Skriftlig Eksamen
Datastrukturer og Algoritmer (DM02)

Institut for Matematik og Datalogi
Odense Universitet

Mandag den 12. januar 1998, kl. 9–13

Alle sædvanlige hjælpemidler (lærebøger, notater, etc.) samt brug af lommeregner er tilladt.

Eksamenssættet består af 4 opgaver på 6 nummererede sider (1–6). Fuld besvarelse er besvarelse af alle 4 opgaver.

De enkelte opgavers vægt ved bedømmelsen er angivet i procent. Der må gerne refereres til algoritmer og resultater fra lærebogen inklusive øvelsesopgaverne. Specielt må man gerne begrunde en påstand med at henvise til, at det umiddelbart følger fra et resultat i lærebogen (hvis dette altså er sandt!). Henvisninger til andre bøger (udover lærebogen) accepteres ikke som besvarelse af et spørgsmål.

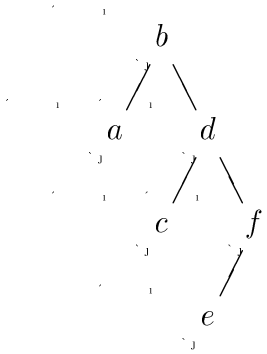
Bemærk, at hvis der er et spørgsmål i en opgave, man ikke kan besvare, må man gerne besvare de efterfølgende spørgsmål og blot antage, at man har en løsning til de foregående spørgsmål.

Opgave 1 (40%)

Vi har et antal nøgler, som vi vil have anbragt i et søgetræ. Vi véd hvor ofte, der vil blive søgt efter de forskellige nøgler, og ønsker at udnytte denne information til at bygge et optimalt søgetræ. Nedenfor ses en række nøgler og et *besøgstal* for hver nøgle.

Nøgle	a	b	c	d	e	f
Besøgstal	30	42	3	18	2	5

Summen af besøgstallene i dette eksempel er 100, og tabellen fortæller dermed, at ud af 100 søgninger vil man søge efter a 30 gange, b 42 gange, c 3 gange, osv. Et søgetræ, der kunne være et fornuftigt bud på et optimalt arrangement, kunne være følgende:



Dette træ har *besøgsværdi* 70. Dette er det totale antal kanter, der skal følges for at lave alle de søgninger, der er angivet i tabellen. Dvs. 30 a søgninger, 42 b søgninger, osv. Besøgsværdien 70 er altså fundet som $30 \cdot 1 + 42 \cdot 0 + 3 \cdot 2 + 18 \cdot 1 + 2 \cdot 3 + 5 \cdot 2$.

Et *optimalt* søgetræ med hensyn til en given tabel er et søgetræ med mindst mulig besøgsværdi.

Spørgsmål a: Man kunne tro, at et optimalt søgetræ altid kan laves ved at anbringe nøglen med størst besøgstal i roden og fortsætte rekursivt med at lave et venstre og højre undertræ af de nøgler, der er mindre end, henholdsvis større end rodens nøgle. Vis ved et eksempel, at dette ikke er tilfældet. □

Nedenfor defineres en funktion OBV, der skal beregne besøgsværdien af et optimalt søgetræ. Funktionen kaldes med en liste af besøgstal, der står i rækkefølge svarende til sorterede nøgler. F.eks. [30, 42, 3, 18, 2, 5] fra eksemplet. Funktionen returnerer et *par* bestående af *det totale antal søgninger* samt *træets besøgsværdi*.

```
def OBV(bt, first, last):
    if first > last:
        return (0,0)
    elif first == last:
        return (bt[first], 0)
    else:
        MinVal = MAXINT
        for i in range(first, last + 1):
            r = bt[i]
            (cl,vl), (cr,vr) = OBV(bt, first, i-1), OBV(bt, i+1, last)
            c, v = cl + r + cr, vl + cl + vr + cr
            if v < MinVal:
                Count, MinVal = c, v
    return (Count,MinVal)

print OBV([30, 42, 3, 18, 2, 5], 0, 5) # udskriver (100, 70)
```

