# Algoritmer og Datastrukturer

Niklas Magnussen - blg515

31. marts 2019

## Indhold

# 1 Del og Hersk

---

- Merge-sort
  - Rekursionsligningen
  - Rekurstionstræer
  - Substitutions metoden
  - Master metoden
- Bevis for nedre grænse for sortering
- Tætteste par af punkter

## 1.1 Merge-sort

Del og hersk er et algoritme paradigme

En del og hersk algoritme deler ofte problemet op rekursivt indtil den når et base case, hvorefter den løser delproblemerne.

Vi tager udgangspunkt i merge-sort, som er en sorterings algoritme der gør brug af del og hersk paradigmet for at sortere. Dens 3 trin er:

- Del et array op i midten så du har 2 arrays der skal sorteres.
- Bliv ved med at dele til hvert array består af 1 element. Voila! Dette array er nu trivielt sorteret.
- Foren 2 sorterede arrays til 1 array, således at det nye array også er sorteret.

Merge-sort gør brug af en vigtig hjælpefunktion **merge**. Merge funktionen forener 2 arrays sådan at de er sorterede efter forening.

**Algorithm 1** Merge hjælpe funktionen

```
1:  A[p..q..r] ← Delarray der skal sorteres
2:  function MERGE(A, p, q, r)
3:      n_1 = q − p + 1
4:      n_2 = r − q
5:      left[1..n_1 + 1]
6:      right[1..n_2 + 1]
7:      for i = 1 to n_1 do
8:          left[i] = A[p + i − 1]            ▷ Fylder left med venstre delarray
9:      for i = 1 to n_2 do
10:         right[i] = A[q + j]               ▷ Fylder right med højre delarray
11:     left[n_1 + 1] = ∞
12:     right[n_2 + 1] = ∞                    ▷ Sætter sentinel værdier for enden
13:     i = 1
14:     j = 1
15:     for k = p to p do
16:         if left[i] ≤ right[j] then
17:             A[k] = left[i]
18:             i = i + 1
19:         else
20:             A[k] = right[j]
21:             j = j + 1
```

Merge funktionen har en køretid på $\mathcal{O}(n)$

Selve Merge-sort funktionen er meget simpel.

**Algorithm 2** Merge-sort funktionen

```
1:  A[p..q] ← Delarray der skal sorteres
2:  function MERGE-SORT(A, p, q)
3:      if p < r then
4:          q = ⌊(p+r)/2⌋
5:          MERGE-SORT(A, p, q)
6:          MERGE-SORT(A, q + 1, r)
7:          MERGE(A, p, q, r)
```

### 1.1.1 Rekursionsligningen

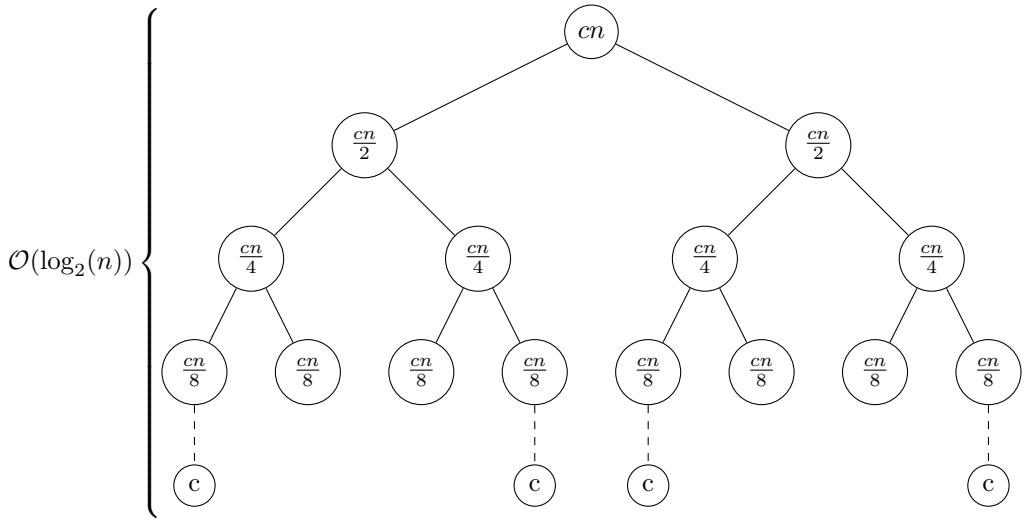
For at beskrive del og hersk algoritmers køretid, skal vi have en måde at beskrive rekursive funktioners køretid. Dette gøres med rekursionsligninger. Hvis vi kigger på merge-sorts køretid $T(n)$, så ser vi, at den splitter arrayet op i to stykker som den rekursivt kalder merge-sort på, det giver $T(n) = T(\lfloor \frac{n}{2} \rfloor) + T(\lceil \frac{n}{2} \rceil)$, det viser sig at vi kan fjerne floor og ceil funktionerne da de ikke ændre køretiden asymptotisk, så vi får $T(n) = 2T(\frac{n}{2})$. Vi skal også huske, at merge-sort kalder merge funktionen i hvert rekursive kald, hvilket lægger $cn$ til hvert rekursive kald, hvor $c$ beskriver de konstanter i merges køretid. Så den endelige rekursionsligning for merge-sort bliver

$$T(n) = 2T(\frac{n}{2}) + cn$$

### 1.1.2 Rekursionstræer

I nogle tilfælde, når man skal beskrive rekursionligninger asymptotisk, kan det være meget nyttigt at tegne et rekursionstræ, der viser per-level-cost af hvert rekursionskald, og hvor dybt rekursionskaldene går. Rekursionstræet for merge-sort ses på figur 1.

Figur 1: Rekursionstræ for merge-sort $T(n) = 2T(\frac{n}{2}) + cn$



For bestemme køretiden ud fra rekursionstræet summeres alle per-level-costs for rekursionslagene. Hver lags cost summeres til $cn$, hvilket er $\mathcal{O}(n)$, og fordi $n$ halveres hver gang er træet $\log_2(n)$ dybt $+1$ for det oprindelige kald. Køretiden er derfor $cnlgn+cn = \mathcal{O}(n\log_2(n))$. For at være mere grundig i sin argumentation, kan man bruge substitutionsmetoden efter man har lavet sit rekursionstræ.

### 1.1.3 Substitutionsmetoden

Substitutionsmetoden er en anden måde at analysere rekursionsligninger. Substitutionsmetoden består af to trin

- Lav et gæt $f(n)$. Lav evt. et rekursionstræ for at få en fornemmelse af rekursionsligningens køretid.
- Bevis at gættet er korrekt. Her bruges matematisk induktion til at vise at rekursionsligningen er $T(n) = \mathcal{O}(f(n))$, ved at antage $T(m) \leq cf(m), m < n$ og derefter vise $T(n) \leq cf(n)$.

I tilfældet med merge-sort, skal vi bevise at $T(n) = \mathcal{O}(nlgn)$, som er vores gæt ud fra rekursionstræet, ved at vise $T(n) \leq cnlgn + cn$, som er vores gæt fra rekursionstræet:

Vi starter med base case. $T(1) = c$, da der ikke bliver lavet nogen rekursive kald. Det giver

$$\begin{aligned} T(1) = c &\leq c1lg1 + c \\ &= c \leq c \end{aligned}$$

Vi antager nu at $T(m) \leq cmlgm$:

***Bevis.***

$$\begin{aligned} T(n) &= 2T(\frac{n}{2}) + n \leq cnlgn + cn \\ &= 2(c\frac{n}{2}lg\frac{n}{2} + c\frac{n}{2}) + cn \leq cnlgn + cn \\ &= 2(\frac{cnlg2}{2} - \frac{cn}{2} + \frac{cn}{2}) + cn \leq cnlgn \\ &= cnlgn - cn + cn + cn \leq cnlgn \qquad = cnlgn + cn \leq cnlgn + cn \end{aligned}$$

Ergo har vi at $T(n) = cnlgn + cn = \mathcal{O}(nlgn)$. ■

### 1.1.4 Master metoden

I Master metoden tager man udgangspunkt i rekursionsligninger med formen

$$T(n) = aT(\frac{n}{b}) + f(n)$$

og sammenligner dem med $n^{\log_b(a)}$. Rekursionsligningen beskriver at vi vi deler vores problem op i $a$ bidder, hver af størrelse $\frac{n}{b}$, og hvor tiden for de ikke-rekursive kald er $f(n)$.

Der er tre tilfælde i master method, som beskrives i **master theorem**

**Theorem 1.1.** *Lad $a \geq 1$ og $b \geq 1$ være konstanter. Lad $f(n)$ være en asymptotisk positiv funktion. Lad $T(n) = aT(\frac{n}{b}) + f(n)$ være en rekursionsligning på de positive heltal, og hvor $\frac{n}{b}$ kan betyde både $\lfloor \frac{n}{b} \rfloor$ og $\lceil \frac{n}{b} \rceil$.*
*Så har $T(n)$ følgende asymptotiske grænser:*

1. *hvis $f(n) = \mathcal{O}(n^{\log_b(a)-\epsilon})$ for $\epsilon > 0$, så er $T(n) = \Theta(n^{\log_b(a)})$*
2. *hvis $f(n) = \Theta(n^{\log_b(a)})$, så er $T(n) = \Theta(n^{\log_b(a)} lgn)$*
3. *hvis $f(n) = \Omega(n^{\log_b(a)+\epsilon})$ for $\epsilon > 0$, så er $T(n) = \Theta(f(n))$*

Hvis vi bruger master metoden på merge-sort, ser vi at vi har $a = 2, b = 2, f(n) = n$, og vi kan sammenligne rekursionsligningen med $n^{\log_b(a)}$:

$$n = \Theta(n^{\log_2(2)}) \Rightarrow f(n) = n = \Theta(nlgn)$$

master method virker kun i de tilfælde hvor $f(n)$ enten er polynomisk større, i 1. tilfælde, og polynomisk mindre, i 3. tilfælde, end $n^{\log_b(a)}$. Det er grunden til at vi inkluderer en konstant $\epsilon > 0$

## 1.2 Bevis for nedre grænse for sortering

Vi har brugt merge-sort som har en køretid på $\mathcal{O}(nlgn)$, som vores eksempel, og det kan vises at den køretid er så god som det kan blive for sammenlignings-sorteringer.

Vi tager udgangspunkt i et **decision-tree**, hvor en knude repræsenter en sammenligning mellem to elementer $i : j$ i listen der skal sorteres. Hvis man går til venstre bytter man hvis $i \leq j$, og til højre bytter man når $i < j$.

**Theorem 1.2.** *Enhver sammenligningssorteringsalgoritme er* $\Omega(nlgn)$.

***Bevis.*** Der er $n!$ permutationer af listen og alle knuderne repræsenterer en forskellig permutation. En sortering svarer derfor til at tage nogle beslutninger i vores decision-tree indtil man ender i den permutation der er den sorterede, og vi skal derfor finde højden $h$ for at finde hvor mange trin algoritmen skal tage.

Vi ved vi har med et fuldt binært træ, derfor er der højst $2^h$ blade, og vi har altså

$$n! \leq 2^h$$

Og vi finder højden ved at tage logaritmen

$$lg(n!) \leq h$$

$n! = \Theta(n^n)$ og vi har derfor

$$nlgn \leq h$$

Ergo er $h = \Omega(nlgn)$. ■

## 1.3 Tætteste par af punkter

Et andet problem der løses med del og hersk, er problemet med at finde det tætteste par af punkter. Givet en mængde punkter $Q$ med $n >\geq 2$ punkter skal vi finde to punkter $p_1 = (x_1, y_1)$ og $p_2 = (x_2, y_2)$ der har den mindste Euklidiske afstand $d(p_1, p_2) = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$. Den naive måde at løse problemet på, er ved at se på alle $\binom{n}{2)=\Theta(n^2)}$ punkter, og returnerer dem med lavest afstand.

Del og hersk algoritmen har rekursionsligning magen til merge-sort – $T(n) = 2T(\frac{n}{2}) + \mathcal{O}(n) = \mathcal{O}(nlgn)$. Hvert rekursive kald tager som argumenter en delmængde $P \subset Q$, arrays $X, Y$ som hver indeholder henholdsvis punkterne i $P$ sorteret efter stigende $x$-koordinat, og stigende $y$-koordinat. De 3 trin i del og hersk er for denne algoritme

- **Del**: Rekursivt del $P$ på en vertikal linje $l$ så vi har $|P_L| = \lfloor \frac{|P|}{2} \rfloor$ og $|P_R| = \lceil \frac{|P|}{2} \rceil$. Vi deler også $X$ og $Y$ op og sorterer delarrays, så vi får

$X_L, X_R$ og $Y_L, Y_r$. Vi stopper de rekursive kald, når $|P| \leq 3$, hvorefter vi bruger den naive måde.

- **Hersk**. Vi prøver at finde de tætteste par af punkter $\delta_L$ i $P_L$ og $\delta_R$ i $P_R$, og vi lader $\delta = min(\delta_L, \delta_R)$.
- **Foren**: Det tætteste par af punkter er enten begge i $P_L$ eller begge i $P_R$, hvor $\delta$ så er den korteste afstand. Det er også muligt at det tætteste par af punkter, er et punkt $p_L \in P_L$ og $p_R \in P_R$, og algoritmen må tjekke for dette tilfælde.

