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.

Sensorveiledning

Den første delen består små oppgaver av typen «sant/usant», verdt totalt 32 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 $2 \cdot \max(n - 16, 0)$, 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 32 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.

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.

Tallene settes inn slik:

3						
3	5					
3	5	10				
3	5	10	8			
3	5	10	8	4		
3	4	10	8	5		
3	4	10	8	5	2	
3	4	2	8	5	10	
2	4	3	8	5	10	7
0	1	2	3	4	5	6

Det er kun den siste linjen `2, 4, 3, 8, 5, 10, 7` som trenger inngå i svaret.

2 poeng for riktig svar. 1 poeng for *nesten* riktige svar, som vil si dersom

- svaret er en min-heap, men ikke den samme heapen som den gitte rekkefølgen vil gi opphav til, eller
- svaret er ett bytte unna løsningensforslaget.

(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)
  | // ...
```

Hovedidéen er å alltid legge til på neste ledige plass, det vil si indeks n. Det er der neste node *må* være for at treet fortsatt skal være komplett. Hvis noden på den nye plassen er mindre enn foreldrenoden så må de bytte plass. Dette gjøres iterativt (eller rekursivt).

I arrayrepresentasjonen av en heap ligger alltid roten på indeks 0. Foreldrenoden til en gitt indeks i ligger på `ParentOf(i)` $= \lfloor \frac{i-1}{2} \rfloor$ for en gitt indeks i.

Her er algoritmen presentert med pseudokode:

```
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)
2     A[n] ← x
3     i ← n
4     while 0 < i and A[i] < A[ParentOf(i)] do
5         A[i], A[ParentOf(i)] ← A[ParentOf(i)], A[i]
6         i ← ParentOf(i)
7     return A
```

Det gis opptil 6 poeng for et godt svar på oppgaven.

- Opptil 1 poeng for å gi hvor elementet skal plasseres.
- Opptil 2 poeng for å få frem at noden skal bytte plass med foreldernoden dersom den er mindre.
- Opptil 3 poeng for en godt skissert algoritme.

(c) Redgjør for kjøretidskompleksiteten til algoritmen.

Algoritmen for innsetting i heap er $\mathcal{O}(\log(n))$. Det er fordi n halveres i hver iterasjon.

Det gis 1 poeng for riktig kjøretidskompleksitet og 1 poeng for redegjørelsen.

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`)`

```
// ···
```

Dette kan løses i $\mathcal{O}(\log(n))$ ved å bruke et binær søk. Binærsøk presentert på forelesning returnerer **`true`**/**`false`**, men kan enkelt modifiseres til å returnere indeksen elementet befinner seg på. I konteksten av denne oppgaven kan vi anta at x er i arrayet, og vi kan la algoritmen være udefinert i tilfellet der x ikke er i arrayet.

Input: Et ordnet array `A` og et element x i `A`
Output: Returner i slik at `A[i]` $= x$
Procedure `BinarySearch(A,` x`)`

```
low ← 0
high ← |A| − 1
while low ≤ high do
    i ← ⌊(low+high)/2⌋
    if A[i] = x then
        return i
    else if A[i] < x then
        low ← i + 1
    else if A[i] > x then
        high ← i − 1
```

Studenten trenger ikke implementere det modifiserte binærsøket, men bør nevne at binærsøket må modifiseres litt.

Med dette kan algoritmen uttrykkes slik:

Procedure `FindSortedIndicesOfDuplicate(A,` x`)`

```
i ← BinarySearch(A, x)
print(i)
if 0 < i and A[i - 1] = x then
    print(i − 1)
if i < n − 1 and A[i + 1] = x then
    print(i + 1)
```

Oppgaven kan også løses ved å løpe gjennom arrayet frem til x, og printe posisjonen og posisjonen etter.

- Opptil 4 poeng for en løsning i logaritmisk tid.
- Opptil 2 poeng for en løsning i lineær tid.
- Det trekkes 1 poeng for feil (eller ikke oppgitt) kjøretidskompleksitet.
- Det trekkes opptil 1 poeng for å ikke nevne at binærsøket må modifiseres.
- Det gis ingen poeng for løsninger over lineær tid.

(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 ≤ i < j < n hvor A[i] = A[j]
Procedure FindSortedDuplicateIndices(A)
    // ···
```

Denne oppgaven løses best på lineær tid, og bør være ganske rett frem.

```
Input: Et sortert array A med n sammenlignbare elementer
Output: Skriver ut de to posisjonene 0 ≤ i < j < n hvor A[i] = A[j]
Procedure FindSortedDuplicateIndices(A)
    for i ← 0 to n − 2 do
        if A[i] = A[i + 1] then
            print(i)
            print(i + 1)
            return
```

- Opptil 3 poeng for en løsning i lineær tid.
- Det trekkes 1 poeng for feil (eller ikke oppgitt) kjøretidskompleksitet.
- Opptil 1 poeng for løsninger over lineær tid.

