Forslag til løsning af eksamenssættet i DM02 januar 2004

(med forbehold for trykfejl)

Opgave 1

Spørgsmål a: Som karakteristisk operation vælges '$*$', da det samlede arbejde udført er proportionalt med arbejdet ved at udføre denne operation.

I den inderste for-løkke udføres '$*$' 1 gang, og den inderste for-løkke gennemløbes i gange for hvert gennemløb af den yderste. Derudover udføres yderligere en '$*$' i hvert gennemløb af yderste forløkke. Den yderste forløkke gennemløbes $n+1$ gange.

I alt udføres '$*$'

$$\sum_{i=0}^{n}(i+1) = \left(\sum_{i=0}^{n} i\right) + n + 1 = \frac{n(n+1)}{2} + n + 1 = \frac{n^2}{2} + \frac{3n}{2} + 1 \in \Theta(n^2)$$

gange.

Spørgsmål b: Vi skal vise, at $\texttt{total} = a_n x^{n-(\texttt{i}+1)} + a_{n-1}x^{n-(\texttt{i}+2)} + \ldots a_{\texttt{i}+2}x + a_{\texttt{i}+1}$ gælder i starten af første gennemløb af løkken og i starten af hvert af de følgende gennemløb.

Første gennemløb: Lige inden for-løkken sættes

$$\texttt{total} = A[\texttt{A.length-1}] = A[n+1-1] = A[n] = a_n.$$

I starten af første gennemløb af løkken er $\mathtt{i} = \mathtt{A.length} - 2 = n - 1$. D.v.s. $n - (i+1) = 0$, og $i + 1 = n$. Dermed er

$$a_n x^{n-(\mathtt{i}+1)} + a_{n-1}x^{n-(\mathtt{i}+2)} + \ldots a_{\mathtt{i}+2}x + a_{\mathtt{i}+1} = a_n.$$

Dvs. $\mathtt{total} = a_n = a_n x^{n-(\mathtt{i}+1)} + a_{n-1}x^{n-(\mathtt{i}+2)} + \ldots a_{\mathtt{i}+2}x + a_{\mathtt{i}+1}$.

Efterfølgende gennemløb: Lad $\mathtt{total}_i$ være værdien af `total` i starten af for-løkken.

$$\begin{aligned}
\mathtt{total}_i &= \mathtt{total}_{i+1}x_0 + A[i+1] \\
&= \mathtt{total}_{i+1}x_0 + a_{i+1} \\
&= (a_n x_0^{n-((i+1)+1)} + a_{n-1}x_0^{n-((i+1)+2)} + \ldots + a_{(i+1)+2}x_0 + a_{(i+1)+1})x_0 + a_{i+1}, \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{ifølge induktionsantagelsen} \\
&= (a_n x_0^{n-(i+2)} + a_{n-1}x_0^{n-(i+3)} + \ldots + a_{i+3}x_0 + a_{i+2})x_0 + a_{i+1} \\
&= a_n x_0^{n-(i+1)} + a_{n-1}x_0^{n-(i+2)} + \ldots + a_{i+3}x_0^2 + a_{i+2}x_0 + a_{i+1}
\end{aligned}$$

Spørgsmål c: Sidste gang, vi står i starten af for-løkken, springer vi straks ud igen uden at opdatere `total`. Da er i $= -1$. Ifølge spm. b gælder da

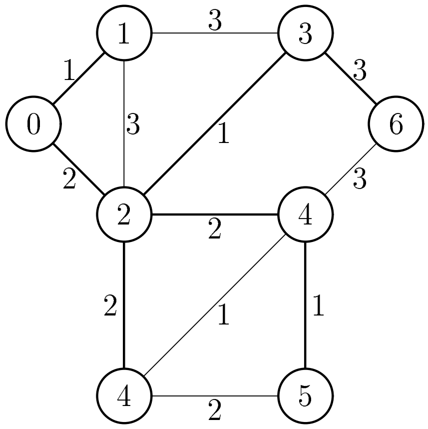
$$\begin{aligned}
\mathtt{total} &= a_n x_0^{n-(-1+1)} + a_{n-1}x_0^{n-(-1+2)} + \ldots + a_{-1+2}x_0 + a_{-1+1} \\
&= a_n x_0^n + a_{n-1}x_0^{n-1} + \ldots + a_1 x_0 + a_0
\end{aligned}$$

