Eksamen i IN2010 høsten 2024

27. November 2024

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 22 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 $2 \cdot \max(n - 11, 0)$, 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 22 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.

Huffman-trær 10 poeng

Vi vil komprimere tekststrenger som består av symbolene `A`, `B`, `C` og `D` ved hjelp av Huffman-koding. De relative frekvensene er gitt av følgende frekvenstabell:

A	B	C	D
1	3	7	2

(a) Hva blir kodelengdene for de ulike symbolene i det tilhørende Huffman-treet?

2 poeng for riktig svar.

Kodelengdene for de ulike symbolene er:

A	B	C	D
3	2	1	3

(b) Hva inneholder løvnodene i det resulterende treet?

1 poeng for riktig svar.

De vil inneholde symbolene og de relative frekvensene.

Generelt i et Huffman-tre har hver node v:

- v.`symbol` – Symbolet til noden (eller **`null`**)
- v.`freq` – Frekvensen til noden
- v.`left` – Venstre barn av noden (eller **`null`**)
- v.`right` – Høyre barn av noden (eller **`null`**)

Et Huffman-tre bygges fra en mengde med par (s, f) der s er et symbol og f er en frekvens.

(c) Beskriv algoritmen for å bygge et Huffman-tre. Du kan skissere den med pseudokode, men en god beskrivelse med naturlig språk gir like god uttelling.

Maksimalt 7 poeng.

Algoritmen består av å opprette en prioritetskø som ordnes etter frekvensen til noden. Vi oppretter en node for hvert symbol- og frekvens par (uten barn) og plasserer disse på køen. Vi vil så ta ut to noder u og v fra prioritetskøen, og lage en ny node w (uten noe satt symbol) som har u og v som barn, og frekvens som svarer til summen av frekvensene til u og v. Den nye noden plasseres på prioritetskøen. Algoritmen terminerer når det kun er én node igjen på prioritetskøen, og den noden vil være rotnoden for Huffman-treet, som returneres som siste steg av algoritmen.

Input: En mengde C med par (s, f) der s er et symbol og f er en frekvens
Output: Et Huffman-tre

```
1  Procedure Huffman(C)
2      Q ← new PriorityQueue
3      for (s, f) ∈ C do
4          Insert(Q, new Node(s, f, null, null))
5      while Size(Q) > 1 do
6          v1 ← RemoveMin(Q)
7          v2 ← RemoveMin(Q)
8          f ← v1.freq + v2.freq
9          Insert(Q, new Node(null, f, v1, v2))
10     return RemoveMin(Q)
```

- 2.5 poeng for å få frem at det brukes en prioritetskø ordnet etter frekvens.
- 2.5 poeng for hvordan to noder slås sammen.
- 1 poeng for termineringskriterie.
- 1 poeng for returverdien.

En student som viser god forståelse for algoritmen kan få full uttelling selv hvis noen av momentene ovenfor ikke blir nevnt eksplisitt.

Etymologisk ordliste 10 poeng

En etymologisk ordliste er et oppslagsverk for et ords opphav. Her er et eksempel fra kategorien Matematikk:

> **heksagon:** seksvinklet figur. Av gresk *hex* «seks» + *gonía* «vinkel».
> *(Kilde: Norsk etymologisk ordbok – tematisk ordnet, Yann De Caprona)*

Anta at et ordobjekt v kun har følgende felter:

- v.`string` – Ordet representert som en streng
- v.`category` – En streng som angir kategorien ordet tilhører

Du kan anta at kategorien til et ord er en av N kategorier, der $N < 100$.

(a) Arrayet A inneholder hele den etymologiske ordlisten ordnet alfabetisk (altså ordnet etter `string`). Skriv en prosedyre som *stabilt* sorterer arrayet A *tematisk* (altså ordnet etter `category`). Prosedyren skal ikke inneholde kall på sorteringsalgoritmer.

```
Input: Et array A med n ordobjekter
Output: Et array A med de samme n ordobjektene som er ordnet etter category
1 Procedure EtymologySort(A)
  | // ···
```

Maksimalt 6 poeng.

