Institut for Matematik og Datalogi
Syddansk Universitet, Odense

21. maj 2020

DM507 — Algoritmer og Datastrukturer

Eksaminatorie-timer uge 20, Forår 2020

Instruktorerne for DM507

Indhold

1 Eksaminatorie-timer I

1.1 Eksamen juni 2014 opgave 1

Angiv løsningerne til følgende rekursionsligninger

i) $T(n) = 2 \cdot T(\frac{n}{3}) + n$: Observer rekursionsligningen er på formen $T(n) = a \cdot T(\frac{n}{b}) + f(n)$ og dermed kan Master Theorem potentielt benyttes.

Her er $a = 2$, $b = 3$ og $f(n) = n$. Da $\alpha = \log_3 2 = 0.6309\ldots$ og

$$f(n) = n = \Omega(n^{\alpha+\varepsilon}) = \Omega(n^{0.6309\cdots+\varepsilon}),$$

for $\epsilon = 0.1$, så kan Case 3 af Master Threorem benyttes, hvis der findes et $c < 1$ og et n_0 sådan at $a \cdot f\left(\frac{n}{b}\right) \leq c \cdot f(n)$, når $n \geq n_0$. Eftersom

$$\begin{aligned} a \cdot f(\frac{n}{b}) &\leq c \cdot f(n) \Leftrightarrow \\ 2 \cdot \frac{n}{3} &\leq c \cdot n \Leftrightarrow \\ \frac{2}{3} &\leq c \end{aligned}$$

er uligheden opfyldt for f.eks. $c = \frac{2}{3}$ og $n_0 = 1$. Altså, Case 3 af Master Theorem kan benyttes. Løsningen er $T(n) = \Theta(f(n)) = \Theta(n)$.

ii) $T(n) = 32 \cdot T(\frac{n}{4}) + n^{2.5}$: Rekursionsligningen er igen på den rigtige form.

Her er $a = 32$, $b = 4$ og $f(n) = n^{2.5}$. Da $\alpha = \log_{32} 4 = 2.5$ og

$$f(n) = n^{2.5} = \Theta(n^{\alpha}) = \Theta(n^{2.5}),$$

så kan Case 2 af Master Theorem benyttes. Løsningen er $T(n) = \Theta(n^{\alpha} \cdot \log n) = \Theta(n^{2.5} \cdot \log n)$.

1.2 Eksamen juni 2014 opgave 2

I denne opgave skal vi afgøre om nogle udsagn er sande eller falske. Inden vi begynder på selve opgaven skal man huske tilbage til definitionerne af O, Θ og ω. Disse kan findes i kapitel 3.1 i Cormen et al.[1] eller på slides 10-15 i [2]. Alle følgende referencer til slides referer til [2].

1) n^2 **er** $O(n^2)$

Som man kan se på slide 11 i [2] for at $f(n) = O(g(n))$ så skal $f \leq g$ i voksehastighed. Her har vi at $f = g$, da $n^2 = n^2$ og derfor må dette udsagn være sandt.

2) n^2 **er** $\Theta(n^2)$

Som man kan se på slide 13 i i [2] for at $f(n) = \Theta(g(n))$ så skal $f = g$ i voksehastighed. Her har vi at $f = g$, da $n^2 = n^2$ og derfor må dette udsagn også være sandt.

3) n^4 **er** $O(5n^3 + 3n^5)$

For at løse den her opgave kan vi bruge metoden på slide 17 for at vurdere om $n^4 = O(5n^3 + 3n^5)$. Det vil sige, vi vil gerne kigge på resultatet af $\frac{n^4}{5n^3 + 3n^5}$ når $n \to \infty$. Her kan vi se at n^5 vokser hurtigere end n^4 og resultatet vil derfor gå mod 0. Ud fra sætning 2 på slide 17 kan vi derfor sige at $n^4 = o(5n^3 + 3n^5)$. Når en funktion f er lille o af anden funktion g, så er f også store O af g som man kan læse på slide 16. Derfor må dette udsagn være sandt.

4) n^4 **er** $\Theta(5n^3 + 3n^5)$

Her kan vi bruge resultatet fra 3), hvor $\frac{n^4}{5n^3 + 3n^5} \to 0$ når $n \to \infty$. Det vil altså sige at $n^4 < 5n^3 + 3n^5$ i voksehastighed og da funktionernes voksehastighed skal være lig med hinanden for at en funktion er Θ af en anden, må dette udsagn være falskt.

5) $n \log n$ **er** $O(n^{1.5})$

Lad os bruge samme metode som i 3), dvs. kigge på resultatet af følgende:

$$\frac{n \log n}{n^{1.5}} \text{ når } n \to \infty$$

Her kan vi omskrive brøken på følgende måde:

$$\frac{n \log n}{n^{1.5}} = \frac{n \log n}{n^1 \cdot n^{0.5}} = \frac{\log n}{n^{0.5}}$$

Vi kan bruge at en hver logaritme er lille o af et hvert polynomium (fra slide 19) til at sige at dette går mod 0. Ud fra sætning 2 på slide 17 kan vi derfor sige at $n \log n = o(n^{1.5})$. Når en funktion f er lille o af anden funktion g, så er f også store O af g som man kan læse på slide 16. Derfor må dette udsagn være sandt.

6) n **er** $O(\log n)$

Da enhver logaritme er lille o af ethvert polynomium må dette udsagn være falskt.

7) $(\log n)^{10}$ **er** $O(n^{0.10})$

Her kan vi igen bruge sætningen på slide 19 om at en hver logaritme (selv opløftet i enhver potens) er lille o af et hvert polynomium sammen med at når funktion f er lille o af anden funktion g, så er f også store O af g. Derfor må dette udsagn være korrekt.

8) 1 **er** $O(n)$

Man kan argumentere for at dette er sandt på mange måder. Hvis vi kigger på funktionerne kan vi se at $1 \leq n$ når $n \geq 1$, hvilket er definitionen på $f(n) = O(g(n))$. Dette udtryk må derfor være sandt.

9) n^2 **er** $O(n^3)$

Her kunne vi bruge metoden med grænseværdier til at vise at dette er sandt. Alternativt kan vi kigge på slide 20, hvor vi i en tidligere opgave har vist at disse er skrevet op efter stigende asymptotisk voksehastighed.

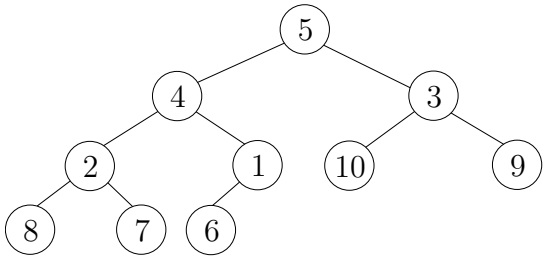
10) n^3 **er** $\omega(n^3)$

For at afgøre dette udtryk kan vi bruge den øverste sætning på slide 17, dvs. kigge på grænseværdien for $\frac{n^3}{n^3}$ når $n \to \infty$. Da vi dividerer to identiske funktioner ville dette altid være 1, og dermed en konstant. Derfor er $n^3 = \Theta(n^3)$. Hvilket vil sige, at det her udsagn må være falskt.

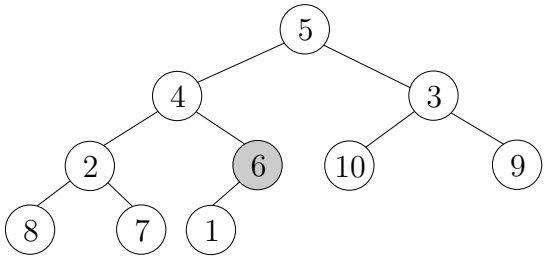
1.3 Eksamen juni 2014 opgave 3

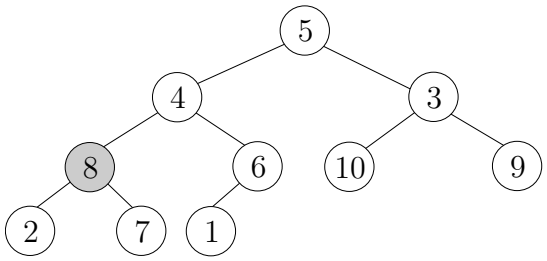
Vi skal udføre Build-Max-Heap på det givne array. Man kan godt udføre Build-Max-Heap ved kun at se på arrayet, men intuitivt er det lidt nemmere at se på det som et binært træ (dog husk, at i praksis arbejder vi stadig kun i arrayet, og laver ikke en seperat træ-struktur ved siden af).

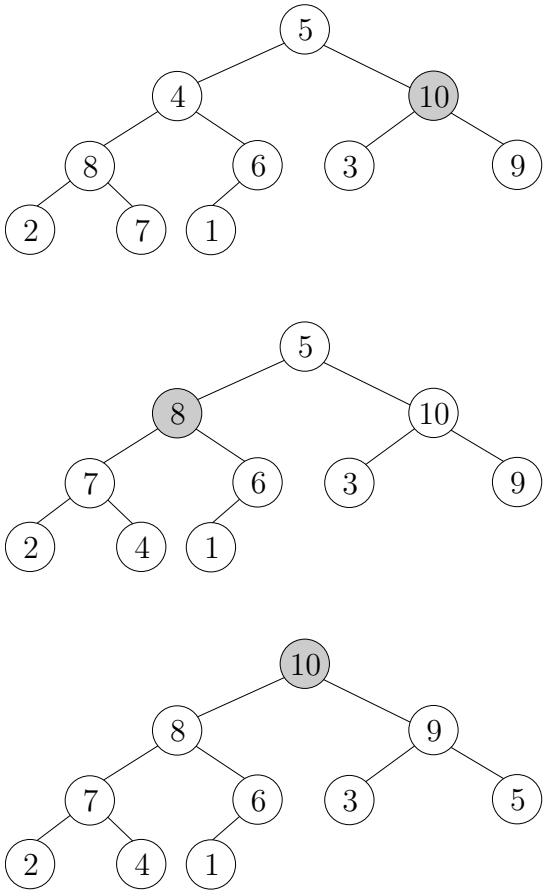
Arrayet svarer til det følgende binære træ:



I Build-Max-Heap kører vi i princippet bare Max-Heapify på alle indre knuder i træet, startende fra det største indeks. I de følgende figurer er dette blevet gjort, hvor den grå knude indikerer den knude der originalt blev kaldt Max-Heapify på:







Og vi kan se, at det sidste træ har heap-facon og er min-heapordnet.

