UNIVERSITETET I OSLO

Det matematisk-naturvitenskapelige fakultet

Eksamen i:	**INF 2220 –** ***Algoritmer og datastrukturer***
Eksamensdag:	**8. desember 2016**
Tid for eksamen:	**09:00 – 13:00 (4 timer)**
Oppgavesettet er på:	**7 sider**
Vedlegg: Ingen	
Tillatte hjelpemidler:	**Alle trykte og skrevne**

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

Generelle råd:

- Skriv **leselig** (uleselige svar teller ikke…)
- Husk å **begrunne** svar
- Skriv **korte** og **presise** kommentarer. Hvis du bruker kjente datastrukturer (list, set, map) 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.
- **Vekten** till en oppgave indikerer **vanskelighetsgraden** og tidsbruken du bør bruke på oppgaven, ut i fra våre estimat. Dette kan være greit å benytte før å disponere tiden best mulig, dvs. Ikke bruk mye tid på en oppgave med liten prosentsats (gå videre).
- **Står det at du skal skrive Java-kode,** så får du ikke poeng for pseudo-kode. Få, små kodefeil som manglende ; eller lignende feil trekker ikke ned i Java-koden.

lykke til!

Ingrid Yu, Arne Maus og Dino Karabeg

Oppgave 1 - Stor O (9 %)

Hva er 'worst-case' tidskompleksiteten til følgende programkoder, som funksjon av input-parameteren n*?*

1a) (3 %)

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

Svar 1.a

O ($\log^2$ n). 1 + 2 + … + log n = (log n + 1) log n / 2

1b) (3 %)

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

Svar 1.b

O (n log n). log 1 + log 2 + … + log n = log n! = O(n log n)

1c) (3 %)

```
int  rec (int n);
    if (n > 1) {
      return 3 * rec (n / 3)
}
```

Svar 1.c
Egentlig uløselig slik den står, ofg det er et OK svar. Skal vel være:

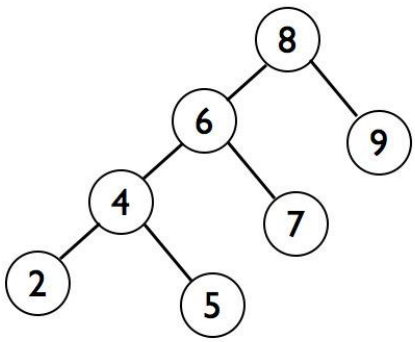
```
int  rec (int n){
    if (n > 1) {
      return 3 * rec (n / 3);
    else return 0;
}
```

Da er den :

O (log n). Time (n) = c + Time (n/3) = c + c + Time ($n/3^2$) =… = c $\log_3$ n = O(log n)

Oppgave 2 Rød-svarte trær (*10 %*)

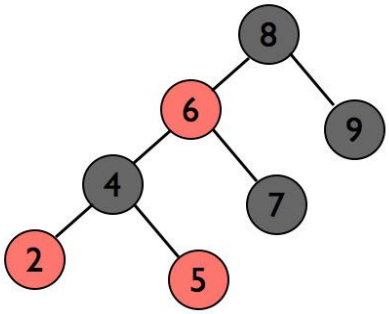
Begge spørsmål tar utgangspunt i dette treet som input.



2a (3 %)

