Tidskompleksitet:
T(n) = Polynomisk utrykk av antall løkker og enkle operasjoner. O øvre grense. Ω nedre grense. Θ øvre og nedre grense, om de er like. En løkke i en løkke hvor størrelse på løkkene er n blir $n^2$. Om disse løkkene er i en løkke med størrelse n blir det nå $n^3$. Om løkkene har forskjellig størrelse blir det n * m. Løkker som kommer sekvensielt blir n + m.
Rekursjon: I rekursjon bruker man masters theorem ved splitt og hersk.
$T(n) = a*T(n/b)+cn^k$ $a$: antall rekursive kall, $b$: brøkdelen av datasetter vi behandler i et rekursivt kall. $cn^k$: kompleksitet av metoden.
1. $b^k < a \rightarrow T(n) \in \Theta(n^{log_b a})$
2. $b^k = a \rightarrow T(n) \in \Theta(n^k * log(n))$
3. $b^k > a \rightarrow T(n) \in \Theta(n^k)$

```
1
2 void double_recursive(int d[], int l, int r)
3 {
4     int m = (l + r) >> 1;
5     return sum(d, l, m) + sum(d, m +1, r);
6 }
```

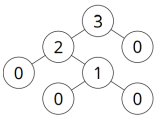
$T(n) = 2 * T(n/2) + c * n^0$ $T(n) = \Theta(n^{log_2 2}) = \Theta(n)$
Lineær rekursjon: Hvis rekursjonsvariabelen minker med $(n-1)$ (antar $a = 1$)får vi $T(n-1) + cn^k \in \Theta(n + cn^k)$
Datatypen liste: Definisjon: Kunne opprette, finne antall elementer, sette inn element, fjerne elementer, tømme lista, finnet et element ved indeks eller verdi, sekvensielt gå igjennom lista, finne max og min, sortere på et hensyn.
Liste:

| Operasjon | Usortert | Sortert |
|---|---|---|
| Lage | O(1) | O(1) |
| Finne lengde | O(1) | O(1) |
| Sette inn | O(1) | O(n) |
| Fjerne | O(1) | O(n) |
| Tømme | O(1) | O(1) |
| Finne på plass | O(1) | O(1) |
| Søke | O(n) | O(log n) |
| Traversere | O(n) | O(n) |
| Finne største | O(n) | O(1) |
| Sortere | O(n log n) | - |

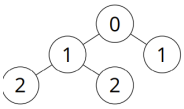
Dobbel- og enkel-lenket:

| Operasjon | Enkel | Dobbel med hale |
|---|---|---|
| Lage | O(1) | O(1) |
| Finne lengde | O(1) | O(1) |
| Sette inn | O(1)/O(n) | O(1) |
| Fjerne | O(n) | O(1) |
| Finne på plass | O(n) | O(n) |
| Søke | O(n) | O(n) |
| Traversere | O(n) | O(n) |
| Finne største | O(n) | O(n) |
| Sortere | $O(n^2)$ | O(n log n) |

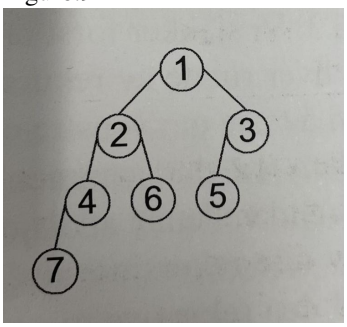
Datatypen Kø: : Kø er en datatype hvor ting blir tatt ut foran og lagt inn bak, FIFO. Bruker en liste. O(1) for innsetting og fjerning av elementer. Bruker $k = (k+1)\%max$, hvor k er start- eller slutt posisjon og max er kapasiteten. Dette gjør at det blir en sirkulær struktur og vi slipper å flytte alle elementene. Datatypen Stakk: : LIFO, last in first out. Hvis vi holder styr på hvor toppen av stakken er, blir det O(1) for å legge inn og hente ut data.
DFS: DFS = Depth-First Search. Velg en startnode og legg denne på stakken. Stakken består av noder som vi har tenkt å besøke. Ta ut noden fra stakken og marker den som besøkt. Vi skal nå behandle noden vi har tatt ut av stakken. I stakken skal du legge alle nodene som kan nåes fra noden som blir behandlet. Gjør dette til alle nodene er behandlet. Velg en random ubesøkt node hvis det ikke er flere noder som kan nås. BFS: BFS = Breath-First Search. Nøyaktig samme som DFS, men med kø i stedet for stakk. Trær: Et tre består av mange noder, hvor man har en rot. Barn er noder under en node, forelder er noden over en node. Noder med samme forelder er søskennoder. Binærtrær: Trær hvor hver node har max 2 barn. Et perfekt binært tre → alle løvnodene befinner seg på nederste nivå, alle indre noder har 2 barn. Komplett → perfekt tre, men det kan mangle noder lengst til høyre i nederste nivå. Fullt → alle indre noder har 2 barn, løvnoder må ikke være nederst. Høyde: Antall lenker mellom noden og den noden som er lengst unna i et av nodens subtrær. Treets høyde er definert som rotas høyde.



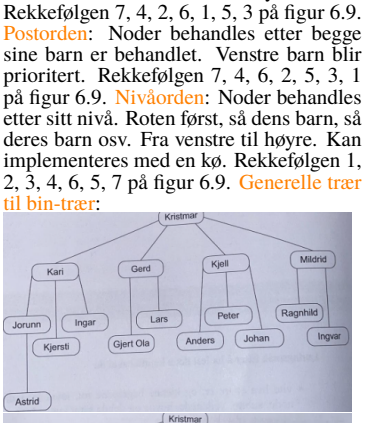

Dybde: Antall lenker mellom noden og rota. Rota har dybde 0.

