Eksamen i IN2010 høsten 2023

15. Desember 2023

Om eksamen

- Eksamen består av en bitteliten oppvarming, etterfulgt av to hoveddeler.
- Den første delen består av små oppgaver, som rettes automatisk. Det gjøres ingen forskjell mellom ubesvart og feil svar; det betyr at det lønner seg å svare på alle oppgavene.
- Den andre delen består av litt større oppgaver hvor du i større grad må programmere (pseudokode), skrive og resonnere.
- Ingen hjelpemidler er tillatt.
- An English translation of the full set of exercises is attached as a PDF.

Kommentarer og tips

- Det kanskje viktigste tipset er *å lese oppgaveteksten svært nøye*.
- Pass på at du svarer på nøyaktig det oppgaven spør om.
- Redgjør for eventuelle antagelser du gjør.
- Pass på at det du leverer fra deg er klart, presist og enkelt å forstå, både når det gjelder form og innhold.
- Hvis du står fast på en oppgave, bør du gå videre til en annen oppgave først.
- Alle implementasjonsoppgaver skal besvares med *pseudokode*. Det viktige er at pseudokoden er *lett forståelig, entydig og presis*.
- En *lett forståelig, entydig og presis* forklaring med naturlig språk, kan være mer poenggivende enn pseudokode som er vanskelig å forstå, tvetydig eller upresis.
- I implementasjonsoppgaver foretrekkes lavere kjøretidskompleksitet.
- Du kan anta at du har algoritmer og datastrukturer kjent fra pensum tilgjengelig, med mindre noe annet er spesifisert.

Oppvarming 2 poeng

(a) Hva er en algoritme? Svar kort (maks fire setninger).

(b) Hva er en datastruktur? Svar kort (maks fire setninger).

Vi er ikke ute etter et «fasitsvar». Vi er ute etter å høre din forståelse av begrepene, og alle rimelige svar gir full uttelling.

Innsetting i heap — 10 poeng

(a) Du er gitt et tomt array på størrelse 7.

0	1	2	3	4	5	6

Sett inn de følgende tallene med min-heap innsetting:

$$3, 5, 10, 8, 4, 2, 7$$

Fyll ut tabellen slik den ser ut etter alle tallene er satt inn. Skriv svaret som en kommaseparert liste.

(b) Forklar kort hvordan algoritmen for innsetting i en min-heap fungerer, og skisser algoritmen med pseudokode. Du kan anta at arrayet i input ikke er fullt.

Input: Et array A som representerer en heap med n elementer, og et element x
Output: Et array som representerer en heap, som inneholder x

```
1 Procedure HeapInsert(A, x)
  | // ...
```

(c) Redgjør for kjøretidskompleksiteten til algoritmen.

Finn duplikat 12 poeng

Du er gitt et array `A` med sammenlignbare elementer. Du får vite at `A` inneholder nøyaktig ett duplikat. Altså er alle elementer i `A` unike, bortsett fra *ett* element, som forekommer nøyaktig to ganger.

(a) Anta at du er gitt et *sortert* array `A`, og får vite at x er duplikatet i `A`. Skriv en effektiv prosedyre som skriver ut de to posisjonene som inneholder x. Oppgi kjøretidskompleksiteten på algoritmen.

Input: Et *sortert* array `A` med n sammenlignbare elementer, og et element x som forekommer nøyaktig to ganger
Output: Skriver ut de to posisjonene $0 \leq i < j < n$ hvor `A[i]` $= $ `A[j]` $= x$ forekommer
Procedure `FindSortedIndicesOfDuplicate(A,` x`)`

```
// ···
```

(b) Anta at du er gitt et *sortert* array `A` (og nå får du ikke oppgitt elementet som er duplisert). Skriv en effektiv prosedyre som skriver ut de to posisjonene til elementet som forekommer to ganger. Oppgi kjøretidskompleksiteten på algoritmen.

