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.

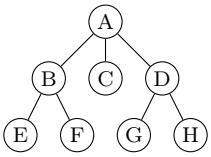
Oppvarming **2 poeng**

- Hva er en algoritme? Svar kort (maks fire setninger).
- 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.

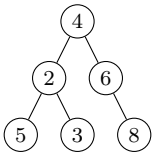
Binære søketrær — 4 poeng

(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

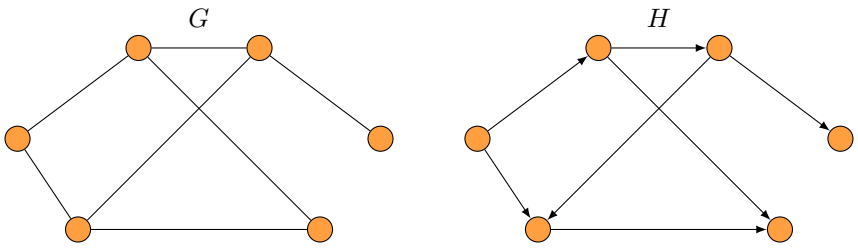
Er hvert binært søketre et AVL tre?

(d) — 1 poeng

Er hvert AVL tre et binært søketre?

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?

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 $h(k, N) = k \texttt{ mod } N$, som for dette eksempelet blir det samme som $h(k) = k \texttt{ mod } 10$. 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.

Søk **12 poeng**

Anta at vi har tre prosedyrer:

- `Search` – som gjør et rett frem søk på lineær tid
- `BinarySearch` – som gjør et binærsøk
- `HeapSort` – sorterer med heapsort

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)
```

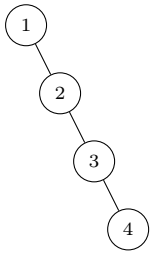
```
1 Procedure Intersection3(A, B)
2 |   HeapSort(A)
3 |   for x in B do
4 |   |   if BinarySearch(A, x) then
5 |   |   |   print(x)
```

```
1 Procedure Intersection4(A, B)
2 |   for x in B do
3 |   |   HeapSort(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 avhengig av hvor stor k (altså lengden av B) er i forhold til n (altså lengden til A).

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?

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 skal vi se på ulike måter å finne subanagrammer fra en gitt ordbok.

(a) 4 poeng

Algorithm 1: 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?

(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 2: 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
```

(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.

(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 `Subanagrams0f2` 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 nedfor med så lav kjøretidskompleksitet som mulig.

Algorithm 3: 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
```

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 4: 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
```

(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.

(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.

(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 5: 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
```

(e) 3 poeng

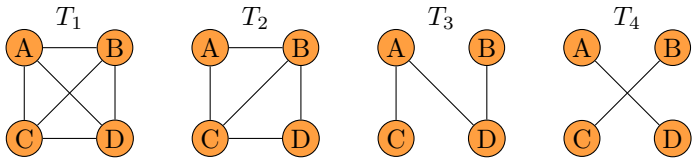
Type 1 er en urettet komplett graf (brevduene)
Type 2 er en urettet graf med en hamiltonsk sykel (sykkelstien)
Type 3 er et urettet tre (nettverkskabler)
Type 4 er en urettet ikke-sammenhengende graf (roping)

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?

(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?

(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.

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 6: 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
```

(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 7: 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.