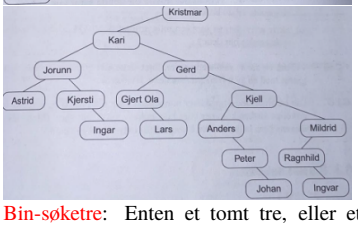



Figur6.9




Traversering: Preorden, inorden, postorden og nivåorden. Preorden: Enhver node behandles før sine barn. Dermed blir venstre noder behandlet i sin helhet først. Rekkefølgen 1, 2, 4, 7, 6, 3, 5 på figur 6.9. Inorden: Noder behandles mellom sine venstre og høyre barn. Roten blir da behandlet i midten om treet er symmetrisk. Rekkefølgen 7, 4, 2, 6, 1, 5, 3 på figur 6.9. Postorden: Noder behandles etter begge sine barn er behandlet. Venstre barn blir prioritert. Rekkefølgen 7, 4, 6, 2, 5, 3, 1 på figur 6.9. Nivåorden: Noder behandles etter sitt nivå. Roten først, så dens barn, så deres barn osv. Fra venstre til høyre. Kan implementeres med en kø. Rekkefølgen 1, 2, 3, 4, 6, 5, 7 på figur 6.9. Generelle trær til bin-trær:



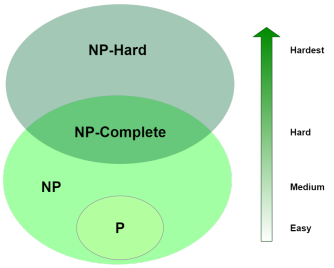

Bin-søketre: Enten et tomt tre, eller et binærtre der nodene har nøkler. Alle nøkkel verdiene til venstre må være mindre, alle nøkler til høyre må være større enn en node sin nøkkel verdi. En inordentraversering vil få ut nodene i sortert rekkefølge. Innsetting: Sammenligne nøkkelverdier til alle verdier i venstre subtre er mindre, og alle verdier i høyre subtre er større. Søk: Samme måte som insetting, tester også etter likhet. Kommer vi til en node uten barnet vi søker etter og ingen annen mulighet, vet vi at det ikke finnes. Sletting: 3 muligheter. Noden har ikke barn, fjern lenken til forelder noden. Har 1 barn, foreldre nodens lenke går nå til barnet sitt barn. Har 2 barn, erstatter nodens data, ikke lenker, med neste node i rekkefølgen (den minste i høyre subtre). Denne noden fjernes deretter fra sin plass. Kjøretid søk og insetting: Innsetting og søk er proporsjonalt med høyden. Kjøretid i verste tilfelle er O(n) hvor n er høyden fordi det blir en lenka liste. I avanserte varianter er treet perfekt balansert, kjøretid blir da $\Theta(logn)$, der n er antall noder. Eksempler er AVL-, rødsvarte- og splinetrær. Generelt tilfelle blir derfor mellom $O(logn)$ og $O(n)$. Noder i et perfekt tre blir alltid $n = 2^{h+1} - 1$ der h → høyde. Kjøretid sletting: Så samt man har noden man skal slette er sletting med 0 eller 1 barn $O(1)$. Har noden 2 barn er det $O(logn)$ dersom $h = logn$; Kjøretid annet: Dybde er worstcase nederste noden, $O(h) \approx O(logn)$. Høyde sjekker alle noder, $T(n) = 2T(n/2) + 1$. Master metode på dette blir $\Theta(n)$. Traversering går igjennom alle noder, dermed $\Theta(n)$.

B-trær: B-tre brukes til disker og databaser. Ikke binære, hundrevis av barn. Alle løvnoder er på samme nivå og alle noder, har mellom $n-1$ og $2n-1$ nøkler. Rota har fra $1 \rightarrow 2n-1$ nøkler. Hver node har 2 lister, en for barn og en for nøkler.

