MAT1030 – Diskret Matematikk

Forelesningsnotater – Våren 2010

Dag Normann/Roger Antonsen
Universitetet i Oslo

Sist oppdatert: 18. mai 2010 14:14

Dette kompendiet er automatisk generert fra materialet som ble presentert på forelesningene og må leses med dette forbeholdet. Den første utgaven av forelesningsnotene ble skrevet våren 2008 av Dag Normann, deretter bearbeidet av Roger Antonsen våren 2009og videre bearbeidet av Dag Normann våren 2010.
Kommentarer, feil og forslag til forbedringer kan sendes til Dag Normann (dnormann@math.uio.no).

Innhold

MAT1030 – Forelesning 1

Algoritmer, pseudokoder, kontrollstrukturer

Dag Normann - 19. januar 2010

Velkommen til MAT1030!

Introduksjon

- Velkommen til MAT1030: Diskret matematikk!
- Plan for i dag:
  - Mest praktiske opplysninger
  - En oversikt over kurset
  - Litt fra kapittel 1 i læreboken
- Før vi begynner
  - Det er lov å stille spørsmål underveis.
  - Alle forelesningene vil bli lagt ut på kursets hjemmeside.
  - Vi begynner kvart over, og ikke senere.
  - "Spørsmål eller kommentarer?"

Undervisning

- **Foreleser**: Dag Normann (dnormann@math.uio.no)
- Tirsdag 12:15–14:00, Auditorium 1, Vilhelm Bjerknes hus
- Onsdag 10:15–12:00, Auditorium 1, Vilhelm Bjerknes hus
- **Plenumsregning**: Knut Berg (knube@student.matnat.uio.no)
- Fredag 12:15–14:00, Auditorium 1, Vilhelm Bjerknes hus
- Knut vil basere seg på regning på tavlen.
- **Åpne grupper/orakler**:
- Mandag - fredag
- 10.15 - 14.00, Auditorium 3, Vilhelm Bjerknes hus.
  - Se kursets hjemmeside for mer informasjon.
  - Prøv å løs ukeoppgavene selv.
  - Bruk plenumsregningne og de åpne gruppene!

Obligatoriske oppgaver og eksamen

- Kurset har to obligatoriske oppgaver.
  - Fredag 3. mars: Innlevering av obligatorisk oppgave 1.
  - Fredag 7. mai: Innlevering av obligatorisk oppgave 2.
- Oppgavene vil bli lagt ut senest 14 dager før.

- Bedømmes til bestått/ikke bestått.
- Begge oppgavesettene må bestås for å kunne gå opp til eksamen.
- Skriftlig eksamen, 3 timer, uten hjelpemidler.
  - Mandag 7. juni kl. 14.30.

Evaluering

- Alle emner i matematikk skal evalueres av studentene.
- Vi skal velge to til fire kontaktpersoner som sammen med foreleser skal bestemme hvordan evalueringen skal foregå.
- Vi kommer tilbake til valg av kontaktpersoner neste uke.

Pensum og lærebok

Referanser

[1] Peter Grossman *Discrete Mathematics for Computing* (3. utgave) Grassroots Series, Palgrave Macmillian (2009) ISBN: 13:978-0230-21611-2

Det vil være mulig å bruke utgave 2.

Pensum er kapittel 1–7 og 9 og utdrag fra kapittel 10, 11 og 13, samt alle forelesningsnotater og øvingsoppgaver som legges ut på kurset hjemmeside.

Forelesningsnotatene vil inneholde lærestoff som ikke står i boka, **og det er også pensum.**

Hva er diskret matematikk?

- Diskret matematikk er et samlebegrep for matematikk hvor kontinuitet, geometri eller algebra ikke spiller noen stor rolle.
- Diskret matematikk er matematikken tilpasset en digital verden, mens mye annen matematikk er tilpasset en analog verden.
- I MAT1030 skal vi ta for oss temaer som er relevante for en grunnleggende forståelse av bruk av, og virkemåte til, datamaskiner.

Emnebeskrivelsen

Tallsystemer, utsagnslogikk med sannhetsverditabeller, litt om kvantorer og utforming av bevis, elementær mengde- , relasjons- og funksjonslære, induktivt definerte strukturer med generelle rekursive konstruksjoner og induksjonsbevis, litt kombinatorikk, grafer og trær og til sist litt om kompleksitet av algoritmer, heri bruk av O-notasjonen. I emnet legges det vekt på utformingen av algoritmer i tilknytning til stoffet.

Hva er innholdet i MAT1030?

- Algoritmer
- Litt om representasjon av tall
- Logikk
  - Relevans for informatikk

  - Innføring i utsagnslogikk
  - Litt om bevisteknikker
- Mengdelære
  - Grunnleggende begreper
  - Relasjoner
  - Funksjoner
- Induksjon og rekursjon
- Kombinatorikk
- Grafteori med anvendelser
- Trær
- Kompleksitet av algoritmer

Hva legger vi vekt på?

- Hvilken relevans har dette stoffet for studier i informatikk?
- Hvordan kan vi finne algoritmer for å løse de problemene som presenterer seg?
- Vi kommer til å arbeide med algoritmer i tilknytning til logikk i større grad enn det boka gjør.
- Undervisningsmaterialet for dette kommer som en del av kompendiet basert på forelesningene.
- Det finnes lenker til de tilsvarende kompendiene fra 2008 og 2009 på semestersidene for MAT1030.
- Pensum vil være basert på årets kompendium.

Kapittel 1: Algoritmer

Algoritmer

En algoritme er en oppskrift som forteller oss hvordan vi skritt for skritt skal kunne oppnå et resultat eller løse et problem. Eksempler på algoritmer kan være:

- Kakeoppskrifter.
- Automatisk innsjekking på fly.
- Beskrivelsen av hvordan man utfører divisjon mellom flersifrede tall.
- Oppskrift på hvordan man løser opp parenteser og trekker sammen flerleddede uttrykk i algebra.

Hva er det som kjennetegner en algoritme?

> Det skal ikke kreves intelligens eller forståelse for å følge den.

- Du skal ikke kunne kjemi for å bake en kake.
- Du skal kunne sjekke inn på fly selv om du har teknologifobi.
- Det er ikke nødvendig å forstå hva man gjør når man utfører en divisjon, regnetrening er det som trengs.
- Mange lærer seg hvordan de kan løse oppgaver i skolealgebra, uten å ha peiling på hva de egentlig driver med.

Vi skal fokusere på algoritmer som

- beregner funksjoner
- avgjør om et objekt eller en datamengde har en gitt egenskap eller ikke
- organiserer gitte data på en ønsket måte (eksempelvis ordner dataene)
- utfører andre oppgaver i tilknytning til matematikk eller informatikk som vi ønsker å kunne få utført.

Hvem ønsker vi å kommunisere algoritmen til, og hvordan skal det gjøres?

1. Kakebakere, flypassasjerer og liknende.
2. Skolebarn/ungdom og studenter som skal lære matematikk.
3. Teknisk kyndige medmennesker som skal "forstå" algoritmen.
4. Datamaskiner som skal utføre algoritmen for oss.

I MAT1030 er gruppe 3 den mest aktuelle.

Pseudokoder

- En pseudokode er en måte å beskrive en algoritme på.
- Pseudokoden beskriver hvordan algoritmen kan følges trinn for trinn.
- En pseudokode formuleres i et språk som er mer teknisk enn naturlige språk og mindre teknisk enn programmeringsspråk.
- Vi skal bruke pseudokoder på samme måte som i læreboka.
- Man må se eksempler, og øve, for å bli flink til å skrive pseudokoder.

Eksempel (Areal av trekant).

1. Input $\mathtt{h}$ [$\mathtt{h}$ er høyden i trekanten.]
2. Input $\mathtt{g}$ [$\mathtt{g}$ er lengden på grunnlinjen i trekanten.]
3. $\mathtt{areal} \leftarrow \frac{\mathtt{h} \cdot \mathtt{g}}{2}$
4. Output $\mathtt{areal}$

Vi kunne ha skrevet en annen pseudokode for å beregne det samme:

Eksempel.

1. Input $\mathtt{h}$
2. Input $\mathtt{g}$
3. $\mathtt{areal} \leftarrow \mathtt{h} \cdot \mathtt{g}$
4. $\mathtt{areal} \leftarrow \frac{\mathtt{areal}}{2}$
5. Output $\mathtt{areal}$

- Et viktig aspekt ved pseudokoder er bruk av variabler

- Variabler er noe som kan gis forskjellige verdier.

Så langt består en pseudokode av en nummerert liste instruksjoner hvor hver instruksjon har et av følgende tre formater:

- Gi en input-verdi til en variabel.
- Gi en variabel en ny verdi, som en funksjon av de eksisterende verdiene på variablene.
- Gi verdien til en av variablene som output, det vil si resultatet av algoritmen.

Vi kan bruke hva vi vil som variable, eksempelvis er h, g og $areal$ variablene i pseudokodene vi har sett på.
Hvis vi skal beregne verdien av en formel for areal, volum, hastighet etter en viss tids fritt fall og liknende, kan vi bruke pseudokoder slik vi har sett dem til nå. Det finnes imidlertid algoritmer, og tilhørende kontrollstrukturer, for å beregne

- $|x|$ fra x
- $n!$ fra n
- ledd nummer n i Fibonacci-følgen

$$1, 1, 2, 3, 5, 8, 13, 21, \ldots$$

og for å undersøke om

- parentesene i et algebraisk uttrykk er satt på lovlig måte
- et naturlig tall er et primtall.

Kontrollstrukturer

- En kontrollstruktur brukes for å styre hvordan, og hvorvidt, de enkle instruksjonene i en pseudokode skal utføres.
- Vi skal innføre de kontrollstrukturene det er aktuelt å bruke i dette emnet via eksempler på bruk. Disse eksemplene skal supplere eksemplene fra læreboka.
- Vi skal benytte oss av de samme kontrollstrukturene som boka.

Eksempel (Absoluttverdi).
Vi skal gi en pseudokode for å beregne absoluttverdien til et tall x:

1. Input x
2. **If** $x < 0$ **then**

   2.1. $x \leftarrow -x$
3. Output x

Eksempel (Avstand).
Vi skal gi en pseudokode for å beregne avstanden mellom to heltall.

1. Input n [n et heltall]

2. Input m [m et heltall]
3. **If** $n < m$ **then**

   3.1. $x \leftarrow m - n$

   else

   3.2. $x \leftarrow n - m$
4. Output x

Vi skal se tre eksempler på hvordan vi kan skrive pseudokoder for algoritmer som beregner

$$n! = 1 \cdot 2 \cdot \ldots \cdot n.$$

I det ene eksemplet bruker vi en while-løkke, i det andre en repeat-until-løkke og i det siste en for-løkke.

Eksempel (while-løkke).

1. Input n [$n \geqslant 1$, n heltall]
2. $x \leftarrow 1$
3. $i \leftarrow 1$
4. **While** $i \leqslant n$ **do**

   4.1. $x \leftarrow x \cdot i$

   4.2. $i \leftarrow i + 1$
5. Output x

Eksempel (repeat-until-løkke).

1. Input n [$n \geqslant 1$, n heltall]
2. $x \leftarrow 1$
3. $i \leftarrow 1$
4. **Repeat**

   4.1. $x \leftarrow x \cdot i$

   4.2. $i \leftarrow i + 1$
5. **until** $i = n + 1$
6. Output x

Eksempel (for-løkke).

1. Input n [$n \geqslant 1$, n heltall]
2. $x \leftarrow 1$
3. **For** $i = 1$ **to** n **do**
   3.1. $x \leftarrow x \cdot i$
4. Output x

Bytte av verdi på variablene, hvordan vi ikke skal gjøre det og hvordan vi skal gjøre det.

Eksempel (Feil måte).

1. $y \leftarrow x$
2. $x \leftarrow y$

Eksempel (Riktig måte).

1. $hjelp \leftarrow x$
2. $x \leftarrow y$
3. $y \leftarrow hjelp$

MAT1030 – Forelesning 2

Kontrollstrukturer, tallsystemer, basis

Dag Normann - 20. januar 2010

Kapittel 1: Algoritmer (fortsettelse)

Kontrollstrukturer

I går innførte vi pseudokoder og kontrollstrukturer. Vi hadde tre typer grunninstruksjoner:

- Input $variabel$
- $variabel \leftarrow term$
- Output $variabel$

Vi hadde fem kontrollstrukturer

- **If** $\cdots$ **then**
- **If** $\cdots$ **then** $\cdots$ **else**
- **While** $\cdots$ **do**
- **Repeat** $\cdots$ **until**
- **For** $\cdots$ **to** $\cdots$ **do**

Vi skal se på noen flere eksempler på pseudokoder.
Det er ingen som vet om algoritmen som er beskrevet i den neste pseudokoden vil terminere for alle input. Det betyr at den muligens **ikke** er en algoritme i bokas forstand. Den forutsetter at vi kan skille mellom partall og oddetall.

Eksempel (Ubegrenset while-løkke).

1. Input x [$x \geqslant 1$ heltall.]
2. $y \leftarrow 0$
3. **While** $x > 1$ **do**
   - 3.1. **If** x er partall **then**
     - 3.1.1. $x \leftarrow \frac{x}{2}$

       else
     - 3.1.1. $x \leftarrow 3x + 1$
   - 3.2. $y \leftarrow y + 1$
4. Output y

Vi kan finne en enkel pseudokode for å finne ledd nr. n i følgen

$$1, 1, 2, 3, 5, 8, 13, 21, 34, 55, \cdots$$

av Fibonacci-tall:

Eksempel (Fibonacci).

1. Input n [$n \geqslant 1$ heltall]
2. $x \leftarrow 1$
3. $y \leftarrow 1$
4. **For** $i = 2$ **to** n **do**
   4.1. $z \leftarrow x$
   4.2. $x \leftarrow x + y$
   4.3. $y \leftarrow z$
5. Output x

- Parenteser i forskjellige former brukes mye i matematikk, informatikk og spesielt i programmer.
- Eksempelvis, i LaTeX forekommer "parenteser" som f.eks.

  `\begin{center} ... \end{center}`

  og

  `\begin{itemize} ... \end{itemize}`

  og det er viktig at de står riktig i forhold til hverandre.
- Dette kontrolleres når dokumentet eller programmer kompileres.
- Det neste eksemplet på en pseudokode er en parentes-sjekker.
- Vi kan kontrollere om en liste venstre og høyreparenteser

  (( ) (( ))) ( ) (( ))) ( )

  er lovlig eller ikke, ved å telle opp og ned – opp ved (og ned ved) – fra venstre mot høyre.
- Hvis vi til slutt ser at vi har like mange parenteser av hvert slag, og aldri underveis har flere) enn (, er uttrykket i orden.
- I det neste eksemplet gir vi to input, lengden av uttrykket og sekvensen av parenteser.
- Vi sjekker uttrykket fra venstre mot høyre.
- Vi bruker variabelen val til å holde orden på om sekvensen så langt er i orden.
- Vi bruker variabelen y til å telle overskuddet av (.

1. Input n [Lengden av uttrykket, antall parenteser totalt]
2. Input $x_1 \cdots x_n$ [En liste av venstre og høyreparenteser]
3. $y \leftarrow 0$
4. $val \leftarrow$ JA
5. **For** $i = 1$ **to** n **do**
   5.1. **If** $x_i =$ (**then**
   5.1.1. $y \leftarrow y + 1$
   else

5.1.2. **If** $y = 0$ **then**

5.1.2.1. $val \leftarrow$ NEI

else

5.1.2.2. $y \leftarrow y - 1$

6. **If** $y > 0$ **then**

6.1. $val \leftarrow$ NEI

7. Output val

Hva skal dere kunne fra kapittel 1?

Forventede ferdigheter:

- Kunne uttrykke en algoritme i pseudokode, og kunne bruke de forskjellige kontrollstrukturene på riktig måte.
- Kunne følge en algoritme gitt ved en pseudokode og inputverdier på variablene skritt for skritt, og kunne holde orden på hvordan verdiene på variablene endrer seg under "utregningen".
- Kunne forklare hvorfor en pseudokode løser den oppgaven den er satt til å utføre, det vil si, kunne gi en muntlig eller skriftlig *dokumentasjon*.

Kapittel 2: Representasjon av tall

Hva som gjennomgås i forelesningene

- Hele kapittel 2 og 3 er pensum.
- Siden dette stoffet er kjent for mange, så overlates mesteparten av dette til den enkelte.
- Det er lite i de senere kapitlene som avhenger direkte av det som står i kapittel 2 og 3.
- I boken står det f.eks. godt forklart – ved hjelp av algoritmer beskrevet ved pseudokoder – hvordan man konverterer fra binære tall til desimaltall og vice versa.
- Vi skal gå raskt gjennom det viktigste her.

Tallmengder

Hvilke tall vi betrakter er avhengig av hva vi ønsker å bruke dem til. I MAT1030 vil vi stort sett betrakte følgende typer tall:

- Naturlige tall $\mathbb{N}$

$$1, 2, 3, \cdots$$

- Hele tall $\mathbb{Z}$

$$\cdots, -3, -2, -1, 0, 1, 2, \cdots$$

- Rasjonale tall $\mathbb{Q}$

Tall som kan skrives som en brøk $\frac{p}{q}$

- Reelle tall $\mathbb{R}$

"alle tallene"

- Mange mener at tall er punkter på tall-linja, og at det ikke spiller noen rolle om vi betrakter 2 som et naturlig tall, et heltall, et rasjonalt tall eller et reelt tall.

- I programmeringsammenheng kan det spille en stor rolle hva slags verdier en variabel kan få lov til å ta, og representasjonen av et tall som et dataobjekt kan variere med hva slags type tall vi betrakter.

Det finnes andre tallmengder som også er av interesse i matematikk og informatikk, eksempelvis

- Komplekse tall
- Algebraiske tall
- Kvaternioner
- Ordinaltall

Representasjon av tall

Så langt tilbake vi har informasjon om, har mennesker og kulturer hatt muntlig og skriftlig språk for tall.

Romertallet

$$\text{MCMXXVIII}$$

er en alternativ måte å skrive

$$1928$$

på.

Hvis vi blir bedt om å skrive et program for addisjon av to tall, betyr det mye om vi bruker den romerske eller dagens måte å skrive tall på.

- De tallene vi bruker til daglig kalles desimaltall, eller tall i 10-tallsystemet.
- Dette er et plass-siffersystem med basis 10.
- Det betyr igjen at hvert siffer angir et antall 10'er potenser, og sifferets posisjon forteller oss hvor stor potensen er.

Eksempel.

- 258 står for
$$2 \cdot 10^2 + 5 \cdot 10^1 + 8 \cdot 10^0.$$
- $3,14$ står for
$$3 \cdot 10^0 + 1 \cdot 10^{-1} + 4 \cdot 10^{-2}$$

Tverrsumtesten (digresjon)

Tverrsummen til et desimaltall er summen av alle sifrene.

Eksempel.

- Tverrsummen til 234 er $2 + 3 + 4 = 9$
- Tverrsummen til 15987 er $1 + 5 + 9 + 8 + 7 = 30$
- Tverrsummen til 2825 er $2 + 8 + 2 + 5 = 17$

- Legg merke til at resten vi får når vi deler tallet på 9 er det samme som vi får når vi deler tverrsummen på 9.
- Kan dette forklares matematisk?

Påstand (Tverrsumtesten).
Hvis vi skriver et tall n på desimalform og lar $T(n)$ være tverrsummen til n, så får vi samme rest når vi deler n på 9 som når vi deler $T(n)$ på 9.

Bevis.
La $a_k \dots a_0$ være desimalformen til n.
Da er

$$n = a_k 10^k + \cdots + a_1 10 + a_0$$

Når $1 \leqslant i \leqslant k$ kan $10^i - 1$ deles på 9, siden sifrene består av bare 9-tall.
Vi har at

$$n = T(n) + a_k(10^k - 1) + \cdots + a_1(10 - 1)$$

(trenger litt ettertanke), og påstanden følger (trenger litt ettertanke til).

Vi får også at vi får den samme resten om vi deler et tall på 3 som når vi deler tverrsummen på 3.

Det betyr at vi kan kontrollere om et tall kan deles på 3 ved å se på tverrsummen.

Er tallet veldig stort, kan vi se på tverrsummen av tverrsummen, osv.

Binære tall

- Det er kulturelt betinget at vi bruker 10 som basis i tallsystemet vårt.
- Alle tall > 1 kan i prinsippet brukes.
- I informatikksammenheng er det like naturlig å bruke 2, 8 og 16 som basistall.
- Bruker vi 2 som basis, sier vi at tallet er på binær form.

Eksempel.
Vi tolker en binær form (hvor alle sifrene er 0 eller 1) omtrent som om det var et desimaltall, bortsett fra at vi erstatter 10 med 2:

- $1010_2 = 1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0 = 8 + 2 = 10_{10}$
- $11011_2 = 16 + 8 + 2 + 1 = 27$
- $100111001_2 = 256 + 32 + 16 + 8 + 1 = 313$

Binær representasjon kan selvfølgelig også brukes til tall som er mindre enn 1.

- $0,100101_2 = \frac{1}{2} + \frac{1}{16} + \frac{1}{64} = \frac{32+4+1}{64} = \frac{100101_2}{64}$
- $0,01101_2 = \frac{1}{4} + \frac{1}{8} + \frac{1}{32} = \frac{8+4+1}{32} = \frac{01101_2}{32}$

Det finnes en enkel prosedyre for å regne ut verdien av et binært tall:

1. Input n [n er antall sifre i sekvensen]
2. Input $x_1 \dots x_n$ [en sekvens av 0'er og 1'ere]
3. $y \leftarrow 0$ [y skal bli verdien på sekvensen tolket som et binært tall]
4. **For** $i = 1$ **to** n **do**
   4.1. $y \leftarrow 2y$
   4.2. **If** $x_i = 1$ **then**
      4.2.1. $y \leftarrow y + 1$
5. Output y

Regn f.eks. ut hva som skjer med input $n = 4$ og $x_1x_2x_3x_4 = 1101$.

Aritmetikk

Vi utfører addisjon, subtraksjon, multiplikasjon og divisjon av tall på binær form omtrent som for tall i 10-tallsystemet, bortsett fra at alt i prinsippet blir mye enklere, den lille addisjonstabellen og den lille multiplikasjonstabellen blir så mye mindre.
Som eksempler regner vi eventuelt følgende stykker på tavla (oppgaver for den som ikke er på forelesningen).

- $17 + 14$
- $17 - 14$
- $5 \cdot 11$
- $11 : 5$ med fire siffer bak komma.

Det er selvfølgelig mulig å finne pseudokoder som uttrykker de algoritmene vi vil bruke, men som i skolematematikken er det her best å demonstrere algoritmene ved eksempler.

Oktal og heksadesimal form

Hvis man bruker 8-tallsystemet arbeider man med tall på oktal form.
Eksempelvis vil vi ha

- $443_8 = 4 \cdot 8^2 + 4 \cdot 8 + 3 = 256 + 32 + 3 = 291_{10}$
- $3,21_8 = 3 + 2 \cdot \frac{1}{8} + \frac{1}{64}$

Hvis man bruker 16-tallsystemet arbeider man med tall på heksadesimal form.
Her må man supplere symbolene $0, 1, \dots 9$ med sifre A, B, C, D, E og F.
Eksempelvis vil

$$2C3_{16} = 2 \cdot 16^2 + 12 \cdot 16 + 3 = 512 + 192 + 3 = 707_{10}.$$

Fordelen med oktal og heksadesimal form er at regning med tall i disse tallsystemene representerer en rasjonalisering av regning med binære tall.

Ved å gruppere tre og tre siffer kan en binær form omgjøres direkte til oktal form:

$$101\ 100\ 001\ 010_2 = 5412_8$$

og ved å gruppere fire og fire sifre kan en binær form omgjøres til heksadesimal form:

$$1011\ 0000\ 1010_2 = B0A_{16}.$$

Oppgave (Tverrsumstest).
Gå tilbake til beviset for at tverrsumstesten for delelighet med 3 og 9 holder i 10-tallsystemet, og finn ut for hvilke tall vi har en tverrsumstest for tall på oktal og heksadesimal form.

MAT1030 – Forelesning 3

Litt om representasjon av tall

Dag Normann - 26. januar 2010

Kapittel 3: Litt om representasjon av tall

Hva vi gjorde forrige uke

- Vi diskuterte løst hva vi skal mene med en algoritme.
- Vi så hvordan vi kan beskrive algoritmer ved hjelp av *pseudokoder*.
- Vi beskrev fem *kontrollstrukturer* som vi vil bruke i våre pseudokoder, og vi så på en del eksempler.
- Vi diskuterte kort det *dekadiske* tallsystemet, binære-, oktale og heksadesimale tall.
- Vi forventer ikke noen stor regneferdighet i tilknytning til tallsystemene, men en viss forståelse av dem.

En rask gjennomgang av kapittel 3

- Kapittel 3 inneholder mye som allerede er velkjent for informatikkstudenter.
- Vi skal derfor gå raskt gjennom de viktigste punktene.
- Resten blir overlatt til selvstudium.
- Dere kan be om hjelp hos plenumsregner eller gruppelærere, men vær forberedt på at noen av dem *ikke* har tatt noe emne i digital representasjon.
- Se forelesningsnotatene for 2008 for noe fyldigere forelesningsnotater.

Introduksjon

- Grunnenheten er en bit.
- En bit vil være i en av to tilstander: 0 eller 1.
- En byte er en sekvens av 8 bit, f.eks.

$$10011001.$$

- I en byte kan vi lagre $2^8 = 256$ forskjellige informasjoner.
  - Dette svarer til alle tall med to sifre i det heksadesimale systemet.
- Hele tall og reelle tall er forskjellige typer tall, og "samme" tall må representeres forskjellig når det oppfattes som et heltall og når det oppfattes som et reelt tall.
- Ved å kjenne til hvordan tall representeres, vil vi kjenne til begrensningene og mulige feilkilder.
  Hvis man skal foreta en numerisk beregning hvor antall avrundinger er i millionklassen, er det viktig å vite hvor stor feil som kan oppstå fordi representasjonen i maskinen ikke er nøyaktig.
  Det finnes uendelig mange tall, men bare endelig mange av dem kan representeres i en konkret maskin.

Litt om representasjon av hele tall

Når vi skal representere hele tall i en datamaskin er det tre spørsmål som må besvares:

1. Hvor mange tall ønsker vi å representere?
2. Hvilke tall ønsker vi å representere?
3. Hvordan vil vi representere dem?

Svaret på spørsmål 1 avhenger av hvor mange bit/byte vi vil sette av for å representere et enkelt tall.
Bruker vi flere bit, kan vi representere flere enkelttall, men vi vil bruke lengere tid på å manipulere tallene, og vi vil ha mindre plass til andre formål.

- Anta – for enkelhets skyld – at vi kun har én byte, det vil si 8 bit.
- Da kan vi representere $2^8 = 256$ forskjellige tall.
- F.eks.
  - alle tall n slik at $0 \leqslant n \leqslant 255$,
  - alle tall n slik at $-255 \leqslant n \leqslant 0$, eller
  - alle tall n slik at $-128 \leqslant n \leqslant 127$.
- Det siste er det vanligste.

- Den vanligste er å la det første bitet være et fortegnsbit, som er 1 hvis n er negativt.
- De resterende 7 bit brukes litt forskjellig avhengig av om n er negativt eller ikke.
- Hvis n er positivt, så bruker vi binærformen til n.
- Hvis n er negativt, så bruker vi binærformen til $n + 128$.

Eksempel.
- La $n = -126$.
- Fortegnsbitet blir 1.
- Vi representerer $n + 128 = 2$ med 7 bit som $000\ 0010$,
- Representasjonen blir derfor $1000\ 0010$.

- Et viktig poeng med representasjoner er at vi så enkelt som mulig skal kunne regne på det som representeres.
- Man kan være fristet til å representere et negativt tall, $-n$, med fortegnsbit 1 og deretter binærformen til n.
- Det er to grunner til at vi vil gjøre det anderledes.
  - Vi ville få to representasjoner av 0 og ingen av -128.
  - Vi ville ikke kunne brukt samme prosedyrer for addisjon og subtraksjon for positive og for negative tall.
  - Dette illustreres best hvis vi bare bruker fire bit i representasjonen:

Heltall	Representasjon med fire bit
-8	1000
-7	1001
-6	1010
-5	1011
-4	1100
-3	1101
-2	1110
-1	1111
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111

Definisjon (Representasjon av hele tall).
Hvis vi har k bit til disposisjon for å representere hele tall, ($k = 8$, $k = 16$ og $k = 32$ er de mest aktuelle) representerer vi alle hele tall a slik at $-2^{k-1} \leqslant a < 2^{k-1}$ på følgende måte:

- Hvis $a \geqslant 0$, er første bit 0 og resten er det binære tallet for a med $k-1$ sifre
- Hvis $a < 0$ er første bit 1 og resten er det binære tallet for $2^{k-1} + a$

Eksempel.

- La $a = -23$. Da er $a < 0$, så første bit må være 1.
- $128 + (-23) = 105$ og $1101001_2 = 105$
- Det gir at representasjonen av a er 11101001.

- Dette synes som en tungvint måte, og det er det.
- Hvis vi gjennomfører eksemplet ved å regne binært, ser kommer vi frem til det som kalles toerkomplementet.

  (Vi regner eksemplet binært på tavlen.)

Definisjon (Toerkomplement).

- La $a_1 \cdots a_k$ være en sekvens av 0'er og 1'ere, med en 0'er lengst til venstre, og med minst én 1'er.
- toerkomplementet til sekvensen får vi ved å starte fra høyre, lese ett og ett siffer og
  1. Alle 0'er til høyre beholdes.

2. Første 1'er fra høyre beholdes.
3. Alle siffer til venstre for første 1'er endres fra 0 til 1 eller fra 1 til 0.

- Hvis $x_1 \cdots x_8$ er datarepresentasjonen av et positivt heltall n , er toerkomplementet representasjonen av $-n$.
- Dette gjelder selvfølgelig om vi bruker 16-bit representasjoner eller 32-bit-representasjoner også.

Litt om representasjon av reelle tall

For å forstå prinsippene for representasjon av reelle tall i en maskin, kan vi se på hvordan reelle tall fremstilles på en lommeregner eller i en tekst som omhandler store eller små tall. I stedenfor å skrive

$$230850100000000000000$$

kan vi skrive

$$0,2308501 \cdot 10^{21}$$

eller

$$2,308501\text{E}20$$

Med untak av at vi vil bruke binære tall i stedenfor vanlige tall fra titallsystemet, vil vi kode reelle tall via tre informasjonsbiter, fortegn, sifrene brukt og eksponent.
Vi vil skrive tallene på normalisert binær form:

- Et fortegn, $+$ eller $-$.
- Et *binært* desimaluttrykk på formen

  $$0,1\cdots$$

  kalt signifikanden.
- En eksponentdel

  $$2^n$$

  hvor n er et heltall.
- Vi har enkle prosedyrer for å utføre aritmetikk på tall på normalisert binær form.

Eksempel.

a) $0,101100101 \cdot 2^{-3}$ er på normalisert binærform.
Vi kunne skrevet dette som $0,000101100101$.

b) $101,0101101$ er ikke på normalisert binærform.
Da burde vi skrevet $0,1010101101 \cdot 2^3$

c) $0,1000010100001 \cdot 2^{28}$ er på normalisert binær form.
I dette tilfellet er det liten grunn til å skrive tallet på eksakt form, og det ville også antydet en større nøyaktighet enn det vi trolig har grunnlag for.

d) $0,11010 \cdot 2^{-87}$ er på normalisert binær form.

- Vi følger læreboken i beskrivelsen av hvordan reelle tall representeres på en datamaskin.
- Vi bruker 32 bit til å representere ett reellt tall.
- Det første bitet bruker vi til å representere fortegnet.
- De neste 8 bit bruker vi til å representere eksponenten.
- De siste 23 bit bruker vi til å representere signifikanden.

- For fortegnet bruker vi 0 for $+$ og 1 for $-$.
- For signifikanden bruker vi det 23-sifrede binære tallet som står bak komma.
  - Dette svarer til mellom 7 og 8 sifre i titallsystemet, så det er den nøyaktigheten vi regner med.
  - Avrundingsfeil vil da være i størrelsesorden 2^{-23}.
- For å representere eksponenten, bruker vi 8 bit.
  - Det gir oss mulighet til å fange opp $2^8 = 256$ forskjellige eksponenter.
  - Vi har bruk for å representere omtrent like mange negative som positive eksponenter.
  - Standard metode er at man representerer en eksponent ved summen av binærformen og 01111111, hvor $01111111_2 = 127$.
  - Dette plasserer representasjonen av 0 omtrent midt i, og gir oss mulighet for å representere alle eksponenter fra -127 til 128.

Eksempel.

Vi vil finne datarepresentasjonen av tallet $2,5_{10}$

Først må vi skrive tallet på binær form:

$$2,5_{10} = 10,1_2.$$

Den normaliserte binære formen er

$$0,101 \cdot 2^2.$$

Eksempel (fortsatt).

Da vil vi bruke 0 for å representere fortegnet, 10000001 for å representere eksponenten og 10100000000000000000000 til å representere signifikanden.

$2,5_{10}$ representeres da av bitsekvensen

$$01000000\ 11010000\ 00000000\ 00000000$$

fordelt på 4 byte.

Eksempel.

Finn representasjonen av

$$-\frac{1}{3}.$$

Binærformen med 23 gjeldende siffer er

$$(-\frac{1}{3})_{10} = -0,01010101010101010101010_2.$$

Normalisert binær form blir da

$$-0,10101010101010101010101 \cdot 2^{-1}.$$

Eksempel (fortsatt).

Da må vi bruke

- 1 for å representere fortegnet.
- 01111110 for å representere eksponenten.
- 10101010101010101010101 for å representere signifikanden.

Representasjonen, fordelt på 4 byte, blir da

$$10111111\ 01010101\ 01010101\ 01010101.$$

Kapittel 4: Logikk

Logikk

Kapittel 4 i læreboka gir en kort innføring i viktige deler av logikken.

Hvorfor skal informatikkstudenter lære noe om logikk?

- von Neumanns konstruksjon av datamaskinen er basert på utsagnslogikk og logiske porter.
- Grunnarkitekturen av datamaskiner har ikke endret seg mye siden den tid, selv om de teknologiske forbedringene har vært enorme.
- Vi bruker språket fra logikk til å formulere forutsetninger P i kontrollstrukturer som

  If P **then** $\cdots$ **else** $\cdots$
- Vi bruker logikk for automatisk å sikre at forskjellige data lagt inn i en base er forenlige med hverandre.
- Vi trenger logikk for å kunne definere hva som menes med korrekthet av et program.
- Kunstig intelligens, sikkerhetsprotokoller for utveksling av informasjon og mye annen avansert bruk av datamaskiner bygger på logikk.

- Logikk, slik vi kjenner faget i dag, har sin opprinnelse fra gresk filosofi og vitenskap.
- Tanken var at et resonnement må kunne stykkes opp i enkelte tankesprang, grunnresonnement, hvor det vil være lett å bestemme om grunnresonnementene er riktige eller representerer feilslutninger.
- Da kan man finne ut av hvilke deler av et argument som baserer seg på ren tankekraft, og hva som baserer seg på unevnte forutsetninger.
- Aristoteles formulerte en del syllogismer som skulle gi oss de lovlige logiske slutningene.
- Vi skal ikke gå inn på den klassiske syllogismelæren, men etterhvert legge grunnlaget for den.

Eksempel.

a) Jeg rekker ikke middagen.
Du kommer senere hjem enn meg.
Altså rekker ikke du middagen.

b) Jeg rekker ikke middagen.
Hvis jeg ikke rekker middagen, rekker ikke du middagen.
Altså rekker ikke du middagen.

I eksempel a) har vi en skjult forutsetning, mens i eksembel b) er argumentet logisk sett riktig. Vi kan skrive dette argumentet om til et annet, hvor b) holder, men a) ikke holder.

Enda et eksempel

Eksempel.

a) Jeg liker ikke Bamsemums.
Du liker alt jeg liker.
Altså liker ikke du Bamsemums.

b) Jeg liker ikke Bamsemums.
Hvis jeg ikke liker Bamsemums, liker ikke du Bamsemums.
Altså liker ikke du Bamsemums.

Her er det opplagt en feil i argument a), mens b) holder fortsatt.
Det vil være vanskelig å programmere en generell argumentsjekker som vil godta a) i det første eksemplet, men ikke i det andre.

MAT1030 – Forelesning 4

Logikk

Dag Normann - 27. januar 2010

Kapittel 4: Logikk (fortsettelse)

Kort oppsummering

I går begynte vi å se på kapitlet om logikk.

Vi snakket litt om den historiske bakgrunnen for logikk, og om hvorfor informatikkstudenter bør lære seg logikk.

Vi så på to eksempler som skal illustrere at det i noen resonnementer vil finnes skjulte antagelser.

Eksemplene fra i går

Eksempel.

(a) Jeg rekker ikke middagen.
Du kommer senere hjem enn meg.
Altså rekker ikke du middagen.

(b) Jeg rekker ikke middagen.
Hvis jeg ikke rekker middagen, rekker ikke du middagen.
Altså rekker ikke du middagen.

Eksempel.

(a) Jeg liker ikke Bamsemums.
Du liker alt jeg liker.
Altså liker ikke du Bamsemums.

(b) Jeg liker ikke Bamsemums.
Hvis jeg ikke liker Bamsemums, liker ikke du Bamsemums.
Altså liker ikke du Bamsemums.

Logisk holdbart argument

Som en tommelfingerregel kan vi si følgende.

> Et argument er *logisk holdbart* hvis vi kan bytte ut delformuleringer som ikke inneholder noe av den logiske strukturen med andre formuleringer uten at argumentet blir feil.

- Hovedutfordringen blir å bestemme hva som tilhører den logiske strukturen og hva vi kan forandre på for å bruke testen over.
- Et argument er logisk holdbart i kraft av sin form, ikke sitt innhold.
- Hvis en datamaskin skal kunne sjekke gyldigheten av et resonnement, må vi eksplisere alle skjulte forutsetninger i resonnementet.
- Vi må også eksplisere hvilke atomære resonnementer som er lovlige, for en maskin kan bare kontrollere om noe er utført i tråd med forhåndsbestemte regler.
- Hvis en maskin skal kunne "forstå" hva som tilhører den logiske strukturen i en formulering, må den knyttes til bruk av spesielle tegn eller ordsekvenser.
- Dette er helt analogt med den rigiditeten som kreves av et program i et programmeringsspråk.

Fire viktige personer




Før vi starter på fagstoffet skal vi trekke frem fire navn som er viktige for utviklingen av logikk i det 20. århundre og for sammenfiltringen av matematisk logikk og teoretisk informatikk.

- Toralf Skolem (1887–1963)
- Kurt Gödel (1906–1978)
- Alan Turing (1912–1954)
- John von Neumann (1903–1957)

Hva skal vi lære av logikk?

- Utsagnslogikk
- Predikatlogikk
- Litt om hvordan man fører bevis
- Algoritmer for å teste om utsagn er logisk holdbare eller ikke

Utsagnslogikk

Definisjon.
Et utsagn er en ytring som enten er sann eller usann.

- Som matematisk definisjon er ikke denne definisjonen spesielt god, ettersom den ikke kan brukes til å bestemme hva som er utsagn og hva som ikke er det.
- Er "Per er en dannet mann" et utsagn?

- Vi vil betrakte dette som et utsagn, ettersom ytringen i en gitt situasjon uttrykker en oppfatning som enten kan aksepteres eller bestrides.
- Vi skal ikke gå nærmere inn på den filosofiske analysen av hva et utsagn er.

Eksempel.

Følgende er eksempler på utsagn, slik vi skal bruke begrepet:

- $2^{10} < 1000$
- $\pi \neq e$
- Anne har røde sko.
- I morgen blir det pent vær.
- Det fins mange grader av uendelighet.

Eksempel.

Følgende ytringer kan ikke oppfattes som utsagn:

- Når går toget?
- Uff!!!
- Dra til deg den lurvete mærschedesen din, eller så kjører jeg på den! (Sitat fra sint trikkefører i Grensen.)
- Måtte sneen ligge lenge og løypene holde seg.

Eksempel.

Vi har sett endel utsagn i forbindelse med formuleringer av kontrollstrukturer i pseudokoder:

- **While** $i > 0$ **do**
- **Repeat** $\cdots$ **until** $x > k$
- **If** x er et partall **then** $\cdots$ **else** $\cdots$

Under en utregning vil verdiene på variablene endre seg, men ved hvert enkelt regneskritt vil "ytringene" enten være sanne eller usanne, og vi ser derfor på dem som utsagn.

Eksemplene på forrige side aktualiserer spørsmålet om matematiske likninger, ulikheter og andre formler hvor det forekommer variable størrelser kan betraktes som utsagn:

- $x^2 + 2x - 1 = 0$
- $\sin^2 x + \cos^2 x = 1$
- $f(x) = f'(x)$

Det første tilfellet er en likning i variabelen x, det andre en kjent identitet fra trigonometrien og det siste en differensiallikning hvor f er den ukjente.

For at vi skal slippe å slåss om dette er eksempler på utsagn eller ikke, innfører vi et nytt begrep, et predikat.

Definisjon.
Et predikat er en ytring som inneholder en eller flere variable, men som vil bli sann eller usann når vi bestemmer hvilke verdier variablene skal ha.

Eksempel.
Alle eksemplene fra forrige side, $x^2 + 2x - 1 = 0$, $\sin^2 x + \cos^2 x = 1$ og $f(x) = f'(x)$, er eksempler på predikater. I de to første tilfellene er x variabelen, og i det siste tilfellet er både f og x variable.

Utsagnsvariable og sannhetsverdier

Det er ikke så viktig å vite hva et utsagn er. Det viktige er at når vi betrakter en ytring som et utsagn, stripper vi ytringen for alt untatt egenskapen at den vil være sann eller usann.

Vi vil bruke bokstaver p, q, r og liknende som utsagnsvariable, det vil si at de kan stå for et hvilket som helst utsagn.

Vi vil la **T** og **F** stå for de to sannhetsverdiene sann og usann (*true* og *false*).

Hver utsagnsvariabel p kan da ha en av verdiene **T** eller **F**.

- Det finnes mange andre bokstaver eller symboler man kan anvende for å betegne sannhetsverdiene.
  - true og false
  - tt og ff
  - $\top$ og $\bot$
  - True og False
  - sann og usann (ordet "gal" anbefales ikke)
  - **S** og **U**

Utsagnslogiske bindeord, konnektiver (og)

Eksempel.
La oss se på følgende pseudokode:

1. Input x [$x \geqslant 0$, x heltall]
2. Input y [$y \geqslant 0$, y heltall]

3. **While** $x > 0$ og $y > 0$ **do**
   3.1. $x \leftarrow x - 1$
   3.2. $y \leftarrow y - 1$
4. $z \leftarrow x + y$
5. Output z

Dette er en algoritme for å beregne $|x - y|$ fra x og y.

Definisjon.

- Hvis p og q er to utsagn, er uttrykket $p \wedge q$ også et utsagn.
- Vi leser det p **og** q.
- $p \wedge q$ er sann hvis både p og q er sanne, ellers er $p \wedge q$ usann.
- Vi kaller ofte $p \wedge q$ for konjunksjonen av p og q.

Definisjonen av når $p \wedge q$ er sann kan gis i form av en tabell.
En slik tabell kaller vi en sannhetsverditabell.
Utarbeidelsen av sannhetsverditabeller vil være en viktig ferdighet i dette kurset:

p	q	$p \wedge q$
T	**T**	**T**
T	**F**	**F**
F	**T**	**F**
F	**F**	**F**

I en matematikk/informatikksammenheng er det greit å bruke symbolet $\wedge$:

- $3 \leqslant x \wedge x \leqslant 5$ er en helt grei formulering.
- **While** $x > 0 \wedge y > 0$ **do** kan være en alternativ måte å starte while-løkka fra eksemplet vårt på.
- Ofte vil man finne at man bruker samme typografi som for andre kontrollstrukturer i denne sammenhengen:

  While $x > 0$ **and** $y > 0$ **do**
- I noen søkemotorer brukes **AND** med store bokstaver for å markere at man leter etter sider hvor to ord forekommer, eksempelvis

  Gödel **AND** Turing.

  Hvis man gjengir sammensetning av utsagn i dagligtale, er det bedre å bruke ordet "og".

  Man må imidlertid være klar over at den utsagnslogiske forståelsen visker ut noen av de nyansene vi kan legge inn i dagligtale.

  I de to første eksemplene på neste side vil utsagnslogikken fange opp meningen, mens vi i de to neste mister mye av meningen.

Eksempel.

1. Per er født i Oslo og Kari er født i Drammen.
2. Jeg liker å spille fotball og jeg liker å drive med fluefiske.
3. Jeg gikk inn i stua og tok av meg skiene.
   Jeg tok av meg skiene og gikk inn i stua.
4. Jeg bestilte snegler til forrett og du forlot meg rasende
   Du forlot meg rasende og jeg bestilte snegler til forrett.

Utsagnslogiske bindeord, konnektiver (eller)

Det neste bindeordet vi skal se på er *eller*.

Dette ordet kan ha to betydninger, og vi må velge en av dem.

Dette kommer vi tilbake til etter et par eksempler.

Eksempel.
La oss ta utgangspunkt i følgende pseudokode:

1. Input x [$x \geqslant 0$, x heltall]
2. Input y [$y \geqslant 0$, y heltall]
3. $z \leftarrow 0$
4. **While** $x > 0$ eller $y > 0$ **do**
   4.1. $x \leftarrow x - 1$
   4.2. $y \leftarrow y - 1$
   4.3. $z \leftarrow z + 1$
5. Output z

Dette gir oss en algoritme for å beregne $max\{x, y\}$.

Hvis vi gir x og y store verdier n og m, vil både x og y ha positive verdier ved oppstart og etter noen få gangers tur i while-løkka.
Vi ønsker ikke at løkka skal stoppe av den grunn.
Derfor vil vi gjerne at et utsagn "p eller q" skal kunne være sant også når både p og q er sanne, i det minste i denne sammenhengen.
Er $2 \leqslant 3$? Er $3 \leqslant 3$?.
I en matematisk sammenheng vil vi gjerne at begge deler skal være sanne, vil jo at $x \leqslant 3$ skal være oppfylt både av de tallene som er ekte mindre enn 3 og av 3 selv.
Det betyr at når et av leddene i et eller-utsagn er sant, vil vi at hele utsagnet skal være sant.
Vi vil bruke symbolet $\vee$ for 'eller', og da er

$$x \leqslant y \text{ det samme som } x < y \vee x = y.$$

Definisjon.

- Hvis p og q er to utsagn, er uttrykket $p \vee q$ også et utsagn.
- Vi leser det p **eller** q.
- $p \vee q$ er sann hvis p, q eller begge to er sanne, ellers er $p \vee q$ usann.
- Vi kaller $p \vee q$ for disjunksjonen av p og q.

Definisjonen av når $p \vee q$ er sann kan også gis i form av en sannhetsverditabelltabell:

p	q	$p \vee q$
T	**T**	**T**
T	**F**	**T**
F	**T**	**T**
F	**F**	**F**

Eksempel.

Følgende eksempler fra dagligtale viser at det er to forskjellige måter å forstå ordet 'eller' på

- Du kan få servere pølser eller du kan få servere pizza i bursdagsselskapet.
- Vil du ha en PC eller vil du ha en Mac?
- Jeg kommer til middag om toget er i rute eller om jeg får sitte på med en kollega.
- Om du leser VG eller om du leser Dagbladet finner du ikke noe stoff om hyperbolsk geometri.

Vi skal bruke den inklusive betydningen av *eller*, og vi bruker symbolet $\vee$ eller kontrollstrukturvarianten **or**.

Vi vil bruke dette bindeordet i en matematikk/informatikksammenheng, og være varsomme med å overføre den inklusive tolkningen til dagligtale.

Ekslusiv *eller* kan også defineres ved en sannhetsverditabell.

Dere utfordres til å gjøre dette selv.

I enkelte programmeringssammenhenger, trenger vi å nyansere forståelsen av $\vee$ og av $\wedge$ ytterligere.

Anta at $P(\vec{x})$ og $Q(\vec{x})$ er to prosedyrer ($\vec{x}$ er en vanlig måte å skrive en generell sekvens av variable på), slik at vi ikke kan være sikre på om de tilhørende programmene alltid terminerer.

Anta at vi bruker et programmeringsspråk som tillater kontrollstrukturer av tilnærmet form

If $P(\vec{x}) > 0$ **or** $Q(\vec{x}) > 0$ **then** $\cdots$

Skal vi da kunne fortsette når $P(\vec{x})$ ikke har noen verdi, men $Q(\vec{x}) > 0$?

Diskusjonen foregår muntlig på forelesningen.

Utsagnslogiske bindeord, konnektiver (ikke)

Det neste ordet vi skal se på er ikke i betydningen *det er ikke slik at*.

Eksempel.

- Månen er ikke full i morgen.
- Hurtigruta går ikke innom Narvik.
- Jeg rekker ikke middagen.
- Jeg liker ikke Bamsemums.

I alle disse tilfellene benekter vi en positiv påstand, eksempelvis "Jeg liker Bamsemums".

Eksempel.

1. Input x [$x \geqslant 1$ heltall]
2. Input y [y heltall, $1 \leqslant y \leqslant x$]
3. **While** $y \neq 0$ **do**
   3.1. $z \leftarrow y$
   3.2. $y \leftarrow rest(x, y)$ [$rest(x, y)$ gir restdelen når x deles på y]
   3.3. $x \leftarrow z$
4. Output x

Dette er en måte å formulere Euklids algoritme på (en måte å finne største felles faktor i to tall på).
Poenget her er formuleringen $y \neq 0$, en benektelse av at $y = 0$.

Vi vil vil bruke et spesielt tegn, $\neg$ for å uttrykke at vi benekter et utsagn.

Definisjon.

- Hvis p er et utsagn, er $\neg p$ et utsagn.
- $\neg p$ får sannhetsverdien **F** om p har sannhetsverdien **T** og $\neg p$ får sannhetsverdien **T** om p har sannhetsverdien **F**.
- Vi kaller $\neg p$ for negasjonen av p.

Vi kan også gi denne definisjonen på sannhetsverditabellform:

p	$\neg p$
T	**F**
F	**T**

Denne tabellen er selvforklarende.

MAT1030 – Forelesning 5

Utsagnslogikk

Dag Normann - 2. februar 2010

Kapittel 4: Logikk (fortsettelse)

Repetisjon

- Forrige gang snakket vi om *utsagn* og *predikater*, og vi innførte bindeordene (konnektivene) $\wedge$ for **og**, $\vee$ for **eller** og $\neg$ for **ikke**.
- Vi så hvordan vi kunne definere disse tre konnektivene ved hjelp av sannhetsverditabeller.
- Spesielt poengterte vi at $\vee$ står for *inklusiv* eller, det vil si at $p \vee q$ er sann når både p og q er sanne.
- Vi tar opp tråden der vi slapp den.

Sammensatte utsagn

Vi skal snu litt på rekkefølgen av stoffet i forhold til læreboka.

Ved å bruke konnektivene $\wedge$, $\vee$ og $\neg$ har vi gitt utsagnslogikken sin fulle uttrykkskraft.

De konnektivene vi skal se på senere, kan erstattes med sammensatte uttrykk hvor vi bare bruker $\neg$, $\wedge$ og $\vee$.

Det er faktisk mulig å klare seg med bare $\neg$ og $\wedge$ eller bare med $\neg$ og $\vee$, men da trenger vi sammensatte utsagn som det er vanskelig å lese.

For å fortsette denne diskusjonen, må vi se på hva vi mener med sammensatte utsagn.

Sammensatte utsagn, bruk av parenteser

Vi har sett at $x \neq 0$ egentlig er en alternativ skrivemåte for $\neg(x = 0)$.

Anta at vi i en programmeringssammenheng har bruk for å uttrykke betingelsen

$$x \neq 0 \text{ og } y > 0.$$

Dette burde vi kunne skrive som

$$\neg(x = 0) \wedge y > 0.$$

Hvis p er utsagnet $x = 0$, q er utsagnet $y > 0$ og r er utsagnet $p \wedge q$, skal $\neg r$ være utsagnet $\neg p \wedge q$?

Det var vel ikke det vi mente, $\cdots$, eller?

Sammensatte utsagn og bruk av parenteser

- Vi vil bruke parenteser for å markere rekkevidden av et konnektiv, det vi si, hva vi mener med p og med q når vi skriver $\neg p$, $p \wedge q$ eller $p \vee q$.
- Vi skal gi en mer formell beskrivelse av hvordan vi skal bruke parenteser senere, men praksis fra skolealgebraen er retningsgivende.

- For eksempel skriver vi
$$\neg(x = 0 \wedge y > 0)$$
hvis vi mener å negere hele konjunksjonen, mens vi skriver
$$\neg(x = 0) \wedge y > 0$$
hvis det bare er $x = 0$ som skal negeres.
- For tydligere å se forskjellen, kan vi regne ut sannhetsverditabellen til de to sammensatte uttrykkene $\neg p \wedge q$ og $\neg(p \wedge q)$.
- En sannhetsverditabell for et sammensatt uttrykk vil være en tabell hvor vi har følgende:
  - En kolonne for hver utsagnsvariabel.
  - En kolonne for hver del av det gitte utsagnet.
  - En rad for hver mulig fordeling av sannhetsverdier på utsagnsvariablene.
  - For hvert delutsagn skriver vi den sannhetsverdien delutsagnet vil ha i hver rad ut fra hvilke sannhetsverdier utsagnsvariablene har.

Eksempel ($\neg(p \wedge q)$).

p	q	$p \wedge q$	$\neg(p \wedge q)$
T	**T**	**T**	**F**
T	**F**	**F**	**T**
F	**T**	**F**	**T**
F	**F**	**F**	**T**

Eksempel ($\neg p \wedge q$).

p	q	$\neg p$	$\neg p \wedge q$
T	**T**	**F**	**F**
T	**F**	**F**	**F**
F	**T**	**T**	**T**
F	**F**	**T**	**F**

Sammenlikner vi høyresidene i de to eksemplene, ser vi forskjellen:

p	q	$\neg(p \wedge q)$	$\neg p \wedge q$
T	**T**	**F**	**F**
T	**F**	**T**	**F**
F	**T**	**T**	**T**
F	**F**	**T**	**F**

- Rekkevidden til tegnet $\neg$ er det minste korrekte utsagnet som står bak tegnet.

- Skal vi negere et sammensatt utsagn, må vi normalt sette parenteser rundt det sammensatte utsagnet.
- Hvis et utsagn med både $\wedge$ og $\vee$ kan forståes på flere måter, må vi bruke parenteser for å presisere rekkeviddene til de to bindeordene.
- Det er ikke noe i veien for å lage sannhetsverditabeller for sammensatte utsagn med tre eller flere utsagnsvariable.
- I det neste eksemplet skal vi se på et sammensatt utsagn $(p \wedge q) \vee (\neg p \wedge r)$.
- Det finnes 8 forskjellige måter å fordele sannhetsverdiene til tre variable på.
- Det betyr at tabellen vår må ha 8 linjer under streken.
- Med fire variable får vi 16 linjer.
- Det vil ikke få plass på skjermen, så da må vi utvikle andre metoder.

Sammensatte utsagn og sannhetsverditabeller

Eksempel ($(p \wedge q) \vee (\neg p \wedge r)$).

p	q	r	$\neg p$	$p \wedge q$	$\neg p \wedge r$	$(p \wedge q) \vee (\neg p \wedge r)$
T	**T**	**T**	**F**	**T**	**F**	**T**
T	**T**	**F**	**F**	**T**	**F**	**T**
T	**F**	**T**	**F**	**F**	**F**	**F**
T	**F**	**F**	**F**	**F**	**F**	**F**
F	**T**	**T**	**T**	**F**	**T**	**T**
F	**T**	**F**	**T**	**F**	**F**	**F**
F	**F**	**T**	**T**	**F**	**T**	**T**
F	**F**	**F**	**T**	**F**	**F**	**F**

"Hvis-så" og "hvis og bare hvis"

Eksempel.

- Hvis $x < 3$, så er $x < 5$.
  - Man skal ikke kunne mye matematikk for å mene at dette må være riktig.
- Utsagnene $x < 3$ og $x < 5$ vil anta forskjellige sannhetsverdier avhengig av hva x er.
- Det vil også det sammensatte utsagnet

$$x < 3 \vee \neg(x < 5).$$

- Er det mulig å definere et utsagnslogisk bindeord $\rightarrow$ slik at:
  - Hvis p og q er utsagn, så vil $p \rightarrow q$ være et utsagn slik at sannhetsverdien til $p \rightarrow q$ avhenger av sannhetsverdiene til p og til q?
  - Når x varierer, skal alltid $x < 3 \rightarrow x < 5$ være sant?

Eksempel (Fortsatt).

- La oss se nærmere på $x < 3 \to x < 5$.
- La $p(x)$ stå for $x < 3$ og la $q(x)$ stå for $x < 5$.
- Hvis $x = 2$ vil både $p(x)$ og $q(x)$ få verdien **T**.
  - $p \to q$ bør bli sann når både p og q er sanne.
- Hvis $x = 4$ vil $p(x)$ få verdien **F**, mens $q(x)$ får verdien **T**.
  - $p \to q$ bør bli sann hvis p er usann mens q er sann.
- Hvis $x = 6$ blir både $p(x)$ og $q(x)$ usanne.
  - $p \to q$ bør også bli sann også når både p og q er usanne.

Eksempel.

- Hvis $x^2 > 0$, så er $x > 0$.
- Mange vil protestere på dette!
- Hvorfor?
- Fordi det finnes moteksempler, eksempelvis $x = -1$.
- Et moteksempel er et eksempel på at en ytring ikke alltid er riktig.
- Et moteksempel til et utsagn "Hvis p, så q" vil alltid være et tilfelle hvor p er sann, mens q er usann.
- Det vil derfor være naturlig å la $p \to q$ være usann når p er sann og q er usann.

Definisjon.

- Hvis p og q er to utsagn, så er $p \to q$ også et utsagn.
- $p \to q$ blir sann hvis q er sann eller hvis p er usann.
- Hvis p er sann og q er usann, lar vi $p \to q$ bli usann.
- Vi vil lese "hvis p, så q".

Vi kan definere $\to$ ved hjelp av følgende sannhetsverditabell.

p	q	$p \to q$
T	**T**	**T**
T	**F**	**F**
F	**T**	**T**
F	**F**	**T**

De som har problemer med at vi her sier at det er sant at noe usant medfører noe sant (eller noe usant) får hente støtte i det kjente sitatet fra Ibsen

hvor Udgangspunktet er galest, blir tidt Resultatet orginalest.

Ibsen fanger her inn essensen av vår definisjon av hvordan sannhetsverdien til

$$\mathtt{Udgangspunkt} \to \mathtt{Resultat}$$

bestemmes av sannhetsverdien til utgangspunktet og til resultatet, og når utgangspunktet er noe som ikke er sant, kan resultatet bli hva som helst.

- Det er ikke så naturlig å bruke $\to$ i programmeringssammenheng. Derfor gir vi ingen eksempler hvor $\to$ brukes i kontrollstrukturer.
- Læreboka bruker ordet *implies* i forbindelse med $\to$.
- Dette er litt uheldig, fordi det lett fører til en sammenblanding av symbolene $\to$ og $\Rightarrow$.
- $x < 5 \Rightarrow x < 3$ er regelrett feil, mens $x < 5 \to x < 3$ er sant for noen verdier av x og usant for andre.
- Dette utdypes mer under forelesningen.

Oppgave.

(a) Finn sannhetsverditabellen til

$$(p \to q) \to p$$

(b) Finn sannhetsverditabellen til

$$(p \to q) \vee (q \to p)$$

(c) Hva ser du i kolonnen lengst til høyre?

- Når vi bruker "hvis-så" i dagligtale, kan vi få noe meningsløst ut av det.
- I de eksemplene som følger kan vi diskutere om tolkningen som logikken forteller oss er riktig stemmer overens med den tolkningen vi vil legge i ytringen som vanlig kommuniserende mennesker:

Eksempel.

- Hvis Noah hadde lært dyrene å svømme, så ville jorda vært overbefolket av løver.
- Hvis ulven spiser Rødhette, vil det bli en Grimm historie.
- Du får gå på kino hvis du vasker opp etter maten.
- Hvis dere avholder reelle demokratiske valg, vil vi gi støtte til oppbyggingen av infrastrukturen.

- I det nest siste eksemplet gis det ikke rom for å få gå på kino hvis oppvasken ikke tas, og i det siste eksemplet er tilbudet om økonomisk støtte helt klart knyttet til kravet om demokrati.
- Oppvask vil være både en nødvendig og en tilstrekkelig betingelse for kinobesøk.

- Vi innfører et siste konnektiv, $\leftrightarrow$ som skal fange opp hvis og bare hvis i samme forstand som $\rightarrow$ fanger opp hvis-så.

Definisjon.

- Hvis p og q er utsagn, er $p \leftrightarrow q$ et utsagn.
- $p \leftrightarrow q$ er sant når både p og q er sanne, og når både p og q er usanne.
- $p \leftrightarrow q$ er sant når p og q har den samme sannhetsverdien.

Vi kan selvfølgelig også definere $\leftrightarrow$ ved en sannhetsverditabell:

p	q	$p \leftrightarrow q$
T	**T**	**T**
T	**F**	**F**
F	**T**	**F**
F	**F**	**T**

Legg merke til at om vi skriver ut sannhetsverditabellene for

- $p \leftrightarrow q$
- $(p \wedge q) \vee (\neg p \wedge \neg q)$
- $(p \rightarrow q) \wedge (q \rightarrow p)$

får vi den samme kolonnen lengst til høyre.

Det er en passende treningsoppgave å skrive ut de to siste tabellene.

Eksempel ($p \rightarrow (\neg p \rightarrow q)$).

p	q	$\neg p$	$\neg p \rightarrow q$	$p \rightarrow (\neg p \rightarrow q)$
T	**T**	**F**	**T**	**T**
T	**F**	**F**	**T**	**T**
F	**T**	**T**	**T**	**T**
F	**F**	**T**	**F**	**T**

Eksempel.

De to neste eksemplene blir bare gjennomgått på tavla:

- $p \rightarrow ((p \vee q) \rightarrow p)$
- $p \wedge (p \rightarrow q) \rightarrow q$

MAT1030 – Forelesning 6

Utsagnslogikk og predikatlogikk

Dag Normann - 3. februar 2010

Kapittel 4: Logikk

Oppsummering

- Vi har nå innført de fem utsagnslogiske bindeordene

$$\wedge, \vee, \neg, \rightarrow, \leftrightarrow .$$

- For hvert av disse bindeordene, eller konnektivene, har vi definert betydningen av dem ved en sannhetsverditabell.
- Vi har sett på hvordan vi kan bygge opp sannhetsverditabeller for sammensatte utsagn.
- Vi fortsetter nå med innføringen av utsagnslogikk.

"En digresjon"

- Hvis vi ønsker å være helt formelle, kan vi definere formelle utsagnslogiske uttrykk på følgende måte, hvor vi skiller mellom variable for grunnutsagn og sammensatte utsagn:
  - Utsagnskonstatene **T** og **F** er utsagn.

    (Som logikere burde vi være enda mer forsiktige her, men vi skal ikke skille mellom en konstant og dens verdi i dette kurset.)
  - Alle utsagnsvariable $p_1, \ldots, p_n$ er utsagn.
  - Hvis p og q er utsagn, er $\neg p$, $(p \wedge q)$, $(p \vee q)$, $(p \rightarrow q)$ og $(p \leftrightarrow q)$ også utsagn.
- En slik definisjon kaller vi en induktiv, eller en rekursiv, definisjon.
- Når vi gir en slik definisjon, begrenser vi bruken av ordet utsagn fra noe vagt, slik vi gjorde det innledningsvis, til noe matematisk presist.
- Vi har en klar parallell i definisjonen av visse programmeringsspråk.
- Vi skal komme tilbake til en mer systematisk drøfting av slike definisjoner litt senere i semesteret.

Oppbygging av utsagn

- Den induktive oppbyggingen av utsagn forteller oss at vi har grunnutsagn og sammensatte utsagn, men også at noen utsagn er mer sammensatte enn andre.
- Når vi kommer til kapitlene om grafer og trær, vil vi se at et sammensatt utsagn kan betraktes som en trestruktur, hvor det gitte utsagnet ligger ved roten, og treet forgrener seg gjennom stadig mindre sammensatte delutsagn, helt til vi finner utsagnsvariablene ved bladene.
- Det første bindeordet vi kommer til når vi skal løse opp et utsagn i delutsagn kalles hovedkonnektivet eller, analogt med i boka, prinsipalkonnektivet.
- Dette illustreres på tavlen.

Mer om parenteser

Eksempel.

$$(p \wedge q \to r) \to (p \to r) \vee (q \to r)$$

- Her mangler det noen parenteser, og for å kunne sette opp sannhetsverditabellen, må vi vite hvilke parenteser som mangler, eller, som er underforstått.
- Vi har tidligere sagt at $\wedge$ og $\vee$ skiller mer enn $\neg$.
- Vi skal også la $\to$ og $\leftrightarrow$ skille mer enn $\wedge$ og $\vee$.
- Det betyr at utsagnet over egentlig skal være

  $$(((p \wedge q) \to r) \to ((p \to r) \vee (q \to r))),$$

  noe som ikke akkurat er lettere å lese.

- Vi skriver ut trestrukturen til dette sammensatte utsagnet på tavlen.
- Som eksempel skriver vi ut en sannhetsverditabell basert på trestrukturen.
- Vi oppdager at kolonnen under det prinsipale konnektivet vil inneholde **T** i alle linjer.
- Det betyr at utsagnet er sant uansett hvilke grunnutsagn vi setter inn for p, q og r.
- Da må $(p \to r) \vee (q \to r)$ være en logisk konsekvens av $p \wedge q \to r$.
  (Vi skal snart definere hva vi mener med logisk konsekvens helt presist.)
- På neste side skal vi se et eksempel på at en ukritisk tolkning av dette i dagligtale gir noe meningsløst.
- La p stå for "Jeg betaler semesteravgiften".
- La q stå for "Jeg får godkjent obligene".
- La r stå for "Jeg kan gå opp til eksamen".
- Da er
  - Hvis jeg betaler semesteravgiften kan jeg gå opp til eksamen eller hvis jeg får godkjent obligene kan jeg gå opp til eksamen.

  en logisk konsekvens av
  - Hvis jeg betaler semesteravgiften og får godkjent obligene kan jeg gå opp til eksamen.

  Er det noe galt her, og i så fall hva?
- Hvordan skal vi forstå utsagn som

  $$p \wedge q \wedge r$$

  og

  $$p \vee q \vee r$$

- I slike tilfeller vil vi få den samme høyrekolonnen i sannhetsverditabellen uansett hvordan vi setter parentesene, så vi kan like godt la det være.

Tautologier og kontradiksjoner

Definisjon.

- La A være et sammensatt utsagn i utsagnsvariablene $p_1, \ldots, p_n$.
- A er en tautologi hvis A får verdien **T** for alle fordelinger av sannhetsverdier på $p_1, \ldots, p_n$, det vil si hvis sannhetsverditabellen til A har bare **T** i høyre kolonne.
- Vi kunne brukt ordene selvoppfyllende eller selvforklarende på norsk, men holder oss til det internasjonalt brukte tautologi.
- Hvis sannhetsverdien til A derimot alltid blir **F**, kaller vi A en kontradiksjon eller en selvmotsigelse.
- En tautologi er, med andre ord, et utsagn som alltid er sant, og en kontradiksjon er et utsagn som alltid er usant.

Eksempel.

- $p \vee \neg p$ er en tautologi.
- $p \wedge \neg p$ er en kontradiksjon.
- $p \to (q \to p)$ er en tautologi.
- $(p \to q) \to ((q \to r) \to (p \to r))$ er en tautologi.
- $(p \leftrightarrow q) \wedge p \wedge \neg q$ er en kontradiksjon.
- $\neg p \to (p \to q)$ er en tautologi.

  Noe av dette har vi regnet på, noe er gitt som oppgaver og resten kan dere godt betrakte som oppgaver.

Logisk ekvivalens

Eksempel ($\neg p \wedge \neg q$).

p	q	$\neg p$	$\neg q$	$\neg p \wedge \neg q$
T	**T**	**F**	**F**	**F**
T	**F**	**F**	**T**	**F**
F	**T**	**T**	**F**	**F**
F	**F**	**T**	**T**	**T**

Eksempel ($\neg(p \vee q)$).

p	q	$p \vee q$	$\neg(p \vee q)$
T	**T**	**T**	**F**
T	**F**	**T**	**F**
F	**T**	**T**	**F**
F	**F**	**F**	**T**

Vi ser at høyrekolonnene er identiske.

Definisjon.

- La A og B være to utsagnslogiske uttrykk.
- Vi sier at A og B er logisk ekvivalente hvis A og B har samme sannhetsverdi uansett hvilke verdier vi gir til utsagnsvariablene.
- Vi skriver

$$A \equiv B$$

når A og B er logisk ekvivalente.

Eksempel.

- $\neg p \wedge \neg q \equiv \neg(p \vee q)$
- $\neg p \vee \neg q \equiv \neg(p \wedge q)$
- $\neg\neg p \equiv p$
- $p \rightarrow q \equiv \neg p \vee q$
- $p \rightarrow (q \rightarrow p) \equiv \mathbf{T}$

Som øvelser bør dere sette opp alle sannhetsverditabellene, og kontrollere at påstandene holder.

Logisk konsekvens

- Logisk ekvivalens er et viktig begrep.
- Logisk konsekvens er et minst like viktig begrep:

Definisjon.

- La A og B være sammensatte utsagn.
- B er en logisk konsekvens av A dersom $A \rightarrow B$ er en tautologi.
- Vi skriver ofte $A \Rightarrow B$ når B er en logisk konsekvens av A.

- Merk at uttrykk som $A \equiv B$ og $A \Rightarrow B$ ligger på utsiden av den formelle utsagnslogikken.

Logiske lover

- Tabell 4.12 på side 56 i læreboka (Side 55 i utgave 2) lister opp en rekke regneregler for utsagnslogikk, kalt "laws of logic".
- Poenget er at man kan regne på et uttrykk ved å bruke disse reglene på deluttrykk, for derved å prøve å forenkle det.
- Det er et faktum (vi ikke skal bevise nå) at vi kan regne oss frem til **T** fra enhver tautologi.
- Vi skal ikke drille inn bruk av disse lovene, men at disse lovene virkelig kan brukes som regneregler, bør bevises, og det skal vi gjøre nå:

Teorem.

- La A være et sammensatt utsagn, og la B være et delutsagn av A.
- La C være et annet utsagn slik at $B \equiv C$ og la D komme fra A ved at vi erstatter en eller flere forekomster av B med C.
- Da er $A \equiv D$.

Bevis.

- I sannhetsverditabellen for A har vi en kolonne for B, og det er bare verdiene i denne kolonnen vi bruker videre.
- Vi ville fått samme sluttresultat om vi hadde brukt en kolonne for C i stedet for den identiske kolonnen for B.
- Dette svarer til å sette opp sannhetsverditabellen for D.

- Noen av de viktigste regnereglene for logikk i boken er:
  - DeMorgans lover:

$$\neg(p \wedge q) \equiv \neg p \vee \neg q$$

$$\neg(p \vee q) \equiv \neg p \wedge \neg q$$

  - Distributive lover:

$$p \wedge (q \vee r) \equiv (p \wedge q) \vee (p \wedge r)$$

$$p \vee (q \wedge r) \equiv (p \vee q) \wedge (p \vee r)$$

- Vi skal snart se på et eksempel på hvordan vi kan vise at et sammensatt utsagn er en tautologi ved å bruke disse regnereglene.
- Vi henviser til betegnelsene i tabellen på side 56 (55).

Strategier

- Det finnes flere metoder eller strategier for å bestemme om et sammensatt utsagn er en tautologi eller ikke.
- Bruk av sannhetsverditabeller er en sikker, men tidkrevende metode.
- Vi skal ikke legge vekt på bruk av regnereglene for logikk her, men hvis man vil bruke dem, kan man gjøre det målrettet, ved å eliminere $\leftrightarrow$ og $\rightarrow$, flytte alle forekomster av $\neg$ så langt inn som mulig og så bruke distributive lover og forkortningsregler.
- Hvis A er et sammensatt utsagn, kan vi "løse likningen" $A = \mathbf{F}$ med hensyn på utsagnsvariablene.

  Hvis likningen ikke har løsning, så er A en tautologi.
- Hva som er mest hensiktsmessig avhenger av hvordan det sammensatte utsagnet ser ut.

Bruk av regneregler

Eksempel ($(p \vee q) \wedge (p \vee \neg q) \rightarrow p$).