Til eksamen skulle svaret have været givet som: $10, 8, 9, 7, 6, 3, 5, 2, 4, 1$

1.4 Eksamen juni 2014 opgave 4

I denne opgave bliver vi givet hash-tabellen i figur 1. Vi får også at vide at vi skal indsætte værdierne 3, 5 og 15 (i den rækkefølge). Vi skal indsætte ved brug af double hashing med auxilary-funktionerne:

$$\begin{aligned} h_1(x) =& (5 \cdot x + 1) \mod 13 \\ h_2(x) =& 1 + (x \mod 12) \end{aligned}$$

Vi skal altså indsætte ved brug af:

$$h(x, i) = (h_1(x) + i \cdot h_2(x)) \mod m$$

Her er $m = 13$ da det er antallet af pladser i tabellen. Information omkring double hashing kan findes i [3, slide 37].

indeks:	0	1	2	3	4	5	6	7	8	9	10	11	12
H:	18		8			6			30	25		2	23

Tabel 1: Input

Indsæt 3:

$$\begin{aligned} h(3,0) =& (((5 \cdot 3 + 1) \mod 13) + 0 \cdot (1 + (3 \mod 12))) \mod 13 \\ =& (3 + 0 \cdot 4) \mod 13 \\ =& 3 \end{aligned}$$

Der er plads i tabellen på indeks 3 så vi indsætter 3 her.

index:	0	1	2	3	4	5	6	7	8	9	10	11	12
H:	18		8	3		6			30	25		2	23

Tabel 2: Indsætning af 3

Indsæt 5:

$$\begin{aligned} h(5,0) =& (((5 \cdot 5 + 1) \mod 13) + 0 \cdot (1 + (5 \mod 12))) \mod 13 \\ =& (0 + 0 \cdot 6) \mod 13 \\ =& 0 \end{aligned}$$

Der er ikke plads i tabellen på plads 0 så vi udregner igen med $i = 1$

$$\begin{aligned} h(5,1) =& (((5 \cdot 5 + 1) \mod 13) + 1 \cdot (1 + (5 \mod 12))) \mod 13 \\ =& (0 + 1 \cdot 6) \mod 13 \\ =& 6 \end{aligned}$$

Der er plads i tabellen på indeks 6 så vi indsætter 5 her.

index:	0	1	2	3	4	5	6	7	8	9	10	11	12
H:	18		8	3		6	5		30	25		2	23

Tabel 3: Indsætning af 5

Indsæt 15:

$$\begin{aligned} h(5,0) =& (((5 \cdot 15 + 1) \mod 13) + 0 \cdot (1 + (15 \mod 12))) \mod 13 \\ =& (11 + 0 \cdot 4) \mod 13 \\ =& 11 \end{aligned}$$

Der er ikke plads i tabellen på plads 11 så vi udregner igen med $i = 1$

$$\begin{aligned} h(15,1) =& (((5 \cdot 15 + 1) \mod 13) + 1 \cdot (1 + (15 \mod 12))) \mod 13 \\ =& (11 + 1 \cdot 4) \mod 13 \\ =& 2 \end{aligned}$$

Der er ikke plads i tabellen på plads 2 så vi udregner igen med $i = 2$

$$\begin{aligned} h(15,2) =& (((5 \cdot 15 + 1) \mod 13) + 2 \cdot (1 + (15 \mod 12))) \mod 13 \\ =& (11 + 2 \cdot 4) \mod 13 \\ =& 6 \end{aligned}$$

Der er ikke plads i tabellen på plads 6 så vi udregner igen med $i = 3$

$$\begin{aligned} h(15,3) =& (((5 \cdot 15 + 1) \mod 13) + 3 \cdot (1 + (15 \mod 12))) \mod 13 \\ =& (11 + 3 \cdot 4) \mod 13 \\ =& 10 \end{aligned}$$

Der er plads i tabellen på indeks 10 så vi indsætter 15 her.

index:	0	1	2	3	4	5	6	7	8	9	10	11	12
H:	18		8	3		6	5		30	25	15	2	23

Tabel 4: Indsætning af 15

Resultatet der skal gives ifølge eksamens-opgaven er: $18, x, 8, 3, x, 6, 5, x, 30, 25, 15, 2, 23$

1.5 Eksamen juni 2014 opgave 5

De første tre iterationer af Radix-sort på følgende array skal vises.

	1	2	3	4	5	6	7	8
A:	8345	7112	1830	5001	4345	2222	9112	6363

Hver iteration af Radix-sort sorterer efter et ciffer af elementerne. Først det første ciffer (mindst betydende), så det andet ciffer (anden-mindst betydende) osv. Sidste iteration sorterer på det sidste ciffer (mest betydende). Sortereringen i hver iteration skal være stabil[1].

Første iteration (sortering på første digit):

	1	2	3	4	5	6	7	8
A:	1830	5001	7112	2222	9112	6363	8345	4345

Anden iteration (sortering på anden digit):

	1	2	3	4	5	6	7	8
A:	5001	7112	9112	2222	1830	8345	4345	6363

Tredje iteration (sortering på tredje digit):

	1	2	3	4	5	6	7	8
A:	5001	7112	9112	2222	8345	4345	6363	1830

[1]Stabil sortering betyder, at hvis nøglerne af to elementer er det samme (i dette tilfælde to cifre er det samme), så vil den af de to elementer, som kom først i input også komme først i output.

1.6 Eksamen juni 2014 opgave 6

I denne opgave er vi givet en fil, som indeholder nedenstående tegn og tilhørende hyppigheder:

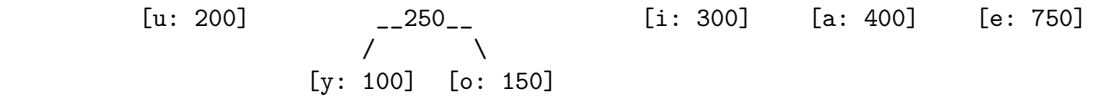
Tegn	a	e	i	o	u	y
Hyppighed	400	750	300	150	200	100

Tabel 5: De givne tegn i filen og deres tilhørende hyppigheder

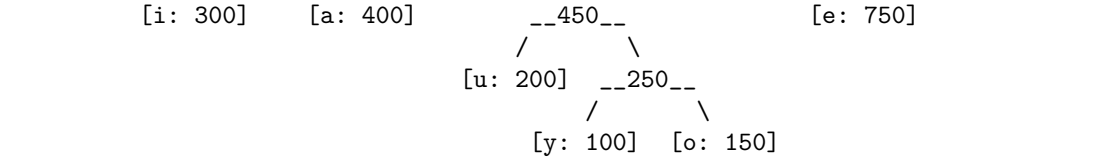
Et muligt Huffman-træ for denne fil konstrueres ved følgende skridt:

1. Skridt: Start med alle tegn som værende blade i Huffman-træet

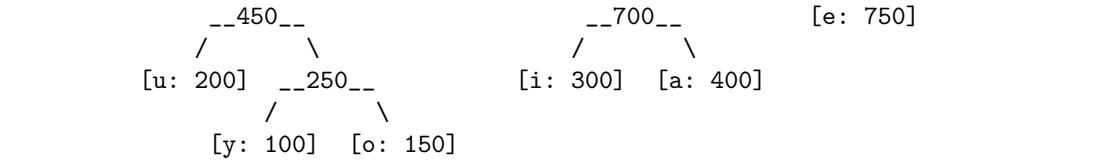
2. Skridt: Sammenlæg blade `[y: 100]` og `[o: 150]` til et træ med rod `250`



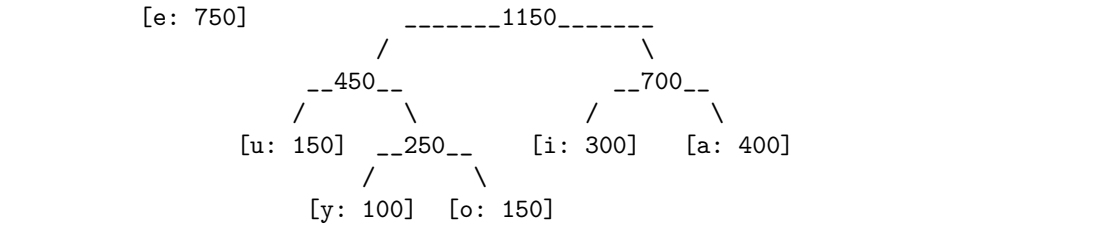
3. Skridt: Sammenlæg bladet `[u: 200]` med træet med rod `250`, så vi får et træ med rod `450`



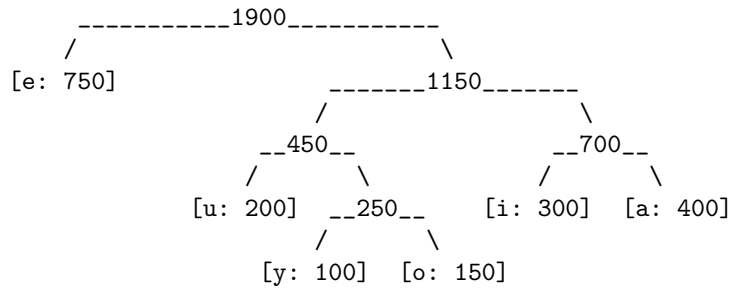
4. Skridt: Sammenlæg blade `[i: 300]` og `[a: 400]`, så vi får et træ med rod `700`



5. Skridt: Sammenlæg træerne med rod `450` og `700`, så vi får et træ med rod `1150`



6. Skridt: Sammenlæg bladet `[e: 750]` med træet med rod `1150`, så vi får et træ med rod `1900`



Når ovenstående træ anvendes i forbindelse med *afkodning* (decoding) eller *kodning* (encoding) af en fil, så læser vi 0 som et venstre skridt og 1 et som et højre skridt på en sti i træet fra roden til et blad.

Vi angiver hvor mange bits en fil fylder når den er kodet vha. det konstruerede Huffman-træ. Dette gøres ved (i) at tælle hvor mange bits der skal bruges til at kode et bestemt tegn og (ii) herefter tage højde for hvor mange af hvert tegn vi skal repræsentere, dvs. hyppigheden af tegnet i filen. Mere præcist observerer vi:

- Karakteren `a` kodes som 111, dvs. 3 bits og vi skal repræsentere 400 af disse karakterer.
- Karakteren `e` kodes som 0, dvs. 1 bit og vi skal repræsentere 750 af disse karakterer.
- Karakteren `i` kodes som 110, dvs. 3 bits og vi skal repræsentere 300 af disse karakterer.
- Karakteren `o` kodes som 1011, dvs. 4 bits og vi skal repræsentere 150 af disse karakterer.
- Karakteren `u` kodes som 100, dvs. 3 bits og vi skal repræsentere 200 af disse karakterer.
- Karakteren `y` kodes som 1010, dvs. 4 bits og vi skal repræsentere 100 af disse karakterer.

