AD noter

Sebastian Rode

10. april 2023

Indhold

1 Asymptotisk Notation

1.1 Big O-notation

Den værst tænkelige køretid for en algortime.

Her er de generelle regler for Big O-notation:

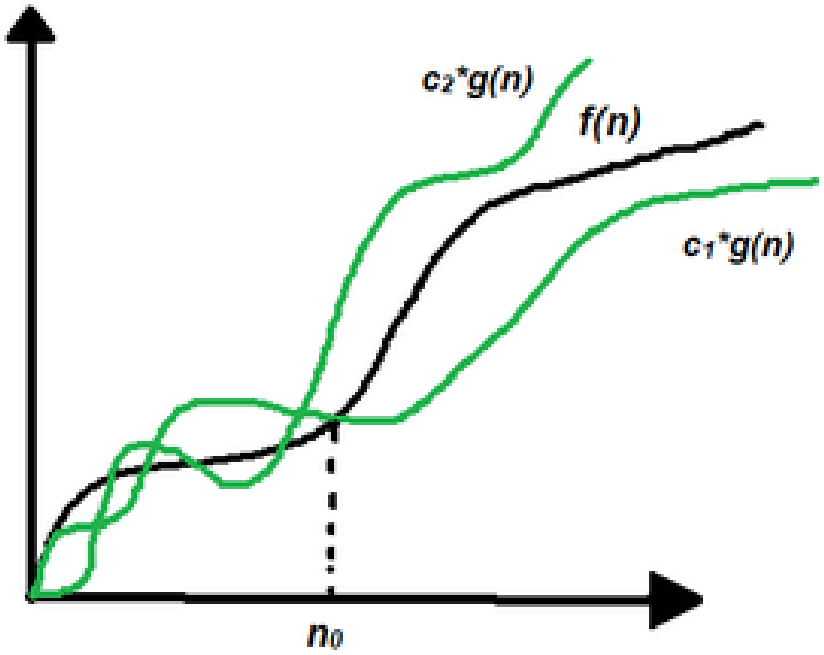
1. Man ignorerer konstanter. Det vil sige 5n $\rightarrow$ O(n)
2. Rækkefølgen fra laveste til største køretid er
   $O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2) < O(2^n) < O(n!)$
3. vi regner ikke på de hurtigere køretider, hvis en funktion har en højere køretidstype i sig.

1.2 Big Ω-notation

Bedst tænkelige køretid. Udregnet ved at tage den hurtigst mulige køretid på en algoritme med n iterationer.

1.3 Big Θ-notation

Ved en tilstrækelig stor n ved man, at algoritmen vil have en køretid mindst eller mest inden for et interval. fx som set på billedet nede under, kan det ses at f(n) er mellem c1 og c2 efter en given n0 størrelse.



1.4 Fremgangsmetode til typeopgaver i eksamen

Ved Big O-notation, se om den til højre er større eller mindre end værdien til venstre. Husk konstanter ikke tæller med, samt kun den langsommeste funktion i stykket regnes på.

Hvis den til venstre er mindre end den til højre svarer man JA.
Hvis den til venstre er større end den til højre svarer man NEJ.

Ved Big Ω-notation er det det omvendte af Big O-notation, hvilket vil sige at:
Hvis den til venstre er mindre end den til højre svarer man NEJ.
Hvis den til venstre er større end den til højre svarer man JA.

ved B

2 Sådan laver man et Induktionsbevis (af Kim)

Kim er manden og har derigennem udarbejdet denne smukke gennemgang af, hvordan man laver et induktionsbevis. Hvis du læser dette, skylder du ham en øl på caféen?

2.1 Generelt om Induktionsbeviser

Generelt ved alle induktionsbeviser, sa er det godt at huske pa de 3 hovedpunkter, som giver overblik og samtidig forsikre os om, at påstanden om den givne algoritme er sand. De 3 hovedpunkter, kan altså ses som en hjælp til os, for at sikre at vi overholder alle krævne for et induktionsbevis. Vi kan ogsa se de 3 hovedpunkter som påstande, P1,P2 og P3, som vi vil løbende "vise" gælder.

2.2 P1) Induktionshypotesen:

Hvilken påstand skal bevises?

Skriv altid op, hvad der skal bevises, samt alle restriktioner og antagelser, der matte gælde. Dette gøres for at fa overblik og struktur, og sa man har gjort det helt klart for læseren, hvad der skal til at ske. Generelt er dette altid en super god vane at have, uanset hvilket matematiske bevis, det skulle handle om.

2.3 P2) Basis-tilfældet:

Holder pastanden for trivielle n? Vis at algoritmen virker for det mest basale valg af n, som overholder alle antagelser. Dette gøres ved indsætte det simpleste tilfælde af n, hvor vi typisk ved præcis hvilken værdi som algoritmen skal give os. Dette kan ses som en simpel kontrol, og gøres for at sikre, at vores algoritme ikke fejler pa det åbentlyse tilfælde, som vi algoritmen skal overholde. Hvis dette tilfælde ikke kan vises, sa kan dette jo være en kraftig indikation pa, at vores algoritme ikke er korrekt skrevet op, eller i værste tilfælde, at algoritmen simpelthen ikke er sand eller giver noget der er sandt, hvilket er hele formålet med algoritmen.

2.4 P3) Induktionsskridtet:

Holder påstanden for alle n?

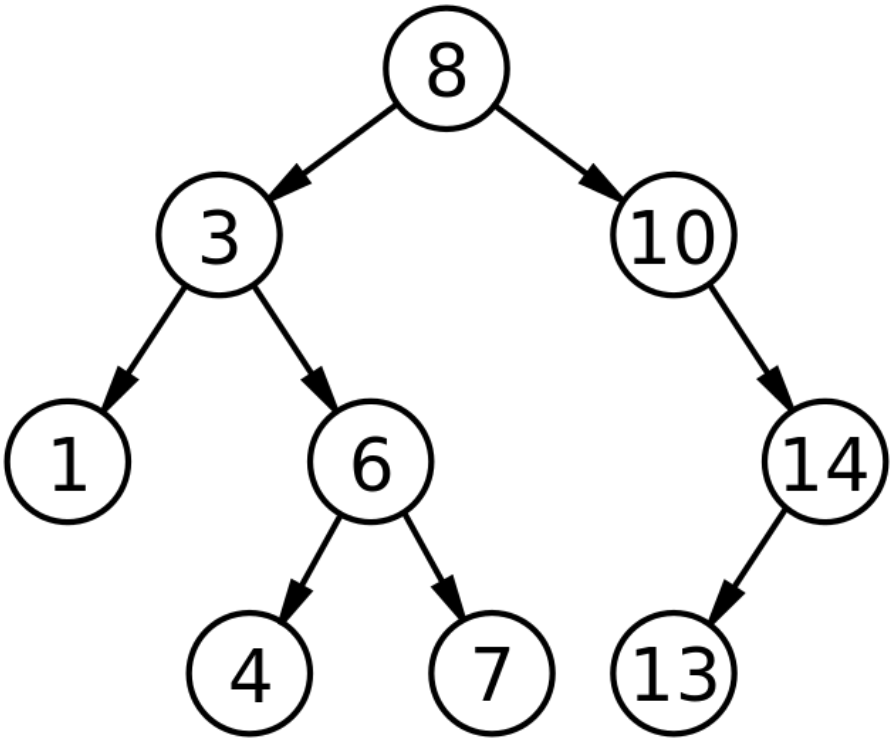
At vise denne påstand, anses typisk som den "sværre" del af beviset, eller faktisk "selve beviset". Det er typisk her, at man bruger mest tid. Vi vil nu antage at algoritmen holder for et vilkårligt valg af n, og benytte dette, til at vise at algoritmen ogsa holder, for de fremtidige eller forrige skridt, hvilket punktnavnet induktionsskridtet prøver at indikere. Og siden n er vilkårligt valgt, viser dette at algoritmen holder for alle n.

Dette gøres ved at indsætte (n+ 1) eller (n-1) i algoritmen. Typisk, efter nogle reduceringer, udregninger og brug af regneregler/Sætninger, vil det give os muligheden for, at kunne benytte antagelserne om virkningen af algoritmen, evt. fra basis-tilfældet eller antagelsen om algoritmen holder for n, for nu, at vise at påstanden for algoritmen gælder.

I alternative tilfælde, er det typisk ikke fordelagtigt, at benytte (n + 1) eller (n - 1), men at man kan prøve sig med (n + i) eller (n - i), for i = 2, 3, 4, ...

3 træer

et søgetræ er sorteret, når alle værdier til højre fra en en knude er større, mens alle værdier til venstre fra knuden er mindre.



En heap er en træ-struktur.

Hver niveau er fyldt ud ligeligt, jo mindre en knude ikke har børn.
Hver knude har max 2 børn

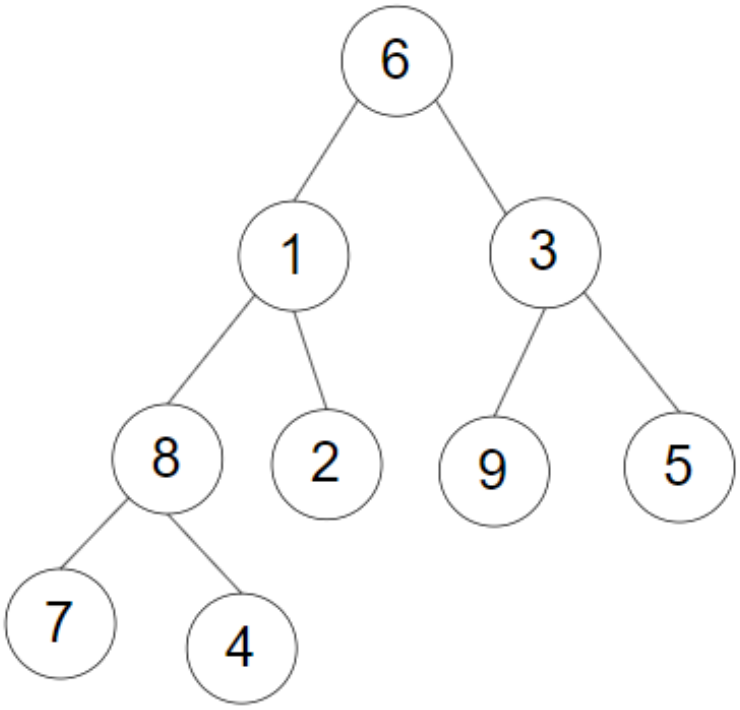
Alle knuder er så venstre som muligt, hvilket vil sige hvert barn er til venstre for dens forældre.

Man kan bruge algortimer som Prim's algortime og Dijkstra's algortime til at skabe heaps.