- $(p \vee q) \wedge (p \vee \neg q) \rightarrow p$
- $\neg((p \vee q) \wedge (p \vee \neg q)) \vee p$ [Eliminasjon av $\rightarrow$]
- $\neg(p \vee q) \vee \neg(p \vee \neg q) \vee p$ [Bruk av DeMorgan]
- $(\neg p \wedge \neg q) \vee (\neg p \wedge \neg\neg q) \vee p$ [To gangers bruk av DeMorgan]
- $(\neg p \wedge (\neg q \vee \neg\neg q)) \vee p$ [Distributiv lov]
- $(\neg p \wedge \mathbf{T}) \vee p$ [Invers lov]
- $\neg p \vee p$ [Identitetsloven]
- $\mathbf{T}$ [Invers lov]

- Vi antydet også at det er mulig å vise at et sammensatt utsagn A er en tautologi ved å vise at likningen

  $$A = \mathbf{F}$$

  ikke holder.
- Vi skal gi ett eksempel på hvordan vi kan "løse" slike likninger.
- Metoden kan være nyttig når A har mange forekomster av $\rightarrow$.
- Programmeringsspråket *PROLOG* er basert på en systematisering av denne metoden, koblet med predikatlogikk.

Eksempel ($(p \rightarrow q) \rightarrow ((q \rightarrow r) \rightarrow (p \rightarrow r))$).

1. $(p \rightarrow q) \rightarrow ((q \rightarrow r) \rightarrow (p \rightarrow r)) = \mathbf{F}$
2. $p \rightarrow q = \mathbf{T}$ (Fra 1.)
3. $(q \rightarrow r) \rightarrow (p \rightarrow r) = \mathbf{F}$ (Fra 1.)
4. $q \rightarrow r = \mathbf{T}$ (Fra 3.)
5. $p \rightarrow r = \mathbf{F}$ (Fra 3.)

6. $p = \mathbf{T}$ (Fra 5.)
7. $r = \mathbf{F}$ (Fra 5.)
8. $q = \mathbf{F}$ (Fra 4 og 7.)
9. $p = \mathbf{F}$ (Fra 2 og 8)
10. $p \neq p$ (Fra 6 og 9.)

En oppgave

Oppgave.

a) Vis at hvis A, B og C er sammensatte utsagn, så vil

$$(A \leftrightarrow B) \leftrightarrow C \equiv A \leftrightarrow (B \leftrightarrow C)$$

og at

$$(A \leftrightarrow B) \equiv (B \leftrightarrow A).$$

b) Forklar hvorfor dette betyr at rekkefølge og parentessetting ikke betyr noe i et utsagnslogisk uttrykk som bare bruker bindeordet $\leftrightarrow$

c) [Vanskelig] Hvordan kan vi lett avgjøre om et slikt uttrykk er en tautologi eller ikke?

Predikatlogikk

- Utsagnslogikk er enkel i den forstand at gitt et utsagnslogisk uttrykk er det muligens tidkrevende, men i prinsippet enkelt, å avgjøre om vi står overfor en tautologi, en kontradiksjon eller noe annet.
- Utfordringen i utsagnslogikk er å finne algoritmer som raskt kan løse denne typen problemstillinger for sammensatte utsagn (med mange utsagnsvariable) som forekommer i praktiske anvendelser.
- Utsagnslogikken er også enkel i den forstand at den er uttrykksfattig, det er mange tilsynelatende logiske slutninger som ikke kan presses inn i formatet til tautologier.
- Vi skal starte med et eksempel.

Eksempel.

Anta at vi vet følgende:

- All fluesopp er giftig.
- Det fins sopp som ikke er giftig.

Da må vi ha lov til å konkludere med

- Det fins sopp som ikke er fluesopp.

Eksempel.

- Vi vet følgende:
  - Alle kvadrattall er $\geqslant 0$.
  - Det fins tall som ikke er $\geqslant 0$

  Da konkluderer vi med
  - Det fins tall som ikke er kvadrattall.

Dette er det samme argumentet i to forkledninger.

MAT1030 – Forelesning 7

Logikk, predikatlogikk

Dag Normann - 9. februar 2010

Kapittel 4: Logikk (predikatlogikk)

Predikatlogikk

- Vi brukte hele forrige uke til å innføre utsagnslogikk.
- Vi lærte å sette opp sannhetsverditabeller, vi så på sentrale begreper som
  - tautologi
  - kontradiksjon
  - logisk ekvivalens
  - logisk konsekvens

  og vi så på måter vi kan regne med utsagnslogiske uttrykk på.
- Når vi er ferdige med avsnittet om predikatlogikk, skal vi presisere læringsmålene nærmere.
- Vi startet såvidt med predikatlogikk sist uke.
- Vi skal ta opp igjen eksemplet fra onsdag.

Eksempel.

Anta at vi vet følgende:

- All fluesopp er giftig.
- Det fins sopp som ikke er giftig.

Da konkluderer vi med:

- Det fins sopp som ikke er fluesopp.

Eksempel.

Anta at vi vet følgende:

- Alle kvadrattall er $\geqslant 0$.
- Det fins tall som ikke er $\geqslant 0$

Da konkluderer vi med:

- Det fins tall som ikke er kvadrattall.

Dette er det samme argumentet i to forkledninger.

- Da vi innledet utsagnslogikken definerte vi et predikat som en ytring med variable, som ville bli sann eller usann hver gang vi gir variablene verdier.
- I det første eksemplet kan vi betrakte *sopp* som en variabel som kan ta en hvilken som helst sopp som verdi.
- Da blir f.eks. *soppen er giftig* og *soppen er en fluesopp* predikater.
- I det andre eksemplet er *tall* en variabel som kan ta alle hele tall som verdi. Da er *tallet er et kvadrattall* og *tallet er* $\geqslant 0$ predikatene.
- Det gjenstår å betrakte uttrykk som *alle sopper* og *det fins tall* som en del av en utvidet logisk struktur.

Eksempel.

- La $f : [a, b] \to \mathbb{R}$ være en funksjon.
- Hvordan skal vi uttrykke

$$f \text{ har et minimumspunkt?}$$

- *Løsning*:
  Det fins en $x \in [a, b]$ slik at for alle $y \in [a, b]$ vil $f(x) \leqslant f(y)$.

Det å finne egne symboler for det fins og for alle blir mer og mer påtrengende.

- Vi ser på et eksempel til:

$$\textit{Det fins ikke noe største primtall}$$

- Vi prøver med litt utsagnslogikk:

$$\neg\textit{(Det fins et største primtall)}$$

- Det vil si at det er ikke slik at det fins et primtall som er større eller lik alle primtallene.
- Vi trenger et mer formelt språk for å få orden på dette!

Kvantorer

Definisjon.

- Hvis P er et predikat og x er en variabel, vil

$$\exists x P$$

  uttrykke at det fins en verdi av x slik at P holder.

$$\forall x P$$

  uttrykker at P holder for alle verdier x kan ha.

Vi kaller $\exists$ og $\forall$ for kvantorer, og vi regner dem som en del av det formelle logiske vokabularet. Vi skal utdype denne definisjonen senere.

Eksempel.

a)

$$\exists x(x \in [a, b] \wedge \forall y(y \in [a, b] \to f(x) \leqslant f(y)))$$

uttrykker at det fins et minimumspunkt for f på $[a, b]$.

b)

$$\neg\exists x(x \text{ primtall} \wedge \forall y(y \text{ primtall} \to y \leqslant x))$$

uttrykker at det ikke fins et største primtall.

- Det kan være lurt å øve seg på å skrive uttalelser i dagligtale om til utsagn med kvantorer, men for det meste vil vi bruke kvantorer når vi trenger matematisk presisjon i matematikk eller informatikk.
- Vi skal se på noen eksempler på hvordan man oversetter fra dagligtale til formelt språk og omvendt.
- Flere eksempler fins i læreboka.

Eksempel.

- *Alle hunder har lopper, men ikke alle hunder har lus.*
  $\forall x\ (\text{hund } x \rightarrow \exists y(\text{loppe } y \wedge x \text{ har } y)) \wedge$
  $\neg\forall x\ (\text{hund } x \rightarrow \exists y(\text{lus } y \wedge x \text{ har } y))$
- *Alle har et søskenbarn på Gjøvik.*
  $\forall x \exists y(y \text{ bor på Gjøvik} \wedge y \text{ er søskenbarn til } x)$
- *Ingen er bedre enn Tor til å fiske laks*
  $\neg\exists x(x \text{ er bedre enn Tor til å fiske laks })$

Eksempel.

- $\forall x \forall y(\exists z(\text{far}(z, x) \wedge \text{far}(z, y)) \rightarrow \text{brødre}(x, y))$
  Hvis to personer har en felles far, er de brødre.
  Dette er selvfølgelig ikke sant, for de kan være søstre.
- $\forall x \exists y(x \text{ har slått } y \wedge y \text{ har slått } x)$
  La oss si at dette dreier seg om fotballag.
  For alle lag fins det et annet lag slik at de har slått hverandre.
- $\neg\forall x \exists y(y \text{ er bestevennen til } x)$
  Ikke alle har en bestevenn.

Det er ikke alltid at den underforståtte logiske strukturen fremkommer av et utsagn i dagligtale:

Eksempel.

Vi har følgende sitat fra høytaleranlegget til NSB 09.02.2010:
Lokaltog til Skøyen klokken 06.50 er innstilt.
Det skyldes *ikke* tilgjengelig materiell.

Her er det uklart (?) hva *ikke* peker på, og det er en underforstått kvantor: Det fins tilgjengelig matriell.
For å fange opp uttrykk som det skyldes, trenger man modallogikk.
Modallogikk står sterkt på IfI.

Eksempel.
(a) $\exists x \forall y (x \leqslant y)$ (b) $\forall y \exists x (x \leqslant y)$

- Rekkefølgen vi skriver kvantorene i betyr mye for hva utsagnet sier:
  (a) sier at det fins et minste objekt.
  (b) sier at det alltid fins et objekt som er mindre eller lik.
- Hvis x varierer over de hele tallene er a) feil, mens b) holder.
- Hvis x varierer over de naturlige tallene, holder a), og b) holder også, fordi for gitt en verdi for y kan vi bruke samme verdi for x.
- Før vi kan bestemme om et utsagn med kvantorer er sant eller usant, må vi vite hvilke mulige verdier variablene kan ta.
- I en programmeringssammenheng vil vi alltid deklarere datatypen til en variabel, og da kan variabelen ta alle verdier i denne datatypen.

  Når er et utsagn med kvantorer logisk holdbart?

  La oss betrakte følgende eksempel:

Eksempel.

$$\forall x \forall y \forall z ((x < y \wedge y < z) \rightarrow x < z).$$

- Selv om vi ikke har bestemt oss for hvilke verdier x, y og z kan ta, uttrykker dette en sammenheng som vi mener er underforstått når vi bruker symbolet $<$.
  - La x, y og z variere over deltakerne i en sjakkturnering.
  - Hvis S_1 og S_2 er to sjakkspillere, så kan vi si at $S_1 < S_2$ hvis S_1 tapte for S_2 i minst et parti.
  - Det er ofte at vi kan finne tre spillere som "slår hverandre".
  - I denne situasjonen er utsagnet over *ikke sant*.

Definisjon.
- Et predikat er en ytring
$$P(x_1, \ldots, x_n)$$
hvor det kan forekomme variable.
- Hvis P er et predikat og x er en variabel, er $\exists x P$ og $\forall x P$ nye predikater hvor variabelen x er bundet.
- Variable som ikke er bundet kalles frie.
- Hvis vi setter inn (lovlige) verdier for de frie variablene i et predikat får vi et utsagn.
- For å bestemme om et utsagn er sant eller usant må vi bestemme variasjonsområdene til alle variablene samt hva andre symboler skal stå for.

Definisjon (fortsatt).

- En setning er et predikat uten frie variable. Dette kalles også ofte for et lukket utsagn.
- En setning er logisk gyldig dersom den er sann uansett hvilke variasjonsområder vi velger og uansett hva vi lar symbolene bety.

Denne definisjonen er ikke matematisk sett helt presis, men den holder for vårt formål.

Eksempel.

- $x < y \to \neg(y < x)$ er et predikat med to frie variable, x og y.
- $\exists x(x < y \to \neg(y < x))$ er et predikat med en fri variabel y og en bunden variabel x.
- $\forall y \exists x(x < y \to \neg(y < x))$ er en setning, fordi begge variablene er bundne.
- For å bestemme om denne setningen er sann eller usann, må vi bestemme oss for hvilke verdier x og y kan ta, og for hva vi mener med $x < y$.
- Hvis vi lar x og y variere over $\mathbb{Z}$ og $<$ være vanlig ordning, kan vi vise at setningen er sann på vanlig matematisk måte.

Eksempel ($\forall y \exists x(x < y \to \neg(y < x))$).

- Beviset kan formuleres slik;

  La y få en vilkårlig verdi a

  La x også få verdien a.

  Siden $a < a$ er usant, må $\neg(a < a)$ være sant, og sannhetsverdien til

  $$x < y \to \neg(y < x)$$

  blir **T** når vi setter inn a for både x og y.

  Merk at a var vilkårlig da vi satte a inn for y, men valgt med omhu da vi satte a inn for x.
- Dette gir oss ingen grunn til å mene at setningen er logisk gyldig.

- Ved hjelp av læreboka listet vi opp en rekke regneregler for utsagnslogikk.
- Det fins tilsvarende regler for regning med uttrykk med kvantorer.
- En alternativ måte er å isolere noen utsagn i predikatlogikk som aksiomer og fastsette noen regler for hvordan man kan bevise andre utsagn fra disse aksiomene.
- Dette er noe som tas opp på et senere trinn i emner både ved Institutt for Informatikk og ved Matematisk Institutt.
- Vi skal se på et par regneregler som vil være utledbare i en slik logikk, men hvor vi kan overbevise oss om gyldigheten her og nå.

- Vi definerte $\equiv$ som en relasjon mellom utsagnslogiske utsagn, men vil utvide bruken til utsagn med kvantorer, når utsagnene åpenbart er sanne under nøyaktig de samme omstendighetene.

Eksempel (DeMorgans lover for kvantorer).
For alle utsagn A og variable x vil

1. $\neg\forall xA \equiv \exists x\neg A$
2. $\neg\exists xA \equiv \forall x\neg A$

- Noen ganger kan det være lettere å argumentere for en abstrakt påstand ved å gi et dekkende eksempel.
- Vi kan argumentere for 1 ved følgende eksempel som dekker alle andre eksempler:
- Vi mener det samme når vi sier
  - Det er feil at alle russere er katolikker.
  - Det fins en russer som ikke er katolikk.
- Vi kan argumentere for 2 ved følgende eksempel:
- Vi mener det samme når vi sier
  - Det fins ingen ærlig politiker.
  - For alle politikere gjelder det at de ikke er ærlige.

Eksempel (Sammentrekning av kvantorer).
For alle utsagn A og B gjelder

1. $\exists xA \vee \exists xB \equiv \exists x(A \vee B)$
2. $\forall xA \wedge \forall xB \equiv \forall x(A \wedge B)$

- Om vi sier

  Det fins en elev i klassen som spiller tennis
  eller det fins en som spiller badminton

  mener vi det samme som om vi sier

  Det fins en elev i klassen som spiller tennis eller badminton.

- Om vi sier

  Alle arbeiderne fikk høyere lønn
  og alle arbeiderne fikk kortere arbeidstid

  mener vi det samme som om vi sier

  Alle arbeiderne fikk høyere lønn og kortere arbeidstid

- Igjen er disse eksemplene dekkende for den generelle situasjonen.
- Det er *VIKTIG* at man ikke trekker $\exists$ over en $\wedge$ eller en $\forall$ over en $\vee$.

Eksempel (To moteksempler).

- Utsagnet

  Noen Nordmenn er mangemillionærer
  og noen Nordmenn lever under fattigdomsgrensen

  er på formen

  $$\exists x M(x) \wedge \exists x F(x).$$

  Utsagnet

  $$\exists x (M(x) \wedge F(x))$$

  uttrykker at noen Nordmenn både er mangemillionærer og samtidig lever under fattigdomsgrensen.

Eksempel (To moteksempler, fortsatt).

- Den første påstanden er nok sann, mens den andre er heller tvilsom.
- Det betyr at de to utsagnene ikke er logisk ekvivalente.

Eksempel (To moteksempler, fortsatt).

- Utsagnet

  Alle barna får tilbud om å stå slalåm eller å gå langrenn

  er på formen

  $$\forall x (S(x) \vee L(x)).$$

  Utsagnet

  $$\forall x S(x) \vee \forall x L(x)$$

  sier at det er det samme tilbudet til alle barna, mens det første utsagnet gir muligheten for at det er et valg.

  Utsagnene er derfor ikke logisk ekvivalente.

Mer om kvantorer

Eksempel.

- Den tekniske definisjonen av at en funksjon f er kontinuerlig i et punkt x er:

  $$\forall \epsilon \exists \delta \forall y (\epsilon > 0 \rightarrow \delta > 0 \wedge (|x - y| < \delta \rightarrow |f(x) - f(y)| < \epsilon))$$

- Hvis vi skal uttrykke at f ikke er kontinuerlig i x må vi negere denne setningen.
- I første omgang bruker vi deMorgans lover for kvantorene, og får

$$\exists\epsilon\forall\delta\exists y\neg(\epsilon > 0 \to \delta > 0 \wedge (|x - y| < \delta \to |f(x) - f(y)| < \epsilon))$$

Eksempel (Fortsatt).

- Ved deretter å bruke reglene for utsagnslogikk kan vi skrive om

$$\neg(\epsilon > 0 \to \delta > 0 \wedge (|x - y| < \delta \to |f(x) - f(y)| < \epsilon))$$

  til

$$\epsilon > 0 \wedge (\delta \leqslant 0 \vee (|x - y| < \delta \wedge |f(x) - f(y)| \geqslant \epsilon))$$

- Vi har tillatt oss å skrive $\geqslant$ i stedenfor $\neg <$.
- Hele uttrykket blir da

$$\exists\epsilon\forall\delta\exists y(\epsilon > 0 \wedge (\delta \leqslant 0 \vee (|x - y| < \delta \wedge |f(x) - f(y)| \geqslant \epsilon)))$$

Eksempel (Fortsatt).

- Det er usikkert om noen får lyst til å studere analyse etter dette.
- Det illustrerer imidlertid at det krever god kontroll over bruk av kvantorer og konnektiver å kunne finne ut av hva det betyr at en viktig matematisk definisjon *ikke* holder i en gitt situasjon.
- Det illustrerer også at det kan gi bedre leselighet om vi "flytter" noe av det som uttrykkes gjennom utsagnslogikk til en begrensning av virkeområdet til kvantoren:

$$\forall\epsilon > 0\exists\delta > 0\forall y(|x - y| < \delta \to |f(x) - f(y)| < \epsilon)$$

  hvor negasjonen blir

$$\exists\epsilon > 0\forall\delta > 0\exists y(|x - y| < \delta \wedge |f(x) - f(y)| \geqslant \epsilon).$$

Oppsummering

Læringsmålene for kapitlet om logikk er:

1. Definisjonene av utsagn og predikat, og å kunne bestemme hvilke ytringer som er utsagn, hvilke som er predikat og hvilke som faller utenfor rammene våre.
2. De logiske bindeordene $\neg$, $\wedge$, $\vee$, $\to$ og $\leftrightarrow$ og hvordan de defineres via sannhetsverditabeller.
3. Sette opp sannhetsverditabellen til et sammensatt uttrykk, og bruke denne til å bestemme om et utsagn er en tautologi, en kontradiksjon eller ingen av delene.

4. Kjenne til logikkens lover og i noen utstrekning kunne bruke dem.
5. Spesielt sentralt står deMorgans lover og de distributive lovene.
6. Kjenne definisjonene av kvantorene $\forall$ og $\exists$ og kjenne deMorgans lover for kvantorer.
7. Kunne uttrykke en sammenheng ved bruk av kvantorer og kunne "forstå" et uttrykk som inneholder kvantorer.

Relevans for informatikk?

- Et naturlig spørsmål nå vil være om predikatlogikk har noen relevans for informatikk.
- Det er ikke naturlig å bruke kvantorer i testuttrykk i pseudokoder, kontrollstrukturer eller i programmeringsspråk bygget over pseudokodefilosofien.
- Grunnen er at det generelt ikke fins noen algoritme for å bestemme om en setning er sann eller usann.
- Hvis kvantorene skal variere over data lagret i en base, trenger ikke sannhetsverdiene til utsagn med kvantorer å være stabile.
- Vi skal se på to eksempler som antyder hvordan bruk av predikater og til dels kvantorer kan være nyttige i en informatikksammenheng.

MAT1030 – Forelesning 8

Logikk, predikatlogikk, bevisteknikker

Dag Normann - 10. februar 2010

Kapittel 4: Mer predikatlogikk

Oppsummering

Vi gjentar læringsmålene for utsagnslogikk og predikatlogikk:

1. Definisjonene av utsagn og predikat, og å kunne bestemme hvilke ytringer som er utsagn, hvilke som er predikat og hvilke som faller utenfor rammene våre.
2. De logiske bindeordene $\neg$, $\wedge$, $\vee$, $\rightarrow$ og $\leftrightarrow$ og hvordan de defineres via sannhetsverditabeller.
3. Sette opp sannhetsverditabellen til et sammensatt uttrykk, og bruke denne til å bestemme om et utsagn er en tautologi, en kontradiksjon eller ingen av delene.
4. Kjenne til logikkens lover og i noen utstrekning kunne bruke dem.
5. Spesielt sentralt står deMorgans lover og de distributive lovene.
6. Kjenne definisjonene av kvantorene $\forall$ og $\exists$ og kjenne deMorgans lover for kvantorer.
7. Kunne uttrykke en sammenheng ved bruk av kvantorer og kunne "forstå" et uttrykk som inneholder kvantorer.

Anvendelser av predikatlogikk

Vi nevnte kort et område hvor predikatlogikk ikke har noen fornuftig funksjon, i tester for styring av **while**-løkker og liknende.

Vi skal se på et par eksempler hvor bruk av predikatlogikk er nyttig.

I innledningen til kapitlet om logikk trakk vi frem en rekke andre eksempler, eksempler som vi ikke kan gå detaljert inn på.

Eksempel (Kvalitetssikring av databaser).

- Anta at vi skal bygge opp en base for registrering av slektskapsforhold.
- Vi vil registrere noen grunnleggende slektskapsforhold.
- For å sikre oss mot at vi lagrer data på feil måte skal vi sette opp visse aksiomer som dataene våre skal respektere.
- Hvis vi kan utlede en kontradiksjon fra de lagrede slektskapsforholdene og aksiomene, har vi foretatt en feillagring.

Eksempel (Fortsatt).

- Noen aksiomer kan være
  - $\text{Far}(x, y) \wedge \text{Far}(x, z) \rightarrow \text{Søsken}(y, z)$
  - $\text{Mor}(x, y) \wedge \text{Mor}(x, z) \rightarrow \text{Søsken}(y, z)$
  - $\text{Søsken}(x, y) \wedge \text{Far}(x, z) \wedge \text{Mor}(y, z) \rightarrow \mathbf{F}$
- Dette sikrer at vi ikke lagrer søsken som foreldre til det samme barnet.
- Riktignok medfører disse aksiomene at alle er sin egen søsken, men siden ingen av oss vil kunne bli både mor og far til det samme barnet, er kvalitetssikringen ivaretatt uansett.

- Hvis en datamaskin skal gi oss en feilmelding ut fra at de dataene vi har lastet inn leder til en kontradiksjon, må den være programmert til å gjøre det.
- En mulighet er å bruke et spesialkonstruert programmeringsspråk PROLOG til dette formålet.
- Et PROLOG-program vil være en liste av kvantorfrie predikater av en bestemt form, og når programmet kjøres innebærer det å vise at predikatene samlet sett er kontradiktoriske.
- PROLOG har sin egen syntaks, men oversatt til vårt språk kan en PROLOG-instruks være på en av tre former:
  1. $A_1 \wedge \cdots \wedge A_n \rightarrow B$
  2. $A_1 \wedge \cdots \wedge A_n \rightarrow \mathbf{F}$
  3. B
- Svært mange relevante sammenhenger mellom data i en base kan formuleres som en PROLOG-instruks.
- Her vil A_i og B være positive grunnpredikater uten bruk av kvantorer eller bindeord, da heller ikke negasjon.
- Eksempler kan være $x < y$, $\text{Far}(x, y)$, Forbudt(Promillekjøring) og "Per bedriver promillekjøring".
- Vi skal gi et veldig enkelt eksempel på hvordan PROLOG kan brukes til å søke etter data i en base.

Eksempel.

- Vi ser litt nærmere på slektskapsbasen vi så på i sted.
- Vi har lagret informasjon om hvem som er mor eller far til hvem. Annen informasjon må utledes.
- På samme måte som vi innfører *Søsken* som en utledet egenskap, kan vi innføre *Farfar* ved $\text{Far}(x, y) \wedge \text{Far}(y, z) \rightarrow \text{FarFar}(x, z)$,
- Tor oppsøker denne basen og lurer på om han har noen farfar.

Eksempel (Fortsatt).

- I basen er det lagret $\text{Far}(\text{Per}, \text{Tor})$ og $\text{Far}(\text{Knut}, \text{Per})$.
- Programmereren som styrer basen legger til aksiomet

$$\text{FarFar}(x, \text{Tor}) \rightarrow \mathbf{F}$$

- Dette sier at x ikke er farfar til Tor.
- Hvis det leder til en motsigelse, vet Tor at han har en farfar registrert i basen.

Eksempel (Fortsatt).

- PROLOG vil nå målrettet prøve å utlede **F** fra det nye aksiomet og den lagrede informasjonen.
- Gangen vil være omtrent som følger:
  1. Fra $\text{FarFar}(x, \text{Tor}) \rightarrow \mathbf{F}$ og $\text{Far}(x, y) \wedge \text{Far}(y, z) \rightarrow \text{FarFar}(x, z)$ kan vi slutte $\text{Far}(x, y) \wedge \text{Far}(y, \text{Tor}) \rightarrow \mathbf{F}$ ved at vi setter inn Tor for z.
  2. Fra $\text{Far}(x, y) \wedge \text{Far}(y, \text{Tor}) \rightarrow \mathbf{F}$ og $\text{Far}(\text{Per}, \text{Tor})$ kan vi slutte $\text{Far}(x, \text{Per}) \rightarrow \mathbf{F}$ ved at vi setter inn Per for y.
  3. Fra $\text{Far}(\text{Knut}, \text{Per})$ og $\text{Far}(x, \text{Per}) \rightarrow \mathbf{F}$ kan vi slutte **F** ved at vi setter inn Knut for x.
- PROLOG vil ikke bare bevise at Tor har en farfar, men den virker slik at den finner frem til en farfar blant dataene.
- For de som bare leser utskriftene: Endel ble forklart muntlig på forelesningen.

Kapittel 4: Bevisteknikker

Litt repetisjon

- Vi har nå gått gjennom både utsagnslogikk og predikatlogikk.
- Vi innførte

  eksistenskvantoren $\exists$ og

  allkvantoren $\forall$.
- Vi så på en del eksempler på oversettelse mellom dagligtale og uttrykk med kvantorer.
- Vi viste noen logiske ekvivalenser:
  - deMorgans lover: $\neg\exists x A \equiv \forall x \neg A$ og $\neg\forall x A \equiv \exists x \neg A$.
  - Sammentrekning: $\exists x A \vee \exists x B \equiv \exists x (A \vee B)$ og $\forall x A \wedge \forall x B \equiv \forall x (A \wedge B)$.

Bevisteknikker

- Den siste delen av kapittel 4 handler om forskjellige bevisteknikker.

- Vi skal se på måter å strukturere et matematisk bevis på.
- Dette er et tema alle studenter i matematikk eller et annet teoretisk fag etterhvert vil kjenne seg igjen i.
- Vi skal se på direkte bevis, bevis ved tilfeller og kontrapositive bevis.
- Senere skal vi føye induksjonsbevis til vår meny av bevisteknikker.
- Vi skal eksemplifisere de forskjellige bevisformene.

Eksempel.

- Vi skal vise at differensen mellom to kvadrattall som kommer etter hverandre i tallrekken er et oddetall.
- Vi kan formulere dette mer matematisk som en påstand:

$$\text{For alle tall } n \text{ er } (n+1)^2 - n^2 \text{ et oddetall.}$$

Bevis.

- Ved 1. kvadratsetning er $(n+1)^2 = n^2+2n+1$, så $(n+1)^2-n^2 = n^2+2n+1-n^2 = 2n+1$.
- Siden $2n+1$ alltid er et oddetall er påstanden vist.

- I dette eksemplet formulerte vi først det vi skulle vise i en matematisk språkdrakt, deretter regnet vi litt på den differensen vi skulle bevise var et oddetall, og endte opp med at det var akkurat et oddetall det var.
- Hvis vi analyserer beviset litt nærmere ser vi at alle oddetallene kan fremkomme som en slik differens, for å få $2n+1$ som verdi, kan vi velge kvadrattallene $(n+1)^2$ og n^2.
- Når vi først har funnet et bevis, kan vi undersøke om samme metode kan gi oss mer innsikt. Det vil ofte være tilfelle, men kan kreve ekstra innsats.
- La oss se om vi kan bruke samme resonnement til å si noe om differensen mellom kubikktall.

Teorem.

- For alle naturlige tall n er $(n+1)^3 - n^3$ et oddetall.

Bevis.

- Vi har at $(n+1)^3 = n^3 + 3n^2 + 3n + 1$, så

$$(n+1)^3 - n^3 = 3n^2 + 3n + 1.$$

- Hvis n er et partall, er $3n^2 + 3n$ også et partall, så $(n+1)^3 - n^3$ er et oddetall.
- Hvis n er et oddetall, er både $3n^2$ og $3n$ oddetall, så $3n^2 + 3n$ er fortsatt et partall, og også i dette tilfellet er $(n+1)^3 - n^3$ et oddetall.
- Dermed er påstanden bevist.

- Som vi ser, brukte vi akkurat samme resonement i starten av disse to bevisene.
- I det andre beviset måtte vi imidlertid etterhvert dele argumentet opp i to tilfeller, ett for at n er et oddetall og ett for at n er et partall. Siden dette dekker alle mulighetene, er beviset fullstendig.
- La oss gi et tredje eksempel på et bevis hvor vi må dele argumentet opp i tilfeller.

Eksempel.

- La oss bevise følgende påstand:
  Hvis n er et helt tall, så kan $n^2 - n$ deles på 6 eller så kan $n^2 + n$ deles på 6.

Bevis.

- Vi har at $n^2 - n = (n-1)n$ og at $n^2 + n = n(n+1)$.
- Nøyaktig ett av tallene $n-1$, n eller $n+1$ kan deles på 3.
- Hvis $n-1$ kan deles på 3 er ett av tallene $(n-1)$ eller n et partall, og da er $(n-1)n$ delelig med 6.
- Hvis $n+1$ er delelig med 3 er ett av tallene n eller $n+1$ partall, så $n(n+1)$ er delelig med 6.
- Hvis n er delelig med 3 ser vi ved samme argument at både $(n-1)n$ og $n(n+1)$ er delelige med 6.
- Tilsammen beviser dette påstanden.

- Dette eksemplet viser hvordan man enkelte ganger må dele et argument opp i tilfeller.
- Det viser imidlertid også at man av og til må få anta at leseren henger med i noen av svingene, uten at alle detaljene som ligger til grunn for beviset blir tatt med.
- Vi har for eksempel tatt det for gitt at leseren er med på at 6 er faktor i $(n-1)n$ når $n-1$ kan deles på 3 og $n-1$ eller n er et partall.
- Vi har heller ikke minnet om det er fordi $n^2 - n = (n-1)n$ at vi har vist påstanden når vi i realiteten viser at $(n-1)n$ eller $n(n+1)$ kan deles på 6.
- Hvor mange detaljer man tar med er en vurderingsak, og vil være avhengig av målgruppen.

Kontrapositive bevis

- De bevisene vi har sett på til nå kalles direkte bevis.
- Dette er for å skille dem fra såkalte kontrapositive eller indirekte bevis.
- Hvis vi går tilbake til utsagnslogikken, ser vi at utsagnene

$$p \to q$$

og

$$\neg q \to \neg p$$

er logisk ekvivalente.

- Det betyr at hvis vi ønsker å vise en påstand på formen $A \to B$, kan vi like gjerne anta $\neg B$ og bevise $\neg A$.
- Et bevis på denne formen kalles kontrapositivt.
- Før vi gir et eksempel, skal vi se på et generelt spesialtilfelle.
- Det å bevise en påstand A er det samme som å vise $\mathbf{T} \to A$.
- Den kontrapositive varianten vil være å vise $\neg A \to \neg \mathbf{T}$, det vil si $\neg A \to \mathbf{F}$.
- $\mathbf{F}$ er selvmotsigelsen i sin reneste form, så en måte å bevise A på vil være å anta $\neg A$, og så utlede en selvmotsigelse.
- Dette er en vanlig måte å bevise teoremer på, enten ved at man antar at det man skal bevise er usant eller som en hjelp til å bevise påstander man trenger underveis i beviset.
- Det klassiske eksemplet på et kontrapositivt bevis er Pythagoreernes argument for at $\sqrt{2}$ ikke er et rasjonalt tall.
- Dette er gitt som en oppgave i boka, og vi skal la det være med det.
- Vi skal gi tre eksempler på kontrapositive bevis.
- Det første er bare for å illustrere metoden, mens det siste viser et meget viktig resultat som berører forståelsen av informatikkens begrensninger.

Teorem.
Alle naturlige tall > 1 er primtall eller kan faktoriseres i primtall.

Bevis.

- Anta at $n > 1$ hverken er et primtall eller kan faktoriseres i primtall.
- Siden n ikke er et primtall, kan n skrives som et produkt $n = ab$ hvor $1 < a < n$ og $1 < b < n$.
- Hvis både a og b enten er primtall eller kan faktoriseres i primtall, vil n kunne faktoriseres i primtall.

Bevis (Fortsatt).

- Det har vi antatt at ikke er tilfelle.
- Derfor fins det $n_1 < n$ slik at $1 < n_1$ og slik at n_1 hverken er et primtall eller kan faktoriseres i primtall.
- Da kan vi fortsette dette resonementet n ganger, og får $1 < n_n < n_{n-1} < \ldots < n_1 < n$ slik at ingen av disse tallene er primtall eller kan faktoriseres i primtall.
- Dette er umulig, siden det ikke fins så mange tall mindre enn n.
- Derfor må forutsetningen, som var at teoremet vårt ikke holder, være feil.

Når vi har lært om induksjonsbevis, vil vi kunne bevise dette på en måte som virker mer overbevisende på matematikere.

Teorem.

La n og m være positive heltall, og la c være den største felles faktoren i n og m.

Da finnes det hele tall a og b slik at

$$c = an + bm.$$

Bevis.

La d være det minste positive hele tallet slik at d kan skrives på formen

$$d = an + bm$$

hvor a og b er heltall.

Bevis (fortsatt).

Siden c er en faktor i både n og m, er c en faktor i d, så $c \leqslant d$.

Det er derfor nok å vise at d er en faktor både i n og i m.

Anta at det ikke er tilfelle.

Ved symmetri kan vi anta at d ikke er faktor i n, og hvis vi deler n på d får vi en rest $r < d$.

Da finnes det k, a og b slik at

$$r = n - kd = n - k(an + bm) = (1 - ka)n + (-kb)m.$$

Dette er umulig siden d er det minste positive tallet som kan skrives som en kombinasjon av n og m.

Dermed er teoremet bevist.

Et langt bevis

- Vi skal vise at under svært generelle forutsetninger er det ikke mulig å løse noen helt grunnleggende problemer om egenskaper ved programmer.
- Vi skal la $\mathcal{P}$ være et programmeringsspråk som har følgende egenskaper:
  - Det er mulig å skrive et program for en prosedyre som i teorien aldri stopper (mens $x \geqslant 0$, sett $x = x + 1$).
  - Vi kan la et program P_2 etterfølge et program P_1 ved en konstruksjon som
    $$P_1; P_2.$$
  - Vi kan skille mellom tilfeller som i **If** ... **then** ... **else**.
  - En hvilken som helst tekstfil, eksempelvis et program, kan tjene som input.
  - Det fins et program for kopiering av en fil, det vil si, som til en inputtekst t gir output tt, dvs t og en kopi av t.

Teorem.

- La $\mathcal{P}$ være et programmeringsspråk med egenskapene på forrige side.
- Da fins det ikke noe program Q for å avgjøre om et annet program P med input t vil stoppe eller fortsette i det uendelige.

Bevis.

- Med et program vil vi her mene tekstfilen som utgjør programmet.
- Hvis P er et program og t er et mulig input, skriver vi $P(t)$ for tilsvarende output.
- Anta at teoremet er feil, og at det fins et program Q slik at om P er et annet program og t er et mulig input, så vil
  - $Q(Pt) = 1$ om $P(t)$ har en verdi, dvs P med input t stopper.
  - $Q(Pt) = 0$ om $P(t)$ i teorien aldri stopper.

Bevis (Fortsatt).

- La C være et program slik at $C(t) = tt$ for alle t og la U være et program som aldri stopper uansett input.
- La R være et program som svarer til
- **If** $Q(C(t)) = 1$ **then** U **else** output 1.
- Husk at R også er tekstfilen til R.
- Anta at $R(R)$ stopper. La oss se på hvordan beregningen til $R(R)$ må se ut.
- Vi har at $Q(C(R)) = Q(RR) = 1$ fordi $R(R)$ stopper.
- Men etter den testen, fortsetter R med U, det vil si at $R(R)$ ikke stopper.

Bevis (Fortsatt).

- Men hvis $R(R)$ ikke stopper, er $Q(C(R)) = 0$, så R gir output 1, hvilket betyr at $R(R)$ stopper likevel.
- Når vi konstruerer R på denne måten, vil beregningen av $R(R)$ stoppe hvis og bare hvis beregningen aldri stopper.
- Dette er en motsigelse, og konklusjonen må være at det ikke fins noe program Q som følger spesifikasjonene.

Konstruktive bevis

- I informatikk kan det ofte være lurt å bruke konstruktive bevis.
- Det betyr at man ikke ukritisk gjør bruk av tautologier som $p \vee \neg p$, men at man skal ha kontroll på hvilken av de to delene som holder.
- Kontrapositive bevis har heller ingen plass i konstruktiv matematikk.
- Fordelen med konstruktive bevis er at man kan trekke algoritmer og annen form for informasjon ut av bevisene.
- Det klassiske beviset for at det alltid fins et større primtall er konstruktivt.
- Vi skal se et eksempel på et bevis som ikke er konstruktivt.

Teorem.
Det fins to irrasjonale tall a og b slik at $a^b \in \mathbb{Q}$.

Bevis.
Vi deler beviset opp i to tilfeller.

- Tilfelle 1: $\sqrt{2}^{\sqrt{2}} \in \mathbb{Q}$.
  Da lar vi $a = b = \sqrt{2}$ og $a^b \in \mathbb{Q}$.
- Tilfelle 2: $\sqrt{2}^{\sqrt{2}} \notin \mathbb{Q}$.
  Da lar vi $b = \sqrt{2}$ og $a = \sqrt{2}^{\sqrt{2}}$.
- Da får vi
$$a^b = (\sqrt{2}^{\sqrt{2}})^{\sqrt{2}} = \sqrt{2}^{(\sqrt{2}\cdot\sqrt{2})} = \sqrt{2}^2 = 2 \in \mathbb{Q}.$$

- Dette eksemplet blir ofte brukt til å illustrere forskjellen på konstruktive bevis og bevis basert på klassisk logikk.
- I klassisk logikk kan vi gjøre uhemmet bruk av antagelsen at enten gjelder en påstand P eller så gjelder negasjonen $\neg P$.

- I konstruktiv matematikk kreves det at vi i tillegg har noe informasjon om hvilken av de to som gjelder, eller i det minste en metode for å avgjøre hvilken av de to som gjelder i en gitt situasjon.
- Ut fra det beviset vi har sett på kan vi ikke si noe sikkert om hvilket par a, b av irrasjonale tall det er som er slik at $a^b \in \mathbb{Q}$, bare at det fins et slikt par.
- Vi ga tidligere et bevis for at hvis c er største felles faktor til n og m, så finnes det heltall a og b slik at
$$c = an + bm.$$
- Hvis vi ser nærmere på det beviset, ser vi at vi ikke har noen informasjon om hvilke tall a og b er.
- Dette teoremet vises vanligvis ved å ta utgangspunkt i Euklids algoritme som vi har eksemplifisert ved en pseudokode.
- Da blir beviset mer konstruktivt, og vi kan utlede en algoritme for å finne a og b fra det beviset.
- Skal vi være helt ærlige, er det egentlig det samme beviset i to utgaver.

MAT1030 – Forelesning 9

Mengdelære

Dag Normann - 16. februar 2010

Kapittel 5: Mengdelære

Oppsummering

- Vi er nå ferdige med kapittel 1–4.
- Vi har dekket
  - algoritmer og pseudokoder
  - tall og tallsystemer
  - litt om representasjon av data
  - utsagnslogikk
  - predikatlogikk
  - litt om hvordan bevis kan være strukturert.
- Noen spørsmål før vi går over til kapittel 5?

Mengder

- De fleste som tar MAT1030 har vært bort i mengder i en eller annen form tidligere.
- I statistikk og sannsynlighetsteori på VGS behandler man utfallsrom og studerer matematiske sannsynligheter eller sannsynlighetsfordelinger basert på eksperimenter på slike utfallsrom.
- Man ser på delmengder av slike utfallsrom og eksempelvis Bayes setning, som omhandler både det vi vil kalle komplement og det vi vil kalle snitt (og med de betegnelsene).
- Mengdebegrepet brukes også i de innledende emnene på universitetet, eksempelvis i form av løsningsmengder for ulikheter.

- En mengde er en samling objekter hvor det er entydig bestemt hvilke objekter som er med i mengden eller ikke.
- Bruken av mengder gjennomsyrer matematikk og andre teoretiske fag.
- Vi skal lære å bruke mengder slik at vi kan uttrykke oss presist om konstruksjoner og begreper av interesse i informatikk i en vid forstand.
- Mengdelære brukes på mange måter som matematikkens grunnlag, noe vi ikke skal legge så stor vekt på.

Vi bruker klammeparenteser { og } for å beskrive mengder.

Vi skal illustrere bruken ved eksempler.

Eksempel.

- $\{0, 1\}$ er mengden av digitale verdier en bit kan ha.
- $\{\mathbf{T}, \mathbf{F}\}$ er mengden av sannhetsverdier.
- $\{2, 3, 5, 7, 11, 13, 17, 19, 23, 29\}$ er mengden av de 10 minste primtallene.
- $\mathbb{N} = \{1, 2, 3, \cdots\}$ er mengden av naturlige tall.

- I de tre første eksemplene har vi listet opp mengdens elementer.
- Mengdene er endelige og i disse tilfellene så små at vi kan beskrive mengden ved hjelp av en liste med klammeparenteser rundt.
- Vi sier at vi har gitt mengden på listeform.
- I en viss forstand har vi beskrevet $\mathbb{N}$ ved hjelp av en liste. I dette tilfellet er listen uendelig lang, og vi har brukt prikkene $\cdots$ for å antyde at opplistingen fortsetter.
- Denne mengden er også gitt på listeform (Læreboka: Enumerated form).
- Bruker vi listeform med prikker eller tilsvarende, må vi være sikre på at leseren vil oppfatte prikkene på samme måte som forfatteren.
- Hvordan ser mengden $\{1, 5, 15, 34, 65, \cdots\}$ ut?
  - Hjelper det å få vite at neste tall er 111?
  - Kunne vi startet med $\{0, 1, \ldots\}$ og forøvrig fått den samme tallmengden basert på den samme regelen?
- Vi trenger et eget symbol for å uttrykke at et objekt er et element i en mengde.

Definisjon.

a) Vi skriver $a \in A$ for å uttrykke at a er et element i mengden A.
b) Vi skriver $a \notin A$ for å uttrykke at a *ikke* er et element i A.

- Vi kunne ha skrevet $\neg(a \in A)$ i stedenfor.
- Det er ikke uvanlig å bruke $/$ som nektingssymbol, som i $3 \not< 2$ og $4 \neq 3$.
- Hvis vi skal beskrive noe mer kompliserte mengder, må vi bruke et litt annet format enn listeform, kalt predikatform i læreboka.
- Dette illustreres også best ved noen eksempler.

Eksempel.

- $\{n \in \mathbb{N} : n \text{ kan deles på } 5\}$
  Mengden av n i $\mathbb{N}$ slik at n kan deles på 5.
- $\{n \in \mathbb{N} : n \text{ er et primtall}\}$
  Mengden av n i $\mathbb{N}$ slik at n er et primtall.

Vi kan uttrykke dette mer formelt som følger.
Da blir det vanskeligere å uttrykke definisjonen med ord.

Eksempel.

- $\{n \in \mathbb{N} \ : \ \exists k \in \mathbb{N}(n = 5k)\}$
  Mengden av n i $\mathbb{N}$ slik at det fins en k i $\mathbb{N}$ slik at $n = 5k$.
- $\{n \in \mathbb{N} \ : \ n \geqslant 2 \wedge \forall m \in \mathbb{N} \forall k \in \mathbb{N}(n = km \rightarrow k = 1 \vee m = 1)\}$
  Mengden av de n i $\mathbb{N}$ som er slik at n er større eller lik 2, og slik at for alle m og k i $\mathbb{N}$ har vi at hvis $n = km$ så er $k = 1$ eller $m = 1$.

Vi ser at den formelle definisjonen faktisk er lettere å lese, mens definisjonen fra forrige side vel egentlig var den som var enklest å oppfatte.
Hvordan vi formulerer definisjonen av en mengde avhenger av hva vi vil kommunisere.

- Vi kaller ofte bruken
  $$\{\cdot : \cdots\}$$
  av parenteser for mengdebyggeren.
- Læreboka bruker kolon, :, i mengdebyggeren.
- Det er like vanlig, om ikke vanligere, å bruke en vertikal strek, | i stedenfor.
- Da ser mengdebyggeren ut som
  $$\{\cdot \mid \cdots\}.$$
- Det er mulig at foreleser, av gammel vane, kommer til å bruke | i stedenfor : noen ganger.
- Noen mengder vil vi referere til så ofte at vi bruker egne symboler for dem.
- Vi har allerede sett at vi bruker $\mathbb{N}$ for mengden av naturlige tall.
- Vi lar $\mathbb{J} = \{\cdots . -3, -2, -1, 0, 1, 2, 3, 4, \cdots\}$ være mengden av hele tall.
  Her følger vi boka, men internasjonalt (og i Norge) er det mer vanlig å bruke $\mathbb{Z}$ for denne mengden.
- Vi lar $\mathbb{R}$ stå for mengden av alle reelle tall.
  Det er så mange reelle tall at vi vanskelig kan liste dem opp, ikke engang ved hjelp av $\cdots$.
- Vi lar $\mathbb{Q}$ betegne mengden av rasjonale tall.
- Det er mulig å definere $\mathbb{Q}$ fra de andre mengdene.
  $$\mathbb{Q} = \{x \in \mathbb{R} \ : \ \exists p \in \mathbb{J} \ \exists q \in \mathbb{N} \ (x = \frac{p}{q})\}.$$
- Det er også mulig å skrive $\mathbb{Q}$ på listeform:
  $$\mathbb{Q} = \{0, -1, 1, -2, -\frac{3}{2}, -\frac{1}{2}, \frac{1}{2}, \frac{3}{2}, 2, -3, -\frac{8}{3}, -\frac{7}{3}, \cdots\}$$
- I en viss forstand er det umulig å definere $\mathbb{N}$ utfra $\mathbb{R}$ alene.
  - Den forstanden faller langt utenfor pensum i MAT1030.
- Vi har allerede truffet på endel mengder i dette emnet, uten å legge vekt på at de er mengder.

Eksempel.

- La VAL_k være mengden av fordelinger av sannhetsverdier **T** og **F** på k utsagnsvariable $p_1, \ldots, p_k$.
- La var være en uendelig mengde $\{p_1, p_2, \ldots\}$ av utsagnsvariable.

  Vi kan arbeide med denne mengden uten å vite nøyaktig hva disse variablene er, det holder at vi har navn for dem.
- La $\mathrm{REP}_{\mathbb{R}}$ være mengden av digitale representasjoner av reelle tall, og la $\mathrm{REP}_{\mathbb{J}}$ være mengden av digitale representasjoner av hele tall som ikke er den samme mengden som mengden av binære representasjoner av hele tall.
- Da vi snakket om hvordan vi skulle "forstå" kvantorer $\exists x$ og $\forall y$, sa vi at vi måtte presisere hvilke verdier x og y kan ta.

  Dette kan vi gjøre mer presist ved å snakke om variasjonsmengden eller tolkningsområdet til den enkelte variable som en mengde.

- Senere skal vi se på datastrukturer som mengder.
- En datastruktur vil bestå av de objektene en variabel i et program kan ta som verdi.
- Vi har tidligere nevnt at vi ofte må deklarere typen til en variabel.
- Dette betyr at vi må avgrense den delen av datastrukturen som den aktuelle variabelen kan hente sine verdier fra.
- For å kunne beskrive datastrukturer, datatyper og for å kunne diskutere hvordan, og i hvilket omfang, objektene i slike strukturer kan representeres som digitalisert informasjon, trenger vi et grunnlag i mengdelære.
- Det er dette grunnlaget vi skal få når vi gjennomgår stoffet i Kapittel 5.

- Vi skal ikke kaste oss inn i den filosofiske diskusjonen om hva en mengde *er* i noen større grad enn i diskusjonen om hva et tall er for noe.
- Det viktigste for oss er å forstå hvordan vi bruker mengder til å uttrykke oss presist.
- Hvis man noen gang har tenkt til å diskutere teoriproblemer med noen andre, er det viktig at man har den samme forståelsen.
- En viktig del av denne forståelsen er å vite når to mengder er like.

Definisjon.
To mengder A og B er like hvis de har nøyaktig de samme elementene.

- Det betyr at mengden er fullstendig bestemt av sine elementer, og det betyr ikke noe hvordan vi beskriver den.

Eksempel.

- $\{2,3,4\} = \{3,4,2\} = \{2,2^2,3\} = \{1+1,2,2+1,3,3+1,2+2,4\}$
- $\{3,4,7\} = \{\text{III},\text{IV},\text{VII}\}$ så lenge det er klart at vi bruker arabiske og romerske måter å uttrykke tall på, mens hvis vi snakker om de konkrete symbolsekvensene er mengdene forskjellige.
- $$\{(x,y) \in \mathbb{R}^2 \,:\, 9x^2 + 16y^2 = 25\} =$$
  $$\{(x,y) \in \mathbb{R}^2 \,:\, (\frac{3x}{5})^2 + (\frac{4y}{5})^2 = 1\}.$$

Eksempel (Tre "dumme" oppgaver).

- Finn
  $$\{x \in \mathbb{R} \,:\, x^4 + 4x^3 + 8x^2 + 8x + 4 = 0\}$$
- Bestem mengden av sannhetsverdifordelinger som gjør
  $$(p \vee q) \wedge (\neg p \wedge \neg q)$$
  sann.
- Finn mengden av flytende-punkt representasjoner av det reelle tallet 0.

- Felles for alle disse tre oppgavene er at mengdene ikke har noen elementer.
- Alle løsningsmengdene har nøyaktig de samme elementene, nemlig ingen, og er derfor like.

Definisjon.
Vi lar $\emptyset$ betegne den tomme mengden, det vil si mengden som ikke har noen elementer.

- Symbolet $\emptyset$ er internasjonalt gjennomført, og man vil ikke finne det forklart andre steder enn i innføringstekster i mengdelære.
- Det er fremkommet ved en "null" med en strek over, og må ikke forveksles med noen bokstav i noe alfabet.
- Vi har at $\emptyset \neq \{\emptyset\}$; den første mengden har ingen elementer mens den andre har ett element, nemlig $\emptyset$ selv.
- Vi har tidligere sagt at når vi bruker en variabel i en programmeringssammenheng, må vi deklarere *typen* til variabelen.

  (Ikke alltid sant; det kan avhenge av programmeringsspråket, men for mange ikke-spesialiserte språk er det tilfelle.)

- Alternativt arbeider vi med en datastruktur hvor vi henter alle variabelverdier fra.
- Vi har ofte definert mengder som $\{n \in \mathbb{N} \ : \ \cdots\}$ eller $\{x \in \mathbb{R} \ : \ \cdots\}$. I slike tilfeller er det ofte klart fra sammenhengen hvilke mengder n eller x skal hentes fra, og det kan være brysomt å måtte presisere det hver gang.
- Hvis A er en mengde, og vi vil se på mengden av objekter som ikke er med i A, vil vi normalt ønske å avgrense oss til de objektene som er av interesse i den aktuelle sammenhengen.
- Alt dette gjør det aktuelt å innføre et eget symbol for en universell mengde, uten at denne universelle mengden trenger å være den samme i enhver sammenheng.

Definisjon.

a) Vi lar $\mathcal{E}$ stå for den universelle mengden.

b) $\mathcal{E}$ vil betegne forskjellige mengder i forskjellige sammenhenger, men skal ligge fast i enhver gitt sammenheng.

c) I noen sammenhenger (definisjoner, oppgaver) vil $\mathcal{E}$ bare betegne en vilkårlig universell mengde, mens i andre sammenhenger vil betydningen av $\mathcal{E}$ bli presisert.

Vi bruker $\mathcal{E}$ fordi læreboka gjør det. Det er imidlertid ingen fast notasjon for den universelle mengden uavhengig av forfatter og anvendelsesområde slik det er med $\emptyset$ for den tomme mengden.

Mengdealgebra

- Vi skal nå se på noen grunnleggende operasjoner på mengder.
- Disse kalles Boolske operasjoner.
- Det er en nær sammenheng mellom Boolske operasjoner og utsagnslogiske bindeord.
- Utsagnslogikk og Boolsk mengdelære er begge instanser av det som kalles Boolsk algebra.
- Et mer systematisk studium av Boolsk algebra er relevant for retninger innen teoretisk informatikk.

Mengdealgebra – union

Eksempel.

- Vi har fått i oppgave å finne en digital representasjon av funksjonen

$$f(x) = \sqrt{x^2 - 1}$$

  og vet at denne funksjonen bare er definert når $|x| \geqslant 1$.
- Det betyr at vi må finne mengden av 32-bit representasjoner av tall $\geqslant 1$ og av tall $\leqslant -1$ og så slå disse mengdene sammen.
- En slik sammenslåing kaller vi en union.

Definisjon.

- Hvis A og B er to mengder, definerer vi unionen av A og B som

$$A \cup B = \{x \,:\, x \in A \vee x \in B\}.$$

- Unionen av A og B består altså av de objektene som er element i minst en av de to mengdene A og B, men også gjerne i begge.

Eksempel.

- $\mathbb{N}$ er unionen av mengdene av positive partall og positive oddetall.
- Løsningsmengden til ulikheten $x^2 > 1$ er

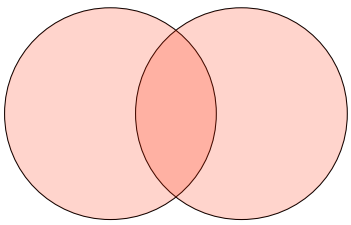
$$\{x \,:\, x < -1\} \cup \{x \,:\, x > 1\}.$$

- Mengden av 32-bitsekvenser som representerer tall x med $|x| \geqslant 1$ kan skrives som

$$\{01\sigma \,:\, \sigma \in \{0,1\}^{30}\} \cup \{11\sigma \,:\, \sigma \in \{0,1\}^{30}\}$$

  hvor $\{0,1\}^{30}$ er en vanlig måte å uttrykke mengden av 30-bits sekvenser.

- En vanlig måte å illustrere Boolske operasjoner på er ved å bruke Venndiagrammer.
- Et Venndiagram for en Boolsk kombinasjon av to eller tre mengder vil bestå av en sirkel for hver mengde, slik at de overlapper hverandre.
- Hvis vi har bruk for å markere den universelle mengden $\mathcal{E}$, gjør vi det ved et rektangel som omslutter alle sirklene.
- Ved å bruke forskjellige skraveringer, kan vi illustrere hvilke punkter som ligger i mengden og hvilke som ikke gjør det.



$A \cup B$

- Vi har en sirkel for hver mengde.
- Vi har fire felter, et for hver kombinasjon av $x \in A$ og $x \in B$.
- Hele det fargede området markerer $A \cup B$.

Mengdealgebra – snitt

Eksempel.

- La A være mengden av naturlige tall n slik at n kan deles på både 6 og 8.
- La B være mengden av reelle tall x slik at

$$((x-1)^2-1)^2+(x^4-16)^4=0$$

- La C være mengden av digitale representasjoner av ikke-negative partall.
- I tilfelle A gir vi to krav direkte.
- I tilfelle B krever vi at vi både må ha at $(x-1)^2-1=0$ og at $x^4-16=0$.
- I tilfelle C krever vi at bitsekvensen må ha den foreskrevne lengde, og både må starte med 0 og ende med 0.

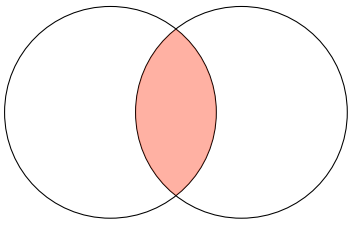
Definisjon.

- La A og B være to mengder.
- Med snittet av A og B mener vi

$$A\cap B=\{x\,:\,x\in A\wedge x\in B\}.$$

- $A\cap B$ består altså av de objektene som er elementer både i A og i B.

Vi kan også illustrere snitt ved et Venndiagram.



$A\cap B$

- Vi farger eller skraverer det feltet som markerer fellesdelen av de to mengdene.

Eksempel.

- Hvis

$$A=\{0,1,2,3,4,5\}$$
$$B=\{1,3,5,7,9\}$$

  er $A\cup B=\{0,1,2,3,4,5,7,9\}$, mens $A\cap B=\{1,3,5\}$.
- Hvis

$$A = \{(x, y) \in \mathbb{R}^2 \ : \ x^2 + y^2 \geqslant 1\}$$
$$B = \{(x, y) \in \mathbb{R}^2 \ : \ x^2 + y^2 \leqslant 1\}$$

er $A \cup B$ hele planet, mens $A \cap B$ er mengden av punkter på enhetssirkelen.

Mengdealgebra – komplement

- Vi har ikke hatt bruk for den universelle mengden $\mathcal{E}$ i definisjonen av union og snitt.
- Vi har formulert definisjonen av $\cup$ og $\cap$ slik at sammenhengen med $\vee$ og $\wedge$ skal komme klart frem.
- Den neste mengdeoperasjonen vi skal se på er komplement.
- Den vil ha et nært slektskap til $\neg$.
- For å definere komplement må vi ha tilgang til en universell mengde.

Eksempel.

1. La P betegne mengden av primtall.

   Et sammensatt tall er et naturlig tall $\neq 1$ som ikke er et primtall.

   Vi kan definere
   $$S = \{n \in \mathbb{N} \ : \ n \notin P \cup \{1\}\}.$$
   Det hadde vært enklere om vi kunnet skrive dette på en kortere måte som vi alle kunne forstå.

Eksempel (Fortsatt).

2. Vi definerer de irrasjonale tallene som mengden av de reelle tallene som ikke er rasjonale.

   Dette kunne vi skrevet som
   $$\{x \in \mathbb{R} \ : \ x \notin \mathbb{Q}\}.$$
   Hvis alle forstår oss om vi skriver $\overline{\mathbb{Q}}$ istedenfor, ville det vært greiere.

Eksempel (Fortsatt).

3. Vi har ord både for partall og for oddetall, men vi har ikke et eget ord for tall som ikke kan deles på 13 (eller 17 for den saks skyld).

   I denne sammenhengen var det klart at vi snakker om hele tall, dvs
   $$\mathcal{E} = \mathbb{J}.$$

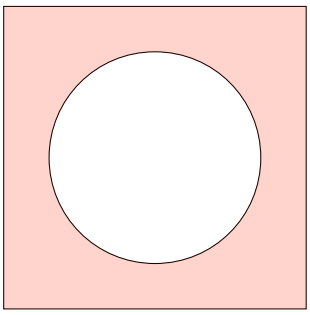
Vi vil si at mengden av tall som ikke kan deles på 13 er komplementet til mengden av tall som kan deles på 13, og vi markerer komplementet til en mengde ved å sette en strek over uttrykket for mengden.

Definisjon.

La $\mathcal{E}$ være en universell mengde og la A være en mengde hvor alle elementene ligger i $\mathcal{E}$.
Med komplementet til A mener vi

$$\overline{A} = \{x \in \mathcal{E} \ : \ x \notin A\} = \{x \in \mathcal{E} \ : \ \neg(x \in A)\}.$$

Når vi skriver $\overline{A}$ skal det alltid være klart at $\mathcal{E}$ er kjent
(eller at vi arbeider generelt med en vilkårlig $\mathcal{E}$.)
Vår notasjon for komplement er ikke uvanlig, men på ingen måte enerådende.
Vi kan beskrive komplementet ved et Venndiagram.



$\overline{A}$

- Det fargede/skraverte feltet markerer komplementet.

Eksempel.

- Hvis $\mathcal{E} = \{0,1,2,3,4,5,6,7\}$ og $A = \{2,3,5\}$, er $\overline{A} = \{0,1,4,6,7\}$.
- Hvis $\mathcal{E}$ er som over og $B = \{1,3,5,7\}$, er $\overline{B} = \{0,2,4,6\}$
- Hvis $\mathcal{E} = \mathbb{J}$ og $B = \{1,3,5,7\}$ er

$$\overline{B} = \{\cdots, -3, -2, -1, 0, 2, 4, 6, 8, 9, 10, \cdots\}.$$

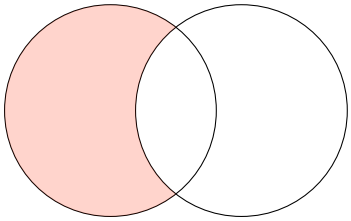
Mengdealgebra – mengdedifferens

- Den siste mengdeoperasjonen vi skal innføre er mengdedifferens.

Definisjon.
Hvis A og B er to mengder, er differansen A minus B definert ved

$$A - B = \{x : x \in A \wedge x \notin B\}$$

Vi bruker ofte betegnelsen mengdedifferens
En alternativ skrivemåte mye brukt i litteraturen er $A \setminus B$.
Vi kan også illustrere mengdedifferens ved et Venndiagram.



$A - B$

- Det fargede/skraverte området markerer differensen
- Vi har ikke hatt bruk for $\mathcal{E}$ her.
- $A - B = A \cap \overline{B}$

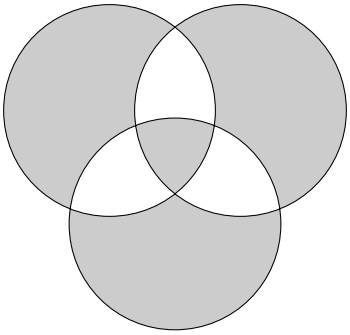
Venndiagrammer

- Den virkelige nytten av Venndiagrammer ligger i at de kan brukes til å studere sammenhengen mellom forskjellige Boolske uttrykk.
- I en viss forstand er bruk av Venndiagrammer en parallell til bruk av sannhetsverditabeller.
- Vi kommer til å vise noen eksempler på tavlen på hvordan vi kan etablere mengdeteoretiske identiteter ved hjelp av Venndiagrammer.

Oppgave.
Vi definerer ofte symmetrisk differens ved

$$A \triangle B = (A - B) \cup (B - A).$$

a) Illustrer $A \triangle B$ ved et Venndiagram.
b) Vis at $(A \triangle B) \triangle C$ kan illustreres ved Venndiagrammet på neste side.
c) Drøft hvorfor dette viser at vi kunne skrevet $A \triangle B \triangle C$ uten bruk av parenteser.



$(A \triangle B) \triangle C$

MAT1030 – Forelesning 10

Mengdelære

Dag Normann - 17. februar 2010

Kapittel 5: Mengdelære

Oversikt

- Tirsdag snakket vi først litt om mengder, og om hvordan vi beskriver en mengde.
- Vi har innført de Boolske operasjonene,
  - union $\cup$: $A \cup B = \{x : x \in A \vee x \in B\}$
  - snitt $\cap$: $A \cap B = \{x : x \in A \wedge x \in B\}$
  - komplement $\overline{A}$: $\overline{A} = \{x \in \mathcal{E} : x \notin A\}$
  - mengdedifferens $A - B$: $A - B = \{x : x \in A \wedge x \notin B\}$

  samt de faste mengdene $\emptyset$ og $\mathcal{E}$.
- Vi tegnet Venndiagrammet tilhørende de forskjellige Boolske operasjonene, og begynte på et eksempel på litt mer avansert bruk av Venndiagrammer.
- Vi fortsetter med flere eksempler (på tavlen).

Mengdeteoretiske lover

Eksempel.

- deMorgans lover
  - $\overline{A \cup B} = \overline{A} \cap \overline{B}$ (som vi så på i går).
  - $\overline{A \cap B} = \overline{A} \cup \overline{B}$
- De distributive lovene
  - $A \cup (B \cap C) = (A \cup B) \cap (A \cup C)$
  - $A \cap (B \cup C) = (A \cap B) \cup (A \cap C)$

Eksempel.

- $A \cap \overline{(B \cup C)} = (A - B) \cap (A - C)$
- $(\overline{A} - B) \cap C = C - (A \cup B)$

Oppgave.

- Vi bruker bare Venndiagrammer for uttrykk med en, to eller tre mengder.
- Tegn et Venndiagram for tre mengder A, B og C, og sett inn sannhetsverdiene for de tre basisutsagnene $x \in A$, $x \in B$ og $x \in C$ i de forskjellige feltene.
- Undersøk hvor mange deler det er mulig å dele planet inn i ved hjelp av fire sirkler.
- Forklar hvorfor dette viser at Venndiagrammer ikke er hensiktsmessige for Boolske uttrykk med mer enn tre mengder.

Inklusjon

Eksempel.

- Det er selvfølgelig slik at alle tall som kan deles på 4 også er partall.

  Vi sier da at mengden av tall delelige med 4 er inneholdt i partallene, eller at den er en delmengde av partallene.
- Mengden av registrerte fødselsnummere er inneholdt i mengden av alle data registrert i skattedirektoratet.
- Mengden av hunder er en delmengde av mengden av dyr.

Definisjon.

Hvis A og B er mengder, sier vi at A er inneholdt i B, eller at A er en delmengde av B, hvis

$$\forall x(x \in A \to x \in B).$$

- Vi skriver

  $$A \subseteq B$$

  for at A er inneholdt i B.

- Vi vil kunne skrive $A \subseteq B$ selv om $A = B$.
- Noen forfattere bruker $A \subset B$ slik vi bruker $A \subseteq B$ mens andre bruker det i betydningen

  $$A \subseteq B \wedge A \neq B.$$

- I dette siste tilfellet vil vi si at A er ekte inneholdt i B.

Eksempel.

- $\{2, 5, 6\} \subseteq \{1, 2, 5, 6, 7\}$ og inklusjonen er ekte.

- I følge læreboka vil
$$\mathbb{N} \subseteq \mathbb{J} \subseteq \mathbb{Q} \subseteq \mathbb{R}.$$
Når vi ser på disse mengdene som datatyper, vet vi at vi må bruke forskjellige måter å representere et tall i $\mathbb{J}$ på, avhengig av om vi ser på tallet som et element i $\mathbb{J}$ eller $\mathbb{R}$.
Denne påstanden er derfor ikke helt uproblematisk, men dog akseptabel for våre formål.
- $\{x : x^2 > 4\} \subseteq \{x : x^2 > 4 \vee x < -1\}$.

Vi kan bruke Venndiagrammer til å vise at et Boolsk uttrykk alltid definerer en delmengde av mengden definert ved et annet Boolsk uttrykk.
Vi skal se et par eksempler på tavlen.

Eksempel.
- $A \cap B \subseteq A \cup B$
- $\overline{A} \cap (B - C) \subseteq B - (A \cap C)$

Disjunkte mengder

Definisjon.
To mengder A og B er disjunkte hvis de ikke har noen felles elementer, det vil si, hvis
$$A \cap B = \emptyset.$$

Eksempel.
- $\{0, 4, 7, 9\}$ og $\{1, 2, 5, 10\}$ er disjunkte.
- $\{0, 4, 5, 7, 9\}$ og $\{0, 2, 5, 10\}$ er ikke disjunkte.
Snittet av disse to mengdene er $\{0, 5\} \neq \emptyset$.

Boolsk algebra

- Det er en nær sammenheng mellom Boolsk mengdealgebra og utsagnslogikk.
- Ved å erstatte A med $x \in A$ oppfattet som en utsagnsvariabel, kan vi spisse $\cup$ til $\vee$, $\cap$ til $\wedge$ og erstatte komplement $\overline{A}$ med $\neg(x \in A)$, og vi får en utsagnslogisk formel.
- Det er da naturlig å erstatte $\emptyset$ med **F** og $\mathcal{E}$ med **T**.
- To mengder, definert fra mengdevariable A, B og liknende ved hjelp av union, snitt og komplement vil alltid være like nøyaktig når oversettelsene er logisk ekvivalente.

- Tabell 5.1 på side 80 (79 i Utgave 2) i læreboka lister noen Boolske identiteter.
- De har sine paralleller i tabellen på side 56 (55) over logikkens lover.
- Vi skal ikke drille regning med disse Boolske identitetene.

En digresjon: Russells Paradoks

- Hvis vi hadde kunnet snakke om mengden av alle mengder, hadde vi hatt en grunn mindre til å bringe inn den universelle mengden $\mathcal{E}$.
- Antagelsen om at det fins en mengde som har alle mengder som elementer, leder imidlertid til en motsigelse som kalles Russells Paradoks.
- Vi gir beviset for Russells Paradoks som en oppgave med hint.

Oppgave.

- Anta at X er en mengde, og at for alle mengder Y vil $Y \in X$.
- La $Z = \{Y \in X \,:\, Y \notin Y\}$.
- Da er $Z \in X$.
- Vis at hvis $Z \in Z$ så vil $Z \notin Z$.
- Vis at hvis $Z \notin Z$ så vil $Z \in Z$.
- Forklar hvorfor dette viser at mengden X ikke kan finnes.

Digital representasjon av mengder

- I utgangspunktet skal det ikke spille noen rolle i hvilken rekkefølge man skriver opp elementene i en mengde.
- Hvis man imidlertid har behov for å representere visse mengder digitalt, må man velge seg en rekkefølge på elementene i den universelle mengden $\mathcal{E}$.
- Vi skal nå se på en metode for digital representasjon av mengder som virker når $\mathcal{E}$ er endelig.
- Hvis $\mathcal{E}$ er en uendelig mengde, må man enten velge en annen metode eller gi opp.

Definisjon.

- Anta at $\mathcal{E}$ har k elementer i rekkefølge
$$\{a_1, \ldots, a_k\}.$$
- La $A \subseteq \mathcal{E}$
- Vi representerer A som informasjon på k bit i rekkefølge, ved at bit nummer i får verdien 1 hvis og bare hvis $a_i \in A$.

- Ved denne måten å representere mengder på blir det svært enkelt å etterlikne de Boolske operasjonene.

- Snitt svarer til punktvis multiplikasjon, union svarer til det å ta maksimumsverdien punktvis og komplement svarer til å skifte verdi i alle bit.
- Vi kommer ikke til å jobbe så mye med digital representasjon av mengder.
- Når vi kommer til kompleksitetsteori, og vi spør om kompleksiteten til et problem som involverer mengder, trenger vi å vite hvordan mengder representeres digitalt.

Kardinaltall

- Hvis vi i noen sammenhenger ønsker å bruke digitale representasjoner av mengder, er det viktig at $\mathcal{E}$ ikke får lov til å være for stor.
- $10^{10^{10}}$ er lett å skrive, men foreløpig har vi ingen datamaskin med så mange bit.
- For å kunne følge med på hvor store mengder vi opererer med, og for å kunne resonnere omkring størrelse på mengder, er det en fordel med en notasjon for størrelsen av mengder.
- Det er dette vi vil fange opp i begrepet kardinaltall.

Definisjon.

- La A være en endelig mengde.
- Med kardinaltallet til A mener vi antall elementer i A.
- Vi skriver $|A|$ for kardinaltallet til A.

Eksempel.

- Hvis $A = \{2, 3, 17, 5, 23, 12, 15\}$ er $|A| = 7$.
- Hvis $B = \{n \in \mathbb{N} : 2n + 1 < 17\}$ er $|B| = 7$.
- Hvis $C = \{x \in \mathbb{R} : x^3 - 6x^2 + 11x - 6 = 0\}$ er $|C| = 3$.
- Hvis $D = \{x \in \mathbb{J} : x^2 = 64\}$ er $|D| = 2$.

Vi skal se på en enkel setning om kardinalitet som kan bli nyttig i avsnittet om kombinatorikk. Det er en setning som har sin parallell i sannsynlighetsteori, og i mange andre sammenhenger hvor man på en eller annen måte måler størrelse på mengder.

Teorem.

La A og B være to endelige mengder.

Da er

$$|A| + |B| = |A \cup B| + |A \cap B|.$$

Bevis.

Først observerer vi at hvis C, D og E er vilkårlige disjunkte mengder, så vil

$$|C \cup D \cup E| = |C| + |D| + |E|.$$

Så lar vi $C = A - B$, $D = A \cap B$ og $E = B - A$.

Fra Venn-diagrammet ser vi at disse er disjunkte.

