Eksamen i IN2010 høsten 2025

5. Desember 2025

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 større oppgaver hvor du blir bedt om å skrive og resonnere.
- Ingen hjelpemidler er tillatt.
- Hele oppgavesettet er vedlagt som PDF.
- 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 ber om.
- Redegjør 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 er lavere kjøretidskompleksitet mer poenggivende.
- Du kan anta at du har algoritmer og datastrukturer kjent fra pensum tilgjengelig, med mindre noe annet er spesifisert.

Sensorveiledning

Den første delen består små oppgaver av typen «sant/usant», verdt totalt 26 poeng. Poenggivningen er slik at det ikke er noen forskjell mellom ubesvart og feil svar. Hver oppgave teller ett poeng, og for å få den totale poengsummen for den første delen skalerer vi poengene med formelen $2 \cdot \max(n - 13, 0)$, hvor n er antall spørsmål som er korrekt besvart. Dette gjør det enklere å korrigere for gjetning. Hvis man gjetter helt tilfeldig på alle oppgavene får man i snitt null poeng, men ved alt rett får man 26 poeng.

Sensorveiledningen inneholder et løsningsforslag på alle oppgaver som ikke rettes automatisk. I tillegg gis det veiledning på hvordan poeng kan fordeles, men sensoren står fritt til å gi eller trekke poeng etter eget skjønn. Det legges alltid vekt på at besvarelsene er presise og enkle å forstå, både med hensyn til form og innhold.

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.

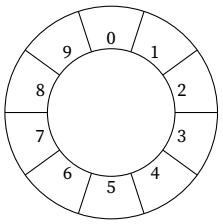
Her finnes det utrolig mange riktige svar! Vi er ikke ute etter en formell definisjon (selv om det selvfølgelig vil være poenggivende), men vil bare få studentene i gang med å tenke på algoritmer og datastrukturer. Her er mitt svar (som på ingen måte bør være førende for hvordan vi tildeler poeng):

- Algoritmer er idéene bak de programmene vi skriver. De angir hva som må gjøres for å løse et gitt problem på en presis og entydig måte. Følger du en algoritme for å løse et problem så skal du også få riktig svar til slutt, og helst på rimelig tid.
- Datastrukturer er måter å organisere data på. Stort sett vil vi organisere dataene etter hva vi oftest ønsker å gripe tak i, slik at vi kan få mer effektive algoritmer.

Vi gir full uttelling for alle rimelige svar. For å *ikke* få full uttelling må man enten la oppgaven stå ubesvart, eller skrive noe som er direkte feil. Det kan trekkes poeng dersom deler av svaret inneholder vesentlige feil.

Separate chaining 10 poeng

(a) Vi starter med et tomt array på størrelse $N = 10$.



Hashfunksjonen du skal bruke er $h(k, N) = k \mod N$, som for dette eksempelet blir $h(k, 10) = k \mod 10$. Altså hasher et tall til sitt siste siffer.

Bruk *separate chaining* til å sette inn disse nøkkel- og verdiparene i den gitte rekkefølgen:

$$(21, a), (54, b), (82, c), (10, d), (20, e), (44, f), (10, g)$$

Skriv ned resultatet etter siste innsetting, med hver linje på formen:

$$i\colon (k_1, v_1), (k_2, v_2), \ldots$$

der i er en indeks i arrayet, og $(k_1, v_1), (k_2, v_2), \ldots$ er nøkkel- og verdipar.

(b) Forklar kort hvordan algoritmen for innsetting ved separate chaining fungerer, og skisser algoritmen med pseudokode. Du kan anta at du har en hashfunksjon h slik at $h(k, N)$ gir et tall mellom 0 og $N - 1$ for en vilkårlig nøkkel k. Du trenger ikke ta høyde for rehashing.

Input: Et array A av størrelse N, en nøkkel k og verdi v
Output: Et array der (k, v) er satt inn med separate chaining

```
1 Procedure SeparateChainingInsert(A, k, v)
  | // ...
```

(c) Redegjør kort for kjøretidskompleksiteten til innsetting ved separate chaining.