Denne oppgaven løses best med bucket sort, og en rimelig gjengiving av bucket sort tilpasset oppgaven gir 6 poeng.

En annen stabil sorteringsalgoritme i $\mathcal{O}(n \cdot \log(n))$ gir opptil 4 poeng. Det trekkes 1 poeng dersom algoritmen ikke er stabil. En sorteringsalgoritme i $\mathcal{O}(n^2)$ gir opp til 2 poeng.

```
Input: Et array A med n ordobjekter
Output: Et array A med de samme n ordobjektene som er ordnet etter category
1  Procedure EtymologySort(A)
2  |  B ← ordered dictionary with an empty list as default value
3  |  for i ← 0 to n − 1 do
4  |  |  k ← A[i].category
5  |  |  Append A[i] to the list B[k]
6  |  j ← 0
7  |  for k in keys of B do
8  |  |  for x in B[k] do
9  |  |  |  A[j] ← x
10 |  |  |  j ← j + 1
11 |  return A
```

(b) Anta nå at du har en ordbok K, som assosierer hver kategori til et intervall som angir hvor kategorien starter og slutter i arrayet. For hvert ordobjekt v vil $K[v.\texttt{category}] = [\texttt{low}, \texttt{high}]$, der `low` og `high` gir indeksen til henholdsvis det første, og det siste, ordobjektet som tilhører kategorien til v. Du skal nå søke etter et gitt ordobjekt i arrayet A slik det blir returnert fra `EtymologySort`.

Oppgi hvordan du kan bruke en algoritme, kjent fra pensum, til å sjekke om et ordobjekt v finnes i arrayet A. Oppgi også kjøretidskompleksiteten til algoritmen.

Maksimalt 4 poeng.

Siden sorteringen ovenfor er stabil, vil ordene være ordnet alfabetisk innenfor hvert tema, så vi kan bruke binærsøk og gjøre dette i $\mathcal{O}(\log(n))$. I stedet for et binærsøk som starter med `low` $= 0$ og `high` $= n - 1$, så vil vi heller starte med `low` og `high` som gitt av $K[v.\texttt{category}]$ og søke etter $v.\texttt{string}$.

Det gis 2 poeng for lineærtidsløsninger.

Jazz eller klassisk? 10 poeng

Du ønsker å invitere venner med på konsert, og er usikker på om du skal invitere til *jazz* eller *klassisk*, som er de to eneste sjangrene du liker. Gjennom et lite musikknettverk har du god oversikt over din egen musikksmak, musikksmaken til dine venner, dine venners venner og dine venners venners venner etc. Musikksmaken til en venn (eller en venns venn etc.) v er gitt av en ordbok M, der $M[v]$ er en mengde av sjangre. For eksempel, kan $M[v]$ være $\{\texttt{"jazz"}, \texttt{"pop"}, \texttt{"folk"}\}$.

(a) Hvordan kan du representere musikknettverket som en graf? Hva representerer nodene og kantene i grafen?

> Maksimalt 1 poeng.
>
> Dette er en urettet graf $G = (V, E)$, der nodene er personer i nettkverket og kantene er vennskap. Det er også mulig å tenke at vennskap er asymmetrisk, som gjør grafen rettet; det gjør ingen forskjell for algoritmen under, og gir like mye uttelling.
>
> Oppgaveteksten er litt uklar (selv om den blir klarere sett i sammenheng med resten av oppgaveteksten). Derfor er vi svært snille i poenggivningen på denne.

Selv om du gjerne vil ha med deg mange på konsert, er det begrenset hvor perifere venner du vil ha med. Du setter en grense k som angir hvor mange venneledd du er villig til å gå gjennom. For eksempel, hvis $k = 1$, vil du kun invitere dine egne venner. Hvis $k = 2$ vil du både invitere dine egne venner og deres venner.

