Innlevering 5 (obligatorisk)

Høst 2024

Om innleveringen

- Det er tillatt — og du oppfordres til — å *levere i grupper på inntil tre personer*, under forutsetning av at alle i gruppen både står for det som leveres og alene kan gjøre rede for alle delene av arbeidet. Grupper skal opprettes i Devilry.
- Det er mulig å levere som en gruppe på den obligatoriske innlevering, selv om de tidligere innleveringsoppgavene er løst individuelt.
- Innleveringen skal leveres i innleverings- og tilbakemeldingssystemet Devilry.
- Innleveringen består av fire oppgaver (og dette dokumentet er på **??** sider).
- For hver oppgave skal innleveringen inneholde:
  - Et Java- eller Python-program for hver oppgave som ber om det.
    - Du kan dele et program i flere filer, men du må sørge for at programmet kan kjøres med én kommando.
  - Én `.pdf`-, `.md`- eller `.txt`-fil som skal inneholde kommandoen som brukes for å kjøre programmet, samt løsninger på oppgaver som skal besvares med pseudokode eller naturlig språk.
- Du skal ikke levere testfiler eller andre ressurser.
- Lenker til ressurssidene finner du på innleveringssiden.[a]
- Med mindre noe annet er spesifisert kan dere bruke hva dere vil fra Java eller Python sine standardbibliotek.
- Kommenter **«Klar for retting»** i Devilry hvis du leverer i god tid før fristen, og innleveringen er klar for retting.

[a] https://www.uio.no/studier/emner/matnat/ifi/IN2010/h24/innleveringer/

Krav til innleveringen

- Alle oppgaver skal være besvart.
- Programmene må kompilere og kunne kjøres med kommandoen dere har oppgitt.
- Det som leveres skal være *lett forståelig, entydig og presist*.
- Det gis inntil *ett* nytt forsøk. For å kunne få et nytt forsøk:
  - Må du ha levert et seriøst forsøk.
  - Må du ha *forsøkt* å løse alle oppgavene.
  - Må du redegjøre for mangler i løsningene din.

Oppgave 1 — Binærsøk med lenkede lister

Råd fra retterne

- Husk å forklare hvordan du kom frem til svaret ditt.
- Tenk først på kjøretidskompleksiteten til binærsøk.
- Tenk så på kjøretidskompleksiteten til oppslag i lenkede lister.

Gi en verste tilfelle kjøretidsanalyse av algoritmen nedenfor, som implementerer binærsøk over *lenkede lister*. Hvordan påvirker valget av datastruktur kjøretidskompleksiteten i dette tilfellet?

ALGORITHM: BINÆRSØK MED LENKEDE LISTER

Input: En ordnet lenket liste A og et element x
Output: Hvis x er i listen A, returner **true** ellers **false**

```
1  Procedure BinarySearch(A, x)
2      low ← 0
3      high ← |A| − 1
4      while low ≤ high do
5          i ← ⌊(low+high)/2⌋
6          if A.get(i) = x then
7              return true
8          else if A.get(i) < x then
9              low ← i + 1
10         else if A.get(i) > x then
11             high ← i − 1
12     return false
```

Forslag til filnavn:

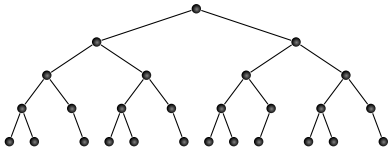
- `oppgave1.{pdf|md|txt}`

Oppgave 2 — Bygge balanserte søketrær

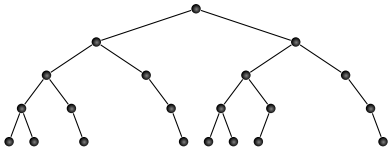
Råd fra retterne

- Pass på at pseudokoden formidler den samme idéen som koden.
- Tenk på følgende spørsmål:
  - Hvilket element skal ligge i roten av treet?
  - Hvordan finner jeg medianverdien fra en prioritetskø?

I denne oppgaven ønsker vi å bygge et *helt balansert binært søketre*. Vi definerer dette som et binært søketre hvor det er 2^d noder med dybde d, der $0 \leq d < h$ og h er høyden på treet[1]. En annen måte å si det samme på er at den korteste og den lengste stien fra roten til et tomt subtre (for eksempel representert med en `null`-peker) har en lengdeforskjell på 0 eller 1.



(a) Et helt balansert binærtre



(b) Et ikke helt balansert binærtre

Du trenger ikke implementere et binært søketre. Alt du trenger å gjøre er å printe ut elementene du får som input i en rekkefølge som garanterer at vi får et balansert tre dersom vi legger elementene inn i binærtreet ved bruk av vanlig innsetting. Dette binære søketreet er *ikke selvbalanserende*. Input består av heltall i sortert rekkeføge, der ingen tall forekommer to ganger (altså trenger du ikke ta høyde for duplikater).