Da er

- $|A \cup B| = |C| + |D| + |E|$
- $|A| = |C| + |D|$
- $|B| = |D| + |E|$

Teoremet følger da ved enkel regning.

Til de som ikke kom til forelesningen: Noe ble forklart ved Venn-diagrammer på tavlen.

Eksempel.

- Hvis alle 16 spillerne på et håndball-lag må kunne brukes i angrep eller forsvar, og vi vet at 7 av spillerne kan brukes i begge posisjoner og at 12 av spillerne kan brukes i forsvar, setter vi opp en likning for antall spillere som kan brukes i angrep:
- $$16 + 7 = x + 12.$$
- Det gir at 11 spillere kan brukes i angrepsspillet.
- Fra Venn-diagrammet ser vi at det vil være fire spillere som bare kan brukes i angrep, og fem spillere som bare kan brukes i forsvar.

- Den tyske matematikeren *Georg Cantor* utviklet en teori for kardinaltallet til en uendelig mengde også.
- Dette skjedde i siste halvdel av 1800-tallet.
- Ut fra Cantors definisjon fins det like mange rasjonale tall og hele tall som naturlige tall, mens det fins ekte flere reelle tall.
- Vi skal begrense oss til kardinalitet av endelige mengder.
- Selv om datamaskiner av natur bare kan håndtere endelig mye informasjon, har imidlertid studiet av uendelige mengder også en plass i informatikken.
- Dette faller ofte utenfor rammen til diskret matematikk og derfor utenfor pensum i MAT1030.

Eksempel.

a) La $A = \{0, 1, 2\}$.

Da har A 8 delmengder: $\emptyset$, $\{0\}$, $\{1\}$, $\{2\}$, $\{0, 2\}$, $\{0, 1\}$, $\{1, 2\}$ og $\{0, 1, 2\}$.

Disse er skrevet opp i en usystematisk rekkefølge.
En mer systematisk måte vil være først å skrive den ene delmengden av $\emptyset$: $\emptyset$,
så resten av delmengdene av $\{0\}$: $\{0\}$
så resten av delmengdene av $\{0,1\}$: $\{1\}$ og $\{0,1\}$
og til slutt resten av delmengdene av $\{0,1,2\}$: $\{2\}$, $\{0,2\}$, $\{1,2\}$ og $\{0,1,2\}$

Eksempel (Fortsatt).
Den naturlige rekkefølgen blir da

$$\{\emptyset,\{0\},\{1\},\{0,1\},\{2\},\{0,2\},\{1,2\},\{0,1,2\}\}$$

b) For å liste opp alle delmengder av $\{0,1,2,3\}$ lister vi først opp alle delmengder av $\{0,1,2\}$ og deretter alle nye delmengder, ved å legge 3 til en av de åtte første.
Det gir
$\{\emptyset,\{0\},\{1\},\{0,1\},\{2\},\{0,2\},\{1,2\},\{0,1,2\},$
$\{3\},\{0,3\},\{1,3\},\{0,1,3\},\{2,3\},\{0,2,3\},\{1,2,3\},\{0,1,2,3\}\}$

Potensmengder

Definisjon.
- La A være en mengde.
- Med potensmengden til A mener vi mengden av alle delmengder av A.

Merk.
- Hvis A er en mengde og $B \subseteq A$ er en vilkårlig delmengde, vil vi for hver $x \in A$ ha to muligheter, $x \in B$ og $x \notin B$.
- En konsekvens er at hvis A er endelig vil potensmengden til A ha $2^{|A|}$ elementer.
- Dette vil ofte bety at det vil ta alt for lang tid å gjennomføre naive algoritmer.

Eksempel.
- La A være en endelig mengde av naturlige tall.

- Vi lar $\sum A$ bety summen av alle tallene i A.
- Partisjonsproblemet er om det fins delmengder B og C av A som er slik at
  1. $A = B \cup C$
  2. $\emptyset = B \cap C$ (de er disjunkte)
  3. $\sum B = \sum C$
- Den første strategien for å løse dette problemet kan være å liste opp alle par $B \subseteq A$ og $C = A - B$, og sjekke, men hvis A har 1000 elementer, er ikke dette praktisk gjennomførbart.
- Ingen vet per i dag om det fins en vesentlig raskere metode for å løse partisjonsproblemet generelt.
- Partisjonsproblemet er et eksempel på et NP-komplett problem.

Merk.

- Potensmengden til A er definert selv om A er uendelig.
- I det tilfellet er ikke alle egenskapene ved potensmengder fullstendig klarlagt ennå.
- Vi ledes langt ut over rammene for diskret matematikk om vi prøver å forstå potensmengden til en uendelig mengde.
- Cantor viste at i en viss forstand er potensmengden til A alltid ekte større enn A.

Ordnede par

- Vi har brukt mengden $\mathbb{R}^2$ av tallpar i tidligere eksempler.
- Alle vet at det er forskjell på tallparene $(2,3)$ og $(3,2)$ i $\mathbb{R}^2$.
- Det betyr at rekkefølgen på tallene i paret spiller en rolle.
- Et slikt par kaller vi et ordnet par.

- Det er ikke bare tall som kan opptre i par.
- Vi kan for eksempel skrive at

  Per og Kari er ektefeller

  og vi mener så absolutt at de utgjør et par.
- I dette tilfellet betyr ikke rekkefølgen noe, men skriver vi

  Kari er kona til Per

  kan vi ikke erstatte det med

  Per er kona til Kari.

- Vi trenger begrepet ordnet par for å kunne snakke presist og generelt om visse former for sammenhenger vi kan finne mellom to objekter.
- Disse objektene kan være tall i en tallmengde.
- De kan imidlertid også være data i en base, data som representerer personer, hendelser, adresser, yrker og mye annet det kan være behov for å registrere.

- Derfor vil vi legge en helt generell definisjon til grunn, når vi definerer hva som menes med et ordnet par.

Definisjon.

La a og b være to objekter.

Det ordnede paret (a, b) av a og b er a og b skrevet i rekkefølge.

To ordnede par (a, b) og (c, d) er like hvis $a = c$ og $b = d$.

Merk.

- Vi har egentlig ikke sagt hva et ordnet par er for noe, bare knyttet det til at objektene settes i rekkefølge.
- Det er definisjonen av når to ordnede par er like som gir oss den ønskede matematiske presisjonen. Det knytter den abstrakte definisjonen opp til skrivemåten vi benytter.

Definisjon.

La A og B være to mengder.

Med det kartesiske produktet $A \times B$ av A og B mener vi

$$A \times B = \{(a, b) \ : \ a \in A \wedge b \in B\}.$$

Betegnelsen henter sitt navn fra den franske matematikeren og filosofen René Descartes.

Eksempel.

- La A være mengden av registrerte norske skøyteløpere og B være mengden av tider mellom 1.30.00 og 2.30.00.

  Da vil registreringer av personlige rekorder på 1500m oppfattes som par i $A \times B$
- Hvis A er mengden av ord skrevet med latinske bokstaver og B er mengden av sider på nettet, leter vi i prinsippet gjennom $A \times B$ når vi søker etter nettsider hvor et bestemt ord forekommer.

  I dette tilfellet er det klart at vi trenger å utvikle spesielle teknikker for å kunne gjøre dette på en effektiv måte, men utviklingen av slike teknikker starter med å forstå kompleksiteten av $A \times B$.

Hvis A og B er endelige mengder, vil

$$|A \times B| = |A| \cdot |B|.$$

For de som har lært om matriser, ser vi sammenhengen med en $n \times m$-matrise.
La $A = \{a_1, \cdots, a_n\}$ og $B = \{b_1, \cdots, b_m\}$.
Da kan vi skrive $A \times B$ som:

$$\left\{ \begin{array}{ccc} (a_1, b_1) & \cdots\cdots & (a_1, b_m) \\ \cdot & & \cdot \\ \cdot & & \cdot \\ \cdot & & \cdot \\ (a_n, b_1) & \cdots\cdots & (a_n, b_m) \end{array} \right\}$$

MAT1030 – Forelesning 11

Relasjoner

Dag Normann - 23. februar 2010

Kapittel 5: Relasjoner

Kommentarer til forelesningen

På grunn av svikt i dataanlegget, ble forelesningen gitt som tavleforelesning.

Dette resulterte i at noen planlagte momenter uteble, mens foreleser snakket om ordninger, et tema planlagt for 24.02.2010.

Samlet sett vil forelesningene 23.02.2010 og 24.02.2010 dekke pensum om relasjoner, med eksempler i tillegg.

Binære relasjoner

- Forrige uke innførte vi mengdebegrepet, og vi så på Boolske operasjoner som union, snitt og komplement.
- Vi lærte å sette opp Venn-diagrammer for å studere resultatet av sammensatte Boolske operasjoner.
- Vi så på inklusjon, det vil si delmengdebegrepet.
- Vi var en rask tur innom kardinalitet.
- Til slutt så vi på ordnede par (a, b) og og kartesisk produkt $A \times B$.
- $A \times B = \{(a, b) : a \in A \wedge b \in B\}$
- Vi fortsetter der vi slapp.

Eksempel.

- Hvis A er mengden av norske statsborgere, vil $A \times A$ være mengden av par av norske statsborgere.

  Det fins mange interessante undermengder av $A \times A$ bestemt av de forskjellige forhold det kan være mellom to personer, eksempelvis

  - kollega av
  - søster til
  - nabo av
  - misunnelig på
  - $\ldots$

  Dette vil lede oss over til avsnittet om relasjoner.

Definisjon.

La A være en mengde.

En binær relasjon på A er en delmengde R av $A^2 = A \times A$.

Merk.

- I senere studier kan dere komme borti relasjoner mellom tre eller flere objekter. Disse er da ikke binære.
- Siden vi bare skal studere binære relasjoner, gjør vi som boken, og dropper ordet "binær".

Eksempel.

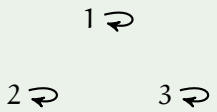
- I kryptografi er moduloregning viktig.
- Hvis p er et primtall og a og b er hele tall, sier vi at $a \equiv_p b$ om p er en faktor i $a - b$.
- Da er $\equiv_p$ en relasjon på de hele tallene.
- Vi kunne like gjerne skrevet
$$(a, b) \in \equiv_p .$$
- Relasjonen $\equiv_p$ og beslektede relasjoner (hvor eksempelvis p ikke er et primtall, men i praksis umulig å faktorisere) spiller en stor rolle i kryptografi.

Vi skal se på hvordan vi kan visualisere relasjoner.

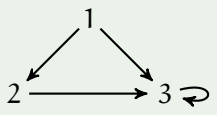
Eksempel.

La $S = \{1, 2, 3\}$.

- $\{(1,1), (2,2), (3,3)\}$ er en binær relasjon på S, skrevet på listeform. På grafisk form ser relasjonen slik ut:



- $\{(1,2), (1,3), (2,3), (3,3)\}$ er også en binær relasjon på S, skrevet på listeform. Grafisk form blir:



Eksempel (Flere eksempler på relasjoner).

- *Likhetsrelasjonen* på en mengde S, $\{(x, x) : x \in S\}$.
- *Mindre enn*-relasjonen på f.eks. naturlige tall, $\{(x, y) : x, y \in \mathbb{N} \text{ og } x < y\}$
- *Foreldrerelasjonen* på f.eks. mengden av mennesker, $\{(a, b) : a \text{ er forelder til } b\}$
- *Delmengde*-relasjonen på en mengde av mengder.
- Og så videre.

- Noen lærebøker vil definere en relasjon fra A til B som en mengde

$$R \subseteq A \times B.$$

- Det kan fins pedagogiske grunner for å gjøre det slik, men
  enhver relasjon fra A til B vil samtidig være en relasjon *på* $A \cup B$.
- Vi vinner ikke tilstrekkelig mye på å arbeide med mere generelle relasjoner sett i forhold til hvor omstendelig det vil bli å formulere det vi ønsker å uttrykke.

Notasjon for relasjoner

- Det å beskrive en relasjon som en mengde av ordnede par gir ikke mye innsikt i hvordan relasjonen ser ut.
- Det å skrive $(a, b) \in R$ representerer også en uvant måte å skrive ting på.
- Ingen av oss har lyst til å begynne å skrive

$$(2, 3) \in <$$

  i stedet for $2 < 3$, eller

$$(3, 3) \in =$$

  i stedet for $3 = 3$, for ikke å snakke om

$$(\emptyset, \{\emptyset\}) \in \in$$

  i stedet for $\emptyset \in \{\emptyset\}$.
- Den første forenklingen vi skal gøre er å skrive aRb når vi mener $(a, b) \in R$.

Beskrivelser av relasjoner

- Hvis A er en liten mengde, fins det to måter å beskrive R på,
  - Ved hjelp av en matrise
  - Ved hjelp av en graf
- Vi skal se på noen eksempler.
- For begge måter å beskrive relasjoner på spiller det en stor rolle hvordan man organiserer elementene i mengden A,

  - i rekkefølge som koordinater
  - som punkter på en tavle eller et ark
- La $A = \{1,2,3,4,5\}$ og la $R = \{(1,3),(2,4),(3,5),(4,1),(5,2)\}$
- Vi skal illustrere R ved hjelp av en 5×5-matrise.
  - Radene, regnet ovenfra, vil representere 1. koordinat.
  - Søylene, regnet fra venstre, vil representere 2. koordinat.
  - Vi markerer elementene i R med **T** og de andre med **F**.
  - Det er like vanlig, og ofte bedre, å bruke 1 og 0.

$$\begin{bmatrix} \mathbf{F} & \mathbf{F} & \mathbf{T} & \mathbf{F} & \mathbf{F} \\ \mathbf{F} & \mathbf{F} & \mathbf{F} & \mathbf{T} & \mathbf{F} \\ \mathbf{F} & \mathbf{F} & \mathbf{F} & \mathbf{F} & \mathbf{T} \\ \mathbf{T} & \mathbf{F} & \mathbf{F} & \mathbf{F} & \mathbf{F} \\ \mathbf{F} & \mathbf{T} & \mathbf{F} & \mathbf{F} & \mathbf{F} \end{bmatrix}$$

- Den grafiske fremstillingen tar vi på tavlen.
- La $A = \{1,2,3,4\}$ og $R = \{(1,2),(1,3),(1,4),(4,1),(3,1),(2,1),(3,3)\}$
- Matriseformen blir

$$\begin{bmatrix} \mathbf{F} & \mathbf{T} & \mathbf{T} & \mathbf{T} \\ \mathbf{T} & \mathbf{F} & \mathbf{F} & \mathbf{F} \\ \mathbf{T} & \mathbf{F} & \mathbf{T} & \mathbf{F} \\ \mathbf{T} & \mathbf{F} & \mathbf{F} & \mathbf{F} \end{bmatrix}$$

- Den grafiske fremstillingen tar vi på tavlen.

Flere begreper

- Erfaringsmessig faller det stoffet vi nå begynner på vanskeligere for mange enn det vi har gått gjennom så langt.
- Det skyldes at stoffet synes abstrakt og at det er mange nye begreper.
- I eksempler vil vi kunne innføre noen nye symboler som vi gir en spesiell betydning.
- Et eksempel vi har sett på er relasjonen $a \equiv_p b$ som uttrykker at p er en faktor i $a - b$.
- En av de ferdighetene vi skal oppøve i MAT1030 er evnen til å lese, og forstå, definisjoner.
- De relasjonene vi vil innføre i eksemplene, vil som oftest ikke være pensum, men evnen til å forstå slike eksempler kan bli prøvet til eksamen.
- Relasjoner med bestemte egenskaper kan dukke opp i så mange forskjellige sammenhenger at det kan være aktuelt å studere dem under ett.
- Det kan også være aktuelt å utvikle program som virker for alle relasjoner av en gitt type.
- Følgende oppgaver trenger strengt tatt det samme programmet:
  - Ordne dagens avisoverskrifter alfabetisk.
  - Ordne deltagerne i et løp etter oppnådd sluttid.
  - Ordne LOTTO-tallene etter størrelse.
  - Ordne studenter etter oppnådde karakterer.

- Det er noe felles ved oppgaven å skulle ordne en mengde, og det er naturlig å isolere de relasjonene som kan oppfattes som ordninger.
  - Det er blant annet noe av det vi skal lære nå.

- Inklusjon mellom mengder er en form for "større enn", men det fins mengder, eksempelvis $\{1,2,3\}$ og $\{2,3,4\}$, som ikke er inneholdt i hverandre noen vei.
- Hva vil sorteringsalgoritmer gjøre hvis vi bruker dem på slike ordninger?
- Det er noen egenskaper ved relasjoner som er så vanlig forekommende (hos de *nyttige* relasjonene) at vi har gitt dem egne navn.
- Vi gir listen først og drøfter hver enkelt egenskap etterpå:

Egenskaper ved binære relasjoner

Definisjon.

- En relasjon R på en mengde A kalles:
  - Refleksiv hvis xRx for alle $x \in A$.
  - Irrefleksiv hvis det ikke fins noen $x \in A$ slik at xRx.
  - Symmetrisk hvis xRy medfører yRx for alle $x, y \in A$.
  - Antisymmetrisk hvis $xRy \wedge yRx$ medfører at $x = y$ for alle $x, y \in A$.
  - Transitiv hvis $xRy \wedge yRz$ medfører xRz for alle $x, y, z \in A$.
- Vi skal bruke den tiden vi trenger til å lære oss disse begrepene og å forstå dem.

Refleksive relasjoner

Eksempel (Refleksiv: xRx for alle x).

- For enhver mengde A vil likhetsrelasjonen $x = y$ være refleksiv på A.
- Hvis X er mengden av delmengder av en mengde $\mathcal{E}$, vil inklusjonsrelasjonen $A \subseteq B$ være refleksiv på X.
- $\leqslant$ er en refleksiv relasjon, uansett om vi ser på den som en relasjon på $\mathbb{N}$, på $\mathbb{Q}$, på $\mathbb{J}$ eller på $\mathbb{R}$.
- $<$ er normalt ikke en refleksiv relasjon, spesielt ikke når vi bruker tegnet på vanlig måte for $\mathbb{N}$ etc.
- Hvis A er mengden av sammensatte utsagn i utsagnsvariablene p, q og r, og ϕ og ψ er sammensatte utsagn, kan vi definere $\phi R \psi$ som

  $$\phi \to \psi \text{ er en tautologi.}$$

  Da er R en refleksiv relasjon.

- Hvis en relasjon gis på matriseform, er det lett å se om relasjonen er refleksiv eller ikke:

$$\begin{bmatrix} \mathbf{T} & \mathbf{T} & \mathbf{F} & \mathbf{F} & \mathbf{F} \\ \mathbf{T} & \mathbf{T} & \mathbf{F} & \mathbf{F} & \mathbf{F} \\ \mathbf{T} & \mathbf{F} & \mathbf{T} & \mathbf{F} & \mathbf{T} \\ \mathbf{T} & \mathbf{T} & \mathbf{F} & \mathbf{T} & \mathbf{F} \\ \mathbf{T} & \mathbf{F} & \mathbf{F} & \mathbf{F} & \mathbf{T} \end{bmatrix}$$

- Ved å se at vi har bare **T** på diagonalen, ser vi at relasjonen er refleksiv.
- Gjør vi en liten forandring, trenger ikke relasjonen lenger å være refleksiv.

$$\begin{bmatrix} \mathbf{T} & \mathbf{T} & \mathbf{F} & \mathbf{F} & \mathbf{F} \\ \mathbf{T} & \mathbf{F} & \mathbf{F} & \mathbf{F} & \mathbf{F} \\ \mathbf{T} & \mathbf{F} & \mathbf{T} & \mathbf{F} & \mathbf{T} \\ \mathbf{T} & \mathbf{T} & \mathbf{F} & \mathbf{T} & \mathbf{F} \\ \mathbf{T} & \mathbf{F} & \mathbf{F} & \mathbf{F} & \mathbf{T} \end{bmatrix}$$

Irrefleksive relasjoner

Eksempel (Irrefleksiv: $\neg(x\mathbf{R}x)$ for alle x).

- $\neq$ er irrefleksiv på alle mengder.
- *Far til* og *Mor til* er irrefleksive relasjoner på enhver forsamling av mennesker.
- $<$ og $>$ er irrefleksive relasjoner på $\mathbb{N}$, $\mathbb{J}$, $\mathbb{Q}$ og $\mathbb{R}$.
- *Ekte inklusjon* er en irrefleksiv relasjon.

Eksempel.

- La X være mengden av sammensatte utsagn i utsagnsvariablene p, q og r, la ϕ og ψ være sammensatte utsagn, og definer relasjonen S ved

  $$\phi S \psi$$

  når

  $$\phi \wedge \psi \rightarrow \mathbf{F}$$

  er en tautologi.
- Da er S hverken refleksiv eller irrefleksiv.

- Hvis relasjonen blir beskrevet ved hjelp av en matrise, er det også enkelt å kontrollere om relasjonen er irrefleksiv:

$$\begin{bmatrix} \mathbf{F} & \mathbf{T} & \mathbf{F} & \mathbf{F} & \mathbf{F} \\ \mathbf{T} & \mathbf{F} & \mathbf{F} & \mathbf{F} & \mathbf{F} \\ \mathbf{T} & \mathbf{F} & \mathbf{F} & \mathbf{F} & \mathbf{T} \\ \mathbf{T} & \mathbf{T} & \mathbf{F} & \mathbf{F} & \mathbf{F} \\ \mathbf{T} & \mathbf{F} & \mathbf{F} & \mathbf{F} & \mathbf{F} \end{bmatrix}$$

- Det er bare å sjekke om det står **F** langs diagonalen.

Symmetriske relasjoner

Eksempel (Symmetrisk: $x\mathrm{R}y$ medfører $y\mathrm{R}x$ for alle x og y.).

- Symmetriske relasjoner er de relasjonene hvor rekkefølgen ikke spiller noen rolle.
- x er gift med y.
- $\phi \wedge \psi$ er ikke en kontradiksjon.
- $\phi \wedge \psi$ er en kontradiksjon.
- n og m har en felles faktor > 1, som en relasjon på $\mathbb{N}$.
- $A \cap B = \emptyset$ som relasjon på potensmengden til $\mathcal{E}$.

- Vi kan undersøke om en relasjon er symmetrisk ved å studere matriserepresentasjonen:

Eksempel.

$$\begin{bmatrix} \mathbf{T} & \mathbf{T} & \mathbf{T} & \mathbf{F} \\ \mathbf{T} & \mathbf{F} & \mathbf{F} & \mathbf{T} \\ \mathbf{T} & \mathbf{F} & \mathbf{T} & \mathbf{T} \\ \mathbf{F} & \mathbf{T} & \mathbf{T} & \mathbf{F} \end{bmatrix}$$

Denne relasjonen er symmetrisk

Eksempel.

$$\begin{bmatrix} \mathbf{T} & \mathbf{F} & \mathbf{T} & \mathbf{F} \\ \mathbf{T} & \mathbf{F} & \mathbf{F} & \mathbf{T} \\ \mathbf{T} & \mathbf{F} & \mathbf{T} & \mathbf{T} \\ \mathbf{F} & \mathbf{T} & \mathbf{T} & \mathbf{F} \end{bmatrix}$$

Denne relasjonen er *ikke* symmetrisk

Antisymmetriske relasjoner

Eksempel (Antisymmetrisk: $x\mathrm{R}y \wedge y\mathrm{R}x$ medfører $x = y$).

- I en antisymmetrisk relasjon skal vi ikke ha andre positive speilsymmetrier om diagonalen enn de som ligger på diagonalen.
- Inklusjon av mengder.
- $\leqslant$ og $<$ i vanlige sammenhenger.
- Antisymmetri er svært ofte knyttet til former for ordninger, og vi kommer tilbake til dette senere

Transitive relasjoner

Eksempel (Transitiv: $xRy \wedge yRz$ medfører xRz).

- $<$ og $\leqslant$ er transitive relasjoner i alle vanlige sammenhenger.
- $\subseteq$ er en transitiv relasjon på potensmengden til $\mathcal{E}$.
- "ϕ er en logisk konsekvens av ψ" er transitiv.
- *Far til* og *Mor til* er ikke transitive, men *etterkommer* er transitiv.
- *Søsken til* er transitiv hvis vi mener helsøsken, men ikke hvis vi inkluderer halvsøsken.

Et eksempel

- Vi skal ta utgangspunkt i noen relasjoner som det kan være aktuelt å studere av praktiske eller teoretiske grunner.
- Vi skal se på hvilke av de fem egenskapene vi har sett på disse relasjonene vil ha.
- Hvis A og B er utsagn i predikatlogikk, lar vi $A \Rightarrow B$ bety at B er en logisk konsekvens av A. Vi sier at A impliserer B.
- Det betyr at B er sann i enhver situasjon som gjør A sann.
- Vi oppfatter $\Rightarrow$ som en relasjon på mengden av utsagn i predikatlogikk.
- $\Rightarrow$ er *refleksiv* fordi $A \Rightarrow A$ for alle utsagn A.
- $\Rightarrow$ er da ikke *irrefleksiv*.
- $\Rightarrow$ er ikke *symmetrisk* (se 1.) eller *antisymmetrisk* (se 2.)
  1. Vi har $x > 0 \Rightarrow x > 0 \vee x = 0$, men omvendingen gjelder ikke.
  2. Utsagnene $\neg(x < 0) \vee x > 0$ og $x < 0 \rightarrow x > 0$ impliserer hverandre, men de er ikke like.
- $\Rightarrow$ er transitiv siden $A \Rightarrow C$ når $A \Rightarrow B$ og $B \Rightarrow C$.

Enda et eksempel

- Når man skal gi en matematisk beskrivelse av hva et program P "gjør", er det ofte to relasjoner som det er aktuelle å studere.
- Vi begrenser oss til språk som minner om pseudokoder.
- Da har vi variable $x_1, \ldots, x_n$ i programmet.
- Underveis vil verdiene på disse variablene endre seg, gjennom instruksjoner som

$$x_i \leftarrow t(x_1, \ldots, x_n).$$

- En fordeling av verdier på variablene kaller vi en valuasjon eller en tilstand.
- La V være mengden av valuasjoner, og la u og v være elementer i V.
- Hvert program $\mathcal{P}$ vil bestemme en relasjon $[\![\mathcal{P}]\!]$ på V, hvor

$$u[\![\mathcal{P}]\!]v$$

  hvis output-valuasjonen er v når inputvaluasjonen er u.

- Hvis $[\![\mathcal{P}]\!]$ er *refleksiv*, betyr det at programmet i realiteten lar alle inputvaluasjoner være uforandret.
- Hvis $[\![\mathcal{P}]\!]$ er *irrefleksiv*, betyr det at programmet gjør endringer uansett input.
- Hvis $[\![\mathcal{P}]\!]$ er symmetrisk, betyr det at hvis vi kjører programmet en gang til, kommer vi tilbake til utgangspunktet.
- Krypteringsmaskinen ENIGMA brukt av tyskerne under krigen hadde den egenskapen.
- Hvis $[\![\mathcal{P}]\!]$ er antisymmetrisk, betyr det at vi aldri kommer tilbake til input-dataene ved å kjøre programmet på output-dataene.
- $[\![\mathcal{P}]\!]$ er i praksis aldri transitiv, siden dette ville medført at vi oppnår det samme om vi kjører programmet to ganger, hvor output overføres til input mellom gangene.

  Dette vil imidlertid være tilfelle om output-dataene alltid inneholder en form for stopp-ordre.

- En annen viktig relasjon i studiet av hva programmer "gjør" er $\vdash^*$
- Et program $\mathtt{P}$ er normalt en form for tekst, og den teksten består av punkter eller instruksjoner.
- Typisk har vi nummerert alle linjene i en pseudokode, slik at vi kan snakke om at vi er i en bestemt posisjon i programmet underveis i kjøringen av programmet.
- La $\mathtt{P}$ være mengden av posisjoner og la fortsatt V være mengden av valuasjoner.
- $\vdash^*$ er relasjonen på $\mathtt{P} \times V$ hvor

$$(\mathfrak{p}, \nu) \vdash^* (\mathfrak{q}, \mathfrak{u})$$

  hvis programmet $\mathcal{P}$, om vi er i posisjon $\mathfrak{p}$ og med valuasjon ν vil komme til posisjon $\mathfrak{q}$ og med valuasjon $\mathfrak{u}$ etter ingen, ett eller flere regneskritt.
- Denne relasjonen er selvfølgelig avhengig av $\mathcal{P}$, og vi kan skrive den $\vdash^*_{\mathcal{P}}$.
- Denne relasjonen er *refleksiv*, fordi vi tillater at vi ikke tar noen regneskritt.

  Den er da normalt ikke irrefleksiv.
- Denne relasjonen er *transitiv*.
- Hvis relasjonen er symmetrisk, er det en katastrofe for programmereren, siden vi da bare har løkkeberegninger.
- Hvis relasjonen er *antisymmetrisk*, vil beregningen aldri gå i løkke.

MAT1030 – Forelesning 12

Relasjoner

Dag Normann - 24. februar 2010

Kapittel 5: Relasjoner

Repetisjon

- En relasjon på en mengde A er en delmengde $R \subseteq A \times A = A^2$.
- Vi har satt navn på visse egenskaper relasjoner som oppstår i anvendelser ofte kan ha.

Definisjon.

- En relasjon R på en mengde A kalles:
  - Refleksiv hvis xRx for alle $x \in A$.
  - Irrefleksiv hvis vi ikke har noen $x \in A$ slik at xRx.
  - Symmetrisk hvis xRy medfører yRx for alle $x, y \in A$.
  - Antisymmetrisk hvis $xRy \wedge yRx$ medfører at $x = y$ for alle $x, y \in A$.
  - Transitiv hvis $xRy \wedge yRz$ medfører xRz for alle $x, y, z \in A$.

Ekvivalensrelasjoner

En viktig klasse av relasjoner er ekvivalensrelasjonene. La oss se på noen eksempler før vi gir den formelle definisjonen.

Eksempel.

a) La $A = \{0, \ldots, 9\}$.

Hvis a og b er elementer i A lar vi aRb hvis $a - b$ er delelig med 3.

Vi ser at R deler A opp i tre disjunkte mengder $B = \{0, 3, 6, 9\}$, $C = \{1, 4, 7\}$ og $D = \{2, 5, 8\}$, hvor hver mengde består av tall som står i innbyrdes relasjon til hverandre, mens vi ikke har noen R-forbindelser på mengdene imellom.

Eksempel (Fortsatt).

b) La $A = \mathbb{R}^2$ og la

$$(x, y)R(u, v) \Leftrightarrow x^2 + y^2 = u^2 + v^2.$$

Vi ser at to punkter står i R-relasjon til hverandre nøyaktig når avstanden til origo er den samme.

Denne relasjonen deler $\mathbb{R}^2$ opp i uendelig mange disjunkte mengder, nemlig sirklene om origo med radius r for $r \geqslant 0$.

Eksempel (Fortsatt).

c) La $A = \mathbb{Q}$ og la pRq om p og q har den samme *heltallsverdien*.

Heltallsverdien til et tall p er det største hele tallet $a \leqslant p$.

I alle disse eksemplene har vi definert en relasjon R ved at to objekter står i relasjon til hverandre når de deler en viss felles egenskap.

Det er denne typen relasjoner vi vil kalle *ekvivalensrelasjoner*.

- Uansett hvilken egenskap det vil være snakk om, vil ethvert objekt dele denne egenskapen med seg selv.

  Vi vil altså kreve av en ekvivalensrelasjon at den skal være *refleksiv*.
- Hvis a og b deler en egenskap, kan vi også snu på ordstillingen og si at b og a deler denne egenskapen.
- Det er altså et rimelig krav til at en relasjon skal kalles en ekvivalensrelasjon at den er *symmetrisk*.
- Hvis a og b deler en egenskap, og b og c deler den samme egenskapen, er den også felles for a og c.

  Vi vil kreve av en ekvivalensrelasjon at den skal være *transitiv*.
- Det viser seg at disse tre egenskapene er tilstrekkelige til å fange opp intuisjonen vår om å formalisere det uformelle "dele visse egenskaper".

  Den formelle definisjonen er:

Definisjon.

La R være en relasjon på en mengde A.

Vi kaller R en ekvivalensrelasjon om R er refleksiv, symmetrisk og transitiv.

Ekvivalensklasser

- En viktig egenskap ved en ekvivalensrelasjon R på en mengde A er at R deler A opp i disjunkte mengder av parvis ekvivalente elementer i A.

  (Disjunkt betyr at snittet er tomt.)
- Disse mengdene kaller vi ekvivalensklasser.

- Vi skal vise denne egenskaper ved ekvivalensrelasjoner helt generelt, men la oss se på noen eksempler først.

Eksempel.

- La $A = \{0, 1, 2, 3, 4, 5\}$
- La $R = \{(0,0), (0,1), (1,0), (1,1), (2,2), (3,3), (3,4),$
  $(3,5), (4,3), (4,4), (4,5), (5,3), (5,4), (5,5)\}$
- I utgangspunktet er det ikke så lett å se hvilke egenskaper R har, men beskriver vi R på matriseform blir bildet klarere:

Eksempel (Fortsatt).

$$\begin{bmatrix} \mathbf{T} & \mathbf{T} & \mathbf{F} & \mathbf{F} & \mathbf{F} & \mathbf{F} \\ \mathbf{T} & \mathbf{T} & \mathbf{F} & \mathbf{F} & \mathbf{F} & \mathbf{F} \\ \mathbf{F} & \mathbf{F} & \mathbf{T} & \mathbf{F} & \mathbf{F} & \mathbf{F} \\ \mathbf{F} & \mathbf{F} & \mathbf{F} & \mathbf{T} & \mathbf{T} & \mathbf{T} \\ \mathbf{F} & \mathbf{F} & \mathbf{F} & \mathbf{T} & \mathbf{T} & \mathbf{T} \\ \mathbf{F} & \mathbf{F} & \mathbf{F} & \mathbf{T} & \mathbf{T} & \mathbf{T} \end{bmatrix}$$

Vi ser at aRb hvis og bare hvis a og b tilhører den samme av delmengdene $\{0, 1\}$, $\{2\}$ og $\{3, 4, 5\}$.

Eksempel.

- Vi definerer relasjonen R på $\mathbb{R}^2$ ved $(x, y)R(u, v)$ om $x + y = u + v$.
- R er en ekvivalensrelasjon.
- Hvis $x + y = k$ vil $(x, y)R(u, v)$ hvis og bare hvis $u + v = k$.
- Denne relasjonen deler planet opp i uendelig mange disjunkte deler, hvor delene er alle linjene med stigningstall -1

Definisjon.

La R være en ekvivalensrelasjon på en mengde A og la $a \in A$.

Vi lar ekvivalensklassen til a være

$$E(a) = \{b \in A \mid aRb\}.$$

Teorem.

La R være en ekvivalensrelasjon på mengden A, $E(a)$ ekvivalensklassen til $a \in A$.

a) For alle $a \in A$ vil $a \in E(a)$.

b) Hvis aRb vil $E(a) = E(b)$.

c) Hvis $\neg(aRb)$ vil $E(a) \cap E(b) = \emptyset$

Dette teoremet sier nettopp at hvis R er en ekvivalensrelasjon på A, så vil vi dele A opp i disjunkte klasser av parvis ekvivalente elementer.

Bevis.

Siden aRa vil $a \in E(a)$. Dette viser a).

La aRb. Skal vise at $E(a) = E(b)$.

Siden vi da også har at bRa er det nok å vise at $E(b) \subseteq E(a)$ for å vise b).

Så la $c \in E(b)$.

Da vil $aRb \wedge bRc$ så aRc fordi R er transitiv.

Det betyr at $c \in E(a)$.

Siden $c \in E(b)$ var valgt vilkårlig, kan vi slutte at $E(b) \subseteq E(a)$.

Bevis (Fortsatt).

Til sist, anta at $c \in E(a) \cap E(b)$.

Da vil $aRc \wedge bRc$.

Ved symmetri og transitivitet for R følger det at aRb.

Snur vi dette argumentet på hodet, får vi at $\neg(aRb) \Rightarrow E(a) \cap E(b) = \emptyset$.

Dette viser c), og teoremet er bevist.

Oppgave.

La A være en mengde og la R og S være ekvivalensrelasjoner på A.

La $T = S \cap R$

a) Forklar hvorfor vi har at aTb hvis og bare hvis $aRb \wedge aSb$ for alle a og b i A.

b) Vis at T er en ekvivalensrelasjon.

c) Finn et eksempel hvor $R \cup S$ ikke er en ekvivalensrelasjon.

Ordninger

- En annen type relasjoner som forekommer ofte er ordninger.
- Vi har en ordning når vi formaliserer "mindre eller lik" , "er svakere enn" og tilsvarende forhold.
- Vi tar definisjonen først, og illustrerer den med noen eksempler etterpå.
- Vi følger boka og definerer:

Definisjon.

La A være en mengde med en relasjon R.

Vi kaller R en partiell ordning hvis

1. R er refleksiv
2. R er transitiv
3. R er antisymmetrisk.

Eksempel.

a) La $\mathcal{E}$ være en universell mengde, og $\subseteq$ være inklusjonsordningen på potensmengden til $\mathcal{E}$.

1. $\subseteq$ er *refleksiv* fordi $A \subseteq A$ for alle mengder A.
2. $\subseteq$ er *transitiv* fordi $A \subseteq B$ og $B \subseteq C$ alltid vil medføre at $A \subseteq C$
3. $\subseteq$ er *antisymmetrisk* fordi $A = B$ når $A \subseteq B$ og $B \subseteq A$.

Dette viser at $\subseteq$ er en partiell ordning.

Eksempel (Fortsatt).

b) La $\leqslant$ være den vanlige "mindre-eller-lik" relasjonen på $\mathbb{J}$.

$\leqslant$ er opplagt både refleksiv, transitiv og antisymmetrisk, så $\leqslant$ er en partiell ordning.

$<$ er derimot *IKKE* en partiell ordning, fordi $<$ ikke er *refleksiv*.

c) Hvis vi ordner studentene i MAT1030 etter oppnådd karakter, med A øverst, F langt nede, men "ikke møtt" og "trukket seg før eksamen" enda lenger nede, har vi ikke en partiell ordning.

Siden det er mer enn 8 studenter, må minst to stykker lide samme MAT1030-skjebne.

Disse vil stå likt, men være forskjellige personer.

Det betyr at denne relasjonen ikke er *antisymmetrisk*

- Det er en viktig forskjell mellom ordningen av potensmengden til $\mathcal{E}$ og ordningen av $\mathbb{J}$.
- Hvis vi har to tall a og b vil en av tre holde:

1. $a = b$
2. $a < b$
3. $b < a$

- For inklusjon har vi en fjerde mulighet, nemlig at ingen av A eller B er inneholdt i den andre.
- Vi fanger opp denne forskjellen i en definisjon:

Definisjon.

La R være en partiell ordning på en mengde A.

R kalles total dersom vi for alle a og b i A har at

$$aRb \vee bRa.$$

Merk.

- Det er for totale ordninger at vi har gode sorteringsalgoritmer.
- Utviklingen av slike algoritmer overlater vi til foreleserne i programmering, de er flinkere til å finne effektive algoritmer.

En utfordring

- Dette er en oppgave for de som har lyst til å se hvor gode de er til å resonnere rundt de generelle definisjonene vi har gitt.
- Vi forventer ikke at studentene skal kunne løse slike oppgaver til eksamen.

Oppgave.

La R være en relasjon på en mengde A.

Vi kaller R en preordning hvis R er transitiv og refleksiv.

Definer relasjonen S på A ved aSb når $aRb \wedge bRa$.

a) Vis at S er en ekvivalensrelasjon på A.

Der er vanlig å skrive A/S for mengden av ekvivalensklasser $E(a)$ til ekvivalensrelasjonen S.

Oppgave (Fortsatt).

Vi definerer en relasjon $\hat{R}$ på A/S ved

$$E(a)\hat{R}E(b)$$

om aRb

b) Vis at det ikke spiller noen rolle hvilke elementer i ekvivalensklassene $E(a)$ og $E(b)$ vi bruker.

c) Vis at $\hat{R}$ er en partiell ordning på mengden av ekvivalensklasser.

En digresjon

- Hvis A er en endelig mengde finnes det $|A|^2$ elementer i $A \times A$, og

$$2^{|A|^2}$$

forskjellige relasjoner på A.

- Hvis eksempelvis $|A| = 10$ vil det finnes 2^{100} relasjoner på A.
- Hvis vi trekker en relasjon helt vilkårlig, ved eksempelvis myntkast for hvert par (a, b), er sannsynligheten forsvinnende liten for at resultatet blir en av de pene relasjonstypene vi har sett på.

MAT1030 – Forelesning 13

Funksjoner

Dag Normann - 2. mars 2010

Kapittel 6: Funksjoner

Forrige uke

- Forrige forelesning snakket vi om relasjoner.
- Vi snakket om ekvivalensrelasjoner og ekvivalensklasser.
- Vi definerte partielle ordninger og totale ordninger.
- Deretter begynte vi å snakke litt om funksjoner.
- Det skal vi fortsette med nå.
- Vi gjentar de siste minuttene fra onsdag.
- Før det: Er det noen spørsmål?

Funksjoner

- Fra skolematematikken kjenner man funksjoner definert ved uttrykk som
  - $f(x) = x^2 + 3x + 4$
  - $g(x) = \sin x$
  - $h(x) = e^{2\ln(x)+27}$
- Vi har også støtt på tallteoretiske funksjoner som
  - Heltallsverdien til $\frac{n}{m}$.
  - Primtall nr. n.
  - Største felles divisor av n og m.

  Felles for alle disse eksemplene er at det er en sammenheng mellom

  argumentet eller argumentene

  til funksjonen, og

  verdien

  til funksjonen.

  Det er vanlig å fremstille en funksjon som en slags svart boks hvor
  - Vi forer boksen med funksjonsargumentene.
  - Boksen bearbeider dataene.
  - Boksen gir oss funksjonsverdien.

  Som intuitivt bilde er dette en grei beskrivelse.
  Som presis matematisk definisjon holder det ikke helt.
  Vi skal stort sett holde oss til presisjonsnivået i boka.

Hovedpoenget er at vi skal tolke ordet *regel* på neste side så liberalt som mulig.

For fullstendighetens skyld, skal vi, om noen minutter, ta med den vanlige formelle definisjonen av hva en funksjon er.

Definisjon.

La X og Y være to mengder.

En funksjon $f : X \to Y$ er en regel som for hver $x \in X$ gir oss en, og bare en, $y = f(x)$ i Y.

Merk.

- Som sagt skal ordet *regel* tolkes så liberalt som mulig, men, som vi skal se, med en smule forsiktighet.
- Det er ikke noe krav om at det skal foreligge noen regler som mennesker eller maskiner kan følge.
- Det eneste kravet er at $f(x)$ skal fins i Y, og at uttrykket $f(x)$ ikke kan være flertydig.

Eksempler

Eksempel (Funksjoner).

- Vi skal se på noe av det vi har snakket om før, og se hvorvidt funksjoner er involvert.
- Vi skal legge vekt på andre eksempler enn de dere kjenner fra skolematematikken.
- La $\mathcal{E}$ være en universell mengde, og la A være en delmengde av $\mathcal{E}$
- Vi definerte $\overline{A} = \{x \in \mathcal{E} \ : \ x \notin A\}$.
- Hver A har en og bare en komplement-mengde, så $f(A) = \overline{A}$ er en funksjon.
- Her er X lik Y lik potensmengden til $\mathcal{E}$.
- På samme måte kan vi oppfatte $\cap$ og $\cup$ som funksjoner.
- Hvis Y er potensmengden til $\mathcal{E}$ og $X = Y^2$, vil $\cap$ og $\cup$ være funksjoner fra X til Y.

Eksempel.

- Quicksort er en algoritme som sorterer elementene i en liste etter størrelse.
- Quicksort gir mening hver gang listen er hentet fra en totalt ordnet mengde.
- Da beregner Quicksort funksjonen som tar en vilkårlig liste som argument og gir den sorterte listen som verdi.

- Vi kan finne en annen algoritme Slowsort for sortering av elementene i en liste, og den vil definere den samme funksjonen, men være en annen algoritme.
- Det er forbindelsen mellom argument og verdi som bestemmer hvilken funksjon vi har, ikke hvordan vi kommer fra argument til verdi.

Definisjonsområdet og verdiområdet

Definisjon.

a) Hvis $f : X \to Y$ er en funksjon kaller vi
   - X for definisjonsområdet til f.
   - Y for verdiområdet til f.

b) Bildet eller bildemengden til f er

$$\{f(x) \ : \ x \in X\}$$

c) Vi kan skrive $f[X]$ for bildet til f.

Eksempel.

- Definer $f(x) = e^x$ som en funksjon fra $\mathbb{R}$ til $\mathbb{R}$.
  Da vil:
  - *Definisjonsområdet* til f være hele $\mathbb{R}$.
  - *Verdiområdet* til f være hele $\mathbb{R}$.
  - *Bildet* til f være mengden av positive reelle tall.

Eksempel.

- La $A \subseteq \mathcal{E}$ være en mengde, og definer

$$f_A(B) = A \cap B$$

  som en funksjon fra potensmengden X til $\mathcal{E}$ til seg selv.
- Da er X både definisjonsområdet og verdiområdet til f_A, mens bildemengden vil være potensmengden til A.

Eksempel.

- Innledningsvis i disse forelesningene konstruerte vi en pseudokode for å beregne $|n - m|$ fra n og m.
- I dette tilfellet var definisjonsområdet mengden av par av ikke-negative hele tall, verdiområdet og bildemengden lik mengden av ikke-negative hele tall.
- Da vi satte opp sannhetsverditabeller for sammensatte utsagn i utsagnsvariablene p, q og r, konstruerte vi i virkeligheten funksjoner fra mengden X av tripler fra $\{\mathbf{T}, \mathbf{F}\}$ til $Y = \{\mathbf{T}, \mathbf{F}\}$.

  To sammensatte utsagn er logisk ekvivalente når disse funksjonene er like.

- En sannhetsverditabell beskriver egentlig en funksjon på tabellform.
- I prinsippet kan alle funksjoner hvor definisjonsområdet er en liten, endelig mengde beskrives på tabellform.
- Vi illustrerer det på tavlen.
- En alternativ måte å beskrive slike funksjoner er ved å bruke et pildiagram, se illustrasjon på tavlen.

Merk.

- Læreboka bruker *domain* for definisjonsområde og *codomain* for verdiområde.
- Det er ikke uvanlig å bruke ordene *domene* og *kodomene* på norsk.
- Vi skal holde oss til betegnelsene *definisjonsområde* og *verdiområde* i disse forelesningene.

Hvis vi skal gi en enda mer stringent innføring i funksjoner, kan vi definere en funksjon som følger:

En funksjon fra A til B er en delmengde $f \subseteq A \times B$ slik at

- For alle $a \in A$ finnes en og bare en $b \in B$ slik at $(a, b) \in f$.
- Vi skriver $b = f(a)$ for $(a, b) \in f$.

Hvis man skal bruke mengdelæren til å legge et grunnlag for matematikken, er dette den *offisielle* definisjonen av hva en funksjon er.

Her er regelen for å beregne $f(a)$ at

$$f(a) \text{ er den } b \text{ som er slik at } (a, b) \in f.$$

Vi skal fortsette med det presisjonsnivået vi har lagt opp til.

Injektive funksjoner

- Funksjoner har en ting felles med relasjoner:

  Noen av dem har spesielle egenskaper som er verd en egen betegnelse.
- Vi skal først se på injektive funksjoner.
- Andre betegnelser er 1-1-funksjoner og enentydige funksjoner.

Eksempel (Injektive funksjoner).

- La w være et 32-bits binært tall, og la $f(w)$ være det heltallet som representeres av w.
- Hvis $v \neq w$ vil $f(v) \neq f(w)$ siden vi ikke bruker to forskjellige binære representasjoner av det samme tallet.
- Hvis vi oppfatter f som en *input-output*-forbindelse, ser vi at det er én input per output.
- Dette er et eksempel på en en-til-en-funksjon eller *injektiv* funksjon.

Eksempel (Fortsatt).

- En telefonselger vil ha datastøttet oppringing til kunder.
- For ikke å irritere kundene unødig, bør ikke telefonselgeren kontakte samme kunde to ganger.
- Hvis $R(n)$ er kunde nummer n selgeren kontakter, betyr dette kravet at $R(n) \neq R(m)$ når $n \neq m$.
- Programmet selgeren støtter seg på må være slik at oppringningsfunksjonen R blir injektiv, eller enentydig.

Den formelle definisjonen vil være:

Definisjon.

La $f : X \to Y$ være en funksjon.

f kalles injektiv hvis vi for alle x og y i X har at

$$x \neq y \Rightarrow f(x) \neq f(y).$$

Merk.

- Vi har brukt den kontrapositive versjonen i definisjonen.
  En ekvivalent formulering vil være
  $$f(x) = f(y) \Rightarrow x = y.$$

Eksempel (Injektive funksjoner).

- La $f(x) = x^2$ være en funksjon fra $\mathbb{R}$ til $\mathbb{R}$.

Da er ikke f injektiv fordi $1 \neq -1$ mens $f(1) = f(-1)$.

- Hvis vi begrenser definisjonsområdet til de ikke-negative tallene $\mathbb{R}_{\geqslant 0}$ blir funksjonen injektiv.
- Funksjonen som ordner en sekvens av ord alfabetisk er ikke injektiv, fordi ord-sekvensene "Per, Pål, Espen" og "Espen, Pål, Per" er forskjellige, men de blir like når vi ordner dem alfabetisk.

Eksempel.

- La $A = \{1, \ldots, 100\}$
- La $f(a)$ være tverrsummen til $a \in A$ når vi antar at vi bruker vanlig titallsystem.
- f er en funksjon, hvor A er definisjonsområdet.
- I dette tilfellet har vi ikke bestemt verdiområdet, men uten å tenke oss om vet vi at tverrsummen vil være et naturlig tall.
- Vi ser derfor på f som en funksjon

$$f : A \to \mathbb{N}$$

Eksempel (Fortsatt).

- Er f injektiv?
- Kravet var at for alle a og b i A, så skal $f(a) \neq f(b)$ når $a \neq b$.
- Men $f(63) = f(72) = 9$, så f er ikke injektiv.
- Kan vi bestemme bildemengden til f?
- Bildemengden vil være $B = \{1, 2, 3, 4, \ldots, 18\}$.

Eksempel (Fortsatt).

- Vi definerer nå to relasjoner på A ved hjelp av f:
- La aRb hvis $f(a) = f(b)$.
- La aSb hvis $f(a) < f(b) \vee (aRb \wedge a \leqslant b)$
- Når vi har relasjoner som dette, bør vi stille følgende spørsmål:
  - Er R eller S refleksiv?
  - Er R eller S irrefleksiv?
  - Er R eller S symmetrisk?
  - Er R eller S antisymmetrisk?
  - Er R eller S transitiv?

Eksempel (Fortsatt).

- La oss se på R først.
  1. Vi ser at R er *refleksiv* fordi $f(a) = f(a)$ for alle $a \in A$.
  2. Vi ser at R er *symmetrisk* fordi $f(b) = f(a)$ når $f(a) = f(b)$.
  3. Vi ser at R er *transitiv* fordi $f(a) = f(c)$ hvis vi har at $f(a) = f(b)$ og at $f(b) = f(c)$.
  4. Siden $A \neq \emptyset$ og R er refleksiv, er ikke R samtidig *irrefleksiv*.
  5. Siden f ikke er injektiv og R er symmetrisk, kan ikke R være *antisymmetrisk*.
- 1., 2. og 3. viser at R er en ekvivalensrelasjon.
- Vi har ikke brukt noen spesielle egenskaper ved A eller f, så relasjoner konstruert på denne måten vil alltid være ekvivalensrelasjoner.

Eksempel (Fortsatt).

- Når vi nå vet at R er en ekvivalensrelasjon, kan vi prøve å finne ekvivalensklassene.
- Vi vil ha en ekvivalensklasse for hver tverrsum:
  1. $\{1, 10, 100\}$
  2. $\{2, 11, 20\}$
  3. $\{3, 12, 21, 30\}$
  4. $\{4, 13, 22, 31, 40\}$
  5. $\{5, 14, 23, 32, 41, 50\}$

     osv.

Eksempel (Fortsatt).

- La oss så se på egenskapene til S.
- $aSb \Leftrightarrow f(a) < f(b) \vee (aRb \wedge a \leqslant b)$
  - S er *refleksiv* fordi $aRa \wedge a \leqslant a$ for alle a, så andre ledd i disjunksjonen er alltid sant.
  - La aSb og bSc.
    - Hvis $f(a) < f(b)$ eller $f(b) < f(c)$ vil $f(a) < f(c)$, og aSc.
    - Hvis $f(a) = f(b) = f(c)$ har vi at $a \leqslant b \leqslant c$, så da har vi også at aSc.
  - Vi ser at S er *transitiv*.
  - Anta at aSb og bSa
    - Da må $f(a) \leqslant f(b)$ og $f(b) \leqslant f(a)$, så $f(a) = f(b)$, det vil si at aRb.
    - Da er $a \leqslant b$ og $b \leqslant a$ så $a = b$
  - Det følger at S er *antisymmetrisk*.
- Konklusjonen er at S er en partiell ordning.

Følgende oppgave skal leses i sammenheng med teksten på de foregående sidene.

Oppgave.
Vis at S er en total ordning, og skriv ned de 10 S-første tallene.

Eksempel (Injektive funksjoner).

- Vi skal se på tre funksjoner, og spørre om de er injektive:
  1. A er alle ord w godkjent som norske ord, og $f(w)$ er ordet w skrevet baklengs.
  2. B er mengden av uendelige desimalutviklinger α og $g(\alpha)$ er det tilsvarende reelle tallet.
  3. C er mengden av positive reelle tall som har en eksakt 32-bits representasjon r, og hvis r representerer tallet x lar vi $h(r)$ være tallet som representerer $\sqrt{x}$.
- f vil være injektiv, for hvis vi speiler to forskjellige ord, vil speilbildene bli forskjellige.

Eksempel (Fortsatt).

- g er ikke injektiv fordi noen reelle tall kan ha to forskjellige desimalutviklinger, eksempelvis

$$0,9999\cdots = 1,0000\cdots.$$

- h kan ikke være injektiv fordi hvis et tall ligger mellom 2^{-24} og 2^{24}, vil kvadratroten ligge mellom 2^{-12} og 2^{12}.

  Vi har altså langt færre binære tall til disposisjon for å representere $\sqrt{x}$ enn x selv, så funksjonen kan ikke være injektiv.
- Her har vi egentlig brukt noe som kalles skuffeprinsippet, dueslagsprinsippet eller på engelsk the pigeon hole principle.

MAT1030 – Forelesning 14

Mer om funksjoner

Dag Normann - 3. mars 2010

Kapittel 6: Funksjoner

Injektive funksjoner

- I går begynte vi på kapitlet om funksjoner

$$f : X \to Y,$$

  og vi brukte betegnelsene
  - definisjonsområdet for X
  - verdiområdet for Y
  - bildemengden $f[X]$ for $\{f(x) : x \in X\}$.
- Det som kjennetegner en funksjon er at det

$$\text{for alle } x \in X \text{ finnes } \textit{en og bare en } y \in Y \text{ slik at } y = f(x).$$

- Vi gikk deretter over til å se på injektive funksjoner:

$$x \neq y \Rightarrow f(x) \neq f(y).$$

- Vi vil normalt bruke Injektive eller enentydige funksjoner når vi skal registrere fysiske objekter digitalt.
- Det er da ofte et poeng at verdiområdet er vesentlig større enn bildemengden.
- Et slående eksempel er registreringen av personer som et 11-siffret tall i form av fødselsnummer.
- Det er viktig at forskjellige personer har forskjellige fødselsnummer.
- Det er også viktig at det finnes langt flere 11-siffrede tall enn personer.
- På denne måten kan de virkelige fødselsnumrene fordeles slik at hvis man skriver feil, vil resultatet normalt ikke bli et gyldig fødselsnummer.
- Hver bankkonto vil ha et kontonummer, også med 11 siffre.
- Forskjellige konti har forskjellige kontonummer.
- Mange konti tillater bruk av betalingskort, og mange kort er utstyrt med PIN-koder.
- Funksjonen som gir PIN-koden til et kort er ikke enentydig.
- Foreleser hadde en gang to kort med samme PIN-kode, et bankkort og et adgangskort til universitetets bygninger.

Eksempel.
- Enkelte ganger kan det være aktuelt å bake en lokal relasjonsdatabase inn i en større.

- La oss anta at endel lokale bibliotek har digitalisert sine bokregistere, og at man nå ønsker å lage en nasjonal base for alle landets biblioteker.
- Da må vi konstruere en injektiv funksjon for hvert lokale bibliotek, som sender representasjonen av hver enkelt bok i den lokale basen på en representasjon av samme bok i den nasjonale basen.

Eksempel (fortsatt).

- Det vil finnes noen funksjoner, som forfatter, utgivelsesår, forlag m.m. som skal bevares.
- I det hele skal all informasjon lagret i den lokale basen kunne gjennfinnes i den nasjonale, mens den nasjonale basen gjerne kan inneholde mer informasjon om hver enkelt bok.
- En injektiv funksjon som bevarer informasjon på denne måten, kalles ofte for en embedding.
- Vi skal ikke gi en presis matematisk definisjon av hva vi mener med en *embedding* i disse forelesningene, men injektive funksjoner, i form av embeddinger, spiller en stor rolle i forståelsen av sammenhenger mellom forskjellige relasjonsdatabaser.

Eksempel.

- Et eksempel på embeddinger er strengt voksende funksjoner.
- Hvis A og B er to mengder, R er en relasjon på A og S er en relasjon på B, er en injektiv funksjon $f : A \to B$ en embedding hvis vi for alle x og y i A har

$$xRy \Leftrightarrow f(x)Sf(y).$$

- Hvis R og S er ordninger svarer dette til at f er strengt voksende.

Surjektive funksjoner

Den neste gruppen av funksjoner vi skal se på er de surjektive funksjonene:

Definisjon.

La $f : X \to Y$ være en funksjon.

f kalles surjektiv hvis bildemengden til f er hele Y.

På engelsk brukes ofte betegnelsen onto og på norsk kan vi si at f går fra X på Y.

Eksempel (Surjektive funksjoner).

- La $f : \mathbb{R} \to \mathbb{R}_{\geqslant 0}$ være gitt ved
$$f(x) = x^2.$$
Da er f surjektiv.
- $f_A(B) = A \cap B$ er surjektiv som en funksjon fra potensmengden til $\mathcal{E}$ til potensmengden til A.
- La $\text{PRIM} : \mathbb{N} \to \mathbb{N}$ være definert ved at $\text{PRIM}(n)$ er primtall nr. n.
Da er ikke PRIM en surjektiv funksjon, fordi verdimengden er hele $\mathbb{N}$, mens bildemengden er mengden av primtall.

Eksempel (Fortsatt).

- Enhver funksjon vil være surjektiv hvis vi setter verdiområdet lik bildemengden.
- Det er ofte i de tilfellene hvor det er uklart hva bildemengden er at det kan være relevant å spørre seg om en funksjon er surjektiv eller ikke.
- Selv om vi finner en algoritme for å beregne en
$$f : \mathbb{N} \to \mathbb{N}$$
trenger det ikke finnes noen algoritme for å avgjøre om $n \in f[\mathbb{N}]$.
En funksjon f og bildemengden til f kan ha forskjellig kompleksitet.

Merk.

- I dette kurset vil injektive funksjoner spille en større rolle enn surjektive funksjoner.
- Hvis vi konstruerer digitale representasjoner for elementene i en mengde A, konstruerer vi samtidig en funksjon F fra representantene for elementene i A til A selv.
- Vi vil normalt ønske å vite om vi får representert alle elementene i A (håpløst hvis ikke A er endelig).
- Dette er det samme som å spørre om F er surjektiv.

Sammensetning av funksjoner

Eksempel (Sammensatte funksjoner).

- Anta at vi har en maskin som benytter 16-bit representasjoner av hele tall.

- La a være et helt tall slik at
$$-32767 \leqslant a \leqslant 32767.$$
- Hvis vi skal beregne $f(a) = -a$ på maskinen trenger vi å
  1. Finne den digitale representasjonen av a.
  2. Beregne den digitale representasjonen av $-a$ fra den digitale representasjonen til a
  3. Finne $-a$ ut fra den digitale representasjonen til $-a$.
- Vi ser at det er tre funksjoner involvert her, og vi har bruk for sammensetningen av dem.

Eksempel (Sammensatte funksjoner).

- I skolematematikken, og i noen kurs i matematisk analyse, ser vi på sammensetninger av funksjoner definert på $\mathbb{R}$ eller delmengder av $\mathbb{R}$.
- Kjerneregelen for derivasjon forteller oss hvordan vi kan derivere slike sammensetninger.

Eksempel (Sammensatte funksjoner).

- La oss se på følgende pseudokode.

1. *Input* x [x naturlig tall]
2. $y \leftarrow 1$
3. **While** $x > 0$ **do**
   3.1 $y \leftarrow 2y$
   3.2 $x \leftarrow x - 1$
4. $z \leftarrow 1$
5. **While** $y > 0$ **do**
   5.1 $z \leftarrow 3z + 1$
   5.2 $y \leftarrow y - 1$
6. *Output* z

Eksempel (Fortsatt).

- Denne pseudokoden deler seg naturlig i to deler.
- Instruksjonene 1. - 3. beregner $y = f(x) = 2^x$.
- Instruksjonene 4. - 6. beregner z som en funksjon $z = g(y)$, hvor vi ikke har noen opplagt formel for $g(y)$.
- Tilsammen vil pseudokoden definere en sammensatt funksjon $z = g(f(x))$.

Definisjon.

La $f : X \to Y$ og $g : Y \to Z$ være to funksjoner.

Vi definerer sammensetningen $h = g \circ f$ som funksjonen

$$h : X \to Z$$

vi får ved først å bruke f på argumentet x og så g på mellomverdien $f(x)$.

Vi skriver også

$$h(x) = g(f(x)).$$

Eksempel.

- La $f : \mathbb{N} \to \mathbb{N}$ være definert ved at $f(n)$ er primtall nummer n og la $g : \mathbb{N} \to \mathbb{N}$ være definert ved at $g(m) = m^2$

  Da er $(g \circ f)(4) = g(f(4)) = g(7) = 49$.

  $(f \circ g)(4) = f(g(4)) = f(16) = 53$.
- Vi ser altså at selv om sammensetningen av funksjonene gir mening for begge rekkefølgene, kan det spille en rolle i hvilken rekkefølge vi setter dem sammen.

Eksempel (Fortsatt).

- La f sende binærformen til et naturlig tall n over til desimalformen og la g gi oss tverrsummen av desimalformen til et tall.

  Da gir det ingen mening å snakke om $f \circ g$ fordi definisjonsområdet til f er mengden av binære representasjoner av naturlige tall og verdiområdet til g er en mengde av naturlige tall.

  I MAT1030 er det viktig å skille mellom tall og de forskjellige representasjonene av tallene.

  $g \circ f$ gir mening. Definisjonsområdet vil være mengden av binære representasjoner, "mellomområdet" vil være mengden av desimaltallsrepresentasjoner og verdiområdet vil være naturlige tall.

  $(g \circ f)(100110) = g(38) = 11$.

Teorem.

La $f : X \to Y$ og $g : Y \to Z$ være to funksjoner.

La $h = g \circ f$ være sammensetningen av f og g.

a) Hvis både f og g er injektive, er h injektiv.

b) Hvis både f og g er surjektive, er h surjektiv.

Bevis.

a) Anta at f og g er injektive.

$x \neq y \Rightarrow f(x) \neq f(y) \Rightarrow g(f(x)) \neq g(f(y))$.

Siden $\Rightarrow$ er *transitiv*, $h(x) = g(f(x))$ og $h(y) = g(f(y))$ følger det at h er injektiv når f og g er det.

b) Anta at f og g er surjektive, og la $z \in Z$ være vilkårlig.

Siden g er surjektiv, fins $y \in Y$ slik at $g(y) = z$.

Siden f er surjektiv, fins $x \in X$ slik at $f(x) = y$.

Da er $z = g(f(x)) = h(x)$.

- For mange programmeringsspråk kan gjennomkjøringen av et program oppfattes som en styrt sammensetning av mange funksjoner.
- Vi så på et eksempel hvor en pseudokode beskrev en algoritme for en sammensatt funksjon.
- Egentlig kan enhver instruksjon

$$x_i \leftarrow t(x_1, \ldots, x_n)$$

i en pseudokode oppfattes som en ordre om at vi skal bruke funksjonen t der og da.
- Siden det å bruke disse funksjonene utgjør enkelttrinnene i beregningen, får vi resultatet etter å ha satt sammen slike instruksjoner.
- *Advarsel!*
Programmer, eller pseudokoder, skal egentlig oppfattes som en generalisering av sammensatte funksjoner, ettersom rekkefølgen vi bruker funksjonene i avhenger av startverdien på variablene.

Inverse funksjoner

Eksempel (Inverse funksjoner).

- Med fare for å overfokusere på et tema skal vi nok en gang se på digital representasjon av tall.
- La X være mengden av reelle tall som har en digital representasjon med enkel presisjon.
- La Y være mengden av 32-bits representasjoner av reelle tall.
- La $F : X \to Y$ være funksjonen som til et tall x gir oss den digitale representasjonen av x.
- La $G : Y \to X$ være funksjonen som til en digital representasjon y av et tall gir oss tallet.
- Da er $G(F(x)) = x$ for alle $x \in X$ og $F(G(y)) = y$ for alle $y \in Y$.
- Vi sier at G er den inverse av F.

Eksempel (Inverse funksjoner).

- Fra skolematematikken og begynneremnene i matematikk har vi flere eksempler på par av funksjoner som "opphever" hverandre:
  1. $f : \mathbb{R} \to \mathbb{R}_{>0}$ definert ved $f(x) = e^x$ og $g : \mathbb{R}_{>0} \to \mathbb{R}$ definert ved $g(y) = \ln(y)$.
  2. Tangensfunksjonen $f : \langle -\frac{\pi}{2}, \frac{\pi}{2} \rangle \to \mathbb{R}$ og Arcustangens-funksjonen $g : \mathbb{R} \to \langle -\frac{\pi}{2}, \frac{\pi}{2} \rangle$.
  3. $f(x) = 2x$ og $g(y) = \frac{y}{2}$
- Vi har utallige eksempler på par av funksjoner som representerer "omvendte regningsmåter av hverandre".

Eksempel (Inverse funksjoner).

- La $f : \mathbb{J} \to \mathbb{J}$ være definert ved at $f(a)$ er heltallsverdien til $\frac{a}{2}$, det vil si det største hele tallet b slik at $b \leqslant \frac{a}{2}$.
- Kan vi finne en omvending av f?
- La oss regne på noen verdier, og la oss se hva som skjer:
- $\cdots$, $f(-2) = f(-1) = -1$, $f(0) = f(1) = 0$, $f(2) = f(3) = 1$, $\cdots$
- Hvis vi skal lage en omvendt funksjon g har vi to valg for $g(1)$, nemlig $g(1) = 2$ og $g(1) = 3$.
- Velger vi $g(1) = 2$, vil $g(f(3)) \neq 3$, og velger vi $g(1) = 3$ får vi $g(f(2)) \neq 2$.
- Vi ser at siden f ikke er injektiv, får vi et problem.

Eksempel (Inverse funksjoner).

- La $P : \mathbb{N} \to \mathbb{N}$ være definert ved at $P(n)$ er primtall nummer n i den voksende opplistingen

$$2, 3, 5, 7, 11, 13, 17, 19, \ldots.$$

- P er injektiv, men P har ingen omvendt funksjon Q.
- Problemet er at vi ikke kan definere eksempelvis $Q(15)$ siden 15 ikke er et primtall og derfor ikke har noe nummer.
- Hvis vi for eksempel prøver oss med $Q(15) = 7$, får vi

$$P(Q(15)) = 17 \neq 15.$$

- Vi ser at problemet ligger i at P ikke er surjektiv.

Definisjon.

La $f : X \to Y$ være en funksjon.

$g : Y \to X$ kalles en invers til f, eller en omvendt funksjon av f, hvis

- $g(f(x)) = x$ for alle $x \in X$.
- $f(g(y)) = y$ for alle $y \in Y$.

Teorem.

a) La $f : X \to Y$ være en funksjon.
Da har f en invers funksjon $g : Y \to X$ hvis og bare hvis f er både injektiv og surjektiv.

b) En funksjon f kan ikke ha mer enn én invers.

Definisjon.

Hvis $f : X \to Y$ har en invers, skriver vi f^{-1} for den inverse.

Bevis.

- Anta først at $f : X \to Y$ er både injektiv og surjektiv.
  La $y \in Y$.
  Siden f er surjektiv, fins det $x \in X$ slik at $f(x) = y$, og siden f er injektiv fins det bare en slik x.
  Da lar vi $g(y) = x$, og g vil være en invers.

Bevis (Fortsatt).

- Anta så at f har en invers g.
  - $x \neq y \Rightarrow g(f(x)) \neq g(f(y)) \Rightarrow f(x) \neq f(y)$.
    Derfor er f injektiv.
  - La $y \in Y$ og la $x = g(y)$.
    Da er $y = f(x)$.
    Det følger at f er surjektiv.
- Definisjonen under første punkt er den eneste måten å finne en invers på, så det kan ikke fins flere.

Oppgave.

a) Vis at hvis $f : X \to Y$ og $g : Y \to Z$ begge har inverse f^{-1} og g^{-1}, så vil sammensetningen

$$h = g \circ f$$

også ha en invers.

b) Under antagelsene i a), vis at $h^{-1} = f^{-1} \circ g^{-1}$.

- Vi kan bruke kvantorer til å gi presise definisjoner av *injektiv*, *surjektiv* og *sammensetning*:
- $f : X \to Y$ er injektiv hvis $\forall x \in X \forall y \in X(f(x) = f(y) \to x = y)$.
- $f : X \to Y$ er surjektiv hvis $\forall y \in Y \exists x \in X(y = f(x))$.
- Hvis $f : X \to Y$, $g : Y \to Z$ og $h = g \circ f$ vil

$$\forall x \in X \forall z \in Z(h(x) = z \leftrightarrow \exists y \in Y(y = f(x) \wedge z = g(y))).$$

De som ønsker å forstå hvordan kvantorer brukes, bør overbevise seg selv om at dette er riktig.

Funksjoner og programmering

- De fleste programmeringsspråk er utstyrt med predefinerte funksjoner.
- For noen av disse språkene kan man også definere sine egne funksjoner.
- En lommeregner har eksempelvis knapper for algebraiske operasjoner, trigonometriske funksjoner , logaritme- og eksponensialfunksjoner m.m.
- Noen lommeregnere tillater også at vi definerer våre egne funksjoner ved hjelp av sammensatte uttrykk, og tildels enkle programmer.
- Matematisk sett opererer disse funksjonene på representasjoner av tall og andre størrelser i lommeregneren eller på datamaskinen.
- Dette diskuteres i tilstrekkelig detalj i læreboka.

- Noen hjelpemidler kan gå under betegnelsen "funksjoner" i en programmeringssammenheng, men er ikke funksjoner i vanlig matematisk forstand.
- De viktigste er de som genererer et tilfeldig element i en mengde.
- Ber vi om en kabal, eller et minesveiperspill, får vi et nytt spill hver gang.
- Skal vi teste om et stort tall n er et primtall utfører vi en algebraisk test på tilfeldig valgte tall $a_1, \ldots, a_k$ mindre enn n.

  For hver test vil svaret *NEI* fortelle oss at n ikke er et primtall, mens svaret *JA* forteller oss at sannsynligheten er $\frac{3}{4}$ for at n er et primtall.
- Denne primtallstesten kan gi forskjellige svar på om n er et primtall, og er derfor ikke en funksjon i matematisk forstand.
- Ved å la antall vilkårlige deltester være stort, eksempelvis 100, er usikkerheten omkring svaret i størrelsesorden 2^{-200}, så for alle praktiske formål er primtallstesten en funksjon.

Beregnbare funksjoner

- IT dreier seg mye om hvordan man løser oppgaver ved hjelp av elektroniske hjelpemidler, fortrinnsvis datamaskiner.
- All IT-aktivitet på maskin-nivå styres av programmer, uansett om vi ser dem eller ikke.
- Hvis man skal kunne forstå informasjonsteknologiens begrensninger, må vi derfor forstå grensene for hva det er mulig å skrive programmer for.
- Alle programmer beskriver egentlig funksjoner, selv om noen argumenter (som maskintid, maskinarkitektur o.a.) ikke er synlig.
- Det er derfor av interesse å studere de funksjonene som lar seg uttrykke ved hjelp av programmer.

- Hvis vi begrenser oss til funksjoner fra $\mathbb{N}_0$ til $\mathbb{N}_0$ har vi gode matematiske karakteriseringer av de beregnbare funksjonene, det vil si de som kan programmeres i et eller annet programmeringsspråk. ($\mathbb{N}_0 = \mathbb{N} \cup \{0\}$)
- Det viser seg at alle programmerbare funksjoner fra $\mathbb{N}_0$ til $\mathbb{N}_0$ kan formuleres som en av våre pseudokoder, hvor vi bare bruker navn på tallene 0 og 1, addisjon og multiplikasjon og Boolske tester uttrykt ved hjelp av $=$ og $<$.

