MAT1030 – Diskret Matematikk

Forelesning 30: Kompleksitetsteori

Roger Antonsen

Institutt for informatikk, Universitetet i Oslo

19. mai 2009

(Sist oppdatert: 2009-05-19 15:04)

Forelesning 30: Kompleksitetsteori

Oppsummering

- I dag er siste forelesning med nytt stoff!
- I morgen blir det ingen forelesning.
- Neste uke ønskereprise og litt om veien videre.
- Innholdet? Opp til dere.
- Nå, først litt repetisjon.
- Deretter, om sortering og gjennomførbarhet.

Oppsummering

- 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

- Dette er et eksempel på en analyse av 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.

Sorteringsalgoritmer

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 $1, 2, 3, 4, 5, 7, 8, 9, 12$.

Sorteringsalgoritmer

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* $x_1, x_2, \ldots, x_n$

Sorteringsalgoritmer

- 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.

Sorteringsalgoritmer

- 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?

Sorteringsalgoritmer

- 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.

Sorteringsalgoritmer

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

Sorteringsalgoritmer

- 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 $O(n^2)$.
- Poenget her at vi trenger en notasjon for å kunne snakke om tidskompleksiteter som er mellom $O(n)$ og $O(n^2)$.

Sorteringsalgoritmer

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.

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 (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 $2 \leqslant m \leqslant \sqrt{n}$.

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 $2^{\frac{k}{2}}$ 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.

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 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.

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.

Vi merker oss følgende sammenhenger.

- 2^n er ikke $O(n^k)$ for noen verdi av k
- 2^n er $O(n!)$

Gjennomførbare algoritmer

- 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.

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.

Slutt