

Overordnede læringsmål

Kunnskap om:

- [X1] Et bredt spekter av etablerte algoritmer og datastrukturer
- [X2] Klassiske algoritmiske problemer med kjente effektive løsninger
- [X3] Komplekse problemer uten kjente effektive løsninger

Skal kunne:

- [X4] Analysere algoritmers korrekthet og effektivitet
- [X5] Formulere problemer så de kan løses av algoritmer
- [X6] Konstruere nye effektive algoritmer

Skal være i stand til:

- [X7] Å bruke eksisterende algoritmer og programvare på nye problemer
- [X8] Å utvikle nye løsninger på praktiske algoritmiske problemstillinger

Forkunnskaper

- [Y1] Kjenne til begreper rundt monotone funksjoner
 - [Y2] Kjenne til notasjonene $\text{roof}(x)$, $\text{floor}(x)$, $n!$ og $a \bmod n$
 - [Y3] Vite hva polynomer er
 - [Y4] Kjenne grunnleggende potensregning
 - [Y5] Ha noe kunnskap om grenseverdier
 - [Y6] Være godt kjent med logaritmer med ulike grunntall
 - [Y7] Kjenne enkel sannsynlighetsregning, indikatorvariable og forventning
 - [Y8] Ha noe kjennskap til rekkesummer
 - [Y9] Beherske helt grunnleggende mengdelære
 - [Y10] Kjenne grunnleggende terminologi for relasjoner, ordninger og funksjoner
 - [Y11] Kjenne grunnleggende terminologi for egenskaper ved grafer og trær
 - [Y12] Kjenne til enkel kombinatorikk, som permutasjoner og kombinasjoner
- Se på kap 3.2, kap 5 (s.118-120), App. A (s.1145-1146 og likning A.5), App. B, App. C (s.1183-1185, C2 og s.1196-1198)

Gjennom semesteret

Læringsmål for hver algoritme

- [Z1] Kjenne den formelle definisjonen av det generelle problemet den løser
- [Z2] Kjenne til eventuelle tilleggskrav den stiller for å være korrekt
- [Z3] Vite hvordan den oppfører seg; kunne utføre algoritmen, trinn for trinn
- [Z4] Forstå korrekthetsbeviset; hvordan og hvorfor virker algoritmen egentlig?
- [Z5] Kjenne til eventuelle styrker eller svakheter, sammenlignet med andre
- [Z6] Kjenne kjøretiden under ulike omstendigheter, og forstå utregningen

Læringsmål for hver datastruktur

- [Z7] Forstå algoritmene (se over) for de ulike operasjonene på strukturen
- [Z8] Forstå hvordan strukturen representeres i minnet

Læringsmål for hvert problem

[Z9] Kunne angi presist hva input er

[Z10] Kunne angi presist hva output er og hvilke egenskaper det må ha

Læringsmål

1 Problemer og algoritmer

Vi starter med fagfeltets grunnleggende byggesteiner, og skisser et rammeverk for å tilegne seg resten av stoffet. Spesielt viktig er ideen bak induksjon og rekursjon: Vi trenger bare se på siste trinn, og kan anta at resten er på plass.

[A1] Forstå bokas pseudokode-konvensjoner

- En måte å spesifisere algoritmer på, uavhengig av programmeringsspråk

[A2] Kjenne egenskapene til random-access machine-modellen (RAM)

- Instruksjoner er utført en etter en, ingen samtidige operasjoner.
- Modellen som brukes i boka når vi ser på kode.

[A3] Kunne definere problem, instans og problemstørrelse

- *Problem:*
 - Relasjonen mellom input og output. Blir løst av en algoritme. Når man definerer problemet, må man beskrive hvilket forhold man ønsker mellom input og output.
- *Instans*
 - En subklasse av problemet
 - En bestemt input
- *Problemstørrelse:*
 - Lagringsplass som trengs for en instans.

[A4] Kunne definere asymptotisk notasjon; O , Ω , Θ , lille O og lille θ (!)

- *Stor- O :* gir en asymptotisk øvre grense for en funksjon ved hjelp av en enklere funksjon. " \leq "
- *Stor- Ω :* gir en asymptotisk nedre grense for en funksjon ved hjelp av en enklere funksjon. " \geq "
- *Stor- Θ :* brukes når en funksjon er både store- O og store- Ω . Det vil si at for store x , så er funksjonen "klemt" mellom to lineære funksjoner. " $=$ "
- *Lille O :* streng mindre enn. "<"
- *Lille Ω (ω):* strengt større enn. ">"

- - $\omega > \Theta(f(n))$ (Lille Omega)
 - $\Omega \geq \Theta(f(n))$ (Store Omega)
 - $\Theta = \Theta(f(n))$ (Store Theta)
 - $O \leq \Theta(f(n))$ (Store O)
 - $o < \Theta(f(n))$ (Lille o)

- *Kompleksitetsregler:*
 - $1 < \ln(n) < n < n^k < k^n < n! < n^n$, der k = en konstant
 - Grønn farge: polynomisk kjøretid
 - Rød farge: Eksponentiell kjøretid

[A5] Kunne definere best-case, average-case og worst-case (!)

- *Kjøretid:* funksjon av problemstørrelse $\rightarrow f(\text{problemstørrelse})$
- *Best-case:* best mulig kjøretid for en gitt størrelse
- *Average-case:* Snittet av kjøretiden til alle instansene. “forventet” kjøretid.
- *Worst-case:* gir en øvre grense på kjøretiden for alle input. Garanterer at algoritmen ikke vil bruke lengre tid. Mest brukt.

[A6] Forstå løkkeinvarianter og induksjon (!)

- *Løkkeinvarianter:*
 - Brukes til bevis for løkker. Må vise tre ting!
 - Vet at feks alt til venstre for et gjeldende tall i er riktig sortert i en liste.
 - Init: Vise at den er sann før første iterasjon.
 - Vedlikehold: Hvis den er sann før en iterasjon av en løkke, er den sann før neste iterasjon også.
 - Terminering: Når løkka er ferdig, gir invarianten oss en brukbar egenskap som hjelper oss å vise at algoritmen er sann.

[A7] Forstå rekursiv dekomponering og induksjon over delproblemer (!)

- Løser en liten del av problemet og sender problemet videre

[A8] Forstå INSERTION-SORT

- Input
 - En sekvens med tall
- Output
 - En sortert sekvens av samme tall
- Info
 - Sjekker gjeldende tall(nøkkelen) og setter det inn der det skal til venstre i lista (som er sortert). Effektiv for å sortere en liten mengde elementer.
- Egenskaper:
 - Sammenligningsbasert
 - In-place: Bytter på to og to elementer
 - Stabil: Vil aldri flytte to like elementer forbi hverandre, uansett om man starter foran eller bak
- Kjøretid:
 - Best case: $O(n)$
 - Average case: $O(n^2)$
 - Worst case: $O(n^2)$
- Figur/Kode:

```
def insertion_sort(A):
    for j in range(1, len(A)):
        key = A[j]

        # Plasserer A[j] inn i den sorterte sublisten [0..j-1]
        i = j-1
        while i >= 0 and A[i] > key:

            # Flytter hvert element en til høyre, så lenge key < A[i]
            A[i+1] = A[i]
            i -= 1

        # Plasserer key på riktig plass
        A[i+1] = key
```

2 Datastrukturer

For å unngå grunnleggende kjøretidsfeller er det viktig å kunne organisere og strukturere data fornuftig. Her skal vi se på hvordan enkle strukturer kan implementeres i praksis, og hva vi vinner på å bruke dem i algoritmene våre.

[B1] Forstå hvordan stakker og køer fungerer

- **Stakker:** last in - first out (LIFO)
 - Som tallerkener stablet oppå hverandre.
 - Stack-empty: sjekker om $S.top == 0$; da er stacken tom
 - Push(S, x): legge til element på toppen. Hvis top er satt til feks indeks 4, og man pusher x , blir elementet som før var på plass 4, flyttet til plass 5.
 - Pop: fjerner øverste element. Topp blir da satt ned -1 plass.
 - 4. La tabellen $S[1..10]$ være $\langle 50, 70, 45, 15, 72, 41, 61, 22, 26, 64 \rangle$ og la $S.top = 5$. Utfør så følgende utsagn, i rekkefølge:


```
1  x = Pop(S)
2  y = Pop(S)
3  Push(S, x)
4  Push(S, y)
```


Hva er innholdet i tabellen S nå?

 - **Merk:** Det spørres her om innholdet i *hele* tabellen, ikke bare i stakken.
 - Svar:** $S = \langle 50, 70, 45, 72, 15, 41, 61, 22, 26, 64 \rangle$
 - Forklaring:** Her er $x = 72$ og $y = 15$. Pop returnerer elementet $S[S.top]$ og dekrementerer $S.top$, men endrer ikke innholdet i S . Push inkrementerer først $S.top$, og setter så inn den angitte verdien som nytt element $S[S.top]$. Effekten her er altså at elementene på posisjon 4 og 5 bytter plass.
- **Køer:** first in - first out (FIFO)
 - Køen går i sirkel
 - Enqueue: legge til element i slutten (tail)
 - Settes inn på $Q.tail, Q.tail+1$ etterpå.
 - Dequeue: henter ut/fjerner element fra starten (head)
 - Henter ut $Q.head, Q.head+1$ etterpå.

[B2] Forstå hvordan lenkede lister fungerer (*LIST-SEARCH, LIST-INSERT, LIST-DELETE, LIST-DELETE', LIST-SEARCH', LIST-INSERT'*)

- Datastruktur hvor objektene blir ordnet i lineær rekkefølge. Rekkefølgen blir bestemt av en peker i hvert objekt. Tanken bak strukturen er at rekkefølgen på elementer opprettholdes ved at hvert element peker til det neste i rekkefølgen.
- Enkle lenkede lister - peker på neste element
- Dobble lenkede lister - peker på neste og forrige element
- Sykliske lenkede lister - previous.head peker på tail, og next.tail peker på head.
- Hvis sortert - minste elementet er head, største er tail
- *List-search:*
 - Finner første element med key k i L og returnerer posisjonen. Worst case $\Theta(n)$, hvis man må søke gjennom hele lista.
- *List-insert:*
 - Gitt et element x hvor key allerede er satt, vil x bli satt inn i front av listen. $O(1)$.
 - Innsetting på slutten $O(n)$.
- *List-delete:*
 - Fjerner et element x fra L . Må gi en peker til x , og så skjøtes x ut av lista ved å oppdatere pekere.
 - For å slette et element med gitt key, må vi først bruke List-search for å finne pekeren.
 - Oppslagstid + $O(1) = O(n)$
- Alternative utgaver av disse (List-insert', list-search', list-delete') bruker noe som kalles sentinels. Dette skal forenkle koden, fordi man ser bort i fra spesialtilfellene ved starten og slutten i koden. Man bruker en $L.Nil$ på startplassen, og bare lar denne være ingenting.

[B3] Forstå hvordan pekere og objekter kan implementeres

- Omtrent som lenkede lister. Peker mot en adresse i minnet som om det var et annet objekt i den adressen.
-  d) Vi ønsker å implementere L som et flerarray av objekter tilsvarende som i Cormen (figur 10.5 s.242). Hvilket av alternativene under for startvariabel L og arrayene $N = \text{next}$, $K = \text{key}$ og $P = \text{prev}$ er korrekt implementert? (5 %)
- ☐ $L = 2$, $N = \langle /, 6, 1, 3, 0, 4 \rangle$, $K = \langle 3, 4, 72, 32, 0, 7 \rangle$, $P = \langle 3, /, 4, 1, 0, 2 \rangle$
- ☐ $L = 4$, $N = \langle 8, 6, /, 1, 0, 0, 0, 2 \rangle$, $K = \langle 7, 32, 72, 4, 0, 0, 0, 3 \rangle$, $P = \langle 4, 8, 2, /, 0, 0, 0, 1 \rangle$
- ☐ $L = 1$, $N = \langle 2, 3, 4, 5, / \rangle$, $K = \langle 4, 7, 32, 72, 3 \rangle$, $P = \langle 1, 2, 3, 4, / \rangle$
- ☒ $L = 7$, $N = \langle 3, 0, 4, /, 0, 1, 6, 0 \rangle$, $K = \langle 32, 0, 72, 3, 0, 7, 4, 0 \rangle$, $P = \langle 6, 0, 1, 3, 0, 7, /, 0 \rangle$

[B4] Forstå hvordan direkte adressering og hashtabeller fungerer (!)

(HASH-INSERT, HASH-SEARCH)

- *Direkte adressering:*
 - Fungerer bra når antall mulige nøkler er rimelig lavt.
 - Bruker nøkkel direkte som indeks. Triviell form for hashing.
 - Kun dersom man ikke kan få kollisjoner, dvs vi antar at ingen har samme nøkkel
 - Tabellen blir laget slik at på $T[\text{key}]$ står verdien til key
 - Alle operasjoner tar $O(1)$

- *Hashtabeller:*
 - Genererer en indeks fra nøkkelverdien. Enveisfunksjon - går ikke tilbake!
 - Hvis antall nøkler er stor, så er dette mer effektivt enn direkte adresserings tabeller
 - Bruker hash-funksjon for nøklene, $h(k)$.

[B5] Forstå konfliktløsning ved kjeding (chaining) (*CHAINED-HASH-INSERT*, *CHAINED-HASH-SEARCH*, *CHAINED-HASH-DELETE*)

- $x.key = m$ og $h(m) = j \rightarrow x$ elementet, m nøkkelen og j hashen
- Hvis flere nøkler hashes til samme posisjon, får vi en kollisjon.
- Løser dette ved kjeding: alle elementer som hasher til samme posisjon blir puttet inn i en lenket liste.
- *CHAINED-HASH-INSERT*
 - Setter inn x i “hodet” til lista $T[h(x.key)]$. Worst case $O(1)$
 - Spesialtilfelle: Hvis man må sjekke om elementet allerede er i lista blir det $O(n)$ fordi man må gå gjennom lista først.
- *CHAINED-HASH-SEARCH*
 - Söker etter et element med nøkkel k i lista.
- *CHAINED-HASH-DELETE*
 - Sletter x fra lista. Gjøres i $O(1)$ tid hvis lista er dobbel lenket.
 - Hvis enkel lenket liste, må man gå gjennom lista først, for å lage *next* verdien til forgjengeren der vi skal fjerne.

[B6] Kjenne til grunnleggende hashfunksjoner

- Dårlig hashfunksjon
 - Alle nøklene peker til samme plass, og vi får en n lang liste på denne plassen. Søk er da $\Theta(n)$.
- God hashfunksjon
 - Hver nøkkel har like stor sannsynlighet for å hashe til hvilken som helst plass, uavhengig av hvor andre nøkler har hashet til. Har som regel ingen måte å sjekke at dette blir gjort riktig.
- Hvordan lage hashfunksjon
 - Divisjonsmetoden:
 - $h(k) = k \bmod m$
 - Unngå at m er “power of 2” (2,4,8,16...)
 - Et primtall som ikke er for nærme en tall som er 2-er potens er ofte et bra valg.
 - Multiplikasjonsmetoden
 - $h(k) = \text{floor}[m (kA \bmod 1)]$ der A er en konstant mellom 0 og 1.
 - Fordel her er at verdien til m ikke er kritisk
- Kjente hashfunksjoner og hvordan det brukes:
 - Lengde, ASCII-verdier \leftarrow sortering
 - RSA, DSA \leftarrow kryptografi

[B7] Vite at man for statiske datasett kan ha worst-case $O(1)$ for søk

- Hvis du vet på forhånd hva alle elementene er, kan man velge en hashfunksjon som gjør at man har worst-case $O(1)$

[B8] Kunne definere amortisert analyse

- I en amortisert analyse regner vi ut den gjennomsnittlige tiden for å utføre en sekvens av datastrukturoperasjoner over alle operasjonene som ble utført.
- Med amortisert analyse kan vi vise at gj.snittskostnaden for en operasjon er liten, hvis vi regner snittet over en sekvens av operasjoner, selv om en enkelt operasjon av sekvensen kan være dyr.
- Ulikt fra average case fordi sannsynlighet ikke er involvert.
- Garanterer “the average performance of each operation in the worst case”, derfor kan det være bedre enn worst-case i mange tilfeller, fordi worst case er for pessimistisk.

[B9] Forstå hvordan dynamiske tabeller fungerer (*TABLE-INSERT*)

- Øker eller minker en tabell. Hvis man får flere elementer inn i tabellen enn det er plasser, vil man allokere en ny og større tabell. Vanlig å gjøre den dobbelt så stor som den forrige. Kopierer over den gamle lista.
- Load-factor α til en ikke-tom tabell er antall elementer lagret i tabellen delt på størrelsen (antall plasser) til tabellen. Tabell full hvis load-factor $\alpha = 1$
- Bruker $O(1)$ ved innsetting, men når tabellen er full vil man bruke lengre tid fordi man lager en ny liste. Tiden man bruker på dette er likevel mindre “overall” enn å skulle bruke lenger, men samme, tid på innsetting hele tiden.
 - Når en tabell er full, kan man ikke bare definere at lengden skal være lenger, man må faktisk finne plass i minnet.
 - Har man en tabell av lengde n , allokere man en ny tabell av lengde $2n$ og kopierer de n første elementene over (med for-løkke f.eks, tar lineær tid).
 - Innsetning i ny tabell tar konstant tid, men reallokeringen av ny tabell tar lineær tid.
 - Hvis vi bare hadde reallokert til en tabell av størrelse $n+1$, ville innsettingen tatt lineær tid hver gang og total kjøretid ville blitt $1+2+3+\dots$ aka kvadratisk.
 - Får i stedet en total kjøretid på $1+2+4+\dots$, lineær, og gj.snittet per operasjon er konstant.
 - Viktig å huske at $1+2+3+\dots+n-1 = n(n-1)/2 = \Theta(n^2)$
og at $1+2+3+\dots+n/2 = n-1 = \Theta(n)$
- Amortisert analyse, fordi totale kjøretiden blir mindre, selv om enkelte operasjoner (lage ny tabell) tar lenger tid
- TABLE-INSERT
 - Beskrevet grovt over, altså konseptet med å legge til element hvis plass, eller allokere ny tabell $2n$ hvis fullt og kopiere over forrige tall.
 - Amortisert kjøretid $O(1)$

3 Splitt og hersk

Rekursiv dekomponering er kanskje den viktigste ideen i hele faget, og designmetoden splitt og hersk er en grunnleggende utgave av det: Del instansen i mindre biter, løs problemet rekursivt for disse, og kombinér løsningene

[C1] Forstå designmetoden divide-and-conquer (splitt og hersk) (!)

- Divide: dele problemet inn i en rekke delproblemer som er mindre instanser av samme problem
- Conquer: “herske” delproblemene ved å løse de rekursivt
- Combine: kombinerer løsningene til delproblemene inn i løsningen for det originale problemet

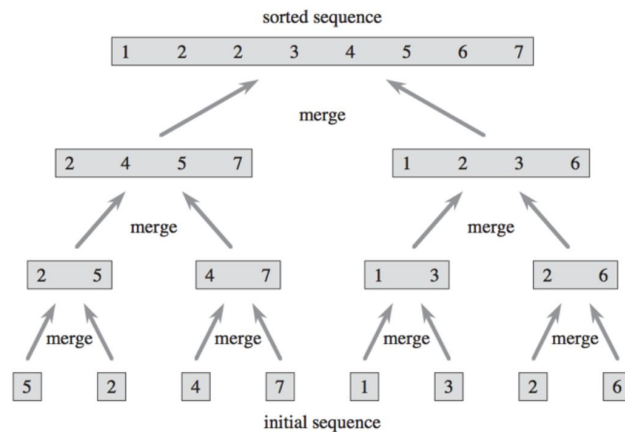
[C2] Forstå BISECT og BISECT' (se appendiks)

- BISECT
 - Binærsøk, bruker splitt og hersk ved å dele opp en sortert liste i to og lete etter tallet rekursivt
 - Sender inn et tall og finner hvor i lista det forekommer. Hvis det ikke forekommer, returneres plassen det ville forekommet i en sortert liste.
 - Hvis elementet forekommer flere ganger, returneres det lengst til venstre.
 - Kjøretid $\Theta(\lg n)$
 - Antall “spørsmål”: $\lg n + 1$
- BISECT'
 - Iterativ utgave, der man kjører while-løkke. Generelt mer effektiv fordi man slipper funksjonskall.

[C3] Forstå MERGE-SORT

- Input
 - En usortert liste
- Output
 - Sortert liste
- Info
 - En splitt og hersk algoritme.
 - Splitter på midten rekursivt til vi har enkeltstående tall (som i seg selv er sortert siden det bare er ett tall). $\Theta(\lg n)$ tid.
 - Sorterer delsekvensene rekursivt ved å bruke merge sort flere ganger. Sammenligner to og to sekvenser.
 - Slår sammen to sorterte delsekvenser for å danne det sorterte svaret.
 - Denne MERGE delen tar $\Theta(n)$ tid fordi man går gjennom lista én gang.
- Egenskaper
 - Sammenligningsbasert
 - Splitt og Hersk
 - Stabil - Bare hvis den velger elementer fra venstre halvdel om elementene er like
- Kjøretid
 - Worst case: $\Theta(n \lg n)$

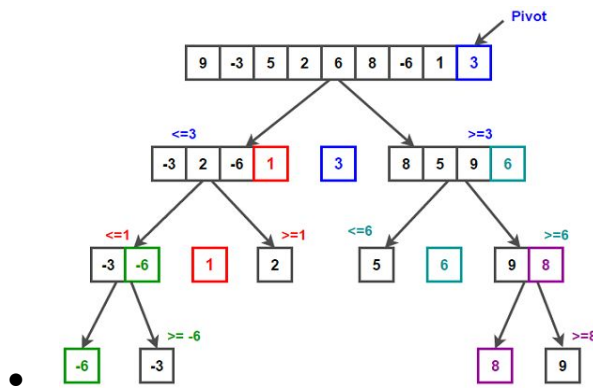
- Kan bevises ved masterteoremet
- Figur



[C4] Forstå QUICKSORT og RANDOMIZED-QUICKSORT

Quicksort:

- Input
 - Usortert liste
- Output
 - Sortert liste
- Info
 - Benytter seg også av splitt og hersk. Populær for store input
 - Velger et pivot-element (PARTITION), kan være feks første eller siste elementet i lista (siste er brukt i Cormen).
 - (1) Deler opp listen til to sublister. Den ene inneholder alle tallene som er mindre enn pivot, den andre alle tallene som er større.
 - (2) Sorterer disse to listene rekursivt vha QuickSort
 - (3) Returnerer liste1 + pivot + liste2
 - Bruker PARTITION
 - Denne velger pivot som deler listen
 - Returnerer indeks til pivot etter at lista er sortert
- Egenskaper
 - Sammenligningsbasert
 - Splitt og hersk
 - In-place
 - Den er rekursiv og “møblerer” om på elementene i returneringsfasen av algoritmen
- Kjøretid
 - Best case og Average case er $O(n \lg n)$ fordi vi gjør $\lg n$ kall til Partition, som er $\Theta(n)$.
 - Worst case forekommer når input-lista allerede er sortert/omvendt sortert (alt ettersom du velger pivot foran eller bak). Dette betyr at etter PARTITION så har den ene siden 0 elementer og den andre $n-1$ elementer. $O(n^2)$
- Figur



Randomized-quicksort:

- Samme algoritme som quicksort, men pivot velges tilfeldig fra lista vha RANDOMIZES-PARTITION.
- Gjøres for å gi færre tilfeller av worst-case.