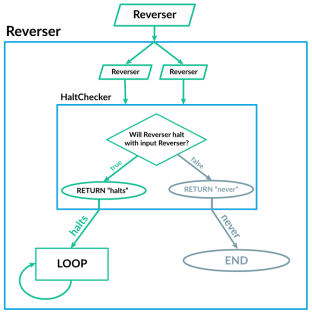
Heap: 2 typer heap, max og min. Komplett bin-tre med nøkkelverdier i nodene. Heap er en delvis ordnet datastruktur, post hoc ergo propter hoc; alt er ikke sortert riktig, men rota blir alltid minst eller størst. Heap brukes til sortering og prioritetskø. Sorteringsalgoritmer: Innsetting: $O(n^2)$ $\Omega(n)$ når tabellen er sortert. Enkel. Bra for små datamengder og tabeller som nesten er sortert (for eksempel slutten av en quicksort). I disse tilfeller kan innsettingssortering være raskere enn $O(nlogn)$-algoritmene. Boble-: $\Theta(n^2)$ Triviell. Velge-: $\Theta(n^2)$ Velge det største tallet og plasser det rett. Få skriveoperasjoner, kan være bra om skriving til minne er tregt/ikke ønskelig. Shell-: Mellom $O(n^{\frac{7}{6}}) - O(n^{\frac{5}{4}})$. Hvorfor er tidskompleksiteten så mongis? I shellsort har vi en $S$-verdi. Denne verdien er først $S = \frac{tabellstorrelse}{2}$. S bestemmer hvor store steg sammenlikningen i sorteringen er. Hvis $S$ er 1 vil det bli en helt lik algoritme som insettingssortering. Tidskompleksiteten avhenger veldig av hvordan $S$ senkes fra n/2 til 1. Hvis $S$ senkes med 2 for hver steg, da blir tidskompleksiteten $O(n^{\frac{3}{2}})$. Hvis $S$ er 2.2 som i kodeeksempelet på arket blir tidskompleksiteten mongis. Ingen har klart å beregne kompleksiteten for varianten av shellshort beskrevet i boka. Flette-: $O(nlog(n))$ fra mastermetoden. algo: 1. Del tabell i 2 deler. 2. Løs de 2 delene rekursivt med flettesortering. 3. en tabell med bare ett element er sortert. 4. flett de sorterte tabellene sammen. Quick-: $O(nlog(n))$ fra mastermetoden. Ansett som den generelt beste sorteringsalgoritmen. Splitt og hersk. algo: 1. Del tabellen i to mindre tabeller. Alle tall i den laveste delen skal ha lavere verdi enn alle tall i den høyeste delen (utføres i splitt() metoden). NB: deler ikke nødvendigvis på midten. median3sort velger delingsverdien, enten medianen (m), venstre endepunkt (v) eller høyre endepunkt (h). median3sort flytter m, v og h slik at de står riktig sortert i forhold til hverandre. 2. De to deltabellene sorteres hver for seg med rekursiv bruk av quicksort. 3. Stopper rekursjonen når vi har 3 elementer igjen. Da tar vi en median3sort. feller: 1. dårlig delingsverdi gir $n^2$ kjøretid (f.eks. hvis vi har en sortert tabell og velger det første elementet som delingsverdi). 2. for dyp rekursjon (fix: f.eks. med å stoppe før rekursjonen blir for dyp og sortere resten med f.eks. innettingsortering eller heapsort). 3. $n^2$ kjøretid ved sortering av duplikater. <,> i indre løkker, heller enn <= og >=. 4. Skjevdeling med 0 og n elementer, som gir uendelig løkke. Unngås med median3sort, som gir oss ihvertfall ett «lite» og ett «stort» tall. Unngås også av andre metoder som ikke sorterer delingstallet omigjen. For hånd: 1. Sammenlign venstre, midten og høyre, bytt om nødvendig. 2. Bytt pivot med element sist i delarray. 3. Venstre-item er første tall større enn pivot fra venstre. 4. Høyre-item er første tall mindre enn pivot fra høyre. 5. Mens venstre idx < høyre idx, bytt. 6. Bytt pivot og venstre item. 7. Del på hver side av pivot, ikke inkluder, og begynn på nytt. Telle-$O(n)$ Radiks-: $O(n)$ Meta: Kan ikke sortere bedre enn $O(nlogn)$ ved å sammenlikne elementer med hverandre. Bevist. Hashtabell:
Finne elementer i $\Theta(1)$ tid. $\alpha$ = lastfaktor = $n/m$. $m$ størrelse, $n$ elementer, $k$ er nøkkel. Hashfunksjoner: burde bruke en god hashfunksjon basert på hvilke nøkler vi kommer til å måtte hashe. Hvis ikke vet på forhånd kan vi bruke en hash som er bra på alt. En hashfunksjon bør avhenge av alle delene av nøklene. Restdivisjon: $h(k) = k \bmod m$. Funker best med m = primtall. Dårlig med m = tierpotens eller toerpotens, f.eks hvis $m = 256$ vil alle nøkler bare avhenge av de siste 8 bitsa ($2^8 = 256$).
Basic multiplikativ hash: $m$ kan være hva som helst. $0 < A < 1$. $A$ funker bra med $\frac{\sqrt{5}-1}{2}$. $h(k) = \lfloor m(kA - \lfloor kA \rfloor) \rfloor$ Rask multiplikativ hash: $m = 2^x$. $A = 1327217885$. $H(x,k,A) = (k * A >> (31 - x))$ , hvor k er antall bits vi ønsker i resultatet. Kollisjoner: Lenka lister: Innsetting: $\Theta(1)$. Uthenting $O(n)$ og $\Omega(1)$ (gjennomsnitt): $\Theta(1+\alpha)$ som blir $\Theta(1)$ så lenge vi ikke fyller hashtabellen med mer enn $\Theta(m)$ elementer. Åpen adressering: Lineær probing: $probe(h,i,m) = (h+i) \bmod m$. h er hashfunksjon, i teller opp for hvert forsøk. Prøver neste posisjon til vi finner en tom plass. Kan få en lang kjede med feilplasserte elementer som må hoppes over ved hvert eneste oppslag. Kvadratisk probing: $probe(h,i,m) = (h + k_1 i + k_2 i^2) \bmod m$. $k_1$, $k_2$ og $m$ kan ikke ha felles faktor, da testes kommer vi ikke til alle indekser. En forbedring til lineær probing; gjør at vi ikke får de samme lange kjedene: tall som kolliderer i ett forsøk vil ikke være nødt til å fortsette å kollidere så fremt de har ulik hashverdi. Dobbel hashing: $probe(h_1,h_2,i,m) = (h_1 + h_2 i) \bmod m$. Består av hashfunksjonene $h_1(k)$ og $h_2(k)$. Vi kan bruke $h_1(k) = k \bmod m$. Vi vil at $h_2(k)$ og $h_1(k)$ skal lage ulik verdi. $h_2(k)$ bør (må ikke) få ulike tall der $h_1(k)$ gir like tall. $h_2(k) \neq 0$. $h_2(k)$ og $m$ MÅ være relativt prime (hvis dette kravet ikke stemmer vil probesekvensen ikke gå innom alle posisjonene og det vil være umulig å plassere et element i noen poster). Med primtall $m$: bruk $h_2(k) = k \bmod (m-1)+1$. Med toerpotens $m$: bruk $h_2(k) = (2|k|+1) \bmod m$ Dette gir et oddetall som er relativt prim med tabellstørrelsen som er toerpotens.