(a) Resultatet av innsettingen ser slik ut:

$$
\begin{aligned}
&0\colon (10, g), (20, e)\\
&1\colon (21, a)\\
&2\colon (82, c)\\
&3\colon\\
&4\colon (54, b), (44, f)\\
&5\colon\\
&6\colon\\
&7\colon\\
&8\colon\\
&9\colon
\end{aligned}
$$

- 3 poeng for riktig svar.
- Det trekkes 1.5 poeng for å ikke overskrive en lik nøkkel med ny verdi.

(b) Det viktigste er at studenten formidler en god forståelse av hvordan separate chaining fungerer.

Innsetting med separate chaining tar utgangspunkt i å bruke hashen av nøkkelen som indeks, og plassere nøkkel-/verdi-paret i en bøtte (som regel implementert som en lenket liste). Dersom nøkkel-/verdi-paret allerede finnes i listen, skal verdien erstattes.

I implementasjonen under antar vi lister som er implementert som dynamiske arrayer.

Input: Et array A av størrelse N, en nøkkel k og verdi v
Output: Et array der (k, v) er satt inn med separate chaining

```
1  Procedure SeparateChainingInsert(A, k, v)
2      i ← h(k, N)
3      B ← A[i]
4      if B = null then
5          A[i] ← a list containing the element (k, v)
6          return A
7      for j ← 0 to |B| − 1 do
8          if key of B[j] is k then
9              B[j] ← (k, v)
10             return A
11     append (k, v) to B
12     return A
```

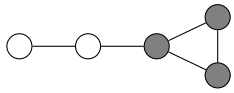
- 5 poeng for en god forklaring med pseudokode.
- Det gis full uttelling for selvforklarende eller godt kommentert pseudokode.
- Det gis inntil 3 poeng dersom pseudokoden mangler.
- Det trekkes 1 poeng for å ikke overskrive en lik nøkkel med ny verdi.
- Det trekkes 1 poeng dersom det ikke kommer klart frem hvordan bøttene initialiseres.

(c) Kjøretiden til separate chaining er $O(n)$ i verste tilfelle, men under antagelsen av en god hashfunksjon og en tilstrekkelig lav load factor vil lengden på hver liste være begrenset av en konstant, som gir forventet $O(1)$.

- 2 poeng.
- 1 poeng for kun verste tilfelle.
- 1 poeng for kun forventet kjøretidskompleksitet.

Sykler i grafer — 11 poeng

(a) Anta at grafen $G = (V, E)$ er en *urettet* graf. Grafen inneholder en sykel hvis det finnes en sti fra en node u tilbake til samme node u som går innom minst én annen node v.



Skriv en prosedyre som avgjør om grafen inneholder en sykel.

```
Input: En urettet graf G = (V, E)
Output: Returnerer true hvis grafen inneholder en sykel, og false ellers
Procedure isCyclicUndirected(G)
    // ...
```

Det finnes flere gode løsninger i $\mathcal{O}(|V|+|E|)$. Den mest nærliggende er kanskje et dybde-først søk.

```
Input: En urettet graf G = (V, E)
Output: Returnerer true hvis grafen inneholder en sykel, og false ellers
Procedure isCyclicUndirected(G)
    visited ← empty set
    for u ∈ V do
        if u ∈ visited then
            continue
        if isCyclicComponent(G, null, u, visited) then
            return true
    return false

Procedure isCyclicComponent(G, p, u, visited)
    add u to visited
    for (u, v) ∈ E do
        if v = p then
            continue
        if v ∈ visited then
            return true
        if isCyclicComponent(G, u, v, visited) then
            return true
    return false
```

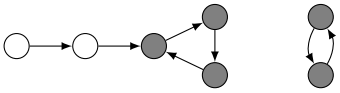
I løsningen over fastslår vi at det finnes en sykel dersom dybde-først søket har en kant til en node som allerede er besøkt, *med mindre* dette er den samme noden som DFS ble kalt fra (altså foreldre-noden i spenntreet). Uten denne begrensningen ville `isCyclicUndirected` svart **`true`** på alle grafer som inneholder en kant.

En annen løsning er å finne komponentene til grafen, og sjekke at hver av komponentene inneholder nøyaktig $|E_c| = |V_c| - 1$, der V_c og E_c henholdsvis er nodene og kantene i kompnenten c.