Dette resulterer endeligt i at filen vil fylde:

$$3 \cdot 400 + 1 \cdot 750 + 3 \cdot 300 + 4 \cdot 150 + 3 \cdot 200 + 4 \cdot 100 = 4450 \text{ bits} \tag{1}$$

Heraf er det antaget at yderligere information, der kan anvendes til at genskabe det konstruerede Huffman-træ, ikke er inkluderet i filen.

1.7 Eksamen juni 2014 opgave 8

I denne opgave ser vi på at sortere n elementer efter værdien af deres nøgler, når det vides at disse nøgler kun antager værdierne 0 og 1. Angiv for hver af algoritmerne CountingSort, InsertionSort, MergeSort og QuickSort, hvilke af nedenstående køretider som er henholdsvis deres worst-case og deres best-case køretid for denne type input.

- A) $O(n)$
- B) $O(n \log n)$
- C) $O(n^2)$

Svar ved at angive indholdet (enten A, B eller C) af indgangene i følgende tabel:

	Worst case	Best case
CountingSort	A	A
InsertionSort	C	A
MergeSort	B	B
QuickSort	C	C

CountingSort: Køretiden for countingsort er $O(n+k)$ og når input kun består af 0 og 1, så er $k=2$. Dermed bliver køretiden $O(n+2)=O(n)$. Der er ikke forskel på best og worst case input for countingsort, da den eneste forskel mellem forskellige typer inputs ligger i værdien af tælleren for enten 0 eller 1 i C arrayet.

InsertionSort: Best case er klart $O(n)$, da der for sorteret input ikke er nogle inversioner som skal fjernes - dermed ender insertionsort bare med at lave et enkelt gennemløb af input. Et worst case input for insertionsort vil være

$$\underbrace{11\ldots1}_{\text{ettaller}}\underbrace{00\ldots0}_{\text{nuller}}$$

hvor der er $\frac{n}{2}$ ettaller og $\frac{n}{2}$ nuller. Dermed vil hvert nul indgå i en inversion med alle $\frac{n}{2}$ ettaller (husk: elementer med samme nøgle indgår ikke i en inversion med andre elementer med samme nøgle). Der er dermed $\frac{n}{2}\cdot\frac{n}{2}=O(n^2)$ inversioner, hvilket gør køretiden er $O(n^2)$.

MergeSort: Køretiden for mergesort er altid $\Theta(n\log n)$, da den laver det samme arbejde uanset input.

QuickSort: Både worst case og best case køretiden er $O(n^2)$. Observer, at eftersom ethvert input består af 0 og 1, så er 0 og 1 de eneste mulige pivot elementer.

- pivot er 0: første kald af `Partition` vil skabe to partitioner, hvor den ene kun består af nuller og den anden kun af ettaller. Disse to partitioner er altså sorteret og vi har tidligere set køretiden for quicksort med sorteret input er $\Theta(n^2)$. En af partitionerne må mindst have $\frac{n}{2}$ elementer, og da den er sorteret, en køretid på $\Omega(n^2)$. Dette betyder køretiden for hele input er $\Omega(n^2)$.
- pivot er 1: dette vil give et dårligt split, hvor man får to partitioner af størrelse $n-1$ og 0. I det efterfølgende rekursive kald af quicksort på partitionen med størrelse $n-1$, vil man få 0 eller 1 som pivot, hvilket enten vil lave to sorterede partitoner eller et dårligt split. En nedre grænse for køretiden kan findes ved følgende argument: i de første rekursive kald er der dårlige splits indtil 0 ender som pivot og laver to sorterede partitioner. Hvis 0 bliver pivot, når partitionen mindst har størrelsen $\frac{n}{2}$, så er køretiden $\Omega(n^2)$ for hele input, da alene for input størelsen $\frac{n}{2}$ gælder køretiden er $\Omega((\frac{n}{2})^2)=\Omega(n^2)$. Hvis 0 bliver pivot, når partitionen er mindre end $\frac{n}{2}$, så er køretiden for hele input $\Omega(n^2)$, da køretiden alene for de første $\frac{n}{2}$ rekursive kald, hvor der opstår dårlige splits, er $n+(n-1)+\cdots+\frac{n}{2}=\Theta(n^2)$.

Vi har argumenteret for, at uanset input, så er køretiden $\Omega(n^2)$. Dette udelukker, at køretiden kan være $O(n)$ eller $O(n\log n)$ for best case og worst case.

1.8 Eksamen juni 2014 opgave 9

I denne opgave skal vi bestemme køretiden for 4 forskellige algoritmer. For at bedømme køretiden kigger vi på hvor mange gange den inderste løkke bliver gentaget.

I **algoritme1** har vi to for-løkker. Den yderste for-løkke kører n gange, først med $i = 1$, så $i = 2$, og til sidst med $i = n$. Den inderste løkke kører fra $j = i$ til n, dvs. i første iteration af den yderste for-løkke gentager den inderste sig n gange. Anden iteration af det yderste (hvor $i = 2$) gentager den inderste for-løkke sig $n - 1$ gange og så videre. Det vil sige, den inderste løkke gentager sig $n + (n-1) + (n-2) + ... + 1$ gange, hvor vi laver konstant arbejde for hver gentagelse. Denne sum af arbejde kan vi omskrive til $1+2+...+n = \frac{(n+1)n}{2} = O(n^2)$.

I **algoritme2** gentager den yderste for-løkke sig ligesom i algoritme1. I algoritme2 er der i stedet for en inderste for-løkke en while-løkke. While-løkken starter hver gang med $s = n$ og ser hvor mange gange s kan blive halveret før den rammer 1. At spørge hvor mange gange et tal kan blive halveret er det samme som at tage log_2 af tallet, hvilket giver køretiden $O(\log n)$ for while-løkken. Da vi n gange gentager processen med at se hvor mange gange vi kan halvere s får vi en køretid på $O(n \log n)$

I **algoritme3** er de to yderste for-løkker identiske med for-løkkerne i algoritme1, der er dog tilføjet en ekstra løkke. Lad os til at starte med overveje hvordan den nye for-løkke opfører sig når i, j bliver større. Til at starte med er $i = 1$ og $j = 1$, og der laves 1 arbejde. Efterfølgende som j vokser laves der tilsvarende lige så mange gentagelser af løkken. Det vil sige når $j = n$ laves der n gentagelser (løkken går fra $k = i(= 1)$ til $k = j(= n)$. Når i vokser laves der tilsvarende mindre gentagelse af løkken, da der startes fra et større i. Hvis vi ser på antallet af gentagelser af den inderste løkke som en sum for hver iteration af den miderste for-løkke får vi n summer. Her vil den første sum være $1+2+...+n$, den næste $1+2+...+(n-1)$, den tredje $1+2+...+(n-2)$ og så videre indtil den sidste sum er 1. Antallet af gentagelser er lig med den asymptotiske køretid, da der i hver gentagelse bliver lavet konstant arbejde.

For at finde en asymptotisk øvre grænse for det samlede arbejde kan vi finde en øvre grænse for antallet af gentagelser. For at finde en sådan øvre grænse kan vi lade alle n summer være $1 + 2 + ... + n$. Det kan vi skrive om på følgende måde:

$$n \cdot (1 + 2 + ... + n) = n \cdot \frac{(n+1)n}{2} = \frac{n(n+1)n}{2} = \frac{n^3 + n^2}{2}$$

Hvilket giver køretiden $O(n^3)$.

At dette er en god øvre grænse kan ses ved, at i halvdelen af iterationerne i den ydre for-løkke er $i \geq \frac{n}{2}$, dvs. en nedre grænse kan findes ved:

$$n \cdot (1 + 2 + ... + \frac{n}{2}) = \Theta(n^3)$$

Køretiden for algoritmen er altså også $\Omega(n^3)$, dvs. den er faktisk $\Theta(n^3)$.

I **algoritme4** har vi en while-løkke som gentages indtil $n = 1$, hvor n bliver halveret i hver iteration. Inde i while-løkken er der en for-løkke som gentages n gange. Den yderste while-løkke gentages $\log n$ gange pga. samme argument som for den inderste for-løkke i algoritme2. Af den grund kunne man forestille sig at køretiden er $O(n \log n)$, men da variablen n opdateres til at være $n/2$ og den inderste for-løkke afhænger af variablen n kan man finde en bedre køretid. Lad os kigge på hvor meget der bliver lavet i de første iterationer. I første iteration laves der n arbejde i den inderste for-løkke. Anden iteration laves der $n/2$, da n er blevet halveret. Den tredje $n/2^2$, og så videre indtil sidste iteration hvor der laves $n/2^{\log n} = n/n = 1$ arbejde. Det

vil sige, det samlede arbejde kan forklares med følgende sum:

$$n/2^0 + n/2^1 + n/2^2 + ... + n/2^{\log n} = \sum_{i=0}^{\log n} n/2^i$$

hvilket må være højst $\sum_{i=0}^{\infty} n/2^i = 2n = O(n)$. Derfor kan man med en bedre analyse sige at køretiden for algoritme4 er $O(n)$.

At summen $\sum_{i=0}^{\infty} n/2^i = 2n$ følger af at i en exponentielt faldene udvikling dominerer det første led. Dette kan ses ved at omskrive summen på følgende måde:

$$\sum_{i=0}^{\infty} n/2^i = n \cdot \sum_{i=0}^{\infty} 1/2^i = n\frac{1}{1-1/2} = 2n$$

Hvor der er brugt at vi kan omskrive $\sum_{k=0}^{\infty} x^k$ til $\frac{1}{1-x}$ fra eq. A.6 på side 1147 i Cormen et al.[1]

2 Eksaminatorie-timer II

2.1 Opgave 1 - Cormen et al. øvelse 22.5-1

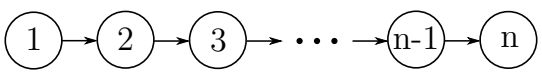
Vi skal se på, hvordan antallet af stærke sammenhængskomponenter (SCCs) kan ændre sig hvis en ny kant bliver tilføjet. Vi ser først på situationen for orienterede grafer, og bagefter ser vi på hvordan antallet af sammenhængskomponenter for uorienterede grafer kan ændre sig.

Orienterede grafer

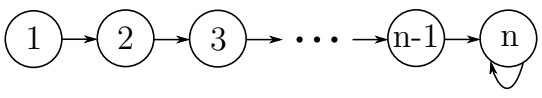
Hvis vi tilføjer en kant kan antallet af SCCs ikke stige da vi ikke kan fjerne en sti fra en knude til en anden knude ved at tilføje en kant (som er det eneste der kan bryde en SCC op i to eller flere SCCs og dermed forøge antallet af SCCs).

Det er klart at antallet af SCCs kan falde hvis vi tilføjer en kant, da denne kant kan tilføje en sti fra en knude til en anden knude, og dermed muligvis slå to (eller flere) SCCs sammen til én SCC, og dermed vil antallet af SCCs falde. Følgende eksempel viser, at ved at tilføje blot én kant, kan vi få antallet af SCCs til at falde med ethvert heltal i intervallet $[0, n-1]$.

