MAT1030 – Diskret matematikk

Forelesning 31:

Dag Normann

Matematisk Institutt, Universitetet i Oslo

19. mai 2008

Informasjon

Jeg er blitt bedt om å opplyse om hvilke forelesninger det er som inneholder eksamensrelevant stoff som ikke står i læreboka.

Det er

- Forelesning 17, 10. mars.
- Forelesning 18, 12. mars.
- Forelesning 26, 28. april.
- Forelesning 27, 30. april.
- Forelesning 28, 5. mai.

Kompleksitetsteori

- Vi nærmer oss slutten, selv på avsnittet om komplekistetsteori.
- Vi har vurdert tre forenklinger, eller tilnærminger, vi kan gjøre når vi skal vurdere tidskompleksiteten av en algoritme
  1. Vurder den mest tidkrevende operasjonen.
  2. Betrakt alltid det verste tilfellet når flere input kan ha samme størrelse.
  3. Anta at input er stort.

Kompleksitetsteori

- I noen eksempler har vi sett på hvor lange løkker vi kan ha, om løkker inneholder underløkker (i tilfellet hvor vi undersøker om en graf er sammenhengende), og vi har funnet frem til en størrelsesorden på kompleksiteten, som $f(n) = n^2$ eller $g(n) = n^3$.
- Vi kan for eksempel konkludere med at regnetiden vil være begrenset av $c \cdot n^2$ for input av størrelse n, hvor vi ikke bryr oss om å finne verdien på c.
- Det er flere grunner til at vi ikke vil bry oss om å finne verdien på c, tre av de viktigste er:
  1. Verdien på c avhenger av programmeringsspråk, maskin og andre varierende forhold.
  2. Selv i en konkret situasjon kan det være vanskelig å bestemme en fornuftig verdi av c.
  3. Det har liten teknologisk interesse i de fleste tilfellene hva verdien på c er.
- Alle disse betraktningene leder opp mot den fjerde tilnærmingen vi skal gjøre, og til det begrepet vi skal innføre.

Kompleksitetsteori

- 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 regel presise.
- Den vil også gjøre det mer presist å avgjøre hva som faktisk er de verste tilfellene.

Kompleksitetsteori

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 finnes en positiv konstant c slik at

$$f(n) \leq c \cdot g(n)$$

for alle tilstrekkelig store n.

Kompleksitetsteori

- Med "tilstrekkelig store" mener vi at det finnes en n_0 slik at ulikheten holder for alle $n \geq n_0$.
- 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:

$$\exists c > 0 \exists n_0 \forall n \geq n_0 (f(n) \leq c \cdot g(n)).$$

Kompleksitetsteori

- 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 bits 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.

Kompleksitetsteori

- 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.

Kompleksitetsteori

Eksempel

La $f(n) = 3n + 2$ og $g(n) = 2n$.

Da er f $O(g)$ fordi $f(n) \leq 2g(n)$ når $n \geq 2$.

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:

$f(n) \leq 10^6 \cdot g(n)$ for alle n.

Vi har også at $f(n) \leq g(n)$ for alle $n \geq 10^6$.

Kompleksitetsteori

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 finnes noen c som duger.

For å vise at c ikke duger, må vi vise at det finnes 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.

Kompleksitetsteori

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 $c = 3 + 10 + 2 + 20 = 35$, det vil si, summen av alle koeffisientene i f.

Husk at $n \geq 1$ her.

Da er

$$f(n) = 3n^4 + 10n^3 + 2n + 20 \leq$$
$$3n^4 + 10n^4 + 2n^4 + 20n^4 =$$
$$(3 + 10 + 2 + 20)n^4 = c \cdot g(n)$$

Kompleksitetsteori

Denne metoden kan brukes til å vise følgende teorem.

Teorem

Hvis f er en polynomfunksjon med grad $\leq k$ 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.

Kompleksitetsteori

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 finnes vilkårlig store n slik at $c \cdot g(n) < f(n)$.

Velger vi $n > (2 + 4 + 6)c = 12c$ får vi

$$f(n) = n^3 > c \cdot (2 + 4 + 6)n^2$$
$$\geq c \cdot g(n) \text{ (som i beviset for teoremet).}$$

Kompleksitetsteori

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 $O(g)$.

Kompleksitetsteori

- 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 finnes det $c > 0$ og n_0 slik at hvis $n \geq n_0$ vil

$$f(n) \leq c \cdot g(n).$$

Kompleksitetsteori

- Videre finnes det $d > 0$ og n_1 slik at hvis $n \geq n_1$ vil

$$g(n) \geq d \cdot h(n).$$

- Hvis vi nå lar $n \geq max\{n_0, n_1\}$ har vi at

$$f(n) \geq c \cdot g(n) \geq c \cdot d \cdot h(n),$$

så konstanten $c \cdot d > 0$ kan brukes til å vise at f er $O(h)$.

- Dette viser at relasjonen er transitiv.

Kompleksitetsteori

- 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 $f(n) \leq 1 \cdot f(n)$, så f er $O(f)$ for alle f.
- Det viser at relasjonen er refleksiv.

Kompleksitetsteori

- 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.

Sortering

- Hvis man skal kunne skaffe seg oversikt over informasjonen i en stor datamengde, er det viktig å kunne sortere disse dataene etter visse kriterier.
- Det finnes en elektronisk tabell over ca. 10.000 vitenskapelige tidskrift som ansatte ved norske universiteter og høyskoler har anledning til å publisere artikler i.
- Denne tabellen inneholder mye informasjon om det enkelte tidskriftet, som ISSN-nummer, navn, fagområde, hvor mange artikler som er trykket der siste år, hvor mange artikler som er trykket der de siste fem årene, samt noen indekser som skal måle kvaliteten på tidskriftet.
- For enkelte formål kan det være aktuelt å sortere disse ca. 10.000 tidskriftene etter navn, for andre etter fagområder, og enkelte ganger er det mest formålstjenlig å sortere tidskriftene etter et av kvalitetsmålene.