  Det er ikke uvanlig for logikere eller folk som arbeider med teoretisk databehandling å la de naturlige tallene starte med 0.

  Vi skal være snille og holde oss til måten boka gjør det på.

  Som en forberedelse til kapittel 7 om induksjon og rekursjon, skal vi se på to pseudokoder hvor vi har pålagt oss å begrense oss til addisjon, multiplikasjon og Boolske tester med $=$ og $<$ (men hvor vi dermed får lov til å bruke $\leqslant$).
- I det første eksemplet skal vi beregne $f(x, y) = max\{0, x - y\}$.
- I det andre eksemplet skal vi beregne $g(x, y) = x^y$.

Eksempel (Beregnbare funksjoner).

1. *Input* x [$x \in \mathbb{N}_0$]
2. *Input* y [$y \in \mathbb{N}_0$]
3. $z \leftarrow 0$
4. **While** $y < x$ **do**

   4.1 $y \leftarrow y + 1$

   4.2 $z \leftarrow z + 1$
5. *Output* z

- Vi har ikke snakket om induksjonsbevis ennå. Det vil være den naturlige metoden for å vise korrekthet av et slikt program.
- I dette tilfellet ser vi at hvis $x \leqslant y$ starter vi ikke løkka i det hele tatt, mens hvis $y < x$ "teller" vi y opp til x samtidig som vi øker verdien av z tilsvarende mye.

Eksempel (Beregnbare funksjoner).

1. *Input* x $[x \in \mathbb{N}_0]$
2. *Input* y $[y \in \mathbb{N}_0]$
3. $u \leftarrow 0$
4. $z \leftarrow 1$
5. **While** $u < y$ **do**
   5.1 $z \leftarrow z \cdot x$
   5.2 $u \leftarrow u + 1$
6. *Output* z

Dette resulterer i at vi multipliserer x med seg selv y ganger, altså at vi beregner x^y.

Dette sto vi igjen med da tiden var ute:

- I programmeringssammenheng er det ikke alltid så lett å vite når et gitt program med et gitt input faktisk gir oss et output i den mengden hvor vi vil ha det.
- I verste fall kan vi skrive programmer for funksjoner hvor det er umulig å bestemme hva definisjonsområdet er.
- Innenfor IT er det derfor naturlig også å studere partielle funksjoner fra en mengde X til en mengde Y.
- Dette vil være funksjoner hvor definisjonsområdet er en delmengde av X og hvor verdiområdet er Y.
- Tolkningen av et program som en funksjon fra et Cartesisk produkt av datatyper til en datatype vil vanligvis være som en partiell funksjon.

MAT1030 – Forelesning 15

Rekursjon og induksjon

Dag Normann - 9. mars 2010

Mengder, relasjoner og funksjoner

Oppsummering

- Vi er nå ferdig med kapitlene 5 og 6 om mengdelære, relasjoner og funksjoner.
- Fra mengdelæren må man beherske følgende:
  - Bruk av mengdebyggeren for å beskrive/definere mengder.
  - Mengdealgebra med union, snitt, $\emptyset$, $\mathcal{E}$ og komplement.
  - Hva vi mener med at en mengde er inneholdt i en annen og med at to mengder er disjunkte.
  - Bruk av Venn-diagrammer til å utforske mengdealgebraiske problemer.
  - Bruk av deMorgans lover og de distributive lovene for å forenkle boolske (mengdealgebraiske) uttrykk.
- Fra avsnittet om relasjoner skal vi ha fått med oss:
  - Hva vi mener med en relasjon, og hva vi mener med at en relasjon er refleksiv, irrefleksiv, symmetrisk, antisymmetrisk og transitiv.
  - Vi må kjenne definisjonene av partiell ordning, total ordning og ekvivalensrelasjon og kunne føre enkle resonementer rundt disse begrepene.
  - Vi må vite hva som menes med ekvivalensklasser og kunne beskrive disse som en del av en oppgaveløsning.
- Det er noen grunnleggende begreper i tilknytning til kapitlet om funksjoner man må kjenne til for å kunne gå eksamensdagen i møte med ro i sinnet.
  - Hva en funksjon er.
  - Injektive funksjoner, også kalt 1-1-funksjoner eller enentydige funksjoner.
  - Surjektive funksjoner, også kalt på (onto).
  - Sammensetning av funksjoner.
  - Omvendte eller inverse funksjoner.

  Det er også viktig å holde orden på hva som menes med:
- *Definisjonsområdet* til en funksjon.
- *Verdiområdet* til en funksjon.
- *Bildemengden* til en funksjon.

  I tillegg bør man kunne vite når
- man kan finne en invers til en funksjon.
- man kan sette sammen to funksjoner.

  Dette sto vi igjen med da tiden var ute sist onsdag:

- I programmeringssammenheng er det ikke alltid så lett å vite når et gitt program med et gitt input faktisk gir oss et output i den mengden hvor vi vil ha det.
- I verste fall kan vi skrive programmer for funksjoner hvor det er umulig å bestemme hva definisjonsområdet er.
- Innenfor IT er det derfor naturlig også å studere partielle funksjoner fra en mengde X til en mengde Y.
- Dette vil være funksjoner hvor definisjonsområdet er en delmengde av X og hvor verdiområdet er Y.
- Tolkningen av et program som en funksjon fra et Cartesisk produkt av datatyper til en datatype vil vanligvis være som en partiell funksjon.

Kapittel 7

Innledning til rekursjon og induksjon

- Vi skal nå starte på avsnittet om rekursive konstruksjoner og bevis ved induksjon.
- Dette er det første stedet hvor årets MAT1030 vil omfatte mer stoff enn det læreboka omfatter.
- Det betyr at forelesningene er å betrakte som pensum, også der de går ut over rammene til læreboka.
- Alt stoff som er eksamensrelevant vil man finne i læreboka eller i forelesningsnotatene som legges ut på nettet.
- Læreboka behandler for det meste rekursjon og induksjon over de naturlige tallene $\mathbb{N}$.
- I en informatikk-sammenheng fins det andre induktivt konstruerte mengder hvor tilsvarende metoder har mening.
- Vi skal etterhvert se på noen generelle og spesielle eksempler av interesse for informatikk.
- Vi skal imidlertid først se på rekursjon i en begrenset, men viktig, forstand.

Rekursjon

Eksempel.

- Vi definerer en funksjon $f : \mathbb{N} \to \mathbb{N}$ ved
  1. $f(1) = 2$
  2. $f(n+1) = 2^{f(n)}$ for alle n.
- Vi har ikke definert f ved en formel, så er f veldefinert?

Eksempel (Fortsatt).

- En test kan jo være om vi er i stand til å skrive et program for f.
- Vi kan oppfatte punktene 1. og 2. på forrige side som en spesifikasjon.

- Vi har tidligere sett hvordan vi kan finne en pseudokode for $g(z) = 2^z$
- Det betyr at vi kan bruke en instruksjon på formen

$$z \leftarrow 2^y$$

  med vissheten om at vi kan erstatte den ene linjen med en pseudokode.
- Da er det lett å lage en pseudokode for f.

Eksempel (Fortsatt).

1. *Input* x $[x \in \mathbb{N}]$
2. $z \leftarrow 2$
3. $i \leftarrow 1$
4. **While** $i < x$ **do**
   4.1 $i \leftarrow i + 1$
   4.2 $z \leftarrow 2^z$
5. *Output* z

Vi kaller $f(x)$ verdien på 2^{er}-tårnet av høyde x.

Eksempel.

- Vårt neste eksempel er en funksjon som brukes mye i matematikk og i sannsynlighetsregning,

$$n \mapsto n!,$$

  eller fakultetsfunksjonen.
- Vi kan bruke omtrent samme formatet som i forrige eksempel.
  1. $1! = 1$
  2. $(n+1)! = n! \cdot (n+1)$ for alle $n \in \mathbb{N}$.
- Vi kan nærmest kopiere pseudokoden fra forrige eksempel, og får følgende algoritme for beregning av $n!$.

Eksempel (Fortsatt).

1. *Input* x $[x \in \mathbb{N}]$
2. $z \leftarrow 1$
3. $i \leftarrow 1$

4. **While** $i < x$ **do**
   4.1 $i \leftarrow i + 1$
   4.2 $z \leftarrow z \cdot (i)$
5. *Output* z

- Læreboka tar utgangspunkt i tallfølger, mens vi tar utgangspunkt i funksjoner.
- Siden funksjoner er et mer generelt begrep, vil det gjøre det enklere å studere rekursjon som et mer generelt fenomen.
- Det er i prinsippet ingen forskjell mellom en uendelig tallfølge og en funksjon definert på $\mathbb{N}$
- Tallfølgen
$$1, 2, 6, 24, 120, 720, \ldots$$
er bare en annen måte å skrive fakultetsfunksjonen på.
- Hvorvidt man i konkrete tilfeller bruker tallfølger eller funksjoner, avhenger av hva som er pedagogisk mest forstandig for anledningen.

- Kan vi gi en bedre begrunnelse for at de to funksjonene vi har sett på er veldefinerte enn at vi kan finne pseudokoder for dem?
- Svaret er selvfølgelig *JA*.
- Vi kan nå alle naturlige tall ved å
  1. Starte med 1
  2. Legge til 1 så mange ganger som nødvendig.
- Hvis vi da definerer en funksjon f ved å bestemme
  1. hva $f(1)$ er
  2. hvordan $f(n+1)$ avhenger av $f(n)$ og n

  har vi bestemt $f(n)$ for alle n.
- Vi kan oppfatte en konkretisering av punktene 1 og 2 over som en spesifikasjon.
- Vi skal se på et eksempel i detalj.

Eksempel.
Vi definerer funksjonen $f(n, m)$ ved rekursjon på n ved
1. $f(1, m) = 2m - 1$
2. $f(n+1, m) = 2f(n, m) - 1$

- Med tilstrekkelig tålmodighet kan vi finne et uttrykk for $f(n, m)$ for hver enkelt n ved følgende.

Eksempel.

1. $f(1, m) = 2m - 1$
2. $f(2, m) = 2f(1, m) - 1 = 2(2m - 1) - 1 = 4m - 3$
3. $f(3, m) = 2f(2, m) - 1 = 2(4m - 3) - 1 = 8m - 7$
4. $f(4, m) = 2f(3, m) - 1 = 2(8m - 7) - 1 = 16m - 15$

   ...

   Vi ser at vi kan gjøre listen av utregninger så lang vi vil, så $f(n, m)$ er definert for alle n og m.

   En annen sak er om vi kan vise den formelen som ser ut til å peke seg ut.

   Da vil vi få bruk for induksjonsbevis.

- Læreboka har brukt **For**-løkker der vi har brukt **While**-løkker.
- Forskjellen er kosmetisk.
- Det viktige er at vi bruker en løkke til å fange opp formatet
  1. $g(1) = a$
  2. $g(n + 1) = f(g(n), n)$
- og at vi har en standard overgang fra en pseudokode for f til en pseudokode for g.
- Vi sier at g er definert fra a og f ved rekursjon.
- Vi beskriver den generelle **For**-løkka på neste side.

1. *Input* n $[n \in \mathbb{N}]$
2. $x \leftarrow a$
3. **For** $m = 2$ **to** n **do**

   3.1 $x \leftarrow f(x, m)$
4. *Output* x

Merk.

Vi sier at klassen av funksjoner programmerbare via en pseudokode er lukket under *definisjoner ved rekursjon*.

Oppgave.

Betrakt følgende pseudokode, hvor det inngår en rekursiv definisjon.

1. *Input* n $[n \in \mathbb{N}]$
2. $x \leftarrow 1$
3. $y \leftarrow 1$

4. $z \leftarrow 1$
5. **For** $m = 2$ **to** n **do**
   5.1 $y \leftarrow y + 1$
   5.2 **For** $k = 1$ **to** y **do**
      5.2.1 $z \leftarrow z + 1$
      5.2.2 $x \leftarrow x + z$
6. *Output* x

Oppgave (Fortsatt).

- Følg beregningen og finn verdien på output for $n = 1$, $n = 2$, $n = 3$ og $n = 4$.
- Hvordan tror du denne følgen fortsetter?
- Vil beregningen stoppe uansett hvilket naturlig tall n vi starter med?

- Til nå har vi bare sett på funksjoner fra $\mathbb{N}_0$ til $\mathbb{N}_0$ definert ved rekursjon.
- Filosofien bak hvorfor rekursive definisjoner gir mening gir oss også muligheten til å betrakte andre definisjonsområder.

Eksempel.

La $f(2) = 1$

Hvis $n \geqslant 2$ definerer vi $f(n+1)$ ved

- $f(n+1) = f(n)$ hvis n ikke er et primtall.
- $f(n+1) = f(n) + 1$ hvis n er et primtall.

Da er $f(n)$ definert for alle tall $n \geqslant 2$ og forteller oss hvor mange primtall det fins $\leqslant n$.

- Foreløpig gir det ikke mening å bruke rekursjon til å definere funksjoner med definisjonsområder som ikke er $\mathbb{N}$, $\mathbb{N}_0$ eller $\{n \in \mathbb{N} : n \geqslant k\}$ for en k.
- Det er imidlertid ingen grunn til at verdiområdet skal bestå av tall, noe vårt neste eksempel vil vise.

Eksempel.

- Vi har en klassisk definisjon av regningsart R_n nummer n:
- $R_1(x, y) = x + y$
- R_2 defineres rekursivt ved

- $R_2(0, y) = 0$
- $R_2(x + 1, y) = R_1(R_2(x, y), y)$

- Hvis $n \geqslant 2$ og R_n er definert, definerer vi R_{n+1} rekursivt ved
  - $R_{n+1}(0, y) = 1$
  - $R_{n+1}(x + 1, y) = R_n(R_{n+1}(x, y), y)$.

Vi ser at R_1 er addisjon, R_2 er multiplikasjon, R_3 er eksponensiering osv.

Eksempel.

- $\mathbb{N} \to \mathbb{N}$ er en vanlig betegnelse på mengden av alle funksjoner fra $\mathbb{N}$ til $\mathbb{N}$.
- Vi kan bruke rekursjon til å definere iterering av en funksjon som følger:
  1. $f^1 = f$
  2. $f^{n+1} = f^n \circ f$
- Vi kan bruke dette til å definere en følge av etterhvert sterkt voksende funksjoner:
  1. $h_1(x) = x + 1$
  2. $h_{n+1}(x) = h_n^{h_n(x)}(x)$

Eksempel (fortsatt).

- Det er lett å regne ut $h_2(2)$, overkommelig å regne ut $h_3(3)$, men vi anbefaler ingen å prøve å regne ut $h_4(4)$.
- Det er ingen stor utfordring å skrive en pseudokode for

$$h(n, x) = h_n(x).$$

- Ta det likevel som en utfordring.

Merk.

- Definisjonen av f^n er gyldig for alle mengder X og alle funksjoner $f : X \to X$.
- Vi sier at en konstruksjon er polymorf hvis den kan gjøres gjeldende for mange spesialtilfeller simultant.
- Dette utdypes mer i spesialiserte emner i informatikkstudiet.

- I informatikksammenheng brukes ofte ordet rekursjon når vi definerer en funksjon f ved hjelp av selvreferanse.
- Vi har tidligere sett på dette eksemplet:

Eksempel.

1. La $n \geqslant 1$
2. La $f(1) = 0$
3. La $f(2n) = 1 + f(n)$
4. La $f(2n+1) = 1 + f(6n+4)$

Problemet var at vi ikke vet om $f(n)$ er definert for alle n.

Eksempel (fortsatt).

- Vi vil imidlertid ha at det finnes en partiell funksjon f som svarer til definisjonen.
- Denne kan vi beskrive ved å definere k-te tilnærming f_k ved rekursjon på k.
  1. $f_k(1) = 0$ for alle k.
  2. $f_1(n)$ er udefinert for $n > 1$.
  3. $f_{k+1}(2n) = f_k(n) + 1$.
  4. $f_{k+1}(2n+1) = f_k(6n+4)$.
- $\{f_k\}_{k\in\mathbb{N}}$ er en voksende følge av partielle funksjoner, og funksjonen f vil være grensen av disse funksjonene når $k \to \infty$.
- Dette er en standard måte å håndtere funksjoner definert ved generell rekursjon på.

Induksjonsbevis

Eksempel.

- La oss gå tilbake til den rekursive definisjonen
  1. $f(1, m) = 2m - 1$
  2. $f(n+1, m) = 2f(n, m) - 1$

  hvor det er naturlig å gjette på at

$$f(n, m) = 2^n \cdot m - (2^n - 1).$$

- Vi har sett at denne formelen stemmer for $n = 1$, $n = 2$, $n = 3$ og $n = 4$ da vi regnet ut

Eksempel (Fortsatt).

1. $f(1, m) = 2m - 1$
2. $f(2, m) = 2f(1, m) - 1 = 2(2m - 1) - 1 = 4m - 3$
3. $f(3, m) = 2f(2, m) - 1 = 2(4m - 3) - 1 = 8m - 7$
4. $f(4, m) = 2f(3, m) - 1 = 2(8m - 7) - 1 = 16m - 15$

- Hvis vi nå forsøker å se om formelen stemmer for $n = 5$ på en slik måte at vi forhåpentligvis finner en forklaring, kan vi regne som følger:

Eksempel (Fortsatt).

$$f(5, m) = 2f(4, m) - 1 =$$

$$2(2^4 m - (2^4 - 1)) - 1 =$$

$$2 \cdot 2^4 m - 2(2^4 - 1) - 1 =$$

$$2^5 m - 2^5 + 2 - 1 =$$

$$2^5 m - (2^5 - 1).$$

Eksempel (Fortsatt).

- I denne utregningen har vi bare brukt at $5 = 4 + 1$
- Vi kunne erstattet 4 med en vilkårlig n og 5 med $n + 1$, og fått utregningen

$$f(n + 1, m) = 2f(n, m) - 1 = 2(2^n m - (2^n - 1)) - 1 =$$

$$2 \cdot 2^n m - 2(2^n - 1) - 1 =$$

$$= 2^{n+1} m - 2^{n+1} + 2 - 1 = 2^{n+1} m - (2^{n+1} - 1).$$

- Dermed har vi gitt det som kalles et induksjonsbevis for at formelen vår er riktig.

- Hvorfor kan vi betrakte argumentet over som et bevis for at formelen holder for alle n?
- Årsaken er at vi nå vet at vi ved direkte utregning kan bevise formelen hver gang noen gir oss en verdi for n, for eksempel $n = 8$.
- Vi vet nå at formelen holder for $n = 5$ og vi vet at siden den holder for $n = 5$ må den holde for $n = 6$.

- Utregningen vår gir imidlertid at formelen også må holde for $n = 7$ og deretter for $n = 8$.
- Siden vi vet at vi med utholdenhet kan fortsette å tenke slik så langt noen kunne ønske, vet vi at formelen vår må holde for alle n.
- Den metoden vi har brukt til å bevise en påstand for alle naturlige tall på kalles som sagt induksjonsbevis.
- I sin enkleste form kan induksjonsbevis formuleres som:

Definisjon.

La $P(n)$ være et predikat med en variabel n for et element i $\mathbb{N}$.

Anta at vi kan bevise

1. $P(1)$
2. $\forall n(P(n) \rightarrow P(n+1))$

Da kan vi konkludere $\forall n P(n)$.

Denne måten å bevise $\forall n P(n)$ på kalles induksjon.

MAT1030 – Forelesning 16

Rekursjon og induksjon

Dag Normann - 10. mars 2010

Forelesning 16

Rekursjon og induksjon

- Tirsdag ga vi endel eksempler på rekursive definisjoner og vi forklarte hva vi mener med induksjonsbevis.
- Vi kommer til å fortsette i dag med å gi eksempler på begge deler.
- Induksjonsbevis er et effektivt matematisk virkemiddel.
- Våre eksempler vil ofte gå ut på å vise at en formel for det generelle leddet i en følge som vi har definert ved rekursjon er riktig.
- I flere eksempler vil den naturlige gangen være

$$\text{problem} \to \text{rekursjon} \to \text{formel} \to \text{induksjonsbevis}$$

- Formatene for å definere en funksjon f på $\mathbb{N}$ ved rekursjon er varianter over
  1. Definer $f(1)$.
  2. Definer $f(n+1)$ som en funksjon av n og $f(n)$.
- Formatene for å bevise et predikat $P(n)$ for alle $n \in \mathbb{N}$ er varianter over
  1. Bevis $P(1)$.
  2. Bevis $P(n) \to P(n+1)$ for vilkårlig $n \in \mathbb{N}$.

Eksempel.

- Definer
  - $f(1) = 1$
  - $f(n+1) = 3f(n) + 1$ for alle $n \in \mathbb{N}$.
- Da har vi
  - $f(1) = 1$
  - $f(2) = 3 \cdot 1 + 1 = 4$
  - $f(3) = 3 \cdot 4 + 1 = 13$
  - $f(4) = 3 \cdot 13 + 1 = 40$
- Vi kan fortsette å regne ut $f(5)$, $f(6)$, $f(7)$, osv. ettersom f er definert ved rekursjon.

Eksempel (Fortsatt).

- Definer
  - $g(1) = 1$
  - $g(n+1) = g(n) + 3^n$
- Da har vi
  - $g(1) = 1$
  - $g(2) = 1 + 3 = 4$
  - $g(2) = 4 + 3^2 = 13$
  - $g(4) = 13 + 3^3 = 40$
- Vi kan fortsette å regne ut $g(5)$, $g(6)$, $g(7)$, osv., og vil finne ut at så langt vi kan se vil $f(n) = g(n)$.

Eksempel (Fortsatt).

- Det er da naturlig å gjette på at $f(n) = g(n)$ for alle n.
- For å vise det, kan vi prøve å vise at de to rekursive definisjonene er de samme, men vi ser jo at rekursjonsskrittet i de to definisjonene ikke likner på hverandre.
- Vi ser at $g(n)$ er summen i en endelig geometrisk rekke

$$g(n) = 1 + 3 + 3^2 + \cdots + 3^{n-1}.$$

- En slik rekke har en kjent sum

$$g(n) = \frac{3^n - 1}{3 - 1} = \frac{3^n - 1}{2}.$$

Eksempel (Fortsatt).

- Siden vi sikkert ikke husker hvordan vi kom frem til denne formelen, og siden vi innfører induksjonsbevis i disse forelesningene, viser vi formelen ved induksjon.

  Setter vi $n = 1$ inn i formelen, får vi

$$\frac{3^n - 1}{2} = \frac{3 - 1}{2} = 1 = g(1)$$

  så formelen stemmer for $n = 1$.
- Anta at formelen stemmer for et tall n, det vil si at

$$g(n) = \frac{3^n - 1}{2}.$$

Eksempel (Fortsatt).

Da er

$$g(n+1) = g(n) + 3^n = \frac{3^n - 1}{2} + 3^n$$

$$= \frac{3^n - 1 + 2 \cdot 3^n}{2} = \frac{3 \cdot 3^n - 1}{2} = \frac{3^{n+1} - 1}{2}$$

som viser at formelen også holder for $g(n+1)$.

Eksempel (Fortsatt).

Hvis $P(n)$ er utsagnet

$$g(n) = \frac{3^n - 1}{2}$$

har vi vist

$P(1)$ for seg

1. $P(1) \to P(2)$
2. $P(2) \to P(3)$
3. $P(3) \to P(4)$

   ...

under ett som spesialtilfeller av $P(n) \to P(n+1)$.

Eksempel (Fortsatt).

Da er eksempelvis $P(17)$ en tautologisk konsekvens av alt det vi har bevist.

Prinsippet bak induksjonsbevis er at vi da vet med sikkerhet at $P(n)$ holder for alle n.

Eksempel (Fortsatt).

La nå $Q(n)$ være påstanden

$$f(n) = \frac{3^n - 1}{2}.$$

Vi skal se at vi også kan vise $\forall n Q(n)$ ved induksjon.

Det vil følge at f og g er de samme funksjonene, eller de samme *følgene*, hvis man ønsker å se på det på den måten.

Eksempel (Fortsatt).

- Induksjonstarten er grei, siden $f(1) = 1 = g(1)$ og vi vet at formelen holder for g.
- Anta så at
$$f(n) = \frac{3^n - 1}{2}.$$
- Da er
$$f(n+1) = 3 \cdot f(n) + 1 = 3 \cdot \frac{3^n - 1}{2} + 1 = \frac{3^{n+1} - 3}{2} + \frac{2}{2} = \frac{3^{n+1} - 1}{2}$$
- Dette viser induksjonskrittet, hvis $Q(n)$ holder, så vil $Q(n+1)$ holde.
- Konklusjonen er at $f(n) = g(n) = \frac{3^n - 1}{2}$ for alle n.

I læreboka presenteres et klassisk eksempel på bruk av induksjon:

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}.$$

Dette en formel man finner igjen i direkte eller beslektet form i mange viktige sammenhenger.

Eksempelvis er det antallet oppgjør i en enkel serie med n lag.

Vi skal se på noen andre, delvis beslektede eksempler.

Eksempel.

- La $f(n)$ være summen av de første n oddetallene, det vil si at
$$f(n) = \sum_{i=1}^{n} (2i - 1)$$
Da er $f(n) = n^2$.
- Vi skal gi et induksjonsbevis.

Eksempel (Fortsatt).

- For å vise starten på induksjonen regner vi ut
$$f(1) = \sum_{i=1}^{1} (2i - 1) = 1 = 1^2.$$
- Deretter må vi gjennomføre induksjonskrittet:
Anta at $f(n) = n^2$ for en n.
Da er $f(n+1) = f(n) + 2n + 1 = n^2 + 2n + 1 = (n+1)^2$.

- Ettersom vi nå har vist både induksjonstarten og induksjonskrittet, følger påstanden ved induksjon.

Eksempel.

- Hvis vi trekker en rett linje gjennom planet, deler vi planet i to.
- Hvis vi trekker en ny linje gjennom planet, deler disse to linjene planet i fire deler.
- Hvis vi prøver oss med tre linjer, greier vi ikke å dele planet i mer enn syv deler, og bruker vi fire linjer greier vi maksimalt å dele planet i 11 deler.
- Kan vi finne en formel for hvor mange felter vi maksimalt kan dele planet i ved hjelp av n linjer?

Eksempel (Fortsatt).

- La $F(n)$ være antall felter vi kan dele planet opp i ved å bruke n rette linjer.
- Da er $F(1) = 2$.
- Selv om vi ikke kjenner $F(n)$ kan vi uttrykke $F(n+1)$ ved hjelp av $F(n)$:
- La $l_1, \ldots, l_n, l_{n+1}$ være $n+1$ rette linjer slik at $l_1, \ldots, l_n$ deler planet opp i $F(n)$ forskjellige felter.
- Den siste linjen l_{n+1} skjærer hver av de andre linjene høyst en gang, så vi får maksimalt n nye skjæringspunkter.

Eksempel (Fortsatt).

- Skjæringspunktene deler l_{n+1} opp i høyst $n+1$ linjestykker, og hvert av disse linjestykkene deler et av de gamle feltene i to.
- Det betyr at vi får maksimalt $n+1$ nye felter.
- Da er $F(n+1) = F(n) + n + 1$
- Den neste jobben blir å finne en formel for $F(n)$ og så vise den ved induksjon.
- Denne typen formler finner man ofte gjennom prøving og feiling basert på erfaring, men det finnes også generelle metoder.

  Vi skal ikke legge så stor vekt på disse metodene i MAT1030.

Eksempel (Fortsatt).

- Vi påstår at $F(n) = 1 + \frac{n(n+1)}{2}$ og vil vise det ved induksjon:
- Induksjonen starter med $n = 1$:
  $1 + \frac{1(1+1)}{2} = 1 + 1 = 2 = F(1)$.
- La oss så gjennomføre induksjonskrittet:
- Anta at

$$F(n) = 1 + \frac{n(n+1)}{2}$$

Eksempel (Fortsatt).

- Da er

$$F(n+1) = F(n) + n + 1 = 1 + \frac{n(n+1)}{2} + (n+1)$$

$$= 1 + \frac{n(n+1) + 2(n+1)}{2} = 1 + \frac{(n+2)(n+1)}{2}.$$

- Skal vi være pedantiske kan vi skrive dette om til

$$1 + \frac{(n+1)((n+1)+1)}{2}.$$

- Induksjonskrittet er gjennomført, så påstanden er bevist.

Oppgave.

- Vi vet at vi kan dele planet opp i to felter ved hjelp av en sirkel.
- Vi vet at to sirkler kan skjære hverandre i to punkter
- Vi vet at $2n$ punkter vil dele en sirkel opp i $2n$ buestykker.
- Bruk dette til å definere funksjonen $G(n)$ ved rekursjon, hvor $G(n)$ er antall områder vi kan dele planet opp i ved hjelp av n sirkler.
- Foreslå en formel for $G(n)$ og se om du kan vise den ved induksjon.

Hvorfor forteller svaret på denne oppgaven oss at Venndiagrammer er uegnet til å studere Boolske kombinasjoner av mange mengder?

Eksempel.

- Enkelte regneoperasjoner tar lengere tid jo større input er.

- Det kan være av interesse å finne ut hvor mange "regneskritt" en oppgave krever, avhengig av hvor stort input er.
- Eksempelvis kan vi prøve å finne ut av hvor mange operasjoner som kreves for å utføre sorteringsalgoritmer i
  - Verste tilfelle
  - I gjennomsnitt

Eksempel (Fortsatt).

- Vanligvis greier man seg med omtrentlige verdier, men ved behov kan man bruke rekursjon og induksjon til å finne nøyaktige svar.
- Vi kan sortere elementene i en liste ved systematisk å bytte om på naboer som ligger i feil rekkefølge.
- La $S(n)$ være det maksimale antall slike bytter vi må foreta oss for å sortere en liste med n elementer.
- Vi ser at $S(1) = 0$
- Hvis listen kommer i fullstendig gal rekkefølge, må alle objektene i listen bytte plass med alle andre.

Eksempel (Fortsatt).

- Antall bytter som da trenges for å sortere $n+1$ objekter er $S(n+1) = n + S(n)$, ettersom vi kan risikere at vi må flytte siste objekt i listen til førsteplass (n bytter) og deretter sortere resten av listen ($S(n)$ bytter.)
- Vi ser ved induksjon at $S(n) = \frac{(n-1)n}{2}$.
- Beviset følger ved samme type utregning i induksjonsskrittet som for forrige eksempel, og vi tar det på tavlen (eller som øvelse for de som leser/repeterer denne teksten).

Merk.

- Det forrige eksemplet er ikke helt realistisk, enhver sorteringsalgoritme vil innebære at man foretar en rekke sammenlikninger og skifte av plasser.
- Hvis vi skal analysere hvor tidkrevende en algoritme kan være, må vi vite hvor mange regneskritt som kreves, og hvor lang tid hvert enkelt skritt tar.
- Induksjonsbevis kan inngå som en del av beviset for at en regneprosess kan utføres raskt, eventuelt for at den tar for lang tid.

Eksempel.

- Vi minner om definisjonen av binomialkoeffisienten

$$\binom{n}{k} = \frac{n!}{k!(n-k)!.}$$

- Formelen

$$\binom{n+1}{k} = \binom{n}{k-1} + \binom{n}{k}$$

kan bekreftes ved enkel regning.

Eksempel (Fortsatt).

- På skolen lærer man at

$$\binom{n}{k}$$

uttrykker på hvor mange måter man kan velge ut k objekter fra en mengde med n objekter på, når $k \leqslant n$.

- Det er ikke alltid så lett å få med seg begrunnelsen for dette.
- Et alternativ kan være å bruke induksjon.

Eksempel (Fortsatt).

- Vi starter med tilfellet $n = 1$.
- Da er $k = 1$, og det fins bare en måte å velge ut ett element fra en mengde på ett element. Binomialkoeffisienten er i dette tilfellet 1, så påstanden holder.
- Induksjonstarten er i boks.

Eksempel (Fortsatt).

- Anta så at formelen holder for n og at vi skal finne ut av på hvor mange måter vi kan plukke k elementer ut av en mengde $\{a_1, \ldots, a_{n+1}\}$ på.
- Hvis $k = n + 1$, fins det nøyaktig en måte, og

$$1 = \binom{n+1}{n+1}.$$

Eksempel (Fortsatt).

- Hvis $k < n+1$, ser vi på to tilfeller:
  1. a_{n+1} er med i den mengden vi plukker ut.
  2. a_{n+1} er ikke med i den mengden vi plukker ut.
- I det første tilfellet må vi plukke ut $k-1$ elementer fra

  $$\{a_1, \ldots, a_n\},$$

  og det kan vi gjøre på

  $$\binom{n}{k-1}$$

  måter.

Eksempel (Fortsatt).

- I det andre tilfellet må vi plukke ut k elementer fra

  $$\{a_1, \ldots, a_n\},$$

  og det kan vi gjøre på

  $$\binom{n}{k}$$

  måter.
- Summen er da

  $$\binom{n+1}{k}.$$

  som angir det totale antall måter vi kan plukke ut k elementer fra en mengde med n elementer på.

Eksempel (Fortsatt).

- Induksjonskrittet sier at hvis binomialkoeffisientene $\binom{n}{k}$ forteller oss, for alle $k \leqslant n$, hvor mange forskjellige delmengder med k elementer det fins av en mengde med n elementer, så vil koeffisientene $\binom{n+1}{k}$ fortelle oss det samme for mengder med $n+1$ elementer.
- Vi kan merke oss at for å vise induksjonskrittet for en k trenger vi induksjonsantagelsen både for k og for $k-1$.

- Både rekursjon og induksjon er mer generelle fenomener enn det vi har gitt inntrykk av her.
- Dette skal vi komme tilbake til, både ved å se på rekurrenslikninger og på rekursjon og induksjon over andre matematiske strukturer enn $\mathbb{N}$ eller $\mathbb{N}_0$.
- Først skal vi imidlertid se på en logikers forklaring på sammenhengen mellom induksjon og rekursjon:
- Anta at vi i utgangspunktet ikke har lært om induksjonsbevis.
- Anta videre at $P(k)$ er et predikat og at vi har bevis for
  1. Induksjonstarten $P(1)$
  2. Induksjonskrittet $P(k) \to P(k+1)$ hvor k er en variabel.
- Ved *rekursjon* kan vi da konstruere et bevis $B(n)$ for $P(n)$ for enhver n ved
  1. La $B(1)$ være beviset vi har for $P(1)$.
  2. La $B(n+1)$ være bygget opp av
     - $B(n)$(som er et bevis for $P(n)$),
     - beviset vi får for $P(n) \to P(n+1)$ ved å sette inn n for k i beviset for induksjonskrittet,
     - og bruk av den utsagnslogiske regelen om at fra A og $A \to B$ kan vi slutte B.
  3. Da vil $B(n+1)$ være et bevis for $P(n+1)$.
- Når vi vet at vi kan konstruere enkeltbevis for hver $P(n)$, kan vi rasjonalisere virksomheten vår og si at vi har et bevis for $\forall n P(n)$.

Rekurrens

- Det i nesten enhver sammenheng mest brukte eksemplet på rekurrens er definisjonen av Fibonacci-tallene:
  - $F(1) = 1$
  - $F(2) = 1$
  - $F(n+2) = F(n+1) + F(n)$ for alle $n \in \mathbb{N}$.
- Vi ser at denne tallfølgen også er fullstendig bestemt, selv om definisjonen ikke helt følger formatet til definisjoner ved rekursjon.
  1. $F(1) = 1$
  2. $F(2) = 1$
  3. $F(3) = F(2) + F(1) = 1 + 1 = 2$
  4. $F(4) = F(3) + F(2) = 2 + 1 = 3$
  5. $F(5) = F(4) + F(3) = 3 + 2 = 5$

  ...
- Vi definerer $F(n)$ direkte for de to minste verdiene av n og lar deretter verdien av $F(n)$ avhenge av verdien av F i de to foregående punktene.
- Da er det ingen grenser for hvor langt vi kan fortsette:
  $F(6) = 8$, $F(7) = 13$, $F(8) = 21$, $F(9) = 34$, $\cdots$.
- Spørsmålet er om vi kan finne en eksplisitt formel for $F(n)$, og helst om vi kan basere dette på en generell forståelse.

- Dette skal vi komme tilbake til.
  Vi skal se på et par andre eksempler først.

Eksempel.

- Likningen
$$F(n+2) = F(n+1) + 2F(n)$$
  bestemmer ikke tallfølgen $F(1)$, $F(2)$, $F(3)$, $\cdots$ fullstendig, men hver gang vi bestemmer oss for hva $F(1)$ og $F(2)$ skal være, blir følgen bestemt ved rekurrens.
- En slik likning kaller vi en rekurrenslikning.
- Ved direkte regning kan vi se at $F_1(n) = 2^n$ og $F_2(n) = (-1)^n$ begge tilfredstiller likningen:
- $2^{n+1} + 2 \cdot 2^n = 2^{n+1} + 2^{n+1} = 2^{n+2}$
- $(-1)^{n+1} + 2 \cdot (-1)^n = (-1)^n(-1+2) = (-1)^n(-1)^2 = (-1)^{n+2}$

Eksempel (Fortsatt).

- Hvis nå A og B er to reelle tall, ser vi at vi også har at
$$F_3(n) = A \cdot 2^n + B \cdot (-1)^n$$
  er en løsning.
- Er det nå noen grunn til å lete etter flere løsninger?
- Svaret er *NEI*, for vi vet at hvis vi bestemmer $F(1) = a$ og $F(2) = b$, har vi bestemt følgen fullstendig.
- Likningene $a = A \cdot 2^1 + B \cdot (-1)^1$ og $b = A \cdot 2^2 + B \cdot (-1)^2$ vil bestemme A og B, slik at løsningen i et konkret tilfelle er en av de vi har sett på.

Eksempel (Fortsatt).

Løsningene er

- $$A = \frac{a+b}{6}$$
- $$B = \frac{2b-a}{3}$$

- Det neste spørsmålet er da selvfølgelig hvordan vi fant på å prøve potenser av 2 og -1.
- Det skal vi komme tilbake til neste uke.
- Vi minner om at neste uke er den siste forelesningsuken før Påske.

MAT1030 – Forelesning 17

Rekurrenslikninger

Dag Normann - 16. mars 2010

Forelesning 17

Rekurrenslikninger

- Forrige gang ga vi en rekke eksempler på bruk av induksjonsbevis og rekursivt definerte funksjoner.
- I et av eksemplene definerte vi binomialkoeffisientene og viste en viktig egenskap til binomialkoeffisienter ved induksjon.
- Både definisjonen og den viste egenskapen er pensum.
- $$\binom{n}{k} = \frac{n!}{k!(n-k)!.}$$
- $$\binom{n+1}{k} = \binom{n}{k-1} + \binom{n}{k}$$
- $\binom{n}{k}$ forteller oss på hvor mange forskjellige delmengder med k elementer det er av en mengde med n elementer.
- Det siste kvarteret snakket vi om rekurrens.
- Dette vil være hovedtemaet i dag.
- Vi skal først se på noen eksempler på rekurrenslikninger som tilnærming til en generell metode for å løse slike likninger.

Eksempel.

- Vi skal lete etter løsninger av rekurrenslikningen

$$F(n+2) = 5F(n+1) - 6F(n)$$

- Vi viser først ved regning at $F_1(n) = 2^n$ og $F_2(n) = 3^n$ er løsninger:
- $5 \cdot 2^{n+1} - 6 \cdot 2^n = 2^n(5 \cdot 2 - 6) = 2^n \cdot 2^2 = 2^{n+2}$
- $5 \cdot 3^{n+1} - 6 \cdot 3^n = 3^n(5 \cdot 3 - 6) = 3^n \cdot 3^2 = 3^{n+2}$
- Det som gjør at denne utregningen fører frem er at 2 og 3 er løsninger av likningen

$$x^2 = 5x - 6$$

  og det er de eneste løsningene.
- Det betyr at man må kunne løse 2. gradslikninger for å kunne løse slike rekurrenslikninger.

I stedet for å skrive at

$$F(n+2) = F(n+1) + 2F(n)$$

skal gjelde for alle $n \in \mathbb{N}$

kan vi skrive at

$$F(n) - F(n-1) - 2F(n-2) = 0$$

for alle $n \geqslant 3$,

og i stedet for å skrive at

$$F(n+2) = 5F(n+1) - 6F(n)$$

for alle $n \in \mathbb{N}$

kan vi skrive at

$$F(n) - 5F(n-1) + 6F(n-2) = 0$$

for alle $n \geqslant 3$.

Dette er strengt tatt uendelig mange likninger i uendelig mange variable $F(n)$, men det gir mening å snakke om løsningsmengden til dette likningsettet.

Vi vil nå etablere den terminologien vi skal bruke i fortsettelsen, og vise hvordan vi kan finne løsningsmengden til slike *rekurrenslikninger*.

Definisjon.

- En 2. ordens lineær homogen rekurrenslikning er en *funksjonslikning* på formen

$$at(n) + bt(n-1) + ct(n-2) = 0.$$

- En *følge* $F : \mathbb{N} \to \mathbb{R}$ er en løsning av rekurrenslikningen hvis $aF(n)+bF(n-1)+cF(n-2) = 0$ for alle $n \geqslant 3$.
- Hvis verdiene for $t(1)$ og/eller $t(2)$ er angitt i tillegg, kalles dette initialbetingelser.
- En løsning må da tilfredstille disse betingelsene.

Eksempel.

- Fibonacci-følgen er bestemt som den eneste løsningen $F : \mathbb{N} \to \mathbb{N}$ av

$$t(n) - t(n-1) - t(n-2) = 0$$

  med initialbetingelser $t(1) = t(2) = 1$
- Vi skal finne en formel for $F(n)$ når vi har utviklet metoden for det.

Merk.

- Vi kaller rekurrenslikningen lineær fordi likningens venstre side er en lineær kombinasjon av $t(n)$, $t(n-1)$ og $t(n-2)$.
  Eksempelvis vil ikke
  $$t(n) - (t(n-1))^2 - t(n-2) = 0$$
  være lineær fordi vi har et ledd av grad 2.

Merk (fortsatt).

- Vi kaller likningen homogen fordi vi ikke har noe ledd som bare avhenger av n.
  Eksempelvis er ikke
  $$t(n) - t(n-1) + 0 \cdot t(n-2) + n = 0$$
  homogen fordi vi har et ledd n.
- Likningen er 2. ordens fordi verdien av $t(n)$ avhenger av to verdier $t(n-1)$ og $t(n-2)$.
  Likningen $t(n) - 2t(n-1) = 0$ er 1. ordens, mens $t(n) - t(n-1) + t(n-2) - t(n-3) = 0$ er 3. ordens.

Merk (Fortsatt).

- Alt vi sier vil kunne gjøres gjeldende for 1. ordens og 3. ordens homogene, lineære likninger også, men vi skal konsentrere oss om de 2. ordens likningene.
- For enkelthets skyld vil vi bruke betegnelsen rekurrenslikning i betydningen 2. ordens lineær homogen rekurrenslikning, ettersom vi vil begrense oss til denne typen rekurrenslikninger.

- Da vi analyserte mulige løsninger av likningen
  $$F(n+2) = 5F(n+1) - 6F(n)$$
  kommenterte vi at vi utnyttet at 3 og 2 er løsninger i likningen
  $$x^2 = 5x - 6$$
  da vi viste at $F_1(n) = 3^n$ og $F_2(n) = 2^n$ er løsninger av rekurrenslikningen.
- Denne innsikten skal vi utvikle til en fullstendig innsikt i hvilke løsninger vi har:

Definisjon.
Hvis
$$at(n) + bt(n-1) + ct(n-2) = 0$$

er en rekurrenslikning, kalles

$$ax^2 + bx + c = 0$$

for den karakteristiske likningen til rekurrenslikningen.

Teorem.

La

$$at(n) + bt(n-1) + ct(n-2) = 0$$

være en rekurrenslikning og la $r \in \mathbb{R}$.

Da er $F_r(n) = r^n$ en løsning av rekurrenslikningen hvis og bare hvis r er en løsning av den *karakteristiske* likningen.

Bevis.

Anta at r er en løsning av den karakteristiske likningen.

Vi setter F_r inn i likningene, og får

$$ar^n + br^{n-1} + cr^{n-2} = r^{n-2}(ar^2 + br + c) = 0,$$

det siste fordi r er løsning av den karakteristiske likningen.

Anta så at $F_r(n) = r^n$ løser rekurrenslikningen.

Setter vi inn for $n = 3$ får vi spesielt

$$ar^3 + br^2 + cr^1 = 0$$

som akkurat sier at $ar^2 + br + c = 0$ eller $r = 0$ Det siste er umulig hvis $c \neq 0$, noe som er tilfelle med en 2. ordens likning.

Dette resultatet forteller oss at vi ofte kan finne to løsninger av en rekurrenslikning.

Hvis løsningene av den karakteristiske likningen inneholder kvadratrot-tegn, vil de eksakte formlene for løsningene gjøre det samme.

Hvis løsningene av den karakteristiske likningen er komplekse tall, vil vi trenge komplekse tall for å beskrive løsningene av rekurrenslikningen.

Vi har imidlertid fortsatt bare to løsninger. Hvordan finner vi flere?

Setningen på neste side er et godt hjelpemiddel.

Teorem.

La

$$at(n) + bt(n-1) + ct(n-2) = 0$$

være en rekurrenslikning ,
og la $F(n)$ og $G(n)$ være to løsninger.
La A og B være reelle tall.
Da er $H(n) = A \cdot F(n) + B \cdot G(n)$ også en løsning.

Bevis.

- Det er bare å sette H inn i rekurrenslikningen og sjekke:

$$\begin{aligned}
&a \cdot H(n) + b \cdot H(n-1) + c \cdot H(n-2) \\
&= a(A \cdot F(n) + B \cdot G(n)) + b(A \cdot F(n-1) + B \cdot G(n-1)) \\
&\quad + c(A \cdot F(n-2) + B \cdot G(n-2)) \\
&= A \cdot (a \cdot F(n) + b \cdot F(n-1) + c \cdot F(n-2)) \\
&\quad + B \cdot (a \cdot G(n) + b \cdot G(n-1) + c \cdot G(n-2)) \\
&= A \cdot 0 + B \cdot 0 = 0.
\end{aligned}$$

Eksempel.

- La oss gå tilbake til likningen for Fibonacci-tallene:

$$t(n) - t(n-1) - t(n-2) = 0.$$

- Den karakteristiske likningen er

$$x^2 - x - 1 = 0$$

og ved abc-formelen har vi løsninger

$$r = \frac{1 \pm \sqrt{5}}{2}.$$

Eksempel (Fortsatt).

Det betyr at vi bør lete etter en løsning på formen

$$F(n) = A(\frac{1+\sqrt{5}}{2})^n + B(\frac{1-\sqrt{5}}{2})^n.$$

Eksempel (Fortsatt).

Vi vet at hvis vi finner A og B slik at initialbetingelsene $F(1) = F(2) = 1$ holder, så må vi ha funnet den eneste løsningen som fins.

Det gir oss to *stygge* lineære likninger i de ukjente A og B

$$A\frac{1+\sqrt{5}}{2} + B\frac{1-\sqrt{5}}{2} = 1$$

$$A(\frac{1+\sqrt{5}}{2})^2 + B(\frac{1-\sqrt{5}}{2})^2 = 1$$

Den som vil løse disse likningene selv, bør lukke øynene på neste side, hvor vi gir løsningene uten mellomregning.

Eksempel (Fortsatt).

Løsningene er

$$A = \frac{\sqrt{5}}{5}$$

og

$$B = -\frac{\sqrt{5}}{5}$$

så formelen for n'te Fibonaccitall $F(n)$ er, utrolig nok,

$$\frac{\sqrt{5}}{5}((\frac{1+\sqrt{5}}{2})^n - (\frac{1-\sqrt{5}}{2})^n)$$

For hver verdi av n er altså dette et naturlig tall.

- Hvis den karakteristiske likningen har to forskjellige løsninger r og s, vil alle løsningene av rekurrenslikningen være på formen

$$F(n) = A \cdot r^n + B \cdot s^n.$$

- Det er fordi hvis vi i tillegg bestemmer verdiene på $F(1)$ og $F(2)$, vil vi kunne løse likningsettet

$$A \cdot r + B \cdot s = F(1)$$

$$A \cdot r^2 + B \cdot s^2 = F(2).$$

- Dermed finner vi en løsning på formen $F(n) = Ar^n + Bs^n$ som oppfyller initialbetingelsene.
- Vi skal regne gjennom et typisk eksempel.

Eksempel.

- Anta at vi skal løse følgende oppgave:
  a) Finn alle løsningene av rekurrenslikningen
  $$t(n) - 2t(n-1) - 3t(n-2) = 0.$$
  b) Finn løsningen fra a) som tilfredstiller initialbetingelsene $F(1) = 1$ og $F(2) = 2$.
- Det første vi må gjøre er å finne den karakteristiske likningen
  $$x^2 - 2x - 3 = 0$$
  og løse den:

Eksempel (Fortsatt).

- $$x = \frac{2 \pm \sqrt{(-2)^2 - 4 \cdot 1 \cdot (-3)}}{2} = \begin{cases} 3 \\ -1 \end{cases}$$
- Siden den karakteristiske likningen har to løsninger, vet vi at den generelle løsningen av rekurrenslikningen er $F(n) = A \cdot 3^n + B \cdot (-1)^n$.
- Dette løser a).

Eksempel (Fortsatt).

- For å løse b), må vi bestemme A og B slik at initialbetingelsene holder.
- Det betyr at vi må løse likningene
  - $3A - B = 1$ ($n = 1$, $F(1) = 1$ var en betingelse.)
  - $9A + B = 2$ ($n = 2$, $F(2) = 2$ var en betingelse.)

Eksempel (fortsatt).

- Legger vi sammen likningene, får vi $12A = 3$, så $A = \frac{1}{4}$ er en løsning.
- Setter vi denne løsningen inn i en av likningene og løser med hensyn på B, får vi $B = -\frac{1}{4}$.

- Løsningen på oppgave b) er derfor

$$F(n) = \frac{1}{4}(3^n - (-1)^n).$$

Merk.

- Det er ikke anledning til å ha med seg hjelpemidler til eksamen i MAT1030.
- Spesielt er det ikke anledning til å ha med lommeregner.
- Det betyr at de som har basert seg på å bruke lommeregner for å løse annengradslikninger må begynne å øve seg på å løse dem for hånd.

- I det forrige eksemplet så vi hvordan vi kan løse enhver rekurrenslikning hvor den karakteristiske likningen har to forskjellige løsninger.
- Hva gjør vi hvis det bare fins en løsning?
- Fortsatt vil det være slik at hvis vi kan finne to "uavhengige" følger som løsninger, vil alle andre løsninger være kombinasjoner av disse.
- Hvis r er løsningen på den karakteristiske likningen, er fortsatt $F(n) = r^n$ en løsning av rekurrenslikningen.
- Målet vårt må derfor være å finne en annen løsning i tillegg.

Eksempel.

- Betrakt rekurrenslikningen

$$t(n) - 4t(n-1) + 4t(n-2) = 0.$$

- Den karakteristiske likningen er

$$x^2 - 4x + 4 = 0.$$

- $r = 2$ er den eneste løsningen av den karakteristiske likningen.
- Da er $F(n) = 2^n$ en løsning av rekurrenslikningen.
- La oss gi to initialbetingelser for å se om det fins noe mønster som antyder hvordan en annen løsning kan se ut:

Eksempel (Fortsatt).

- $G(1) = 1$ og $G(2) = 0$:
  Regning gir oss at
  $G(3) = -4$, $G(4) = -16$, $G(5) = -48$ og $G(6) = -128$

Hvis vi nå prøver å sette faktoren 2^n utenfor $G(n)$, får vi at

$G(1) = 2^{-1} \cdot 2^1$, $G(2) = 0 \cdot 2^2$, $G(3) = -2^{-1} \cdot 2^3$, $G(4) = -2 \cdot 2^{-1} \cdot 2^4$, $G(5) = -3 \cdot 2^{-1} \cdot 2^5$ og $G(6) = -4 \cdot 2^{-1} \cdot 2^6$.

Det vil være naturlig å gjette på at $G(n) = (2-n) \cdot 2^{-1} \cdot 2^n$ er løsningen av rekurrenslikningen.

Vi skal se at det er tilfelle.

Eksempel (Fortsatt).

- Hvis vi har gjettet riktig, vil $H(n) = n \cdot 2^n$ også være en løsning av rekurrenslikningen, siden H kan skrives som en kombinasjon av vårt forslag til $G(n)$ og av $F(n)$
- Omvendt, hvis $H(n)$ er en løsning av rekurrenslikningen, er $G(n)$ en kombinasjon av $F(n)$ og $H(n)$, så da er $G(n)$ en løsning av rekurrenslikningen (og den som oppfyller initialbetingelsene).
- I så fall er svaret så pent at det er verd et forsøk med et generelt bevis.

Teorem.

La

$$at(n) + bt(n-1) + ct(n-2) = 0$$

være en rekurrenslikning hvor den karakteristiske likningen bare har en løsning r.

Da er $H(n) = nr^n$ en løsning av rekurrenslikningen.

Bevis.

Løsningen av den karakteristiske likningen er

$$r = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}.$$

Siden det bare er en løsning er $b^2 = 4ac$, fordi det som står under rottegnet må være null.

Vi setter $H(n)$ inn i likningen, og får

1. $anr^n + b(n-1)r^{n-1} + c(n-2)r^{n-2}$
2. $= r^{n-2}(anr^2 + bnr - br + cn - 2c)$
3. $= r^{n-2}(n(ar^2 + br + c) - br - 2c)$
4. $= -r^{n-2}(br + 2c)$.

Bevis (Forklaring).

- I linje 1 setter vi $H(n)$ inn i likningen.
- I linje 2 setter vi den felles faktoren r^{n-2} utenfor samtidig som vi løser opp parentesene $(n-1)$ og $(n-2)$.
- I linje 3 setter vi n utenfor de leddene hvor n er faktor.
- I linje 4 utnytter vi at r er en løsning av den karakteristiske likningen, så $ar^2 + br + c = 0$.
- For å vise at $H(n)$ er en løsning av rekurrenslikningen, er det altså nok å vise at $br + 2c = 0$
- Her skal vi bruke at r er den eneste løsningen.

Bevis (Fortsatt).

- Siden vi har en 2. ordens likning, vil $a \neq 0$ og $c \neq 0$.
- Siden vi bare har en løsning, er
$$r = \frac{-b}{2a}.$$
- Setter vi dette inn i $br + 2c$ får vi
$$b \cdot \frac{-b}{2a} + 2c.$$

Bevis (Fortsatt).

- Satt på felles brøkstrek er dette
$$\frac{-b^2 + 4ac}{2a}$$
som er lik 0 fordi $b^2 - 4ac = 0$.
- Det var det som gjensto å bevise, så teoremet er vist.

Merk.

Vi har nå funnet to løsninger av rekurrenslikningen både i det tilfellet hvor den karakteristiske likningen har to løsninger, og i det tilfellet hvor den bare har en løsning.

Siden vi kan finne en lineær kombinasjon av slike løsninger for enhver initialbetingelse $t(1) = a$ og $t(2) = b$, har vi i realiteten funnet en generell løsning.

Hvis r og s er to forskjellige løsninger av den karakteristiske likningen, er

$$F(n) = Ar^n + Bs^n$$

den generelle løsningen av rekurrenslikningen.

Merk (Fortsatt).

Hvis r er den eneste løsningen av den karakteristiske likningen er

$$F(n) = (A + Bn)r^n$$

den generelle løsningen av rekurrenslikningen.

Eksempel.

- Anta at vi har fått følgende oppgave:
- La $t(n) - t(n-1) + \frac{1}{4}t(n-2) = 0$ være en rekurrenslikning.

a) Finn den generelle løsningen $F(n)$ til likningen.

b) Finn løsningen $G(n)$ som oppfyller at $G(1) = G(2) = 1$.

c) Finn løsningen $H(n)$ som oppfyller at $H(1) = 1$ og $H(2) = 0$.

Eksempel (Fortsatt).

Først må vi finne den karakteristiske likningen

$x^2 - x + \frac{1}{4} = 0$.

Vi finner at $r = \frac{1}{2}$ er den eneste løsningen.

Da er den generelle løsningen

$$F(n) = (A + nB)(\frac{1}{2})^n.$$

Dette løser a)

Eksempel (Fortsatt).

For å løse punkt b), bruker vi de to initialbetingelsene til å finne to likninger med A og B som ukjente

- $n = 1$: $(A + B)\frac{1}{2} = 1$, det vil si $A + B = 2$.
- $n = 2$: $(A + 2B)(\frac{1}{2})^2 = 1$, det vil si $A + 2B = 4$.

Disse har løsninger $A = 0$ og $B = 2$, så svaret på b) er

$$G(n) = 2n(\frac{1}{2})^n.$$

Eksempel (Fortsatt).

For å løse c) setter vi 1 og 0 i høyresidene i likningene over, og får

- $A + B = 2$
- $A + 2B = 4$

som har $A = 4$ og $B = -2$ som løsninger.

Løsningen på c) er derfor

$$H(n) = (4 - 2n)(\frac{1}{2})^n.$$

Spiller så rekurrenslikninger noen rolle i informatikken?

Hvis man skal analysere kompleksiteten av en algoritme, det vil si finne ut av hvor mange regneskritt som trenges som funksjon av størrelsen på input, risikerer man at resultatet blir en inhomogen rekurrenslikning. (Dette er observert på vitenskapelig foredrag om temaet).

Da vil regnetiden vokse eksponensielt med størrelsen på input.

Hvis grunntallet i denne eksponenten ikke er så mye større enn 1, er ikke det nødvendigvis ødeleggende for nytten av algoritmen.

Det er stor forskjell på en algoritme hvor regnetiden er begrenset av $1000(1,08)^n$ og en der regnetiden er begrenset av $2(1,23)^n$, hvor n er antall bit i input. n trenger ikke å være så veldig stor før den første trolig er mer effektiv enn den andre.

MAT1030 – Forelesning 18

Generell rekursjon og induksjon

Dag Normann - 17. mars 2010

Forelesning 18

Rekurrenslikninger

- Forrige uke så vi på rekurrenslikninger.
- En rekurrenslikning er en funksjonslikning på formen

$$at(n) + bt(n-1) + ct(n-2) = 0$$

  hvor en løsning er en funksjon
  $F : \mathbb{N} \to \mathbb{R}$ slik at

$$aF(n) + bF(n-1) + cF(n-2) = 0$$

  for alle $n \geqslant 3$ (eller for alle n hvor n, $n-1$ og $n-2$ er i definisjonsområdet til F, når vi vil bruke maskineriet vårt i en mer generell situasjon).
- Den karakteristiske likningen er da

$$ax^2 + bx + c = 0$$

- Hvis r og s er to forskjellige løsninger av den karakteristiske likningen, er den generelle løsningen av rekurrenslikningen

$$F(n) = Ar^n + Bs^n$$

  hvor A og B er vilkårlige reelle tall.
- Hvis den karakteristiske likningen bare har en løsning r, er den generelle løsningen av rekurrenslikningen

$$(A + Bn)r^n.$$

- Hvis vi i tillegg har krav om at $t(1) = a$ og $t(2) = b$, kan vi bestemme A og B i den generelle løsningen ved å sette inn for $n = 1$ og $n = 2$ i den generelle løsningen, og løse likningene mhp A og B.
- *Dette vil alltid fungere*.
- Boka ser bare på tilfellet med initialbetingelser på $t(1)$ og $t(2)$.
- Hadde vi satt krav til $t(17)$ og $t(256)$, eller til t-verdien for to andre, forskjellige tall, kunne vi fortsatt bestemt A og B fra den informasjonen.
- Initialbetingelser for $n = 0$ og $n = 1$ kan gi den enkleste regningen.
- Dette skal vi se nærmere på om en stund.

- Hvis vi har en rekurrenslikning

$$at(n) + bt(n-1) + ct(n-2) = 0$$

  hvor $a = 0$ eller $c = 0$, er likningen strengt tatt ikke av 2. orden, og vi må være litt forsiktige.
- Dette er nøyaktig den situasjonen hvor vi kan ha at 0 er en rot i den karakteristiske likningen.
- Da vil den generelle løsningen være på formen

$$F(n) = Ar^n$$

  hvor $r \neq 0$ er en rot i den karakteristiske likningen.
- Egentlig kan vi her betrakte den karakteristiske likningen som en førstegradslikning med r som den eneste løsningen.

Eksempel.

- En litt håpløs måte å sende en kryptert binær sekvens på vil være å sende 10 eller 01 valgt vilkårlig der det skulle stått 1 og 0 der det skulle stått 0.
- Det er da opp til mottageren, som er den eneste som kjenner krypteringsmåten, å liste opp alle mulige opprinnelige meldinger og finne den som gir mening.
- Hva er det maksimale antall $F(n)$ opprinnelige meldinger som kan ligge bak en mottatt bitsekvens av lengde n?

Eksempel (Fortsatt).

- Hvis $n = 1$ har vi bare en mulighet, den sendte biten er 0
- Hvis $n = 2$ har vi to muligheter, bitsekvensen representerer 1 eller bitsekvensen representerer 00.
- For $n \geqslant 3$ har vi to muligheter:
  - Siste siffer er 0 og representerer en 0. Det totale antall muligheter i den situasjonen er $F(n-1)$ ettersom resten av meldingen også skal representere en bitsekvens.
  - De siste to sifrene representerer 1. Dette svarer egentlig til $F(n-2)$ muligheter totalt.

Eksempel (Fortsatt).

- Svaret på problemet får vi ved å løse rekurrenslikningen

$$t(n) = t(n-1) + t(n-2)$$

  med initialbetingelser $t(1) = 1$ og $t(2) = 2$
- Løsningen er da at $F(n)$ er Fibonaccitall nr. $n + 1$, noe som viser at metoden er svært upraktisk.

- Vi har vært lojale mot læreboka og latt løsninger av rekurrenslikninger være følger, eller funksjoner definert på $\mathbb{N}$.
- Det er imidlertid ikke noe i veien for at vi ser på løsninger definert på hele $\mathbb{J}$, eller fra 0 og oppover.
- Vi kan tolke likningen
$$t(n) - t(n-1) - t(n-2) = 0$$
for Fibonacci-følgen som en likning der n varierer over hele $\mathbb{J}$.
- Det ville eksempelvis gitt oss at vi kan finne $F(0)$ fra
$$F(2) - F(1) - F(0) = 0,$$
hvilket gir $F(0) = 0$.
- Vi kan godt fortsette nedover med $F(1) - F(0) - F(-1) = 0$ og finne at $F(-1) = 1$.
- Den praktiske nytten vil være at det ofte er enklere å bestemme løsningen til en rekurrenslikning med initialverdier fra $F(0)$ og $F(1)$ fordi de lineære likningene vil bli penere.
- Hvis s og r er løsninger av den karakteristiske likningen, kan vi finne A og B fra
  - $A + B = F(0)$
  - $Ar + Bs = F(1)$
- Har vi bare en løsning r, er forenklingen ved å gå til $F(0)$ enda større.
  - $A = F(0)$
  - $(A + B)r = F(1)$
- Bruker man rekurrenslikningen til å regne ut $F(0)$ er dette en *lovlig måte* å løse oppgaver på.

Eksempel.
- Vi har gitt rekurrenslikningen
$$t(n) - t(n-1) - 2t(n-2) = 0$$
og skal finne løsningen som tilfredstiller initialbetingelse $F(1) = 3$ og $F(2) = 5$.
- Den karakteristiske likningen er
$$x^2 - x - 2 = 0$$
som har løsninger $r = 2$ og $s = -1$.

Eksempel (fortsatt).
- Den generelle løsningen er derfor
$$F(n) = A \cdot 2^n + B \cdot (-1)^n.$$
- Vi ser at $F(0) = 1$ er en alternativ initialbetingelse ved å se på $F(2) - F(1) - 2F(0) = 0$.

- Da løser vi likningene $A + B = 1$ og $2A - B = 3$ og får $A = \frac{4}{3}$ og $B = -\frac{1}{3}$.
- Det kan være en smaksak hva som er den enkleste metoden i hvert enkelt tilfelle.
- Hvis røttene til den karakteristiske likningen er kompliserte uttrykk med kvadratrøtter, er det normalt enklere å ta utgangspunkt i $F(0)$ og $F(1)$ for å bestemme A og B.

Rekursjon og programmering

- Vi startet innføringen av rekursjon med å gi eksempler på hvordan vi kunne finne pseudokoder som svarer til rekursive konstruksjoner.
- Vi kan minne om at hvis
  - $f(1) = a$
  - $f(n+1) = g(f(n), n)$

  er en rekursiv funksjon, og vi har en pseudokode for g, kan vi erstatte denne pseuokoden (som en del av en større kode) med

  $$x \leftarrow g(i, j)$$

  i betydningen at variabelen x får verdien til f når inputvariablene får verdiene til i og j.
- Det er flere måter vi kan lage en pseudokode for g på, vi skal se på to av dem:

Eksempel.

1 *Input* n $[n \in \mathbb{N}]$
2 $x \leftarrow a$
3 *Output* x
4 **For** $i = 1$ **to** $n - 1$ **do**
  4.1 $x \leftarrow g(x, i)$
  4.2 *Output* x

Merk.

Denne pseudokoden vil skrive ut $f(1), f(2), \ldots, f(n)$ i rekkefølge.
Hvis vi bare vil ha ut $f(n)$ blir koden enda enklere:

Eksempel.

1 *Input* n $[n \in \mathbb{N}]$

2 $x \leftarrow a$

3 **For** $i = 1$ **to** $n - 1$ **do**

3.1 $x \leftarrow g(x, i)$

4 *Output* x

- Læreboka har et lite avsnitt om plassen til rekursjon i enkelte programmeringsspråk.
- Dette er lesestoff, som ikke blir utdypet på forelesningene.
- Enkelte programmeringsspråk tillater til og med en sterkere form for rekursjon, selvkallende prosedyrer.
- Rekursjon er et spesialtilfelle av selvkallende prosedyrer, hvor vi definerer en funksjon $f(x)$ ved å bruke verdier $f(y)$ for enkelte y avhengige av x.
- En slik definisjon kan lett lede til løkkeberegninger eller uendelige beregninger, uten at det er lett å se hvorfor det er tilfelle.
- Vi skal ikke komme nærmere inn på det etter følgende eksempel som vi har sett før i flere varianter.

Eksempel.

- Da vi ga eksempler på pseudokoder, ga vi et eksempel på en algoritme som ingen ennå vet om vil terminere for alle verdier på input, og vi ga algoritmen i form av en pseudokode.
- Våre pseudokoder fanger ikke opp muligheten for selvkallende prosedyrer, men hadde vi hatt den muligheten, kunne vi betraktet følgende som en meningsfylt algoritme.

Eksempel (Fortsatt).

- La $f(1) = 1$.
- La $f(n) = f(\frac{n}{2}) + 1$ hvis n er et partall.
- La $f(n) = f(3n + 1) + 1$ hvis $n > 1$ er et oddetall.
- Vi kan da eksempelvis regne ut

$$f(20) = f(10) + 1 = f(5) + 2 = f(16) + 3$$

$$= f(8) + 4 = f(4) + 5 = f(2) + 6 = f(1) + 7 = 8$$

- f er veldefinert som en *partiell* funksjon som vi ikke vet om er *total*.

Oppgave.

- Det er ikke helt sant at vi ikke kan bruke pseudokoder til å lage algoritmer som svarer til selvkallende prosedyrer.
- Siden dette ikke er en del av MAT1030-pensum skal vi ikke legge stor vekt på det.
- Den som har lyst, kan imidlertid prøve å lage en pseudokode som beregner funksjonen definert ved
  - $f(n) = n^2$ hvis n kan deles på 3.
  - $f(n) = f(5n + 1)$ hvis n ikke kan deles på 3.
- Vil algoritmen gi et svar uansett hva input er?

Skal du løse denne oppgaven må du sannsynligvis bevise noe ved induksjon, og du må selv kunne formulere det du skal bevise.

Generell induksjon og rekursjon

- Vi har argumentert for at konstruksjoner ved rekursjon virker og at induksjon er en gyldig bevisteknikk.
- Vi har begrunnet dette med at vi kan nå alle naturlige tall ved å starte med 1 og så legge til 1 så mange ganger vi trenger.
- En måte å forklare hvorfor induksjonsbevis er matematisk holdbare argumenter på er følgende:
- Vi har at $\mathbb{N}$ er den minste mengden som oppfyller
  - $1 \in \mathbb{N}$
  - Hvis $x \in \mathbb{N}$ vil $x + 1 \in \mathbb{N}$
- Hvis vi da viser en egenskap P ved induksjon, viser vi at
  - $1 \in \{y \,:\, P(y)\}$
  - Hvis $x \in \{y \,:\, P(y)\}$ vil $x + 1 \in \{y \,:\, P(y)\}$.
- Siden $\mathbb{N}$ var den minste mengden med denne egenskapen, må $\mathbb{N} \subseteq \{y \,:\, P(y)\}$,
  eller, som vi sier, $P(x)$ holder for alle $x \in \mathbb{N}$.
- I logikk og informatikk (og også innen andre deler av matematikken og i andre fag) ser man definisjoner som likner på vår beskrivelse av $\mathbb{N}$; man beskriver en strukturert mengde ved å si hva som kommer inn som start og hvordan man finner mer komplekse elementer av mengden.
- Innen informatikk og logikk beskriver vi gjerne formelle språk på den måten.
- Det er viktig for de som skal studere f.eks. informatikk videre at man har en god forståelse av induksjon og rekursjon og at man har kjennskap til rekursjon over andre strukturer enn $\mathbb{N}$.
- I de følgende eksemplene skal vi anta at vi har et alfabet som består av alle de symbolene vi kan finne på tastaturet til en standard datamaskin (norsk standard om presisjonen er nødvendig), hvor tomrom er å betrakte som et eget symbol. Vi bruker bokstaver i *kursiv* som variable over bokstavene på tastaturet, og vi kan bruke andre bokstaver i kursiv som variable for ord, hvor:

- Et ord er en ordnet sekvens av bokstaver, hvor bokstavene er skrevet uten ekstra tegn i mellom.
- Det er vanlig å la e betegne det tomme ordet.
- Vi føyer en bokstav til et ord ved ganske enkelt å skrive det inntil ordet på høyre side.
- Dette kan vi bruke til å gi en induktiv beskrivelse av mengden av ord.

Definisjon.

Mengden av ord er den minste mengden som oppfyller

- Det tomme ordet e er et ord.
- Hvis w er et ord og $\mathfrak{b}$ er en bokstav, er $w\mathfrak{b}$ et ord.

Merk.

- Dette eksemplet virker en smule kunstig, fordi vi ikke oppfatter ord slik, selv om de fleste av oss starter med tom linje, og så skriver en og en bokstav fra venstre mot høyre.
- Det spiller imidlertid en rolle for hvordan ord representeres som data om de sees på som ordnede sekvenser med en gitt lengde eller som bygget opp ved at nye bokstaver legges til.
- Ord, slik vi har definert dem, er et spesialtilfelle av lister.
- En liste oppfattes som oftes som at enten er den den tomme listen e, eller så er den et ordnet par av siste element og resten av listen.
- Mange tenker seg at ytterste element (*hodet*) i en liste står til venstre for resten (*halen*) av listen.
- Programmeringspråket LISP er basert på listerekursjon.

Eksempel.

Som et eksempel på en rekursivt definisjon av en funksjon på mengden av ord kan vi betrakte PL (Push Left) som virker på et ord w og en bokstav $\mathfrak{a}$ ved:

- $\mathrm{PL}(e, \mathfrak{a}) = \mathfrak{a}$
- $\mathrm{PL}(w\mathfrak{b}, \mathfrak{a}) = \mathrm{PL}(w, \mathfrak{a})\mathfrak{b}$

Det som her skjer er at PL tar for seg et ord w og en bokstav $\mathfrak{a}$, og skriver bokstaven *foran* ordet, slik at vi får $\mathfrak{a}w$.

Egentlig burde vi vist denne egenskapen ved PL ved induksjon, men vi avstår i denne omgangen.

Eksempel (Fortsatt).

Følgende regne-eksempel viser hvordan PL virker:

$\mathtt{PL}(\mathtt{aba}, \mathtt{c}) =$

$\mathtt{PL}(\mathtt{ab}, \mathtt{c})\mathtt{a} =$

$\mathtt{PL}(\mathtt{a}, \mathtt{c})\mathtt{ba} =$

$\mathtt{PL}(e, \mathtt{c})\mathtt{aba} =$

caba

Eksempel.

Med utgangspunkt i eksemplet PL fra forrige side skal vi definere en speilingsfunksjon R ved rekursjon. R vil ta et ord som input og output vil være ordet skrevet baklengs:

- Vi definerer R ved:
  - $\mathtt{R}(e) = e$
  - $\mathtt{R}(w\mathtt{b}) = \mathtt{PL}(\mathtt{R}(w), \mathtt{b})$

Igjen overlater vi bekreftelsen av at funksjonen virker i henhold til spesifikasjonen til den enkelte.

Vi skal se på et eksempel, og i tillegg gi en oppgave, som skal hjelpe til med dette.

Eksempel.

Vi skal vise ved et regneeksempel i full detalj at $\mathtt{R}(\mathtt{abc}) = \mathtt{cba}$ slik R og PL er definerte.