En noe mer innsiktsfull løsning er å se at grafen ikke inneholder sykler hvis og bare hvis $|E| = |V| - |C|$, der C er mengden av komponenter. Altså inneholder grafen ingen sykler hvis antall kanter svarer til antall noder minus antall komponenter.

- Inntil 5 poeng for en korrekt løsning i $\mathcal{O}(|V|+|E|)$ som er godt formidlet med pseudokode.
- Trekk 2 poeng hvis man ikke tar høyde for at noden må være ulik foreldrenoden i spenntreet.
- Inntil 3 poeng dersom løsningen er i $\mathcal{O}(|V| \cdot (|V|+|E|))$.

(b) Anta at grafen $G = (V, E)$ er en *rettet* graf. Grafen inneholder en sykel hvis det finnes en sti fra en node u tilbake til samme node u.



Skriv en prosedyre som avgjør om grafen inneholder en sykel.

```
Input: En rettet graf G = (V, E)
Output: Returnerer true hvis grafen inneholder en sykel, og false ellers
1 Procedure isCyclicDirected(G)
  | // ...
```

Dette bør løses på lineær tid. Mest naturlig er å bruke topologisk sortering (som oppdager sykler). Algoritmen for topologisk sortering plasserer først alle noder med inngrad 0 på en stack. Noder plukkes av stacken så lenge den ikke er tom, og for hver node som plukkes av fjernes alle dens utgående kanter (som fører til at dens naboer får lavere inngrad). Hvis en node får inngrad 0 plasseres den på stacken. Grafen er asyklisk hvis og bare hvis alle nodene blir prosessert.

```
1  Procedure isCyclicDirected(G)
2  |  stack ← empty stack
3  |  processed ← 0
4  |  for v ∈ V do
5  |  |  if indegree of v is 0 then
6  |  |  |  stack.push(v)
7  |  while stack is not empty do
8  |  |  u ← stack.pop()
9  |  |  processed ← processed + 1
10 |  |  for (u, v) ∈ E do
11 |  |  |  remove u from the incoming edges of v
12 |  |  |  if indegree of v is 0 then
13 |  |  |  |  add v to stack
14 |  return processed < |V|
```

En annen mulighet er å finne de sterkt sammenhengende komponentene av grafen, og sjekke om det finnes komponenter av størrelse større enn én.

Det er også fullt mulig å skrive et modifisert dybde-først søk som oppdager sykler. I en slik løsning vil man typisk skille mellom en node er *uoppdaget*, *under prosessering* eller *ferdig prosessert*. Dersom man under et søk finner en kant til en node som er *under prosessering* kan man konkludere med at grafen inneholder en sykel. En annen lignende (og nærliggende) løsning er å sjekke om en node er på *stacken* (fordi da er den under prosessering); med en vanlig stack vil dette gi $\mathcal{O}(|V| \cdot (|V| + |E|))$.

- Inntil 6 poeng for en korrekt løsning i $\mathcal{O}(|V| + |E|)$ som er godt formidlet med pseudokode.
- Inntil 3 poeng dersom løsningen er i $\mathcal{O}(|V| \cdot (|V| + |E|))$.

Anti-AVL 12 poeng

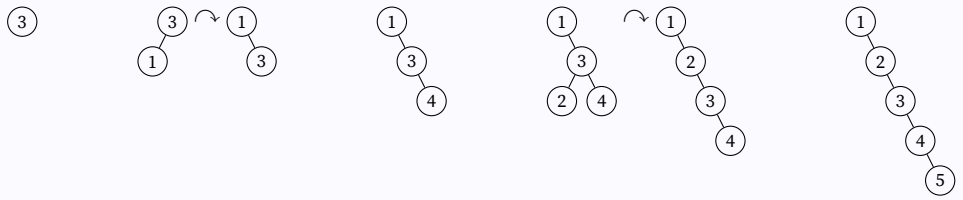
AVL-trær bruker venstre- og høyrerotasjoner for å sørge for at det binære søketreet ender opp balansert etter innsetting og sletting. Her er innsetting for en variant av AVL-trær som vi kaller *Anti-AVL*.

```
Input: Rotnoden v av et Anti-AVL-tre B og et
       element x
Output: Et Anti-AVL-tre med x satt inn
1  Procedure InsertAntiAVL(v, x)
2      if v = null then
3          v ← new Node(x)
4      else if x < v.element then
5          v.left ← InsertAntiAVL(v.left, x)
6      else if x > v.element then
7          v.right ←
            InsertAntiAVL(v.right, x)
8
9      if v.left ≠ null then
10         return RightRotate(v)
11
12     return v
```

```
Input: En node z
Output: Roten til det
        høyreroterte treet
1  Procedure RightRotate(z)
2      y ← z.left
3      T2 ← y.right
4
5      y.right ← z
6      z.left ← T2
7
8      return y
```