Spørgsmål b: Forklar kort i ord, hvordan OBV virker. Herunder hvad **for**-løkken prøver igennem, og hvad `cl + r + cr` samt `vl + cl + vr + cr` beregner. ☐

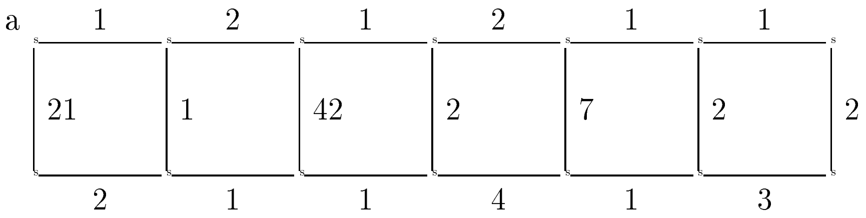
Spørgsmål c: Vis (evt. gennem et eksempel), at OBV beregner samme delresultater adskillige gange. ☐

Spørgsmål d: Anvend dynamisk programmering til at ændre OBV, så delresultater kun beregnes én gang. ☐

Spørgsmål e: Hvad bliver tidskompleksiteten af algoritmen, når der anvendes dynamisk programmering? ☐

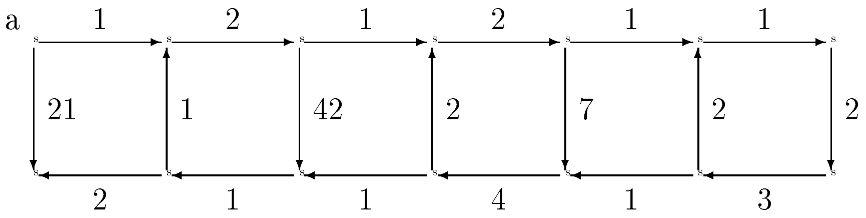
Opgave 2 (25%)

Betragt nedenstående ikke-orienterede, vægtede graf.



Spørgsmål a: Angiv ved en tegning et lettest udspændende træ for grafen, samt dets vægt. □

Vi ser nu på en klasse af vægtede *orienterede* grafer kaldet *frem-og-tilbage* grafer. Knuderne i en sådan graf er organiseret i to lige lange rækker, og der er et ulige antal knuder (mindst 3) i hver række. I øverste række går kanterne mod højre og i nederste række mod venstre. Desuden er der én kant mellem hvert par af knuder i samme position i de to rækker. Disse kanter går skiftevis ned og op (startende med ned). Nedenstående graf er et eksempel på en frem-og-tilbage graf.



Spørgsmål b: Forklar, hvordan man kan beregne korteste veje fra øverste venstre hjørne (markeret med "a" i eksemplet) til alle andre knuder i en sådan graf i tid $O(n)$; dvs. i tid proportional med antallet af knuder n i grafen. (Dijkstra's algoritme er *ikke* hurtig nok.) □

Det følgende spørgsmål handler *ikke* om frem-og-tilbage grafer.

Dijkstra's algoritme forudsætter, at vægtene på kanterne er ikke-negative. Det er klart, at hvis der er negative kredse, så er problemet slet ikke veldefineret, da stilængderne kan gøres vilkårligt lave. Husk, at en kreds er en sti, der på et tidspunkt vender tilbage til udgangspunktet. Den kaldes negativ, hvis summen af kanternes vægte er negativ.

Spørgsmål c: Find et eksempel, der viser, at Dijkstra's algoritme faktisk ikke virker, hvis der er negative vægte, selv hvis der ikke er negative kredse (det er ikke nødvendigt at bruge ret mange knuder). Det skal forklares, hvordan Dijkstra's algoritme afvikles på det eksempel, du angiver. □

Opgave 3 (15%)

Vi har givet en sorteret liste af heltal som f.eks. følgende:

```
A = [0,1,4,5,6,7,9,10,11,16,17,19,21,25,27,28,29]
```

Nu laves der et antal opdateringer, der hver især består i, at talværdien på en plads i listen forøges eller formindskes. Listen er herefter ikke nødvendigvis sorteret.