Fargelegg treet slik at det blir et rødt-svart tre

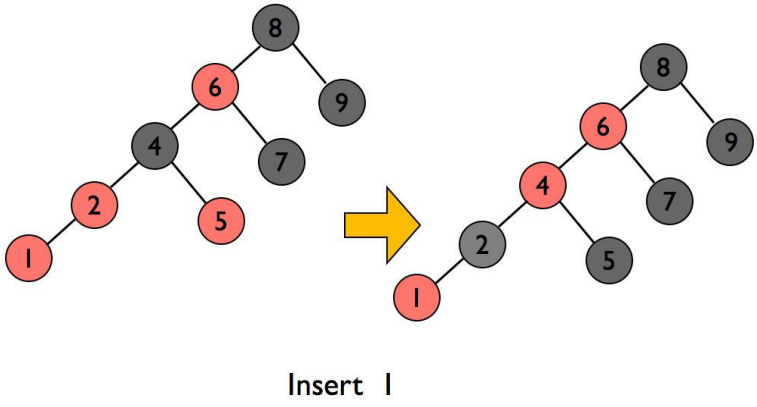
Svar 2a

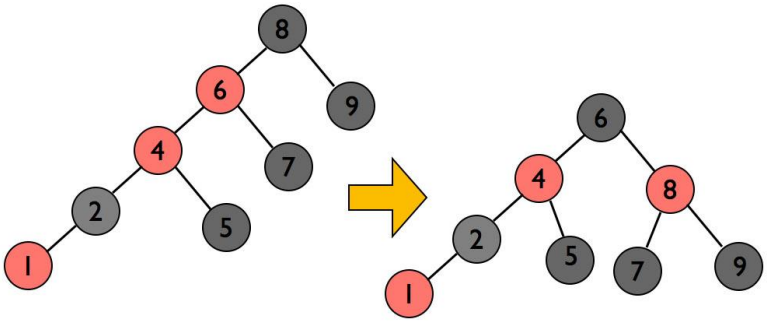


2b (7 %)

Hva skal treet se ut etter at tallet 1 har blitt satt inn? *Hvis alle skritt (rebalanseringer, fargeleggingen)*

Svar 2b





Insert 4

Oppgave 3 Sorteringsoppgave (35 %)

Her er en versjon av innstikksortering:

```
/**
* Innstikksorter a[] 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++)  {
         t = a[k+1];
         if (a[k] > t)  {
               i = k;
                while (i >= v && a[i] > t) {
                     a[i+1] = a[i];
                     i--;
                 }
               a[i+1] = t;
          }
      }
} // end insertSort
```

Nå tror din foreleser at han har funnet en ny og meget bedre rekursiv versjon av denne.
Algoritmen for

```
void superInnstikk (int [] a, int v, int h) {
    ..<din kode her>..
}
```

er:

Hvis lengden av det du skal sortere er lenger enn 200 så kaller du bare **`superInnstikk`** rekursivt to ganger med hver sin like store halvdel av

området a[v..h] (hvis lengden er et oddetall blir de to delene bare nesten like lange, men det gjør ingen ting).

Hvis lengden av området som skal sorteres er <= 200, så skal du fortsatt dele området i to like deler, men nå sortere hver av dem med kall på **`insertSort`** som gitt ovenfor.

Etter at du da rekursivt har kommet i bunn og sortert to halvdeler med **`insertSort`**, skal du i hvert kall på **`superInnstikk`**, på tilbaketrekking (backtrack) skjøte sammen de to sorterte halvdelene som du da har etter følgende regler (du trenger en peker, **`hPek`** for Høyre del, initielt er **`hPek = h;`**):

Du skal ***ikke*** behandle de to delene likt, men du sitter nå med to sorterte halvdeler: Venstre og Høyre. Hvis øverste element i Høyre er større enn øverste element i Venstre, flytter du bare pekeren **`hPek`**, én plass nedover.

Hvis Venstre sitt øverste element er større enn Høyre sitt element der høyre-pekeren peker, dvs: **`a[hPek]`**, bytter du først bare de to elementene (slik at det største kommer på Høyre del). Du senker fortsatt **`hPek`** med en. Det elementet du satte inn i toppen av Venstre tar du så og innstikksorterer nedover i Venstre; skyver på elementene slik at det nye elementet kommer sortert inn i Venstre slik det også gjøres i koden ovenfor.

Når pekeren i Høyre er ferdig med hele Høyre del, altså har gått gjennom alle elementene der, er du ferdig og dette kallet på **`superInnstikk`** og kan da returnere til den som kalte den med sin del av **`a[]`** – dvs. **`a[v..h]`** sortert.