Vi vil bare bruke symbolet e når vi ellers måtte skrevet *ingenting*, så $e$ab er det samme som ab.

$\mathtt{R}(\mathtt{abc}) =$

$\mathtt{PL}(\mathtt{R}(\mathtt{ab}), \mathtt{c}) =$

$\mathtt{PL}(\mathtt{PL}(\mathtt{R}(\mathtt{a}), \mathtt{b}), \mathtt{c}) =$

$\mathtt{PL}(\mathtt{PL}(\mathtt{PL}(\mathtt{R}(e), \mathtt{a}), \mathtt{b}), \mathtt{c}) =$

$\mathtt{PL}(\mathtt{PL}(\mathtt{PL}(e, \mathtt{a}), \mathtt{b}), \mathtt{c}) =$

$\mathtt{PL}(\mathtt{PL}(\mathtt{a}, \mathtt{b}), \mathtt{c}) =$

$\mathtt{PL}(\mathtt{PL}(e, \mathtt{b})\mathtt{a}, \mathtt{c}) =$

$\mathtt{PL}(\mathtt{ba}, \mathtt{c}) =$

$\mathtt{PL}(\mathtt{b}, \mathtt{c})\mathtt{a} =$

$\mathtt{PL}(e, \mathtt{c})\mathtt{ba} =$

cba.

Oppgave.

a) Ved å bruke den rekursive definisjonen av PL, vis hvordan vi skritt for skritt kan finne verdiene av
   - $\mathrm{PL}(e, \mathrm{d})$
   - $\mathrm{PL}(\mathrm{a}, \mathrm{d})$
   - $\mathrm{PL}(\mathrm{ab}, \mathrm{d})$
   - $\mathrm{PL}(\mathrm{aba}, \mathrm{d})$

   Husk at *variablene* $\mathfrak{a}$ og $\mathfrak{b}$ kan stå for hvilken som helst av *bokstavene* a, b og d.

b) Vis, ved å bruke definisjonen av R og egenskapen til PL at $\mathrm{R}(\mathrm{abac}) = \mathrm{caba}$.

Eksempel.

Den siste funksjonen vi skal se på i forbindelse med den induktive definisjonen av mengden av ord er sammensetningsfunksjonen $w * v = wv$.

Denne kan også defineres ved rekursjon som følger:

- $w * e = w$
- $w * (v\mathrm{b}) = (w * v)\mathrm{b}$

Vi kunne gjort det vanskeligere, nemlig brukt rekursjon på første argument:

- $\mathrm{S}(e, v) = v$
- $\mathrm{S}(w\mathrm{b}, v) = \mathrm{S}(w, \mathrm{PL}(v, \mathrm{b}))$

Hvis vi nå vil vise at $\mathrm{S}(w, v) = w * v$ for alle ord w og v, kan vi ha god bruk for induksjon.

- Automatateori er den matematiske teorien for studiet av formelle regnemaskiner som tar for seg et inputord, og enten prosesserer et outputord eller svarer på et spørsmål om inputordet.
- Flere av disse formelle maskinene leser inputordet fra venstre mot høyre, og utfører en beregning underveis.
- Eksempler på dette er de såkalte endelige tilstandsmaskinene og pushdownautomater eller stakkautomater.
- Hvordan en slik automat virker på et ord defineres ved rekursjon på oppbyggingen av ordet.
- Vi skal ikke innføre automatateori, men vi skal i eksempler og oppgaver se på et par tilfeller hvor rekursjon på ord kan brukes til å erstatte slike formelle regnemaskiner.
- De kraftigste formelle maskinene fanges ikke opp av ord-rekursjon.

Eksempel.

- La w være et ord hvor vi vet at bare bokstavene a og b forekommer.

- Vi vil finne en algoritme som avgjør om det er like mange bokstaver av hvert slag, eller om det er et overskudd av den ene.
- Som en hjelpefunksjon definerer vi en funksjon $f(w)$ slik at $f(w)$ er differansen mellom antall a'er og antall b'er i w.
  - $f(e) = 0$
  - $f(wa) = f(w) + 1$
  - $f(wb) = f(w) - 1$

Eksempel (Fortsatt).

- Vi kan da regne ut

$$f(aababba) = f(aababb) + 1 = f(aabab) - 1 + 1 = f(aabab)$$

$$= f(aaba) - 1 = f(aab) = f(aa) - 1 = f(a) = f(e) + 1 = 1.$$

Eksempel (Fortsatt).

- Siden ord er ordnede sekvenser av symboler, kan vi finne pseudokoder som erstatter ordrekursjon.
- Vi skal gi en pseudokode for beregning av f fra dette eksemplet.
- Her er det viktig at *input* er et ord hvor bokstavene a og b forekommer, og at *output* er et helt tall.
- Vanligvis må man deklarere typene til variablene som en del av programmet, men det formaliserer vi ikke her.

Eksempel (Fortsatt).

1 *Input* n [$n \in \mathbb{N}_0$]
2 *Input* w [$w = v_1 \cdots v_n$ er et ord av lengde n]
3 $x \leftarrow 0$
4 **For** $i = 1$ **to** n **do**
  4.1 **If** $v_i = a$ **then**
    4.1.1 $x \leftarrow x + 1$
    else
    4.1.2 $x \leftarrow x - 1$
5 *Output* x

Merk.

- Hvis vi arbeider med algoritmer som skal virke på ord i et alfabet eller lister av data, vil det være aktuelt å innføre egne kontrollstrukturer for listerekursjon.
- Det fins programmeringsspråk av teoretisk interesse, og også av praktisk betydning, hvor det er enkelt å uttrykke listerekursjon og andre former for generell rekursjon.

Påskeferie

Uansett hva klokken er nå, har vi gått igjennom nok stoff, så det gjenstår bare å si Lykke til med midtermin til de som skal ta en slik en.

GOD PÅSKE

MAT1030 – Forelesning 19

Generell rekursjon og induksjon

Dag Normann - 6. april 2010

Forelesning 19

Repetisjon

- Vi er godt i gang med kapitlet om rekursjon og induksjon.
- Vi har sett på hvordan vi kan definere funksjoner eller tallfølger ved rekursjon over $\mathbb{N}$ og på hva det vil si å bevise en påstand ved induksjon.
- Vi så hvordan vi, på en uniform måte, kan omforme en definisjon ved rekursjon til en pseudokode.
- Deretter brukte vi cirka en uke på å se på rekurrenslikninger og på hvordan man finner den generelle løsningen og spesielle løsninger gitt initialbetingelser.
- I siste time før Påske så vi på et eksempel på en induktiv definisjon, definisjonen av ord over et alfabet som lister.
- Vi definerte mengden av ord som et eksempel på en induktivt definert mengde ved
  - Det tomme ordet e er et ord.
  - Hvis w er et ord og a er en bokstav, vil wa være et ord.
- Når vi lagrer et ord som et dataobjekt, vil et ord beskrevet på denne måten ofte representeres anderledes enn hvis vi representerer et ord som en ordnet sekvens.
- Vi ga eksempler på tre funksjoner på mengden av ord, funksjoner som vi definerte ved rekursjon over oppbygningen av et ord:
  - $\mathrm{PL}(w, a) = aw$
  - $\mathrm{R}(w)$ er speilvendingen av w
  - $v * w = vw$.

Generell induksjon og rekursjon

Fra nå av skal vi anta at vi har definert mengen av ord, og at vi kan foreta oss de operasjoner på ord og bokstaver som vi finner nødvendige.

Hvorvidt det er naturlig å bruke *ordrekursjon* eller ikke, avhenger av hvordan vi representerer ord i en datamaskin.

Definisjon.
Et formelt språk er en mengde av ord.

Eksempel.

- Mengden av syntaktisk korrekte program i et gitt programmeringsspråk er et formelt språk.
- Mengden av utsagnslogiske formler i utsagnsvariablene $\mathtt{p}$, $\mathtt{q}$ og $\mathtt{r}$ er et formelt språk, hvis vi regner $\neg$, $\wedge$, $\vee$, $\rightarrow$ og $\leftrightarrow$ med blant bokstavene.
- Mengden av ord fra *delalfabetet* $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$ som uttrykker kubikktall er et formelt språk.
- Mengden av ord som betyr noe både på tysk og spansk er et formelt språk.

Merk.

- Mange av de språkene som oppstår i en informatikk-sammenheng er definert induktivt.
- Den begrensningen vi ga til at symbolene skal fins på tastaturet, er ikke vanlig.
- Det vanlige er at man for ethvert formelt språk også presiserer hvilket alfabet som skal brukes.
- Vår opprinnelige definisjon av formelle språk er brukbar for studiet av programmeringsspråk, men det er viktige eksempler som går ut over begrensningen til tastaturet.

Eksempel.

- Vi skal nå se hvordan vi kan gi en helt presis definisjon av de utsagnslogiske formlene.
- For at ikke eksemplet skal bli for omfattende, dropper vi bindeordene $\rightarrow$ og $\leftrightarrow$ i denne definisjonen.
- Vi lar $\mathtt{p}$, $\mathtt{q}$ og $\mathtt{r}$ være utvalgte bokstaver.
- Vi skal gi en induktiv definisjon av mengden av utsagnslogiske formler hvor vi bruker $\mathtt{p}$, $\mathtt{q}$ og $\mathtt{r}$ som utsagnsvariable, bindeordene $\neg$, $\wedge$ og $\vee$, og hvor vi har formalisert bruken av parenteser.
- Hva vi mener med *induktiv* definisjon vil vi la komme frem gjennom eksemplene.

Eksempel (Fortsatt).

1. $\mathtt{p}$, $\mathtt{q}$ og $\mathtt{r}$ er formler.
2. Hvis $\mathtt{A}$ er en formel, er $\neg\mathtt{A}$ en formel.
3. Hvis $\mathtt{A}$ og $\mathtt{B}$ er formler, er $(\mathtt{A} \wedge \mathtt{B})$ og $(\mathtt{A} \vee \mathtt{B})$ formler.
4. Mengden av formler er den minste mengden som oppfyller 1., 2. og 3. over.

- Når man blir vant til å arbeide med induktive definisjoner, vil vanligvis punktet tilsvarende 4. være underforstått.
- Definisjonen av *ord* hadde langt på vei det samme formatet.

Eksempel (Fortsatt).

- Vi har allerede gitt en viktig definisjon ved rekursjon på oppbyggingen av en formel, uten at vi har sagt direkte at det var det vi gjorde.
- En sannhetsverdifunksjon er en funksjon hvor definisjonsområdet er mengden av fordelinger av sannhetsverdier til variablene p, q og r, og verdiområdet er $\{\mathbf{T}, \mathbf{F}\}$.
- Ved hjelp av sannhetsverditabeller har vi definert funksjonene $F_\neg$, $F_\wedge$ og $F_\vee$, hvor eksempelvis $F_\neg(\mathbf{F}) = \mathbf{T}$, $F_\wedge(\mathbf{T}, \mathbf{F}) = \mathbf{F}$ og $F_\vee(\mathbf{T}, \mathbf{F}\} = \mathbf{T}$.
- Hvis (s_p, s_q, s_r) er en ordnet sekvens av tre sannhetsverdier, definerer vi

  $$F_A(s_p, s_q, s_r)$$

  ved rekursjon på oppbyggingen av A som følger:

Eksempel (Fortsatt).

- $F_p(s_p, s_q, s_r) = s_p$
- $F_q(s_p, s_q, s_r) = s_q$
- $F_r(s_p, s_q, s_r) = s_r$
- $F_{\neg A}(s_p, s_q, s_r) = F_\neg(F_A(s_p, s_q, s_r))$
- $F_{(A \vee B)}(s_p, s_q, s_r) = F_\vee(F_A(s_p, s_q, s_r), F_B(s_p, s_q, s_r))$
- $F_{(A \wedge B)}(s_p, s_q, s_r) = F_\wedge(F_A(s_p, s_q, s_r), F_B(s_p, s_q, s_r))$
- For å kunne forstå denne definisjonen trenger vi mye av det vi har lært, blant annet:
  - Generelle funksjoner
  - Sammensetning av funksjoner
  - Rekursive definisjoner i en generell situasjon.

Eksempel (Fortsatt).

- Vi bestemmer hvordan sannhetsverdifunksjonen skal se ut direkte for de enkleste formlene, nemlig utsagnsvariablene.
- For sammensatte formler vil sannhetsverdifunksjonen avhenge av hvilke sannhetsverdifunksjoner vi har for delformler.
- Det er akkurat denne delen av rekursjonen vi følger når vi skriver ut en sannhetsverditabell.

- Hvis vi skal bruke utsagnslogikk i en programmeringssammenheng, vil det være programmer basert på en slik rekursiv definisjon som ligger til grunn når maskinen beregner verdien av en Boolsk test.

- I vårt neste eksempel skal vi se på en definisjon hvor vi bruker simultan rekursjon.
- Simultan rekursjon innebærer at vi definerer to funksjoner f og g samtidig ved rekursjon.
- For å illustrere hva vi mener med "samtidig" skal vi se på et enkelt eksempel:

Eksempel.

- La $f(1) = 1$
- La $g(1) = 2$
- La $f(n+1) = f(n) \cdot g(n)$ for alle $n \in \mathbb{N}$.
- La $g(n+1) = f(n) + g(n)$ for alle $n \in \mathbb{N}$.
- Vi ser at da kan vi regne ut $f(2) = 1 \cdot 2$ og $g(2) = 1 + 2 = 3$.
- Da er $f(3) = 2 \cdot 3 = 6$ mens $g(3) = 2 + 3 = 5$.
- Slik kan vi fortsette å regne ut resultatene parvis, men vi kommer ingen vei hvis vi bare prøver å regne ut verdiene til f eller bare verdiene til g, vi må regne ut begge funksjonene *simultant*, det vil si *samtidig*.
- Begrunnelsen for at simultan rekursjon er en lovlig definisjonsform er den samme som for vanlig rekursjon.

Eksempel (Fortsatt).

- Hvis vi lar $h(n) = (f(n), g(n))$, det vil si, det ordnede paret av $f(n)$ og $g(n)$, er h en funksjon definert på $\mathbb{N}$ og med verdiområde $\mathbb{N} \times \mathbb{N}$.
- Lar vi $t : \mathbb{N} \times \mathbb{N} \to \mathbb{N} \times \mathbb{N}$ være definert ved

  $$t(a, b) = (a \cdot b, a + b)$$

  ser vi at vi kan erstatte definisjonen av f og g med følgende rekursive definisjon av h:
  - $h(1) = (1, 2)$
  - $h(n+1) = t(h(n))$
- Det er ikke noe spesielt med dette eksemplet, vi kunne gjort en tilsvarende omskrivning hver gang vi definerer funksjoner ved simultan rekursjon.

Eksempel.

Det er ofte mye lettere å studere egenskapene til en utsagnslogisk formel, eksempelvis å undersøke om den er en tautologi, en kontradiksjon eller noe annet, om negasjonstegnet bare står i posisjon rett foran en utsagnsvariabel.

En slik formel sies å være på svak normalform.

Intuitivt kan vi finne en svak normalform ved å bruke deMorgans lover til å skyve negasjonstegnet innover.

Skal vi formulere dette presist, må vi bruke formatet til rekursive definisjoner.

Vi må definere den svake normalformen $SF(A)$ til A og den svake normalformen $SF_\neg(A)$ til $\neg A$ simultant.

Eksempel (Fortsatt).

- La $SF(A) = A$ og $SF_\neg(A) = \neg A$ hvis A er en utsagnsvariabel p, q eller r.
- $SF(\neg A) = SF_\neg(A)$
- $SF_\neg(\neg A) = SF(A)$
- $SF(A \vee B) = SF(A) \vee SF(B)$, $SF(A \wedge B) = SF(A) \wedge SF(B)$.
- $SF_\neg(A \vee B) = SF_\neg(A) \wedge SF_\neg(B)$, $SF_\neg(A \wedge B) = SF_\neg(A) \vee SF_\neg(B)$

- Heller ikke dette eksemplet er valgt bare for å gjøre MAT1030 vanskelig.
- Når man skal skrive programmer for å undersøke om en utsagnslogisk formel er oppfyllbar eller en kontradiksjon, er det til stor hjelp om man kan anta at formelen er på svak normalform.

Eksempel.

Som et eksempel skal vi se på hvordan vi finner den svake normalformen til $(\neg p \wedge q) \vee \neg(p \wedge \neg r)$:

$SF((\neg p \wedge q) \vee \neg(p \wedge \neg r)) =$

$SF(\neg p \wedge q) \vee SF(\neg(p \wedge \neg r)) =$

$(SF(\neg p) \wedge SF(q)) \vee SF_\neg(p \wedge \neg r) =$

$(SF_\neg(p) \wedge q) \vee (SF_\neg(p) \vee SF_\neg(\neg r)) =$

$(\neg p \wedge q) \vee (\neg p \vee SF(r)) =$

$(\neg p \wedge q) \vee (\neg p \vee r)$.

Eksempel.

- Riktig bruk av parenteser er viktig for at programmer skal være syntaktisk korrekte, eller for at matematiske uttrykk skal gi mening.
- I dette eksemplet skal vi se på mengden av korrekte parentesuttrykk, hvor vi bare bruker "("og ")" .
- Vi skal se på hvordan vi kan bygge opp mengden av korrekte parentesuttrykk ved induksjon.
- Deretter skal vi vise at den samme mengden kan beskrives på en annen måte, en måte vi kan bruke for maskinell kontroll av om parenteser er satt riktig eller ikke.
- Vi trenger induksjonsbevis både over oppbyggingen av korrekte parentesuttrykk og over de ikke-negative hele tallene for å vise dette.

Eksempel (Fortsatt).

- Til sist skal vi vise hvordan den alternative skrivemåten kan brukes til å finne en pseudokode som avgjør om et uttrykk med parenteser er korrekt eller ikke.

  De korrekte parentesuttrykkene er den minste mengden av ord som oppfyller

1. Det tomme ordet e er et korrekt parentesuttrykk.
2. Hvis v er et korrekt parentesuttrykk, er $w = (v)$ et korrekt parentesuttrykk.
3. Hvis u og v er korrekte parentesuttrykk, er *sammensetningen* $w = uv$ et korrekt parentesuttrykk.

   Vi skal studere dette viktige formelle språket litt nærmere.

Eksempel (Fortsatt).

Påstand 1

Hvis w er et korrekt parentesuttrykk, vil antall høyre og venstreparenteser i w være det samme.

Bevis

Vi bruker induksjon på oppbyggingen av et parentesuttrykk.

I dette tilfellet er påstanden egentlig opplagt, men vi beviser den ved induksjon likevel, som et eksempel.

Hvis $w = e$ er antall høyre og venstreparenteser begge lik 0, og derfor like.

La $w = (v)$ og anta at påstanden holder for v.

Eksempel (Fortsatt).

Da er både antall venstre- og høyreparenteser i w én større enn tilsvarende tall for v, som er like.

Derfor er de like for w også.

La så $w = uv$ og anta at påstanden holder for både u og for v.

Antall venstreparenteser i w er summen av antall venstreparenteser i u og antall venstreparenteser i v.

Ved induksjonsantagelsen er dette det samme som antall høyreparenteser i u og i v.

Da må summen også være den samme.

Eksempel (Fortsatt).

- Vi definerte mengden av de korrekte parentesuttrykkene som den minste mengden X slik at
  - $e \in X$
  - $v \in X \Rightarrow (v) \in X$
  - $u \in X \wedge v \in X \Rightarrow uv \in X$
- Vi har bevist at mengden Y av ord som har like mange venstre- og høyreparenteser, og ingen andre symboler, oppfyller
  - $e \in Y$
  - $v \in Y \Rightarrow (v) \in Y$
  - $u \in Y \wedge v \in Y \Rightarrow uv \in Y$
- Det betyr at $X \subseteq Y$, og det er en reformulering av Påstand 1.

Eksempel (Fortsatt).

Påstand 2

Hvis w er et korrekt parentesuttrykk, og vi leser w fra venstre mot høyre, finner vi ikke noe sted hvor det har vært flere høyre- enn venstreparenteser.

Bevis

Igjen bruker vi induksjon på oppbyggingen av w som et korrekt parentesuttrykk.

Hvis $w = e$ er dette opplagt riktig.

Anta at påstanden holder for v, og la $w = (v)$.

Når vi leser w fra venstre mot høyre har vi alltid én venstreparentes mer når vi leser w enn når vi er på tilsvarende sted i v.

Eksempel (Fortsatt).

Det betyr at vi har et positivt overskudd av $($ hele tiden mens vi leser v-delen av $(v$, og vi får ikke noe overskudd av $)$ til slutt heller.

Anta at påstanden holder for u og for v, og at $w = uv$.

Når vi leser w fra venstre mot høyre kan vi først ikke opparbeide noe overskudd av $)$ mens vi leser u-delen, og deretter heller ikke mens vi leser v-delen.

Dermed holder påstanden for w også.

Eksempel (Fortsatt).

Påstand 3

La w være et ord hvor vi bare har brukt symbolene "$($" og "$)$".

Hvis w oppfyller konklusjonene i Påstand 1 og Påstand 2, vil w være et korrekt parentesuttrykk.

Bevis

Her lønner det seg å bruke induksjon på lengden av ordet w.

I motsetning til vanlig induksjon, starter beviset her med lengde 0, som er lengden til det tomme ordet.

Hvis lengden av w er 0, det vil si hvis $w = e$, er w et korrekt parentesuttrykk pr. definisjon.

La $w \neq e$ og anta at påstanden holder for alle kortere ord v (Kommentar kommer senere).

Eksempel (Fortsatt).

Vi må dele beviset opp i to tilfeller:

Tilfelle 1: Når vi leser w fra venstre mot høyre finner vi ikke noe sted før til slutt at w har like mange høyre- som venstreparenteser.

Siden w oppfyller påstandene, må w være på formen (v), og siden vi er i tilfelle 1 vil også v oppfylle Påstand 1 og Påstand 2.

Ved induksjonsantagelsen er v korrekt, og da er $w = (v)$ korrekt.

Eksempel (Fortsatt).

Tilfelle 2: Vi kan dele w opp i to ekte delord, $w = uv$, slik at u har like mange venstre- som høyreparenteser.

Da må det samme gjelde for v, siden det gjelder for w.

Videre får vi ikke noe overskudd av høyreparenteser mens vi leser u, siden det er det samme som å lese w. Siden vi ikke starter med noe overskudd av venstreparenteser etter å ha lest u, vil situasjonen være lik om vi leser v direkte eller etter å ha lest u.

Det betyr at Påstand 1 og Påstand 2 holder for både u og v.

Eksempel (Fortsatt).

Ved induksjonsantagelsen er da u og v korrekte.

Da er $w = uv$ også korrekt.

Siden dette argumentet dekker alle mulighetene, er Påstand 3 vist ved induksjon over $\mathbb{N}_0$

Eksempel (Fortsatt).

- La oss oppsummere dette eksemplet.
- For å vise Påstand 1 brukte vi induksjon over oppbyggingen av et uttrykk som et korrekt parentesuttrykk.
- Vi brukte induksjon over den samme induktivt definerte mengden for å bevise Påstand 2.
- For å bevise Påstand 3 brukte vi imidlertid induksjon over $\mathbb{N}_0$.
- Det er mulig å skrive om bevisene for Påstand 1 og for Påstand 2 slik at de blir beviser ved induksjon over $\mathbb{N}_0$, eksempelvis bruke induksjon på antall parenteser i uttrykket.
- Dette er en omskrivning som i innlæringsfasen kan gjøre det lettere å forstå generell induksjon, men som på sikt gjør det mer tungvint å formulere bevisene.

Eksempel (Fortsatt).

- Noen vil stusse over at vi ikke brukte den vanlige formuleringen med å vise $P(1)$ og at $P(n) \Rightarrow P(n+1)$ for alle n.
- Det vi brukte her var et annet induksjonsprinsipp:

  Anta at vi kan vise for alle n at

  $$\forall m < n P(m) \to P(n)$$

- Da kan det ikke fins noe minste tall n slik at $P(n)$ er usann.
- Dermed vet vi at $P(n)$ er sann for alle n.
- Det er fullt ut akseptabelt å bruke denne alternative formen i bevis.
- Vi skal komme tilbake med et eksempel til på denne formen for induksjon.

Eksempel (Fortsatt).

- Hvordan kan vi så finne en pseudokode som avgjør om et parentesuttrykk er korrekt eller ikke?
- Vi skriver en kode for funksjonen som teller overskudd av "(" i forhold til ")".
- Underveis kontrolleres det at vi ikke får for mange ")".
- Til sist kontrolleres det at vi hadde like mange av hver.
- n hentes fra $\mathbb{N}_0$, alle v_i'ene er parenteser, x varierer over $\mathbb{J}$ og y tar JA/NEI som mulige verdier.
- Pseudokoden, som egentlig er en repetisjon av hva vi har sett på før, kommer på neste side.

1 *Input* n
2 *Input* $w = v_1 \cdots v_n$
3 $x \leftarrow 0$
4 $y \leftarrow JA$
5 **For** $i = 1$ **to** n **do**
  5.1 **If** $v_i = ($ **then**
    5.1.1 $x \leftarrow x + 1$
    else
    5.1.2 $x \leftarrow x - 1$
  5.2 **If** $x < 0$ **then**
    5.2.1 $y \leftarrow NEI$
6 **If** $x > 0$ **then**
  6.1 $y \leftarrow NEI$
7 *Output* y

Merk.

- I virkelighetens verden har vi flere typer parenteser å holde styr på.
- Skal vi lage en algoritme som virker i en slik situasjon, kan vi ikke la x ta verdier i mengden av hele tall, men i mengden av lister av venstreparenteser.
- Hver gang vi treffer på en venstreparentes av et slag, legger vi den til listen.
- Hver gang vi treffer på en høyreparentes, må vi sammenlikne den med venstreparentesen ytterst i listen (hodet).
- Vi trenger et språk for håndtering av lister for å kunne skrive pseudokoder basert på denne skissen.
- For de som senere lærer om "push-down"-automater, vil det å lage en automat som realiserer denne algoritmen være en enkel oppgave.

Eksempel.

- La oss gå tilbake til den formen for induksjon som vi brukte for å vise at alle parentesuttrykk som oppfyller konklusjonene i Påstand 1 og 2 vil være korrekte.
- Da vi diskuterte kontrapositive argumenter, ga vi et eksempel på hvordan man kan vise at alle tall $n \geqslant 2$ kan faktoriseres i primtall (hvor det å konstatere at et tall er et primtall betraktes som en faktorisering).
- Det er lettere å formulere beviset hvis man kan bruke induksjon.
- La $n \geqslant 2$, og anta at påstanden holder for alle m slik at $2 \leqslant m < n$.
- Hvis n ikke er et primtall, fins tall $m < n$ og $k < n$ slik at $n = mk$.

Eksempel (Fortsatt).

- Ved antagelsen kan både m og k faktoriseres i primtall.
- Ved å bruke alle disse primtallsfaktorene sammen, får vi en primtallsfaktorisering av n.

Merk.

Dette argumentet gir oss ikke lov til å konkludere at det fins nøyaktig en måte å faktorisere et tall på, bare at det fins minst en.

- I eksemplet over viste vi en egenskap $P(n)$ ut fra antagelsen om at $P(m)$ og $P(k)$ holder for to utvalgte tall $m < n$ og $k < n$.
- Vi konkluderte med at $P(n)$ må holde for alle n.
- Vi kan gi en generell kontrapositiv begrunnelse for at det må være slik.
- Anta at det fins et tall n_1 slik at $P(n_1)$ ikke holder.
- Ved argumentet vårt må det da fins $n_2 < n_1$ slik at heller ikke $P(n_2)$ holder.
- Vi kan fortsette med å finne $n_3 < n_2$, $n_4 < n_3$ osv. slik at $P(n_i)$ ikke holder for noen $i \in \mathbb{N}$.

! Det fins imidlertid ingen strengt synkende følge av naturlige tall, så dette er umulig.

- Vi kan i prinsippet bruke induksjon som bevisform for enhver ordning som ikke tillater noen uendelig strengt synkende følge.
- Vi skal ikke presse denne sitronen lenger, det er sikkert surt nok for mange.

MAT1030 – Forelesning 20

Kombinatorikk

Dag Normann - 7. april 2010

Fortsettelse fra 06.04.2010

Rekursjon og induksjon

- I går så vi på formelle språk generelt, og på de formelle språkene av utsagnslogiske formler og av korrekte parentesuttrykk spesielt.
- Eksemplene vi har sett på bruk av rekursjon på oppbyggingen av ord eller formler er så sentrale at de kan bli brukt som grunnlag for oppgaver gitt til eksamen eller til Oblig. 2.
- Det neste eksemplet har ikke den samme betydningen i informatikk, og kan derfor best sees på som et eksempel for innøvelse av forståelse og ferdigheter.
- I eksemplet beskriver vi "den universelle datatype", en universell mengde hvor vi kan finne alle *virkelige* datatyper som delmengder.

Eksempel.

- Vi definerer de hereditært endelige mengdene HF som den minste mengden slik at
  - $\emptyset \in HF$
  - Hvis $X = \{a_1, \ldots, a_n\}$ er en endelig delmengde av HF, så er $X \in HF$.
- Vi definerer $f : HF \to \mathbb{N}_0$ ved rekursjon over HF ved
  - $f(\emptyset) = 0$
  - $f(\{a_1, \ldots, a_n\}) = 2^{f(a_1)} + \cdots + 2^{f(a_n)}$.

  Her må vi anta at mengden er beskrevet uten gjentagelse, det vil si at alle a_i'ene er forskjellige.

Eksempel (Fortsatt).

- Ved induksjon over HF ser vi at $f(X) \in \mathbb{N}_0$ for alle $X \in HF$.

  Hvis $X = \emptyset$ ser vi det direkte, og hvis $X = \{a_1, \ldots, a_n\}$ kan vi bruke induksjonsantagelsen som sier at $f(a_i) \in \mathbb{N}_0$ for alle i.
- Hvis $k \in \mathbb{N}_0$ vil det fins en og bare en $X \in HF$ slik at $f(X) = k$.
- Her bruker vi induksjon på k.

  For $n = 0$ ser vi at det bare er $X = \emptyset$ som kan gi $f(X) = 0$ siden $2^{f(a_i)} > 0$ for alle mulige $a_i \in X$.

For $k > 0$, fins det en og bare en måte å skrive k som en sum av forskjellige 2'erpotenser på, og hver eksponent k_i vil komme fra en og bare en mengde a_i.

Det betyr at vi kan finne X slik at $f(X) = k$ fra k.

Oppgave.

La HF og f være definert som i eksemplet over, og la $S : HF \to HF$ være definert ved

$$S(X) = X \cup \{X\}.$$

a) Vis at $X \in HF \Rightarrow S(X) \in HF$.

b) La $X, Y \in HF$.
Vis at $X \in Y \Rightarrow f(X) < f(Y)$.

c) Forklar hvorfor det ikke fins noen $X \in HF$ slik at $X \in X$

d) La $N \subseteq HF$ være den minste mengden slik at $\emptyset \in N$ og slik at hvis $X \in N$ så vil $S(X) \in N$.
Drøft sammenhengen mellom N og $\mathbb{N}_0$.

- Hvis vår induktive definisjon av mengden av utsagnslogiske formler skulle vært gitt i en lærebok i informatikk på et avansert nivå, eller i en forskningsartikkel, ville den sett ut som noe slikt:

  Utsagnsvariabel ν

  $\nu ::= p \mid q \mid r$

  Formel A

  $A ::= \nu \mid \neg A \mid (A \wedge A) \mid (A \vee A)$

- De to første linjene skal leses som følger:

  I denne definisjonen lar vi ν betegne en vilkårlig utsagnsvariabel.

  ν kan stå for p, q eller r.

- De to neste linjene skal leses som følger:

  I denne definisjonen lar vi A betegne en vilkårlig formel.

  En formel kan enten være en utsagnsvariabel, fremkommet fra en annen formel ved å skrive $\neg$ foran, eller fremkommet fra to andre formler ved å skrive $\wedge$ eller $\vee$ mellom, og parenteser rundt.

- Vi kan bruke den samme effektive notasjonen for å definere mengden av korrekte parentesuttrykk:

  Parentesuttrykk P

  $P ::= e \mid (P) \mid PP$

  som uttrykker at vi får parentesuttrykkene ved å starte med det tomme ordet e og deretter enten sette parenteser rundt et korrekt uttrykk eller sette to korrekte uttrykk sammmen.

- Som tidligere nevnt, liker informatikere og logikere å la de naturlige tallene starte med 0, altså å arbeide med $\mathbb{N}_0$ i stedet for $\mathbb{N}$.
- Følgende definisjon er påtruffet i informatikklitteratur:

  $n : \mathsf{NAT}$

  $n ::= 0 \mid S(n)$.
- Dette skal leses som at vi definerer en datatype NAT ved å la n stå for et vilkårlig objekt, og vi finner objektene i NAT, enten som *symbolet* 0 eller som en *symbolsekvens* $S(w)$ hvor w er en symbolsekvens vi allerede vet er av type NAT.

Oppsummering

Kapittel 1–3

- algoritmer
- pseudokoder
- kontrollstrukturer
- representasjon av tall (hele og reelle tall)
- tallsystemer

Kapittel 4

- logisk holdbare argumenter
- utsagnslogikk og predikatlogikk
- utsagnslogiske bindeord og kvantorer
- parenteser
- sannhetsverditabeller, tautologier, kontradiksjoner
- logisk konsekvens, logiske lover
- bevisteknikker

Kapittel 5

- mengdelære
- notasjon
- mengdealgebra (union, snitt, komplement, mengdedifferens)
- Venndiagrammer
- kardinaltall, potensmengder, ordnede par
- binære relasjoner og egenskaper ved disse
- ekvivalensrelasjoner og partielle ordninger

Kapittel 6

- funksjoner
- terminologi, verdiområde, definisjonsområde
- surjektive og injektive funksjoner
- sammensetning og invers av funksjoner

Kapittel 7

- rekursjon og induksjon
- rekursive funksjoner
- induksjonsbevis
- rekurrenslikninger
- induktivt definerte mengder, f.eks.
  - mengden av ord over et alfabet
  - mengden av utsagnslogiske formler
  - mengden av parentesuttrykk
- generell rekursjon og induksjon

Kapittel 9: Kombinatorikk

Kombinatorikk

- Kombinatorikk er studiet av *opptellinger*, *kombinasjoner* og *permutasjoner*.
- Vi finner svar på spørsmål "Hvor mange måter ...?" uten å telle.
- Viktig del av f.eks. kompleksitetsanalyse av algoritmer.
  - Hvor mye *tid* bruker en algoritme?
  - Hvor mye *plass* bruker en algoritme?
- Grunnleggende, nyttig og fascinerende matematikk som dere må beherske.
- Vi bruker cirka to dobbeltimer på kapittel 9.

Eksempel.

- Spørsmål:

  Når de syv rette lottotallene på en kupong med 34 tall er trukket ut, hvor mange forskjellige rekker har seks rette?
- Svar:

  Det er syv forskjellige måter å plukke ut seks rette fra de syv rette.

  Det er 27 måter å velge ut det ene tallet som er feil.

  Det gir $7 \cdot 27 = 189$ forskjellige rekker med seks rette.

- Kombinatorikk inngår som et vesentlig element i sannsynlighetsteori.
- Kombinatorikk inngår også når man skal vurdere hvor lang tid et program trenger for å nå i mål og når man skal vurdere hvor stor lagringsplass man må sette av for at et program eller en programpakke skal få det nødvendige arbeidsrommet.
- Lagring av data i forskjellige registere kan illustreres ved lagring av kuler i bokser.
- Et naturlig spørsmål vil da være hvor mange forskjellige måter dette kan gjøres på.
- I dette eksemplet skal vi anta at vi har tretten kuler og fire bokser.

- Hvis vi spør om på hvor mange måter vi kan fordele 13 kuler på fire forskjellige bokser, er det to mulige presiseringer.

Eksempel (Tilfelle 1).

- Alle kulene er forskjellige.
- Da har vi 13 kuler, og vi har fire muligheter for plassering av hver kule.
- Det gir

$$4^{13}$$

mulige fordelinger.

Eksempel (Tilfelle 2).

- Kulene er like, mens boksene fortsatt er forskjellige, kall dem A, B, C og D.
- La

$$\mathrm{xxxxxxxxxxxxxxxx}$$

være en mengde med 16 elementer (en mindre enn antall kuler pluss antall bokser).

- Det fins

$$\binom{16}{3} = \frac{16!}{3! \cdot 13!} = 960$$

måter å omgjøre tre x til X på.

Eksempel (Tilfelle 2, fortsatt).

- Eksempel

$$\mathrm{xxxXxxxxXxxXxxxx}$$

- I dette tilfellet plasserer vi tre kuler i A (foran første X), fire kuler i B (mellom første og andre X), to kuler i C (mellom andre og tredje X) og fire kuler i D (bak siste X).
- Alle plasseringer av de tre X'ene gir oss en fordeling av kulene på de fire boksene.

Eksempel (Tilfelle 2, fortsatt).

- Omvendt vil en fordeling av 13 kuler på boksene A, B, C og D gi oss en plassering av X'ene.
- Har vi to kuler i A, to i B, fem i C og fire i D svarer det til

$$\mathrm{xxXxxXxxxxxXxxxx}.$$

- Dette eksemplet er hva vi kaller generisk.
- Det betyr at vi kan lese en generell setning ut av eksemplet:

Teorem.
Det finnes

$$\left(\begin{array}{c} n+m-1 \\ m-1 \end{array} \right) = \frac{(n+m-1)!}{(m-1)! \cdot n!}$$

måter å fordele n like objekter på m ulike beholdere på.

Merk.

- Vi kunne ha formulert problemet om antall fordelinger også i det tilfellet hvor det ikke er forskjell på boksene.
- Det krever imidlertid at vi går ut over læreboka i en retning som ikke er prioritert, så det skal vi la ligge.
- Vi kunne også ha sett på tilfellet der vi har seks hvite og syv røde kuler.
- Dette er en utfordring dere bør kunne håndtere selv når vi er ferdige med kapittel 9.

- Vi skal komme tilbake til opptelling av mulige fordelinger i forskjellige situasjoner.
- Først skal vi imidlertid se på en sammenheng mange kjenner fra sannsynlighetsteorien.
- I læreboka går den under betegnelsen Inklusjons- og eksklusjonsprinsippet.

Inklusjons- og eksklusjonsprinsippet

Eksempel.

- På en skole er det 177 elever som driver aktiv idrett.

  103 av elevene er aktive om sommeren og 85 av elevene er aktive om vinteren.

  Det betyr at det er 188 aktiviteter fordelt på 177 elever.

  For at dette skal stemme, må 11 av elevene drive både sommer- og vinteridrett, siden det er 11 flere aktiviteter enn det er elever.
- Hvis vi lar S være mengden av elever som driver sommeridrett og V være mengden av elever som driver vinteridrett, ser vi at
  - $|S| = 103$
  - $|V| = 85$
  - $|S \cup V| = 177$
  - $|S \cap V| = 11$

Eksempel (Fortsatt).

- Det siste tallet regnet vi ut på grunnlag av de tre første.
- Vi ser at $|S| + |V| = |S \cup V| + |S \cap V|$ fordi det at det er flere aktiviteter enn elever skyldes at noen driver to aktiviteter, og differensen mellom antall aktiviteter og antall elever må være nøyaktig antallet på de som driver både sommer- og vinteridrett.

Teorem (Inklusjons- og eksklusjonsprinsippet).

La A og B være to endelige mengder.

Da er $|A \cup B| = |A| + |B| - |A \cap B|$.

Bevis.

Hvis vi først teller opp elementene i A og deretter elementene i B, har vi talt elementene i $A \cap B$ to ganger.

For å få antall elementer i $A \cup B$ må vi derfor trekke fra det vi har talt for mye, nemlig antallet i $A \cap B$.

Merk.

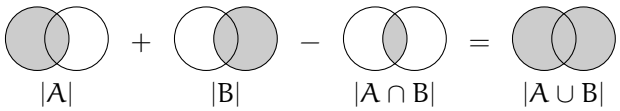
- Det er en nær sammenheng mellom *inklusjons- og eksklusjonsprinsippet* og en tilsvarende lov om sannsynlighet:

$$P(A) + P(B) = P(A \cup B) + P(A \cap B)$$

når A og B er uavhengige hendelser og P måler en sannsynlighet.

- Begge lovene illustreres greit med et Venn-diagram, hvor man ser at hvis vi skraverer sirkelskiven som markerer A i en retning og sirkelskiven som markerer B i en annen retning, er det akkurat feltet som markerer $A \cap B$ vi skraverer to ganger.

- Inklusjons- og eksklusjonsprinsippet:



Eksempel.

- Anta at vi har fått følgende oppgave:

  Av 231 studenter var det 174 som greide oppgave 1 og 175 som greide oppgave 2.

  Alle studentene greide minst en oppgave.

  Hvor mange studenter greide begge oppgavene?

Eksempel (Fortsatt).

- Løsning:

  La A være mengden av studenter som greide oppgave 1 og B mengden av studenter som greide oppgave 2.

  Da er $|A \cup B| = 231$, $|A| = 174$ og $|B| = 175$.

  Inklusjons- og eksklusjonsprinsippet sier oss at

$$231 = 174 + 175 - |A \cap B|.$$

  Det gir oss at

$$|A \cap B| = 174 + 175 - 231 = 118.$$

  Det var 118 studenter som greide begge oppgavene.

Eksempel.

La oss se på følgende oppgave:

- Medlemmene i et idrettslag blir bedt om å registrere seg på nettet med navn, adresse og enten e-postadresse eller mobilnummer.

  Ledelsen ønsker å automatisere utsendelsen av informasjon, uten å bruke to informasjonskanaler til samme medlem, men av de 728 medlemmene er det 94 som har oppgitt både e-postadresse og mobilnummer.

  Når vi vet at 562 medlemmer oppga e-postadresse, hvor mange oppga da mobilnummer?

Eksempel (Fortsatt).

- Løsning:

  La A være mengden av medlemmer som oppga e-postadresse og B være mengden av de som oppga mobilnummer.

  Da sier inklusjons- og eksklusjonsprinsippet at

$$728 = 562 + |B| - 94$$

så

$$|B| = 728 + 94 - 562 = 260.$$

Det var 260 medlemmer som oppga mobilnummer.

Multiplikasjonsprinsippet

- Det neste prinsippet for beregning av antall muligheter vi skal se på er multiplikasjonsprinsippet.
- Multiplikasjonsprinsippet sier at hvis vi skal treffe en serie uavhengige valg, vil det totale antall muligheter være produktet av antall muligheter ved hvert valg.
- Igjen fins det en klar parallell i sannsynlighetsteori, hvor vi fins sannsynligheten for at en serie uavhengige hendelser finner sted ved å ta produktet av sannsynlighetene for enkelthendelsene.
- Vi skal illustrere dette prinsippet ved et par eksempler.

Eksempel.

- Et norsk registreringsnummer for bil består av to store bokstaver og fem siffre.
- Vi bruker ikke bokstavene G, I, O, Q, Æ, Ø eller Å, fordi de enten ikke forekommer utenlands, eller fordi de kan forveksles med tall eller andre bokstaver.
- Da står vi igjen med $22 \cdot 22 = 484$ bokstavkombinasjoner.

Eksempel (Fortsatt).

- Første siffer i nummeret må være et av de ni tallene 1, 2, 3, 4, 5, 6, 7, 8, 9, mens de fire andre sifrene kan hentes fra alle de ti tallsymbolene.
- Det gir tilsammen

$$22 \cdot 22 \cdot 9 \cdot 10 \cdot 10 \cdot 10 \cdot 10 = 43.760.000$$

mulige registreringsnummere på norske biler.

Eksempel (Fortsatt).

- Svenskene bruker tre bokstaver og tre tall, og hadde de begrenset seg til de bokstavene vi bruker i Norge, ville de kunne registrert færre biler.

Oppgave.

Hvor mange bokstaver må svenskene tillate for å kunne registrere like mange biler (eller fler) enn det nordmennene kan?

Eksempel.

- Amerikanske lærebokforfattere lever i den tro at amerikanske collegestudenter lever en ikke ubetydelig del av livet sitt med å spise sammen.
- Derfor er følgende oppgave typisk for amerikanske lærebøker i diskret matematikk.
- En sandwich-bar tilbyr:
  1. Fire typer brød: Fint, mellomgrovt, grovt og glutenfritt.
  2. Tre typer smøring: Smør, majones og sennep.
  3. Seks typer hovedpålegg: Kalkun, roastbeef, skinke, tunfisk, skalldyr og soyaprotein.
  4. Fire typer tilbehør: Stekt bacon, salat, agurk og tomat.
  5. Tre valg på dressing, Thousen Islands, tomatdressing og hvitløksdressing.
- Hvor mange forskjellige sandwicher er det mulig å komponere?

Eksempel (Fortsatt).

Selv om ikke alle sammensetningene vil være like vellykkede rent smaksmessig, er det ingen føringer på hvilke valg som kan kombineres.

Da finner vi det totale antall muligheter ved å bruke multiplikasjonsprinsippet.

- Vi har da

$$4 \cdot 3 \cdot 6 \cdot 4 \cdot 3 = 864$$

forskjellige sammensetninger.

MAT1030 – Forelesning 21

Mer kombinatorikk

Dag Normann - 13. april 2010

Kapittel 9: Mer kombinatorikk

Oppsummering

- Forrige uke startet vi på kapitlet om kombinatorikk.
- Vi så på hvordan vi kan finne antall måter å fordele n like objekter på k ulike beholdere på.
- Dette skal vi komme tilbake til.
- Vi så på inklusjons- og eksklusjomsprinsippet:

$$|A \cup B| + |A \cap B| = |A| + |B|$$

- Videre så vi på multiplikasjonsprinsippet.
- Det skal vi fortsette med i dag.

Multiplikasjonsprinsippet

Etter dagens forelesning skal følgende oppgave være lett:

Oppgave.

a) Vi skal fordele syv like hvite kuler og seks like røde kuler på fire forskjellige bokser.
Hvor mange måter kan dette gjøres på?

b) Hvis vi krever at de hvite kulene skal ligge i de tre første boksene og de røde i de tre siste, hvor mange mulige fordelinger har vi da?

c) Løs a) hvis vi i utgangspunktet bare hadde tre bokser, og sammenlikn svaret med svaret fra b).
Forklar det du observerer.

- Multiplikasjonsprinsippet:
Hvis vi skal treffe en serie uavhengige valg, vil det totale antall muligheter være produktet av antall muligheter ved hvert valg.

$$|A \times B| = |A| \cdot |B|$$

Antall elementer i det kartesiske produktet $A \times B$ er antall elementer i A multiplisert med antall elementer i B.

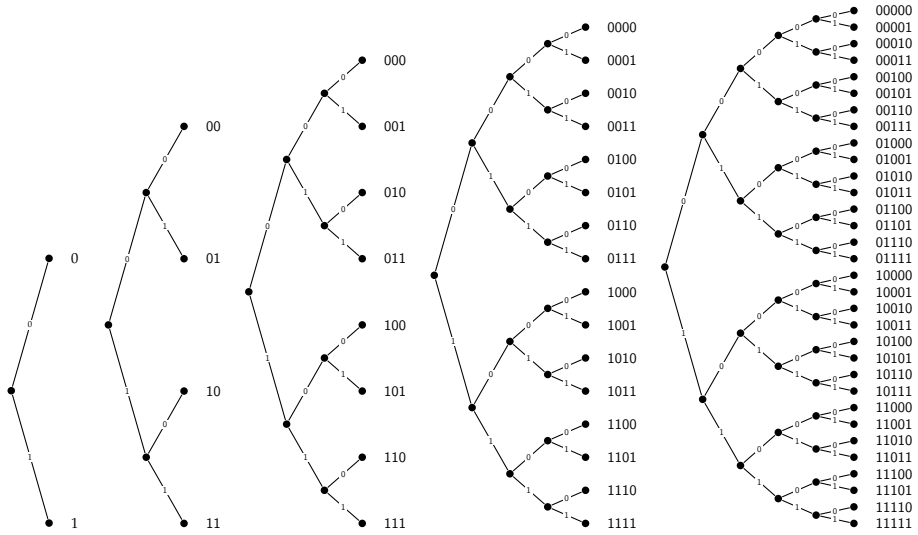
- Både inklusjons- og eksklusjonsprinsippet og multiplikasjonsprinsippet kan generaliseres til flere enn to mengder.

– Generaliseringen av inklusjons- og eksklusjonsprinsippet til tre mengder ses ved hjelp av Venn-diagrammer. Det vil vi ikke få bruk for.
– Generaliseringen av multiplikasjonsprinsippet blir

$$|A_1 \times A_2 \times \cdots \times A_n| = |A_1| \cdot |A_2| \cdot \cdots \cdot |A_n|$$

Det vil vi få bruk for.

Eksempel - det er 2^n binære tall av lengde n

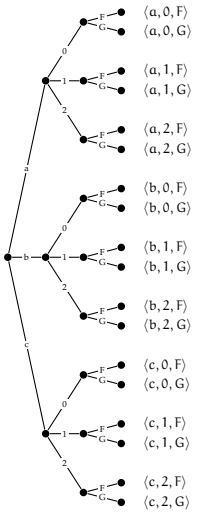


Eksempel - $\{a, b, c\} \times \{0, 1, 2\} \times \{F, G\}$

Multiplikasjonsprinsippet gir oss følgende:

$$\begin{aligned}
&|\{a, b, c\} \times \{0, 1, 2\} \times \{F, G\}| \\
&= |\{a, b, c\}| \cdot |\{0, 1, 2\}| \cdot |\{F, G\}| \\
&= 3 \cdot 3 \cdot 2 \\
&= 18
\end{aligned}$$

Vi kan illustrere det slik:



Permutasjoner

- Det neste vi skal se på er hva vi mener med en permutasjon og på hvordan vi kan telle opp antall permutasjoner av en ordnet mengde.

- En permutasjon er en endring av en rekkefølge, eller en omstokking.
- Når vi stokker en kortstokk er poenget at kortene skal ligge i en annen rekkefølge, og med et fremmedord kan vi si at vi permuterer kortene.
- Vi skal se på noen eksempler.

Eksempel.

- På hvor mange forskjellige måter kan vi skrive tallene 1, 2 og 3 i rekkefølge?
- Vi har tre valg for hvilket tall vi vil skrive først: 1, 2 eller 3.
- For hvert av disse valgene har vi to valg for hvilket som blir det neste tallet:
  Starter vi med 1 må det neste tallet være 2 eller 3, starter vi med 2 må det neste tallet være 1 eller 3 og starter vi med 3 må det neste tallet være 1 eller 2.
- Har vi bestemt hvilke to tall vi skriver først, gir det siste tallet seg av seg selv.
- Det fins altså $3 \cdot 2 = 6$ måter å skrive disse tre tallene i rekkefølge på.

Eksempel.

- Hvis vi utvider eksemplet vårt fra forrige side til å omfatte tallene 1, 2, 3 og 4 vil antall permutasjoner vokse til $4! = 24$ og tar vi med 5 i tillegg er antallet $5! = 120$.
- I det siste tilfellet har vi først fem valg for hvilket tall som skal skrives først, deretter fire valg for tall nr. 2, tre valg for tall nr. 3 og to valg for tall nr. 4. Det siste tallet gir seg selv.
- Generelt fins det $n!$ permutasjoner av tallene $1, \ldots, n$.
- Dette svarer også til hvor mange rekkefølger vi kan sette n elementer i. Eksempelvis kan syv studenter ordnes på $7! = 6720$ måter.

Definisjon (Permutasjon).
En *permutasjon* er en endring av rekkefølgen av elementene i en ordnet mengde. Vi sier også at en *permutasjon* av en mengde er en ordning av elementene i mengden.
Her bruker vi ordet *permutasjon* slik boka tillater det, men det vanlige er å oppfatte en permutasjon som en omstokking, elementene bytter plass med hverandre.

Eksempel.
Permutasjonene av $\{A, B, C\}$ er ABC, ACB, BAC, BCA, CAB, CBA.

- Det er $n!$ permutasjoner av en mengde med n elementer.

- Og vi vet (selvfølgelig) at $n! = n \cdot (n-1) \cdot (n-2) \cdots 1$
- I eksempelet har vi 3 elementer og $3! = 3 \cdot 2 \cdot 1 = 6$ permutasjoner.

Eksempel.

- Et kjent problem i litteraturen er Den handelsreisendes problem (The traveling salesman).
- Hvis vi har gitt n byer som skal besøkes, og vi kjenner avstanden mellom to og to av byene, hva er da den korteste veien gjennom alle byene?
- Det er ennå ingen som har kommet opp med et program som løser dette problemet når antall byer er stort, som for eksempel alle tettsteder i Norge med mer enn 300 innbyggere.
- Vi skal se på hva dette problemet kan ha med antall permutasjoner å gjøre.

Eksempel (Fortsatt).

Hvis antall byer som skal besøkes er mindre, blir selvfølgelig oppgaven gjennomførbar.

Anta at vi har fått i oppdrag å skrive et program som finner den korteste reiseruten fra by A til by B, og som går gjennom ti andre byer $C_1, \ldots, C_{10}$ i en eller annen rekkefølge.

Igjen kan vi anta at alle avstander er kjent.

En måte å gjøre dette på er å liste opp alle mulige rekkefølger vi kan besøke byene $C_1, \ldots, C_{10}$ i, regne ut alle reiselengdene og så velge ut den korteste.

Problemet er at det fins $10! = 3.628.800$ forskjellige rekkefølger vi kan velge mellom.

Eksempel (Fortsatt).

- Det betyr altså at det fins over tre og en halv million måter å reise fra Oslo til Kirkenes på, når reisen skal gå via
  - Kristiansand
  - Stavanger
  - Bergen
  - Molde
  - Kristiansund
  - Trondheim
  - Bodø
  - Narvik
  - Tromsø
  - Alta

Eksempel (Fortsatt).

Øker vi antall byer som skal besøkes til 12, hvis vi for eksempel vil besøke Haugesund og Levanger i tillegg, vil vi være i nærheten av 400.000.000 enkeltruter, og da begynner de raske maskinene å slite.

Det vil gå flere generasjoner maskiner mellom hver gang vi kan øke antall byer med 1 hvis vi bruker denne naive måten.

Eksempel (Fortsatt).

- Det man i praksis gjør er å akseptere at det er dumt å bruke år på å finne ut av om man kan spare noen få kilometers reise, og utvikler raske algoritmer som gir effektive reiseruter, uten å garantere at den finner den mest effektive.
- Det fins elektroniske reiseplanleggere som må forene hensynet til kort regnetid og et godt resultat.
- Tilsvarende optimeringsproblemer finner man for effektiv utnyttelse av lagerplass, effektiv organisering av produksjonsleddene i en bedrift og liknende.

Eksempel.

- Dette eksemplet er stjålet fra en tidligere lærebok i diskret matematikk, den gang det het MA 108.
- Hvor mange ord kan vi skrive ved hjelp av bokstavene i

  MISSISSIPPI?

- Det er 11 bokstaver, og har vi en blytype for hver bokstav, kan vi sette disse i $11!$ forskjellige rekkefølger.
- Det gir oss 39 916 800 forskjellige rekkefølger.

Eksempel (Fortsatt).

- Rekkefølgen vi setter de to P'ene i, betyr imidlertid ikke noe for resultatet. Det alene halverer antall ord vi kan skrive.
- Det er fire I'er og fire S'er. Den innbyrdes rekkefølgen blant I'ene og blant S'ene betyr heller ikke noe for hvordan det ferdige ordet ser ut.
- Det er $4! = 24$ måter å trykke de fire S'ene og $4! = 24$ måter å trykke de fire I'ene på.
- Det betyr at antall forskjellige ord vi kan skrive er

$$\frac{11!}{4! \cdot 4! \cdot 2!} = 34650.$$

Oppgave.

Hvor mange forskjellige ord kan vi skrive ved å stokke om på bokstavene i ordet

PUSLESPILL

Regn ut svaret fullstendig.

Ordnet utvalg

Vi skal nå se på det som kalles ordnet utvalg fra en mengde.

Eksempel.

Oppgave:

I et barneskirenn er det med 20 barn.

Det er lov til å opplyse om hvem som tok de tre første plassene, mens resten ikke skal rangeres.

Hvor mange forskjellige resultatlister kan man få?

Eksempel (Fortsatt).

Løsning:

Det fins 20 mulige vinnere, deretter 19 mulige andreplasser og til sist 18 mulige tredjeplasser.

Det fins altså $20 \cdot 19 \cdot 18 = 6840$ forskjellige resultatlister.

- Legg merke til at

$$20 \cdot 19 \cdot 18 = \frac{20 \cdot 19 \cdot 18 \cdot 17 \cdot 16 \cdots 2 \cdot 1}{17 \cdot 16 \cdots 2 \cdot 1} = \frac{20!}{(20-3)!}$$

- Vi skal nå definere dette mer generelt og bruke notasjonen ${}^{20}P_3$ for dette tallet.

Definisjon.

- La r og n være naturlige tall slik at $r \leqslant n$.
- Med ${}^{n}P_r$ mener vi $\frac{n!}{(n-r)!}$

Merk.

- ${}^{n}P_{r}$ forteller oss hvor mange måter vi kan trekke r elementer i *rekkefølge* ut fra en mengde med n elementer på.

- Når $n = r$ bruker vi at $0! = 1$. Da får vi

$$ {}^{n}P_{n} = \frac{n!}{(n-n)!} = \frac{n!}{0!} = \frac{n!}{1} = n! $$

- Det er som forventet, siden det er $n!$ permutasjoner av en mengde med n elementer i.

Eksempel.

- En idrettsleder har syv løpere i stallen sin, og skal velge ut fire av dem til å delta i en stafett.
- I et stafettlag spiller rekkefølgen stor rolle, især om idrettsgrenen er langrenn og det er to etapper i klassisk og to i fristil.
- Da er det ${}^{7}P_{4} = \frac{7!}{3!} = 7 \cdot 6 \cdot 5 \cdot 4 = 840$ forskjellige mulige laguttak.

Kombinasjoner

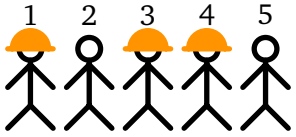
$$ \binom{n}{k} $$

angir hvor mange delmengder med k elementer det finnes av en mengde med n elementer

- Vi har tidligere vist dette ved induksjon.
- Det er også mulig å vise dette rent kombinatorisk, som vi skal gjøre snart.
- Slike tall kalles blant annet for *binomialkoeffisienter*.

[fragile]

- Anta at vi skal fordele tre oransje hatter,  , på fem barn.



- Hver kombinasjon svarer til en delmengde av $\{1,2,3,4,5\}$.
- Antall måter å velge tre barn på *i rekkefølge* er ${}^{5}P_{3} = 5 \cdot 4 \cdot 3 = 60$.
- Her vil hver kombinasjon av hatter, f.eks. $\{1,3,4\}$, bli talt $3! = 6$ ganger, som 134, 143, 314, 341, 413, og 431.
- Hvis vi skal ta høyde for dette, så må vi dele på 6.
- Antall måter å fordele hattene er derfor $60/6 = 10$, som er $\binom{5}{3}$.

Teorem.

- La A være en mengde med n elementer, og la $0 \leqslant k \leqslant n$.
- Da finnes det
$$\binom{n}{k}$$
forskjellige delmengder B av A.

Bevis (Nytt, og fritt for induksjon).

- Antall måter å velge k elementer *i rekkefølge* fra A på er
$${}^{n}P_{k} = \frac{n!}{(n-k)!}$$
- For hver delmengde B med k elementer, så fins det $k!$ forskjellige ordnede utvalg fra A som gir oss B.
- Da må antall mengder B med k elementer være
$$\frac{{}^{n}P_{k}}{k!} = \frac{n!}{(n-k)! \cdot k!} = \binom{n}{k}$$

- Legg merke til at
$$\binom{n}{k} = \binom{n}{n-k}$$
- Hvorfor det?
- For eksempel, hvis vi har en mengde med 20 elementer, så er det $\binom{20}{18}$ delmengder med 18 elementer.
- For hver slik mengde, så har vi også en mengde med 2 elementer.
  - Vi kan f.eks. lage en funksjon som til enhver delmengde av størrelse 18 gir en delmengde av størrelse 2.
  - Denne funksjonen vil være både surjektiv og injektiv.
- Derfor har vi at $\binom{20}{18} = \binom{20}{2} = \frac{20 \cdot 19}{2 \cdot 1} = 190$.
- Antall delmengder av størrelse k må være lik antall delmengder av størrelse $n-k$.
- Det er like mange måter å velge n elementer på som det er å måter å *velge bort* n elementer på.

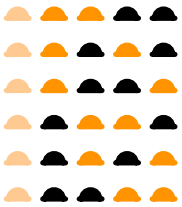
Binomialkoeffisientene

- Tallene $\binom{n}{k}$ kalles blant annet for *binomialkoeffisienter*.
- Følgende (rekursive) sammenheng var utgangspunktet for et tidligere induksjonsbevis:
$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$$

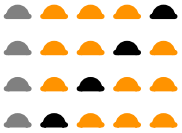
- Hvorfor er det slik? La oss se på et eksempel.
- Tre,  , av fem hatter,  , skal være oransje.
- Hvor mange måter kan dette gjøres på?

$$\binom{5}{3} = \binom{4}{2} + \binom{4}{3}$$

- Hvis den første hatten er oransje, så må to av de fire resterende hattene være oransje.Det er $\binom{4}{2} = 6$ måter å gjøre dette på.



- Hvis den første hatten er svart, så må *tre* av de fire resterende hattene være oransje. Det er $\binom{4}{3} = 4$ måter å gjøre dette på.



- Dette forteller oss at det er to ekvivalente måter å definere binomialkoeffisientene på:
  - Ved hjelp av fakultetsfunksjonen og brøk:

$$\binom{n}{k} = \frac{n!}{(n-k)! \cdot k!}$$

  - Ved rekursjon:

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$$

Pascals trekant

- Pascals trekant er en måte å skrive opp alle binomialkoeffisientene på ved hjelp av formelen

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$$

- Husk at

$$\binom{n}{0} = \binom{n}{n} = 1$$

- Vi får følgende bilde.

$$\begin{array}{ccccccccccccc}
 & & & & & & \binom{0}{0} & & & & & & \\
 & & & & & \binom{1}{0} & & \binom{1}{1} & & & & & \\
 & & & & \binom{2}{0} & & \binom{2}{1} & & \binom{2}{2} & & & & \\
 & & & \binom{3}{0} & & \binom{3}{1} & & \binom{3}{2} & & \binom{3}{3} & & & \\
 & & \binom{4}{0} & & \binom{4}{1} & & \binom{4}{2} & & \binom{4}{3} & & \binom{4}{4} & & \\
 & \binom{5}{0} & & \binom{5}{1} & & \binom{5}{2} & & \binom{5}{3} & & \binom{5}{4} & & \binom{5}{5} & \\
\binom{6}{0} & & \binom{6}{1} & & \binom{6}{2} & & \binom{6}{3} & & \binom{6}{4} & & \binom{6}{5} & & \binom{6}{6}
\end{array}$$

- Vi finner igjen mange kjente tallrekker i Pascals trekant

  - De naturlige tallene: 1,2,3,4,5,6,...
  - De såkalte triangulære tallene: 1,3,6,10,15,21,...
  - Toerpotensene: 1,2,4,8,16,32,...
  - Kvadrattallene: 1,4,9,16,25,...
  - Fibonacci-tallene (selv om de er godt gjemt): 1,1,2,3,5,8,13,21,...
  - Og mange flere...

1
1 1
1 2 1
1 3 3 1
1 4 6 4 1
1 5 10 10 5 1
1 6 15 20 15 6 1
1 7 21 35 35 21 7 1
1 8 28 56 70 56 28 8 1
1 9 36 84 126 126 84 36 9 1
1 10 45 120 210 252 210 120 45 10 1
1 11 55 165 330 462 462 330 165 55 11 1
1 12 66 220 495 792 924 792 495 220 66 12 1
1 13 78 286 715 1287 1716 1716 1287 715 286 78 13 1
1 14 91 364 1001 2002 3003 3432 3003 2002 1001 364 91 14 1
1 15 105 455 1365 3003 5005 6435 6435 5005 3003 1365 455 105 15 1

Tilbake til binomialkoeffisientene

- Nesten alle vet at $(a+b)^0 = 1$.
- Alle vet at $(a+b)^1 = a+b$.
- Mange vet at $(a+b)^2 = a^2 + 2ab + b^2$.
- De fleste kan regne ut at $(a+b)^3 = a^3 + 3a^2b + 3ab^2 + b^3$.
- Noen greier til og med å regne ut at $(a+b)^4 = a^4 + 4a^3b + 6a^2b^2 + 4ab^3 + b^4$.
- Noen bør begynne å ane at det er en sammenheng med Pascals trekant.
- Siden $(a+b)^5 = a^5 + 5a^4b + 10a^3b^2 + 10a^2b^3 + 5ab^4 + b^5$ blir den anelsen bekreftet.

Teorem (Generalisering av 1. kvadratsetning).
For alle tall a og b og alle hele tall $n \geqslant 0$ har vi

$$(a+b)^n = a^n + \binom{n}{1}a^{n-1}b^1 + \binom{n}{2}a^{n-2}b^2 + \cdots$$

$$\binom{n}{k}a^{n-k}b^k + \cdots + \binom{n}{n-1}ab^{n-1} + b^n$$

$$= \sum_{k=0}^{n}\binom{n}{k}a^{n-k}b^k$$

Bevis.

- $(a+b)^n = \underbrace{(a+b)\cdot(a+b)\cdots(a+b)}_{n \text{ forekomster}}$

- Hvis vi multipliserer dette ut, så får vi 2^n ledd.
- Hvert ledd består av n faktorer, hvor hver faktor er enten a eller b, f.eks. $abaa\cdots b$.
- Hvis $B \subseteq A = \{1, \ldots, n\}$, så lar vi B svare til leddet hvor faktor nummer i er b hvis $i \in B$ og a ellers.
- F.eks. vil $\{1, 4, 5\}$ svare til leddet $\underset{1}{b}\underset{2}{a}\underset{3}{a}\underset{4}{b}\underset{5}{b}\underset{6}{a}\cdots\underset{n}{a}$
- Hvert ledd kommer fra en og bare en mengde B; det vi har beskrevet er en surjektiv og injektiv funksjon fra potensmengden av A til leddene vi får når vi regner ut $(a+b)^n$.

Bevis (Fortsatt).

- Det fins $\binom{n}{k}$ delmengder av A med k elementer.
- Da fins det $\binom{n}{k}$ ledd med k b'er og $n-k$ a'er.
- Disse leddene ordnes til
$$\binom{n}{k} a^{n-k} b^k$$
- Dette er nøyaktig leddet med indeks k i teoremet.
- Siden k er vilkårlig må formelen i teoremet gi oss verdien på $(a+b)^n$.
- Dette avslutter beviset.

Oppsummering av regneprinsipper

- Ordnet utvalg med repetisjon: n^r
  - Hvor mange binære tall av lengde 5 fins det?
  - Det er $2^5 = 32$.
- Ordnet utvalg uten repetisjon: ${}^nP_r$
  - På hvor mange måter kan vi trekke to kort fra en kortstokk?
  - Det er ${}^{52}P_2 = 52 \cdot 51 = 2652$.
- Permutasjoner: $n!$
  - På hvor mange måter kan vi stokke om ordet LAKS?
  - Det er $4! = 4 \cdot 3 \cdot 2 \cdot 1 = 24$.
- Kombinasjoner: $\binom{n}{k}$
  - Hvor mange delmengder av $\{a, b, c, d, e\}$ har to elementer?
  - Det er $\binom{5}{2} = \dfrac{5 \cdot 4}{2 \cdot 1} = 10$
- Vi skal se på noen flere eksempler.

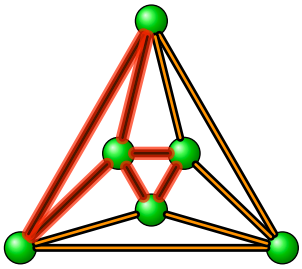
Først en liten digresjon om store tall

I kombinatorikk kommer vi fort opp i *veldig* store tall.

- Bredden til et hårstrå: 10^6 atomer.
- Atomer i en vanndråpe: 10^{21} atomer.
- Atomer i universet: 10^{80} atomer.
- Antall forfedre 265 generasjoner tilbake = antall atomer i universet.
- Og disse tallene er ganske små...
- Allikevel kan vi representere dem og regne på dem uten store problemer.

Grafteori

- Neste gang begynner vi med *grafteori*.
- En *graf* består av *noder* og *kanter*:



- Oppgave: klarer dere å tegne denne på et ark uten å løfte blyanten og uten å gå over en kant to ganger?

MAT1030 – Forelesning 22

Grafteori

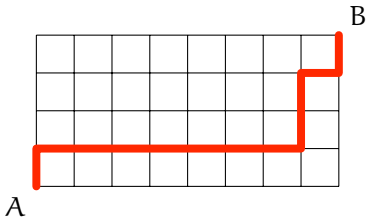
Dag Normann - 14. april 2010

Kombinatorikk

Oppsummering av regneprinsipper

- Ordnet utvalg med repetisjon: n^r
- Ordnet utvalg uten repetisjon: ${}^nP_r$
- Permutasjoner: $n!$
- Kombinasjoner: $\binom{n}{k}$
- Vi skal se på noen flere eksempler.

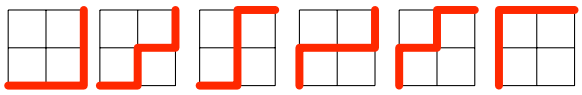
[fragile]Eksempel På hvor mange forskjellige måter kan vi gå fra A til B i dette 8 ganger 4-rutenettet? Vi må gå ett steg av gangen og kun oppover eller til høyre.



- Hvor mange slike stier er det?
- Denne stien kan representeres som $\uparrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\uparrow\uparrow\rightarrow\uparrow$.
- Dette er et ord på 12 tegn over alfabetet $\{\uparrow, \rightarrow\}$ hvor fire av tegnene er $\uparrow$ og åtte av tegnene er $\rightarrow$.
- Hvor mange slike ord er det? Det er $\binom{12}{4} = \frac{12 \cdot 11 \cdot 10 \cdot 9}{4 \cdot 3 \cdot 2 \cdot 1} = 495$

Eksempel

- Vi sjekker at det stemmer for 2 ganger 2-rutenettet:

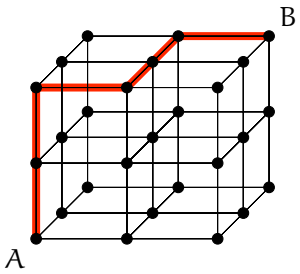


- Antall ord blir i dette tilfellet:

$$\binom{4}{2} = \frac{4 \cdot 3}{2 \cdot 1} = 6$$

som stemmer...

[fragile]Oppgave På hvor mange forskjellige måter kan vi gå fra A til B i denne $2 \times 2 \times 2$-kuben? Vi må gå ett steg av gangen og kun oppover, til høyre eller innover?



Eksempel

- (Dette ble også nevnt på forrige forelesning i et introduksjonseksempel om kuler og bokser.)
- På hvor mange ulike måter kan de 6 karakterene A til F gis til 10 studenter?
- Vi ikke er interessert i hvilken karakter en bestemt student får, men kun antallet av hver karakter i fordelingen.
- Det er 6 muligheter for hver av de 10 studentene.
- Kan vi bruke multiplikasjonsprinsippet og si at svaret er 6^{10}? `Nei`
- La oss lage 6 båser og putte studentene i hver sin bås avhengig av hvilken karakter de får.
- 2 studenter per karakter, og ingen stryk kan representeres slik:

- Fordelingen 123211 kan representeres slik:

- At alle stryker kan representeres slik:

- Hvor mange slike kombinasjoner fins det?
- Av 15 tegn må 5 være røde streker.
- Antall muligheter må være

$$\binom{15}{5} = \frac{15 \cdot 14 \cdot 13 \cdot 12 \cdot 11}{5 \cdot 4 \cdot 3 \cdot 2 \cdot 1} = 3003$$

Eksempel

- Mer generelt har vi

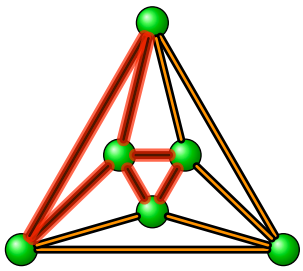
$$\binom{n+k-1}{k-1}$$

  som gir hvor mange forskjellige måter vi kan fordele n identiske elementer i k forskjellige beholdere på.
- Dette kalles også for uordnet utvalg *med repetisjon*.

Grafteori

Grafteori

- I går ga vi en smaksprøve på *grafteori*.
- Vi sa at en *graf* består av *noder* og *kanter*:

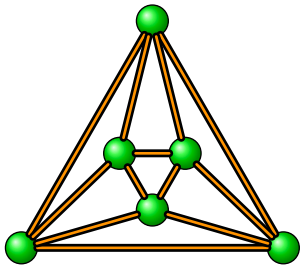


- Vi ga følgende oppgave: Klarer dere å tegne denne på et ark uten å løfte blyanten og uten å gå over en kant to ganger?

