Algdat Oppsummering, eksamen-ting

Jim Frode Hoff

November 18, 2012

1 Definisjoner

1.1 Ordliste

Problem	En oppgave definert på generell input
Probleminstans	Et problem med gitt input
Iterasjon	En gjennomkjøring av en gjenntatt handling
Asymtpoisk notasjon	Notasjon hvor man kun tar vare på største ledd
$O(x)$ kjøretid	Øvre grense (maksimal kjøretid)
$\Omega(x)$ kjøretid	Nedre grense (minimal kjøretid)
$\Theta(x)$ kjøretid	Øvre og nedre grense for kjøretid
$T(x)$ kjøretid	Nøyaktig tid for en gitt probleminstans

1.2 Asymtopisk notasjon

En notasjon hvor man kun tar vare på det elementet som vokser kjappest. Det vil si, når man har en funksjon $f(n) = n^3/10 + 1000n^2 + 1\ 000\ 000n$ så vil man velge å beskrive den asymtopisk som en $g(n) = n^3$. Grunnen til dette er fordi at når n går mot uendelig så vil $n^2/10$ overstige både $1000n^2$ og $1\ 000\ 000n$ på et tidspunkt, og etter det fortsette med å øke eksponensielt.

1.3 Stor 'O'-notasjon

år kjøretiden for en algoritme er gitt ved $O(n)$ betyr det at dette er den (asymtopiske) verste kjøretiden for algoritmen, dvs med verst tenkelig input. Et godt eksempel på dette er Quick-sort, som har en kjøretid på $O(n^2)$ i verste tilfelle, men fortsatt har en kjørtid på $\Theta(n\ log\ n)$

1.4 Theta-notasjon

Kjøretiden for alle algoritmer kan representeres som en funksjon av input. La oss kalle denne funksjonen $f(n)$.
Det spesielle med $\Theta(x)$ er at den ligger over $c_1 f(n)$, men under $c_2 f(n)$ når n er større en en konstant n_0. Det vil si at man har funnet en funksjon som ikke vil være raskere, eller tregere enn grunnfunksjonen over en gitt n_0

2 Rekursjon

Rekursjon går ut på å bruke samme funksjon flere ganger, med et subsett av opprinnelig input. De fleste algoritmer kan implementeres enten iterativt (et funksjonskalll som løser problemet), eller rekursivt. Et godt eksempel på dette er fakultet.
La oss si at du skal lage en funksjon $Fak(X)$, hvor målet er å finne X-fakultet. Det kan løses rekursivt med følgende pseudo-kode:

function FAK(x)
  if $x \leq 1$ **then**
    return 1
  else
    return $X * Fak(X - 1)$
  end if
end function

Her ser vi at funksjonen kalletr seg selv med et subsett av input $(X - 1)$, og har en veldig viktig end-case. Alle rekursive algoritmer trenger en end-case for å avslutte loop'en med kall.

3 Enkel kombinatorikk

Gitt en komplett graf G med V noder, hvor mange kanter har G?
Svar: En komplett graf betyr at hver node har en kant til hver av de andre nodene. Hver node har derfor V-1 kanter som går ut fra seg selv. Men vi vil ikke telle samme kant flere ganger, så det blir (V-1) + (V-2) + (V-3) ... + 2 + 1.
Forenklet blir dette $\sum_{i=1}^{V}(V-I)$, eller enda enklere: $\underline{\underline{\frac{V(V-1)}{2}}}$

4 Masterteoremet

Motivasjonen til å lære seg masterteoremet er for å enkelt kunne løse rekunrenser. Ta for eksempel analysen av Merge-sort. Hvis vi antar at den orginale listen inneholderen 2'er-potens antall elementer vil vi for hvert kall til Merge-Sort få 2 nye metodekall med halve listen i hver, helt til antall elementer i lista er lik 1. Det vil si at $T(n) = 2T(n/2)$ så lenge $|n| > 1$