3.1 Læs heap fra array

Man læser en heap fra et array fra venstre til højre, hvor index 0 er roden. index 1 og 2 er rodens børn, hvorefter knude 3-7 er de næste børn osv. man kan tegne det således, at hvert lag går fra venstre til højre. når laget har alle de børn det kan have, går man videre til næste lag.

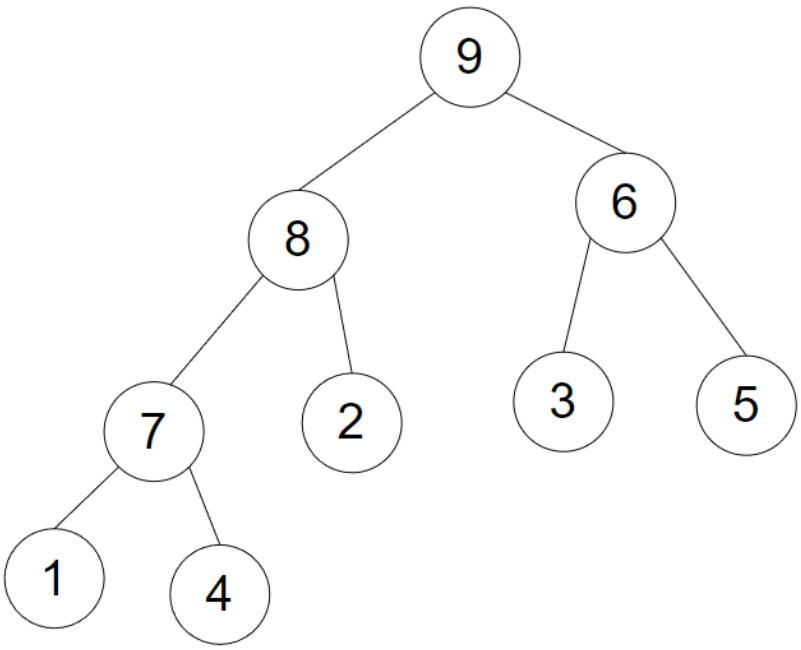
fx vil arrayet [6, 1, 3, 8, 2, 9, 5 7, 4] kunne tegnes således:



3.2 Sådan laver man en Max-heap

1. start med at tegn dit træ færdigt ud fra dit array.
2. sammenlign af den højeste værdi af et barn med dens forældre.
3. swap knuderne, hvis værdien af forældren er mindre end barnet (enten til højre eller til venstre).
4. gentag indtil det største element er roden, og der ikke er flere elementer der kan swappes.

Det vil sige, hvis man laver build-max-heap på arrayet fra oven over, vil træet se således ud:



3.3 sådan laver du en min-heap

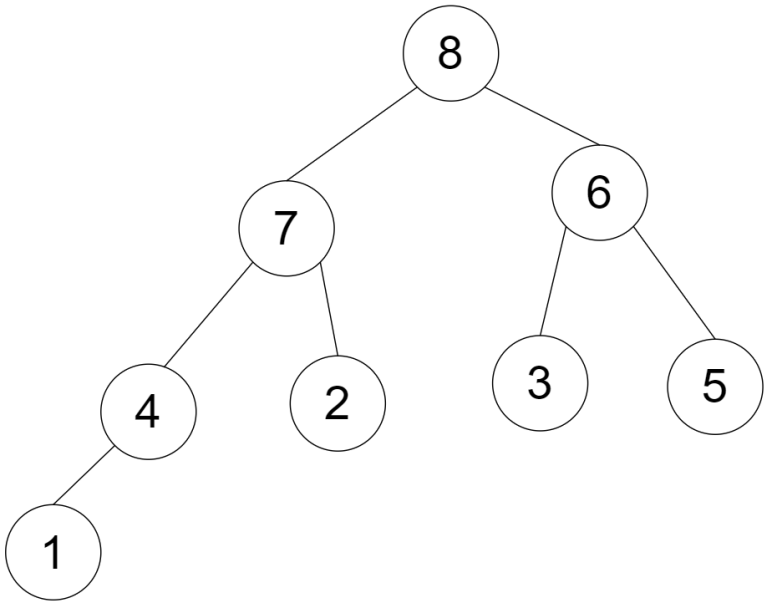
direkte modsætning af max-heap:

1. lav et barn i slutningen af din heap (laveste niveau).
2. swap således at forældre altid er mindst.
3. bliv ved indtil det mindste tal er roden.

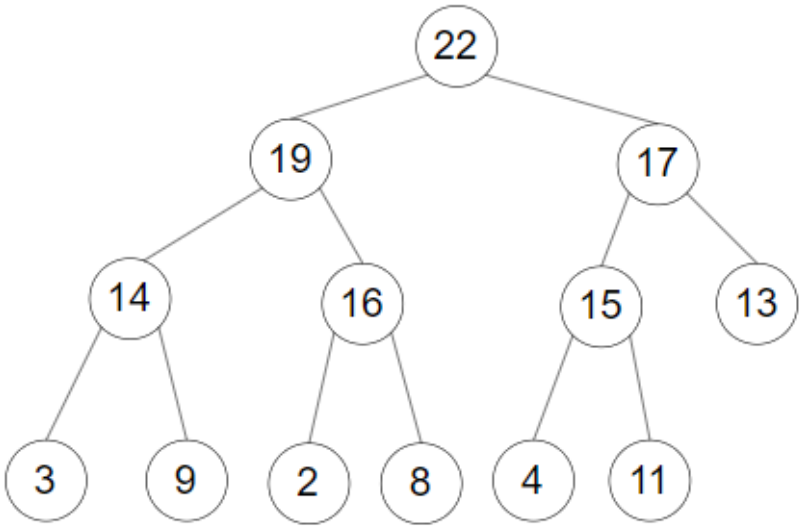
3.4 Heap-Extract-Max og Heap-Extract-Min

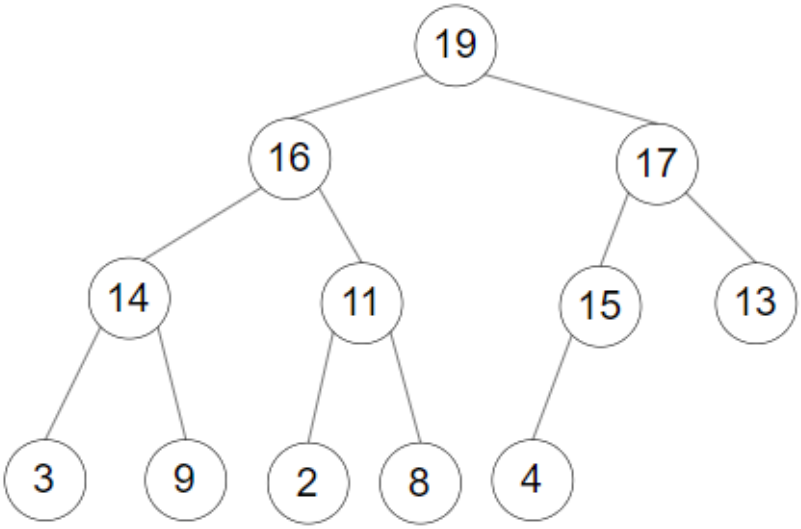
Man tager roden af sin heap, som enten er min eller max, og tager ud. derefter tager man det sidste element i sit træ og putter på rodens plads. Derfra bobler man roden ned til sin nye korrekte placering.

Fx hvis man brugte Heap-Extract-Max på det den max-heap der er på billedet oven over, ville den ende med at se således ud:



Et eksempel mere er på træet her, hvor først ses træet, derefter hvordan det ser ud, når man har lavet Extract-Max:





3.5 rød-sort træer

Egenskaber ved et rødt-sort træ:

1. Roden er altid sort
2. Blade (knuder uden børn) er altid sorte.

3. Hvis en knude er rød, er begge dens børn sorte.

4. alle veje fra en given knude til et blad (knude uden børn) i undertræet har samme antal sorte knuder.

5. Et rødt-sort træ har en sort hæjde, bh.

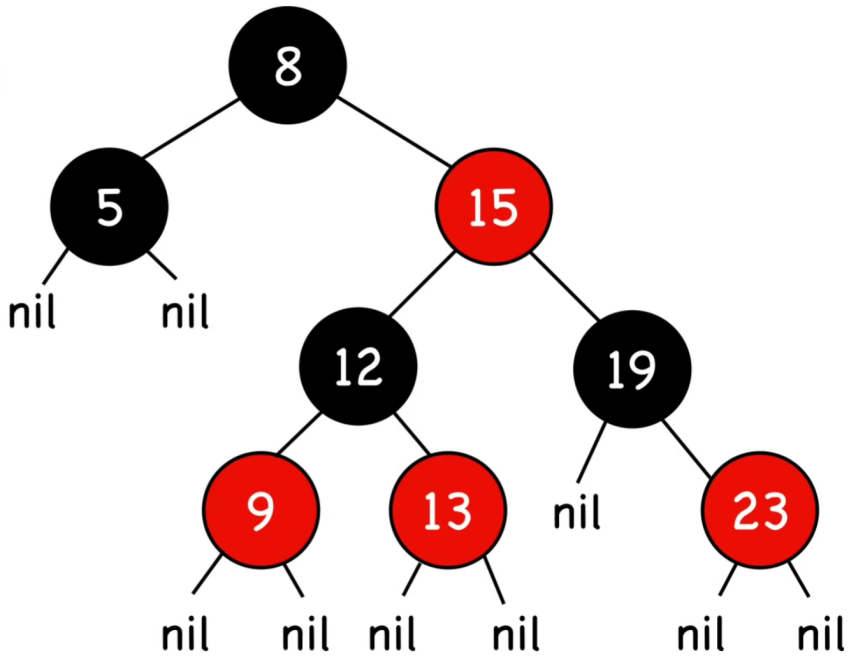
bh er højden af de sorte knuder til en anden given knude. bh af træet er højden af sorte knuder fra bunden til roden. Givet at alle veje fra en given knude til et blad har samme antal sorte knuder, vil der være den samme bh om man starter i venstre eller højre side af træet.

For rød-sorte træer gælder dermed følgende:

1. De kræver 1 ekstra bit af information i lagring, for at vise om de er røde eller sorte.

2. Den længeste vej fra roden til den længeste NIL er ikke længere end den dobbelte distance fra roden til den korteste NIL.
   - Det vil sige den korteste vej har kun sorte knuder
   - Den længste vej skifter mellem røde og sorte knuder.

Her er et eksempel på et rødt-sort træ:



3.5.1 rød-sort rotationer

Hvis man sletter eller tilføjer elementer til sit træ, kan det skabe uorden i et rødt-sort træ, da dens egenskaber stadig skal overholdes. Dette kan løses ved hjælp af rotationer.

