Eksamen i IN2010 høsten 2020

10. Desember 2020

Om eksamen

- Eksamen består av en blanding av flervalgsoppgaver, tekstsvar, og koding. Alle besvarelser skal skrives inn i Inspera, det er ingen mulighet for opplasting av håndskrevede svar.
- På flervalgsoppgaver gir korrekte svar full uttelling, gale svar gis 0 poeng (man får ikke negative poeng).
- Besvarelsen skal være et selvstendig arbeid.
- Alle hjelpemidler er tillatt (lærebok, nettressurser, notater, etc.).
- Under eksamen er det ikke tillatt å samarbeide eller kommunisere med andre personer om oppgaven eller å dele sitt arbeid med andre.
- På nettsiden om brukerstøtte for hjemmeeksamen finner du kontaktpunkter for spørsmål knyttet til gjennomføringen.
- For øvrig gjelder informasjonen på nettsiden om eksamensavvikling ved MN høsten 2020.

Kommentarer

- Det kanskje viktigste tipset er *å lese oppgaveteksten svært nøye*.
- Pass på at du svarer på nøyaktig det oppgaven spør om.
- 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. Forelesningsmateriale kan anses som gode eksempler på pseudokode, men det er også fullt mulig å skrive mer Java- eller Python-aktig syntaks.

Om sensur

- Eksamen vil bli vurdert med en bokstavkarakter. Det legges stor vekt på at besvarelsene er oversiktlige og at forklaringene er gode.
- Etter eksamen kan man trekkes ut til en samtale for å kontrollere eierskap til sin besvarelse. Denne samtalen har ikke noen innvirkning på sensuren eller karakteren, men kan lede til at instituttet oppretter en fuskesak. Dette er beskrevet på UiOs nettside om kontrollsamtaler og nettside om rutiner for behandling av mistanke om fusk.

> Sensorveiledningen inneholder et løsningsforslag på alle oppgaver. I tillegg gis det veiledning på hvordan poeng kan fordeles, men sensoren står fri 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):

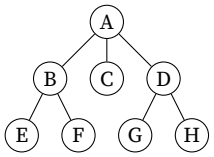
- Algoritmer er idéene bak de programmene vi skriver. De angir hva som må gjøres for å løse et gitt problem på en presis og entydig måte. Følger du en algoritme for å løse et problem så skal du også få riktig svar til slutt, og helst på rimelig tid.
- Datastrukturer er måter å organisere data på. Stort 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.

Binære søketrær — 4 poeng

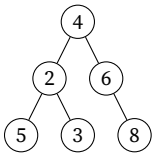
(a) — 1 poeng

Treet til høyre er *ikke* et binærtre. Hvilken node må fjernes for at det skal bli et binærtre?



(b) — 1 poeng

Treet til høyre er *ikke* et binært søketre. Hvilken node må fjernes for at treet skal bli et binært søketre?



(c) — 1 poeng

Er hvert binært søketre et AVL tre?

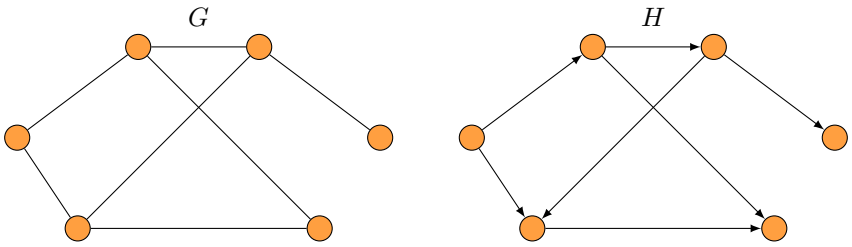
(d) — 1 poeng

Er hvert AVL tre et binært søketre?

(a) Noden som inneholder C.
(b) Noden som inneholder 5.
(c) Nei.
(d) Ja.

Grafegenskaper 8 poeng

Vi er gitt en urettet graf G, og en rettet graf H som ser slik ut:



- Må vi legge til eller fjerne kanter for at G skal bli sammenhengende?
- Må vi legge til eller fjerne kanter for at G *ikke* skal bli sammenhengende?
- Må vi legge til eller fjerne kanter for at G skal bli et tre?
- Må vi legge til eller fjerne kanter for at G *ikke* skal bli et tre?
- Må vi legge til eller fjerne kanter for at H skal bli en DAG?
- Må vi legge til eller fjerne kanter for at H *ikke* skal bli en DAG?
- Må vi legge til eller fjerne kanter for at H skal bli sterkt sammenhengende?
- Må vi legge til eller fjerne kanter for at H *ikke* skal bli sterkt sammenhengende?

- Må vi legge til eller fjerne kanter for at G skal bli sammenhengende?
  - den er allerede sammenhengende, så ingenting
  - legge til kanter
  - fjerne kanter
- Må vi legge til eller fjerne kanter for at G *ikke* skal bli sammenhengende?
  - den er allerede ikke sammenhengende, så ingenting
  - legge til kanter
  - fjerne kanter
- Må vi legge til eller fjerne kanter for at G skal bli et tre?
  - den er en allerede et tre, så ingenting
  - legge til kanter
  - fjerne kanter
- Må vi legge til eller fjerne kanter for at G *ikke* skal bli et tre?
  - den er allerede ikke et tre, så ingenting
  - legge til kanter
  - fjerne kanter
- Må vi legge til eller fjerne kanter for at H skal bli en DAG?
  - den er en allerede en DAG, så ingenting
  - legge til kanter
  - fjerne kanter
- Må vi legge til eller fjerne kanter for at H *ikke* skal bli en DAG?
  - den er allerede ikke en DAG, så ingenting
  - legge til kanter
  - fjerne kanter
- Må vi legge til eller fjerne kanter for at H skal bli sterkt sammenhengende?
  - den er allerede sterkt sammenhengende, så ingenting
  - legge til kanter
  - fjerne kanter
- Må vi legge til eller fjerne kanter for at H *ikke* skal bli sterkt sammenhengende?
  - den er allerede ikke sterkt sammenhengende, så ingenting
  - legge til kanter
  - fjerne kanter

Linear probing 2 poeng

Vi starter med et tomt array på størrelse 10.

