IDATT2101 ALGDAT

Kompleksitet

Asymptotisk notasjon

$4n^2 + 3n + 1 \approx n^2$ ved $n \to \infty$

Denne brukes til grenser for kjøretid:

O – øvre, Ω – nedre, Θ – begge/lik.

Kompleksitet i praksis

$O(n)$, $O(n^2)$ osv. oppnås ofte med nestede løkker som teller $0 \to n$. Med if-break inni loop kan vi få ulik O og Ω.

Rekursjon

Krav til rekursjon

1. En basis som kan løses trivielt.
2. Problem som kan deles opp i mindre biter til man når basistilfellet.

Rekursiv kompleksitet

$$T(n) = \begin{cases} 1 \text{ når } n = 1 \\ aT\left(\frac{n}{b}\right) + cn^k \text{ når } n > 1 \end{cases}$$

$b^k \blacksquare a$	$T(n) \in \Theta(\blacksquare)$
$<$	$n^{\log_b a}$
$=$	$n^k * \log n$
$>$	n^k

a: antall rekursive kall, b: brøkdel av datasett brukt i ett kall, cn^k: kompleksitet.

<u>NB</u>: cn^k regnes ut med vanlig metode.

Rekursjon vs. Iterativ løsning

Hvis man enkelt kan løse problemet iterativt bør man det med tanke på overhead.

Sortering

Beste mulige kompleksitet er $O(n)$ siden alle elementer må sjekkes minst en gang.

Innsettingssortering – $O(n^2), \Omega(n)$

Start med ett kort på hånda, og sett inn ett og ett kort på riktig sted.

+ Små og allerede sorterte datasett.

− Dårlig på store datasett.

Boblesortering – $\Theta(n^2)$

Gå gjennom tabellen $n - 1$ ganger og bytt plass på naboer i feil rekkefølge. Store tall synker, små tall bobler opp.

+ Veldig enkel å implementere.

− Lite brukt: innsetting er like enkel/bedre.

Velgesortering – $\Theta(n^2)$

Velger det største tallet og setter dette på riktig plass. Gjenta til man er ferdig.

+ Bytter plass på færre tall enn boble.

Kvadratiske sorteringsalgoritmer

Alle sorteringsalgoritmer som bytter naboer med hverandre, får kompleksitet $\Omega(n^2)$.

Shellsort – $O(n^2)$

Forbedring av innsetting. Starter med avstand $\frac{n}{2}$ og går ned til 1 (nabo).

+ Merkbart bedre enn innsetting.

− Fortsatt $O(n^2)$ som kompleksitet.

Flettesortering – $O(n * \log n)$

Deler tabellen i to rekursivt og fletter løsningene sammen til en.

+ Bedre enn kvadratiske algoritmer.

− Krever en ekstra tabell (plass).

Quicksort - $O(n \log n)$ i snitt pga. pivot

Deler data i to med en pivot-verdi. Alle tall større/mindre på hver sin side. Rekursiv.

+ Raskeste kjente for generelle data.

− Små datasett. Dårlig plassert pivot.

Dual Pivot Quicksort

Forbedring av quicksort som er rundt 10-20% raskere. Deler data i tre, i stedet for to, og oppnår færre sammenlikninger og rekursive kall.

Tellesortering – $\Theta(n + k)$

Teller opp forekomster og skriver riktig antall i stigende rekkefølge til en ny tabell.

+ Veldig effektiv når tallene ligger tett.

− Krever heltall i $0 \dots k$. Ekstra hjelpetabell.

Lister, kø og stack

Enkeltlenket liste

Noder med verdi og referanse til neste.

<u>Eksempel</u>: $A{:}\,1 \to B{:}\,7 \to C{:}\,2 \to D{:}\,4$

Dobbeltlenket liste

Samme som 1, men og referanse til forrige.

<u>Eksempel</u>: $A{:}\,1 \leftrightarrow B{:}\,7 \leftrightarrow C{:}\,2 \leftrightarrow D{:}\,4$

Kø (bruker tabell/enkeltlenket liste)

Uttak alltid foran, innsetting alltid bak.

<u>Innsetting</u>: $O(1)$. <u>Uttak</u>: $O_{tabell}(n)$, $O_{liste}(1)$

Stack (bruker tabell/enkeltlenket liste)

Uttak alltid foran, innsetting alltid foran.