Hovedregelen:
$T(N) = aT(n/b) + f(n)$

La $A = aT(n/b)$ og $B = f(n)$, for så å gjøre om begge til asymtopisk notasjon.
Case 1: $A < B \rightarrow T(n) = O(n^{log_b a - c})$, som for en $c > 0 = \Theta(n^{log_b a})$
Case 2: $A = B \rightarrow T(n) = \Theta(n^{log_b a}\ lg\ n)$.
Case 3: $A > B \rightarrow T(n) = \Omega(n^{log_b a + c})$, som for en $c < 1 = \Theta(f(n))$

Grunnen til at jeg bruker A og B er fordi jeg prøver å få ting til å høre sammen med programmering. På denne måten blir A og B variabler, mens $aT(n/b)$ og $f(n)$ blir metodekall.

5 Pensum-problemer (2010)

Problem	**Algoritmer**	**Kjøretid**
Sortering	Selectionsort	$\Theta(n^2)$
	Insertionsort	$O(n^2), \Omega(n)$
	Mergesort	$\Theta(n\ log\ n)$
	Heapsort	$\Theta(n\ log\ n)$
	Quicksort	$\Theta(n\ log\ n), O(n^2)$
	Countingsort	$\Theta(n), O(n+k)$
	Radixsort	$\Theta(n), O(nk)$
	Bucketsort	$\Theta(n), O(n^2)$
Søking	Binærsøk	$\Theta(log\ n)$
	Bredde-først-søk	$O(V), \Omega(1)$
	Dybde-først-søk	$O(V), \Omega(1)$
Minimalt spenntre	Prims algoritme	$O(E\ log\ V)$
	Kruskals algoritme	$O(E\ log\ E)$
Korteste vei (en-til-alle)	DAG-shortest path	$\Theta(V)$
	Dijkstras	$\Theta(V + E\ log\ E)$
	Bellman Ford	$\Theta(VE)$
Korteste vei (alle-til-alle)	Floyd-Warshall	$\Theta(V^3)$

5.1 Sorterings-problemet

Man ønsker å ut fra en input $A = [a_1, a_2, a_3 ... a_{k-1}, a_k]$ kunne omrokkere elementene slik at man får en ny liste $B = [a_1 \leq a_2 \leq a_3 \leq .. \leq a_{k-1} \leq a_k]$ på en så effektiv måte som mulig. Når man ikke kan gjøre noen antagelser om input er det ikke realistisk å få til en bedre kjøretid enn $O(n\ log\ n)$.

6 Datastrukturer

Kø - Som en fysisk kø: FIFO → First In, First Out
Stack - Som en stabel: man legger ting oppå hverandre, og tar ut det øverste elementet i stabelen. LIFO → Last In, First Out

6.1 Heap

Heaps er den datastrukturen flest har problem med å forstå, samtidig som at det kanskje er den mest nyttige. Styrken til en heap, i forhold til en liste, er at den er utrolig rask på å finne det største/minste elementet. Skal man feks be en liste om å finne det minste elementet må man iterere gjennom hele listen, og derfor få en kjøretid på $\Theta(n)$. En heap klarer denne oppgaven med kjøretid $O(n\ log\ n)$.

Måten en heap fungerer på er nesten litt "magisk". Ved å alltid la hver node i et binærtre ha en egenskap, så får man sutomagisk en heap. denne egenskapen er: elementene som er direkte under noden har lavere verdi enn noden. Så hvis

7 Sortering

Sorteringsproblemet er definert slik:
Man skal ut fra en liste A=$[a_1, a_2, a_3, ...a_k]$ kunne modifisere listen så den oppfyller kravet A=$[a_1 \leq a_2 \leq a_3 \leq ... \leq a_k]$

Eks: [10, 2, 8, 4] → [2, 4, 8, 10]

7.1 Selectionsort

Denne metoden er en av de enkleste metodene å sortere på, og en av de treigeste.
For hver iterasjon finner man det største usorterte elementent, og bytter ut med det siste usorterte elementet.

