Kompleksitet

Om p=0 og koden er: for (int i = 0; p <= n; i++) { p = p + i}, da blir O(sqrt(n))
Om: for (int i = 0 eller n; i < n eller i>1; i = i*eller i=i/2) {stmt} $O(\log_2 n)$

Grenser:
Θ -> felles, O -> øvre grense, Ω -> nedre grense

Rekursiv kompleksitet:
$aT(n/b) + cn^k$, a = antall rekursive kall, b = brøkdelen av datasett vi behandler, cn^k = kompleksitet for metoden (vanlige løkker). K=ant løkker
$b^k < a$, $\Theta(n^{\log_b a})$. $b^k = a$, $\Theta(n^k \log n)$. $b^k > a$, $\Theta(n^k)$, konstanter: $a \geq 1$, $b > 1$ $c,k \geq 0$

Rekursjon:
Splitt og hersk: deler opp problemene i underproblem, løse disse, og sette sammen disse løsningene til en løsning for det opprinnelige problemet.

Sortering

- Minste kjøretid for sortering uansett er **O(n)** fordi minst alle elementer må sjekkes.

Insettingssortering ($O(n^2)$, $\Omega(n)$):
- setter inn ett og ett element blant de sorterte i venstre del. Går videre til neste usorterte element i tabellen osv.
- effektiv på SMÅ datasett og spesielt god om data er delvis sortert fra før av. Svak på større datasett.

Boblesortering ($\Theta(n^2)$):
- lite brukt ettersom innsettingssortering er like enkel og som regel raskere i praksis. Jobber nedefra og oppover, derfor "bobler" det minste tallet opp til laveste plass.
- ved hvert gjennomløp sjekkes alle tall som står etter hverandre, og bytter plass hvis de står feil i forhold til hverandre -> små tall bobler opp, store tall synker ned

DualPivotQuickSort (10-20% raskere enn QS):
- deler i 3 deler: mindre enn p1, mellom p1 og p2, og større enn p2
- noe mer arbeid men 10-20% raskere enn vanlig QS fordi det blir færre sammenlikninger -> mindre rekursjon, og cache-effekten skjuler ekstra arbeidet

QuickSort (i gjennomsnitt O(n log n) rekursiv metode**:**
- brukes mest, men ikke så effektiv på små datamengder
- plukker ut en delingsverdi (pivot) hvor alt som er mindre plasseres nedenfor og alt større ovenfor. Har nå 3 deler (delingsverdien, som nå står på rett plass, små verdier på ene siden og store verdier på andre siden). De to sidene sorteres rekursivt og splittes i enda 2 deler per side -> slik fortsetter det til alle deler har tabellstørrelse 1 og da er tabellen sortert.
- ulempe: om pivotelementet er veldig skeivt i tabellen. verste tid på O (n^2)

Velgesortering (($\Theta(n^2)$):
- finn største verdi og sett på plass n-1, finner nest største og setter på n-2 osv. til nest minste er satt på plass 1. da vil plass 0 alltid være minste element fordi alle andre er satt på riktig plass.

Flettesortering (O(n log n)):
- rekursiv sortering av tabellen delt i to, så flettes de sorterte deltabellene sammen til en sortert tabell
- trenger en ekstra tabell -> ulempe og tar mer plass

Tellesortering (O(n)):
- sorterer kun heltall -> veldig stabil
- lønner seg ekstra mye når tallene ligger tett og evt. Med mange like verdier
- ulempe: har en ekstra hjelpetabell

Shellsort ($O(n^2)$):
- Sammenligner og flytter først tall som ligger langt unna hverandre, n/2. Når store og lange hopp er gjennomført så gjøres en vanlig innsettingssortering.
- Den verste kompleksiteten er O(n2). Når S synker kan vi få kompleksitet på O(n3/2). Om S deles på 2.2 for hver omgang av den ytre løkka blir kompleksiteten i nærheten av O(n7/6)-O(n5/4)

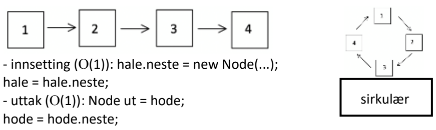
Heapsort (O(n log n), Ω(n)):
- lager heap av usortert data og tar vare på lengden av heapen. Har ei løkke som plukker ut største tallet og setter inn igjen i tabellen bakenfor heapen. Selve heapen blir mindre og mindre mens en større og større del av tabellen blir sortert. Holder alltid oversikt over heap-lengden.
- samme kompleksitet som flettesortering MEN klarer seg uten det ekstra minnet som flettesortering trenger, som er stor fordel for heapsort. (I praksis er QS raskere).

Topologisk sortering (Θ(N + K)):
- planlegger aktiviteter som avhenger av hverandre -> fag på skolen eller klassetrinn f.eks.
- betingelsene kan representeres med en rettet graf med én node per aktivitet.
- lage en MULIG sortering av en slik graf med avhengigheter
- graf må være asyklisk (altså IKKE rundturer)
- resultatet er er ei lenka liste med alle nodene i en mulig rekkefølge. Kjører DFS på hver node (som er nummerert)
- DFS legger nodene i resultatlista i samme rekkefølge som de blir ferdige, altså når alle kantene ut fra dem er ferdig undersøkt.