<u>Innsetting</u>: $O(1)$. <u>Uttak</u>: $O(1)$.

Trær

En graf som har ingen sykler. Ikke-lineær datastruktur.

<u>Rot</u>: Noden som ikke har en forelder.

<u>Løv-node</u>: Noder som ikke har noen barn.

<u>Grad</u>: Antall barn en node har.

<u>Dybde</u>: Antall kanter fra noden til rota.

<u>Høyde</u>: Antall kanter fra laveste løv-node.

Binærtrær

Et tre som har maks to barn per forelder.

<u>Perfekt binærtre</u>: Alle noder har to barn, og løvnodene ligger på samme nivå.

<u>Komplett binærtre</u>: Perfekt, men kan mangle løvnoder nederst til høyre.

<u>Fullt binærtre</u>: Alle noder har 0/2 barn.

<u>Innsetting</u>: Sett inn som rot hvis en ikke eksisterer, ellers sammenlikn nøkkelverdier. Ny verdi er: mindre→venstre, større→høyre.

<u>Søking</u>: Samme søk som innsetting.

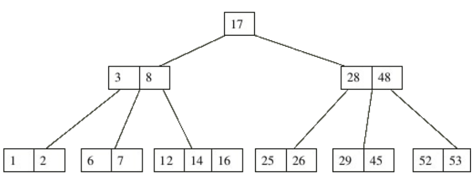
<u>Sletting</u>: Har noden ett barn flyttes denne opp. Har noden to barn erstatter vi noden med den minste noden i høyre subtre. Har noden ingen barn kan vi slette den fritt.

<u>Kompleksitet</u>: Alle handlingene $O(h)$.

B-trær

Datastruktur for effektiv bruk av disker. Mye brukt i databasesystemer. Et tre der man kan ha flere verdier i hver node.

Krav til b-tre av orden m: Noder har maks m barn, alle indre noder har minst $\left\lceil \frac{m}{2} \right\rceil$ barn. En indre node med k barn har minst $k - 1$ nøkler. Alle løvnoder er i samme dybde.



Heap og prioritetskø

Et komplett binærtre hvor hver node inneholder en nøkkelverdi. Delvis ordning gjør heap veldig rask til å hente max/min.

Heapegenskapen

I en max/min-heap har alle noder nøkkelverdi større/mindre eller lik begge barnenodenes. Rota er størst/minst.

Prioritetskø

Datastruktur der man plukker ut basert på et elements prioritet. Ikke når det ble til. Veldig raskt å plukke ut viktigste element.

Heap som tabell

$i_{foreldrenode} = \left\lfloor \frac{i_{node} - 1}{2} \right\rfloor$

$i_{venstre\ barn} = 2i_{node} + 1$

$i_{høyre\ barn} = 2i_{node} + 2$

Heapsort – $O(n \log n), \Omega(n)$

Lager en heap av usortert data og plukker ut det største/minste elementet. Fortsetter til alle elementene er plukket ut.

+ Like god som flettesort uten ekstra tabell.

Hashtabeller

Datastruktur hvor man kan bruke tekst, objekter eller uegnede tall som nøkkel.

<u>Lastfaktor</u>: Andel plass brukt i tabell. $\alpha = \frac{n}{m}$

m: Størrelse på tabell, n: Antall elementer.

Dårlig lastfaktor: $\alpha \to 0$, Full tabell: $\alpha \to 1$.

Hashfunksjon

En funksjon som tar ønsket nøkkel inn, og gir verdier i $0..m$ ut. Jevnt fordelt. Beregningen må skje i $O(1)$ tid.

<u>Hash med restdivisjon</u>: $h(k) = k \bmod m$

<u>Multiplikasjon</u>: $h(k) = \lfloor m(kA - \lfloor kA \rfloor) \rfloor$

Kollisjoner

To poster med ulik nøkkel kan få samme hash. Tabellposisjoner kan lagre ett element, så kollisjonen må håndteres.

Kollisjonshåndtering

<u>Lenkede lister</u>: Tabellelementene er hoder på lenkede lister som kan holde elementer.

<u>Åpen adressering</u>: Hvis ønsket plass er tatt, prob etter andre posisjoner. Tre måter:

<u>Lineær</u>: Gå 1 bort fram til ledig.

  + Neste ledige plass er enkel.

