Algdat Eksamensforelesning 2018

Sander Lindberg

November 2018

Innhold

1 Hvordan gå frem for å lese

Handler mye om forståelse, ikke like eksamener hvert år. Les boken (!!!) gjør eksamensoppgaver

2 Kjøretider

Hvor lang tid en algoritme bruker. Ligger en graf over en annen, har den lengre kjøretid. Bryr seg ikke om konstanter når **N** blir stor. Dette er fordi en konstant ganget med f.eks O(n$\hat{2}$) gjør ikke såå mye forskjell.

2.1 O-notasjon

$f(n) = O(g(n))\ f(n) \leq c * g(n)\ x > n_0$

2.2 Θ-notasjon

$f(n) = \Theta(g(n))$

2.3

3 Rekurrenser

Løs rekurrensen $T(n) = \frac{T(2n)}{2} - n, T(1) = 1$ **oppgi svaret med asymptotisk notasjon**

$$T(n) = \frac{2n}{2} - nT(n) + n = \frac{T(2n)}{2} 2T(n) + 2n = T(2n) = T(m) = 2T(\frac{m}{2}) + m$$

$M = 2n \implies \Theta(nlogn)$

3.1 Masterteoremet

La a ≥ 1 og b > 1 være konstanter. La $f(n)$ være en funksjon og la $T(n)$ være definert av a og b

Har en rekurrens $T(n) = aT(\frac{n}{b} + f(n))$

1. Hvis $f(n) \in O(n^{log_b(a-\epsilon)})$ er $T(n) \in \Theta(n^{log_b(a))})$
2. Hvis $f(n) \in \Theta(n^{log_b(a)})$ er $T(n) \in \Theta(n^{log_b(a)} * log(n))$
3. Hvis $f(n) \in \Omega(n^{log_b(a+\epsilon)})$ er $T(n) = \Theta(f(n))$

3.2 Variabelskifte

Har rekurrensen $T(n) = 2T(\sqrt{n}) + log(n)$. Kan vi bruke masterteoremet? Nei. Prøv å skrive om inputen slik at vi kan bruke masterteoremet.

$M = log_2(n)$

$n = 2^m$ (logaritmeregler)

$\sqrt{n} = n^{\frac{1}{2}} = 2^{\frac{m}{2}}$

$T(n) = 2T(2^{\frac{m}{2}}+) + m$

Bruker variabelskifte

$S(m) = T(2^m)$

$S(\frac{m}{2}) = T(2^{\frac{m}{2}})$

$T(n) = 2S(\frac{m}{2}) + m$

Nå kan vi bruke masterteoremet.

$m^{log_2(2)} = m \implies$ case 2 i masterteoremet

$T(n) = \Theta(m * log(m)) = \Theta(log(n) * log(log(n)))$

4 Sorteringer

Et par tips:

- Må kunne kjøretidene til alle, best case, averge case og worst case.
- Veldig bra videoer på Youtube av algoritmene.

Egenskaper til sorteringsalgoritmer:

- Sammenligning/ikke sammenligning (sammenligning kan ikke bedre enn nlogn)
- Best/average/worst
- Minnebruk - in-place?
- Stabil -> Flere like elementer, holder på plassen sin. Bytter ikke om 2 og 2 f.eks
- Parallelliserbarhet (Kan kjøre den på flere kjærner)

Sammenligningsbasert:

- Bubble
- Insertion

- Selection

4.1 Heapsort

Er et tre. Max-heapify minste elementer nederst og største høyest. Lar det største elementet bytte plass med nest største

Kjøretid: nLogN

4.2 Bubble sort

Best : $\Theta(n)$
Average : $\Theta(n^2)$
Worst : $\Theta(n^2)$

4.3 Insertion sort

Best : $\Theta(n)$
Average : $\Theta(n^2)$
Worst : $\Theta(n^2)$
Minne : $\Theta(1)$
In place : Ja
Stabil : Ja
Parallelliserbar : Nei

4.4 Selection sort

Best : $\Theta(n^2)$
Average : $\Theta(n^2)$
Worst : $\Theta(n^2)$
Minne : $\Theta(1)$
In place : Ja
Stabil : Nei
Parallelliserbar : Nei

4.5 Quicksort

Best : $\Theta(n * log(n))$

Average : $\Theta(n * log(n))$

Worst : $\Theta(n^2)$

Minne : $\Theta(log(n))$

In place : Ja

Stabil : Nei

Parallelliserbar : Ja

Velger et element i listen. Alle element større enn dette blir flyttet over på høyre side. Gjør dette helt til den er sortert.