  Hvis $p_L$ og $p_R$ er de tætteste par af punkter, så må de være indenfor et område der stikker $\delta$ ud fra $l$ på begge sider. For at finde tætteste par af punkter inden for dette vertikale område gør vi følgende:

  1. Lad $Y'$ være alle de punkter inden for det vertikale område, sorteret efter stigende $y$-koordinat.
  2. For hvert punkt $p \in Y'$ tjek de næste 7 punkter efter $p$, og hold øje med den korteste afstand $\delta'$ til disse punkter.
  3. Hvis $\delta' < \delta$ så returner $\delta'$, ellers returner $\delta$. I begge tilfælde returnerer vi også de to tætteste punkter.

**Korrekthed**:
Det er let at se hvordan den vælger $\delta$, så det vi skal vise for at vise korrekthed, er hvordan vi finder $\delta'$. Vi vil nu bevise hvorfor vi kun skal tjekke afstanden mellem punkt $p \in Y'$ og de 7 næste punkter i $Y'$.

***Bevis.*** Vi antager at den korteste afstand efter en forening, er mellem punkterne $p_L \in P_L$ og $p_R \in P_R$ med afstand $\delta' < \delta$. Siden de befinder sig i det vertikale område, må de være inden for en afstand af $\delta$ i deres vertikale afstand, og inden for en afstand af $\delta$ til $l$. Dette giver et $\delta \times 2\delta$ rektangel som de kan befinde sig i.

Vi viser nu hvorfor der højst kan være 8 punkter i sådan et rektangel. Hvis vi den ene halvdel af rektanglet, den side der er i $P_L$, har vi et kvadrat med sidelængder $\delta$. Det højeste antal punkter vi kan have i det kvadrat, med sidelængder ikke

mindre end $\delta$, er 4; et i hvert hjørne. Hvis vi antager at der er et 5. punkt i kvadratet, så må det ligge et sted, hvor dets afstand til et af de andre punkter er mindre end $\delta$, hvilket vil betyde at den korteste afstand ikke er mellem $p_L$ og $p_R$, men mellem to punkter i $P_L$. Det samme gør sig gældende i kvadratet i $p_R$. Der er 4 punkter som ligger på linjen $l$ – to af hjørnerne i $P_L$ og to i $P_R$. Siden $Y'$ er sorteret, behøver vi kun tjekke de 7 punkter efter et givet punkt, for at finde det tætteste. ■

For at opnå en køretid på $\mathcal{O}(nlgn)$ **præsorterer** vi $X$, og $Y$, før det første rekursive kald, og i hvert rekursive kald, splitter vi dem i en *omvendt* `MERGE`, der splitter dem i $X_L, X_R$ og $Y_L, Y_R$ således, at de splittede arrays stadig er sorterede, hvilket tager $\mathcal{O}(n)$ tid. Da vi laver rekursive kald på $|P_L| = \lfloor \frac{|P|}{2} \rfloor$ og $|P_R| = \lceil \frac{|P|}{2} \rceil$, er det let at se rekursionsligningen $T(n) = 2T(\frac{n}{2}) + \mathcal{O}(n) = \mathcal{O}(nlgn)$.

# 2 Fibonacci Hobe

---

- Operationer på en Fibonacci hob
- Potentiale funktionen
- Mergeable hob
- Knude operationer

## 2.1 Operationer på en Fibonacci hob

Fibonacci hobe er en **mergeable heap** datastruktur, hvilket vil sige, at den består af en disjunkt mængde af hobe der understøtter nogle bestemte operationer til at skabe og flette hobe.

| Operation | Binært træ (worst-case) | Fibonacci hob (amotiseret) |
|---|---|---|
| `MAKE-HEAP` | $\Theta(1)$ | $\Theta(1)$ |
| `INSERT` | $\Theta(lgn)$ | $\Theta(1)$ |
| `MINIMUM` | $\Theta(1)$ | $\Theta(1)$ |
| `EXTRACT-MIN` | $\Theta(lgn)$ | $\mathcal{O}(lgn)$ |
| `UNION` | $\Theta(n)$ | $\Theta(1)$ |
| `DECREASE-KEY` | $\Theta(lgn)$ | $\Theta(1)$ |
| `DELETE` | $\Theta(1)$ | $\Theta(1)$ |

Fibonacci hoben består af en cirkulær **doubly linked-list**, kaldet rod-listen, som består af rødderne til nogle min-hobe. Hver rod, har en peger til den næste rod, og selve Fibonacci hoben er en peger til den mindste af rødderne, så man altid kan hive det mindste element ud af Fibonacci hoben. Hver knude i en hob kan have adskillige børn, og børnene er også forbundet af en cirkulær doubly linked-list, så hvert barn har en peger til sin forrige og næste søskende.

Hver knude $x$ har følgende felter

- $x.left, x.right$ venstre og højre pegere i den liste knuden befinder sig i. Enten rod-listen eller en børne-liste
- $x.child$ er en peger til et vilkårligt barn i sin børne-liste
- $x.parent$ er en peger til sin forælder. Hvis denne peger er `NIL` er knuden en rod
- $x.degree$ er antallet af børn
- $x.mark$ er en boolsk værdi

Selve Fibonacci hoben $H$ har kun to felter; $H.min$ er pegeren til den mindste knude i rod-listen. $H.n$ er antallet af knuder i Fibonacci hoben.

## 2.2 Potentiale funktionen

Når vi skal udregne den amortiserede pris for de forskellige operationer, bruger vi en potential funktion

$$\Phi(H) = t(H) + 2m(H)$$

Hvor $t(H)$ er antallet af rødder i rod-listen, og $m(H)$ er antallet af markerede knuder i Fibonacci hoben. Vi antager at vi starter med en tom Fibonacci hob, og $\Phi(H)$ starter med at være 0, og bliver aldrig negativ.
Vi har også, at en knudes maksimale degree, i et træ med $n$ knuder, er $D(n) = \mathcal{O}(lgn)$.

# 3 Balancerede Binære Søgetræer

---

- Operationer på et binært søgetræ
  - Trævandring
  - Søgning på et binært søgetræ
  - Indsættelse og sletning
- Rød-sort træ - Balanceret binært søgetræ
  - En øvre grænse for højden
  - Indsættelse
  - Sletning
- Krydsende linjesegmenter

## 3.1 Operationer på et binært søgtræ

### 3.1.1 Trævandring

**Theorem 3.1.** *Lad $x$ være en rod til deltræ med $n$ knuder, så vil et kald til en trævandringsfunktion e.g.* `INORDER-TREE-WALK(x)` *tage $\Theta(n)$ tid.*

***Bevis.*** Det er let at se at det er $\Omega(n)$, da vi skal besøge alle $n$ knuder, så vi mangler bare at vise at den er $\mathcal{O}(n)$.

Vi siger at prisen for at lave checket ($x \neq$`NIL`) er $c > 0$, så $T(0) = c$. For $n > 0$ lader vi $x$ have to børn, så $x.left.size = k$ og $x.right.size = n - k - 1$, så er tiden for træet

$$T(n) \leq T(k) + T(n - k - 1) + d$$

Hvor $d > 0$ er tiden det tager for indmaden i algoritmen e.g. `PRINT`.

Vi viser nu $T(n) = \mathcal{O}(n)$ ved at bruge substitutionsmetoden til at vise $T(n) \leq (c+d)n+c$ ($T(n)$ er det første test plus indmaden for hvert rekursivt kald, plus den sidste test når $n = 0$):
Først viser vi base-case

$$\begin{aligned} T(0) = c &\leq (c + d)0 + c \\ &= c \leq c \end{aligned}$$

Nu antager vi at $T(m) \leq (c + d)m + c$ for $m < n$

$$\begin{aligned} T(n) &\leq T(k) + T(n - k - 1) + d \leq (c + d)n + c \\ &= ((c + d)k + c) + ((c + d)(n - k - 1) + c) + d \leq (c + d)n + c \\ &= ck + dk + c + cn - ck - c + dn - dk - d + c + d \leq (c + d)n + c \\ &= cn + dn + c \leq (c + d)n + c \end{aligned}$$

Vi har nu vist at $T(n) = \Omega(n)$ og $T(n) = \mathcal{O}(n)$, ergo $T(n) = \Theta(n)$. ■

### 3.1.2 Søgning på et binært træ

**Theorem 3.2.** *Søgningsoperationerne* `SEARCH, MINIMUM, MAXIMUM, SUCCESSOR,` *og* `PREDECESSOR` *har alle en køretid på $\mathcal{O}(h)$ på et binært søgetræ med højde $h$.*

***Bevis.*** I `SEARCH`$(x)$ kan vi, på grund af binær søgetræsegenskaben, følge en simpel path fra roden til knuden $x$, ved at se om $x$ er større eller mindre end den nuværende knude. Siden en simpel path er $\mathcal{O}(h)$ er `SEARCH` $= \mathcal{O}(h)$.

`MINIMUM` og `MAXIMUM` er endnu lettere. På grund af binær søgetræsegenskaben, er den mindste knude bladet længst til venstre, og den største knude er bladet længst til højre. Igen er det en simpel path som er $\mathcal{O}(h)$.

`PREDECESSOR` og `SUCCESSOR` er lidt mere komplicerede. Her udnytter vi også binær søgetræsegenskaben. For at finde den mindste knude der er større end $x$, er der to tilfælde at tage højde for. Hvis $x$ højre deltræ findes, så er den næste knude den mindste knude i højre deltræ, som vi kan finde i $\mathcal{O}(h)$ tid. Hvis ikke, så går vi iterativt op til vores forælder, indtil vores forælder er et venstre barn af en knude, hvorefter vi returnerer vores bedsteforælder. I værste tilfælde skal vi følge en simpel path fra et blad til roden, hvilket også er $\mathcal{O}(h)$. Det er symmetrisk at finde den største knude der er mindre end $x$.

Vi har nu vist at alle søgningsoperationer er $\mathcal{O}(h)$ på et binært søgetræ med højde $h$. ■

#### Indsættelse og sletning

**Theorem 3.3.** *Vi kan implementerer `INSERT` og `DELETE` i $\mathcal{O}(h)$ tid, på et binært søgetræ med højde $h$.*

***Bevis.*** ■

## 3.2 Rød-sort træ - Balanceret binært søgetræ

Et rød-sort træ er et binært træ med et ekstra felt i hver knude, dets farve $x.color$ som er enten rødt eller sort, heraf navnet.
Et rød-sort træ er et **balanceret** binært træ, som sørger for at holde en øvre grænse på højden $h$. Det understøtter følgende egenskaber:

- Hver knude er enten rød eller sort.
- Roden af træet er sort.

- Hvert blad er $T.nil$.
- Hvis en knude er rød, er begge dets børn sorte.
- For hver knude, alle simple paths ned til et blad indeholder lige mange sorte knuder.

Rød-sort træet bruger et **NIL** felt $T.nil$ til at fungere som alle bladene og som rodens forælder. $T.nil$ har farven sort, så alle blade i træet bliver set som sorte.

Vi definerer også en funktion $bh(x)$ på en knude $x$ til at være den knudes **black-height**, som er antallet af sorte knuder, eksklusiv sig selv, i en simpel path fra $x$ ned til et blad.

### 3.2.1 En øvre grænse for højden

**Lemma 3.4.** *Et rødt-sort træ med n knuder har en højde på højst* $2lg(n+1) = \mathcal{O}(lgn)$

***Bevis.*** Vi starter med at vise, at et deltræ med rod $x$ har $2^{bh(x)} - 1$ knuder. Vi beviser det med induktion på højden af deltræet.
Hvis $h = 0$ har vi at $x$ må være et blad og deltræet under $x$ har derfor

$$2^{bh(x)} - 1 = 2^o - 1 = 1 - 1 = 0$$

knuder under sig.

Hvis $h > 0$, er $x$ roden til et deltræ og har to børn, som hver især har enten $bh(x)$ eller $bh(x) - 1$, hvis de er henholdsvis røde eller sorte. Det betyder også, at hvert af børnenes deltræer har mindst $2^{bh(x)-1} - 1$ knuder under sig, og $x$ har derfor

$$(2^{bh(x)-1} - 1) + (2^{bh(x)-1} - 1) + 1 = 2^{bh(x)} - 1$$

knuder under sig, og vi har nu vist den første påstand.

For at færdiggøre beviset, lad vi højden af træet være $h$. Ifølge den 4. rød-sort egenskab, er mindst halvdelen af knuderne, eksklusiv roden, i en simpel path

ned af træet, sorte. Derved er $bh(x)$ af roden $x$ mindst $\frac{h}{2}$, og vi har derfor

$$\begin{aligned} n &\geq 2^{\frac{h}{2}} - 1 \\ n+1 &\geq 2^{\frac{h}{2}} \\ lg(n+1) &\geq \frac{h}{2} \\ 2lg(n+1) &\geq h \end{aligned}$$

Ergo har vi vist at højden er $h$ er højst $2lg(n+1)$ og derved $h = \mathcal{O}(lgn)$. ■

Ved at vise at rød-sort træer har en øvre grænse på højden af $\mathcal{O}(lgn)$, betyder det, at alle operationer på et binært træ der havde $\mathcal{O}(h)$ har nu en køretid på $\mathcal{O}(lgn)$.

