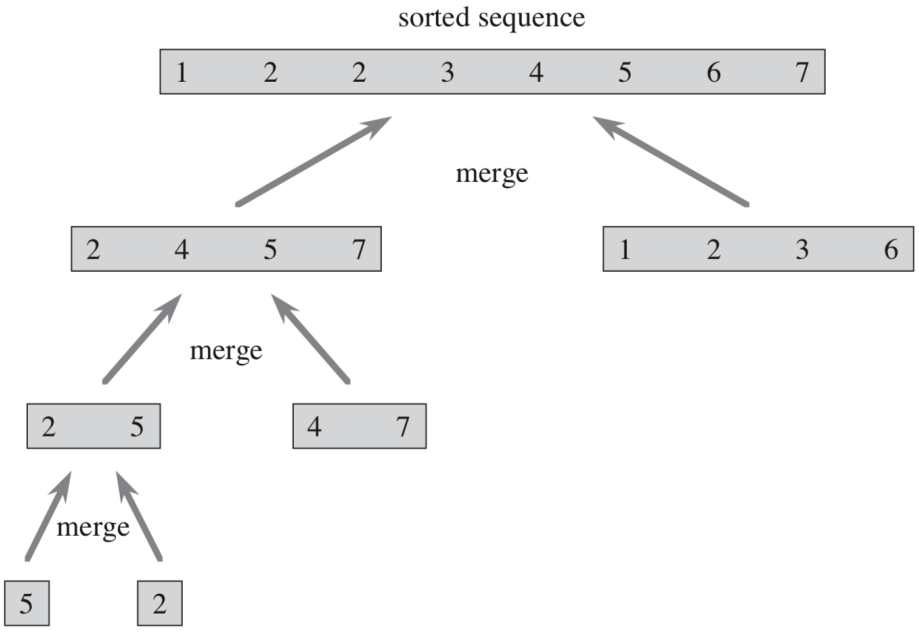
1 Divide and Conquer (Del-og-hersk)

- **Paradigmet**
  - *Del* problemet ind i en mængde af delproblemer som er mindre instanser af det samme problem.
  - *Hersk* (løs) delproblemerne ved at løse dem rekursivt. Hvis delproblemerne er tilpas små, så løs dem bare "direkte".
  - *Kombinér* løsningerne til delproblemerne sammen til én løsning til det oprindelige problem.
  - Minder en smule om Dynamisk Programmering, men forskellen er, at problemerne ikke har optimal delstruktur, altså der er ikke overlappende delproblemer.
  - Eksempler på algoritmer der benytter det er `QuickSort`, `Find-Maximum-Subarray` og `MergeSort`, som jeg vil gå mere i dybden med nu.
- **Håndkørsel af `MergeSort`**



Figur 1: Tegn et træ ala dette, hvor du så kun viser ned i venstre side. Sørg for at holde det 8 elementer langt.

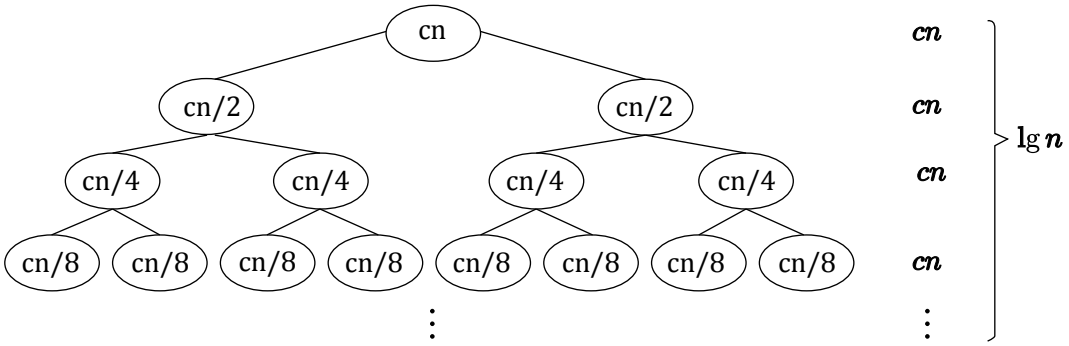
Jeg vil vise det på en 2-talspotens, for det gør det lidt nemmere at illustrere, men princippet ville være det samme for andre tal. Del arrayet op i to (ca.) lige store dele, kald `Merge-Sort` på disse og `Merge` til sidst de nu sorterede delarrays. Algoritmen kører i $O(n \lg n)$ tid, og det vil jeg nu bevise.

- **Rekursionsrelationer**
  Rekursionsrelationer er de typer relationer vi typisk vil få ved del og hersk algoritmer. Som et eksempel ses her rekursionsrelationen for `MergeSort`, hvor vi undlader at skrive casen når $n = 1$ eksplicit og ignorer, at vi egentlig skal tage floor og ceiling, altså tager udgangspunkt i den case hvor vi har en 2-talspotens. Vi får denne rekursionsrelation, da vi åbner to nye delproblemer der er af halv størrelse af det originale, og derudover til sidst skal samle det originale:

$$T(n) = 2T(n/2) + cn$$

- **Rekursionstræer**
  Kan, hvis man er meget præcis, bruges til at bevise en rekursionsrelation. Bruges dog oftest til

at komme på et godt gæt man så kan bevise med substitutionsmetoden. Her ses rekursionstræet vi får for `MergeSort` idet vi bruger ovenstående rekursionsrelation:



Figur 2: Rekursionstræ for `MergeSort`

Vi ser, at hvert niveau i træet bidrager med cn og der er $\lg n$ af disse led. Derfor gætter vi på, at køretiden af `MergeSort` er $O(n \lg n)$. Vi gætter på dette:

$$T(n) = 2T(n/2) + n \qquad \text{har køretid} \qquad O(n \lg n)$$

- **Substitutionsmetoden**

  1. Gæt på en løsning (enten ud fra erfaring eller ved at tegne et rekursionstræ)
  2. Bevis din løsning er sand ved hjælp af induktion (OBS: Vær opmærksom på vi ikke kan være så ligeglad med konstanter som vi ellers tit er med O-notation). Hvis vi løber ind i et problem kan det ofte være en hjælp at trække et led af en lavere orden fra.
  3. Hvis du stadig ikke kan bevise det, så lav et nyt gæt og prøv igen.

Til dette bevis skulle vi normalt bruge stærk induktion, men da vi antager det er en 2-talspotens er det ikke nødvendigt.

Base case:
Vi skal nu vise $T(n) \leq cn \lg n$, hvor vi antager $T(1) = 1$. Vi ser, at vi ikke kan vise det for $n = 1$ da vi så får $cn \lg n = 0$ da $\lg 1 = 0$.

For $n = 2$:

$$T(2) = 2T(\lfloor 2/2 \rfloor) + 2 = 2T(1) + 2 = 4 \leq c2 = c2 \lg 2$$

Hvilket er sandt for $c \geq 2$. Nu går vi til induktionssteppet:

Induktionsstep:

$$\begin{aligned} T(n) &\leq 2c \frac{n}{2} \lg(n/2) + n && (1)\\ &= cn \lg(n/2) + n && (2)\\ &= cn \lg n - cn \lg 2 + n && (3)\\ &= cn \lg n - cn + n && (4)\\ &\leq cn \lg n && (5) \end{aligned}$$

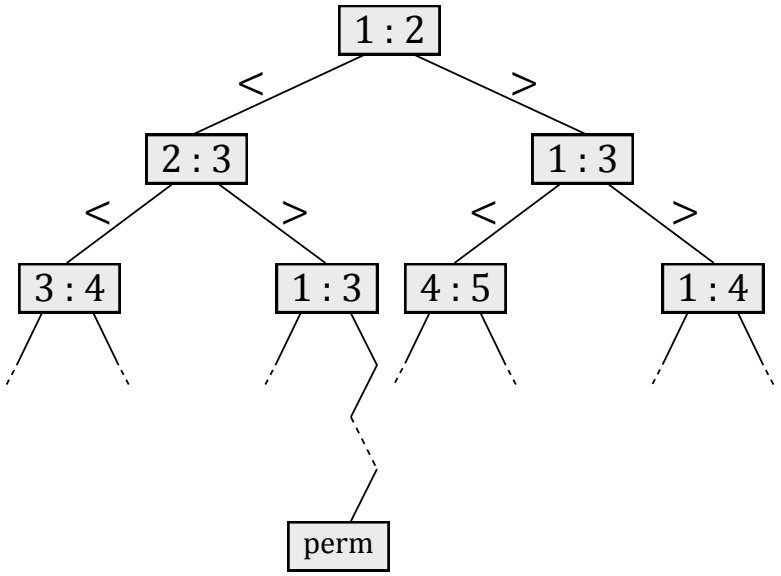
I Eq. (1) benytter vi blot vores induktionsantagelse.
I Eq. (2) ganger vi 2 ind i brøken.
I Eq. (3) deler vi logaritmen op jf. logaritmeregneregler.
I Eq. (4) fjerner vi multiplikationsleddet $\lg 2 = 1$.
I Eq. (5) ved vi der gælder, at $-cn + n \leq 0$ så længe $c \geq 1$.

- **Mastermetoden**
  Lad $a \geq 1$ og $b > 1$ være konstanter samt $f(n)$ en funktion, og lad $T(n)$ være defineret ved rekursionsrelationen:
  $$T(n) = aT(n/b) + f(n)$$
  hvor n/b kan være være både $\lfloor n/b \rfloor$ eller $\lceil n/b \rceil$. Så er der tre forskellige cases, og vi kan næsten altid beregne $T(n)$ asymptotiske grænse.

- **Lower bound for comparison sort**
  Vi antager for at gøre det nemmere, at der ikke er elementer som er ens. Derudover vil vi kun bruge operationen $a < b$ til at få informationer om to elementers relation, da alle andre giver ækvivalent information.

  Antag vi får et input array A med n elementer. Nu anvender vi en comparison sort S på denne. Det S gør, er at permutere elementer i A til en sorteret liste.

  Denne permutation er unikt defineret ud fra de sammenligninger algoritmen har foretaget. Alle disse udfald kan vi repræsentere i et beslutningstræ. Alle simple veje fra rodknuden til et blad er alle de forskellige permutationer der findes.



  Lad nu antallet af blade være b og højden h. Såfremt træet er perfekt balanceret kan vi presse 2^h blade ind i træet. Og vi ved også, at alle permutationer forekommer i bladene, derfor må $b \geq n!$ for ellers ville der ikke være plads til alle permutationer i bladene. Derfor får vi:

  $$2^h \geq b \geq n!$$

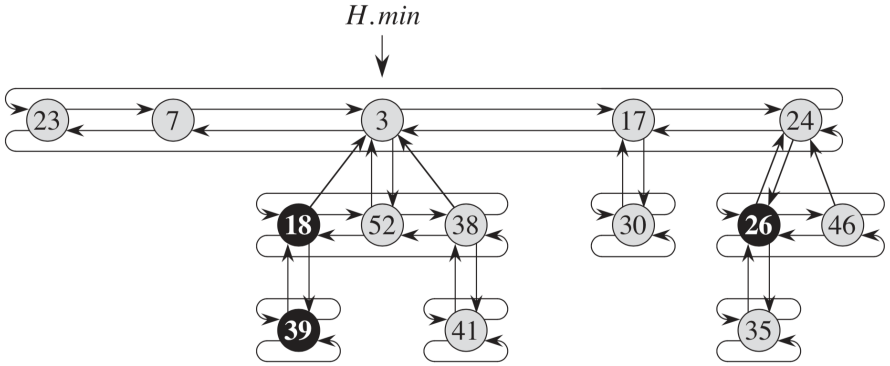
  Vi kan tage 2tals-logaritmen hvorved vi får:

  $$h \geq \lg(n!) = \Omega(n \lg n)$$

  Herved har vi altså bevist, at der er et input af længde n som kræver at der bliver foretaget $h = \Omega(n \lg n)$ sammenligner.

