Kompleksitet

IN2010 – Algoritmer og datastrukturer

Lars Tveito

Institutt for informatikk, Universitetet i Oslo
larstvei@ifi.uio.no

Oversikt

Oversikt

- Så langt har vi studert problemer som det finnes gode og effektive løsninger på
- Vi har sett en rekke teknikker som kan brukes til å løse mange problemer
- Nå skal vi se på problemer hvor ingen av disse teknikkene har ledet til en effektiv løsning
  - Problemer som det kanskje ikke finnes en effektiv løsning på
- Vi skal også se at noen problemer er *uløselige*
- Vi skal endre perspektiv:
  - Fra å tenke på kompleksiteten til en konkret algoritme
  - Til å heller tenke på kompleksiteten til problemet selv

Kompleksiteten til algoritmer

Kompleksiteten til algoritmer

- Kompleksiteten til en algoritme sier noe om hvor mange ressurser den krever
- Som regel er vi interessert i tidsbruk og minnebruk
- Her vil vi konsentrere oss om tidsbruk
- Vi kan ofte resonnere oss frem til hvor mange steg en algoritme krever
  - Eller angi en øvre grense for antall steg
- Hvor fort antall steg vokser i forhold til størrelsen på input gir *kjøretidskompleksiteten* til algoritmen

En ineffektiv løsning på et enkelt problem

ALGORITHM: SUMMER TALLENE FRA 1 TIL n

Input: Et naturlig tall n
Output: Summen av alle tallene fra 1 til n

```
1 Procedure SumTo(n)
2     sum ← 0
3     for i ← 1 to n do
4         for j ← 1 to i do
5             sum ← sum + 1
6     return sum
```

- Ytre løkke: n iterasjoner
- Indre løkke: Varierer med i
- Totalt antall iterasjoner:

$$1 + 2 + \cdots + n = \frac{n \cdot (n+1)}{2}$$

Hvordan regne iterasjoner

- Summere alle iterasjoner:

$$\overbrace{1 + 2 + \cdots + n}^{n}$$

- Å summere første og siste tall gir $n + 1$
- Dersom vi summerer opp alt to ganger får vi:

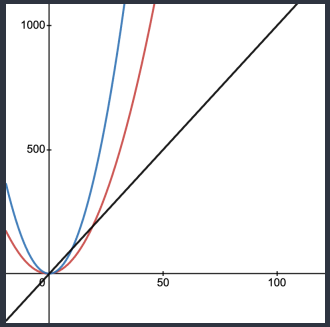
$$\begin{array}{rccccccc} & 1 & + & 2 & + & \cdots & + & n \\ + & n & + & n-1 & + & \cdots & + & 1 \\ \hline = & n+1 & + & n+1 & + & \cdots & + & n+1 \end{array}$$

- Merk at $\overbrace{(n+1) + (n+1) + \cdots + (n+1)}^{n} = n \cdot (n+1)$
- Hvis vi halverer resultatet får vi:

$$1 + 2 + \cdots + n = \frac{n \cdot (n+1)}{2}$$

Litt om kjøretidskompleksitet

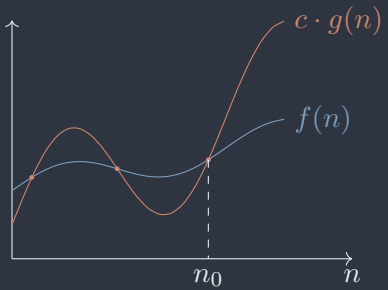
- Kjøretidskompleksitet handler om hvordan kjøretiden vokser når input blir større
- Eksempel: Graf som viser vekst av $10 \cdot n$, n^2 og $\frac{n(n+1)}{2}$
- Kurvene bevarer samme form selv om vi endrer konstantfaktorer
  - n^2 og $\frac{n(n+1)}{2}$ vokser kvadratisk
  - $10 \cdot n$ vokser lineært
- Stor $\mathcal{O}$ notasjon fanger generelle trender, men ignorerer detaljer
- Det gir opphav til følgende perspektiv:
  - Vekst er viktigere enn konstantfaktorer
  - Tregeste leddet styrer veksten
- Det gjør det enklere å resonnere om kjøretid