  − Lange kjeder med kollisjoner.

<u>Kvadratisk</u>: Gå $steg\#^2$ bort til ledig.

  + Naboverdier får ikke samme kjede.

  − Like verdier har alltid samme kjede.

<u>Dobbel hashing</u>: Steglengde er avhengig av en andre hashfunksjon.

  + Like verdier ikke alltid samme kjede.

  − Krever en ekstra hashfunksjon.

Uvektede grafer

<u>Spor</u>: En vei som ikke repeterer kanter.

<u>Sti</u>: En vei som ikke repeterer hjørner/noder.

<u>Selvløkke</u>: Kant som fra/til samme node.

<u>Rundtur</u>: Vei med start/slutt i samme node.

<u>Enkel vei</u>: Vei med kun unike noder.

Implementasjon

Naboliste: Tabell med lenkede lister der index er fra og elementer i lister er til.
Tabell: En $n \times n$ tabell med bits viser kanter.

Bredde-først-søk (BFS) – $O(N + K)$
Start i en node s og finn alle noder som kan nås fra denne og legg de i kø. Når alle direkte koblinger er funnet, gå videre til neste node i køen. Gjenta til ferdig.

Dybde-først-søk (DFS) – $O(N + K)$
Start i en node s. Gå videre til en nabo. Hvis vi kommer til en besøkt node, går vi tilbake til forrige node og prøver neste kant. Fortsett fram til alle noder er besøkt.
+ Går raskere ned i dybden enn BFS.
− Veier funnet med DFS er ikke nødvendigvis korteste vei.

Topologisk sortering – $\Theta(N + K)$
Sorterer elementer basert på hvilke elementer som må være foran andre. Kjører DFS på hver nummererte node og legger dem i resultatlista når alle kanter ut fra dem har blitt undersøkt ferdig.
Eksempel: Studieplanlegging/kompilering.
NB: Krever asyklisk graf for å virke.

Vektede grafer
Grafer som har data/vekt på kantene.
Eksempel: Kjøretid, avstand, hastighet.

Korteste vei-problemet
Ønsker å optimalisere for vekten, uavhengig av hva den representerer. Ofte avstand.

Dijkstras algoritme – $\Theta(N^2)$
Grådig algoritme for korteste vei. Start i node s og finn avstanden fra s til andre. Når alle kantene til s er utforsket, gå til noden med lavest avstandsestimat og gjenta.
− Ingen kanter kan ha negativ vekt.

Bellman Ford-algoritmen – $\Theta(NK)$
Likner på Dijkstra's algoritme.
+ Håndterer kanter med negativ vekt.

A* (A-star) – $\Theta(N^2)$
En forbedring av Dijkstra's algoritme som tar med estimert avstand til mål i prioritet.
+ Ellipseformet søk, innom færre noder.
− Ikke like bra til å finne alternative veier.

ALT – A*, landemerker, triangellikhet
En forbedring av A* som bruker Dijkstra's til å finne nøyaktige avstander fra landemerker til andre noder. Disse brukes til estimat.
+ Bedre estimater enn A* uten landmerker.
− Krever preprosessering av kartet og plass.

Minimale spenntrær
Spenntre: Et tre som kobler sammen alle nodene i en urettet vektet graf.
Kostnad: Summen av vekten til et spenntre.
Minimalt spenntre: Det spenntreet med lavest mulig kostnad. (Kan være flere.)
Bruksområde: Nettfiber, makking av vei.

Kruskals algoritme – $O(K \log \mathrm{K})$
Grådig algoritme for minimalt spenntre. Starter med alle nodene som individuelle trær, og kobler trærne med billigste mulige kant. Vi er ferdig når det gjenstår ett tre.

Prims algoritme – $O(K \log N)$
Start med node s og bygg spenntreet ved å koble til én og én node. Noder som ikke er besøkt prioriteres etter avstand fra treet. Vi er ferdig når alle nodene er tilkoblet.

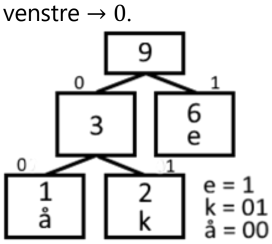
Maksimal flyt
Et flytnettverk er en praktisk bruk av grafer. Vi har en kilde og et sluk, og kanter med en maksimal flyt.
$|f|$: Total flyt i en graf, $f(n, m)$: Flyt i kant.

