Skriftlig Eksamen
Datastrukturer og Algoritmer (DM02)

Institut for Matematik og Datalogi
Syddansk Universitet, Odense

Onsdag den 31. januar 2001, kl. 9–13

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

Eksamenssættet består af 4 opgaver på 7 nummererede sider (1–7). 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. 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 (så vidt det er muligt) besvare de efterfølgende spørgsmål og blot antage, at man har en løsning til de foregående spørgsmål.

Opgave 1 (25%)

Et *palindrom* er en tekst, der er ens forfra og bagfra. Eksempler er “bob”, “abba” “kajak” og “enafdemderredmedfane”.

Vi ser nu på følgende problem: Givet en streng, hvad er det mindste antal palindromer, strengen kan opdeles i? Da en streng, der består af kun ét tegn, er et palindrom, findes der altid en opdeling i et antal palindromer, og dermed findes der altid et mindste antal.

Nedenfor ses nogle af de mulige opdelinger af strengen “parallelle” i palindromer:

```
"parallelle" = "p" + "ara" + "l" + "lel" + "l" + "e"

"parallelle" = "p" + "ara" + "ll" + "elle"

"parallelle" = "p" + "ara" + "ll" + "e" + "ll" + "e"
```

Den, der består af færrest, er den i midten, og man kan faktisk ikke opdele denne streng i færre end 4.

Nedenstående PYTHON-funktion beregner dette minimale antal. Funktionskaldet `MP("parallelle", 0, 10)` returnerer altså tallet 4. Vi antager, at vi har en funktion `IsPalindrome` til rådighed, der givet en streng og to index i og j afgør, om delstrengen fra og med index i til (men ikke med) index j er et palindrom.

```
def MP(s, i, j):
    if IsPalindrome(s, i, j):
        return 1
    else:
        m = j - i
        for k in range(i + 1, j):
            p = MP(s, i, k) + MP(s, k, j)
            if p < m: m = p
        return m
```

Spørgsmål a: Forklar, hvordan resultatet beregnes. Herunder hvad de indgående variabler anvendes til. □

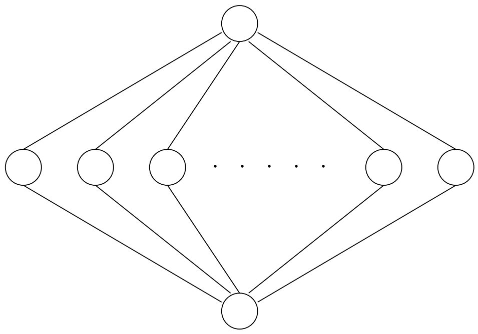
Spørgsmål b: Vis (gerne ved et eksempel), at MP generelt under den rekursive beregning af svaret kaldes med samme argument flere gange. □

Spørgsmål c: Lav en ny version af MP med dynamisk programmering. □

Spørgsmål d: Redegør for tabelstørrelse og kompleksitet i den udgave, hvor der anvendes dynamisk programmering. □

Opgave 2 (25%)

I denne opgave vil vi se på *diamantgrafer*. En diamantgraf er en vægtet ikke-orienteret graf af en bestemt form som illustreret nedenfor (vægtene er ikke vist).



En diamantgraf med n knuder, $n \geq 3$, består altså af en top-knude, en bund-knude, samt $n-2$ mellem-knuder, der hver har en kant til top-knuden og en kant til bund-knuden.

Alle vægtene er større end eller lig med nul.

I denne opgave er vi interesserede i at beregne et letteste udspændende træ (LUT) for diamantgrafer.

Se på nedenstående algoritme, som vi vil kalde Algoritme Forkert.

- Find en mellem-knude u, der har mindst sum af vægte på sine to kanter.
- Inkludér u's to kanter i LUT.
- For alle mellem-knuder $v \neq u$: inkludér v's letteste kant i LUT.

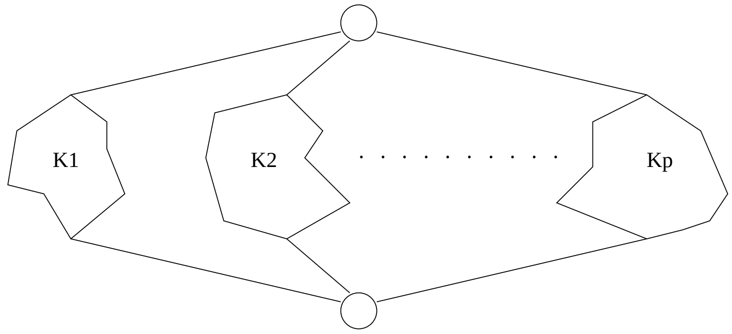
Spørgsmål a: Vis, at Algoritme Forkert ikke altid beregner et letteste udspændende træ for en diamantgraf. □

Spørgsmål b: Vis, at vægten af et træ beregnet af Algoritme Forkert kan være vilkårligt tæt på 50% tungere end det letteste. □

Vi antager nu, at vi anvender en kantlisterepræsentation. Dvs. at hver knude har en liste af sine naboknuder.

