UNIVERSITETET I OSLO

Det matematisk-naturvitenskapelige fakultet

Eksamen i	INF2220 — Algoritmer og datastrukturer
Eksamensdag:	16. desember 2013
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.
- **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.
- I Oppgave 6b.3 står det **(kan puffes)** og det betyr at du får karakter E og ikke F, dersom du ikke svarer på denne deloppgaven.

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

(Fortsettes på side 2.)

Oppgave 1 Tidskompleksitet (vekt 10%)

Hva er *worst case*-tidskompleksiteten til følgende implementasjoner, som en funksjon av parameteren n:

1a For-løkker (vekt 5%)

```
int s = 0;

for (int i = 1; i <= n; i++) {
  for (int j = 1; j <= i; j++) {
    for (int k = i; k <= j; k++) {
      s = s + k ;
    }
  }
}
```

1b Rekursjon (vekt 5%)

```
int proc(int n) {
  int x = 1;

  if (n > 1) {
    for (int i = 1; i <= n - 1; i++) {
      x = x + proc(i);
    }
  }
  return x;
}
```

Oppgave 2 Binære søketrær (vekt 15%)

Fullfør de to programsegmentene under slik at de implementerer følgende oppgaver korrekt:

2a Finn noder i søketrær (vekt 5%)

Gitt et binært søketre og et heltall x. Metoden `find` skal returnere noden V som har verdi lik x (du kan forutsette at den finnes). Fullfør de tre kodelinjene som starter med `return`.

```
class BinNode {
  int value;
  BinNode leftChild;
  BinNode rightChild;

  BinNode find(int x) {
```

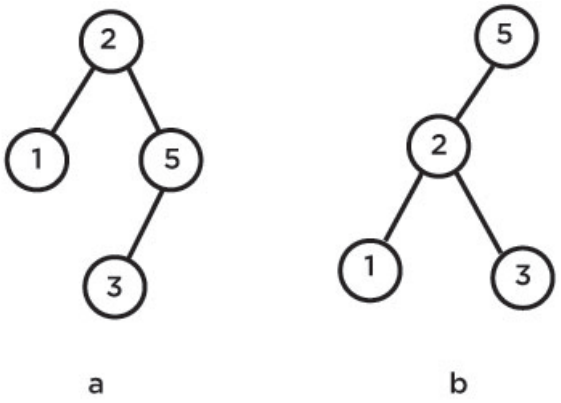
(Fortsettes på side 3.)

```
    if (value == x) {
      return ........;
    }
    else if (value < x) {
      return ........;
    }
    else {
      return ........;
    }
  }
}
```

2b Rotering i søketrær (vekt 6%)

Under er koden til en del av et program som skal gjøre en node V til den nye rotnoden av et binært søketre ved gjentatte ganger å bytte V med foreldrenoden sin (se eksempelet i Figur 1). Koden nedenfor utfører kun et enkelt bytte. Du skal fullføre koden og sørge for at søketre-egenskapen opprettholdes. Du kan anta at V er den noden vi ønsker å bytte, og at P er foreldrenoden til V.

```
if (V == P.leftChild) {
    P.left = ..................;
     ................;
     ................;
}
else { // V is the right child of its parent P
     ................;
     ................;
     ................;
}
```



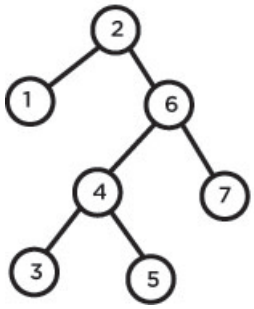
Figur 1: Noden med verdi 5 i figur **a** blir den nye rotnoden i figur **b**

2c Resultat av kjøring av algoritmen (vekt 4%)

La $x = 4$ og T rotnoden i et binært søketre som avbildet i Figur 2. Anta at rotasjonsalgoritmen fra 2b er utvidet til å hensynta ev. forfedrenoder til P. Hva

(Fortsettes på side 4.)

blir resultatet av å utføre denne fullstendige algoritmen på disse parameterene? Dvs. en algoritme som finner noden med verdi 4 under T (ved å kalle `T.find(4)`) og deretter utfører en serier av bytter til denne noden blir den nye rotnoden. Tegn det nye treet.