4.6 Merge sort

Best : $\Theta(n * log(n))$

Average : $\Theta(n * log(n))$

Worst : $\Theta(n * log(n))$

Minne : $\Theta(n)$

In place : Nei

Stabil : Ja

Parallelliserbar : Ja

Deler seg i to helt til det er mange lister med et element. Merger de sammen i riktig rekkefølge (Rekursivt)

4.7 Counting sort

Best : $\Theta(n + k)$

Average : $\Theta(n + k)$

Worst : $\Theta(n + k)$

Minne : $\Theta(n + k)$

In place : Nei

Stabil : Ja

Parallelliserbar : Nei

(K er det største elementet i listen) Lager en ny liste med hvor mange vi har av hvert element. Altså har du 5 0'ere vil index 0 være 5 i den nye listen. Legger sammen indexene. Altså har du 4 1'ere, vil index 1 i den nye listen være 9.

Da vet du at du skal først ha 5 0'ere og deretter fra index 9-5 skal du ha 1'ere til index 9.

4.8 Radix sort

Best : $\Theta(d(n+k))$

Average : $\Theta(d(n+k))$

Worst : $\Theta(d(n+k))$

Minne : $\Theta(n+k)$

In place : Nei

Stabil : Ja

Parallelliserbar : Nei

Bruker andre algoritmer. F.eks bucket sort.

4.9 Bucket sort

Best : $\Theta(n)$

Average : $\Theta(n)$

Worst : $\Theta(n^2)$

Minne : $\Theta(n)$

In place : Nei

Stabil : Ja

Parallelliserbar : Nei

Lager bøtter". F.eks en bøtte 0 - 0.1, andre 0.1-0.2 osv.. Så du får bøtter med intervaller av elementer du skal sorterte.

4.10 Hvor fort kan vi gå?

Sammenligninsbasert - Kan ikke kjøre raskere enn n*log(n)

4.11 Eksamensoppgaver

H2011, oppgave 2c

> En venn av deg påstår han har utviklet en generell prioriteringskø der operasjonen for å legge til et element, å finne maksimum og å ta ut maksimum alle har kjøretid O(1) i verste tilfelle. Forklar hvorfor dette ikke kan stemme

Må være en sammenligningsalgoritme i grunn. (De er generelle). Sammenligningsbaserte algoritmer kan ikke kjøre raskere enn n*log(n) og dermed funker ikke O(1)

H2010, Oppgave 1g

> Heapsort er optimal, men Radix-sort har bedre asymptotisk kjøretid. Forklar hvordan dette henger sammen

H2014, oppgave 3a

> Du ønsker å sorterte en sekvens. Det er velkjent at sammenligningsbaserte algortimer har maks n*log(n). Anta elementene er reelle tall, distribuert etter en gitt sannsynlighetsfordeling som kan beregnes i konstat tid for ethvert element. Hva er den beste forventede kjøretiden du kan få og hvordan kan du oppnå den?

Kan oppnå lineær tid ved bruk av bucket sort. (Den eneste som har sannsynlighetsfordeling)

> Anta nå elementene er positive heltall og a_i < p(n) der p er et polynom. Hva er den beste forventede kjøresiden du kan få, og hvordan?

Veldig lett å tenke counting sort, men dette er feil, da counting sort har O(n+k). Setter du dette inn i p = n$\hat{3}$ får du (n+k)$\hat{3}$ som er dårlig. Svaret er: Radix sort, fordi du får linær tid ganger antall siffer.