Opgave 2

Spørgsmål a: Det nemmeste er at bruge Dijkstras algoritme:

Først tildeles a's naboer et afstandsestimat; b får estimatet 1 og e får estimatet 2. Da b har det mindste estimat, føjes den til træet og har altså afstanden 1 til a. Nu har c afstandsestimat 1+3=4. e får ikke opdateret sit estimat, da 1+3>2. Nu er e den knude blandt træets naboer, som har det laveste estimat. Derfor tilføjes e, og afstanden fra a til e må derfor være 2. Herefter ændres c's afstandsestimat fra 4 til 3. Knuderne f og g får begge estimatet 4. Dermed er c den næste knude, der tilføjes. Så tilføjes f og g, derefter h, og til sidst d.

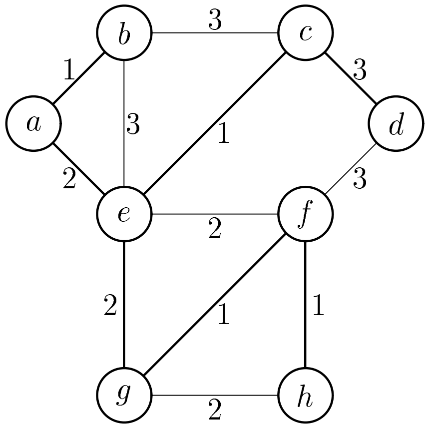
På denne måde fås korteste-vej-træet markeret med fed nedenfor. Afstandene er skrevet i de enkelte knuder.



Spørgsmål b: En nem måde at løse opgaven på er at anvende Kruskals algoritme:

Man ser på kanterne enkeltvis i rækkefølge efter vægt; kanter med lav vægt først. En kant føjes til træet, hvis der ikke opstår en kreds ved tilføjelsen. Alt efter i hvilken rækkefølge man behandler kanter med samme vægt, kan der komme forskellige træer ud af det, men de har alle minimum vægt.

I dette tilfælde kommer alle kanter med vægt 1 med i træet. Derefter kan man ikke tilføje kanten mellem g og h uden at lave en kreds, og man kan ikke tilføje både kanten mellem e og f og kanten mellem e og g uden at lave en kreds. Til sidst er man nødt til at vælge en d's to kanter, selvom de har vægt 3.



Spørgsmål c: Når alle kanter har samme vægt, er ethvert udspændende træ et letteste udspændende træ. Bredde-først-søgning (eller dybde-først-søgning) giver et udspændende træ.

Spørgsmål d: Ideen er, at hvis der kun er få mulige kantvægte, kan vi lave en mere effektiv version af Prims algoritme ved at implementere prioritetskøen som et array bestående af k hægtede lister — en for hver mulig vægt.

Når man møder en ny knude, indsættes den i listen svarende til vægten af den kant, som forbinder den til træet. Når man til en knude, man har set før, langs en kant med lavere vægt end vægten svarende til den liste, knuden befinder sig i, flytter man knuden til listen svarende til den lavere vægt.

For at kunne slette en knude fra en liste i konstant tid, skal listerne være dobbelt-hægtede, og der skal være referencer mellem hver knude i adjacens-listerne og den tilsvarende knude i prioritetskøen.

Når man skal finde den næste knude, som skal indsættes i træet, løber man array'et igennem fra starten. Så snart man finder en plads med en ikke-tom liste, slettes det første element fra denne liste.

Nu kan man altså opdatere en knudes nøgle i konstant tid — dette sker højst m gange — og man kan finde den næste knude, der skal tilføjes træet i tid $O(k)$. Da vi skal tilføje præcis $n-1$ knuder ud over roden, er den samlede tid $O(kn+m)$.