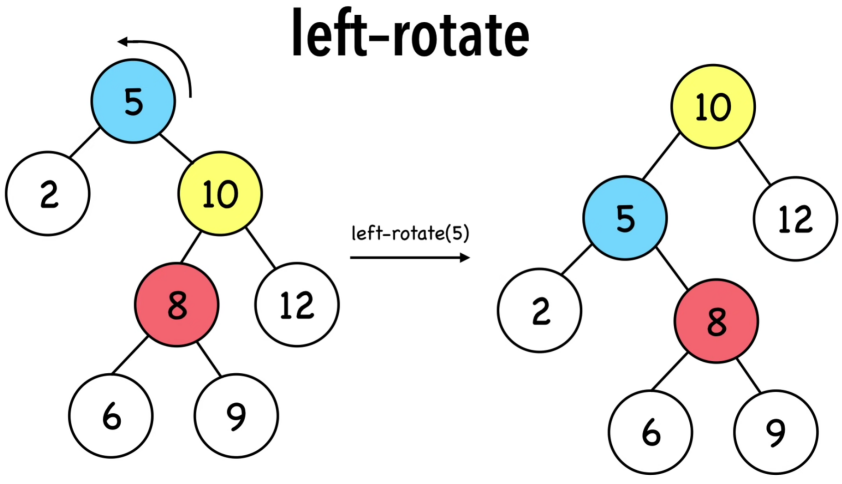
Når man skal lave en rotation, skal man huske følgende:

1. Man ændrer på strukturen af træet ved at omrokkerer et undertræ.
2. Målet er at gøre træet kortere - rød-sort træer har en max højde på $O(logn)$ - Man kan gøre træet mindre ved at rykke store undertræer op og mindre undertræer ned.
3. rotation ændrer ikke på rækkefølgen af elementer. Med det menes der, at mindre elementer stadig står til venstre, med større elementer til højre.

Der findes 2 typer af rotationer - højre og venstre rotationer.

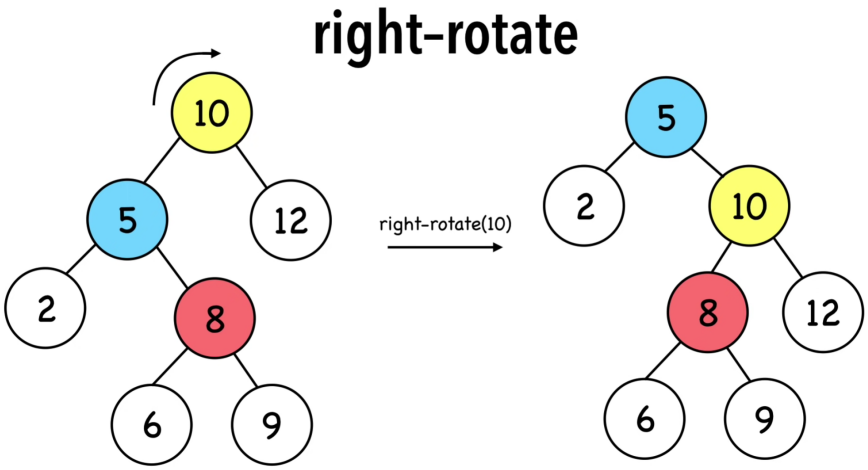
Vi starter med en venstre-rotation. Følgende træ kan skrives på array-form [5, 2, 10, 8, 12, 6, 9].
Jeg springer farverne over for nu, men highlighter nogle knuder, for at vise rotationen.

Jeg vil gerne roterer 5 til venstre, altså en rotation venstre-rotation(5):



5 rykkes ned til venstre, mens dens højre barn bliver til roden. 10's venstre barn 8 er større end 5, så dets undertræ rykkes til højre for 5.

Det samme gælder hvis man vil lave en højre-rotation. Hvis vi prøver at rotere 10 mod højre, rykkes 5 op som rod. 8 er større end 5, så det skal tilpasses på højre side af 5. det er mindre end 10, så den ryger ind på venstre side af 10.



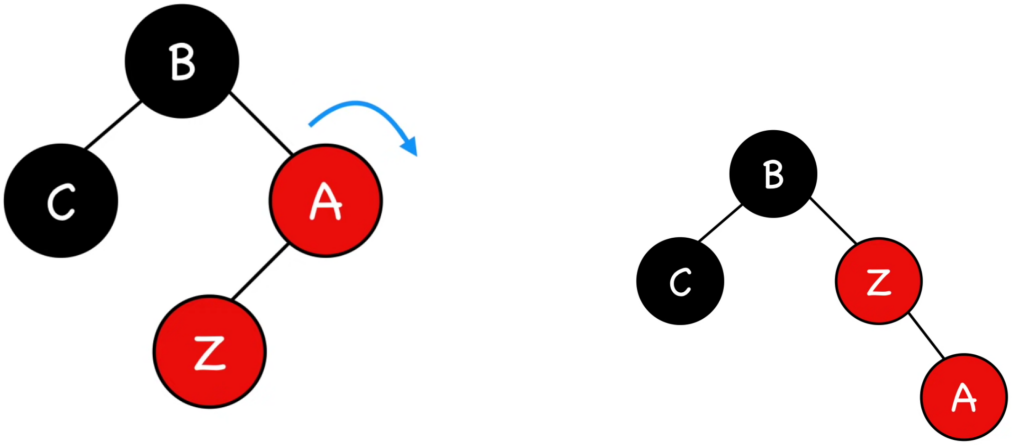
3.5.2 insertion i rød-sort træer

Fremgangsmetoden er som følger:

1. indsæt elementet og farv det rødt.
2. genfarv og roter knuder for at fixe det rød-sorte system.

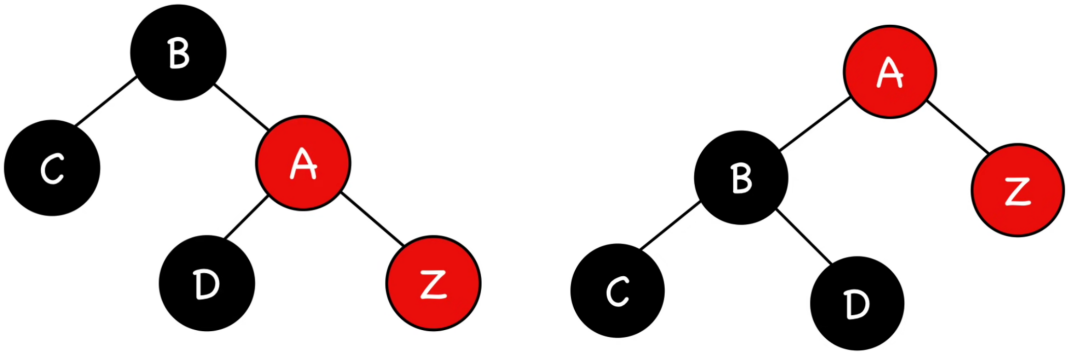
Der er 4 forskellige scenarier vi kan støde ind i, når vi indsætter, som hver har sin løsning.

1. det indsatte element er roden. Vi løser dette ved at farve det sort.
2. Hvis det indsatte elements "onkel"(knuden der er 2 oppe og 1 ned i den modsatte retning af elementet vi indsætter) er rødt, genfarver vi elementets forældre, bedsteforældre og onkel.
3. Hvis det indsatte elements onkel er sort og en trekant dannes, således forældren er rød, roterer vi på forældren og genfarver.



(På billedet er der ikke genfarvet endnu)

4. Hvis det indsatte elements onkel er sort og en linje dannes, således en forældre og et andet barn er rødt, roterer vi på bedsteforældren og genfarver.



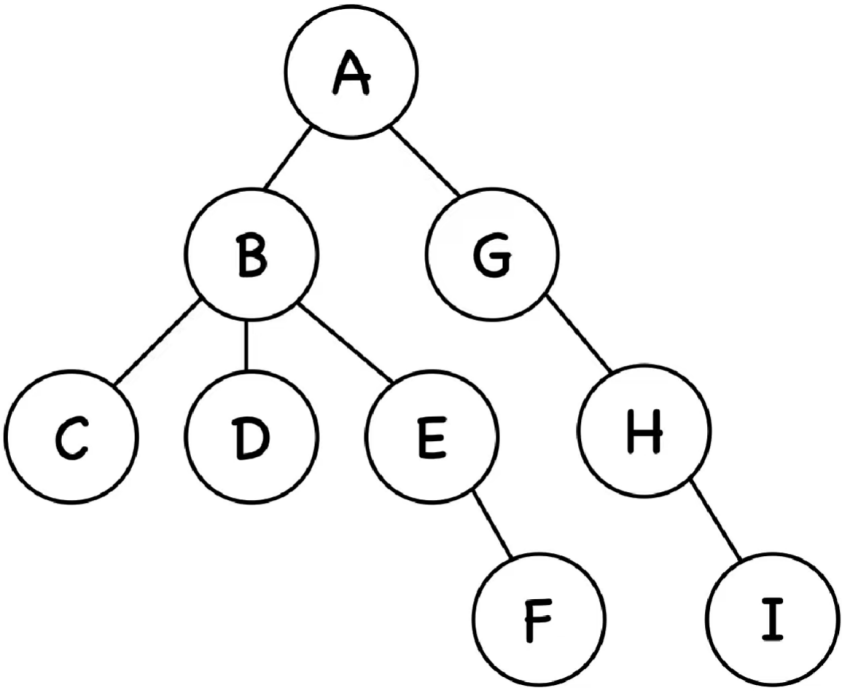
(På billedet er der ikke genfarvet endnu)

3.6 DFS - Dybde-først søgning

For at søge gennem et træ eller en vægtet graf med DFS, tager man og går "i dybden"med den knude man er kommet til, før man går til andre knuder /

undertræer. Det vil sige, at man prioriterer at gennemføre en given knude, før man starter på et andet undertræ.

Her er et eksempel på et træ:



Ifølge DFS starter vi i rodden. Da B kommer før G, går vi til venstre. Vi færdiggører vores søgning af undertræ B før vi går til undertræ G. Dermed går vi først til C i undertræ B, da det er mindre end D og E osv. Til sidst ender man med en søgning der returnerer [ABCDEFGHI].