Ford Fulkerson-metoden – $O(NK^2)$
Vi setter flyten til alle kanter til 0. Deretter finner vi en vei som øker flyten til målet. Vi øker flyten med den laveste kapasiteten i veien. Gjenta til vi har maks flyt.
NB: Når vi øker flyten i en kant, senker vi flyten i motsatt kant med samme tall. Da vil algoritmen snu flyten om dette er nyttig.
+ Kan anvende både DFS og BFS.
− Kan ta lang tid ved verste tilfelle.

Edmond Karp-metoden
Nesten som Ford Fulkerson, men tar i bruk korteste vei og kun BFS.

Avanserte programmeringteknikker
Grådige algoritmer
Algoritmer som gjør det som er best i øyeblikket uten å tenke på helheten. Gjør aldri om på et valg når dette er tatt.
Korteste vei: Dijkstra's algoritme.
Minimale spenntrær: Prim's og Kruskal's
Tekstkomprimering: Huffman-koding
+ God løsning uten endring underveis.
− Ikke nødvendigvis beste løsning.

Tekstsøk og datakompresjon
Huffman-koding
Bygger et tre med frekvensen til tegn i en tekst. Mest brukt på toppen. Bytekoder finnes med treet. Gå til høyre → 1. Gå til venstre → 0.



Boyer-Moore-tekstsøk
Se på siste tegn i søketeksten, hvis den ikke passer flytter vi så langt vi kan. Hvis det passer, se på nest siste. m: Søkeordlengde.

	r	a	b	a	r	b	r	a
0	b	r	*a*					
1			b	r	*a*			
2				b	r	*a*		
3						**b**	**r**	**a**

Upassende tegn: Hvis et tegn ikke finnes i søketeksten kan vi flytte $m - t$ steg fram. t: Hvilket tegn i søketeksten som bommet.
Passende endelse: Hvor overlapper søkeordet med seg selv? Hvis -NE i ENE passet, vil det ikke være noe poeng et hopp.
Galil's regel: Når vi flytter søkeordet kortere enn vi sjekket forrige runde, er det ingen vits i å sjekke det overlappende området.

Lempel-Ziv-tekstkoding
Bruker en sirkulær buffer til å sjekke bakover i teksten vi allerede har sett. Når vi finner en bit som passer, erstatter vi denne med en referanse bakover i teksten.

Bitoperasjoner
Datamaskiner bruker binære tall, og bitoperasjoner er derfor veldig effektive.

Høyreskift
Flytter alle bits en plass til høyre. Dette blir det samme som $\left\lfloor \frac{x}{2} \right\rfloor$.
Notasjon: $a \gg= 1 \rightarrow a = a \gg 1$
Eksempel: $1101_2(13_{10}) \gg 1 = 110_2(6_{10})$

Venstreskift
Flytter alle bits en plass til venstre. Dette blir det samme som $x * 2$.
Notasjon: $a \ll= 1 \rightarrow a = a \ll 1$
Eksempel: $110_2(6_{10}) \ll 1 = 1100_2(13_{10})$

Bitwise XOR (^): 1001 ^ $1010 = 1100$
Bitwise OR (|): $1001 \mid 1010 = 0011$
Bitwise AND (&): $1001 \,\&\, 1100 = 1000$
Bitwise NOT (~): $1010 \rightarrow 0101$

NP-kompletthet
P: Problemet kan løses i polynomisk tid.
NP: Svaret kan sjekkes i polynomisk tid.
NPC: Vanskelige problem i NP. Alle NP-problemer kan omformes til NPC-problem.
NP-hard: Det finnes problem i NPC som kan reduseres til disse i polynomisk tid.

Problem	NP-komplett variant	NP-hard variant
Traveling salesman	Reiserute med kostnad under x	Billigste av alle mulige reiseruter
Ryggsekkproblemet	Få med varer for verdi V	Finn største verdi vi kan få med oss
Komplett subgraf	Finn en *komplett* subgraf med x noder	Finn *største komplette* subgraf
		Halting-problemet
	3SAT	
Delsum	Av n tall, finn x som summerer til 0	
Isomorfi	Er G_1 isomorf med en subgraf av G_2?	
Hamiltonsyklus	Finn en vei innom alle noder én gang	

Håndskrevne kommentarer