Lister

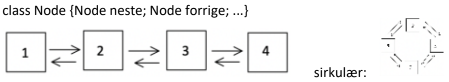
Stack:
- lineær datastruktur -> LIFO (last in first out)

Enkeltlenka liste (sortere: O(n log n)):

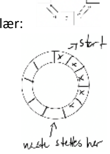


- innsetting (O(1)): hale.neste = new Node(...);
hale = hale.neste;
- uttak (O(1)): Node ut = hode;
hode = hode.neste;

Dobbellenka liste (sortere: O(n log n)):
class Node {Node neste; Node forrige; ...}



sirkulær:



Kø:
kan implementeres med liste eller tabell
Tabell: innsetting bakerst: O(1), uttak forrerst: O(n)
Liste: begge blir O(1)

Trær

- datastruktur av typen graf
- ulineær datastruktur -> har en rot (slik som lenka liste)
- har noder hvor hver kan lenke til mer enn en annen barnenode. Alle har forelder utenom rota
- noder uten barn = løvnoder/ytre noder
- noder med barn = indre noder

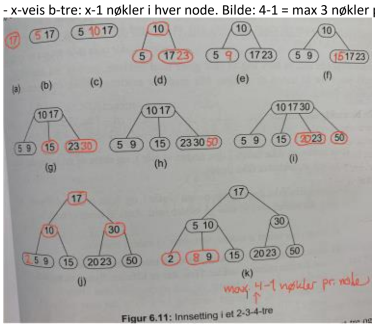
Fritt tre: sammenhengende, asyklisk graf
Tre med rot: fritt tre + en rot
Ordnet tre: bestemt rekkefølge på nodene
Skog: flere trær

Dybde:
- antall lenker mellom noden og rota (rota har dybde 0)
- hvis nodene har foreldrelenke: følg foreldrelenkene opp til rota. Kan gjøres med en enkel løkke.
- hvis nodene ikke har foreldrelenke: start i rota, let i begge subtrær til noden påtreffes, tell antall lenker (enklest da med rekursiv algoritme)

Høyde:
- antall lenker mellom noden og den noden som er lengst unna i et av nodens subtrær (treets høyde er definert som rotas høyde)
- høyden til en node er 1 mer enn høyden til det høyeste subtreet.
- rekursiv algoritme: finn først høyden til venstre subtre, så høyre subtre -> velg den største av de to og legg til én

Binærtre:
- maks to barn pr forelder
- er ordnet, barnas rekkefølge er vesentlig
- kjøretiden ved innsetting, sletting og søk øker proporsjonalt med høyden til treet
- Binært søketre: hver node har nøkkel og alle noder i venstre subtre har mindre nøkkel og alle noder i høyre subtre har større nøkkel
- perfekt binærtre: hvis alle løvnodene befinner seg på nederste/samme nivå og alle indre noder har to barn
- komplett: perfekt, men trenger ikke fult antall løvnoder
- fullt: alle indre noder har 2 barn (hver forelder har 2 barn). Løvnoder trenger ikke være på samme nivå
- innhold: info + tre lenker til andre noder (forelder + 2 barn). Treet er ordnet
- nodens nøkkel er større enn nøklene til alle noder i venstre subtre, og mindre enn alle nøklene i nodens høyre subtre
- innsetting: er rota tom -> sett inn der. Hvis nye nodens nøkkelverdi er mindre enn nøkkelverdien til den vi sammenlikner med, skal vi gå til venstre, er den større går vi til høyre. Fortsetter til vi finner en node som mangler det barnet vi skulle ha gått videre til -> der settes noden
- søking: skjer omtrent som innsetting, men må teste på like nøkler -> finner lik nøkkel -> return node, else return null (finnes da ikke)
- sletting: **1:** noden har ikke barn -> lenka fra foreldrenoden som lenker til noden, settes = null. **2.** noden har ett barn -> foreldrenoden sin lenke til noden som skal slettes settes til å lenke til nodens barn. **3.** noden har to barn -> ertstatter nodens element (alle data bortsett fra lenkene) med elementet i noden som kommer like etter sorteringsrekkefølgen, som er den minste i høyre subtre – denne noden vil oppfylle kravet om at nøkkelverdien er større enn alle nøkkelverdier i venstre subre og er mindre enn alle nøkkelverdier i høyre subtre

B-trær:
- en datastruktur som brukes for å få effektiv bruk av disker og andre sekunærlager -> mye brukt i databasesystemer
- ikke binær, kan ha hundrevis av barn -> alle barn på samme nivå og alle noder har mellom n-1 og 2n-1 nøkler for en eller annen n > 1
- søker: hente en node på hvert nivå -> fordel med høyde på treet. Velger n slik at én node tar et helt antall sider (pages) på disken, noe som gir effektiv framhenting.
- x-veis b-tre: x-1 nøkler i hver node. Bilde: 4-1 = max 3 nøkler pr node.