1 Fibonacci Heaps

- **Motivation**
  - God amortiseret køretid, mange af operationerne er i konstant tid.
  - Gør den velegnet til applikationer som benytter den mange gange. Særligt når `Extract-Min` og `Delete` bliver kaldt få gange relativt set.
  - Bl.a. Dijkstra's algoritme og Prim's algoritme (Minimum Spanning Trees) gør brug af Fibonacci Heaps.
  - Ligesom vi her vil tale om en Min-Heap kunne man ligeledes lave en Max-Heap.
- **Struktur**
  - Min-heap-egenskab: Key'en for en knude er altid $\geq$ key'en for dens forælder. En heap H har pointeren $H.min$ til minimumsknuden ved roden af træet.
  - Knuden x har pointerne/attributerne
    - $x.p$ - Forælder-knude
    - $x.child$ - En vilkårlig af børnene
    - $x.left$ og $x.right$
    - $x.mark$ - Sandhedsværdi som markerer om knuden har mistet ét barn siden den sidst blev gjort til et barn af en anden knude. Nye knuder er umarkerede, og bliver derudover umarkerede hver gang de bliver gjort til barnet af en anden knude.
    - $x.degree$ - Antallet af børn i barnelisten (f.eks. for knuden x med $x.key = 3$ i figuren nedenfor er $x.degree = 3$).
  - Når en Fibonacci Heap H er tom er $H.min = NIL$.
  - Vi bruger en cirkulær doubly linked lists til at forbinde rodknuderne og børnelisterne for en bestemt knude (se nedenfor for illustration):

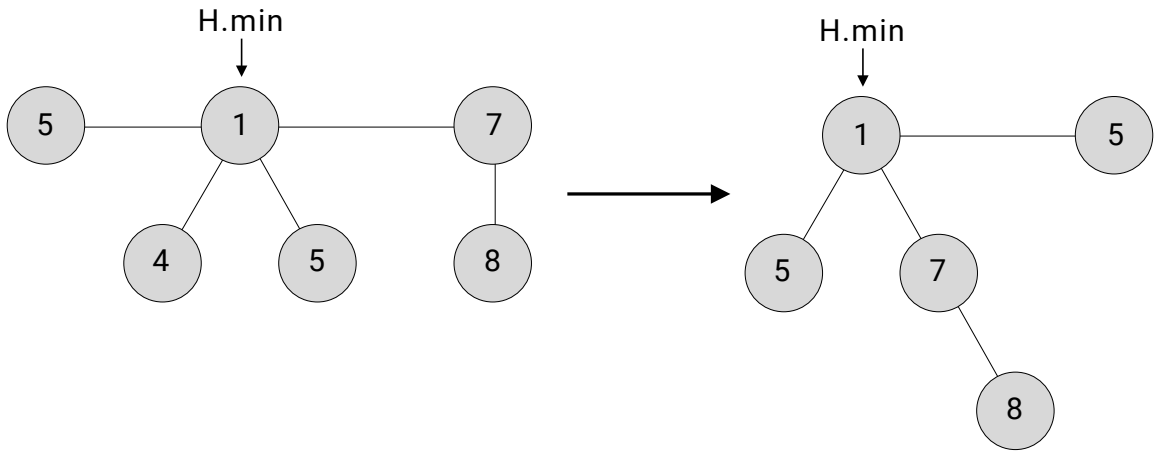


  - Operationernes amortiserede køretid og argumenter (H = heap, x = knude og k = ny key):

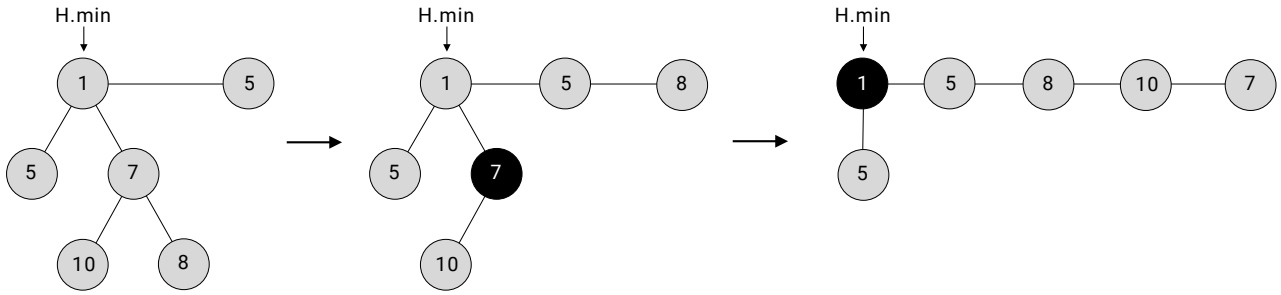
Operation	Amortiseret køretid
`Make-Heap`()	$\Theta(1)$
`Insert`(H, x)	$\Theta(1)$
`Minimum`(H)	$\Theta(1)$
`Union`(H_1, H_2)	$\Theta(1)$
`Decrease-Key`(H, x, k)	$\Theta(1)$
`Extract-Min`(H)	$\Theta(\lg n)$
`Delete`(H, x)	$\Theta(\lg n)$

  - Det tager lang tid at søge efter et element, så i ovenstående køretider antager vi at vi allerede har en pointer direkte til den knuden x vi gerne vil udføre en operation ved.

- **Håndkørsel af `Extract-Min`**
  Her er hoben før operationen og hoben efter tegnet. `Extract-Min` er det hvor vi laver et array $A[0..D(n)]$ og ser på degree af hver knude i roden.



- **Håndkørsel af `Decrease-Key`**
  Tegn en ekstra knude under 7 på det træ du kom frem til før. Fjern knuderne med key 10 og key 8. Træet før vi begynder, efter første decrease og efter andet decrease vil se således ud:



- **Potentialefunktion**
  - Definer potentialefunktionen:

$$\Phi(H) = \underbrace{t(H)}_{\text{Antal rod-knuder}} + 2\overbrace{m(H)}^{\text{Antal markerede knuder}}$$

  - Gyldighed: Vi ser at den er gyldig, da idet der er ingen knuder i vors Fib-Heap vil den være 0. Ellers kan der kun være et positivt antal knuder og/eller markerede knuder, og derfor vil den ellers altid være positiv.
  - Den amortiserede cost af enhver operation er dens faktiske cost plus ændringen i potentialet. Den totale amortiserede cost af n operationer er: *(Vi ved dette fra kapitel 17, uddyb ikke)*

$$\sum_{i=1}^{n} \hat{c}_i = \left(\sum_{i=1}^{n} c_i\right) + \Phi(D_n) - \Phi(D_0)$$

  - Antag øvre grænse $D(n) = O(\lg n)$ for den maksimale degree på enhver knude i en n-knude stor Fib-Heap. *(Kursorisk, skal ikke bevises)*

- **Bevis for amortiseret køretid for `Extract-Min` (s. 517 bund - 518)**

Faktisk cost: $O(D(n))$ fra først maksimalt at flytte $D(n)$ børn op i roden og for at initialisere vores $D(n)$ lange degree-liste A samt at finde $H.min$ til sidst.

Derudover får vi i værste fald at antallet af rod-knuder lige efter at have `Consolidate`'d er $D(n) + t(H) - 1$ hvor:

$$\underbrace{D(n)}_{\text{Børnene fra den udtrakte knude}} + \overbrace{t(H)}^{\text{De originale rodknuder}} \underbrace{-1}_{\text{Det udtrakte element}}$$

Vi ved, at vi hver eneste gang i loopet enten rykker én frem eller linker to knuder sammen, og derfor er det totale arbejde i værste fald følgende, hvor vi antager uligheden:

$$O\Big(D(n) + t(H)\Big) \leq D(n) + t(H)$$

Potentiale: Potentialet før er $t(H) + 2m(H)$
Potentialet efter er $(D(n) + 1) + 2m(H)$ (siden højest $D(n) + 1$ rødder er tilbage da det er længden af vores array A og ingen knuder bliver markeret).

Udregning:

$$\begin{aligned}
\hat{c}_i &= c_i + \Phi(H') - \Phi(H) && (1)\\
&= D(n) + t(H) + \Big((D(n) + 1) + 2m(H)\Big) - \Big(t(H) + 2m(H)\Big) && (2)\\
&= 2D(n) + 1 && (3)\\
&= O(D(n))\\
&= O(\lg n)
\end{aligned}$$

Intuition: Vores cost af at udføre hver eneste link af to knuder bliver betalt for af reduktionen i potentialet der sker når et link reducerer antallet af knuder i træet med 1.

- **Bevis for amortiseret køretid for `Decrease-Key` (s. 520-522)**

Faktisk cost: Selve det at formindske vores key tager konstant tid. Herefter kan der være c kald hvor forælderknuderne går fra sand til falsk så de skal flyttes op i roden, som hver for sig tager konstant tid, hvorved vi får en køretid på c.

Potentiale: For alle c rekursive kald pånær det sidste fjerner vi markeringen for forælderknude og flytter vores nuværende knude op i roden. Altså vil der efter udførslen være følgende antal træer:

$$t(H') = t(H) + c$$

Derudover vil der højst være følgende antal markerede knuder, da $c-1$ træer blev umarkeret i processen, mens den allersidste muligvis blev markeret:

$$m(H') = m(H) - c + 2$$

Udregning: Ændringen i potentiale vil højst være:

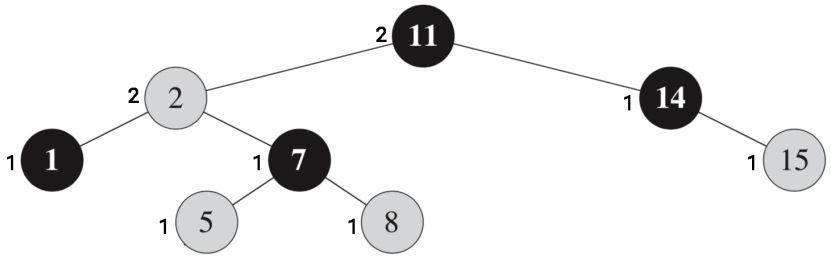
$$\begin{aligned}
\Phi(H') - \Phi(H) &= \Big((t(H) + c) + 2(m(H) - c + 2)\Big) - \Big(t(H) + 2m(H)\Big)\\
&= \Big((t(H) + c) + 2m(H) - 2c + 4\Big) - \Big(t(H) + 2m(H)\Big)\\
&= 4 - c
\end{aligned}$$

Herved fås den amortiserede køretid til at være:

$$\hat{c}_i = c + (4 - c) = O(1)$$

1 Balancerede binære søgetræer

- **Motivation**
  - En effektiv datastruktur, som tilbyder worst-case garantier for ting som `Insertion`, `Deletion` og `Search`.
  - Bliver f.eks. brugt i plane-sweep algoritmen (der ser om linjesegmenter skærer hinanden), samt i Linuxkernen og til at understøtte forskellige operationer i flere programmeringssprog.
- **Struktur for binære søgetræer**
  - Attributer for hver knude: *key* samt $left$, $right$ og p.
  - Opsat så vi altid har $\leq$ elementer i venstre deltræ og $>$ elementer på højre deltræ.
  - Mange operationer, inklusiv `Insert`, `Search`, `Delete` samt `Max` og `Min` kan nu udføres i $O(h)$ tid. Ved tilfældige indsættelser kan man vise at højden vil blive $O(\lg n)$, men vi er ikke altid garanteret at indsættelserne er tilfældige.
- **Struktur for rød-sorte træer**
  - Ligesom binære søgetræer med den ene ekstra attribut *color*, som kan være rød eller sort. Løser problemet med at garantere en højde $h = O(\lg n)$.
  - Opfylder 5 rød-sorte egenskaber:
    1. Hver knude er enten rød eller sort
    2. Roden er sort
    3. Alle blade er $T.nil$, som er sort
    4. Hvis en knude er rød, så er begge dens børn sorte
    5. For hver eneste knude gælder, at alle simple veje fra knuden til dens efterkommere har det samme antal sorte knuder. Denne kalder vi $bh(x)$.
  - Tegn figuren. Knudernes black-height $bh(x)$ er skrevet til venstre for dem:



- **Bevis for at højden $h = O(\lg n)$ (s. 309)**
  - Jeg vil bevise, at et rødt-sort træ x med n interne knuder har en højde

$$h = O(\lg n)$$

  - Starter med Lemma. Lad $s(x)$ betegne antallet af interne knuder i et deltræ for knuden x. Vi starter så med ved stærk induktion at vise, at dette deltræ har mindst $2^{bh(x)} - 1$ interne knuder.

$$s(x) \geq 2^{bh(x)} - 1 \tag{1}$$

  - Base case (x har en højde $h = 0$):
    - $s(x) = 0$, da x med den højde så må være et eksternt blad.
    - $bh(x) = 0$ og derfor $2^{bh(x)} - 1 = 0$.
    - Vi ser at Eq. (1) er overholdt.
  - Induktionsstep:

* Lad os nu sige knude x har højde $h = k$. Så vil dens to børn $x.left$ og $x.right$ have en højde der højest er $k - 1$. Pga. vores induktionsantagelse ved vi, at Eq. (1) gælder for dem. Vi får nu:

$$\begin{aligned} s(x) &= s(x.left) + s(x.right) + 1 && (2)\\ &\geq 2^{bh(x.left)} - 1 + 2^{bh(x.right)} + 1 && (3)\\ &\geq 2^{bh(x)-1} - 1 + 2^{bh(x)-1} - 1 + 1 && (4)\\ &= 2^{bh(x)} - 1 && (5) \end{aligned}$$

I Eq. (3) benytter vi vores induktionsantagelse.
I Eq. (4) benytter vi, at et barn har en sort-højde $bh(x)$ (rød) eller $bh(x) - 1$ (sort).
I Eq. (5) lader vi to 1-taller gå ud med hinanden og sætter de potenser sammen.

