Skriftlig Eksamen
Datastrukturer og Algoritmer (DM02)

Institut for Matematik og Datalogi
Odense Universitet

Torsdag den 2. januar 1997, 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 (30%)

Vi ser på rektangler af højde 2 og længde $n \geq 2$. Se eksempel nedenfor.

0	5	1	7	6	1	2	1	0	6	3	9
0	3	0	4	2	5	3	3	6	7	5	6

To felter er *naboer*, hvis de står ved siden af hinanden eller ovenover hinanden. En *nabovej* er en sekvens af felter, hvor alle par af på hinanden følgende felter er naboer. Vejen skal gå fra et af de to yderfelter til venstre til et af de to yderfelter til højre. Yderfelterne til venstre er altid 0, mens alle andre felter har værdier større end eller lig med 0. Nedenfor ses et eksempel på en nabovej.

0	5	1	7	6	1	2	1	0	6	3	9
0	3	0	4	2	5	3	3	6	7	5	6

Omkostningen af en nabovej er summen af tallene i dens felter. Nabovejen ovenfor har altså en omkostning på 62.

I denne opgave er vi interesseret i at finde omkostningen af den (en) billigste nabovej. Følgende rekursive function kan bruges til det. Parameteren `A` er et rektangel som illustreret ovenfor. Mere præcist er `A` en liste af små lister, hvor de små lister har længde to. F.eks. er `A[0]` lig med `[0,0]`, og `A[3]` lige med `[4,7]`.

```
def f(A, x, y):
    if x == len(A):
        return 0
    else:
        h = f(A, x + 1, y)
        v = f(A, x + 1, 1 - y)
        return A[x][y] + min(h, A[x][1 - y] + v)
```

Spørgsmål a: Forklar kort i ord, hvad `f(A, x, y)` beregner, og hvordan den gør det. Herunder hvad `h` og `v` bruges til, samt hvad `1 - y` skal betyde. □

Spørgsmål b: Hvad er kompleksiteten af algoritmen som funktion af n? Argumenter for dit svar. □

Spørgsmål c: Vis, at `f` kaldes med præcis de samme parametre adskillige gange (f.eks. ved at vise et eksempel). □

Spørgsmål d: Lav en bedre udgave ved at anvende dynamisk programmering. Hvad bliver kompleksiteten af løsningen? Argumenter for dit svar. □

Opgave 2 (30%)

Nedenfor er angivet en funktion, der sorterer elementer i en liste i ikke-aftagende orden.

```
def sort(A):
   i = 0
   while i < len(A): # I
      for j in range(len(A)-(i+1)):
         if A[j] > A[j+1]:
            A[j], A[j+1] = A[j+1], A[j]
      i = i + 1
```

En sorteringsmetode er *stabil*, hvis elementer med samme nøgle altid vil stå i samme indbyrdes rækkefølge i det sorterede output som i det oprindelige input.

Spørgsmål a: Er sorteringsmetoden ovenfor stabil? Argumenter for dit svar. □

Vi definerer nu, at en liste er i *c-uorden*, hvor c er et heltal, hvis ethvert element i listen er anbragt højst c pladser fra den plads, det ville stå på, hvis listen var sorteret.

Spørgsmål b: Angiv en en lille ændring i algoritmen, sådan at den nu kører i tid $O(n)$ på lister, der er i c-uorden (den behøver *ikke* længere virke på lister, der ikke er i c-uorden). Argumenter for at ændringen virker. □

Vi ser nu igen på den oprindelige algoritme (resten af opgaven antager altså *ikke*, at input er i c-uorden).

Spørgsmål c: Argumenter for, at

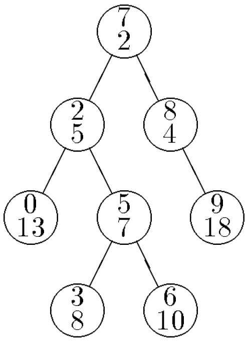
$$i \leq len(A) \;\wedge\; \begin{array}{l}\text{udsnittet i } A \text{ fra og med } len(A)-i \text{ til og med } len(A)-1 \text{ er} \\ \text{sorteret og indeholder de største elementer i } A\end{array}$$

er en invariant for `while`-løkken (udsnittet er tomt første gang, man kommer til `while`-løkken). □

Spørgsmål d: Vis, at funktionen er korrekt. Dvs. at den terminerer, og at listen på det tidspunkt er sorteret. □

Opgave 3 (20%)

I denne opgave skal vi se på en implementation af en symboltabel kaldet en *treap* (sammentrækning af *tree* og *heap*). Implementationen er i form af et søgetræ med elementer, der både har en nøgle og en prioritet. Vi antager dog, at ingen nøgler får tildelt samme prioritet. Nøglerne skal nu opfylde den sædvanlige søgetræsbetingelse (binary search tree invariant), mens prioriteterne skal opfylde den sædvanlige heap-betingelse (heap invariant). Nedenfor ses en treap. Nøglerne står øverst i knuderne og prioriteterne nederst.