(b) Skriv en algoritme som tar en graf $G = (V, E)$, en ordbok M, en startnode $s \in V$, og et positivt heltall k som input. Den skal returnere `"jazz"` eller `"klassisk"` avhengig av hvilken sjanger som flest liker innen k ledd av venner. Du kan anta at det ikke er like mange som liker jazz og klassisk.

Input: En graf $G = (V, E)$, en ordbok M, en startnode $s \in V$ og et heltall k
Output: En streng `"jazz"` eller `"klassisk"`

```
1 Procedure JazzOrClassical(G, M, s, k)
    // ···
```

Maksimalt 7 poeng.

Dette kan løses med en traversering fra s som stopper ved dybde k, der vi teller opp antall jazz og klassisk. Det bør *ikke* gjøres en full traversering.

```
1  Procedure JazzOrClassicalVisit(G, u, k, visited)
2      if k = 0 then
3          return
4      add u to visited
5      for (u, v) ∈ E do
6          if v ∉ visited then
7              JazzOrClassicalVisit(G, v, k − 1, visited)
8
9  Procedure JazzOrClassical(G, M, s, k)
10     visited ← empty set
11     JazzOrClassicalVisit(G, s, k, visited)
12     jazz ← 0
13     classiscal ← 0
14     for v ∈ visited do
15         if "jazz" ∈ M[v] then
16             jazz ← jazz + 1
17         if "klassisk" ∈ M[v] then
18             classical ← classical + 1
19     if jazz > classical then
20         return "jazz"
21     return "klassisk"
```

Det trekkes 3 poeng dersom k ikke tas høyde for i det hele tatt. Det trekkes 1–2 poeng dersom k ikke håndteres riktig avhengig av hvor godt forsøket er.

(c) Nå vil du sende ut invitasjoner. Du ønsker *først* å invitere dine egne venner, *deretter* dine venners venner, og så videre. Hvilken algoritme, kjent fra pensum, egner seg til å sende ut invitasjoner på denne måten?

Maksimalt 2 poeng.

Her ønsker vi å gå *lagvis* gjennom grafen, som oppnås enkelt ved hjelp av et bredde-først søk.

Python 3.6 — 10 poeng

Anta at Python sine ordbøker (`dict`), frem til versjon 3.6, var implementert med linear probing slik det har blitt undervist i IN2010. (Dette er en liten modifikasjon av sannheten).

I desember 2012 kom Raymond Hettinger med et forslag til en endring av den underliggende representasjonen av ordbøker. I forslaget beskriver Hettinger at den gjeldende representasjonen er unødvendig ineffektiv.

Den organiserer dataene i et array `entries`, der hvert element har plass til en nøkkel k og en verdi v. Arrayet har størrelse N, og mye ledig plass. Hettinger foreslår å omorganisere dataene. Nøkkel-verdi-parene bør i stedet lagres i et tettpakket, dynamisk array `entries` med n elementer, der n er antall nøkkel-verdi-par. Elementene i `entries` kan refereres til fra et annet array `indices` med N elementer (som fremdeles har mye ledig plass).

For eksempel, følgende ordbok:

```
d = {}
d["timmy"] = "red"
d["barry"] = "green"
d["gudio"] = "blue"
```

var tidligere representert slik:

```
entries = [(_, _),
           ("barry", "green"),
           (_, _),
           (_, _),
           (_, _),
           ("timmy", "red"),
           (_, _),
           ("guido", "blue")]
```

der `(_, _)` indikerer en ledig plass. Forslaget er å representere den slik:

```
indices = [_, 1, _, _, _, 0, _, 2]
entries = [("timmy", "red"), ("barry", "green"), ("guido", "blue")]
```

Forslaget over førte til at implementasjonen av ordbøker i Python ble endret fra og med versjon 3.6.

(a) Forklar kort, med naturlig språk, hvordan algoritmene for oppslag og innsetting med linear probing må endres for å ta høyde for den nye strukturen.