- Hermed har vi bevist vores Lemma. Nu skal bevise vores påstand. Vi benytter nu egenskab nr. 4, nemlig at hver rød knude har to sorte børn. Dvs. antallet af røde knuder på en vej fra roden ikke kan være med end $h/2$. Derfor må antallet af sorte knuder på sådan en vej minimum være $h/2$. Derfor får vi, at $bh(r) \geq h/2$.
- Nu samler vi disse uligheder hvorved vi får:

$$\begin{aligned} n = s(r) \geq 2^{bh(r)} - 1 &\geq 2^{h/2} - 1\\ &\Updownarrow\\ n + 1 &\geq 2^{h/2}\\ \lg(n+1) &\geq h/2\\ 2\lg(n+1) &\geq h\\ h &= O(\lg n) \end{aligned}$$

- Det er hermed bevist.

Opretholdelse af egenskaber:
Jeg har hermed bevist, at højden på et RB-søgetræ er $O(\lg n)$. For at det skal gøre sig gældende er det meget vigtigt at vi overholder de fem egenskaber. Det kan vi gøre ved:

1. LR (`Left-Rotation`)
2. RR (`Righ-Rotation`)
3. Farve en knude sort eller rød

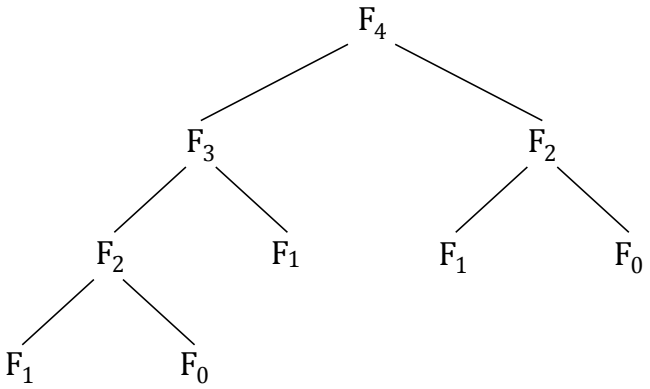
Vi kan have to cases når vi indsætter eller sletter en knude, det er:

1. Vi overskrider ingen egenskaber og kan bare fortsætte
2. Vi overskrider egenskaber, og skal rette op på det.

Hvis det er case 2, så kan vi løse problemet lokalt, og hvis problemet bliver ved med at probagere op ad træet, helt op til roden er rød, så maler vi den bare sort.

1 Dynamisk Programmering

- **Paradigmet**
  - Minder en anelse om del og hersk-paradigmet: Del problemet ind i mindre delproblemer, løs delproblemerne rekursivt og kombiner herefter løsningerne.
  - Forskellen ligger i, at problemerne har *optimal delstruktur*. Det vil sige delproblemerne på en eller anden måde overlapper. Altså når flere delproblemer har de samme deldelproblemer. Det kunne f.eks. være når vi skal beregne et Fibonacci tal eller i det jeg vil vise med LCS.
- **Eksempel med Fibonacci**



Figur 1: Problemer der skal løses i Fib. Her ser vi, at F_2 optræder flere gange.

Uden DP er køretiden eksponentiel, med DP er køretiden $O(n)$ (såfremt vi kan lægge store tal sammen i konstant tid).

- **Beskrivelse af LCS-problemet**
  Vi vil gerne finde Longest Common Subsequence af f.eks. to DNA-strenge.
  Hvis vi f.eks. har $S_1 = ACCG$ og $S_2 = AG$ ville den længste den længste sekvens være AG, og længden er hermed 2.
- **Theorem 15.1 (*Optimal delstruktur af en LCS*)**
  Lad $X = \langle x_1, x_2, ..., x_m \rangle$ og $Y = \langle y_1, y_2, ..., y_n \rangle$ være sekvenserne (f.eks. DNA-strenge), og lad $Z = \langle z_1, z_2, ..., z_k \rangle$ være en hvilken som helst LCS af X og Y:

  1: Hvis $x_m = y_n$, så er $z_k = x_m = y_n$ og Z_{k-1} er en LCS af X_{m-1} og Y_{n-1}.

  2: Hvis $x_m \neq y_n$ og $z_k \neq x_m$, så medfører det at Z er en LCS af X_{m-1} og Y.

  3: Hvis $x_m \neq y_n$ og $z_k \neq y_n$, så medfører det at Z er en LCS af X og Y_{n-1}.

- **Beviser af Theorem 15.1**
  Alle er modstridsbeviser. Tegn strenge ala. følgende for at understøtte argumentation:

```
X: *****A
Y: ****A
Z: **A
```

  1: Antag $z_k \neq x_m$. Så kunne vi appende $x_m = y_n$ på Z, hvorved vi fik en CS af X og Y med en længde $k + 1$.
  Men vi antog jo netop at Z var en *longest* common subsequence med længden k, og derfor har vi en modstrid. Altså er $z_k = x_m = y_n$.

Antag Z_{k-1} ikke LCS af X_{m-1} og Y_{n-1}. Da Z_{k-1} er CS af X_{m-1} og Y_{n-1}, men ikke den længste pr. vores antagelse, så må der findes en anden delstreng W som er CS af X_{m-1} og Y_{n-1}, og som er længere hvorved $|W| > |Z_{k-1}| = k - 1$.
Men så kunne vi appende $x_m = y_n$ til W, og herved få en CS af X og Y med længden $|W| + 1 > k$. Nu har vi fundet en fælles delstreng af X og Y, som er længere end vores længste delstreng, hvilket er en modstrid.

2: Antag Z ikke LCS af X_{m-1} og Y. Z er stadig CS fordi vi antager $z_k \neq x_m$. Fordi vi antager det ikke er den længste delstreng, så findes der en delstreng W som er CS af X_{m-1} og Y og længere end Z, altså $|W| > |Z|$.
Men W er også CS af X og Y, og dermed længere end vores længste delstreng, hvilket er en modstrid.

3: Symmetrisk med 2.

- **Håndkørsel af `LCS`**
  - Ud fra vores Theorem kan vi se, at vi får følgende formel for hvad der skal være på plads $c[i, j]$, idet det angiver længden af en LCS for X_i og Y_j:

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

  Vi ser at det er rekursivt defineret, og løsningerne til visse delproblemer vil optræde flere gange. Derfor er det optimalt at løse med dynamisk programmering. Jeg vil nu give en håndkørsel af en implementering der er bottom-up:
  - Eksempel hvor $X = ABA$ og $Y = BA$:

		j	0	1	2
i				B	A
0			0	0	0
1	A		0	↑ 0	↖ 1
2	B		0	↖ 1	↑ 1
3	A		0	↑ 1	↖ 2

  - Vi får nu en køretid $O(nm)$ og et umiddelbart pladsforbrug $O(nm)$.

    Såfremt vi kun er interesserede i hvor lang den længste delstreng er, men ikke hvilke bogstaver den består af, kan vi forbedre pladsforbruget til $O\Big(\min(n, m)\Big)$. Det gøres ved at vælge den mindste delstreng til at være i toppen, og kun gemme den nuværende række samt rækken lige over.
  - Fordelene ved bottom-up, som vi bruger her, er at vi undgår rekursion og derfor ofte vil have lavere konstantled i forhold til top-down. Fordelen ved top-down er tilgengæld, at vi kun beregner de delproblemer som der faktisk er behov for, for at løse det oprindelige problem hvorimod bottom-up regner alle.

- **Bonusviden om `Cut-Rod` til eventuelle spørgsmål**

1. Der er et optimalt indeks i hvor vi kan cutte stangen op i to mindre delstange (delproblemer) - denne kan være evt. være 0:

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

2. Antag vi får det første cut der skal laves fra venstre side med længde i, så får vi:

$$r_n = p_i + r_{n-i}$$

3. Da får vi følgende formel:

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

4. Dette kan vi meget nemt implementere som en rekursiv algoritme, hvor vi så får en eksponentiel køretid. Ved at bruge DP og gemme resultaterne når vi har beregnet dem får en køretid på $O(n^2)$.

1 Grådige algoritmer

Paradigmet

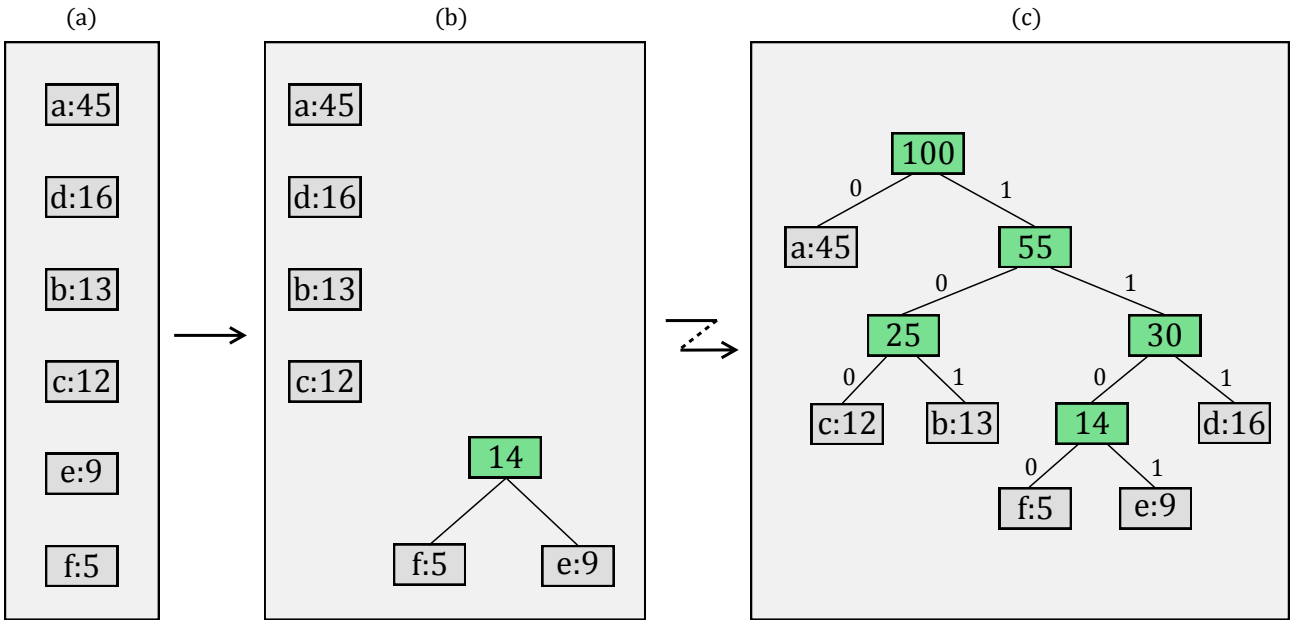
- Grådige algoritmer minder en smule om dynamisk programmering, i og med vi har en række delproblemer der afhænger af løsningen på andre delproblemer.
- Forskellen er, at med en grådig algoritme laver vi altid den løsning som på nuværende tidspunkt ser bedst ud, uden at kigge på fremtidige delproblemer.
- Hermed laver vi en lokal optimal løsning, som vi håber vil lede til en global optimal løsning.
- **Egenskaber for problemerne:**
  Greedy choice property:
  Vil sige at vi kan samle en global optimal løsning ved at lave lokale optimale (grådige) valg. Altså, når vi beregner hvilket valg vi skal tage, kan vi tage det valg som er bedst for det nuværende problem uden at tage højde for løsningerne til delproblemerne. Hermed får vi hele tiden et lignende, men mindre, delproblem.

