

01. Problemer og algoritmer

Bonus: induksjonseksempel, L-problemet, sum ved rekursjon og iterasjon med induktivt bevis

Forelesning 1

Vi starter med fagfeltets grunnleggende byggesteiner, og skisserer et rammeverk for å tilegne seg resten av stoffet. Spesielt viktig er ideen bak induksjon og rekursjon: Vi trenger bare se på *siste trinn*, og kan *anta* at resten er på plass.

Pensum

- ☐ Innledningen til del I
- ☐ Kap. 1. The role of algorithms in computing
- ☐ Kap. 2. Getting started: Innl., 2.1 og 2.2
- ☐ Kap. 3. Characterizing running times: Innl., 3.1 og 3.2

Læringsmål

- [A₁] Forstå bokas *pseudokode*-konvensjoner
- [A₂] Kjenne egenskapene til *random-access machine*-modellen (RAM)
- [A₃] Kunne definere *problem*, *instans* og *problemstørrelse*
- [A₄] Kunne definere *asymptotisk notasjon*, O , Ω , Θ , o og ω .
- [A₅] Kunne definere *best-case*, *average-case* og *worst-case*
- [A₆] Forstå at alle av O , Ω , Θ , o og ω kan beskrive *best-*, *worst-* og *average-case*
- [A₇] Forstå *løkkeinvarianter* og *induksjon*
- [A₈] Forstå *rekursiv dekomponering* og *induksjon over delproblemer*
- [A₉] Forstå INSERTION-SORT

1

mlh@ntnu.no

Forkunnskaper og pseudokode

Forkunnskaper til emner er logaritmer, spesielt logaritmisk notasjon og regneregler for logaritmer slik at det er enklere å se at teoremer som masterteoremet går opp. Det er i tillegg nødvendig å kjenne til modulo operatoren som er resten av en brøk.

Læreboka er CLRS (https://en.wikipedia.org/wiki/Introduction_to_Algorithms) 4. edition. Boka har gått igjennom betydelige endringer siden 3. edition og anbefales dermed ikke. Pseudokoden i boka antar i de fleste tilfeller at indeksing av lister begynner på 1.

Problemløsning og dekomponering

Det finnes mange problemer som datamaskiner kan løse, men ikke alle løsninger til problemene fungerer, spesielt ikke på stor skala.

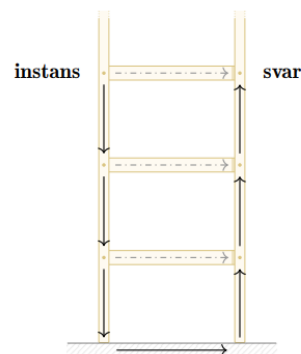
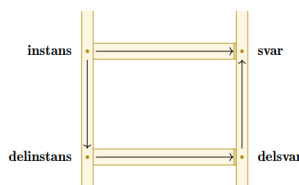
Eksempel fra forelesning: hvis vi skulle brute-force matching mellom donor og mottaker så hadde vi kun matchet i overkant av hundre personer på 10000000... ganger universets levetid. Dette er estimert til å ta en halvtime for alle menneskene på kloden med Chandran-Hochbaum algoritmen.

Det er derfor viktig at vi bruker en mer optimal løsning for å løse problemet. Hvis vi bruker en ordentlig algoritme istedenfor å brute-force problemet kan vi nå realistiske kjøretider.

Et problem kan beskrives som en general relasjon mellom input og output. Begrepet for en enkel, konkret input er en instans. Siden algoritmer ofte jobber på en eller annen type datastruktur som en liste eller et tre kan det være nyttig å dekomponere problemet i mindre problemer (instanser).

På denne måten kan vi følge en metode lik matematisk induksjon for å skape løsninger for problemene. Vi bryter problemet ned til et "base case" som vi kan bevise at stemmer. Deretter bygger vi opp løsningen ved hjelp av induksjon.

$$\begin{array}{c} \text{vilkårlig} \\ \downarrow \\ \frac{P(a)}{\forall x P(x)} \end{array} \quad \begin{array}{c} P \quad H \\ \vdots \\ Q \\ \hline P \Rightarrow Q \quad H \end{array} \quad \frac{P \Rightarrow Q, P}{Q}$$



For å gjøre det enklere å bryte opp problemet kan vi bruke følgende tabell for å få en fin mental oversikt over problemet vi ønsker å løse. Først fyller vi ut "Instans" og "Løsning"

som tilsvarer input og output til problemet. Deretter jobber vi oss nedover på vestre siden og til slutt går vi opp på høyre side og løser resten av problemet. Nedenfor er eksempel for løsning av insertion-sort.

Instans	Løsning
Dekomponering	Kombinasjon 1B
Delløsning	Delløsning 1B
Grunntilfelle	Grunnløsning

Instans	Løsning
En sekvens $A[1:n]$	$A[1:n]$, sortert
Dekomponering	Kombinasjon 1B
Legg $A[n]$ til side	Sett inn $A[n]$ på rett plass i sortert $A[1:n-1]$
Delløsning	Delløsning 1B
$A[1:n-1]$	$A[1:n-1]$, sortert
Grunntilfelle	Grunnløsning
$A[1]$	$A[1]$

I kode har vi to måter å løse slike rekursive induksjonsproblemer; løkker og rekursjon. En løkkeinvariant er et delsteg i løsningen for en iterasjon av løsningen til problemet. Vi beviser at løsningen vår er rett for grunntilfellet, og gjennom hvert steg i løkkeinvarianten, beviser at delløsningen er rett både før og etter. På denne måten kan vi induktivt bevise at sluttløsningen vår er riktig. Om dette gjøres ved en faktisk løkke eller rekursjon har ikke noe å si. Som alltid, må vi bevise at løkka eller rekursjonen stopper.