NP-kompletthet P: Mengden av problemer som kan løses og sjekkes på polynomisk tid. Eksempel: Gange to tall, eller om en streng er et palindrom (at strengen er den samme reversert). NP: Mengden av problemer hvor løsningsforslag kan sjekkes på polynomisk tid, men kan ikke NØDVENDIGVIS løses på polynomisk tid. Å løse problemet må gjøres med bruteforce algoritme. $P \subset NP$. Om $P = NP$ er det vanskeligste problemet innenfor datavitenskap. Eksempel: Kan noen av $n$ heltall summeres til 0? NPC: Vanskelige problemer i NP. $NPC \subset NP$. Beste kjente løsning er i $O(2^n)$. Problemer i NP kan omformes til NPC i polynomisk tid. En løsning for ett NPC problem i polynomisk tid kan brukes for å løse alle NP problemer i polynomisk tid. Eksempel: Hamiltonsyklus. I en graf med $n$ noder, finnes det en sti som går gjennom alle noder én gang? Kan prøve alle veier $O(n!)$. Løsningsforslag kan sjekkes i $O(n)$. NP-hard: Problem er NP-hardt ⇔ et NPC problem kan omgjøres til dette problemet i polynomisk tid. Svar må ikke kunne sjekkes i polynomisk tid. $NPC \subset$

$NP$-hard.




Traveling salesman, NPC case study: Gitt en liste med byer og vektede kanter mellom byene. Finn en vei som er innom alle byene hvor distansen/cost/vekt er under x. NPH-variant: Finn den mest optimale veien som er innom alle byene. Halting-problemet: Vi har et program $Q$ som tar inn et program P sammen med et gitt input I. Vi spør "Vil program P stoppe med input I?". Hvis det stopper, looper vi for alltid. Ellers stopper vi. Om vi bruker $Q$ som input for både P og I får vi et paradoks: $Q$ sier at $Q_1$ stopper så $Q$ looper (paradoks) eller $Q$ sier at $Q_1$ looper så $Q$ stopper (paradoks).



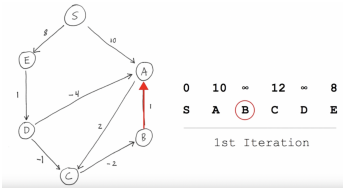

Avanserte proggeteknikker: Rå-kraft algoritmer utnytter datamaskinens kraft til å løse oppgaver. Å bruke en slik algoritme kan være lettere å implementere, men kjøretid kan ofte bli eksponentiell. De er vanligvis brukbare med polynomisk kompleksitet. Splitt og hersk-algoritmer deler et problem i flere deler, løser de ulike delene og finner så en løsning på hele problemet. En løsning på et subproblem kan lagres og brukes flere ganger. Splitt og hersk starter på toppen, jobber seg nedover til basis og jobber seg så oppover igjen. Probabilistisk algoritme bruker tilfeldighet for å finne global maksimum og ikke bare lokal. Denne tilfeldigheten er størst i starten og reduseres invers-proporsjonalt med iterasjoner. Algoritmen vil bare velge en bedre løsning gitt denne tilfeldigheten bestemmer det.
Dynamisk programmering (DP): DP ligner på splitt og hersk. DP hjelper når vi deler opp et stort problem, for så å løse samme delproblem flere ganger. DP starter nederst og lagrer løsninger mens algoritmen beveger seg oppover i rekursjonen. Eks:

```
int fib(int n)
{
   if (n <= 1)
      return n;
   return fib(n-1) + fib(n-2);
}
```

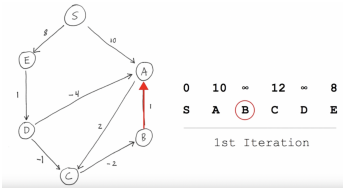
```
f[0] = 0;
f[1] = 1;

for (i = 2; i <= n; i++)
{
    f[i] = f[i-1] + f[i-2];
}

return f[n];
```

Vil DP fungere på sorteringsalgoritme? Sannsynligvis ikke. Tellesortering er IKKE dynamisk programmering. Siden DP handler om å løse samme problem flere ganger fordi vi lagrer det i en liste, vil ikke dette virke bra med å loope igjennom en sortert liste. Når trenger man grådig alg vs. dynamisk progg : endelig valg -> bør prøve grådig vs. foreløpig valg som kanskje må omvurderes -> dynamisk