[C5] Kunne løse rekurrenser med substitusjon, iterasjonsmetoden, rekursjonstrær og masterteoremet (!)

- Substitusjon
 - (1) Gjett formen for løsningen
 - Ved feks $T(n) = 2T(\text{floor}[n/2] + k) + n$, der k bare er et tall, vil man tippe $O(n \lg n)$
 - (2) Bruk matematisk induksjon for å finne konstantene og vis at løsningen fungerer

11. Din venn Lurvik mener følgende rekurrens har løsning $T(n) = n^3 - n^2$:

$$T(n) = \begin{cases} 0 & \text{if } n = 1, \\ 8T(n/2) + n^2 & \text{if } n > 1, \end{cases}$$

der $n = 2^k$, for et positivt heltall k . Vis at hun har rett.

Svar: Viser for eksempel ved induksjon (substitusjonsmetoden).

Grunntilfelle: $T(1) = 1^3 - 1^2 = 0$.

Induktivt trinn: Antar $T(n/2) = (n/2)^3 - (n/2)^2 = n^3/8 - n^2/4$.

Substituerer dette inn i rekurrensen: $T(n) = 8(n^3/2^3 - n^2/2^2) + n^2 = n^3 - 2n^2 + n^2 = n^3 - n^2$.

Forklaring: Svaret holder for grunntilfellet og videreføres i det induktive trinnet, og holder dermed for alle naturlige tall n . Den induktive antagelsen er at $T(m) = m^3 - m^2$, for alle positive heltall $m < n$. Mer spesifikt antar vi altså dette for $m = n/2$.

-
- Iterasjon
 - Ligner litt på rekurrenstrær, men jobber mer direkte på ligningene. Bruker rekurrensen selv til å ekspandere de rekursive termene.

$$\begin{aligned} T(0) &= 0 \\ T(n) &= T(n-1) + 1 \quad (n \geq 1) \end{aligned}$$

Her ønsker vi å bli kvitt $T(n-1)$ på høyre side i ligningen. Om vi antar at $n > 1$, så kan vi bruke definisjonen $T(n) = T(n-1) + 1$ til å finne ut hva $T(n-1)$ er, nemlig $(T((n-1)-1) + 1) + 1$ eller $T(n-2) + 2$. Det er altså én *ekspansjon* (eller iterasjon). Vi kan fortsette på samme måte, trinn for trinn, helt til vi ser et mønster dukke opp:

$$\begin{aligned} T(0) &= 0 \\ T(n) &= T(n-1) + 1 & (1) \\ &= T(n-2) + 2 & (2) \\ &= T(n-3) + 3 & (3) \\ &\vdots & \vdots \\ &= T(n-i) + i & (i) \\ &= T(n-n) + n = n & (n) \end{aligned}$$

Først ekspanderer vi altså $T(n)$, så $T(n-1)$ og så $T(n-2)$, og så videre, hele tiden ved hjelp av den opprinnelige rekurensen. Linjenummeret (til høyre) viser hvor mange ekspansjoner vi har gjort. Etter hvert klarer vi altså å uttrykke resultatet etter et vilkårlig antall ekspansjoner, i . Her kan i være 1, 2, 3, eller et hvilket som helst annet tall, så lenge vi ikke ekspanderer oss «forbi» grunntilfellet $T(0) = 0$. Og det er nettopp grunntilfellet vi ønsker å nå; vi vil gi T argumentet 0. For å få til det må vi altså få $n-i$ til å bli 0, og altså sette $i = n$. Dette gir oss svaret,

○ som vist i siste linje. Et litt mer interessant eksempel finner du i appendiks C, der

- **Rekursjonstrær**

- Hver node representerer “kostnaden” til hvert subproblem et sted i mengden av rekursive funksjonskall.
- Summerer kostnaden i hvert nivå av treet for å få en mengde av hvert nivå sin kostnad, og deretter summerer alle nivåkostnader for å finne den totale kostnaden til rekursjonen.
- Brukes mest for å få en “god gjetning”, for deretter å anvende substitusjon.

- **Masterteoremet**

- $T(n) = a \cdot T(n/b) + f(n)$, hvor n/b = delproblemstørrelse og a = antall delproblemer
- Regn ut $k = \log_b a$
- Del inn i et av tilfellene
 - Tilfelle 1: $f(n) = O(n^{k-\epsilon})$
 - $\rightarrow T(n) = \Theta(n^k)$
 - Tilfelle 2: $f(n) = \Theta(n^k)$
 - $\rightarrow T(n) = \Theta(n^k \lg n)$
 - Tilfelle 3: $f(n) = \Omega(n^{k+\epsilon})$
 - $\rightarrow T(n) = f(n)$
 - Tilfelle 4: Ingen
 - Et spesialtilfelle: $T(n) = 8T(n/2) + n^3 \lg n \rightarrow k = 3$
 - Der gjelder ikke tilfelle 1-3
- Brukes når problemet kan deles inn i to like store delproblemer. Kan kun brukes på rekurrensen på formen over.

10. Løs følgende rekurrens, der $n \geq 0$ er et heltall:

$$T(n) = \begin{cases} 0 & \text{hvis } n = 1, \\ 10\,000 T(n/10) + n^4 + n^2 & \text{hvis } n > 1. \end{cases}$$

○

Oppgi svaret i Θ -notasjon.

Svar: $\Theta(n^4 \lg n)$

Forklaring: Kan for eksempel løses med masterteoremet (tilfelle 2), der $a = 10\,000$, $b = 10$, $\lg_b a = 4$ og $f(n) = n^4 + n^2 = \Theta(n^4)$.

○

Rekursive kall	Størrelse, delproblem	Arbeid i hvert kall	Rekurrens	Kjøretid
Ett	Redusert med 1	Konstant	$T(n) = T(n-1) + \Theta(1)$	$\Theta(n)$
Ett	Halvert	Konstant	$T(n) = T(\frac{n}{2}) + \Theta(1)$	$\Theta(\lg n)$
Ett	Redusert med 1	Lineært	$T(n) = T(n-1) + \Theta(n)$	$\Theta(n^2)$
Ett	Halvert	Lineært	$T(n) = T(\frac{n}{2}) + \Theta(n)$	$\Theta(n)$
To	Redusert med 1	Konstant	$T(n) = 2T(n-1) + \Theta(1)$	$\Theta(2^n)$
To	Halvert	Konstant	$T(n) = 2T(\frac{n}{2}) + \Theta(1)$	$\Theta(n)$
To	Redusert med 1	Lineært	$T(n) = 2T(n-1) + \Theta(n)$	$\Theta(2^n)$
To	Halvert	Lineært	$T(n) = 2T(\frac{n}{2}) + \Theta(n)$	$\Theta(n \lg n)$

Merk at siste rekursen er løst med masterteoremet, mens de andre er løst uten.

tid = # subproblemer \times tid per subproblem

•

[C6] Forstå hvordan variabelskifte fungerer

• Triks

- $T(\text{sqrt}(n)) = \lg n$
 $m = \lg n$
 $S(m) = T(n) = T(2^m)$
 $T(2^{m/2}) = S(m/2)$
 $S(m/2) = m$
 $S(m) = 2$
 $T(n) = 2 \lg n$

Vi vil løse rekurrensen

$$T(n) = 2T(4\sqrt{n}) + \lg n$$

Vi substituerer $m = \lg n$, som betyr at $n = 2^m$ og $\sqrt{n} = n^{1/2} = 2^{m/2}$.

$$T(2^m) = 2T(4 \cdot 2^{m/2}) + m = 2T(2^2 2^{m/2}) + m = 2T(2^{m/2}) + m$$

Videre definerer vi en ny funksjon $G(m) = T(2^m)$, som lar oss skrive

$$G(m) = T(2^m) = 2T(2^{m/2} + 2) + m = 2G(m/2 + 2) + m$$

Denne kan vi løse med masterteoremet. Vi får at $G(m) = \Theta(m \lg m)$. Deretter må vi substituere tilbake med $m = \lg n$, som gir

•
$$G(m) = T(2^m) = T(n) = \Theta(m \lg m) = \Theta(\lg n \lg \lg n)$$

4 Rangering i lineær tid

Vi kan ofte få bedre løsninger ved å styrke kravene til input eller ved å svekke kravene til output. Sortering basert på sammenligninger ($x <= y$) er et klassisk eksempel: I verste fall må vi bruke $\lg n!$ sammenligninger, men om vi antar mer om elementene eller bare sorterer noen av dem så kan vi gjøre det bedre.

[D1] Forstå hvorfor sammenlikningsbasert sortering har en worst-case på $\Omega(n \lg n)$ (!)

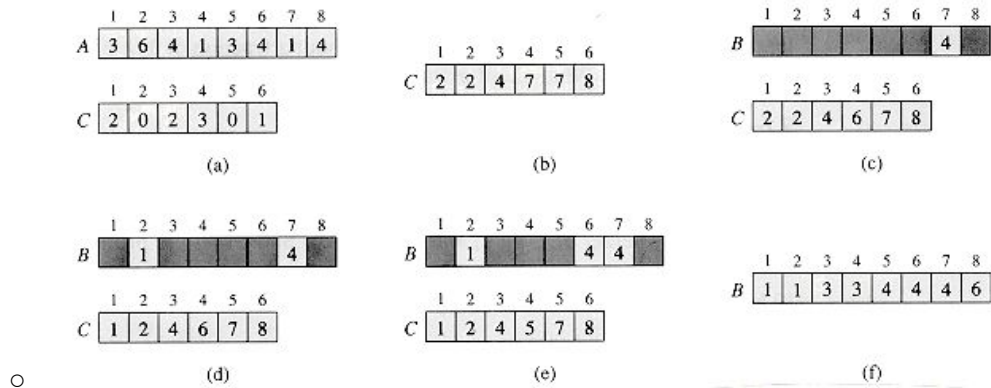
- De sammenligner to og to elementer.
- Høyde på treet er $\lg n$, utføres n ganger.
- Bevis:
 - Høyden h til et tre har l løvnoder som kan nåes i en sammenlikningssortering med n elementer. Fordi hver av de $n!$ permutasjonene til inputen forekommer som en viss løvnode, har vi at $n! \leq l$. Et binærtre med høyde h har 2^h blader, har vi at $n! \leq l \leq 2^h$. Tar logaritmen og får $h \geq \lg(n!) \rightarrow \Omega(n \lg n)$

[D2] Vite hva en stabil sorteringsalgoritme er

- En stabil sorteringsalgoritme er en algoritme som bevarer rekkefølgen basert på sorteringskriterie. Tall med samme verdi vil stå i samme rekkefølge i output arrayen som i input arrayen.
- Viktig når man har noe med samme verdi, men som peker til et annet element.

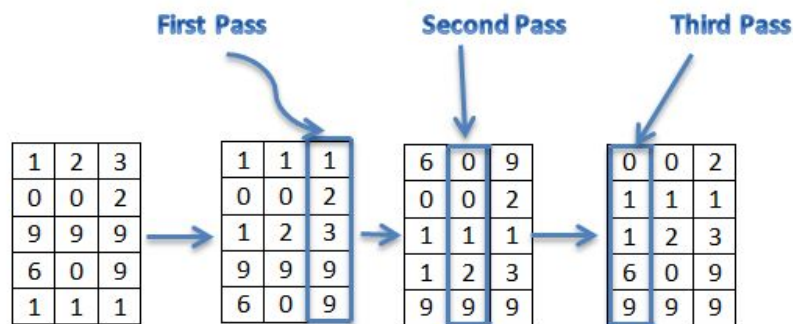
[D3] Forstå COUNTING-SORT, og hvorfor den er stabil

- Input
 - En liste $A[1...n]$
 - Liste $B[1...n]$ som holder den sorterte outputen
 - Heltall k , som er øvre grense for tallene som skal sorteres, altså høyeste tallet i lista
 - Liste $C[0...k]$ gir midlertidig arbeidslagring, lages i algoritmen
- Output
 - Sortert liste
- Info
 - Holder rede på 3 lister
 - C-lista forteller hvor mange forekomster det er av hvert tall. Hvis det er fire 1ere, vil indeks plass 1 ha verdien 4. Dette er ved initialisering.
 - Deretter oppdateres C lista slik at indeks plassen gir informasjon om hvor mange tall som er mindre eller lik gjeldene tall. Hvis det står fem på indeks plass 6, betyr det at det er fem tall i lista som er mindre eller lik tallet 6.
- Egenskaper
 - Stabil
 - Viktig å gå bakover når man skal plukke ut elementer til den nye listen, ellers ustabil
 - Stabil pga linja: $C[i] = C[i] + C[i-1]$
 - Ofte brukt som subsortering i Radix Sort fordi den er stabil
- Kjøretid
 - $\Theta(n + k)$
 - Bruker Counting-sort vanligvis når $k = O(n)$, slik at det blir $\Theta(n)$.
- Figur



[D4] Forstå RADIX-SORT, og hvorfor den trenger en stabil subrutine (!)

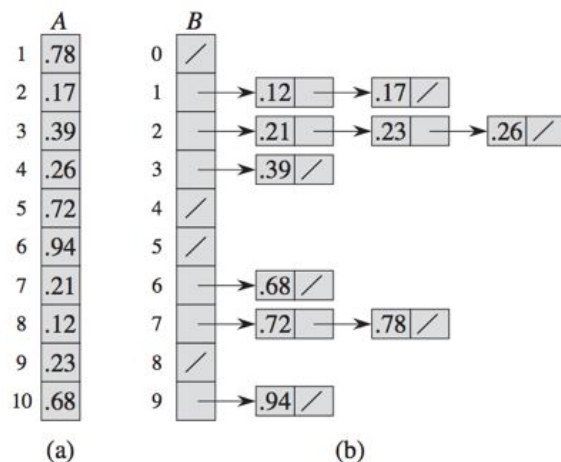
- Input
 - En liste A
 - Et tall d som beskriver høyeste orden av tall som skal sorteres (1-ere, 10-ere, 100-ere osv, eller dag, måned, år)
- Output
 - Sortert liste
- Info
 - Brukes til kortstokksortering
 - Sorterer hver rad for seg vha. en stabil sorteringsalgoritme
 - Starter på minste signifikante bit (LSB)
- Egenskaper
 - Må bruke en stabil sorteringsalgoritme
- Kjøretid
 - $\Theta(d(n + k))$, gitt n d -sifrede tall hvor hvert siffer kan være en av k mulige verdier.
 - Dette er når den stabile sorteringsalgoritmen bruker $\Theta(n + k)$ tid.
- Figur



[D5] Forstå BUCKET-SORT

- Input
 - Antar at input er generert tilfeldig og at tallene er uniformt og uavhengig fordelt på intervallet $[0,1)$
 - Tar inn en array A

- Output
 - Sortert liste
- Info
 - Deler intervallet $[0,1)$ inn i n like store “buckets”
 - Distribuerer tallene inn i disse bucketsene
 - Pga antagelsen om at input er fordelt, vil ikke mange tall falle i samme bucket
 - Sorterer én og én bucket, og deretter legger hver bucket inn i lista i rekkefølge
 - Det opprettes en liste $B[0..n-1]$ med lenkede lister/buckets
 - Sorterer med insertion sort, som tar lineær tid fordi det er få elementer i hver bølge.
- Egenskaper
 - Stabil, bygger en stack i hver bølge. Dette basert på at man bruker insertion sort. Står stabil i foiler og boka
 - In-place, må lage nye “bølger” som blir en ny datastruktur i minnet
 - Rask fordi den gjør antagelser på input
- Kjøretid
 - Best og Average $\Theta(n + k)$
 - Worst case $\Theta(n^2)$. Dette skjer når alle tall havner i samme bølge
- Figur



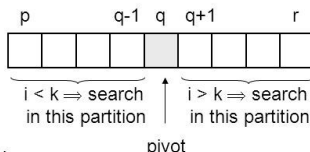
[D6] Forstå RANDOMIZED-SELECT

- Input
 - Array A
 - Pivotelement p
 - Sluttall r
 - Ønske om å finne i -te elementet i $A[p..r]$
- Output
 - returnerer det i -te minste tallet i arrayen $A[p..r]$
- Info
 - Hybrid av quicksort og binærsøk
 - Jobber kun på én side av partisjoneringen
 - Så lenge den finner i -te elementet, trenger ikke resten å være sortert
- Egenskaper
 - Splitt og hersk

- Sammenligner tall i randomized-partition
- Sorterer ikke nødvendigvis
- Kjøretid
 - Average - $\Theta(n)$
 - Worst - $\Theta(n^2)$, når pivot er valgt ugunstig, men unngår som regel dette ved randomized-partition
- Figur

Randomized Select

Alg.: RANDOMIZED-SELECT(A, p, r, i)



```

if p = r
  then return A[p]
q ← RANDOMIZED-PARTITION(A, p, r)
k ← q - p + 1
if i = k
  then return A[q]
elseif i < k
  then return RANDOMIZED-SELECT(A, p, q-1, i)
else return RANDOMIZED-SELECT(A, q+1, r, i-k)
  
```

Try: $A = \{1, 4, 2, 6, 8, 5\}$

10

[D7] Forstå SELECT

- Som Randomized-Select, finner den et ønsket element gjennom rekursiv partisjonering av input.
- Vil derimot denne gangen garantere en god split under partisjoneringen.
- Bruker en modifisert Partition, kalt Partition-Around, der man sender inn pivot x som input og bruker while-løkke helt til $i=x$. Deretter kjører Partition.
- (1) Del opp de n elementene i input til $\lfloor n/5 \rfloor$ (trenger ett oddetall, og 3 funker ikke) grupper med 5 elementer i hver, og på det meste en gruppe bestående av de gjenværende $n \bmod 5$ elementene
- (2) Finn medianen til hver av de $\lceil n/5 \rceil$ gruppene ved å sortere elementene (≤ 5) med insertion sort, og velg deretter median
- (3) Bruk select rekursivt for å finne medianen x av de $\lceil n/5 \rceil$ medianene. Hvis det er partalls medianer blir x den mindre medianen
- (4) Partisjonér input rundt median av medianer x ved å bruke den modifiserte versjonen av Partition.
- (5) Dersom $i=k$, returner x , hvis ikke fortsett rekursivt

5 Rotfaste trestrukturer

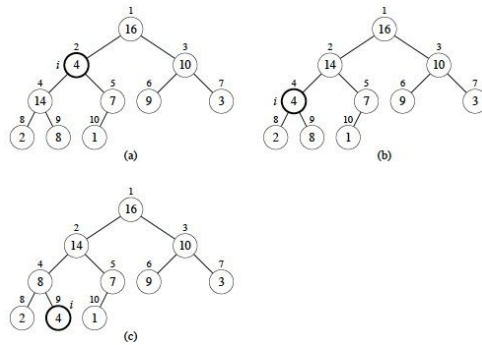
Rotfaste trær gjenspeiler rekursiv dekomponering. I binære søketrær er alt i venstre deltre mindre enn rota, mens alt i høyre deltre er større, og det gjelder rekursivt for alle deltrær!

Hauger er enklere: Alt er mindre enn rota. Det begrenser funksjonaliteten, men gjør dem billigere å bygge og balansere.

[E1] Forstå hvordan heaps fungerer, og hvordan de kan brukes som prioritetskøer (!)