3.6.1 Typer af kanter i en DFS

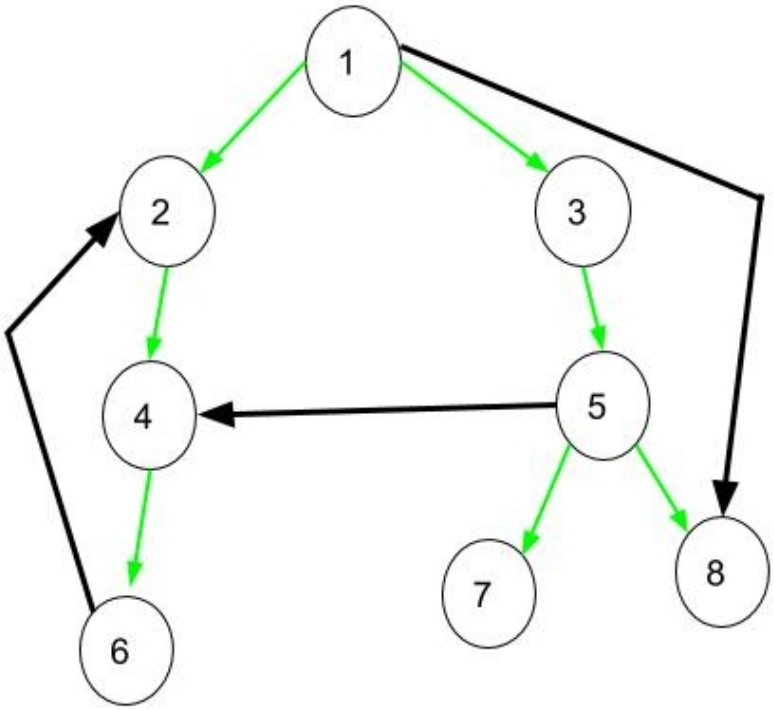
Der findes 4 typer af kanter i en DFS.

1. Tree edge
   En kant indgår i træet efter at have gennemløbet DFS, altså hvis 2 punkter er forbundet direkte til hinanden. De grønne kanter på billedet nedenfor er Tree Edges.

2. Forward Edge
   en kant (u,v) således at v er en nedarver, men ikke direkte del af DFS. kanten fra 1 til 8 i billedet nedenfor er en forward edge

3. Back Edge
   en kant (u, v) således v er en forfædre til knude u, men ikke direkte forbundet i DFS. kanten fra 6 til 2 på billedet nedenfor er en Back Edge.

4. Cross Edge
   En kant der forbinder 2 knuder der ikke deler en fælles forfædre og efterkommer. kanten mellem knude 5 og 6 er en Cross Edge på billedet neden for.



Figur 1: Typer af kanter

3.7 BFS - bredte-først søgning

Med BFS gennemgår man først alle knuder fra roden, før man gennemgår undertræerne til knuderne. Det vil sige man søger gennem et træ eller graf vandret i stedet for lodret (som er DFS).

Hvis vi kigger på samme træ som vi så på i DFS, ville søgningen returnerer [ABGCDEHFI].

3.8 Dijkstra's algoritme

En algortime til at finde afstanden fra en knude til alle andre knuder i en vægtet graf / træ.

Man starter med et lave en liste over sine ubesøgte knuder. Derefter vælger man en startknude.

Fra sin start knude vælger man den rute, som er kortest mellem de mulige ruter til en ny ubesøgt knude. Hvis man ikke har gået nok skridt til at nå til en ny knude endnu, er distancen til knuden ∞.

Efter man har nået en knude, går man tilbage til sit startpunkt og leder efter den knude, der har næstkortest distance, eftersom man nu har fået adgang til en knude mere. Den nye knude føjes til listen. Hvis adgangen til en ny knude gør ruten til en allerede opdaget knude kortere, opdateres listen over korteste vej til den givne knude.

Man tager ALTID udgangspunkt i sin startknude. Det vil sige man kun kigger på distancen til en given knude ud fra hvor langt der er til den fra startknuden, og ikke på distancen til en ny knude fra en given mellemknude.

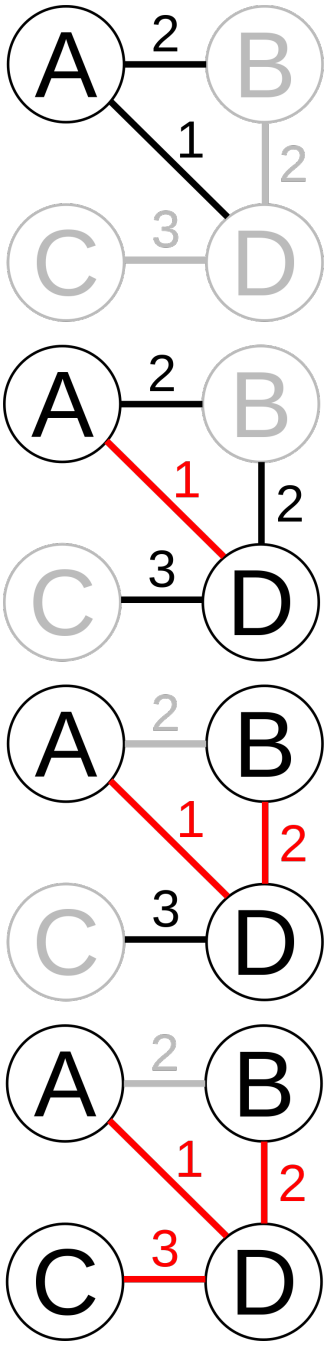
3.9 Prim's algoritme

Prim's algortime er en grådig algortime til at finde distancer i en vægtet graf. At den er grådig vil sige, den altid tager den lokalt bedste løsning og ikke tager højde for en potentiel global bedre løsning.

Først laver man en tom liste. Lad os sige man har en graf, hvorfra man starter ved en knude A, som man tilføjer som første element til sin liste. Derfra tager går man til den næste knude med den korteste distance, og tilføjer til listen. Det næste element er det element der har den korteste distance til et eksisterende element i listen. (Hvis der er ens distancer, vælger man bare en af dem). Sådan gennemløber man alle elementer og tilføjer til sin liste - altid vælger den næste distance der er kortest i forhold til et tilføjet element.

På billedet nedenfor bliver listen fyldt ud således:
$[A]$
$[A, D]$
$[A, D, B]$

$[A, D, B, C]$

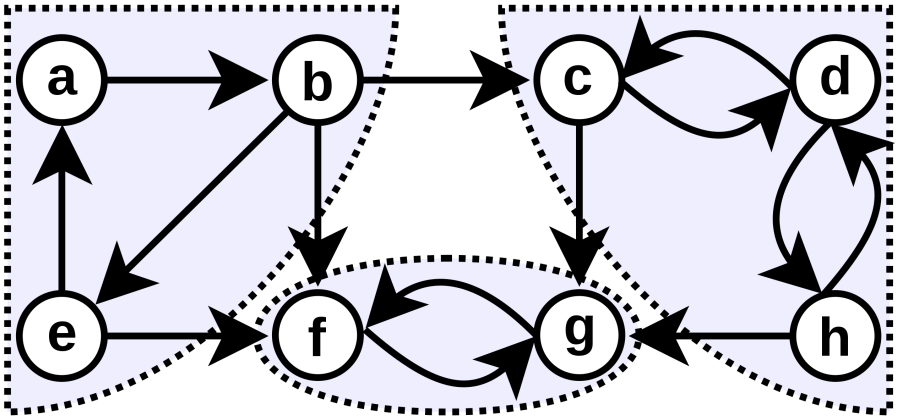


Figur 2: Eksempel på prim's algortime

3.10 Stærke sammenhængskomponenter / strongly connected components

Stærke sammenhængskomponenter er knuder i en graf, som forbindes begge veje.

For at finde antallet af stærke sammenhængskomponenter, ser man på hvor mange dele der hænger stærkt sammen med hinanden / er i en lykke. på billedet nede under er der 3 stærke sammenhængskomponenter:



3.11 Minimum Spanning Tree (MST)

Et MST er et delsæt af et spanning tree med kanter der er forbundet og vægtet, som forbinder alle knuder samlet med mindst total kant-vægt.

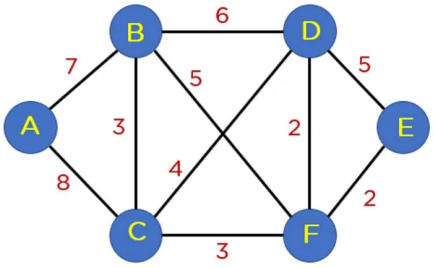
3.11.1 Kruskals algoritme

Algortime til at lave en MST.

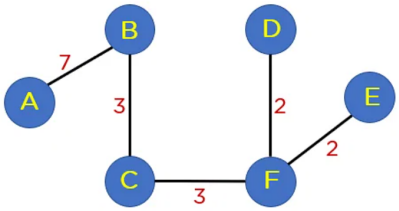
1. Sorter alle kanter ud fra kantvægt
2. vælg den mindste kant
3. Hold øje med om en ny kant laver et loop i træet
4. Hvis et loop ikke formes, inkluder kanten i MST, ellers smid den ud.
5. Gentag step 2 til 5 indtil din MST inkluderer |V|-1 kanter.

Efter at have fulgt overstående skridt, vil du have et MST.

Eksempel:



Removing parallel edges or loops from the graph.



Minimum Spanning Tree.

4 Invarianter

Invarianter er udsagn som altid er rigtigt. I forbindelse med psudokode er det altså udsagn, ofte om lykker, som er sande, lige meget hvilken interation lykken eller koden er nået til.