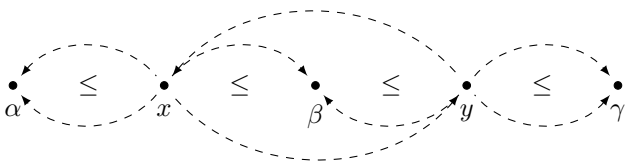
### 3.2.2 Rotation

Rotation er en ombytning af pointere, som bruges til at opretholde rød-sort egenskaber efter en indsættelse eller sletning.

Left-rotate


Intuitionen for hvorfor man må rykke på pegerne kan ses her.
Nedenfor ses et diagram af forholdene mellem knuderne $x, y$ og deltræerne $\alpha, \beta, \gamma$. Pilene er en peger til en knudes børn. Så længe en peger for en knudes højre barn, peger til en knude til højre, og vice versa for venstre peger, så kan man rykke rundt på pegerne.

Før left-rotate




Efter left-rotate

Right-rotate er den inverse af left-rotate.

### 3.2.3 Indsættelse

Vi kan indsætte i $\mathcal{O}(lgn)$ tid, med en lille ændring i algoritmen. Ved kald af `INSERT`$(T, z)$ indsætter vi $z$ i vores træ som normalt og giver den en rød farve $z.color = red$. Derefter bruger vi en hjælpe funktion `RB-INSERT-FIXUP`, som sørger for at træet stadig overholder rød-sort egenskaberne efter indsættelse.

Køretiden af en indsættelse er $\mathcal{O}(lgn)$. Det tager $\mathcal{O}(lgn)$ at vandre ned af træet for at finde den rigtige plads at indsætte, og derefter tager `RB-INSERT-FIXUP` $\mathcal{O}(lgn)$ da den går fra bladet op til roden.

### 3.2.4 Sletning

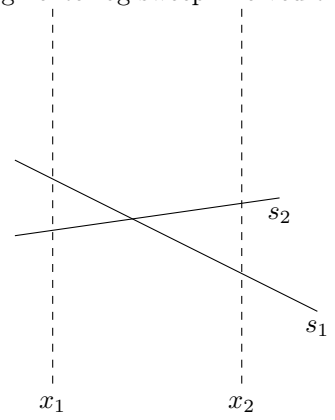
Sletning tager også $\mathcal{O}(lgn)$ tid. Her bruges også en hjælpefunktion `RB-DELETE-FIXUP`, til at vedligeholde rød-sort egenskaberne, som er $\mathcal{O}(lgn)$.

## 3.3 Krydsende linjesegmenter

Hvis man har en mængde linjesegmenter og man ønsker at finde ud af om nogen af linjesegmenterne krydser hinanden kan man bruge en metode kaldet **sweeping**.
Sweeping algoritmen bruger en lodret **sweep line** som kører over mængden af linjesegmenter, og vi betegner den dimension som sweep line bevæger sig over, $x$-aksen, en dimension af tid. Vi antager to ting om vores linjesegmenter, ingen af linjesegmenterne er lodrette linjer, og højst to linjesegmenter kan overlappe på et punkt.

Vi bestemmer en relation $\succeq$ mellem to segmenter $s_1$ og $s_2$ i forhold til deres $y$ position ved et bestemt tidspunkt. Vi siger at $s_1 \succeq_x s_2$, hvis $s_1$ ligger højere end $s_2$ ved tidspunkt $x$.

Figur 2: Linjesegmenter og sweep line ved tidspunkt $x_1$ og $x_2$




Som kan ses på figur 2, så er $s_1 \succeq_{x_1} s_2$ og $s_2 \succeq_{x_2} s_1$.
Vi holder styr på forholdene mellem alle segmenterne, i en rangeret orden. Et segment kommer med i ordenen når sweep line passere segmentets venstre ende, og den bliver taget ud af ordenen når sweep line passere dens højre ende.

Når to linjesegmenter krydser hinanden kan det ses på deres forhold før og efter sweep line har passeret deres krydsningspunkt. Hvis vi har to segmenter $e, f$, så vil $e \succeq_v f$, hvor $v$ er tidspunket før krydset, og $f \succeq_w e$, hvor $w$ er tidspunktet efter krydset. Præcis i krydsningspunktet vil både $e \succeq f$ og $f \succeq e$.

Selve sweep algoritmen sorterer alle linjesegmenterne i forhold deres ender, således at de venstre endepunkter kommer før de højre, hvis de er ens så kommer den med laveste $y$-værdi først. Vi holder styr på dette i et rød-sort træ, hvor, når vi sammenligner knuder i træet, laver vi krydsprodukt mellem to linjesegmenters endepunkter.

Algoritmen til at finde krydsende linjesegmenter bruger 4 hjælpefunktioner til rød-sort træet:

- `INSERT`$(T, s)$ indsætter linjesegment $s$ i træet $T$
- `DELETE`$(T, s)$ fjerner linjesegment $s$ fra træet $T$
- `ABOVE`$(T, s)$ returnere linjesegmentet over $s$, således at $s' \succeq s$. Svarer til `SUCCESSOR` i binær søgetræet.

- `BELOW`$(T, s)$ returnere linjesegmentet under $s$, således at $s \succeq s'$ Svarer til `PREDECESSOR` i binær søgetræet.

**Algorithm 3** Algoritme der finder krydsende linjesegmenter

1: $S$ en mængde af $n$ linjesegmenter
2: **function** ANY-SEGMENTS-INTERSECT($S$)
3: $T$ = Rød-sort træ
4: Sorter alle linjesegmenternes endepunkter, venstre før højre, lavest før højest.
5: **for each** endepunkt $p$ i den sorterede liste **do**
6: **if** $p$ er venstre endepunkt af linjesegment $s$ **then**
7: INSERT($T, s$)
8: **if** (ABOVE($T, s$) findes og krydser $s$) **or** (BELOW($T, s$) findes og krydser $s$)) **then**
9: **return** TRUE
10: **if** $p$ er højre endepunkt af linjesegment $s$ **then**
11: **if** både ABOVE($T, s$) og BELOW($T, s$) findes, og ABOVE($T, s$) krydser BELOW($T, s$) **then**
12: **return** TRUE
13: DELETE($T, s$)
14: **return** FALSE

Algoritmen bruger to datastrukturer: **sweep-line status** (**fejelinje**) holder relationer mellem punkterne som sweep-line har mødt. Til denne bruger vi et binært søgetræ, hvor binær søgetræs egenskaben er, at en knude (linjesegmen) $s_2$ er venstre barn af en forælder $s_1$, hvis $s_1 \succeq s_2$, og er højre barn hvis $s_2 \succeq s_1$. Derudover bruger vi også **event-point schedule** (**hændelseslisten**), som er et array af punkter sorteret efter stigende $x$-koordinat. Vi tilføjer et linjesegment til sweep-line status, når et punkt i event-point schedule er et venstre endepunkt, og fjerner linjesegmentet, når punktet er et højre endepunkt.

Algoritmen looper over event-point schedule punkterne $p$, Hvis det punkt er et venstre endepunkt, tjekker vi punktets linjesegment $s$ `ABOVE` og `BELOW` om de krydser med $s$, hvis de gør, returnerer vi `TRUE`. Hvis $p$ er et højre endepunkt,

tjekker vi om segmenterne `ABOVE` og `BELOW` om de krydser med hinanden, og returnerer `TRUE` hvis de gør. Til sidst returnerer vi `FALSE`, hvis ingen krydsende linjesegmenter blev fundet.

**Køretiden** af algoritmen er $\mathcal{O}(nlgn)$. Vi sorterer alle punkterne, hvilket kan gøres i $\mathcal{O}(nlgn)$ med f.eks. merge-sort. Derefter looper vi over alle $2n$ punkter, hvor $n$ er antallet af linjesegmenter, og tiden inde i loopet er $O(lgn)$, siden vi bruger et rød-sort træ, til at implementerer sweep-line status. Selve algoritmen til at finde om to linjesegmenter krdyser, er $\mathcal{O}(1)$, da den kan implementeres med simple krydsprodukt operationer. Derfor er køretiden for denne algoritme $\mathcal{O}(nlgn)$.

Der er $k$ krydsende linjesegmenter – $k = \mathcal{O}(n^2)$, da der er $\binom{n}{2}$ par af linjesegmenter. Hvis algoritmen skulle returnerer alle linjesegmenterne, ville tiden være $\mathcal{O}((n+k)lgn)$ da vi for hver omgang af loopet, som er $O(n)$, skal tjekke $O(k)$ krydsende linjesegmenter, hvis vi finder et kryds. Hvis vi ikke tjekkede for alle kryds, ville algoritmen kun returnerer det første andet linjesegment som $s$ krydsede med, og ikke dem alle. Hvis der viser sig at være $k = n^2$ kryds, er køretiden $\mathcal{O}((n+n^2)lgn) = \mathcal{O}(n^2lgn)$.

# 4 Dynamisk Programmering

---

- Elementer af dynamisk programmering
- Rod-Cutting
  - Køretid
  - Optimal delstruktur
  - Overlappende delproblemer
  - Top-down
  - Bottom-up
- Longest Common Subsequence
  - Optimal delstruktur
  - Overlappende delproblemer
  - dynamisk programmering algoritmen
  - Recap af de 4 trin

## 4.1 Elementer af dynamisk programmering

Dynamisk programmering, ligesom del og hersk, løser problemer ved at dele dem op og løse underproblemerne. En forskel er dog, at problemerne skal udvise to egenskaber, for at dynamisk programmering er praktisk at bruge: Problemet skal udvise **optimal delstruktur**, hvilket betyder at løsningen til et problem indeholder løsningen til delproblemer. Den anden egenskab et dynamisk programmering problem skal udvise er **overlappende delproblemer**, hvilket betyder at det samme delproblem skal løses flere gange for at nå frem til en løsning til det endelige problem.

Der er 4 trin man følger for et dynamisk programmering problem:

- Karakteriser en optimal løsnings struktur.
- Rekursivt definer værdien af en optimal løsning.
- Bestem værdien af en optimal løsning.
- Konstruer selve den optimale løsning ud fra de løste værdier.

De første 3 trin giver værdien til en optimal løsning, mens det sidste trin er valgfrit, og giver løsningen der hører til værdien.

## 4.2 Rod-Cutting

Man får givet en stav af længde $n$ og en liste $p_i$ med priser, således at en stav af længde $k$ har prisen $p_k$. Med en stav af længde $n$ er der $2^{n-1}$ måder at skære staven på - ved hver længde kan man enten vælge at skære, eller ikke at skære. En måde at beskrive prisen man kan få for en stav er, ved at skære staven i to dele og rekursivt checke den optimale pris, og så vælge maximum af de priser

$$r_n = max(p_n, r_1 + r_{n-1}, r_2 + rn - 2, ..., r_{n-1} + r_1)$$

En lettere måde at beskrive den rekursive løsning er, at ved at vælge den maksimale pris ved at sælge en stav af længde $i$ plus den optimale pris ved en stav af $n - i$

$$r_n = max_{1 \leq i \leq n}(p_i + r_{n-i})$$

### 4.2.1 Køretid

Køretiden af den rekursive rod-cutting algoritme er $\mathcal{O}(2^n)$, da den tjekker alle $2^{n-1}$ løsninger. Mere formelt kan vi se på køretiden $T(n)$ for algoritmen, hvor vi beskriver køretiden som en rekursionsligning $T(n) = 1 + \sum_{i=0}^{n-1} T(i)$, hvor 1 er kaldet for roden, det oprindelige kald til $T(n)$, og det næste led summerer over alle de rekursive kald der bliver foretaget.

#### Bevis af køretid

Vi beviser køretiden ved substitutionsmetoden, hvor base case er $T(0) = 1 + \sum_{i=0}^{0-1} T(i) = 1 = 2^0$. Vi går nu ud fra at $T(m) = 2^m$, for $m < n$.

***Bevis.***

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

Køretiden for rod-cutting er $\mathcal{O}(2^n)$. ■

Denne køretid er ikke praktisk for store længder $n$, men heldigvis kan den forbedres ved brug af dynamisk programmering.

### 4.2.2 Optimal delstruktur

En egenskab der skal til for at kunne benytte dynamisk programmering til at løse et problem er, at problemet skal udvise optimal delstruktur. Det betyder, at løsningen til et problem indeholder løsningen til underproblemer. I tilfældet med rod-cutting algoritmen, ses det i, at når vi skal finde den optimale pris for en stav af længde $n$, skal vi finde den optimale pris for stavlængder op til $n-1$.

### 4.2.3 Overlappende underproblemer

En anden egenskab der skal gøre sig gældende for, at dynamisk programmering er praktisk at bruge er, at der skal udvises overlappende underproblemer.

Overlappende underproblemer er, når det samme underproblem skal løses flere gange. I rod-cutting ses det i, at den optimale pris for at finde den optimale pris af en stavlængde $n$, skal man finde den optimale pris for stavlængden $i < n$ $2^{n-i}$ gange.