desmos

Hvordan $\mathcal{O}$ er definert

- La $f(n)$ være kjøretiden for input av størrelse n og la g være en funksjon fra heltall til positive tall
- $f(n)$ er i $\mathcal{O}(g(n))$ hvis det finnes en positiv konstant c, slik at $f(n) \leq c \cdot g(n)$ for alle tilstrekkelig store verdier av n
  - Denne sier at $f(n)$ ikke vokser raskere enn $g(n)$
  - $g(n)$ er en *øvre skranke* for $f(n)$
  - $f(n)$ er begrenset av $g(n)$
- $f(n) = 3 \cdot n^2 + 5 \cdot n + 2$ er i $\mathcal{O}(n^2)$
  - Fordi $4 \cdot n^2$ er størst så lenge $n > 6$



Tilbake til det enkle problemet

- Prosedyren for å summere opp tallene 1 til n var *veldig* inneffektiv
- Den er ikke inneffektiv fordi den er kvadratisk
  - Men fordi det er så lett å se at den kan løses mer effektivt!

```
1 Procedure SumTo(n)
2 |  sum ← 0
3 |  for i ← 1 to n do
4 |  |  sum ← sum + i
5 |  return sum
```

  - Denne er i $\mathcal{O}(n)$
- I analysen over utledet vi en formel som løser problemet direkte

```
1 Procedure SumTo(n)
2 |  return n·(n+1)/2
```

  - Denne er i $\mathcal{O}(1)$

Kompleksiteten til problemer

Kompleksiteten til problemer

- Hittil har vi konsentrert oss på kompleksiteten til en *løsning*
- I kompleksitetsteori er studerer heller vi problemers *iboende* vanskelighetsgrad
- Vanskelighetsgraden er gitt av den *mest effektive* algoritmen som løser problemet
- Dersom du skal finne kompleksiteten til et *problem* er det ikke tilstrekkelig å analysere kompleksiteten til én foreslått løsning
- Du må også kunne bevise at det umulig kan eksistere en mer effektiv algoritme
- Dette er en betydelig vanskeligere øvelse

Avgjørelsesproblemer

- I kompleksitetsteori begrenser vi oss som regel til *avgjørelsesproblemer*
- De tar en *instans* av problemet og et *spørsmål* som besvares med med JA eller NEI
- Tidligere har vi sett på problemer som har hatt rikere output
  - Som regel kan vi formulere et *lignende* avgjørelsesproblem
- Hvis en algoritme eksisterer for et problem, kan den brukes til å besvare det relaterte avgjørelsesproblemet
- Rikere varianter av problemer er *minst like vanskelig* som deres relaterte avgjørelsesproblemer
- Avgjørelsesproblemer er enklere å formalisere som *formelle språk*

Avgjørelsesproblemer (eksempler)

SORT
Instans: To arrayer A_1 og A_2.
Spørsmål: Består A_2 av de samme elementene som A_1 i sortert rekkefølge?

MST-k
Instans: En graf G og et tall k.
Spørsmål: Finnes det et spenntre over G med total vekt mindre enn k?

SCC-k
Instans: En graf G og et tall k.
Spørsmål: Har G minst k sterkt sammenhengende komponenter?

Kompleksitetsklassene P og NP

- Avgjørelsesproblemer kan klassifiseres inn i *kompleksitetsklasser*
- Dette kan vi se på som et forsøk på å gruppere dem etter *hvor vanskelige* de er
  - Samtidig er det en formalisering av hva *hvor vanskelig* egentlig betyr
- Fra dette perspektivet, er vi ikke lenger så opptatt av *nøyaktig* kjøretidskompleksitet
  - Men heller grupper av problemer som har omtrent samme kompleksitet
- Vi skal se på de to mest kjente, og kanskje mest nyttige, kompleksitetsklassene

Kompleksitetsklassen P

- Et avgjørelsesproblem er i P hvis det finnes en algoritme som løser det i polynomiell tid
  - Det vil si at algoritmen er i $\mathcal{O}(n^k)$ for en konstant k
- En intuitiv måte å se for seg polynomiell tid er et program som inneholder flere nøstede løkker

```
1 Procedure Polynomial(n)
2     for i_1 ← 0 to n − 1 do
3         for i_2 ← 0 to n − 1 do
4             ⋮
5             for i_k ← 0 to n − 1 do
6                 Constant(i)                          // O(n^k)
```

