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.