### 4.2.4 Top-down

Man kan forbedre $\mathcal{O}(2^n)$ ved at bruge dynamisk programmering. I top-down bruger man et array eller en anden datastruktur til at gemme løsningen til underproblemer, så næste gang et rekursivt kald støder på det samme underproblem skal det ikke løses igen, men kan læses fra datastrukturen på konstant tid.

Nedenunder ses algoritmerne til memoized cut-rod. `MEMOIZED-CUT-ROD` initialiserer et array $r$, som funktionen `MEMOIZED-CUT-ROD-AUX` bruger til at gemme løsninger til delproblemer.

**Algorithm 4** Top-down funktion bruger tabel til memoization.

1: $p[1...n]$ er array med stavpriser for stav af længde $i = 1, 2, ..., n$
2: **function** MEMOIZED-CUT-ROD($p, n$)
3: **for** $i = 0$ **to** $n$ **do**
4: $r[i] = -\infty$
5: **return** MEMOIZED-CUT-ROD-AUX($p, n, r$)

**Algorithm 5** Selve cut-rod funktionen til memoization.

```
1: p[1...n] er array med stavpriser for stav af længde i = 1, 2, ..., n
2: r[1...n] er array med optimal løsning på delproblemer af cut-rod
3: function MEMOIZED-CUT-ROD-AUX(p, n, r)
4:     if r[n] ≥ 0 then
5:         return r[n]
6:     if n == 0 then
7:         q = 0
8:     else
9:         q = −∞
10:        for i = 1 to n do
11:            q = max(q, p[i]+MEMOIZED-CUT-ROD-AUX(p, n − i, r))
12:    r[n] = q
13:    return q
```

### 4.2.5 Bottom-up

I bottom-up metoden starter man med at løse, som navnet hentyder, fra bunden og op. Et problem bliver ikke løst, før delproblemer, som det problem afhænger af, er blevet løst. Her er der ikke brug for en datastruktur til at gemme på løsningerne til delproblemer, da løsningen bruges umiddelbart efter den er fundet.

**Algorithm 6** Selve cut-rod funktionen til memoization.

```
1: p[1...n] er array med stavpriser for stav af længde i = 1, 2, ..., n
2: function BOTTOM-UP-CUT-ROD(p, n,)
3:     r[1...n] =nyt array
4:     r[0] = 0
5:     for j = 1 to n do
6:         q = −∞
7:         for i = 1 to j do
8:             q = max(q, p[i] + r[j − i]
9:         r[j] = q
10:    return r[n]
```

Både top-down og bottom-up giver en ny køretid på $\Theta(n^2)$). I bottom-up er det åbenlyst, da vi iterer over to for-loops. I top-down skyldes det at vi har et for-loop, og hver løsning bliver løst 1 gang.

## 4.3 Longest common subsequence

En delfølge af en følge $X$ består af elementerne fra $X$ hvor nogle er blevet udeladt. F.eks. $X = \langle 1, 2, 3, 4, 5 \rangle$ kunne delfølger være $X' = \langle 1, 2 \rangle, \langle 1, 4, 5 \rangle$ etc.

Longest common subsequence algoritmen finder den længste delfølge som er en delfølge af to følger $X$ og $Y$.
En naiv måde at løse LCS problemet er, at sammenligne alle delfølger af $X$ med alle delfølger af $Y$. Siden en delfølge kan tage den binære beslutning om at inddrage eller udelade et element af den oprindelige følge, er der $2^n$ delfølger, derfor giver det sig at denne metode er upraktisk.

### 4.3.1 Optimal delstruktur

For at vi kan bruge dynamisk programmering til at forbedre LCS, skal vi vise at LCS udviser optimal delstruktur, ved at en LCS af to følger indeholder LCS til "prefix"af de følger, hvor en prefix $i$ af en følge $X$ er de første $i$ elementer af $X$, således at $X_i = \langle x_1, x_2, ..., x_i \rangle$.

#### Bevis for at optimal delstruktur af LCS

**Theorem 4.1.** *Hvis vi har to følger $X = \langle x_1, x_2, ..., x_m \rangle$ og $Y = \langle y_1, y_2, ..., y_n \rangle$, og vi har en LCS af de to følger $Z = \langle z_1, z_2, ..., z_k \rangle$.*

1. *hvis $x_m = y_n$ så $z_k = x_m = y_n \Rightarrow Z_{k-1}$ er en LCS af $X_{m-1}$ og $Y_{n-1}$*
2. *hvis $x_m \neq y_n$ så $z_k \neq x_m \Rightarrow Z$ er en LCS af $X_{m-1}$ og $Y$*
3. *hvis $x_m \neq y_n$ så $z_k \neq y_n \Rightarrow Z$ er en LCS af $X$ og $Y_{n-1}$*

Altså hvis $Z$ er en LCS af $X$ og $Y$, så må $Z_{k-1}$ være en optimal delfølge for et prefix af $X$ og $Y$.

Tilfælde 2 og 3 er symmetriske, hvor $X$ og $Y$ er byttet rundt.

***Bevis.*** Vi starter med 1. tilfælde:
Hvis $Z$ er en LCS, hvor $x_m = y_n$ og vi antager at $z_k \neq x_m$, så kan vi blot tilføje $x_m = y_m$ til $Z$, hvor vi så får en LCS af længde $k+1$, hvilket er i modstrid med at $Z$ allerede var en LCS.
Nu har vi at $Z_{k-1}$ er en LCS af $X_{m-1}$ og $Y_{n-1}$. Lad os antage at der findes en fælles delfølge $W$ af $X_{m-1}$ og $Y_{n-1}$ som har en længde større end $k-1$. Siden $x_m = y_n$ kan vi tilføje $x_m = y_n$ til $W$, som så får en længde $k+1$, hvilket er i modstrid med antagelsen at $Z$ var en LCS.
Altså, hvis $x_m = y_n$, så er $Z_{k-1}$ en LCS af $X_{m-1}$ og $Y_{n-1}$.

2. tilfælde:
Hvis $x_m \neq y_m$ og vi antager at $z_k \neq x_m$, så må $Z$ være en LCS af $X_{m-1}$ og $Y$. Hvis vi antager at der findes en fælles delfølge $W$ af $X_{m-1}$ og $Y$, som har længde større end $k$, så ville $W$ også være en fælles delfølge af $X_m$ og $Y$, siden vi fra tilfælde 1 ved at vi ikke kan tilføje $X_m = y_n$. Dette er i modstrid med at $Z$ er en LCS af $X$ og $Y$, siden $W$ har længde større end $k$
Altså, hvis halerne i $X$ og $Y$ ikke er ens, men $z_k = y_n$, så fjerner vi $i$ elementer halen af $X$, indtil vi finder et fælles element. $Z_{k-1}$ er så en delfølge af $X_{m-i}$ og $Y_{n-1}$

3. tilfælde er symmetrisk med 2. tilfælde, med $X$ og $Y$ byttet rundt. ■

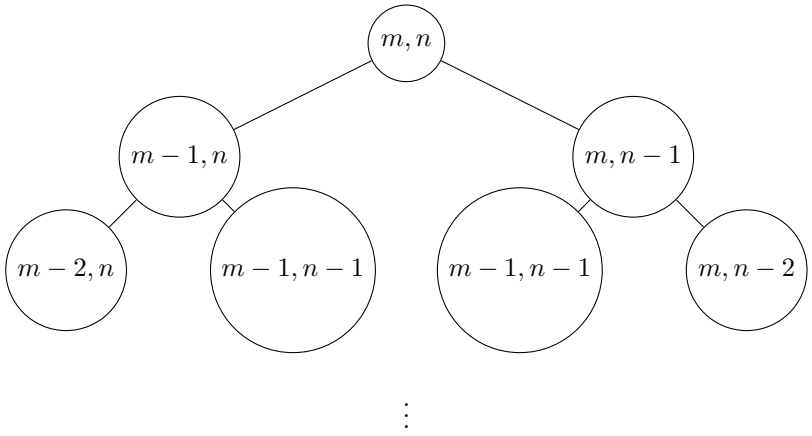
### 4.3.2 Overlappende delproblemer

LCS algoritmen udviser også overlappende delproblemer. Vi viste før, at hvis $x_m \neq y_n$, så skal vi løse to delproblemer; at finde en LCS til $X_{m-1}$ og $Y$, og finde en LCS til $X$ og $Y_{n-1}$.

$$c[i,j] = \begin{cases} 0 & \text{hvis } i = 0 \text{ or } j = 0 \\ c[i-1, j-1] & \text{hvis } i, j > 0 \text{ og } x_i = y_j \\ max(c[i-1,j], c[i,j-1]) & \text{hvis } i, j > 0 \text{ og } x_i \neq y_j \end{cases} \quad (1)$$

Hver af disse to delproblemer skal så løse tilsvarende delproblemer. Et træ for de overlappende delproblemer kan ses her:

Figur 3: Overlappende delproblemer. Knuderne viser prefix $m$ for $X$, og $n$ for $Y$.




### 4.3.3 Dynamisk programmering algoritmen

Hvis vi bruger dynamisk programmering til at gemme løsninger til delproblemerne kan vi se, at vi skal løse $\Theta(mn)$ unikke delproblemer, hvilket også giver køretiden $\mathcal{O}(mn)$. Løsningen til LCS problemet kan udregnes ved ligning (1), hvor $c$ arrayet indeholder alle delproblemerne. For at konstruere selve følgen, skal vi også gemme værdien i hver af de to følger $X$ og $Y$ i et array, når $x_i = y_j$.

### 4.3.4 Recap af de 4 trin

De fire trin der hører til at løse LCS problemet med dynamisk programmering er:

- Vi bestemte hvordan strukturen for en LCS var, ved at vise optimal delstruktur.
- Derefter bestemte vi rekursivt hvordan vi skulle finde den optimale løsning, og vi så at problemet udviste overlappende delproblemer.
- Vi bruger ligning (1) til at beskrive en algoritme der returnerer et 2-dimensionelt array $c$ hvor $c[m, n]$ indeholder længden af LCS'en til $X_m$ og

$Y_n$

- Valgfrit kan algoritmen også returnere et 2-dimensionelt array $b$ der viser hvordan vi kan konstruere et LCS.

# 5 Grådige Algoritmer

---

- Elementer af en grådig algoritme
- Huffmann code
  - Bygge en Huffman code
  - Bevis af greedy choice
  - Bevis af optimal delstruktur

## 5.1 Elementerne af en grådig algoritme

Grådige algoritmer er algoritmer der læser et problem ved altid at tage den beslutning der, lokalt, er den bedste beslutning. Grådige algoritmer, ligesom dynamisk programmering, løser problemer der udviser **optimal delstruktur**, men i modsætning til dynamisk programmering, der sørger for at tage en global optimal løsning, ved at bruge delløsninger, så tager den grådige algoritme det valg der fører til en lokal optimal løsning, hvilket i mange tilfælde også fører til den globalt optimale løsning.
Vi kalder den egenskab hvor en lokal optimal beslutning fører til en global optimal løsning, for **greedy choice** egenskaben.

## 5.2 Huffman code

Huffman code er en metode til at komprimere data på. I vores tilfælde vil vi kigge på at komprimere en streng af character tegn.
Hvert tegn har en bit streng der repræsenterer det tegn, f.eks. ASCII tegn er 8-bit **fixed-length code**, men det vil også sige, at hvis har en fil med 1000 tegn, så skal vi minimum bruge 8000 bits for at alle tegn med. Hvis man kun har begrænset antal tegn, kan man benytte **variable-length code**. Her giver vi tegn der optræder oftest en kortere bit længde end tegn der optræder få gange, vi sørger også for at den code vi bruger er en **prefix (free) code**, betydende at intet tegns codeword svarer til prefix af et andet tegns codeword e.g. 110 er prefix af 1101, derfor kan begge de to codewords ikke bruges.

### 5.2.1 Bygge en Huffman code

Algoritmen til at lave Huffman code ud fra et alfabet af tegn, går ud på at konstruere et binært træ, hvor, når man vandrer igennem træet fra roden af, så når man tager ned til venstre barn, så tilføjer man et 0, og når man tager ned mod højre barn tilføjer man 1.

**Algorithm 7** $N(C, i)$ with memoization

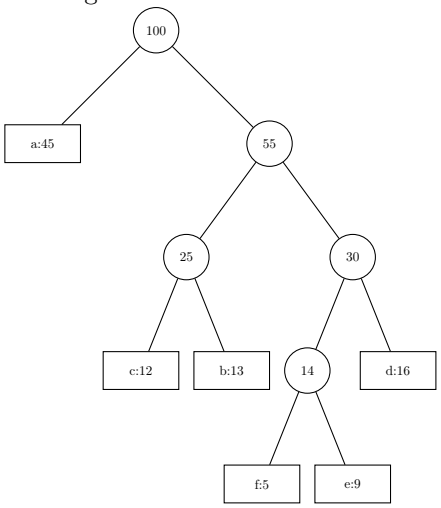
```
1:  function HUFFMAN(C)
2:      n = |C|
3:      Q = C                                   ▷ min-prioritetskø
4:      for i = 1 to n − 1 do
5:          lav en ny knude z
6:          z.left = EXTRACT-MIN(Q)
7:          z.right = EXTRACT-MIN(Q)
8:          z.freq = z.left.freq + z.right.freq
9:          INSERT(Q, z)
10:     return EXTRACT-MIN(Q)
```

Algoritmen laver en prioritetskø $Q$, så tegnet med den laveste frekvens ligger først, derefter laver den en knude der forbinder de to elementer i køen med laveste frekvens og bygger træet op af.