(*PARENT, LEFT, RIGHT, MAX-HEAPIFY, BUILD-MAX-HEAP, HEAPSORT, MAX-HEAP-INSERT, HEAP-EXTRACT-MAX, HEAP-INCREASE-KEY, HEAP-MAXIMUM* og tilsvarende for min-heaps)

- Trestruktur slik at man kan aksessere elementene i $O(n \lg n)$ tid.
- Binær heap - to "barn" til hver node, totalt utfyllt bortsett fra nederste nivå, som er fylt fra venstre
 - Roten er $A[1]$, alle elementer i lista har sin egen node i treet
 - To typer: Max-Heap og Min-Heap
 - Max-Heap property er at Parent alltid er større eller lik barna. Brukes til heapsort
 - Min-Heap property er at Parent alltid er mindre eller lik barna. Brukes til prioritetskøer
 - Høyden til en node er definert som den lengste enkle veien fra noden til en løvnode.
 - Høyden til treet er definert som høyden til roten
 - Høyden til en heap med n elementer er $\Theta(\lg n)$
 - Operasjoner på binær heap er proporsjonal med høyden, og bruker derfor $O(\lg n)$ tid
 - Høyden i en binær min-heap er bare avhengig av antallet verdier, og er $\lfloor \lg n \rfloor$
- **PARENT(i)**
 - return $\lfloor i / 2 \rfloor$
- **LEFT(i)**
 - return $2i$
- **RIGHT(i)**
 - return $2i + 1$
- **MAX-HEAPIFY**
 - Skal opprettholde heap-egenskapen
 - Input: Liste A og index i
 - Lagrer en verdi *largest* av barna til den gjeldende noden
 - Kaller rekursivt på seg selv helt til heap-egenskapen er opprettholde
 - Kjøretid: $\Theta(1)$ for å fikse forholdet mellom nodene $A[i]$ og LEFT/RIGHT + tiden for å kjøre MAX-HEAPIFY på et subtree til et av barna.
 - Barnets subtree har maks størrelse $2n/3$, og vha masterteorem tilfelle 2 finner vi kjøretiden $O(\lg n)$
 - En annen måte å fremstille dette på er $O(h)$, der h er høyden til en node.
 - Worst-case skjer når laveste nivå av treet akkurat er halvfullt

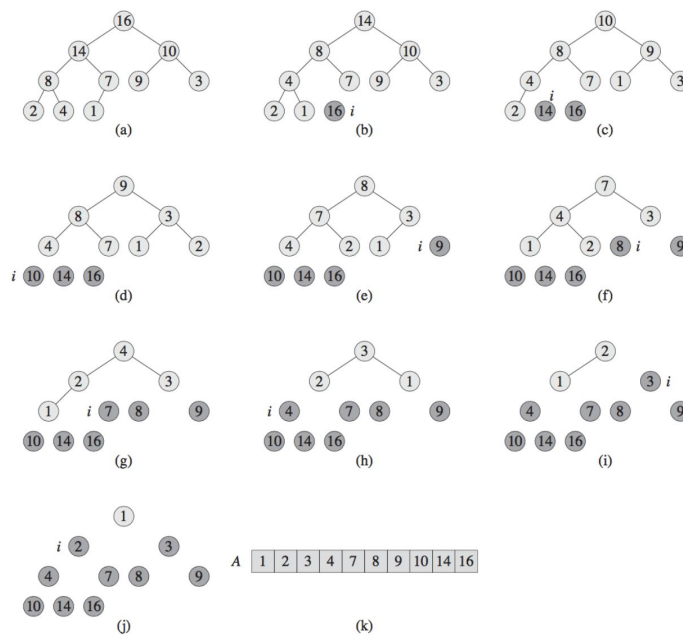


- BUILD-MAX-HEAP(A)
 - Får inn en liste som skal gjøres til en heap
 - Setter heap-size som lengden
 - Itererer fra $i = \lfloor A.length / 2 \rfloor$ ned til 1
 - Kjører deretter MAX-HEAPIFY på alle nodene
 - Kjøretid: $O(\lg n)$ for hvert kall til MAX-HEAPIFY, dette gjøres $O(n)$ ganger, gir en kjøretid på $O(n \lg n)$. Men kan vise at kjøretiden faktisk er $O(n)$.
- Prioritetskø
 - MAX og MIN prioritetskø
 - Datastruktur for å opprettholde et sett S med elementer, hver assosiert med en verdi kalt *key*.
 - Kan brukes til å fordele arbeidsoppgaver, slik at de viktigste blir gjort først
- HEAP-MAXIMUM(A)
 - return $A[1]$
 - $\Theta(1)$
- HEAP-EXTRACT-MAX(A)
 - Fjerner og returnerer det største elementet (størst key) i heapen (roten)
 - Kjører MAX-HEAPIFY på i løkke for $(A.length/2$ til 1)
 - $O(\lg n)$ fordi konstant arbeid + MAX-HEAPIFY
- HEAP-INCREASE-KEY(A, i, key)
 - Øker verdien til nøkkelen på plass i til *key*.
 - Kjøretid $O(\lg n)$
- MAX-HEAP-INSERT(A, key)
 - Legger til en ny node med verdi -inf
 - Kjører så HEAP-INCREASE-KEY(A, A.heap-size, key)
 - Kjøretid $O(\lg n)$

[E2] Forstå HEAPSORT

- Input
 - En liste A
- Output
 - En sortert liste
- Info
 - Gjør lista om til heap, sorterer, og deretter tilbake til liste igjen
 - Kjører først BUILD-MAX-HEAP på lista
 - Deretter løkker fra lengden av lista ned til 2
 - Bytter $A[1]$ med $A[i]$. Tar ut roten, legger til nederste løvnode som rot

- Heapsize blir mindre
 - Kjører MAX-HEAPIFY(A, 1)
 - Heap-egenskapen er opprettholdt, og vi gjentar oppskriften
- Egenskaper
 - Splitt og hersk
 - In-place
 - Bruker eksisterende tre for å swappe elementer
- Kjøretid
 - $O(n \lg n)$, siden BUILD-MAX-HEAP bruker $O(n)$ og hver av de $n-1$ kallene til MAX-HEAPIFY tar $O(\lg n)$ tid.
- Figur



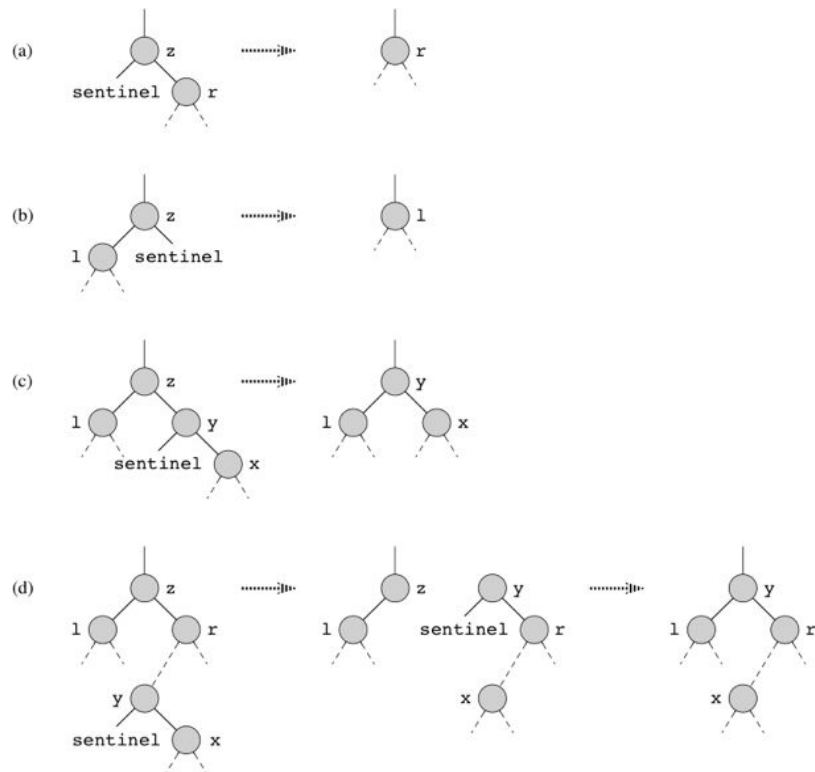
[E3] Forstå hvordan rotfaste trær kan implementeres

- Binære trær
 - Med left og right som barn
 - Dersom $x.p(\text{arent}) = \text{NIL}$, er x roten
- Rotfestede trær med ubundet forgreining
 - Vi kan utvide representasjonen av et binært tre til en klasse av trær der antall barn til hver node er på det meste en konstant k - vi bytter *left* og *right* attributtene til *chil1*, *chil2*...*childk*.
 - Kan bruke $O(n)$ minne for en vilkårlig n 'te rotfestet tre
 - Left-child, right-sibling representasjon
 - $x.\text{left-child}$ peker på barnet mest til venstre
 - $x.\text{right-sibling}$ peker på søsken til x rett til høyre

[E4] Forstå hvordan binære søketrær fungerer (!) (INORDER-TREE-WALK, TREE-SEARCH, ITERATIVE-TREE-SEARCH, TREE-MINIMUM, TREE-MAXIMUM, TREE-SUCCESSOR, TREE-PREDECESSOR, TREE-INSERT, TRANSPLANT, TREE-DELETE)

- Vi kan representere et binært søketre som en lenket datastruktur der hver node er et objekt.
- I tillegg til en *key* og et sett med data, har hver node attributtene *left*, *right* og *p* som peker til nodene korresponderende til henholdsvis sitt venstre barn, høyre barn og parent.
- Dersom et barn eller forelder mangler er den gjeldende attributtens verdi NIL.
- Rotnoden er den eneste noden i treet som har forelder lik NIL
- Binært søketre egenskapen
 - For en node x , er barnet til venstre mindre enn eller lik verdien til x
 - Motsatt er barnet til høyre større enn eller lik verdien til x
- INORDER-TREE-WALK(x)
 - Printer verdiene i treet i sortert rekkefølge
 - Kaller seg selv rekursivt på $x.left$
 - Kaller seg selv rekursivt på $x.right$
 - Bruker $\Theta(n)$ tid
- TREE-SEARCH(x, k)
 - Tar inn roten og en verdi den vil finne
 - Hvis verdien er mindre enn x , kall rekursivt på venstre side. Hvis den er større, kall rekursivt på høyre side
 - Kjøretid $O(h)$ der h er høyden på treet
 - Høyden på et tre måles nivå for nivå, der rotnoden ikke teller med
 - Antall kanter i den lengste stien fra rota til en løvnode
- ITERATIVE-TREE-SEARCH(x, k)
 - Samme formål, men rekursjonen byttes ut med en while-løkke
- TREE-MINIMUM(x)
 - Finner minste elementet i treet
 - While-løkke der man itererer seg ned til verdien helt til venstre nederst i treet
 - $O(h)$
- TREE-MAXIMUM(x)
 - Finner maksimum
 - $O(h)$
- TREE-SUCCESSOR(x)
 - Finner etterfølgeren til x
 - $O(h)$
- TREE-PREDECESSOR(x)
 - Finner forgjengeren til x
 - $O(h)$
- TREE-INSERT(T, z)
 - Setter inn verdien z i binærtreet på riktig plass
 - $O(h)$
- TRANSPLANT(T, u, v)
 - Brukes ved sletting
 - Erstatte et subtre som barn av sin forelder med et annet subtre.
 - Når TRANSPLANT erstatter subtreet med rot u med subtre med rot v , blir node u sin parent, node v sin parent, og u sin parent ender opp med å ha v som sitt barn

- Litt tricky opplegg, dette



-
- TREE-DELETE(T, z)
 - Sletter et element z fra treet
 - Bruker TRANSPLANT for å koble sammen foreldre og barn der man fjerner
 - $O(h)$

[E5] Vite at forventet høyde for et tilfeldig binært søketre er $\Theta(\lg n)$

- Vit det

[E6] Vite at det finnes søketrær med garantert høyde på $\Theta(\lg n)$

- Vit det

6 Dynamisk programmering

På sett og vis en generalisering av splitt og hersk, der delproblemer kan overlappe: I stedet for et tre av delprogram-avhengigheter har vi en rettet asyklisk graf. Vi finner og lagrer del-løsninger i en rekkefølge som stemmer med avhengighetene.

[F1] Forstå ideen om en delproblemgraf (!)

- Delproblemgrafen er en rettet graf, med en node for hvert distinkt delproblem. Grafen har en rettet kant fra noden for delproblemet x til noden for delproblemet y , dersom en optimal løsning for x avhenger av en optimal løsning for delproblemet y .
- Størrelsen til delproblemgrafen $G=(V,E)$ kan hjelpe oss å forstå kjøretiden til en algoritme med dynamisk programmering.
- Kjøretiden er summen av antall ganger vi må løse et delproblem

- Typisk er kjøretiden til et delproblem proporsjonal med utgående kanter i delproblemgrafene
- → kjøretiden er lineær (antall kanter og noder)
- Overlappende delproblemer
 - *Nyttig* når vi har overlappende delproblemer.
 - *Korrekt* når vi har optimal substruktur.
- Optimal substruktur handler om at vi bygger optimale løsninger ut fra optimale del-løsninger
 - Korteste vei består av korteste vei.
 - Lengste vei består ikke av lengste veier. Kan ikke deles opp på samme måte.

[F2] Forstå designmetoden dynamisk programmering (!)

- Løser problemer ved å kombinere løsningene til delproblemer. Gjelder når delproblemene overlapper - dvs. delproblemene har samme delproblemer.
- Ligner på splitt og hersk, men er bedre fordi den "husker" utregnede løsninger og lagrer de i en liste
- Beslektet med lineær programmering. Brukes som en optimeringsmetode. Handler om å utføre en serie med valg. Ting endrer seg over tid - derav dynamisk.
- Brukes ved optimaliseringsproblemer
- For å kunne bruke dynamisk programmering på et optimeringsproblem, må det ha optimal delstruktur og overlappende delproblemer.
- Oppskrift (fra Cormen):
 - (1) *Characterize the structure of an optimal solution.* Dekomponering. Måter å identifisere delproblemer på.
 - (2) *Recursively define the value of an optimal solution.* Definer verdien til en optimal løsning rekursivt.
 - (3) *Compute the value of an optimal solution.* Regn ut svaret.
 - (4) *Construct an optimal solution from computed information.* Konstruer en optimal løsning fra resultatene
- Handler om å løse distinkte delproblemer kun en gang. Når man har løst det lagrer man det, slik at man bare kan slå opp resultatet uten å regne det neste gang.
- Kan brukes til å feks finne det n'te Fibonacci tallet

[F3] Forstå løsning ved memoisering (top-down) (!)

- Memoisering
 - Lagre en verdi som vi kan se på/bruke igjen senere
- Skriver prosedyren rekursivt på en naturlig måte, men modifisert til å lagre resultatet av hvert delproblem. Sjekker først om den tidligere har løst samme delproblem.
- Starter øverst i treet, ved roten (top down), jobber seg rekursivt nedover, og returnerer derfra. Når den har funnet verdien på vei "tilbake til roten" i rekursjonen, regner den seg aldri nedover igjen, fordi den har lagret verdiene.
- *example:* If you are calculating the Fibonacci sequence `fib(100)`, you would just call this, and it would call `fib(100)=fib(99)+fib(98)`, which would call `fib(99)=fib(98)+fib(97)`, ...etc..., which would call `fib(2)=fib(1)+fib(0)=1+0=1`. Then it would finally resolve `fib(3)=fib(2)+fib(1)`, but it doesn't need to recalculate `fib(2)`, because we cached it.

- Kjøretid: $\Theta(n^2)$

[F4] Forstå løsning ved iterasjon (bottom-up)

- Iterativ metode fordi vi bruker loops
- Løser alle de mindre problemene først, helt til man kommer til det store problemet. Men på dette tidspunkt er allerede delproblemene løst
- Å løse et bestemt delproblem avhenger kun av å løse mindre delproblemer. Sorterer delproblemene etter størrelse og løser de i rekkefølge, minst først.
- Løse et bestemt problem; har allerede løst alle de mindre delproblemene som problemet består av og lagret løsningen.
- Kjøretid: $\Theta(n^2)$

[F5] Forstå hvordan man kan rekonstruere en løsning fra lagrede beslutninger

- Vi lagrer ikke bare den mest optimale løsningen, men også hvilket valg som ledet til den optimale verdien

[F6] Forstå hva optimal delstruktur er

- Med en optimal delstruktur menes det at en optimal løsning til delproblemet også er en optimal løsning til problemet.
- Et problem må ha optimal substruktur dersom en optimal løsning skal inneholde optimale løsninger på delproblemer
- Finne optimale delstrukturer
 - (1) Vise at en løsning til et problem består av et valg. Å ta dette valget gir en eller flere delproblemer å løse
 - (2) Gitt et problem, får man gitt et valg som leder til en optimal løsning. Ikke tenk på hvordan man kan ta dette valget, bare anta det man har fått.
 - (3) Gitt et valg, må man velge hvilke delproblemer som følger og hvordan man best karakteriserer "rommet" av delproblemene
 - (4) Viser at løsningen til delproblemene brukt i en optimal løsning til et problem også selv må være optimale

[F7] Forstå hva overlappende delproblemer er

- Overlappende delproblemer -> bruk dynamisk programmering
 - At man løser samme problem igjen og igjen, istedet for å alltid lage nye delproblemer, som feks splitt og hersk
- Når en rekursiv algoritme gjentar det samme problemet gjentatte ganger, sier vi at optimaliseringsproblemet har overlappende delproblemer.

[F8] Forstå eksemplene stavkutting og LCS

- *Stavkutting*
 - Problem innen dynamisk programmering der du har en stav med en viss pris på ulike lengder av staven. Staven skal kuttet slik at du får mest mulig verdi ut av staven
 - Kan dele opp en stav på lengde n i 2^{n-1} deler
 - Input
 - En lengde n og priser p_i for lengder $i = 1, \dots, n$

- Output:
 - Lengder l_1, \dots, l_k der summen av lengdene er n og totalprisen $r_n = p_{l_1} + \dots + p_{l_k}$ er maksimal
- Bestem max inntekter r_n som er mulig å oppnå ved å kutte opp stangen og selge
- Her lagrer vi ikke løsninger på delproblemene. Veldig lite effektivt! Eksponentiell kjøretid
- Bruker dynamisk programmering for å konvertere det til en effektiv algoritme: løse hvert distinkt delproblem en gang og lagre det. Kan enten velge top-down med memoisering eller bottom-up.
- CUT-ROD(p, n)
 - p liste med priser og n lengde
 - linja $q = \max_{1 \leq i < n} (p_i + \text{CUT-ROD}(p, n-i))$ som utfører selve utførelsen
 - $r_n = \max_{1 \leq i < n} (p_i + r_{n-i})$ mer generelt
- Denne må modifiseres for å ikke bli eksponentiell. Legg inn memoisering (top-down) eller iterative metode (bottom-up) for å lagre svarene, og hvor det kom fra
- Begge versjoner gir $\Theta(n^2)$ kjøretid
- Må løse $\Theta(n)$ delproblemer for å finne optimal løsning
- Når man skriver denne i boka, lager man tre rader, en for lengden i , en for prisen p og en for hvor du skal kutte s
- For å få optimal løsning for lengde 3, kutter du først av lengde 1, deretter lengde 2. Når du har kuttet lengde 1, flytter du $i-1$ plass tilbake, og ser hvor mye du skal kutte der
-

Lengde(i)	1	2	3	4
Pris(p)	2	4	5	8
Kutt(s)	1	2	1	4

- LCS:

- Gitt en sekvens X og en sekvens Y , sier vi at Z er en felles subsekvens av X og Y hvis Z er en subsekvens av både X og Y .
- Kan bruke Bottom-up DP. Har $\Theta(mn)$ distinkte delproblemer
- Input
 - To sekvenser, $X = \langle x_1, \dots, x_m \rangle$ og $Y = \langle y_1, \dots, y_n \rangle$
- Output
 - En sekvens $Z = \langle z_1, \dots, z_k \rangle$ og indekser $i_1 < \dots < i_k$ og $l_1 < \dots < l_k$ der $z_{ij} = x_{i_j}$ og $z_{ij} = y_{l_j}$ for $j = 1 \dots k$ der Z har maksimum lengde
- Lager en svær jævla tabell der man legger inn verdier og piler
- Sammenligner med den over og ved siden av gjeldende rute hvis ikke $x_i == x_j$, sender pila til venstre \leftarrow hvis $x_{i-1,j} > x_{i,j-1}$, sender opp \uparrow hvis $x_{i,j} = x_{i,j-1}$
- Hvis de er like, så plusser man på 1 og sender pila slik \nwarrow
- Kjøretid $O(n^2)$

		j	0	1	2	3	4	5	6
i	x_i	y_j		B	D	C	A	B	A
			0	0	0	0	0	0	0
0			0	0	0	0	0	0	0
1	A		0	↑	↑	↑	↖	←	↖
2	B		0	↖	←	←	↑	↖	←
3	C		0	↑	↑	↖	←	↑	↑
4	B		0	↖	↑	↑	↑	↖	←
5	D		0	↑	↖	↑	↑	↑	↑
6	A		0	↑	↑	↑	↖	↑	↖
7	B		0	↖	↑	↑	↑	↖	↑

[F9] Forstå løsningen på 0-1-ryggsekkproblemet (se appendiks D) (KNAPSACK, KNAPSACK')

- Følgende problem
 - En tyv skal tjele med seg noe fra et hus. Vil fylle sekken slik at han får med seg mest mulig verdi
 - To valg, enten å ta med en ting, eller ikke (man kan ikke dele en tv i 2 og bare ta med halvparten), derav navnet 0-1 knapsack
 - Løses ved dynamisk programmering
 - Ja, vi tar med gjenstand i Vi løser så problemet for gjenstander $1, \dots, i-1$ men der kapasiteten er redusert med w_i . Vi legger så til v_i til slutt.
 - Nei, vi tar ikke med gjenstand i . Vi løser så problemet for gjenstander $1, \dots, i-1$, men kan fortsatt bruke hele kapasiteten. Til gjengjeld får vi ikke legge til v_i til slutt.
- Er et NP-hardt problem, ingen har funnet polynomisk løsning på det.
 - Kjøretiden er $\Theta(nW)$, siden det er nW delproblemer
 - Størrelsen blir da $\Theta(n + \lg W)$, siden vi bare trenger $\Theta(\lg W)$ bits for å lagre parameteren W .
 - Poenget er altså at W vokser eksponentielt som funksjon av $\lg W$, og kjøretiden er, teknisk sett, eksponentiell.
 - Dette er kanskje enda enklere å se hvis vi lar m være antall bits i W , så vi kan skrive kjøretiden som $T(n, m) = \Theta(n2^m)$.
 - Dette er ikke en polynomisk kjøretid. Kjøretider som er polynomiske hvis vi lar et tall fra input være med som parameter til kjøretiden (slik som $\Theta(nW)$, der W er et tall fra input, og ikke direkte en del av problemstørrelsen) kaller vi pseudopolynomiske.
- KNAPSACK(n, W)
 - Med memoisering eller bottom-up (figur)

```

KNAPSACK'(n, W)
1 let K[0..n, 0..W] be a new table
2 for j = 0 to W
3   K[0, j] = 0
4 for i = 1 to n
5   for j = 0 to W
6     x = K[i - 1, j]
7     if j < wi
8       K[i, j] = x
8     else y = K[i - 1, j - wi] + vi
8       K[i, j] = max(x, y)

```

- Input: Verdier v_1, \dots, v_n , og vekter w_1, \dots, w_n og en kapasitet W
- Output: Indekser i_1, \dots, i_k slik at $w_{i_1} + \dots + w_{i_k} \leq W$ og totalverdien $v_{i_1} + \dots + v_{i_k}$ er maksimal
- Sjuk tabell her og

$W=10$

v_i	10	40	30	50
w_i	5	4	6	3

$$K[i, w] = \max(K[i-1, w], v_i + K[i-1, w - w_i])$$

$i \backslash w$	$w=0$	$w=1$	$w=2$	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0
$w_i=5\text{kg}$ 1	0	0	0	0	0	10	10	10	10	10	10
4kg 2	0	0	0	0	40	40	40	40	40	50	50
6kg 3	0	0	0	0	40	40	40	40	40	50	70
3kg 4	0	0	0	50	50	50	50	90	90	90	90

$K[i, w]$

www.LogicHeap.com

7 Grådige algoritmer

Grådige algoritmer består av en serie med valg, og hvert valg tas lokalt: Algoritmen gjør alltid det som ser best ut her og nå, uten noe større perspektiv. Slike algoritmer er ofte enkle; utfordringen ligger i å finne ut om de gir rett svar

[G1] Forstå designmetoden grådighet (!)

- Handler om å ta det valget som ser best ut der og da.
- Tar et lokalt optimalt valg i håp om at det vil lede til den globale optimale løsningen
- Der dynamisk programmering ville sett på alle alternativer, der vil grådighet på lokal basis velge det som ser best ut. Løser kun de delproblemene som blir valgt.