> Maksimalt 4 poeng. Omtrent 2 poeng for oppslag og 2 poeng for innsetting.
>
> I den nye strukturen vil oppslaget fremdeles starte med å beregne hashen $i = h(k, N)$ av nøkkelen k, der N er størrelsen av `indices`. Vi vil så slå opp `indices`$[i]$, og dersom den ikke er tom, så vil vi sammenligne nøkkelen i `entries`, som da kan finnes på `entries`$[$`indices`$[i]]$. Dersom nøkkelen ikke stemmer overens med nøkkelen k, vil vi øke i med 1 (modulo N), og slå opp på nytt. Dersom vi treffer en tom plass avsluttes søket.
>
> Det viktigste poenget for oppslag er at vi slår opp i `entries` via `indices`.
>
> For innsetting finner vi første ledige plass i `indices` med samme teknikk som ved oppslag. Hvis den første ledige plassen er j, så settes `indices`$[j]$ til størrelsen av `entries`. Til slutt legger vi til nøkkel-verdi-paret på slutten av `entries`.

(b) Hva er en rehash? Hva er det som forårsaker en rehash?

> Maksimalt 2 poeng.
>
> En rehash forårsakes av at load faktoren $\frac{n}{N}$ er for høy, altså at det ikke er så mye ledig plass. Da øker vi størrelsen på det underliggende arrayet (som var av størrelse N), og setter inn alle elementene på nytt (med innsettingsalgoritmen til linear probing).

(c) Hvordan vil en rehash foregå i den nye strukturen?

Maksimalt 2 poeng.

I den nye strukturen vil `entries` kunne forbli uendret. Vi vil heller øke størrelsen på `indices`, og gjøre innsetting på alle elementene som beskrevet ovenfor (med unntak av å legge til i `entries`).

(d) Hva er fordelene med den nye strukturen?

Maksimalt 2 poeng.

Her er det flere fordeler:

- Elementene kan hentes ut i en forutsigbar rekkefølge (altså, rekkefølgen de ble satt inn i).
- Iterering krever n steg i stedet for N steg (og N er alltid større enn n).
- Rehashing påvirker ikke `entries`.
- Lavere minnebruk, som følge av at det største arrayet `indices` kun trenger å inneholde indekser. I praksis bruker den 20% til 25% mindre minne. Disse tallene tallene er ikke mulig å forutsi fra beskrivelsen over, men man kan kanskje se at det vil resultere i lavere minnebruk.

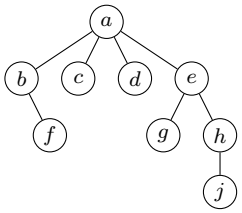
Det gis full uttelling hvis to av momentene ovenfor blir nevnt. Listen over er ikke nødvendigvis komplett, så andre gode poenger kan også gi uttelling.

Laveste felles forfeder 12 poeng

Den *laveste felles forfederen* av to noder u og v i et tre T er den laveste (altså noden med høyest dybde) som har både u og v som en etterkommer.

Her er noen eksempler fra treet ved siden av:

- Laveste felles forfeder av a og a er a
- Laveste felles forfeder av b og e er a
- Laveste felles forfeder av g og j er e
- Laveste felles forfeder av e og j er e



(a) Forklar kort hvorfor det for hvert par av noder må eksistere en felles forfeder.

Maksimalt 1 poeng. Hvert par av noder vil ha en felles forfeder, fordi rotnoden er en forfeder av alle noder i treet.

(b) Anta at hvis v er en node, har v kun følgende felter:

- v.`parent` – foreldernoden til v
- v.`children` – barnenodene til v
- v.`depth` – dybden til v

Gi en algoritme som, gitt to noder u og v, finner og returnerer deres laveste felles forfeder.

```
Input: To noder u of v
Output: Laveste felles forfeder av u og v
1 Procedure LowestCommonAncestor(u, v)
  |  // ···
```

Maksimalt 5 poeng.

```
Input: To noder u of v
Output: Returnerer laveste felles forfeder av u og v
1 Procedure LowestCommonAncestor(u, v)
2   if u = v then
3   |  return u
4   if u.depth < v.depth then
5   |  return LowestCommonAncestor(u, v.parent)
6   return LowestCommonAncestor(u.parent, v)
```

En god løsning vil være i $\mathcal{O}(h)$ der h er høyden til treet (slik som den over). Maksimalt 3 poeng for løsninger i lineær tid.