0	1	2	3	4	5	6	7	8	9

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

Bruk *linear probing* til å sette inn disse tallene i den gitte rekkefølgen:

$$93, 48, 74, 99, 29, 13, 45$$

Fyll ut tabellen slik den ser ut etter alle tallene er satt inn med linear probing.

Det resulterende arrayet ser slik ut:

29			93	74	13	45		48	99
0	1	2	3	4	5	6	7	8	9

Søk 12 poeng

Anta at vi har tre prosedyrer:

- `Search` – som gjør et rett frem søk på lineær tid
- `BinarySearch` – som gjør et binærsøk
- `Sort` – sorterer i $\mathcal{O}(n \cdot \log(n))$

Prosedyrene nedenfor tar et array `A` med heltall av størrelse n og et array `B` av størrelse k som input. Alle skal printe tallene som er i både `A` og `B`.

```
1 Procedure Intersection1(A, B)
2 |  for x in B do
3 |  |  if Search(A, x) then
4 |  |  |  print(x)
```

```
1 Procedure Intersection2(A, B)
2 |  for x in B do
3 |  |  if BinarySearch(A, x) then
4 |  |  |  print(x)
```

```
1 Procedure Intersection3(A, B)
2 |  Sort(A)
3 |  for x in B do
4 |  |  if BinarySearch(A, x) then
5 |  |  |  print(x)
```

```
1 Procedure Intersection4(A, B)
2 |  for x in B do
3 |  |  Sort(A)
4 |  |  if BinarySearch(A, x) then
5 |  |  |  print(x)
```

Tre av disse prosedyrene gir alltid riktig svar, og én kan gi feil svar. Vi lurer på når man burde bruke hvilken variant av `Intersection` avhengig av hvor stor k (altså lengden av `B`) er i forhold til n (altså lengden til `A`). Du trenger bare å oppgi ett riktig svar hvis flere prosedyrer gir opphav til samme kjøretidskompleksitet.

(a) Hvis `B` inneholder ett element, altså $k = 1$, hvilken prosedyre gir riktig svar med minimal kjøretidskompleksitet?

(b) Hvis `B` inneholder $\log(n)$ elementer, altså $k = \log(n)$, hvilken prosedyre gir riktig svar med minimal kjøretidskompleksitet?

(c) Hvis `B` inneholder n elementer, altså $k = n$, hvilken prosedyre gir riktig svar med minimal kjøretidskompleksitet?

(d) Hvis `B` inneholder n^2 elementer, altså $k = n^2$, hvilken prosedyre gir riktig svar med minimal kjøretidskompleksitet?

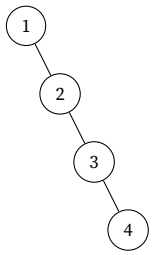
`Intersection2` gir ikke nødvendigvis riktig svar, fordi man kan ikke uten videre anta at input er sortert. `Intersection4` gir aldri best kjøretid, fordi å sortere for hver iterasjon er overflødig.

Det mest interessante tilfellet er der $k = \log(n)$. Da er `Intersection1` i $O(n \cdot k) = O(n \log(n))$ og `Intersection3` er i $O(n \log(n) + \log(n) \cdot k) = O(n \log(n) + \log(n) \cdot \log(n)) = O(n \log(n))$. Altså er begge svar korrekte, og gir full uttelling.

- Hvis `B` inneholder ett element, altså $k = 1$, hvilken prosedyre gir riktig svar med minimal kjøretidskompleksitet?
  - `Intersection1`
  - `Intersection2`
  - `Intersection3`
  - `Intersection4`
- Hvis `B` inneholder $\log(n)$ elementer, altså $k = \log(n)$, hvilken prosedyre gir riktig svar med minimal kjøretidskompleksitet?
  - `Intersection1`
  - `Intersection2`
  - `Intersection3`
  - `Intersection4`
- Hvis `B` inneholder n elementer, altså $k = n$, hvilken prosedyre gir riktig svar med minimal kjøretidskompleksitet?
  - `Intersection1`
  - `Intersection2`
  - `Intersection3`
  - `Intersection4`
- Hvis `B` inneholder n^2 elementer, altså $k = n^2$, hvilken prosedyre gir riktig svar med minimal kjøretidskompleksitet?
  - `Intersection1`
  - `Intersection2`
  - `Intersection3`
  - `Intersection4`

AVL

9 poeng



Vanligvis gjør vi AVL-innsetting på AVL-trær, men vi kan bruke samme prosedyre på et ordinært binært søketre (gitt at hver node inneholder høyden for sitt subtre).

Til høyre ser du et binært søketre, som *ikke* er et AVL-tre. Sett inn 5 i treet ved bruk av AVL-innsetting.

(a)

2 poeng

Hvordan ser treet ut etter den første rotasjonen i innsettingen?

(b)

4 poeng

Merk at en dobbelrotasjon telles som to enkle rotasjoner.
Hvor mange enkle rotasjoner blir utført under innsetting?

(c)

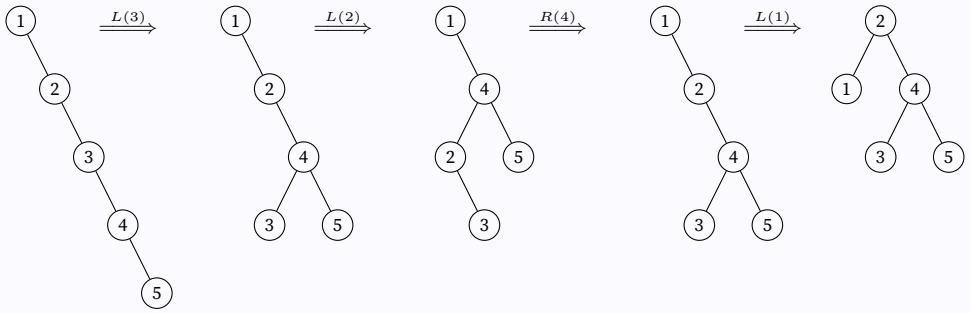
1 poeng

Er det resulterende treet et AVL-tre?

