Forelesning 29

Kompleksitetsteori

Dag Normann - 7. mai 2008

KAPITTEL 13: Kompleksitetsteori

Meldinger:

Det blir hovedsaklig tavleregning på plenumsregningen i morgen.

15/5-08 vil D.N. vikariere for Roger på plenumsregningen.

Oppgavene til neste uke vil i hovedsak bestå av gamle eksamensoppgaver.

Kompleksitetsteori

- Vi skal nå starte på siste kapittel i boka,
  og dermed siste del av pensum.
- 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.

Kompleksitetsteori

Eksempel.

- Sommeren 1986? fikk D.N. 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.

Kompleksitetsteori

Eksempel (Fortsatt).

- Logikk-gruppen hadde to nyinnkjøpte Xerox arbeidsplassmaskiner, og første forsøk på å imøtekomme ønsket var å skrive et LISP-program (med hjelp fra stipendiat Erik Colban) 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* $x_1, \ldots, x_{56}$.

Kompleksitetsteori

Eksempel (Fortsatt).

- Etter en god lunsj ble det klart at dette ville ta hele dagen, og trolig natten med.
- Vi måtte finne en raskere algoritme.
- Vi laget en algoritme basert på en matematisk analyse av problemet (analysen tok ca. 10 minutter)
- Den nye algoritmen ga svaret i løpet av ca 2 sekunder.

Kompleksitetsteori

Eksempel (Fortsatt).

- 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 39, mellom 30 og 39 osv.
- Legger vi sammen får vi fordelingen av tverrsummer for alle tall mellom 0 og 99.

Kompleksitetsteori

Eksempel (Fortsatt).

- 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 i antall lodd, og ikke i antall lodd.
- Vi skal komme tilbake til slike fenomener senere, og sette navn på dem. Forklaringen er justert i forhold til det som ble sagt på forelesningen.
-

Kompleksitetsteori

- I kompleksitetsteori er det ofte to størrelser man prøver å finne
  - Hvor lang tid tar det å følge en algoritme.
  - Hvor mye lagerplass må man sette av for at algoritmen skal ha den informasjonen den trenger til enhver tid.
- Vi skal konsentrere oss om tidskompleksitet, *time complexity* og la plasskompleksitet, *space complexity* være udiskutert.

Kompleksitetsteori

- 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.

Kompleksitetsteori

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 kan avhenge av hvor stort minne vi har satt av til hjelp for programmet.
- Svaret kan avhenge av om vi må dele ressursene med andre.
- Svaret vil avhenge av input, og av hvordan input representeres digitalt.
- med mer, med mer.

Vi trenger imidlertid ofte ikke å kjenne alle disse forholdene for å kunne sammenlikne algoritmer eller for å vurdere om en algoritme er praktisk gjennomførbar eller ikke.

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.

Kompleksitetsteori

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

Kompleksitetsteori

Eksempel (Fortsatt).

- Vi har gitt en pseudokode for å beregne

$$f(n) = \sum_{j=1}^{n!} j.$$

- Vi må anta at det normalt krever mere 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 ca. $n!$ 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.

Kompleksitetsteori

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

Kompleksitetsteori

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.

Kompleksitetsteori

- 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.

Kompleksitetsteori

- 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 utspendte 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.