5 Kompeksitet/NP

5.1 NP

Alle problemer som ikke kan løses i polynomisk tid, men kan verifiseres i polynomisk tid.

5.2 $P \subseteq Np$

5.3 NPC

Må være i NP Må være minst like vanskelig som alle andre problemer i NP

(Np hard hvis 2 er oppfylt)

5.4 Reduksjon

Tar et problem X og omformulerer det til problemet Y Har vi et NPC problem til vårt, har vi vist at vårt er minst like vanskelig som NPC Reduser fra venstre til høyre. Sett opp:

$NPC \leq NP$ f.eks

$A -> B = A \leq B$

H2008, oppgave 4f

> Du har problemene A,B,C. Alle er NP. Du vet A er i P og B i NPC.

For å bevise at C er i P må C reduseres til A i polynomisk tid

For å bevise at C er i NPC må B reduseres til C i polynomisk tid

Hvis B kan reduseres til A i polynomisk tid, så er P = NP

5.5 Hvordan få maks poeng uten å løse oppgaven

1. Tenk, hvilket problem minner om problemet?
2. Sett opp ulikheten. (Hvordan kan vi vise at dette er NPC/NP hardt/NP/P, (NP-H <= U))
3. Skriv opp at du vil redusere problemet du har funnet.
4. ?????
5. Profit

6 Datastrukturer

6.1 Stack

Sist inn - først ut

Push() O(1), Pop() O(1), Peek().

For å hente ut element midt i stacken -> Må ta ut alle over først. Tar altså O(n) tid.

6.2 Queue

Push() O(1), Peek(), Pop() O(1) (O(n) om du skal ha et element et annet sted i stacken). Sist inn sist ut. (Først inn, først ut FIFO)

6.3 Double ended queue

Kan pushe og poppe på begge ender av køen.

6.4 Linked list

Et element peker på det neste elementet. Fordelen - Trenger ikke sette av veldig lang plass i minnet, som med array. Kan ikke ta ut tind av linked list med indexer. Hente ut elementer tar O(n) tid. Må begynne fra starten og søke nedover. O(1) tid for å legge til elementer. Da du bare legger det først. Sette inn midt i listen O(n) tid. Å slette et element - begynne på starten, sette pekeren på elementet før det du skal slette til den etter det du skal slette. Tar O(n) tid.

6.5 Double linked list

Samme som linked list, bare at den har pekere frem og tilbake (også start og slutt).

6.6 Trær

Binærtre -> Hver node har maks to barn. Høyden på treet er log(n). Flytte et element nedenfra oppover tar O(log(n)) tid. Samme med å slette.

7 Hashing

Er en måte å indexsere elementer på, ved å komprimere lagringsplassen. Si du skal lagre masse navn. På maskinen 0'ere og 1'ere. Har et univers av nøkler, vi vil bare ha 4 av de når vi skal lagre 4 navn. Har en hashfunksjon som finner en hash. Lagrer navnet på hashen vi finner.

Tar ca O(1) tid å slå opp i hastabell

7.1 Hash chaining

Hvis to navn får samme hash f.eks 2, skal begge legges på samme sted i hashtabellen. Hvis vi vil ta vare på begge to, må vi kanskje gjøre tabellen større, eller vi kan bruke hash chaining. Lager en lenket liste. Altså lagrer en lenket liste på hashen (?). Har du en god hashfunksjon, vil disse listene bli kortere.

H2014, Oppgave 2d

> Hva karakteriserer en god hashfunkjson?

Fører til få kollisjoner, så listene blir korte. Uniform fordeling. Hasher like sannsynlig til punktene. Funksjonen må være deterministisk.

8 Dynamisk programmering

Tankesett som må øves på. Gjelder å løse oppgaver med dynamisk programmering. Er mange algoritmer i pensum. Mange nevnes ikke at er dynamiske.