Eksempel-input	Eksempel-output
0	5
1	8
2	10
3	9
4	7
5	6
6	2
7	4
8	3
9	1
10	0

(a) Du har fått et *sortert array* med heltall som input. Lag en algoritme som skriver ut elementene i en rekkefølge, slik at hvis de blir plassert i et binært søketre i den rekkefølgen så resulterer dette i et *balansert* søketre.

- Skriv pseudokode for algoritmen du kommer frem til.
- Skriv et Java eller Python-program som implementerer algoritmen din. Det skal lese tallene fra `stdin` og skrive dem ut som beskrevet ovenfor.

(b) Nå skal du løse det samme problemet, men denne gangen kun ved bruk av *prioritetskø*. Altså kan ikke algoritmen din bruke andre datatyper enn prioritetskøer, men til gjengjeld kan du bruke så mange prioritetskøer du vil!

- Skriv pseudokode for algoritmen du kommer frem til. Her kan du anta at elementene allerede er plassert på en prioritetskø, og at input kun består av en prioritetskø med heltall.

[1] Merk at dette er veldig likt definisjonen av et *komplett* binærtre, som forklares i forelesningen om prioritetskøer, men uten kravet om at noder med dybde h er plassert så langt til venstre som mulig.

- Skriv et Java eller Python-program som implementerer algoritmen din. Programmet må først plassere elementene som leses inn på en prioritetskø, og derretter kalle på implementasjonen av algoritmen du har kommet frem til.

  For Java kan du bruke `PriorityQueue`[2]. De eneste operasjonene du trenger å bruke fra Java sin `PriorityQueue` er: `size()`, `offer()` og `poll()`. Merk at `offer()` svarer til `push()`, og `poll()` svarer til `pop()`.

  For Python kan du bruke `heapq`[3]. De eneste operasjonene du trenger er: `heappush()` og `heappop()`, samt kalle `len()` for å få størrelsen på prioritetskøen.

Forslag til filnavn:

- `oppgave2.{pdf|md|txt}`
- `oppgave2a.py` eller `Oppgave2a.java`
- `oppgave2b.py` eller `Oppgave2b.java`

[2] https://docs.oracle.com/javase/8/docs/api/java/util/PriorityQueue.html
[3] https://docs.python.org/3/library/heapq.html

Oppgave 3 — Sortering

Råd fra retterne

- Dette får du til!

I denne oppgaven skal du implementere en sorteringsalgoritme!

Du kan velge mellom:

- Merge sort
- Quicksort
- Heapsort[4]

Input og output

Input består av n heltall. For hvert tall i er det slik at $0 \leq i \leq 2^{31} - 1$. Output skal være de samme n tallene i sortert rekkefølge.

Eksempel-input	Eksempel-output
80	7
91	12
7	13
33	33
50	50
70	70
13	80
321	91
12	321

Input skal leses fra fil, hvor filnavnet blir gitt som kommandolinjeargument. Output skal skrives til en fil med samme navn, avsluttet med `out`. Hvis inputfilen eksempelvis heter `input` skal outputfilen hete `input_out`.

Forslag til filnavn:

- `oppgave3.py` eller `Oppgave3.java`

[4] Merk at Heapsort ikke er gått gjennom i detalj på forelesning ennå. Se kapittel 5, seksjon 4 i Goodrich & Tamassia.

Oppgave 4 — Six degrees of IMDB

Råd fra retterne

- Du kan konstruere grafen i flere steg.
- Husk at du ofte kan spare tid ved å lagre mer.
- `HashMap` og `HashSet` i Java, og `dict` og `set` i Python er fine ting!

Datasettet

> Datasettet vi har generert skal kun brukes for læring, og kan ikke benyttes for andre formål. Du kan lese mer om bruk av IMDB sine datasett her.

Vi skal bygge en ganske stor graf basert på data fra IMDB. Den får rundt hundre tusen noder og fem millioner kanter! På den kan vi utføre mange forskjellige grafalgoritmer. Fordi grafen er stor, vil du fort få en følelse for effektiviteten av algoritmene dine.

I grafen vi skal bygge, er hver node en skuespiller, og to skuespillere har en kant mellom seg for hver film de har spilt sammen i. Kantene er merket med en film som har en rating. Dermed får vi en *urettet* graf med *merkede*, *parallelle* og *vektede* kanter.

Alternativt kan vi se på grafen som en *urettet* og *vektet* graf med *to forskjellige nodetyper*, en type for skuespillere og en type for filmer. I en slik graf vil hver skuespiller kun være koblet til filmer, og hver film er kun koblet til skuespillere.