(c) Anta nå at treet er et *binært søketre av unike heltall*. Hvis v er en node, så har v kun følgende felter:

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

Gi en algoritme som, gitt roten av treet og elementene x og y, returnerer den laveste felles forfeder av nodene som henholdsvis har elementene x og y der $x < y$. Du kan anta at begge elementene x og y finnes i treet.

```
Input: En node v, og to verdier x og y der x < y
Output: Laveste felles forfeder av nodene som har elementene x og y
1 Procedure LowestCommonAncestorBST(v, x, y)
  |  // ···
```

Maksimalt 5 poeng.

Input: En node v og to verdier x og y der $x < y$
Output: Returnerer laveste felles forfeder nodene som har elementene x og y

```
1 Procedure LowestCommonAncestorBST(v, x, y)
2 |  if x ≤ v.element ≤ y then
3 |  |  return v
4 |  if y < v.element then
5 |  |  return LowestCommonAncestorBST(v.left, x, y)
6 |  return LowestCommonAncestorBST(v.right, x, y)
```

En god løsning vil være i $\mathcal{O}(h)$ der h er høyden til treet (slik som den over). Maksimalt 3 poeng for løsninger i lineær tid.

(d) Hva er kjøretidskompleksiteten til algoritmen din fra (c) for et tre med n noder i verste tilfelle? Hva er kjøretidskompleksiteten dersom du antar at treet er balansert?

Hvis du ikke har svart på (c) kan du oppgi svaret for din løsning på (b) i stedet.

Maksimalt 1 poeng for riktig kjøretidskompleksitet.

Mediankø 12 poeng

En *mediankø* er en abstrakt datatype for en samling som støtter følgende operasjoner:

- `insert`(Q, x) – plasserer et element x på køen
- `removeMedian`(Q) – fjerner og returnerer *median*-elementet fra køen

Medianen er det «midterste» elmentet i samlingen, dersom elementene ordnes fra minst til størst. Dersom vi setter inn tallene $8, 3, 1, 5$ og 7 i en mediankø, så vil:

- første kall på `removeMedian` gi 5
- andre kall på `removeMedian` gi 3
- tredje kall på `removeMedian` gi 7
- fjerde kall på `removeMedian` gi 1
- femte kall på `removeMedian` gi 8

Merk at vi velger det minste av de to midterste elementene når køen inneholder et partall antall elementer, som vist i eksempelet ovenfor.

(a) Hvis vi setter inn tallene $4, 1, 8, 3, 2$ på en mediankø, hva vil de tre første kallene på `removeMedian` gi? Oppgi svaret som en kommaseparert liste med tall.

> 1 poeng for $3, 2, 4$.

(b) Forklar *kort* hvordan du vil representere en mediankø.

> 1 poeng for en forståelig forklaring. Må ses i sammenheng med neste deloppgave.

(c) Forklar hvordan du ville ha implementert operasjonene `insert` og `removeMedian`. Du kan bruke pseudokode som en del av forklaringen din, men det er ikke et krav. Alle datastrukturer nevnt i pensum er tilgjengelige for deg.

Maksimalt 6 poeng.

Det finnes mange måter å gå løs på dette problemet. En naiv løsning er å sette elementene inn i en usortert struktur (liste eller dynamisk array), og sortere med en effektiv sorteringsalgoritme for å finne medianen ved behov. Maksimalt 2 poeng for slike løsninger.