(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 ≤ i < j < n hvor A[i] = A[j]
Procedure FindDuplicateIndices(A)
    // ···
```

Dette kan også løses på (amortisert og forventet) lineær tid ved å bruke et hash map eller et hash set. Studenten kan anta at disse datastrukturene har konstant tid på de relevante operasjonene (altså kreves det ingen bemerkning om amortisert eller forventet kjøretidskompleksitet).

```
Input: Et sortert array A med n sammenlignbare elementer
Output: Skriver ut de to posisjonene 0 ≤ i < j < n hvor A[i] = A[j]
Procedure FindSortedDuplicateIndices(A)
    firstSeen ← empty hash map
    for i ← 0 to n − 1 do
        x ← A[i]
        if x is a key in firstSeen then
            print(firstSeen[x])
            print(i)
            return
        firstSeen[x] ← i
```

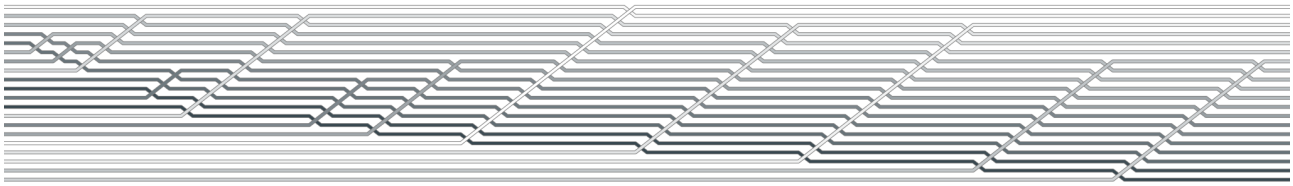
Den kan løses i lineæritmisk tid ved å sortere med en god sammenligningsbasert sorteringsalgoritme. Merge sort og Heapsort er gode valg. Quicksort er også et godt valg (men bør også nevnes at den strengt tatt er kvadratisk i verste tilfelle). Bucket- og Radix sort kan ikke brukes her, siden alt vi vet om elementene er at de er sammenlignbare.

Etter sortering kan verdien av duplikatet finnes på lineær tid ved å bruke løsningen fra forrige deloppgave. Deretter kan arrayet løpes over nok en gang, og skrive ut de posisjonene duplikatet befinner seg på.

Oppgaven kan også løses i kvadratisk tid ved å sammenligne hvert element med alle de andre.

- Opptil 5 poeng for en løsning på lineær tid.
- Opptil 4 poeng for en løsning i lineæritmisk tid.
- Opptil 1 poeng for høyere enn lineæritmisk tid.
- Det trekkes 1 poeng for feil (eller ikke oppgitt) kjøretidskompleksitet.

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.

Gnome sort fungerer ved å løpe over elementene, og når den finner et element som er større enn det foregående elementet skal et flyttes bakover i arrayet. Oppførselen til Gnome sort er nesten helt lik Insertion sort, men den gjør dette med å kun vedlikeholde én indeks. Den mest vesentlige forskjellen fra Insertion sort er at den vil gjøre dobbelt så mange sammenligninger i det som svarer til én innsetting i Insertion sort.

Det er altså en sorteringsalgoritme med kvadratisk kjøretidskompleksitet (i likhet med bubble sort og selection sort). Den gjør akkurat like mange bytter som Insertion sort, men en god del flere sammenligninger.

På grunn av kjøretidskompleksiteten vil den være langt mindre egnet for store arrayer enn de raskeste sammenligningsbaserte sorteringsalgoritmene vi kjenner fra pensum (Merge, Heap og Quick).