Vi har generert to TSV-filer (publisert på ressurssiden) som sammen utgjør grafen:

`movies.tsv` Hver linje består av fire felter, separert med `\t`:

`tt-id` Tittel Rating Antall Stemmer

> Hver film er unikt identifisert ved en `tt`-id, der hver `tt`-id er en streng som begynner med `tt` etterfulgt av 7 eller 8 siffer. Filmen har en tittel og en rating (et tall mellom 0.0 og 10.0) og et antall stemmer. Ingen av oppgavene vil benytte seg av antall stemmer, så det feltet kan ignoreres.

`actors.tsv` Hver linje består av

`nm-id` Navn $\texttt{tt-id}_1$ $\texttt{tt-id}_2$ ... $\texttt{tt-id}_k$

> Hver skuespiller er unikt identifisert ved en `nm`-id, der hver `nm`-id er en streng som begynner med `nm` etterfulgt av 7 eller 8 siffer, og et navn. De k siste kolonnene på hver linje er filemene skuespilleren har spilt i, gitt som `tt`-id-er.

Bygg grafen

For å kunne implementere grafalgoritmer trenger vi først en graf å jobbe med. Her må vi gjøre et valg om hvordan vi vil *representere* grafen. Du står fritt til å velge den representasjonen du tenker vil være enklest å jobbe med. Det er ingen restriksjoner på hva du kan bruke fra Java eller Pythons standardbibliotek.

Merk at en skuespiller kan ha spilt i en film som vi ikke har data om. Det vil si at en skuespiller kan ha en `tt`-id som ikke forekommer i `movies.tsv`. Disse `tt`-id-ene skal ignoreres.

Deloppgaver:

1. Skriv et Java eller Python-program som leser inputfilene (`movies.tsv` og `actors.tsv`) og bygger grafen. For testing kan det være lurt å starte med de små inputfilene (`marvel_movies.tsv` og `marvel_actors.tsv`).
2. For å sjekke om resultatet er rimelig skal du nå kunne skrive en prosedyre som teller antall noder og antall kanter. Skriv ut resultatet, som kan se slik ut:

```
Oppgave 1

Nodes: 126196
Edges: 5342530
```

Det kan finnes andre representasjoner av grafen som egner seg for å løse resten av oppgavene, som gjør at dere får andre tall, og det er helt greit.

Six Degrees of IMDB

Basert på konseptet six degrees of separation har det blitt laget noen veldig kule applikasjoner, som for eksempel Six Degrees of Wikipedia. Vi skal nå gjøre noe lignende for IMDB (inspirert av Six Degress of Kevin Bacon).

Skriv en prosedyre som, gitt to skuespillere, finner en korteste sti som forbinder dem. Programmet skal kunne skrive ut stien, og inneholde både nodene (skuespillerene) og kantene som forbinder dem (se eksempelutskrift nedenfor).

Skriv ut korteste stier for følgende skuespillere (gitt som nm-id-er):

nm-id_1	nm-id_2	Navn_1	Navn_2
nm2255973	nm0000460	Donald Glover	Jeremy Irons
nm0424060	nm8076281	Scarlett Johansson	Emma Mackey
nm4689420	nm0000365	Carrie Coon	Julie Delpy
nm0000288	nm2143282	Christian Bale	Lupita Nyong'o
nm0637259	nm0931324	Tuva Novotny	Michael K. Williams

Skriv ut resultatet på søkene, som bør se slik ut:

```
Oppgave 2

Donald Glover
===[ LA Originals (7.2) ] ===>  Terry Crews
===[ Inland Empire (6.8) ] ===>  Jeremy Irons

Scarlett Johansson
===[ Rough Night (5.2) ] ===>  Kate McKinnon
===[ Barbie (7.1) ] ===>  Emma Mackey

Carrie Coon
===[ His Three Daughters (8.5) ] ===>  Natasha Lyonne
===[ But I'm a Cheerleader (6.8) ] ===>  Julie Delpy

Christian Bale
===[ The Promise (6.1) ] ===>  Oscar Isaac
===[ Star Wars Episode IX: The Rise of Skywalker (6.4) ] ===>  Lupita Nyong'o

Tuva Novotny
===[ Dear Alice (5.5) ] ===>  Danny Glover
===[ LUV (5.9) ] ===>  Michael K. Williams
```

Merk at utskriften ikke trenger å se nøyaktig slik ut, men det er viktig at det er lett å lese ut hvilke noder og kanter stien går gjennom. Det finnes ofte flere stier av samme lengde; det er ingen krav til hvilken sti som finnes, så lenge det ikke finnes en kortere.

Forslag til filnavn:

- `oppgave4.py` eller `Oppgave4.java`