- Forskjell på dynamisk programmering og grådighet;
 - Begge utnytter optimal delstruktur
 - *Dynamisk*:
 - Løs delproblemer rekursivt
 - Bygg løsning på beste delløsning
 - *Grådighet*:
 - Løs det mest lovende delproblemet rekursivt
 - Bygg løsning på denne delløsningen
 - Kan være enklere å implementere og ha bedre kjøretid
- DP vil fortsatt fungere - akkurat som for D&C
- Bruke dynamisk programmering der grådighet kan brukes: Kaster bort tid, for løser mange delproblemer man ikke trenger å løse
- Ting å identifisere:
 - (1) Globalt optimalitetskriterium
 - (2) Lokalt optimalitetskriterium
 - (3) Konstruksjonstrinn
- Ting å vise:
 - (1) Grådighetsegenskapen
 - Vi kan velge det som ser best ut, her og nå
 - (2) Optimal substruktur
 - En optimal løsning bygges av optimale delløsninger
- *Grådig valg + optimal delløsning gir optimal løsning*
 - (1) Formulér som optimeringsproblem der vi tar et valg så ett delproblem gjenstår
 - (2) Vis at det alltid finnes en optimal løsning som tar det grådige valget
 - (3) Vis at optimal løsning på grådig valgt delproblem gir globalt optimal løsning
- Bevis ved forsprang:
 - Vis at grådighet er minst like bra som alle andre algoritmer for hvert trinn; det følger at den gir en optimal løsning
 - Vis f.eks. induktivt at siste sluttidspunkt i grådig aktivitetsutvalg er minst like tidlig som enhver annen løsning

[G2] Forstå grådighetsegenskapen (the greedy-choice property) (!)

- Handler om at man kan danne en globalt optimal løsning ved å ta lokalt optimale (grådige) valg.
- Vi kan velge det som ser best ut, her og nå.

[G3] Forstå eksemplene aktivitet-utvalgelse og det fraksjonelle ryggsekkproblemet

- Aktivitet-utvalgelse:
 - Ønsker å velge en max-size delmengde av gjensidig compatible aktiviteter. Har et sett $S=\{a_1, a_2, \dots, a_n\}$ med n aktiviteter.
 - Hver aktivitet a_i har en starttid s_i og en sluttid f_i , hvor $0 \leq s_i < f_i < \infty$.
 - a_i og a_j er compatible hvis $[s_i, f_i)$ og $[s_j, f_j)$ ikke overlapper, dvs. $s_i \geq f_j$ eller $s_j \geq f_i$
 - *Input: Intervaller $[s_i, f_i), \dots, [s_j, f_j)$*

- *Output: flest mulig ikke-overlappende intervaller*
- Ønsker å velge størst mulig delmengde av ikke-overlappende aktiviteter
- Valg: Et intervall som skal bli med
- Løs begge delproblemer rekursivt og legg til 1
- Trenger ikke å se på alle delproblemene
 - Det vil alltid lønne seg å ta med intervallet som slutter først \rightarrow blokkerer minst
- Greedy choice for aktivitet-utvalgelse:
 - Velge aktiviteten som er ferdig først, altså a_1
 - Finne aktiviteter som starter etter at a_1 er ferdig
 - La $S_k = \{a_i \in S : s_i \geq f_k\}$ være settet med aktiviteter som starter etter at a_k er ferdig. Hvis a_1 er en optimal løsning, da er en optimal løsning til det originale problemet inneholdt av a_1 og alle aktivitetene i en optimal løsning til delproblemet S_1 .
- Kan bruke dynamisk programmering, men det er ikke nødvendig. Kan i stedet gjentatte ganger velge den aktiviteten som er ferdig først, beholde kun aktiviteter som er kompatible med denne aktiviteten, og gjenta til ingen aktiviteter gjenstår.
- Grådige algoritmer har typisk top-down design.
- RECURSIVE-ACTIVITY-SELECTOR(s, f, k, n)
 - Returnerer et max-size set av gjensidig kompatible aktiviteter i S_k .
 - Kaller seg selv rekursivt
 - Kjøretid: $\Theta(n)$
- GREEDY-ACTIVITY-SELECTOR(s, f)
 - Iterativ versjon av rec-act-sel(s, f, k, n).
 - Antar at input aktivitetene er ordnet etter monotont økende sluttid. Samler valgte aktiviteter i et set A og returnerer settet når den er ferdig.
 - Kjøretid: $\Theta(n)$
- Ryggsekkproblemet - fractional knapsack:
 - Samme som 0-1 knapsack, men kan nå ta med seg deler av en ting, istedet for å velge enten ta med eller ikke
 - Finner kilosprisen til hvert element, og tar med så mye som mulig av den som er verdt mest per kilo, fyller deretter neste og neste helt til kapasiteten er full
 - Må sortere elementene mtp kilopris, kjører derfor på $O(n \lg n)$ tid
 - Input: verdier v_1, \dots, v_n , vekter w_1, \dots, w_n og en kapasitet W
 - Output: Indekser i_1, \dots, i_k og en fraksjon $0 \leq \epsilon \leq 1$ slik at $w_{i_1} + \dots + w_{i_{k-1}} + \epsilon * w_{i_k} \leq W$ og totalverdien $v_{i_1} + \dots + v_{i_{k-1}} + \epsilon * v_{i_k}$ er maksimal
 - Optimal substruktur

Fractional Knapsack-Greedy Solution

• Greedy-fractional-knapsack (w, v, W)

1. for $i = 1$ to n
2. do $x[i] = 0$
3. weight = 0
4. while weight < W
5. do $i =$ best remaining item
6. if weight + $w[i] \leq W$
7. then $x[i] = 1$
8. weight = weight + $w[i]$
9. else
10. $x[i] = (W - \text{weight}) / w[i]$
11. weight = W
12. return x

○

[G4] Forstå HUFFMAN og Huffman-koder

- Huffman koder bruker en tabell over hvor ofte en karakter/bokstav/siffer forekommer for å bygge opp en optimal vei å nå hver karakter som en binær string
- De karakterene som brukes oftest, vil ha kortest mulig "binær kost", fordi de representeres flere ganger. Altså vil bokstaven e ha kortere binærfremstilling enn feks x i en norsk tekst med huffmankoding
- Huffman koder komprimerer data veldig effektivt. Vanlig å spare 20-90%.
- Vil lage binær kode for tegn. Tegnene har frekvenser. Kodene kan ha varierende lengde. Vil minimere forventet kodelengde.
- *Prefiks-kode*: Ingen koder er prefiks av andre. Kan representeres som stier i binærtre, med tegn som løvnoder.
- I et Huffman tre skal dybden være minst mulig.
- Prøver å lage en grådig algoritme:
- Vi kan "slå sammen" to partielle løsninger ved å la én bit velge mellom dem.
- Grådighet: slå alltid sammen de sjeldneste, siden den ekstra bit-en da koster minst
- *Huffman(C)*:
 - Input: Alfabet $C = \{c, \dots\}$ med frekvenser $c.\text{freq}$
 - Output: Binær koding som minimerer forventet kodelengde $\sum_{c \in C} (c.\text{freq} * \text{length}(\text{code}(c)))$
 - Starter med et sett av $|C|$ blader og utfører en sekvens med $|C|-1$ sammenslåings operasjoner. Returnerer roten til treet.
 - Kjøretid: $O(n \lg n)$

8 Traversering av grafer

Vi traverserer en graf ved å besøke noder vi vet om. Vi vet i utgangspunktet bare om startnoder, men oppdager naboene til dem vi besøker. Traversering er viktig i seg selv, men danner også ryggraden til flere mer avanserte algoritmer

[H1] Forstå hvordan grafer kan implementeres

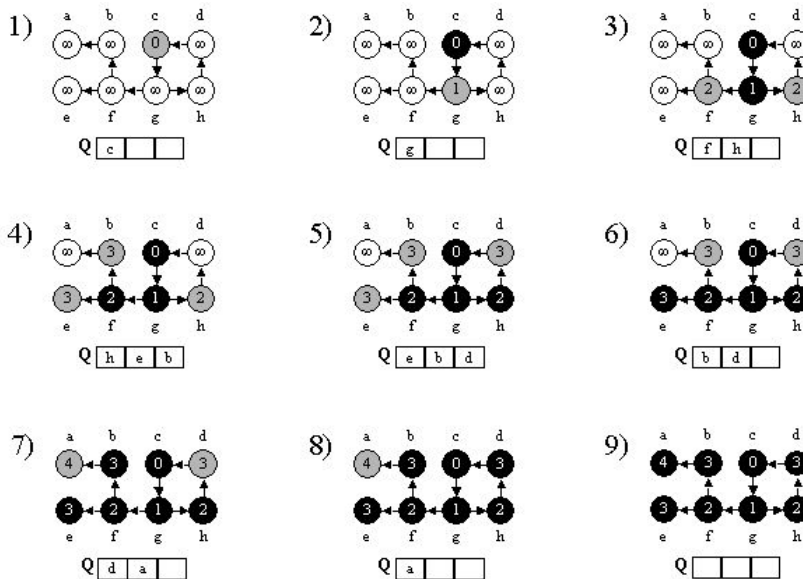
- Graf $G = (V, E)$, der V er noder (vertices) og E er kanter (edges)
- Kan representere grafer på to standard måter:
 - Nabolister - samling av tilstøtende lister
 - Nabomatrise - matrise

- Begge gjelder både urettede og rettede grafer.
- Nabolister representasjonen gir en mer kompakt måte å representere sparsomme grafer, og blir derfor vanligvis valgt. Brukes der $|E|$ er mye mindre enn $|V|^2$
- Kan foretrekke nabomatrise-representasjon når det er snakk om grafer som er tette, eller vi fort må bestemme om det finnes en kant mellom to gitte noder. Brukes der $|E|$ er nær $|V|^2$ eller når vi kjapt trenger å finne ut om det er en kant som binder to gitte noder
- Nabomatriser ofte raskere, men tar mer plass, selv om dette kommer an på problemet
- Finnes flere representasjoner enn dette
- Naboliste
 - Har en liste med alle tallene, som peker på en lenket liste. Denne inneholder hvem som er naboene.
 - Hvis G er en rettet graf; summen av lengdene til alle nabolistene er $|E|$
 - Hvis G er en urettet graf; summen av lengdene til alle nabolistene er $2 * |E|$
 - Kan lett tilpasse tilstøtende lister til å representere *vektede grafer*; grafer hvor hver kant har en assosiert vekt, typisk gitt av en vektfunksjon $w: E \rightarrow \mathbb{R}$
 - Krever $\Theta(V+E)$ minne
 - Hvis $|V| = |E|$ vil nabolister være den beste måten å lagre G mest plasseffektivt
- Nabomatrise
 - For en graf $G=(V,E)$ antar vi at nodene er nummerert $1, 2, \dots, |V|$. Da består nabomatrise-representasjonen til G av $|V| * |V|$ matriser $A = (adj_{ij})$ slik at $adj_{ij}=1$ hvis $(i,j) \in E$, 0 ellers.
 - Krever $\Theta(V^2)$ minne
 - Kan også representere vektet grafer.
- Selv om nabolister er minst like plasseffektivt, er nabomatriser enklere. For ikke-vektet grafer krever nabomatriser kun ett bit per inngang.

[H2] Forstå BFS, også for å finne korteste vei uten vektor

- Bredde først søk er en grafraverseringsalgoritme/søkealgoritme som baserer seg på at den for hver gang den besøker en node legger til barne/naboene til noden i en gitt rekkefølge (feks alfabetisk/leksikografisk) i en kø
 - (1) Gitt en graf, en startnode og en rekkefølge å legge til naboer i, oppretter vi en queue og legger startnoden inn i køen. Dette er foreløpig eneste element i køen
 - (2) Så lenge det er ett eller flere elementer i køen, ta ut det første elementet fra køen(dequeue), og legg til naboene til dette elementet til i køen i gitt rekkefølge
- Dersom man søker etter et element i grafen, kan man terminere søket når BFS har oppdaget elementet
- Initialisert et tre med startnodeverdi = 0 og resten = inf. Hvis node v har avstand 4 fra startnode u , skal verdien settes til 4
- Lager også et BF-tre, som inneholder alle noder som kan nåes
 - Når en node er oppdaget, legges den til i treet
 - En node har høyst 1 forelder, fordi den oppdages for første gang bare én gang
- Fungerer på både rettede og urettede grafer

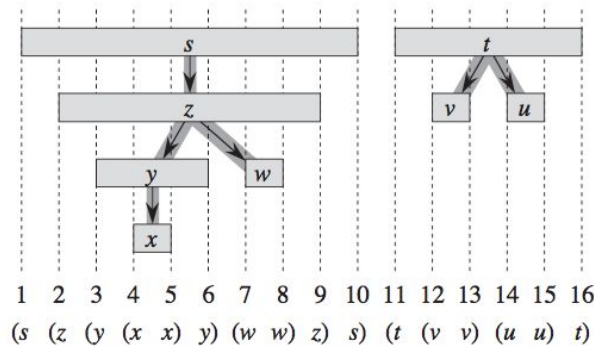
- Typisk eksamensgreie:
 - Hvis alle kantvekter har lik verdi k , og vi vil finne korteste vei, er det det samme som at alle kantvekter har verdi 1, og BFS vil være beste algoritme
- Kjøretid $O(|V| + |E|)$
 - Enqueue og Dequeue tar $O(1)$ tid, og blir brukt på $O(V)$ stk.
 - Går gjennom hver naboliste høyst 1 gang. Summen av lengden på alle nabolistene er $\Theta(E)$
 - Initialisering er $O(V)$
 - Gir den totale kjøretiden $O(|V| + |E|)$



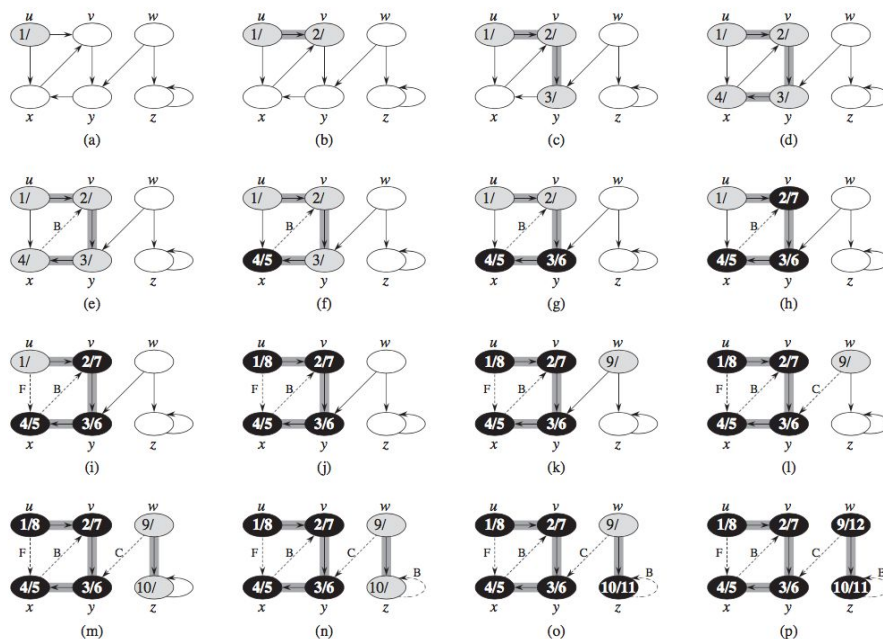
[H3] Forstå DFS, og parentesteoremet

- Som navnet - søk dybere i grafen hvis det er mulig
- Algoritmen utforsker kantene ut fra den nyligste oppdagede noden v , som fortsatt har ikke-utforskede kanter. Når alle av v 's kanter har blitt utforsket, går prosedyren tilbake til noden v kom fra for å se etter ikke-utforskede kanter.
- Jobber rekursivt
- Ved å merke start og sluttider kan DFS brukes til topologisk sortering
- Forgjenger subgrafene til DFS danner derfor en dybde-først skog med flere dybde-først trær, kantene i E_{pi} er tre-kanter
- For å lage dybde først skog, tidsstempler man hver node - start og sluttidspunkt
 - Hvert flytt til en ny node er én telling
- Kjøretid
 - $\Theta(V + E)$
 - DFS-Visit blir kalt på én gang per node
 - I DFS-Visit kjøres en løkke $\Theta(E)$ ganger
- Parantesteoremet - ett av disse tre forholdene holder
 - (1) Intervallene $[u.d, u.f]$ og $[v.d, v.f]$ er helt disjunkte, og hverken u eller v er en etterkommer den andre i dybde-først skogen.
 - (2) Hele intervallet $[u.d, u.f]$ er i intervallet $[v.d, v.f]$, og u er en etterkommer av v i ett dybde-først-tre.
 - (3) Hele intervallet $[v.d, v.f]$ er i intervallet $[u.d, u.f]$, og v er en etterkommer av u i ett dybde-først-tre.

- Figur gir nok bedre forklaring, man ser på søket i dybden som parenteser inni hverandre



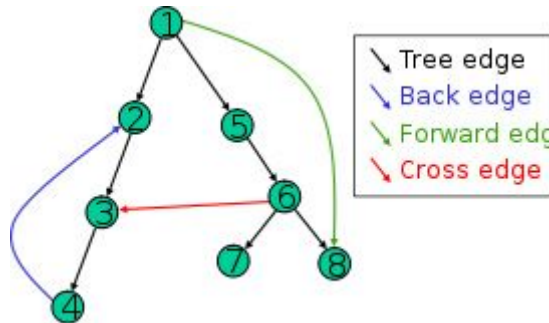
○



[H4] Forstå hvordan DFS klassifiserer kanter

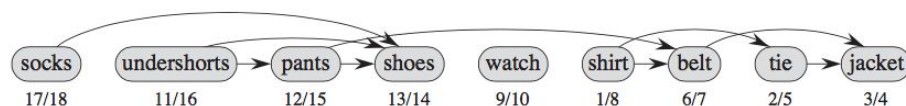
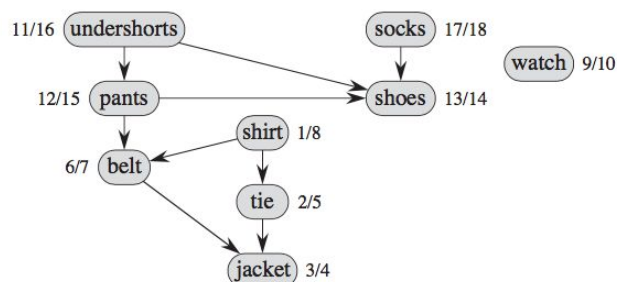
- Fire type kanter
- (1) Tree edges
 - Kanter i DF-skogen. Kanten (u,v) er en tre-kant dersom v først ble funnet ved utforskning av kanten (u,v)
 - Møter en hvit node
- (2) Back edges
 - Kantene (u,v) som forbinder en node u til en forgjenger v i et DF-tre. Vi ser på selv-løkker, som kan forekomme i rettede grafer til å være back edges.
 - Hvis man kjører DFS på en graf og finner en backedge, så har man en sykel
 - Møter en grå node
- (3) Forward edges
 - Non-tree edges (u,v) som forbinder en node u til en etterkommer v i ett DF-tre
 - Møter en svart node
- (4) Cross edges

- Er alle andre kanter. De kan gå mellom noder i samme DF-tre, så lenge en av nodene ikke er en forgjenger til den andre, eller så kan de gå mellom noder i forskjellige DF-trær
- (Møter en svart node)
- Hvis det er en urettet graf, har vi bare tree edges eller back edges



[H5] Forstå TOPOLOGICAL-SORT

- Gir nodene en rekkefølge
- Foreldre før barn - evt: alle kommer etter avhengigheter
- Krever at grafen er DAG (directed acyclic graph)
- Som om man skal kle på seg om morgenen, man kan ikke ta på bukse før man har tatt på seg truse (med mindre du er crazy banana loco)
- Bruker DFS og noterer ned discovertime og finishtime, men det er finishtime som brukes i kallet!
 - Sorteres etter synkende finish-tid
- For å dobbelsjekke at man har sortert riktig, kan man sette alle nodene opp i rekkefølge fra venstre mot høyre, høyest sluttid til lavest. Trekk så kanter fra nodene slik de er beskrevet i treet oppgaven oppgir.
 - Hvis alle kantene peker til høyre i rekka med noder, har du riktig
 - Eller: Hvis du har en kant som peker til venstre, har du sortert feil
- Kjøretid $\Theta(V + E)$
 - DFS $\Theta(V + E)$ og $O(1)$ tid å innsette hver av de $|V|$ nodene i en lenket liste



[H6] Forstå hvordan DFS kan implementeres med en stakk

- Blir da som BFS, bare med LIFO-kø aka stakk. "Køen" blir en kallstakk

- Bruker stakk i stedet for rekursjon, denne er LIFO og man må pushe barna til den nåværende noden på stakken i omvendt rekkefølge
 - (1) Gitt en graf, en startnode og en rekkefølge å legge til noder i, oppretter vi en stakk, og legger startnoden inn i stakken. Stakken har nå startnoden som eneste element
 - (2) Så lenge det er ett eller flere elementer i stakken, pop det øverste elementet og legg (push) barna på stakken, i motsatt rekkefølge. Altså hvis rekkefølgen på barna er ABC, må de legges i stakken CBA

[H7] Forstå hva traverseringstrær (som bredde-først og dybde-først-trær) er

- Når vi traverserer lagres forgjenger for hver node
- Disse kan danne et tre, hvor vi ser hvilken vei vi har gått for å komme til hver node
- Grunnen til at en LIFO-kø gir oss samme atferd som en rekursiv traversering (altså DFS) er at vi egentlig bare simulerer hvordan rekursjon er implementert. Internt bruker maskina en kallstakk, der informasjon om hvert kall legges øverst, og hentes frem igjen når rekursive kall er ferdige.

[H8] Forstå traversering med vilkårlig prioritetskø (!)

- Prosedyren BFS (på side 595) kan tilpasses til å oppføre seg omtrent likt med DFS (side 605), ved å bytte ut FIFO-køen Q med en LIFO-kø, eller stakk (stack).
- Vi mister da tidsmerkingen (v.d og v.f), men rekkefølgen noder farges grå og svarte på vil bli den samme.
- En annen forskjell er at DFS, slik den er beskrevet i boka, ikke har noen startnode, men bare starter fra hver node etter tur, til den har nådd hele grafen; sånn sett er BFS mer beslektet med DFS-Visit.
- Køen Q i BFS er rett og slett en liste med noder vi har oppdaget via kanter fra tidligere besøkte noder, men som vi ennå ikke har besøkt.
- Noder er hvite før de legges inn, grå når de er i Q og svarte etterpå.
- Det at vi bruker en FIFO-kø er det som lar BFS finne de korteste stiene til alle noder, siden vi utforsker grafen «lagvis» utover, men vi kan egentlig velge vilkårlige noder fra Q i hver iterasjon, og vi vil likevel traversere hele den delen av grafen vi kan nå fra startnoden
- Hvordan vi velger å traversere, handler om hvordan vi ønsker å prioritere noder.
 - Andre prioriteringer vil gi andre traverseringsalgoritmer, som for eksempel den såkalte A* -algoritmen (ikke pensum).