(a) Sett inn tallene $3, 1, 4, 2, 5$ med `InsertAntiAVL`.

- Beskriv kort hvordan det resulterende Anti-AVL-treet ser ut.
- Angi det totale antallet rotasjoner som forekommer i løpet av innsettingene.



Treet har 1 som rot, 2 som høyre-barn av roten, 3 som høyre barn av 2, og så videre. Altså er det en sortert lenket liste hvis du anser `right` som en neste-peker.

Det gjøres 2 rotasjoner total.

- Inntil 2 poeng.
- 1 poeng for beskrivelsen og 1 poeng for antall rotasjoner.

(b) Beskriv mer generelt hvordan Anti-AVL-trær ser ut.

Et Anti-AVL-tre er et binært søketre der hver node kun har et høyre-barn. Det vil se ut som en sortert lenket liste av høyre-pekere.

- Inntil 2 poeng.

(c) Et binært søketre er et binærtre der elementet i hver node er:

- større enn alle elementer i venstre subtre, og
- mindre enn alle elementer i høyre subtre.

Har Anti-AVL-trær den samme egenskapen? Forklar.

Ja, Anti-AVL er binære søketrær. En høyre-rotasjon bevarer BST-egenskapen. Siden innsetting i Anti-AVL-trær er identisk med ordinære binære søketrær, bortsett fra at det gjør en høyre-rotasjon i tilfellet der noden har et venstre barn, så vil innsetting fremdeles respektere BST-egenskapen.

- Inntil 3 poeng for en god forklaring.

(d) Hva er kjøretidskompleksiteten på oppslag i Anti-AVL-trær? Du kan anta at oppslag er implementert identisk som ordinære binære søketrær og AVL-trær. Begrunn svaret kort.

Oppslag i et Anti-AVL-tre er i $\mathcal{O}(n)$, der n er antall noder i treet. Hvis vi skal slå opp det største elementet i treet, vil vi følge $n - 1$ høyre-barn, som er $\mathcal{O}(n)$.

- Inntil 2 poeng.
- 1 poeng for riktig svar og 1 poeng for begrunnelsen.

(e) Beskriv kort hvordan innsetting i Anti-AVL-trær skiller seg fra innsetting i AVL-trær.

Den viktigste, og klareste forskjellen, er at AVL-trær bygger et svært balansert binært søketre, men Anti-AVL-trær bygger et svært *ubalansert* binært søketre.

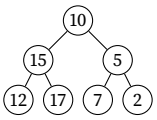
AVL-trær er avhengig av at hver node lagrer høyden til sitt subtre, foretar rotasjoner dersom høydeforskjellen mellom to subtrær overstiger 1. Anti-AVL-trær sjekker kun om det venstre subtre er tomt.

- Inntil 3 poeng.
- 2 poeng for å peke ut den veldig tydelige forskjellen, og 1 poeng for å peke ut at AVL-trær gjør rotasjoner basert på høyden.

Level-order 12 poeng

En *level-order*-traversering besøker nodene i et tre etter *dybde*. Det vil si at roten besøkes først, så alle barna til roten, så alle barna til rotens barn, og så videre.

En *level-order*-traversering av følgende tre kan, for eksempel, besøke nodene i rekkefølgen $10, 15, 5, 12, 17, 7, 2$.



Merk at output for algoritmene under er en *liste*, men det er en liten detalj som ikke er vesentlig for *algoritmene*. Vi trekker ikke poeng for dette alene, men det kan være med på å trekke i samspill med andre unøyaktigheter.