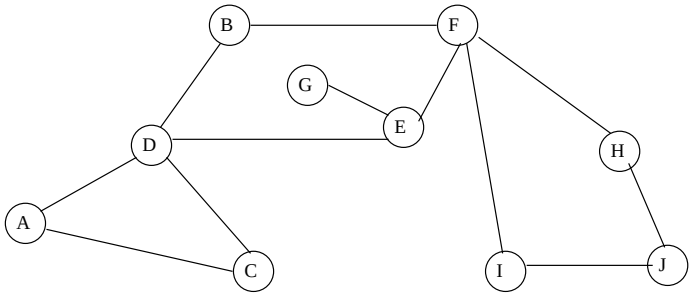


Figur 2: Et binært søketre

Oppgave 3 Grafer (vekt 26%)

3a Biconnectivity (vekt 4%)

Finn alle *articulation points* for grafen under. Vis et dybde først-spenntre som starter fra node A samt *Num*- og *Low*- numrene for hver node.



3b Løkker i ikke-sammenhengende grafer (vekt 10%)

La G være en graf med følgende engenskaper:

- den er urettet.
- den *kan* være ikke-sammenhengende.
- den *kan* inneholde løkker.

1. Anta at G har n noder og m sammenhengende komponenter. Hvor mange kanter kan G maksimalt inneholde dersom grafen skal være asyklisk?

(Fortsettes på side 5.)

2. Skriv en metode som finner *minimum* antall kanter som må fjernes fra G for at den resulterende grafen skal bli asyklisk. I tillegg til at metoden skal returnere antall kanter som må fjernes fra G for å gjøre den asyklisk, skal metoden også identifisere hvilke kanter som kan fjernes (det holder at disse skrives ut til skjerm). Det er tillatt å bruke hjelpemetoder hvis du ønsker det.

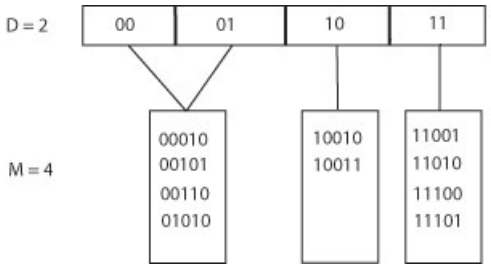
3c Hamiltonsk sti (vekt 12%)

En *Hamiltonsk sti* er en sti gjennom en graf som besøker alle nodene i grafen **en** gang. La grafen $G = (V, E)$ være en *rettet asyklisk graf* (DAG).

Skriv en metode `HamiltonskSti` slik at den, gitt grafen G, returnerer *true* hvis G inneholder en Hamiltonsk sti og ellers returnerer *false*. Algoritmen skal ha *lineær* tidskompleksitet. Begrunn hvorfor algoritmen din tilfredstiller tidskravet.

Oppgave 4 Utvidbar hashing (vekt 10%)

Figuren under viser en utvidbar hashtabell



Figur 3: En utvidbar hashtabell

1. Vis hashtabellen du får ved å sette inn **00000** i Figur 3?
2. Vis hashtabellen du får ved å sette inn **11111** i Figur 3?

Oppgave 5 Tekstalgoritmer (vekt 9%)

5a Boyer Moore (vekt 5%)

La nålen være: **cdfcfcfc**

1. Beregn *good suffix shift* til nålen.
2. La *bad character shift* bestå av bl.a.: (f,1) (d,6) (c,2) (v,8).
   Det er funnet en mismatch mellom nål og høystakk:

(Fortsettes på side 6.)

```
··· c d f c f v f c ···
    c d f c f c f c
              ↑
```

Hvor langt vil Boyer Moore-algoritmen flytte nålen? Begrunn!