  Optimal delstruktur:
  Vise sige, at hvis der fandtes en optimal løsning der indeholder det grådige valg, så består den optimale løsning af det grådige valg + en løsning til det delproblem vi har tilbage når vi har lavet det grådige valg.

- Eksempler på grådige algoritmer er activity-selection, Dijkstras algoritme og Huffman-koder. Jeg vil her gå mere i dybden med Huffman-koder.

Håndkørsel af `Huffman`

- Løser problemet med at vi gerne vil sende noget information, men ønsker at gøre brug af, at der er en hvis redundans i informationen. Ved at kode de karakterer der optræder flest gange med den laveste kode får vi den mindst mulige filstørrelse. Men hvordan finder vi ud af, hvordan vi skal kode dem, så vi overholder dette samtidig med at slutkodningen ikke kan tolkes på flere måder?
- Vi indfører følgende notation:
  - T er et parse-træ for en præfikskode.
  - C er sættet af karakterer, f.eks. $C = \{a, b, c, d, e, f\}$. For hver karakter $c \in C$ har vi:
  - f_c for frekvensen af denne karakter.
  - $d_T(c)$ for dybden af bladet i T.
  - $B(T) = \sum_{c \in C} f_c \cdot d_T(c)$ er omkostningen af træet. Vi ønsker naturligvis at minimere $B(T)$.
- Algoritmen virker ved at tage de to elementer med lavest frekvens og sætte dem sammen så de bliver søskende i et deltræ med en forælder der har en frekvens som er summen af dens børn. Dette gør algoritmen $n - 1$ gange, så alle knuder til sidst er sat sammen til et og samme deltræ.



Figur 1: **(a)** Algoritmen idet alle karakterer $c \in C$ er sat ind i vores datastruktur, men inden vi har kørt nogle iterationer.
(b) Algoritmen efter første iteration, hvor vi vælger f og e fordi de har lavest frekvens. De sættes sammen og der laves en forældreknude med frekvensen $f_f + f_e = 5 + 9 = 14$.
(c) Algoritmen efter alle $n - 1$ iterationer af for-loopet er kørt og vi har vores færdige parse-træ T. Nu ville bitrepræsentationen for f.eks. $a = 0$ og $f = 1100$.

- Køretiden afhænger af hvor hurtigt vi kan finde elementet med lavest frekvens og tage det ud, samt hvor hurtigt vi kan indsætte nye elementer. Hvis vi f.eks. bruger Fibonacci Heaps kan vi både køre `Extract-Min` og `Insert` i $O(\lg n)$ tid.
  Vi ser derudover, at for hver iteration fjerner vi to elementer og tilføjer et, altså er der netto ét element mindre at håndtere hver gang, hvorved vi får $\Theta(n)$ iterationer. Således bliver den totale køretid:
  $$O(n \lg n)$$

Bevis af korrekthed af `Huffman`
Vi har lige bestemt et parse-træ T ved at bruge Huffmans algoritme. Nu vil jeg gerne bevise, at $B(T)$ er minimal. Vi skal bruge et Lemma jeg ikke decideret vil bevise, men som giver god mening ud fra Figur 1.

Lemma: Hvis vi lader W være alle knuder pånær roden, så har vi for ethvert parse-træ T at:

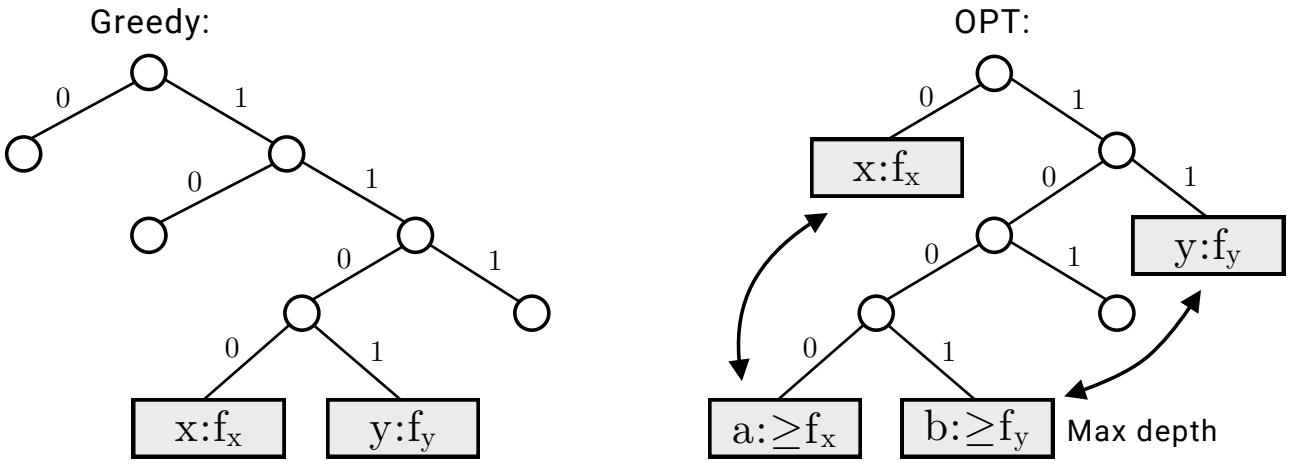
$$B(T) = \sum_{c \in C} f_c \cdot d_T(c) = \sum_{v \in W} f_v$$

Dvs. vores frekvens-dybde sum er lig summen af alle frekvenserne af knuder der ligger i W. Se f.ek.s på (c) i Figur 1.

For nu at bevise, at `Huffman` giver en optimal præfikskode skal vi vise greedy choice property og optimal delstruktur.

Greedy choice property:
Vi skal altså vise, at det grådige valg er med i mindst en af de optimale løsninger. Dvs. når vi finder de to symboler med lavest frekvens, sætter dem sammen, og giver dem en fælles forælder, så findes der en optimal løsning som har de her to knuder med mindst frekvens som søskendeblade i den optimale løsning.
Lad os kalde de to karakterer med lavest frekvens for x og y.

Vi har greedy choice property fra det faktum, at et optimalt træ OPT kan blive transformeret om til et andet træ OPT' som har det grådige valg (at x og y er søskendeblade):



Figur 2: Bemærk der med a og b i ulighederne i figuren menes frekvensen af de to, altså f_a og f_b

På Figur 2 har vi først outputtet fra `Huffman` (dvs. x og y er søskendeblade) og derefter et arbitrært optimalt træ OPT. Men OPT kan vi nu lave om til et nyt træ OPT' der også har det grådige valg. Det kan vi, fordi ulighederne på figuren gælder da vi netop har defineret x og y til at være de elementer med henholdsvis lavest og næstlavest frekvens.

Frekvensen af a er mindst lige så stor som frekvensen af x. Så ved at flytte a op kan vi få en kortere bitrepræsentation til noget der optræder lige så mange gange eller hyppigere end x. Her ses et formelt bevis, hvor $OPT = T$ er træet før ombytningen og $OPT' = T'$ er træet efter ombytningen af x og y:

$$\begin{aligned} B(T') &= B(T) - f_a \cdot d_T(a) - f_x \cdot d_T(x) \\ &\quad + f_a \cdot d_T(x) + f_x \cdot d_T(a) && (1) \\ &= B(T) + \underbrace{(f_a - f_x)}_{\geq 0}\underbrace{(d_T(x) - d_T(a))}_{\leq 0} && (2) \\ &\leq B(T) \implies T' \text{ er optimal} && (3) \end{aligned}$$

Eq. Eq. (1) Vi ved, at omkostningen af træet T' hvor x byttes med a svarer det til det gamle træ, og hvor vi så tager højde for de ændringer vi laver. Vi tager udgangspunkt i det optimale træ T i alle led.

Eq. Eq. (2) Vi sætter de to frekvenser sammen til et led samt de to dybder til et led. Det første led er ≥ 0 da det er vores antagelse jf. vores greedy choice property. Det andet led er ≤ 0, da vi valgte a til at have maksimal dybde.

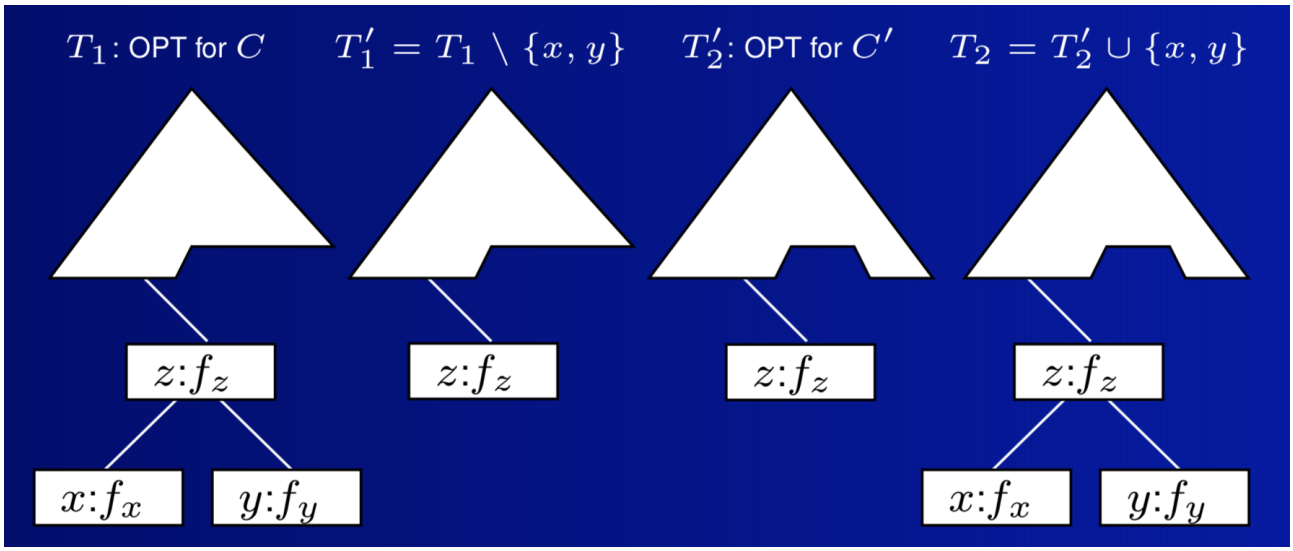
Eq. Eq. (3) Vi ser, at vi muligvis ganger et positivt med et negativt tal, og derfor må det led totalt blive ≤ 0. Således får vi $B(T') \leq B(T)$, som medfører at T' er optimal.

Altså har vi nu vist, at det grådige valg hvor x og y er søskendeblade er med i mindst en af de optimale løsninger.

Optimal delstruktur:
Det vi nu vil vise er, at hvis der fandtes en optimal løsning der indeholder det grådige valg, så består den optimale løsning af det grådige valg + en løsning til det delproblem vi har tilbage når vi har lavet det grådige valg.

Det delproblem vi har tilbage er et nyt alfabet $C' = (C \setminus \{x, y\}) \cup \{z\}$, altså hvor vi har fjernet x og y fra det oprindelige problem og indsat en ny "karakter" z med $f_z = f_x + f_y$.

Nu skal vi vise, at hvis det grådige valg er i et optimalt træ T_1 for C, så er $T_1' = T_1 \setminus \{x, y\}$ et optimalt træ for C'.
Derudover indfører vi T_2', som er en optimal løsning til vores problem med det reducerede alfabet C', og $T_2 = T_2' \cup \{x, y\}$ som er den optimale løsning med søskendebladene.



Vi ønsker at vise, at $B(T_1') \leq B(T_2')$, da T_2' netop er defineret til at være en optimal løsning for delproblem C'. Ved at vise dette viser vi nemlig, at delløsningen T_1' faktisk er en optimal løsning for C'.

Fra figuren ser vi:

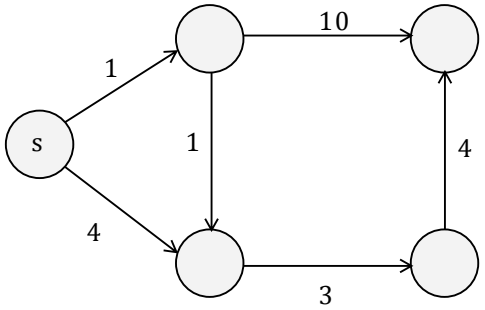
$$B(T_1') = B(T_1) - f_x - f_y \leq B(T_2) - f_x - f_y = B(T_2')$$

Lighederne gælder, da vi jf. vores Lemma ved at $B(T)$ er lig summen af frekvenserne af alle knuder pånær roden.