(a) Anta at treet er en *binær heap*, representert ved et array A med n elementer. Gi en algoritme som returnerer en liste med elementene fra A i *level-order*.

```
Input: En binær heap, representert ved et array A med n elementer
Output: En liste med elementene i heapen i level-order
1 Procedure LevelOrderHeap(A)
  | // ···
```

En heap er allerede i *level-order*, så det er tilstrekkelig å returnere heapen som en liste.

```
1 Procedure LevelOrderHeap(A)
2 |  list ← an empty list
3 |  for i ← 0 to n − 1 do
4 |  |  append A[i] to list
5 |  return list
```

- Inntil 2 poeng.

(b) Anta at treet er et *binært søketre* med n noder, der hver node v har følgende felter:

- v.`element` — verdien som er lagret i noden
- v.`left` — venstre barn av v
- v.`right` — høyre barn av v

Gi en algoritme som returnerer en liste med elementene fra treet i *level-order*.

```
Input: Roten v av et binært søketre med n noder
Output: En liste med elementene i søketreet i level-order
1 Procedure LevelOrderBST(v)
  | // ···
```

Dette gjøres enklest ved hjelp av et bredde-først søk fra rotnoden. Siden input er et tre er det ikke nødvendig å bruke et `visited`-set.

```
1  Procedure LevelOrderBST(v)
2      list ← an empty list
3      queue ← an empty queue
4      if v ≠ null then
5          queue.enqueue(v)
6      while queue is not empty do
7          u ← queue.dequeue()
8          append u.element to list
9          if u.left ≠ null then
10             queue.enqueue(u.left)
11         if u.right ≠ null then
12             queue.enqueue(u.right)
13     return list
```

- Inntil 6 poeng.
- Inntil 2 poeng trekk for manglende **`null`**-sjekker (altså, ikke ta høyde for at en node mangler venstre/høyre barn).
- Trekk 3 poeng for bruk av stack i stedet for kø (som gir dybde først, i stedet for bredde først).
- Vi trekker ikke for overflødig bruk av `visited`-set (men det trekker litt ned på helhetsinntrykket).

(c) Oppgi kjøretidskompleksiteten for algoritmene dine i (a) og (b).

Begge er $\mathcal{O}(n)$, siden de flytter elementene over i en liste.

- Inntil 1 poeng for riktig kjørekompleksitet på begge algoritmer.

(d) Se på følgende algoritme:

```
Input: Roten v av et binært søketre
Output: Et nytt binært søketre
1  Procedure ConstructBST(v)
2      B ← empty binary search tree
3      nodes ← LevelOrderBST(v)
4      for u in nodes do
5          B ← Insert(B, u)
6      return B
```

Hva kan du si om relasjonen mellom input-treet og output-treet?

Input-treet og output-treet er *like*. Roten vil plasseres først, så vil dens to barn plasseres, og deretter deres barn og så videre.

- Inntil 3 poeng for riktig svar.
- Oppgaven krever ikke en begrunnelse.

Finn k minste — 12 poeng

Gitt et array A med n sammenlignbare elementer, ønsker vi å finne de k minste elementene i A der $1 \leq k \leq n$.

```
Input: Et array A av n sammenlignbare elementer, og et heltall 1 ≤ k ≤ n
Output: En liste med de k minste elementene fra A
1 Procedure FindMin(A, k)
  | // ···
```

Skriv en prosedyre som beregner `FindMin(`A`, `k`)` i:

> Merk at et korrekt svar på deloppgave (c) også er et korrekt svar på (a) og (b). Dermed kan man få full uttelling dersom man oppgir at svaret på (c) også er et svar på (a) og (b).
>
> Merk igjen at output for algoritmene under er en *liste*, men det er en liten detalj som ikke er vesentlig for *algoritmene*. Vi trekker ikke poeng for dette alene, men det kan være med på å trekke i samspill med andre unøyaktigheter.