(d)

2 poeng

Hva er verdien i rotnoden etter innsetting?



(a) Se det andre treet i illustrasjonen.
(b) Det blir gjort 4 rotasjoner.
(c) Ja, det blir et AVL-tre
(d) Verdien i rotnoden er 2.

Subanagram poeng

Vi ønsker å lage en nettside for å hjelpe sleipe Scrabble-spillere. En Scrabble-spiller har noen bokstaver som de skal prøve å forme ord fra, og disse ordene må stå i en ordbok. I denne oppgaven ignorerer alle andre Scrabble-regler, og fokuserer på det vi skal kalle *subanagrammer*.

La oss anta at bokstavene man har tilgjengelig er gitt som en streng S, og at et ord W også er gitt som en streng. Vi sier at W er et *subanagram* av S, dersom W kan skrives ved å kun bruke bokstaver fra S.

Merk at vi ikke tillater gjenbruk av bokstaver fra S. For eksempel er `"hole"` et subanagram av `"ehlo"`. Derimot er `"hello"` *ikke* et subanagram av `"ehlo"`, fordi det ikke er to `"l"`-er i `"ehlo"`. Både `"hole"` og `"hello"` er subanagrammer av `"hheelloo"`.

Her er noen eksempler på subanagramer av `"aghilmort"`:

- `"algorithm"`
- `"logarithm"`
- `"alright"`
- `"right"`
- `"math"`
- `"git"`

I oppgavene som følger ser vi på ulike måter å finne subanagrammer fra en gitt ordbok.

(a) 4 poeng

ALGORITHM: ER W ET SUBANAGRAM AV S?

Input: En streng W og en streng S

Output: Returner **true** hvis W et subanagram av S, **false** ellers

```
1 Procedure IsSubanagramOf1(W, S)
2     r ← copy of W
3     for c in S do
4         if c is in r then
5             remove an occurrence of c from r
6     return r.length = 0
```

Algoritmen ovenfor skal sjekke om ordet W kan skrives med bokstavene fra S.

- Anta at en streng fungerer akkurat som et array med bokstaver.
- Å kopiere et array gjøres i lineær tid.
- Å sjekke om et element er i et array gjøres i lineær tid.
- Å fjerne et element fra et array gjøres i lineær tid.
- Å finne lengden av et array gjøres i konstant tid.
- Vi lar w angi størrelsen på W og s angi størrelsen på S.

Hva er kjøretidskompleksiteten til algoritmen?

- $O(w + s)$
- $O(s^2)$
- $O(w \log(s))$
- $O(s \cdot w)$
- $O(w^2)$

(b) 7 poeng

Du skal nå implementere et alternativ til `IsSubanagramOf1`. Du skal ta utgangspunkt i en *frekvenstabell*. Anta at du har en prosedyre `FreqTable` tilgjengelig, som bygger et map fra bokstaver til antall forekomster i lineær tid.

Hvis F er en frekvenstabell for strengen `"abbcccdddd"` vil for eksempel F.`get("a")` returnere 1 og F.`get("d")` returnere 4. Du kan anta at F.`get`(x) returnerer 0 for alle bokstaver som ikke er med i ordet frekvenstabellen bygges fra. Du kan oppdatere verdien i frekvenstabellen ved å bruke F.`put`, som et vanlig map. Både `get` og `put` er i $O(1)$.

Fullfør implementasjonen av `IsSubanagramOf2`, slik at den har kjøretidskompleksitet $O(w + s)$, der w er størrelsen av W og s er størrelsen av S.

ALGORITHM: ER W ET SUBANAGRAM AV S?

Input: En streng W og en streng S
Output: Returner **true** hvis W et subanagram av S, **false** ellers

```
1 Procedure IsSubanagramOf2(W, S)
2 |  F ← FreqTable(W)
  |  // Fyll ut
```

Å bygge en frekvenstabell gjøres på lineær tid. En fin måte å løse denne oppgaven er å lage en frekvenstabell for begge strengene (som kan gjøres i $O(w + s)$) og sjekke at det for hver bokstav i W er minst like mange av den bokstaven i S:

```
1 Procedure IsSubanagramOf2(W, S)
2 |  F_w ← FreqTable(W)
3 |  F_s ← FreqTable(S)
4 |  for c ∈ W do
5 |  |  if F_w.get(c) > F_s.get(c) then
6 |  |  |  return false
7 |  return true
```

Det finnes mange andre fremgangsmåter her, der man for eksempel heller dekrementerer hver bokstav i S fra frekvenstabellen til W, og sjekker at summen ender opp med å bli 0 eller mindre. Første linje i prosedyren `IsSubanagramOf2` er veiledene, men den kan godt fjernes (hvis de for eksempel heller belager seg på en frekvenstabell for S).

Alle varianter av denne fremgangsmåten (altså benytter seg av en eller to frekvenstabeller på fornuftig vis) kan gi full uttelling dersom pseudokoden er lett forståelig og presis.

Vi trekker poeng for uklarheter og upresise svar. Dersom koden ikke kjører i $O(w + s)$ får man ingen poeng.