Figur 6.11: Innsetting i et 2-3-4-tre

2, 3 eller 4 barn

Heap

- Komplett binærtre hvor nodene har sammenlignbare nøkkelverdier
- max-heap: hver node har en nøkkelverdi som er større enn eller lik begge barnenodenes
- min-heap: hver node har en nøkkelverdi som er mindre enn eller lik begge barnenodenes – rota har dermed minste verdi
- delvis ordning: gjør at heapoperasjoner er raske
- heap som tabell: sparer plass og tid i forhold til nodeobjekter med referanse til barnenoder. Nodenes posisjoner i tabellen kan beregnes -> for noden i posisjon i_{node} kan vi finne indeks til foreldre- og barnenoder slik:

indeks, i	0	1	2	3	4	5	6
tabell, t[i]	14	8	10	7	2	2	9

$$i_{\text{foreldrenode}} = \lfloor (i_{\text{node}} - 1)/2 \rfloor$$
$$i_{\text{venstre barn}} = 2 i_{\text{node}} + 1$$
$$i_{\text{høyre barn}} = 2 i_{\text{node}} + 2 = 2(i_{\text{node}} + 1)$$

	Heap	sortert tabell	usortert tabell
Initialisering	$\Theta(n)$	$O(n \log n)$	$\Theta(1)$
finne maks	$\Theta(1)$	$\Theta(1)$	$O(n)$
hent_maks	$O(\log n)$	$O(1)$	$O(n)$
justere prioritet	$O(\log n)$	$O(n)$	$\Theta(1)$
sette inn ny	$O(\log n)$	$O(n)$	$\Theta(1)$
fjerne node	$O(\log n)$	$O(n)$	$\Theta(1)$

- sletting: slette elementet og hente inn siste element og flytter opp – tar så å setter den i riktig posisjon. Lag_heap: Θ(n), nXhent_max og tot: O(nlogn),Ω(n)

Prioritetskø (PQ):
- legge inn og ta ut elementer som har ulik prioritet. Forskjellen på PQ og andre lagringsstrukturer er at vi lett kan plukke ut elementet med den høyeste prioriteten. Ser at dette er enkelt når en max-heap brukes for å implementere PQ. (F.eks. av OS og prosesskjøring -> hvilken prosesser bør prioriteres osv)

Bit-operasjoner

- bit-operasjoner er veldig raske fordi så å si alle datamaskiner bruker binære tall. Fordi prosessoren er laget med binær digital elektronikk.
- bitverdi 0 representeres med 0 volt, 1 med høyere spenning

Høyreskift:
- int a = 154;
a >>= 1; // a = a >> 1; -> tilsvarer a = a/2; som blir 77
eks.: 10011010 (154) og høyreskifter 1 blir: 01001101 (77)
- et høyreskift tilsvarer divisjon med 2

Venstreskift:
Int a = 5;
a <<= 3; // a = a << 3; -> tilsvarer a = a*8;
eks.: 000101 (5) og venstreskift 3 blir: 101000 (40)
- et venstreskift tilsvarer multiplikasjon med 2

Bitvis xor (^): 1 der bitene er ulike, 0 der de er like
Bitvis and (&): a & b er 1 hvis både a og b er 1, ellers 0:
210   11010010
124 & 01111100
80   01010000   -> int a = 210, b = 124; int c = a & b;
- raskere beregning med maskering (&) enn «mod»
- if((tall % 2) == 1)... blir med maskering: if((tall & 1) == 1)
- «& 1» plukker ut siste siffer:
- x%4 plukker ut siste 2 siffer, samme som x & 3
- x%8 plukker ut siste 3 siffer, samme som x & 7
- generelt: erstatt «x mod 2» med «x & (2^n-1)»

Bitvis or (|): omvendt av bitvis and
Bitvis not (~): inverter alle bitene

IP-adresser og nettmasker:
- ruting: er 10.50.82.218/22 på samme nett som 10.50.83.167?
-> /22 delen betyr nettmakse med 22 1´ere og resten 0'ere (32 bit)

```
 Min ip base10:        10        50        82       218
       Binær ip: 00001010001100100101001011011010
      Nettmaske: 11111111111111111111110000000000
    Nettadresse: 00001010001100100101000000000000
 Net.adr.base10:        10        50        80         0
Annen ip base10:        10        50        83       167
       Binær ip: 00001010001100100101001110100111
      Nettmaske: 11111111111111111111110000000000
    Nettadresse: 00001010001100100101000000000000
 Net.adr.base10:        10        50        80         0
```
tre kjappe instr og JA!

Kryptografi:
- scramble data og kunne dekode senere.
- xor brukes ofte: «melding xor nøkkel» gir uleselig tekst, motsatt igjen gir klartekst -> nøkkel må være like lang som tekst -> upraktisk
- miksing: flytte alle bitene i melding. Gjør det vanskelig å gjette seg frem. «and» og «or» kan brukes. Skift og rotasjon kan flytte bitene rundt innengor en byte eller int.
- kryptering ved bruk av bit-operasjoner forsinker ikke komm. -> viktig for både komm. over nett og for å jobbe mot kryptert HD.

Datakompresjon:
- presse sammen data mest mulig. Kan se på repetisjoner i tekst og erstatte like ord. Se Huffmantre!