For å kunne bruke dynamisk programmering på problemet ha:

- Optimal substruktur
- Overlappende subproblemer
- Subproblemene må være uavhengige

Gjør dynamisk programmering med Memoisering. Hvor må jeg begynne for å bygge meg oppover til en løsning?

Typiske dynamiske algoritmer:

- Rod cutting
- Fibonacci

Naiv (dum) løsning: Lett, men eksponentiell tid
Memoisering: Linær tid
Bottom up: Linær tid

8.1 Fibonacci

Mange funksjonskall som er det samme. F.eks fib(5), regner ut fib(3) flere ganger, fib(2) flere ganger. Dette kalles delproblemer.

Uten memoisering:

```
def fib(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n-1)+fib(n-i)
```

For å forbedre: Lagre delproblemene (lagre fib(2) og fib(3) f.eks)

Med memoisering:

```
memo = {}
def fib(n):
    if n in memo:
        return memo[n]

    f = 0
    if n == 0:
        f = 0
    elif n == 1:
        f = 1
    else:
        f = fib[n-1] + f[n-2]
    memo[n] = f

    return f
```

8.2 Rod cutting

Har en tømmerstokk som du skal kutte opp og tjene mest penger. Får betalt for lengde på tømmerstokken. Har overlappenge delproblemer.

$r_n = max(p_i + r_{n-i}), 1 \leq i \leq n$

H2010, oppgave 1h

Hvorfor er det ikke alltid nyttig å bruke memoisering i rekursive algoritmer?

Delproblemene overlapper ikke alltid.

8.3 Hvorgan gå frem?

1. Beskriv strukturen til en optimal løsning, tenk på hvilke valg som må gjøres
2. Definer rekursivt verdien til en optimal løsning
3. Regn ut verdien til en optimal løsning
4. Bygg opp en optimal løsning basert på beregnet informasjon

H2012, oppgave 1f

> I ryggsekkproblemet, la c[i, w] være optimal verdi for de i første objektene, med en kapasitet på w. La v_i og w_i være henholdsvis verdien og vekten til objekt i. Fyll ur rekurrensen for c[i, w], hvis i antar at i > 0 og w_i <= w

c[i, w] = max{v_i+c[i-1, w-w_i], c[i-1, w]}

8.4 Grådig programmering

Krav:

1. Optimal substruktur

"Det lokalt beste valget vil være det globalt beste valget". Altså, gjør det som virker best her og nå.

0-1 knapsack = Dynamisk Fractional knapsack = Grådig

Fractional: Ta med så mye du kan av det beste. Har du for mye "gull", fyll opp resten med "sølv".

8.5 Huffman codes

Hvordan komprimere data/tekst så mye som mulig?

Vi har en tekst med 100000 bokstaver; a,b,c,d,e,f

	a	b	c	d	e	f
Frequency (thousands)	45	13	12	16	9	5
Fixed-length codeword	000	001	010	011	100	101
Variable-length codeword	0	101	100	111	1101	1100

La oss bruke prefix kode.

Krav til prefix koder:

1. Ingen kode er en prefix til en annen kode
2. Hva med denne koden?
    - a = 0, B = 101, c = 100
    - Dette er en gyldig prefix kode!

8.6 Amortized analysis

Av og til kan vanlig kjøretidsanalyse være litt for pessimistisk... Hva om algoritmen som regel er kjapp, men av og til bruker den lengre tid på en operasjon? Vil det ikke gi mer mening å snakke om "gjennomsnittlig kjøretid"?

9 Graf-algoritmer

9.1 Binære søketrær

Struktur for å kjapt og greit kunne søke igjennom liste for å finne elementet du ønsker.

Hvordan funker det?

Barna til venstre er lavere enn forelderen. Alle til høyre er høyere.

Bygger et tre med "Insert". Legge høyere til høyre og lavere til venstre. (Starter alltid på toppen).

Er ikke nødvendigvis optimalt. Optimale kalles "Optimalt binært søketre".

Hvordan søke? Sjekke er tallet lavere eller høyere? Gå til venstre eller høyre osv, til du kommer til tallet.