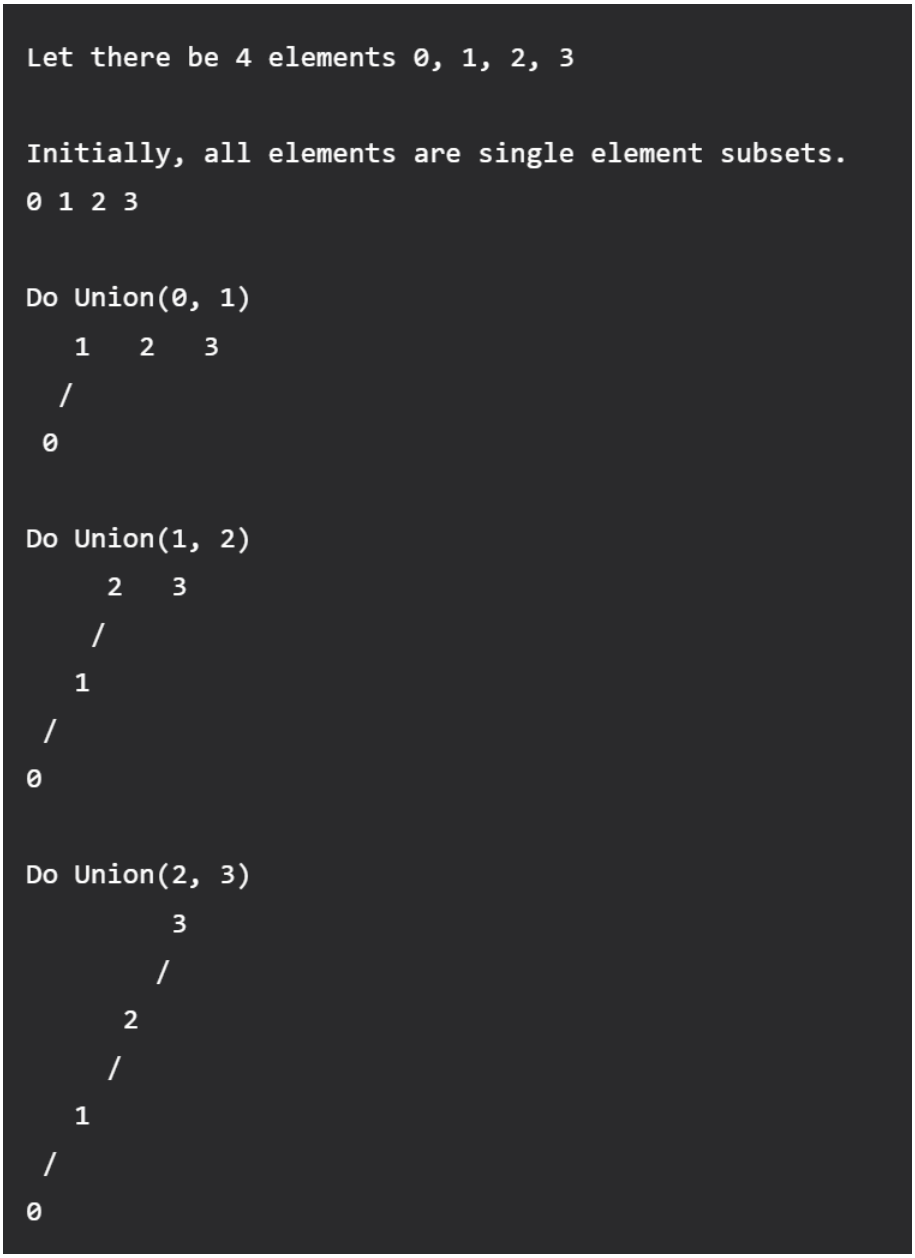
5 Disjoined Sets

5.1 Union-find

Når man laver Union-find findes der en række af operationer:

1. Makeset(x): Skaber en knude til træet.
2. Union(x, y) / Merge(x, y) / Link(x, y): sætter 2 knuder sammen. den første knude bliver hægtet på den anden, så fx Union(a, b) hægter a som et barn til b.
3. Find-set(x) returnerer roden til x. Man kan bruge diverse metoder til at lave et træ, som er bedst struktureret til at finde dette (union by rank og path-compression).

Her er et eksempel med elementerne 0, 1, 2 og 3, hvor der er kaldt makeset() på dem alle. Derefter bliver der kaldt union på dem, således strukturen bliver længst mulig, altså den værst tænkelige køretid:

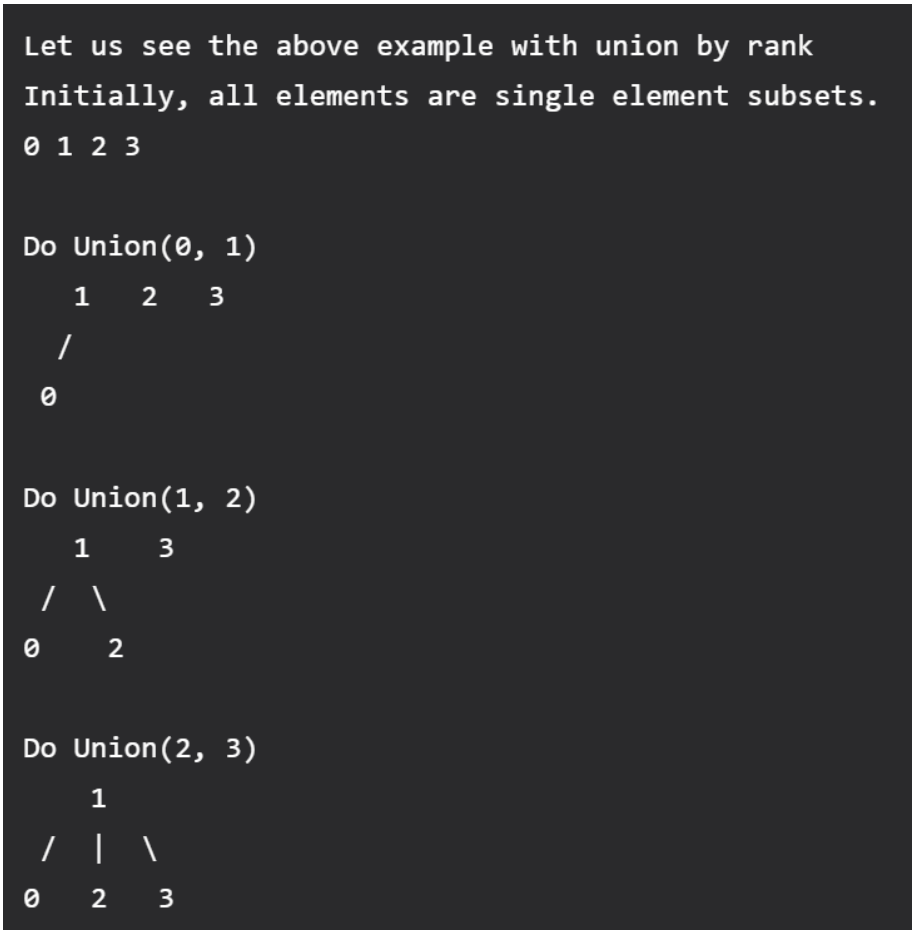


Figur 3: union af elementer

For at få en bedre køretid, kan man hægte elementer på træet ud fra rank i stedet for højde. Her fra hægter man et element på en liste, baseret på hvilket af elementerne der har den højeste rank.

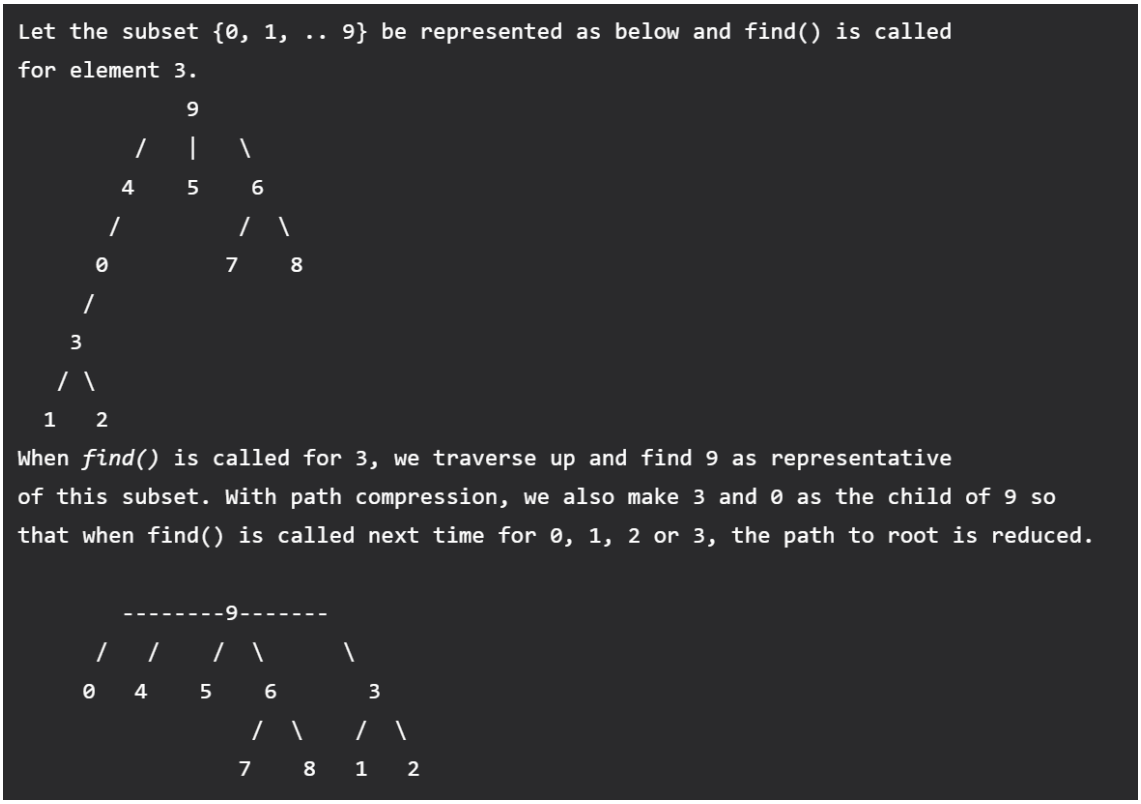
I eksemplet nede under hægtes 0 på 1. I undertræet har 0 en rank 0 og

1 har en rank 1. Når der så bliver kaldt union(1, 2), hægtes 2 på 1, da 1 har højest rank. Når der så kaldes union(2,3) hægtes 3 også på 1, da 1 stadig har den højeste rank.



Figur 4: Union by rank

En anden måde at få bedre køretid på, er ved hjælp af path compression. Når man kalder find-set(x), hægtes x på den øverste representant y i undertræet. Derefter hægtes alle andre undertræer i undertræet også fast på y, således træet bliver væsentligt lavere.



Figur 5: Path Compression

6 Amortized Analysis

Udgangspunkt i en stack med multipop. Den har 4 operationer den skal kunne udfører:

1. Push(S,x): pusher element x ind på stacken S
2. Pop(S): popper det øverste element på stacken S
3. Multipop(S,k) Popper k elementer fra stacken (k er positivt tal. Hvis S $<$ k, popper alle elementer.)
4. Stack-Empty(S): outputs True hvis stacken er tom, ellers False

Hvert kald tager $\Theta(1)$ konstant worst case tid.
Multipop tager $\Theta(min\{s, k\})$ worst case tid

Antag vi laver n operationer på en tom stack, hvad er så den totale worst case køretid på operationerne?
Hver operation tager $\Theta(n)$ tid og der er n operationer, så worst case er $O(n^2)$.
Dette er ikke realistisk, da hver eneste operation ikke vil tage worst case tid at udføre i et normalt miljø.

Vi kan bruge metoder til at finde en mere præcis køretid

6.1 Aggregate Analysis

I aggregate analysis regner man på en upper bound T(n) af den totale worst case tid af n operationer. Derefter udregner man en upper bound gennemsnitlig cost / køretid, kaldet amortizied cost, udtrykt ved T(n)/n.