Uligheden gælder, da vi definerede at T_1 er en optimal løsning til C.

1 Minimum Spanning Trees

- **Motivation samt beskrivelse af problemet**
  - Løser det problem at forbinde en graf af knuder med den minimale omkostning.
  - Bliver ofte benyttet til f.eks. at designe elektriske kredsløb.
  - Kan modelleres som en graf $G = (V, E)$ hvor V er sættet af knuder og E er sættet af kanter. For hver kant $(u, v) \in E$ skal der være en vægtfunktion $w(u, v)$ som siger hvad omkostningen er.
  - Vi vil gerne finde det acyklise subset $T \subseteq E$ som forbinder alle knuderne, og minimerer den totale vægt.
  - Tegn en graf her hvor du beskriver terminologien ud fra, følgende er en god størrelse:



- **Generisk løsning**
  - Terminologi:
    - Cut: En partitionering $(S, V - S)$ af V i grafen $G = (V, E)$. Vil sige at vi deler grafen op i to subsets.
    - Krydse et cut: Når en kant (u, v) kryder cuttet $(S, V-S)$ er det fordi en af endepunkterne er i S og den anden er i $V - S$.
    - Respektere et sæt A af kanter: Når et cut ikke resulterer i, at nogle af kanterne i sættet A krydser cuttet.
    - Light kant: En kant, som både kryder et cut og hvis vægt er minimum af alle de kanter der kryder cuttet (NB: Der kan være flere for en iteration).
    - Sikker kant: En kant vi kan tilføje til subsettet A, som vedligeholder at A stadig er et subset af et Minimum Spanning Tree
  - Algoritme:

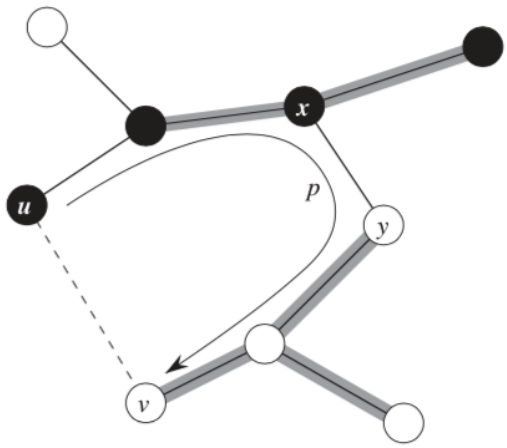
Algorithm 1: Generic-MST

```
1 Generic-MST(G, w)
2     A = ∅
3     while A ≠ et Spanning Tree
4         find en kant (u, v) sikker for A
5         A = A ∪ {(u, v)}
6     return A
```

  - Trivielt ser vi, at for hver iteration vil A være et subset af et MST da der kun tilføjes sikre kanter, og når algoritmen slutter er alle kanter tilføjet til A i et MST.

- **Bevis af Theorem 23.1 (Genkendelse af sikre kanter)**
  - *Def:* For grafen $G = (V, E)$ med vægtfunktion w defineret for E, så lad A være et subset af E som er inkluderet i et MST. Lad $(S, V - S)$ være ethvert cut af G som respekterer A, og lad (u, v) være en light kant der krydser dette cut.
    Så er kanten (u, v) sikker for A.

- Lad os nu sige at vi har et Minimum Spanning Tree T, som også inkluderer subsettet A, og antage at T ikke indeholder light kanten (u, v).
- *Mål:* Vi konstruerer nu et nyt MST T' som inkluderer $A \cup \{(u, v)\}$ ved at bruge en cut-and-paste teknik, hvorved vi viser at (u, v) er en sikker kant for A.



Figur 1: Sorte knuder i S, hvide knuder i $V - S$. Kanterne i MST'et T er vist, men ikke alle kanter i grafen G er vist. Kanterne i A er markeret og (u, v) er en light kant der krydser cuttet $(S, V - S)$.

- Vi laver nu et cut $(S, V - S)$ som respekter subsettet A.
- Kanten (u, v) danner en cyklus med kanterne på den simple vej p fra u til v i T, som set på Figur 1.
- Siden u og v er på modsatte sider af cuttet $(S, V - S)$, så må minimum en af kanterne i T være på den simple vej p og samtidig krydse cuttet. Lad os kalde den kant (x, y).
- Kanten (x, y) er ikke i A, siden cuttet respekterer A.
- Da vi har at (x, y) er en del af den unikke simple vej fra u til v, så vil det at fjerne den skære T over i to komponenter.
- Tilføjer vi nu (u, v) genforbinder vi komponenterne og får et nyt spanning tree

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

- Herefter skal vi vise at T' er et Minimum Spanning Tree. Siden (u, v) er en light kant der kryder $(S, V - S)$ og (x, y) også krydser dette cut, så får vi:

$$\begin{aligned} w(T') &= w(T) + \overbrace{w(u, v) - w(x, y)}^{\leq 0 \text{ da } w(u,v) \leq w(x,y)} \\ &\leq w(T) \end{aligned}$$

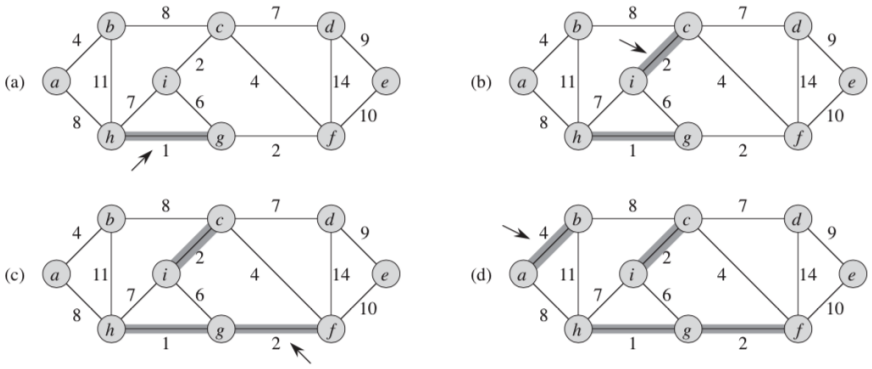
  Men vi havde før defineret T til at være et minimum spanning tree, så vi har også $w(T) \leq w(T')$, altså må T' også være et MSP.
- Vi skal stadig vise at (u, v) faktisk er en sikker kant for A.
  Da $A \subseteq T$ og $(x, y) \notin A$ må vi have at $A \subseteq T'$. Derfor må også $A \cup \{(u, v)\} \subseteq T'$.
  Siden T' er et minimum spanning tree må der gælde, at (u, v) er en sikker kant for A.

- **Håndkørsel af Kruskals algoritme**
  - Grådig algoritme *(Skriv ikke op, men for forståelse under forberedelse)*:

Algorithm 2: MST-Kruskal

```
1  MST-Kruskal(G, w)
2      A = ∅
3      foreach vertex v ∈ G.V
4          Make-Set(v)
5      sort the edges of G.E into increasing order by weight w
6      foreach edge (u, v) ∈ G.E in sorted order
7          if Find-Set(u) ≠ Find-Set(v)
8              A = A ∪ {(u, v)}
9              Union(u, v)
10     return A
```

  - Altså:
    Start med et tomt sæt A
    Lav et nyt sæt for hver knude
    Sorter alle kanter efter stigende vægt w
    For hver eneste kant i sorteret rækkefølge, hvis de to knuder forbundet af kanten ikke er en del af samme sæt, så tilføj kanten til A og foren de to knuder.
  - Illustration:



Figur 2: Kruskals algoritme *(fortsætter, kun første fire steps)*

  - Køretiden afhænger af hvor hurtigt vi kan tjekke om to elementer er en del af samme sæt samt forene dem. Vi bruger disjoint set forests.

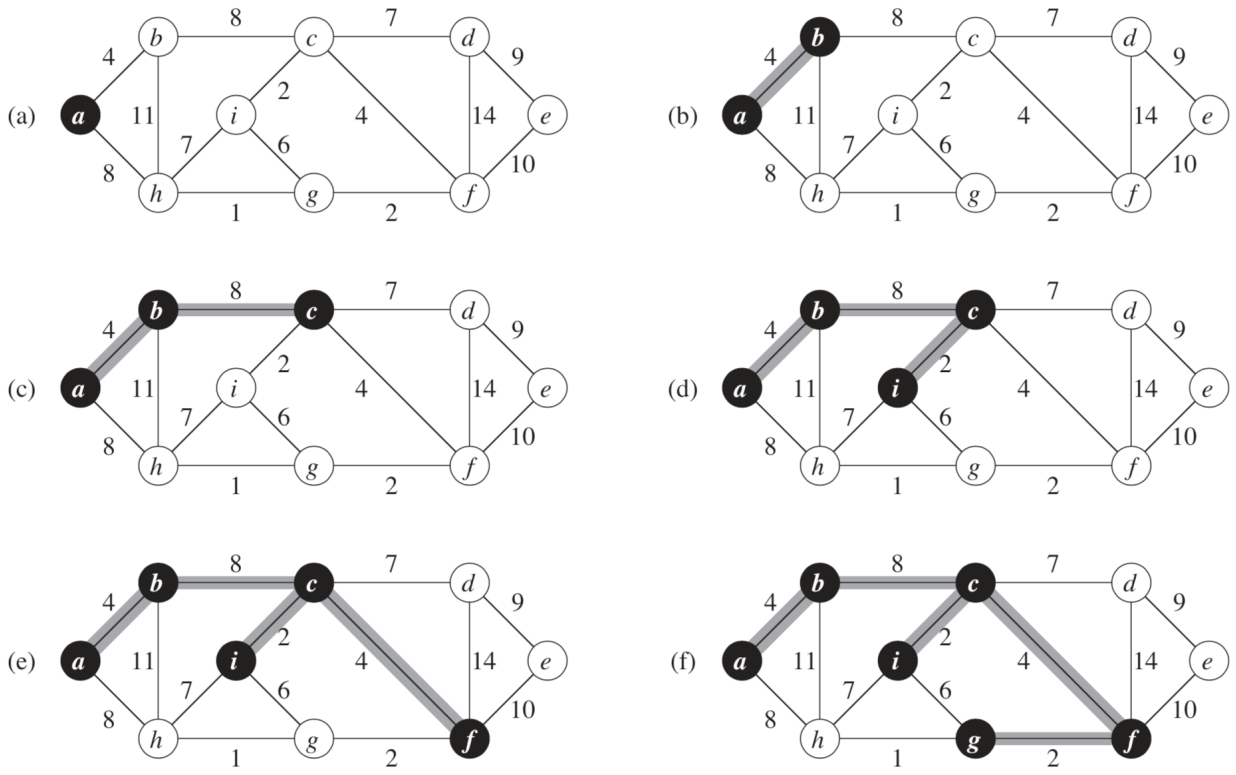
    Sortering af kanterne tager $O(E \lg E)$ tid. Efter det udfører vi $O(E)$ `Find-Set` og `Union`, som sammen med $|V|$ `Make-Set` operationer tager $O\Big((V + E)\ \alpha(V)\Big)$-tid.
    $\alpha(V)$ er en funktion der vokser *meget* langsomt

    Da vi antager at grafen er forbundet har vi $|E| \geq |V| - 1$, og derfor kan vi omskrive ovenstående til $O(E\alpha(V))$ tid.

