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.

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.

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?

(b) Hva inneholder løvnodene i det resulterende treet?

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.

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)
  | // ···
```

(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.

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 $\{$`"jazz"`, `"pop"`, `"folk"`$\}$.

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

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)
  | // ···
```

(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?

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.

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

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

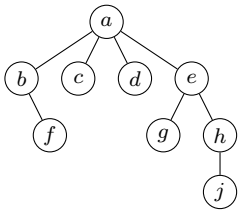
(d) Hva er fordelene med den nye strukturen?

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.

(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)
  | // ···
```

(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)
  | // ···
```

(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.

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.

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

(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.

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

(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.

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.

(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.