3a) (20%)

Skriv sekvensiell Java-kode for denne rekursive algoritmen.
(for å sortere en array a[], kaller man metoden slik : **`superInnstikk(a, 0, a.length-1);`**)

```
static void superInnstikk (int [] a, int v, int h) {
        int i,temp, half =(v+h)/2, lasti = half;

        if ((h-v) > 200 ){
                superInnstikk (a, v, half);
                superInnstikk (a, half+1, h);
        } else {
                insertsort (a,v,half);
                insertsort (a, half+1,h);
        }
```

```
                    // now backtrack
                    for (int hPek = h  ; hPek < half; hPek--){
                            temp = a[hPek];
                            if (a[half] > temp){
                                    // bytt toppen av Vestre og a[hPek]
                                       swap (a, half, hPek);

                                     // 3 a :
                                      for ( i=half; i > v && a[i-1]> temp; i -- ){
                                                a[i]= a[i-1];
                                       }

                                       // 3c: first move before last move (N.B Korr)
                                       for (int k = half; k > lasti; k--)
                                         a[k] = a[k-1];

                                       for ( i=lasti; i > v && a[i-1]> temp; i -- ){
                                                a[i]= a[i-1];
                                       }
                                       lasti = i;
                                       a[i] = temp;
                            } // end if
                    }// end hPek
          }// end insertSortPlus

          // swap elements v and r in a[]
          static void swap(int[] a, int v, int h) {
                    int temp = a[v];
                    a[v] = a[h];
                    a[h] = temp;
          } // end swap
```

3b) (3%)

Forklar hvorfor denne virkelig sorterer a[] og begrunn kort hvilken orden O() i kjøretid denne algoritmen har.

Denne sorterer a[v..h] fordi begge de to halvdelene forblir sorterete etter hovedløkka med hPek og alle i Høyre blir >= alle i Venstre (Høyre: Fordi vi evt får inn et nytt element i a[hPek]som er større enn det som var der (da er Høyre trivielt sorteret nedover a[half+1..hPek]).

Ovenfor hPek: a[hPek +1..h] er også Høyre sortert fordi hvis a[hPek+1] nå skulle være mindre enn a[hPek] så skulle den vært byttet med et elenement a[half] som var >= det som a[hPek] ble sammenlignet med. Altså er Høyre også sortert a[hPek..h]. Venstre er trivielt sortert hele tiden fordi vi gjør innstikksortering. Etter hovedløkka er er også a[half] <= a[half+1] (hPek == half+1). Altså er Høyre og Venstre begge sortert og Venstre <=Høyre. Da er samenstillingen av Vestre og Høyre: a[v..h]

sortert.

Orden på algoritmen er $O(n^2 logn)$ fordi vi gjør *logn-log100* innstikksorteringer i Venstre: $(100^2) + 100^2 + 200^2 + 400^2 .. + n/2*n/2 \sim (logn - log100)(n/2)^2$ operasjoner. $= O(n^2 logn)/4 = O(n^2 logn)$. I Høyre gjøres 'bare' n operasjoner $((100^2 + 200/2 + 400/2 + ..+ n/2) = (100^2 + n) = O(n)$.

3c) (5%)

En liten forbedring av algoritmen er hvordan innesettingen av et nytt element i Venstre gjøres – vi kan etter hvert under innsettingen i Venstre gjøre denne litt raskere. Skriv Java-kode for dette.

Se koden ovenfor. Vi vet at når vi bytter inn et element fra Høyre, er det <= det forrige vi byttet inn i Venstre. Vi kan altså først skyve alle elementene ett hakk høyreover fra forrige plass vi satte inn et element fra Høyre. Etter den plassen a[lasti], må vi teste for å finne riktig plass i Vestre for a[hPek].

3d) (7%)

Forklar, ikke skriv kode, for hvordan du rimelig lett kan parallellisere denne algoritmen.

Parallelliseringen går som parallelliseringen av Kvikksort. På nedtur starter vi tråder i stedenfor rekursive kall omtrent så mange som vi har kjerner – dvs ned til nivå 3 i treet hvis vi har 8 kjerner.. Etter det vil hver tråd utføre den sekvensielle superInnstikk-algoritmen. Vi får ingen konflikt med skriving her, fordi hver tråd behandler hver sin del av a[].