- Alle algoritmer vi har sett i kurset så langt har vært polynomielle
- Husk at $\mathcal{O}$ kun gir en øvre grense
  - For eksempel er en algoritme i $\mathcal{O}(\log n)$ også i $\mathcal{O}(n^2)$
- Problemer i P anses som regel som *håndterbare* problemer

Sudoku $n \times n$

- I klassisk Sudoku får du et delvis utfylt 9×9 grid
- En løsning er et utfylt brett, der hver rad, kolonne og boks inneholder tallene 1 til 9 nøyaktig én gang
- Avgjørelsesproblemet:
  SUDOKU $n \times n$
  Instans: Et delvis utfylt $n \times n$ Sudoku-brett.
  Spørsmål: Har brettet en gyldig løsning?
- Det er ingen kjent polynomielltidsløsning for dette problemet
  - Det er heller ikke kjent om det i det hele tatt *kan eksistere* en løsning i polynomiell tid
- Dette problemet vokser fort:
  - For $n = 4$ finnes det 288 gyldige brett
  - For $n = 9$ finnes det $6\,670\,903\,752\,021\,072\,936\,960$ gyldige brett
- Det er allikevel enkelt å *sjekke* at en gitt løsning er gyldig

7	1	4	3	5	6	8	2	9
5	8	2	4	1	9	6	7	3
6	9	3	7	2	8	4	5	1
3	7	9	8	6	4	5	1	2
2	5	6	1	9	3	7	8	4
1	4	8	2	7	5	9	3	6
4	2	5	6	3	7	1	9	8
8	3	7	9	4	1	2	6	5
9	6	1	5	8	2	3	4	7

Verifisering av løsninger med sertifikater

- Avgjørelsesproblemer tar input og svarer JA eller NEI
- Gitt en JA-instans av et avgjørelsesproblem kan vi *bevise* det ved å fremvise et *sertifikat*
- For Sudoku er spørsmålet om et gitt brett har en løsning
  - Dersom svaret er JA, kan vi skrive en algoritme som verifiserer det i *polynomiell tid*
  - En slik algoritme tar instansen og det løste brettet og sjekker at alle regler er fulgt
  - Det løste brettet fungerer her som et *sertifikat*
- Denne *verifikasjonen* kan gjøres på polynomiell tid
  - Selv hvis det å *finne* sertifikatet ikke kan det

Kompleksitetsklassen NP

- Et avgjørelsesproblem er i NP hvis en løsning kan verifiseres i polynomiell tid
- Alle problemer i P er også i NP
  - Hvis vi kan løse problemet i polynomiell tid, så kan vi også verifisere en løsning i polynomiell tid



- NP er en forkortelse for «Nondeterministic Polynomial time»
- Alle problemer i NP er *minst like vanskelige* som de i P
- Det er ikke visst om $P = NP$ eller $P \neq NP$
  - Et av de største uløste problemene i matematikk og informatikk
  - Løsning gir en premie på én million dollar

Reduksjoner

Oversettelse mellom problemer

- Problemløsing involverer ofte å oversette problemene til andre problemer som er lettere å løse
- I kompleksitetsteori gjør vi dette systematisk gjennom *reduksjoner*
- Reduksjoner gir oss en måte å sammenligne vanskelighetsgraden av ulike problemer

Polynomtidsreduksjoner

- En polynomtidsreduksjon fra problem A til problem B oversetter instanser av A til instanser av B i polynomiell tid
- Korrekt reduksjon krever at JA-instanser av A blir JA-instanser av B, og tilsvarende for NEI-instanser
- Alle avgjørelsesproblemer i P kan reduseres til hverandre

Polynomtidsreduksjoner (eksempel)

EVEN
Instans: Et heltall n.
Spørsmål: Er n et partall?

ODD
Instans: Et heltall n.
Spørsmål: Er n et oddetall?