	Average	Worst
Insert	O(lg(n))	O(n)
Search	O(lg(n))	O(n)
Remove	O(lg(n))	O(n)
Traversal	O(n)	O(n)

9.2 Representasjon av grafer

Nabomatriser og nabolister, G = (V, E)

9.3 Traversering

Gå igjennom grafer.

G = (V, E), V = Noder, E = kanter

9.3.1 DFS

LIFO (Last in first out). Går utover og følger til du ikke kommer lenger. Gå tilbake til den siste du hoppet over.

9.3.2 BFS

FIFO (First in first out). Oppdager alle barn til en node. Går til den første du fant, oppdager alle barn til denne. Går til den andre du fant på starten og finner alle barnene dens osv...

9.3.3 Kjøretider BFS, DFS

BFS	O(V+E)
DFS	O(V+E)

9.4 Eksamensoppgaver

H2012, oppgave 1d

> Tegn er eksempel på en urettet, sammenhengende grad med en markert startnode, der BFS vil finne korteste veier til de andre nodene, mens DFS garantert ikke vil det, samme hvilken rekkefølge naboene besøkes i. Bruk så få noder som mulig

Svar: Tre noder med kanter i mellom seg, En trekant.

H2011, oppgave 1f

> Hvilken algoritme vil du bruke for å finne korteste vei mellom to noder i en urettet, uvektet graf?

Svar: BFS

H2014, oppgave 4b

> Hvilket problem løser Ford-Fulkerson? Hvilke antagelser må eller kan du gjøre, og hvordan påvirker de kjøretiden?

Løser maks flyt og en må bruke en algoritme som BFS f.eks for å gjøre Ford-Fulkerson, som gir kjøre til på $O(VE\hat{2})$

H2011, oppgave 1a

Anta at nodene i en rettet graf kan ordnes fra venstre til høyre slik at alle kantene peker fra venstre mot høyre. Hva kalles en slik ordning?

Svar: Topologisk sortering

V2012, oppgave 1a

Anta du har en DAG G med n noder og m kanter. Hva er kjøreditden for å finne en topologisk sortering av G?

Svar: Fikk ikke me dmeg :c

H2010

Hvilke krav må stilles til en rettet graf for at den skal kunne sorteres topologisk?

Svar: Ingen sykler.

9.5 Maks flyt

Har en start og slutt (Source, drain (start og sluk)) . Det finnes veier mellom forskjellige noder, som har forskjellig vekt. Vekten betyr at veien har kapasitet på så så mye.

Har en graf med forskjellige veier og kapasitet. To paralelle kanter? -> legger til en node.

Begynner ved å lage residualnettverk. Deler opp så du har ”Resten” ut fra noden og det som ”står over streken” inn til noden. Reverserer flyten.

Flytforøkende sti - Finner en vei i resudialnettverket fra S til T som kan øke flyten. Gjentar til vi ikke kan øke mer.

Kalles Ford-Fulkersons metode (ikke algoritme). Kjøretid = Finne flytforøkende sti: $\Theta(V+E)AKA\Theta(E)$ Hvor mange ganger må vi finne flytforøkende sti= $O(|F^*|) = \Theta(E|F^*|)$

Edmond Karp kjøretid: $O(VE\hat{2})$

9.6 Topologisk sortering

Ordne nodene på en spesiell måte. Bortover langs en linje, slik at alle kantene går fra venstre til høyre.

9.6.1 DAG

Directed Acyclic Graph Ingen sykler, må kunne bli topologisk sotrtert. En topologisk sortering av en DAG G er en lineær ordning av nodene i G slik at hvis G inneholder en kant (u, v) kommer u før v i ordningen. (Cormen)

- Kjør DFS på alle noder

- Når en node er ferdigprossert legges den først i den topologiske ordningen.

9.7 Heap

En forelder er større enn sine barn. Hver node har 2 barn.