Den er sammenligningsbasert, og er derfor vanskelig å sammenligne direkte mot Bucket- og Radix sort.

Algoritmen er (i likhet med Insertion sort) stabil, som er enkelt å se fordi den kun bytter elementer som står direkte ved siden av hverandre. Den er også in-place, fordi den ikke bruker andre datastrukturer enn arrayet som er gitt som input.

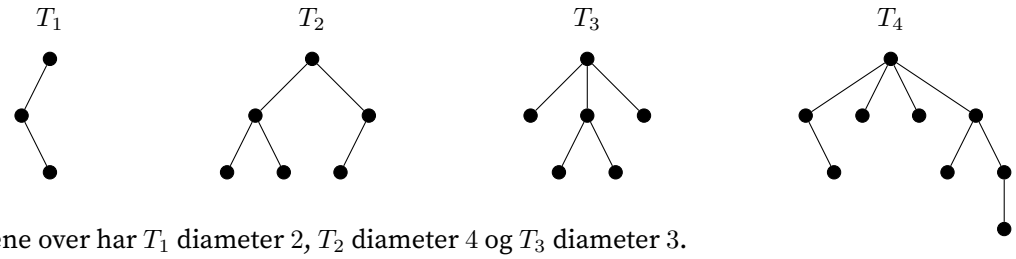
Dette er en litt friere oppgave, som gjør at sensor må yte eget skjønn. Her er noen veiledende kommentarer om poenggivning:

- Opptil 3 poeng for en god forklaring.
- Opptil 2 poeng for riktig kjøretidskompleksitet.
- Opptil 2 poeng for å «plassere» Gnome sort riktig i forhold til andre sorteringsalgoritmer.
- Opptil 2 poeng for en redegjørelse om stabilitet og in-place (disse krever lite forklaring).
- For full uttelling må studenten identifisere det svært nære beslektskapet med Insertion sort.

Det vektlegges at studenten viser god vurderingsevne med tanke på å trekke frem viktige momenter for å tegne et godt bilde av sorteringsalgoritmen sammenlignet med andre sorteringsalgoritmer. Lykkes de med det kan det gis noe mer uttelling enn hva punktene over foreslår.

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?

> Svaret er 5. Det gis 1 poeng for riktig svar.

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.

> Nei. Se for deg et komplett binærtre med mange noder; der vil den lengste stien gå gjennom rotnoden. Hvis du lager en ny rotnode som har det komplette binærtreet som sitt venstre subtre, så vil den lengste stien gå gjennom de samme nodene, og ikke gjennom den nye rotnoden.
> - Opptil 2 poeng for hele deloppgaven.
> - Det gis 1 poeng for riktig svar.
> - Det gis 1 poeng for en god begrunnelse.

(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.

Hvis vi først antar at rotnoden er med i den lengste stien, så vil den lengste stien gå fra den dypeste noden i det venstre subtreet, via rotnoden, og ned til den dypeste noden i det høyre subtreet. Høyden av et subtre angir antall kanter til den dypeste noden i treet.

Hvis vi antar at h_L og h_R er høydene til venstre og høyre subtre av rotnoden, så vil diameteren være gitt av $h_L + h_R + 2$.

Mer generelt vil den lengste stien som går gjennom en node være gitt av $h_L + h_R + 2$ gi (der vi antar at h_L og h_R er høydene av den aktuelle noden).

Altså kan problemet løses ved å for hver node beregne lengden på den lengste stien den inngår i, og til slutt returnere den lengste av disse lengdene. Dette kan gjøres i lineær tid.