En løsning som bruker sortert innsetting i en liste får lineær tid på innsetting, og det vil være mulig å holde styr på hvor medianen er, slik at den kan finnes i konstant tid. Hvis man holder styr på hvor midten er, og bruker en dobbeltlenket liste, så kan man fjerne det minste elementet i konstant tid. Maksimalt 4 poeng for en slik løsning.

Den beste løsningen (vi har funnet) er å bruke to prioritetskøer (eller binære heaps); en max-heap og en min-heap, som lar oss få logaritmisk tid på begge operasjoner. Vi lar $Q.\texttt{small}$ være en max-heap og $Q.\texttt{large}$ være en min-heap. Hvis vi sørger for at $Q.\texttt{small}$ inneholder den minste halvparten av elementene og $Q.\texttt{large}$ inneholder den største halvparten, vil toppen av $Q.\texttt{small}$ alltid være medianen.

Merk at denne løsningen, som involverer å holde to datastrukturer like store, ligner på oppgaven Teque, som er blitt gitt som en innleveringsoppgave.

```
1  Procedure balance(Q)
2  |  if |Q.large| + 1 < |Q.small| then
3  |  |  x ← removeMax(Q.small)
4  |  |  insert(Q.large, x)
5  |  if |Q.small| < |Q.large| then
6  |  |  x ← removeMin(Q.large)
7  |  |  insert(Q.small, x)
8
9  Procedure insert(Q, x)
10 |  if Q.small is empty or x < peek(Q.small) then
11 |  |  insert(Q.small, x)
12 |  else
13 |  |  insert(Q.large, x)
14 |  balance(Q)
15
16 Procedure removeMedian(Q)
17 |  x ← removeMax(Q.small)
18 |  balance(Q)
19 |  return x
```

For full uttelling må forklaringen være dekkende nok til at man kan implementere algoritmene med utgangspunkt i forklaringen.

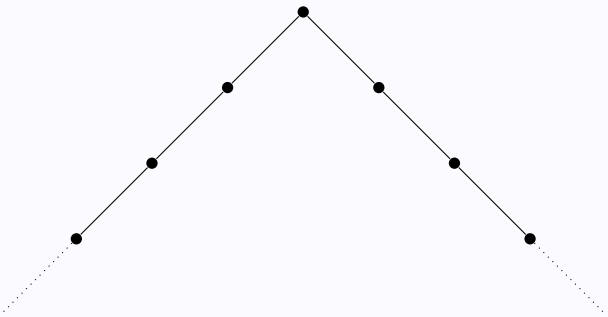
(d) Basert på forklaringen din i (c), hva kan du si om kjøretidskompleksiteten til `insert` og `removeMedian`?

Maksimalt 2 poeng, der det gis 1 poeng for hver prosedyre. For løsningen ovenfor er begge prosedyrer i $\mathcal{O}(\log(n))$.

(e) Hvis vi tar n elementer fra en mediankø ved hjelp av `removeMedian`, og setter dem inn i et binært søketre, én etter én, hvordan vil det resulterende binære søketreet se ut? Forklar kort.

Maksimalt 2 poeng.

I venstre subtre fra roten vil alle noder kun ha venstre barn, og i høyre subtre fra roten vil alle noder kun ha høyre barn. Det vil altså se slik ut:



Rundtur med Den Norske Turistforeningen (DNT) 12 poeng

Fagutvalget ved institutt for informatikk har bestemt seg for å arrangere en rundtur i et vakkert norsk fjellandskap for alle IFI-studenter. Dette sammenfaller med at Den Norske Turistforeningen har lansert en ny app, sammen med offentlige datasett, slik at det er gode muligheter for å gjøre et godt informert rutevalg. Siden de fleste studenter på IFI har en forkjærlighet for grafer, er det ikke til å unngå at de vil bruke datasettet til å bygge en graf som fanger nok informasjon til å kunne algoritmisk bestemme den beste rundturen.

Grafen som bygges er en rettet graf $G = (V, E)$, der hver node $u \in V$ representerer en lokasjon, og en kant $(u, v) \in E$ representerer en turvei fra u til v. Ingen noder har en kant til seg selv.