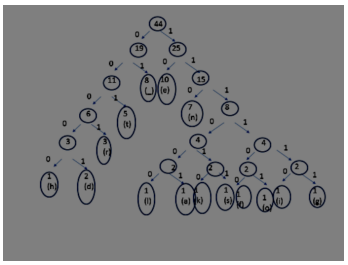
Ryggsekkproblemet: Vi har $n$ varer. En vare med index i har pris $p_i$ og vekt $v_i$. Hvordan skal vi få med varer med høyest verdi (pris) når vi bare kan bære V vekt? Det eksisterer ikke en enkel grådig algoritme for dette problemet. Hva går problemet ut på?: Hvis vi velger en vare $v_i$ må løse problemet "få med mest verdi når vekten er $V - v_i$. Når vi velger $v_j$ må vi løse samme problemet 2 ganger til bare med vekt $V - v_i - v_j$ og $V - v_j$. Etter hvert som vi tester flere forskjellige vekter, ser vi at samme problem blir løst flere ganger. Derfor kan det være lurt å lagre verdiene i en tabell når vi løser problemet. Vi kan bruke dynamisk progging for å løse dette. NB: V og v må være heltall for å kunne bruke dynamisk progging. Maks flyt: Ønsker å finne maksimal flyt fra kran (K) til sluk (S). Begreper. **Flyt**: Hvor mye som flyter igjennom en kant. **Kapasitet**: Hvor mye som *kan* flyte gjennom en kant. **Residual/rest-kapasitet**: kapasitet - flyt. **Flytøkende vei**: Én vei fra K til S der kapasiteten ikke er fullt utnyttet. **Restnett**: En graf som viser uutnyttet kapasitet i flytnettverket. **Flytkanellering**: I blant kan vi øke flyten ved å «snu» flyten gjennom en kant som har uhensiktsmessig flyt. Hvis det flyter 3 enheter fra node 1 til node 2 vil det være en restkapasitet på 3 fra node 2 til 1. En kant hvor det ikke flyter noe vil altså ha en reversert restkapasitat på 0, og en kant hvor det er en flyt $f$ vil ha en reversert restkapasitet på $f$. Når vi tidligere har lagt på flyt gjennom en kant, kan det hende at en senere flytøkende vei går gjennom samme kant i motsatt retning. Da legger vi ikke på flyt i motsatt retning, men kutter egentlig ut flyt som viste seg å gå i feil retning. Algoritmen kansellerer altså flyt som har blitt lagt på tidligere Nice? Barnajs! Ford fulkerson:1: Finn en hvilken som helst flytøkende vei. 2: Finn kanten med minst restkapasitet i denne veien. 3: Øk flyten langs hele veien tilsv. denne restkapasiteten (og dermed også dekrementer kapasiteten med flyten). Metoden har en svakhet; den kan ta lang tid! Edmonds Karp:Finner flytøkende veier ved å gjøre BFS søk. Start BFS i kildenoden. Finner (og bruker opp) de flytøkende veiene som har færrest kanter. $O(NK^2)$. Generelt om flytnettverk:: All transport må gå hele veien fra K til S, ingen mellomlagring. Flyt i en kant kan maksimalt være lik kapasiteten, men kan være mindre. Positiv flyt i en retning, kan sees på som negativ flyt i motsatt retning. Det er mest sannsynlige lettere å bruke Ford Fulkerson for hånd! Niceu! Min/-max spenntre (urettet graf): Kruskals algoritme: $\Theta(E * log(E))$ Ta kanten E som har minst/mest verdi og fargelegg denne. Ikke fargelegg kanten hvis fargeleggingen gjør at det ikke blir et tre. Gjør dette til du er igjennom alle kantene. Prims algoritme: $O(E * log(V))$ Prim er en grådig algoritme. Start i en vilkårlig node. Velg korteste kant fra denne og marker noden som besøkt. Gjør dette for alle noder uten å ødelegge treet. Dijkstra: Med heap: $\Theta((V + E)log(V))$. Finner korteste vei mellom startnode og alle andre noder. Kan ikke brukes med negative kanter og er en grådig algoritme. Negative kanter går ikke ettersom dijkstra går ut ifra at en besøkt node ikke kan ha kortere vei til seg. Dette er fordi dijkstra er en grådig algoritme. Algoritme: Sett distanser til alle noder til uendelig. Startnoden får distanse 0. Repeter for ubesøkte noder: Sett distanser til naboer av nåværende node til det minste av nåværende distanse og distansen til nåværende node. Velg så nærmeste node som nåværende om den ikke er valgt enda og sett den til besøkt. Bellman-Ford: $O(|V| * |E|)$ I motsettning til Dijkstra kan Belleman-Ford bli brukt på grafer som er negativt vektet, så lenge det ikke er en negativ syklus man kan nå fra startnoden. Algo: Iterer igjennom alle nodene en gang og oppdaterer vektene på hver node.



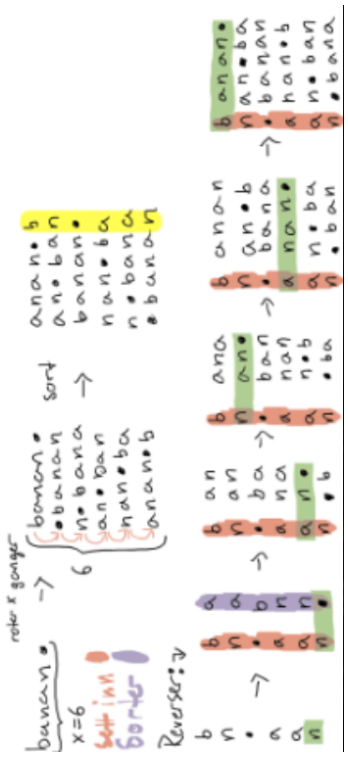

På bilde kan man se hvordan det ser ut midt i første "syklus" av algoritmen. Gjør sykluser helt til avstanden i nodene ikke forandrer seg. A*: Variasjon av Dijkstra som typisk bruker geografisk eller euclidisk avstand i sammeligningen mellom hvilken node som er neste man skal velge.