9 Minimale spenntær

Her har vi en graf med vekter på kantene, og ønsker å bare beholde akkurat de kantene vi må for å koble sammen alle nodene, med en så lav vektsum som mulig. Erke-eksempel på grådighet: Velg én og én kant, alltid den billigste lovlige.

[I1] Forstå skog-implementasjonen av disjunkte mengder

(CONNECTED-COMPONENTS, SAME-COMPONENT, MAKE-SET, UNION, LINK, FIND-SET)

- En disjunkt-sett datastruktur vedlikeholder en samling $S = \{S_1, S_2, \dots, S_k\}$ av disjunkte dynamiske sett

- Vi identifiserer hvert sett med en representant. Denne kan ikke være i et annet sett, da settene er disjunkte
- Skal slå sammen mange mengder, sjekker om to noder har samme representant
- Hver node har en peker til foreldrenode
 - Ender opp med en rot
 - Rota representerer mengden
- Heuristikk: forbedre
- Union by rank-heuristikk
 - Rang er øvre grense for nodehøyde, så man alltid putter det korteste treet under det høyeste hvis man skal koble de sammen. Hvis de har lik rang, øker man rangen til den øverste med 1.
 - Under operasjonen UNION plasseres noden med høyest rank over den andre noden
- Path compression-heuristikk
 - Brukes under find-set operasjoner for å få hver node på “find path” til å peke på roten. Endrer ingen ranger.
 - Path compression gjør at man ikke “gror” lange rekker av noder, men at nodene ligger nærmere roten.
 - Man må da ikke traversere gjennom mange noder på metoden FIND-SET (som forflytter seg oppover i treet til man finner roten, som er treet’s “representative”)
- Sammen gir union by rank og path compression en worst case på $O(m \cdot \alpha(n))$, hvor $\alpha(n)$ er en veldig tregt voksende funksjon. Tilnærming $\alpha(n) \leq 4$.
- MAKE-SET
 - Lager nytt sett hvor x er eneste medlem
- UNION
 - Forener to sett til et nytt sett som er unionen av de to, og fjerner de to tidligere fra samlingen
- LINK
 - To røtter; en henges under den andre. Den med lavest rang peker på den med høyest. Hvis de er like, øke rangen med 1.
- FIND-SET
 - Returnerer peker til representanten av settet som inneholder x
- En av de mange bruksområdene til disjunkte-sett datastrukturen er å kunne definere de koblede komponentene i en urettet graf.
- CONNECTED-COMPONENTS
 - Bruker disjunkte-sett operasjonene til å regne ut de koblede komponentene i grafen.
- CONNECTED-COMPONENTS
 - Når CONNECTED-COMPONENTS har prosessert grafen, kan prosedyren SAME-COMPONENTS svare på om to noder er i den samme koblede komponenten.
- Disjunkte-sett skoger
 - En raskere implementasjon av disjunkte sett er at vi representerer settene med rotfestede trær, der hver node inneholder ett medlem og hvert tre representerer ett sett.

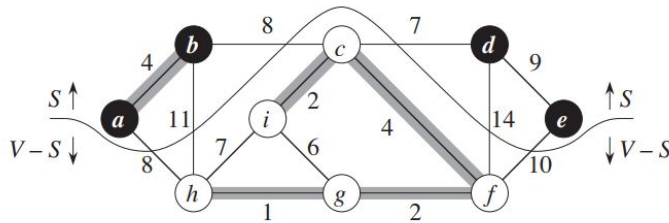
- I en *disjunkt-sett skog* peker hvert element kun til sin forelder. Roten i hvert tre innholder representativen og sin egen forelder.
- Vi utfører de tre disjunkt-sett operasjonene følgende.
 - Operasjonen *Make-Set* lager helt enkelt et tre med kun en node.
 - Vi bruker *Find-Set* ved å følge forelder-pekerne helt til vi finner roten av treet.
 - *Union* operasjonen får roten til det ene treet til å peke til roten til det andre.
- Kjøretid $O(m \lg n)$
 - Der n er antall MAKE-SET operasjoner, m er totalt antall MAKE-SET, UNION og FIND-SET operasjoner. Antar at de n MAKE-SET operasjonene er de første n operasjonene som blir gjort

[I2] Vite hva spenltrær og minimale spenltrær er

- *Spenltrær*
 - Et tre som spenner grafen G . Blir formet av et asyklisk subset som kobler sammen alle noder.
- *Minimale spenltrær (MST)*
 - Det rimeligste spennertreet av alle mulige. Kan finnes flere.
 - Ingen sykler, alle noder involvert, lavest total kantsum som mulig
 - Hvis alle kantene i en sammenhengende graf har forskjellige vekter, vil grafen kun ha ett minimalt spennertre.
- Et minimalt spennertre er et tre som er innoen alle nodene nøyaktig én gang, og som har den lavest mulige kombinerte kantvekten.

[I3] Forstå GENERIC-MST (!)

- Legger til en og en kant i det minimale spennertreet, så lenge det ikke bryter med invarianten (ingen sykler)
- Bruker grådighet
- Kan ha negative kanter
- Input:
 - En urettet graf $G=(V,E)$ og en vektfunksjon $w = E \rightarrow \mathbb{R}$
- Output:
 - En asyklisk delmengde $T \subseteq E$ (bytte til mengdetegn) som kobler sammen nodene i V og som minimerer vektsummen $w(T) = \sum_{(u,v) \in T} w(u,v)$
- Vi utvider en kantmengde (partiell løsning) gradvis
- Invariant: Kantmengde utgjør en del av et minimalt spennertre
- Et kutt $(S, V-S)$, og sier at en kant *krysser* kuttet, dersom nodene den tilhører er i hver sin del av settet etter kuttet
- Sier at et kutt respekterer et sett A av kanter dersom ingen kanter i A krysser kuttet
- En kant er en lett kant som krysser et kutt dersom dens vekt er minimumet av enhver kant i kuttet



En måte å se et kutt av grafen på forrige figur på. De svarte nodene er i settet S , og de hvite er i settet $V - S$. (d, c) er den unike lette kanten.

- En “trygg kant” er en kant som bevarer invarianten
 - La $G = (V, E)$ være en sammenhengende, urettet graf med vekter definert på E .
 - La A være et subsett av E som er inkludert et minimalt spennetre for G , la $(S, V - S)$ være et kutt i G som respekterer A , og la (u, v) være en lett kant som krysser $(S, V - S)$.
 - Da er kanten (u, v) en trygg kant for A . Derfor er lette kanter, trygge kanter.
- Snitt i Kruskal og Prim:
 - Kruskal - Snittet går rundt det treet som har den billigste kanten mellom to forskjellige trær
 - Prim - Snittet går mellom noene i treet og resten av nodene
- “Exchange argument”
 - Ta en optimal (eller vilkårlig) løsning som ikke har valgt grådig
 - Vis at vi kan endre til det grådige valget uten å få en dårligere løsning
 - En lett kant over et snitt som respekterer løsningen vår er trygg
 - Vi kan løse problemet grådig!

GENERIC-MST(G, w)

```

1   $A = \emptyset$ 
2  while  $A$  does not form a spanning tree
3      find an edge  $(u, v)$  that is safe for  $A$ 
4       $A = A \cup \{(u, v)\}$ 
5  return  $A$ 
```

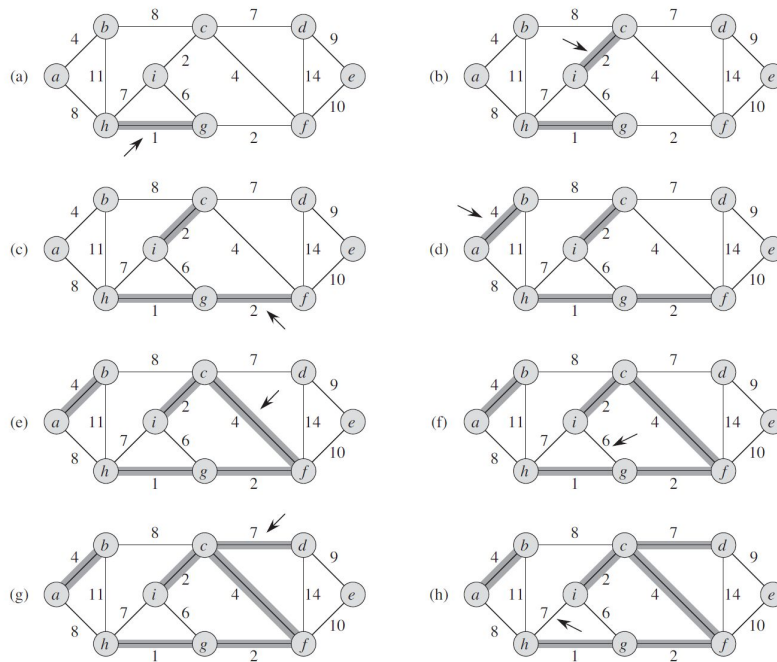
[I4] Forstå hvorfor lette kanter er trygge kanter

- En lett kant er den kanten mellom et subset S og $V-S$ som har lavest verdi/kostnad
- Denne er trygg fordi den bevarer invarianten
 - Vi får et MST ved å koble disse sammen, og har ingen sykler

[I5] Forstå MST-KRUSKAL

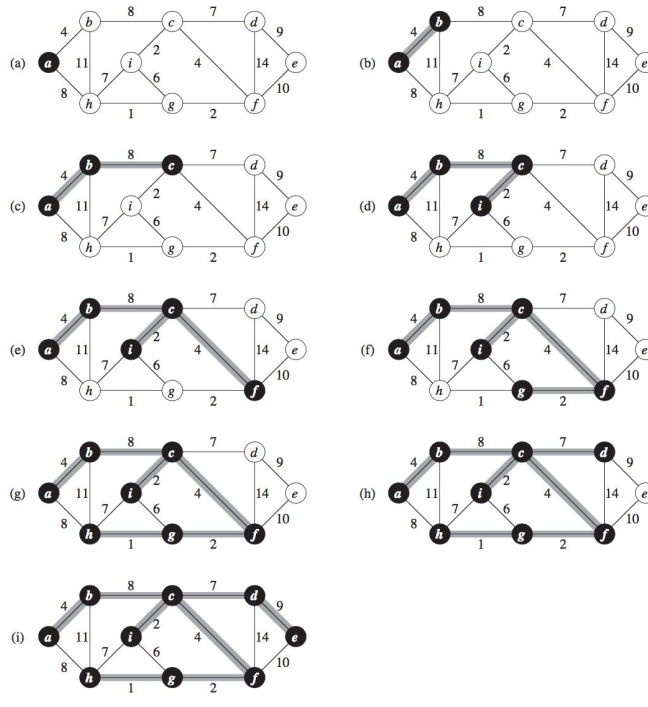
- Jævlig koselig algoritme, it's bringing sexy back
- En kant med minimal vekt blant de gjenværende er trygg så lenge den ikke danner sykler
 - Altså velg den kanten med minst vekt uansett hvor i nodekartet det befinner seg
 - Hvis noen kantvekter er like (små), så velger man etter hva som er oppgitt (feks leksikografisk/alfabetisk)

- Gjør dette i hele treet, men pass på å ikke lage sykler
 - Tilslutt har man et MST
- Grådig algoritme
- Kjøretid
 - Average $O(|E| \lg |V|)$



[I6] Forstå MST-PRIM

- Bygger ett tre gradvis; en lett kant over snittet rundt treet er alltid trygg
- → Kan implementeres vha. traversering
- → For enkelhets skyld: legg alle noder inn fra starten, med uendelig dårlig prioritet
- (1) Begynn i en node
- (2) For alle ikke-valgte noder, hold rede på billigste kant som knytter tre til node
- (3) Hvert steg, velg den noden det er billigst å gå til
- Bruker min-prioritets-kø
 - Har noder i sin kø
- Farging som for BFS:
 - kanter mellom svarte noder er endelige
 - beste kanter for grå noder også uthevet
 - boka uthever bare kantene i spenntreet
- Grådig algoritme
- Kjøretid $O(V \lg V)$
 - Ved binær heap $O(E \lg V)$
 - Ved Fibonacci heap $O(V \lg V + E)$



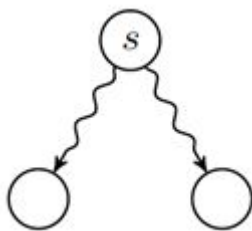
10 Korteste vei fra én til alle

Bredde-først-søk kan finne stier med færrest mulige kanter, men hva om kantene har ulik lengde? Det generelle problemet er uløst, men vi kan løse problemet med gradvis bedre kjøretid for grafer (i) uten negative sykler; (ii) uten negative kanter; og (iii) uten sykler. Og vi bruker samme prinsipp for alle tre!

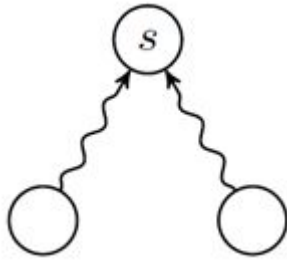
[J1] Forstå ulike varianter av korteste-vei- eller korteste-sti-problemet

(Single-source, single-destination, single-pair, all-pairs)

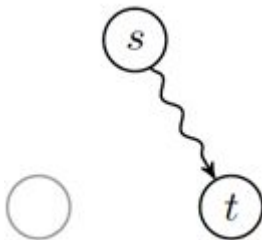
- Single-source shortest path (en til alle)



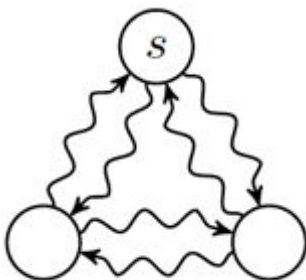
-
- Single destination (alle til en) - løses som SSSP i omvendt graf
 - Ved å reversere retingen til hver kant i grafen, kan vi redusere dette problemet til et *single-source problem*



-
- Single-pair shortest path (en til en) - Har ikke noe bedre enn SSSP
 - Løser det som SSSP
 - Alle kjente algoritmer for dette problemet har samme *worst-case* kjøretid som den beste single-source algoritmen.



-
- All-pairs shortest path (alle til alle)
 - Vi kan løse dette problemet ved å kjøre en *single-source* algoritme en gang fra hver node, men vi kan i mange tilfeller løse den raskere.



•

[J2] Forstå strukturen til korteste-vei-problemet

- I et korteste vei problem blir vi gitt en vektet, rettet graf $G = (V, E)$, med en vektfunksjon $w : E \rightarrow \mathbb{R}$ som mapper vektene til et sett kanter. Vekten $w(p)$ av veien $p = \langle v_0, v_1, \dots, v_k \rangle$ er summen av vektene til kantene på veien: $w(p) = \sum w(v_{i-1}, v_i)$.
- **Input:**
 - En rettet graf $G=(V,E)$, vektfunksjon $w:E \rightarrow \mathbb{R}$ og node $s \in V$.
- **Output:**
 - For hver node $v \in V$, en sti $p = \langle v_0, v_1, \dots, v_k \rangle$ med $v_0=s$ og $v_k=v$, som har minimal vektsum:
$$w(p) = \sum_{i=1}^k w(v_{i-1}, v_i) .$$
-
- Vi kaller også w lengde: $\delta(s, v) = w(p)$ er avstanden fra s til v .

- En korteste vei fra noden u til noden v er definert som enhver vei p med vekt $w(p) = \delta(u,v)$.
- Korteste-vei algoritmer avhenger typisk av egenskapen om at en korteste vei mellom to noder inneholder andre korteste veier innad.
 - Merk at optimal substruktur er en av nøkkelindikatorene på at dynamisk programmering og den grådige metoden muligens tar sted.
- Delveier av korteste veier er korteste veier
- Noen instanser i SSSP-problemet kan inkludere negative kanter

[J3] Forstå at negative sykler gir mening for korteste enkle vei (simple path)

- Kan ikke ha en korteste enkle vei med en negativ sykel.
 - Da vil korteste vei bare gå i en negativ sykel i uendeligheten, fordi den vil jo finne "korteste" vei
- En negativ sykel kan ikke være koblet opp mot "kilden", eller punktet som vi prøver å komme til.
- Hvis det er en negativ sykel som er koblet til kilden så sier vi at det finnes en korteste vei som er negativ uendelig.
- Hvis en negativ sykel ikke er koblet opp mot kilden så sier vi at det at denne syklen gir et tall som er positiv uendelig.
 - Fordi vi initierer først alle noder med uendelig, og siden veien fra kilden er den som gir tall til noder så vil denne negative sykel være uendelig i all evighet.
- Kan heller ikke inneholde en positiv sykel
 - Dersom man fjerner syklen ville man fått en enda kortere vei med samme kilde og destinasjon
 - Dersom vi har en sykel med vekt 0, vil det fortsatt finnes en korteste vei uten denne syklen.
 - Derfor sier vi at når vi finner korteste vei, har de ingen sykler, de er enkle veier
 - Kan altså ha positiv sykel i grafen, men en slik sykel vil ikke velges som korteste vei, da den øker hver runde i syklen man går. Det blir dust å gjøre.

[J4] Forstå at korteste enkle vei er "ekvivalent" med lengste enkle vei

- Altså at det er samme type problem, ikke ekvivalent i ordets betydning
- En enkel sti er en sti uten sykler
- En korteste sti er alltid enkel
- Negativ sykel? Ingen sti er kortest
- Det finnes fortsatt en kortest enkel sti
- Å finne den effektivt: uløst (NP-hardt)
- Korteste enkle vei er ekvivalent med lengste enkle vei da man bare kan velge de lengste veiene istedet for de korteste
 - Ganger alle vektene med -1
- Viktig å se på forskjellen korteste enkle vei og korteste vei
 - Korteste og lengste vei vil være forskjellig, men korteste enkle vei og lengste enkle vei vil være samme med motsatt fortegn

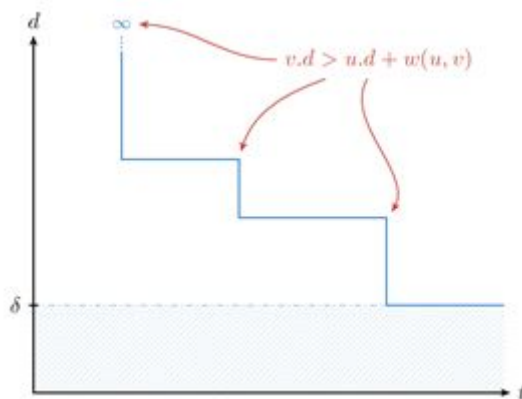
- Poenget med dette er at transformasjonen er såpass billig at om vi hadde klart å løse ett av problemene effektivt, så kunne vi automatisk ha løst det andre.
- Men lengste enkle vei er et uløst problem – og det impliserer at korteste enkle vei også er uløst (mer spesifikt, NP-hardt).

[J5] Forstå hvordan man kan representere et korteste-vei-tre

- Representerer korteste veier likt som et BF-tre.
 - Har for hver node en forgjenger som er en annen node eller NIL
 - Får en kjede av forgjengere langs korteste vei
 - Ulikt fra BF-tre ved at den inneholder korteste veier fra kilden definert på kant-vekter, og ikke antall kanter
- Et korteste-vei tre med rot s er en rettet subgraf $G' = (V', E')$, hvor $V' \subseteq V$ og $E' \subseteq E$ slik at:
 - (1) V' er settet med alle noder nåbare fra s i G
 - (2) G' former et rotfestet tre med rot s
 - (3) For alle noder $v \in V'$, er den unike veien fra s til v i G' den korteste veien fra s til v i G

[J6] Forstå kant-slakking (edge relaxation) og RELAX (!)

- Algoritmene brukt for SSSP bruker teknikken slakking/relaxing.
- Hvis veien til node u ($u.d$) pluss lengden av kanten mellom u og v ($w(u,v)$) er kortere enn den opprinnelige veien til v , oppdateres lengden og u blir satt til v sin forgjenger
- $\delta(s, v) \leq v.d$
- \rightarrow avstanden fra s til v : lengden til korteste vei
- \rightarrow et overestimat: det beste vi har funnet så langt!



- Hver gang vi finner en snarvei $s \leadsto u \rightarrow v$, synker estimatet
- Prøv å forbedre overestimatet $v.d$ ved hjelp av $u.d$ og $w(u,v)$

RELAX(u, v, w)

- 1 if $v.d > u.d + w(u, v)$
- 2 $v.d = u.d + w(u, v)$
- 3 $v.\pi = u$

- Forberedelse til “korteste vei ved slakking” fra s til alle:

INITIALIZE-SINGLE-SOURCE(G, s)

1 **for** each vertex $v \in G.V$

2 $v.d = \infty$

3 $v.\pi = \text{NIL}$

• 4 $s.d = 0$

- Dijkstra's algoritme og DAG-Shortest-Path slakker hver kant nøyaktig én gang.
- Bellman-Ford algoritmen slakker hver kant $|V| - 1$ ganger.

[J7] Forstå ulike egenskaper ved korteste veier og slakking (triangle inequality, upper-bound property, no-path property, convergence property, path-relaxation property, predecessor-subgraph property)

- *Triangle inequality*
 - For enhver kant $(u, v) \in E$, har vi at $\delta(s, v) \leq \delta(s, u) + w(u, v)$.
- *Upper-bound property*
 - Vi har alltid at $v.d \leq \delta(s, v)$ for alle noder $v \in V$, og når $v.s$ får verdien $\delta(s, v)$, endres den aldri.
- *No-path property*
 - Dersom det ikke er noen vei fra s til v , da vil vi ha at $v.d = \delta(s, v) = \infty$.
- *Convergence property*
 - Dersom $s \rightsquigarrow u \rightarrow v$ er en korteste vei i G for noen $u, v \in V$, og dersom $u.d = \delta(s, u)$ før enhver slakking av kanten (u, v) , da vil $v.d = \delta(s, v)$ for alltid etterpå.
- *Path-relaxation property*:
 - Om p er en korteste vei fra s til v og vi slakker kantene til p i rekkefølge, så vil v få riktig avstandsestimat. Det gjelder uavhengig om andre slakkinger forekommer, selv om de kommer innimellom.
- *Predecessor-subgraph property*
 - Så fort $v.d = \delta(s, v)$ for alle $v \in V$, er forgjenger delgraf en korteste-vei tre med rot

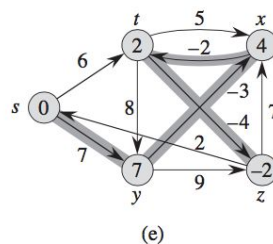
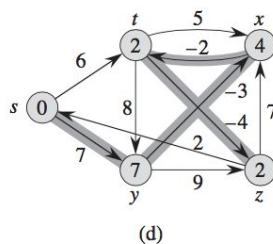
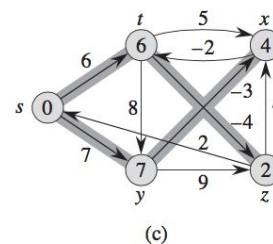
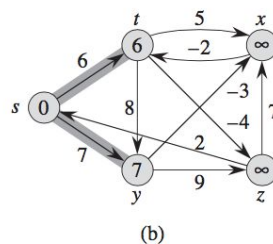
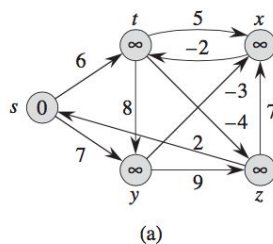
[J8] Forstå BELLMAN-FORD

- Generell algoritme - derfor også ineffektiv
- Slakk alle kantene til det bare *må* bli rett!
- Kan ha negative kanter
- Rekursiv
- Ingen stier har flere enn $|V|-1$ kanter, så går gjennom alle $|V|-1$ ganger. Alle stier må nå ha fått kantene slakket i rekkefølge! Kjører en iterasjon til for å sjekke om det fortsatt finnes snarveier. Isåfall; da har vi kommet borti en negativ sykel. Ellers; svaret vi fant må være rett!
 - Returnerer false hvis negativ sykel oppdaget
 - Returnerer true ellers
- (1) Initialiser algoritmen ved å sette avstand fra startnoden til startnoden lik 0, og alle andre avstander lik $+\infty$
- (2) Sammenlign alle avstander, og oppdater hver node med den nye avstanden.