Hashtabeller

- oppnår ytelse ved å spre nøkler jevnt utover en stor tabell og dobbelhashing er god til dette -> unngår lange kjeder
- problemer ved bruk i moderne prosessorer: cache mye raskere enn alt annet og ideelt om man jobber innenfor denne. Hashtabeller som er større enn minnet og det blir spredning fører til swapping.
- bruker en hashfunksjon for å produsere en tabellindex: tabell [hashThis(nøkkel)]
-> ved hjelp av hashfunksjon beregner vi da index
-> blir raskere oppslag selv om selve nøkkelen ikke er egnet
-lastfaktor: for en hashtabell med størrelse m som inneholder n elementer, er lastfaktoren: $\alpha = n/m$
-> dårlig lastfaktor når α er nær 0
-> full tabell når $\alpha = 1$ og overfylt når $\alpha > 1$
- gode hashfunksjoner er raske til å beregne, gir verdi mellom 0 og tabellstørrelse, gir god spredning og gir få kollisjoner.
-> håndtering av kollosjoner: avvise kolliderende elementer, bruke lenka liste, åpen adressering

Bruk av lenka liste:
- hver tabellposisjon er et listehode. Når flere elementer hasher til samme index, lenker vi dem der
- fordel: tabellen blir aldri full, kan få $\alpha > 1$
- ulempe: det får med plass til neste-referanse

Bruk av tabell (åpen addressering):
- linær probing: «neste ledig plass». Minus ved denne er at man kan få lange kjeder med kollisjoner. Aller verdier som havner i dette området må gå igjennom samme lange kjede
- kvadratisk probing: prøver å unngå de lange kjedene i lineær probing. Noen elementer kolliderer og lager kollisjonskjede. Andre elementer som treffer lenger ut i denne kjeden, vil ikke følge samme kjede pga. et andregradsledd i probingen. Dermed finner de fortere ledig plass. Ulempe: vil også oppstå kollisjonskjeder
- dobbelhashing: det beste alternativet! Unngår kvadratisk probing ved å bruke to hashfunksjoner og sprer evt. kollisjoner jevnt utover -> unngår lange kollisjonskjeder. Bruker den andre hashfunksjonen kun hvid det blir kollisjon på første. Husk: ingen felles faktor ved utregning. Og bør ha tabellstørrelse i primtall. Brukte dobbelhashing og tabell med lenka liste på øvingen.
- tabellen kan ikke overfylles uansett. Bør ha 25% overhead
- hashfunksjon 1 (k mod m) hvor m = primtallet (tabellstørrelse) og k = random tall (verdi i tabellen)
- hashfunksjon 2 (k mod (m – 1) + 1) hvor k og m er primtall for å unngå evig løkke. K sendes inn fra hashfunksjon 1.

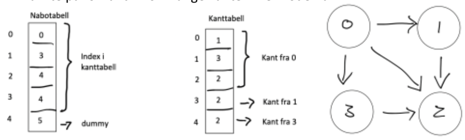
Uvektede grafer:

- graf: en datastruktur som består av noder, og en rekke forbindeler mellom disse. Forbindelsene kalles kanter
- nabo: kanten n -> m vil si m er navo til n
- vei: en kjede noder som henger sammen
- rundtur: vei som ender i startnoden
-> en graf med rundtur(er) er syklisk
-> en graf uten rundturer er asyklisk
- veilengde: antall kanter i veien

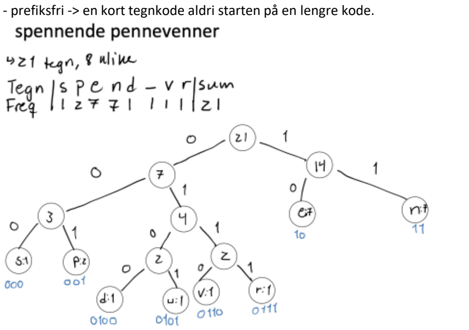
Implementasjon:
Holde orden på kantene: naboliste eller nabotabell (disse to gir retta grafer). For å lage uretta graf -> lag alle aknter to ganger, en gang i hver retning.
Nabotabell: tar høyde for at alle kan ha forbindelse og er en todimensjonal tabell: Kant [N][N]
- Kant[i][j] forteller om det er fobindelse mellom i og j
-> kan bruke true/false for uvektet
-> bruke vekter (avstand, tid osv.) for vektet graf - ulempe: lett for å sløse med plass - ulempe: tar tid å finne alle kanter fra en node - fordel: lett å finne alle kanter ut fra en node

Naboliste1:
- tabell med noder hvor hver node er listehode og hver kant er et listeelement
-> referanse til neste kant og referanse til hvelken node kanten går til (evt. vekt også)
- trenger ikke lagre kanter som ikke finnes
- fordel: sparer plass i forhold til nabotabell
- fordel: lett å finne alle veier ut fra en node
- ulempe: vanskelig å finne alle veier inn til en node
- lenka liste kan utvides etterhvert – trenger ikke vite antall noder og kanter på forhånd

Naboliste2:
- en nodetabell og en kanttabell
- hver node har en index til sin første kant i kanttabellen. Resten følger på høyere indexer i kanttabellen fram til der neste node har sin første kant
-> dummynode til slutt for å vite hvor siste node slutter
-> krever ikke objektorientering, kan bruke bare int og virker derfor for alle språk
-> enkelt å progge, kompakt i memory
-> må vite på forhånd hvor mange kanter hver node har