På bilde ser man at nodene som er nærmest målet blir prioritert. Med grafer som vi bruker vil A* for det meste bevege seg i en oval mot målet, i stedet for en sirkel rundt start som i Dijkstra. ALT: Tekstsøk: Naiv algoritme gjør et flytt på søkeord per sjekk. Gir $O(n \cdot m)$ $\Omega(n)$. Boyer-Moore er bedre: Ser på sistetegn i søkeord. Hvis den ikke er lik flytter vi så langt vi kan. Ellers ser vi på nestsiste osv. Algoritmen bruker 3 regler: Upassende tegn: Preprosesserer tekst til en todimensjonal tabell som gir antall flytt basert på indeks til upassende tegn og indeks på tegnet i søkeordet. $\Omega(n/m)$. Kan gi maksimalt antall flytt lik antall tegn i søkeord. Evt.: antall flytt: m - t hvor m er søkeordlengde og t er indeks på hvilket tegn som ikke passet fra høyre. Passende tegn: Om søkeord som overlapper seg selv: Tabell forteller antall flytt basert på antall tegn som passer. Blant ulike regler for antall steg man skal flytte, velger vi den som gir maks flytt. Galil: Trenger ikke å sjekke overlappende område om vi flyttet kortere enn forrige sjekk. $O(n)$ $\Omega(n/m)$. Datakompresjon: Run-length coding: Reduserer repitisjoner av bytes til et tall. AAAABC → 4ABC. Legger til et tall for hvor mange bytes som ikke er komprimert. Dårlig for repeterte sekvenser av samme flere ulike bytes. Lempel-ziv: Setter inn referanser: Repeter X tegn som er Y tegn bakover. Må også ha tall som teller antall tegn til neste referanse. Må bestemme størrelse på fremoverbuffer og søkebuffer. Fra Helge Chadling: Kompresjon oppnås ved å erstatte gjentatte strenger med referanse til den forrige strengen. Når man komprimerer, leser man gjennom teksten. Strenger man ikke har sett før, skriver man opp som vanlig. Strenger man har sett, erstattes med en referanse som forteller hvor langt bakover man må gå for å finne kopien, og hvor lang kopien skal være. Dermed er det mulig å pakke ut igjen senere. Hvis referansene er kortere enn strengene de erstatter, oppnår vi kompresjon. Demonstrer Lempel-Ziv kompresjon på «fort, fortere, fortest» Her bruker jeg klammer [] for å vise referanser. Første tall er angir «tegnbakover», andre tall angir hvor mange som skal kopieres: «fort, [6,4]ere[9,7]st» I alt 11 tegn erstattes av to referanser. Lempel-ziv welsh: Bygger opp en ordliste og bytter ut tegnsekvenser med tall fra ordlista. Kan være raskere men samme kompresjon som Lempel-Ziv. Huffman: Lager et huffman tre, binær tre, hvor hver node er et tegn med frekvensen av den i teksten. Høyest frekvens på toppen. Bytekoden til et node er traverseringen gjennom treet. 0: Gå til venstre, 1: Gå til høyre. Grunnen til at avkodingen er entydig er at ingen av kodene er et prefis for en annen kode. Denne algoritmen har egenskapen av: prefiksfrie koder. Huffman tre for "dette er en helt annen tekst enn den forrige".



Burrows Wheela:




Bildet er rotert 90 grader Ikke kompresjon, men en transformasjon som samler like tegn sammen. Sett et bestemt tegn på slutten av tegnsekvensen. Roter sekvensen for å få alle mulige kombinasjoner av den. Sorter de ulike sekvensene på hverandre. Siste tegn i hver sekvens utgjør den ferdige transformasjonen. Reversering: Går i en løkke hvor vi legger til den transformerte sekvensen, setter den først og sorterer til det er lagt til like mange sekvenser som tegn i den transformerte. Riktig rad er den som har det bestemte tegnet på siste plass. Move to front: Komprimerer ikke, men forbereder. Initialiser tabell hvor hver celle er indeksen. For hvert tegn i input: Finn tegn i tabellen og skriv til output. Flytt tegnet vi fant til første plass i tabellen. Dette gir mer effektiv Huffmankoding. Adaptiv: Lager nytt Huffmantre ettersom en tekst endrer seg. F.eks. norsk til engelsk. Aritmetisk: Bedre enn Huffman ved å bruke mindre enn 1 bit for symboler som har mindre sjanse for å komme. Problem kan være at datamaskiner er trege med desimaltall. RBZip2: En kombinasjon av: Run-length coding, Burrows Wheeler, Move-To-Front, run-length coding igjen og huffmankoding.
Topologisk sortering:Vi ønsker å finne en mulig rekkefølge, f.eks. produksjonsplanlegging i en urettet graf. En graf med én eller flere sykler kan ikke topologisk sorteres. Ofte er det mange gyldige løsninger. **Algo:** 1. Vi starter DFS i en tilfeldig node. Hver gang DFS er ferdig med å behandle en node, lenkes den inn først i resultatlista. Så lenge det er urørte noder igjen, starter vi nye DFS. Med naboliste blir det $\theta(N + K)$. Figur 6.9 er et tre og vil dermed ikke ha noen sykler. Antar kantene er urettet nedover nivåene. Vi kan da gjøre en topologisk sortering. Starter i node 1. Beveger oss til 3 og så 5. I node 5 kan vi ikke besøke flere noder. Node 5 blir nå ferdig behandlet og lenkes inn først i resultatlista. Så er vi tilbake i node 3. Kan ikke besøke flere noder fra node 3. Node 3 er dermed også ferdig behandlet og lenkes

inn først i resultatlista. osv osv. En mulig topologisk sortering: 1, 2, 4, 7, 6, 3, 5.

Sterk sammenhengende komponenter:1. Gjør samme fremgangsmåte som på en topologisk sortering. 2. Transponer grafen. 3. Gjør så en ny "topologisk sortering" (er ikke faktisk en topologisk sortering, men er den samme fremgangsmåten), men starter med den første noden i resultatlista. Resultatlista kan ansees som en stakk. Fortsett å pop ut noder til alle noder er besøkt. Sykler blir nå sterke sammenhengende komponenter. Noder som står alene blir også en sterk sammenhengende komponent.