- Grafer fins overalt rundt oss!
- Grafteori er en viktig del av både *anvendt* og *teoretisk* matematikk.
- Grafteori brukes til å modellere problemer innenfor mange områder: informatikk, biologi, samfunnsvitenskap, lingvistikk, fysikk ...
- Mange problemer kan *representeres* ved å bruke grafer.
- Vi har møtt idéen om *representasjon* flere ganger allerede.
- Vi representerer f.eks. reelle tall ved hjelp av binære tall.
- Vi kan representere et matematisk problem som et annet som er enklere å løse.
- En representasjon gjør at vi kan se bort fra det som er irrelevant. Vi fanger inn *essensen*.
- Det er akkurat det som skjer i grafteori.

En graf

- En *graf* består av *noder* () og *kanter* ().
- Vi har til nå sett et eksempel på en graf:



- Klarte dere å tegne denne på et ark uten å løfte blyanten og uten å gå over en kant to ganger?
- Etter disse forelesningene i grafteori skal alle klare å besvare dette spørsmålet umiddelbart.
- Vi skal se at oppgaven er ekvivalent med å finne en såkalt *Eulersti*.

Søkealgoritmer for grafer

- Søkealgoritmer for grafer er et viktig tema.
- Kjente søkealgoritmer for grafer er f.eks.:
  - *Prims algoritme* for å finne et *minimalt spenntre* i en vektet graf. (Kruskals algoritme for samme problem.)
  - *Dijkstras algoritme* for å finne *minste avstand*, eller *korteste sti*, i en vektet graf.
- Vi kan søke bredde først eller dybde først
- For veldig mange grafproblemer har man ikke funnet *effektive* algoritmer.

Grafteori - noen eksempler

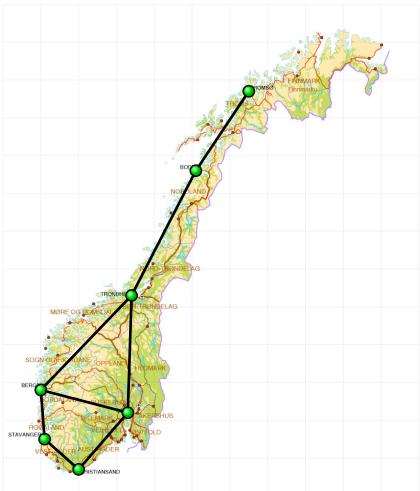
Eksempler på grafer

Grafer kan representere mange forskjellige ting.

- Nodene kan være studentene på Blindern, og kantene kan representere at to kjenner hverandre.
- Nodene kan være tilstandene som et dataprogram er i, og kantene kan representere overganger mellom tilstandene.
- En graf kan representere lenkestrukturen på et nettsted, hvor nodene er nettsider og kantene er lenker.
- Listen fortsetter: elektroniske kretser, molekyler i kjemi, datanettverk, analyse av nettverkstrafikk...
- Et *tre* er en spesiell type graf. Vi kommer til trær i kapittel 11.
- Vi skal se på flere eksempler på grafer før vi begynner med teorien.

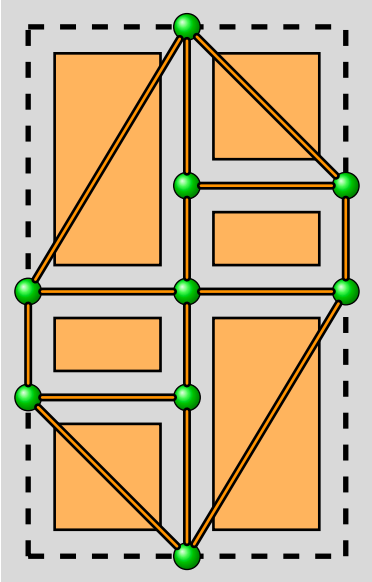
Kart som grafer

- Et kart kan gi utgangspunkt for flere forskjellige grafer.
- En mulighet er at
  - nodene representerer *byer*
  - kantene representerer *veier*
- En annen mulighet er at
  - nodene representerer *områder*, f.eks. fylker
  - kantene representerer *grenser*
- Når vi har representasjonen, så kan vi egentlig glemme det opprinnelige kartet.



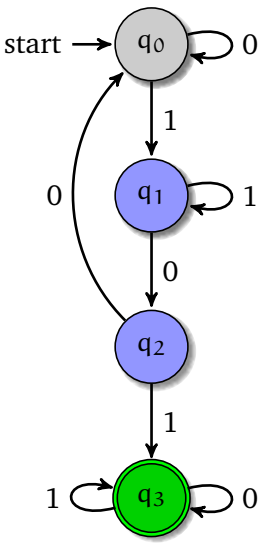
Veinett som grafer

- Et veinett kan representeres som en graf.
- Vi kan la hvert *kryss* svare til en node.
- Vi kan la *veiene* som forbinder krysse-ne svare til kantene.
- Når vi har tegnet opp grafen, så kan vi resonnere om den i stedet for om selve veinettet.



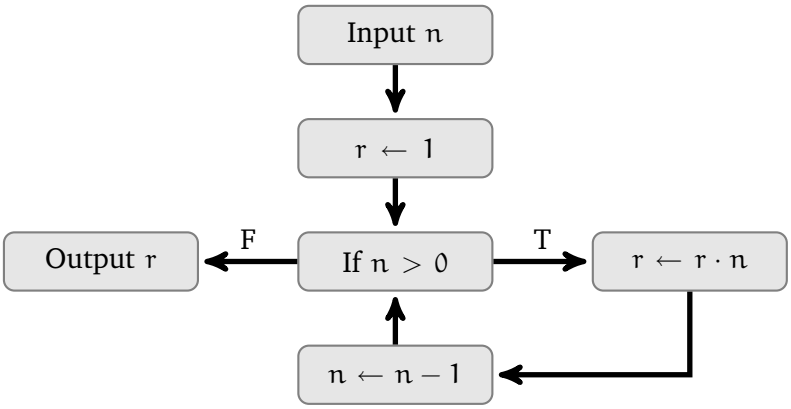
Endelige tilstandsmaskiner som grafer

- Vi kan tenke på *endelige tilstandsmaskiner* som grafer.
- Her kalles q_0, q_1, q_2 og q_3 *tilstander* og overgangene kalles *transisjoner*.
- Vi har en starttilstand, q_0 og en såkalt *aksepterende* tilstand, q_3.
- Hvis vi begynner med tallet 1101 og følger kantene etter hvert som vi leser siffer, så ender vi opp i q_3. Siden det er en aksepterende tilstand, er 1101 *akseptert*.
- Tallet 100 aksepteres ikke, siden vi ender opp i tilstand q_0, som ikke er aksepterende.
- Ser du hvilke tall som aksepteres?



Flytdiagrammer som grafer

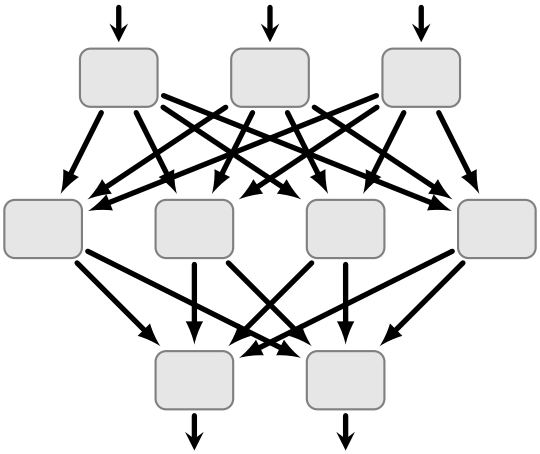
- Vi kan tenke på *flytdiagrammer* som grafer.
- I følgende eksempel representerer nodene programinstruksjoner.



- Hvilket program er dette?

Flytdiagrammer og multiplikasjonsprinsippet

- Multiplikasjonsprinsippet igjen.



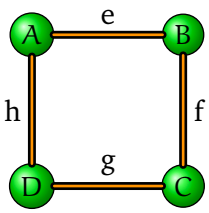
Grafteori - definisjoner og begreper

Grafteori - definisjoner og begreper

Definisjon (Graf).
En *graf* G består av en ikke-tom mengde *noder* V og en mengde *kanter* E, slik at enhver kant forbinder nøyaktig to noder med hverandre eller en node med seg selv.

- Dette er med vilje litt upresist.
- Vi presiser heller etter hvert når vi trenger det.
- På engelsk brukes begrepene
  - *vertex/vertices* om noder, og
  - *edges* om kanter.

- Vi tegner noder slik: 
- og kanter slik:
- Det er ikke viktig akkurat *hvordan* vi tegner grafer; det er *strukturen* i graf som er viktig, hvilke noder som er forbundet med hvilke via en kant.



Her er A, B, C og D noder, mens e, f, g og h er kanter.

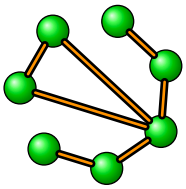
Definisjon (Inntil/naboer).
En kant ligger *inntil* (engelsk: *incident*) nodene som forbindes av den. To noder er *naboer* (engelsk: *adjacent*) hvis de forbindes av en kant.

- Kanten e ligger inntil nodene A og B.
- Nodene B og C er naboer, siden de forbindes av kanten f.

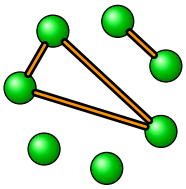
Sammenhengende grafer

En graf trenger ikke å være *sammenhengende*.

Definisjon (Sammenhengende).
En graf er *sammenhengende* (engelsk: *connected*) hvis det er mulig å komme fra enhver node til enhver annen node ved å følge kantene.



En *sammenhengende* graf.



En *usammenhengende* graf.

Tomme grafer og løkker

En graf trenger ikke å ha noen kanter, men den må ha minst én node. Grafer uten kanter kalles *nullgrafer* eller *tomme grafer* (engelsk: *null graph*).

En tom graf.

En graf kan ha *løkker* (engelsk: *loop*), en kant som går fra en node til den samme noden.

En graf med en løkke.

Parallelle kanter og enkle grafer

En graf kan ha *parallelle* kanter, to eller flere kanter som forbinder de samme to nodene.

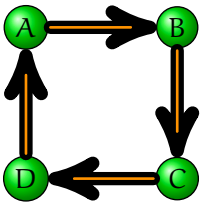
En graf med parallelle kanter.

Definisjon (Enkel).
En graf er *enkel* (engelsk: *simple*) hvis den ikke har løkker eller parallelle kanter.

- Det er ganske vanlig å definere grafer slik at løkker og parallelle kanter ikke forekommer.

Rettede grafer

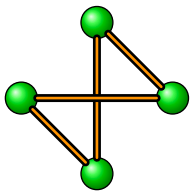
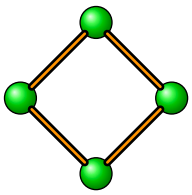
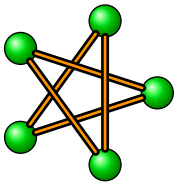
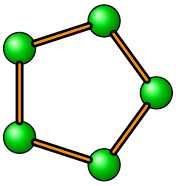
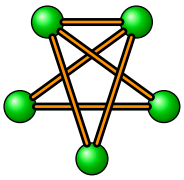
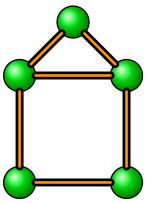
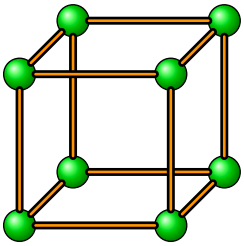
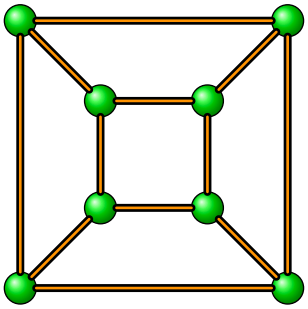
Definisjon.
En *rettet graf* (engelsk: *directed*) er en graf hvor hver kant har en retning.



- En *relasjon* kan ses på som en rettet graf.
- Denne grafen svarer til relasjonen $\{\langle A, B\rangle, \langle B, C\rangle, \langle C, D\rangle, \langle D, A\rangle\}$.
- Hvis vi har en *symmetrisk* relasjon, riktignok, så kan vi tenke på denne som en vanlig (urettet) graf.
- Foreløpig skal vi ikke snakke om rettede grafer.

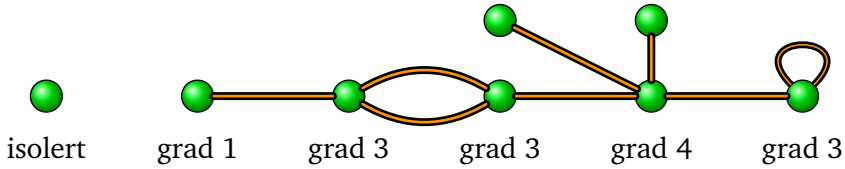
Måter å tegne opp grafer på

- Det fins ikke noen unik måte å tegne opp en graf på.
- Vi bryr oss for eksempel ikke om hvor lange kantene er, om de er bøyd, etc.
- Det eneste som spiller noen rolle er om to noder er forbundet med en kant.
- La oss se på noen eksempler. Følgende par av grafer er identiske, men tegnet opp på forskjellige måter.
- Vi forestiller oss at kantene er elastiske og at vi flytter om på nodene.
- Vi skal etter hvert presisere dette gjennom begrepet *isomorfi*. Følgende par av grafer kalles *isomorfe*.

Graden til noder

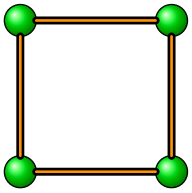
Definisjon (Grad).
Graden (engelsk: *degree*) til en node v er antall kanter som ligger inntil v. En løkke teller som to kanter. Med $\deg(v)$ mener vi graden til v. En node med grad 0 kalles *isolert*.



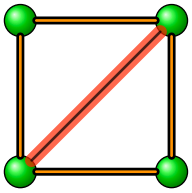
Teorem.
Summen av gradene til alle nodene i en graf er lik 2 ganger antallet kanter. Hvis V er mengden av noder og E er mengden av kanter, så har vi

$$\sum_{v \in V} \deg(v) = 2|E|.$$

- Hver *kant* som legges til i en graf vil øke summen av gradene med to.
- La oss se på et eksempel.



Antall kanter er 4.
Summen av gradene er 8.

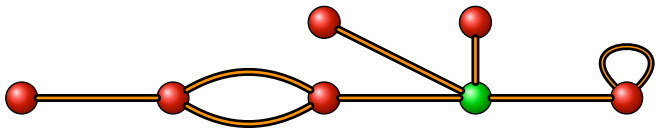


Antall kanter er 5.
Summen av gradene er 10.

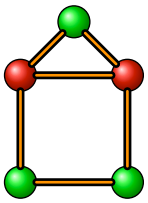
Bevis.
Hvis vil legger sammen gradene til alle nodene, så vil hver kant telle to ganger, siden hver kant ligger inntil to noder.

Håndhilselemmaet

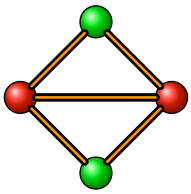
Lemma (håndhilselemmaet).
Det er alltid et partall antall noder av odde grad i en graf.



Her er det 6 noder (markert med rødt) med odde grad.



Her er det 2 noder av odde grad.



Her er det 2 noder av odde grad.

- Hvis vi forestiller oss mange mennesker samlet i et rom og at man håndhilser på hverandre, så må antallet av de som håndhilser på et odde antall personer være et partall.
- Vi kan representere denne situasjonen ved å representere menneskene som noder. En kant vil da representere at to personer håndhilser på hverandre.
- Det kalles et lemma fordi det ikke er så interessant i seg selv, men er nyttig for å bevise andre lemmaer og teoremer.
- Vi skal nå bevise håndhilselemmaet.

Bevis (håndhilselemmaet).
La G være en graf. Vi deler mengden V av noder inn to: V_o er de som har odde grad (de som var røde) og V_p er de som har lik grad (de som var grønne). Vi har vist et teorem som sier at summene av gradene til *alle* nodene er to ganger antall kanter.

$$\sum_{v \in V} \deg(v) = 2|E|$$

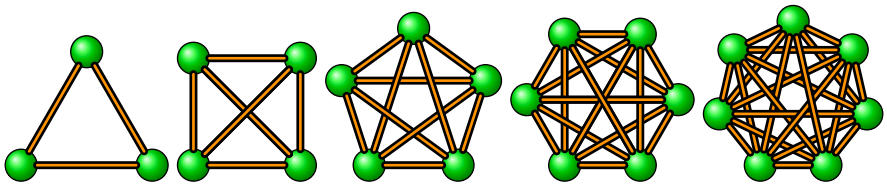
Siden vi har delt opp mengden av noder i to, kan vi skrive dette slik:

$$\sum_{v \in V_o} \deg(v) + \sum_{v \in V_p} \deg(v) = 2|E|$$

Siden $2|E|$ er et partall og summen av gradene til nodene i V_p er et partall, så må summen av gradene til nodene i V_o også være et partall. Siden hver node i V_o har odde grad, så må det være et partall antall av dem.

Komplette grafer

Definisjon (Komplett graf).
En enkel graf er *komplett* hvis hver node er nabo med enhver annen node.



Komplette grafer (K_3, K_4, K_5, K_6, K_7).

- Hvor mange kanter er det i en komplett graf?
- K_3 har 3 kanter. K_4 har 6 kanter. K_5 har 10 kanter. K_6 har 15 kanter. K_7 har 21 kanter. Er det noen som ser et mønster?

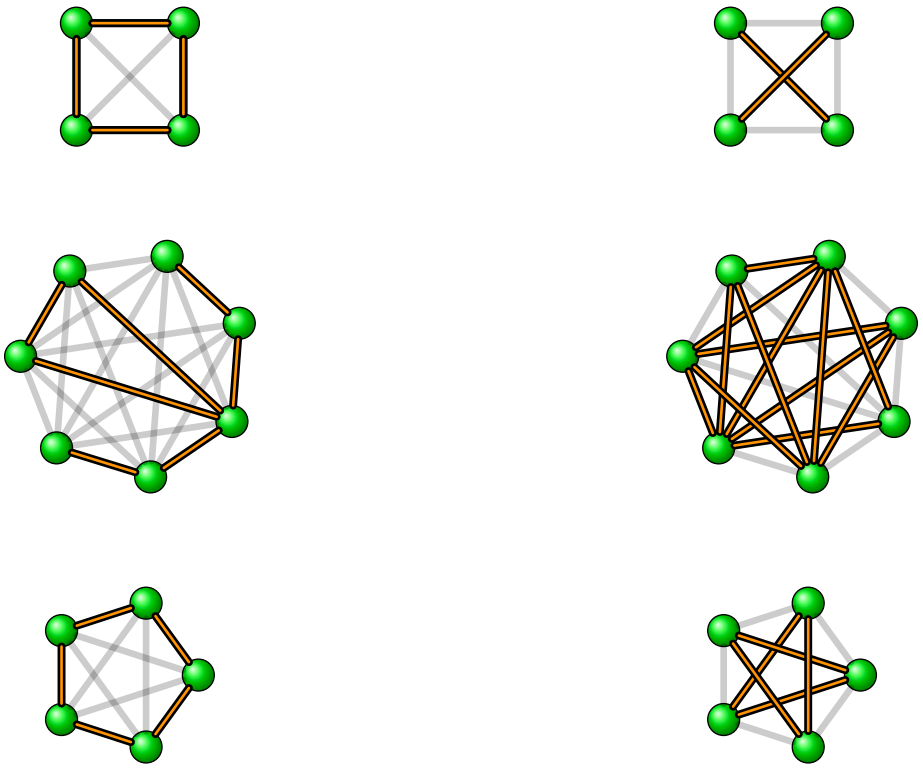
Teorem.
Det er $\binom{n}{2}$ kanter i en komplett graf med n noder.

- Det er fordi vi har en kant for hver mengde av noder med kardinalitet 2.
- K_n representerer kampoppsettet i en enkel serie som omfatter n lag.
- For det vanlige oppsettet med hjemme- og bortekamper vil kampoppsettet representeres av en komplett rettet graf.

Komplementet til en graf

Definisjon (Komplement).
La G være en enkel graf. Da er *komplementet* til G grafen som har de samme nodene som G, men hvor to noder er naboer hvis og bare hvis nodene *ikke* er naboer i G. Vi skriver $\overline{G}$ for komplementet til G.

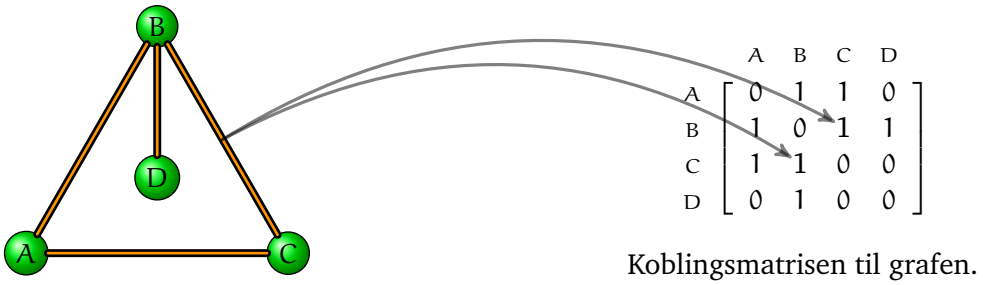
Vi skal se på noen grafer og deres komplementer.



- I det siste tilfellet fikk vi ikke noen *ny* graf når vi tok komplementet.
- Slike grafer kalles *selv-komplementære*.

Matriserepresentasjoner

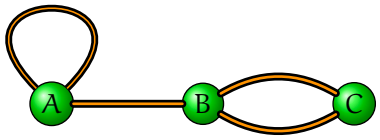
På samme måte som med relasjoner, så har grafer en matriserepresentasjon. Vi kaller en slik matrise for en *koblingsmatrise* (engelsk: *adjacency matrix*).



Koblingsmatrisen til grafen.

Definisjon (Koblingsmatrise).
Hvis G er en graf med n noder, $v_1, \ldots, v_n$, så er koblingsmatrisen til G en n-ganger-n-matrise hvor tallet i rad i og kolonne j er antall kanter mellom v_i og v_j.

$$\begin{array}{c} \\ A \\ B \\ C \end{array} \begin{array}{c} \begin{array}{ccc} A & B & C \end{array} \\ \begin{bmatrix} 1 & 1 & 0 \\ 1 & 0 & 2 \\ 0 & 2 & 0 \end{bmatrix} \end{array}$$



Koblingsmatrisen til grafen.

- Legg merke til at vi kan speile en matrise om diagonalen.
- Det er fordi vi kun ser på *urettede* grafer.
- Hvis vi ser på rettede grafer, så kan vi ikke speile matrisen om diagonalen.
- Vi kunne også speile om diagonalen for *symmetriske relasjoner*.
- En forskjell mellom symmetriske relasjoner og grafer er at vi tillater *parallelle* kanter i grafene.
- De kan vi ikke fange inn ved hjelp av en relasjon.
- Det fins flere matriser for samme graf, avhengig av rekkefølgen vi gir nodene i.

MAT1030 – Forelesning 23

Grafteori

Dag Normann - 20. april 2010

Grafteori

Repetisjon og mer motivasjon

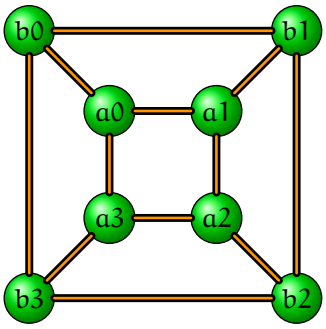
- Først litt repetisjon
- En graf består av *noder* og *kanter*
- Kanter ligger *inntil* noder, og noder kan være *naboer*.
- Vi bør kjenne til begrepene om *sammenhengende* grafer, *tomme* grafer, *løkker*, *parallelle* kanter, *enkle* grafer og *komplette* grafer.
- Hver node har en *grad*.
  - Summen av gradene til alle nodene i en graf er lik 2 ganger antallet kanter.
  - Håndhilselemmaet: Det er alltid et partall antall noder av odde grad i en graf.
- Vi skal kjenne til *komplementet* av en graf og *matriserepresentasjoner*.

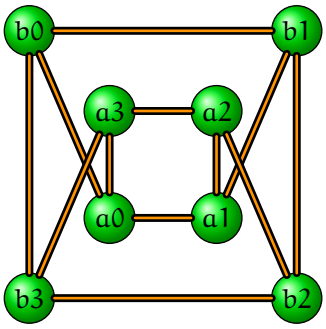
- Grafer kan brukes til å representere omtrent alt som fins av relasjoner.
- Mange algoritmer/egenskaper kan forstås bedre ved å bruke grafer.
- Ofte er løsningen å kunne identifisere et problem som et *grafteoretisk* problem.

Grafisomorfier

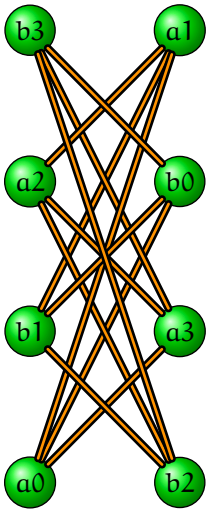
- Vi skal nå møte begrepet *isomorfi* for første gang i dette kurset.
- Isomorfibegrepet er veldig generelt og brukes overalt i matematikk.
- Ordet kommer fra gresk og betyr formlik (iso = lik, morf = form).
- Isomorfe objekter skal ha samme *form*, men kan ha ulikt innhold.
- En isomorfi er en funksjon som er injektiv og surjektiv – det er altså en en-til-en-korrespondanse – og som bevarer bestemte egenskaper.
- Intuitivt, så sier vi at to matematiske objekter er *isomorfe* hvis de er “strukturelt like”.
- Hvis vi har tegnet opp to grafer og kan komme fra den ene grafen til den andre ved kun å flytte på noder og endre på lengdene på kantene, så er grafene *isomorfe*.
- Vi har ikke lov til å legge til eller ta bort noder eller kanter, eller dele kanter eller noder i to...

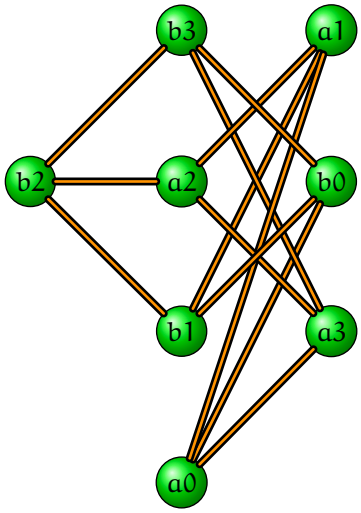
- Følgende to grafer er isomorfe.

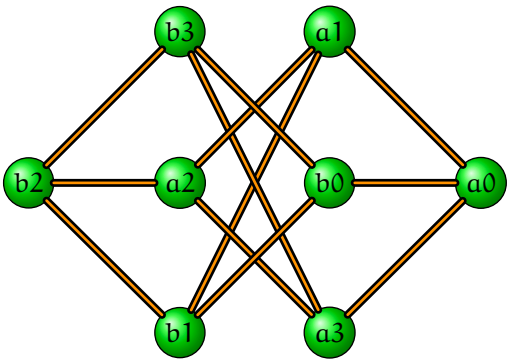


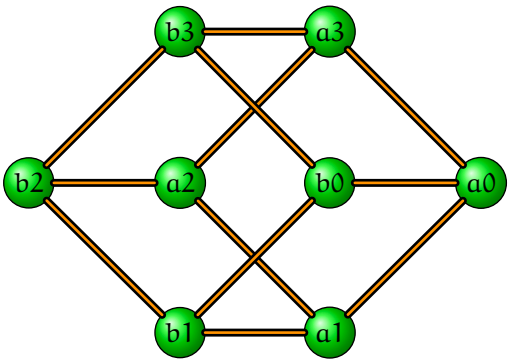


- Disse to grafene er også isomorfe med følgende grafer.

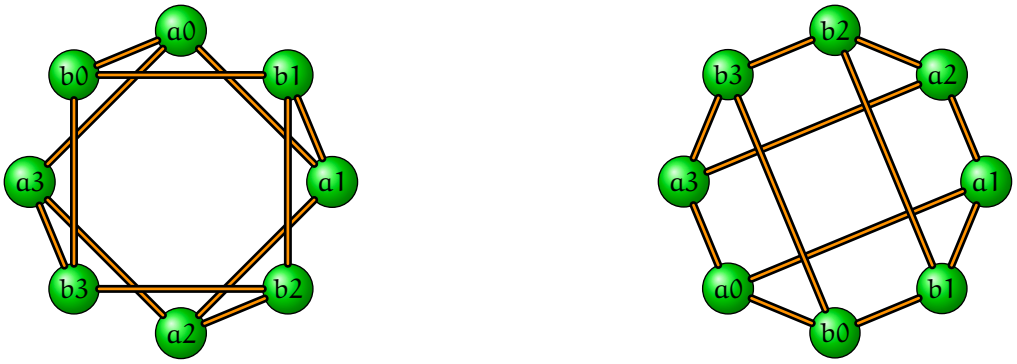








- De neste par grafene er også er isomorfe med disse.



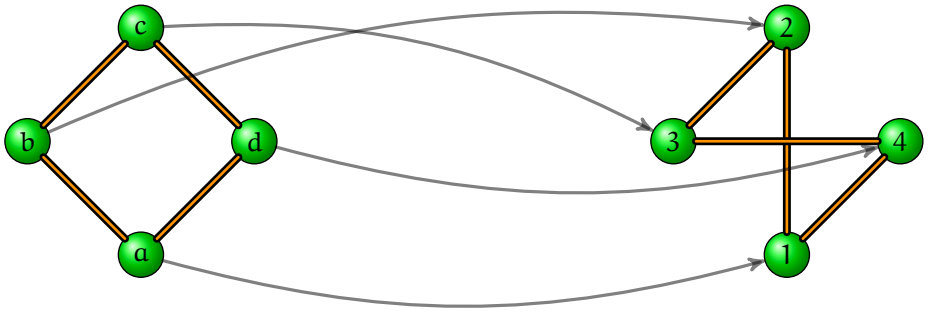
- Vi skal nå gjøre isomorfibegrepet helt presist.
- Hvis G og H er to grafer, så er en isomorfi mellom grafene en funksjon fra nodene i G til nodene i H som oppfyller bestemte egenskaper.

Definisjon (Isomorfi).
La G og H være to enkle grafer slik at $V(G)$ er mengden av noder i G og $V(H)$ er mengden av noder i H. En *isomorfi* fra G til H er en funksjon $f : V(G) \to V(H)$ med følgende egenskaper:

- f er surjektiv og injektiv
- Nodene u og v er naboer i G *hvis og bare hvis* nodene $f(u)$ og $f(v)$ er naboer i H.

Eksempel.

- La grafen G bestå av nodene $\{a, b, c, d\}$ og kantene $\{ab, bc, cd, da\}$.
- La grafen H bestå av nodene $\{1, 2, 3, 4\}$ og kantene $\{14, 34, 12, 32\}$.
- Funksjonen f slik at $f(a) = 1$, $f(b) = 2$, $f(c) = 3$ og $f(d) = 4$ er en isomorfi.
- F.eks. ser vi at a og b er naboer i G (siden ab er en kant).
- Da må $f(a)$ og $f(b)$, som er 1 og 2, være naboer i H.
- Det stemmer, siden 12 er en kant i H.



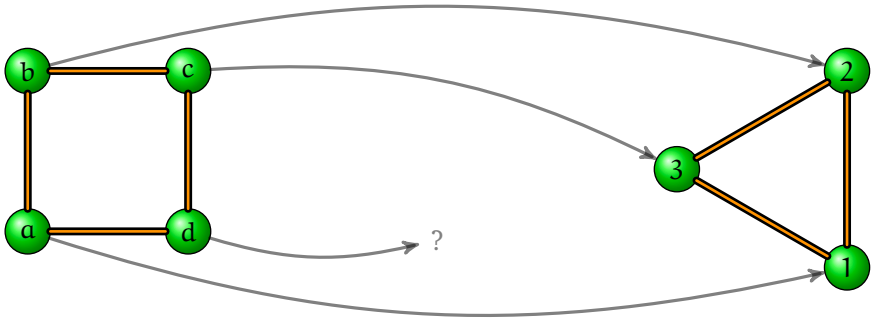
Merk.

- Det står "*hvis og bare hvis*" i definisjonen:
- Nodene u og v er naboer i G *hvis og bare hvis* nodene $f(u)$ og $f(v)$ er naboer i H.
- *Hvis*-delen av definisjonen er denne påstanden:
  - Nodene u og v er naboer i G *hvis* nodene $f(u)$ og $f(v)$ er naboer i H.
  - Det betyr at hvis nodene u og v *ikke* er naboer, så er heller ikke nodene $f(u)$ og $f(v)$ naboer.
  - "Naboer i $H \rightarrow$ Naboer i G"
- *Bare hvis*-delen av definisjonen er denne påstanden:
  - Nodene u og v er naboer *bare hvis* nodene $f(u)$ og $f(v)$ naboer.
  - Det betyr at hvis nodene u og v er naboer, så er også nodene $f(u)$ og $f(v)$ naboer.
  - "Naboer i $G \rightarrow$ Naboer i H"

- Vi har definert isomorfi mellom enkle grafer.
- Vi kan utvide definisjonen til å gjelde endelige grafer generelt.
- Da krever vi at antall kanter mellom u og v i G skal være det samme som antall kanter mellom $f(u)$ og $f(v)$ i H.
- For enkle grafer kan dette antallet være 0 (om u og v ikke er naboer) og 1 (om u og v er naboer).
- For å vise at to grafer er isomorfe, må man gi en funksjon og argumentere for at funksjonen har egenskapene som er nødvendige.
- Det fins per i dag ingen effektiv algoritme for å avgjøre om to grafer er isomorfe.
- For å vise at to grafer *ikke* er isomorfe, så er det tilstrekkelig å finne en "grafteoretisk egenskap" som kun den ene av grafene har.
- En grafteoretisk egenskap er en egenskap som bevares under "lovlige" tranformasjoner, som å flytte rundt på nodene, gjøre kantene lengre/kortere, etc.
- Noen av de enkleste grafteoretiske egenskapene er f.eks.:
  - Hvor mange noder en graf har.
  - Hvor mange kanter en graf har.

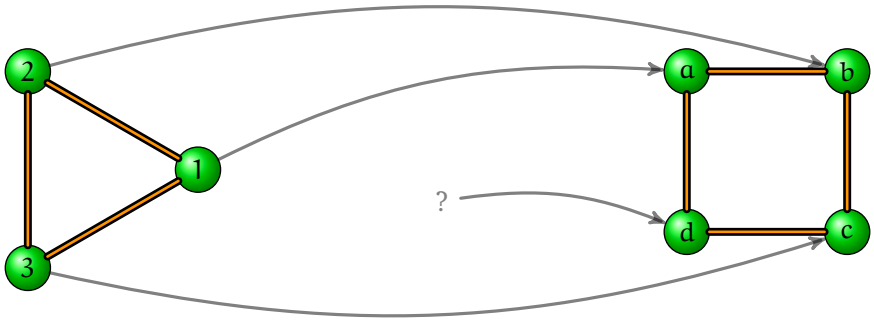
– Hvor mange noder av en bestemt grad en graf har.
– Korteste avstand mellom to noder.

- Er følgende grafer isomorfe?



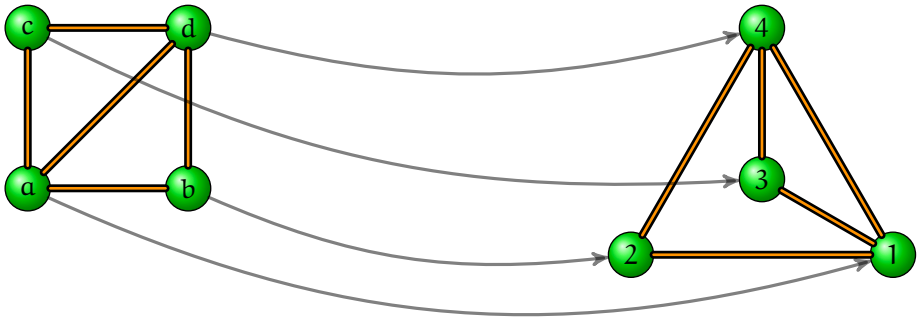
- Nei, de er ikke isomorfe.
- Grafen til høyre har *færre* noder enn grafen til venstre, så ingen funksjon fra den venstre grafen til den høyre kan være *injektiv* eller *en-til-en*.

- Er følgende grafer isomorfe?



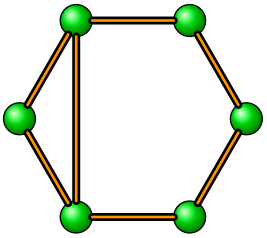
- Nei, de er ikke isomorfe.
- Grafen til høyre har *flere* noder enn grafen til venstre, så ingen funksjon fra den venstre grafen til den høyre kan være *surjektiv* eller *på*.

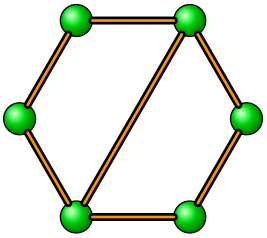
- Er følgende grafer, G og H, isomorfe?



- Ja, de er isomorfe.
- Funksjonen $f : V(G) \to V(H)$ gitt ved $f(a) = 1$, $f(b) = 2$, $f(c) = 3$ og $f(d) = 4$ er en isomorfi.
- Vi ser at u og v er naboer i G hvis og bare hvis $f(u)$ og $f(v)$ er naboer i H.

- Er følgende grafer isomorfe?





- Nei, de er ikke isomorfe.
- Grafen til venstre inneholder tre noder som alle er relatert til hverandre; det gjør ikke grafen til høyre.

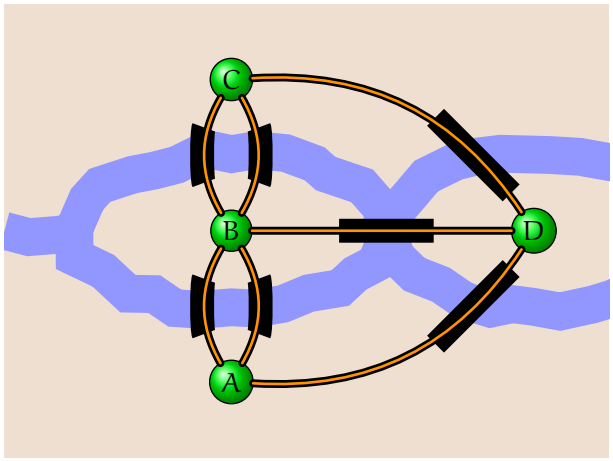
Noen kommentarer

- Å avgjøre om to grafer er isomorfe er i bunn og grunn å finne ut om det er den samme grafen man har å gjøre med.
- Hvis en rekke grafer er gitt, så ønsker vi å finne ut om noen av dem er isomorfe for å unngå å gjøre overflødig arbeid.
- Det er ingen som har klart å lage en *effektiv* algoritme (polynomiell tid) for å avgjøre om grafer er isomorfe (i det generelle tilfellet).
- Mange spesialtilfeller, f.eks. trær, vet man mye om.
- I praksis så klarer man å lage ganske effektive algoritmer allikevel, men i *verste* tilfelle må man backtracke over alle $n!$ mulige omdøpinger av nodene.
- Å finne isomorfier fra en graf til seg selv er en måte å avdekke *symmetrier* på.
- Å finne ut om en graf er en *delgraf* av en annen er et annet, men relatert, problem. (Ofte vanskeligere.)

Stier og kretser

- Vårt neste tema er *stier* (engelsk: *path*) og *kretser* (engelsk: *circuit*).
- Vi skal begynne med det klassiske eksemplet om *Königsbergs broer*.
- Kort fortalt har vi sju broer som forbinder fire landområder.
- Spørsmålet er om det går an å gå en tur i Königsberg slik at man går over hver av de sju broene *nøyaktig* én gang.
- Dette er kjent for å ha blitt løst av Leonhard Euler omkring 1735.
- Vi skal se at oppgaven er den samme som å finne en *Eulersti* i grafen som representer Königsberg.

Königsbergs broer



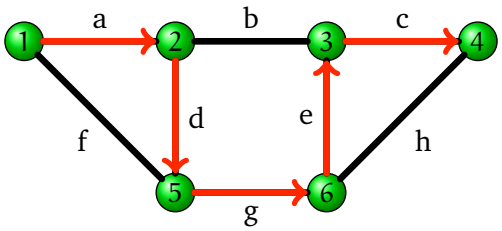
Vi representerer situasjonen med en graf. Spørsmålet blir nå om det er mulig å gå over alle kantene nøyaktig en gang.

Definisjon (Sti).
En *sti* av lengde n i en graf er sekvens av noder og kanter på formen

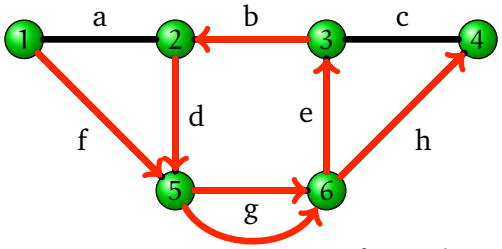
$$v_0 e_1 v_1 e_2 v_2 \dots e_n v_n$$

hvor e_i er en kant som forbinder v_{i-1} og v_i for $i \in \{1, 2, \dots, n\}$. En sti hvor $v_0 = v_n$ kalles en *krets*. Vi sier at sekvensen er en sti fra v_0 til v_n.

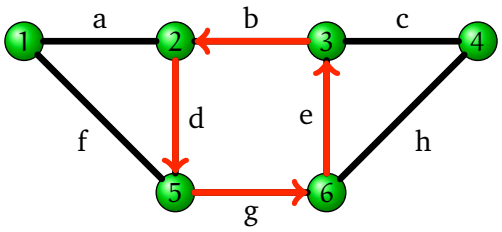
- En sti kalles også for en *vei*.
- *Lengden* til en sti er det samme som antall kanter i stien.
- En enkelt node er både en sti og en krets av lengde 0.
- Hvis grafen er *enkel*, tillater vi oss å skrive $v_0 v_1 v_2 \dots v_n$ for stien.
- Vær oppmerksom på at terminologien for grafteori varierer fra lærebok til lærebok.
- Vi ser på noen eksempler.



1a2d5g6e3c4 er en sti fra 1 til 4.

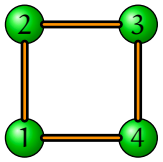


1f5g6e3b2d5g6h4 er en sti fra 1 til 4.



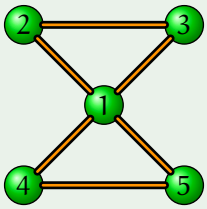
5g6e3b2d5 er en krets som begynner og slutter i 5.

- Hvis en graf er enkel fins det ingen parallelle kanter, og da kan vi betegne en sti som en sekvens av noder.
- Det er vanlig å identifisere kretser som kun er forskjellige med hensyn på startnode eller rekkefølge.



- Her er 12341 en krets.
- Vi identifiserer denne med kretsene 23412, 34123, etc., og 43214, 32143, etc.

Oppgave.
Hvor mange *forskjellige* kretser som inneholder alle kantene nøyaktig én gang fins i følgende graf?



- Vi kan definere mengder av stier *induktivt* på følgende måte.

Definisjon (Mengden av stier - induktivt).
- En sekvens som består av en node v er en sti.

- Hvis p er en sti, og det siste elementet i p er noden u, og det går en kant fra u til v, så er sekvensen pev en sti.
- Mengden av stier er den minste mengden som oppfyller disse to kravene.

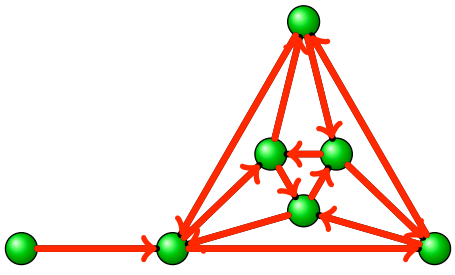
- Når vi nå har begrepet om en sti kan vi definere *sammenhengende grafer* mer presist.

Definisjon (Sammenghengende).
En graf er *sammenhengende* hvis det for hvert par av noder u og v fins en sti fra u til v.

- Vi kan definere en ekvivalensrelasjon R på mengden av noder i en graf ved å si at uRv skal holde hvis det fins en sti fra u til v.
- Det er lett å sjekke at R er transitiv, refleksiv og symmetrisk.
  (Vi tegner og forklarer på tavla.)
- Ekvivalensklassene definert av R kalles for *komponentene* til grafen.
- En graf kan deles opp i "sammenhengende delgrafer" på en slik måte.
- En sammenhengende graf er en graf som består av en komponent.

Definisjon (Eulersti/Eulerkrets).
La G være en sammenhengende graf. En *Eulersti* er en sti som inneholder hver kant fra G nøyaktig én gang. En *Eulerkrets* er en Eulersti hvor den første og den siste noden sammenfaller. En sammenhengende graf som har en Eulerkrets kalles *Eulersk*. En sammenhengende graf som har en Eulersti, men *ikke* en Eulerkrets, kalles *semi-Eulersk*.

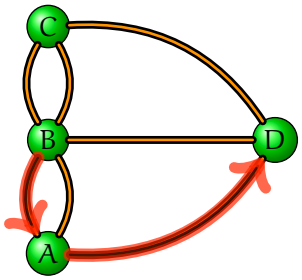
- Noen kommentarer er på sin plass.
- I en Eulersti har vi lov til å gjenta noder, men ikke kanter.
- Eulerstier kalles også for *Eulerveier*.
- Finner vi en Eulersti i denne grafen?
- Vi må i hvert fall begynne eller slutte i noden helt til venstre, siden den har grad 1.



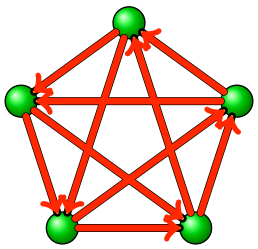
- Vi fant en Eulersti.
- En Eulerkrets er riktignok umulig, på grunn av noden til venstre.
- Ved å ta bort denne, så får vi en Eulerkrets.

Tilbake til Königsberg

- Problemet om Königsbergs broer blir nå: Har grafen over Königsberg en Eulersti?
- En sti må gå innom noden A minst en gang.
- Hvis vi går inn i A og ut igjen, så gjenstår én kant.
- For å gå over alle tre kantene som ligger inntil A, så må man enten begynne eller slutte i A.
- Det samme gjelder for C, og D og B, siden de alle har *odde grad*.



- Vi ser om vi finner en Eulerkrets i den komplette grafen med fem noder.

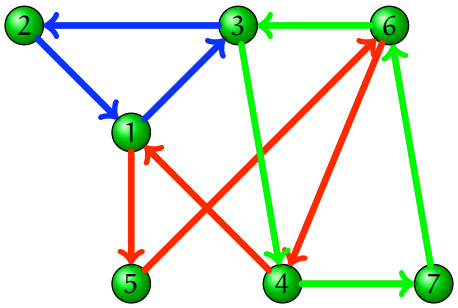


- Det var greit.
- Vi observerer at hver node har grad 4, et partall.
- Vi skal nå se at det er en sammenheng mellom eksistensen av Eulerstier/-kretser og hvorvidt gradene til nodene er partall.
- Anta at vi har en graf med en Eulerkrets.
- Hva er da gradene til nodene i grafen?
- Vi kan skrive Eulerkretsen som sekvensen $v_0e_1v_1e_2v_2 \dots e_nv_n$, hvor $v_n = v_0$.
- Graden til en node u må være 2 ganger antall ganger den forekommer i sekvensen.
- Enhver node i grafen må da ha grad lik et partall.
- Vårt neste teorem sier det omvendte, nemlig at hvis hver node i en graf har grad som er et partall, så må grafen inneholde en Eulerkrets.

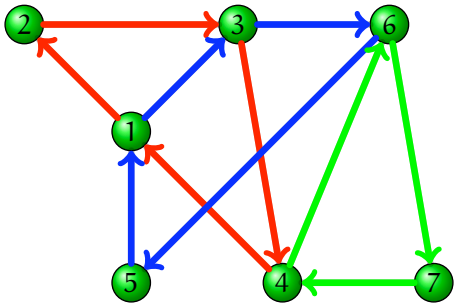
Teorem.
La G være en sammenhengende graf.

1. Hvis graden til enhver node i G er et partall, så inneholder G en Eulerkrets.
2. Hvis nøyaktig to noder i G har *odde* grad, så inneholder G en Eulersti som begynner i en node av odde grad og som slutter i en node av odde grad.
3. Hvis G har mer enn to noder av odde grad, så inneholder grafen *ikke* en Eulersti.

- Boka beviser påstand 1 ved å gi en algoritme for å konstruere en Eulerkrets for en graf hvor hver node har et partall som grad.
- Vi må da argumentere for at algoritmen er *korrekt*, at den alltid vil finne en Eulerkrets.
- I dette tilfellet er det ganske greit å se.
- Vi skal se at påstand 1 medfører påstand 2, at en graf som inneholder nøyaktig to noder av odde grad inneholder en Eulersti.
- La oss se på intuisjonen bak algoritmen.



- Hvis vi begynner i node 2, så finner vi kretsen 2132.
- Siden ubrukte kanter ligger inntil både node 1 og 3, forsøker vi å *utvide* vår nåværende krets ved å legge til nye kretser.
- Hvis vi begynner i node 1, så finner vi kretsen 15641.
- Vi kan *sette sammen* disse kretsene og få kretsen 21564132.
- Vi finner en krets til, 47634.
- Vi får da kretsen 215647634132 som er en Eulerkrets.
- Her er en annen måte å sette sammen flere kretser til en Eulerkrets på, med den samme grafen.

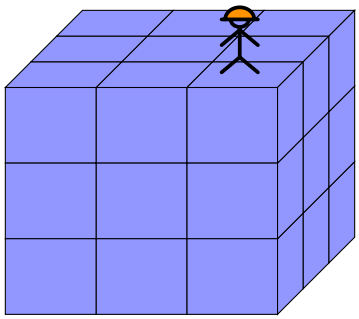


- Hvis vi begynner med en sti som kun inneholder én node og utvider stien stegvis ved å legge til kanter og stier, så må vi før eller siden komme tilbake til den første noden i stien, slik at vi får en *krets*.
- Grunnen er at hver node har grad som er et partall.
- Hver gang vi "går inn i" en node i stien, som ikke er startnoden, så må finnes en ubrukt kant slik at vi kan "gå ut" igjen.
- Før eller siden må vi komme tilbake til startnoden.

1. Input en Eulergraf G med noder V og kanter E
2. krets ← en node fra V

3. **While** $E \neq \emptyset$ **do**
   3.1. $i \leftarrow$ den første noden i krets med en kant fra E som ligger inntil i
   3.2. $v \leftarrow i$; $nykrets \leftarrow i$
   3.3. **Repeat**
      3.3.1. $e \leftarrow$ en kant fra E som ligger inntil v
      3.3.2. $v \leftarrow$ noden som er nabo med v via e
      3.3.3. $nykrets \leftarrow$ sammensetningen av $nykrets$ og e og v
      3.3.4. $E \leftarrow E - \{e\}$
      until ingen kant fra E ligger inntil v
   3.4. krets $\leftarrow$ sammensetningen av krets før i, $nykrets$ og krets etter i
4. Output krets

En nøtt om en maur og en kube



- Anta at en maur sitter på utsiden av en 3x3x3-kube.
- Mauren spiser seg inn i den første lille kuben (av i 27 kuber).
- Er det mulig for mauren å spise seg gjennom alle kubene og så komme ut igjen samme sted?
- Mauren har kun lov til å gå en kube av gangen og ikke på skrå.

MAT1030 – Forelesning 24

Grafer og trær

Dag Normann - 21. april 2010

Grafteori

Oppsummering

- Vi har sett på isomorfibegrepet for grafer.
- To grafer er isomorfe hvis alle de viktige egenskapene er de samme.
- Mer presist:

  > Det fins en bijeksjon mellom nodene og mellom kantene slik at bildet av en kant går mellom bildet av to noder hvis og bare hvis kanten går mellom nodene.

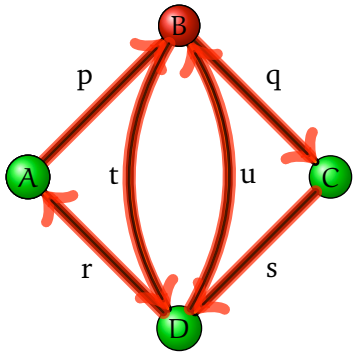
- Vi definerte stier og kretser
- En sti er en følge av noder og kanter slik at vi går fra node til node via kantene mellom dem.
- En krets er en sti som begynner og slutter samme sted.
- To kretser er like uavhengig av hvor vi starter kretsen som sti, og uavhengig av retningen vi oppgir for stien.
- For en presis definisjon trenger vi å bruke en ekvivalensrelasjon på mengden av stier.
- En Eulerkrets er en krets som inneholder hver kant nøyaktig én gang.
- En Eulersti er en sti med samme egenskap.
- En sammenhengende graf har en Eulerkrets hvis graden til alle nodene er et partall.
  En slik graf kalles en Eulergraf.
- En sammenhengende graf har en Eulersti hvis høyst to noder har et oddetall som grad.
  En graf som har to noder med odde grad er semi-Euler.
- Vi beskrev en pseudokode for å finne en Eulerkrets i en Eulergraf.
- I dag skal vi gi et fullstendig bevis for teoremet om Eulergrafer, men først skal vi repetere pseudokoden:

1. Input en Eulergraf G med noder V og kanter E
2. $krets \leftarrow$ en node fra V
3. **While** $E \neq \emptyset$ **do**
   - 3.1. $i \leftarrow$ den første noden i $krets$ med en kant fra E som ligger inntil i
   - 3.2. $v \leftarrow i$; $nykrets \leftarrow i$
   - 3.3. **Repeat**
     - 3.3.1. $e \leftarrow$ en kant fra E som ligger inntil v
     - 3.3.2. $v \leftarrow$ noden som er nabo med v via e
     - 3.3.3. $nykrets \leftarrow$ sammensetningen av $nykrets$ og e og v
     - 3.3.4. $E \leftarrow E - \{e\}$

until ingen kant fra E ligger inntil v

3.4. krets $\leftarrow$ sammensetningen av krets før i, nykrets og krets etter i

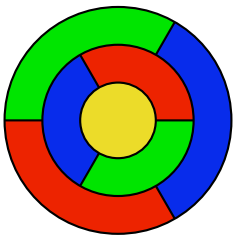
4. Output krets



$$E = \{p,q,s,r,t,u\}$$
$$i = AB$$
$$krets = ApBtDuBqCsDrA$$
$$nykrets = ApBqCsDrABtDuB$$

Digresjon: Firefargeproblemet

- I mange, mange år var følgende et åpent matematisk problem:
  Anta at vi har et plant kart over landområder (land, fylker, stater o.l.). Er det alltid mulig å trykke kartet ved hjelp av bare fire farger slik at to landområder som grenser opp mot hverandre alltid har forskjellig farge?
- Hvis vi representerer landene som noder og grensene som kanter, er dette egentlig et grafteoretisk problem.
- Grafteori, som en matematisk tung disiplin, har mye å hente fra forsøkene på å løse dette problemet.
- Måten problemet ble løst på har interesse i seg selv.
- De som løste det, reduserte problemet til et stort antall enkelttilfeller, som deretter ble sjekket av en datamaskin.
- Var det mennesker eller datamaskinen som løste problemet?

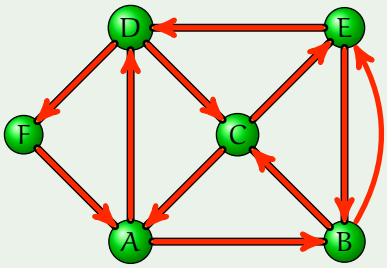


Eulerstier en gang til

- Vi har sett på en algoritme som fant en Eulerkrets når det var mulig.
- Den kan også brukes til å finne en Eulersti, det vil si en sti som er innom alle kantene nøyaktig én gang, men som kan begynne og slutte på forskjellige steder.
- Disse stedene må da være de to nodene med odde grad.
- Utvider vi grafen med en kant mellom disse to nodene, kan vi lage en Eulerkrets.

- Tar vi bort den nye kanten, får vi en Eulersti.

Eksempel.



- Det er to strategier for å bevise en setning som Eulers.
- Vi kan bruke et induksjonsbevis.
- Induksjonsbevis gir ofte opphav til algoritmer, og den algoritmen boka presenterer kan sees på som utledet av et induksjonsbevis.
- Vi kan også bevise teoremet *direkte*, uten å argumentere via algoritmen.
- Her følger et direkte bevis for at det er alltid fins en Eulerkrets hvis hver node har grad lik et partall.
- Det er også mulig å trekke en algoritme ut av dette beviset, og algoritmen blir veldig lik den vi så sist.

Bevis.
La G være en sammenhengende graf hvor gradene til alle nodene er partall. La S være en sti

$$v_0 e_1 v_1 e_2 v_2 \dots e_n v_n$$

av maksimal lengde slik at ingen kant forekommer to ganger. Vi skal vise at S er en Eulerkrets. Vi skal gjøre dette ved å vise følgende tre påstander:

(a) S er en krets.
(b) S inneholder alle nodene i grafen.
(c) S inneholder alle kantene i grafen.

Siden ingen kant forekommer to ganger, må S være en Eulerkrets.

Bevis (Fortsatt).
(a) Noden v_0 må være lik v_n. Når vi går ut av den første noden, v_0, via kanten e_1, så bruker vi opp én av kantene som ligger inntil v_0. For hver node vi går inn i og ut av, så bruker vi opp

to kanter. Når vi er fremme ved den siste noden i stien, v_n, så fins det ingen ubrukt kant som ligger inntil v_n. Hadde det vært en slik kant, så ville vi hatt en sti som var lenger enn S, og da hadde ikke S vært maksimal. Siden graden til v_n er et partall, så må vi tidligere i stien ha gått ut av v_n. Den eneste muligheten er at v_n er lik v_0. Dermed er S en *krets*.

Bevis (Fortsatt).
(b) S må bestå av alle nodene i grafen. Det er fordi grafen er sammenhengende og S er maksimal.
Hvis en node v ikke hadde vært med, så kunne vi ha laget en sti som var lenger enn S.

Bevis (Fortsatt).
(c) S inneholder alle kantene fra grafen.
Anta for motsigelse at det fins en kant e, som forbinder nodene u og v, som ikke er med i S.
Siden S inneholder alle nodene fra grafen, så må v være lik v_k for en passende k.
Da kan vi lage en sti som er lenger enn S ved å begynne med ue og fortsette med S:

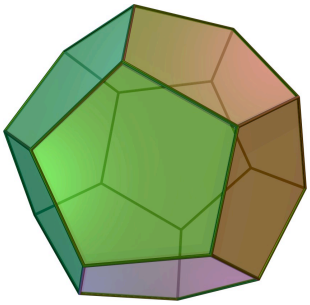
$$ue\underbrace{v_k e_{k+1} v_{k+1} \dots e_n v_n e_1 v_1 e_2 v_2 \dots e_{k-1} v_{k-1} e_k v_k}_{S}$$

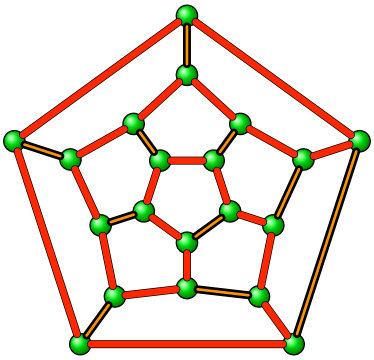
Hamiltonstier

- Vi må også si litt om stier som inneholder alle *nodene* i en graf, uavhengig hvorvidt alle kantene er med eller en kant er med flere ganger.
- "Den handelsreisendes problem" er et slikt problem, hvor man er ute etter den *korteste* stien som går gjennom alle byene i en mengde.

Definisjon.
La G være en sammenhengende graf. En *Hamiltonsti* er en sti som inneholder hver node fra G nøyaktig én gang. En *Hamiltonkrets* er en Hamiltonsti hvor den første og den siste noden sammenfaller. En sammenhengende graf som har en Hamiltonkrets kalles *Hamiltonsk*.

- Hamiltons puzzle tar utgangspunkt i et dodekaeder (et av de fem Platonske legemene) hvor hvert hjørne er merket med navnet på en by. Spørsmålet han stilte var om det var mulig å reise gjennom alle byene nøyaktig én gang. Vi ser at dette spørsmålet er det samme som om den tilhørende grafen har en Hamiltonsti.





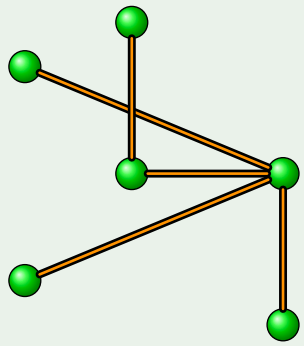
- Euler studerte også et tilsvarende problem: når det er mulig for en springer å gå over *alle* rutene på sjakkbrett av ulike størrelser.
- Det er ingen som har klart å lage en *effektiv* algoritme for å finne ut om det fins en Hamiltonkrets i en graf.
- Dette er "like vanskelig" som å bestemme om et utsagnslogisk utsagn er en tautologi.
  [Det tilhører klassen av **NP**-komplette problemer.]
- I praksis er det sjeldent at man virkelig trenger å finne en Hamiltonkrets.
- Ofte er det tilstrekkelig å finne en Eulerkrets, eller greit å gå over noder flere ganger.
- Det fins mange spesialtilfeller og heuristikker man kan benytte seg av.

Kapittel 11: Trær

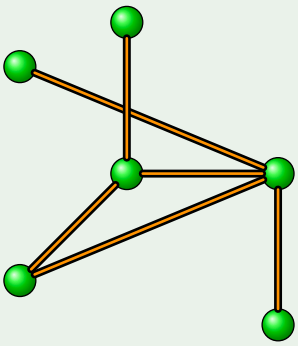
Trær

- Et tre er en spesiell type graf.
- Intuitivt er et tre noe som vokser fra en rot
  og så forgrener seg uten noe sted å vokse sammen igjen.
- Vi kan se på et biologisk tre som en graf, ved å la hvert forgreningspunkt være nodene, og delene av en stamme, gren eller kvist mellom to forgreningspunkter være kantene.
- Vi skal gi en presis definisjon av når en graf kan betraktes som et tre.
- Men, hvorfor skal vi lære om trær?
- Hvis en graf representerer et nettverk, vil et tre svare til et nettverk hvor det bare fins én sti fra en node til en annen.
- Hvis nettverket består av kabler eller andre medier som formidler informasjon, kan det være hensiktsmessig at signaler bare går langs én vei, slik at systemet ikke forstyrres av at samme informasjon kommer med små tidsforskjeller.
- Dataobjekter som sammensatte algebraiske uttrykk, utsagnslogiske formler eller program har ofte en trestruktur som beskriver hvordan komplekse objekter er bygget opp fra enklere objekter.
- For å undersøke om et utsagn formalisert i matematikken kan bevises eller ikke, kan man prøve å bygge opp et tre av utsagn hvor forgreningen stopper når vi har nådd aksiomene.
- Denne naive idéen danner grunnlaget for enkelte automatiske bevissøkere.

Eksempel.



Eksempel.



Definisjon.

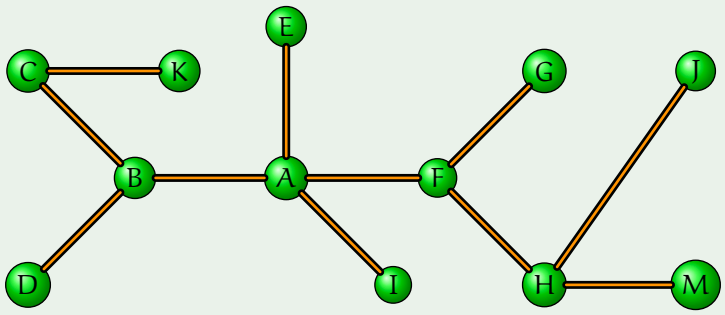
a) En sykel (engelsk: *cycle*) i en graf er en sti med følgende egenskaper.
   - Stien inneholder minst en kant.
   - Ingen kant forekommer mer enn én gang.

– Stien er en krets, det vil si, den begynner og slutter i samme node.

En sykel med n kanter kalles en n-sykel.

b) En graf er et tre hvis grafen er sammenhengende og grafen ikke inneholder noen sykler.

Eksempel.



Vi ser at denne grafen har 12 noder og 11 kanter.

Eksempel.

Et tre trenger ikke å ha noen forgreningspunkter:

Her har vi 5 noder og 4 kanter.

- I de eksemplene vi har sett på har vi alltid endenoder i et tre, det vil si noder av grad 1.
- Husk at en graf alltid har minst en node.

  Grafen med en node og ingen kanter er et tre. Alle andre trær vil ha endenoder.
- I de eksemplene vi har sett har alle trærne en node mer enn de har kanter.

  Dette er en egenskap som alle endelige trær har.

  Det er ingenting i definisjonen av grafer og trær som sier at de skal være endelige, men vi kommer til å begrense oss til endelige grafer og trær hvis vi ikke sier noe annet. Boka forutsetter også at vi bare arbeider med endelige grafer og trær i dette kurset.

Teorem.

a) Hvis et tre har minst en kant, så har treet en node med grad 1 (En slik node kaller vi en endenode eller bladnode).

b) I ethvert tre fins det nøyaktig én node mer enn det fins kanter.

Bevis.

a) La

$$v_0 e_1 \cdots e_n v_n$$

være en sti med maksimal lengde hvor ingen kant forekommer to ganger.

Siden grafen er et tre, kan ikke stien være innom samme node to ganger.

Endenodene v_0 og v_n må være bladnoder, siden vi ellers ville kunnet gjøre stien lengere.

Bevis (Fortsatt).

b) Vi bruker induksjon på antall noder i treet.

Hvis det bare fins en node, har vi ingen kanter, og påstanden stemmer.

Hvis det fins mer enn en node, kan vi anta at påstanden holder for alle mindre trær.

Tar vi bort en bladnode og den ene kanten som ligger inntil denne noden, får vi et mindre tre.

Siden vi har tatt bort en node og en kant, og ved induksjonsantagelsen da har en node mer enn vi har kanter, må dette være tilfellet i det opprinnelige treet også.

Resonementet illustreres på tavla.

Vektede grafer

- Hvis en graf representerer et veinett er det av interesse å vite hvor lange de enkelte veistrekningene er.
- Hvis en graf representerer et ledningsnett, kan anleggskostnader og driftskostnader ved de enkelte strekningene være av interesse.
- Hvis nodene i en graf står for land og kantene for grenseoverganger mellom dem, kan tollsatsene eller andre egenskaper ved de forskjellige grenseovergangene bety noe.
- Siden vi har mange eksempler på grafer hvor det er viktige tallstørrelser knyttet til de enkelte kantene, studerer vi vektede grafer som et eget begrep.

Definisjon.

En vektet graf er en graf hvor hver kant har fått en vekt, et positivt reelt tall.

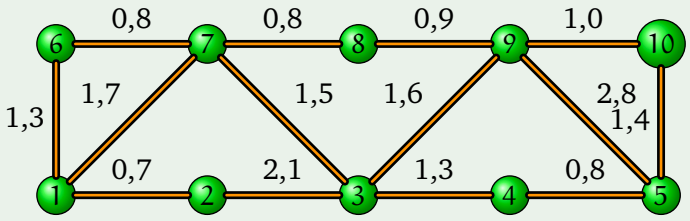
Merk.

- Formelt sett kan vi definere en vektet graf som et par (G, f) hvor G er en graf og f er en funksjon fra mengden av kanter i G til de positive reelle tallene.

- Vi har altså bruk både for ordnede par og for funksjoner for å gi en skikkelig definisjon.

Eksempel.

La oss se på et eksempel:



- Det er nå mulig å trekke kabler mellom disse skjematisk tegnede byene, hvor kostnaden f.eks. er målt i antall NOK 10^7.
- Kan vi fjerne noen av kantene slik at anleggskostnadene blir minst mulig, men vi fortsatt forbinder alle byer med kabler?

- Så lenge grafen inneholder kretser, må det være greit å ta bort en kant i kretsen.
- Vi bør derfor finne det mest kostnadseffektive deltreet som når over alle nodene.
- Vi skal komme tilbake til dette eksemplet når vi har diskutert algoritmen som ligger bak.

Utspennende trær

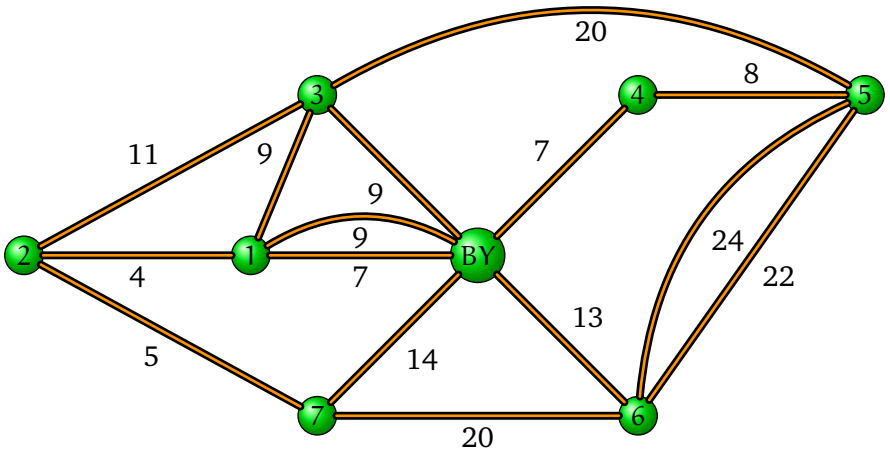
Definisjon.

- La G være en sammenhengende graf, og la T være et deltre av G. Det betyr her at T og G har de samme nodene, alle kantene i T er kanter i G, men noen kanter i G kan mangle i T.
- Vi sier at T spenner ut G hvis alle nodene i G ligger inntil en kant i T. (Tegning på tavla.)
- Husk at et tre er en sammenhengende graf, så dette betyr at alle par av forskjellige noder i G kan forbindes med en (og bare en) sti i T.
- Hvis G er en vektet graf, er problemet å finne et tre T som spenner ut G slik at summen av vektene på kantene i T blir minst mulig.

- Det kan finnes mange forskjellige utspennende trær i en graf.
- Hvert slikt tre vil ha en samlet vekt, ved at vi legger sammen vektene på kantene.
- I en situasjon hvor vektene representerer kostnader, og hvor det er teknologisk nødvendig eller tilstrekkelig å erstatte grafen med et utspennende tre, er det av interesse å kunne finne et utspennende tre med minst mulig vekt.
- Det fins effektive algoritmer for å kunne gjøre dette.
- Vi skal se på en slik algoritme: Prims algoritme.

En kommunegraf

- Vi skal se på et realistisk eksempel på en situasjon som langt på vei kan modelleres som en vektet graf, og hvor det vil være relevant å finne en Eulerkrets eller sti, en Hamiltonkrets og et minimalt utspennende tre for å løse visse samfunnsoppgaver.
- I virkelighetens verden finner man ofte ikke en Eulersti når man trenger en eller en Hamiltonkrets når man trenger en, men som vi skal se, kan man alltid finne minimale utspennende trær.
- Vårt eksempel er en graf som modellerer veinettet mellom de lokale tettstedene i en kommune, og vektingen av kantene er antall kilometer hver enkelt veistrekning er på. Grafen er ikke *enkel*, men bortsett fra det er den som en vektet graf.



- Snøbrøyterne: *Fins det en Eulersti?*
- Postutkjørerne: *Fins det en Hamiltonkrets?*
- Bredbåndutbyggerne: *Fins det et minimalt utspennende tre?*

Oppgave.

a) Avgjør om det fins en Eulerkrets eller en Eulersti i kommunegrafen, og finn i så fall denne.

b) Er spørsmålet om det fins en Hamiltonkrets det rette spørsmålet?
Kunne postutkjørerne stilt et mer fornuftig grafteoretisk spørsmål?

c) Finn et minimalt utspennende tre (bruker stoff fra resten av forelesningen).
For å få en vektet graf i tråd med definisjonen, kan du ta bort unødige kanter med mye vekt der det fins parallelle kanter.

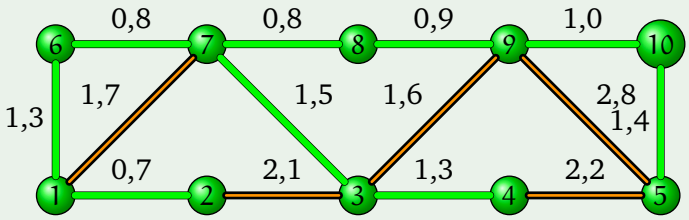
Prims algoritme

- Prims algoritme gir en metode for å finne det minimale utspennende treet til en vektet graf.
- I læreboka står det en pseudokode for Prims algoritme.
- Her vil vi beskrive algoritmen litt mer uformelt.