6.1.1 stack eksempel

Vi kigger på eksemplet med multipop-stacken oven over.
- Vi ved der maksimalt er n push operationer, så derved kan vi binde worst case tid på push og pop-operationer.
- antallet af pop operationer kan ikke være større end antallet af push operationer, da vi jo ikke kan poppe den tomme luft.
- Derved er den totale worst case tid for alle n operationer O(n).
- Den amortiserede cost per operation er O(n)/n = O(n).

6.1.2 Binary Counter

Vi vil implementere en binær tæller, der starter på 0 og vil tælle k bits med operationen INCREMENT.
Tælleren går tilbage til 0 efter 2^k INCREMENTs (overflow).

psudokode for INCREMENT:

```
INCREMENT(A)
i = 0
while i < A.length and A[i] == 1
    A[i] = 0
    i = i + 1
if i < A.length
    A[i] = 1
```

Køretiden af INCREMENT er proportional med hvor mange bits der bliver opereret på. Med det menes der, hvis man fx skal skifte 3 bits fra 1 til 0, tager det længere tid end hvis man skal ændre et enkelt 0 til 1.

Counter value	A[7]	A[6]	A[5]	A[4]	A[3]	A[2]	A[1]	A[0]	Total cost
0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	1	1
2	0	0	0	0	0	0	1	0	3
3	0	0	0	0	0	0	1	1	4
4	0	0	0	0	0	1	0	0	7
5	0	0	0	0	0	1	0	1	8
6	0	0	0	0	0	1	1	0	10
7	0	0	0	0	0	1	1	1	11
8	0	0	0	0	1	0	0	0	15
9	0	0	0	0	1	0	0	1	16
10	0	0	0	0	1	0	1	0	18
11	0	0	0	0	1	0	1	1	19
12	0	0	0	0	1	1	0	0	22
13	0	0	0	0	1	1	0	1	23
14	0	0	0	0	1	1	1	0	25
15	0	0	0	0	1	1	1	1	26
16	0	0	0	1	0	0	0	0	31

Figur 6: oversigt over cost i binary counter

Her fra ses det, at worst case køretiden er proportional med antallet af bits der kan flippes, hvilket er O(k). (antallet af bits). Vi flipper i worst case alle bits.
Derved kan vi få en average time bound: O(k)

Hvad hvis vi bruger aggregate tid?
Hvis vi kigger på den mindst betydende bit, bit A[0], så kan vi se den flipper hver gang der er et kald til INCREMENT(A).
Men hvis vi kigger på den næstmindst-betydende bit, A[1], så flipper den kun hver anden gang.
Generelt kan dette udtrykkes, som bit A[i] flippes ved hvert 2^i kald, hvor i går fra i = 0, ..., k-1
Derudover gælder det også, at A[i] flipper et total af $[n/2^i]$ antal gange. Det vil sige jo større i er, jo færre gange flipper den i'de bit.

Derved kan vi udrenge det totale antal af flips vi laver over n operationer:

$$\sum_{i=0}^{k-1} \lfloor \frac{n}{2^i} \rfloor \leq \sum_{i=0}^{k-1} \frac{n}{2^i} = n \sum_{i=0}^{k-1} \frac{1}{2^i} < \sum_{i=0}^{\infty} \frac{1}{2^i} = 2n$$

Derved kan vi sige at worst case tid for alle n operationer er O(n), fordi tiden

er proportionel med hvor mange bits vi flipper.

Derved ses det at den amortiserede tid er O(n)/n = O(1)

6.2 The Accounting Method

Lad os antage vi har en serie af n operationer på en data struktur (kunne være eksemplet med stacken eller binary counter).

Lad tiden for den i'de operation koste c_i, i = 1, ..., n

I accounting metoden knytter man kunstige priser kaldet $\hat{c_i}$, ..., $\hat{c_n}$ for den n'de operation.

Vi kalder $\hat{c_i}$ den amortiserede pris for den i'de operation.

Det vil sige, at i modsætning til aggregate analysis, hvor man siger omkostningen for en operation er den gennemsnitlige omkostning for n operationer, så i definerer vi i accounting method den amortiserede omkostning til at være $\hat{c_i}$.

$\hat{c_i}$ har 3 muligher:

1. $\hat{c_i} > c_i$, vi overbetaler omkostningen med en mængde af $\hat{c_i} - c_i$, som bliver lagret som credit i specifikke objekter i strukturen, som så kan bruges senere.
2. $\hat{c_i} < c_i$, vi betaler for lidt for den i'de operation med en mængde på $c_i - \hat{c_i}$, som vi så betaler med credit vi har lagret fra en tidligere operation.
3. $\hat{c_i} = c_i$

Der er en vigtig ulighed som gælder for alle sekvenser af operationer:

$$\sum_{i=1}^{n} c_i \leq \sum_{i=1}^{n} \hat{c_i}$$

Det vil sige den totale kunstige omkostning skal være større eller lig med den reele omkostning, hvilket giver os en øvre grænse.

6.2.1 stack eksempel

Faktiske priser på reele omkostninger (skaleret med konstant):

1. Push: 1
2. Pop: 1
3. Multipop: $min\{s, k\}$

Så vælger vi de amortiserede omkostninger:

1. Push: 2
2. Pop: 0
3. Multipop: 0

Derved bliver det meget nemt at finde den amortiserede omkostning, da man bare kan tælle antallet af Push-operationer. Den højeste potentielle omkostning bliver derved 2n, ved n operationer.

En pop-operation kan ikke blive udført, hvis stacken er tom. Dermed sikrer vi, at push bliver udført først og vi derigennem aldrig får negativ credit.

$2n/n = 2$. Det giver os en O(2), som amortiseret er O(1).
Det giver os en average cost på O(1), som er det samme vi fik ved at bruge aggregate analysis tidligere.

6.3 The Potential Method

Potentialemetoden minder meget om accounting metoden, men hvor vi i accounting gemmer priser på objekter i strukturen, så gemmer vi credits et centralt sted. (metaforisk set en bank.)

Derudover har vi en potentiale funktion ϕ tilknyttet, som udtrykker hvor mange credits vi har i banken på nuværende tidspunkt.

Forestil n operationer på en datstruktur, D_0 udtrykker dataen før den første operation, og hvor D_i udtrykker dataen efter den i'de operation fra i = 1, ..., n.
Vi siger $\phi(D_i)$ er den mængde credits vi har efter D_i operationer.

Derfra definerer vi den amortiserede omkostning $\hat{c_i}$ til at være:

$$\hat{c_i} = c_i + \phi(D_i) - \phi(D_{i-1})$$

Det giver os 3 muligher for $\phi(D_i)$'s størrelse:

1. $\phi(D_i) - \phi(D_{i-1}) > 0$, vi overbetaler den i'de operation og putter $\phi(D_i) - \phi(D_{i-1})$ credit i banken.
2. $\phi(D_i) - \phi(D_{i-1}) < 0$ vi underbetaler den i'de operation og må bruge $\phi(D_i) - \phi(D_{i-1})$ credit fra banken.
3.

Lige som i accounting metoden, så kræver vi dette forhold mellem c_i og $\hat{c_i}$:

$$\sum_{i=1}^{n} c_i \leq \sum_{i=1}^{n} \hat{c_i}$$

Dette kan vi opnå, ved at sige:

$$\sum_{i=1}^{n} \hat{c_i} = \sum_{i=1}^{n} (c_i + \phi(D_i) - \phi(D_{i-1})) = (\sum_{i=1}^{n} c_i) + \phi(D_n) - \phi(D_0)$$

Ved at kræve at $\phi(D_n) > \phi(D_0)$ kan vi få den ønskede ulighed:

$$\sum_{i=1}^{n} \hat{c_i} = (\sum_{i=1}^{n} c_i) + \phi(D_n) - \phi(D_0) \geq \sum_{i=1}^{n} c_i$$

Kort sagt betyder dette udtryk at vi aldrig må komme under det antal credit vi har i banken til at starte med. Så hvis vi fx starter med at have 3 credit i banken må vi aldrig komme under 3.

Siden vi ikke kender n, så kræver vi $\phi(D_i) \geq \phi(D_0)$ for alle $i \geq 0$. Sagt med andre ord, den første operation skal give credit i banken. Hvis det er tilfældet, siger vi ϕ er valid (en gyldig potentialefunktion).

Typisk vælger man ϕ så $\phi(D_0) = 0$ og $\phi(D_i) \geq 0$ for alle $i \geq 0$. Dermed er ϕ altid gyldig.

6.3.1 eksempel på stacken

D_0 er den oprindelige stack og D_i er stacken efter den i'de operation.

Vi vælger $\phi(D_i)$ til at være antallet af elementer på stacken D_i fra i = 0, ..., n

Dette må være en gyldig potentialefunktion, da $\phi(D_0) = 0$ og $\phi(D_i) \geq 0$ for alle i.

1. lad $i \in \{1, ..., n\}$ være givet og tag højde for den i'de operation

2. $\phi(D_i)$ er antallet på stacken D_i

3. Hvis den i'de operation er Push:
   $\phi(D_i) - \phi(D_{i-1}) = 1$

   Derfra kan vi finde den amortiserede omkostning ud fra den reele omkostning, med det udtryk for den amortiserede omkostning vi definerede i starten af metoden:
   $\hat{c_i} = c_i + \phi(D_i) - \phi(D_{i-1}) = 1 + 1 = 2$

4. Hvis den i'de operation er Pop: $\phi(D_i) - \phi(D_{i-1}) = -1$ $\hat{c_i} = c_i + \phi(D_i) - \phi(D_{i-1}) = 1 - 1 = 0$

5. Hvis det er Multipop:
   Lad antallet af elementer i stacken være k og antallet af elementer vi popper være k'
   Lad $k' > 0$, antallet af elementer vi popper
   $\phi(D_i) - \phi(D_{i-1}) = -k'$
   $\hat{c_i} = c_i + \phi(D_i) - \phi(D_{i-1}) = k' - k' = 0$

Derigennem kan vi at i alle tilfælde vil $\hat{c_i} \leq 2$, som giver en amortiseret omkostning på O(1). (ligesom vi fandt med accounting metoden.)

6.4 Applications

7 Del og Hersk (Divide and conquer)

7.1 Divide and conquer

En dynamisk programmeringsalgortime er en algortime, som deler sit problem op i mindre dele og løser dem rekursivt. Derefter sætter den de løste delproblemer sammen til en løsning.

7.2 recursion tree method

Den rekursive træ-metode går ud på at tegne et rekursitvt træ, for at finde ud af hvor meget den kommer til at koste, rekursivt.