Insertion Sort

Insertion Sort er en sorteringsalgoritme som fungerer ved å ta hvert element i lista, og flytte det mot venstre helt til det ligger i riktig posisjon i den sorterte lista. Insertion Sort er en stabil sorteringsalgoritme som betyr at den beholder rekkefølgen på like elementer.

- Best case: $O(n)$ hvis lista er sortert fra før av
- Worst case: $O(n^2)$ hvis lista er sortert fra høyt til lavt (revers)
- Average case: $O(n^2)$

algdat/InsertionSort.scala at main · matsjla/algdat

This file contains bidirectional Unicode text that may be interpreted or compiled differently than what appears below. To review, open the file in an editor that reveals hidden Unicode characters. Learn

 <https://github.com/matsjla/algdat/blob/main/src/main/scala/com/supergrecko/dsa/sorting/InsertionSort.scala>

matsjla/algdat

Implementation of all algorithms and data structures from TDT4120 at NTNU in Scala



1 Contributor 0 Issues 0 Stars 0 Forks

Asymptotisk notasjon

Asymptotisk notasjon er notasjon som ofte brukes for å vise hvor ressurseffektiv en algoritme eller løsning er. Man ser ofte på kjøretid, men man kan også ha asymptotisk notasjon av minnebruk, instruksjonssykluser og liknende data. Nedenfor er skalaen på de forskjellige formene.

asymptotisk notasjon

Det store tallet vi hadde tidligere hadde en eksponent på 85 430...

$\lg n$	logaritmisk	$2 \times 10^{949\,978\,419\,116\,565}$
\sqrt{n}		1×10^{31}
n	lineær	3×10^{15}
$n \lg n$	linearitmisk	7×10^{13}
n^2	kvadratisk	6×10^7
n^3	kubisk	1×10^5
2^n	eksponentiell	51
$n!$	faktoriell	17

Tallene her er altså hvor stor n kan bli før det tar mer enn ett år å bli ferdig, dersom vi bruker f(n) mikrosekunder, der f(n) er funksjonen til venstre.

Alt på formen n^k kaller vi *polynomisk*
(Ta en kikk på Problem 1-1)

Når vi bruker asymptotisk notasjon er vi kun interessert i en veldig grov størrelsesorden. Dermed dropper vi konstanter og lavere ordens ledd.

$$\Theta(n^2 + n + 10) = \Theta(n^2)$$

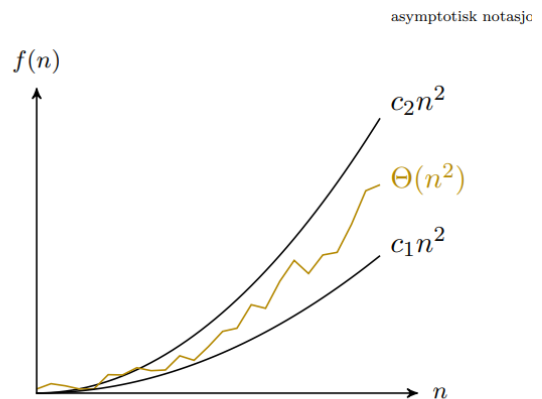
Reell eller nøyaktig kjøretid kan være varierende eller udefinert. Vi ønsker dermed å finne en form som likner kurven. Vi bruker tre forskjellige målinger i asymptotisk notasjon. Det er viktig å vite at det finnes ingen direkte sammenheng mellom $O(n)$, $\Theta(n)$, og $\Omega(n)$ og best/worst/average case kompleksitet for en algoritme. Asymptotisk notasjon er bare en måte å definere best/worst/average case.

- $O(n)$ eller big-o definerer den øvre grensa. Dette betyr at funksjonen kan være opptil n i kompleksitet, men også veldig mye mindre.

- $\Theta(n)$ eller stor theta definerer "tightest bound". Dette betyr at funksjonen vil være tilnærmet n . Det impliserer også $O(n) = \Omega(n) \implies \Theta(n)$
- $\Omega(n)$ eller stor omega definerer den nedre grensa. Dette betyr at funksjonen har ihvertfall n i kompleksitet, men det er ingen øvre grense som betyr at den kan være uendelig kompleks.

I tillegg har vi $o(n)$ og $\omega(n)$ som er øvre, men ikke lik, og nedre, men ikke lik grenser.

Når en skal regne på asymptotisk notasjon hjelper det å tenke på at notasjonen $O(n)$ definerer mengden av alle funksjonene med kompleksitet n eller mindre. Det resulterer ofte i at $\Omega(n)$ dominerer et uttrykk. Dessuten er $f(n) = O(n)$ notasjon misbruk da $O(n)$ er en mengde.



Ligger mellom to skaleringer av samme kurve, for store n