```
1  Procedure Diameter(v)
2      d ← 0
3
4      Procedure DiameterRec(v)
5          if v = null then
6              return −1
7          h_L ← DiameterRec(v.left)
8          h_R ← DiameterRec(v.right)
9          d ← max(d, h_L + h_R + 2)
10         return 1 + max(h_L, h_R)
11
12     DiameterRec(v)
13     return d
```

Her er problemet løst ved å gå gjennom treet én gang. Den definerer en indre prosedyre for å unngå å bruke en global variabel, men dette er helt uvesentlig med tanke på vurdering av oppgaven.

Det er muligens enklere å løse dette ved å dele opp problemet i flere steg. For eksempel kan høyden i hver node beregnes først, deretter en lokal diameter for hver node, og til slutt løpe over alle nodene og finne den største diameteren. Ingen slike avveininger har noen påvirkning på poenggivningen.

Det kan være fort gjort å ende opp med en kvadratisk løsning.

- Opptil 8 poeng for en løsning i lineær tid.
- Opptil 5 poeng for en løsning i kvadratisk tid.
- Opptil 3 poeng for gode forsøk, men som ikke vil fungere.

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

1 poeng for riktig kjøretidskompleksitet avhengig av svaret som ble gitt i forrige deloppgave. Det kan gis poeng selv hvis det er store mangler i forrige deloppgave.

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?

> Problemet bør tenkes på som en komplett, vektet og urettet graf, der hver node i representerer en bygning og hver kant representerer en tunnel som kan bygninges ut. Kantene er vektet etter hvor mye det koster å grave tunnelen.
>
> Opptil 2 poeng for en god forklaring av hva slags graf oppgaven handler om.

(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.

> Vi kan løse problemet ved å finne et minimalt spenntre over G. Algoritmen vi kjenner best fra kurset er Prim sin algoritme for minimale spenntrær.
>
> Den fungerer ved å bygge et tre fra en vilkårlig valgt startnode. Naboene av startnoden plasseres på en prioritetskø, der vekten på kanten utgjør prioriteten. Grafen traverseres videre ved å plukke noder fra prioritetskøen, og legge dem til i treet dersom noden ikke allerede er i treet.
>
> Når alle noder er i treet vil vi ha bygninget et minimalt spenntre for G.
>
> Algoritmen er i $\mathcal{O}((|V| + |E|) \cdot \log(|V|))$, som kan forenkles til $\mathcal{O}(|E| \cdot \log(|V|))$ fordi grafen er sammenhengende. Siden grafen er komplett kan kjøretidskompleksiteten også uttrykkes som $\mathcal{O}(|V|^2 \cdot \log(|V|))$. Alle disse gir like god uttelling.
>
> - Opptil 5 poeng på deloppgaven for en god forklaring av en effektiv forklaring av en algoritme for minimale spenntrær.
> - Opptil 2 poeng for datastrukturene.
> - Opptil 2 poeng for forklaringen.
> - 1 poeng for riktig kjøretidskompleksitet.

(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.

Det resulterende tunnelsystemet er et *tre*, som betyr at det kun finnes én sti mellom hvert par av noder. Derfor kan du finne stien i $\mathcal{O}(|V|+|E|)$, med en hvilken som helst traversering, for eksempel BFS eller DFS.

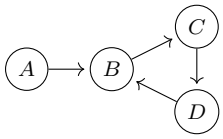
Å bruke Dijkjstra her er ikke helt veldefinert, fordi vektene på kantene som er beskrevet i oppgaven har med kostnaden av å bygninge tunnelene, ikke hvor lang tid det tar å gå i dem. Det vil riktignok gi riktig svar, fordi vektene ikke spiller noen rolle når det kun finnes én sti mellom hvert par av noder.

- Opptil 3 poeng for en løsning med BFS eller DFS.
- Opptil 1 poeng for en løsning med Dijkstra.
- 1 poeng for riktig kjøretidskompleksitet (for en løsning med Dijkstra må man ha riktig kjøretidskompleksitet for å få uttelling for deloppgaven).

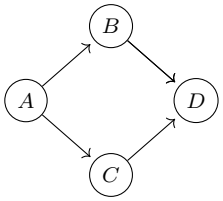
Dependency hell 12 poeng

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.

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 isCyclic(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 $\mathcal{O}(|V| \cdot (|V| + |E|))$.

- Opptil 5 poeng for en godt beskrevet algoritme som løser problemet i $\mathcal{O}(|V| + |E|)$.
- Opptil 3 poeng dersom løsningen er i $\mathcal{O}(|V| \cdot (|V| + |E|))$.

(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.

Det er tilstrekkelig å gjøre en traversering for hver node med inngrad null. Hvis vi sier at V_S er nodene i V som har inngrad null vil algoritmen være i $\mathcal{O}(|V_S| \cdot (|V| + |E|))$. Hvis vi antar at det kun er én node med inngrad null, så er algoritmen i $\mathcal{O}(|V| + |E|)$. Hvis vi antar at alle nodene har inngrad null, så algoritmen i $\mathcal{O}(|V|)$ (fordi alle søk vil starte og stoppe i samme node). Verste tilfelle må ligge et sted mellom disse ytterkantene, og være begrenset til $\mathcal{O}(|V| \cdot (|V| + |E|))$. Det er allikevel en betydelig forbedring fra å søke fra alle nodene.

```
1  Procedure isDiamond(G)
2      V_S ← the empty set
3
4      for v ∈ V do
5          if indegree of v is 0 then
6              add v to V_S
7
8      for s ∈ V_S do
9          stack ← stack containing s
10         visited ← empty set
11
12         while stack is not empty do
13             u ← stack.pop()
14             if u ∈ visited then
15                 return true
16             add u to visited
17             for (u, v) ∈ E do
18                 add v to stack
19
20     return false
```

En lignende løsning er å ta utgangspunkt i alle noder som har inngrad større eller lik to, og søke i den reverserte grafen.

Uten disse forbedringene vil en løsning typisk gjøre én traversering for hver node, som er i $\mathcal{O}(|V| \cdot (|V| + |E|))$.

- Opptil 7 poeng for en god løsning som er omtrent like effektiv som forslaget ovenfor.
- Opptil 5 poeng for en god løsning som gjør en traversering per node i grafen.
- Opptil 4 poeng for en god løsning som ikke er i $\mathcal{O}(|V| \cdot (|V| + |E|))$.

Generelt bør det gis noe uttelling for de som viser god forståelse for problemet, og skisserer en løsning som kunne fungert.