UNIVERSITETET I OSLO

Det matematisk-naturvitenskapelige fakultet

Eksamen i	INF2220 — Algoritmer og datastrukturer
Eksamensdag:	14. desember 2015
Tid for eksamen:	14.30 – 18.30
Oppgavesettet er på 11 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 du 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 til følgende kodesegmenter:

1a For-løkker (vekt 4%)

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

Solution: $O(n^3)$

1b Rekursjon (vekt 3%)

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

Solution: uendelig

1c Rekursjon (vekt 3%)

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

Solution: $O(2^n)$

Oppgave 2 Trær (vekt 20%)

Til oppgavene under kan vi anta at treet består av noder av en nodeklasse som den under:

```
class Node{
   int value;
   Node left, right;
```

(Fortsettes på side 3.)

```
}
```

2a Minste element i et binærtre (vekt 3%)

Skriv en rekursiv metode for å finne det minste elementet i et binærtre uten søkeegenskaper (ikke et binært søketre).
Metoden skal ha signaturen `int min(Node n).`
Solution:

```
int min(Node n){
  if(n == null) return Integer.MAX_VALUE;

  int minLeft = min(n.left);
  int minRight = min(n.right);
  return smallest(minLeft, minRight, n.value);
}

int smallest(int x, int y, int z){
  return Math.min(x, Math.min(y, z));
}
```

2b Antall noder i et binærtre (vekt 3%)

Skriv en rekursiv metode for å telle antall noder i et binærtre.
Metoden skal ha signaturen `int count(Node n).`
Solution:

```
int count(Node n){
 if(n == null) return 0;
 return count(n.left) + count(n.right) + 1;
}
```

2c Median (vekt 10%)

Bruk metodene fra oppgave 2a og 2b til å lage en metode som finner medianen av dataene lagret i treet (medianen av en mengde tall er det tallet som det finnes like mange elementer som er større enn som det finnes elementer som er mindre enn). Lag gjerne hjelpemetoder.
Solution:

```
int median(Node n){
 int size = count(n);
 int x = 0;
 for(int i = 0; i < size / 2; i++){
   x = min(n);
   update(n, x);
 }
```

(Fortsettes på side 4.)

```
 if(size % 2 == 1){
   return min(n);
 }
  //Denne trenger vi ikke når vi antar odde antall elementer
  //else{
  // return (x + min(n)) / 2;
  //}
}


//Løsningsforlsaget endrer på value. En annen mulighet er å fjerne
//noden med verdi x
boolean update(Node n, int x){
 if(n == null) return false;
 if(n.value == x){
   n.value = Integer.MAX_VALUE;
   return true;
 }
 if(update(n.left, x)){
   return true;
 }
 if(update(n.right, x)){
   return true;
 }
 return false;
}
```

2d Tidskompleksitet (vekt 4%)

Hva er tidskompleksitet til løsningen din i oppgave 2c? Begrunn svaret ditt.
Solution:
Depeding on the solution. For the solution above we have $O(n^2)$. More precisely, one count (linear time) + n/2 min and updates (both linear time) which gives us $O(n^2)$

Oppgave 3 Julebord (vekt 30%)

I denne oppgaven skal du planlegge sitteplassordningen for et julebord. Du har en liste V over gjestene av type Person:

```
class Person{
 int id;

 ...

}
```

(Fortsettes på side 5.)

3a Bordplassering (vekt 15%)

Tenk deg at du også har fått en oppslagstabell T der $T[u.id]$ for $u \in V$ gir en liste over gjestene (av typen Person) som u kjenner. Hvis u kjenner v så kjenner v også u. Du er pålagt å ordne sitteplasser slik at enhver gjest ved et bord kjenner alle andre gjester som sitter på det samme bordet, enten direkte eller gjennom noen andre gjester som sitter på det samme bordet. For eksempel, hvis x kjenner y og y kjenner z, da kan x, y og z sitte ved det samme bordet.