Knuderne i træet er objekter af følgende type:

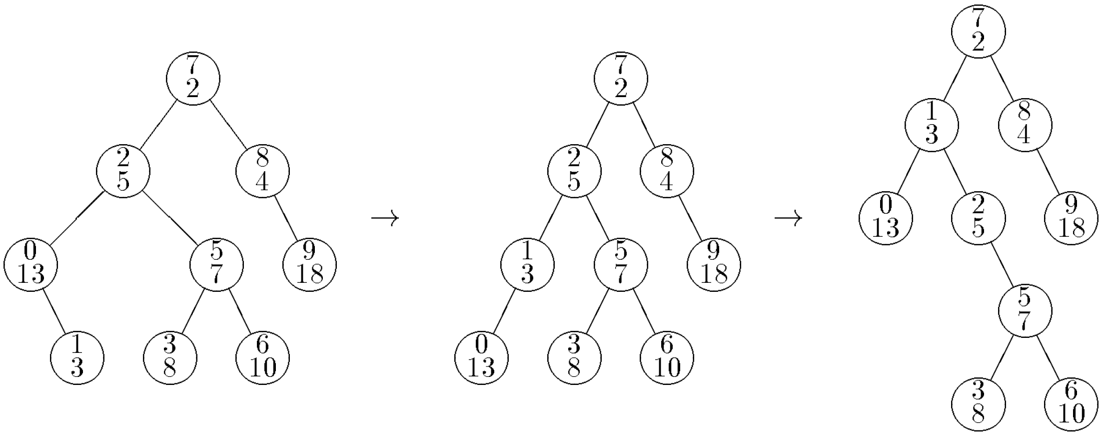
```
class Node:

    def __init__(self, k, p, l, r):
        self.key = k
        self.pri = p
        self.left = l
        self.right = r
```

Variablen `root` antages i det følgende at referere til træets rod.

Spørgsmål a: Skriv en funktion `MinKeys`, så kaldet `MinKeys(root, m)` udskriver præcis de nøgler, der er mindre end eller lig med `m`. Kompleksiteten af funktionen skal være proportional med træets højde plus antal nøgler, der er mindre end `m`. □

Spørgsmål b: Skriv en funktion `MinPris`, så kaldet `MinPris(root, m)` udskriver præcis de prioriteter, der er mindre end eller lig med `m`. Kompleksiteten af funktionen skal være proportional med antallet af prioriteter, der er mindre end `m`. □

Indsættelse i en treap klares som følger: Først indsættes elementet som sædvanligt i et søgetræ efter nøglen. Derefter roteres der (venstre- og højrerotationer), indtil heap-betingelsen igen er opfyldt. Indsætter vi f.eks. elementet med nøgle 1 og prioritet 3, laves der en venstrerotation fulgt af en højrerotation. Se nedenfor.

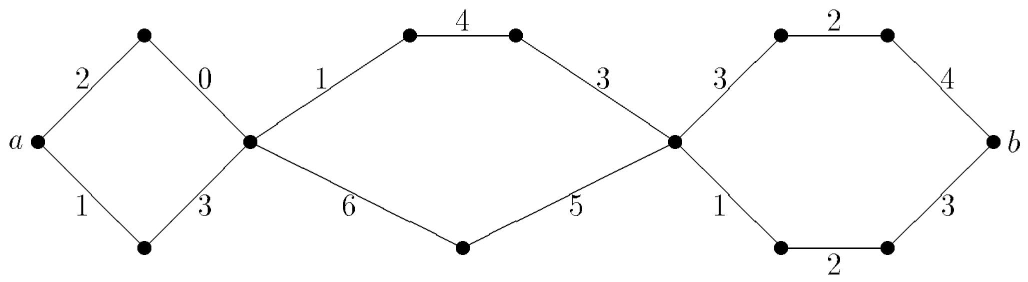


Spørgsmål c: Vis, at hvis n elementer indsættes i en oprindeligt tom treap i voksende rækkefølge med hensyn til nøglerne, så foretages der et amortiseret konstant antal rotationer pr. indsættelse.

Vink: Hvad sker der med længden af højrestien under en indsættelse? □

Opgave 4 (20%)

Opgaven drejer sig om en type ikke-orienterede, vægtede grafer, kaldet *ringgrafer*, der består af en kæde af ringe. Der er mindst to ringe, og enhver ring indeholder mindst tre kanter. Vægtene er ikke-negative heltal. Man har givet referencer til to punkter, a og b; ét i hver yderring. Nedenfor ses et eksempel, hvor kæden består af tre ringe



Lad n betegne antallet af knuder i grafen og m antallet af kanter.

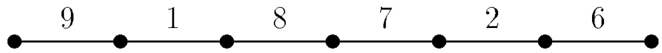
Spørgsmål a: Vis, at der findes en konstant c, så $m \leq cn$. □

Antag, at grafen er repræsenteret ved den sædvanlige kantlisterepræsentation. Dvs. hver knude har tilknyttet en liste af referencer til præcis de andre knuder, den er forbundet til.

Spørgsmål b: Forklar kort i ord, hvordan man altid kan finde vægten (cost) af et letteste udspændende træ (minimum spanning tree) for en ringgraf i tid $O(n)$. □

I Kingston vises det, at kompleksiteten af Kruskal's algoritme er $O(m \log m)$ for sammenhængende grafer. Det kunne dog sagtens være, at algoritmen afvikles hurtigere på særlige (simple) graftyper.

En *liniegraf* er en ikke-orienteret, sammenhængende, vægtet graf, der består af en sekvens af punkter, hvor de to endepunkter kun har en kant, mens resten har to kanter; én til hver naboknude. Et eksempel ses nedenfor.



Det er klart, at det letteste udspændende træ for en liniegraf simpelt hen består af alle kanter. Følgende spørgsmål drejer sig om at vise, at Kruskal's algoritme ikke en gang er lineær på liniegrafer.

Som prioritetskø og disjunkte mængder i Kruskal's algoritme anvendes naturligvis implementationerne fra bogen ("heap" og "Galler-Fischer").

Spørgsmål c: Argumenter for, at man for ethvert k kan lave en liniegraf med $n \geq k$ knuder og tildele kanterne vægte, sådan at Kruskal's algoritme vil tage tid proportionalt med $n \log n$. □