Veiledende poengfordeling:

- Minus 2 poeng for å bytte om < og > (altså sjekke om F_w.`get`$(c) < F_s$.`get`(c))
- Minus 2 poeng for å bytte om S og W (ett sted)
- Totalt 2 poeng for å sjekke om frekvensene er like/ulike, altså sjekke for anagram istedenfor subanagram

(c) 6 poeng

Strategi 1

```
1  Procedure SubanagramsOf1(D, S)
2      r ← empty list
3      for i ← 0 to d − 1 do
4          W ← D[i]
5          if IsSubanagramOf2(W, S) then
6              add W to r
7      return r
```

I pseudokoden ovenfor er vi gitt en ordbok `D` av lengde d, som er et array av strenger. Den bruker `IsSubanagramOf2` fra forrige deloppgave til å finne alle ord i ordboka som er et subanagram av S. Å finne subanagrammer med denne teknikken kjører i $O(d \cdot (w + s))$, der d er størrelsen på ordboka, w er størrelsen til det største ordet i ordboka, og s er størrelsen til S.

Strategi 2

```
1  Procedure Build(D)
2      M ← empty map
3      for i ← 0 to d − 1 do
4          sortedword ← Sort(D[i])
5          if M.get(sortedword) is empty then
6              M.put(sortedword, empty list)
7          add D[i] to M.get(sortedword)
8      return M
9  Procedure SubanagramsOf2(M, S)
10     r ← empty list
11     for x in sorted substrings of S do
12         append M.get(x) to r
13     return r
```

I pseudokoden ovenfor lager vi et hashmap, der hver nøkkel k er en streng med bokstaver sortert i alfabetisk rekkefølge, og verdien er en liste med anagrammer av k som finnes i ordboka. For å finne alle subanagrammer i ordboken, må vi slå opp på alle sorterte substrenger av S i hashmapet. Å finne subanagrammer med denne teknikken kjører i $O(2^s)$, altså eksponensiell tid med hensyn til størrelsen av S.

Drøft fordeler og ulemper mellom strategi 1 og strategi 2 med tanke på kjøretid.

Dette er en oppgave som går ut på å gjøre rimelige antagelser og anvende dem til å si noe fornuftig om hvilken fremgangsmåte man bør bruke. Dette er den oppgaven fra eksamenssettet hvor sensoren er nødt til å anvende mest skjønn i poengsettingen. Tanken bak oppgaven er å få studenten til å reflektere over hvilken fremgangsmåte man bør velge i lys av kjøretidskarakteristikkene til de to strategiene. Det bør legges vekt på at teksten er *godt skrevet* (dette betyr ikke at det skal trekkes poeng for enkle skrivefeil, men at teksten må være tydelig og enkel å forstå; en fin tommelfingerregel er at en setning man ikke forstår etter to gjennomlesninger ignoreres). Under lister vi noen refleksjoner vi ønsker å se.

En viktig observasjon er at `SubanagramsOf1` er avhengig av både d, w og s, men `SubanagramsOf2` kun er avhengig av s.

Man kan forvente at ord i en ordliste er forholdsvis korte, mens ordlisten selv er stor til sammenligning. Strengen S (som er bokstavene man har tilgjengelig til å forme ord fra) kan man også forvente er forholdsvis kort. Da vil d være det dominerende leddet for `SubanagramsOf1`, fordi at man for et enkelt oppslag er nødt til å søke gjennom hele ordlisten. En fordel med `SubanagramsOf1` er at kjøretiden påvirkes i liten grad av størrelsen på inputstrengen S; altså vil ikke kjøretiden påvirkes dramatisk dersom en bruker skriver inn en kort tekststreng S, til sammenligning med en lang tekststreng.

Prosedyren `SubanagramsOf2` er kun avhengig av størrelsen på S, men til gjengjeld er kjøretiden *eksponensiell* med hensyn til s. Vi vet at eksponensiell tid sjeldent er en god ting. Men det er viktig å huske på at S antageligvis er ganske liten, og ordboken er veldig stor til sammenligning. Dersom vi setter en fast størrelse for d og w (altså størrelsen på ordlista og det lengste ordet i ordlista), så er spørsmålet hvor stor s må bli før det lønner seg å gå gjennom hele ordboken. Dette besvares enklest eksperimentelt (og våre eksperimenter tilsier at for ordboken sowpods så går denne grensen rundt $s = 17$). Det vi uansett kan si sikkert er at kjøretiden til `SubanagramsOf2` vil vokse dramatisk, men dersom vi antar at størrelsen på S holder seg rimelig liten, så kan dette fremdeles være en god strategi.

For å kunne bruke `SubanagramsOf2` må vi først bygge en datastruktur som har litt verre kjøretid enn `SubanagramsOf1` (fordi vi sorterer ordene). Et fint pragmatisk argument er at dette vil kun gjøres en gang (når serveren starter opp), og under antagelsen om at ordene er korte, er denne kostnaden neglisjerbar.

Merk at dette er en drøftningsoppgave av en grunn: det finnes ikke et enkelt fasitsvar. Dersom man skal implementere dette vil man måtte avgjøre strategi eksperimentelt; dersom man ikke setter noen begrensninger på bokstavene brukeren har tilgjengelig S, så vil man kanskje ønske å benytte strategi 2 dersom $s < 17$ og strategi 1 ellers.

Veiledende poengfordeling:

- 3 poeng for å nevne at strategi 2 lønner seg hvis S er liten og D er stor
- 1 poeng for en velskreven teskt
- 1 poeng for å drøfte og argumentere for påstandene
- 1 poeng for å gjøre rimelige antagelser
- Minus 1 poeng for å ikke ha noen referanse til O-notasjonen (men snakker generelt om at «når ordboka er stor» og «ordene korte»).
- Minus 3 poeng for å snakke om n uten definisjon; her er jo det hele poenget

(d) 6 poeng

Nå som nettsiden vår har funksjonalitet for å finne subanagrammer av en streng S, gjenstår det bare å presentere resultatene for brukeren. Anta at subanagrammene som blir returnert av `SubanagramsOf2` er gitt som et array, sortert *alfabetisk*. Vi ønsker at subanagrammene skal vises sortert etter *lengde*, og at ord av samme lengde sorteres alfabetisk, slik som i følgende eksempel:

Input	Output
`"algorithm"`	`"algorithm"`
`"alright"`	`"logarithm"`
`"git"`	`"alright"`
`"logarithm"`	`"right"`
`"math"`	`"math"`
`"right"`	`"git"`

Fyll ut pseudokode nedenfor med så lav kjøretidskompleksitet som mulig.