Nedenunder er et eksempel på et færdigt Huffman træ med et alfabet $C$, hvor hvert tegn har frekvens $c.freq$.

$$C = \begin{array}{c|c|c|c|c|c} \text{a} & \text{b} & \text{c} & \text{d} & \text{e} & \text{f} \\ \hline 45 & 13 & 12 & 16 & 9 & 5 \end{array}$$

Figur 4: Huffman træ for $C$



Med fixed-length vil filstørrelsen (hvor mange bits vi skal bruge) være 8 gange så stor som antallet af tegn, hvis vi bruger ASCII. For at udregne hvor mange bits vi skal bruge med den nye prefix-code, skal vi summere produktet af tegnenes frekvens og deres dybde i træet, $d_T(c)$, som er antallet af bits i tegnets nye code. Den cost-funktion kalder $B(T)$.

$$B(T) = \sum_{c \in C} c.freq \cdot d_T(c)$$

For at bevise korrektheden af algoritmen, skal vi vise at den har greedy choice egenskaben og optimal delstruktur.

### 5.2.2 Bevis af greedy choice

Det grådige valg algoritmen tager er altid at vælge de to tegn med lavest frekvens til at være børn af en ny knude. Altså at $x, y$ har samme længde og kun skiller sig fra hinanden i den sidste bit.

**Lemma 5.1.** *Lad $x, y$ være to tegn i et alfabet $C$, med laveste frekvens. Der findes en optimal prefix-code, således at koden for $x$ og $y$ har samme længde, og skiller sig kun ud i den sidste bit.*

Altså, at træet $T$ med der indeholder det grådige valg, er optimal.

***Bevis.*** Vi starter med at have et træ $T$ som ikke indeholder det grådige valg, hvor $x$ og $y$ ikke er naboer. Vi ændrer så i træet indtil $x$ og $y$ er naboer, så de får samme længde og kun skiller sig ud fra hinanden i den sidste bit.
Lad $a, b$ være to knuder med maksimal dybde. Vi antager følgende:

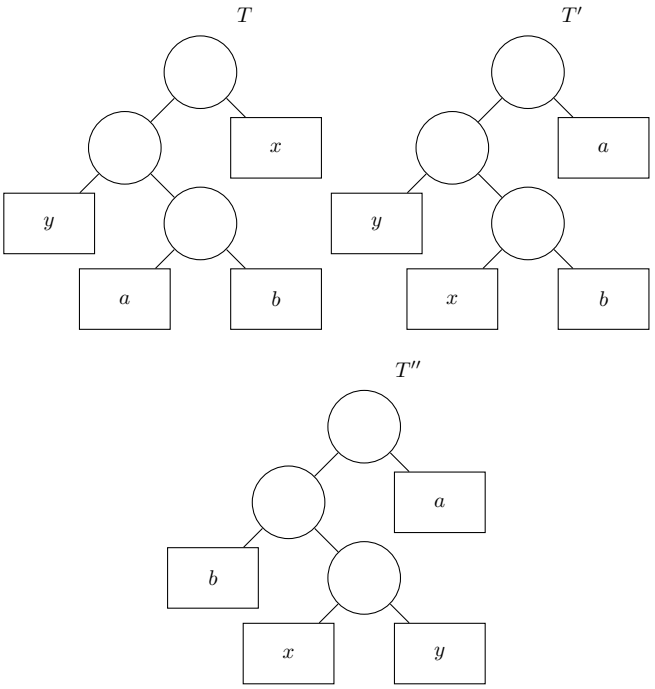
- $x.freq \leq y.freq$
- $a.freq \leq b.freq$
- $x.freq \leq a.freq$
- $y.freq \leq b.freq$

- $x.freq < b.freq$

Hvis $x.freq = b.freq$ så vil alle frekvenserne være ens, og ingen ændringer i træet vil kunne ændre i dens cost.

Første trin i beviset er at tage træet $T$ og bytte plads med $a$ og $x$, så vi får et nyt træ $T'$. Derefter tager $T'$ og bytter plads med $b$ og $y$ så vi får $T''$.

Figur 5: Trin i rykning af træ $T \to T' \to T''$




For at se om træet $T'$ stadig er optimalt, så skal $B(T') - B(T) \leq 0$.

$$\begin{aligned} B(T) - B(T') &= \sum_{c \in C} c.freq \cdot d_T(c) - \sum_{c \in C} c.freq \cdot d_{T'}(c) \\ &= x.freq \cdot d_T(x) + a.freq \cdot d_T(a) - x.freq \cdot d_{T'}(x) + a.freq \cdot d_{T'}(a) \\ &= x.freq \cdot d_T(x) + a.freq \cdot d_T(a) - x.freq \cdot d_T(a) + a.freq \cdot d_T(x) \\ &= (a.freq - x.freq)(d_T(a) - d_T(x)) \end{aligned}$$

Siden $x.freq \leq a.freq$ må første led være ikke negativt, og da vi ved at $a$ havde maksimal dybde i træet, må $d_T(a) - d_T(x)$ også være ikke negativt, det bety-

der at $0 \leq B(T) - B(T')$ og $T'$ er derfor stadig optimal. Samme argument kan bruges i $B(T') - B(T'')$, da vi nu rykker på $y$ og $b$.

Ergo er træet $T''$, der indeholder det grådige valg, hvor $x, y$ er naboer, optimalt.

■

### 5.2.3 Bevis af optimal delstruktur

**Lemma 5.2.** *Lad $x, y \in C$ være tegn med minimal frekvens. Lad $C'$ være et nyt alfabet hvor $x, y$ er fjernet og erstattet af $z$, således at $C' = C - \{x, y\} \cup \{z\}$, og hvor $z.freq = x.freq + y.freq$.*
*Lad $T$ være et træ for en optimal prefix-code for $C$, så vil $T'$, der er det træ hvor $x, y$ er fjernet, være en optimal prefix code for $C'$.*

Altså, hvis $T$ er et optimalt træ for $C$, så må $T'$, der er et deltræ af $T$, også være optimalt for $C'$, som er et delalfabet af $C$. Ergo udvises der optimal delstruktur.

***Bevis.*** Træet $T'$ har cost $B(T') = B(T) - x.freq - y.freq$, siden det er det samme træ som $T$, bare hvor værdien $z.freq = x.freq + y.freq$ ligger et lag højere, så $d_T(z) = y.freq - 1 = x.freq - 1$.

Vi vil nu bevise at deltræet $T'$ kun er optimalt, hvis $T$ også er optimal. Vi vil bevise det med en modstrid. Vi antager at $T'$ stadig er det optimale træ for $C'$ og $T$ ikke er det optimale træ for $C$, men at der findes et træ $T''$ som er optimal for $C$, så der gælder $B(T'') < B(T)$.

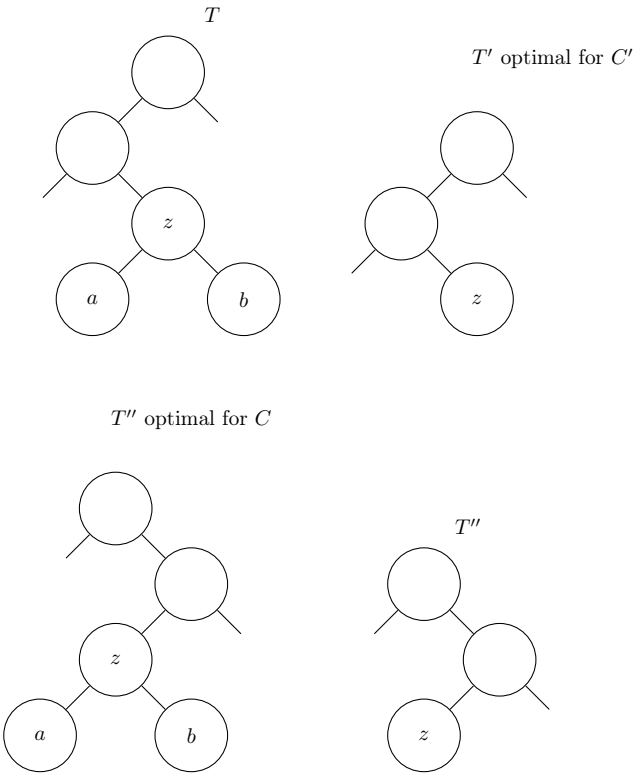
Vi laver nu et træ $T'''$ ud fra $T''$, som er et træ for $C'$, altså hvor vi fjerner $x, y$. Hvis vi nu regner på prisen af vores deltræer, har vi

$$\begin{aligned} B(T''') &= B(T'') - x.freq - y.freq \\ &< B(T) - x.freq - y.freq \\ &= B(T') \end{aligned}$$

$B(T''') < B(T')$ er en modstrid, da vi antog at $T'$ var det optimale træ for $C'$. Derfor må det optimale træ for $C'$ være et deltræ af det optimale træ for $C$, og derfor være en optimal delstruktur. ■

***Bevis.*** Vi har nu bevist korrektheden af Huffman codes ved at bevise at den udviser de to egenskaber: Greedy choice property og optimal delstruktur. ■

Figur 6: Optimal delstruktur: Billede modstriden, at $T$ ikke er optimal for $C$ når $T'$ er optimal for $C'$



# 6 Minimalt Udspændte Træer

---

- Udspændte træer
- Kruskals algoritme
  - Disjunkte mængder
  - Køretid af Kruskals algoritme
- Prims algoritme
  - Prioritetskø

## 6.1 Udspændte træer

givet en graf $G = (V, E)$, så er $T \subset G.E$ et udspændt træ, således at $w(T) = \sum_{e \in T} w(e)$, hvor $w(u, v)$ er en vægt-funktion der bestemmer vægten af en kant der går mellem knuderne $u, v$. Et minimalt udspændt træ er, som navnet hentyder, et udspændt træ for $G$ med den laveste samlede vægt.

Vi bruger en grådig algoritme til at bygge et MST, hvor det grådige valg er at tilføje en kant $e \in G.E$ og tilføje den til en mængde $A$. For at tilføje kanter til $A$ holder vi øje med følgende invariant:

Før vi tilføjer en ny knude, udgør $A$ en delmængde af et MST.

Selve algoritmen går ud på at finde en **safe edge**, som er en kant $e$ der overholder invarianten og så tilføje den til $A$, og dette gør man iterativt, indtil $A$ udgør et udspændt træ for $G$.

- **Initiering**: Den tomme mængde er en delmængde af et MST.
- **Vedligeholdelse**: Vi tilføjer kun safe-edges, og overholder derfor invarianten.
- **Afslutning**: Siden alle kanterne vi har tilføjet har overholdt invarianten, må alle kanter i $A$ være i et MST, derfor må $A$ være et MST.

For at finde en safe-edge skal vi finde en **light-edge**, som er en minimumsvægtet kant der forbinder et **cut** af $G$. Et cut er en opdeling af $G.V$, således at $G$ bliver splittet i to nye disjunkte grafer indeholdende knuderne $S$ og $V - S$.

**Theorem 6.1.** *Lad $G = (V, E)$ være en forbundet graf med en vægtsfunktion $w(u, v)$ i $G.E$. Lad $A$ være en delmængde af $G.E$, som er inkluderet i et MST for $G$.*
*Lad $(S, V - S)$ være et cut af $G$ der respekterer $A$, og lad $(u, v)$ være en light-edge som forbinder det cut, så vil $(u, v)$ være en safe-edge for $A$.*

***Bevis.*** Vi beviser det med modstrid. Lad $T$ være et MST der indeholder $A$, og som ikke indeholder en light-edge $(u, v)$. Vi bygger nu et nyt MST $T'$, som indeholder $A \cup (u, v)$, og derved vise at $(u, v)$ er en safe-edge.

$(u, v)$ forbinder cuttet $(S, V - S)$, og siden $G$ er forbundet, må der være en kant $(x, y)$ fra $u \rightsquigarrow v$ der også forbinder cuttet $(S, V - S)$. $(x, y)$ er ikke i $A$. Vi laver nu et nyt træ, vi bruger $(u, v)$ til at forbinde cuttet i stedet for $(x, y)$:

$$T' = T - \{(x, y)\} \cup \{(u, v)\}$$

Siden $(u, v)$ er en light-edge for cuttet, så må $w(u, v) \leq w(x, y)$, hvilket betyder

$$\begin{aligned} w(T') &= w(T) - w(x, y) + w(u, v) \\ &\leq w(T) \end{aligned}$$

Derfor må $T'$ også være et MST for $G$. Det sidste vi mangler at vise er, at $(u, v)$ er en safe-edge for $A$, hvilket kan ses ved at $A \cup \{(u, v)\} \subset T'$. ■

## 6.2 Kruskals algoritme

Den første algoritme til at finde MST er Kruskals algoritme. Denne algoritme bruger en **skov** af træer, som er alle knuderne $G.V$. Derefter forbinder den de knuder som udgør en safe-edge.