En rekursiv ligning består af 2 dele; en rekursiv del kaldet T(n) og en ikke rekursiv del, der kan have forskellige værdier, men ofte bliver betegnet med en eller anden konstant c eller cn. Et tal foran T(n) = 2T(n/2) eller T(n)

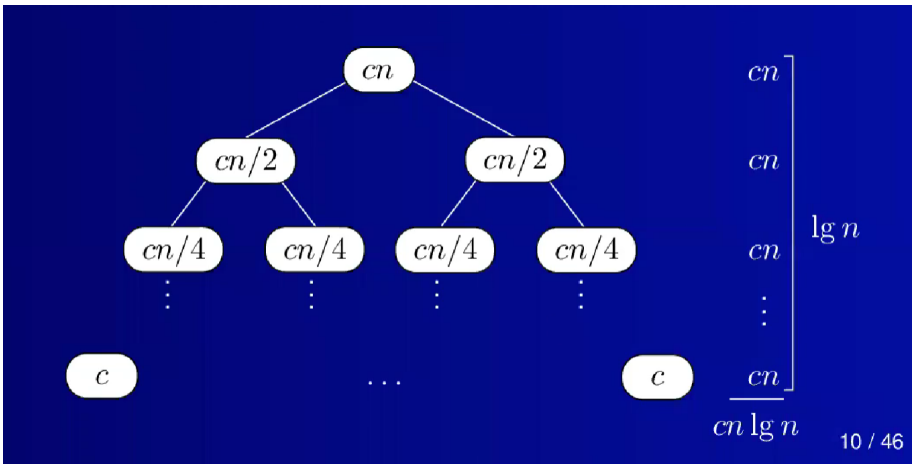
= 4T(n/2) signalerer hvor mange gange rekursionen splittes op ved hvert kald.

Til at forklare metoden, bruger vi merge sort som eksempel.

Merge sorts rekursive køretid er T(n) = 2T(n/2)+Θ(n) for alle n > 1.
Derfor ved vi, at T(n) ≤ 2T(n/2) + cn, for en eller anden konstant c > 0.

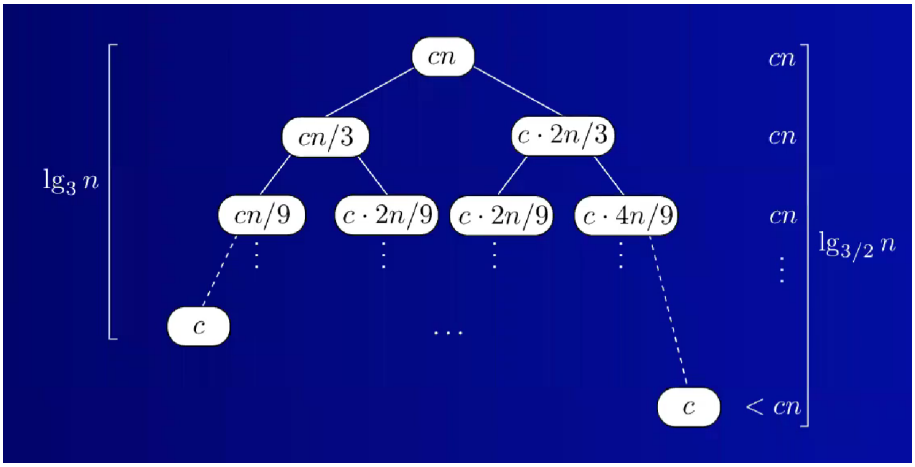
et rekursivt træ har en knude for hvert rekursive kald.
Omkostningen er tiden brugt i den ikke-rekursive del af kaldet.

I merge sorts tilfælde er det ikke-rekursive kald cn:



Figur 7: Det ses for hvert skridt ned i træet bliver listen halveret, og dermed også omkostningen for at sortere. Til sidst er omkostningen en enkelt konstant c, som er der er n mængder af. Den totale omkostning er proportional med træets højde i log n.

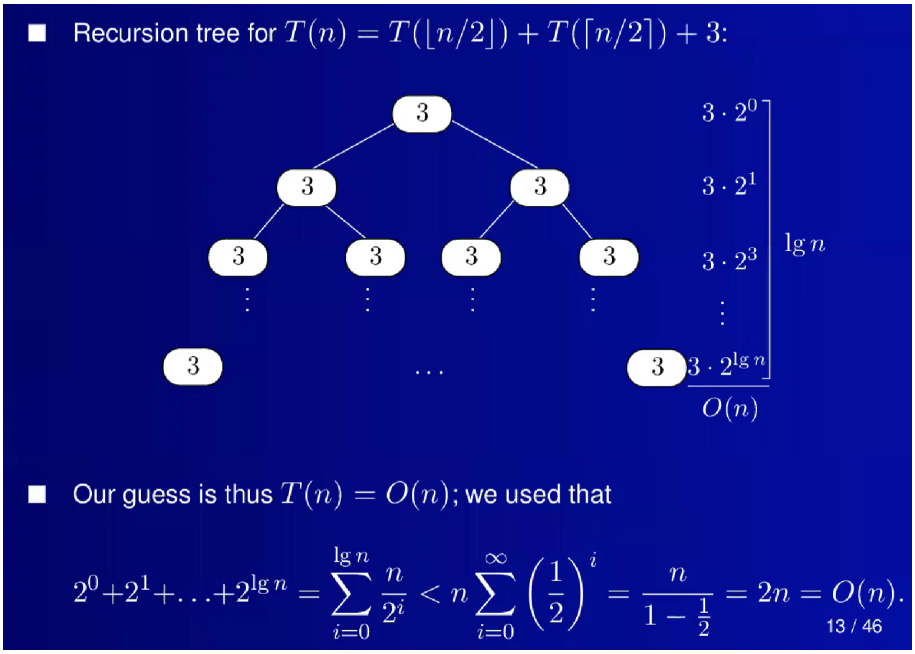
Vi kan også se på et andet tilfælde end merge sort. Her er $T(n) = T(\lfloor n/3 \rfloor + T(\lfloor 2n/3 \rfloor) + cn$. Dermed kan vi se, at den ene rekursion bliver større end den anden, da de henholdsvis bliver splittet op i en tredjedel og 2 tredjedele.



Figur 8: ubalanceret rekursivt træ. Den ene side kommer til at veje mere end den anden.

Vi kan her fra se
$T(n) = T(\lfloor n/3 \rfloor + T(\lfloor 2n/3 \rfloor) + cn > cnlg_{3/2}n = O(nlogn)$. konstante faktorer bliver taget ud.

Vi prøver med et eksempel mere.
I dette rekursionstræ er konstanten 3 for hver gren i træet. Træet er også balanceret, så dets højde er log n. vægten af træet er derfor $3 * 2^n$, hvor n er antal niveauer nede fra roden (læg mærke til på billedet er der en fejl. 3. niveau skal være $3 * 2^2$.)

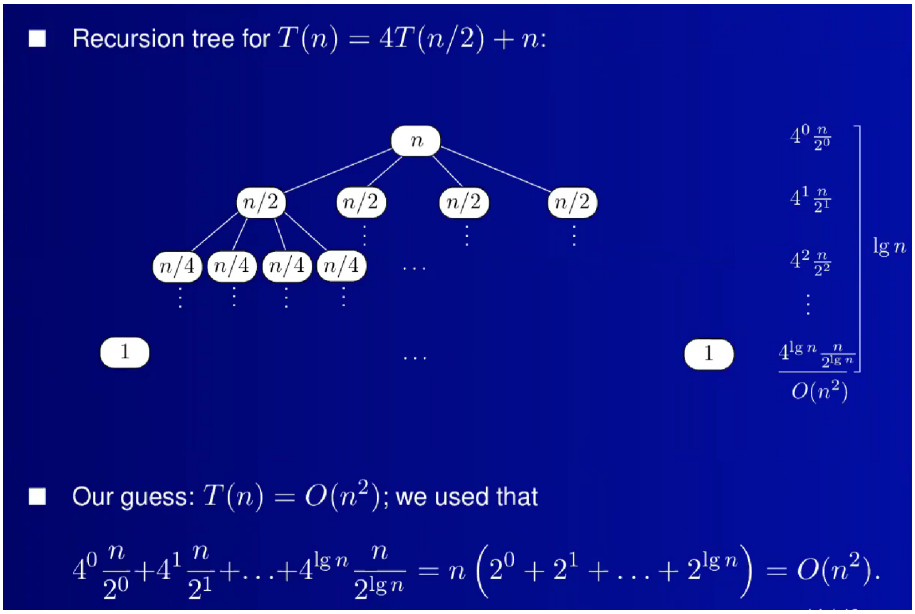


Figur 9: ved at summere træets vægte med dens vækst får man O(n) køretid.

Sidste eksempel:
Træet for T(n) = 4T(n/2) + n:

da træet indeholder 4T, betyder det for hver rekursion splittes det i 4 nye grene, men da dets vægt er (n/2) bliver omkostningen kun halvværet for hvert split.

På billedet nede under ses træet med dets omkostninger ude i siden.



Figur 10: Nederst ses den sidste omkostning. $4^{logn}\frac{n}{2^{logn}} = n^2$, så derigennem ses det, at løsningen må være $T(n) = O(n^2)$

7.3 merge Sort

Good-to-knows omkring merge sort:

1. Merge-sort(A, p, q, r) betyder A er listen, p er det sidste element i listen, q er pivot-elementet, r er det første element i listen.
2. Merge sort laver n*2-1 sammenligninger
3. Merge sort laver merge n-1 gange
4. worst case kørertid er $\theta(nlogn)$

7.4 Substitution Method

Substitutionsmetoden bruges til at finde øvre og nedre bounds i rekursive kald ved hjælp af matematisk induktion.

Metoden består af følgende skridt:

- Gæt formen på løsningen, typisk med O-notation
- Man bruger stærk induktion på n til at finde de skjulte konstanter der får løsningen til at virke.

recursion tree method kan bruges til at komme frem med et gæt.
Derefter beviser vi formelt at gættet er korrekt.
I rekursion tydeliggøres worst case tid af T(n) sjældent for konstanten n.

7.4.1 Eksempel

Løs rekursionen $T(n) = 2T(\lfloor n/2 \rfloor) + n, n > 1$

- Vi går ud fra T(1) = 1, da det tager konstant tid at løse problemer af størrelsen n (og 1).
- Vi gætter: T(n) = O(n log n).
- Vi viser: $T(n) \leq cnlogn$ for alle $n \geq n_0$, for en konstant $c > 0$ og $n_0 > 0$.
- Vi laver induktion over $n \geq n_0$; vi kan vælge c og n_0 lige som vi vil for at få beviset til at passe.
- Et Problem: uligheden T(n) $\leq$ cn log n er falsk, hvis n = 1, lige meget valget af c > 0. Det er fordi log 1 = 0, så c* 1 log 1 bliver også til 0. Men vi kan vælge n_0 som vi vil, så dermed kommer vi uden om problemet.
- Derfor vælger vi kun at bevise uligheden for n $\geq$ 2 ($n_0 = 2$)

Så hvad skal vores base case(s) være?