Oppgave 4 Jobbfordeling (19 %)

Et velkjent problem innen optimering og kombinatorikk kan beskrives som følgende: Gitt n jobber og m maskiner, hvor n > m, samt en liste over hvor lang tid hver av de n jobbene tar (positive heltall), finn en fordeling av disse n jobbene på disse m maskinene slik at totaltiden (= slutt-tidspunktet for den siste jobben) blir minst mulig. Det er ingen avhengigheter mellom jobbene så alle n jobber kan utføres parallelt og kan fordeles fritt på maskinene, men har man først startet en jobb på en bestemt maskin kan den ikke avbrytes men må fullføres på den maskinen.

For eksempel: Hvis n = 6, m = 2 og tidene er gitt ved (4, 2, 6, 2, 4, 1) (dvs. jobb 0 tar 4 tidsenheter, jobb 1 tar 2 tidsenheter, ...osv.) kan vi fordele jobbene på de to maskinene ved følgende fordeling:

Maskin 0: 0, 1, 4
Maskin 1: 2, 3, 5

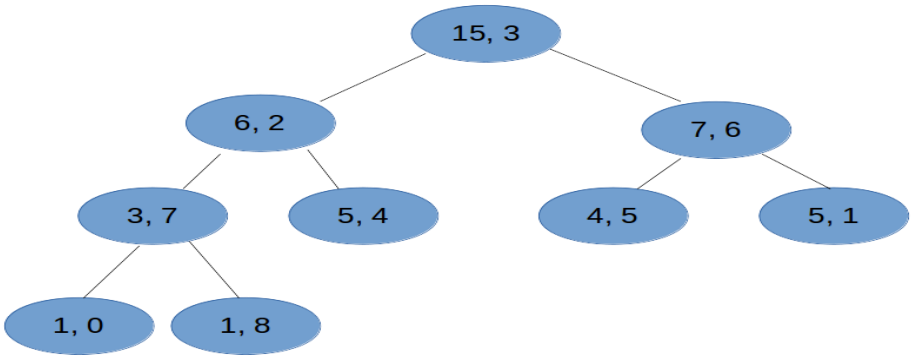
hvilket gir tiden 4 + 2 + 4 = 10 for maskin 0 og 6 + 2 + 1 = 9 for maskin 1, dvs.

totaltid 10 (maksimum av 10 og 9). Denne fordelingen kan representeres ved en array av lengde n, der det j-te elementet er k hvis jobb j fordeles til maskin k. I vart eksempel vil dette være (0, 0, 1, 1, 0, 1).

For å finne den optimale løsningen må man generelt bruke algoritmer med eksponentiell kompleksitet. Imidlertid kan man finne en god tilnærming med følgende algoritme: Hvis vi fremdeles har jobber igjen som ikke er fordelt på en maskin, finn den største av disse og la den maskinen (eller en av de maskinene) som utfører minst jobb til nå utføre denne jobben. Dette kan implementeres ved å bruke to heaps, en maksimumsheap (høyt tall = høy prioritet) for jobbene, og en minimumsheap (lavt tall = høy prioritet) for maskinene.

4a (4%)

Anta at du har fått følgende 9 jobber (n=9) med tider gitt ved
(1, 5, 6, 15, 5, 4, 7, 3, 1). Vi setter inn 9 par (tid, jobb-ID) i en prioritetskø basert på en maksimumsheap. Tegn heapen du får ved å sette inn parene ett om gangen i den gitte rekkefølgen.
Løsning:



4b (6%)

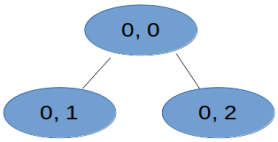
Anta at du er gitt m = 3 maskiner, alle med starttid 0 (ingen jobber fordelt til noen maskiner). Vi lager en prioritetskø for maskinene basert på en minimumsheap; maskinene er sortert etter tiden de allerede bruker for å utføre jobbene de har fått tildelt. Vis de to første rundene av følgende algoritmen:

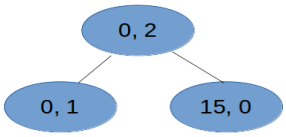
1. Hent ut det største elementet fra prioritetskøen for jobber fra deloppgave (a).
2. Hent ut det minste elementet fra prioritetskøen for maskiner.
3. Fordel jobben (fra pkt. 1) til rett maskin og beregn ny tid for denne maskinen.
4. Legg maskinen tilbake i prioritetskøen for maskiner, med ny tid.