Sortering

- Denne tabellen, og hvordan faglaugene forholder seg til informasjonen i den, har betydning for deler av finansieringen av universiteter og høyskoler.
- Det er derfor viktig at de som bruker denne tabellen til å formulere kriteriene for denne finansieringen raskt kan sortere dataene slik man har bruk for der og da.
- Det vi skal se på nå er en enkel metode for å sortere data, og på hvordan denne metoden kan forbedres slik at kompleksiteten bringes betraktelig ned.

Sortering

- Begge de algoritmene vi skal se på illustrerer nytten av å forstå teorien rundt relasjoner.
- Vi vil bruke symbolet $<$ og forestille oss at vi arbeider med tall, men vi har bare bruk for at $<$ er en transitiv, irrefleksiv relasjon slik at vi for alle a og b har at $a = b$, $a < b$ eller $b < a$, men at vi ikke har flere muligheter.
- La oss se på et eksempel.
- Vi har gitt 10 tall uten noen spesiell orden, $5, 9, 4, 1, 7, 12, 3, 6, 2, 8$.
- Vi vil sortere disse i voksende orden.
- Dette vil vi i første omgang gjøre i ti operasjoner.

Sortering

Eksempel (Sortering av $5, 9, 4, 1, 7, 12, 3, 6, 2, 8$)

1. Sorter 5 ; Rest $9, 4, 1, 7, 12, 3, 6, 2, 8$
2. Sorter $5, 9$; Rest $4, 1, 7, 12, 3, 6, 2, 8$
3. Sorter $5, 9, 4 \to 5, 4, 9 \to 4, 5, 9$; Rest $1, 7, 12, 3, 2, 8$
4. Sorter $4, 5, 9, 1 \to 4, 5, 1, 9 \to 4, 1, 5, 9 \to 1, 4, 5, 9$; Rest $7, 12, 3, 2, 8$
5. Sorter $1, 4, 5, 9, 7 \to 1, 4, 5, 7, 9$; Rest $12, 3, 2, 8$
6. Sorter $1, 4, 5, 7, 9, 12$; Rest $3, 2, 8$
7. Sorter $1, 4, 5, 7, 9, 12, 3 \to \cdots \to 1, 4, 3, 5, 7, 9, 12 \to 1, 3, 4, 5, 7, 9, 12$; Rest $2, 8$
8. Sorter $1, 3, 4, 5, 7, 9, 12, 2 \to \cdots \to 1, 3, 2, 4, 5, 7, 8, 9, 12 \to 1, 2, 3, 4, 5, 7, 9, 12$; Rest 8
9. Tilsist flytter vi 8 nedover i den sorterte delen av listen til vi finner plassen dens, og den sorterte listen blir $1, 2, 3, 4, 5, 7, 8, 9, 12$.

Sortering

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** $j \geq 1$ *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* $x_1, x_2, \ldots, x_n$

Detaljforklaring finner sted muntlig på forelesningen.

Sortering

- 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.

Sortering

- 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?

Sortering

- 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.

Sortering

- 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

  $1, 3, 7, 8, 9, 12, 14, 22, 23, 25, 31, 37, 40, 41, 44, 47, 50$

Sortering

- 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.
- Poenget her at vi trenger en notasjon for å kunne snakke om tidskompleksiteten til algoritmen.

Sortering

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 mer effektive sorteringsalgoritmen vil ha en tidskompleksitet som er $O(n \cdot lg(n))$, n fordi vi fortsatt må plassere n objekter på riktig plass, men $lg(n)$ fordi dette er tidskompleksiteten til den nye plasseringsalgoritmen.

Gjennomførbare algoritmer

- Vi har lovet 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 sett på 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.

Gjennomførbare algoritmer

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.

Gjennomførbare algoritmer

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, og det er antall bits 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 $2 \leq m \leq \sqrt{n}$.

Gjennomførbare algoritmer

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 $n^{\frac{k}{2}}$ antall divisjoner vi må utføre for å bestemme om n er et primtall eller ikke.
- I kryptografi er vi interesserte i primtall med hundre siffre eller mer, eller helst i produkter av to eller tre slike primtall.
- Da vil de naive metodene sprenge alle grenser for anstendig kompleksitet.

Gjennomførbare algoritmer

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 Hamilton-sti på som er vesentlig mer effektive enn den naive, prøv alle stier med $n-1$ kanter og se om en av dem tilfeldigvis skulle være en Hamiltonsti.

Gjennomførbare algoritmer

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.

Gjennomførbare algoritmer

Definisjon

Vi sier at en algoritme er gjennomførbar (tractable på engelsk) hvis tidskompleksiteten er $O(n^k)$ for en k.

Gjennomførbare algoritmer

Merk

- 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. k ligger gjerne 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.

Gjennomførbare algoritmer

- 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.

Gjennomførbare algoritmer

- 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.
- 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**.
- 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.

- Det finnes mange temaer som faller inn under sekkebetegnelsen diskret matematikk som vi ikke har tatt opp ennå.
- Det finnes også temaer det ville vært naturlig for oss å ta opp, eksempelvis sammenhengen mellom trerekursjon og kompleksitet.
- Siden vi dog trenger noe tid til repetering, og pensum har blitt passe stort, gjenstår det bare å si at presentasjonen av nytt stoff i MAT1030, våren 2008, har tatt

SLUTT