Listen

```
B = [0,1,4,3,6,7,12,10,11,16,17,15,21,25,26,28,29]
```

er blevet ændret på fire pladser i forhold til den oprindelige liste `A`; nemlig på index 3, 6, 11 og 14.

Vi véd om listerne, at der ingen dubletter er (heller ikke efter opdateringerne), og at der efter en opdatering er mindst to uændrede elementer på hver side af et ændret.

Vi er interesserede i at få en udskrift af index på ændrede elementer. Da opdateringerne foretages direkte i den oprindelige liste, kendes den oprindelige liste ikke længere, så ændrede elementer, der stadig står sorteret, kan naturligvis ikke opdages. Hvis man ikke kan afgøre, hvilket af to elementer, der er blevet ændret, accepterer vi at få begge index udskrevet.

På listen `B` ovenfor ville vi forvente følgende udskrift:

```
2 3
6
11
```

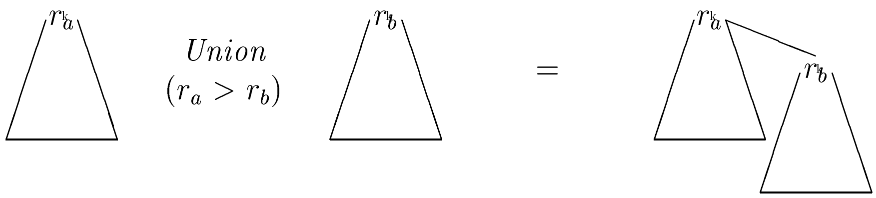
Vi kan nemlig ikke konstatere ændringen på index 14, og vi kan heller ikke se, om den ændring, der faktisk skete på index 3, skete på index 2 eller 3.

Spørgsmål a: Skriv en PYTHON funktion, der tager en opdateret liste som argument og udskriver den omtalte liste af index, der har ændret sig. □

Opgave 4 (20%)

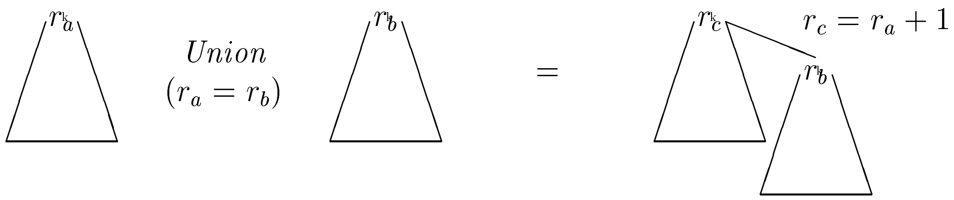
I forbindelse med disjunkte mængder (disjoint sets) sikrede vi logaritmisk højde ved at sammensætte træer afhængig af deres størrelse. I denne opgave ser vi på et alternativ til den metode, hvor vi i stedet for at gemme antallet af knuder, giver hver knude en *rang*.[1]

Når en ny knude laves (operationen *MakeSet*), gives knuden rang 0. Når to træer sættes sammen med *Union*, gøres roden med mindst rang til barn af den anden rod som nedenfor:



Der er altså ingen knuder, der får deres rang ændret.

Hvis rødderne har samme rang, vælges vilkårligt, og roden efter sammensætningen får sin rang talt op med én:



Operationen *Find* ændrer ikke strukturen.

Spørgsmål a: Vis ved induktion på antallet af operationer, der udføres, at der er mindst 2^r knuder i et træ med rang r. $\square$

Spørgsmål b: Argumentér for, at hvis en knude med rang r har et barn med rang r', så er $r > r'$. $\square$

Spørgsmål c: Vis, at *Find* er $O(\log n)$, hvor n er antallet af elementer i strukturen. $\square$

[1] Dette har intet at gøre med den rang, der defineres i forbindelse med splay trees.