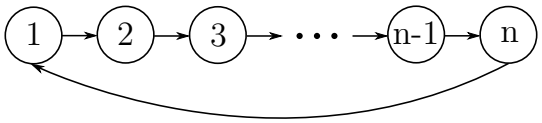
I følgende graf er der n knuder, og der er n SCCs (der findes intet par af knuder, hvor begge knuder har en sti til hinanden):



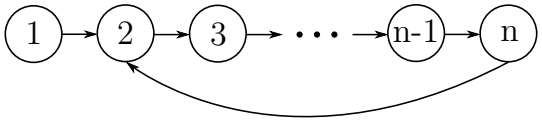
Hvis vi tilføjer en kant fra knude n til knude n, så ændrer antallet af SCCs sig ikke:



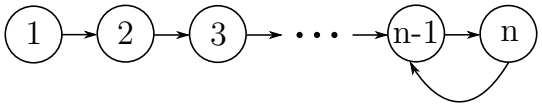
Hvis vi tilføjer en kant fra knude n til knude 1, så er der nu en sti fra enhver knude til enhver anden knude, dvs. alle knuderne er i samme SCC, og antallet af SCCs er faldet fra n til 1, dvs. det er faldet med $n-1$:



Hvis vi tilføjer en kant fra knude n til knude 2, så er der nu en sti fra enhver knude til enhver anden knude undtaget til knude 1. Dvs. alle knuderne på nær knude 1 er i samme SCC, dvs. der er nu 2 SCCs, og antallet er faldet med $n-2$:



Det samme kan vi gøre med alle knuderne hele vejen op til knude $n-1$, hvor der nu kun er en sti mellem $n-1$ og n (og mellem n og $n-1$), dvs. der er $n-1$ SCCs, og antallet af SCCs er faldet med 1:



Dvs. vi kan få antallet til at falde med $0, 1, 2, ..., n-1$. Vi kan ikke få antallet til at falde med n, da der højst er n SCCs i en graf, og der mindst er 1 SCC.

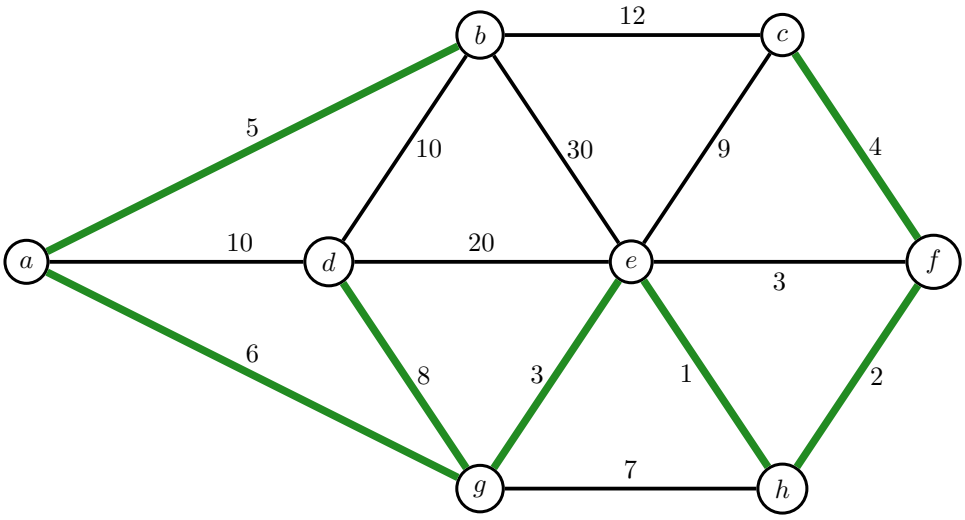
Uorienterede grafer

Når vi tilføjer en kant mellem u og v i en uorienteret graf vil der (hvis der ikke allerede er det i forvejen) blive tilføjet en sti både fra u til v og fra v til u. Der vil samtidig også blive tilføjet stier (hvis der ikke allerede er det i forvejen) mellem alle knuder som har en sti til u og alle knuder som har en sti til v. Sagt på en anden måde: hvis der bliver tilføjet en kant mellem to knuder fra to forskellige sammenhængskomponenter (CCs), så vil disse to CCs blive slået sammen til en enkelt CC da alle knuder blandt disse to CCs nu har en sti mellem sig. Hvis en kant bliver tilføjet mellem to knuder der allerede er i samme CC, sker der ikke noget med antallet af CC.

Opsummering: når der tilføjes en kant kan antallet af CCs enten forblive det samme, eller antallet kan falde med en. Dette er de eneste to muligheder, da enhver kant der tilføjes enten tilføjes mellem knuder fra to forskellige CCs eller knuder fra samme CC.

2.2 Opgave 2 - Eksamen juni 2010, opgave 2, spørgsmål d

I denne opgave skal vi vise et minimum spanning tree (MST) for en graf med vægtede kanter. I figur 1 ses markeret med grøn det MST der fremkommer ved brug af både Kruskal's eller



Figur 1: input graf

Prim's algoritme. Forklaringer og beviser for korrekthed af disse algoritmer kan findes i [4, slides 28 - 47].

Ved brug af Kruskal's algoritme bliver kanterne behandlet i følgende rækkefølge:

- Tilføj kanten (e, h) med vægt 1.
- Tilføj kanten (f, h) med vægt 2.
- Tilføj kanten (e, g) med vægt 3.
- Udelad kanten (e, f) selvom den kun har vægt 3, fordi både e og f allerede er en del af det træ vi har opbygget.
- Tilføj kanten (c, f) med vægt 4.
- Tilføj kanten (a, b) med vægt 5.
- Tilføj kanten (a, g) med vægt 6.
- Udelad kanten (g, h) selvom den kun har vægt 7, fordi g og h allerede er en del af det træ vi har opbygget.
- Tilføj kanten (d, g) med vægt 8.

Ved kørsel af Kruskal's algoritme på denne graf bliver det altid samme kørsel og resultat.

Ved brug af Prim's algoritme med start i knuden a bliver kanterne behandlet i følgende rækkefølge:

- Tilføj kanten (a, b) med vægt 5 (fordi det er kanten med mindst vægt henover det cut der adskiller a og resten af grafen).
- Tilføj kanten (a, g) med vægt 6.

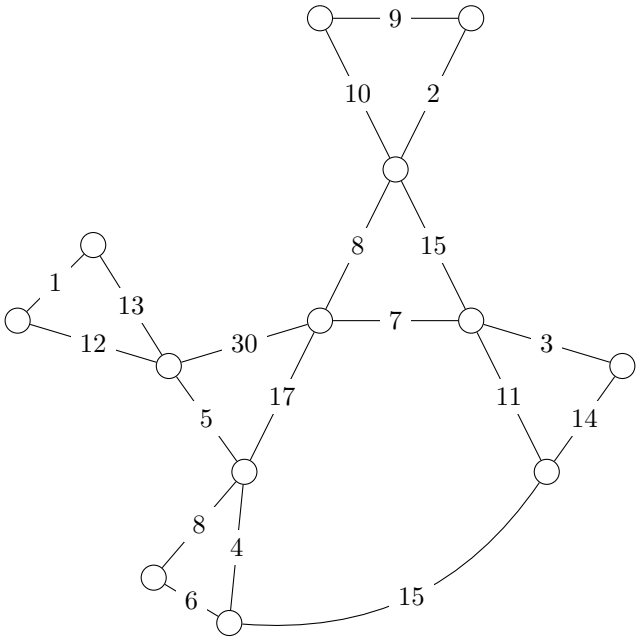
- Tilføj kanten (e, g) med vægt 3.
- Tilføj kanten (e, h) med vægt 1.
- Tilføj kanten (h, f) med vægt 2.
- Udelad kanten (e, f) med vægt 3, fordi e og f allerede er forbundet.
- tilføj kanten (c, f) med vægt 4.
- Udelad kanten (g, h) med vægt 7, fordi g og h allerede er forbundet.
- tilføj kanten (d, g) med vægt 8.

Ved kørsel af Prim's algoritme på denne graf bliver det altid samme resultat, men rækkefølgen af kanter der bliver behandler kan variere alt efter hvilken knude man starter i.

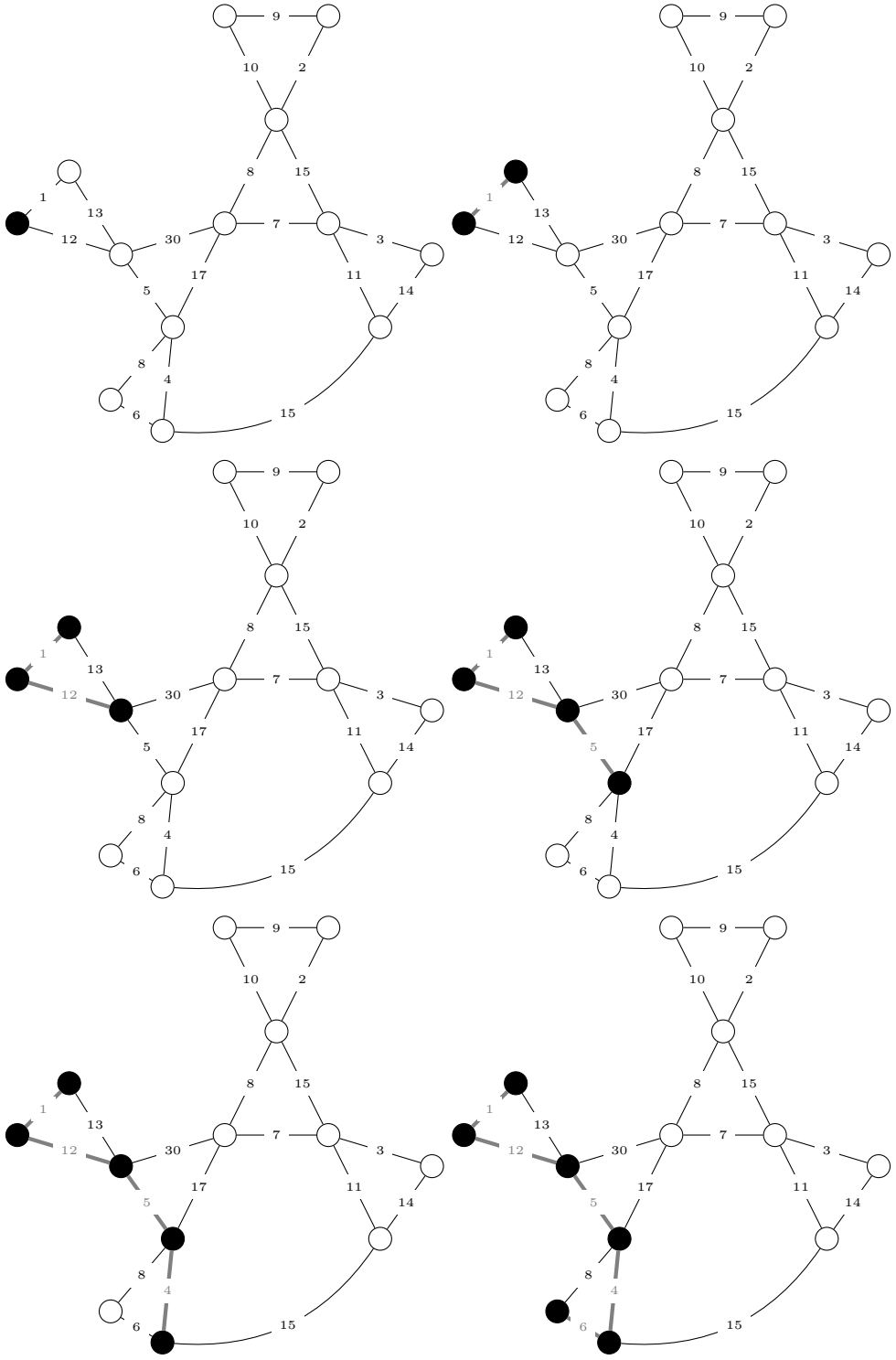
Den samlede vægt for dette MST er $1 + 2 + 3 + 4 + 5 + 6 + 8 = 29$.