- (3) Sjekk om det finnes negative sykler
- Kjøretid:
 - Slakker hver kan $|V| - 1$ ganger, $O(E)$
 - Initialisering $O(V)$
 - Totalt: $O(VE)$

Operasjon	Antall	Kjøretid
Initialisering	1	$\Theta(V)$
RELAX	$V - 1$	$\Theta(1)$
RELAX	$O(V)$	$\Theta(1)$

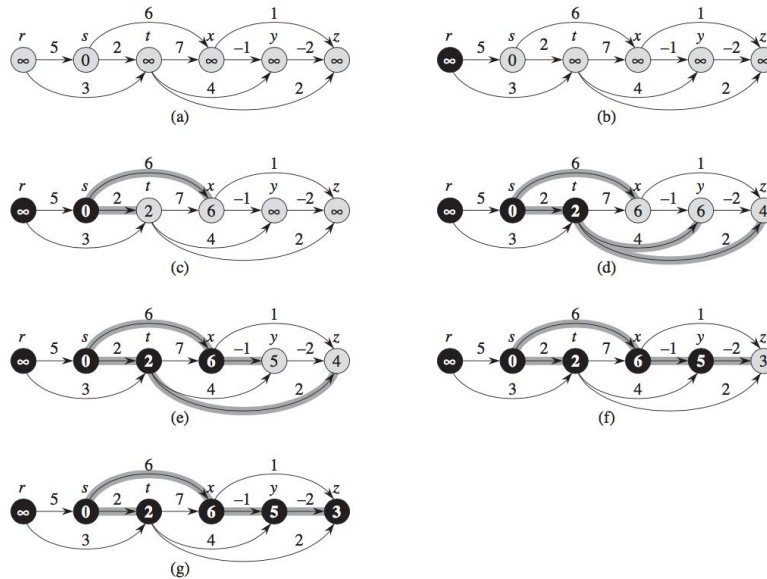
-
- Om et estimat endres, så var tidligere slakking fra noden bortkastet
→ slakk kanter fra v når v.d ikke kan forbedres.



[J9] Forstå DAG-SHORTEST-PATH

- Men hvordan vet vi at v.d ikke kan forbedres? (fra siste punkt om Bellman-Ford)
 - *Strategi 1:* Slakk kanter ut fra noder i topologisk sortert rekkefølge (krever rettet asyklisk graf, DAG). Blir rett fordi når alle inn-kanter er slakket kan ikke noden forbedres, og kan trygt velges som neste. Ved å slakke kantene til en vektet DAG ifølge en topologisk sortering av nodene, kan vi regne ut den korteste veien fra en enkel kilde.
- Korteste vei er godt definert i en DAG, siden det hverken finnes negative kanter eller sykler
- Starter med å topologisk sortere DAG-en til en lineær ordning på nodene. Vi skal bare gå over nodene en gang i den topologiske sorterte rekkefølgen.
- Kan ha negative kanter
- Når vi prosesserer hver node, slakker vi hver kant som forlater noden
- Slakker utkantene til nodene fra venstre mot høyre
 - Slakker hver node nøyaktig én gang
- Kjøretid
 - Topologisk sortering tar $\Theta(V + E)$ tid

- Kallet til INITIALIZE-SINGLE-SOURCE tar $\Theta(V)$ tid
- Totalt: $\Theta(V + E)$



[J10] Forstå koblingen mellom DAG-SHORTEST-PATH og dynamisk programmering (!)

- Korteste vei problemet har optimal delstruktur. Delproblemene er avstanden fra kildenoden til inn-naboer, velg den som gir best resultat
- I DP med memoisering utfører vi implisitt DFS på delproblemene, og får automatisk en topologisk sortering - problemene løses etter delproblemer. Delproblemer er avstanden fra s til inn-naboer, velg den med best resultat.
- Bottom-up - kantslukking av inn-kanter i topologisk sortert rekkefølge (pulling)
 - Gir samme svar: Kantslukking av ut-kanter i topologisk sortert rekkefølge (reaching)

[J11] Forstå DIJKSTRA

- Hvordan vet vi at v.d ikke kan forbedres? (fra siste punkt om Bellman-Ford)
 - Strategi 2: Velg den gjenværende noden med lavest estimat.
 - Gjenværende noder kan bare forbedres ved å slakke fra andre gjenværende noder. Det laveste estimatet kan dermed ikke forbedres
 - Dette gjelder bare hvis vi ikke har negative kanter
- Fungerer bare hvis kantene er ikke-negative
 - Dersom det finnes negative sykler, vil ikke algoritmen terminere som normalt
- Grådig algoritme
- Mest effektiv når det brukes heap
- (1) Initialer avstander. Sett startnoden til 0, og alle andre til $+\infty$
- (2) Initialiser besøkt. Sett startnoden til besøkt, og alle andre til ikke besøkt
- (3) Se på naboene, beregn avstand
- (4) Oppdater avstander
- (5) Gå til den noden med lavest avstandsverdi fra startnoden, og merk den som lest
- (6) Gjenta steg 3-5 helt til alle er besøkt eller en evt sluttnode er besøkt
- Slakker hver node én gang
 - Slakker alle utkantene til den noden v med minst v.d

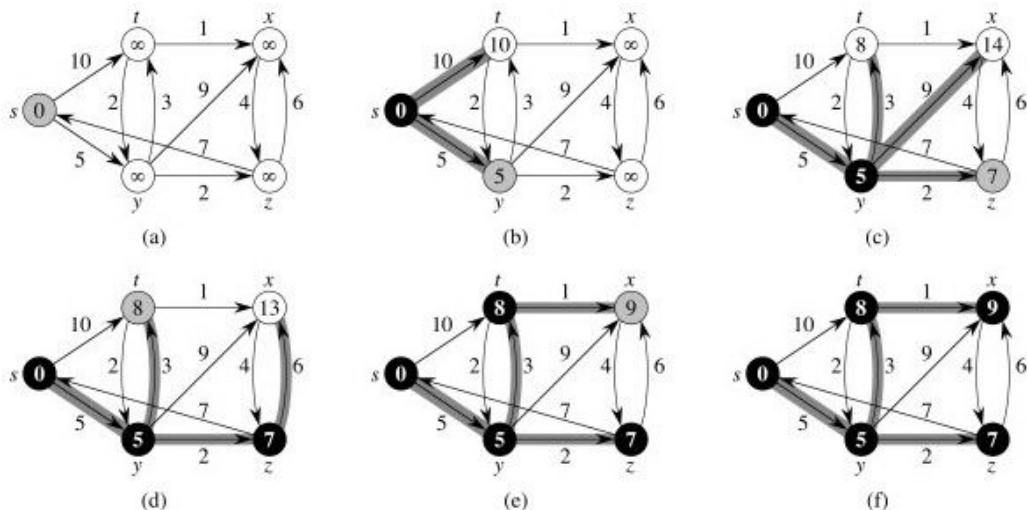
- Kjøretid binær heap - $O(E \lg V + V \lg V)$
 - EXTRACT-MIN bruker $\Theta(\lg V)$ tid for binær heap, $O(V)$ ved tabell
 - Dette skjer V ganger, så får tot $\Theta(V \lg V)$, $O(V^2)$ for tabell
 - RELAX utføres E ganger, for binær heap må man her kalle Decrease-Key dersom RELAX slår til. Kostnad for hver kant er derfor $O(\lg V)$ for binærheap, ved tabell er det $O(1)$
 - Gir totalt $O(E \lg V)$ vs $O(E)$
 - Ved Fib. heap ville totale kjøretid vært $O(V \lg V + E)$
 - Ved array $O(V^2)$

DIJKSTRA(G, w, s)

```

1  INITIALIZE-SINGLE-SOURCE( $G, s$ )
2   $S = \emptyset$ 
3   $Q = G.V$ 
4  while  $Q \neq \emptyset$ 
5       $u = \text{EXTRACT-MIN}(Q)$ 
6       $S = S \cup \{u\}$ 
7      for each vertex  $v \in G.\text{Adj}[u]$ 
8          RELAX( $u, v, w$ )

```



11 Korteste vei fra alle til alle

Vi kan finne de korteste veiene fra hver node etter tur, men mange av delproblemene vil overlappe - om vi har mange nok kanter lønner det seg å bruke dynamisk programmering med dekomponeringen "Skal vi innom k eller ikke?"

[K1] Forstå forgjengerstrukturen for alle-til-alle-varianten av korteste-vei-problemet (PRINT-ALL-PAIRS-SHORTEST-PATH)

- Løser korteste vei alle-til-alle på en nabomatrise
 - Regner da ikke bare ut en matrise med vekter på kantene mellom nodene, men også en forgjengermatrise
- Hvilke noder er i stien fra i til j ?
 - Går da gjennom forgjengerne
- Metode for å printe korteste vei

```

PRINT-ALL-PAIRS-SHORTEST-PATH( $\Pi, i, j$ )
1  if  $i == j$ 
2    print  $i$ 
3  elseif  $\pi_{ij} == \text{NIL}$ 
4    print "no path from"  $i$  "to"  $j$  "exists"
5  else PRINT-ALL-PAIRS-SHORTEST-PATH( $\Pi, i, \pi_{ij}$ )
6    print  $j$ 

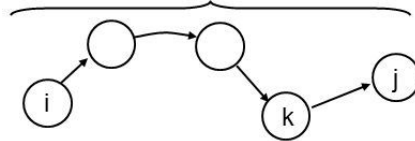
```

- Representert ved $l_{ij}^{(m)} = \min\{l_{ik}^{(m-1)} + w_{kj}\}$

Recursive Solution

- $l_{ij}^{(m)}$ = weight of shortest path $i \rightsquigarrow j$ that contains at most m edges

- $m = 0$: $l_{ij}^{(0)} = \begin{cases} 0 & \text{if } i = j \\ \infty & \text{if } i \neq j \end{cases}$



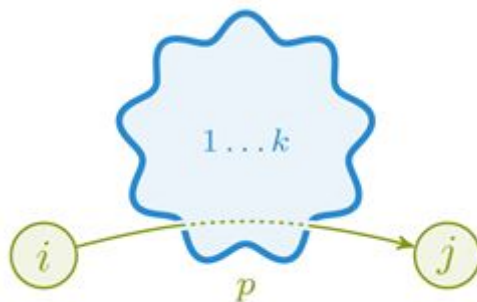
- $m \geq 1$: $l_{ij}^{(m)} = \min \{ l_{ij}^{(m-1)}, \min_{1 \leq k \leq n} \{ l_{ik}^{(m-1)} + w_{kj} \} \}$
 $= \min_{1 \leq k \leq n} \{ l_{ik}^{(m-1)} + w_{kj} \}$
 - Shortest path from i to j with at most $m - 1$ edges
 - Shortest path from i to j containing at most m edges, considering all possible predecessors (k) of j

7

[K2] Forstå FLOYD-WARSHALL

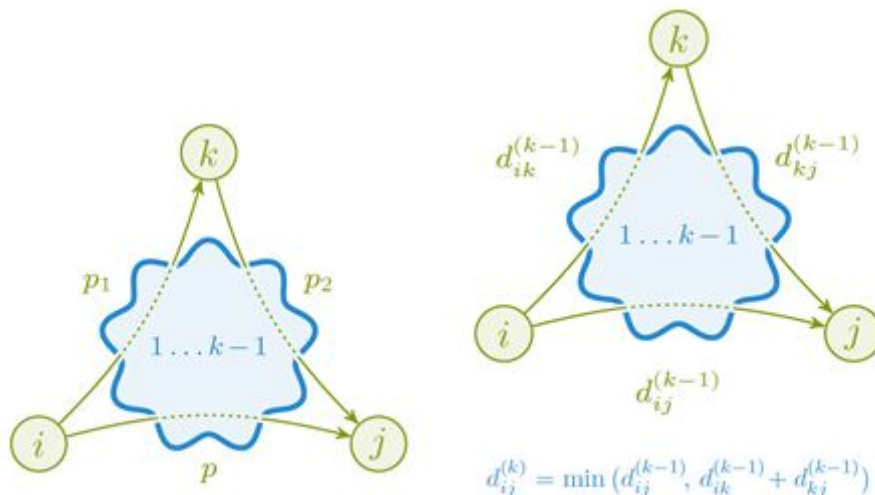
- Dynamisk programmerings algoritme
- Kunne kjørt en-til-alle algoritme på hver node, men vil ta lenger tid:
 - › DIJKSTRA med tabell: $O(V^3 + VE)$
 - › ... med binærhaug: $O(VE \lg V)$
 - › ... med Fib.-haug: $O(V^2 \lg V + VE)$
 - › BELLMAN-FORD: $\Theta(V^2E)$
- Input:

- En vektet, rettet graf $G=(V,E)$ uten negative sykler, der $V=\{1,\dots,n\}$, og vektene er gitt av matrisen $W=(w_{ij})$
- Output:**
 - En $n \times n$ -matrise $D=(d_{ij})$ med avstander, dvs. $d_{ij} = s(i,j)$
- Returnerer en forgjengermatrise $\Pi = (\pi_{ij})$
- Målsetting:**
 - Vil tillate negative kanter
 - Vil ha lavere asymptotisk kjøretid...
 - ...og ha lavere konstantledd
- For hver tildeling av nodene i, j og k sjekker den om det finnes en raskere vei fra i til j som går gjennom k .
- Algoritmen lager matriser $D^{(k)}$ der $D^{(k)}[i][j]$ gir korteste vei fra i til j som kun passerer noder nummerert k eller lavere.
- Fungerer på negative kanter, ikke negative sykler, men kan brukes til å finne negative sykler
- Algoritmen ser etter mellomliggende noder, som gjør at veien mellom to noder blir mindre
- Hva blir "koordinatene" til delproblemene?
 - $\pi_{ij}^{(k)} =$ forgjengeren til j om vi starter i i og går via noder fra $\{1\dots k\}$
 - $d_{ij}^{(k)} =$ korteste vei fra i til j via noder fra $\{1\dots k\}$



○

- Som for ryggsekkproblemet: skal k være med eller ikke?
- Stiene p, p_1 og p_2 går kun via noder fra $\{1\dots k-1\}$



- For d_{ij} : Vi kan ha gått gjennom k i én av delstiene, siden vi blander iterasjoner - men vi antar at det ikke er noen negative sykler, og en positiv sykel vil aldri lønne seg (og vil dermed ikke bli med) => kan bruke én tabell.

$$d_{ij}^{(k)} = \begin{cases} w_{ij} & \text{if } k = 0, \\ \min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}) & \text{if } k \geq 1. \end{cases}$$

•

$$\pi_{ij}^{(0)} = \begin{cases} \text{NIL} & \text{if } i = j \text{ or } w_{ij} = \infty, \\ i & \text{if } i \neq j \text{ and } w_{ij} < \infty. \end{cases}$$

•

$$\pi_{ij}^{(k)} = \begin{cases} \pi_{ij}^{(k-1)} & \text{if } d_{ij}^{(k-1)} \leq d_{ik}^{(k-1)} + d_{kj}^{(k-1)}, \\ \pi_{kj}^{(k-1)} & \text{if } d_{ij}^{(k-1)} > d_{ik}^{(k-1)} + d_{kj}^{(k-1)}. \end{cases}$$

•

•

- For π_{ij} : Baserer seg på valget vi gjør for d_{ij} => holder med én tabell
- FLOYD-WARSHALL(W):

FLOYD-WARSHALL(W)

1 $n = W.rows$

2 $D^{(0)} = W$

3 **for** $k = 1$ **to** n

4 let $D^{(k)} = (d_{ij}^{(k)})$ be a new $n \times n$ matrix

5 **for** $i = 1$ **to** n

6 **for** $j = 1$ **to** n

7 $d_{ij}^{(k)} = \min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)})$

8 **return** $D^{(n)}$

•

- For hver node i og j ; hva er korteste vei $i \leadsto j$?
- Forenklet: Bruker bare D heller enn $D^{(k)}$

FLOYD-WARSHALL'(W)

1 $n = W.rows$

2 initialize D and Π

3 **for** $k = 1$ **to** n

4 **for** $i = 1$ **to** n

5 **for** $j = 1$ **to** n

6 **if** $d_{ij} > d_{ik} + d_{kj}$

7 $d_{ij} = d_{ik} + d_{kj}$

8 $\pi_{ij} = \pi_{kj}$

•

- Kjøretid: $\Theta(n^3)/(V^3)$
 - Kjører tre for-løkker på størrelse $|V|$

- Det er $|V|$ noder vi skal gå igjennom, og for hver node kan man variere startnoden med $|V| - 1$ muligheter, og sluttnoden $|V| - 2$ muligheter.
- Dijkstra bruker også $O(V^3)$ på alle-til-alle, men operasjonene per ledd i Floyd-Warshall er så mye mindre at denne vil lønne seg.

$$D^{(0)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \Pi^{(0)} = \begin{pmatrix} \text{NIL} & 1 & 1 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & \text{NIL} & \text{NIL} \\ 4 & \text{NIL} & 4 & \text{NIL} & \text{NIL} \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(1)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \Pi^{(1)} = \begin{pmatrix} \text{NIL} & 1 & 1 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & \text{NIL} & \text{NIL} \\ 4 & 1 & 4 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(2)} = \begin{pmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \Pi^{(2)} = \begin{pmatrix} \text{NIL} & 1 & 1 & 2 & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & 2 & 2 \\ 4 & 1 & 4 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(3)} = \begin{pmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & -1 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \Pi^{(3)} = \begin{pmatrix} \text{NIL} & 1 & 1 & 2 & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & 2 & 2 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(4)} = \begin{pmatrix} 0 & 3 & -1 & 4 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix} \quad \Pi^{(4)} = \begin{pmatrix} \text{NIL} & 1 & 4 & 2 & 1 \\ 4 & \text{NIL} & 4 & 2 & 1 \\ 4 & 3 & \text{NIL} & 2 & 1 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ 4 & 3 & 4 & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(5)} = \begin{pmatrix} 0 & 1 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix} \quad \Pi^{(5)} = \begin{pmatrix} \text{NIL} & 3 & 4 & 5 & 1 \\ 4 & \text{NIL} & 4 & 2 & 1 \\ 4 & 3 & \text{NIL} & 2 & 1 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ 4 & 3 & 4 & 5 & \text{NIL} \end{pmatrix}$$

•

[K3] Forstå TRANSITIVE-CLOSURE

- Samme som Floyd-Warshall (TC er egentlig Warshall), men kun opptatt av om det finnes en sti eller ikke, bryr seg ikke om kantvektene
- Lager en binærmatrix, med 0 og 1 for hhv ikke sti og sti. Hvis sti fra A til B $\rightarrow 1$
- *Input:*
 - En rettet graf $G=(V,E)$
- *Output:*
 - En rettet graf $G^*=(V,E^*)$ der $(i,j) \in E^*$ hvis og bare hvis det finnes en sti fra i til j i G
- \rightarrow Vi kommer til å returnere en nabomatrix T for G^* .
- Traversér fra hver node

- Kjøretid: $V \times \Theta(E+V) = \Theta(VE+V^2)$
- Bra når vi har få kanter, f.eks. $E = o(V^2)$
- Mye overhead; høye konstantledd
- Målsetting:
 - Vi fokuserer på tilfellet $E = \Theta(V^2)$
 - Vil ha lavere konstantledd
- Observasjon:
 - Korteste stier har felles segmenter
 - Overlappende delproblemer

$$t_{ij}^{(k)} = t_{ij}^{(k-1)} \vee (t_{ik}^{(k-1)} \wedge t_{kj}^{(k-1)})$$

-
- Det er trygt å bruke én tabell. Fordi det ikke er farlig at resultatet kommer fra denne iterasjonen. Dette er fordi det ikke gjør noe om vi går innom k mer enn én gang (sykel); hvis det finnes en sti “med” en sykel, så finnes det også en sti “uten” en sykel!

- TRANSITIVE-CLOSURE(G):

```

TRANSITIVE-CLOSURE( $G$ )
1   $n = |G.V|$ 
2  let  $T^{(0)} = (t_{ij}^{(0)})$  be a new  $n \times n$  matrix
3  for  $i = 1$  to  $n$ 
4      for  $j = 1$  to  $n$ 
5          if  $i == j$  or  $(i, j) \in G.E$ 
6               $t_{ij}^{(0)} = 1$ 
7          else  $t_{ij}^{(0)} = 0$ 
8  for  $k = 1$  to  $n$ 
9      let  $T^{(k)} = (t_{ij}^{(k)})$  be a new  $n \times n$  matrix
10     for  $i = 1$  to  $n$ 
11         for  $j = 1$  to  $n$ 
12              $t_{ij}^{(k)} = t_{ij}^{(k-1)} \vee (t_{ik}^{(k-1)} \wedge t_{kj}^{(k-1)})$ 
13 return  $T^{(n)}$ 

```

- - Kan endre linje 5 til “if (i,j) is in $G.E$ ” vil det være mulig å bruke algoritmen til å finne ut om en rettet graf er asyklisk. Diagonalen er normalt kun 1ere, men ved denne endringen vil diagonalen bli satt til bare 0ere. Når man da finner en sykel, får man 1 på diagonalen
- For hver node i og j ; finnes det en sti $i \rightsquigarrow j$?
- Forenklet utgave: Bruker bare en tabell, T

```

TRANSITIVE-CLOSURE'(G)
1   $n = |G.V|$ 
2  initialize T
3  for  $k = 1$  to  $n$ 
4      for  $i = 1$  to  $n$ 
5          for  $j = 1$  to  $n$ 
6               $t_{ij} = t_{ij} \vee (t_{ik} \wedge t_{kj})$ 
7  return T

```

-
- Kjøretid: $\Theta(n^3)$

12 Maksimal flyt

Et stort skritt i retning av generell lineær optimering (såkalt lineær programmering). Her ser vi på to tilsynelatende forskjellige problemer, som viser seg å være *duale* av hverandre, noe som hjelper oss med å finne en løsning.