Vektede grafer:
Korteste vei problemet varianter:
- en til alle: korteste vei fra en til alle
- alle til en: korteste vei fra alle andre noder inn til en målnode
- alle til alle: korteste vei fra alle noder til alle andre
Minimale spenntrær:
- spenntre: uvekta og uretta graf
- grafen beskriver punkter og (mulige) koplinger
- trenger ikke alle mulige forbindelse, bare nok til at grafen akkurat henger sammen. Trenger ikke rundtur
- en graf uten rundturer er et tre, når alle noder er med – et spenntre -> plukke vekk unødvendige kanter
- et minimalt spenntre er et hvor summen av vektene er minst mulig
- andvendelser: planlegge graving, rør, strømledninger
-> mange mulige traséer, kan ha ulik pris pga. lengde og vanskeligheter i terreng
-> kretskort/chip: kortere forbindelser er billigere
Dijkstras algoritme(Usortert tabell: O(N2)):
- finner alle veier inn til målnoden -> fint om man kjører feil og ny vei må finnes
- tåler ikke kanter med negativ vekt
- bruker PQ. Grådige valg. Velger ut den noden med lavest verdi og ofte vil alle noder og kanter bli undersøkt i en sirkulær bevegelse ut fra startnoden og til den treffer målnoden
s ut av køen, med prioritet/avstand 0.
Setter avstandene $a = 11$, $b = 10$ og $c = 15$.
b ut av køen, med prioritet 10. Setter avstanden $m = 25 + 10 = 35$
a ut av køen, med prioritet 11. Ingen oppdateringer.
c ut av køen, med prioritet 15. Setter avstanden $m = 15 + 5 = 20$, fordi $20 < 35$
m ut av køen, med prioritet 20. Den korteste veien ble altså $s \to c \to m$ med lengde 20
A*: videreutvikling av Djikstra
- har en «retningssans» for å finne korteste vei fra startnode til sluttnode. Ser på summen av avstand fra startnode, og estimerer avstand til målnoden når den velger neste node. (Må ha et estimat som aldri blir for høyt). Et slikt estimat baseres gjerne på rettlinjet avstand fra noden til målet og krever dermed informasjon om nodens plassering på kartet. Dette spisser søket; det får form av en ellipse som peker mot målet. Finner en vei raskere enn Djikstra, men lønner seg mindre til å finne andre alternative veier. Søker færre noder enn Djikstra.
s ut av køen, med prioritet $0 + 18 = 18$. Setter avstand:prio $a = 11 : 32$, $b = 10 : 35$ og $c = 15 : 20$. c ut av køen, med prioritet 20. Setter avstand:prio $m = 20 : 20$. m ut av køen, med prioritet 20. Korteste vei fra s til m er funnet.
Bellman-Ford:
- minner om Djikstra, men kan brukes med negative verdier på kantene.
Prims:
- nesten identisk med Djikstras. Bruker minimalt spenntre, ved å koble på en og en node basert på en PQ. Prims ser på avstand fra nærmeste node i spenntreet mens Djikstra ser på avstand fra en startnode.
Kruskals:
- ser på hver node som et frittsående tre. Velger laveste vektede kanter og slår sammen trærne. Bruker minimalt spenntre.
Maksimum flyt:
- kantene har en maks kapasitet. Kan ikke overskride denne. Tenk vannrør – større tall jo mer plass til vann i røret. Må ha konstant flyt gjennom hele. Helst det du starter med i kilden skal du få med ut til sluket!
- husk å hele tiden oppdater kapasiteten til kant (rør)
- symmetri: hvis vi sender noe tilbake kan vi trekke fra den verdien og får denne verdien motsatt vei. Nodene har ingen lagringsplass. Kan sende inn så mye vi vil så lenge ingen grenser overtredes. Ingen verdier kan blir liggende igjen.
- kansellering: få maks verdi i S (sluk). Kan gjøres ved å utnytte alle kantene på ulike måter / endre flyten til å utnytte andre kanter.
Maksimum – Ford Fulkerson:
- sier noe om måten du oppnår maksimal flyt -> kan andvende ulike teknikker som DFS eller BFS
- kan ta lang tid
- ser på restnettet (bare de tilgjengelige verdiene). Øker flyten så mye som mulig på veien med minst mulig tilgjenglighet. Dette gjentas til alle mulige veier er makset, uten å overskrivegrenser eller etterlate verdier fra K (kilde)
Maksimum – Edmond Karp:
- nesten som FF, men tar utgangspunk i korteste vei først og bruker kun BFS. Ser på restnettet og øker flyten så mye som mylig på den korteste veien. Dette gjentas til alle mulige veier er makset. — Kompleksitet O(NK^2)
Grådige algoritmer:
tar grådige valg. Tenker ikke framover, emn heller «hva er best akkurat nå». Går aldri tilbake på et valg. Rask og effektiv hvis det fungerer på problemet.
-> spenntrær: Prism og Kruskals
-> korteste vei: Djikstra
-> kompresjon: Huffman
Huffman:
- brukes i datakompresjon -> teller opp hvor mange ganger tegn forekommer (frekvens):
- tegn med høy frekvens får kort kode (få bits)
- algoritme: lag en tre-node for hvert tegn, som har tegnets frekvens som verdi. Slå sammen de to som har lavest frekvens til et tre:
-> dette er det GRÅDIGE VALGET. Gjøres ved å gi dem en felles rotnode. Den nye noden får verdi lik summen av de to nodene.
-> til slutt samlet i et binærtre -> Huffmantre
- prefiksfri -> en kort tegnkode aldri starten på en lengre kode.