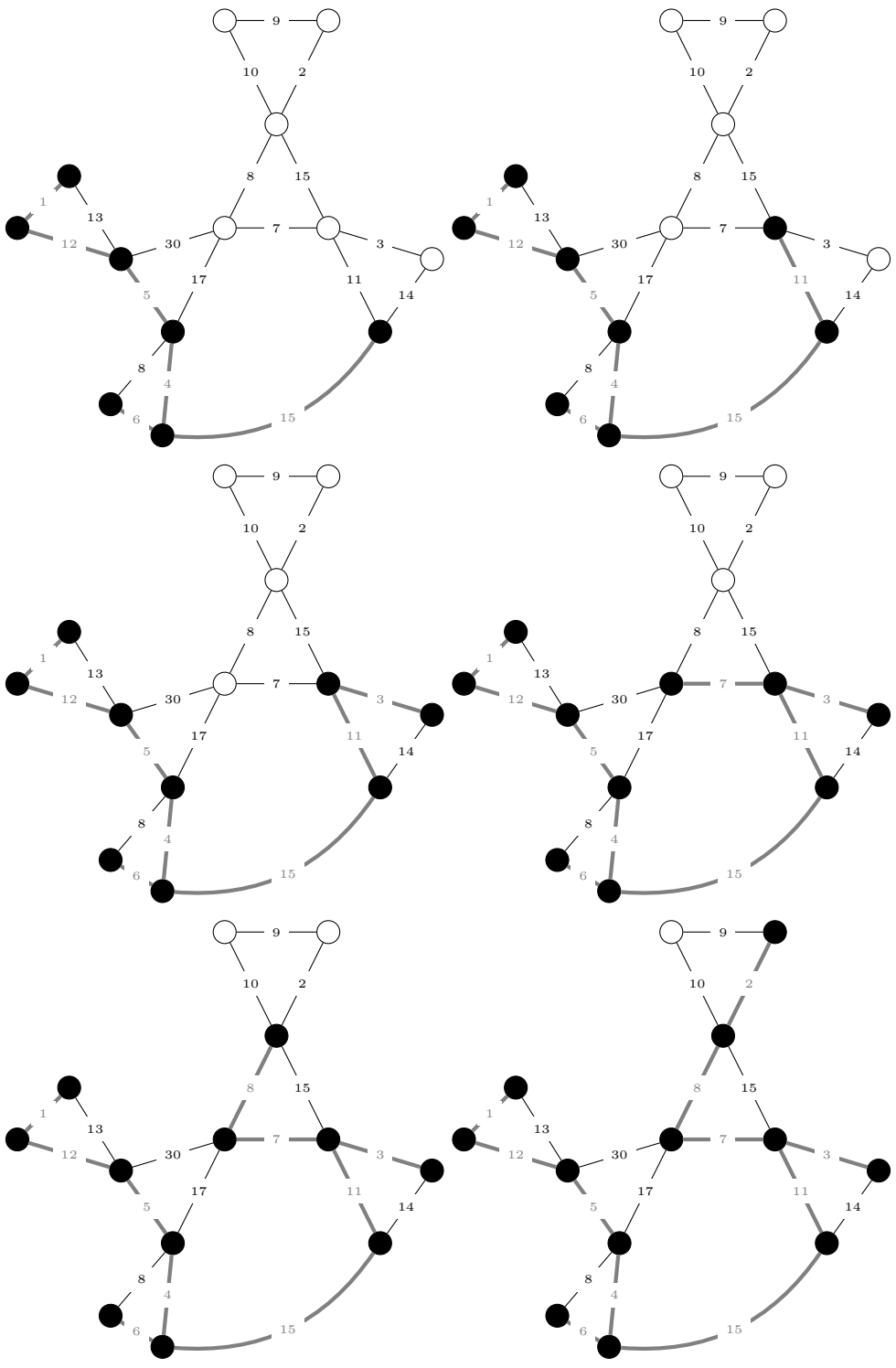
2.3 Opgave 3 - Eksamen januar 2008, opgave 2, spørgsmål b

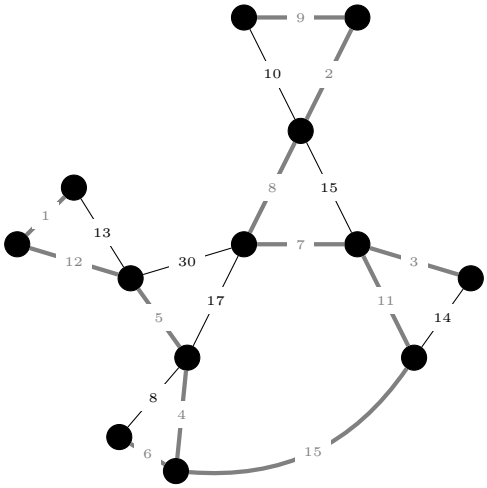
Prim-Jarníks algoritme og Kruskals algoritme skal bruges til at finde letteste udspændende træ (MST) af nedenstående vægtede graf.



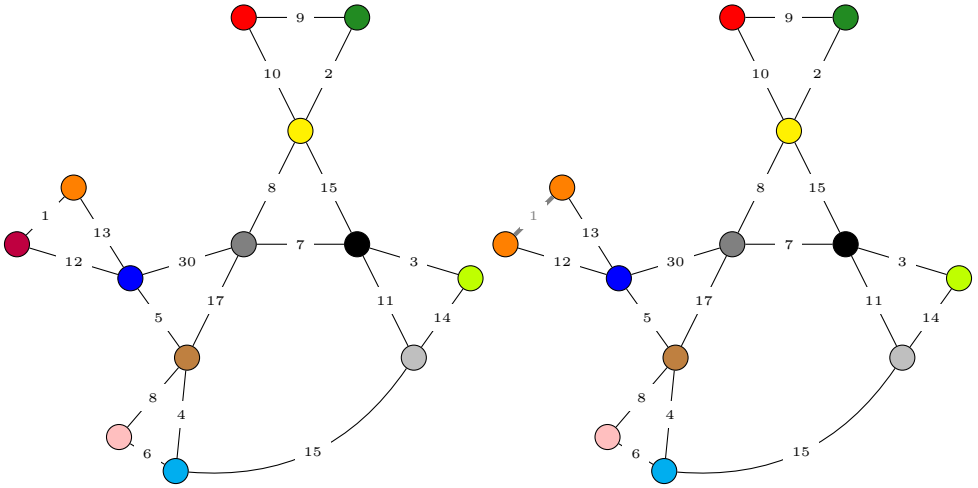
Først køres Prim-Jarníks algoritme[1, p.634]. Sorte knuder er blevet en del af træet, og grå kanter er trækanterne mellem disse knuder.