For hver runde, vis hvordan prioritetskøene endres (du trenger ikke vise mellomsteg).

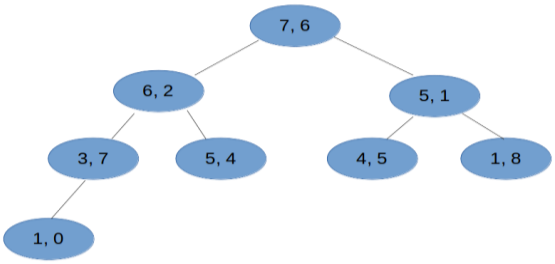
Løsning:

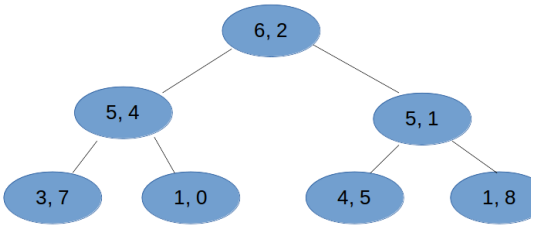
Etter 1 og 2 runde for maskin heap Etter 1 og 2 runde for jobb heap


0, 0
0, 1
0, 2


0, 2
0, 1
15, 0


0, 1
15, 0
7, 2


7, 6
6, 2
5, 1
3, 7
5, 4
4, 5
1, 8
1, 0


6, 2
5, 4
5, 1
3, 7
1, 0
4, 5
1, 8

4c (9%)

Skriv metoden i Java som implementerer algoritmen over. Metoden skal ta inn heltall n og m og en array int[] t, og skal returnere en array av lengde n som viser den endelige fordelingen av de n jobbene på de m maskinene.

Du kan anta at du har følgende klasse tilgjengelig:

```
class Pair implements Comparable {
   int time;
   int id;

   Pair(int time, int id) {
      this.time = time;
      this.id   = id;
   }

   public int compareTo(Object o) {
      Pair a2 = (Pair) o;
      return this.time - a2.time;
   }
}
```

En minimumsheap av slike par kan opprettes ved

```
PriorityQueue<Pair> minHeap = new PriorityQueue(n);
```

og en maksimumsheap ved

```
PriorityQueue<Pair> maxHeap = new PriorityQueue(n,
   new Comparator<Pair>() {
      public int compare(Pair a1, Pair a2) {
         return a2.compareTo(a1);
      }
   })
```

Løsning (c)

```
public static int[] jobscheduling(int n, int m, int[] times) {

      // make heap for jobs
      PriorityQueue<Pair> jobs = new PriorityQueue(n,
          new Comparator<Pair>() {
             public int compare(Pair a1, Pair a2) {
                return a2.compareTo(a1);
             }
          });

      // insert all jobs
```

```
    for(int i = 0; i < n; i++) {
        Pair p = new Pair(times[i], i);
        jobs.add(p);
    }

    // make heap for machines
    PriorityQueue<Pair> machines = new PriorityQueue(n);

    // insert with initial time 0
    for(int i = 0; i < m; i++) {
        Pair p = new Pair(0, i);
        machines.add(p);
    }

    // assign jobs using an array
    int assignment[] = new int[n];

    while(jobs.size()!=0) {
        // get largest job, smallest machine
        Pair jobpair = jobs.poll();
        Pair machinepair = machines.poll();

        // assign + calculate new time
        assignment[jobpair.id] = machinepair.id;
        machinepair.time += jobpair.time;

        // put machine pair back on heap
        machines.add(machinepair);
    }

    return assignment;
}
```

Oppgave 5 Korteste enkle sykel (15%)

I denne oppgaven skal du skrive pseudokode for to effektive algoritmer som gitt en rettet graf G med positive vekter returnerer vekten av den korteste enkle sykelen i G. Dersom G ikke inneholder noen sykel, retunerer du vekt ∞.