Eks: [10*, 2, 8, 4] → [4, 2, 8*, 10] → [4*, 2, 8, 10] → [2*, 4, 8, 10]
* = største usorterte element på tidspunktet

Denne algoritmen er veldig treg. Det vil ta n operasjoner for å finne det største elementet, og siden dette gjøres n ganger (ved hver iterasjon). Dermed får man $O(n^2)$ kjøretid.

7.2 Insertionsort

Her går vi frem på en forholdsvis lik metode. Vi tar det første usorterte elementet x, og sammenligner det med de sorterte elementene. Hvis man når et element som er mindre enn x stopper man gjennomgangen, og tar for seg neste x.

Eks: [7*, 2, 8, 4, 3] → [7, 2*, 8, 4, 3] → [2, 7, 8*, 4, 3] → [2, 7, 8, 4*, 3] → [2, 4, 7, 8, 3*] → [2, 3, 4, 7, 8]
* = *Første usorterte element på tidspunktet*

Dette fungerer på samme måte som at man plasserer et kort på riktig sted i en kortspill-hånd.

Denne algoritmen har i likhet med Selectionsort en worst-case kjøretid på $O(n^2)$, men har en bedre best-case. Hvis listen allerede er sortert vil vi faktisk få en kjøretid på $T(n)$.

7.3 Mergesort

Vi har allerede fått en mye bedre best-case kjøretid på problemet, men vi vil gjerne få ned worst-case også. Her kommer Mergesort inn som en letytforståelig metode.
Mergesort deler listen i to helt til den har en stor samling av lister med 1 element. Dermed har man jo en samling av sorterte lister. Hurra!

Hva gjør man så når man har masse sorterte lister? jo, man slår dem sammen. Dette gjøres ved å sammenligne det første elementet i 2 lister, og legge den minste fremst i en ny liste. Så fjerner man dette elementet i den opprinnelige listen, og sammenligner igjen. Dette gjøres for hver liste rekursivt, som igjen gir oss en sortert liste med alle elementene. Hurra!

For å splitt listen helt ned til 1 element pr liste vil kreve N-1 oppdelinger, som asymtopisk blir $O(n)$. Sammenligningen vil kreve $log_2 n$ sammenslåinger, hvor hver sammenslåing vil kreve $n/2$ sammenligninger. Det gir oss $T(n-1) + T(log_2 n) * T(n/2) = T(n-1) + T((n/2)log_2 n) = \Theta(nlogn)$.

Dette er en KRAFTIG forbedring av worst-case. Det å sammenligne Mergesort og Selectionsort er som å sammenligne lineært søk og binærsøk.

7.4 Quicksort

Quicksort benytter seg av et pivot-element. Dette er et valgt element som blir sammenlignet med alle andre elementene, og deler opp listen i 2. Det er "det som er mindre eller lik pivot" og "Det som er større". Pivot-elementet ligger da i midten, og er sortert. Videre tar man hver av de to nye del-listene, og utfører samme prosedyre.

7.4.1 Randomisering

Med tradisjonell quicksort vil man velge det første eller siste elementet som pivot hver gang. Dette vil gi en worst-case på allerede sorterte lister. Måten å unngå dette er med randomisering. Da velger man enkelt nok pivot-element helt tilfeldig, og jobber med det.

7.4.2 Selection

8 Hashing

Direct-address tabell
Optimal når mengden av mulige nøkler (m) er forholdsvis lav. Implementeres ved å lage en tabell med m felter, hvor hver slot representerer hver nøkkel.