Kommentar: Løsningen minder meget om løsningen til opg. 24.3-6 fra ugeseddel 14.

Spørgsmål e:

$$\begin{aligned} kn + m \in o(m \log n) &\Leftrightarrow \\ kn \in o(m \log n) &\Leftrightarrow \\ k \in o\left(\frac{m}{n} \log n\right) & \end{aligned}$$

Første "$\Leftrightarrow$" følger af $m \in o(m \log n)$.

Opgave 3

Spørgsmål a: $\langle 2, 3, 8, 9\rangle$ eller $\langle 2, 5, 8, 9\rangle$.

Spørgsmål b: Det første, man undersøger, er, om det kan betale sig at tage $A[1]$ med i delfølgen, dvs. om man kan opnå en længst mulig delfølge, hvis man tager $A[1]$ med, eller om man kan opnå en længere voksende delfølge ved at udelade $A[1]$. Dette undersøges med kaldet lvd(A,1,0): Med lvd(A,2,0) beregner man længden af den længst mulige delfølge, som ikke indeholder $A[1]$, og lvd(A,2,1)+1 er længden af den længst mulige delfølge, som indeholder — dvs. starter med — $A[1]$. Bemærk, at man altid kommer til at foretage kaldet lvd(A,2,1), fordi $A[0] = 0$, og alle tal i følgen er positive, dvs. $A[1] > A[0]$.

Generelt beregner man med lvd(A, i, j), hvor mange af tallene fra $A[i..k]$, der kan føjes til en delfølge, som slutter med $A[j]$. Med kaldet lvd$(A, i+1, j)$ behandler man det tilfælde, hvor $A[i]$ udelades. Hvis $A[i] > A[j]$, er det lovligt at føje $A[i]$ til en delfølge, som slutter med $A[j]$, så i dette tilfælde undersøger man med kaldet lvd$(A, i+1, i)$, hvor mange af tallene i $A[i+1..k]$, der kan tilføjes efter en delfølge, som slutter med $A[i]$. Hvis lvd$(A, i+1, i) + 1 >$ lvd$(A, i+1, j)$, får man den længste delfølge ved at tage $A[i]$ med.

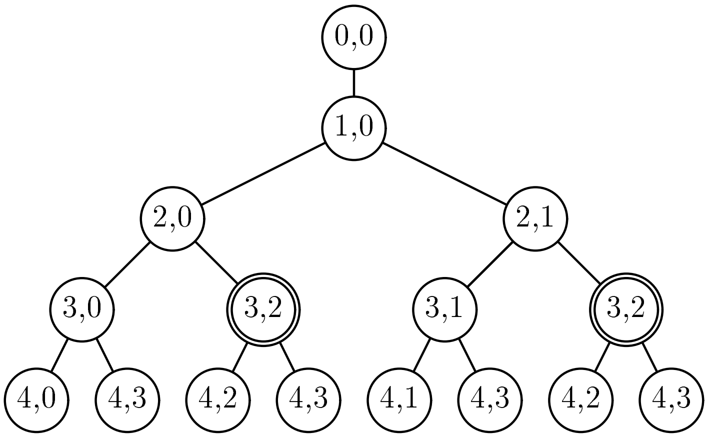
Med kaldet lvd$(A, 1, 0)$ beregner man altså længden af den længste voksende delfølge af $A[1..k]$.

Kommentar: Der var en del, som stort set blot refererede koden linie for linie. Det er ikke tilstrækkeligt. Man skal give et overblik over, hvad der foregår, og hvorfor det virker.

Spørgsmål c: Eks: $\langle 1, 2, 3 \rangle$.

Da følgen er voksende, vil vi komme ind i `if`-sætningen i hvert kald.

Rekursionstræet kommer derfor til at se ud som vist nedenfor. Parametrene i, j til kaldet står i hver knude. Som det ses, foretages kaldet lvd(A,3,2) to gange.



Kommentar: Det er ikke så interessant, at f.eks. kaldet lvd$(A, 4, 2)$ foretages flere gange, for i dette kald sammenligner man blot i med A.length, og 0 returneres. Dvs. man laver ikke ekstra arbejde i forhold til at slå op i en tabel.