ALGORITHM: SORTERING AV SUBANAGRAMMER

Input: Et array A av størrelse n som inneholder strenger

Output: Et array med de samme n strengene sortert etter lengde der ord av samme lengde er sortert alfabetisk

```
1 Procedure SortSubanagrams(A)
  | // Fyll ut
```

Dette lukter bucket sort!

```
1  Procedure SortSubanagrams(A)
2      B ← empty hashmap
3      for i ← 0 to n − 1 do
4          k ← A[i].length
5          if B[k] = null then
6              B[k] ← empty list
7          append A[i] to B[k]
8
9      j ← 0
10     for keys k of B in descending order do
11         for x in B[k] do
12             A[j] ← x
13             j ← j + 1
14     return A
```

Denne løsningen er en tilpasset bucket sort, der vi bruker et hashmap for å oppbevare bøttene. Dette fordi den lengste strengen i arrayet kan være lenger enn størrelsen av arrayet selv (selv om dette ikke er sannsynlig gitt konteksten). Siden vi traverserer nøklene i hashmapet i avtagende rekkefølge gir dette $O(N \log(N) + n)$ kjøretid, der N er antall nøkler. Merk at $N \leq n$ fordi vi ikke kan ha flere ulike lengder enn vi har strenger. Denne løsningen er *minst* like bra som en $n \log(n)$ algoritme, og i konteksten av strenger i en ordbok vil det være langt raskere enn en $n \log(n)$ algoritme.

- En god løsning med bucket sort gir full uttelling.
  - Vi trekker opptil 3 poeng dersom kategoriene er uspesifisert.
  - Vi trekker 1 poeng dersom strengene er sortert fra kortest til lengst.
  - Vi trekker ikke for bruk av arrays som bøtter.
- Vi gir opptil 4 poeng for en *stabil* sorteringsalgoritme i $O(n \log(n))$ med sortering på lengde.
- Vi gir opptil 2 poeng for en *stabil* sorteringsalgoritme i $O(n^2)$.
- Vi gir opptil 2 poeng for andre løsninger som vil fungere og er godt presentert.
  - Dette gjelder ikke dersom noen finner på en god løsning som vi ikke har tenkt på ennå.

Fire typer grafer poeng

Du ønsker å forbedre kommunikasjonsnettverket i en liten by. Historisk sett har all kommunikasjon mellom husstander i byen skjedd ved hjelp av brevduer. På denne måten kunne alle nå hverandres hus direkte. Men brevduer er upraktiske, så du skal finne bedre måter for husstandene å kommunisere med hverandre på. Du har utforsket følgende muligheter:

- Den nye startupen HamiltonBikes har begynt å bygge nye sykkelstier og kommer til å drifte brevlevering mellom alle hus. De garanterer at man kan ta seg en sykkeltur som går innom alle husene i byen, uten å måtte sykle forbi det samme huset to ganger.
- Husene kan forbindes med nettverkskabel, men med hensyn til kostnaden, vil redundante forbindelser unngås.
- Noen bor så nærme hverandre at naboer kan rope for rask én-til-én kommunikasjon.

Du innser at hver kommunkasjonsform kan formaliseres som en graf der nodene er husstandene og kantene er forbindelsene mellom disse. Du identifiserer fire typer grafer, som hver tilsvarer en kommunikasjonsform:

Type 1 er en urettet komplett graf (brevduene)
Type 2 er en urettet graf med en hamiltonsk sykel (sykkelstien)
Type 3 er et urettet tre (nettverkskabler)
Type 4 er en urettet ikke-sammenhengende graf (roping)

Alle grafer i denne oppgaven har mer enn to noder.

Graftypene er gjentatt der du trenger det, så du trenger ikke å huske dem.

(a) 7 poeng

Din første oppgave er å sjekke om HamiltonBikes virkelig leverer det de lover. De tilbyr å sende deg et kart over sykkelstiene og et forslag på en runde som går forbi hvert hus nøyaktig én gang. Du oversetter problemet til et avgjørelsesproblem:

Type2	
Instans:	En graf G
Spørsmål:	Inneholder G en hamiltonsk sykel?

Du bestemmer deg for å lage en effektiv algoritme som verifiserer **Type2**. Algoritmen tar en graf og et sertifikat som input, og svarer JA hvis sertifikatet bekrefter at grafen er av type 2, NEI ellers. Algoritmen skal kjøre i polynomisk tid.

HamiltonBikes leverer sertifikatet som et array C av noder i G, slik at hver node er med nøyaktig én gang. Skriv pseudokode for verifikatoren `Type2Verifier`.

ALGORITHM: VERIFIKATOR FOR **TYPE2**

Input: En graf G og et array av noder C
Output: Returner JA hvis C viser at G er av type 2, NEI ellers

```
1 Procedure Type2Verifier(G, C)
    // Fyll ut
```

ALGORITHM: VERIFIKATOR FOR **TYPE2**

Input: En graf G og et array av noder C
Output: Returner JA hvis C viser at G er av type 2, NEI ellers

```
1 Procedure Type2Verifier(G, C)
2     (V, E) ← G
3     N ← |C|
4     for i ← 0 to N − 1 do
5         if (C[i], C[(i + 1) mod N]) ∉ E then
6             return NEI
7     return JA
```

Det viktige i denne oppgaven er at man sjekker at etterfølgende par av noder i C svarer til kanter i G. Det er også nødvendig å sjekke at siste og første node i C utgjør en kant i G; dette gjør vi i siste iterasjon av loopen, der $i = N - 1$, hvor vi da sjekker om `(C[N - 1], C[0])` er en kant i G (fordi $N \bmod N = 0$).

