Skriftlig Eksamen
Datastrukturer og Algoritmer (DM02)

Institut for Matematik og Datalogi
Odense Universitet

Fredag den 5. januar 1996, kl. 9–13

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

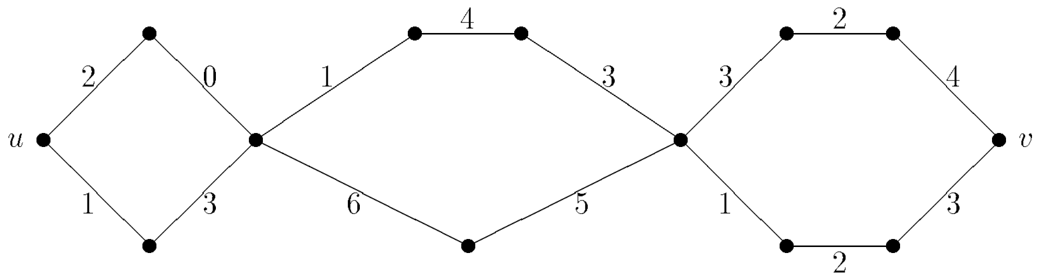
Eksamenssættet består af 4 opgaver på 5 nummererede sider (1–5). Fuld besvarelse er besvarelse af alle 4 opgaver.

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 (15%)

Opgaven drejer sig om en type ikke-orienterede, vægtede grafer, kaldet *bæltegrafer*. En bæltegraf er en kæde af ringe med to særlige knuder kaldet *endepunkterne*. Der findes ét endepunkt i hver af de yderste ringe. Nedenfor ses et eksempel, hvor kæden består af tre ringe, og endepunkterne er u og v.



I det følgende bruges n for antallet af knuder, m for antallet af kanter, og k for antallet af ringe i en bæltegraf. Antag, at grafen er repræsenteret ved, at hver knude har tilknyttet en liste af alle naboknuder (adjacency lists).

Spørgsmål a: Hvad er m udtrykt ved n og k i en vilkårlig bæltegraf. □

Spørgsmål b: Forklar i ord, hvordan man altid kan finde den korteste vej mellem endepunkterne (u og v) i en bæltegraf i tid $O(n)$. □

Opgave 2 (25%)

Opgaven drejer sig om sortering. Vi har givet en liste af heltal A med index fra 0 til n. Elementerne fra index 1 til n skal sorteres. Det antages, at A[0] findes, og at tallet i A[0] er mindre end alle andre tal i A. Vi antager, at A ikke indeholder dubletter. Dvs. alle tal er forskellige. Nedenfor er angivet to sorteringsalgoritmer.

Udvalgssortering

```
for i = 1 to n−1 do
    m = i
    for j = i+1 to n do
        if A[j] < A[m] then
            m = j
    if i ≠ m then
        temp = A[i]
        A[i] = A[m]
        A[m] = temp
```

Indsættelsessortering

```
for i = 1 to n do
    j = i
    x = A[j]
        while A[j−1] > x do
            A[j] = A[j−1]
            j = j−1
        A[j] = x
```

Spørgsmål a: Hvad er kompleksiteten af de to sorteringsalgoritmer, hvis de udelukkende køres på lister, der i forvejen *er* sorterede? For hver sorteringsalgoritme angives én af de tre muligheder: $O(n^2)$, $O(n \log n)$ eller $O(n)$, samt en kort begrundelse (angiv i hvert tilfælde den mindste øvre grænse for kompleksiteten). □

Vi definerer nu, at en liste er i *c-uorden*, hvor c er et heltal, hvis enhver nøgle i listen er anbragt højst c pladser fra den plads, den ville stå på, hvis listen var sorteret. I det følgende lille eksempel er A[0] ikke medtaget.