Spørgsmål d:

```
int lvd(int[] A, int i, int j) {
    if (A[i][j] == -1){
        if (i == A.length) A[i][j] = 0;
        int max = lvd(A,i+1,j);
        if (A[i] > A[j]) {
            int m = lvd(A,i+1,i)+1;
            if (m > max) max = m;
        }
        A[i][j] = max;
    }
    return A[i][j];
}
```

Tabelstr.: $O(k^2)$

Køretid: $O(k^2)$, da hver indgang i tabellen beregnes i konstant tid, bortset fra de rekursive kald. Worst-case er, når hele følgen er voksende.

Opgave 4

Spørgsmål a: Mindste nøgle ligger i roden, da den ligger i et lige lag.

Største nøgle er den største blandt rodens to børn, da hver af disse er større end eller lig alle nøgler i sit undertræ samt roden.

Spørgsmål b:

1. Elementet indsættes "sidst" i min-max-hoben, ligesom i en alm. hob. Da den nye nøgle er mindre end sin forælder, giver dette ikke uorden i de ulige lag — dvs. enhver knude i et ulige lag er stadig større end eller lig med hver af sine efterkommere.

2. Da den nye nøgle er mindre end sin forælder, som ellers skal være mindst i sit undertræ, byttes de to om. Dette ændrer heller ikke på, at enhver knude i et ulige lag er større end eller lig alle sine efterkommere.

3. Nu er den nye nøgle mindre end sin bedsteforælder (roden), så disse to byttes om.

Hvis træet havde været højere, ville man fortsætte med at ombytte den indsatte nøgle k_1 med dens bedsteforælder k_2, så længe $k_1 < k_2$. Efter en sådan ombytning står enhver knude i et ulige lag rigtigt i forhold til sine efterkommere, og alle knuder i lige lag, evt. bortset fra k_1's forfædre, står rigtigt i forhold til deres efterkommere. Det ses af følgende.

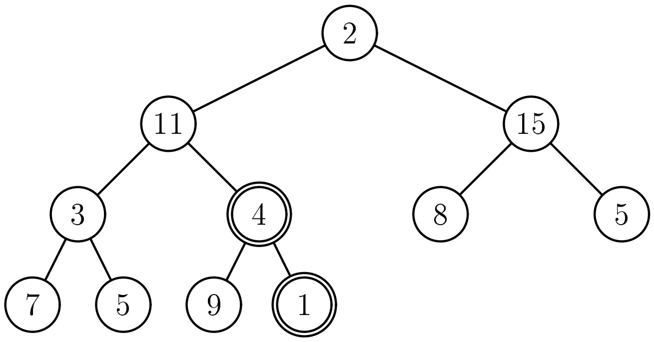
Før ombytningen gjaldt:

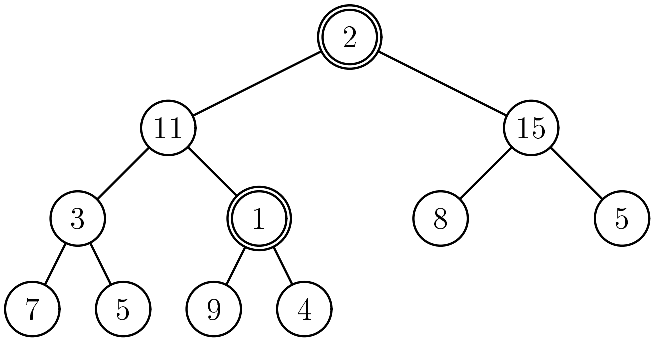
1. k_2 var mindre end eller lig sine efterkommere.