Veiledende poengfordeling:

- 5 poeng for å sjekke om $e \in E(C) \Rightarrow e \in E(G)$; minus 1 poeng om de glemmer siste kant
- 2 poeng for signatur og generell oversiktlig besvarelse
- Ingen poeng for å finne en generell sykel, f.eks. ved hjelp av BFS/DFS

(b) 3 poeng

Forklar kort (maks 4 setninger) hvorfor svaret på forrige deloppgave viser at avgjørelsesproblemet **Type2** er i NP. Ikke gi en detaljert kjøretidsanalyse av pseudokoden din for `Type2Verifier`.

Dersom svaret ditt på forrige deloppgave har mangler, kan du gjøre nødvendige antagelser uten å få følgefeil.

> `Type2Verifier` kjører i polynomiell tid. Derfor kan JA-instanser av **Type2** verifiseres i polynomiell tid. Dette er kravet for å være i NP.

(c) 3 poeng

Type 1 er en urettet komplett graf (brevduene)
Type 2 er en urettet graf med en hamiltonsk sykel (sykkelstien)
Type 3 er et urettet tre (nettverkskabler)
Type 4 er en urettet ikke-sammenhengende graf (roping)

Alle grafer i denne oppgaven har mer enn to noder.

En fordel med sykkelstiene til HamiltonBikes er redundans: hvis for eksempel alle sykkelstiene rundt et hus må stenges på grunn av byggearbeid, så er det fortsatt mulig å sykle rundt i byen.

Forklar kort (maks 4 setninger) hvorfor grafer av type 2 er 2-sammenhengende.

Hvis grafen inneholder en hamiltonsk sykel kan alle noder nå alle andre på minst to måter. Dette er ved å følge sykelen «den ene eller den andre» veien. Her bruker vi at grafen er urettet.

(d) 8 poeng

Type 1 er en urettet komplett graf (brevduene)
Type 2 er en urettet graf med en hamiltonsk sykel (sykkelstien)
Type 3 er et urettet tre (nettverkskabler)
Type 4 er en urettet ikke-sammenhengende graf (roping)

Alle grafer i denne oppgaven har mer enn to noder.

Bystyret skjønner ikke grafteori, så de vil gjerne at du viser hvordan man skulle kunne komme seg rundt i byen dersom det gjøres byggearbeid. Du bestemmer deg for å lage et program som skriver ut to distinkte stier fra et gitt hus til et annet.

Du tar altså utgangspunkt i en graf $G = (V, E)$ som er av type 2, og to noder s og t i grafen.

Skriv pseudokode for en prosedyre `TwoPaths`, som skriver ut to distinkte stier fra s til t.

Du er gitt et sertifikat C, som beskrevet i deloppgave (a).

Ikke tenk på formatet på utskriften; det viktige er at nodene skrives ut i riktig rekkefølge.

ALGORITHM: SKRIV UT TO DISTINKTE STIER FRA s TIL t

Input: En graf G, to noder s og t og et sertifikat C
Output: Skriver ut to distinkte stier fra s til t

```
1 Procedure TwoPaths(G, C, s, t)
  | // Fyll ut
```

Siden C er et sertifikat for at G inneholder en hamiltonsk sykel, så utgjør C en sykel som inneholder alle noder i G. Det vil si at vi har en sykel som nødvendigvis inneholder både s og t. Alt vi trenger å gjøre er å finne s og så følge sykelen i begge retninger til vi finner t, hvor hver retning svarer til en distinkt sti mellom s og t.

```
1  Procedure TwoPaths(G, C, s, t)
2  |  N ← |C|
3  |  si ← 0
4  |  while C[si] ≠ s do
5  |  |  si ← si + 1
6  |  i ← si
7  |  while C[i] ≠ t do
8  |  |  print(C[i])
9  |  |  i ← (i + 1) mod N
10 |  print(C[i])
11 |  i ← si
12 |  while C[i] ≠ t do
13 |  |  print(C[i])
14 |  |  i ← (i − 1) mod N
15 |  print(C[i])
```

Veiledene poengfordeling:

- 2 poeng for sti en vei
- 2 poeng for sti motsatt vei
- 2 poeng for å huske modulo-operator eller lignende for å få til syklisk iterasjon
- 2 poeng for riktig oppsett (signatur/ryddig svar)
- Ingen poeng for å finne alle stier i G ved hjelp av DFS
- 1-2 poeng hvis de finner 2 mulige distinkte stier og stopper etter det.

(e) 3 poeng

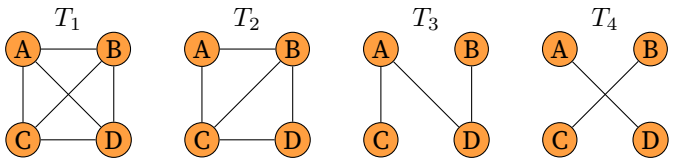
Type 1 er en urettet komplett graf (brevduene)
Type 2 er en urettet graf med en hamiltonsk sykel (sykkelstien)
Type 3 er et urettet tre (nettverkskabler)
Type 4 er en urettet ikke-sammenhengende graf (roping)

Alle grafer i denne oppgaven har mer enn to noder.

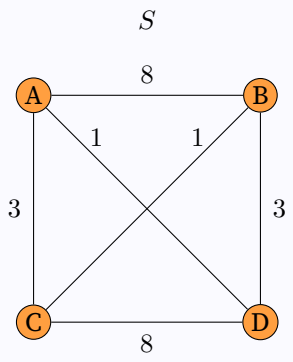
Vi har lyst til å få oversikt over alle kommunikasjonsmåter i byen. Derfor gir vi forskjellig vekt til de ulike kanttypene, basert på kostnad. Deretter slår vi disse sammen til en graf S. Der det er flere kanter beholder vi bare den med lavest vekt.

