MAT1030 – Forelesning 31

Repetisjon

Dag Normann - 18. mai 2010

(Sist oppdatert: 2010-05-18 14:11)

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

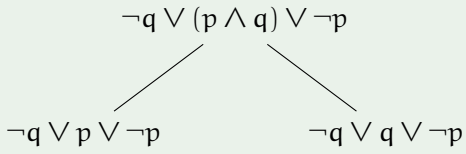
$$\neg q \vee p \vee \neg p$$

og

$$\neg q \vee q \vee \neg p.$$

Eksempel (Fortsatt).

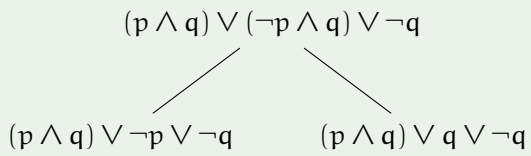
- Da kan vi starte konstruksjonen av et bevistre ved å bruke tilsvarende forgrening:



- Vi ser at vi allerede har fått aksiomer på de to barna, så vi kan bruke dem som bladnoder, og har et bevistre.

Eksempel.

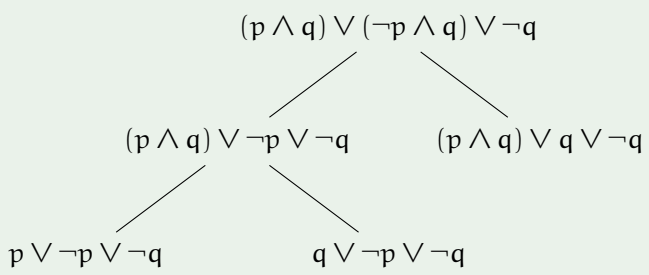
- $(p \wedge q) \vee (\neg p \wedge q) \vee \neg q$
- Her er det to ledd med $\wedge$, og vi kan velge hvilket vi vil løse opp først.
- Vi velger det andre og får forgreningen



- 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.