- til induktionsskridtet skal hypotesen holde for $\lfloor n/2 \rfloor$
- i vores tilfælde har vi brug for 2 basis-tilfælde, da n = 2 og n = 3 begge giver en løsning på 1, da både 2/2 og 3/2 = 1, når vi runder ned med floor.
-

Vi starter med Induktionen:

- Siden $T(n) = 2T(\lfloor n/3 \rfloor) + n$ for $n > 1$, så har vi T(2) = 4 og T(3) = 5.
- ved at vælge n > 2, så har vi $T(n) \leq cnlgn$ for n = 2,3
- for induktionsskridtet antager vi n > 3 og antager, at T(m) $\leq$ cm log m for alle m = 2, 3, .., n-1
- Det vil sige $T(\lfloor n/2 \rfloor) \leq c\lfloor n/2 \rfloor log(\lfloor n/2 \rfloor)$

- Induktionsskridtet (viser det holder for n):

$$T(n) = 2T(\lfloor n/2 \rfloor) + n$$

$$\leq 2c\lfloor n/2 \rfloor log(\lfloor n/2 \rfloor) + n$$

$$\leq cnlog(n/2) + n$$

$$= cnlogn - cnlog2 + n$$

$$cnlogn - cn + n$$

$$\leq cnlogn$$

7.5 Quicksort

- bruger del og hersk til at sortere et array A
- Den vælger et pivotelement (i bogen det sidste element i listen, men man kan også bare vælge et tilfældigt element. Så kaldes algoritmen for Random Quicksort)
- den finder alle elementer i listen der er mindre end det valgte element og putter til venstre, alle elementer der er større og putter til højre.
- Så sorterer den de 2 splittede lister rekursivt.
- Merge sort sorterer in-place. Det vil sige den kun kræver en konstant mængde plads ekstra for at sorterer.
- Worst case for quick sort er når splittene er så ubalanceret som muligt.

Psudokode:

```
Quicksort (A, p, r)
1   if p < r
2       q = PARTITION(A, p, r)
3       Quicksort(A, p, q-1)
4       quicksort =(A, p+1, r)
```

A = array
p = første elemement
r = sidste element

PARTITION: omrokkerer A[p...r] og returnerer et index q, så A[1...q-1] indeholder elementer der er mindre end r og A[q+1...r] indeholder elementer

der er større end q.

Partition kan laves i $\theta(n)$ tid (lineær tid).

7.5.1 sketchy eksamenshack (don't know if work xd)

hvis en rekursionsligning er på formen $T(n) = T(n-1) + nx^y$, så er (n-1) det samme som at gange n med det der bliver plusset på.

fx:
$T(n) = T(n-1) + n = n^2$

7.6 Masters Theorem

Masters Theorem går ud på at sætte værdier a og b ind i en rekursionsligning, for at få et ønsket svar.

Fx hvis man har en ligning på formen T(n) = aT(n/b) + f(n), hvor a $\geq$ 1 og b $>$ 1 som konstanter, så kan man bruge masters metoden.
Se a som antallet af rekursive kald man laver, b er problemstørrelsen, når man går 1 niveau ned i rekursionen. f(n) er den ikke-rekursive del af ligningen.

Hvis en ligning er på den form, kan man bruge master metoden. Master metoden har 3 tilfælde:

- Hvis $f(n) = O(n^{log_b a - \epsilon})$, for en konstant $\epsilon > 0$, så
  $T(n) = \theta(n^{log_b a})$
  (hvis ϵ er tilstrækkelig lille, gælder udtrykket)
- Hvis $f(n) = \theta(n^{log_b a})$, så
  $T(n) = \theta(n^{log_b a} logn)$
- hvis $f(n) = \Omega(n^{log_b a + \epsilon})$ for en konstant $epsilon < 1$ og alle tilstrækkeligt store n, så
  $T(n) = \theta(f(n))$

7.6.1 Procedure til eksamen

Hvis $a \geq 1$ og b $>$ 1, brug følgende:

- Hvis $n^{log_b a}$ $>$ f(n)
  $Theta(n) = T(n) = (nlogba)$

- Hvis $n^{log_b a}$ = f(n)
  T (n) = (nlogbalogn)
- Hvis $n^{log_b a}$ < f(n)
  T (n) = (f (n))

7.6.2 Merge sort som eksempel: $T(n) = 2T(n/3) + \Theta(n)$

- a = 2, b = 2, $f(n) = \Theta(n)$
- Derfra kan vi se at $f(n) = \Theta(n^{log_b a})$ (den står på Theta form).
- Det vil sige vi har $T(n) = \Theta(n^{log_b a} logn)$
- Masters metoden ville også virke, selv om man bruger floor / ceiling.

7.6.3 Exempel 1

$T(n) = T(n/2) + t(n/2) + 3$

Vi går ud fra T(n) = aT(n/b) + f(n), hvor a $\leq$ 1 og b > 1 som konstanter Derved kan T blive asymptotisk bundet som oven over.

- der er 2 rekursive kald, så vi kan sætte a = 2
- Vi kan se b = 2
- Vi kan se f(n) = 3.
- vi udregner $n^{log_b a} = n$. Hvis f(n) er polomielt mindre end $n^{log_b a - \epsilon}$, så gælder tilfælde 1 fra master metoden. Det er rigtigt, da n er en konstant 3, så den er polomielt mindre.
- Dermed får vi løsningen skal være $T(n) = \theta(n^{log_b a} logn) = \Theta(n)$

7.6.4 Exempel 2

$T(n) = 4T(n/2) + n^2$

- Vi kan se a = 4, b = 2, f(n) = n^2
- Siden $n^l og_b a = n^2$, vi er i 2. tilfælde af master metoden.
- Løsning: $T(n) = \Theta(n^{log_b a} logn) = \Theta(n^2 logn)$

7.6.5 Exempel 3

$T(n) = 7T(n/8) + nlogn$

- Vi kan se a = 7, b = 8, f(n) = $nlogn$
- Siden $n^log_b a = n^{0.9357\ldots}$, vi er i 3. tilfælde af master metoden.
- vi har $af(n/b) = 7(n/8)log(n/8) < \frac{7}{8}nlogn = \frac{7}{8}f(n)$
- vi vælger $c = \frac{7}{8}$
- Løsning: $T(n) = \Theta(f(n)) = \Theta(nlogn)$

7.6.6 Exempel 4

køretiden for Strassens algortime. hurtig måde at gange 2 n x m matricer sammen på
$T(n) = 7T(n/2) + dn^2$

- Vi kan se a = 7, b = 2, f(n) = dn^2
- Siden $n^log_b a = n^{2.807\ldots}$, vi er i 1. tilfælde af master metoden.
- Løsning: $T(n) = \Theta(n^{log_b a}) = \Theta(n^{2.807\ldots})$

8 Dynamisk programmering

Bed til gud

Figur 11: Fold dine hænder op mod den eneste ene og bed til dine tilfældige krydser bliver sat korrekt. Hvis du ikke vælger korrekt, var du alligevel aldrig en af guds udvalgte. Lige som jesus døde på korset, får du fejl i Dynamisk Programmering, men dit offer vil rede menneskeheden (det normalfordelte gennemsnit).

Når du nu har bedt til gud, kan du se på denne smukke gennemgang af opgaven om dynamisk programmering, udarbejdet af Kim (manden). Hvis du læser dette, skylder du ham n+1 øl på caféen?

8.1 Dynamisk programmering øveeksamen opgave gennemgang (af Kim)

Kim's gennemgang bruger induktion.

8.1.1 P1) Hvad skal bevises?

Vi skal bevise ved induktion, at

$$d_n = \frac{n(n-3)}{2}$$

for n 3, hvor dn står for antal diagonaler af en given n-kant(n-polygon), gælende for 3-kanter og størrere polygoner.

8.1.2 P2) Holder påstanden for trivielle n?

Vi vil nu vise basis-tilfældet n = 3, derfor

$$d_3 = \frac{(3)(3-3)}{2} = \frac{(3(0)}{2} = 0$$

Dette er hvad vi forventer algoritmen skal give, da en 3-kant har 0 diagonaler. Algoritmen overholder altså basis-tilfældet.

8.1.3 P3) Holder påstanden for alle n?

Vi antager at algoritmen passer for n. Derfor antager vi nu at

$$d_n = \frac{n(n-3}{2}$$

for n 3.

8.1.4 Delmål 1)

Vi vil nu vise at algoritmen gælder for (n + 1).
Vi indsætter derfor nu (n + 1) i algoritmen ovenover og reducerer ved hjælp af 1. kvardatsætning, hvorfor at

$$d_{n+1} = \frac{n+1)((n+1)-3)}{2} = \frac{(n^2+1+2n)-(3n+3)}{2} = \frac{n^2+1+2n-3n-3}{2}$$

Vi reducerer, omrokkere og sætter n ud for parentes. så derfor

$$d_{n+1} = \frac{n^2+1+2n-3n-3}{2} = \frac{n(n-3)+(2n-3)}{2}$$

Vi husker på ovenstående formel, til smart brug senere. Vi fortsætter derfor med reduceringerne, så

$$d_{n+1} = \frac{n(n-3)+2n-2}{2} = \frac{n^2-3n}{2} + \frac{2n-3}{2}$$

Vi genkender udtrykket $\frac{n(n-3)}{2}$ som d_n pr. antagelse fra (P3), og anser $\frac{2n-2}{2}$ bare som en rest. Vi har nu at

$$d_{n+1}\frac{n^2-3n}{2} + \frac{2n-3}{2} = d_n + \frac{2n-2}{2} = d_n$$

Vi isolerer nu d_n, som et smart trick, hvorfor at

$$d_{n+1} = d_n + \frac{2n-3}{2} = d_n \Leftrightarrow d_n = d_{n+1} - \frac{2n-2}{2}$$

8.2 Delmål 2)

Siden vi har
$d_n = d_{n+1} - \frac{2n-2}{2}$,
da indser vi, at vi blot skal vise
$d_n = d_{n+1} - \frac{2n-2}{2} = \frac{n(n-3)}{2}$, for at være færdig.

Derfor erstatter vi nu d_{n+1} med formlen fra tidligere, hvorfor vi har at

$$d_n = d_{n+1} - \frac{2n-2}{2} = \frac{n(n-3) + (2n-2}{2} - \frac{2n-2}{2}$$

$$= \frac{n(n-3) + (2n-2) - (2n-2)}{2} = \frac{n(n-3) + 0}{2}$$

$$\frac{n(n-3)}{2}$$

Hvilket var hvad vi skulle vise. Dermed har vi vist at algoritmen gælder for (n + 1), og induktionsbeviset er fuldendt.