(a) $\mathcal{O}(k \cdot n)$

> En naturlig algoritme i $\mathcal{O}(k \cdot n)$ er å finne det minste elementet, deretter finne det nest minste, og så videre, k ganger. En annen måte å se på det er å gjøre k iterasjoner av `SelectionSort`. Løsningen under er utledet fra `SelectionSort`.

```
1  Procedure FindMin(A, k)
2  |  list ← an empty list
3  |  for i ← 0 to k do
4  |  |  m ← i
5  |  |  for j ← i + 1 to n − 1 do
6  |  |  |  if A[j] < A[m] then
7  |  |  |  |  m ← j
8  |  |  append A[m] to list
9  |  |  A[i], A[m] ← A[m], A[i]
10 |  return list
```

- Inntil 3 poeng for en løsning i $\mathcal{O}(k \cdot n)$.
- Trekk 1 poeng dersom man ikke ser vekk fra elementer som allerede er lagt til i resultatet.
- Trekk 1 poeng dersom man ikke håndterer at to elementer kan være like.

(b) $\mathcal{O}(n \cdot \log(n))$

> Denne løsningen svarer til å sortere arrayet med en $\mathcal{O}(n \cdot \log(n))$ sorteringsalgoritme, og plukke ut de k første.

```
1 Procedure FindMin(A, k)
2 |  list ← an empty list
3 |  A ← MergeSort(A)
4 |  for i ← 0 to k do
5 |  |  append A[i] to list
6 |  return list
```

- Inntil 4 poeng for en korrekt løsning.
- Inntil 3 poeng dersom `Quicksort` benyttes (denne er i $\mathcal{O}(n^2)$, men er vanligvis rask).
- Inntil 1 poeng for løsninger som baserer seg på en kvadratisk sorteringsalgoritme.

(c) $\mathcal{O}(n \cdot \log(k))$ *eller* $\mathcal{O}(n + k \cdot \log(n))$

Problemet kan løses i $\mathcal{O}(n \cdot \log(k))$ ved å bruke en max-heap til å holde på de k minste elementene.

```
1  Procedure FindMin(A, k)
2      list ← an empty list
3      heap ← an empty max-heap
4      for i ← 0 to k − 1 do
5          add A[i] to heap
6      for i ← k to n − 1 do
7          heap.push(min(heap.pop(), A[i]))
8      for i ← 0 to k − 1 do
9          prepend heap.pop () to list
10     return list
```

Merk at man også kan anta at heapen har metode `peek`, og kun poppe dersom $A[i]$ er mindre enn det største elementet på heapen.

Problemet kan løses i $\mathcal{O}(n + k \cdot \log(n))$ ved å først bygge en min-heap i lineær tid (vi kjenner at vi kan bygge en max-heap på linær tid fra forelesning, og en min-heap er helt tilsvarende), deretter poppe de k minste elementene fra en heap med n elementer.

```
1  Procedure FindMin(A, k)
2      list ← an empty list
3      heap ← BuildMinHeap(A)
4      for i ← 0 to k − 1 do
5          append heap.pop () to list
6      return list
```

- Inntil 5 poeng for en korrekt løsning i $\mathcal{O}(n \cdot \log(k))$ *eller* $\mathcal{O}(n + k \cdot \log(n))$.

Kollektivtilbudet i Oslo — 15 poeng

I det siste bystyremøtet ble det sluttet flere vedtak vedrørende kollektivtilbudet i Oslo. La kollektivtilbudet være representert som en sammenhengende, urettet og vektet graf $G = (V, E)$, der $u \in V$ er en holdeplass, en kant $\{u, v\} \in E$ er en *strekning* og vektfunksjonen w angir reisetid (i minutter) for en strekning.

(a) Grafen er *sammenhengende* og *urettet*. Forklar kort hva hver av egenskapene vil si for en som reiser kollektivt i Oslo.

At grafen er sammenhengende vil si at en reisende kan reise mellom to vilkårlige holdeplasser ved å bruke kollektivtilbudet.

At grafen er urettet vil si at en som har reist en strekning alltid kan reise den samme strekningen tilbake.

- Inntil 1 poeng.

(b) Bystyret har vedtatt at det skal bygges nye holdeplasser langs alle *strekninger* som overstiger k minutter. Under bygningsarbeidet vil det ikke være mulig å reise langs noen av disse strekningene. Skriv en algoritme som tar grafen som representerer det nåværende kollektivtilbudet, samt en grense k, og sjekker om det vil være mulig å reise mellom alle par av holdeplasser under bygningsarbeidet.

