UNIVERSITETET I OSLO

Det matematisk-naturvitenskapelige fakultet

Eksamen i	INF2220 — Algoritmer og datastrukturer
Eksamensdag:	12. desember 2014
Tid for eksamen:	14.30 – 18.30
Oppgavesettet er på 8 sider.	
Vedlegg:	Ingen
Tillatte hjelpemidler:	Alle trykte eller skrevne

Kontroller at oppgavesettet er komplett før
du begynner å besvare spørsmålene.

Innhold

Generelle **råd**:

- Skriv **leselig** (uleselige svar er alltid feil . . .)
- Husk å **begrunne** svar og hold kommentarene dine korte og presise.
- Hvis du bruker kjente datastrukturer som (list, set, map, binær-tre) trenger du ikke forklare hvordan de fungerer. Generelt: hvis du bruker abstrakte datatyper fra biblioteket kan du bruke disse uten å forklare hva de gjør.
- Hvis skriver pseudokode, må den være detaljert. Det må komme klart fram hvilken datastruktur du bruker, hva initialiseringen består av og hva hovedløkkene gjør.
- Du skal bruke Java.
- **Vekten** til en oppgave indikerer vanskelighetsgraden og tidsbruken du bør bruke på den oppgaven. Dette kan være greit å benytte for å disponere tiden best mulig.

Lykke til!
Ingrid Chieh Yu, Dino Karabeg og Arne Maus

(Fortsettes på side 2.)

Oppgave 1 Tidskompleksitet (vekt 10%)

Hva er *worst case*-tidskompleksiteten (big O) til følgende kodesegmenter:

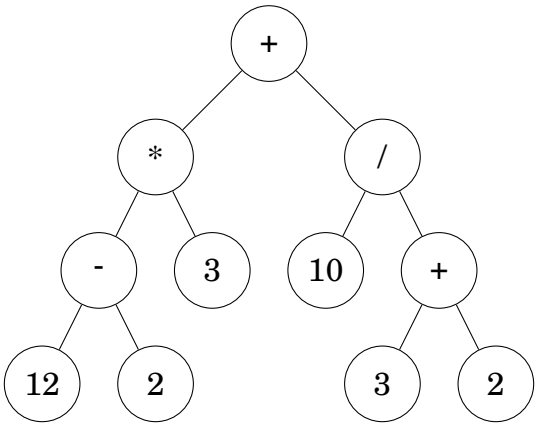
1a (vekt 5%)

```
int x = 0;
for (int i = n; i >= 1; i = i/2){
    for (int j = 1; j <= i; j++){
       x = x + i + j;
    }
}
```

1b (vekt 5%)

```
int x = 0;
for (int j = 1; j <= n; j++){
    for (int i = n; i >= j; i = i/2){
        x = x + i + j;
    }
}
```

Oppgave 2 Uttrykkstrær og rekursjon (vekt 11%)



Figur 1

Figur 1 viser et *uttrykkstre*. Et uttyksetre er en entydig måte å representere et regnestykke, der verdien til en node tilsvarer enten resultatet av operanden den innholder utført på nodens to barn eller verdien til tallet i noden.

2a Evaluering av uttrykkstreet (vekt 2%)

Evaluer uttrykkstreet i Fig. 1, og skriv resultatet.

(Fortsettes på side 3.)

2b Rekursjon (vekt 7%)

Implementer en rekursiv metode med signaturen `public void evalTree (BinTree t)` som skal evaluere et uttrykstre. Programmet skal kombinere klassene `MyStack` og `BinTree` med *postorder traversering* av trær. Du kan anta at klassen metoden ligger i har initialisert en instanse av klassen `MyStack` med tilstrekkelig størrelse med navn `stack`.

```
public class MyStack {
   private int maxSize;
   private int[] stackArray;
   private int top;
   public MyStack(int s) {
      maxSize = s;
      stackArray = new int[maxSize];
      top = -1;
   }
   public void push(int j) {
      stackArray[++top] = j;
   }
   public long pop() {
      return stackArray[top--];
   }
}

class BinTree {
   String value;
   BinTree left;
   BinTree right;
}
```

En stack er en implementasjon av konseptet *LIFO datahåndtering*. Se for deg en stabel med tallerkener der du enten setter en tallerken på toppen av stabelen med *push* eller henter en tallerken fra toppen av stabelen med *pop*. Verdiene som skal lagres i treet er representert med strenger og inneholder enten en av de aritmetiske operatorene "+" , "-" , "*" og "/ " eller en tallverdi. Programmet skal fungere som følger: Når en node inneholder en streng med et tall så skal integerverdien til strengen pushes på stacken. Når en node inneholder en operand så skal man først evalure verdien til de to barna før man poper høyre- og venstrebarnets verdier av stacken og utfører operatoren på de to operandene. Resultatet av verdien skal pushes på stacken. Når programmet har kjørt ferdig skal stacken kun inneholde verdien uttrykkstreet evaluerer til.