Her er fire eksempelgrafer, en av hver type, hvor

- kanter i T_1 har vekt 10
- kanter i T_2 har vekt 8
- kanter i T_3 har vekt 3
- kanter i T_4 har vekt 1



Fyll ut kantvektene i grafen S.



(f) 4 poeng

Type 1 er en urettet komplett graf (brevduene)
Type 2 er en urettet graf med en hamiltonsk sykel (sykkelstien)
Type 3 er et urettet tre (nettverkskabler)
Type 4 er en urettet ikke-sammenhengende graf (roping)

Alle grafer i denne oppgaven har mer enn to noder.

Du konstruerer den sammenslåtte grafen G på samme måte som i forrige deloppgave utifra de følgende grafene:

- G_1 av type 1, hvor alle kanter har vekt 10,
- G_2 av type 2, hvor alle kanter har vekt 8,
- G_3 av type 3, hvor alle kanter har vekt 3,
- G_4 av type 4, hvor alle kanter har vekt 1.

Nå har du lyst til å finne den billigste måten å forbinde husene i byen. Derfor har du bestemt deg for å bruke Kruskals algoritme på den sammenslåtte grafen G.

Besvar spørsmålene nedenfor, og begrunn svaret *kort* (maks 4 setninger).

- Finnes det nøyaktig én billigste måte å forbinde husene i byen?
- Er det mulig for alle i byen til å kommunisere bare ved bruk av roping? Med andre ord, kunne vi ha bygget et spenntre for G, med kun kanter fra G_4?
- Din kollega sier at hvis G_4 ikke har noen kanter, så er det minimale spenntreet for G nøyaktig G_3. Stemmer dette?
- Hva er vekten på den *siste* kanten Kruskals algoritme velger når den bygger spenntreet?

- Nei, det er ikke garantert fordi flere kantar kan ha samme vekt. Et unikt minimalt spenntre er bare garantert når ingen kanter har samme vekt.
- Nei, fordi G_4 ikke er sammenhengende. Poenget med spenntreet er at det skal spenne ut grafen, vi må derfor bruke andre kanter enn kun de fra G_4.
- Ja. Dette er fordi G_3 er av type 3 og derfor er et tre. Hvis G_4 ikke inneholder noen kanter, vil alle de billigste kantene være i G_3 så spenntreet er derfor også minimalt.
- 3. Kruskal terminerer når siste kant er valgt, og det er den billigste kanten som forbinder de to siste spenntrærne (T_1 og T_2). Denne kanten kan ikke være i G_4 som forklart i punkt 2 over. Siden G_3 er et spenntre, som forklart i punkt 3, må det finnes en kant som forbinder enhver ikke-triviell todeling av nodene. Dette er fordi hvis det var mulig å lage T_1 og T_2 uten at det fantes en slik kant i G_3 så hadde ikke G_3 vært et spenntre.

Kun halv uttelling hvis studenten forklarer ut fra grafene fra forrige deloppgave istedenfor for en generell graf. Manglende begrunnelse gir ikke uttelling.

(g) 2 poeng

Type 1 er en urettet komplett graf (brevduene)
Type 2 er en urettet graf med en hamiltonsk sykel (sykkelstien)
Type 3 er et urettet tre (nettverkskabler)
Type 4 er en urettet ikke-sammenhengende graf (roping)

Alle grafer i denne oppgaven har mer enn to noder.

La $G = (V, E)$ være en graf og la s og t være noder i V. Vi skal nå finne lengden til den korteste stien fra s til t.

- Hvilken algoritme løser problemet mest effektivt hvis du vet at G er av type 1 og alle kantene har vekt 10?
- Hvilken algoritme løser problemet mest effektivt hvis du vet at G er av type 2 og alle kantene har vekt 8?

- Hvilken algoritme løser problemet mest effektivt hvis du vet at G er av type 1 og alle kantene har vekt 10?
  - DFS
  - BFS
  - Dijkstra
  - Prim
  - Bellman Ford
  - Topologisk sortering
  - En algoritme som bare returnerer 10
- Hvilken algoritme løser problemet mest effektivt hvis du vet at G er av type 2 og alle kantene har vekt 8?
  - DFS
  - BFS
  - Dijkstra
  - Prim
  - Bellman Ford
  - Topologisk sortering
  - En algoritme som bare returnerer 8

Merk at det er en liten svakhet i oppgaven her, hvor vi burde spesifisert at $s \neq t$. Det er allikevel ikke direkte feil, fordi en sti ikke kan inneholde samme node flere ganger; altså er det å gå fra s til s ikke en sti.

(h) 8 poeng

Nå skal du finne den korteste kommunikasjonsveien fra posten (A) til alle hus i byen, formalisert som noder i en mengde V. Du har representert alle kommunikasjonstyper i de følgende kantmengdene:

- E_1 inneholder en kant mellom hvert par av noder i V.
  Alle kanter i E_1 har vekt 10.
- $E_2 = \{\{A, B\}, \{B, C\}, \{C, D\}, \{D, E\}, \{E, F\}, \{F, G\}, \{G, H\}, \{E, H\}, \{F, H\}, \{F, A\}\}$
  Alle kanter i E_2 har vekt 8.
- $E_3 = \{\{A, C\}, \{B, C\}, \{C, D\}, \{D, E\}, \{E, F\}, \{F, G\}, \{G, H\}\}$
  Alle kanter i E_3 har vekt 3.
- $E_4 = \{\{A, D\}, \{C, D\}, \{B, D\}, \{F, G\}\}$
  Alle kanter i E_4 har vekt 1.
- $V = \{A, B, C, D, E, F, G, H\}$

Finn de korteste stiene fra A til alle andre noder. Du kan bruke kanter fra alle fire kantmengdene. Husk at det kan være flere kommunikasjonsveier mellom hus, altså at det er parallelle kanter mellom nodene.