- En instans av **EVEN** er et heltall n
- Instansen kan oversettes til en instans av **ODD** ved å transformere n til $n + 1$
- Denne er korrekt, fordi
  - en JA-instans av **EVEN** blir en JA-instans av **ODD**
  - en NEI-instans av **EVEN** blir en NEI-instans av **ODD**
  - reduksjonen er i polynomiell tid
- Siden vi kan redusere **EVEN** til **ODD**, så kan vi konkludere med at **ODD** er *minst like vanskelig* som **EVEN**
- Reduksjonen fra **ODD** til **EVEN** er helt tilsvarende
  - Derfor er **EVEN** minst like vanskelig som **ODD**
  - Altså er problemene like vanskelige

Polynomtidsreduksjoner (eksempel)

SORT
Instans: To arrayer A_1 og A_2.
Spørsmål: Består A_2 av de samme elementene som A_1 i sortert rekkefølge?

SCC-k
Instans: En graf G og et tall k.
Spørsmål: Har G minst k sterkt sammenhengende komponenter?

- Alle avgjørelsesproblemer i P kan polynomtidsreduseres til hverandre
- Vi rekker å *løse problemene* i polynomiell tid
- **SCC-k** kan reduseres til **SORT**
- Løs **SCC-k**(G, k)
  - Hvis løsningen er JA, oversett instansen til $A_1 = [1, 2, 3]$, $A_2 = [1, 2, 3]$
  - Hvis løsningen er NEI, oversett instansen til $A_1 = [1, 2, 3]$, $A_2 = [2, 1, 3]$

NP-harde problemer

- Et problem er NP-hardt dersom det er *minst like vanskelig* som alle problemer i NP
- **SAT** var det første problemet som ble bevist å være NP-hardt
- For å vise at et problem A er NP-hardt, kan man redusere et kjent NP-hardt problem, som **SAT**, til A

NP-komplette problemer

- Et problem A er NP-komplett hvis det både
  1. er i NP (kan verifiseres i polynomiell tid)
  2. er NP-hardt (hvert problem i NP kan polynomtidsreduseres til A)
- Disse problemene er antatt vanskelige å løse, men løsningene er enkle å verifisere
- NP-komplette problemer spiller en viktig rolle i kryptografi

Men er $P = NP$?

- Alle NP-komplette problemer kan reduseres til hverandre
- Dersom man finner en polynomiell løsning på *ett* NP-komplett problem
  - så kan alle NP-komplette problemer reduseres til dette problemet
  - og med det har man bevist at $P = NP$
- Dersom en slik løsning er rask nok til å kunne beregnes på moderne datamaskiner, så ville det hatt enorme konsekvenser

Beregnbarhet

- Vi avslutter med en kort historie om beregnbarhet
- Spørsmålet er om det finnes problemer så vanskelige at det ikke går an å beregne et svar, uansett hvor lang tid det tar
  - Finnes det problemer som er grunnleggende uløselige?
  - Altså hvor det kreves uendelig tid for å besvare spørsmålet
- Dette var spørsmålet David Hilbert stilte i begynnelsen av 1900-tallet
- Svaret kom fra Alonzo Church og senere Alan Turing i 1936
  - Det finnes uløselige problemer!

Stoppeproblemet er uløselig

- Det grunnleggende uløselige problemet kalles *stoppeproblemet*
- Det presenteres som et avgjørelsesproblem:
  HALT
  Instans: En algoritme A, og input x til A.
  Spørsmål: Terminerer A på input x?
- Stoppeproblemet er uløselig, og dette kan bevises ved et motsigelsesbevis

Motsigelsesbevis for stoppeproblemet

- Anta for motsigelse at det finnes en algoritme H som gir riktig svar på alle instanser av **HALT**
- Konstruerer algoritme D som tar en algoritme A som input:
  - Sjekk om $H(A, A)$, terminerer A med seg selv som input?
    - Hvis JA: gå inn i en evig løkke
    - Hvis NEI: terminer med svar JA
- Hva skjer hvis vi kjører $D(D)$, altså D med D som input?
  - Dersom $H(D, D)$ svarer at $D(D)$ terminerer, så terminerer det ikke
  - Dersom $H(D, D)$ svarer at $D(D)$ ikke terminerer, så svarer $D(D)$ JA
- I begge tilfeller leder antagelsen om at H finnes til en motsigelse
- Derfor kan ikke en slik algoritme H eksistere
- Med andre ord: ingen algoritme kan løse stoppeproblemet