Listen

3	2	0	7	4	6	9	8

er i 2-uorden, som det ses ved at sammenligne med den sorterede liste

0	2	3	4	6	7	8	9

Det ses, at 3 står to pladser væk fra sin rigtige plads, 2 står rigtigt, 0 står to pladser forkert, 7 står også to plads forkert, 4 står én plads forkert, osv. Listen er derimod ikke i 1-uorden, netop fordi der er elementer, der står *mere* end én plads forkert. Hvis en liste er i 0-uorden, er den altså helt sorteret.

Spørgsmål b: Hvad er kompleksiteten af de to sorteringsalgoritmer, hvis de udelukkende køres på lister, der er i c-uorden for en konstant $c > 0$? For hver sorteringsalgoritme angives én af de tre muligheder: $O(n^2)$, $O(n \log n)$ eller $O(cn)$, samt en kort begrundelse (angiv i hvert tilfælde den mindste øvre grænse for kompleksiteten). □

Spørgsmål c: Angiv, om der kan laves ganske små ændringer i algoritmerne, så kompleksiteten på lister, der er i c-uorden ($c > 0$), kommer ned på $O(cn)$ (hvis det ikke sker automatisk). Bemærk, at algoritmerne nu *ikke* behøver at virke på input, der ikke er i c-uorden. Forklar kort, hvori ændringerne består. □

Opgave 3 (25%)

Opgaven drejer sig om at udregne den mest profitable plan for at investere et antal kroner i en række virksomheder.

Vi har givet d virksomheder nummereret fra $1 \ldots d$. Desuden har vi funktionen *udbytte* til rådighed. Vi antager, at $udbytte(p, c)$ for vilkårlige p og c i konstant tid returnerer det forventede udbytte af at investere p kroner i virksomhed c. Alle værdier er heltal større end eller lig med nul.

Ved at kalde $invester(p, 1)$, hvor pseudo-koden for *invester* er givet nedenfor, kan vi få beregnet det maksimale forventede udbytte af at investere p kroner i de d virksomheder.

```
function invester(p,c)
    if c > d then
        return 0
    else
        max = −1
        for i = 0 to p do
            temp = udbytte(i,c) + invester(p−i,c+1)
            if temp > max then
                max = temp
        return max
```

Spørgsmål a: Forklar kort i ord, hvad *invester*(p, c) beregner. □

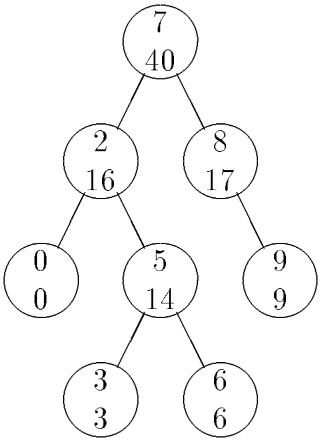
Spørgsmål b: Vi betragter nu kaldet *invester*$(p, 1)$. Gør rede for, at de samme resultater beregnes flere gange. Dvs. at der findes q og c, så kaldet *invester*(q, c) foretages adskillige gange. □

Kompleksiteten af algoritmen er eksponentiel, netop fordi de samme kald laves gentagne gange.

Spørgsmål c: Konstruér en bedre løsning ved at anvende dynamisk programmering. Hvad bliver kompleksiteten udtrykt i p og d? □

Opgave 4 (35%)

Opgaven drejer sig om søgetræer. Vi ser på søgetræer, hvor knuderne som sædvanligt har en *nøgle* og en *værdi*. Vi bestemmer os for, at værdien i en knude skal være summen af alle nøglerne i knudens undertræ (husk, at det inkluderer knuden selv). Nedenfor ses et eksempel. Nøglerne står øverst og værdifeltet nederst. Vi antager, at ingen nøgler optræder to gange.