1. Dersom du må representere dette som et graf-problem, hva slags graf vil dette være? Og hva representerer noder og kanter i grafen?
2. Implementer en effektiv graf-algoritme som, gitt V og T som input, returnerer det minste antall bord som er nødvendig for å oppnå dette kravet.
3. Hva er kjøretiden til algoritmen din? Begrunn kort.

Solution:

1. max 2 points. An undirected graph. Guests as vertices and edges between two vertices means the two guests know each other. Table T represents the adjacency lists for the vertices.
2. max 10 points. If we start from one vertex s and search the graph using breadth-first search (BFS) or depth-first search (DFS), all the guests that are reachable from s can sit at the same table, and additional tables are needed for vertices that are unreachable from s . Hence, to find the minimum number of tables, we can iterate through s ∈ V . If s is not visited, increment the number of tables needed and call dfs-visit(s,T) or BFS(s, T) , marking vertices as visited during the traversal. Return the number of tables needed after iterating through all the vertices. This problem is equivalent to finding the number of connected components in the graph. Here is the pseudocode:

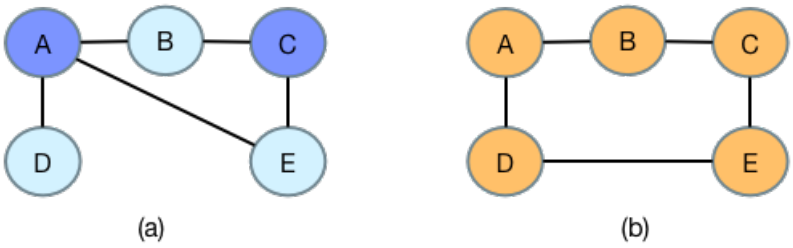
```
int numTables(V, T){
  int n = 0
  for s ∈ V
    if (s.visited == false){
         n = n + 1
         s.visited = true
         dfsVisit(s, T)
    }
  return n
}

void dfsVisit(u, T){
  for v ∈ T[u.id]
     if (v.visited == false){
        v.visited = true
        dfsVisit (v, T)
     }
}
```

3. max 3 points. if using DFS as above the running time is O(V + E) because every vertex or edge is visted exactly once.

(Fortsettes på side 6.)

3b Uvenner (vekt 15%)

Anta nå at det kun er 2 bord, og du får en annen tabell S der $S[u.id]$ for $u \in V$ gir en liste over gjestene som er på dårlige vilkår med u. Hvis v er uvenner med u, er u også uvenner med v. Målet ditt er nå å ordne sitteplasseringen slik at ingen gjester som sitter på samme bord er uvenner med hverandre. (I denne oppgaven skal vi ikke ta hensyn til om gjestene kjenner hverandre)
Figur 1 viser to grafer der gjester er representert som noder og en kant mellom to noder betyr at disse gjestene er uvenner med hverandre. For grafen **(a)** ser vi at det er mulig å ha A og C på et bord og B, D og E på et annet bord, mens for **(b)** ser vi at dette er umulig.



Figur 1

Implementer en effektiv algoritme som gitt listene V og S som input returnerer *true* hvis du kan plassere alle gjestene på to bord, ellers returnere *false*.
Note that like the previous task, the graph may not be connected.

```
bool twoTables(V,S){
  int white = 0
  for s ∈ V
    if (s.visited == false) // s is not visited
      if (dfsVisit(s, S, white)== false){
         return false
      }
  return true
}

bool dfsVisit(u, S,colorTOapply){
 if (u.visited == false){
    u.visited = true
    u.color = colorTOapply
    for v ∈ S[u.id]
       if (dfsVisit(v,S,1-colorTOapply) == false)
           return false
 }
 else if (u.color =/= colorTOapply)
    return false

 return true
}
```

(Fortsettes på side 7.)

Oppgave 4 Spenntre (vekt 10%)

La $G = (V, E)$ være en sammenhengede, urettet graf og la T være et minimalt spenntre av G. Anta nå at vi endrer vekten til kanten $e = (u, v)$ i G.