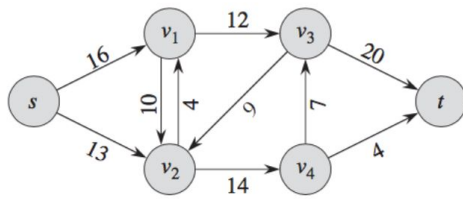
[L1] Kunne definere flytnettverk, flyt og maks-flyt-problemet

- Flyt kan visualiseres ved for eksempel et rørsystem for å levere vann i en by, eller som et nettverk med ulike kapasitet på de ulike kablene. Maks flyt er hvor mye som faktisk strømmer gjennom nettverket.
- Det kan finnes kanter med veldig liten kapasitet som hindrer flyt, så uansett om alle de andre kantene har stor kapasitet, vil maks flyt avhenge av den minste kanten dersom det ikke er noen vei rundt den.
- Maksimal flyt er nådd hvis og bare hvis residualnettverket ikke har flere flytforøkende stier.
- Flytnettverk: Rettet graf $G=(V,E)$
 - Kapasiteter $c(u,v) \geq 0$
 - Kanter som ikke finnes har ingen kapasitet
 - Kilde og sluk $s, t \in V$
 - Forenklet antagelse: Alle noder er på en sti fra s til t
 - $v \in V \Rightarrow s \rightsquigarrow v \rightsquigarrow t$
 - Ingen løkker (self-loops), Merk: vi kan ha sykler
 - Tillater ikke antiparallelle kanter
 - $(u,v) \in E \Rightarrow (v,u) \notin E$
 - $(u,v) \notin E \Rightarrow c(u,v)=0$
- Flyt: En funksjon $f: V \times V \rightarrow \mathbb{R}$
 - Flyt inn i en node = flyt ut av en node
 - Flyt ut av en kilde = flyt inn i et sluk
 - $0 \leq f(u,v) \leq c(u,v)$
 - $u \neq s, t \Rightarrow \sum_v f(v,u) = \sum_v f(u,v)$
- Flytverdi: $|f| = \sum_v f(s,v) - \sum_v f(v,s)$

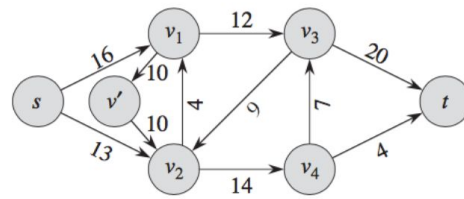
- Maks-flyt problemet:
 - *Input*: et flytnettverk G
 - *Output*: En flyt f for G med maks flyt, $|f|$

[L2] Kunne håndtere antiparallelle kanter og flere kilder og sluk

- Antiparallelle kanter
 - At det er rettede kanter. Hvis et flytnettverk er laget med kant fra a til b , har vi ikke en kant b til a
 - Splitt den ene med en node
 - Sett kant fra node a til x , og x til b i stedet, der x er en ny node
 - Kapasiteten til de to nye kantene er lik kapasiteten til originalkanten

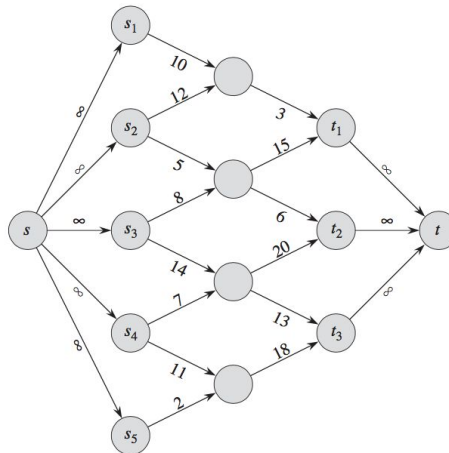
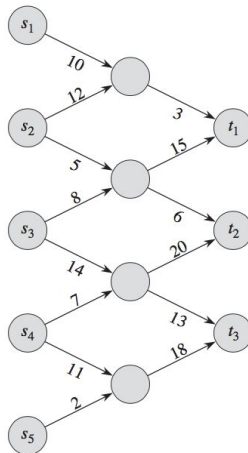


(a)



(b)

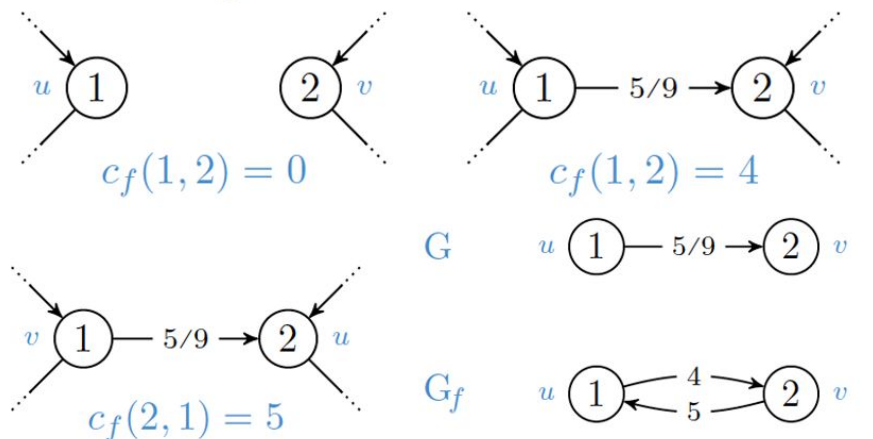
- Flere kilder og sluk
 - Legg til super-kilde og super-sluk



[L3] Kunne definere residualnettverket til et nettverk med en gitt flyt (!)

- Restnettverk:
 - Kan sende tilbake flyt gjennom en kant motsatt vei, hvis dette vil øke flyten i nettverket generelt
 - Disse reverserte kantene i restnettverket lar algoritmen sende tilbake flyt som den allerede har sendt langs kanten.
 - Det er ekvivalent med å senke flyten på kanten.
 - Å sende flyt langs en kant, der det allerede går flyt, i et restnettverk er også kjent som oppheving.
 - Det er dette bakoverkantene i restnettverket representerer.
 - Engelsk: residual network
 - Det sendes 5 flyt

- Fremoverkant: ved ledig kapasitet, kapasitet-flyt = 4 sendes
- Bakoverkant: 5 flyt
- Altså sender man motsatt verdi den samme veien. Hvis 5 sendes i flytnettverk, sendes 4 den samme veien i restnettverket, og 5 andre vei.



-
- Enkleste tilfellet: en av kantene i flytnettverket har ledig kapasitet så har vi også en kant i restnettverket som har lik kapasitet.

[L4] Forstå hvordan man kan oppheve (cancel) flyt

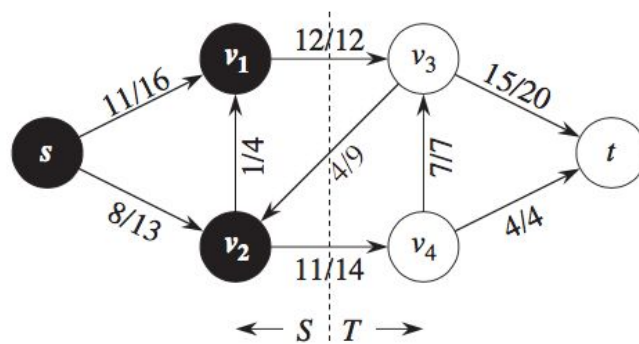
- Flytoppheving:
 - Vi kan "sende" flyt baklengs langs kanter der det allerede går flyt
 - Vi opphever da flyten, så den kan omdirigeres til et annet sted
 - Det er dette bakoverkantene i restnettverket representerer

[L5] Forstå hva en forøkende sti (augmenting path) er

- En sti fra kilde til sluk som lar oss øke flyten
 - En sti fra kilde til sluk i restnettverket
 - Langs fremoverkant: flyten kan økes
 - Langs bakoverkant: flyten kan omdirigeres
 - Altså: en sti der den totale flyten kan økes
- Gitt et flytnettverk som antas å ha maks flyt, kan dette sjekkes ved å kjøre Ford-Fulkerson eller Edmonds-Karp og se om man finner en flytforøkende sti. Gjør man det så var ikke flyten optimal i utgangspunktet

[L6] Forstå hva snitt, snitt-kapasitet og minimalt snitt er

- Snitt i flytnettverk: Partisjon (S, T) av V
- \rightarrow s i S og t i T
- \rightarrow Netto-flyt:
 - Lik uansett hvor du kutter



-
- Nettoflyten lags kuttet blir: $f(S, T) = f(v_1, v_3) + f(v_2, v_4) - f(v_3, v_2) = 12 + 11 - 4 = 19$
- Kapasiteten blir da: $c(S, T) = c(v_1, v_3) + c(v_2, v_4) = 12 + 14 = 26$
- Minimalt snitt
 - Et minimalt snitt i et nettverk er et snitt der kapasiteten er minst av alle snitt av nettverkene

[L7] Forstå maks-flyt/min-snitt-teoremet (!)

- Dersom f er en flyt i et flytnettverk G med kilde s og sluk t , da er de følgende forholdene ekvivalente
 - (1) f er en maksimal flyt i G
 - (2) Restnettverket G_f har ingen økende stier
 - (3) $|f| = c(S, T)$ for et snitt (S, T) av G .
- Et minimalt kutt vil kutte der det går minst kapasitet - og det gir mening fordi den minste kapasiteten i et flytnettverk er begrensningen på maks flyt gjennom et nettverk
- Maks flyt = (kapasiteten til) min snitt

[L8] Forstå FORD-FULKERSON-METHOD og FORD-FULKERSON

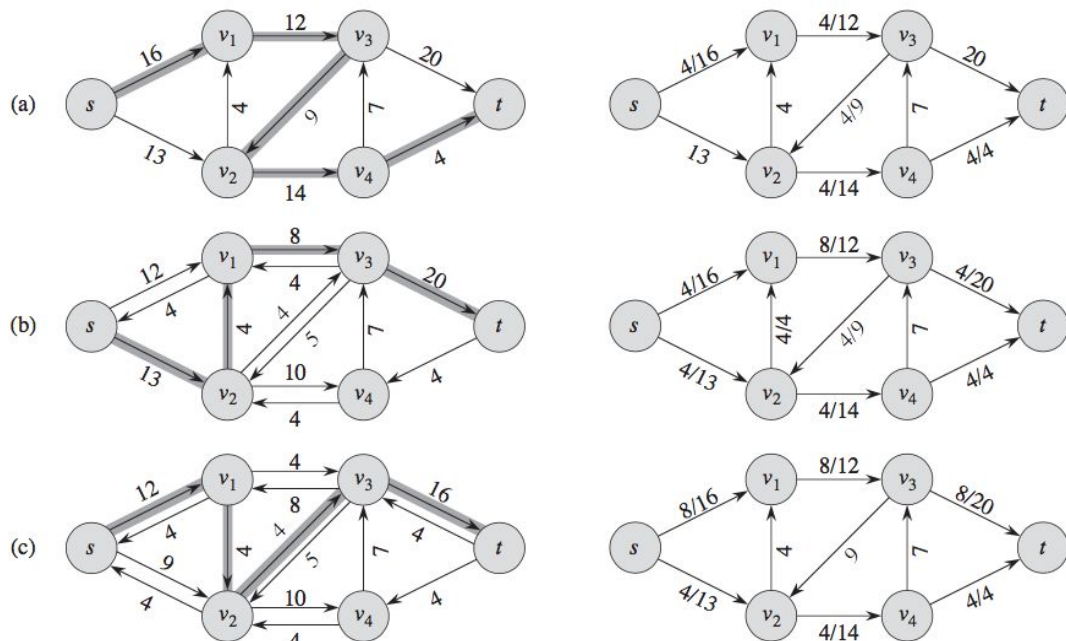
- Ford-Fulkerson:
 - Velger litt random en sti
 - Finn økende stier så lenge det går
 - Deretter er flyten maksimal
 - Om vi bruker BFS: "Edmonds-Karp"
- Normal implementasjon:
 - (1) Finn økende sti først
 - (2) Finn så flaskehalsen i stien
 - (3) Oppdater flyt langs stien med denne verdien


```

FORD-FULKERSON( $G, s, t$ )
1  for each edge  $(u, v) \in G.E$ 
2       $(u, v).f = 0$ 
3  while there is a path  $p$  from  $s$  to  $t$  in  $G_f$ 
4       $c_f(p) = \min \{c_f(u, v) : (u, v) \text{ is in } p\}$ 
5      for each edge  $(u, v)$  in  $p$ 
6          if  $(u, v) \in E$ 
7               $(u, v).f = (u, v).f + c_f(p)$ 
8          else  $(v, u).f = (v, u).f - c_f(p)$ 

```

-
- Kjøretid $O(E |f^*|)$
 - Dersom vi sier at f^* gir oss den maksimale flyten som vi kan oppnå.
 - Da vil vi på det meste kjøre while-løkken for å finne en forøkende sta, $|f^*|$ ganger, da flyten f må øke med minst en enhet av gangen.
 - Hver iterasjon av while-løkken tar $O(E)$ tid, samme gjør initialiseringen på linje 1-2.



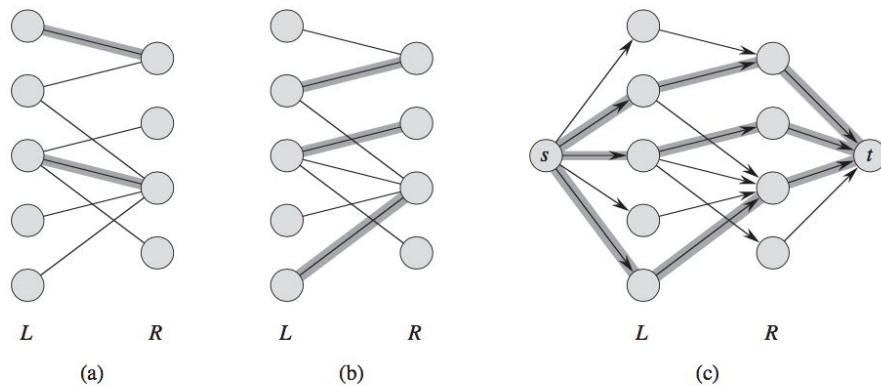
[L9] Vite at FORD-FULKERSON med BFS kalles Edmonds-Karp-algoritmen

- Kan forbedre grensen på Ford-Fulkerson ved å finne en forøkende sti med BFS.
 - Velger en forøkende sti som den korteste fra s til t , hvor hver kant har en enhet-vekt. Dette er Edmonds-Karp
- “Flette inn” BFS
 - Finn flaskehalser underveis
 - Hold styr på hvor mye flyt vi får frem til hver node
 - Traverser bare noder vi ikke har nådd frem til enda
- v.a: mulig økning (augmentation)

- Hvor mye mer flyt får vi til å sende fra s til v ? Gitt at flaskehalsen $c_f(s \rightsquigarrow v)$ for en eller annen sti $s \rightsquigarrow v$. Etter traversering er $t.a = c_f(p)$ for en forøkende sti p (eller o).
- Avstander synker ikke i residualnettverket
- En kant (u,v) kan være flaskehals maks annenhver iterasjon: Den forsvinner, og må dukke opp igjen
- Vi velger korteste økende stier:
 - Dermed må v først være 1 kant lenger unna enn u
 - Så, idet (u,v) dukker opp igjen, må u være 1 lenger unna enn v
 - Når (u,v) så er kritisk igjen, har altså avstanden til u økt med minst 2
- Kjøretid
 - Dermed kan vi maks ha $O(VE)$ iterasjoner
 - Kjøretid for hver iterasjon er $O(E)$
 - Kjøretid $O(VE^2)$

[L10] Forstå hvordan maks-flyt kan finne en maksimum bipartitt matching

- Matching: sier at en node v er matchet av matchingen M dersom en node i i M har v i seg, hvis ikke er v unmatched
- Maksimum matching er dermed matching med maksimum kardinalitet, altså flest mulig kanter, $|M|$ størst mulig
- Vi kan se på problemet som at vi har n antall nyredonorer, også har vi m pasienter som venter på en nyre. Det vi skal finne ut, er det maksimale antall med matcher, det vil si maksimale antall personer som kan få en nyre.
- Vi lar da nodene i R representere donorene, og L representere pasientene, og kantene mellom dem representerer om nyrene er kompatibel med pasienten.
- *Matching*: Delmengde $M \subseteq E$ for en urettet graf $G=(V,E)$
- \rightarrow Ingen av kantene i M deler noder
- \rightarrow *Bipartite matching*: M matcher partisjonene
- Bipartitt matching
 - *Input*: En bipartite urettet graf $G=(V,E)$
 - *Output*: En matching $M \subseteq E$ med flest mulig kanter, dvs., der $|M|$ er maksimal
 - Kan løses med Ford-Fulkerson i $O(VE)$ tid
 - Alle kapasiteter er 1
 - Edmonds-Karp vil finne korteste vei. Etter hvert vil korteste vei kunne bli ganske lang, for å opprettholde maks flyt



[L11] Forstå heltallsteoremet (integrality theorem) (!)

- For heltallskapasiteter gir Ford-Fulkerson heltallsflyt
- Dersom kapasitetsfunksjonen c kun tar på seg heltallsverdier, da vil maksimumflyten f produsert av *Ford-Fulkerson-metoden* ha den egenskapen at $|f|$ er en heltall.
- Generelt, vil flyten mellom to noder $f(u,v)$ være et heltall for alle noder u og v .

13 NP-komplett

NP er den enorme klassen av ja-nei-problemer der ethvert ja-svar har et bevis som kan sjekkes i polynomisk tid. Alle problemer i NP kan i polynomisk tid reduseres til de såkalt komplette problemene i NP. Dermed kan ikke disse løses i polynomisk tid, med mindre alt i NP kan det. Ingen har klart det så langt...

[M1] Forstå sammenhengen mellom optimerings- og beslutnings-problemer

- *Problem*: en relasjon mellom input og output
- Input og output kan være vilkårlige abstrakte objekter
- Et problem kalles *konkret* hvis input og output er bitstrenger.
- *Verifikasjonsalgoritme*:
 - Tar inn sertifikat og instans
 - Sjekker om løsningen stemmer, verifiseres hvis 1/JA
 - Kaller gjerne da løsningene et “sertifikat” eller “vitne”
 - Sertifikat: Bitstreng som brukes som bevis for ja-svar
- Optimaliseringsproblem
 - Vil finne den mest optimale løsningen (korteste vei feks)
 - NP-komplett gjelder ikke optimaliseringsproblemer direkte
 - Ikke nødvendigvis noe vitne
- Beslutningsproblem
 - Kan tenke på det som å stille spørsmålet “Finnes det et vitne?”
 - JA/NEI, 0/1
 - Mer generelt: egentlig et hvilket som helst ja/nei-spørsmål
 - Lag beslutningsproblem med terskling. Avgjøres vha. optimering, som da er like vanskelig.

- Selv om NP-komplette problemer er begrenset til et rike beslutningsproblemer, kan vi dra nytte av det praktiske *forholdet* mellom optimaliseringsproblemer og beslutningsproblemer.
- Vi kan vanligvis *caste* et gitt optimaliseringsproblem som et relatert beslutningsproblem ved å legge inn en bundet verdi for å bli optimalisert.
 - For eksempel er et avgjørelsesproblem relatert til *Kortest-vei is Sti*:
 - Gitt en rettet graf G , noder u og v , og et heltall k , eksisterer en sti fra u til v bestående av maksimalt k kanter?
 - Vi vil her kunne løse *Sti* ved å løse *Korteste-vei*, og så sammenligne antall kanter i korteste vei med verdien til beslutningsproblemet k
 - Beslutningsproblemet er *lettere*, eller “ikke vanskeligere”, enn optimaliseringsproblemet.
- Angitt på en måte som er mer relevant for NP-fullstendighet; hvis vi kan bevise at et *beslutningsproblem* er vanskelig, gir vi også bevis for at det relaterte *optimaliseringsproblemet* er vanskelig.
- Kompleksitetsklasse:
 - En mengde språk
 - P: språkene som avgjøres i polynomisk tid
 - Disse problemene kan vi løse i praksis
- Hvis $P=NP$ kan vi finne optimum vha. binær søk med terskelen

[M2] Forstå koding (encoding) av en instans

- En koding av et sett S av abstrakte objekter er en mapping e fra S til et sett med binære strenger
- For at et dataprogram skal klare å løse et abstrakt problem, må vi representere probleminstansene på en måte som programmet skjønner.
- Vi *koder* instanser og resultater som bits
- Bruker koding for å mappe abstrakte problemer til konkrete problemer
 - Gitt et abstrakt beslutningsproblem, kan vi mappe et sett av instanser til et relatert konkret beslutningsproblem
- For et sett av instanser I sier vi at to enkodings e_1 og e_2 er *polynomiske relaterte* dersom det finnes to polynomisk-tid funksjoner f_{12} og f_{21} slik at for hver $i \in I$, har vi at $f_{12}(e_1(i)) = e_2(i)$, og $f_{21}(e_2(i)) = e_1(i)$.