Hash tabell
En mer minnebesparende måte å lage en oppslagstabell på. Her lager man en mindre tabell, hvor hvert felt har en egen nøkkel. For at alle nøklene skal kunne få tilgang blir nøklene kjørt gjennom en *hash-funksjon*før den ferdige hash'en slår opp i tabellen. Svakheten med dette er at kollisjoner kan opstå. Dvs at flere aktuelle nøkler kan gi samme hash-verdi.
Når hashtabellen nærmer seg å bli full vil søketiden for å finne ledig plass bli større
Hashtabell - Tabell som lagrer data, basert på nøkler
Hashfunksjon - Funksjon som tar inn data, og genererer nøkler.

9 Ymse usortert

9.1 Binære søketrær

Et binært søketre er, som navnet tilsier, et binært tre. Det har egenskapen at på et gitt deltre, så vil alt på ventresiden være mindre eller lik rotnoden, og alt på høyresiden vil være større. Denne egenskapen gjelder også for alle deltrær i det binære søketreet.

Når man skal bygge et binært søketre, så tar man sitt første tall, og setter denne som rotnode. Når man setter inn flere tall så søker man seg frem til posisjonen tallet skulle ha vært på, og setter det der. Dette kan føre til veldig ubalanserte trær, men det er slik man gjør det (ifølge Åsmund Eldhuset).

9.2 NPC

NP - Nondeterministic Plynomial
Decision problem
Ja-svar skal kunne verifiseres i polynomisk tid
P
Decision problem
Kan løses i polynomisk tid
P er et subset av NP

NPC
Vanskeligste problemene i NP
NPC er et subset av NP.

Hvis et problem $A \in NPC$, så stemmer også $A \in NP$. Greia er nemlig at NP består kun av P og NPC. Ting som ikke er i P eller NPC er heller ikke i NP.
NP-Hard er problemer som kan beskrives som like vanskelige som NPC. Disse er ikke et subset av NP, og et ja-svar trengs ikke å kunne verifiseres i polynomisk tid.

Liste over NPC-problemer:
SATISFIABILTY
CLIQUE

VERTEX COVER
HAM-CYCLE
TRAVELLING SALESMAN (TSP)
SUBSET-SUM

9.3 Multithreading

Relativt nytt pensum, som baserer seg på flerkjerne-maskiner.
T_p = tid for kjøring med p prosessorer.
$T_p \geq \frac{T_1}{p}$ (Work-law)

Det er viktig å holde koden sin fri for "Raise-conditions"

9.4 Dynamisk programmering

Rod-cutting
tanken er at man har en stav av et eller annet metall, firmaet ditt selger metallstykker med flere lengder. Det er ikke nødvendigvis staver av alle lengder, så feks lengde 2 trenger ikke å eksistere.
Rekursivt løse et problem ved å teste alle utfall av problemet.
krav:
Optimal substruktur
Uavhengige delproblemer

9.4.1 Memoisering

Ved hjelp av en såkalt "DP-tabell" kan man lagre kjente utfall. Gitt følgende pseudo-kode:

function FIBONACCI(n)
  if $n \leq 1$ **then**
    return 1
  else
    $a = Fibonacci(n-1)$
    $b = Fibonacci(n-2)$
    return $a + b$
  end if

end function

og input 5, så vil man etter hvert lære seg hva forskjellige fibonacci-tall er. Man vil da gjerne lagre kjente resultat, siden dette vil spare maskinen for mye utregning, og da heller slå opp i minnet. Eksempelvis som denne pseudokoden:

```
dict ← hashmap
function FibMemo(n)
    if n in dict then
        return dict[n]
    else if n ≤ 1 then
        dict[n] = 1
    else
        dict[n] = FibMemo(n − 1) + FibMemo(n − 2)
        return a + b
    end if
end function
```

10 Kilder

Litt fra foiler, litt fra Kormen, men for det meste fra meg selv. Yupp, meg selv. Dette dokumentet er skrevet mest for min egen del, for å repetere det som er pensum høst 2011. Det er fullt mulig at noe av det som står her er feil. For det beklager jeg, og kan egentlig bare trøste deg med at jeg i så fall sitter med feil informasjon også.

Lesing og gjenngivning på eget ansvar.