Her kan vi gjøre et dybde-først søk, der kanter med vekt som overstiger k ignoreres, og sjekke om alle noder nås gjennom én traversering.

```
1  Procedure CheckConnected(G, k)
2      stack ← stack initialized with an arbitrary vertex u ∈ V
3      visited ← empty set
4
5      while stack is not empty do
6          u ← stack.pop()
7          if u not in visited then
8              add u to visited
9              for (u, v) in E do
10                 if w(u, v) < k then
11                     add v to stack
12
13     return visited = V
```

- Inntil 4 poeng for en løsning i $\mathcal{O}(|E|)$.
- Inntil 1 poeng dersom kanter med vekt som overstiger k benyttes.

(c) Det viser seg at algoritmen fra deloppgave (b) svarer **`false`** for k-en bystyret hadde i tankene. Gi en algoritme som bystyret kan bruke til å finne den minste verdien for k som algoritmen fra deloppgave (b) svarer **`true`** for.

En naturlig løsning er å beregne det minimale spenntreet for grafen, og finne verdien til den kanten med høyest vekt.

```
1  Procedure FindMinimalK(G)
2      T ← Prim(G)
3      k ← ∞
4      for (u, v) in edges of T do
5          k ← max(k, w(u, v))
6      return k
```

- Inntil 4 poeng for en løsning i $\mathcal{O}(|E| \cdot \log(|V|))$.

(d) Det siste året har det vært to hendelser der det har vært full stopp ved en holdeplass. Det ledet til at reisende ikke kunne nå destinasjonen sin. Bystyret har vedtatt en garanti: dersom det blir full stopp ved en holdeplass, skal det fremdeles være mulig for reisende å nå sin destinasjon via en alternativ rute.

Garantien gjelder for alle holdeplasser.

Forklar hvordan vi kan bruke en algoritme kjent fra pensum til å sjekke om kollektivtilbudet leverer denne garantien.

> Her ønsker vi å sjekke om grafen er 2-sammenhengende; altså skal grafen forbli sammenhengende hvis vi fjerner en vilkårlig node (sammen med dens tilhørende kanter). Fra pensum kjenner vi en algoritme som finner alle separasjonsnoder i lineær tid, og algoritmen `IsBiconnected` sjekker at det ikke finnes noen separasjonsnoder i grafen. Dette er det samme som å sjekke at garantien holdes for alle holdeplasser.
>
> - Inntil 3 poeng for en løsning i $\mathcal{O}(|V| + |E|)$.

(e) Møtet til bystyret varte for lenge, så de bestilte pizza, og etter det fløt det over med nye kreative idéer. Den siste i denne rekken er følgende: for å avlaste de mest trafikkerte rutene, vil de gi *negative billettpriser* på de minst trafikkerte strekningene.

Før de setter i gang med prosjektet må de forsikre seg om at det ikke er mulig å opptjene seg vilkårlig høye summer i negative billettpriser ved å reise rundt og rundt.

Billettprisen for en strekning er gitt av en vektfunksjon c. En strekning kan være svært trafikkert i én retning, men lite trafikkert i den andre, derfor kan $c(u, v)$ være ulik $c(v, u)$ for en strekning $\{u, v\} \in E$.

Forklar hvordan vi kan bruke en algoritme kjent fra pensum til å sjekke om det finnes måter å opptjene vilkårlig høye summer i negative billettpriser.

> Her ønsker vi å finne negative sykler i en graf. Algoritmen `BellmanFord` finner korteste stier i en rettet graf *og* identifiserer negative sykler. Dermed kan vi kalle på `BellmanFord` med grafen G fra en vilkårlig node $u \in V$ og vektfunksjonen c og returnere **`false`** dersom den ikke rapporterer at det finnes negative sykler.
>
> Merk at `BellmanFord` finner avstanden fra én node til alle andre noder, derfor er input både grafen og en startnode. Siden grafen er urettet spiller det ingen rolle hvilken node vi starter fra.
>
> - Inntil 3 poeng.
> - Trekk 1 poeng dersom det ikke er spesifisert hvilken node algoritmen skal starte fra.