5a (4%)
Forklar hvorfor dybde først-søk ikke egner seg her.
Løsning:

(a)
1)DFS not always find shortest cycle. There is the problem of exploring already explored vertices. The fully explored vertices will be marked by dfs, so when following another path that leads to an alredy explored v, dfs will not re-explore the vertice again even if following this node v will lead to another cycle.
If one use DFS to enumerate all paths, it is exponetial number of paths to explore. Hence not effective.

5b (6%)
Skriv en algoritme der du bruker Dijkstra for å finne den korteste enkle sykelen. Hvis du bruker Dijkstra slik som den er (umodifisert), trenger du ikke å gjengi koden, du kan da anta at du har en metode med signatur void dijkstra (Vertex s) . Modifiserer du Dijkstra, må du vise den modifiserte Dijkstras algoritme.

Hva er kompleksiteten til algoritmen din?

(b)
best = ∞
For each vertex s of G do
{ // Run Dijkstra's algorithm from s by calling dijkstra(s) following signature inWeiss
For each edge t → s do
best = min(best, weight(t,s) + t.dist) //
}
return best

O(V (V + E)logV) using priority queue, if not O(n^3)

5c (5%)
Skriv en algoritme der du løser problemet ved bruk av Floyd i din algoritme.
(c)

med Floyd:

Siden Floyd finner korteste vei mellom alle par av noder, kan vi gå gjennom alle parene og finne par av noder som minimerer dist(u,v) + dist(v,u):

Kjøre Floy slik som på forelesningen og deretter:
For each pair of vertices u, v
If (dist[u][v] + dist [v] [u] < best)
best = dist[u] [v]+ dist [v] [u]

return best

Oppgave 6, Spørsmål og svar (12 %)

Svar "sant" eller "falsk", og vis oss at du forstår akkurat hvorfor, og relaterte grunnleggende ideer, i et kort avsnitt som begrunner svaret.

6a (2%)

$3n^2 = O(n^3)$
Her er det to gode svar: **Sant** fordi etter definisjonen er $O(n^3)$ er funksjon som er større enn $3n^2$.
Oftest vil vi ha 'minste funksjon som er >= uttrykket og da ville vi si **Falsk** fordi $3n^2$ også er $O(n^2)$

6b (2%)

"Å gi en polynomisk algoritme" betyr det samme som "å gi en effektiv løsning".

False (but look at the answer) – efficiency of a solution is relative to the complexity of the problem. An algorithm is "efficient" if the two complexities match one another. We say, however, that a *problem* has an efficient solution if it has a polynomial-time algorithm.

6c (2%)

Det finnes minst to distinkte grunner til hvorfor insetting i en hashtabell kan ta lineær tid.

True – one reason is collisions (when the table has degenerated so that we need a linear number of rehashes); another reason is rehashing (when the table has become so full that rehashing is needed).

6d (2%)

Comparison sorting kan løses i worst-case lineær tid (O(n)).

False – it was proven in class (a lower bound) that comparison sorting *requires* time *n log n*.

6e (2%)

Etter en insetting er gjørt i et balansert rød- svart tre, kan treet alltid rebalanseres med maksimum 2 rotasjoner.

False – O(*log n*) rebalancing operations may be needed (all the way up to the root of the tree).

6f (2%)

Branch and bound kan brukes til å løse noen NP-komplette problemer i polynomisk tid.

False (but look at the answer) – any (worst-case, but this is implied) polynomial algorithm for an NP-complete problem would automatically mean (be a proof that) P = NP. What is true, however, is that B&B can vastly speed up solutions to certain practical instances of certain NP-complete problems**.**

---------- slutt på oppgavesettet ----------