### 6.2.1 Disjunkte mængder

Kruskals algoritme bruger disjunkte mængder til at holde kanterne i MST.

Disjunkte mængder, er en datastruktur der understøtter følgende operationer og egenskaber.

- Hver mængde har et kanonisk element, kaldet mængdens repræsentant.
- `Make-Set(`$x$`)` laver en ny mængde med $x$ som repræsentant.
- `Find-Set(`$x$`)` returnerer repræsentanten for mængden der indeholder elementet $x$.

- `Union`$(x, y)$ Forbinder de to mængder med repræsentanter $x$ og $y$, således at den nye mængdes repræsentant er en af de to.

Der er to måder vi kan implementere disjunkte mængder på: Enten kan vi bruge en **linked-list**repræsentation, hvor hver mængde er en linked-list, med et mængde objekt, der peger på repræsentanten og hvor alle elementer peger på. Den anden metode, som vi bruger til Kruskal, er **Disjunkt-mængde skove**, som er en skov af disjunkte træer, hvor roden er repræsentanten af mængden.

De tre operationer kan implementeres på følgende måde:

- `Make-Set`$(x)$: Vi laver et nyt træ, kun indeholdende $x$, og vi sætter $x.\pi$ til at være sig selv, derved er den roden. Dette tager $\mathcal{O}(1)$ tid.
- `Find-Set`$(x)$: Vi følger $x = x.\pi$ op indtil $x = x.\pi$, hvorefter vi returnerer repræsentanten. Dette tager $\mathcal{O}(h)$ tid.
- `Union`$(x, y)$: Vi indsætter roden af det ene træ ind i det andet træ. Dette kan i værste fald føre til en hægtet-liste af elementer, derfor indføre vi to heuristikker: **Union by rank** og **path compression**. I union by rank holder hver knude et estimat af højden af dets deltræ, dets **rank**, og derfor kan vi forbinde det mindre træ på det større træ. Path compression indføres i `Find-Set`$(x)$, hvor, så længe $x \neq x.\pi$ sætter vi $x.\pi =$ `Find-Set`$(x)$, dette minimerer højden af træet, og gør forældrestigen så kort som muligt.

### 6.2.2 Køretid af Kruskals algoritme

Vi bruger **disjoint-set forets** til at udvikle MST med Kruskals.
Første trin er at oprette en skov af mængder indeholdene alle knuderne i $G.V$, derefter sorteres alle kanterne efter stigende vægt. Derefter looper vi over alle kanterne $(u, v)$, og så længe `Find-Set`$(u) \neq$ `Find-Set`$(v)$ så forbinder vi mængderne med `Union`$(u, v)$. Til sidst returnerer vi den disjunkte mængde, som, hvis $G$ er forbundet, er et MST.

Køretiden af Kruskals algoritme er $\mathcal{O}(ElgV)$. Først laver vi $|V|$ disjunkte træer, som er $\mathcal{O}(V)$, herefter looper vi over alle kanterne, og forbinder kantens knuder. Siden vi

## 6.3 Prims algoritme

Prims algoritme fungerer meget som Dijkstras korteste vej algoritme. Her har vi en prioritetskø af $v.key$, som er en værdi der er den mindste vægt af en kant der forbinder til $v$. Vi initialiserer alle $v.key = \infty$ og $v.\pi$, derefter vælges en arbitrær knude $r$ til roden, og $r.key = 0$. Derefter lægges alle knuder ind i prioritetskøen $Q = G.V$. Vi udvælger knuden $u$ med den mindste $key$ i $Q$, som starter med at være $r$ og tjekker alle dens naboer $v$, og retter $v.key = w(u, v)$ hvis $w(u, v) < v.key$, og sætter $v.\pi = u$. Når køen er tom, betyder det at alle knuderne er i træet, og vi kan eventuelt returnerer rod-knuden.

Algoritmen overholder følgende invarianter:

1. $A = \{(v, v.\pi) : v \in G.V - \{r\} - Q\}$
2. Alle knuder i MST er knuder i $G.V - Q$
3. For alle knuder $v \in G.V$, hvis $v.key < \infty$, så er $v.\pi \neq$ `NIL` og $(v, v.\pi)$ er en light-edge der forbinder $v$ til en knude der allerede er i MST.

Køretiden af Prims algoritme er $\mathcal{O}(E + VlgV)$ hvis vi implementerer vores prioritetskø med en Fibonacci hob, da `EXTRACT-MIN` tager $\mathcal{O}(lgV)$, som udføres hvert loop $\mathcal{O}(V)$ gange, og `DECREASE-KEY` operation tager $\mathcal{O}(1)$ tid og udføres $\mathcal{O}(E)$ gange. Dette giver $\mathcal{O}(E + VlgV)$.

# 7 Korteste Vej

---

- Korteste vej problemet
  - Optimal delstruktur
  - Cykler i grafen
  - Egenskaber af en korteste vej
- Bellman-Ford
  - Køretid
  - Korrekthed
- Dijkstra
  - Korrekthed
  - Køretid

## 7.1 Korteste vej problemet

Givet en **orienteret graf** $G = (V, E)$ med en vægtsfunktion$w(u, v)$ for $(u, v) \in G.E$, altså $w : E \to \mathbb{R}$, så går korteste vej problemet ud på at finde en vej $p$, som er en følge af kanter i $G.E$, med en minimal vægt $w(p)$, hvore vægtsfunktionen på en vej, er summen af vægtene i dens kanter.

Vi definerer vægten af den korteste vej fra $u$ til $v$ til at være

$$\delta(uv) = \begin{cases} minw(p) & \text{hvis der findes en vej } p \text{ fra } u \text{ til } v \\ \infty & \text{ellers} \end{cases}$$

Der er 4 varianter af korteste vej problemet:

- **Single-pair** bestemmer den korteste afstand fra en knude $u$ til en knude $v$.
- **Single-source** bestemmer den korteste afstand fra en knude $s$ til alle andre knuder i $G.E$.
- **Single-target** bestemmer den korteste afstand fra alle knuder i $G.E$ til en knude $t$.
- **All-pairs** bestemmer den korteste afstand fra alle knuder i $G.E$ til alle andre knuder i $G.E$.

De to algortimer vi kommer til at kigge på er single-source korteste vej.

### 7.1.1 Optimal delstruktur

Problemet at finde den korteste vej udviser optimal delstruktur.

**Lemma 7.1.** *Givet en orierenteret graf $G = (V, E)$ med vægtsfunktion $w(u, v)$. Lad $p = \langle v_0, v_1, ..., v_k \rangle$ være den korteste vej $v_0 \rightsquigarrow v_k$. For alle $i, j$ således at $0 \leq i \leq j \leq k$, lad $p_{ij} = \langle v_i, v_{i+1}, ..., v_j \rangle$ være en delvej $v_i \rightsquigarrow v_j$, så er $p_{ij}$ den korteste vej fra $v_i$ til $v_j$.*

Altså, alle delveje af en korteste vej, er også selv korteste veje.

***Bevis.*** Hvis vi kigger på delvejene $v_0 \rightsquigarrow v_i \rightsquigarrow v_j \rightsquigarrow v_k$, så har vi at

$$w(p) = w(p_{0i}) + w(p_{ij} + w(p_{jk})$$

Antag så, at der findes en anden vej $p'_{ij}$ således at $w(p'_{ij}) < w(p_{ij})$, så kan vi indsætte $p'_{ij}$ i $p$, og få

$$w(p') = w(p_{oi}) + w(p'_{ij}) + w(p_{jk}) < w(p)$$

Hvilket er i modstrid med at $p$ var en korteste vej. ■

### 7.1.2 Cykler i grafen

Hvis vi tillader at kanter kan have en negativ vægt, så kan vi løbe ind i problemer med negative cykler i grafen.
Hvis grafen $G$ indeholder en cykel, som kan nås fra vores startknude $s$, hvor summen af vægtene i cyklen er negativ, så er korteste vej problemet udefinerbart. Siden cyklen har negativ sum, kan vi bare kører flere omgange om cyklen, og få en vej, som har $\delta(s,t) = -\infty$.

Det giver heller ikke mening at inkluderer positive cykler i den korteste vej. Hvis en vej $p = s \rightsquigarrow t$ indeholder en positiv cykel, så må der være en knude $u$ i cyklen, hvor $s \rightsquigarrow u$ og $u \rightsquigarrow t$. Da cyklen starter og slutter i $u$, kan vi lave en vej $p'$ hvor vi har fjernet alle knuderne i cyklen, undtagen $u$ fra $p$, og have $w(p') < w(p)$.

En korteste vej indeholder derfor ingen cykler.

### 7.1.3 Egenskaber af en korteste vej

Korteste vej algoritmerne bruger en teknik **relaxation**. Hver knude $v$ har et felt $v.d$ som er en øvre grænse for den korteste vej $s \rightsquigarrow v$. Derudover har de også et felt $v.\pi$, som er pegeren til den foregående knude i den korteste vej, således at vi kan konstruere selve den korteste vej.

Alle knuder bliver initialiseret til $v.d = \infty$ og $v.\pi = \texttt{NIL}$, udover $s.d = 0$. Selve relax funktionene tager to knuder $u$ og $v$, og ser om $v.d > u.d + w(u,v)$, hvis

den er, bliver $v.d = u.d + w(u,v)$ og $v.\pi = u$. Altså, hvis $p_{su} + w(u,v) < p_{sv}$, så kan vi sige, at den korteste vej fra $s$ til $v$ går igennem $u$ – $s \rightsquigarrow u \rightsquigarrow v$.

Vi bruger følgende egenskaber af en korteste vej:

- **Trekants uligheden**

  **Lemma 7.2.** *Lad $G = (V, E))$ være en orienteret graf med vægtsfunktion $w(u,v)$ på en kant $(u,v) \in G.E$. Lad $s$ være startknuden. Der gælder for alle kanter $(u,v)$*

  $$\delta(s,v) \leq \delta(s,u) + w(u,v)$$

  .

  ***Bevis.*** Antag at $p$ er den korteste vej fra $s$ til $v$, så kan der ikke findes en vej med lavere vægt fra $s$ til $v$, så derfor, hvis kanten $(u,v)$ er i $p$, så må $p$ gå igennem $u$, altså $p_{sv} = p_{su} + (u,v)$ ■

- **En øvre grænse egenskaben**

  **Lemma 7.3.** *Lad $G = (V, E))$ være en orienteret graf med vægtsfunktion $w(u,v)$ på en kant $(u,v) \in G.E$. Lad $s$ være startknuden. Lad $s$ være startknuden. grafen bliver intitialiseret således at alle knuder $v \neq s \in G.V$ har $v.d = \infty$. Der gælder at $v.d \geq \delta(s,v)$, som invariant over alle relaxation operationer. Når $v.d$ får sin nedre grænse $\delta(s,v)$ forbliver den uændret.*

  ***Bevis.*** Vi beviser invarianten $v.d \geq \delta(s,v)$ med induktion over antallet af relaxations.
  Base-case er sandt, siden $v.d = \infty \Rightarrow v.d \geq \delta(s,v)$.
  For det induktive trin kalder vi relax på en kant $(u,v)$, hvor vi antager at $v.d > \delta(s,u) + w(u,v)$, da relax ellers ikke ændrer noget og er derfor sandt.

  $$\begin{aligned} v.d &= u.d + w(u,v) \\ &\geq \delta(s,u) + w(u,v) && \text{(Det induktive trin)} \\ &\geq \delta(s,v) && \text{(Trekants uligheden)} \end{aligned}$$

Grunden til $v.d$ ikke ændrer sig når den når sin nedre grænse er, at når $v.d = \delta(s, v)$, kan den ikke blive lavere, da $\delta(s, v)$ er den korteste vægtede vej. Den kan heller ikke stige, da relax kun ændrer værdien hvis $v.d > \delta(s, u) + w(u, v)$. ■

- **Ingen vej egenskaben**

**Corollary 7.3.1.** *Lad $G = (V, E))$ være en orienteret graf med vægtsfunktion $w(u, v)$ på en kant $(u, v) \in G.E$. Lad $s$ være startknuden. Lad $s$ være startknuden. Hvis der ikke findes en vej fra $s$ til en anden knude $v \in G.V$, så har vi at $v.d$ er initialiseret til $v.d = \infty = \delta(s, v)$, og den ændrer sig ikke over alle relax trin på kanterne i $G.E$.*

***Bevis.*** På grund af den øvre grænse egenskab, har vi at $\infty = \delta(s, v) \leq v.d$ og derfor må $v.d = \infty$, og kan ikke blive lavere. ■

**Lemma 7.4.** *Lad $G = (V, E))$ være en orienteret graf med vægtsfunktion $w(u, v)$ på en kant $(u, v) \in G.E$. Lad $s$ være startknuden. Lad $(u, v) \in G.E$. Lige efter et kald på relax $(u, v)$ gælder at $v.d \leq u.d + w(u, v)$.*

***Bevis.*** Hvis kaldet til relax ændrer $v.d$, så må det gælde før kaldet, at $v.d > u.d + w(u, v)$, derfor må $v.d = u.d + w(u, v)$ efter kaldet. Hvis vi ikke ændrer $v.d$ efter kaldet, må det i forvejen gælde, at $v.d \leq u.d + w(u, v)$. ■