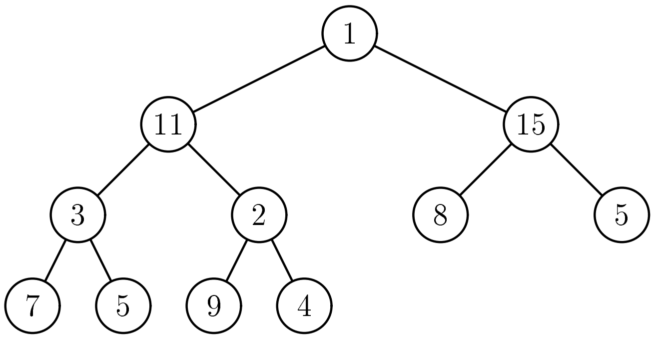
2. k_1 var mindre end eller lig k_2.

Det medfører, at efter ombytningen gælder:

- k_2 er mindre end eller lig sine efterkommere (følger af 1.).
- k_2 er mindre end eller lig sin forælder (følger af 1.).
- k_1 er mindre end eller lig sine efterkommere (følger af 1. og 2.).
- k_1 er mindre end eller lig sin forælder (følger af 1. og 2.).







Spørgsmål c: EXTRACT-MIN: Det sidste element i min-max-hoben flyttes op i roden. Først bobles det nye rod-element ned, 2 lag ad gangen. Når elementet står rigtigt i forhold til de andre min-lag, checkes om det også står rigtigt i forhold til max-lagene, og evt. bobles det op, til der også er orden i max-lagene. At boble op igennem max-lagene ændrer ikke på, at der er orden i min-lagene (samme begrundelse som i spm. b).

<u>EXTRACT-MIN</u>(A)
Hvis A.heapsize < 1
    error "heap underflow"
min $\leftarrow A[1]$
$A[1] \leftarrow A[A.\text{heapsize}]$
$A[A.\text{heapsize}] --$
HEAPIFY-MIN(A,1)
Returner min

<u>HEAPIFY-MIN</u>(A, i)
$\triangleright$ Bobl $A[i]$ ned, til der er orden i min-lagene
min-indeks $\leftarrow$ min-grandchild(i)
Så længe A[min-indeks] $> A[i]$
    Ombyt $A[i]$ og A[min-indeks]
    $i \leftarrow$ min-indeks
    min-indeks $\leftarrow$ min-grandchild(min-indeks)
$\triangleright$ Bobl $A[i]$ op, til der er orden i max-lagene
Hvis $A[i] > A[\text{parent}(i)]$
    Ombyt $A[i]$ og $A[\text{parent}(i)]$.
    $i \leftarrow \text{parent}(i)$
    Så længe $A[i] > \text{parent}(\text{parent}(i))$
        Ombyt $A[i]$ og $\text{parent}(\text{parent}(i))$
        $i \leftarrow \text{parent}(\text{parent}(i))$
Ellers hvis $A[i] > A[\text{min-child}(i)]$
    Ombyt $A[i]$ og A[min-child]

<u>min-child</u>(i)
Hvis $A[i]$ har mindst et barn, returneres indeks på det barn, som har den mindste nøgle (børnene har indeks $2i$ og $2i + 1$).
Ellers returneres i.

<u>min-grandchild</u>(i)
Hvis $A[i]$ har mindst et barnebarn, returneres indeks på det barnebarn, som har den mindste nøgle (børnebørnene har indeks $4i$ til $4i + 3$).
Ellers returneres i.

<u>parent</u>(i)
Hvis $A[i]$ ikke er roden, returneres indeks på $A[i]$'s forælder (dvs. $\lfloor i/2 \rfloor$).
Ellers returneres i.

<u>EXTRACT-MAX</u>(A)
Hvis A.heapsize < 1
    error "heap underflow"
max-indeks $\leftarrow$ max-child(1)
A[max-indeks] $\leftarrow$ A[A.heapsize]
A[A.heapsize] $--$
HEAPIFY-MAX(A,max-indeks)
Returner max

Resten er som for EXTRACT-MIN, blot med '$<$' og '$>$' byttet rundt og 'min' og 'max' byttet rundt.

Kommentar: Dette er "13-spørgsmålet", dvs. det er klart det sværeste i sættet. Det betyder dog ikke, at man helt skal opgive at svare på spørgsmålet; man kan sagtens få nogle point for et svar, som ikke er fuldstændig korrekt/har alle detaljer med.