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;
}
```

Solution:
for correction: all or nothing

1. $O(n^2)$ (uttrykket' s =s+k' utføres bare $O(n^2)$ ganger - faktisk vel ca. $n^2/2$ ganger, fordi innerste løkke (k) bare enten går null ganger(j<i) eller 1 gang (når j = i)

2. $O(2^n)$ (antall kall dobler seg når vi går fra k til k+1)

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

(Fortsettes på side 3.)

starter med `return`.

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

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

Solution:

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

  BinNode find(int x) {
    if (value == x) {
      return this;
    }
    else if (value < x) {
      return rightChild.find(x);
    }
    else {
      return leftChild.find(x);
    }
  }
}
```

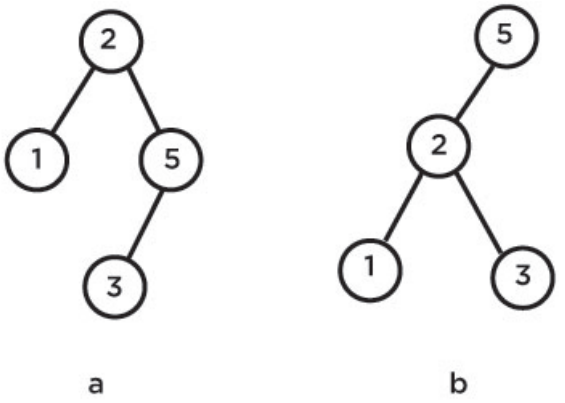
2b Rotering i søketrær (vekt 6%)

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 = ..................;
     ................;
     ................;
}
```

(Fortsettes på side 4.)

```
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**

Solution:

```
if (V == P.left)  {
  P.left = V.right;
  V.right = P;
}
else {
  P.right = V.left
  V.left = P
}
```

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

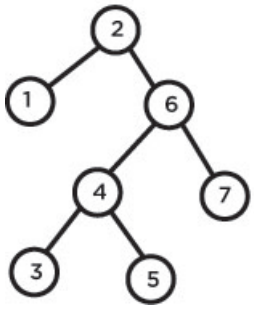
La $x = 4$ og T rotnoden i et binært søketre som avbildet i Figur 2. Anta at rotasjonsalgoritmen fra 2b er utvidet til å hensynta ev. forfedrenoder til P. Hva blir resultatet av å utføre denne fullstendige algoritmen på disse parameterene? Dvs. en algoritme som finner noden med verdi 4 under T (ved å kalle `T.find(4)`) og deretter utfører en serier av bytter til denne noden blir den nye rotnoden. Tegn det nye treet.
Solution:

Oppgave 3 Grafer (vekt 26%)

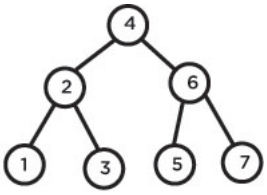
3a Biconnectivity (vekt 4%)

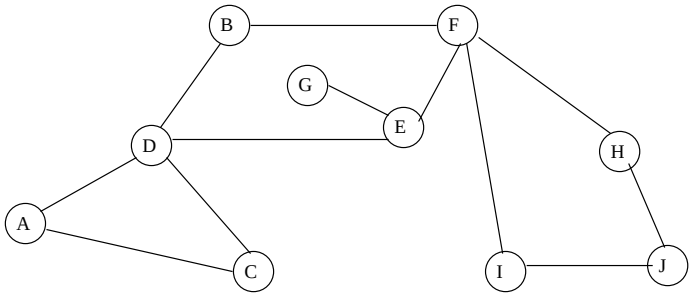
Finn alle *articulation points* for grafen under. Vis et dybde først-spenntre som starter fra node A samt *Num*- og *Low*- numrene for hver node.

(Fortsettes på side 5.)



Figur 2: Et binært søketre





Solution: E, F, D. Note that A is not an articulation point. The test of A as root would mark it correctly.

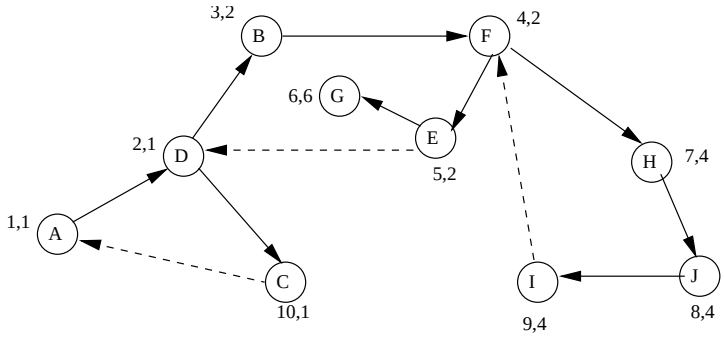
3b Løkker i ikke-sammenhengende grafer (vekt 10%)

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?
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.

(Fortsettes på side 6.)



Solution:
For correction: 3 points + 7 points

1. The answer is based on the observation that a connected component, having k vertices, must have at most k-1 edges to be acyclic (and at least k-1 to be connected). If there are n vertices in the graph and if there are m connected components, then there must be at most n-m edges remaining in the graph to make it acyclic.