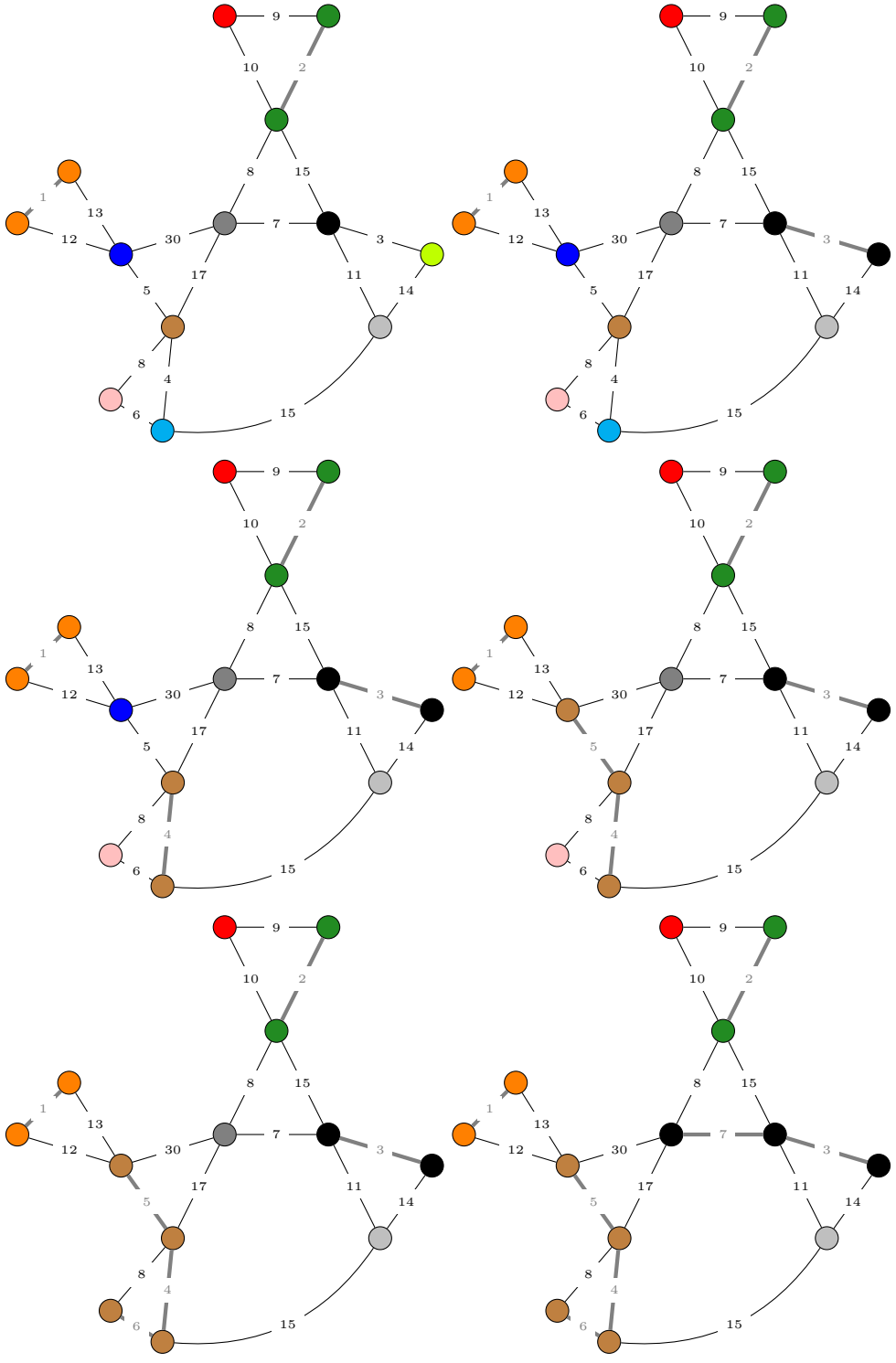

9
10
2
8
15
1
13
12
30
7
3
5
17
11
14
8
4
6
15

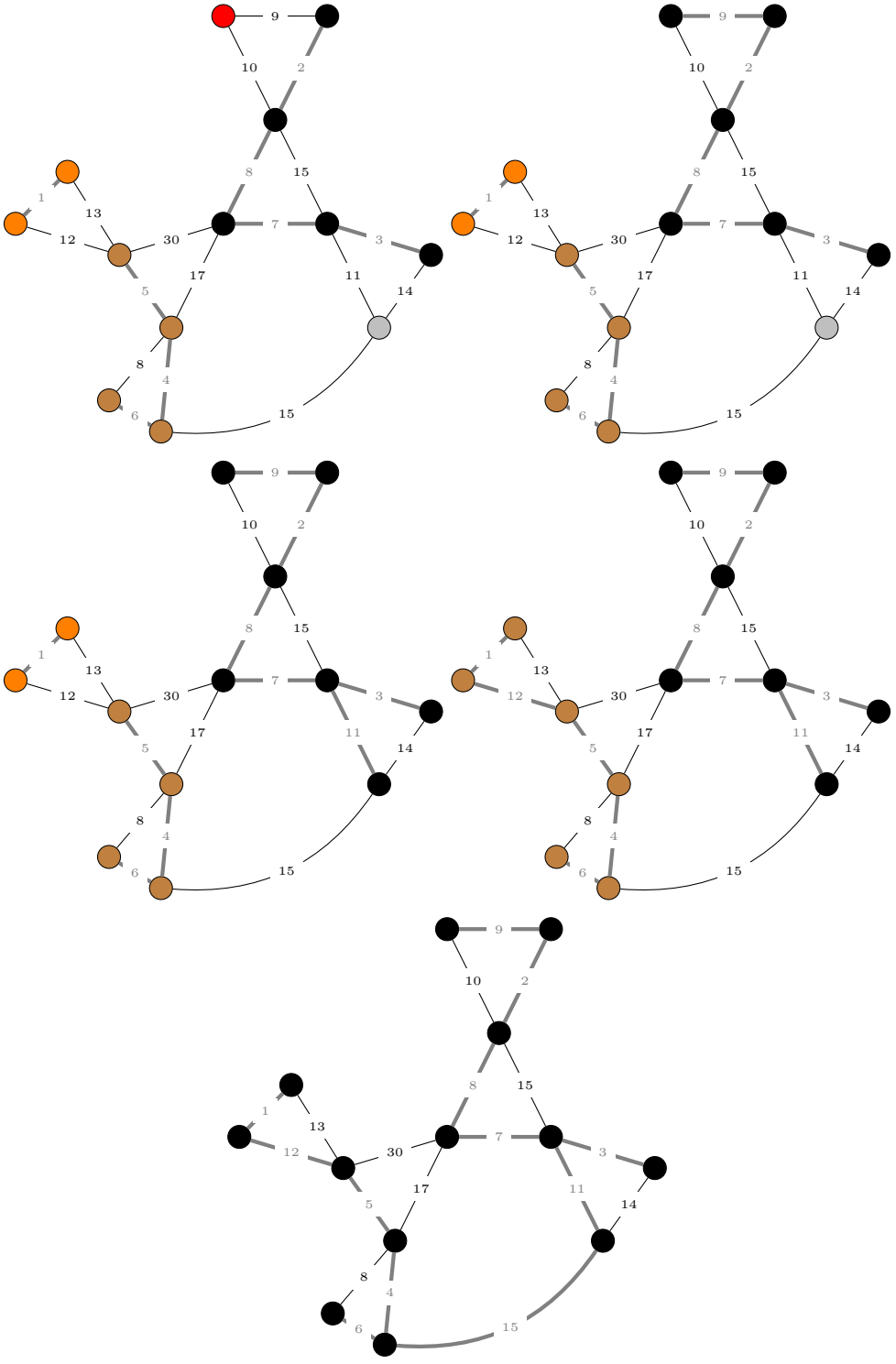

9
10
2
8
15
1
13
7
12
30
3
5
17
11
14
8
4
6
15



Derefter køres Kruskals algoritme[1, p.631]. Hver farve repræsenterer en sammenhængskomponent, og grå kanter repræsenterer trækanterne mellem knuderne i sammenhængskomponenterne.






9
10
2
8
15
1
13
7
12
30
3
17
11
14
5
8
4
6
15

2.4 Opgave 4 - Cormen et al. øvelse 23.2-4

I følgende opgave angiver vi hvor hurtigt Kruskal's algoritme kan køre, når alle kanterne E i grafen $G = (V, E)$ har vægte i intervallet fra 1 til $|V|$. For at kunne gøre dette observerer vi som udgangspunkt, at Kruskal's algoritme på en sammenhængende graf, bruger tid på (Cormen et al. Section 23.2):

- Sortering af kanter: $O(|E| \log(|E|))$ tid, ved brug af en sammenligningsbaseret sorteringsalgoritme.
- Make-Set, Union og Find-Set operationer: $O(|E|\alpha(|V|))$ (hvis $|E| \geq |V| - 1$).

Den samlede køretid vil være $O(|E| \log(|E|) + |E|\alpha(|V|)) = O(|E| \log(|E|))$. Vi ser i denne sammenhæng at sorteringen af kanterne vil være den mest krævende opgave.

Med dette som udgangspunkt, ser vi at Kruskal's algoritme, på en sammenhængende graf hvor kanterne har vægte i intervallet 1 til $|V|$, kan gøres hurtigere ved brug af countingsort, da vi så kan lette en af de mest krævende del af algoritmen. Mere konkret ser vi at Kruskals algoritme i dette tilfælde vil bruge tid på:

- Sortering af kanter: $O(|E|)$ tid, ved brug af en lineær-tid sorteringsalgoritme.
- Make-Set, Union og Find-Set operationer: $O(|E|\alpha(|V|))$ (hvis $|E| \geq |V| - 1$).

Den samlede køretid vil være $O(|E|+|E|\alpha(|V|)) = O(|E|\alpha(|V|))$. Vi ser i denne sammenhæng at mængde operationerne vil være den mest krævende opgave.

2.5 Opgave 5 - Eksamen juni 2014, opgave 7

Denne opgave handler om at bruge Kruskals algoritme til at finde et MST for nedenstående graf $G = (V, E)$. Vi ser i spørgsmål a b og d på situationen efter at algoritmen har undersøgt 7 kanter (dvs. har lavet 7 iterationer af det andet for-loop på side 631 i lærebogen).

Spørgsmål a: Angiv hvilke kanter der er valgt til at indgå i MST'et (dvs. er i A) efter at Kruskals algoritmen har undersøgt 7 kanter. En kant med ende punkter u og v skrives som sædvanligt (u, v). I hver kant, angiv endepunkterne i alfabetisk rækkefølge.

De valgte kanter er

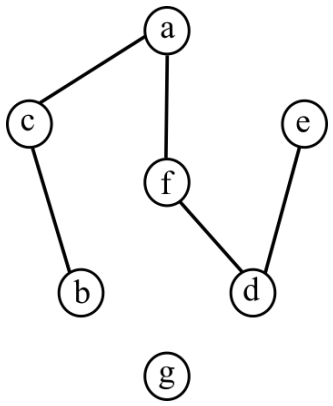
Iteration	Valgt kant
1	(b, c)
2	(a, f)
3	(a, c)
4	(d, e)
5	[Ingen valgt]
6	(d, f)
7	[Ingen valgt]

Udførslen af de første 7 iterationer af Kruskals algoritme er illustreret i Figur 3-10 (side 23).

Spørgsmål b: Angiv sammenhængskomponenterne som kanterne fra spørgsmål a giver anledning til, dvs. angiv sammenhængskomponenterne i grafen $G' = (V, A)$. Hver sammenhængskomponent angives som en liste af knuder. I hver liste, angiv endepunkterne i alfabetisk rækkefølge.

For G' er sammenhængskomponenterne

- $\{a, b, c, d, e, f\}$
- $\{g\}$



Figur 2: Illustration af G'

Spørgsmål c: Angiv vægten af et minimum udspændende træ (MST) for hele grafen G.

Det er ikke nok at udføre de 7 iterationer i andet for-loop af Kruskals algoritme for at opnå et MST for hele G. Man kan satens køre algoritmen færdig for at opnå et MST, men man kan også observere følgende:

- der eksisterer et MST som indeholder A (de allerede valgte kanter). Dette følger af virkemåden af Kruskals algoritme.
- $S = \{g\}$ er et cut og ingen kanter i A går henover cuttet (ingen kanter i A er i $S \times (V - S)$).
- kanten (d, g) er den letteste kant blandt kanterne henover cuttet

Derved følger det af *Cut*-sætningen [4, s. 8], at (d, g) er "safe"at tilføje til de allerede valgte kanter. Dvs. der eksisterer et MST som indeholder $A \cup (d, g)$. Eftersom grafen bestående af den originale mængde af knuder og kanterne $A \cup (d, g)$ er et træ samt at der eksisterer et MST, som indeholder dets kanter, så må det være et MST for hele G.