- Det viser seg at hvis man bygger opp et tre ved i hvert skritt å gjøre det som i øyeblikket virker mest fornuftig, så kommer man frem.
- Vi skal trolig ikke gi et korrekthetsbevis for Prims algoritme, men det forventes at man kan praktisere den på eksempler.
- Vi beskriver Prims algoritme litt annerledes enn den er formulert i læreboka, men effekten er den samme, vi får det samme treet bygget opp i den samme rekkefølgen.
- La G være en vektet, sammenhengende graf med noder $V = \{v_1, \ldots, v_n\}$.
- La T_1 være treet som består av v_1 og ingen kanter.
- Start med node v_1 og la $V_1 = \{v_2, \ldots, v_n\}$, altså resten av nodene.
- Finn $v_{i_1} \in V_1$ med minimal avstand til v_1 via kant e_1.
- Vi får V_2 ved å fjerne v_{i_1} fra V_1 og vi får T_2 ved å legge til v_{i_1} og e_1 til T_1.
- Deretter fortsetter vi med alltid å velge den ubrukte noden som ligger nærmest, via en kant, til treet bygget opp så langt, og vi bygger ut treet med denne noden og den tilsvarende kanten.
- Siden grafen er sammenhengende, vil vi alltid finne en ny node som er "nabo" til treet bygget opp på et gitt tidspunkt, og da finner vi alltid en ny node som ligger nærmest.
- Vi skal illustrere hvordan denne algoritmen virker på eksemplet vi har gitt på en vektet graf.

Eksempel (Fortsatt).

- Vi ser på hvordan man ved hjelp av Prims algoritme, skritt for skritt, kan bygge opp et utspennende tre med minimal vekting.
- Vi starter i Node 1.



MAT1030 – Forelesning 25

Trær

Dag Normann - 27. april 2010

Forelesning 25

Litt repetisjon

- Vi har snakket om grafer og trær. Av begreper vi så på var følgende:
- Eulerstier og Eulerkretser
- Hamiltonstier og Hamiltonkretser
- Trær
- Vektede grafer
- Minimale utspennende trær
- Et tre er en sammenhengende graf uten sykler.
- En sykel er en krets hvor samme kant ikke kan forekomme to ganger *og hvor samme node ikke kan forekomme to ganger*.
- Dette siste kravet ble uteglemt forrige uke, uten at det endrer hva vi mener med et tre.
- Et tre kan faktisk defineres som en sammenhengende graf hvor det finnes én node mer enn antall kanter.
- Vi viste at dette er en egenskap ved trær.
- Omvendingen gis som en utfordring til de som har lyst til å prøve seg på et bevis.
- Vi minner om at en *vektet graf* er en enkel graf hvor hver kant har en vekt, et positivt reelt tall (Alternativt: Et ikke-negativt tall).
- Hvis G er en sammenhengende graf, vil et utspennende tre være en delgraf T slik at
  - T har de samme nodene som G.
  - T er et tre, det vil si, T er sammenhengende og inneholder ingen sykler.
- Vi vet at hvis G har n noder, vil et utspennende tre ha $n-1$ kanter.
- Omvendt, vil et tre med $n-1$ kanter ha n noder.
- En konsekvens er at hver gang vi velger ut $n-1$ kanter fra G slik at vi ikke får noen sykler, så får vi et utspennende tre.
- Forrige uke så vi på det som kalles Prims algoritme.
  Vi tar den fra begynnelsen igjen.

Prims algoritme

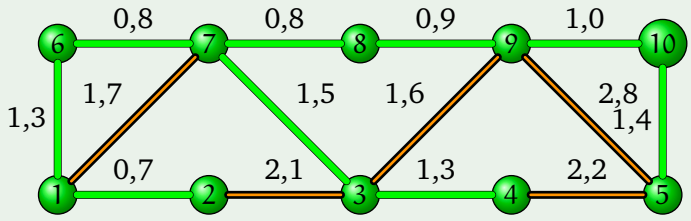
- Prims algoritme gir en metode for å finne det minimale utspennende treet til en vektet graf.
- I læreboka står det en pseudokode for Prims algoritme.
- Her vil vi først beskrive algoritmen litt mer uformelt.

- Det viser seg at hvis man bygger opp et tre ved i hvert skritt å gjøre det som i øyeblikket virker mest fornuftig, så kommer man frem.
- Vi skal ikke gi et korrekthetsbevis for Prims algoritme, men det forventes at man kan praktisere den på eksempler.
- Vi beskriver Prims algoritme litt annerledes enn den er formulert i læreboka, men effekten er den samme, vi får det samme treet bygget opp i den samme rekkefølgen.

- La G være en vektet, sammenhengende graf med noder $V = \{v_1, \ldots, v_n\}$.
- La T_1 være treet som består av v_1 og ingen kanter.
- Start med node v_1 og la $V_1 = \{v_2, \ldots, v_n\}$, altså resten av nodene.
- Finn $v_{i_1} \in V_1$ med minimal avstand til v_1 via kant e_1.
- Vi får V_2 ved å fjerne v_{i_1} fra V_1 og vi får T_2 ved å legge til v_{i_1} og e_1 til T_1.
- Deretter fortsetter vi med alltid å velge den ubrukte noden som ligger nærmest, via en kant, til treet bygget opp så langt, og vi bygger ut treet med denne noden og den tilsvarende kanten.
- Siden grafen er sammenhengende, vil vi alltid finne en ny node som er "nabo" til treet bygget opp på et gitt tidspunkt, og da finner vi alltid en ny node som ligger nærmest.
- Vi skal illustrere hvordan denne algoritmen virker på eksemplet vi har gitt på en vektet graf.

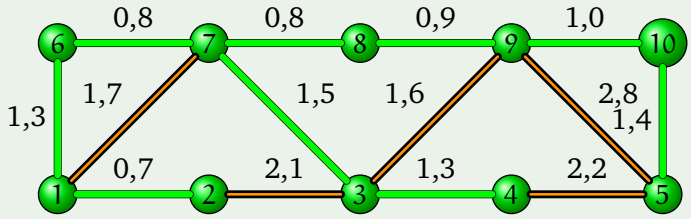
Eksempel (Fortsatt).

- Vi ser på hvordan man ved hjelp av Prims algoritme, skritt for skritt, kan bygge opp et utspennende tre med minimal vekting.
- Vi starter i Node 1.



Eksempel (Fortsatt).

- Så lenge vi har listet opp kantene i rekkefølge og alltid velger den kanten med minst vekt som kommer først i listen vår, spiller det ingen rolle hvor vi starter.
- Hvis vi starter i Node 8 bygger vi opp treet slik.



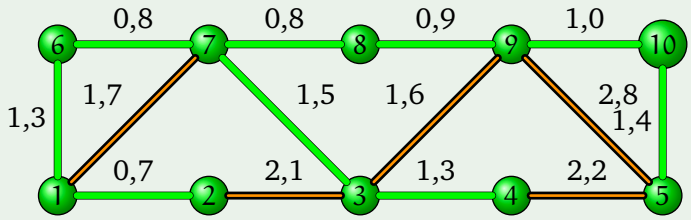
- Det ble det samme treet til slutt.

- Vi gir en pseudokode for Prims algoritme.
- Den ser litt annerledes ut enn den som står i boka, men effekten, skritt for skritt, er den samme.

1 *Input* $V = \{v_1, \ldots, v_n\}$ [Nodene i G]
2 *Input* $E = \{e_1, \ldots, e_k\}$ [Kantene i G]
3 $T \leftarrow \{v_1\}$
4 $K \leftarrow \emptyset$
5 **While** $E \neq \emptyset$ **do**
  5.1 $x \leftarrow$ *første* $e_i \in E$ *slik at* e_i *ligger inntil en node i* T *og har minimal vekt blant disse.*
  5.2 $y \leftarrow$ *noden ved* x som ikke ligger i T
  5.3 $K \leftarrow K \cup \{x\}$
  5.4 $T \leftarrow T \cup \{y\}$
  5.5 $E \leftarrow E - \{e_i$; e_i *ligger inntil to noder i* $T\}$.
6 *Output* (T, K).

- Det fins en fremgangsmåte som ikke er avhengig av at vi starter noe sted, og som også vil gi oss det samme treet.
- Her bygger vi ikke opp et tre, men små deltrær som til sist vil vokse seg sammen til et tre.
- Vi utnytter at vi får et utspennende tre bare vi tar med $n-1$ kanter uten å lage noen sykler.
- Hver gang legger vi til en ny kant med minimal vekt slik at vi ikke får noen sykler.
- Fins det flere aktuelle kanter med samme minimale vekt, velger vi den som står først i listen vår.
- I eksemplet vårt vil da det utspennende treet bygge seg opp slik som på neste side.

Eksempel (Fortsatt).



- Vi får fortsatt det samme treet.
- I utgave 3 blir denne algoritmen omtalt (i oppgavedelen) som Kruskals algoritme.

Vi regner et annet eksempel på tavla, både ved bruk av Prims algoritme og Kruskals algoritme.

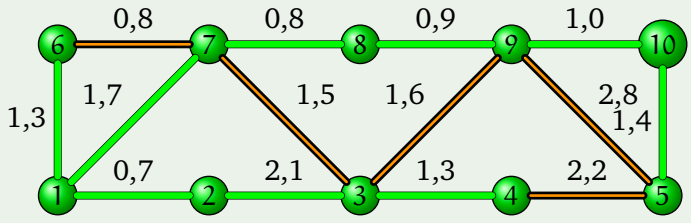
Dijkstras algoritme

- Et annet naturlig problem i forbindelse med vektede grafer er å finne et utspennende tre slik at hver node har en minimal avstand til en gitt "sentralnode".
- Her tenker vi oss at vektene svarer til lengder av de enkelte kantene.
- Det fins effektive algoritmer for dette også.
- Vi skal gi en uformell beskrivelse av Dijkstras algoritme og vise hvordan den virker på eksemplet vårt.
- Vi skal se at vi denne gangen får et annet tre.
- For dette problemet er også treet vi får avhengig av hvilken node som velges som "sentrum".

- Anta at vi har en vektet graf G med en opplisting av n noder og endel kanter.
- Vi starter med en *sentrumsnode* v_1 og lar T_1 bestå av noden v_1 og ingen kanter.
- Ved rekursjon på $i \leqslant n$ konstruerer vi et tre T_i med i noder og $i-1$ kanter fra G.
- Vi konstruerer T_{i+1} fra T_i når $i < n$ ved følgende prosedyre:
  1. Finn den noden v utenom T_i og den kanten e som er slik at e forbinder v til T_i, og vi oppnår den kortest mulige veien fra v_1 til v, via T_i og e, på denne måten.
  2. Fins det flere like gode alternativer, velg den v'en og deretter den e'en som står først i listene.
  3. Utvid T_i til T_{i+1} ved å legge til noden v og kanten e.
- Dijkstras algoritme er beskrevet i form av en pseudokode i boka.
- Det er meningen at dere skal kunne finne det utspennende treet som gir de korteste stiene fra provinsen til sentrum når dere har fått gitt en vektet, sammenhengende graf.
- La oss se på eksemplet vårt en gang til.

Eksempel.

- Vi starter i Node 1
- Vi får et nytt tre.



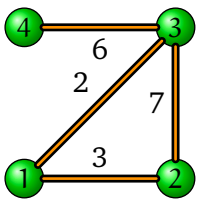
- Vi illustrerer Dijkstras algoritme ved to eksempler på tavla.
- Noen har lært andre måter å gjennomføre Prims algoritme, Kruskals algoritme og Dijkstras algoritme på enn slik vi gjør det her.
- Bruker dere en alternativ metode, må dere forklare den for sensor.

- Dijkstras algoritme kan også brukes til å finne korteste sti mellom to noder.
- Det vil være stien mellom noedne med minst mulig samlet vekt.
- Vi lar en av nodene være startnoden i Dijkstras algoritme, og så følger vi algoritmen til den andre noden er med i treet vi konstruerer.

Representasjoner

- Når vi beskriver algoritmer som finner bestemte typer trær i vektede grafer, ligger det selvfølgelig under at vi tenker oss at disse algoritmene skal kunne programmeres.
- Det betyr at det må være mulig å representere disse vektede grafene digitalt.
- Vi har tidligere sett hvordan grafer kan representeres som matriser.
- Siden matriser kan representeres digitalt, betyr det at også grafer kan representeres digitalt.
- Vektede grafer kan også representeres som matriser.
- Vi følger boka, og lar ∞ representere at det ikke fins noen kant mellom to noder.
- Hvis vi tolker grafen som en strømkrets hvor vektene representerer motstanden i hver enkelt ledning, vil det at vi ikke har noen direkte kobling mellom to noder svare til at vi har en ledning med uendelig motstand mellom dem.
- Vi illustrerer dette med et eksempel.

En vektet graf.



Matriserepresentasjonen.

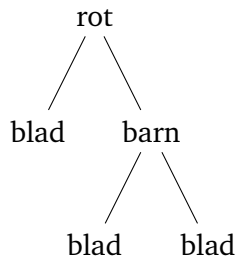
	1	2	3	4
1	0	3	2	∞
2	3	0	7	∞
3	2	7	0	6
4	∞	∞	6	0

Denne representasjonen må *ikke* forveksles med matriserepresentasjonen av grafer som ikke er enkle.

Trær med rot

- Til nå har vi sett på trær som sammenhengende grafer uten sykler.
- Det betyr at vi ikke har noe dynamisk bilde av disse trærne, de har ikke noe startpunkt eller noen rot.
- For mange anvendelser av teorien for trær, er det nyttig å kunne betrakte den ene noden som roten til treet, den noden alt annet har vokst ut fra.
- Formelt sett er et tre med rot definert som et grafteoretisk tre hvor en av nodene er utpekt som rot.

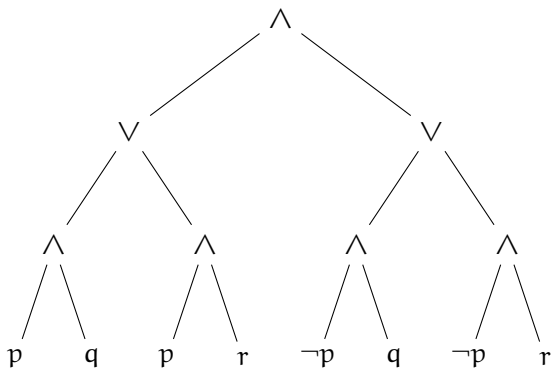
- Det er imidlertid vanlig å tegne slike trær litt annerledes enn trær uten rot.



- Roten tegnes øverst, og treet "vokser" nedover.
- Nodene ligger i "lag" avhengig av avstanden til roten.
- Vi kan snakke om barna til en node, og om bladene til et tre med rot.

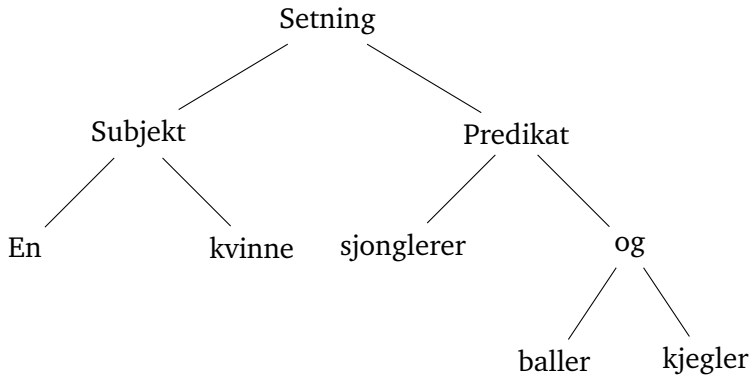
Vi skal se på endel eksempler før vi går nærmere inn på terminologien.
Det er ikke noe i veien for å tegne roten nederst eller til venstre eller høyre på arket, det er bare ikke vanlig.

- Vi kan bruke trær til å gi en grafisk fremstilling av en utsagnslogisk formel.
- La $A = ((p \wedge q) \vee (p \wedge r)) \wedge ((\neg p \wedge q) \vee (\neg p \wedge r))$.
- Mengden av formler er bygget opp induktivt, og oppbyggingen av hver formel kan beskrives som et tre:

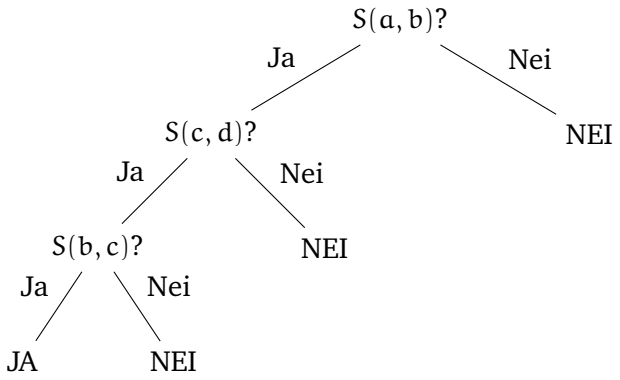


- Et tre som det på forrige side kaller vi ofte et syntakstre.
- Et syntakstre forteller oss hvordan et ord i et litt komplisert formelt språk er bygget opp av enklere ord.
- Syntakstrær kan brukes i grammatikkanalyse av setninger i naturlige språk.
- Da setter man opp et tre som beskriver hvordan en setning f.eks. er bygget opp av subjekt og predikat, hvordan subjektet kan bestå av en artikkel og et substantiv, osv.
- Som et eksempel skal vi se et slikt tre på neste side, uten at bruken av slike trær skal være noe tema for disse forelesningene.

- *En kvinne sjonglerer baller og kjegler.*
- Denne setningen er bygget opp slik.



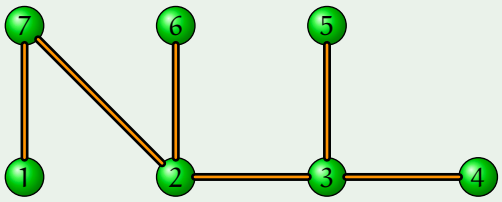
- Trær kan ofte brukes til å beskrive forskjellige algoritmer hvor man stiller visse spørsmål, og prosessen videre avhenger av svarene på de enkelte spørsmålene.
- I læreboka står det et eksempel på et tre av spørsmål som kan brukes til å bestemme rekkefølgen på tre forskjellige tall a, b og c.
- Vi skal se på en spørsmålsserie som avgjør om fire individer, a, b, c og d, tilhører samme art.
- Vi skriver $S(x, y)$ for at x og y tilhører samme art.
- Det å tilhøre samme art er en ekvivalensrelasjon, og korrekthet av programmet bygger utelukkende på dét (på samme måte som korrekthet av eksemplet i boka bygger på at vi har en total ordning).



- Det er verd å merke seg at prosedyren over faktisk sjekker om a, b, c og d står i relasjon S til hverandre når S er en transitiv relasjon.
- Vi sjekker først $S(a, b)$.
- Får vi negativt svar gir algoritmen negativt svar.
- Får vi positivt svar sjekker vi $S(c, d)$.
- Får vi positivt svar her også sjekker vi til slutt $S(b, c)$.
- Hvis S er transitiv vet vi at a, b, c og d ligger etter hverandre i S-relasjonen.
- En algoritme som virker for veldig mange tolkninger av input kalles gjerne en polymorf algoritme; det vil si at den virker på mange strukturer.
- Sorteringsalgoritmer er eksempler på polymorfe algoritmer.
- Hvis vi tar utgangspunkt i et tre uten rot, og så tegner det som et tre med rot, vil det visuelle resultatet avhenge mye av hvor vi plasserer roten.

Oppgave.

- Tegn følgende tre som et tre med rot når vi plasserer roten henholdsvis i nodene 1, 3 og 6:



Trær med rot

Definisjon.

- La T være et tre med rot (anta at vi tegner T med roten øverst).
- Med nivået til en node mener vi antall kanter mellom noden og roten.
- Hvis det fins en kant mellom node a og b, og a har lavest nivå (ligger øverst i tegningen) sier vi at b er barnet til a.
- Hvis det fins en sti mellom to noder slik at
  - en kant k i stien ligger inntil nodene a og b (det vil si at sekvensen akb er en del av stien) nøyaktig når b er barnet til a,

  så vil den som har det høyeste nivået (ligger nederst i tegningen) være etterkommer til den andre, som omvendt er forgjenger til den første.

Definisjon (Fortsatt).

- En node som ikke har noen barn er et blad eller en løvnode.
- En gren er en sti fra roten til et blad.
  (Noen vil kalle dette en maksimal gren og la en gren være en sti fra roten til en node.)

Oppgave.
Vis at det finnes en bijeksjon mellom mengden av blader og mengden av grener i et tre med rot.

MAT1030 – Forelesning 26

Trær

Dag Normann - 28. april 2010

Forelesning 26

Litt repetisjon

- Prims algoritme
  - finne det minste utspennende treet i en vektet graf
  - en *grådig* algoritme i den forstand at den vurderer lokalt hva som er det beste neste skrittet
- Dijkstras algoritme
  - en av nodene er *sentrum*
  - finne det treet som gir kortest mulig vei fra hver av de andre nodene til sentrum
- Matriserepresentasjoner
- Trær med rot
  - en av nodene har status som *rot*

Trær med rot

- La T være et tre med rot (anta at vi tegner T med roten øverst).
- Vi definerte:
- Nivået til en node.
- Hva vi mener med et barn til en node.
- Hva vi mener med en etterkommer til en node.
- Hva vi meder med en forgjenger til en node.
- En node som ikke har noen barn er et blad eller en løvnode.
- En gren er en sti fra roten til et blad.

Binære trær

- Veldig mange trær har den egenskapen at hvis en node ikke er en bladnode, så har den nøyaktig to barn.
- Vi skiller ut disse ved en egen betegnelse.

Definisjon.

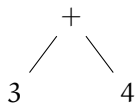
- Et binært tre er et tre med rot slik at følgende holder.
  1. Enhver node er enten en bladnode eller har nøyaktig to barn.

2. Hvis en node har to barn, vil det ene barnet betegnes som barnet til venstre og det andre som barnet til høyre.

- Det venstre deltreet får vi ved å fjerne roten og barnet til høyre og alle dets etterkommerne. Tilsvarende for det høyre deltreet. (I et deltre blir det ene barnet den nye roten.)
- I et tre med kun én node kan vi ikke snakke om deltrær.

Traverseringer

- En traversering av et tre innebærer at vi leser nodene i treet i en bestemt rekkefølge og utfører operasjoner (som å skrive symboler) i en bestemt rekkefølge.
- Eksempel



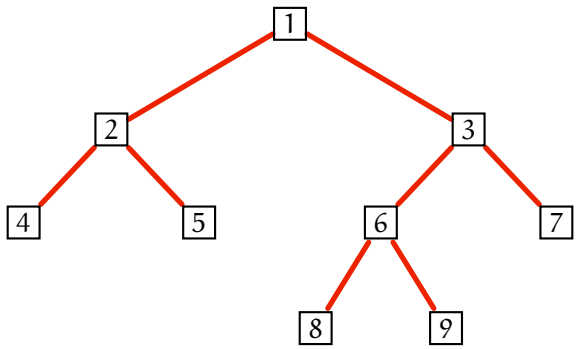
- Vi skal se på tre vanlige måter å traversere et tre på.
  - *in-order* traversering – svarer til *infiks* notasjon: $3+4$
  - *pre-order* traversering – svarer til *prefiks* notasjon: $+34$
  - *post-order* traversering – svarer til *postfiks* notasjon: $34+$

In-order-traversering

- Her er algoritmen for den traverseringen som gir *infiks* notasjon hvis input er et syntakstre.

Algoritme *in-order_traverse(*T*)*:

1. **If** T ikke er et blad **then**
   1.1. *in-order_traverse(venstre deltre av* T*)*
2. Output roten til T
3. **If** T ikke er et blad **then**
   3.1. *in-order_traverse(høyre deltre av* T*)*



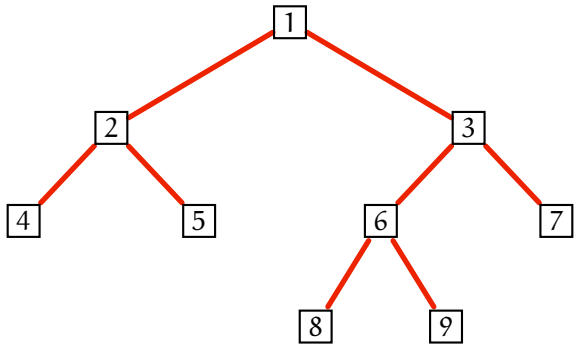
in-order-traversering gir 4 2 5 1 8 6 9 3 7

Pre-order-traversering

- Her er algoritmen for den traverseringen som gir *polsk prefiks* notasjon hvis input er et syntakstre.

Algoritme *pre-order_traverse(*T*)*:

1. Output roten til T
2. **If** T ikke er et blad **then**
   2.1. *pre-order_traverse(venstre deltre av* T*)*
3. **If** T ikke er et blad **then**
   3.1. *pre-order_traverse(høyre deltre av* T*)*



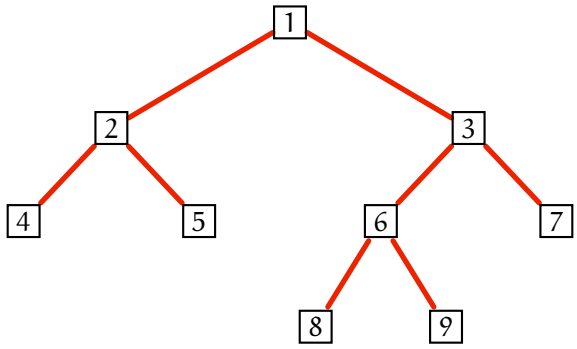
pre-order-traversering gir 1 2 4 5 3 6 8 9 7

Post-order-traversering

- Her er algoritmen for den traverseringen som gir *baklengs polsk* notasjon, eller *postfiks* notasjon, hvis input er et syntakstre.

Algoritme *post-order_traverse(*T*)*:

1. **If** T ikke er et blad **then**
   1.1. *post-order_traverse(venstre deltre av* T*)*
2. **If** T ikke er et blad **then**
   2.1. *post-order_traverse(høyre deltre av* T*)*
3. Output roten til T



post-order-traversering gir 4 5 2 8 9 6 7 3 1

Notasjon: infiks, prefiks, postfiks

- Når vi skriver utsagnslogiske eller algebraiske uttrykk, er vi vant til å skrive symbolene $\wedge, \vee, +, \cdot$, etc. *mellom* deluttrykkene.
- Denne skrivemåten er historisk betinget og kalles for infiks.
- Fordelen med forlengs og baklengs polsk notasjon, eller *prefiks* og *postfiks*, er at man slipper parenteser.
- Disse kan være bedre egnet for innmating i algoritmer.
- Programmeringsspråket *LISP* er basert på bruk av polsk notasjon, og i "gamle dager" måtte lommeregnere programmeres med baklengs polsk notasjon.
- Brukere av editoren *Emacs* vil oppdage at den innebygde kalkulatoren bruker baklengs polsk notasjon (RPN).

Eksempel

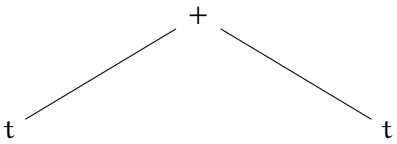
- Vi skal gi en grammatikk som definerer alle termer i symbolene 0, 1, $+$ og $\times$ i forlengs polsk, eller prefiks, notasjon.
- En term er et uttrykk som beskriver et tall. Vi skal se på alle måter å beskrive tall på hvor vi bruker symbolene nevnt over.
- Poenget med polsk notasjon er at funksjonssymbol, logiske bindeord og andre symboler vi bruker til å binde sammen enkle uttrykk til mer komplekse settes først, og så kommer deluttrykkene etter, uten parenteser.
- En grammatikk er et sett regler som forteller oss hvilke ord som tilhører det formelle språket vi vil beskrive.
- I informatikk-litteratur har man, som tidligere nevnt, utviklet en rask skrivemåte for slike grammatikker.

  Term t
  $\mathrm{t} ::= 0 \mid 1 \mid +\mathrm{tt} \mid \times\mathrm{tt}$

- Denne definisjonen skal leses som følger:
- Mengden av *termer* er den minste mengden som oppfyller
  - 0 og 1 er termer.
  - Hvis t og s er termer, er $+\mathrm{ts}$ og $\times\mathrm{ts}$ også termer.
- Vi har sett på tilsvarende konstruksjoner da vi så på formelle språk definert ved generell induksjon.
- En induktiv definisjon forteller oss at det ligger en *trestruktur* bak hvert ord i språket.
- Hvordan kan vi bruke trær til å bestemme om et ord i alfabetet $\{0, 1, +, \times\}$ er en term eller ikke,

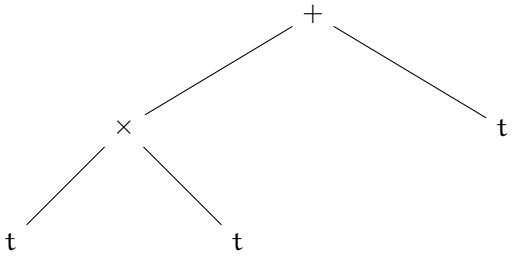
  og hvordan kan vi bruke trær til å finne en mer lesbar form av termen?

- Er $+ \times 00 \times +100$ en term slik det ble definert på forrige side?
- Vi kan prøve å utvikle et syntakstre for dette ordet, hvor vi bruker bokstaven t for å markere at her må det stå en enklere term.
- Første tilnærming til syntakstreet må være

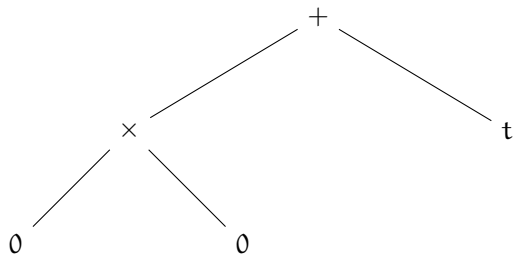


hvor $\mathrm{tt} = \times 00 \times +100$

- Den første ukjente termen må begynne med $\times$, så syntakstreet må se ut som



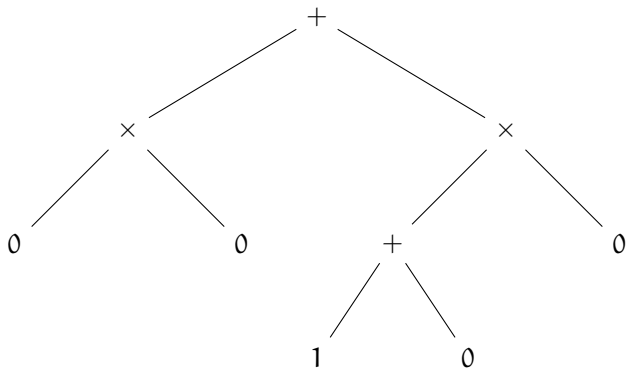
hvor $\mathrm{ttt} = 00 \times +100$. Her kan vi se direkte hva de tre t-ene må stå for, så vi får treet



hvor $\mathrm{t} = \times + 100$.

- Vi kan fortsette å avsløre hvordan syntakstreet må se ut ved å lese problemordet vårt fra venstre mot høyre.
- Vi ser at neste term er et produkt hvor første faktor er summen av 1 og 0 og andre faktor er 0

  (Vi er ikke interessert i verdien av denne termen, bare om det er en term).
- Det gir oss følgende fullstendige syntakstre.



Skrevet på vanlig infiks-form får vi

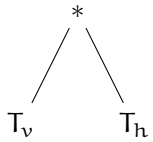
$$(0 \times 0) + ((1 + 0) \times 0).$$

Mer om notasjon

- Når man bruker baklengs polsk notasjon på en lommeregner, taster man inn tall og funksjonsuttrykk som $+$, exp, sin i rekkefølge.
- Hver gang man taster inn et funksjonsuttrykk, vil lommeregneren oppfatte siste tall, eller de to siste tallene, som input, og erstatte disse med funksjonsverdien.
- Det er altså funksjonsverdien som oppfattes som det siste tallet du tastet inn i fortsettelsen.
- Det er ikke vanskelig å se at lommeregneren kan behandle en sekvens av tall og funksjoner på bare en måte. Det betyr at et uttrykk i baklengs polsk notasjon bare kan tolkes på en måte.
- Det samme gjelder da selvfølgelig for forlengs polsk notasjon.
- Syntakstreet for en term eller et utsagnslogisk uttrykk er uavhengig av om vi har brukt vanlig infiks notasjon, forlengs polsk eller baklengs polsk notasjon.
- Når syntakstreet er gitt, så kan disse uttrykkene gjenskapes ved hjelp av de ulike traverseringsalgoritmene.
- Det er enkelt å lage en algoritme som tar et uttrykk, sjekker om det er korrekt og som eventuelt gir syntakstreet som output.

Induksjon og rekursjon for binære trær

- Mengden av binære trær kan også defineres *induktivt*.
- Utgangspunktet, eller *induksjonstarten*, blir nulltreet, treet som består av en node og ingen kanter.
- Denne noden er da både *rot* og *blad*.
- *Induksjonskrittet* består i at vi tar to binære trær T_v og T_h, en ny rotnode og to nye kanter, mot venstre til roten i T_v og til høyre mot roten i T_h.



Sammensetning av to binære trær til ett.

- Vi kan oppfatte denne sammensetningen som en form for *sum* av to binære trær: Vi legger sammen to trær og får et nytt tre.
- Vi kan godt skrive $T_v \oplus T_h$ for denne sammensetningen av trær.
- Merk at den kommutative loven ($x \oplus y = y \oplus x$) ikke gjelder; det er essensielt hvilket av trærne som settes til venstre og hvilket som settes til høyre.
- Som grafer betyr det ikke så mye, men for trerekursjon er det viktig, siden vi der kan referere til venstre og høyre deltre.
- Vi skal nå se på en form for produkt av trær.
- Vi skriver $*$ for nulltreet.

Definisjon.
Vi definerer "treproduktet" $\otimes$ ved

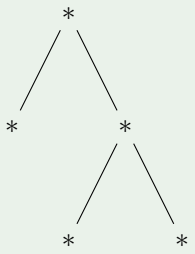
- $* \otimes S = S$
- $(T_v \oplus T_h) \otimes S = (T_v \otimes S) \oplus (T_h \otimes S)$

- Poenget med å gi denne definisjonen er å gi et eksempel på hvordan man kan definere ting ved rekursjon på oppbyggingen av et tre.
  - Vi illustrerer hva som skjer ved et par eksempler på tavla.
  - Vi ser at effekten er å erstatte alle bladnodene i T med kopier av S.
- Generelt kan vi definere en funksjon f ved rekursjon over oppbyggingen av binære trær ved følgende.
  1. Bestemme hva $f(*)$ er, når $*$ er nulltreet.
  2. Bestemme hvordan $f(T)$ avhenger av de to deltrærne $f(T_v)$ og $f(T_h)$, når T er et sammensatt tre.

Oppgave.
Definer treeksponering S^T ved trerekursjon på T ved

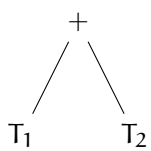
1. $S^* = *$
2. $S^T = (S^{T_v} \otimes S) \oplus (S^{T_h} \otimes S)$ når T er et sammensatt tre.

La T og S begge være treet



Finn S^T.

- Hvis vi går tilbake til syntakstrær, så kan vi se hvordan de tre ulike notasjonsformene kan defineres via trerekursjon.
- De tre algoritmene svarer til hver sin rekursive funksjon.
- Vi definerer funksjonen $\texttt{infiks}$ ved trerekursjon på følgende måte.
  - Hvis roten i T er en bladnode med merke a, så lar vi $\texttt{infiks}(T) = a$.
  - Hvis T er på formen

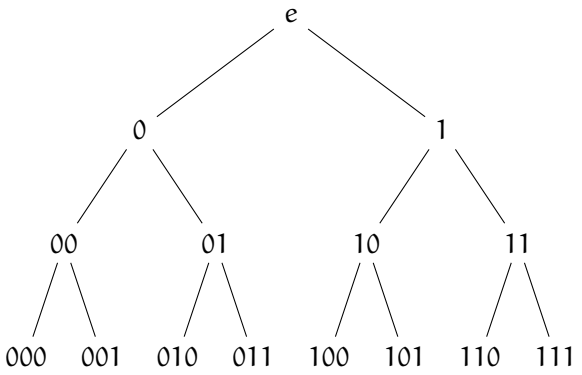


lar vi $\mathit{infiks}(T) = (\mathit{infiks}(T_1) + \mathit{infiks}(T_2))$.

- De to andre funksjonene er definert på samme måte for bladnoder, og på følgende måte for sammensatte trær.
  - $\mathtt{prefiks}(T) = +\mathtt{prefiks}(T_1)\mathtt{prefiks}(T_2)$
  - $\mathtt{postfiks}(T) = \mathtt{postfiks}(T_1)\mathtt{postfiks}(T_2)+$

Bitsekvenser og binære trær

- Det er en intim sammenheng mellom bitsekvenser og binære trær.
- For hver node i et binært tre kan vi lage oss en tilsvarende bitsekvens ved rekursjon på nivået (avstanden til roten) til noden.
- La $\mathtt{bit}(*)$ være den tomme sekvensen hvis $*$ er rotnoden.
- Hvis a er en node med to barn, b til venstre og c til høyre, og $\mathtt{bit}(a) = x_1 \cdots x_k$, lar vi
  - $\mathtt{bit}(b) = x_1 \cdots x_k 0$.
  - $\mathtt{bit}(c) = x_1 \cdots x_k 1$.
- Denne definisjonen illustreres på tavla.
- Omvendt vil en endelig mengde X av bitsekvenser, eller 0-1-sekvenser, bestemme et binært tre hvor vi først ser på alle delsekvenser av sekvensene i X, så lar bladnodene være de minimale bitsekvensene som ikke er delsekvens av noen sekvens i X og til slutt organiserer dette til et tre ved å la den tomme sekvensen bli roten, og så gå til venstre eller høyre avhengig av om neste bit er 0 eller 1.
- Vi illustrerer denne konstruksjonen på tavla.
- Hvis vi markere nodene i et binært tre med bitsekvensene, så får vi følgende bilde.



Litt om strømmer

- En digital strøm er en uendelig følge $\{x_n\}_{n\in\mathbb{N}}$ hvor hver x_i er en bit, markert som 0 eller 1.
- En digital strøm kan oppfattes som en strøm av data på digital form.
- Anta at vi har en prosedyre hvor input kan være en digital strøm og hvor output er en eller annen melding på digital form.
- Det vil finnes situasjoner hvor vi aldri får noe output hvis input er spesielt ekle digitale strømmer, men normalt vil vi at prosedyren skal avsløre om den digitale strømmen vi mottar er uten interesse, og skal avslutte med en melding om det.

- Vi tenker oss altså en situasjon hvor prosedyren avslutter med et svar uansett hvilken strøm den fores med.
- For enhver strøm finnes det da en endelig del som er stor nok til at prosedyren vår kan gi et output på grunnlag av denne.
- Det er fordi prosedyren vår bare kan utnytte endelig mye informasjon om hver enkelt strøm.
- La T være treet av endelige bitsekvenser som er så små at prosedyren vår ikke har nok grunnlag i disse til å gi et output.
- Er T et endelig tre?
- Vi skal vise at det er tilfelle.
- Beviset er et eksempel på et kontrapositivt bevis, altså på et bevis hvor vi antar at konklusjonen er feil, og resonnerer oss frem til at da er premissene feil.
- Anta derfor at treet er uendelig.
- Da må venstre deltre være uendelig eller høyre deltre være uendelig (eller begge).
- Start en digital strøm med 0 om venstre deltre er uendelig og med 1 om det er endelig. La T_1 være det tilsvarende uendelige deltreet.
- Fortsett strømmen med 0 om venstre deltre i T_1 er uendelig og med 1 om det er endelig (da er høyre deltre i T_1 uendelig).
- Slik fortsetter vi ved å gå til venstre når deltreet i den retningen er uendelig, og til høyre når det er nødvendig for fortsatt å ha et uendelig deltre.
- På den måten bygger vi opp en digital strøm som prosedyren vår ikke kan gi noe output fra, for da ville den gjøre det fra en endelig del av strømmen.
- Vi har imidlertid sørget for at enhver endelig del av den strømmen vi konstruerer, ligger i T, og derfor er utilstrekkelig for dette.
- Påstanden vi nå har vist, har den praktiske konsekvensen at hvis vi først har greid å lage en prosedyre som gir et svar uansett hvilken digital strøm vi forer den med, så finnes det en øvre grense for hvor lenge vi må vente på et svar, uavhengig av hva input er.
- Dette er et eksempel på en påstand hvor vi må gi et indirekte bevis, eller i det minste gå utenom den konstruktive delen av matematikken.
- Dette er ikke noe tema i MAT1030, og vi skal ikke forfølge dette aspektet videre.

MAT1030 – Forelesning 27

Trær

Dag Normann - 4. mai 2010

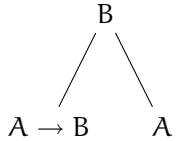
Forelesning 27

Oppsummering fra sist

- Forrige onsdag så vi på binære trær.
- For oss er syntakstrær viktige eksempler på binære trær.
- Vi så på forskjellige måter å traversere et binært tre på.
- Disse brukte vi til å representere termer via syntakstrær og de tre vanlige notasjonsformene for termer:
  1. *Infix* eller vanlig notasjon.
  2. *Prefix* eller polsk notasjon.
  3. *Postfix* eller baklengs polsk notasjon.
- Vi så på de binære trærne som en induktivt definert mengde, og på det tilhørende prinsippet for definisjoner ved rekursjon.
- Vi oppfattet prosedyrene som skriver ut de tre notasjonsformene fra syntakstreet som eksempler på trerekursjon.
- Det er meningen at dere skal kunne finne et syntakstre fra en formel eller en term, og at dere skal kunne skrive formelen eller termen med de tre notasjonsformene.
- Vi vil anvende dette tankegodset når vi kommer til unifiseringsalgoritmer.
- Først skal vi se på en annen type trær.

Bevistrær

- En annen type trær som spiller en stor rolle i logikk, men også i informatikk, er bevistrærne.
- Et bevistre er et merket tre hvor hver node er merket med en formel.
- Bladnodene vil være opplagt sanne, aksiomer i en eller annen formalisert teori, og merket til en foreldrenode vil følge logisk fra merkene til barna, ut fra visse prinsipper.
- En mulighet er at vi kan få lov til å ha deler av et bevistre som ser ut som



  som uttrykker at hvis vi har bevist A og $A \to B$, så kan vi konkludere B.
- Et slikt bevistre vil da være en garantist for at formelen som merker roten må være en konsekvens av de aksiomene som er brukt.

- Vi skal ikke la dette utvikle seg til et kurs i logikk, eller bevisteori, men som et eksempel på bruk av trær, skal vi se hvordan vi kan finne et bevis for et utsagnslogisk uttrykk, et tre som vil være en garantist for at uttrykket er en tautologi.
- Det er et poeng at hvis uttrykket er på *svak normalform*, så har vi en prosedyre for å omforme syntakstreet til et forsøksvis bevistre, og alle bladene blir aksiomer nøyaktig når utgangspunktet var en tautologi.
- Vi minner om at et utsagnslogisk uttrykk er på svak normalform hvis vi har følgende.
  - Kun bindeordene $\wedge$, $\vee$ og $\neg$ brukes.
  - $\neg$ kan kun stå rett foran en utsagnsvariabel.
- Som eksempler på rekursive konstruksjoner som går ut over rekursjon over $\mathbb{N}$ så vi på hvordan vi systematisk kan
  - fjerne forekomster av $\rightarrow$ og $\leftrightarrow$ i et utsagn slik at vi får et ekvivalent utsagn med bare $\wedge$, $\vee$ og $\neg$, og
  - som simultanrekursjon, skrive om utsagnene A og $\neg A$ til svak normalform.
- Disse konstruksjonene kan også formuleres ved hjelp av rekursjon på syntakstrær.
- Siden vi begrenser oss til utsagn på svak normalform, kan vi ikke bruke den regelen vi ga eksempel på, ettersom $\rightarrow$ ikke inngår i vokabularet.
- Vi skal tillate en type forgrening, eller slutningsregel.

$$\begin{array}{ccc} & A_1 \vee \cdots \vee A_n \vee (B \wedge C) & \\ \swarrow & & \searrow \\ A_1 \vee \cdots \vee A_n \vee B & & A_1 \vee \cdots \vee A_n \vee C \end{array}$$

- Vi skal ikke la rekkefølgen av delutsagnene i et $\vee$-utsagn bety noe.
- De eneste bladnodene vi vil akseptere er noder med merkene

$$A_1 \vee \cdots \vee A_n \vee p \vee \neg p$$

  hvor p er en utsagnsvariabel,

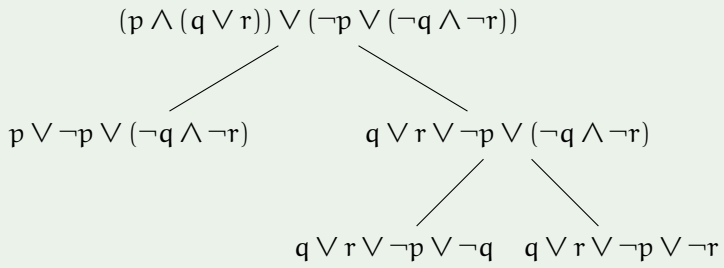
  altså disjunksjoner som inneholder både en utsagnsvariabel og negasjonen dens.
- Slike disjunksjoner er opplagt tautologier, og vil tjene som aksiomer.
- Vi skal se på et par eksempler hvor vi starter med et utsagn og utvikler et bevistre for dette utsagnet.
- Eksemplene forklares på tavla, om nødvendig.

Eksempel.

$$\begin{array}{ccc} & (p \wedge q) \vee (\neg p \vee \neg q) & \\ \swarrow & & \searrow \\ p \vee \neg p \vee \neg q & & q \vee \neg p \vee \neg q \end{array}$$

- Når vi bruker den distributive loven på $p \wedge q$ får vi en konjunksjon av to opplagte tautologier.
- Tautologier på denne formen har vi kalt aksiomer.
- Resultatet blir et bevistre.

Eksempel.



La oss nå se på prosedyren for å lage et forsøksvis bevistre fra et utsagnslogisk uttrykk.

Hvis vi har et utsagn A på svak normalform, kan A skrives på formen

$$A = A_1 \vee \cdots \vee A_n$$

hvor $n \geqslant 1$ og hver A_i enten er en konjunksjon, en utsagnsvariabel p eller negasjonen $\neg p$ av en utsagnsvariabel.

Hvis det finnes i og j slik at A_i er en utsagnsvariabel, og A_j er negasjonen av den samme, lar vi roten være bladnode merket med A, og vi har et bevistre.

Hvis alle A_i er utsagnsvariable eller negasjonen av slike, men vi er ikke i situasjonen over, lar vi roten være en bladnode, men konkluderer med at vi ikke har noe bevistre.

Ellers går vi til den minste i slik at $A_i = C \wedge D$.

Da lager vi to barn, hvor vi erstatter A_i med C når vi bygger treet videre under det ene barnet og vi erstatter A_i med D når vi bygger treet videre under det andre barnet.

Hvis begge disse deltrærne blir bevistrær, har vi konstruert et bevistre, ellers har vi ikke gjort det.

Vi illustrerer prosedyren på utsagnet

$$(\neg p \vee (p \wedge (p \vee q))) \wedge (p \vee \neg q \vee (q \wedge \neg p))$$

på tavla.

- Algoritmen for å finne et bevistre, om mulig, kan utvides til utsagn med kvantorer.
- Da er det ikke sikkert at vi alltid får et svar på om det finnes et bevis.
- Formelle bevis spiller en viss rolle om man ønsker å trekke ut numerisk informasjon rundt det beviste utsagnet.
- Formelle bevis spiller også en rolle i utviklingen av automatiserte korrekthetsbevis for programmer.
- Det neste temaet spiller også en rolle i anvendelser av logikk innen IT.

Unifisering

- Vi skal avslutte stoffet om trær med å se på en teknikk som kalles unifisering.
- Hvis vi har to termer hvor det forekommer variable, er det da mulig å erstatte disse variablene med andre termer slik at resultatene blir like?
- Vi skal begrense oss til et enkelt tilfelle, men den generelle unifiseringsalgoritmen spiller en stor rolle i logikkprogrammering og i automatisk bevissøk.

Eksempel.

La

$$t = (x + 0) \times ((0 + 0) \times x)$$
$$s = ((1 + 0) + 0) \times (y \times (1 + 0))$$

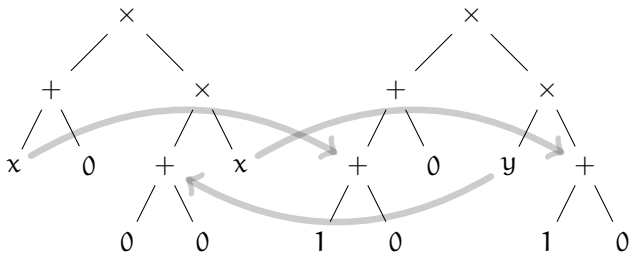
Er det mulig å erstatte x og y med termer slik at de to uttrykkene blir syntaktisk like?

I dette tilfellet kan vi sette inn $1 + 0$ for x og $0 + 0$ for y og begge uttrykkene blir

$$((1 + 0) + 0) \times ((0 + 0) \times (1 + 0)).$$

Dette ser vi lettest ved å betrakte syntakstrærne:

$$t = (x + 0) \times ((0 + 0) \times x)$$
$$s = ((1 + 0) + 0) \times (y \times (1 + 0))$$



- $x = (1 + 0)$
- $y = (0 + 0)$

Eksempel.

Nå lar vi

$$t = (x + 0) \times ((0 + 0) \times x)$$
$$s = ((1 + 0) + 0) \times (y \times (1 + y))$$

Hvis vi skal unifisere disse to uttrykkene, det vil si erstatte x og y med termer slik at s og t blir like, må vi få til at

$x + 0$ og $((1 + 0) + 0)$ blir like

$(0 + 0) \times x$ og $y \times (1 + y)$ blir like

samtidig.

Fra den første linjen ser vi at vi må sette inn $1 + 0$ for x, og hvis vi gjør det i den andre linjen, reduserer vi problemet vårt til å finne y slik at $y \times (1 + y)$ og $(0 + 0) \times (1 + 0)$ blir syntaktisk like.

Vi ser direkte at det er umulig.

Eksempel.

- Kan vi unifisere termene

$$(x \times (1 + 0)) + (((0 + 1) + z) \times (1 + x))$$

og

$$((0 + 1) \times (z + 0)) + ((y + 1) \times (1 + x))$$

det vil si, kan vi finne andre termer vi kan sette inn for x, y og z slik at de to uttrykkene blir like?

- Dette kan vi gjøre ved å sammenlikne termene systematisk og se om vi får fremtvunget hva vi skal erstatte x, y og z med for at resultatet skal bli vellykket.

Eksempel (Fortsatt).

- Først ser vi at begge termene har $+$ som hovedsymbol (dette ser vi lettere ut fra syntakstreet), og skal vi unifisere termene, må vi unifisere termene

  $(x \times (1 + 0))$ og $((0 + 1) \times (z + 0))$

  $(((0 + 1) + z) \times (1 + x))$ og $((y + 1) \times (1 + x))$

  simultant, det vil si vi må sette inn de samme termene for x, y og z i begge tilfellene.

Eksempel (Fortsatt).

Vi ser at begge termene i det første paret er produkttermer og det samme gjelder for begge termene i det andre paret. Oppgaven blir derfor å unifisere alle disse fire parene simultant:

1. x og $0 + 1$
2. $1 + 0$ og $z + 0$
3. $(0 + 1) + z$ og $y + 1$

4. $1 + x$ og $1 + x$

Vi ser at linje 1 forteller oss at vi må sette inn $0 + 1$ for x og i linje 4 har vi to like termer. De to andre linjene løser seg opp i fire nye linjer:

Eksempel (Fortsatt).

Vi må kunne unifisere

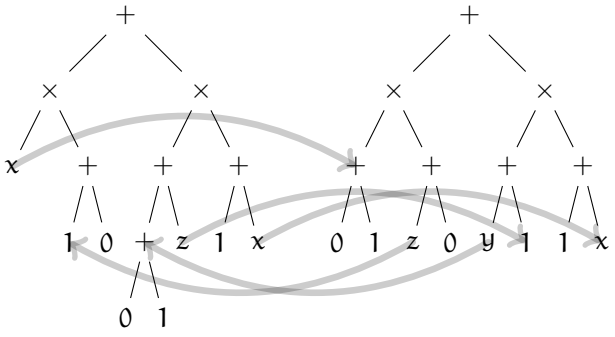
1. 1 og z
2. 0 og 0
3. $0 + 1$ og y
4. z og 1

simultant.

Nå har vi nådd bunnen av det som blir rekursjonen, og vi ser at hvis vi setter inn 1 for z og $0 + 1$ for y, i tillegg til at vi skulle sette inn $0 + 1$ for x, så får vi unifisering av de to første termene.

$$(x \times (1 + 0)) + (((0 + 1) + z) \times (1 + x))$$

$$((0 + 1) \times (z + 0)) + ((y + 1) \times (1 + x))$$



- $x = (0 + 1)$
- $z = 1$
- $y = (0 + 1)$

- Vi skal se på flere illustrerende eksempler før vi beskriver den endelige algoritmen for unifisering.
- Unifiseringen vil være mislykket hvis vi på et trinn må unifisere to forskjellige termer uten fri variable.
- Det er hele tiden viktig å tenke på at det er de syntaktiske uttrykkene som skal bli like, ikke bare de numeriske verdiene.

- For at en maskin skal kunne teste om en overgang er logisk korrekt, må uttrykk som spiller rollen som utsagnsvariable være syntaktisk like.
- Unifisering spiller en viktig rolle i logikkprogrammering, for eksempel i programmeringsspråk som PROLOG.

Eksempel.

- Unifiser $(1 + x) + (y + z)$ og $(1 + y) + (x + 0)$.
- Dette krever, i to trinn, at vi skal kunne unifisere
  1. 1 og 1
  2. x og y
  3. y og x
  4. z og 0

  simultant.
- Bare vi setter inn den samme termen for x, y og vi setter inn 0 for z, får vi en unifisering uansett.
- Vi vil derfor falle ned på den mest generelle unifiseringen, hvor vi beholder en av variablene x og y.
- Svaret blir derfor $(1 + x) + (x + 0)$

Eksempel.

- Vi skal unifisere $(1 + x) + (1 + y)$ og $(1 + 0) + (1 + (x + 1))$
- Dette reduseres til oppgaven å skulle unifisere
  1. $1 + x$ og $1 + 0$
  2. $1 + y$ og $1 + (x + 1)$

  simultant.

Eksempel (Fortsatt).

- Disse løser seg opp i fire oppgaver om å unifisere
  1. 1 og 1
  2. x og 0
  3. 1 og 1
  4. y og $x + 1$

  simultant.
- Linjene 1 og 3 er uproblematisk.
- I linje 2 ser vi at vi må sette inn 0 for x

- Det betyr at vi må omforme linje 4 til å skulle unifisere y og $0 + 1$, noe vi kan gjøre på direkten.

Eksempel.

- Anta at vi skal prøve å unifisere $x + (y + 0)$ og $(x + 1) + (1 + 0)$.
- Da må vi kunne unifisere
  1. x og $x + 1$
  2. $(y + 0)$ og $(1 + 0)$

  simultant.
- Andre linje er uproblematisk, men første linje er umulig, det finnes ingen term t slik at t og $t + 1$ er syntaktisk like.
- Hvis vi i forsøket på å unifisere par av termer må erstatte en variabel med en større term hvor variabelen forekommer, må vi konkludere med at unifisering er umulig.
- Ser vi på eksemplet $x + x$ og $y + (y + 1)$ reduseres det til samme type umulighet.

- Vi skal nå beskrive en rekursiv prosess som avgjør om det er mulig simultant å unifisere en endelig mengde par av termer i språket vårt.
- Etter at vi har gjort det, skal vi se på et eksempel på hvor vi kan få bruk for unifisering, og hvordan vi må tilpasse en konkret situasjon til en som kan håndteres av algoritmen vår.
- Vi vil starte med en endelig liste $x_1, \ldots, x_k$ av variable og to lister $t_1, \ldots, t_n$ og $s_1, \ldots, s_n$ av termer hvor disse variablene kan forekomme, og vi vil bestemme om det er mulig å erstatte variablene med termer $r_1, \ldots, r_k$ slik at hver t_i blir lik sin makker s_i.
- Får vi til det, sier vi at vi unifiserer parene simultant.
- Hvis t_1 er en variabel x_i og x_i forekommer i s_1, men s_1 er mer kompleks, gi opp unifiseringen. Det samme gjelder om situasjonen er omvendt.
- Hvis t_1 er variabelen x_i og x_i ikke forekommer i s_1, noterer vi at vi skal bruke sluttverdien av s_1 som termen r_i som skal erstatte x_i.

  Deretter erstatter vi alle andre forekomster av x_i med s_1 og fortsetter unifiseringsalgoritmen.

  I dette tilfellet har vi oppnådd å redusere antall variable med 1.

  Et korrekthetsbevis for algoritmen vil i første omgang være ved induksjon over antall variable.
- Hvis s_1 er en variabel, mens t_1 ikke er det, fortsetter vi på tilsvarende måte.
- Hvis s_1 og t_1 er samme term, bare stryker vi dette paret fra listen og fortsetter.
- Hvis ingen av tilfellene over gjelder har vi to muligheter:
  - s_1 og t_1 er åpenbart forskjellige, eksempelvis ved at den ene er en sum og den andre et produkt, den ene er et tall mens den andre er et funksjonsuttrykk eller de er forskjellige tall.

I dette tilfellet konkluderer vi med at unifisering er umulig.

– De er begge en sum eller de er begge et produkt.

Da erstatter vi paret s_1, t_1 med to par av mindre uttrykk, paret av de første addendene (faktorene) og paret av de andre addendene (faktorene) slik vi har sett eksempler på.

Deretter fortsetter vi algoritmen fra start.

- Hvis vi ikke finner ut underveis at unifisering er umulig, vil denne fremgangsmåten finne frem til de mest generelle termene $r_1, \ldots, r_k$ vi kan erstatte $x_1, \ldots, x_n$ med for å unifisere alle parene simultant.
- Korrekthetsbeviset er ved induksjon over antallet k av variable, med en underinduksjon over antall symboler sammenlagt i de to listene (dette antallet kan øke når vi kvitter oss med en variabel).
- Det finnes andre, og i praksis mer effektive, måter å gjennonføre unifisering på; hovedpoenget her var å vise prinsippene bak algoritmen.

  Det er ikke meningen at dere skal kunne gjengi denne algoritmen, men at dere skal kunne bestemme for hånd, i forholdsvis enkle tilfeller, om termer lar seg unifisere, og i tilfelle, gjennomføre det.

  Det kan i det minste være en fordel til eksamen å vite hva unifisering innebærer.

  Nå skal vi svare på spørsmålet om hva dette skal være godt for.

  Som tidligere nevnt spiller unifisering en rolle i automatisk bevissøk, eksempelvis i forbindelse med PROLOG.

  Vi skal se på et eksempel på hvordan vi systematisk kan søke etter et bevis for en påstand i et veldig enkelt logisk system.

  Eksemplet er så enkelt at vi ikke trenger hjelp av datamaskin til å finne et bevis, så det er mest til informasjon og motivasjon.

- Anta at vi har to aksiomer som vi kan bruke til å bevise at enkelte termer beskriver mindre tall enn andre termer:
  – $x < x + 1$
  – $x < y \wedge y < z \rightarrow x < z$.
- Så ønsker vi å søke etter et bevis for at

$$1 < ((1 + 1) + 1) + 1.$$

- Vi kan ikke finne en unifisering med aksiom 1, så håpet må være at vi har kommet frem til denne ulikheten som en anvendelse av aksiom 2.
- PROLOG vil da unifisere problemet vårt med konklusjonen i aksiom 2, slik at vi får $x = 1$ og $z = ((1 + 1) + 1) + 1$.
- Dette følger fra $1 < y \wedge y < ((1 + 1) + 1) + 1$, og PROLOG vil lete etter en verdi for y slik at begge delene av denne konjunksjonen kan bevises.
- $1 < y$ lar seg unifisere med aksiomet $x < x + 1$ ved å la $x = 1$ og $y = 1 + 1$.
- Prøver vi denne veien, må vi også prøve å bevise den andre delen av konjunksjonen for denne verdien av y, nemlig at $1 + 1 < ((1 + 1) + 1) + 1$.
- Igjen ser vi at dette ikke kommer direkte fra aksiom 1, så skal vi kunne bevise denne påstanden, må det være som en konsekvens av aksiom 2 og et bevis for en instans av

$$1 + 1 < y \wedge y < ((1 + 1) + 1) + 1.$$

- Setter vi inn $(1 + 1) + 1$ for y, i et forsøk på å la første del av denne konjunksjonen være en direkte konsekvens av aksiom 1, ser vi at også andre del blir en direkte konsekvens av aksiom 1.
- Vi har dermed systematisk lett oss frem til et bevis for den opprinnelige ulikheten.
- I virkelighetens verden kan vi trenge mer komplekse unifiseringer når vi prøver å finne bevis for påstander, og det å organisere søket på en slik måte at vi ofte raskt finner bevis der de finnes er et viktig teknologiskaspekt.
- Med dette gir vi oss med unifisering.
- Dette avslutter også innføringen i grafer og trær.

MAT1030 – Forelesning 28

Kompleksitetsteori

Dag Normann - 5. mai 2010

Forelesning 28: Kompleksitetsteori

Introduksjon

- Da er vi klare (?) for siste kapittel, om kompleksitetsteori!
- I denne siste delen skal vi spørre oss om det er mulig å måle hvor lang tid det tar å utføre en algoritme og hva en slik måling innebærer.
- Det vil være interessant å vite om en gitt algoritme kan utføres av en datamaskin innen rimelig tid, og hvis vi har to algoritmer som skal løse den samme oppgaven, om den ene er raskere enn den andre.
- Før vi diskuterer hva disse spørsmålene kan bety, la oss se på et eksempel på dårlig og god programmering for å løse en oppgave.

Et eksempel

- Årets foreleser i MAT1030 fikk en gang en telefon fra en person som ville starte et lotteri.
- Det skulle trykkes tre og en halv millioner lodd, med nummer fra 1 til 3.500.000.
- Tanken var at man skulle utbetale ekstragevinster for enkelte tverrsummer av loddnummeret.
- Tverrsummen vil være et tall mellom 1 og 56.
- Ønsket var at en matematiker skulle hjelpe til med å finne ut hvor mange lodd som vil ha tverrsum i når i varierer fra 1 til 56.
- Et første forsøk på å imøtekomme ønsket var å la en stipendiat skrive et LISP-program som realiserte følgende algoritme.

1. **For** $i = 1$ **to** 56 **do**
   1.1. $x_i \leftarrow 0$
2. **For** $n = 1$ **to** $3.500.000$ **do**
   2.1. $i \leftarrow \text{tverrsum}(n)$
   2.2. $x_i \leftarrow x_i + 1$
3. *Output* $x_1, \ldots, x_{56}$.

- Etter en god lunsj ble det klart at dette ville ta hele dagen, og trolig hele natten.
- Vi måtte finne en raskere algoritme.
- Vi laget en algoritme basert på en matematisk analyse av problemet.
- Analysen og programmeringen tok ca. 10 minutter.
- Den nye algoritmen ga svaret i løpet av ca. 2 sekunder.
- Vi ser kort på tankegangen bak den nye algoritmen.

- Vi kjenner til hvordan tverrsummene fordeler seg for tall fra 0 til 9.
- Tverrsummene for tall fra 10 til 19 fordeler seg nesten likt, bare forskjøvet et tall oppover.
- Tilsvarende kan vi lett finne fordelingen av tverrsummene for tall mellom 20 og 29, mellom 30 og 39 osv.
- Legger vi sammen får vi fordelingen av tverrsummer for alle tall mellom 0 og 99.
- Forskyver vi denne fordelingen ett tall opp, får vi fordelingen av tverrsummer mellom 100 og 199.
- Ti nye runder gir oss altså fordelingen av tverrsummene for alle tall mellom 0 og 999.
- Ti nye runder gir oss fordelingen opp til 9.999, osv.
- Tilslutt må vi justere tallene litt slik at vi får fordelingen av tverrsummer for tall mellom 1 og 3.500.000.
- Som en generell algoritme ser vi at antall regneoperasjoner er proporsjonalt med antall siffre, og ikke med antall lodd.
- Vi skal komme tilbake til slike fenomener senere, og sette navn på dem.

Kompleksitetsteori

- I kompleksitetsteori er det ofte to størrelser man prøver å finne
  - Hvor lang tid tar det å følge en algoritme.
  - Hvor mye lagringsplass må man sette av for at algoritmen skal ha den informasjonen den trenger til enhver tid.
- Vi skal konsentrere oss om tidskompleksitet (engelsk: *time complexity*) og la plasskompleksitet (engelsk: *space complexity*) være udiskutert.
- Vi skal ikke ta sikte på å gi en innføring i kompleksitetsteori som en del av den teoretiske informatikken, men at dere etter endt MAT1030 kan:
  a) Vurdere to algoritmer mot hverandre for å kunne vurdere hvilken som vil være mest tidseffektiv.
  b) Vurdere om en algoritme er gjennomførbar innen akseptabel tid for input av den størrelsen man ønsker at den skal virke for.
- Hvis vi har skrevet en algoritme på pseudokodeform, er spørsmålet om hvor lang tid det tar å følge algoritmen et upresist spørsmål av flere grunner:
  - Svaret avhenger av hvilket programmeringsspråk vi benytter.
  - Svaret avhenger av hvilken maskin vi benytter.
  - Svaret avhenger av hvor stort minne vi har satt av.
  - Svaret avhenger av om vi må dele ressursene med andre.
  - Svaret avhenger av input og hvordan input representeres digitalt.
- Vi trenger ikke å kjenne alle disse forholdene for å kunne sammenlikne algoritmer eller vurdere om en algoritme er praktisk gjennomførbar.
- Vi skal lære å se bort fra det uvesentlige, og derigjennom få et grunnlag for å vurdere den omtrentlige kompleksiteten av en algoritme.
- Det vil være den omtrentlige tidsbruken som funksjon av størrelsen på input vi vil være på jakt etter.

Første tilnærming

Eksempel.

1 *Input* n [n naturlig tall]
2 $x \leftarrow 1$
3 **For** $i = 2$ **to** n **do**
  3.1 $x \leftarrow x \cdot i$
4 $y \leftarrow 0$
5 **For** $j = 1$ **to** x **do**
  5.1 $y \leftarrow y + j$
6 *Output* y

Eksempel (Fortsatt).

- Vi har gitt en pseudokode for å beregne

$$f(n) = \sum_{j=1}^{n!} j.$$

- Vi må anta at det normalt krever mer tid å multiplisere to tall enn å summere dem.
- På den annen side skal vi utføre $n - 1$ multiplikasjoner i den første delen, mens vi skal utføre $n! - 1$ addisjoner i andre del.
- n skal ikke være så veldig stor før andre del av algoritmen tar vesentlig lengere tid å utføre enn første del.
- Det er i andre del at kompleksiteten ligger.

Eksempel (Fortsatt).

1 *Input* n [n naturlig tall]
2 $x \leftarrow 1$
3 **For** $i = 2$ **to** n **do**
  3.1 $x \leftarrow x \cdot i$
4 $y \leftarrow \frac{x(x+1)}{2}$
5 *Output* y

Eksempel (Fortsatt).

- Den nye pseudokoden gir oss nøyaktig den samme funksjonen.
- Her vil fortsatt den første delen innebære at vi må foreta $n-1$ multiplikasjoner, mens den andre delen innebærer essensielt en multiplikasjon og en divisjon med 2.
- Nå er det den første delen som vil være mest tidkrevende.
- Vi har funnet en raskere algoritme for å beregne den samme funksjonen.

- Lærebokas første tilnærming til kompleksiteten av en algoritme lyder, oversatt til norsk:

  Tell bare de mest tidkrevende operasjonene.
- En operasjon kan være en enkel regneoperasjon, en **for**-løkke, en sammenlikning av størrelser, en annen form for løkke eller noe annet.
- Hvis løkker inngår i algoritmen, vil ofte lengden på løkkene bestemme hvor tidkrevende algoritmen er.
- Det kan derfor være lurt, slik vi gjorde i eksemplet, å studere lengden på de enkelte løkkene.
- Mange addisjoner kan overskygge langt færre multiplikasjoner, selv om det er mer tidkrevende å utføre en multiplikasjon enn en addisjon.
- Hvis koden inneholder **while**-løkker eller **until**-løkker, kan det være vanskelig å sammenlikne tidsbruken med tidsbruken til andre løkker.
- La oss se på Prims algoritme i lys av første tilnærming.
- I Prims algoritme har vi listet opp nodene i en vektet graf, og så har vi listet opp kantene i grafen sammen med sine vekter.
- I Prims algoritme har vi en hovedløkke hvor vi i løpet av løkka legger en ny kant til det utspennende treet.
- I skritt nr. i skal vi ta for oss hver av de $n-i$ nodene som ikke har kommet med i treet, se på alle kantene fra disse nodene til treet bygget så langt og plukke ut den av disse kantene som har minst vekt.
- Den mest tidkrevende enkeltoperasjonen vil være å vurdere om en kant er kandidat til å bli lagt til treet, samt å sammenlikne vekten av hver enkelt kant med vekten til en tidligere utplukket kandidat.
- Vi skal senere komme med en måte å formulere omtrent hvor mange slike grunnoperasjoner vi må utføre.

Andre tilnærming

Eksempel.

1 *Input* n [n naturlig tall]
2 *Input* $x_{n-1} \cdots x_1$ [Hver x_i lik 0 eller 1]
3 $x_n \leftarrow 0$
4 $i \leftarrow 1$

5 **While** $x_i = 1$ **do**

5.1 $x_i \leftarrow 0$

5.2 $i \leftarrow i + 1$

6 $x_i \leftarrow 1$

7 *Output* $x_n \cdots x_1$

Eksempel (Fortsatt).

- Denne pseudokoden gir en algoritme for å legge 1 til det binære tallet $x_{n-1} \cdots x_1$.
- Hvis vi starter med $n = 20$ og det binære tallet 11111111111111111111, vil **while**-løkka gjentas nitten ganger, og vi tester om den skal brukes 20 ganger.
- Hvis vi starter med det binære tallet 11111111111111111110 utfører vi testen for **while**-løkka bare en gang.
- Siden den eneste kontrollen vi har over hvor mange ganger denne løkka må gjentas er antall siffre i det binære tallet, lar vi det være målet på hvor lang tid vi bruker.

- For endel algoritmer vil tiden vi bruker kunne avhenge av om vi er heldige med valg av input eller ikke.
- Når vi skal vurdere kompleksiteten til en algoritme, kan det ofte være hensiktsmessig å vurdere tidsbruken i de verste tilfellene.
- Det er dette læreboka setter opp som tilnærming nr 2, etter at man har vurdert hvilken del av programmet det er som overskygger de andre delene i tidsbruk:

Hvis tidsbruken varierer for forskjellige input av samme størrelse, ta utgangspunkt i det verste tilfellet.

Eksempel.

- Vi har gitt en sammenhengende graf og skal avgjøre om grafen har en Eulerkrets eller ikke.
- Når vi skal vurdere kompleksiteten av en algoritme, er det viktig hvordan vi representerer input.
- Her vil vi anta at grafen er gitt som en symmetrisk matrise, hvor tallet i rad i og kolonne j angir hvor mange kanter det er mellom nodene i og j.
- Vi antar som før at tallene på diagonalen er det dobbelte av antall løkker ved den tilsvarende noden.
- Graden til en node er da summen av alle tallene langs tilsvarende rad (eller søyle).

Eksempel (Fortsatt).

- Vi bestemmer om grafen har en Eulerkrets ved å summere tallene i hver rad til vi finner et oddetall.
- Har grafen en Eulerkrets, må vi summere tallene i alle radene, så hvis n er antall noder, må vi utføre $n(n-1)$ addisjoner og sjekke at n tall er partall.
- Hvis grafen ikke har en Eulerkrets kan vi slippe billig fra det og utføre bare $n-1$ addisjoner.
- Den dominerende prosessen i det verste tilfellet er det å summere tallene i alle radene, så det er de operasjonene vi legger til grunn når vi vurderer kompleksiteten.

Eksempel (Fortsatt).

- Anta nå at vi ikke visste at grafen var sammenhengende.
- Er det ødeleggende for kompleksiteten av problemet hvorvidt grafen har en Eulerkrets at vi må undersøke om den er sammenhengende?
- Vi kan uformelt beskrive en prosedyre som undersøker om en graf er sammenhengende på følgende måte:
  - Vi vil finne sammenhengskomponenten til node 1:
  - La A være $n \times n$-matrisen til G hvor $a_{i,j}$ er tallet i rad i og søyle j.
  - La $X_1 = \{1\}$
  - Ved rekursjon for $k < n$, la $X_{k+1} = X_k \cup \{j \leqslant n \mid \exists i \in X_k(a_{i,j} > 0)\}$.
  - G er sammenhengende hvis $X_n = \{1, \ldots, n\}$.

Eksempel (Fortsatt).

