MAT1030 – Forelesning 29

Kompleksitetsteori

Dag Normann - 11. mai 2010

(Sist oppdatert: 2010-05-11 14:23)

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.