Input: Et *sortert* array `A` med n sammenlignbare elementer
Output: Skriver ut de to posisjonene $0 \leq i < j < n$ hvor `A[i]` $= $ `A[j]`
Procedure `FindSortedDuplicateIndices(A)`

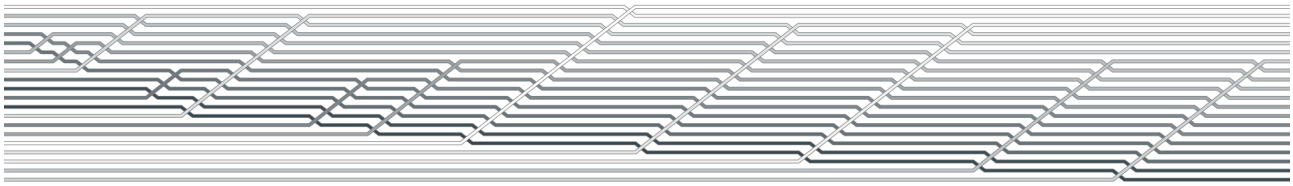
```
// ···
```

(c) Anta at du er gitt et array `A` (nå kan du ikke anta at arrayet er sortert). Skriv en effektiv prosedyre som skriver ut de to posisjonene til elementet som er duplisert. Oppgi kjøretidskompleksiteten på algoritmen.

Input: Et array `A` med sammenlignbare n elementer
Output: Skriver ut de to posisjonene $0 \leq i < j < n$ hvor `A[i]` $= $ `A[j]`
Procedure `FindDuplicateIndices(A)`

```
// ···
```

Gnome sort 10 poeng



Gnome sort er en enkel sorteringsalgoritme som *ikke* er kjent fra pensum. Illustrasjonen ovenfor er generert fra en kjøring av Gnome sort. Pseudokode for algoritmen er gitt nedenfor.

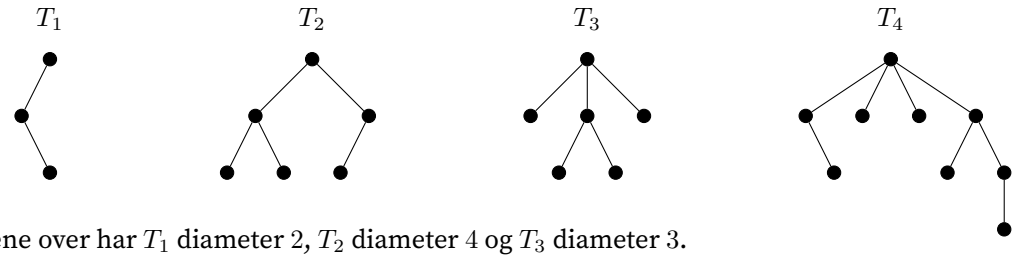
```
Input: Et array A med n elementer
Output: Et sortert array med de samme n elementene
1 Procedure GnomeSort(A)
2     i ← 0
3     while i < n do
4         if i > 0 and A[i-1] > A[i] then
5             A[i-1], A[i] ← A[i], A[i-1] // swap A[i] and A[i - 1]
6             i ← i − 1
7         else
8             i ← i + 1
```

I denne oppgaven skal du drøfte fordeler og ulemper ved Gnome sort sammenlignet med andre sorteringsalgoritmer kjent fra pensum. Teksten din bør inneholde:

- en kort forklaring på hvordan Gnome sort fungerer (med naturlig språk),
- kjøretidskompleksiteten til Gnome sort (sammen med en kort begrunnelse),
- en sammenligning av Gnome sort med sorteringsalgoritmer fra pensum, og
- en kort redegjørelse for hvorvidt Gnome sort er stabil og/eller in-place.

Diameteren til et tre 12 poeng

Vi definerer *diameteren* til et tre som lengden til den lengste stien mellom to noder.



I eksemplene over har T_1 diameter 2, T_2 diameter 4 og T_3 diameter 3.

(a) Hva er diameteren til T_4?

I de neste deloppgavene begrenser vi oss til *binære* trær. Hvis v er en node, så er:

- v.`left` venstre barn av v
- v.`right` høyre barn av v

(b) Vil den lengste stien i et binært tre alltid gå gjennom rotnoden? Begrunn svaret.

(c) Skriv en prosedyre som finner diameteren til et gitt *binærtre*.

```
Input: Rotnoden v av et binærtre
Output: Returnerer diameteren til treet
1 Procedure Diameter(v)
  |  // ···
```

Lavere kjøretidskompleksitet er mer poenggivende.

(d) Oppgi kjøretidskompleksiteten på algoritmen din for et binærtre med n noder.