- **Konvergens egenskaben**

**Lemma 7.5.** *Lad $G = (V, E))$ være en orienteret graf med vægtsfunktion $w(u, v)$ på en kant $(u, v) \in G.E$. Lad $s$ være startknuden og lad*

$$s \rightsquigarrow u \rightarrow v$$

*være den korteste vej i $G$ for knuderne $u$ og $v$. Knuderne blivere initialiseret, og hvis efter en række relax trin, at $u.d = \delta(s, u)$, vil $v.d = \delta(s, v)$ efter det næste relax trin, og alle trin efter.*

***Bevis.*** Vi ved fra den øvre grænse egenskab, at hvis $u.d = \delta(s, v)$, vil den forblive sådan efter alle fremtidige relax kald. Efter et relax kald på kanten

$(u, v)$ har vi

$$\begin{aligned} v.d &\leq u.d + w(u,v) && \text{(Fra Lemma 7.4)} \\ &= \delta(s,u) + w(u,v) \\ &= \delta(s,v) && \text{(Fra Lemma 7.1, } p_{uv} \subset p_{sv}\text{)} \end{aligned}$$

Hvis $v.d \leq \delta(s,v)$, må $v.d = \delta(s,v)$. ■

- **Vej-relaxation egenskaben**

  **Lemma 7.6.** *Lad $G = (V,E))$ være en orienteret graf med vægtsfunktion $w(u,v)$ på en kant $(u,v) \in G.E$. Lad $s$ være startknuden. Hvis $p = \langle v_0, v_1, ..., v_k \rangle$, hvor $v_0 = s$, er den korteste vej fra $s \rightsquigarrow v_k$. Hvis vi, efter at initialiseret, har relaxet kanterne $(v_0,v_1),(v_1,v_2),...,(v_{k-1},v_k)$, så vil $v_k.d = \delta(s,v_k)$ efter det sidste relax kald, og alle fremtidige kald. Denne egenskab gælder også hvis relax bliver kaldt på andre kanter i $e \in G.E$.*

  ***Bevis.*** Vi viser ved induktion, at efter $i$ relax kald på $p$, så er $v_i.d = \delta(s,v_i)$.
  Base case er når $i = 0$, så har vi at $v_0.d = s.d = 0 = \delta(s,s)$. Nu antager vi at $v_{i-1}.d = \delta(s,v_{v-1})$, og vi kalder relax på kanten $(v_{i-1},v_i)$. På grund af konvergens egenskaben, vil $v_i.d = \delta(s,v_i)$ efter relax kaldet. ■

## 7.2 Bellman-Ford

Bellman-Ford algoritmen løser korteste vej problemet hvor vi tillader negative vægte. Algoritmen initialiserer alle knuderne, og derefter kører den et loop $|G.V| - 1$ og for hvert loop kalder den relax på alle kanter $(u,v) \in G.E$. Altså bliver hver kant relaxet $|V| - 1$ gange, og til sidst tjekker algoritmen alle kanter for

$$v.d > u.d + w(u,v)$$

Hvis en kant opfylder dette, så må det betyde, at grafen indeholder en negativ cykel, og algoritmen returnerer `FALSE`, ellers returnerer den `TRUE`, og den korteste vej kan konstrueres ved at følge en kants forælder – $v.\pi$.

### 7.2.1 Køretid

af Bellman-Ford er $\mathcal{O}(VE)$ da initialisering tager $\Theta(E)$ og vi går igennem alle $\Theta(E)$ kanter $|V|-1$ gange, altså $O(VE)$. Det sidste loop er $\mathcal{O}(E)$.

Korrektheden viser vi, ved at vise at tjekket vi laver i det sidste loop er korrekt.

**Lemma 7.7.** *Lad $G=(V,E)$ være en orienteret graf med vægtsfunktion $w(u,v)$ på kanter $(u,v)\in G.E$. Lad $s$ være startknuden.*
*Vi antager at $G$ ikke indeholder negative cykler, som kan nås fra $s$. Så efter $|V|-1$ relaxations af alle kanter, har vi at $v.d=\delta(s,v)$ for alle knuder $v$ der kan nås fra $s$.*

***Bevis.*** Vi bruger vej-relaxations egenskaben for grafen $G$. lad $p\langle s,v_1,v_2,...,v_k\rangle$ være den korteste vej $s\rightsquigarrow v_k$. Fordi korteste veje er simple, er der højst $|V|-1$ knuder i $p$, og derfor har vi $k\leq|V|-1$. Efter $i$ relaxationskald, har vi, på grund af vej-relaxationsegenskaben, at $v_k.d=\delta(s,v_k)$. ■

### 7.2.2 Korrekthed

**Theorem 7.8** (Korrekthed af Bellman-Ford)**.** *Vi kører Bellman-Ford på en orienteret graf $G=(V,E)$ med vægtsfunktion $w:G.E\to\mathbb{R}$. Lad $s$ være startknuden.*
*Hvis $G$ ikke har nogen negative cykler, så returnerer Bellman-Ford* `TRUE`*, og $v.d=\delta(s,v)$ for alle $v\in G.V$, og forgænger delgrafen $G_\pi$udgør et træ med rod $s$.*
*Hvis $G$ har negative cykler, der kan nås fra $s$, returnerer Bellman-Ford* `FALSE`*.*

***Bevis.*** Vi kigger på første tilfælde, hvor $G$ ikke har negative cykler.
Vi har at $v.d=\delta(s,v)$. Hvis $s\rightsquigarrow v$, så beviser lemma 7.8 dette, ellers hvis der ikke er en vej fra $s$ til $v$, så beviser ingen vej egenskaben, at det også gælder. Forgænger delgraf egenskaben viser at $G_\pi$ er korteste vej træet.
Nu viser vi at Bellman-Ford returnerer `TRUE`, ved at vise

$$\begin{aligned} v.d &= \delta(s,v) \\ &\leq \delta s,u+w(u,v) \text{ (Ved trekants uligheden)} \\ &= u.d+w(u,v) \end{aligned}$$

Antage at der findes en negativ cykel, så vil vi vise at Bellman-Ford returnerer `FALSE`.

Lad cyklen være $c = \langle v_0, v_1, ..., v_k \rangle$ hvor $v_0 = v_k$, så har vi

$$\sum_{i=1}^{k} w(v_{i-1}, v_i) < 0$$

Vi antager nu at Bellman-Ford returnerer `TRUE`, så gælder $v_i.d \leq v_{i-1}.d + w(v_{i-1}, v_i)$, for alle $i = 1, 2, ..., k$. Hvis vi summere alle ulighederne, har vi

$$\begin{aligned}\sum_{i=1}^{k} v_i.d &\leq \sum_{i=1}^{k} (v_{i-1}.d + w(v_{i-1}, v_i)) \\ &= \sum_{i=1}^{k} v_{i-1}.d + \sum_{i=1}^{k} w(v_{i-1}, v_i)\end{aligned}$$

Da $v_0 = v_k$ har vi også at

$$\sum_{i=1}^{k} v_i.d = \sum_{i=1}^{k} v_{i-1}.d \Rightarrow \sum_{i=1}^{k} v_i.d - \sum_{i=1}^{k} v_{i-1}.d = 0$$

altså

$$0 \leq \sum_{i=1}^{k} w(v_{i-1}, v_i)$$

Hvilket er i modstrid med at cyklen har negativ vægt. Altså, en korteste vej med negativ vægt kan ikke returnere `TRUE`, og må derfor returnerer `FALSE`. ■

## 7.3 Dijkstra

Dijkstras algoritme finder den korteste vej på grafer uden negative vægte, altså $w(u, v) \geq 0$ for alle $(u, v) \in G.E$.

Algoritmen holder en mængde $S$ af knuder, hvis korteste vej er fundet, og for iteration vælger den en ny knude i $v \in V - S$ med det laveste estimat $v.d.$. I implementationen bruger vi en min-prioritets kø til at kunne tage en knude med minimum $v.d$ værdi.

Først initialiseres alle knuder, $S = \emptyset$, og $Q = G.V$. Herefter trækker vi en knude $u$ fra $Q$ med minimums estimat. Som starter ud med at være $s.d = 0$. Vi tilføjer

den til $S$, og så relaxer vi alle kanter $(u, v)$ for $v \in u.adj$, hvor $adj$ er knudens naboliste.
Algoritmen holder en invariant $Q = G.V - S$ ved begyndelsen af hver iteration.

### 7.3.1 Korrekthed

**Theorem 7.9** (Korrekthed af Dijkstras algoritme)**.** *Vi kører Dijkstras algoritme på en orienteret graf $G = (V, E)$ med vægtsfunktion $w : G.E \to \mathbb{R}_{\geq 0}$. Lad $s$ være startknuden.*
*Algoritmen slutter med $u.d = \delta(s, u)$ for alle knuder $v \in G.V$.*

***Bevis.*** Vi beviser det med følgende invarians:

Ved begyndelsen af hver iteration, er $v.d = \delta(s, v)$ for $v \in S$.

Vi viser at når vi tilføjer en knude $u \in G.V$ til $S$, så er $u.d = \delta(s, u)$, da vi ved fra den øvre grænse egenskab, at den ikke vil ændre sig efter.

- **Initialisering**: I starten er $S = \emptyset$ og invarianten er trivielt sand.
- **Vedligeholdelse**: Vi ønsker at vise $u.d = \delta(s, u)$ når vi tilføjer $u$ til $S$. Lad os antage at $u$ er den første knude hvor $u.d \neq \delta(s, u)$ når vi tilføjer den til $S$. Vi ved at $u \neq s$ fordi $s$ er allerede i $S$, og fordi $S \neq \emptyset$, må der være en vej fra $s$ til $u$, ellers ville $u.d = \infty = \delta(s, u)$ ifølge ingen vej egenskaben.

  Siden der er en vej fra $s$ til $u$, må der også være en korteste vej fra $s$ til $u$. Før $u$ kommer i $S$, der der en knude $x \in S$ og en knude $y \notin S$, således at $s \rightsquigarrow x \to y \rightsquigarrow u$.
  Vi påstår at $y.d = \delta(s, y)$ når $u$ kommer i $S$. Vi ved at $x.d = \delta(s, x)$, og på grund af konvergens egenskaben har vi at $(x, y)$ blev relaxed, og derfor

er $y.d = \delta(s, y)$. Fordi $y$ kommer før $u$, har vi

$$\begin{aligned} y.d &= \delta(s, y) \\ &\leq \delta(s, u) \\ &\leq u.d \text{ (Ved øvre grænse egenskaben)} \end{aligned}$$

Men fordi vi vælger den knude med lavest estimat i $V - S$ gælder også $u.d \leq y.d$, hvilket fører til $y.d = \delta(s, y) = \delta(s, u) = u.d$, hvilket er i modstrid med, at $u.d \neq \delta(s, u)$.
Derfor må $u.d = \delta(s, u)$ når vi tilføjer $u$ til $S$.

- **Afslutning**: Når vi har tilføjet den sidste knude i $S$ er $Q = \emptyset$, hvilket må betyde at $S = V$. Derfor er $u.d = \delta(s, u)$ for alle $u \in G.V$.

■

### 7.3.2 Køretid

Vi kan opnå en køretid på $\mathcal{O}(VlgV + E)$ ved at bruge en Fibonacci hob til at implementere min-prioritetskøen. Vi laver $|V|$ inserts når vi initialiserer $Q = G.V$, hvilket er $\Theta(lgV)$. Vi laver $|V|$ `EXTRACT-MIN` som har en amortiseret køretid på $\mathcal{O}(lgV)$, og så i hvert loop laver vi højst $|E|$ `DECREASE-KEY`, som har en amortiseret tid på $\Theta(1)$. Dette giver at vi har

$$|V| \cdot \mathcal{O}(lgV) + |E| \cdot \Theta(1) = \mathcal{O}(VlgV + E)$$

# 8 Amortiseret Analyse

---

- Analysere en binær tæller
  - Aggregate analysis
  - Accounting method
- Potential method - Dynamisk tabel
  - Indsæt element
  - Fjern element

## 8.1 Analysere en binær tæller

Amortiseret analyse går ud på at tage gennemsnittet af en række operationers worst-case køretid. Det eksempel vi bruger til at beskrive de forskellige metoder til amortiseret analyse, er en binær tæller.

**Algorithm 8** Inkrementere en binær tæller

```
1: A[0...k − 1] array af bits som vi inkrementerer.
2: function INCREMENT(A)
3:     i = 0
4:     while i < k and A[i] == 1 do
5:         A[i] = 0
6:         i = i + 1
7:     if i < k then
8:         A[i] = 1
```

Funktionen kører over arrayet indtil den finder et sted at hvor den kan flippe en bit til 1. Det er let at se at vi looper over hele arrayet i worst-case, hvilket gør køretiden af `INCREMENT` $\mathcal{O}(k)$. Hvis vi så ønsker at inkrementerer et array $n$ gange, får vi en køretid på alle operationer af $\mathcal{O}(nk)$.

### 8.1.1 Aggregate analysis

Problemet med binær tællerens $\mathcal{O}(nk)$ køretid er, at vi næsten aldrig skal loope over hele arrayet. Vi kigger først på den amortiserede køretid med aggregate analysis.