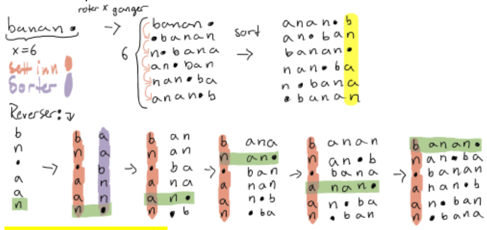
Traversering av grafer/ grafalgoritmer:
Bredde først søk (BFS):
- ønsker korteste vei fra en startnode s, til alle andre noder
- finner først alle naboer til s, deretter naboers naboer osv.
-> husker veien fra s og utvoer ved at hver node har en forgjenger
- ved bruk av naboliste: for hver node vi finner må vi sjekke alle dens kanter. Hvis grafen henger sammen, alle N nodene-> O(N + K)
-Nabotabell: For hver node må vi sjekke N mulige kanter = O(N^2)
Dybde først søk (DFS):
- prøver å følge én vei så langt som mulig. Hvis det ikke går, gå tilbake til forrige node og prøv andre veier ut.
- merker nodene som «besøkt» og besøker ikke samme flere ganger. Opererer rekursift; hvis en node ikke er besøkt, merk den og kjør DFS på dens naboer.
- Naboliste: O (N + K),Nabotabell: O(N^2)
Topologisk sortering (TS):
- aktuelt når vi planlegger aktiviteter som avhenger av hverandre, dvs. noen må være ferdig før andre kan begynne (f.eks. studieplanlegging)
- betingelsene kan representers med en rettet graf med én node per aktivitet. TS går ut på å ta en slik graf og sette nodene i en mulig rekkefølge (kan være flere riktige)
-> grafen må være asyklisk (kun én vei mellom noder!)
- samme tankegang som DFS:
-> starter DFS i tilfeldi node og hver gang DFS er ferdig med en node, lenkes den inn forrerst i resultatlista
-> så lenge det er urørte noder igjen, starter vi ny DFS og får så samme kompleksitet som DFS. - Naboliste: O (N + K),Nabotabell: O(N^2)
Sterkt sammenhengende komponenter:
- def.: en rettet graf, eller en del av en slik, er sterkt sammenhengende hvis det finnes vei fra enhver node til enhver annen. Dette må gjelde begge veier, vi trenger både a -> b og b -> a. Topologisk sortert-ingen rundturer-hver node egen komponent
- finnes slik: kjør et eller flere DFS til alle noder er funnet
- bygg opp liste, så de siste ferdige nodene kommer først
- lag den omvendte grafen (samme noder, snu kanter)
- kjør en serie DFS på den omvendte grafen, bruk ferdig-listen som startliste. De ulike DFS-trærne blir sterkt sammenhengende komponenter
- ved bruk av naboliste: Θ(N+K)
Tekstsøk og datakompresjon
Boyer-Moore:
- se på siste tegn i søketeksten først -> hvis det passer, flytt søketesksten så laaaangt vi kan:

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

feil i *kursiv* og tegn som passer i **fet skrift**
Regelen om upassende tegn:
- hvis tegnet ikke fins i søketeksten -> flytt m steg fram. Hvis tegnet fins til venstre i søkeordet, kan vi flytte ordet så det passer med teksten -> flytte m steg: med mismatch på nestsiste flytt m-1 osv.
- trenger en todimensjonal tabell: **1.** en indeks er det upassende tegnet. **2.** den andre indeksen er posisjonen i søketeksten. Verdien i cellen er hvor langt vi kan flytte framover:

	m	e	t	e	o	r	i	t	t	s	t	e	i	n
0	m	e	*n*	e										
1				m	e	n	*e*							

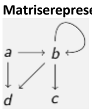
Lempel-Ziv Welsh:
- søker bakover i sirkulært buffer. Leser ett og ett tegn. Bygger ordliste underveis (starter med 1-byte «ord»).
Burrows Wheeler transformasjonen (BWT):
- ikke komprimering i denne algoritmen, men transformerer en blokk
- transformerer repeterte sekvenser til repeterte tegn og repeterte tegn er lettere å komprimvere videre