[M3] Forstå hvorfor løsningen vår på 0-1-ryggsekkproblemet ikke er polynomisk

- Den dynamisk programmerte algoritmen for 0-1 knapsack problemet har en kjøretid på $O(nW)$, hvor n er antall elementer og W er den maksimale vekten som knapsack-en kan holde.
- Dette er ikke en polynomisk-tid algoritme for noen fornuftig representasjon av input. I en fornuftig representasjon er alle numeriske verdier (vektene og verdiene, etc.) gitt i binærtall.
- For å representere verdien W , trenger vi $\lg W$ bits. Dermed blir kjøretiden $O(nW)$ eksponentiell i størrelsen til input.
 - Kalles også pseudopolynomisk

[M4] Forstå forskjellen på konkrete og abstrakte problemer

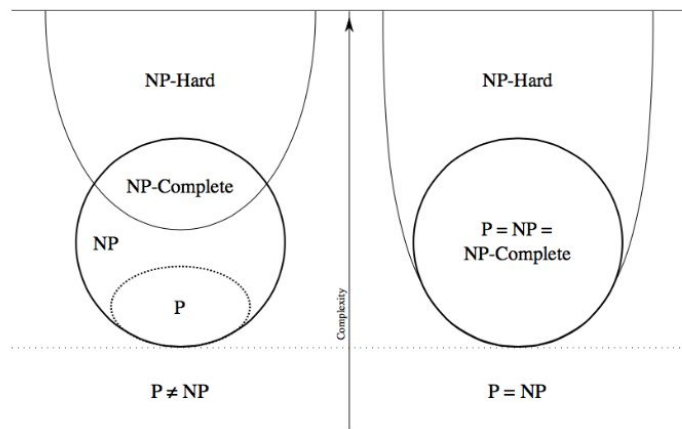
- Et problem kalles *konkret* hvis input og output er bitstrenger.
 - Et konkret problem er polynomisk-tid løsbart dersom det finnes en algoritme som kan løse den på $O(nk)$ tid, for en konstant k .
 - Vi definerer den *komplekse klassen* P som et sett av konkrete beslutningsproblemer som er polynomisk-tid løsbare.
- Abstrakte problemer
 - En funksjon som apper et sett av instanser til et løsningssett
 - For det meste optimaliseringsproblemer

[M5] Forstå representasjonen av beslutningsproblemer som formelle språk

- Et språk L over et alfabet Σ er et sett av strenger laget av symboler i Σ .
- Rammeverket for formelle språk lar oss konsist uttrykke relasjonen mellom beslutningsproblemer og algoritmer som løser de.
- En algoritme A aksepterer en streng s dersom gitt input x gir $A(x) = 1$
- En algoritme avviser en streng dersom $A(x) = 0$
- Et språk er bestemt av en algoritme A dersom hver binærstreng i L er akseptert av A og hver binærstreng ikke i L er avvist av A
- Et språk L er akseptert i polynomisk tid av en algoritme A hvis det er akseptert av A , og hvis det finnes en konstant k slik at for alle strenger med lengde n i L , aksepterer A input x på $O(n^k)$ tid
- Et språk er bestemt i polynomisk tid av en algoritme A , hvis det eksisterer en k slik at for alle strenger x i $\{0,1\}^*$ av lengde n , algoritmen bestemmer at x er i L på $O(n^k)$ tid

[M6] Forstå definisjonen av klassen P

- $P = \text{Polynomial time}$
- Klassen P er slike problemer som kan løses i polynomisk tid. Ekvivalent med å kunne lese bare ja-instanser i polynomisk tid.
- Ethvert problem i P er også i NP , siden dersom et problem er i P kan vi løse det i polynomisk tid, selv uten å bli gitt et vitne.
- $P \subseteq NP$
- $P \subseteq \text{co-}NP$
- Men vet ikke om $P = NP$, i så fall kan man løse alle problemer i NP
- Et språk L er i P hvis det eksisterer en algoritme med følgende egenskaper
 - Den tar inne en bitstreng x som eneste parameter
 - Hvis $x \in L$ så returnerer algoritmen 1
 - Hvis $x \notin L$ så returnerer algoritmen 0
 - Den har polynomisk kjøretid



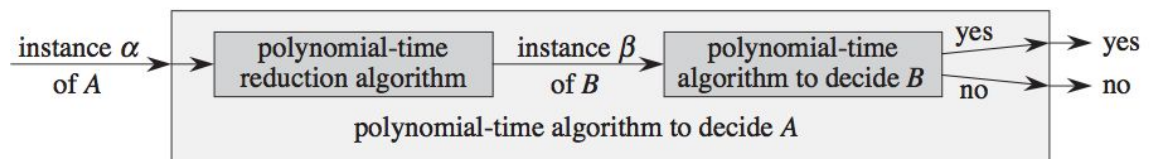
[M7] Forstå definisjonen av klassene NP og co-NP

- NP = Non-deterministic polynomial time
 - Ethvert problem som kan verifiseres i polynomisk tid
 - Blir gitt et vitne på en løsning, kan man bekrefte at vitnet er korrekt
 - Man “kommer opp med” et svar, og sjekker om det er riktig
 - NP: Ja-svar har vitner som kan sjekkes i polynomisk tid
 - Et språk L er i NP hvis og bare hvis det eksisterer en algoritme med følgende egenskaper
 - Den tar inn to bitstrenger x og y
 - Hvis $x \in L$ så skal det eksistere en y som gjør at algoritmen returnerer 1
 - Denne y må ha polynomisk lengde som funksjon av lengden til x
 - Hvis $x \notin L$ så skal det ikke eksistere noen slik y
 - Den har polynomisk kjøretid
- co-NP:
 - Nei-svar har vitner som kan sjekkes i polynomisk tid, falsifiseres
 - “Kommer opp med et svar”, og kan si at det ikke er riktig
 - co-NP er ikke komplementet til NP
 - Et språk L er i co-NP hvis og bare hvis komplementet til L er i NP.
 - Det vil si at det finnes et sertifikat når x ikke er med i L , heller enn når x er med i L

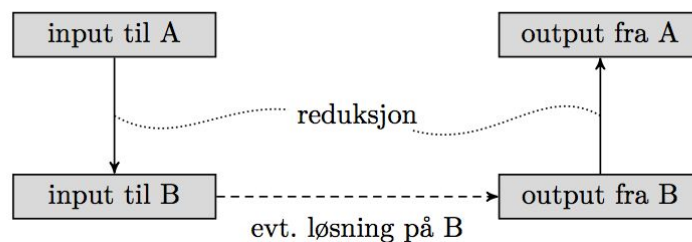
[M8] Forstå redusibilitets-relasjonen \leq_p

- Reduksjoner
 - Sammenligner et kjent problem og et ukjent problem, og sjekker om det ukjente problemet er minst like vanskelig som det kjente
 - Hvis vi reduserer A til et løsbart problem B , må A være løsbart
 - Hvis B -siden er polynomisk, er A -siden polynomisk
 - Motsatt er nødvendigvis ikke riktig, hvis selve reduksjonen er eksponentiell
 - La oss se på et beslutningsproblem A , som vi ønsker å løse i polynomisk tid. Vi kaller inputen til en problem for instansen. La det være slik at vi allerede vet hvordan vi kan løse et annet beslutningsproblem B i polynomisk tid.

- Til sist, la det være slik at vi har en prosedyre som *transformerer* enhver instans α av A til en instans β i B , med følgende egenskaper:
 - Transformasjonen tar polynomisk tid.
 - Svarene er det samme. Det vil si at svaret for α er "ja" hvis og bare hvis svaret for β også er "ja".
- Vi kaller en slik prosedyre i polynomisk tid en reduksjonsalgoritme og det gir oss en måte å løse problem A i polynomisk tid:



-
- (1) Gitt en instans α av problem A , bruker vi en polynomisk reduksjonsalgoritme som transformerer den til en instans β av problem B .
- (2) Kjør beslutningsalgoritmen for B , i polynomisk tid, på instansen β .
- (3) Bruk svaret for β som svar for α



-
- Trekker ut i fra dette følgende beslutninger
 - (1) Hvis vi kan løse B , så kan vi løse A
 - (2) Hvis vi ikke kan løse A , så kan vi ikke løse B
 - (3) Hvis vi ikke kan løse B , sier det ingenting om A
 - (4) Hvis vi kan løse A , sier det ingenting om B
- Hvis A kan reduseres til B i polynomisk tid skriver vi $A \leq_p B$
- Boken (Cormen m.fl) trekker frem et eksempel der førstegradsligningen $ax+b=0$ kan transformeres til $0x^2+ax+b=0$.
- Alle gyldige løsninger for annengradsligningen er også gyldige løsninger for førstegradsligningen.
- Ideen bak eksempelet er å vise at et problem, X , kan reduseres til et problem, Y , slik at inputverdier for X kan løses med Y .
- Kan du redusere X til Y , betyr det at å løse Y krever minst like mye som å løse X , dermed må Y være minst like vanskelig å løse som X .
 - Det er verdt å merke seg at reduksjonen ikke er gjensidig, du kan dermed ikke bruke X til å løse Y

[M9] Forstå definisjonen av NP-hardhet og NP-komplethet (!)

- NPC - Et problem som er i NP og er like vanskelig som ethvert problem i NP
 - Hvis ethvert problem i NPC kan løses i polynomisk tid, så har alle problemene i NP en polynomisk-kjøretid algoritme

- NPH - NP-harde problemer kan ikke løses i polynomisk tid, altså er ikke i NP. Det sies å være en klasse av problemer som er minst like vanskelige som det vanskeligste problemet i NP-klassen.
 - Sagt på en annen måte: problemer som lar seg redusere fra NP-problemer i polynomisk tid, men som ikke nødvendigvis lar seg verifisere i polynomisk tid med en gitt løsning.
 - NP-harde problemer som lar seg verifisere i polynomisk tid kalles NP-komplette, altså de samme type problemene, men som befinner seg i NP
 - Et problem er NP-Hardt dersom alle problemer i NP kan reduseres til det
- Vi prøver ikke å vise at det finnes en effektiv algoritme for å løse problemet, men heller vise at ingen effektiv algoritme eksisterer
- Appellerer kun til beslutningsproblemer, ikke optimaliseringsproblemer
 - Men: kan skrive om optimaliseringsproblem til et beslutningsproblem
 - Optimalisering - Finn korteste vei fra v til u
 - Beslutning - Finnes det en korteste vei fra v til u med k kanter? JA/NEI
 - Optimaliseringsproblemet er “vanskelig”, mens beslutningsproblemet er “hvertfall ikke vanskeligere”
 - Hvis vi kan bevise at et beslutningsproblem er vanskelig, kan vi bruke det til å vise at det tilsvarende optimaliseringsproblemet også er vanskelig
- Antar at et kjent problem A er NPC, kan bevise at B også er NPC (eller vanskeligere) med reduksjon

[M10] Forstå den konvensjonelle hypotesen om forholdet mellom P, NP og NPC

- P vs. NP: Om vi kan løse problemet, så kan vi verifisere det med samme algoritme, og bare ignorere sertifikatet. Dvs. $P \subseteq NP$ og $P \subseteq \text{co-NP}$
- Vi vet ikke om $P = NP \cap \text{co-NP}$
- 4 mulige scenarier; men ingen vet hvem som stemmer!
- Hvis $P = NP$ kan vi svare på om det finnes et vitne, og *rekonstruere* vitnet
- Dersom et NPC problem er løsbart i polynomisk tid, da er $P = NP$
- Motsatt, dersom et problem i NP ikke er løsbart i polynomisk tid, er ingen NPC problem løselige i polynomisk tid
- Alle problemer i NP til reduseres til problemene i NPC i polynomisk tid
- NPC er NP-harde problemer som er i NP

[M11] Forstå beviset for at CIRCUIT-SAT er NP-komplett

- Problem
 - En logisk krets bestående av NOT/INV, ADD og OR gates
 - Problemet er “Gitt en krets med de nevnte portene over, er kretsen tilfredsstilt (blir output 1)”
- Gitt en krets C, kan vi prøve å sjekke om den er tilfredsstilt ved å sjekke alle mulige inputs.
 - Hvis kretsen har k inputs, må vi sjekke 2^k mulige utfall, og det er ikke bra
 - Altså tar checkingen av hver $\Omega(2^k)$ tid, som er superpolynomisk i størrelsen til kretsen
- Dette er NP-komplett, og har vi sterke beviser på at ingen polynomisk algoritme eksisterer som løser Circuit SAT

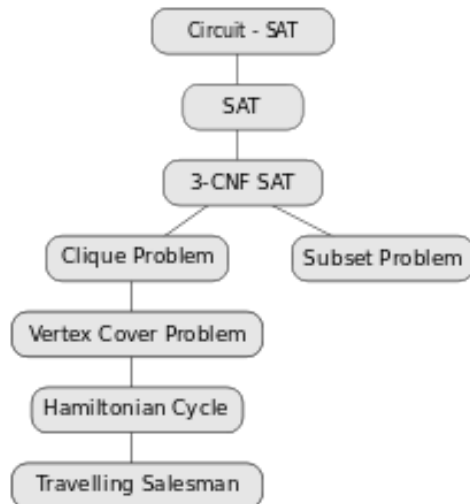
- For å vise at noe er NP-komplett, må vi vise at (1) det er i NP og (2) det er NP-hardt

14 NP-komplette problemer

Om du står overfor et NP-komplett problem, så er det viktig å kunne bevise det. Vi har sett at alt i NP kan reduseres til CIRCUIT-SAT direkte, men vi kan også vise kompletthet indirekte, ved å redusere videre fra CIRCUIT-SAT.

[N1] Forstå hvordan NP-kompletthet kan bevises ved én reduksjon (!)

- Reduserer nedover i hierarkiet - altså fra Ham-cycle til TSP



[N2] Kjenne de NP-komplette problemene CIRCUIT-SAT, SAT, 3-CNF-SAT, CLIQUE, VERTEX-COVER, HAM-CYCLE, TSP og SUBSET-SUM (!)

- **CIRCUIT-SAT** - logisk krets som skal bli 1
- **SAT** - Boolsk formel med n variabler
 - Kan man sette variablene i en kombinasjon slik at formelen gir True
- **3CNF-SAT (Conjunctive normal form)** - logisk formen som består av 3 variabler (mer spesifikt OR av tre literaler)
 - formula = AND of clauses(bestemmelser/klausuler)
 - clause = OR of literals (literater/konstanter/variabler)
 - literal = x_i / NOT x_i
 - 2CNF-SAT kan løses polynomisk, mens 3CNF SAT er NP-deilig
 - Kan brukes som en bipartitt graf
- **CLIQUE** - Største "gjeng" av noder som har kanter til alle andre i gjengen
 - Beslutningsproblem: Finnes en clique av lengde k i denne grafen?
 - Finn en kompliment-graf (altså de nodene som ikke hadde kanter mellom seg i cliquen, skal nå ha kanter mellom seg, og vice versa)
 - Denne grafen vil være et independent set (ingen av nodene som var med i klikken har nå kanter til hverandre)
- **VERTEX-COVER** - Dekke alle kanter i en graf ved å velge færrest mulig noder
- **HAM-CYCLE** - Finne en sykel i en graf som går gjennom hver node kun én gang, og der startnoden også er sluttnoden
- **TRAVELLING-SALESMAN** - Hva er beste måte å komme seg gjennom n byer på, på kortest mulig tid/avstand/lengde tilbakelagt

[N3] Forstå NP-komplekthetsbevisene for disse problemene

[N4] Forstå at 0-1-ryggsekkproblemet er NP-hardt

[N5] Forstå at lengste enkle-vei-problemet er NP-hardt

- Se [J4]

[N6] Være i stand til å konstruere enkle NP-komplekthetsbevis

Rekker vi må kunne:

Håndtrykksformelen

Dette er én av to enkle men viktige formler som stadig dukker opp når man skal analysere algoritmer. Formelen er:

$$\sum_{i=0}^{n-1} i = \frac{n(n-1)}{2}$$

Den kalles gjerne *håndtrykksformelen*, fordi den beskriver antall håndtrykk som utføres om n personer skal hilse på hverandre. Vi kan vise dette ved å telle antall håndtrykk på to ulike måter. De to resultatene må da være like.

Telling 1: La oss se på personene én etter én. Første person hilser på alle de andre $n - 1$ personene. Andre person har allerede hilst på første person, men bidrar med $n - 2$ nye håndtrykk ved å hilse på de gjenværende. Generelt vil person i bidra med $n - i$ nye håndtrykk, så totalen blir

$$(n - 1) + (n - 2) + \cdots + 2 + 1 + 0.$$

Om vi snur summen, så vi får $0 + 1 + \cdots + (n - 1)$, så er dette den venstre siden av ligningen. Dette oppstår typisk i algoritmer de vi utfører en løkke gjentatte ganger, og antall iterasjoner i løkka øker eller synker med 1 for hver gang, slik:

```
1  for i = 1 to n - 1
2      for j = i + 1 to n
3          i tar j i hånden
```

Telling 2: Vi kan også telle på en mer rett frem måte: Hver person tar alle de andre i hånden, og inngår dermed i $n - 1$ håndtrykk. Hvis vi bare teller hver enkelt person sin halvdel av håndtrykket, får vi altså $n(n - 1)$ *halve* håndtrykk. To slike halve håndtrykk utgjør jo ett helt, så det totale antallet håndtrykk blir den høyre siden av ligningen, nemlig $n(n - 1)/2$.

Man kan også gjøre om en sum av denne typen ved å brette den på midten, og legge sammen første og siste element, nest første og nest siste, etc. Vi får da

$$(n - 1 + 0) + (n - 2 + 1) + (n - 3 + 2) + \dots$$

Hvert ledd summerer til $n - 1$, og det er $n/2$ ledd. Mer generelt er summen av en aritmetisk rekke (der vi øker med en konstant fra ledd til ledd) lik gjennomsnittet av første og siste ledd, multiplisert med antallet ledd i rekken.

Utslagsturneringer

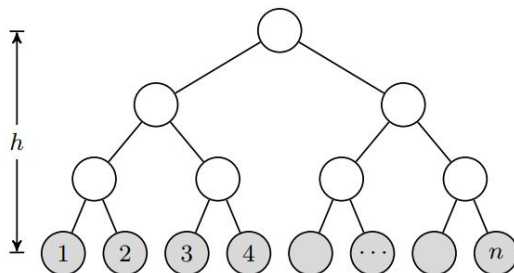
Dette er den *andre* av de to sentrale formlene:

$$\sum_{i=0}^{h-1} 2^i = 2^h - 1$$

Det vil si, de første toerpotensene summerer til én mindre enn den neste. En måte å se dette på er av totalssystemet, der et tall $a = 11 \cdots 1$ med h ettall etterfølges av tallet $b = 100 \cdots 0$, som består av ett ettall, og h nuller. Her er $a = 2^0 + 2^1 + \cdots + 2^{h-1}$ og $b = 2^h$, så $a = b - 1$.

Et grundigere bevis kan vi få ved å bruke samme teknikk som før, og telle samme ting på to ulike måter. Det vi vil telle er antall matcher i en utslagsturnering (*knockout*-turnering), det vil si, en turnering der taperen i en match er ute av spillet. Dette blir altså annerledes enn såkalte *round robin*-turneringer, der alle møter alle – for dem kan vi bruke håndtrykksformelen for å finne antall matcher, siden hver match tilsvarer ett håndtrykk.

Telling 1: Vi begynner med å sette opp et *turneringstre*:



Her er deltakerne plassert nederst, i løvnodene; hver av de tomme, hvite nodene representerer en match, og vil etterhvert fylles med vinneren av de to i nodene under, helt til vi står igjen med én vinner på toppen. Vi går altså gjennom h runder, og i hver runde organiseres de spillerne som gjenstår i par som skal møtes. Om vi nummererer rundene baklengs, etter hvor mange runder det er igjen til finalen, så vil antall matcher i runde i være 2^i . Det totale antall matcher er altså $2^0 + 2^1 + \cdots + 2^{h-1}$, som er venstresiden av ligningen vår.

Telling 2: Antall deltakere er $n = 2^h$. I hver match blir én deltaker slått ut, helt til vi sitter igjen med én vinner etter finalen. Antall matcher trenger vi for å slå ut alle bortsett fra én er $n - 1$, eller $2^h - 1$, som er høyresiden i ligningen.

Det er forøvrig også viktig å merke seg *høyden til treet*: $h = \log_2 n$. Dette er altså antall doblinger fra 1 til n , eller antall halvinger fra n til 1. Det er også en sammenheng som vil dukke opp ofte.

Oversikt sorteringsalgoritmer:

- **Sammenligningsbasert**
 - Sammenligner to elementer for å se hvem som skal stå først i sekvensen.
Worst case $\Omega(n \lg n)$.
- **Stabil**
 - Like elementer blir “samlet” i samme rekkefølge som før sortering
- **In-Place**
 - Bruker eksisterende struktur uten å lage en ny kopi
- **Splitt og hersk**
 - Deler opp sekvensen i mindre biter for å få kontroll over listen

Navn	Stabil	In-place	Divide & Conquer	Sammenligningsbasert	Kjøretid	Kommentar
Insertion sort	x	x		x	B - $O(n)$ A - $O(n^2)$ W - $O(n^2)$	Effektiv for å sortere en liten mengde elementer.
Merge sort	x*		x	x	W - $\Theta(n \lg n)$	Splitter opp og kaller MERGE for å slå sammen. *Kan implementeres ikke-stabil ved noen få endringer på koden.
Quicksort		x	x	x	A - $O(n \lg n)$ W - $O(n^2)$	Randomized-quicksort kan brukes for å minske muligheten for worst case. Kan gjøres stabil, men mer effektiv uten.
Counting sort	x				$\Theta(n + k)$, der k vanligvis er $O(n)$	Fast af. Brukes ofte av Radix sort. Antar at verdiene er heltall i et begrenset verdiområde
Bucket sort	x	x			A - $\Theta(n + k)$	Bruker insertion sort

					$W - \Theta(n^2)$	
Bubble sort	x	x		x	B - $O(n)$ A og W - $O(n^2)$	
Heap Sort		x	x		$O(n \lg n)$	Kan bruke prioritetskø
Randomize d-select		x(kanksje)	x	x	A - $\Theta(n)$ W - $\Theta(n^2)$	Sorterer ikke, men finner i-te element
Select		x	x	x	$O(n)$	Medianer av medianer

Oversikt korteste vei algoritmer:

Navn	Kjøretid	Negative kanter	Negative sykler	Type	Kommentar
DAG-shortest-path	$\Theta(V + E)$	x		SSSP	Topologisk sortering først, eksempel på DP, bruker DFS
Bellman-Ford	$O(VE)$	x	return False	SSSP	Oppdager negative sykler, slakker hver kant $ V - 1$ gang, generell og treg
Dijkstra	Binær heap - $O(E \lg V + V \lg V)$			SSSP Grådig	Kan bruke array/liste og fib heap også. Bruker tabelliste når vi har veldig mange kanter (flere enn $\Theta(V^2/\lg V)$)
Floyd-Warshall	$\Theta(V^3)$	x	Finne et negativt tall i diagonale n	APSP	Holder styr på korteste vei og forgjengere. Kan også implementeres slik at den

					holder styr på andre ting (eksamensoppgave).
Transitive-Closure	$\Theta(V^3)$	mulig?		APSP	Binærmatrise
BFS	$\Theta(V + E)$			Traversering	Kan brukes i korteste vei
DFS	$\Theta(V + E)$			Traversering	Kan ikke garantere korteste vei (trekant med 3 noder er eksempel der den ikke vil finne korteste vei, men BFS vil)