- I denne algoritmen har vi en hovedløkke i n trinn.
- Hvert trinn i løkka består av en gjennomløpning av alle par av noder, for å se om det finnes en kant som forbinder den ene noden med sammenhengskomponenten bygget opp så langt.
- Det gir n løkker hvor vi foretar n^2 tester i hver løkke.
- Tilsammen foretar vi n^3 tester.
- Det å undersøke om en graf er sammenhengende krever altså flere operasjoner enn det å undersøke om den har en Eulerkrets, når vi gjør det på denne måten.

Eksempel.

- Det neste eksemplet som skal belyse tilnærming 2 er Euklids algoritme.

- Euklids algoritme er en selvkallende algoritme som finner det største felles mål for to tall.
- Det største felles målet er det samme som den største felles faktoren.
- Hvis $n \geqslant m$ er to naturlige tall vil $\texttt{Euklid}(n, m)$ være
  - m hvis m er en faktor i n.
  - $\texttt{Euklid}(m, k)$ hvor k er resten når vi deler n på m når m ikke er en faktor i n.
- Euklids algoritme er rask, selv for store tall.

Eksempel (Fortsatt).

- Hvis vi følger Euklids algoritme for to tallpar som ligger nær hverandre ser vi at det likevel kan være forskjeller i hvor raskt algoritmen gir et svar.
  1. $(80, 32) \to (32, 16)$ som gir svar 16.
  2. $(81, 32) \to (32, 17) \to (17, 15) \to (15, 2) \to (2, 1)$ som gir svaret 1
- Hvordan skal vi så kunne finne de verste tilfellene?
- Følg med på den overraskende fortsettelsen!

Eksempel (Fortsatt).

- Det minste par av forskjellige tall som gir oss svaret med en gang er $(2, 1)$
- Det minste tallet > 2 som gir 1 som rest når vi deler det med 2 er $1 + 2 = 3$
- Det minste tallet > 3 som gir 2 som rest når vi deler det med 3 er $3 + 2 = 5$.
- Det minste tallet > 5 som gir 3 som rest når vi deler det med 5 er $5 + 3 = 8$

Eksempel (Fortsatt).

- Hvis vi begynner med et par av Fibonaccitall (F_{n+1}, F_n) vil Euklids algoritme gi oss paret (F_n, F_{n-1}) i neste omgang.
- Dette er de verste tilfellene, det vil si de tilfellene hvor vi bruker lengst tid i forhold til hvor store tallene er.
- Dette var neppe en anvendelse Fibonacci hadde i tankene, men hvem vet?

Tredje tilnærming

- Når vi skal vurdere om en algoritme er raskere enn en annen, er det ikke sikkert at det er relevant for alle input.

- Det kan lønne seg å benytte en algoritme som arbeider raskere for store input, der tiden vi bruker faktisk kan ha økonomisk betydning, selv om en annen algoritme er bedre for små input.
- Vi skal først illustrere dette ved å gå gjennom et eksempel i boka, ettersom dette eksemplet i seg selv er viktig.
- Det dreier seg om effektiv eksponensiering, det vil si, om en metode for raskt å kunne beregne store potenser av et tall.
- Eksemplet har samme verdi om vi regner potenser av reelle tall, naturlige tall eller hele tall, så det presiserer vi ikke.

Eksempel.

- Vi kan definere funksjonen $f(x, n) = x^n$ ved rekursjon som følger:
  - $x^0 = 1$
  - $x^{n+1} = x^n \cdot x$
- Skal vi bruke denne til å beregne 3^8 får vi følgende beregning:

Eksempel (Fortsatt).

1. $3^0 = 1$
2. $3^1 = 1 \cdot 3 = 3$
3. $3^2 = 3 \cdot 3 = 9$
4. $3^3 = 9 \cdot 3 = 27$
5. $3^4 = 27 \cdot 3 = 81$
6. $3^5 = 81 \cdot 3 = 243$
7. $3^6 = 243 \cdot 3 = 729$
8. $3^7 = 729 \cdot 3 = 2187$
9. $3^8 = 2187 \cdot 3 = 6561$

Eksempel (Fortsatt).

- En alternativ måte å beregne 3^8 på kan være:
  1. $3^2 = 3 \cdot 3 = 9$
  2. $3^4 = 3^2 \cdot 3^2 = 9 \cdot 9 = 81$
  3. $3^8 = 3^4 \cdot 3^4 = 81 \cdot 81 = 6561$
- Her bruker vi bare tre multiplikasjoner i motsetning til seks.
- Skulle vi beregnet 3^{16} ville vi etter den første metoden måtte utføre 8 nye multiplikasjoner, mens vi etter den nye metoden klarer oss med en til:

$$3^{16} = 3^8 \cdot 3^8 = 6561 \cdot 6561 = 43046721$$

- Dette går faktisk fortere, selv for hånd.
  (Eller gjør det det?)

- Med utgangspunkt i siste eksempel, skal vi nå beskrive to algoritmer for eksponensiering, og sammenlikne dem.
- Vi har sett på hvordan vi kan beregne x^1, x^2, x^4, x^8 og så videre ved gjentatt kvadrering.
- Hvordan skal vi for eksempel kunne utnytte dette til å beregne x^{13}?
- Vi vet at $x^{13} = x^8 \cdot x^4 \cdot x$
- Vi vet at 13, representert som binært tall, er 1101_2
- En strategi kan derfor være at vi beregner x, x^2, x^4 og x^8 samtidig som vi ser på binærrepresentasjonen av 13 for å se hvilke av disse tallene som skal inngå som et produkt i x^{13}.
- Siden 13 faktisk er gitt ved sin binære representasjon i en datamaskin, er dette veldig gunstig.
- Vi skal gi en fullstendig pseudokode for å beregne x^n når n er gitt på binær form, men først skal vi se på et eksempel.

Eksempel.

- Vi vil beregne 3^{22}
- $22 = 16 + 4 + 2$ så binærformen til 22 er 10110
- Vi vil beregne to følger:
  1. Den ene er 3, 3^2, 3^4, 3^8 og 3^{16} slik vi har sett før.
  2. Den andre er produktet av de tallene i den første følgen som inngår i 3^{22} etterhvert som vi kommer til dem.
- Vi ser på hvilke tallpar vi får underveis, og hvordan vi kommer frem til dem:
  1. $y_1 = 3$ og $z_1 = 1$ fordi siste siffer i 10110 er 0.
  2. $y_2 = 3 \cdot 3 = 9$ og $z_2 = 9 \cdot 1 = 9$
  3. $y_3 = 9 \cdot 9 = 81$ og $z_3 = 81 \cdot 9 = 279$
  4. $y_4 = 81 \cdot 81 = 6561$ og $z_4 = 279$
  5. $y_5 = 6561 \cdot 6561 = 43046721$ og $z_5 = 43046721 \cdot 279 = 12010035159$
- Svaret er 12010035159.

1 *Input* x [x et reelt tall]
2 *Input* k [k antall siffre i binærrepresentasjonen av n]
3 *Input* $b_k \cdots b_1$ [Binærrepresentasjonen av n]
4 $y \leftarrow x$

5 $z \leftarrow 1$

6 **For** $i = 1$ **to** k **do**

6.1 **If** $b_i = 1$ **then**

6.1.1 $z \leftarrow y \cdot z$

6.2 $y \leftarrow y \cdot y$

7 *Output* z

- Denne pseudokoden er litt anderledes enn den som står i boka.
- Skal vi beregne x^2 tar denne prosedyren litt mer tid enn den definert ved rekursjon, ettersom vi her får både å regne ut x^2 og $x^2 \cdot 1$, men for store n er denne algoritmen vesentlig raskere.
- Vi kan lage en "dum" algoritme som regner ut $x = 0$ ved rekursivt å multiplisere 0 med 2^n, på følgende måte.

Eksempel.

1 *Input* n [n naturlig tall]

2 $x \leftarrow 0$

3 **For** $i = 1$ **to** n **do**

3.1 $x \leftarrow 2x$

4 *Output* x

- Vi kan finne en annen "dum" algoritme som beregner den samme funksjonen.

Eksempel (Fortsatt).

1 *Input* n

2 $x \leftarrow \frac{3 \cdot 5 - 15}{n \cdot (n+1)}$

3 *Output* x.

- I det siste eksemplet må vi foreta fem regneoperasjoner, mens i det første eksemplet er antall regneoperasjoner avhengig av n.
- For små n vil den første algoritmen faktisk gi raskere svar, også fordi vi der kan arbeide med hele tall, mens vi må arbeide med flytende reelle tall i den andre algoritmen.
- For store input er imidlertid den andre, direkte metoden raskere enn den første.
- Ved å følge tredje tilnærming, stopper all diskusjon om hvilken av to dumme algoritmer som er best.
- Hvis input er lite, vil de fleste algoritmer gi oss et svar innen rimelig tid, og det spiller ikke så stor rolle hvilken algoritme vi velger hvis det er flere mulige.
- Hvis input er stort, kan en ineffektiv algoritme bruke ødeleggende mye mer tid enn en effektiv algoritme.

- Det er derfor at tidsbruken for store inputverdier er det mest interessante.
- Dette er samlet i tredje tilnærming:

Anta at input er stort

Måle kompleksitet med funksjoner

- Vi har sammenliknet algoritmer, og vi har drøftet kompleksitet i visse tilfeller, men vi har ikke sagt så mye om hva slags funksjoner vi vil bruke til å måle kompleksitet med.
- Data er gitt på digital form, og det er naturlig å måle størrelsen på input ut fra hvor mange bit som brukes til å representere input.
- Vi antar fra nå av at størrelsen på input måles i antall bit som brukes i representasjonen.

Eksempel.

- La oss gå tilbake til eksemplet om grafer og problemet om å avgjøre om en graf er sammenhengende eller ikke.
- Siden løkker og parallelle kanter ikke kan gjøre en graf mer sammenhengende, kan vi godt begrense dette problemet til enkle grafer, det vil si grafer uten løkker og parallelle kanter.
- Uten å gå i detalj, kan vi si at for å representere en enkel graf med n noder, trenger vi et antall bit begrenset av $k \cdot n^2$ hvor k er et tall uavhengig av n men avhengig av hvordan vi velger å representere grafen digitalt.

Eksempel (fortsatt).

- Snur vi dette, ser vi at hvis m er antall bit i input, er antall noder i grafen begrenset av et tall $a \cdot \sqrt{m}$ hvor a er en konstant uavhengig av m.
- Da vi lagde en prosedyre for å bestemme om en graf med n noder er sammenhengende eller ikke, forestilte vi oss en prosess i følgende trinn:
  1. Velg ut en node.
  2. I $n - 1$ runder, utvid noden til en maksimal sammenhengende delgraf, ved i hvert trinn å legge til de nye nodene som kan nås fra delgrafen bygget opp så langt ved å legge til en kant.
  3. Undersøk om det fins noder som ikke er med i sammenhengskomponenten.

Eksempel (fortsatt).

- I hvert skritt i hovedløkka, gikk vi gjennom alle kantene, for å se om en av endenodene lå i grafen konstruert så langt.
- Hvis input er på m bit, har vi ca. $m^{\frac{1}{2}}$ trinn i hovedløkka og vi må (i verste tilfelle) teste ca. $\frac{1}{2} \cdot m$ kanter.

- Siden vi opererer med cirkatall, vi skal se på de verste tilfellene og bare på den mest tidkrevende delen av algoritmen, får vi at tidsbruken er omtrent $m^{\frac{3}{2}}$ hvor m er antall bit i input.
- Vi skal etterhvert være litt mer presis i hva vi mener med "cirka".

Definisjon.
En polynomfunksjon er en funksjon på formen

$$f(n) = a_k n^k + a_{k-1} n^{k-1} + \cdots + a_1 n + a_0$$

Vi antar normalt at $a_k \neq 0$, og da er k graden til funksjonen.

- I noen tilfeller er det viktig å skille mellom polynomfunksjonen og polynomet, som er det definerende uttrykket.
  - Dette er ikke så viktig for oss.
- Hvis graden til en polynomfunksjon f er større enn graden til en annen funksjon g, vil $f(n) > g(n)$ bare n er stor nok.
- Det betyr at hvis kompleksiteten til to algoritmer er gitt ved polynomfunksjoner, kan vi bruke tilnærming 3 og bestemme hvilken som er den raskeste hvis gradene er forskjellige.

Eksempel.
- Vi har gitt et stort tall på binær form og vil undersøke om tallet er et av Fibonacci-tallene.
- Det gitte tallet er representert ved n bit.
- Vi setter av fire n-bits områder R_1, R_2, R_3 og R_4 hvor det gitte tallet ligger i R_1.
- Vi starter med å laste binærkoden til 1 i R_2 og binærrepresentasjonen til 2 i R_3
- Dette tar $n + n$ enkeltoperasjoner (siden vi vi må rydde R_2 og R_3 for søppel).

Eksempel (fortsatt).
- Deretter starter vi en løkke hvor vi
  1. Laster summen av tallene i R_2 og R_3 inn i R_4. Dette tar ca $2n$ regneskritt, siden vi må holde orden på eventuell mente.
  2. Sammenlikner verdien av R_1 og R_4. Er de like, svarer vi JA, er tallet i R_4 størst, svarer vi NEI og er tallet i R_1 fortsatt størst, fortsetter vi prosessen.
  3. Laster tallet i R_3 over i R_2 og deretter tallet i R_4 over i R_3 Dette tar ca $2n$ regneskritt.

- Antall ganger vi må gjennomføre denne løkka er tilnærmet proporsjonal med n ettersom Fibonaccitallene øker tilnærmet eksponensielt.
- Det betyr at vi kan bruke en annengradsfunksjon til å beskrive den omtrentlige tidsbruken, $a \cdot n$ løkker som hver bruker ca $b \cdot n$ regneskritt.

- I det forrige eksemplet så vi at hvis m er et tall gitt på binær form med n sifre, fines det en konstant c slik at antall regneskritt som skal til for å avgjøre om m er et Fibonaccitall eller ikke er begrenset av
$$f(n) = c \cdot n^2.$$
- Vi var ikke spesielt ivrige etter å finne en konkret verdi på c, av forskjellige grunner:
  1. c vil avhenge av hvilket språk vi bruker og faktisk av hvilken maskin vi bruker.
  2. Den virkelige tiden avhenger vel så mye av hvor kraftig maskinvare vi disponerer som hvor liten vi kan få verdien på c til å bli.
  3. Den teknologiske utviklingen gjør at selv store verdier for c er uten betydning for effekten av denne algoritmen.
- Det som ville hjulpet var om vi kunne bringe kompleksiteten ned fra, si $40 \cdot n^2$ til $1.000 \cdot n$.

MAT1030 – Forelesning 29

Kompleksitetsteori

Dag Normann - 11. mai 2010

Forelesning 29: Kompleksitetsteori

Oppsummering

- Forrige gang startet vi på kapitlet om kompleksitetsteori.
- Vi er interessert i å kunne si noe om hvor lang tid det tar å følge en algoritme.
- Målet er at vi skal kunne sammenlikne tidsbruken til forskjellige algoritmer, for å vurdere hvilken som er mest tidseffektiv.
- I tillegg skal vi kunne vurdere hvorvidt et program basert på en algoritme kan forventes å terminere for de ønskede input innen akseptabel tid.
- Vi følger boka og er i ferd med å se på fire viktige aspekter, eller *tilnærminger*, for å vurdere effektiviteten av en algoritme.

- Første tilnærming: Tell bare de mest tidkrevende operasjonene.
- Andre tilnærming: Hvis tidsbruken varierer for forskjellige input av samme størrelse, ta utgangspunkt i det verste tilfellet.
  - Vi ønsker å finne et *generelt* svar og ikke måtte kjøre algoritmen for alle mulige input.
  - For mange algoritmer avhenger tiden ofte av om vi er heldige med valg av input eller ikke.
  - Når vi skal vurdere kompleksiteten til en algoritme, så er det derfor hensiktsmessig å vurdere tidsbruken i de *verste* tilfellene.

- Det norske ordet "tilnærming" er normalt en grei oversettelse av det engelske "approximation", men det vil gi en riktigere intuisjon om vi erstattet det med forenkling.
- Målet med disse tilnærmingene er at det skal bli mulig å sammenlikne algoritmer, og da viser det seg at det er enkelte forenklinger som gir det mest nyttige bildet.
- Hvis vi oppfatter ordet tilnærming slik at det står for en tilnærmet beskrivelse av kompleksiteten til en algoritme, er dette noenlunde dekkende.
- Vi avsluttet forrige gang midt i et eksempel og tar opp det igjen nå.

Måle kompleksitet med funksjoner

- Vi startet med å se på polynomfunksjoner.
- Hvis graden til en polynomfunksjon f er større enn graden til en annen funksjon g, vil $f(n) > g(n)$ bare n er stor nok.
- Det betyr at hvis kompleksiteten til to algoritmer er gitt ved polynomfunksjoner, kan vi bruke tilnærming 3 og bestemme hvilken som er den raskeste hvis gradene er forskjellige.

Eksempel.

- Vi har gitt et stort tall på binær form og vil undersøke om tallet er et av Fibonacci-tallene.
- Det gitte tallet er representert ved n bit.
- Vi setter av fire n-bits områder R_1, R_2, R_3 og R_4 hvor det gitte tallet ligger i R_1.
- Vi starter med å laste binærkoden til 1 i R_2 og binærrepresentasjonen til 2 i R_3
- Dette tar $n + n$ enkeltoperasjoner (siden vi vi må rydde R_2 og R_3 for søppel).

Eksempel (fortsatt).

- Deretter starter vi en løkke hvor vi
  1. Laster summen av tallene i R_2 og R_3 inn i R_4. Dette tar ca $2n$ regneskritt, siden vi må holde orden på eventuell mente.
  2. Sammenlikner verdien av R_1 og R_4. Er de like, svarer vi JA, er tallet i R_4 størst, svarer vi NEI og er tallet i R_1 fortsatt størst, fortsetter vi prosessen.
  3. Laster tallet i R_3 over i R_2 og deretter tallet i R_4 over i R_3 Dette tar ca $2n$ regneskritt.
- Antall ganger vi må gjennomføre denne løkka er tilnærmet proporsjonal med n ettersom Fibonaccitallene øker tilnærmet eksponensielt.
- Det betyr at vi kan bruke en annengradsfunksjon til å beskrive den omtrentlige tidsbruken, $a \cdot n$ løkker som hver bruker ca $b \cdot n$ regneskritt.

- I det dette eksemplet så vi at hvis m er et tall gitt på binær form med n sifre, fines det en konstant c slik at antall regneskritt som skal til for å avgjøre om m er et Fibonaccitall eller ikke er begrenset av
$$f(n) = c \cdot n^2.$$
- Vi var ikke spesielt ivrige etter å finne en konkret verdi på c, av forskjellige grunner:
  1. c vil avhenge av hvilket språk vi bruker og faktisk av hvilken maskin vi bruker.
  2. Den virkelige tiden avhenger vel så mye av hvor kraftig maskinvare vi disponerer som hvor liten vi kan få verdien på c til å bli.
  3. Den teknologiske utviklingen gjør at selv store verdier for c er uten betydning for effekten av denne algoritmen.
- Det som ville hjulpet var om vi kunne bringe kompleksiteten ned fra, si $40 \cdot n^2$ til $1.000 \cdot n$.

Fjerde tilnærming og O-notasjon

- Fjerde tilnærming lyder:

  Vi skiller ikke mellom to tidskompleksiteter hvis vekstraten til den ene er et konstant multiplum av vekstraten til den andre.
- Etter at vi nå har innført fire prinsipper for tilnærminger, hvorav tre av dem er mer å betrakte som tommelfingerregler enn matematisk presise regler, skal vi innføre den såkalte O-notasjonen (ikke "null", men bokstaven O).

- Ved hjelp av den blir faktisk bruk av første, tredje og fjerde tilnærming presise.
- Den vil også gjøre det mer presist å avgjøre hva som faktisk er de verste tilfellene (andre tilnærming).

Definisjon.
Tidskompleksiteten til en algoritme er en funksjon f fra $\mathbb{N}$ til $\mathbb{N}$, slik at når argumentet n er størrelsen på input (målt i antall bit), så er verdien $f(n)$ er *det maksimale antallet av mest tidskrevende operasjoner utført* når størrelsen på input er n.

Definisjon.
La f og g være tidskompleksiteter, det vil si, funksjoner fra $\mathbb{N}$ til $\mathbb{N}$.
Vi sier at f er $O(g)$ hvis det fins en positiv konstant c slik at

$$f(n) \leqslant c \cdot g(n)$$

for alle tilstrekkelig store n.

- Med "tilstrekkelig store" mener vi at det fins en n_0 slik at ulikheten holder for alle $n \geqslant n_0$.
- Skulle vi gitt denne definisjonen mer presist, måtte vi bruke kvantorene vi lærte om tidligere i semesteret.
- Da ser definisjonen av at f er $O(g)$ slik ut:

$$\exists c > 0 \exists n_0 \forall n \geqslant n_0 (f(n) \leqslant c \cdot g(n)).$$

- Med denne notasjonen, og i lys av et eksempel vi har sett på før, kan vi si at tidskompleksiteten for den naturlige algoritmen som undersøker om en graf er sammenhengende og har en Eulerkrets er $O(n^{\frac{3}{2}})$, når n er antall bit vi trenger for å representere grafen.
- I motsetning til tidligere formuleringer som "størrelsesorden er...", er dette et presist matematisk utsagn, og dekker alle reelle implementeringer av algoritmen vår.
- Bruken av denne notasjonen er så viktig at vi skal spandere på oss endel eksempler for å få litt intuisjon rundt den.
- Vi minner om at en polynomfunksjon er en funksjon

$$f(n) = a_k n^k + \cdots + a_1 n + a_0.$$

- Vi skal anta at alle koeffisientene er i $\mathbb{N}_0$, det vil si ikkenegative hele tall.
- Videre vil vi normalt anta at $a_k > 0$, og polynomfunksjonen har da grad k.
- Vi skal se på sammenhengen mellom O-notasjonen og polynomfunksjoner.

Eksempel.

- La $f(n) = 3n + 2$ og $g(n) = 2n$.
- Da er f $O(g)$ fordi $f(n) \leqslant 2g(n)$ når $n \geqslant 2$.

Eksempel.

- La $f(n) = 10^6 \cdot n$ og la $g(n) = n^2$.
- Er f $O(g)$?
- Vi kan lett finne en verdi av c som viser dette veldig enkelt:
- $f(n) \leqslant 10^6 \cdot g(n)$ for alle n.
- Vi har også at $f(n) \leqslant g(n)$ for alle $n \geqslant 10^6$.

Eksempel.

- La $f(n) = n^2$ og la $g(n) = 10^6 \cdot n$.
- Er f $O(g)$?
- I dette tilfellet er svaret negativt.
- For å vise det, må vi vise at det ikke fins noen c som duger.
- For å vise at c ikke duger, må vi vise at det fins vilkårlig store n slik at $c \cdot g(n) < f(n)$.
- La $n > 10^6 \cdot c$.
- Da er $f(n) = n^2 > 10^6 \cdot c \cdot n = c \cdot g(n)$.
- Dette viser at svaret er negativt.

Eksempel.

- La $f(n) = 3n^4 + 10n^3 + 2n + 20$ og la $g(n) = n^4$.
- Er f $O(g)$?
- Svaret er *JA*, og vi skal gi et argument som er så generelt at det tjener som bevis for neste teorem.
- La $c = 3 + 10 + 2 + 20 = 35$, dvs., summen av alle koeffisientene i f.
- Husk at $n \geqslant 1$ her.

$$\begin{aligned} f(n) &= 3n^4 + 10n^3 + 2n + 20 \\ &\leqslant 3n^4 + 10n^4 + 2n^4 + 20n^4 \\ &= (3 + 10 + 2 + 20)n^4 \\ &= c \cdot g(n) \end{aligned}$$

Denne metoden kan brukes til å vise følgende teorem.

Teorem.
Hvis f er en polynomfunksjon med grad $\leqslant k$ vil f være $O(n^k)$.

- Hva hvis graden til f er større enn graden til g?
- Vi har sett et eksempel på dette hvor f ikke er $O(g)$.
- Det gjelder helt generelt, og vi skal se på et eksempel som illustrerer det.

Eksempel.
- La $f(n) = n^3$ og la $g(n) = 2n^2 + 4n + 6$.
- La c være en vilkårlig positiv konstant.
- Vi vil vise at det fins vilkårlig store n slik at $c \cdot g(n) < f(n)$.
- Velger vi $n > (2 + 4 + 6)c = 12c$ får vi

$$f(n) = n^3 > c \cdot (2 + 4 + 6)n^2 \geqslant c \cdot g(n)$$

(som i beviset for teoremet).

Vi kan oppsummere dette med følgende observasjon:

Korollar.
a) Hvis f og g er to polynomfunksjoner og f er $O(g)$, vil graden til f være mindre eller lik graden til g.
b) Omvendt, hvis f og g er to polynomfunksjoner slik at graden til f er mindre eller lik graden til g vil f være $O(g)$.

- Vi har definert relasjonen

$$f \text{ er } O(g)$$

og det ville vært dumt å ikke benytte anledningen til å repetere litt om relasjoner i denne forbindelse.
- Vi husker at en relasjon R er transitiv hvis

$$aRb \wedge bRc \Rightarrow aRc.$$

- Er O-notasjonstrelasjonen transitiv?

- La oss drive litt undersøkende matematikk og anta at f er $O(g)$ og at g er $O(h)$.
- Da fins det $c > 0$ og n_0 slik at hvis $n \geqslant n_0$ vil

$$f(n) \leqslant c \cdot g(n).$$

- Videre fins det $d > 0$ og n_1 slik at hvis $n \geqslant n_1$ vil

$$g(n) \leqslant d \cdot h(n).$$

- Hvis vi nå lar $n \geqslant max\{n_0, n_1\}$ har vi at

$$f(n) \leqslant c \cdot g(n) \leqslant c \cdot d \cdot h(n),$$

  så konstanten $c \cdot d > 0$ kan brukes til å vise at f er $O(h)$.
- Dette viser at relasjonen er transitiv.
- Vi husker også at en relasjon R kalles refleksiv hvis aRa for alle a i grunnmengden.
- Er relasjonen

$$f \text{ er } O(g)$$

  refleksiv?
- For alle funksjoner f og for alle tall n er $f(n) \leqslant 1 \cdot f(n)$, så f er $O(f)$ for alle f.
- Det viser at relasjonen er refleksiv.
- På generelt grunnlag kan vi da definere relasjonen

  f og g har samme kompleksitet

  ved f er $O(g)$ og g er $O(f)$.
- Siden vi tar utgangspunkt i en relasjon som er transitiv og refleksiv, får vi en ekvivalensrelasjon på denne måten.
- Ekvivalensklassene til denne relasjonen kaller vi ofte kompleksitetsklasser og de svarer til mengder av funksjoner hvor alle har samme kompleksitet ut fra forenklingene 1, 3 og 4.
- Dette er et eksempel på hvordan man kan bruke teorien for relasjoner til å gjøre et upresist begrep "vokser omtrent like fort" til et presist begrep.
- To polynomfunksjoner tilhører samme ekvivalensklasse nøyaktig når graden er den samme.
- Med dette avslutter vi innføringen i O-notasjonen.

Sorteringsalgoritmer

- Vi skal studere en sorteringsalgoritme som eksempel på hvordan vi bestemmer kompleksiteten til en algoritme.
- Sorteringsalgoritmer er en nyttig og viktig anvendelse av kompleksitetsteori.
- Vi skal se på enkle eksempler, som sortering av tall i stigende rekkefølge.
  - Dette kan knyttes til teorien om relasjoner.
  - Vi ser på relasjonen $<$ over tall.
  - Men, vi har kun bruk for at $<$ er en transitiv og irrefleksiv relasjon slik at for alle a og b, så har vi at $a = b$, $a < b$ eller $b < a$.

- Vi skal sortere følgende ti tall i stigende rekkefølge.

$$5, 9, 4, 1, 7, 12, 3, 6, 2, 8$$

- Dette vil vi i første omgang gjøre i ti operasjoner.

Eksempel (Sortering av $5, 9, 4, 1, 7, 12, 3, 6, 2, 8$).

1. $5, 9, 4, 1, 7, 12, 3, 6, 2, 8$
2. $5, 9, 4, 1, 7, 12, 3, 6, 2, 8$
3. $5, 9, 4 \to 5, 4, 9 \to 4, 5, 9, 1, 7, 12, 3, 2, 8$
4. $4, 5, 9, 1 \to 4, 5, 1, 9 \to 4, 1, 5, 9 \to 1, 4, 5, 9, 7, 12, 3, 2, 8$
5. $1, 4, 5, 9, 7 \to 1, 4, 5, 7, 9, 12, 3, 2, 8$
6. $1, 4, 5, 7, 9, 12, 3, 2, 8$
7. $1, 4, 5, 7, 9, 12, 3 \to \cdots \to 1, 4, 3, 5, 7, 9, 12 \to 1, 3, 4, 5, 7, 9, 12, 2, 8$
8. $1, 3, 4, 5, 7, 9, 12, 2 \to \cdots \to 1, 3, 2, 4, 5, 7, 8, 9, 12 \to 1, 2, 3, 4, 5, 7, 9, 12, 8$
9. Tilsist flytter vi 8 nedover i den sorterte delen av listen til vi finner dens plass, og den sorterte listen blir $1, 2, 3, 4, 5, 7, 8, 9, 12$.

Her er sorteringsalgoritmen fra boka.

1 *Input* $x_1, x_2, \ldots, x_n$
2 **For** $i = 2$ **to** n **do**
  2.1 *plassér* $\leftarrow x_i$
  2.2 $j \leftarrow i - 1$
  2.3 **While** $j \geqslant 1$ *and* $x_j >$ *plassér* **do**
    2.3.1 $x_{j+1} \leftarrow x_j$
    2.3.2 $j \leftarrow j - 1$
  2.4 $x_{j+1} \leftarrow$ *plassér*
3 *Output* $x_1, x_2, \ldots, x_n$

- La oss nå prøve å analysere kompleksiteten til denne algoritmen.
- Vi tar for oss ett og ett element fra den opprinnelige listen, og plasserer det på sin rette plass i forhold til den sorterte versjonen av den delen som kom foran.
- Det gir en hovedrunde med lengde n
- I hvert skritt i denne hovedrunden, må vi sammenlikne det objekter vi skal plassere med elementene i den ferdigsorterte delen av listen.
- Vi kan risikere å måtte sammenlikne det nye objektet med alle de som kom først.
- Hvis den opprinnelige listen kom ordnet helt motsatt av hva vi ønsker, skjer dette hver gang.
- Det vil gi oss

$$1 + 2 + \cdots + (n-1) = \frac{(n-1)n}{2}$$

antall sammenlikninger.

- Siden det er disse sammenlikningene som er mest tidkrevende, kan vi konkludere med at tidskompleksiteten til denne algoritmen er $O(n^2)$.
- Er det mulig å være mer effektiv?
- Når vi skal sortere en liste med n elementer, er vi nødt til, på en eller annen måte å plassere alle n elementer på riktig plass.
- Det sier seg selv at dette må skje i omtrent n omganger.
- I den algoritmen vi så på brukte vi i gjennomsnitt $\frac{n}{2}$ antall sammenlikninger for å plassere et objekt i en allerede ordnet liste, i det verste tilfellet.
- Her er det rom for betydlige forbedringer.
- La oss se på et eksempel.
- Vi har gitt en ordnet liste på 16 objekter, eksempelvis tallene

  $$1, 3, 7, 9, 12, 14, 22, 23, 25, 31, 37, 40, 41, 44, 47, 50$$

  og vi vil finne plassen til tallet 8 i denne listen på en måte som kan inngå i en effektiv algoritme.
- Hvis vi bruker metoden fra i sted, vil vi foreta 14 tester.
- Etter den nye metoden vil vi starte med å sette det nye tallet inn i midten:

  $1, 3, 7, 9, 12, 14, 22, 23, 8, 25, 31, 37, 40, 41, 44, 47, 50$
- Vi ser at midten er for langt oppe, så vi hopper ned til midten av den delen av listen som ligger under:

  $1, 3, 7, 9, 8, 12, 14, 22, 23, 25, 31, 37, 40, 41, 44, 47, 50$
- Tallet ligger fremdeles for høyt, så vi gjør det samme en gang til:

  $1, 3, 8, 7, 9, 12, 14, 22, 23, 25, 31, 37, 40, 41, 44, 47, 50$
- Nå kom vi for langt ned, så vi flytter oss opp igjen, halvparten så langt som vi flyttet sist.
- Det gir

  $1, 3, 7, 8, 9, 12, 14, 22, 23, 25, 31, 37, 40, 41, 44, 47, 50$
- Ved systematisk å omtrent halvere den delen av den opprinnelige listen det nye objektet skal plasseres, vil antall trinn i plasseringsalgoritmen reduseres fra å være proporsjonal med n til å bli proporsjonal med antall sifre i n. (Spiller det noen rolle om vi snakker om binær representasjon eller dekadisk representasjon?)
- I boka står det en pseudokode for en sorteringsalgoritme basert på dette prinsippet.
- Det er ikke noe stort poeng å gjengi denne koden her så sent i semesteret.
- Tidskompleksiteten til denne algoritmen er fremdeles $O(n^2)$, men ikke $O(n)$.
- Vi opplever det likevel som at vi har funnet en bedre algoritme.
- Poenget her at vi trenger en notasjon for å kunne snakke om tidskompleksiteter som er mellom $O(n)$ og $O(n^2)$.

Definisjon.
Hvis n er et tall, lar vi

$$\lg n$$

være tallet m slik at $2^m = n$.

Vi kan kalle dette for binærlogaritmen til n.

- For alle praktiske formål i kompleksitetsteori, kunne vi brukt funksjonen som gir antall sifre i binærrepresentasjonen av n i stedenfor.
- Den mest effektive sorteringsalgoritmen har en tidskompleksitet som er $O(n \cdot \lg n)$. (Se oppgavene i boka.)
- Man bør lese boka og forstå hvorfor følgende er tilfelle.
  - $\lg n$ er $O(n)$
  - n er ikke $O(\lg n)$

Gjennomførbare algoritmer

- Vi har snakket om at vi skal lære å vurdere om en algoritme kan gjennomføres i løpet av realistisk tid.
- Som de gode matematikere vi har blitt skal vi selvfølgelig gi en presis definisjon av hva som menes med en gjennomførbar eller overkommelig algoritme.
- Vi har snakket om algoritmer hvor kompleksiteten er $O(n \cdot \lg(n))$, $O(n^{\frac{3}{2}})$ og $O(n^2)$.
- Alle disse er gjennomførbare.
- Vi skal se på noen algoritmer som ikke er gjennomførbare for store input.

Eksempel.

- Vi har laget en algoritme som avgjør om et utsagnslogisk uttrykk er en tautologi eller ikke.
- Den består i at vi skriver opp sannhetsverditabellen til uttrykket.
- Hvis n er antall symboler i uttrykket, vil antall søyler i tabellen i verste fall være $O(n)$, mens antall linjer i verste fall er $O(2^n)$.
- Tidskompleksiteten av sannhetsverditabellmetoden er altså i $O(n \cdot 2^n)$, og for store input er dette ikke gjennomførbart.

Eksempel.

- Det finnes ingen virkelig effektiv metode for å avgjøre om et naturlig tall er et primtall på, og de som er lette å forstå er i alle fall ikke effektive.
- Siden det er størrelsen av input som teller (antall bit i binærrepresentasjonen av tallet), er det antall sifre i input som er utgangspunktet for å vurdere kompleksiteten.
- Den naive måten å undersøke om n er et primtall på er å undersøke om n har noen faktor m med $2 \leqslant m \leqslant \sqrt{n}$.

Eksempel (Fortsatt).

- Det holder selvfølgelig å gjøre dette for primtallene mellom 2 og $\sqrt{n}$, men da må vi kaste bort tid på å bestemme hvilke av disse tallene som er primtall, så det er ikke nødvendigvis så lurt.
- Hvis k er antall sifre i n, er $\frac{k}{2}$ omtrent antall sifre i $\sqrt{n}$, og det er omtrent $2^{\frac{k}{2}}$ antall divisjoner vi må utføre for å bestemme om n er et primtall eller ikke.
- I kryptografi er vi interesserte i primtall med hundre sifre eller mer, eller helst i produkter av to eller tre slike primtall.
- Da vil de naive metodene sprenge alle grenser for anstendig kompleksitet.

MAT1030 – Forelesning 30

Kompleksitetsteori

Dag Normann - 12. mai 2010

Forelesning 30: Kompleksitetsteori

Oppsummering

- I dag er siste forelesning med nytt stoff!
- Etter det vil vi begynne på en full repetisjon av pensum.
- Nå, først litt repetisjon av kompleksitetsteorien.
- Kompleksitetsteori: Om algoritmers *tidsforbruk*.
- Fire tilnærminger:
  1. Tell bare de mest tidkrevende operasjonene.
  2. Ta utgangspunkt i de verste tilfellene.
  3. Anta at input er stort.
  4. Ikke skill mellom to tidskompleksiteter hvis vekstraten til den ene er et konstant multiplum av vekstraten til den andre.
- Tidskompleksiteten til en algoritme: En funksjon fra $\mathbb{N}$ til $\mathbb{N}$.
- O-notasjon:
  - f er $O(g)$ hvis det fins en positiv konstant c slik at $f(n) \leqslant c \cdot g(n)$ for alle tilstrekkelig store n.
  - Viktig: Forstå hva det vil si at en funksjon f *ikke* er $O(g)$.
  - Hvis f er en polynomfunksjon med grad $\leqslant k$, så vil f være $O(n^k)$.

Sorteringsalgoritmer

- V i så på to sorteringsalgoritmer.
- I begge tilfellene sorterer vi en liste på n tall ved å plassere ett og ett riktig i forhold til de som allerede er sortert.
- Vi så på to måter å plassere *det neste tallet* på rett plass på:
  1. Sammenlikn med ett og ett tall i den sorterte delen for å finne rett plassering.
  2. Sammenlikn med midten, midten av resten, osv. til vi finner rett plass.
- Metode 1. gir større kompleksitet enn metode 2.

Definisjon.
Hvis n er et tall, lar vi

$$\lg n$$

være tallet m slik at $2^m = n$.

Vi kan kalle dette for binærlogaritmen til n.

- For alle praktiske formål i kompleksitetsteori, kunne vi brukt funksjonen som gir antall sifre i binærrepresentasjonen av n i stedenfor.
- Den mest effektive sorteringsalgoritmen har en tidskompleksitet som er $O(n \cdot \lg n)$. (Se oppgavene i boka.)
- Man bør lese boka og forstå hvorfor følgende er tilfelle.
  - $\lg n$ er $O(n)$
  - n er ikke $O(\lg n)$

Gjennomførbare algoritmer

- Vi har snakket om at vi skal lære å vurdere om en algoritme kan gjennomføres i løpet av realistisk tid.
- Som de gode matematikere vi har blitt skal vi selvfølgelig gi en presis definisjon av hva som menes med en gjennomførbar eller overkommelig algoritme.
- Vi har snakket om algoritmer hvor kompleksiteten er $O(n \cdot \lg(n))$, $O(n^{\frac{3}{2}})$ og $O(n^2)$.
- Alle disse er gjennomførbare.
- Vi skal se på noen algoritmer som ikke er gjennomførbare for store input.
- Vi avsluttet gårsdagens forelesning med å se på to eksempler.
- Det å avgjøre om et utsagn er en tautologi eller ikke krever $O(n \cdot 2^n)$ regneskritt om vi bruker sannhetsverdimetoden.
- For å faktorisere et primtall med n siffre, må vi i prinsippet lete gjennom $2^{\sqrt{n}}$ mulige faktorer. Det tar lang tid.
- Vi skal se på et eksempel til.

Eksempel.

- La G være en sammenhengende graf.
- Hvordan skal vi gå frem for å bestemme om grafen har en Hamiltonsti, det vil si en sti som er innom hver node nøyaktig en gang?
- Hvis n er antall noder i grafen, vil en Hamiltonsti ha $n-1$ kanter
- Det finnes ingen kjent måte å undersøke om G har en Hamiltonsti på som er vesentlig mer effektiv enn den naive; prøv alle stier med $n-1$ kanter og se om en av dem tilfeldigvis skulle være en Hamiltonsti.

Eksempel (Fortsatt).

- I det verste tilfellet er antall stier i G med $n-1$ kanter $O(\binom{n^2}{n-1})$, det vil si

$$\frac{(n^2)!}{(n^2-n+1)!(n-1)!}$$

- Dette er et tall som faktisk er større enn 2^{n-1}, så algoritmen er ikke imponerende effektiv.

Definisjon.

Vi sier at en algoritme er gjennomførbar (tractable på engelsk) hvis tidskompleksiteten er $O(n^k)$ for en k.

Vi merker oss følgende sammenhenger.

- 2^n er ikke $O(n^k)$ for noen verdi av k
- 2^n er $O(n!)$

- Det er flere grunner til at man har falt ned på dette som en fornuftig definisjon.
- Tidligere erfaringer tilsa at hvis en algoritme er gjennomførbar i henhold til denne definisjonen, kan den brukes i praksis.
- Det er ofte slik at k ligger rundt tre eller lavere.
- Ganske overraskende viste en gruppe indere for noen år siden at det finnes en algoritme som avgjør om et tall er et primtall eller ikke som faller inn under denne definisjonen, men der var k (og konstanten c) så stor at algoritmen hadde mer teoretisk enn praktisk verdi.
- Definisjonen er også ganske robust, selv om forskjellige matematiske modeller for hva en beregning består i kan gi forskjellige verdier på graden.

- Vi skal avslutte disse forelesningene med å snakke bittelitegrann om **P** og **NP**.
- **P** er klassen av problemer som kan løses i polynomisk tid, det vil si de som kan løses av en gjennomførbar algoritme slik vi har definert det.
- Eksempler på problemer som ligger i **P** er om en graf er sammenhengende og om den har en Eulerkrets, om to termer lar seg unifisere, om et uttrykk svarer til en term på polsk form og etterhvert om et tall er et primtall eller ikke (det kom som en overraskelse).
- **NP** er grovt sagt klassen av problemer hvor vi med flaks bare trenger å bruke polynomisk tid for å løse det den ene veien, mens vi tilsynelatende bruker eksponensiell tid om løsningen går den andre veien.

- Hvis G er en graf, og noen streker opp en Hamiltonsti, er det raskt å få bekreftet at det er en Hamiltonsti det er, mens hvis det ikke finnes noen Hamiltonsti trenger vi lang tid.
- Hvis A er et uttrykk som ikke er en tautologi, kan vi få vite det veldig fort hvis vi tilfeldigvis prøver den fordelingen av sannhetsverdier som gjør utsagnet usant, mens vi fortsatt må skrive ut hele sannhetsverditabellen hvis utsagnet er en tautologi.
- Gitt en eske med mange puslespill-brikker, er det lett å vise at brikkene kan settes sammen til et bilde hvis vi greier å legge brikkene riktig én for én, mens det tar lang tid å bli sikker på at det er feil ved noen av brikkene, om det er tilfelle.

- Det store åpne problemet er om disse mengdene av problemer er de samme, eller om det finnes problemer som er i **NP** men ikke i **P**.
- Litteratur rundt **P** = **NP**-problemet finner dere på nettet og i en rekke lærebøker om automatateori eller kompleksitetsteori, eller i en NORMAT-artikkel fra 2005 (Årgang 53, bind 1).
- Dette er et av de syv milleniumsproblemene i matematikk, og det er en dusør på $\$10^6$ for hvert av de seks som står fortsatt uløst.

- Den som løste det eneste milleniumsproblemet som er løst, sa *nei takk* til pengene.

Slutt

Repetisjon

Repetisjon av hele pensum

- Vi går systematisk gjennom alle delene av pensum og trekker frem de ferdighetene man må beherske for å være sikker på å greie 90% av eksamensettet.
- For å være sikker på greie 100% må man kunne alt som er oppgitt som pensum.
- Oppgaver kan hentes fra hele pensum, men hovedvekten blir på stoff som ikke er testet gjennom de to obligatoriske oppgavene.
- Hvis man kan løse alle oppgaver av den typen som er gitt som treningsoppgaver gjennom semesteret, ligger man meget godt an.
- Vi skal se på noen eksempler, men det er ikke slik at alle eksemplene dekker oppgaver som blir gitt til eksamen, og heller ikke slik at vi illustrerer alle eksamensoppgavene med eksempler her.

Kapittel 1

- I Kapittel 1 så vi på algoritmebegrepet generelt og kontrollstrukturer spesielt.
- Vi har brukt kontrollstrukturer når det har passet seg slik gjennom hele semesteret.
- Viktige deler av læringsmålet i dette emnet er at studentene skal
    * Kunne utarbeide en kontrollstruktur som en nærmere presisering av en algoritme.
    * Kunne lese en kontrollstruktur og forstå sammenhengen mellom inputverdiene og outputverdiene.
    * I tilknytning til Kapittel 13, kunne vurdere tidskompleksiteten.

Kapittel 2 og 3

- Kapittel 2 og 3 omhandler tallsystemer og prinsipper for digital representasjon av hele tall og reelle tall.
- Mye av dette stoffet er behandlet mer grundig i andre emner.
- Forståelsen av digital representasjon ble forsiktigvis testet i det første obligatoriske oppgavesettet.
- Vi har etterhvert hatt bruk for å vise hvordan relasjoner, grafer, trær og andre dataobjekter kan representeres i en datamaskin.
- Det er lite aktuelt med en oppgave som behandler stoff fra disse kapitlene direkte, men man kan ha fordel av å forstå hvordan grafer, trær og andre objekter representeres, især i forbindelse med en eventuell oppgave i kompleksitetsteori.

Kapittel 4

- I dette kapitlet om logikk har vi konsentrert oss om
    – utsagnslogikk

  - kvantorer
  - føring av argumenter
- I tillegg er det en del snakk om betydningen av logikk i programmering.
- Dette siste er mest motiverende, men en motivasjon man bør ta med seg til senere emner.
- I utsagnslogikk har vi lagt mest vekt på det formelle språket, hvor vi tar utgangspunkt i noen utsagnsvariable, og bygger opp utsagn fra utsagnsvariablene ved å bruke bindeordene (eller konnektivene) $\neg$, $\vee$, $\wedge$, $\rightarrow$ og $\leftrightarrow$.
- Et typisk eksempel kan være

$$A = \neg(p \rightarrow q) \rightarrow \neg q \vee p.$$

- Vi lærte hvordan vi kunne spare på parenteser, slik at vi vet hvordan parentesene egentlig skal settes i eksemplet over.
- Vi skal kunne avgjøre om et utsagn som det over er en tautologi, en kontradiksjon eller ingen av delene.
- Den sikreste måten å gjøre det på er å sette opp en sannhetsverditabell.
- Tabellen til utsagnet over finner vi på neste side.

p	q	$p \rightarrow q$	$\neg(p \rightarrow q)$	$\neg q$	$\neg q \vee p$	A
T	**T**	**T**	**F**	**F**	**T**	**T**
T	**F**	**F**	**T**	**T**	**T**	**T**
F	**T**	**T**	**F**	**F**	**F**	**T**
F	**F**	**T**	**F**	**T**	**T**	**T**

Vi ser at utsagnet er en tautologi.

- Vi lærte også om bruk av kvantorer.
- Kvantorene $\exists$ - "det eksisterer", og $\forall$ - "for alle", brukes for å gi matematiske definisjoner og setninger en presis formulering.
- Kvantorer ble ikke noe sentralt tema dette året, men vi fikk bruk for dem da vi skulle undersøke om f er $O(g)$ for bestemte funksjoner f og g.
- Det er ofte viktig å ha preise definisjoner når man skal vise at en egenskap ikke holder.
- Det siste temaet vi tok opp under dette kapitlet var bevisformer hvor vi skiller mellom direkte bevis og indirekte eller kontrapositive bevis.
- I et kontrapositivt bevis antar vi at den påstanden vi vil vise er usann, og så beviser vi at noen av forutsetningene heller ikke kan være sanne.
- Vi har sett på noen eksempler på bevis, men har ikke laget noe stort nummer av det.
- Det er meningen at dere skal kunne gjennomføre et enkelt resonement om dere blir bedt om det.

Kapittel 5

- Kapittel 5 tar opp et bredt tema over få sider.
- Først har vi en innføring i Boolsk mengdelære med snitt, union, komplement og mengdedifferens.
- Det er viktig at dere kjenner definisjonene av $A \cap B$, $A \cup B$, $\overline{A}$ og $A - B$.

- To mengder er like hvis de har de samme elementene.
- $A \subseteq B$, A er inneholdt i B, hvis alle elementene i A også er elementer i B.
- Når vi snakker om komplementet $\overline{A}$, har vi alltid antatt at vi har en universell mengde $\mathcal{E}$.
- Hvis vi har et Boolsk uttrykk hvor det inngår to eller tre vilkårlige mengder, kan vi undersøke egenskapene ved dette uttrykket ved å bruke et Venndiagram
- Et eksempel på en mulig oppgave kan være å bruke et Venndiagram til å avgjøre om følgende inklusjon alltid vil holde:

$$(A \cap \overline{B}) - C \subseteq \overline{C - (B \cup A)}.$$

- Vi tar den på tavla.
- Kardinaliteten til en mengde er et finere ord for hvor mange elementer mengden har.
- Vi har lært om ordnede par, kartesisk produkt og om potensmengder.
- Potensmengden til A er mengden av alle delmengder av A.
- Dere bør vite at hvis A har n elementer, så har A 2^n delmengder, og A^2 har n^2 elementer.

MAT1030 – Forelesning 31

Repetisjon

Dag Normann - 18. mai 2010

Forelesning 31: Repetisjon

- Vi går systematisk gjennom alle delene av pensum og trekker frem de ferdighetene man må beherske for å være sikker på å greie 90% av eksamensettet.
- For å være sikker på greie 100% må man kunne alt som er oppgitt som pensum.
- Oppgaver kan hentes fra hele pensum, men hovedvekten blir på stoff som ikke er testet gjennom de to obligatoriske oppgavene.
- Hvis man kan løse alle oppgaver av den typen som er gitt som treningsoppgaver gjennom semesteret, ligger man meget godt an.
- Vi skal se på noen eksempler, men det er ikke slik at alle eksemplene dekker oppgaver som blir gitt til eksamen, og heller ikke slik at vi illustrerer alle eksamensoppgavene med eksempler her.

Kapittel 5

- Siste delen av dette kapitlet omhandler relasjoner.
- En relasjon på en mengde A er en delmengde R av A^2.
- Vi har sett på fem egenskaper en relasjon kan ha
  - Transitiv: $xRy \wedge yRz \Rightarrow xRz$ for alle x, y og z.
  - Refleksiv: xRx for alle x.
  - Irrefleksiv: Ikke xRx for noen x.
  - Symmetrisk: $xRy \Rightarrow yRx$ for alle x og y.
  - Antisymmetrisk: $xRy \wedge yRx \Rightarrow x = y$ for alle x og y.
- I enkle tilfeller bør vi kunne bestemme om en relasjon er refleksiv, om den er symmetrisk og om den er transitiv.
- En relasjon som er refleksiv, symmetrisk og transitiv kalles en ekvivalensrelasjon.
- En ekvivalensrelasjon R på en mengde A vil dele A opp i disjunkte ekvivalensklasser av parvis ekvivalente objekter.
- I enkle situasjoner bør dere være i stand til å beskrive ekvivalensklassene til en ekvivalensrelasjon.

Eksempel.

La $A = \{2, 3, 4, 6, 8, 9, 12, 16, 18, 24, 27, 32, 36, 48\}$.

Hvis n og m er i A, lar vi nRm hvis n og m har like mange faktorer når de er fullstendig faktorisert.

Kan vi finne ekvivalensklassene?

Ekvivalensklassene vil svare til delmengder av A som har den interessante egenskapen felles.

Den interessante egenskapen er her antall faktorer.

Eksempel (Fortsatt).

1 En faktor: $\{2, 3\}$
2 To faktorer: $\{2^2, 2 \cdot 3, 3^2\} = \{4, 6, 9\}$
3 Tre faktorer: $\{2^3, 2^2 \cdot 3, 2 \cdot 3^2, 3^3\} = \{8, 12, 18, 27\}$
4 Fire faktorer: $\{2^4, 2^3 \cdot 3, 2^2 \cdot 3^2\} = \{16, 24, 36\}$
5 Fem faktorer: $\{2^5, 2^4 \cdot 3\} = \{32, 48\}$

Kapittel 6

- Selv om funksjonsbegrepet gjennomsyrer mye av matematikken, har vi ikke lagt mye vekt på generell funksjonslære.
- Funksjonsbegrepet gir oss et språk som vi kan bruke i andre deler av matematikken og informatikken.
- Ved hjelp av det språket kan vi gi mening til at to grafer "egentlig" er like, det vil si isomorfe.
- En algoritme vil gjerne definere en funksjon, funksjonen som fra en input-verdi gir oss den tilsvarende outputverdien.
- Vi brukte også funksjoner for å kunne måle tidskompleksiteten til en algoritme.
- Til eksamen kan det være aktuelt å kjenne til hvordan man setter sammen to funksjoner, kjenne igjen en injektiv (eller surjektiv) funksjon om man får bruk for det, men læringsmålet er i hovedsak å kunne bruke funksjonsbegrepet i sammenhenger utenom Kapittel 6, så det blir ikke lagt mye vekt på rendyrkede oppgaver om vilkårlige funksjoner.

Kapittel 7

- Et av de viktigste kapitlene i pensum er kapitlet om induksjon og rekursjon.
- Vi starter med å se på hvordan man definerer en følge ved rekursjon:

  $f(1) = 1$

  $f(n) = 2f(n-1) + 1$ når $n > 1$.
- Poenget med rekursjon er at ved hjelp av disse to linjene har vi bestemt hva $f(n)$ må være for alle $n \geqslant 1$.
- Vi har at $f(2) = 2f(1) + 1 = 2 + 1 = 3$
- Vi har at $f(3) = 2f(2) + 1 = 6 + +1 = 7$
- Vi har at $f(4) = 2f(3) + 1 = 14 + 1 = 15$

- Slik kan vi fortsette.
- Hvis vi nå ser at $f(1) = 2^1 - 1$, at $f(2) = 2^2 - 1$, at $f(3) = 2^3 - 1$ og at $f(4) = 2^4 - 1$, er det nærliggende å tro at $f(n) = 2^n - 1$ for alle $n \in \mathbb{N}$.
- Det er til og med nærliggende å tro at denne egenskapen har noe med hvordan vi definerte f å gjøre.
- Vi bruker et induksjonsbevis for å vise at $f(n) = 2^n - 1$ for alle n.
- Induksjonsbevis er delt opp i induksjonstart og induksjonskritt
- Induksjonstarten tilsier at vi skal vise at $f(1) = 2^1 - 1$, og det holder siden både høyre og venstre sider har verdi 1.
- I induksjonskrittet antar vi at $f(n-1) = 2^{n-1} - 1$, hvor $n > 1$, og vi skal vise at da er $f(n) = 2^n - 1$.
- I dette tilfellet er det rett frem.
- En ulempe ved dette formatet på induksjonsbevis er at det gjør det lett å bli opptatt av formen bevisene får på bekostning av forståelsen av hvorfor bevisformen gir riktige svar.
- Vi har derfor lagt vekt på å bruke rekursjon og induksjon i en mer generell sammenheng.
- Det ene tilfellet er hvor grunnlaget for induksjonen er mer enn et tall.
- Dette er systematisk gjennomført for rekurrenslikninger.
- Rekurrenslikninger er et takknemlig oppgavestoff, og vi skal ta et eksempel.

Eksempel.

- Anta at vi har gitt rekurrenslikningen

$$F(n) = F(n-1) + 2F(n-2).$$

- Denne likningen kan også skrives som

$$F(n) - F(n-1) - 2F(n-2) = 0.$$

- Vi finner den generelle løsningen av en slik likning ved å se på den karakteristiske likningen

$$r^2 - r - 2 = 0$$

som har løsninger $r = 2$ og $r = -1$.

Eksempel (Fortsatt).

- Den generelle løsningen av likningen er da

$$f(n) = A \cdot 2^n + B \cdot (-1)^n.$$

- Hvis vi nå i tillegg får vite at vi skal ha at $f(1) = 1$ og $f(2) = 11$, har vi fått to initialbetingelser
- Den rekursive definisjonen forteller oss at

$F(3) = F(2) + 2F(1) = 11 + 2 = 13$

$F(4) = F(3) + 2F(2) = 13 + 22 = 35$

osv.

- Vi kan finne hele løsningen ved å løse likningene

  $A \cdot 2^1 + B \cdot (-1)^1 = 1$

  $A \cdot 2^2 + B \cdot (-1)^2 = 11$

  som gir $A = 2$ og $B = 3$.

Eksempel (Fortsatt).

- Den spesielle løsningen er derfor

$$f(n) = 2 \cdot 2^n + 3 \cdot (-1)^n.$$

- Hvis vi nå blir bedt om å bekrefte denne formelen ved et induksjonsbevis, kan vi argumentere som følger:
- Ved at vi har løst de to førstegradslikningene, vet vi at formelen stemmer for $n = 1$ og for $n = 2$
- Vi lar nå $n > 2$ og vi antar at formelen stemmer for $n - 1$ og $n - 2$.
- Da kan vi bruke rekurrensdefinisjonen, regne litt, og se at formelen stemmer for n også
- Da kan vi konkludere med at vi har et induksjonsbevis.

- Vi så også på induktivt definerte strukturer generelt.
- Vi har en induktivt definert struktur hvis vi strukturen (eller mengden) er konstruert ved at vi har noen basisobjekter, og noen prinsipper for hvordan vi konstruerer nye objekter fra eksisterende objekter.
- Vi har sett på mengden av ord som bygget opp fra det tomme ordet ved etter tur å legge nye bokstaver til høyre for ordet.
- Vi har sett på mengden av utsagnslogiske uttrykk som bygget opp fra utsagnsvariable ved å bruke de logiske bindeordene.
- Etterhvert så vi på de binære trærne som bygget opp fra den enkle bladnoden ved hjelp av binære forgreninger.
- Hver gang vi har en induktivt definert struktur, har vi et grunnlag for å konstruere funksjoner ved rekursjon, og et grunnlag for å bevise setninger ved induksjon.
- Vi får da en rekursjon(induksjon)start for hvert basisobjekt og et rekursjon(induksjon)skritt for hver måte å konstruere nye objekter på.
- Vi har sett dette eksemplifisert gjennom konstruksjon av svak normalform og gjennom eliminering av $\rightarrow$ og $\leftrightarrow$ fra utsagnslogiske uttrykk, og vi så på eksempler som sammenbinding av ord, speiling av ord etc.
- Vi la stor vekt på bruk av trerekursjon i et senere kapittel.

Kapittel 9

- Avsnittet om kombinatorikk er lite sammenliknet med tilsvarende avsnitt i andre kurs i diskret matematikk.
- Mye av det som står her er dekket av pensum i Videregående skole.
- Man bør kunne kjenne binomialkoeffisientene, deres definisjon og bruksområde.
- Man bør også kjenne til formelen for hvor mange måter man kan velge ut k elementer i rekkefølge fra en mengde med n elementer på, men her legger vi ikke vekt på at man husker notasjonen.
- Pensum er stort sett det som står i boka, eller gjennomgått som eksempler på forelesningen, og vi bruker ikke mer tid på det her.

Kapittel 10

- I grafteori har vi lagt mest vekt på ordinære grafer, og mindre vekt på rettede grafer.
- Det er noen begreper det vil kunne være en fordel å kjenne til, som
  - Node
  - Kant
  - Graden til en node
  - Løkke
  - Parallelle kanter
  - Enkel graf
  - Krets
  - Sti
  - Sammenhengende graf
- Vi har behov for å vite hvordan grafer representeres digitalt i Kapittel 13.
- Ellers er de viktigste ferdighetene fra dette kapitlet å kunne vurdere om en graf har en Eulersti eller en Eulerkrets.
- I de tilfellene hvor grafen har en Eulersti eller en Eulerkrets, skal man kunne finne en slik en.
- To grafer G og H er isomorfe hvis det finnes en bijeksjon f fra nodene til G til nodene til H og en bijeksjon g fra kantene til G til kantene til H slik at s er en kant mellom a og b hvis og bare hvis $g(s)$ er en kant mellom $f(a)$ og $f(b)$.
- Det er meningen at dere skal resonnere rundt isomorfi mellom grafer.

Kapittel 11

- En sykel i en graf er en sti som begynner og slutter i samme node, men som ellers ikke er innom samme node to ganger.
- En sykel kan da heller ikke inneholde samme kant to steder.
- Et tre er en sammenhengende graf som ikke inneholder noen sykel.
- Et tre vil alltid være en enkel graf, ettersom to parallelle kanter danner en sykel.
- I et tre er alltid antall kanter en mindre enn antall noder.
- Dette er en viktig kunnskap for å kunne besvare enkle spørsmål.

- Vår første anvendelse av trær er i forbindelse med enkle, vektede grafer.
- En graf er vektet hvis hver kant er utstyrt med et ikke-negativt tall, en vekt.
- Et utspennende tre i en enkel graf er en delgraf som omfatter alle nodene og som er et tre.
- Prims algoritme står sentralt i pensum, og vil normalt bli tema for en eksamensoppgave.
- Med Prims algoritme skal man finne et utspennende tre i en vektet graf som har minimal samlet vekt.
- I Pims algoritme starter man i et vilkårlig punkt.
- Deretter bygger man opp et tre, ved i hvert skritt å legge en kant som forbinder den delen av treet man har bygget opp med en *ny* node.
- (I boka er dette formulert som at man ikke innfører noen sykel.)

- Vi har også sett på Kruskals algoritme.
- Der bryr vi oss ikke om å bygge et tre hele tiden, bare om å legge til en ny kant *med minimal vekt* slik at vi ikke *introduserer en sykel*.
- Hvis det ikke blir presisert at man skal bruke Prims algoritme, kan man bruke Kruskals algoritme for å finne det minimale utspennende treet.
- Blir man bedt om å bruke Prims algoritme, og å angi rekkefølgen på kantene, må man angi startnoden.

- Den andre algoritmen vi har sett på i forbindelse med vektede grafer er Dijkstras algoritme.
- Dijkstras algoritme er også eksamensrelevant.
- Poenget her er å starte med en utvalgt node, og så finne det utspennende treet som gir minimal avstand fra hver av de andre nodene til den utvalgte.
- Siden to noder i et tre er forbundet med en og bare en sti, lar vi "avstand" mellom to noder bety summen av vektene på kantene i denne stien.
- Dijkstras algoritme lar oss også bygge opp treet node for node og kant for kant, men i hvert skritt legger vi nå til en kant til en ny node som gir oss en minimal ny avstand til sentrumsnoden.

- De andre typene trær vi har sett på er trær med rot, merkede trær og binære trær.
- Vi har spesielt lagt vekt på å studere syntakstrær og bevistrær.
- I begge disse tilfellene snakker vi om merkede, binære trær, og dermed spesielt om trær med rot.
- Et bevistre gir oss en mulighet for å analysere om et utsagn på svak normalform er en tautologi eller ikke.
- Hver node er merket med en disjunksjon ($\vee$) av formler på svak normalform, hvor hvert ledd i disjunksjonen enten er en literal, det vil si en utsagnsvariabel eller negasjonen av en utsagnsvariabel, eller en konjunksjon ($\wedge$).
- En bladnode hvor det forekommer både en utsagnsvariabel og negasjonen dens, kalles et aksiom.

- Vi minner først om hva det vil si at et utsagn er på svak normalform.
- I følge definisjonen er et utsagn på svak normalform hvis vi bare bruker bindeordene $\neg$, $\vee$ og $\wedge$, og hvor $\neg$ alltid forekommer rett foran en utsagnsvariabel.

- Eksempelvis er
$$(\neg p \vee q) \wedge (p \vee \neg q)$$
på svak normalform, mens
$$\neg(p \vee q)$$
ikke er det.
- Vi kan se på mengden av uttrykk på svak normalform som en induktivt definert mengde, ved at basisuttrykkene er literaler som p og $\neg q$, og hvor vi bare bruker $\wedge$ og $\vee$ for å bygge opp mer komplekse uttrykk.
- Bevistrærne tar utgangspunkt i at utsagn på formen $A_1 \vee \cdots \vee A_n$ vil være en tautologi hvis det finnes en i og j slik at $A_i = \neg A_j$.
- Vi opphøyer derfor slike utsagn til aksiomer. De er både tautologier, og det er effektivt å la en datamaskin teste om et utsagn på svak normalform oppfyller denne egenskapen.
- La nå $D \vee A \vee C$ og $D \vee B \vee C$ være to utsagn, og anta at vi vet at disse to utsagnene er sanne i en gitt situasjon.
- Da vet vi at $D \vee (A \wedge B) \vee C$ også er sann.
- Dette kan vi se ved å se på en sannhetsverditabell med 8 linjer!
- Vi skriver ut tabellen på tavla.
- I alle linjene hvor både $D \vee A \vee C$ og $D \vee B \vee C$ får verdien **T**, vil også $D \vee (A \wedge B) \vee C$ få verdien **T**.
- Da kan vi opphøye
Fra $D \vee A \vee C$ og $D \vee B \vee C$ kan vi slutte $D \vee (A \wedge B) \vee C$
til en logisk regel (hvor det ikke trenger å stå noe på plassen til D eller til C.
- Et bevistre er et merket, binært tre, hvor alle nodene er merket med utsagn på svak normalform, hvor alle bladnodene er aksiomer, og hvor merket til en forgreningsnode alltid følger fra merkene til barna ved regelen over.
- Det er to fordeler med slike bevistrær:
  1. Det finnes effektive algoritmer for å teste om et merket tre er et bevistre eller ikke.
  2. Hvis A er et utsagn på svak normalform, finnes det en lett forståelig strategi for å lage et bevistre for A (det vil si at rotnoden er merket med A) hvor vi ofte raskt kan bestemme om A er en tautologi eller ikke.
- I de verste tilfellene trenger vi fortsatt eksponensielt lang tid, men de verste tilfellene oppstår ofte ikke i praktiske anvendelser.
- Det vi forsøksvis har prøvd å lære bort i disse forelesningene, etter ønske fra grupperinger ved IfI, er hvordan vi lager et bevistre fra et utsagn.
- Vi skal se på et par eksempler til.

Eksempel.

- $$\neg q \vee (p \wedge q) \vee \neg p$$

- Hvis vi prøver å bruke den logiske regelen baklengs, se vi at dette utsagnet er en konsekvens av utsagnene

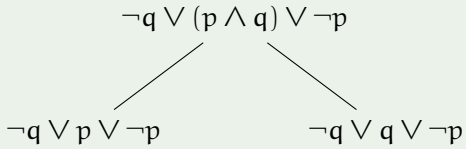
$$\neg q \vee p \vee \neg p$$

og

$$\neg q \vee q \vee \neg p.$$

Eksempel (Fortsatt).

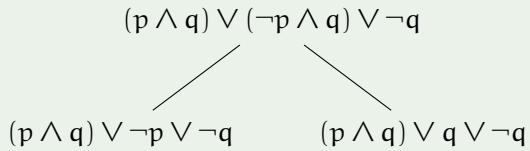
- Da kan vi starte konstruksjonen av et bevistre ved å bruke tilsvarende forgrening:



- Vi ser at vi allerede har fått aksiomer på de to barna, så vi kan bruke dem som bladnoder, og har et bevistre.

Eksempel.

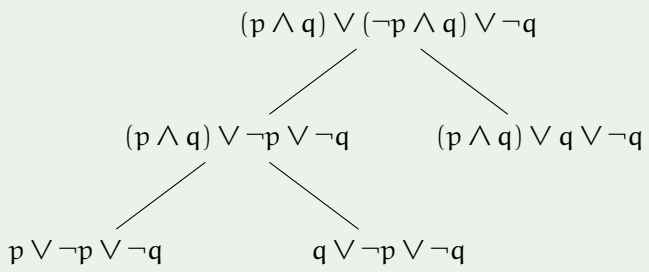
- $(p \wedge q) \vee (\neg p \wedge q) \vee \neg q$
- Her er det to ledd med $\wedge$, og vi kan velge hvilket vi vil løse opp først.
- Vi velger det andre og får forgreningen



- Her er ikke venstre barn et aksiom, men høyre barn er det.

Eksempel (Fortsatt).

- Vi prøver oss derfor med en ytterligere forgrening fra venstre barn.
- Det gir treet



- Vi ser at dette treet er et bevistre.

- Denne metoden vil alltid gi oss et tre.
- Hvis vi ender opp med en bladnode som ikke er et aksiom, er utsagnet vi startet med ikke en tautologi.
- Hvis vi ender opp med aksiomer i alle bladnodene, er utsagnet en tautologi.
- Ytterligere utdypinger kan vi ta på tavlen om ønskelig.
- Vi har ikke fått tid til å trene så mye på arbeid med bevistrær som ønskelig, men de inngår i pensum likevel.
- Hvis vi utvider språket til å omfatte kvantorer, spiller slike bevistrær en stor rolle, både i de teoretiske studiene og i praktisk bevissøk.
- I forbindelse med syntakstrærne har vi sett på infiks notasjon, polsk notasjon og baklengs polsk notasjon.
- Det er meningen at man skal kunne skrive ned syntakstreet til en term eller et uttrykk, og ved hjelp av det kunne finne frem til hvordan termen eller uttrykket ser ut med hhv. infiks, polsk og omvendt polsk notasjon.
- I denne forbindelse har vi også sett på unifisering og unifiseringsalgoritmen.
- Unifiseringsproblemet generelt går ut på at vi har n par t_1, s_1 opp til t_n, s_n av termer i et språk.
- I disse termene kan det forekomme variable $x_1, \ldots, x_k$.
- Er det mulig å erstatte alle variablene $x_1, \ldots, x_k$ med termer $r_1, \ldots, r_k$ slik at begge sider av hvert enkelt par blir syntaktisk like?
- Unifisering er eksamensrelevant.
- Hvis det blir gitt en oppgave om unifisering til eksamen, av den type vi er vant med, vil vi etter opfordring fra en student bruke $*$ for multiplikasjon i stedet for $\times$, for lettere å kunne skille multiplikasjon fra variabelen x.

Kapittel 13

- Det siste kapitlet omhandler algoritmer og kompleksitet.
- Gjennomgangen av dette kapitlet gikk litt fort på slutten, men det kan være aktuelt med en enkel oppgave fra dette stoffet.
- Det som da vil være aktuelt er å finne frem til tidskompleksiteten til algoritmen bak en enkel pseudokode, å kunne beskrive denne ved hjelp av O-notasjonen, og å kunne vurdere om algoritmen er gjennomførbar (tractable) eller ikke.

- Husk at en algoritme er, per definisjon, gjennomførbar hvis tidskompleksiteten er $O(n^k)$ for et naturlig tall k.
- Vi har tidligere sagt at det ikke er aktuelt med eksamensoppgaver som i vanskelighetsgrad og form går ut over det som er gitt som øvingsoppgaver gjennom semesteret.
- Det skader likevel ikke å trene litt.
- Vi ser på to eksempler til.

Oppgave.

Betrakt følgende pseudokode:

1 *Input* k [$k \in \mathbb{N}$]
2 *Input* $n_1, \ldots, n_k$ [Sekvens av hele tall med standard digital representasjon]
3 $x \leftarrow n_1$
4 **For** $i = 2$ **to** k **do**
  4.1 **If** $x < n_i$ **then**
    4.1.1 $x \leftarrow n_i$
5 *Output* x

Oppgave (Fortsatt).

a) Forklar sammenhengen mellom input og output.
b) Finn et uttrykk for tidskompleksiteten.

Løsning

a) Siden vi hele veien lar x ha verdien til den største n_i lest så langt, vil x til slutt bli det største av tallene $n_1, \ldots, n_k$.
b) Siden vi bruker standard digital representasjon på tallene n_i, har antall bits vi bruker til å representere hvert enkelt tall en fast lengde.

   Derfor er k et mål på hvor stort input er.

   Den mest tidkrevende enkeltoperasjonen er sammenlikningen av x og n_i, en operasjon vi utfører k ganger.

   Tidskompleksiteten blir da $O(k)$.

Oppgave.

Betrakt følgende pseudokode:

1 *Input* k [$k \in \mathbb{N}$]
2 *Input* n [$n \in \mathbb{N}$]

3 $x \leftarrow n$

4 **For** $i = 2$ **to** k **do**

4.1 **If** x like tall **then**

4.1.1 $x \leftarrow \frac{x}{2}$

else

4.1.2 $x \leftarrow 3x + 1$

5 *Output* x

Oppgave (Fortsatt).
Bestem tidskompleksiteten til algoritmen over.

Løsning

Når ikke annet er nevnt, skal vi anta at input er gitt på binær form.

La m være antall bits vi bruker til å representere input.

m er av størrelsesorden $\lg(k) + \lg(n)$.

De tre leddene i hovedløkka er omtrent like arbeidskrevende, så det blir lengden på hovedløkka som bestemmer tidskompleksiteten.

Lengden på denne løkka er $k - 1$, som er $O(2^m)$ hvor m er størrelsen på input.

Siden vi uansett om prosessen stabiliserer seg eller ikke insisterer på å gjennomføre $k - 1$ runder, vil $O(2^m)$ være tidskompleksiteten.