2. Run a DFS based algo to number the vertices in the order in which they are visited during DFS (like done in class). Use this visiting info to detect back edges (as in the algo for articulation points seen in class) and keep count of them. The total number of back edges is the minimum number of edges to be removed to eliminate cycles in the graph. Removing the back edges will eliminate the cycles. There must be an outer loop for the basic DFS algorithm to ensure every vertex, and hence every connected component, is visisted). This algorithm also identify the edges to be removed.

```
int backEdgesInComp(v)
{
    int backEdges = 0;
    v.num = counter++;
    v.status = visited;
    for each Vertex w adjacent to v
        if (w.status =/= visited){
            w.parent = v;
            backEdges+=backEdgesInComp(w);
        }
        else if (v.parent =/= w && w.num <= v.num){ //identify the back edge
          backEdges = backEdges+1;
          print(w,v) //for simplicity, just print out the edge
        }
    return backEdges;
}


int minEdgesInG(G) {
  int totalEdgesToRemove = 0;

  For each v in G {         //must visit every connected component
    if(v.status =/= visited){
      int edges = backEdgesInComp(v);
```

(Fortsettes på side 7.)

```
        totalEdgesToRemove = totalEdgesToRemove + edges;
      }
    }
    return totalEdgesToRemove;

}
```

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.

Solution: topSort O(|V|+|E|) + edge test O(|V|) (adjacency matrix O(1) for each test)

```
//modify topsort from M.A. Weiss
//the topsort part is the same from M.A. Weiss

Bool hamilstonskSti(G) {
  bool isHamiltonskSti = true;
  Queue<Vertex> q = new Queue<Vertex>();
  int counter = 0;
  Vertex [] topsort = new Vertex [|V|];

    for each Vertex v
        if(v.indegree == 0)
            q.enqueue(v);

    while(!q.isEmpty())
    {
      Vertex v = q.dequeue();
      topsort[counter++] = v;

      for each Vertex w adjacent to v
          if(--w.indegree == 0)
              q.enqueue(w);
    }

    //her comes the new part
    if(counter != |V|)
      ishamilstonskSti = false

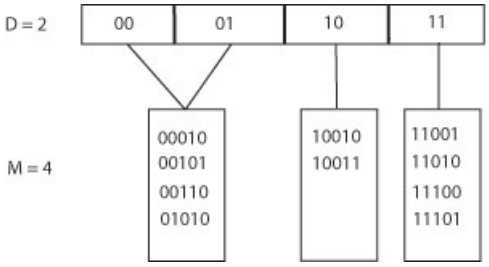
    else
    {
      for (int i;i<|V|-1; i++)
      {
        if (!(topsort[i],topsort[i+1]) in G){
          isHamilstonskSti = false;
          break;
```

(Fortsettes på side 8.)

```
            }
        }
    }
    return isHamilstonskSti;
}
```

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?

Solution: Solution should be obvious: a) splits the first block, b) splits the index and then the block accordingly.

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:

$$\begin{array}{cccccccccc} \cdots & c & d & f & c & f & v & f & c & \cdots \\ & c & d & f & c & f & c & f & c & \\ & & & & & & \uparrow & & & \end{array}$$

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

Solution:

1. good suffix shift: 7 7 7 2 7 4 7 1

0	$!c$	1
1	$!fc$	7
2	$!cfc$	4
3	$!fcfc$	7
4	$!cfcfc$	2
5	$!fcfcfc$	7
6	$!dfcfcfc$	7
7	$!cdfcfcfc$	7

2. 4 using good suffix shift as bad character shift gives 2.

5b Huffman koding (vekt 4%)

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

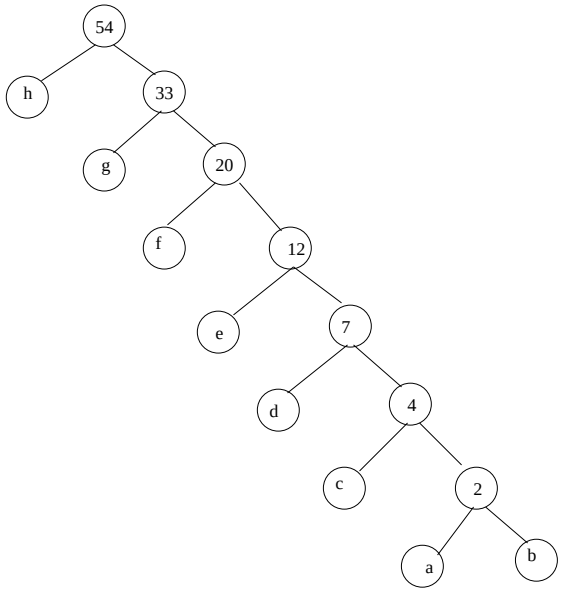
- **a:1**
- **b:1**
- **c:2**
- **d:3**
- **e:5**
- **f:8**
- **g:13**
- **h:21**

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.

Solution:

1. ```
   a = 1111110
   b = 1111111
   c = 111110
   d = 11110
   e = 1110
   f = 110
   g = 10
   h = 0
   ```

(Fortsettes på side 10.)
   ```



2. $a, 1^{n-2}0$
   $b, 1^{n-2}1$
   $c, 1^{n-3}0$
   ...
   nth alfabet,0

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--;
            }
            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

(Fortsettes på side 11.)

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 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%.

(Fortsettes på side 12.)

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"?

Solution: 2% for each sub questions

1. Big-O shows what is essential, how fast the complexity grows when n increases; exact time depends on the machine.
2. Classes reflect the complexity of problems, and what solutions are appropriate / efficient.
3. False. the proof is by diagonalization. To use reduction, we need a problem that is already proven undecidable.
4. When the disk is used for storing data, and the time to read a disk block dominates the running time.
5. Intractable problems have no efficient algorithms; while in principle solvable, the exact solutions take forbiddingly long time (years, or centuries) in the worst case

(Fortsettes på side 13.)