    Vi har, at $\alpha(|V|) = O(\lg V) = O(\lg E)$, og derfor kan vi skrive køretiden fra det som $O(E \lg E) = O(E \lg V)$. Vi ser derfor, at algoritmen umiddelbart er upper-boundet af sorteringen. Da det er et MST vi skal finde, kan vi dog også antage, at det er en forbundet graf, hvorved $|E| \leq |V|^2$, så $\lg E = \lg V^2 = 2 \lg V = O(\lg V)$. Derfor kan vi skrive den endelige køretid som:

$$O(E \lg E) = O(E \lg V)$$

- **Håndkørsel af Prims algoritme**
  - Grådig algoritme
  - Vi starter ved en rodknude r og tilføjer herefter altid en light kant der krydser cuttet $(S, V - S)$. Har den effekt at vores subset A hele tiden er et sammenhængende træ.
  - Virker ved at holde styr på en $v.key$ og $v.\pi$ (parent-pointer) for alle knuder $v \in V$, som til at starte med sættes til henholdsvis ∞ og `NIL`. Herefter opdateres de omkringliggende kanter til den knude vi pt. kigger på såfremt værdierne er bedre.
  - Illustration:



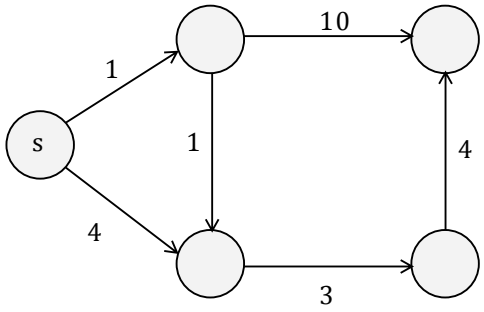
Figur 3: Prims algoritme *(fortsætter, kun første seks steps).*
Bemærk at vi i step (b)-(c) både kunne have valgt enten kanten (b, c) eller kanten (a, h).

  - Køretiden afhænger af hvor hurtigt vi kan køre `Extract-Min` og `Decrease-Key`. Derfor er det optimalt at bruge Fibonacci Heaps. Der er $|V|$ elementer at holde styr på, så hver `Extract-Min` operation vil tage $O(\lg n)$ amortiseret tid. Derudover udfører vi $2|E|$ `Decrease-Key` operationer, som hver amortiseret tager $O(1)$ tid. Altså vil den totale køretid blive:

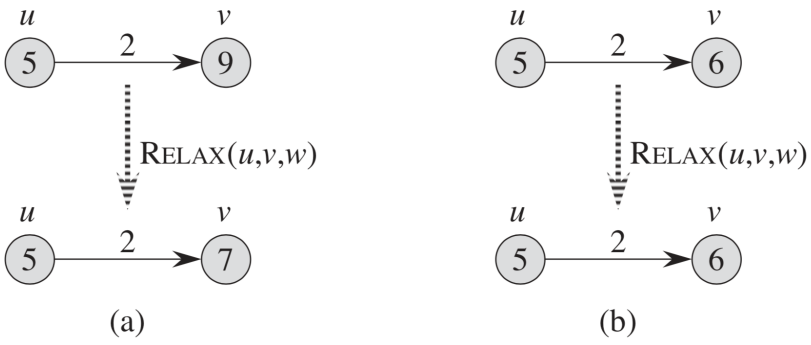
$$O(V \lg V + E)$$

1 Shortest Path

- **Motivation samt beskrivelse af problemet**
  - Åbenlyst utrolig brugbart. Kan bruges både bogstaveligt til korteste vej, men man kan også have en vægtfunktion der er baseret på tid, materialeomkostninger, etc.
    Jeg vil her tale om "Single Source Shortest Path", altså når vi ønsker at finde den korteste vej fra en kildeknude til alle andre knuder i grafen.
  - Givet en graf $G = (V, E)$ og en vægtfunktion $w(E)$, find da vejen p fra f.eks. knude u til v med den laveste samlede vægt $w(p)$ således at $w(p) = \delta(u, v)$.
  - Såfremt der ikke findes en vej fra u til v har vi defineret $\delta(u, v) = \infty$.
  - Tegn denne graf:



- **Init og Relaxation**
  - Vi indfører nu for alle knuder $v \in V$ en attribut $v.d$, som vi kan se som en upper bound for vægten af en kortest vej fra kildeknuden s til v. Denne kaldes "kortest-vej-estimat".
  - Derudover indfører vi attributten $v.\pi$ for alle knuder, som er dens forgænger (altså hvilken knude vi lige kom fra for at opnå den pt. korteste vej fundet).
  - `Init` sætter for alle knuder $v.d = \infty$ og $v.\pi = NIL$.
  - Vi indfører relaxtion for to knuder u og v der virker på den måde, at såfremt vi kan få et mindre "kortest-vej-estimat", så ændrer vi estimatet til dette og sætter v's forgænger til u på følgende måde:



Figur 1: Her relaxer vi kanten (u, v). I (a) kan vi forbedre "kortest-vej-estimateren" og opdaterer derfor værdierne for v, mens vi ikke ændrer noget i (b).

- **Bevis på at problemet udviser optimal delstruktur**
  - Lad $p = \langle v_0, v_1, ..., v_k \rangle$ være en kortest vej fra v_0 til v_k.
    For ethvert i og j således at

$$0 \leq i \leq j \leq k \qquad \text{lad} \qquad p_{ij} = \langle v_i, v_{i+1}, ..., v_j \rangle$$

være en delvej fra v_i til v_j. Så er p_{ij} en kortest vej fra v_i til v_j.

– *Bevis:* Lad os opløse vejen til

$$v_0 \overset{p_{0i}}{\rightsquigarrow} v_i \overset{p_{ij}}{\rightsquigarrow} v_j \overset{p_{jk}}{\rightsquigarrow} v_k$$

Så har vi, at

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

Lad os nu antage, at der findes en vej p'_{ij} fra v_i til v_j med en vægt $w(p'_{ij}) < w(p_{ij})$. Men så ville vi få, at

$$w(p) > w(p_{0i}) + w(p'_{ij}) + w(p_{jk})$$

hvilket er en modstrid, da vi antog at p var den korteste vej fra v_0 til v_k.

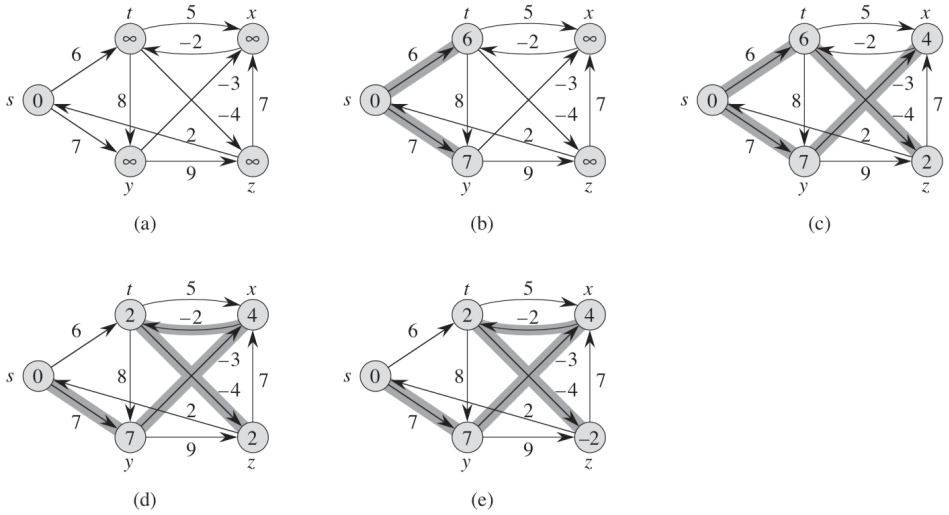
- **Håndkørsel af `Bellman-Ford`**

– Algoritme: *(Ikke skriv op, men for forståelse)*

Algorithm 1: MST-Kruskal

```
1 Bellman-Ford(G, w, s)
2     Initialize-Single-Source(G, s)
3     for i = 1 to |G.V| − 1
4         foreach edge (u, v) ∈ G.E
5             Relax(u, v, w)
6     foreach edge (u, v) ∈ G.E
7         if v.d > u.d + w(u, v)
8             return FALSE
9     return TRUE
```

Altså:
Initialiser alle vægte
Relax alle kanter $|V| - 1$ gange
Test for alle kanter (u, v) om $v.d > u.d + w(u, v)$ - hvis ja, returner `False`
Ellers returner `True`.



Figur 2: Illustration. Vær her opmærksom på, at kanterne relaxes i en bestemt rækkefølge og mellemresultaterne kunne se anderledes ud hvis de blev relaxet i en anden rækkefølge.

– Køretiden for Bellman-Ford er $O(VE)$, da det største bidrag kommer fra vi går igennem alle E kanter $V - 1$ gange.

- **Bevis af korrekthed af `Bellman-Ford`**
  - Tre dele: Path-relaxation property, Lemma 24.2, Theorem 24.4
  - Path-relaxation property
    * *Def:* Givet vi har en kortest vej $p = \langle v_0, v_1, ..., v_k \rangle$ fra $s = v_0$ til v_k. Så vil den sekvens af relaxation steps, som inkluderer i rækkefølge at relaxe

      $$(v_0, v_1), (v_1, v_2), ..., (v_{k-1}, v_k)$$

      medføre at $v_k.d = \delta(s, v_k)$. Dette gælder uanset om vi relaxer andre knuder indimellem.
    * *Bevis:* Induktionsbevis, hvor vi viser, at efter den i'te kant er relaxet har vi, at $v_i.d = \delta(s, v_i)$.
    * For vores basis $i = 0$ ser vi det er sandt, da $v_0.d = s.d = 0 = \delta(s, s)$.
    * For det induktive step antager vi, at $v_{i-1}.d = \delta(s, v_{i-1})$ og ser på hver der sker når vi relaxer kanten (v_{i-1}, v_i). Her har vi en konvergensegenskab jeg ikke vil bevise som da siger, at $v_i.d = \delta(s, v_i)$. Denne lighed er vedligeholdt for hele tiden fremover. Hermed er det bevist.
  - Lemma 24.2 ("Kortest-vej-estimatet" er korrekt ved terminering)
    * *Def:* Efter de $|V| - 1$ iterationer af for-loopet har vi, at $v.d = \delta(s, v)$ for alle knuder der kan nås fra kildeknuden (Vi antager at der ikke er nogle negative cykler).
    * *Bevis:* Tag udgangspunkt i enhver knude v som kan nås fra s, og lad $p = \langle v_0, v_1, ..., v_k \rangle$ hvor $v_0 = s$ og $v_k = v$ være en kortest vej fra s til v. Fordi korteste veje er simple, så har vi at p højest har $|V| - 1$ kanter, så:

      $$k \leq |V| - 1$$

      Vi relaxer netop alle kanter så mange gange, heriblandt kanten (v_{i-1}, v_i) i den i'te iteration. Ved at tage udgangspunkt i path-relaxation property har vi da ved terminering, at:

      $$v.d = \delta(s, v)$$
  - Theorem 24.4 (Korrekthed af `Bellman-Ford`)
    * *Def:* Hvis en graf G ikke har nogle negative cykler, så returnerer algoritmen `True` og har beregnet en korteste vej korrekt til alle knuder fra kildeknuden.
      Hvis G indeholder negative cykler, så returnerer algoritmen `False`.
    * *Bevis:* Først beviser vi det tilfælde, når G ikke har negative cykler. Da har vi lige bevist Lemma 24.2, og har altså derfor for alle knuder at $v.d = \delta(s, v)$. Vi har en "predecessor-subgraph property" som jeg ikke vil bevise, men af den følger, at så vil deres forgænger-attributer være sat korrekt og vi herved får den korteste vej. Hvis knuden ikke kan nås fra kildeknuden, så har vi en anden egenskab, "no path property", som viser at $v.d = \infty$ hvilket er korrekt jf. vores definition.
    * Herefter skal vi vise at den returnerer `True`.
      Når den terminerer har vi for alle knuder, at:

      $$\begin{aligned} v.d &= \delta(s, v) \\ &\leq \delta(s, u) + w(u, v) \qquad \text{(pga. trekantsuligheden)} \\ &= u.d + w(u, v) \end{aligned}$$

      Ingen af testene vil således returnere `False`, og derfor vil den returnere `True`.
    * Nu beviser vi, at den returnerer `False` når der er en negativ cyklus i G. Lad os kalde denne cyklus $c = \langle v_0, v_1, ..., v_k \rangle$, hvor $v_0 = v_k$. Så har vi at den totale vægt af c er mindre end 0 pr. definition.
      Antag nu for modstrid, at Bellman-Ford returnerer `True`. Altså vil

      $$v_i.d \leq v_{i-1}.d + w(v_{i-1}, v_i) \qquad \text{(for } i = 1, 2, ..., k)$$

Ved at summere ulighederne for cyklerne får vi:

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

$$= \sum_{i=1}^{k} v_{i-1}.d + \sum_{i=1}^{k} w(v_{i-1}, v_i) \tag{2}$$

Men da vi har at $v_0 = v_k$ vil hver knude i c indgå præcis én gang i både hvad der er på venstresiden i Eq. (1) og venstre led i Eq. (2), og disse to udtryk er derfor lig hinanden. Derved får vi, at

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

Men vi antog netop, at vores cyklus var negativ, hvilket er en modstrid. Hermed er det bevist at algoritmen korrekt vil returnere `False` såfremt der er en negativ cyklus.