Den $i$'te bit bliver flippet hver $2^i$'te gang, og $\lfloor \frac{n}{2^i} \rfloor$ gange i alt. Det totale antal bits der bliver flippet er derfor

$$\sum_{i=0}^{k-1} \lfloor \frac{n}{2^i} \rfloor = \mathcal{O}(n)$$

For at få den amortiserede køretid af hver `INCREMENT` tager vi gennemsnittet af alle bit flips.

$$\frac{\mathcal{O}(n)}{n} = \mathcal{O}(1)$$

### 8.1.2 Accounting method

Accounting metoden tildeler vi operationer en ny pris, deres amortiserede pris. Nogle operationers amortiserede pris kan være dyrere end deres rigtige pris, og nogle har en billigere pris. Hvis en operations amortiserede pris er højere end den rigtige pris, gemmes det overskydende som kredit, og hvis en operations amortiserede pris er lavere, må den låne kredit gemt på datastrukturen til at betale for den.Den amortiserede pris skal være sådan, at der ikke kan være negativ kredit gemt, og hvis det gælder, så er den totale køretid derfor summen af alle operationernes amortiserede pris.

I vores eksempel med binær tælleren, giver vi den operation der flipper en bit til 1 en pris på 2, og det at flippe en bit til 0 en pris på 0. Intuitionen er, at en bit kan ikke flippes til 0 før den har været flippet til 1, derfor betaler den operation der flipper den til 1 både for sig selv og for operationen der flipper tilbage til 0. Når vi udfører en `increment` flipper vi kun 1 bit til 1, og derfor koster selve `INCREMENT` 2, og derfor er antallet af kredit gemt lig med antallet af 1'ere i arrayet. Den totale køretid er summen af alle operationers pris $2n = \mathcal{O}(n)$, og selve increment er $2 = \mathcal{O}(1)$.

## 8.2 Potential method - Dynamisk tabel

I potential metoden, bruger vi en funktion $\Phi(D_i)$, som er en funktion af vores datastruktur efter $i$ operationer, og som vi kalder potentialet. Vi udregner så den amortiserede pris af den $i$'te operation, som den rigtige pris af operationen plus ændring i potentialet efter operationen.

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

Ligesom accounting metoden, så kan potentialet efter en operation bruges til at betale for andre operationer senere hen, så derfor, så længe potentialet ikke er negativt, så er summen af de amortiserede køretider en øvre grænse for operationernes køretider.

Vi bruger et andet eksempel end en binær tæller – Vi kigger på en dynamisk tabel, hvor vi fordobler længden når det er fyldt, og halverer når det er for kort.

Vi definerer **load-factor** $\alpha(T)$ til at være antallet af elementer divideret med antallet af pladser i tabellen.
Vi antager at vores dynamiske tabel understøtter følgende operationer, hvis amortiserede køretider vi vil kigge på:

- `TABLE-INSERT` lægger et element ind i vores tabel. Vi fordobler når tabel er fyldt, altså når load-factoren $\alpha(T) = 1$
- `TABLE-DELETE` fjerner et element fra vores tabel. Vi halverer når der er en fjerdedel ledige pladser, altså når load-factoren $\alpha(T) = 1/4$

Vores dynamiske tabel fungerer således, at hvis tabellen bliver fyldt efter en `TABLE-INSERT`, altså den har en load-factor på 1, så allokerer vi en ny tabel af dobbelt størrelse. Umiddelbart har `TABLE-INSERT` en worst-case køretid på $\mathcal{O}(n)$ siden den skal kopiere $n$ elementer fra den gamle tabel til den nye, men vi kan vise at den har en amortiseret køretid på $\mathcal{O}(1)$, da den ikke skal udvide tabelen særligt ofte.

Vi definere en potential funktion på tabellen efter $i$ operationer

$$\Phi(T_i) = \begin{cases} 2 \cdot T_i.num - T_i.size & \text{hvis } \alpha(T_i) \geq 1/2 \\ T_i.size/2 - T_i.num_i & \text{hvis } \alpha(T_i) < 1/2 \end{cases}$$

Vi lægger mærke til nogle egenskaber af vores potentiale funktion.
Hvis $\alpha(T) = 1$, så er $T.num = T.size$, og $\Phi(T) = 2T.num - T.num = T.num$, hvilket betyder at vi har nok potentiale til at rykke alle elementer ind i den større tabel.
Ligeledes, når vi skal halverer, så er $\alpha(T) = 1/4$, og $\Phi(T) = 2T.num - T.num = T - num$ og vi har igen nok potentiale til at rykke alle elementer ind i den mindre tabel. Hvis $\alpha(T) = 1/2$, så er potentialet 0, og vi kan frit indsætte og fjerne, uden at skulle bekymre om tabelstørrelsen.

Nu ser vi på $n$ `TABLE-INSERT` og `TABLE-DELETE` operationer på vores dynamiske tabel. Der er 6 tilfælde vi kigger på.

| TABLE-INSERT | | TABLE-DELETE | |
|---|---|---|---|
| $\alpha(T_{i-1}) \geq 1/2 \Rightarrow$ | $\alpha(T_i) \geq 1/2$ | $\alpha(T_{i-1}) < 1/2 \Rightarrow$ | $\alpha(T_i) < 1/2$ |
| $\alpha(T_{i-1}) < 1/2 \Rightarrow$ | $\alpha(T_i) < 1/2$ | $\alpha(T_{i-1}) \geq 1/2 \Rightarrow$ | $\alpha(T_i) \geq 1/2$ |
| | $\alpha(T_i) \geq 1/2$ | | $\alpha(T_i) < 1/2$ |

### 8.2.1 Indsæt element

Vi starter med at kigge på den amortiserede køretid i tilfælde af `TABLE-INSERT`. Vi ser på når $\alpha(T_{i-1}) \geq 1/2$, på de to tilfælde, hvor vi enten fordobler, eller ikke fordobler.

Hvis en indsættelse ikke fordobler tabellen, så har vi at $T_i.num = T_{i-1}.num + 1$ og $T_i.size = T_{i-1}.size$:

$$\begin{aligned}
\hat{c_i} &= c_i + \Phi(T_i) - \Phi(T_{i-1}) \\
&= 1 + (2T_i.num - T_i.size) - (2T_{i-1}.num - T_{i-1}.size) \\
&= 1 + (2(T_{i-1}.num + 1) - T_{i-1}.size) - (2T_{i-1}.num - T_{i-1}.size) \\
&= 1 + 2 \\
&= \underline{3}
\end{aligned}$$

Og hvis en indsættelse fordobler, så har vi at $c_i = T_i.num$, siden vi skal rykke alle elementerne,
$T_{i-1}.num = T_i.num - 1$, og $T_i.size = 2T_i.num$, og $T_{i-1}.size = T_{i-1}.num = T_i.num - 1$:

$$\begin{aligned}
\hat{c_i} &= c_i + \Phi(T_i) - \Phi(T_{i-1}) \\
&= T_i.num + (2T_i.num - T_i.size) - (2T_{i-1}.num - T_{i-1}.size) \\
&= T_i.num + (2T_i.num - 2(T_i.num - 1)) - (2(T_i.num - 1) - (T_i.num - 1)) \\
&= T_i.num + 2 - (2T_i.num - 2 - T_i.num + 1) \\
&= T_i.num + 2 - T_i.num + 1 && = 2 + 1 = \underline{3}
\end{aligned}$$

Hvis vi indsætter et element når $\alpha(T) \geq 1/2$, så er den amortiserede pris altid $\hat{c} = 3$.

Nu Kigger vi på indsættelse hvor $\alpha(T_{i-1}) < 1/2$.
Første tilfælde er når $\alpha(T_i) < 1/2$. Her har vi at $T_{i-1}.num = T_i.num - 1$ og $T_{i-1}.size = T_i.size$:

$$\begin{aligned}
\hat{c_i} &= c_i + \Phi(T_i) - \Phi(T_{i-1}) \\
&= 1 + (T_i.size/2 - T_i.num) - (T_{i-1}.size/2 - T_{i-1}.num) \\
&= 1 + (T_i.size/2 - T_i.num) - (T_i.size/2 - (T_i.num - 1)) \\
&= 1 - 1 \\
&= \underline{\underline{0}}
\end{aligned}$$

Når $\alpha(T_i) \geq$
Her gør vi brug af, $T_i.num = \alpha(T_i) \cdot T_i.size$:

$$\begin{aligned}
\hat{c_i} &= c_i + \Phi(T_i) - \Phi(T_{i-1}) \\
&= 1 + (2T_i.num - T_i.size) - (T_{i-1}.size/2 - T_{i-1}.num) \\
&= 1 + (2(T_{i-1}.num + 1) - T_{i-1}.size) - (T_{i-1}.size/2 - T_{i-1}.num) \\
&= 1 + 3T_{i-1}.num - \frac{3}{2}T_{i-1}.size + 2 \\
&= 1 + 3\alpha(T_{i-1}) \cdot T_{i-1}.size - \frac{3}{2}T_{i-1}.size + 2 \\
&< 1 + \frac{3}{2}T_{i-1}.size - \frac{3}{2}T_{i-1}.size + 2 \\
&= 1 + 2 \\
&= \underline{\underline{3}}
\end{aligned}$$

Altså er den amortiserede cost af `TABLE-INSERT` grænset op ad til, af konstanten 3.

### 8.2.2 Fjern element

Nu kigger vi på `TABLE-DELETE` når $\alpha(T_{i-1}) < 1/2$. Ligesom med indsættelse, skal vi kigge på tilfældet hvor vi skal halvere tabellen, og hvor vi ikke skal.

Først kigger vi på tilfældet hvor vi ikke halverer.

$$\begin{aligned}
\hat{c_i} &= c_i + \Phi(T_i) - \Phi(T_{i-1}) \\
&= 1 + (T_i.size/2 - T_i.num) - (T_{i-1}.size/2 - T_{i-1}.num) \\
&= 1 + (T_i.size/2 - T_i.num) - (T_i.size/2 - (T_i.num + 1)) \\
&= 1 - T_i.num + T_i.num + 1 \\
&= \underline{\underline{2}}
\end{aligned}$$

I tilfældet hvor vi skal halverer, så er vi at $c_i = T_{i-1}.num$, siden vi skal rykke alle elementer i den mindre tabel.
Derudover har vi, at $T_i.size/2 = T_{i-1}.size/4 = T_{i-1}.num = T_inum + 1$, siden vi ved vi har $\alpha(T_i) = 1/4$ når vi halverer:

$$\begin{aligned}
\hat{c_i} &= c_i + \Phi(T_i) - \Phi(T_{i-1}) \\
&= T_{i-1}.num + (T_i.size/2 - T_i.num) - (T_{i-1}.size/2 - T_{i-1}.num) \\
&= (T_i.num + 1) + ((T_i.num + 1) - T_i.num) - (2(T_i.num + 1) - (T_i.num + 1)) \\
&= T_i.num + 1 + 1 - (T_i.num + 2 - 1) \\
&= T_i.num + 2 - T_i.num - 1 \\
&= \underline{\underline{1}}
\end{aligned}$$

Til sidst kigger vi på de to tilfælder hvor $\alpha(T_{i-1}) \geq 1/2$, og der så enten gælder at $\alpha(T_i) < 1/2$ eller $\alpha(T_i) \geq 1/2$.

Hvis $\alpha(T_i) < 1/2$.
Her bruger vi igen at vi ved, at $T_i.num = \alpha(T_i) \cdot T_i.size$:

$$\begin{aligned}
\hat{c_i} &= c_i + \Phi(T_i) - \Phi(T_{i-1}) \\
&= 1 + (T_i.size/2 - T_i.num) - (2T_{i-1}.num - T_{i-1}.size) \\
&= 1 + (T_i.size/2 - T_i.num) - (2(T_i.num + 1) - T_i.size) \\
&= 1 + \frac{3}{2}T_i.size - 3T_i.num_i - 2 \\
&= -1 + \frac{3}{2}T_i.size - 3\alpha(T_i) \cdot T_i.size \\
&< -1 + \frac{3}{2}T_i.size - \frac{3}{2}T_i.size \\
&= \underline{\underline{-1}}
\end{aligned}$$

Hvis $\alpha T_i \geq 1/2$:

$$\begin{aligned}
\hat{c_i} &= c_i + \Phi(T_i) - \Phi(T_{i-1}) \\
&= 1 + (2T_i.num - T_i.size) - (2T_{i-1}.num - T_{i-1}.size) \\
&= 1 + (2T_i.num - T_i.size) - (2(T_i.num + 1) - T_i.size) \\
&= 1 + 2T_i.num - T_i.size - 2T_i.num - 2 + T_i.size \\
&= 1 - 2 \\
&= \underline{\underline{-1}}
\end{aligned}$$

Nu har vi vist at to egenskaber holder – Load-factoren $\alpha(T_i)$ er grænset ned ad til, af en konstant $1/4$, og vores to tabel operationer er grænset op ad til af konstanten 3.
$n$ operationer er derfor $\mathcal{O}(n)$ hvilket giver at hver operation er $\mathcal{O}(1)$. ■