2c Operasjoner på stack (vekt 2%)

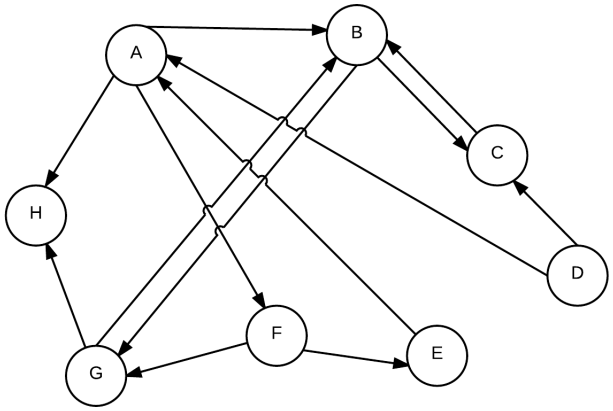
Skriv de 10 første push- og pop-operasjonene som resulterer når programmet ditt er anvendt på uttrykkstreet i Fig. 1. (F.eks, push 4; pop 4; push 7; ...)

(Fortsettes på side 4.)

Oppgave 3 Grafer (vekt 14%)

3a Sterkt sammenhengede komponenter (SCCs) (vekt 7%)

Gitt en rettet graf med noder $A, \ldots, H$ som vist i Figur 2.



Figur 2: En rettet graf

1. Hvilke sterkt sammenhengende komponenter (Strongly Connected Components (SCCs)) har grafen i Figur 2? Du skal illustrere hvordan SCCs er funnet algoritmisk ved å vise trinnene i algoritmen.
2. Gitt en vilkårlig rettet graf G, la G' være en rettet graf der hver node i G' representerer en SCC av G. For nodene u og v i G' finnes det en kant (u, v) hvis det finnes en kant i G som forbinder SCCene som tilsvarer u og v. Kan G' sorteres topologisk? Begrunn kort.

3b Korteste vei (vekt 7%)

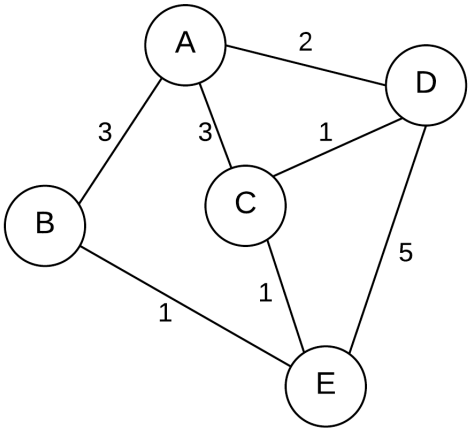
Korteste vei-en til alle-problemet (single-source shortest path problem) handler om å finne korteste vei fra en startnode til alle andre noder i en graf. Det kan være mer enn en vei som har den minste kosten. For eksempel, for grafen G i Figur 3 med startnode A, har vi 3 stier fra A til E med kost 4 ($\langle A, C, E\rangle$, $\langle A, D, C, E\rangle$ og $\langle A, B, E\rangle$).

Implementer en algoritme som i tillegg til å finne korteste vei fra en gitt startnode s til alle andre noder w også beregner antall korteste veier fra s til w. La `int num_paths` være den variabelen i klassen `Vertex` som inneholder denne informasjonen. Du kan anta at grafen ikke har negative kanter.

Oppgave 4 Produktsum (vekt 13%)

Gitt en liste av n heltall, $v_1, \ldots, v_n$, er *produktsummen* av listen den *største* sum man kan lage ved å multiplisere tilstøtende elementer i listen. Hvert element kan

(Fortsettes på side 5.)



Figur 3: Grafen G

maksimalt bli matchet med en av sine naboer. For eksempel, for listen $1, 2, 3, 1$ er produktsummen 8 ($= 1 + (2 \times 3) + 1$) og for listen $2, 2, 1, 3, 2, 1, 2, 2, 1, 2$ er produktsummen 19 ($= (2 \times 2) + 1 + (3 \times 2) + 1 + (2 \times 2) + 1 + 2$).

4a Optimalisering (vekt 6%)

Gitt en liste av n heltall for $n \geq 2$. Hva er den optimale produktsummen $OPT(j)$ for de første j elementene i listen? Du skal altså finne ut hva $OPT(j)$ gir for $j \in \{0, 1, \ldots, n\}$.

4b Dynamisk programmering (vekt 7%)

Gi en dynamisk programmeringsløsning for dette problemet. Implementer metoden `prodSum` som returnerer produktetsummen av de `j` første elementene av en liste `v`:

```
int prodSum(int [] v, int j) {

... //din kode

}
```

Metoden trenger kun å returnere selve produktsummen.

Oppgave 5 Tekstalgoritmer (vekt 10%)

5a Boyer Moore (vekt 4%)

Beregn *good suffix shift* til nålen:

SUSSENUSS

(Fortsettes på side 6.)