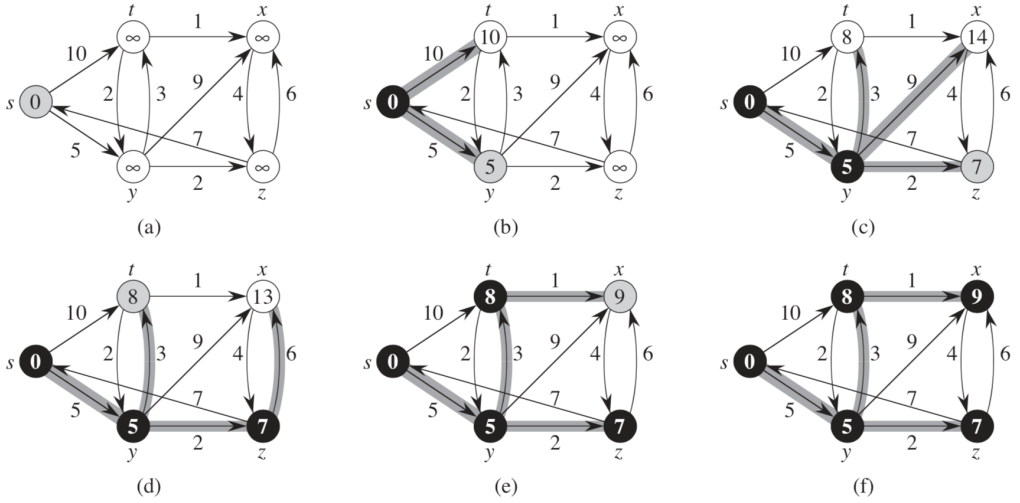
- **Håndkørsel af `Dijkstra`'s algoritme**
  - Antager alle kanter har en ikke-negativ vægt.
  - Algoritme: *(Ikke skriv op, men for forståelse)*

Algorithm 2: Dijkstra

```
1 Dijkstra(G, w, s)
2     Initialize-Single-Source(G, s)
3     S = ∅
4     Q = G.V
5     while Q ≠ ∅
6         u = Extract-Min(Q)
7         S = S ∪ {u}
8         foreach vertex v ∈ G.Adj[u]
9             Relax(u, v, w)
```

Altså:
Initialiser alle vægte
Løbende find knuden med lavest "kortest-vej-estimat" og relax alle dens kanter til omkringliggende knuder.

  - Illustration:



Figur 3: Dijkstras algoritme

– Køretiden afhænger af hvilken underliggende datastruktur vi bruger, hvor Fibonacci Heaps er mest optimalt. Vi udfører

  1. $|V|$ `Extract-Min` operationer som hver tager $O(\lg V)$ tid.
  2. $|E|$ `Decrease-Key` operationer (da vi kigger på alle kanter præcis én gang) som hver amortiseret tager $O(1)$ tid.

  Herved fås vi altså en samlet køretid:

$$O(V \lg V + E)$$

- **Korrekthed af `Dijkstra`'s algoritme**
  - Det vi skal vise for at Dijkstra er korrekt, så skal vi vise, at når man tilføjer en knude til løsningsmængden S, så er det fordi vi allerede har fundet den korteste vej. Vi skriver løkkeinvarianten:

    Claim: Når u tilføjes til S har vi at $u.d = \delta(s, u)$.

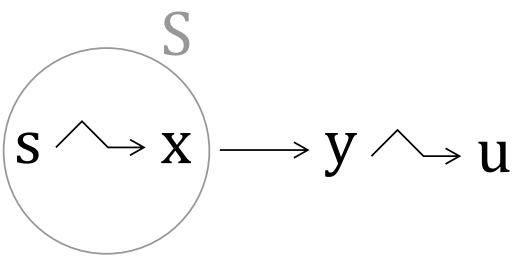
    Base case:

    $$s.d = \delta(s, s) = 0$$

    Induktionsstep:
    Vi bruger stærkt induktion, hvor vi antager at alle de knuder vi allerede har tilføjet til S er den korteste vej.
    Antag nu for modstrid at vi tilføjer knuden u, men $u.d \neq \delta(s, u)$ - hvorved vi får følgende pr. upper bound property:

    $$u.d > \delta(s, u)$$

    Der må findes en vej fra s til u, da hvis der ikke gjorde, så var den korteste vej uendelig pr. upper bound property (da hvis der ikke gjorde, så kunne den aldrig nogensinde blive uendelig pga. no path property).
    Når vi tilføjede den, så ville den altså allerede være lig den korteste vej. Så lad os sige at den korteste vej går gennem kanten (x, y). Vi definerer y til at være den første knude i stien fra s til u som ikke allerede er tilføjet til løsningsmængden S:

    Dvs. at x (som evt. godt kan være s) er i vores løsningssæt S, som pr. den stærke induktionsantagelse må have, at $x.d = \delta(s, x)$. Vi ved derudover, at da vi tilføjede x, så har vi relaxet alle udadgående kanter.
    Pr. convergence property betyder det, at $y.d = \delta(s, y)$. Derudover ved vi, at y kommer før u i den korteste vej til u. Da vi ikke har nogle negative kanter, så må den korteste vej til y må være mindre end eller lig den korteste vej fra s til u, hvilket er mindre end den distance vi har fundet til u pr. vores modstridsantagelse:

    $$y.d = \delta(s, y) \leq \delta(s, u) < u.d$$

    Vi har nu både y og u, og hvis de var ens, så havde vi allerede fundet den korteste vej til u når vi tilføjer den. Derfor må de være forskellige. Men vi har tænkt os at tilføje u til S, dvs.

    $$u.d \leq y.d$$

    (ellers ville vi være igang med at tilføje y pr. Dijkstras algoritme).
    Vi ser nu, at vi får følgende ulighed hvilket er en modstrid, og derfor er Dijkstra korrekt:

    $$y.d < u.d \leq y.d$$

- **Egenskaber**
  - **Trekantsulighed:** For enhver kant $(u, v) \in E$ har vi at $\delta(s, v) \leq \delta(s, u) + w(u, v)$.
  - **Upper-bound property:** Vi har altid at $v.d \geq \delta(s, v)$ for alle knuder $v \in V$, og når $v.d$ får værdien $\delta(s, v)$ ændres den aldrig.
  - **No-path property:** Hvis der ikke er nogen vej fra s til v, så har vi altid at $v.d = \delta(s, v) = \infty$.
  - **Konvergensegenskab:** Hvis $s \rightsquigarrow u \rightarrow v$ er en kortest vej for $u, v \in V$ og $u.d = \delta(s, u)$ på ethvert tidspunkt før vi relaxer kanten (u, v), så vil $v.d = \delta(s, v)$ efter vi relaxer og for altid fremover.
  - **Path-relaxation property:** Hvis $p = \langle v_0, v_1, ..., v_k \rangle$ er en kortest vej fra $s = v_0$ til v_k, og vi relaxer kanterne af p i rækkefølgen $(v_0, v_1), (v_1, v_2), ..., (v_{k-1}, v_k)$, så vil $v_k.d = \delta(s, v_k)$. Dette holder uanset hvad vi relaxer i mellemtiden.
  - **Predecessor subgrapgh-property:** Når $v.d = \delta(s, v)$ for alle $v \in V$, så er forgænger-delgrafen et kortest vej træ med rod i s.

1 Amortiseret analyse

- **Motivation**
  - Nogle gange er vi for en datastruktur ikke interesseret i dens worst case køretid hver eneste gang en operation udføres, men i stedet interesseret i hvad average case er når vi udfører en række af instruktioner.
  - Jeg vil her illustrere koncepterne på nogle lidt tænkte eksempler, men amortiseret analyse er også relevant i Fibonacci Heaps, og dermed i Dijkstras algoritme og Prims algoritme (MST), samt i disjoint-set forests hvilket benyttes i Kruskals algoritme (MST).
- **Aggregeret analyse**
  - Antag vi har en stack der har operationerne `Push(S, x)`, `Pop(S)`, `Stack-Empty(S)` som alle tager $\Theta(1)$-tid.
    Derudover har vi `MultiPop(S, k)`, som kalder `Pop` og `Stack-Empty` et antal gange der svarer til minimum af antal elementer og en parameter k vi giver den. Denne vil køre i $\Theta(\min(s, k))$-tid.
  - Hvis vi udfører n operationer på en stack der til at starte med er tom ser vi, at vi worst-case får en køretid på $O(n^2)$.
  - Vi beregner et upper bound $T(n)$ for alle n operationer og får derved den amortiserede cost pr. operation til $T(n)/n$.
  - Nu udnytter vi, at der højest kan være n `Push`-operationer og der kan ikke være flere `Pop`-operationer (inklusiv dem i `MultiPop`) end `Push`-operationer.
  - Så får vi at worst-case for alle n operationer er $O(n)$, og herved at den amortiserede cost pr. operaton er $O(n)/n = O(1)$.
- **Accounting method**
  - Den rigtige cost for den i'te operation er c_i
    Den amortiserede cost for den i'te operation er $\hat{c}_i$.
  - Hvis $\hat{c}_i > c_i$, så overcharger vi operationen og har på den måde noget "kredit" at bruge af. Hvis det omvendte gør sig gældende, så undercharger vi og bruger herved noget af den kredit vi har bygget op.
  - Der skal ALTID gælde:
    $$\sum_{i=1}^{n} c_i \leq \sum_{i=1}^{n} \hat{c}_i$$
    Hvis vi nu kan få et upper bound på $\sum_{i=1}^{n} \hat{c}_i$ får vi også et upper bound på den faktiske omkostning.
  - Nu henholdsvis får/vælger vi følgende værdier:

Operation	**Faktisk cost**	**Amortiseret cost**
`Push`	1	2
`Pop`	1	0
`MultiPop`	$\min(s, k)$	0

    Vi kan se på det på den måde, at ved `Push` betaler vi både for selve operationen med 1 samt lægger 1 kredit på elementet.
  - `Pop` bliver undercharged med 1, men denne cost betaler vi med den kredit vi har lagt på elementet. Vi ser at vi aldrig får en kredit der er negativ.

- Vi ser, at for enhver sekvens af n operationer har vi

$$\sum_{i=1}^{n} c_i \leq \sum_{i=1}^{n} \hat{c}_i \leq 2n$$

og herved bliver den gennemsnitlige cost ≤ 2, hvorved vi får en gennemsnitlig køretid på $O(1)$.

- **Potentialemetoden - Generelt**
  - Minder lidt om accounting method bortset fra at kreditten ikke er gemt på enkelte elementer men i stedet i "banken". Mængden af kredit i banken er udtrykt ved en potentialefunktion Φ.
  - Hvis vi udfører n operationer på en datastruktur hvor D_i er strukturen efter den i'te operation for $i = 1, ..., n$, så skriver vi $\Phi(D_i)$ som symbol for kreditten der er gemt med den nuværende struktur.
  - Vi definerer den amortiserede cost $\hat{c}_i$ som den faktiske cost c_i plus forskellen mellem hvad der er i potentialet nu og hvad der var lige før.

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

Hvis $\Phi(D_i) - \Phi(D_{i-1}) > 0$ kan man sige at vi putter kredit i banken, og hvis det er mindre tager vi kredit fra banken.

  - Samme egenskab som for accounting method skal gælde:

$$\sum_{i=1}^{n} \hat{c}_i \geq \sum_{i=1}^{n} c_i \tag{1}$$

  - Vi får summen af den amortiserede cost til at blive:

$$\begin{aligned}
\sum_{i=1}^{n} \hat{c}_i &= \sum_{i=1}^{n} \Big(c_i + \Phi(D_i) - \Phi(D_{i-1}) \Big) \\
&= c_1 + \Phi(D_1) - \Phi(D_0) \\
&+ c_2 + \Phi(D_2) - \Phi(D_1) \\
&+ \vdots \\
&+ c_n + \Phi(D_n) - \Phi(D_{n-1}) \\
&= \left(\sum_{i=1}^{n} c_i \right) + \Phi(D_n) - \Phi(D_0)
\end{aligned}$$

Da vi ser at alle potentiale-led pånær det første og sidste går ud med hinanden i summen. Hvis vi sørger for at $\Phi(D_n) \geq \Phi(D_0)$ ser vi, at vi vil opfylde Eq. (1).