(a) Det viser seg at grafen har en ganske spesiell egenskap: *Hver node inngår i makismalt én sykel*. Hver sykel representerer en *rundtur*.

Gi en algoritme som finner alle rundturene i grafen, og oppgi kjøretidskompleksiteten. Du kan kalle på algoritmer kjent fra pensum.

Maksimalt 5 poeng.

Her er det tilstrekkelig å svare at vi finner de sterkt sammenhengende komponentene av G og returnerer alle komponenter som har størrelse større enn én. Dette er i $\mathcal{O}(|V| + |E|)$.

```
Input: En graf G = (V, E)
Output: En mengde av mengder av noder
Procedure RoundTrip(G)
    roundtrips ← empty set
    components ← StronglyConnectedComponents(G)
    for component ∈ components do
        if 1 < |component| then
            add component to roundtrips
    return roundtrips
```

Det trekkes 1 poeng for å ikke utelukke komponenter som kun inneholder én node.

(b) En av rundturene fra (a) blir valgt som reisemål, hvor t er en node i rundturen. Turen starter ved et *utfartssted*, som er enkelt å komme seg til med kollektivtransport. Et utfartssted er identifisert ved noder som har inngrad 0. Det siste som gjenstår i planleggingen er å velge et egnet utfartssted turen skal starte ved for å nå t.

Datasettet inneholder heldigvis også informasjon om hvor lang tid en gjennomsnittlig turgåer bruker langs en turvei. Dette blir gitt som en vektfunksjon w, slik at for hver kant $(u, v) \in E$, så angir $w(u, v)$ tidsbruk som et positivt heltall. Vi vil bruke vektene til å finne ut hvilket utfartssted som er best egnet for å komme seg til reisemålet.

Input er en rettet og vektet graf $G = (V, E)$ med vektfunksjon w, en mengde med utfartssteder U og en node t i reisemålet. Output er noden $s \in U$ slik at avstanden mellom s og t er så lav som mulig.

Gi en algoritme som effektivt finner utfartsstedet som har lavest avstand til t i reisemålet, og oppgi kjøretidskompleksiteten. Du kan kalle på algoritmer kjent fra pensum.

Maksimalt 7 poeng.

En mulig løsning er å bruke Dijkstra sin algoritme for korteste stier fra hver node $s \in U$, og velge noden som har kortest avstand til t. Dette er i $\mathcal{O}(|U| \cdot (|V| + |E|) \cdot \log(|V|))$. En beskrivelse av en slik løsning kan få opp til 4 poeng.

En bedre løsning er å heller finne korteste avstander (med Dijkstra) *fra t i den reverserte grafen G_r*, altså grafen der alle kanter er snudd. I en slik løsning kan man først beregne avstander fra t til alle noder, og deretter gå gjennom alle noder i U og finne ut hvilken som er nærmest t. Dette er i $\mathcal{O}((|V|+|E|) \cdot \log(|V|))$, altså samme kjøretidskompleksitet som Dijkstra. En slik løsning kan få opp til 7 poeng.

Input: En graf $G = (V, E)$ med vektfunksjon w, en mengde med noder U og en node t
Output: En node $s \in U$ som minimerer avstanden til t

```
Procedure BestStartLocation(G, U, t)
    G_r ← ReverseGraph(G) // Anta at vektfunksjonen også reverseres
    dist ← Dijkstra(G_r, t)
    s ← arbitrary node in U
    for u ∈ U do
        if dist[u] < dist[s] then
            s ← u
    return s
```

En alternativ løsning er å legge til en ny node a i grafen, og legge til en kant (a, s) med $w(a, s) = 0$ for alle noder $s \in U$. Hvis vi gjør Dijkstra med a som startnode, så kan man finne det beste utfartsstedet ved å finne den *andre* noden i den korteste stien fra a til t, som er garantert å være et utfartssted. Den vil være like effektiv som løsningen over og kan også få opp til 7 poeng.