Spørgsmål c: Beskriv selv en $O(n)$ algoritme, der faktisk beregner letteste udspændende træ for en diamantgraf. Argumentér for korrekthed og kompleksitet. □

Vi ser nu på en ny klasse af grafer, vi kunne kalde *overordnede diamantgrafer*; stadig vægtede og ikke-orienterede. Denne type grafer har overordnet form som en diamantgraf, men det, der før var en mellem-knude, kan nu være en helt generel sammenhængende graf. Se eksemplet nedenfor, hvor vi har p generelle grafer K_1 til K_p.



Det er velkendt, at vi på en sammenhængende graf med m kanter kan beregne et letteste udspændende træ i tid $O(m \log m)$, f.eks. vha. Kruskals algoritme.

Spørgsmål d: Antag, at hver komponent har præcis $\log m$ kanter, når man medregner de to kanter fra komponenten til henholdsvis top- og bund-knuden (dvs. at der er $p = \frac{m}{\log m}$ komponenter).

Forklar, hvordan man kan beregne letteste udspændende træ for en sådan overordnet diamantgraf i tid asymptotisk bedre end $O(m \log m)$.

Argumentér for, hvordan kompleksiteten opnås. □

Opgave 3 (25%)

Et *palindrom* er en tekst, der er ens forfra og bagfra. Eksempler er "bob", "abba" "kajak" og "enafdemderredmedfane".

Som det også kan ses af udsagnet *Final*, afgør nedenstående algoritme, om en streng t er et palindrom.

Husk, at `t[-(i+1)]` blot er kortere notation for `t[len(t)-(i+1)]`.

I hele denne opgave står x/y for *heltalsdivision*. Dvs. at hvis y ikke går op i x, rundes resultatet ned til nærmeste hele tal. F.eks. er $7/2 = 3$.

```
# Pre: true
i = 0
while i < len(t)/2 and t[i] == t[-(i+1)]: # I
    i = i + 1
# Post: (i = len(t)/2) ⇔ ∀j ∈ {0,...,len(t)/2 − 1}: t[j] = t[−(j + 1)]
IsPal = (i == len(t)/2)
# Final: IsPal ⇔ ∀j ∈ {0,...,len(t)/2 − 1}: t[j] = t[−(j + 1)]
```

Spørgsmål a: Vis, at følgende udsagn I er en invariant for while-løkken:

$$i \leq len(t)/2 \ \wedge\ \forall j \in \{0, \ldots, i-1\}\colon\ t[j] = t[-(j+1)]$$

□

Spørgsmål b: Vis terminering, blandt andet ved at angive en termineringsfunktion. □

Spørgsmål c: Vis, at umiddelbart efter while-løkken gælder udsagnet *Post*. □

Opgave 4 (25%)

Vi ser på opdelingen af de naturlige tal i et endeligt antal disjunkte intervaller, der til sammen udgør alle de naturlige tal. Det sidste interval vil derfor altid indeholde "resten" af tallene, så det vil være uendeligt stort.

Her er et eksempel:

$$\begin{array}{l} [0,1,2,3,4] \\ [5] \\ [6,7,8] \\ [9,10] \\ [11,12,13,\ldots] \end{array}$$

Det er ikke hensigtmæssigt explicit at gemme store intervaller (og da slet ikke et uendeligt interval), så derfor vil vi se på en alternativ repræsentation, som vi vil kalde en *delepunktsrepræsentation*.

Eksemplet ovenfor har følgende delepunktsrepræsentation:

$$\{0,5,6,9,11\}$$

Dvs. at et interval repræsenteres ved det første tal i intervallet, som kaldes intervallets repræsentant.

Vi vil nu interessere os for følgende tre operationer, der alle skal kunne kaldes med ethvert naturligt tal:

> `rep(`x`)`: Returnerer repræsentanten for det interval, som x ligger i.
>
> `cut(`x`)`: Gør ingenting, hvis x er repræsentanten for det interval x ligger i. Ellers opdeles intervallet $[a,\ldots,b]$ i to intervaller $[a,\ldots,x-1]$ og $[x,\ldots,b]$.
>
> `paste(`x`)`: Gør ingenting, hvis x tilhører det sidste interval. Ellers kombineres det interval, x ligger i, med det efterfølgende interval.

Spørgsmål a: Forklar, hvad sker der med delepunktsrepræsentationen ved en `cut` og ved en `paste`. □

Vi lader n betegne antallet af intervaller på et givet tidspunkt.

Spørgsmål b: Forklar, hvordan man vha. et splay-træ kan lave en datastruktur for dette problem, så alle tre operationer få kompleksitet amortiseret $O(\log n)$. □

Vi vil nu yderligere interessere os for følgende operation:

`next(`x`,`s`)`: Gør ingenting, hvis x tilhører det sidste interval. Ellers returneres repræsentanten for det første interval af længde mindst s, der kommer efter det interval, som x ligger i.

Spørgsmål c: Forklar, hvordan man kan lave en datastruktur, så alle fire operationer på intervalopdelinger kan udføres i amortiseret $O(\log n)$. $\square$