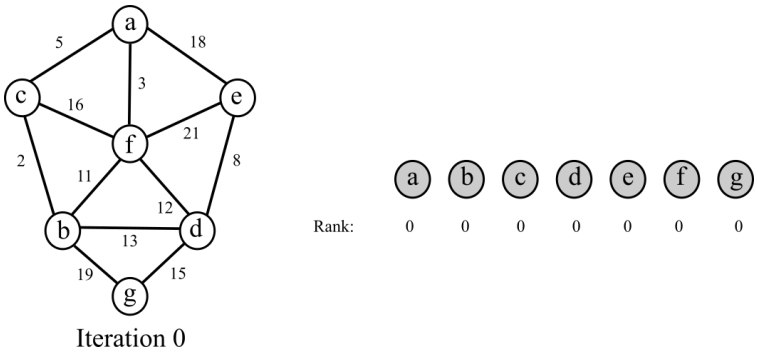
Summeres vægtene for de valgte kanter, så fås vægten af det MST for hele G beskrevet ovenover som

$$2 + 3 + 5 + 8 + 12 + 15 = 45$$

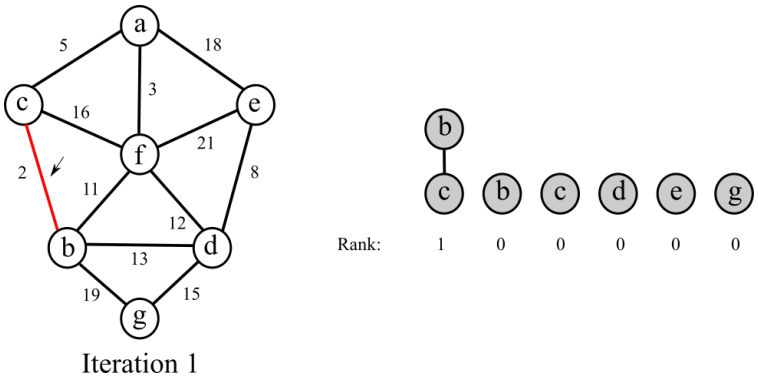
Spørgsmål d: Vi antager nu at Kruskal bruger en disjoint-set datastruktur der er implementeret via en skov af træer som i lærebogens afsnit 21.3, under brug afbåde union-by-rank og path-compression heuristikken. Hvis der under `Union` laves et `Link(x,y)` på to knuder x og y med samme rank, antages det i dette spørgsmål at knuden med det alfabetisk mindste navn bliver den nye rod. Angiv udseendet af disjoint-set skoven efter at Kruskals algoritme har under-søgt 7 kanter. Hvert træ i skoven angives ved at skrive en liste af kanterne i det, samt hvilken knude som er roden. Angiv også rangen af roden.

I Figur 3-10 er disjoint-set skoven angivet for hver iteration. Ud fra Figur 10 kan vi for hvert træ aflæse dets kanter, dets rod og rangen af roden:

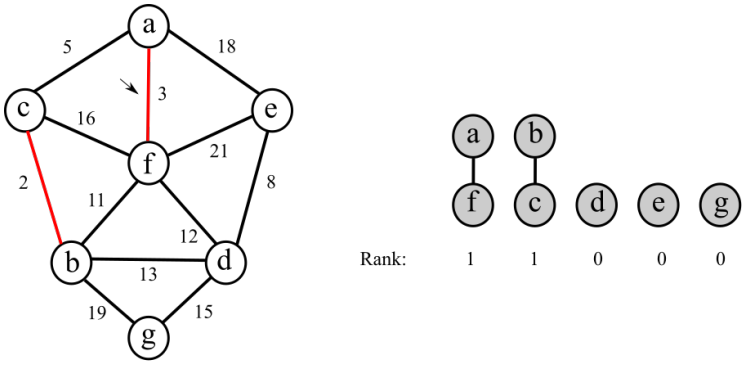
- Træ 1: (a, f), (a,b), (a, d), (b, c), (d, e) (roden = a, rank = 2)
- Træ 2: Ingen kanter i træet (roden = g, rank = 0)



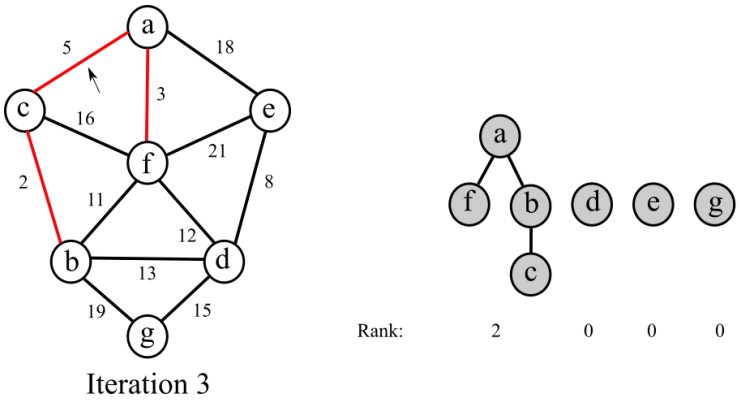
Figur 3: Illustration af iteration af andet for-loop i Kruskals algoritme. Til venstre ses grafen med de valgte kanter markeret rødt. Til højre ses udseendet af disjoint-set skoven ved hver iteration. Ovenstående gælder også for Figur 4-10



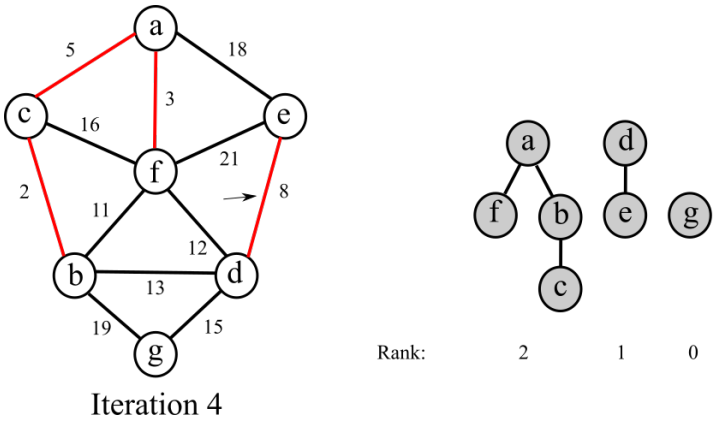
Figur 4



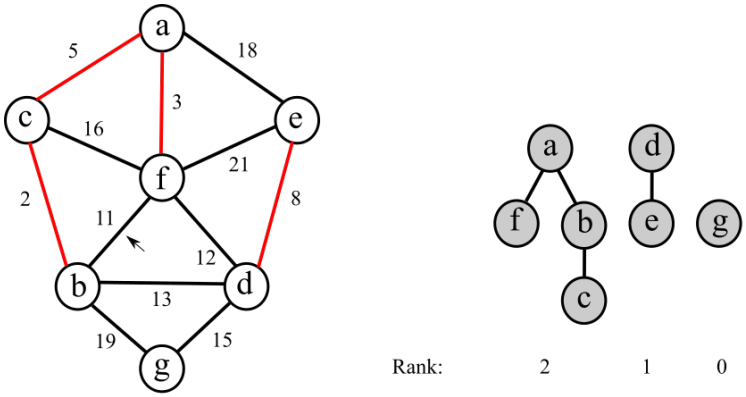
Figur 5



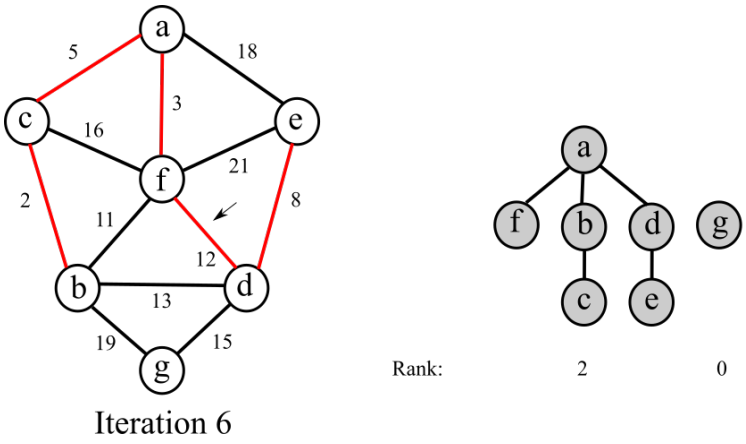
Figur 6



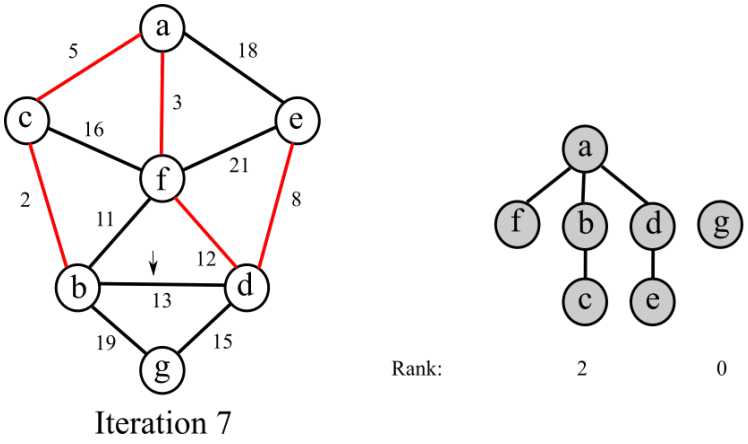
Figur 7



Figur 8



Figur 9



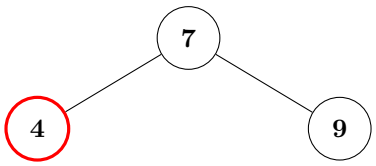
Figur 10

2.6 Opgave 6 - Eksamen juni 2012, opgave 2

Spørgsmål a

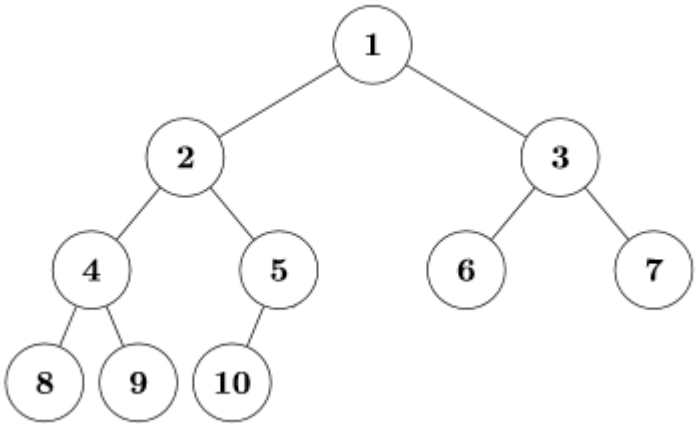
Vi skal bedømme hvilke af fire arrays repræsenterer min-heaps. En min-heap skal være i heap-orden og have heap-facon, vi kan for de givne arrays allerede se at de må være i heap-facon da hele arrayet er fyldt ud. Derfor skal vi her kun tjekke for hvert array om dette er i heap-orden. Det kan vi lettere bedømme ved at lave array repræsentationen om til et binært træ og sikre os at heap-ordenen holder undervejs. Dvs. at alle forældre i træet skal have en nøgle som er lig med eller mindre end deres børns nøgler.