Bit operasjoner:

| Bit Operasjon | Symbol | Eksempel |
|---|---|---|
| Høyre shift | » | x » 1 = x/2 |
| Venstre shift | « | x « 1 = 2x |
| XOR | ^ | 1001 $^{1}$010 = 1100 |
| OR | \| | 1001 \| 1010 = 0011 |
| AND | & | 1001 & 1100 = 1000 |
| NOT | | 1010 = 0101 |

Oppgaver fra tidligere eks:

Forklar hvorfor Dijkstras algoritme ikke fungerer korrekt på grafer som har kanter med negative vekter:

Dijkstras algoritme ser på hver node bare én gang, og regner deretter noden som «ferdig behandlet». For at dette skal gi rett resultat, må en node med kort avstand gjøres ferdig før alle noder med lengre avstand. Algoritmen bruker en prioritetskø for å finne den uferdige noden med kortest avstand, og sjekker om kantene ut fra den gir kortere veier til naboene. Alle ferdige noder har kortere avstand enn den noden som er under behandling. Alle uferdige noder har lenger avstand. Med positive kantvekter kan vi aldri få kortere vei inn mot en ferdigbehandlet node, fordi noden vi jobber med allerede har høyere avstand. (En positiv kantvekt kan bare gjøre avstanden enda større.) Med negative kantvekter kan en node som ligger lenger unna, likevel gi kortere vei inn til en «ferdig» node. I så fall må kantene ut fra denne noden undersøkes på nytt, men det gjør ikke Dijkstras algoritme. Derfor kan den mislykkes med negative kanter. Dette kan skrives på mange andre måter. Noen viste hvordan dette går galt, med en eksempelgraf med negativ kant.

beskriv halting-problemet, og forklar hvorfor det ikke er løsbart.

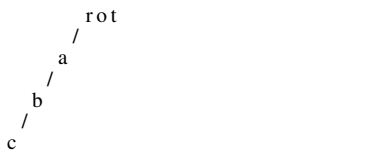
Beskrivelse: Halting-problemet går ut på å kunne si hvorvidt et program vil stoppe eller ikke, med en gitt input. Et program som løser dette problemet, vil altså ta et annet program som input. Det er generelt uløselig, fordi: La oss si at vi har et program H, som løser halting-problemet. Da kan vi lage et program P,som tar et program X som input. Ved hjelp av H sjekker P om X vil stoppe eller ikke når det får sin egen kildekode som input. Hvis H sier X vil stoppe, kjører P en uendelig løkke. Hvis H sier X stopper ikke, vil P avslutte. Hvis vi kjører P med sin egen programkode som input, får vi en umulig situasjon. Altså er programmet H umulig.

Kan vi bruke dynamisk programmering for å lage en enkel og effektiv sorteringsalgorit- me? Hvis ja, foreslå en slik algoritme. Hvis nei, forklar hvorfor dette er vanskelig. Sannsynligvis ikke? Dynamisk programmering hjelper når vi deler opp et stort problem, for så å løse samme delproblem flere ganger. Sortering kan deles opp, slik f.eks. quicksort og flettesortering gjør. Men det er lite sannsynlig at vi ser samme delproblem flere ganger. Å sjekke om en deltabell matcher en tidligere sortert tabell, vil lett ta mer tid enn det sorteringsarbeidet vi sparer. analysen av Prims algoritme, står det at O ((N+ K) log N) kan forenkles til O (K log N). Forklar hvorfor dette stemmer.Prims algoritme finner et minimalt spenntre. Den brukes derfor bare på grafer som henger sammen, for ellers finnes ikke noe spenntre. For at grafen skal henge sammen, må k >= N - 1. I asymptotisk analyse kan vi se bort fra "-1", og sitter igjen med K >= N. Når K er størst, kan vi fjerne N fra K + N, dermed forenkles uttrykket til O (K log N).

Hva vil det si, at et problem er i kompleksitetsklassen NP ? Gi også eksempel på et problem som er i NP, og fortell hva problemet går ut på.

Etter definisjonen: Problemer i NP har løsning som kan sjekkes i polynomisk tid (der n er størrelsen på problemet.) Problemet kan ikke nødvendigvis løses på polynomisk tid. Mange problemer fra pensum, som travelling salesman. For full uttelling må det stå hva det presenterte problemet går ut på

Tegn et binærtre med fire noder. Det skal være slik at enten vi skriver ut nodene i postor- denrekkefølge eller in-orden, så får vi samme rekkefølgeVed in-orden utskrift, behandles først nodens venstre barn. Så skrives nodens eget innhold, og til slutt behandles nodens høyre barn. Ved postorden utskrift behandles først nodens venstre barn, så nodens høyre barn, og til slutt skrives nodens eget innhold. Forskjellen ligger i behandling av høyre barn, så hvis postorden og in-orden skal gi samme resultat, må vi ha et tre hvor alle barn er til venstre:




in-orden og postorden vil skrive «c b a rot»

Når vi implementerer quicsort, er det lett å gjøre feil som gir unødvendig dårlig ytelse. Fortell om slike problemer, og hvordan vi unngår dem.• n2 skjevdeling ved sortering av sorterte data. unngå ved å velge midterste element som delingstall, evt. median av 3/5/7/... • n2 skjevdeling ved sortering av duplikater. <,> i indre løkker, heller enn <= og >= • for dyp rekursjon. Test dybden og bytt til heapsort, eller iterer på det største interval- let • skjevdeling med 0 og n elementer, som gir uendelig løkke. Unngås med median3sort, som gir oss ihvertfall ett «lite» og ett «stort» tall. Unngås også av andre metoder som ikke sorterer delingstallet omigjen. • Bedre ytelse ved å bytte til innsettingssort når intervallene blir tilstrekkelig små