Max-heapify	O(lg(n))
Build-heap	O(n)
Extract-max	O(log(n))
Heapsort	O(nlog(n))
Increase-key	O(log(n))
Heap-insert	O(log(n))

9.7.1 Max-Heapify

Lager heap. Passer på at alle foreldre er større enn barna. Swap meg med det største barnet. Kall max-heapify på nytt. Hvis ikke, swap igjen osc...

9.7.2 Build-Heap

Har en tilfedlig liste. Kall max-heapify på halvparten. Halvparten fordi halvparten har barn. O(n)

9.7.3 Heapsort

Ta det øverste elementet og legg det sist i en liste. Swap størst med minst, kjør max-heapyify og gjenta. O(nlog(n))

9.7.4 Increase-key

Samme som max-heapify, bare gjør det andre vei. Sjekker om barnet er større enn forelderen i stedet for om forelderen er større enn barna.

9.8 Minimale spenntrær

Senntre - Et utvalg kanter i en urettet og sammenhengende graf som danner et tre på en slik måte at alle noder er med i treet.

Minimalt spenntre - Spenntrør som minimerer summen av kantene i et spenntre.

Gjør det grådig:

- Lokalt optimaliserte valg gir oss en globalt optimal løsning
- Legge til en og en ”trygg” kant

(En trygg kant er den minste kanten vi finner.)

9.9 Kruskal

- Sorter alle kantene etter stigende kantvekt
- Legg til kanter så lenge de ikke lager en sykel

Kjøretid: O(E log(V))

9.10 Prim

- Start med en tilfeldig node som startnode til treet T
- Legg hele tiden til den billigste kanten som utvider treet T med en ny node

Kjøretid:

O(E log(V)) -> med binary heap
O(E + Vlog(V)) -> med Fibonacci heap

9.11 Korteste vei - en til alle

Liker ikke negative kanter, ihvertfall ikke negative sykler. Liker heller ikke positive sykler. De ignorerer vi bare.

> Vi har en rettet vektet graf G = (V, E) og skal finne korteste vei fra S til T. En kortesete ei til node T er en sti fra T til S som minimerer summen av kantene langs stien.

9.11.1 Relaxtion

```
Initialize-single-source(G, s)
    for each vertex v in G-v
        v.d = inf
        v.pi = NIL
    s.d = 0


Relax(u,v,w)
    if v.d > u.d + w(u,w)
        v.d = u.d + w(u,v)
        v.pi = u
```

9.11.2 Dag-sortest Path

- Topologisk sorter grafen
- For hver node u i den topologiske ordningen
  - For hver nabo v av u
    - Relax(u, v)

Kjøretid:

Topologisk sortering	$\Theta(E+V)$
Initialisere nodeavstander	$\Theta(V)$
Relax	$\Theta(E)$
Totalt	$\Theta(E+V)$

9.11.3 Bellman-Ford

- Gjør |V| - 1 ganger:
  - For alle kanter (u,v)
    - Relax(u,v)

Kjøretid: $\Theta(VE)$

9.11.4 Dijkstra

- La Q være en min-prioriteringskø

- Legg alle noder i Q
- Så lenge Q ikke er tom
  - u = Q.pop()
  - for hver nabo v av u:
    - Relax(u,v)

Kjøretid:

Array: O(V$\hat{2}$)

Binary heap: O((V+E)*log(V))

Når vi pop'er u vil u.d være korteste vei.

Altså: Når vi velger å besøke en node u har vi allerede funnet korteste vei til u

Dijstra + negative kanter != sant

9.12 Korteste vei alle-alle

9.12.1 Floyd-Warshall

- Alle-Til-alle
- Nabomatriser

9.13 Sammenlikning

Algoritme	Kjøretid	OBS
DAG shortest path	O(V+E)	Kun på DAG
BFS	O(V+E)	Kun hvis alle kanter har samme vekt
Dijkstra	O((V+E)lgV)	kke negative kanter
Bellman Ford	O(VE)	
Floyd-warshall	O(V3)	
Johnsons	O(V2lgV+VE)	Bra for sparse graphs (E<