For at omskrive det første array til en binært træ repræsentation lader vi det første element være roden, og de næste to element være rodens børn. Men allerede her sker der en overtrædelse af heap-orden, da 4 er mindre en 7, men 7 er forældre til 4. I figur 11 kan man se det ufærdige træ.



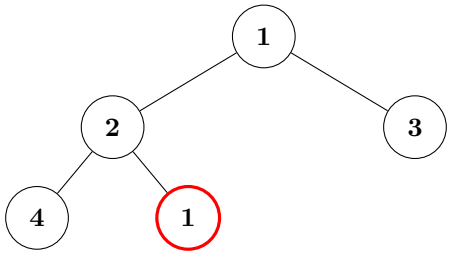
Figur 11: Starten af array A_1 som binært træ. Knuden som overtræder heap-ordenen er markeret med rød

Arrayet A_2 er i sorteret orden og må derfor være i heap-orden som vi har vist i en tidligere opgave. Vi kan dog skrive det op som et binært træ for at overbevise os selv endnu mere. Det gør vi ligesom ved A_1 ved at skrive det første element som roden og de næste to elementer børnene, de næste 4 som børnenes børn og så videre. Det resulterer i følgende træ, hvor man kan se at alle forældrenes nøgler er mindre end eller lig med deres børns nøgler.



Figur 12: Array A_2 som binært træ

I Arrayet A_3 kan man se at der optræder et 1 midt i arrayet man kan derfor intuitivt forestille sig at denne vil blive et barn af et af elementerne mellem roden og elementet med nøglen 1. Og da ingen af disse elementer har en nøgle der er mindre eller lig med 1 kan man udelukke at A_3 er i heap-orden. Dog for at være helt sikker kan man igen tegne træet. Det gør vi på samme måde som ved A_2 indtil vi har tegnet det miderste element med nøglen 1. Det resulterer i følgende træ.

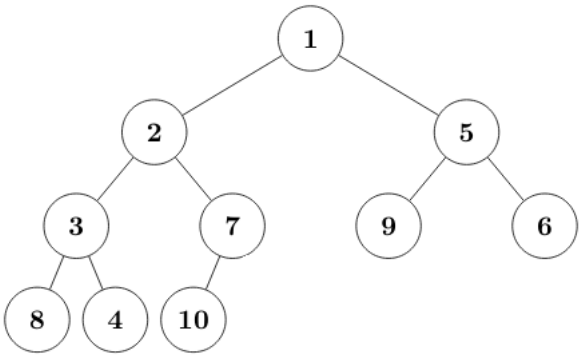


Figur 13: Starten af Array A_3 som binært træ. Knuden med nøglen 1 som overtræder heap-ordenen er markeret med rød

Arrayet A_4 indeholder kun elementer med nøglen 1, og siden en forældre gerne må have samme nøgle som sine børn må dette array være i heap-orden.

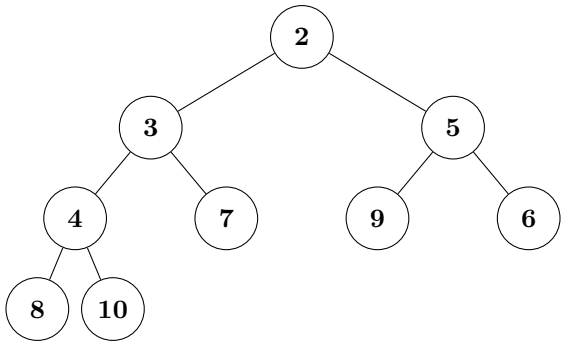
Spørgsmål b

Vi skal udføre heap-extract-min på det array A_5. For at gøre det lettere kan vi først tegne heapen som et binært træ.



Figur 14: Array A_5 som et binært træ

For at udføre heap-extract-min flytter vi først knuden med nøglen 10 op på rodens plads og så udfører vi min-heapify på den. Med min-heapify tager vi den mindste af 10s nye børn, dvs. 2 og bytter rundt på den og 10. Så kalder vi min-heapify på 10s nye plads, hvor vi ender med at bytte rundt på 3 og 10. Nu er 10 på knuden med nøglen 3s plads i figur 15 og vi ender med at bytte rundt på knuderne 4 og 10. Dette giver følgende træ.



Figur 15: Array A_5 efter heap-extract-min

2.7 Opgave 7 - Eksamen juni 2012, opgave 6

2.7.1 Spørgsmål a

i	53	52	26	13	12	6	3	2	1
k	0	0	1	2	2	3	4	4	5

2.7.2 Spørgsmål b og c

Vi laver her både spørgsmål b og c, da c fås fra termination delen af invariantbeviset.

Initialization

Til at starte med er $i = n$ og $k = 0$.

i) $\lfloor \log i \rfloor + k = \lfloor \log n \rfloor + 0 = \lfloor \log n \rfloor$ og invarianten er sand.

ii) i er et heltal, da $i = n$, og vi antager at n er et heltal. Invarianten er derfor sand.

Maintenance

Lad indholdet af variablerne før en iteration være i og k, og efter iterationen være i' og k'. n ændrer sig ikke, så indholdet af den er hele tiden givet ved n.

Vi ser på 2 cases. Når i er lige og når i er ulige.

Antag at i er lige.

Her gør algoritmen at:

$$\begin{aligned} i' &= \frac{i}{2} \\ k' &= k + 1 \end{aligned}$$

Vi vil gerne vise, at

$$\lfloor \log i' \rfloor + k' = \lfloor \log n \rfloor$$

Vi indsætter udtrykkene for i' og k':

$$\lfloor \log i' \rfloor + k' = \lfloor \log \frac{i}{2} \rfloor + (k+1) = \lfloor \log i \rfloor - 1 + k + 1 = \lfloor \log i \rfloor + k$$

Ved sidste lighedstegn bruger vi at $\lfloor \log \frac{i}{2} \rfloor = \lfloor \log n \rfloor - 1$, som vi er givet i opgavebeskrivelsen.

Da vi per invarianten har at $\lfloor \log i \rfloor + k = \lfloor \log n \rfloor$, har vi nu også at $\lfloor \log i' \rfloor + k' = \lfloor \log n \rfloor$, og invarianten i) er dermed også sand til næste iteration. ii) er også sand til næste iteration, da i er lige (og heltal per invarianten), og halvdelen af et lige heltal er et heltal.

Antag at i er ulige.

Her gør algoritmen at:

$$\begin{aligned} i' &= i - 1 \\ k' &= k \end{aligned}$$

Vi vil gerne vise, at

$$\lfloor \log i' \rfloor + k' = \lfloor \log n \rfloor$$

Vi indsætter udtrykkene for i' og k':

$$\lfloor \log i' \rfloor + k' = \lfloor \log(i-1) \rfloor + k = \lfloor \log i \rfloor + k$$

Ved sidste lighedstegn bruger vi at $\lfloor \log(i-1) \rfloor = \lfloor \log i \rfloor$, som vi er givet i opgavebeskrivelsen, og kan bruge da vi ved at i er et ulige heltal.

Da vi per invarianten har at $\lfloor \log i \rfloor + k = \lfloor \log n \rfloor$, har vi nu også at $\lfloor \log i' \rfloor + k' = \lfloor \log n \rfloor$, og invarianten i) er dermed også sand til næste iteration. ii) er også sand til næste iteration, da i er et heltal per invarianten, og når man trækker 1 fra et heltal har man stadig et heltal.

Termination

Vi viser først, at løkken rent faktisk stopper. Løkken er i gang så længe $i > 1$. Hvis i er ulige trækker vi 1 fra i, dvs. i bliver 1 mindre. Hvis i er lige, halverer vi i, og da $i > 1$ og heltal, så er $\frac{i}{2} \leq i - 1$. Dvs. i bliver altid mindst 1 mindre i hver iteration, og løkken må derfor stoppe på et tidspunkt.

Når løkken stopper ved vi, at invarianten gælder. Dvs. efter løkken gælder det at $\lfloor \log i \rfloor + k = \lfloor \log n \rfloor$.

Hvis ingen iterationer har været udført, må $i = 1$, da $i = n$ til at starte med, $n \geq 1$, og grunden til vi ikke udførte en iteration må være fordi $i \leq 1$.

Hvis der har været udført mindst én iteration, lad os da se på den sidste iteration. Lad i være indholdet af i lige før den sidste iteration og i' være indholdet efter den sidste iteration. Da der blev udført en iteration må $i > 1$, og da i er heltal per invarianten, må $i \geq 2$. Hvis i blev halveret i iterationen er $i' \geq 1$, og hvis der blev trukket 1 fra i er $i' \geq 1$. Men da det var den sidste iteration må $i' \leq 1$, dvs. det må gælde at $i' = 1$.

Vi kan nu indsætte dette i invarianten, og se at $\lfloor \log 1 \rfloor + k = \lfloor 0 \rfloor + k = 0 + k = k = \lfloor \log n \rfloor$. Dvs. efter løkken er $k = \lfloor \log n \rfloor$, og da vi returnerer k, returnerer vi $\lfloor \log n \rfloor$, som er det algoritmen skal, og vi har vist at algoritmen fungerer korrekt.

2.7.3 Spørgsmål d

I mindst hver anden iteration vil i være lige, og vil derfor blive halveret (hvis i er ulige i en iteration bliver der trukket en fra i, og i er dermed lige i den næste iteration). Til at starte med er $i = n$, og løkken stopper når $i \leq 1$. Dvs. algoritmen bruger højst $O(\log n)$ tid.

Hver iteration giver også højst en halvering, da vi enten halverer i, eller også trækker vi en fra i, og siden $\frac{i}{2} \leq i - 1$, så når vi trækker en fra i halverer vi højst i. Dvs. algoritmen bruger mindst $\Omega(\log n)$ tid.

Dvs. algoritmens køretid bliver $\Theta(\log n)$.

Litteratur

[1] T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, third edition, 2009.

[2] Rolf Fagerberg. Asymptotisk analyse af algoritmers køretider. URL `https://imada.sdu.dk/~rolf/Edu/DM507/F20/asymptotiskAnalyseAfAlg.pdf`, 2020.

[3] Rolf Fagerberg. Dictionaries. URL `https://imada.sdu.dk/~rolf/Edu/DM507/F20/dictionarySlides.pdf`, 2020.

[4] Rolf Fagerberg. Minimum udspændende træer. URL `https://imada.sdu.dk/~rolf/Edu/DM507/F20/mstSlides.pdf`, 2020.