- **Potentialemetoden - Stack-eksempel**
  - Vi vælger en potentialefunktion $\Phi(D_i) =$ antallet af elementer på stacken.
    Vi ser tydeligt at Eq. (1) er overholdt, da $D_0 = 0$ og $\Phi(D_i) \geq 0$ for alle i.
  - Nu skal vi upper bound'e dette.
  - `Push`:
    - $\Phi(D_i) - \Phi(D_{i-1}) = 1$ (der tilføjes et element til stacken)
    - Derved fås $\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) = 1 + 1 = 2$
  - `Pop`:
    - $\Phi(D_i) - \Phi(D_{i-1}) = -1$ (der fjernes et element fra stacken)
    - Derved fås $\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) = 1 - 1 = 0$

- `MultiPop`:
  - Lad os sige vi popper $k' > 0$ elementer.
  - $\Phi(D_i) - \Phi(D_{i-1}) = -k'$ (da der fjernes k' elementer fra stacken)
  - Derved fås $\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) = k' - k' = 0$
- For alle operationer $i = 1, ..., n$ gælder, at $\hat{c}_i \leq 2$. Herved får vi følgende ulighed:

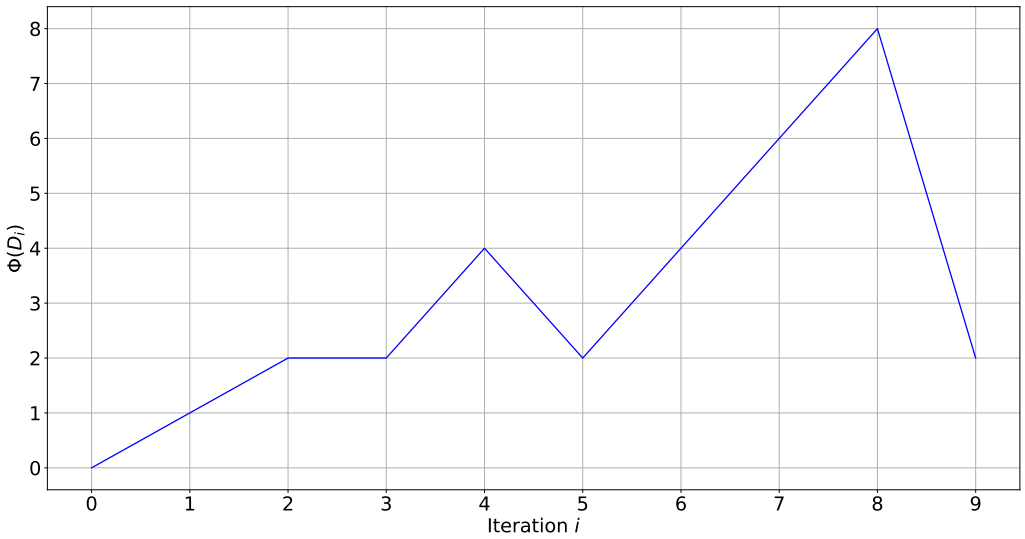
$$\sum_{i=1}^{n} c_i \leq \sum_{i=1}^{n} \hat{c}_i \leq 2n$$

  Og derved er den gennemsnitlige tid brugt pr. operation $2n/n = O(1)$.

- **Potentialemetoden - Dynamisk tabel kun med `Insert`**
  - Nu indfører vi en abstrakt datastruktur T, som vi kan tænke på som en tabel. Den understøtter `Insert` og `Delete`, bruger et array til at gemme sin data og bruger $O(k)$ tid på at allokere/free'e et array af størrelse k. For nu antager vi, at T kun understøtter `Insert`.
  - Vi ønsker at T dynamisk allokerer et nyt større array til sig selv når det gamle array er for småt til at indeholde alle elementerne.
  - Indfør følgende notation:
    - num_i: Antallet af elementer i T efter den i'te operation
    - size_i: Størrelsen af arrayet for T efter den i'te operation
    - $\alpha_i = \text{num}_i/\text{size}_i$, loadfaktoren af T efter den i'te operation (hvis $\text{size}_i = 0$, definer $\alpha_i = 1$).
  - Vi starter med at T er tom. Herefter siger vi, at lige før vi indsætter det i'te element, hvis $\text{num}_{i-1} = \text{size}_{i-1}$ (svarende til loadfaktoren $\alpha_{i-1} = 1$) så ekspanderer vi T til et array der er dobbelt så stort, 2size_{i-1} og kopierer de gamle elementer over.
  - Vi ser at tabelekspandering tager $O(\text{num}_i)$ worst-case tid. Så hvis den i'te operation kræver en ekspandering kan vi sætte $c_i = \text{num}_i$.
  - Hvis ingen ekspandering er krævet kan vi sætte $c_i = 1$.

  - Definer potentialefunktionen

$$\Phi(D_i) = 2\text{num}_i - \text{size}_i$$



Figur 1: Potentialet $\Phi(D_i)$ efter hver iteration i

  - Først vis den er valid ved at vise $\Phi(D_0) = 0$ og $\Phi(D_i) \geq 0$ for alle i.

- Tydeligvis er $\Phi(D_0) = 0$ da T er tom til at starte med. Og da vi har, at T altid som minimum er halvt fyld, så har vi også $\Phi(D_i) \geq 0$ for alle i. Derfor gælder Eq. (1).
- Da Eq. (1) gælder kan vi få et upper bound for $\sum_{i=1}^{n} c_i$ ved at upper bound'e $\sum_{i=1}^{n} \hat{c}_i$.
- Hvis der IKKE sker tabelekspandering får vi følgende amortiserede cost:

$$\begin{aligned}\hat{c}_i &= c_i + \Phi(D_i) - \Phi(D_{i-1}) \\ &= 1 + (2\text{num}_i - \text{size}_i) - (2\text{num}_{i-1} - \text{size}_{i-1}) \\ &= 1 + (2\text{num}_i - \text{size}_i) - (2(\text{num}_i - 1) - \text{size}_i) \\ &= 3\end{aligned}$$

  Vi får dette da size ikke ændrer sig, og det forrige antal elementer svarer til det nuværende antal elementer minus en, da vi jo netop har tilføjet en. 3-tallet fås ved at se, at ting går ud med hinanden.
- Hvis der SKER tabelekspandering OG $i = 1$ får vi den amortiserede cost 2.
- Hvis der SKER tabelekspandering OG $i > 1$ får vi:

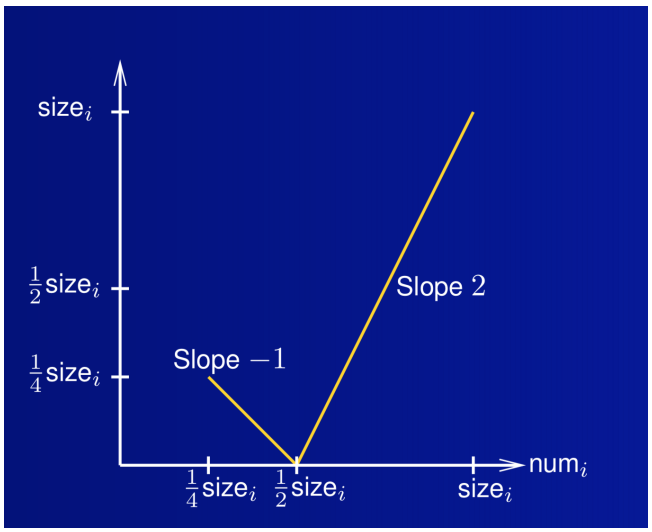
$$\begin{aligned}\hat{c}_i &= c_i\Phi(D_i) - \Phi(D_{i-1}) \\ &= \text{num}_i + (2\text{num}_i - \text{size}_i) - (2\text{num}_{i-1} - \text{size}_{i-1}) \\ &= \text{num}_i + \Big(2\text{num}_i - 2(\text{num}_i - 1)\Big) - \Big(2(\text{num}_i - 1) - (\text{num}_i - 1)\Big) \\ &= 3\end{aligned}$$

  Hvordan disse numre fås kan ræsonneres ud fra grafen, bare husk at alle variabler skal være num_i *til sidst.*

- **Potentialemetoden - High-level bevis når vi har `Insert`/`Delete`**
  - Forklaring af `Delete`: Vi ønsker at kunne fjerne elementer fra tabellen igen, og såfremt vi når ned under en hvis loadfaktor α_i at bruge et mindre array til at holde dem. Naiv implementation vil være at gå til mindre array når $\alpha_i < 1/2$, men så kunne vi potentielt få et problem hvis vi indsætter og sletter mange gange meget tæt på en 2-tals potens.
  - I stedet vælger vi at gøre det når vi når ned under en loadfaktor $\alpha_i < 1/4$, og så kan man faktisk vise at vi understøtter `Insert` og `Delete` i $O(1)$ amortiseret tid.
  - Vi vælger nu potentialemetoden:

$$\Phi(D_i) = \begin{cases} 2\text{num}_i - \text{size}_i & \text{if } \alpha_i \geq 1/2 \\ \text{size}_i/2 - \text{num}_i & \text{if } \alpha_i < 1/2 \end{cases} \tag{2}$$

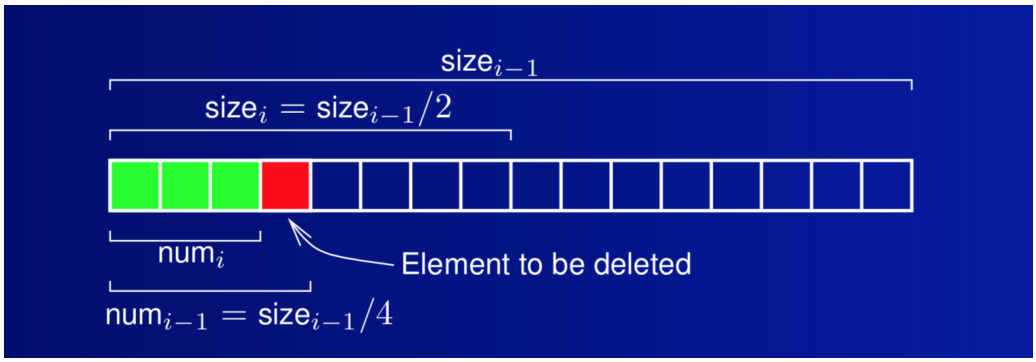
    Da får vi følgende graf:



Figur 2: Hvordan grafen vokser

- Vi observerer, at lige før en tabel ekspandering/forkortelse har Φ værdien num_i. Lige efter er tabellen cirka halv fuld og derfor dropper potentialet til omkring 0. Dette drop i potentiale "betaler" for ekspanderingen/forkortelsen.
- Når der ikke sker en ekspandering/forkortelse, så er den amortiserede cost højest 3 siden den absolutte hældning af Φ højest er 2.

- **Potentialemetoden - Bevis for amortiseret cost af `Delete`**
  - Hvis der ikke sker nogen tabelforkortelse, så kan vi ud fra grafen argumentere for, at $\hat{c}_i \leq 2$ da $c_i = 1$.
  - Hvis der SKER en tabelforkortelse, så vil ét element slettes og num_i elementer flyttes til det forkortede array, så den faktiske cost er $c_i = \text{num}_i + 1$. Vi har at loadfaktoren vil være under $1/2$ for både før og efter operationen, så vi bruger case 2 i Eq. (2) for begge.
  - Da får vi:



$$\begin{aligned}
\hat{c}_i &= c_i + \Phi(D_i) - \Phi(D_{i-1}) \\
&= \text{num}_i + 1 + (\text{size}_i/2 - \text{num}_i) - (\text{size}_{i-1}/2 - \text{num}_{i-1}) \\
&= \text{num}_i + 1 + ((\text{num}_i + 1) - \text{num}_i) - (2(\text{num}_i + 1) - (\text{num}_i + 1)) \\
&= 1
\end{aligned}$$

Husk igen på at vi skal have num_i *som eneste variabel, og ræsonner da ud fra figuren.*