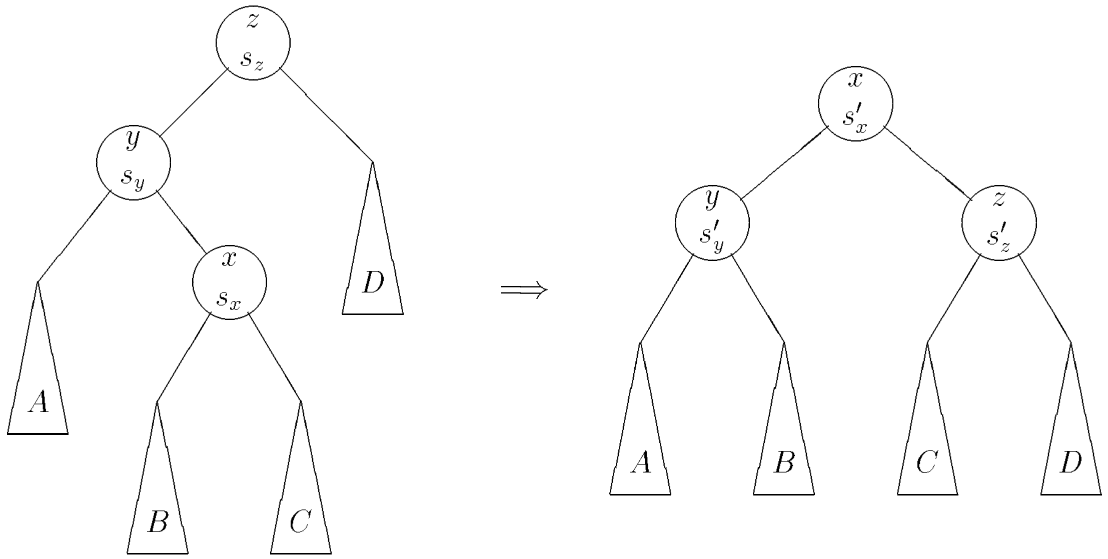


En knude har altså fire felter: *key*, *val*, *left* og *right* til henholdsvis nøgle, værdi, venstre barn og højre barn. Desuden kan man antage, at variablen *root* refererer til træets rod.

Spørgsmål a: Beskriv detaljeret (gerne i PYTHON eller MODULA-2), hvordan man for en givet nøgle k kan beregne summen af alle nøgler *større end eller lig med* k. Kompleksiteten skal være begrænset af en konstant gange træets højde. NB! k behøver ikke findes i træet. □

Spørgsmål b: Forklar, hvordan man givet *to* nøgler $k_1 < k_2$ kan beregne summen alle nøgler i træet, der ligger *mellem* k_1 og k_2 (begge nøgler inklusive). Kompleksiteten skal være begrænset af en konstant gange træets højde. Vink: anvend algoritmen fra spørgsmål a. □

Følgende transformation på et søgetræ kendes fra Kingston side 112.



Spørgsmål c: Forklar, hvordan værdifelterne kan bringes i orden i konstant tid, hvis ovenstående transformation foretages et sted i træet. Man kan antage, at X, Y og Z er variabler, der refererer til knuderne med nøglerne henholdsvis x, y og z efter transformationen. $\square$

Vi er nu interesseret i en operation *average*, som givet to nøgler $k_1 < k_2$ finder *gennemsnittet* af nøgleværdier i træet, der ligger mellem k_1 og k_2 (begge nøgler inklusive). I eksemplet først i opgaven med $k_1 = 4$ og $k_2 = 8$ fås et gennemsnit på $\frac{26}{4}$, da nøglerne 5, 6, 7 og 8 ligger mellem 4 og 8.

Spørgsmål d: Beskriv en struktur, der kan understøtte operationerne *insert* og *delete*, som de kendes fra Kingston side 103–104, samt operationen *average*, alle i tid $O(\log n)$ amortiseret, hvor n er antallet af elementer i strukturen på et givet tidspunkt. Forklar, hvordan strukturen vedligeholdes under operationerne. $\square$