5b Huffman koding (vekt 4%)

La de 8 første alfabetene ha følgende frekvenstabell basert på de 8 første fibonacci-tallene:

- **a:1**
- **b:1**
- **c:2**
- **d:3**
- **e:5**
- **f:8**
- **g:13**
- **h:21**

1. Hva er Huffmankoden for disse alfabetene? Du må vise Huffmantreet basert på frekvenstabellen.
2. Generaliser ditt svar for å finne den optimale koden når frekvensene er de første n Fibonacci-tallene.

Oppgave 6 Sortering (vekt 20%)

6a Innstikksortering (vekt 6%)

Her er en versjon av innstikksortering:

```
/**
* Innstikksorter a[i] fra og med plass v til og med plass h
*   - dvs. a[v..h]
************************************/
void insertSort(int [] a, int v, int h)   {
    int i, t;
    for (int k = v ; k < h; k++) {
        if (a[k] > a[k+1]) {  // KAN FJERNES 1
            t = a[k+1];
            i = k;
            while (i >= v && a[i] > t) {
                a[i+1] = a[i];
                i--;
```

(Fortsettes på side 7.)

```
            }
            a[i+1] = t;
        }                          // KAN FJERNES 2
    }
} // end insertSort
```

To av linjene er merket med kommentar // KAN FJERNES.
Begrunn **hvorfor** begge disse to linjene kan fjernes samtidig, og at metoden fortsatt vil virke som innstikk-sortering. Gi også en kort vurdering om dette er en fornuftig endring av koden.

6b Flettesortering (vekt 14%)

Du skal nå skrive en rekursiv versjon av flettesortering, som består av to metoder (omtrent som quicksort). Her er den første oppgitt:

```
void fletteSort(int [] a) {
  if (a.length <= 50) insertSort(a,0,a.length-1);
  else {
    int [] b = new int [a.length];
    flette(a,b,0,a.length-1);
  }
} // end fletteSort
```

Du skal skrive den rekursive metoden `flette`:

```
void flette(int [] a, int [] b, int v, int h){

  <... din kode her ...>

}// end flette
```

som sorterer en del av en heltallsarray fra og med element `a[v]` til og med element `a[h]` - dvs: `a[v..h]` som følger:

1. Du skal bruke `insertSort` fra Oppgave 6a som sub-algoritme, dele arrayen `a[]` i to på hvert nivå og gå rekursivt ned i hver halvdel til lengden av det du skal sortere er ≤ 50. Da skal du sortere den delen med `insertSort`.
2. På 'backtrack' (etter retur av de to rekursive kallene i `flette`), skal du flette de to sorterte delene av `a[v..h]` ved:
   i. Flette-sortere dem over i 'sin del' av arrayen `b` - dvs. i `b[v..h]`.
   ii. Kopiere den sorterte (nå dobbelt så lange) sekvensen fra `b[]` tilbake til samme plasser i `a[]`.
   iii. Når da det opprinnelige, første kallet på `flette` returnerer, er hele `a[]` sortert.
3. (kan puffes) Forklar hvordan alle kopieringene fra `b[]` tilbake til `a[]` kan unngås ved å endre på de aktuelle (kall-) parametrene på de rekursive

(Fortsettes på side 8.)

kallene på `flette` samt legge til én parameter i metoden `flette`: **int dybde**, som er dybden i rekursjonstreet (dvs. økes med 1 for hvert kall). Her skal du **ikke** skrive kode, bare forklare hvilke endringer du vil gjøre med koden du skrev på punkt 2 for å få til flettesortering uten kopiering.

Oppgave 7 Diverse oppgaver (vekt 10%)

Gi et kort svar, ikke mer enn 3 setninger, til hver av følgende spørsmål. Hvert spørsmål teller 2%.

1. Hvorfor bruker vi Big-O ($O(f(n))$) for å estimere kompleksitet av algoritmer? Hvorfor ikke bruke eksakte funksjoner, eller faktiske kjøretid?
2. Hva oppnår vi ved å dele problemer i kompleksitetsklasser?
3. Er uavgjørbarheten av Stoppe-problemet (*Halting problem*, det første problemet bevist å være uløsbart) bevist ved hjelp av reduksjon?
4. I hvilke situasjoner er det best å bruke utvidbar hashing?
5. Når sier vi at et problem er "intractable"?