Fyll ut tabellen med korrekt minimalavstand fra A for hver node i V.

A	0
B	2
C	2
D	1
E	4
F	7
G	8
H	10

HTML poeng

Du har tatt på deg en jobb hos en bedrift som har en eldre nettside skrevet i ren HTML. Nylig har de fått nyss om at nettsiden fungerer svært dårlig, og de (som har lite intern kompetanse på slikt) har gitt opp og håper at den unge lovende informatikeren skal finne ut av hva i all verden som foregår. Nettsiden består av flere tusen håndskrevne HTML-sider; all snakk om å «lage noe nytt og fresht» blir møtt med spørrende blikk.

Det tar ikke spesielt lang tid før du finner en feil som gjentar seg i mange av sidene det er rapportert feil på. Mange av de håndskrevne HTML-filene er feilformaterte! Du merker deg at det mange steder åpnes en `<div>`-tag, som aldri lukkes med en `</div>`. Andre steder lukkes det en `</div>`-tag, der den aldri har blitt åpnet. Her er et eksempel på en feilformatert HTML-fil:

```
<div>
  <div>
    Velkommen til nettsiden vår!
  <div>
   Se våre nyeste saker!
    <div>
    Vi har ansatt en ung person! Kanskje blir nettsiden oppdatert nå?
    FØLG MED PÅ UTVIKLINGEN. Hilsen Kåre.
    </div>
</div>
```

(a) 4 poeng

Du innser at det første du må gjøre er å finne alle filene som inneholder feil. Etter litt Unix-magi sitter du igjen med filer der all tekst bortsett fra `<div>` og `</div>` er fjernet. Alt går på skinner frem til du må løse en liten nøtt:

> *Hvordan sjekker man at det for hver tag som åpnes med `<div>` også lukkes med `</div>`, og at ingen tag lukkes før den er åpnet?*

Du må skrive en prosedyre `GoodDivs` som sjekker dette, med andre ord, at `<div>`-tagene er velformaterte. Den tar et array `A` som argument, der hvert element enten er strengen `"<div>"` eller strengen `"</div>"`.

ALGORITHM: AVGJØRE OM `<div>`-TAGENE ER VELFORMATERTE

Input: Et array med `"<div>"` og `"</div>"` av lengde n
Output: Returner **true** hvis tag-ene er velformaterte, **false** ellers

```
1 Procedure GoodDivs(A)
  | // Fyll ut
```

Dette er en forholdsvis enkel oppgave, som først og fremst var ment som en oppvarming til oppgave (b). Merk at et (riktig) svar på (b) også egentlig er et riktig svar på denne oppgaven.

Den enkleste måten å gå frem på her er å løpe gjennom alle tagene og øke en teller ved `"<div>"` og senkes ved `"</div>"`. Dersom telleren på noe tidspunkt blir negativ kan vi returnere **false**. Etter alle tag-ene er løpt gjennom må vi sjekke at telleren er 0.

```
1  Procedure GoodDivs(A)
2  |  i ← 0
3  |  for tag in A do
4  |  |  if tag = "<div>" then
5  |  |  |  i ← i + 1
6  |  |  else
7  |  |  |  i ← i − 1
8  |  |  if i < 0 then
9  |  |  |  return false
10 |  return i = 0
```

Veiledende poengfordeling:

- Totalt 1 poeng for løsninger som sjekker direkte at antall == antall
- Totalt 1 poeng for løsninger som sjekker at alle etterfølges av en direkte
- Minus 2 poeng for å glemme å sjekke om antall noen gang overstiger antall
- Minus 1 poeng for å glemme å sjekke om vi ender opp med likt til slutt

(b) 8 poeng

`GoodDivs` fungerer som den skal og finner mange filer som inneholder feilformatert HTML. Dessverre oppdager den ikke alle filene det er rapportert feil på. Du observerer at de inneholder samme feil, men med andre type tag-er! Altså er det ikke bare `<div>`-tagene som fører til problemer, men også andre tag-er som `<head>`, `<body>`, `<p>` og flere andre. Du finner ut at prosedyren `GoodDivs` må generaliseres til en prosedyre `GoodTags`.

> *Hvordan sjekker man at det for hver tag som åpnes også lukkes med en tag av samme type, og at ingen tag lukkes før den er åpnet?*

Du har allerede en enkel prosedyre `isOpen`, som tar en streng og returnerer **`true`** hvis det er en åpne-tag og **`false`** hvis det er en lukke-tag. Altså vil `isOpen("<tag>")` returnere **`true`** og `isOpen("</tag>")` returnere **`false`**. I tillegg har du en prosedyre som gir deg hvilken type en tag har; for eksempel vil `TagType("<div>")` gi `"div"`, og `TagType("</p>")` gir `"p"`.

ALGORITHM: AVGJØRE OM TAG-ENE ER VELFORMATERTE

Input: Et array `A` med tag-er av lengde n
Output: Returner **`true`** hvis tag-ene er velformaterte, **`false`** ellers

```
1 Procedure GoodTags(A)
  |  // Fyll ut
```

Hint: Her kan et lurt valg av datastruktur være til stor hjelp.

Oppgaveteksten åpner for at `["<div>", "<p>", "</div>", "</p>"]` teller som velformatert. Dette var ikke intensjonen. Løsninger som teller dette som velformatert vil også kunne gi full uttelling. Det kan enkelt gjøres ved å ha et map fra tag-type til en teller, som brukes på samme måte som forrige deloppgave.

Løsningen som følger intensjonen (som faktisk vil validere balanserte HTML-filer) kan se slik ut:

```
1 Procedure GoodDivs(A)
2   s ← empty stack
3   for tag ∈ A do
4     if isOpen(tag) then
5       s.push(tag)
6       continue
7     if s is empty or TagType(s.pop()) ≠ TagType(tag) then
8       return false
9   return s is empty
```

Her vektes poeng og trekk vanligvis på samme måte som i (a).