1. Forklar hvilke endringer av e som vil gjøre at T ikke lenger er et minimal spenntre av G. (NB. e er ikke nødvendigvis en kant i T).

2. Forklar en effektiv algoritme som ved minimale endringer på T gjør at T blir et minimalt spenntre igjen. Algoritmen kan ikke endre på vektene i grafen.

Solution: For 1:

- case 1: Suppose $e = (u, v)$ is in T, then T may no longer be an MST if the cost of e becomes larger than the cost of an edge not in T that is in any path between u and v.

- case 2: Suppose $e = (u, v)$ is not in T, then T may not be an MST if the cost of e becomes smaller than the largest cost in the path between u and v in T.

For 2:

- for case 1: e is removed such that now T is disconnected into trees T_1 and T_2(you can show that these are MSTs for the corresponding induced subgraphs), then the minimum cost edge joining a vertex in T_1 to a vertex in T_2 is added to make the MST.

- for case 2: the edge with largest cost in the path between u and v in T is removed and new edge is added to T as before.

Oppgave 5 Good Suffix Shift (vekt 5%)

Beregn good suffix shift til nålen:

sissossiss

Solution:

```
!s = 2
!ss = 1
!iss = 3
!siss = 9
!ssiss = 6
!ossiss = 6
!sossiss = 6
!ssossiss = 6
!issossiss = 6
!sissossiss = 6
```

(Fortsettes på side 8.)

Oppgave 6 Sortering (vekt 15%)

I denne oppgaven skal du gjøre endringer og forbedringer av quicksort og begrunne de endringene du gjør. I forelesningene 5. november ble følgende alternative kode for quicksort vist fram (metoden 'bytt' er triviell, endres ikke, og derfor ikke gitt her):
Her er koden fra 5. november (du kan anta at high > low):

```
void sekvQuick(int[] a, int low, int high) {
   // bare sorter arraysegments > 1
   int ind = (low+high)/2,
   piv = a[ind];
   int større=low+1, // hvor lagre neste 'større enn piv'
       mindre=low+1;   // hvor lagre neste 'mindre enn piv'

   bytt (a,ind,low);   // flytt 'piv' til a[lav] , sortér resten

   while (større <= high) {
          // test om vi har et 'mindre enn piv' element
          if (a[større] < piv) {
               // bytt om a[større] og a[mindre], få en liten ned
               bytt(a,større,mindre);
               ++mindre;
          } // end if - fant mindre enn 'piv'
          ++større;
   } // end gå gjennom a[low+1..high]

   bytt(a,low,mindre-1);  // Plassert 'piv' mellom store og små

   if (mindre-low > 2) sekvQuick (a, low,mindre-2); // sortér alle < piv
                                                    //  (untatt piv)
   if (high-mindre > 0) sekvQuick (a, mindre, high); // sortér alle >= piv
} // end sekvensiell Quick
```

Av og til må man sortere store mengder av data med få mulige verdier. F.eks hvis du skal sortere alle innbyggere i Norge (5,2 mill personer) på alder (0-115 år) blir det mange like verdier ('dubletter') - om lag 60 000 personer med samme alder i de fleste årene. Lag tillegg/endringer til koden ovenfor slik at vi i størst mulig grad ikke sorterer videre på elementer som er lik 'piv'. Du skal ikke her ha en 100% løsning, bare en løsning som ganske raskt vil løse dette problemet, og som i alle fall ikke sorterer videre hvis alle elementene har samme verdi som 'piv' i det området vi nå sorterer. Du skal skrive *hele* koden slik den nå ser ut med endringene du gjør på dette punktet. Begrunn også på noen få linjer hvordan dette nå løser på en bra måte problemet med å sortere mange like verdier.
Solution:

(Fortsettes på side 9.)