Grafer, relasjoner og språk:
Isomorfi: G er isomorf på G' hvis det finnes en bijeksjon g: V(G) -> V(G') og h: E(G) -> E(G') som bevarer kant-hjørne-funksjonene til G og G' slik at for alle v element i V(G) og alle e element i E(G) er v et endepunk for e <-> g(v) er et endepunkt for h(e).
- isomorfiinvarianter: en egenskap P kalles isomorfiinvariant hvor G har egenskap P og G' er isomorf med G, så har G' denne egenskapen P. Hvis to grafer ikke deler en isomorfiinvarient er de ikke isomorfe!
1. å ha n kanter. **2.** å ha m hjørner. **3.** å ha et hjørne av grad k. **4.** å ha m hjørner av grad k. **5.** å ha krets av lengde k. **6.** å ha en simpel krets av lengde k. **7.** å ha m kretser av lengde k. **8.** å være sammenhengende. **9.** å ha en Eulerkrets. **10.** å ha en Hamiltonkrets.
-> ikke isomorfe = kontrapositiven. Altså hvis to grafer ikke deler en isomorfiinvariant.
Graf med hjørner v og w i G:
- en *vei* er en endelig alternerende rekke av kanter og hjørner
- et *spor* er en vei fra v til w som ikke repeterer noen kant
- en *sti* er et spor som ikke repetetet noe hjørne
- en *rundtur/lukket vei* er en vei som starter og slutter i samme node
- en *krets* er en lukket vei som ikke repeterer noen kant
- en *simpel krets* er en krets som ikke repeterer hjørne bortsett fra første og siste node

En **komplett graf** K_n **med** n **hjørner** der n er et positivt heltall, er en simpel graf med n hjørner og nøyaktig en kant mellom hvert par av hjørner.

-> komplett graf: hver node har grad n-1. Total grad: n * (n-1)
- sammenhengende grafer: for hvert par av hjørner i grafen finnes en STI mellom dem. Hvis v og w er en del av en krets i G og en kant fjernes fra kretsen, finnes ennå et spor fra v til w. Hvis G er sammenhengende og har en ikke triviel krets, kan en kant fjernes uten at G blir usammenhengende.
- Eulerkrets: En graf G har en Eulerkrets hvis og bare hvis G er sammenhengende og ethvert hjørne i G har partallsgrad
- Eulerspor: G være en graf, og v og w to noder i G. G har et Eulerspor v-w når G er sammenhengende, v og w odde grad, og alle andre noder partallsgrad
- Hamiltonkrets: simpel krets som inneholder alle noder. Samme antall noder og kanter, og ethvert hjørne har partallsgrad.
Matriserepresentasjoner av rettede grafer:



fra hjørne denne siden

$$\begin{array}{c|cccc} & a & b & c & d \\ \hline a & 0 & 1 & 0 & 1 \\ b & 0 & 1 & 1 & 1 \\ c & 0 & 0 & 0 & 0 \\ d & 0 & 0 & 0 & 0 \end{array} = M_G$$

- for urettede grafer vil du måtte henvise til begge veier!
- finne veier av lengde 2: gang med seg selv en gang (rad * hver kollonne). Antall enere i resultat blir veier av lengde to i grafen G
Trær:
- kretsfritt og sammenhengende.
- for ethvert positivt heltall n, har et tre med n hjørner n-1 kanter
- for ethvert positivt heltall n, så hvis G er en sammenhengende graf med n hjørner og n-1 kanter, er G et tre
- rottrær: et tre der et hjørne kalles roten. Nivået til et hjørne i et trottre er antall kanter langs den unike stien mellom det og roten
- binærtrær: et rottre der enhver forelder har maksimalt to barn.
-> hvis k er et positivt heltall og T er et fullt binærtre med k grenhjørner (foreldre), har T 2k + 1 hjørner og k+1 løvhjørner.
-> Hvis T er et binært tre med t løvhjørner og høyde h, så er t <= 2^h. Finnes det er binært tre med høyde 5 og 38 løvhjørner? 2^5 = 32 løvhjørner så nei t er kan ikke være 38.
Relasjoner:

Gitt $A = \{1, 2, 3\}$ *og* $B = \{0, 1, 2, 4\}$. *Da kan vi definere relasjonen* $\leqslant$ *på dem som følger:*

$$\leqslant = \{(1, 1), (1, 2), (1, 4), (2, 2), (2, 4), (3, 4)\}.$$

Definisjon (Funksjon)

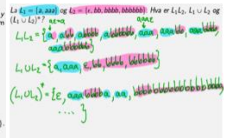
En **funksjon fra en mengde** A **til en mengde** B er en relasjon f fra A til B slik at
1. For alle $x \in A$ finnes en $y \in B$ slik at $(x, y) \in f$.
2. For alle $x \in A$, $y, z \in B$ så hvis $(x, y) \in f$ og $(x, z) \in f$, så er $y = z$.

- ekvivalensrelasjon: R være binærrelasjon på en mengde A
-> refleksiv: R er refleksiv hvis for alle x ∈ A så er (x, x) ∈ R
-> symmetrisk: R er symmetrisk hvis for alle x, y ∈ A så hvis (x, y) ∈ R er også (y, x) ∈ R
-> transitiv: R er transitiv hvis for alle x, y, z ∈ A så hvis (x, y) ∈ R og (y, z) ∈ R så er (x, z) ∈ R
- transitiv tillukning av en relasjon: gjøre en relasjon transitiv
-> ekvivalensklasser:
- mengder: har en mengde A {r,s,t,v,u}. For å vise en oppdeling av A med ekvivalensrelasjon kan det ikke være overlapp av elementer -> Kan ha A_1 {r,s} og A_2 {t,v}, men ikke da A_3 {v,u}. Kan heller ikke mangle ett element fra mengden A.
- partielle ordninger: må være antisymmetrisk, refleksiv og transitiv.
-> antisymmetrisk: la R være en relasjon på mengde A. Da sier vi at R er antisymmetrisk hvis for alle a og b i A, *hvis* a R b og b R a så er a = b. R {(0,0), (0,1), (0,2), (1,1), (1,2)} er antisymmetrisk
- leksikografisk ordning: tenke alfabetisk rekkefølge fra relasjon-graf
- Hassediagram: knyttet til partiell ordning. Forenkling av grafen til relasjonen. 1. Fjern alle looper. 2. Plassér hjørnene slik at alle piler peker oppover. 3. Fjern alle piler som følger av transitivitet. 4. Fjern retningen på alle piler. -> sørtst, minst, minimale og maksimale elmt.
- kjeder: lengde til kjede hvor B delmengde i A er elementer i B-1
Språk og regulære uttrykk:

La Σ være et endelig alfabet. Gitt strenger x og y over Σ er **sammenlenkingen av** x **og** y strengen du får når du setter tegnene i y etter de i x. Videre, gitt to språk L og L', definerer vi tre nye språk som følger:
- **Sammenlenkingen av** L **og** L', gitt ved
$$LL' = \{xy | x \in L, y \in L'\}.$$
- **Unionen av** L **og** L', gitt ved
$$L \cup L' = \{x | x \in L \text{ eller } x \in L'\}.$$
- **Kleenetillukningen av** L, gitt ved
$$L^* = \{x | x \text{ er en sammenlenking av et endelig antall strenger i } L\}.$$

Merk at $\epsilon \in L^*$.

La $L_1 = \{a, aaa\}$ og $L_2 = \{\epsilon, bb, bbbb, bbbbbb\}$. Hva er L_1L_2, $L_1 \cup L_2$ og $(L_1 \cup L_2)^*$?

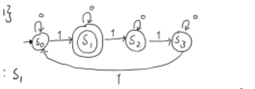


For et endelig alfabet Σ, har vi en funksjon L som assosierer ethvert regulært uttrykk r med et språk over Σ. Da kaller vi $L(r)$ for **språket definert av** r, og definerer det som følger:
(1) $L(\varnothing) = \varnothing$, $L(\epsilon) = \{\epsilon\}$, $L(a) = \{a\}$ for hver $a \in \Sigma$.
(2) Hvis $L(r)$ og $L(s)$ er språkene definert av de regulære uttrykkene r og s over Σ, så er
(i) $L(rs) = L(r)L(s)$
(ii) $L(r|s) = L(r) \cup L(s)$
(iii) $L(r^*) = (L(r))^*$.

Odde 1ere: 0*10*(10*10*)*, ant 1 delelig 3: (0|2)*(111)*(0|2)*, aldri a etter c: (a|b)*(b|c)*, max 2 y: x*(y|yx*y| ϵ)x*, partall y max 1 x: (yy)*(x|yxy| ϵ)(yy)*
- automater:

		0	1
→	s_0	s_0	s_1
◎	s_1	s_1	s_2
	s_2	s_2	s_3
	s_3	s_3	s_0

Inputalfabet: {0,1}
starttilstand: s_0
$S = \{s_0, s_1, s_2, s_3\}$
Aksept tilstilstand: s_1



- kontekstfrie språk: en grammatikk. Strenger S laget etter regler. Holder på helt til vi har en streng uten midlertidige symboler.
Beskriv en måte å generere strengen *babbab* på i grammatikken G gitt under:

$$V = \{a, b, S, A\}, \Sigma = \{a, b\}, R = \{\underbrace{S \to AA}_{1}, \underbrace{A \to AAA}_{2}, \underbrace{A \to a}_{3}, \underbrace{A \to bA}_{4}, \underbrace{A \to Ab}_{5}\}$$

$$S \to AA \xrightarrow{4} bAA \xrightarrow{3} baA \xrightarrow{4} babA \xrightarrow{4} babbA \xrightarrow{5} babbAb \xrightarrow{3} babbab$$

Annet:
- **P:** mengden av alle problemer som kan løses i polynomisk tid.O(1), O(n) osv.
- **NP**: mengden av alle problemer der svaret kan sjekkes i polynomsik tid.
- **NP-komplette:** problemer er løsbare, og løsninger kan sjekkes i polynomisk tid. Alle NP-problem kan omformes (reduseres) til et NPC-problem i polynomisk tid.
- **NP-hard**: mengden av alle problemer som er slik at det finnes et NPC-problem som kan omformes til dette problemet i polynomisk tid. NP- hardt problem er det ikke nødvendigvis mulig å sjekke løsninger i polynomisk tid
- eks.: anta at du har n positive og negative heltall og skal finne et utvalg (en dekmengde) av de som gir sum lik 0. I hvilken mengde gitt over vil du plassere dette problemet? NP-komplett. Med n tall finnes det 2^n mulige delsummer som må sjekkes. Men når et svar er funnet, kan det sjekkes på O(n) tid.
Gi eksempel på ett NP-hardt problem, og ett NP-komplett problem.

Vi har sett på mange problemer, Noen har både en NP-hard og en NP-komplett variant. Eksempler som kan brukes:

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	