Dijkstras algoritme og Bellman-Ford algoritmen løser samme problem, men med ulik kompleksitet. Grei ut om når vi bruker den ene algoritmen, og når vi bruker den andre.Dijkstras algoritme håndterer ikke negative kanter, så for slike grafer er vi nødt til å bruke Bellmann-Ford algoritmen. Ellers foretrekker vi lavest kompleksitet. Dijkstras algoritme har kompleksitet O ((N+K)log N), Bellman-Ford er O (KN) For de fleste grafer (med positiv vekting) vil dermed Dijkstras algoritme være å foretrekke. Unntaket er hvis K < log N . For en slik graf, (som har for få kanter til å henge sammen), vil Bellman-Ford algoritmen være raskere

Prioritetskøer kan implementeres som henholdsvis usortert tabell, heap og fibonacci-heap. Fortell om fordeler og ulemper ved hver av dem..Usortert tabell Fordeler Enkel å implementere. Prioritetsendring O (1) Ulempe Å finne eller hente ut minimum tar O (n) tid heap Kjapp, men noe mer komplisert enn usortert tabell. Å finne minimum tar O (1) tid. Å hente ut minimum eller endre prioritet tar O (log (n)) tid. fibonacci-heap Kjøretider som for heap, bortsett fra at prioritet endres i O (1) tid. Ulempen er at strukturen er mer komplisert å programmere. Dijkstra med fibonacci-heap er $O(N * log(N) + K)$.

velgesort:pseudokode

```
velgesort([]t) {
    for (i = t.length - 1; i > 0; --i)
      {
        int max = 0;
        for (j = 1; j < i; ++j) {
            if (t[j] > t[max]) max = j
    ;
        }
        if (max != i) bytt(t, i, max);
    }
}
```

boblesortering:pseudokode

```
boblesort([t]) {
    for (i = t.length-1; i>0; --i) {
        for (j=0; j < i; ++j) {
            if (t[j] > t[j+1])
            bytt(t, j, j + 1);
        }
    }
}
```

innsettingsortering:pseudokode

```
innsettingssort([]t) {
    for (j = 1; j < t.length; ++j) {
        int bytt = t[j];
        //Sett t[j] p  rett plass:
        i = j - 1;
        while(i >= 0 && t[i] > bytt)
      {
            t[i + 1] = t[i];
            --i;
        }
        t[i + 1] = bytt;
    }
}
```

flettesort:pseudokode

```
flettesort([]t, v, h) {
    if (v < h) {
        m = (v + h) / 2;
        flettesort(t, v, m);
        flettesort(t, m + 1, h);
    flett(t, v, m, h);
    }
}

flett([]t, v, m, h) {
    []ht = new int[h - v + 1];
    i = 0, j = v, k = m + 1;
    while (j <= m && k <= h) {
        ht[i++] = (t[j] <= t[k])
      ?
        t[j++] : t[k++];
    }
    while (j <= m) ht[i++] = t[j
     ++];
    for (i = v; i < k; ++i) t[i]
      = ht[i - v];
}
```

tellesort:pseudokode

```
tellesort([]inn, []ut, k) {
    i, n = inn.length;
    []ht = new int[k + 1];
    for (i = 0; i <= k; ++i) ht[i] = 0;
    for (i = 0; i < n; ++i) ++ht[inn[i]];
    for (i = 1; i <= k; ++i)
    ht[i] += ht[i - 1];
    for (i = n - 1; i >= 0; --i)
    ut[--ht[inn[i]]-1] = inn[i];
}
```

Shellsort:pseudokode

```
shellsort([]t) {
    s = t.length / 2;
    while(s > 0){
        for(i = s; i<t.lenght;++i){
            int j = i, flytt = t[i];
            while(j >= s && flytt < t[j-s]) {
                t[j] = t [j - s];
                j -= w;
            }
            t[j] = flytt;
        }
        s = (s == 2) ? 1 : (int)(s / 2.2);
    }
}
```

Rasksort:pseudokode

```
def median3sort([]t, v, h):
 m = (v + h) / 2
if t[v] > t[m]
    bytt(t, v, m)
if t[m] > t[h]:
    bytt(t, m, h)
    if t[v] > t[m]
        bytt(t, v, m)

return m

def splitt([]t, v, h):
iv, ih
m = median3sort(t, v, h)
dv = t[m]  # delingsverdi
bytt(t, m, h - 1)
for (iv = v, ih = h - 1;;)
    while (t[++iv] < dv)
    while (t[--ih] > dv)
    if iv >= ih
        break;

    bytt(t, iv, ih)

bytt(t, iv, h-1)
return iv

def quicksort([]t, v, h):
if (h - v > 2)
    delepos=splitt(t, v, h)
    quicksort(t, v, delepos-1)
    quicksort(t, delepos+1, h)
else
    median3sort(t, v, h)
```

fordel med quicksort, og insertion sort

Fortell om sterke og svake sider ved quicksort og innsettingssortering. Gi eksempel på når du vil bruke den ene sorteringa, og når du vil bruke den andre.

| | Quicksort | Innsettingssort |
|---|---|---|
| Sterke | Store datasett | • Små datasett, $n < 100$<br>Nesten/helt sorterte sett |
| Svake | Datasett laget spesielt for $n^2$ worst case | Store tabeller |
| Når bruke | For det meste | Hjelp for quicksort, datasett vi vet er små/passende |