5b Huffman-koding (vekt 6%)

Gitt følgende tekst:

```
bibbidi-bobbidi-boo
```

1. Vis frekvenstabellen og Huffman-treet for setningen.
2. Hva er Huffman-koden for de ulike tegnene?

Oppgave 6 Internettsøkeprogrammer (vekt 26%)

Et problem for internettsøkeprogrammer som Google, Bing og DuckDuckGo, er at et søk kan generere millioner av svar (søker du på 'Obama' i Google får du 595 millioner svar!). Disse svarene er mer eller mindre relevante for den som søkte. Vedkommende søker har heller aldri muligheter for å se på alt. Hen vil se på det mest relevante. Hver side (av de mange millioner treff) kan vi anta har en relevans-score, som er et heltall som er slik at jo større dette tallet er, desto mer relevant er vedkommende side.

Vi skal hjelpe DuckDuckGo til å skrive, og senere i oppgave 6c parallellisere, løsningen på dette problemet. Anta at du har n svar og at relevans-scoren ligger lagret i heltallsarrayen $a[0..n-1]$. En triviell sekvensiell løsning i Java er å bruke Javas innebygde sorterings algoritme (Arrays.sort(int [ ] a)), sortere hele arrayen og så plukke ut de 50 største tallene. Men dette tar alt for lang tid. En klart raskere algoritme er følgende:

1. Vi antar først at de 50 tallene er i $a[0..49]$ er de største og innstikksorterer dette området i $a[\,]$ i synkende rekkefølge (du må trivielt skrive om vanlig innstikksortering til å sortere i synkende rekkefølge).
2. Da vet vi at det minste tallet av de 50 første tallene i $a[\,]$ da ligger i $a[49]$. Dette tallet sammenligner vi etter tur med hvert element i $a[50..n-1]$. Finner vi element $a[j] > a[49]$, så :
   a. Bytt $a[49]$ med $a[j]$
   b. Innstikk-sorter det nye elementet inn i $a[0..48]$ i synkende rekkefølge.
3. Når pkt. 2 er ferdig, ligger de 50 største tallene i $a[0..49]$ og ingen av øvrige tallene er overskrevet eller ødelagt.

6a Sortering i synkende rekkefølge (vekt 6%)

Skriv først en sekvensiell metode:

```
void baklengsInnstikkSortering(int[] a, int fra, int til)
```

som sorterere området $a[fra..til]$ i en større array $a[\,]$ i *synkende* rekkefølge.

(Fortsettes på side 7.)

6b Finn mest relevante svar (vekt 10%)

Skriv en sekvensiell metode:

```
void finn50Største(int[] a) {..<din Java-kode her>..}
```

som implementerer algoritmen som skisseres ovenfor og som kaller først metoden fra oppgave 6a. For at du skal få bedre enn C på denne deloppgaven, må du gjøre noe annet enn bare å kalle metoden fra oppgave 6a hver gang vi har fått et nytt, større element i $a[49]$.

Skriv en ny metode for dette enklere tilfellet hvor bare $a[49]$ muligens er feilplassert i $a[0..49]$

6c Parallellitet (vekt 10%)

Skriv med ord (ikke skriv kode) hvordan du ville parallellisere dette problemet med tråder hvis du har en multikjerne maskin med k kjerner. Svar da minst på følgende spørsmål:

- Hvordan ville du dele data mellom de trådene du starter?
- Når og hvordan vil du synkronisere de trådene du har (både main-tråden og de trådene du startet for å få beregne svaret for dette problemet)?
- Hvordan vil du sette sammen beregningene fra trådene til et samlet svar på problemet; det å finne de 50 største tallene i en meget stor array $a[0..n]$?

(For n = 100 millioner er den sekvensielle algoritmen ca. 70 ganger raskere og den parallelle algoritmen med 8 kjerner, 273 ganger raskere enn Arrays.sort).

Oppgave 7 Algoritmer og kompleksitet (vekt 16%)

For hver av påstandene under: svar på hvorvidt den er *sann* eller *usann* og gi en kort begrunnelse på svaret ditt der du skal vise forståelse for konseptene som er innvolvert.

7a (vekt 2%)

Vi vet at vi kan finne et element i en liste med *n* elementer på *O(log n)* tid ved å holde listen sortert og gjøre et binærsøk. Vi vet også at vi kan sortere *n* elementer på *O(n log n)* tid. Men vi kan aldri vite hvorvidt dette er den ideelle ytelsen vi kan oppnå og hvorvidt det er mulige å lage bedre algoritmer.

7b (vekt 2%)

B-trær tillater oss å ha en mer effektiv bruk av lagringsplass, men det er ingen forskjell i kjøretiden til b-trær sammenlignet med andre balanserte søketrær.

(Fortsettes på side 8.)

7c (vekt 2%)

Man kan gjøre søk mer effektivt med hashing enn det som er mulig med binærsøk og binære søketrær.

7d (vekt 2%)

Dobbel hashing er brukt for å unngå sekundære kollisjoner (secondary clustering).

7e (vekt 2%)

Utvidbar hashing løser problemet med kollisjoner ved å lagre verider i en lenket liste.

7f (vekt 2%)

Ved å representere problemer som formelle språk oppnår vi et systematisk og enhetlig syn på problemer og algoritmer.

7g (vekt 2%)

Det er viktig å kunne skille mellom problemene som har løsninger i polynomiell tid og de som ikke har det.

7h (vekt 2%)

Alan Turing lagde den første kommersielle datamaskinen, kalt Turingmaskinen.