N.B. Det er to like raske og gode løsninger (Rødt – se tilleggene)

Løsning 1:
Hewr er ideenn å hoppe iver alle elementer som etter sorteing og som står inntil a[mindre] og som (tilfeldigvis) er lik *piv.*

```
void sekvQuick( int[] a, int low, int high) {
        // bare sorter arraysegments > len =1
        int ind =(low+high)/2,
        piv = a[ind];
        int større=low+1, // hvor lagre neste 'større enn piv'
        mindre=low+1;   // hvor lagre neste 'mindre enn piv'
        bytt (a,ind,low);   // flytt 'piv' til a[lav] , sortér resten

        while (større <= high) {
                // test om vi har et 'mindre enn piv' element
                if (a[større] < piv) {
                        // bytt om a[større] og a[mindre], få en liten ned
                        bytt(a,større,mindre);
                        ++mindre;
                } // end if - fant mindre enn 'piv'
                ++større;
        } // end gå gjennom a[low+1..high]

        bytt(a,low,mindre-1);  // Plassert 'piv' mellom store og små

        større = mindre;  // mindre might be high+1
        while (større <= high && a[større] == piv ) større++;

        if ( mindre-low > 1) sekvQuick (a, low,mindre-1);  // sortér alle < piv
        if ( high-større > 0) sekvQuick (a, større , high);  // sortér alle > piv

} // end sekvensiell Quick
```

Alle elementer som er >= piv ligger fra a[mindre-1] og oppover. Denne løkka fjerner alle a[j] == piv som ligger inntil a[mindre-1] og hvis alle er like, fjerner vi alle og da all videre rekursjon.

Løsning 2:

Her er ideen å flytte ***alle*** elementer som er lik *piv* inntil piv og så hoppe over dem. Ulempen er selvsagt at vi gå gjennom halve arrayen en gang til.

```
void sekvQuick( int[] a, int low, int high) {
        // bare sorter arraysegments > len =1
        int ind =(low+high)/2,
        piv = a[ind];
        int større=low+1, // hvor lagre neste 'større enn piv'
        mindre=low+1;   // hvor lagre neste 'mindre enn piv'
        bytt (a,ind,low);   // flytt 'piv' til a[lav] , sortér resten

        while (større <= high) {
                // test om vi har et 'mindre enn piv' element
                if (a[større] < piv) {
                        // bytt om a[større] og a[mindre], få en liten ned
                        bytt(a,større,mindre);
                        ++mindre;
                } // end if - fant mindre enn 'piv'
                ++større;
        } // end gå gjennom a[low+1..high]

        bytt(a,low,mindre-1);  // Plassert 'piv' mellom store og små

        større = mindre;  // mindre might be high+1
        int start = mindre;
        while (start <= high ){
                if( a[start] == piv ){
                        bytt(a,start,større);
                        større++;
                }
                start++;
        }
        if ( mindre-low > 1) sekvQuick (a, low,mindre-1);  // sortér alle < piv
        if ( high-større > 0) sekvQuick (a, større , high);  // sortér alle > piv

} // end sekvensiell Quick
```

Oppgave 7 Diverse Oppgaver (vekt 10%)

For hver av påstandene under: svar på hvorvidt den er sann eller usann og gi en kort begrunnelse (maks 3 setninger) på svaret ditt. Hvert spørsmål teller 2%.

1. Søking i binære søketrær utføres i verste fall på logaritmisk tid.
2. Utvidbar hashing brukes når vi ikke vet størrelsen på datamengden.
3. Klassen NP-komplett inneholder noen uløselige problemer.
4. Noen uløselige problemer kan løses i polynomisk tid gjennomsnittlig (average-case).
5. Det er ikke mulig å bevise at en algoritme er en tidsoptimal (så effektiv som mulig) løsning på et problem.

Solution:

1. Falsk; sant bare for balanserte søketrær.
2. Falsk; Utvidbar hashing brukes når tabellen lagres på disken.
3. Falsk; alle NP-komplette problemer kan løses (i eksponensiel tid).
4. Falsk; hvis en instans krever uendelig tid, så blir det uendelig tid on average
5. Falsk; noen problemer har bevisbare 'lower bounds'; bevist for sortering og søking i timen.