Blindern-problemet 10 poeng

Blindern-problemet er en utfordring som er kjent for mange studenter ved Universitet i Oslo, som oppstår når man må gå fra en forelesning til en annen på motsatt side av campus på et knapt kvarter. Du er blitt bedt om å konsultere i UiO sitt nye prosjekt hvor det skal utvikles et tunnelsystem som forbinder alle bygningene som hører til campus.

Det er allerede kartlagt hvor mye det vil koste å grave tunnel mellom hvert par av bygninger (og anta at ingen av de kartlagte tunnelene kolliderer). Din oppgave er å finne den mest kostnadseffektive måten å grave et tunnelsystem på, ut ifra de kartlagte tunnelene, slik at man kan gå mellom alle bygningene kun ved å bruke tunnelsystemet.

(a) Forklar kort hvordan dette problemet kan uttrykkes som et grafproblem. Svaret ditt bør inkludere:

- Hva slags graf (rettet/urettet og vektet/uvektet) egner seg her?
- Hva representerer nodene?
- Hva representerer kantene?

(b) Oppgi en egnet algoritme, kjent fra pensum, som kan brukes til å finne den mest kostnadseffektive måten å grave ut et tunnelsystem på, slik at man kan komme seg mellom alle bygningene kun ved å bruke tunnelsystemet. For algoritmen må du oppgi:

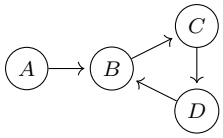
- de mest sentrale datastrukturene den bruker,
- en *kort* forklaring på hvordan algoritmen fungerer, og
- kjøretidskompleksiteten på algoritmen.

(c) Prosjektet blir ferdig i rekordfart og er klar til å brukes. Beskriv, med naturlig språk, en effektiv algoritme for å finne korteste sti fra en bygning til en annet kun ved å bruke tunnelsystemet. Du kan bruke algoritmer kjent fra pensum *uten* å gjøre rede for dem. Oppgi kjøretidskompleksiteten på algoritmen.

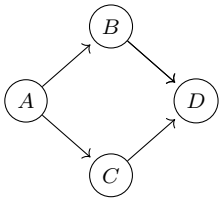
Dependency hell 12 poeng

I moderne programvareutvikling er det vanlig at en applikasjon er avhengig av mange *biblioteker*, som igjen kan avhenge av andre biblioteker. Et bibliotek kan ses på som en samling med kode som er designet for gjenbruk. Bibliotekene som brukes for å bygge en applikasjon kalles *avhengighetene* til applikasjonen. Avhengighetene mellom biblioteker er gitt som en rettet graf $G = (V, E)$, der en node representerer et bibliotek, og hvor det går en kant fra u til v dersom u bruker v som et bibliotek.

Eksemplene under er situasjonener ofte omtalt som *dependency hell*.



Sirkulær avhengighet



Diamantavhengighet

Sykliske avhengigheter er som regel uønsket fordi en oppdatering av et av bibliotekene i en sykel vil kunne påvirke alle de andre bibliotekene i sykelen. *Diamantavhengighet* er også som regel uønsket fordi en oppdatering kan føre til at man blir avhengig av forskjellige versjoner av det samme biblioteket samtidig.

Du skal utvikle algoritmer som kan hjelpe med å identifisere slike uønskede situasjoner.

(a) Mer presist sier vi at grafen inneholder en *syklisk avhengighet* dersom det finnes en sti fra en node v til den samme noden v som består av minst to noder.

Skriv en prosedyre som, gitt en rettet graf $G = (V, E)$, returner **`true`** dersom grafen inneholder *sykliske avhengigheter*, og **`false`** ellers. Dersom prosedyren din benytter seg av algoritmer kjent fra pensum må du redgjøre kort for hvordan de fungerer.

(b) I denne deloppgaven kan du anta at grafen ikke inneholder sykliske avhengigheter.

Mer presist sier vi at grafen inneholder en *diamantavhengighet* dersom det finnes to distinkte stier fra u til v. Husk at to stier er distinkte dersom de ikke deler noen noder eller kanter utenom u og v.

Skriv en prosedyre som, gitt en rettet asyklisk graf $G = (V, E)$, returnerer **`true`** dersom grafen inneholder *diamantavhengigheter*, og **`false`** ellers.