Løsnings forslag i java In115, Våren 1998

Oppgave 1

```
 // Inne i en eller annen klasse
private char S[];
private int pardybde;
private int n;

public void lagAlle(int i)
{
  if (i==n) brukS();
  else
    {
      /* Sjekker om det er plass til flere stjerner og man i tillegg
       * får plass til de sluttparantesene det er påkrevd å ha. */
      if (pardybde <= n-i-1)
        {
          S[i] = '*';
          lagAlle(i+1);
        }

      // Avslutter evt parantes og fortsetter rekursivt utover i arrayen.
      if (pardybde > 0)
        {
          S[i] = ')';
          pardybde--;
          lagAlle(i+1);
          pardybde++;
        }

      // Sjekker om det er plass til ett nytt parantes par i arrayen
      if (pardybde+1 <= n-i-1)
        {
          S[i] = '(';
          pardybde++;
          lagAlle(i+1);
          pardybde--;
        }
    }
}

 // Kallet som setter det hele igang...
...
lagAlle(0);
```

Oppgave 2a

Vi ser på binære trær, satt sammen av noder på formen:

```
class Node
{
  Node vsub, hsub    // Høyre og venstre subtre
  ...                // Evt andre variable
```

```
}
```

Ønsker her å traversere treet infiks og sende med nodens dybde +1 til nodens barn. Starter med kallet på roten med dybde null.

```
public void DIsekvens(Node n, int d)
{
  if (n==null) return;  // Returnerer dersom noden er null
  DIsekvens(n.vsub, d+1);
  System.out.print(d + " ");
  DIsekvens(n.rsub, d+1);
}

// Det hele starter med kallet: (som det ikke var spurt om)
DIsekvens(root, 0);
```

Oppgave 2b

Indreprosedyrer finnes ikke i Java, så vi løser det heller med å la variablene ligge globalt i objektet og lar hovedmetoden være konstruktør og den indre metoden en metode i objektet.

Ideen med metoden under er å bygge subtrærne ved å dele sekvensen inn i subtrær etter den laveste dybden, nemmelig roten, altså indeksen med verdi 0. Deretter har vi to områder som hver har en ener i seg, som vi deler ved eneren, og slik forsetter vi innover. En standard Divide and Conquer metode som til slutt har bygget treet via rekusrive kall.

```
public class ByggTre
{
  private int DI[];
  private Node root;

  public ByggTre(int di[])
  {
    DI = di;
    root = rekBygg(0, DI.length-1, 0);
  }

  public Node rekBygg(int fra, int til, int d)
  {
    Node rn = new Node();

    if (fra > til) return null;

    boolean done = false;
    int i=fra-1;
    while(++i<=til && !done)
      if (DI[i] == d) done = true;

    if (done)
      {
        rn.vsub = rekBygg(fra, i-1, d+1);
        rn.hsub = rekBygg(i+1, til, d+1);
      }
    else  // Feil tilfelle, ikke spurt om..
      throw new Exception("Feil i sekvensen");

    return rn;
  }
}
```

Oppgave 2c

Analysen ligner mye på den for QuickSort, i og med at det fra ett kall som behandler et visst segment gjøres rekusive kall som tar seg av de to "halvdelene" (ikke nødvendigvis like store) av dette segmentet, og at behandlingen i hver proseedyre er proposjonal med lengden av det feltet den skal behandle. Vi får da at et rimelig balansert tre vil gi oss $O(n \log n)$. Er treet venstre tungt derimot, må vi lete gjennom hele feltet i hvert kall på rekBygg, som medfører at vi får lete lengder:
$n, n-1, n-2, ..., 3, 2, 1$.
Denne summen summerer opp til $O(n^2)$

Oppgave 2d

Denne varianten arbeider rekursivt, slik at den gjør lesning i innfiks rekkefølge etter som treet bygges. Videre er det slik at når vi skaper en node A så vil vi ha lest nøyaktig så langt i DI som frem til den "venstereste" noden i subtreet med rot A. Vi vil altså ofte skape noden lenge før vi har lest frem til denne noden i DI sekvensen. Prosedyren kan gå som følger:

```
// Inne i en klasse som forrige oppgave
  private int DI[];
  private int i;  // for posisjon

  public Node rekBygg2(int d)
  {
    Node n = new Node();

    if (DI[i] > d)   // Venstre subtre ikke tomt
      n.vsub = rekBygg2(d+1);

    if (DI[i] != d)  // Test, faller naturlig, men ikke krevet
      throw new Exception (" Feil ");

    i++;
    if (i <=DI.length && DI[i] > d )
      n.hsub = rekBygg2(d+1);

    return n;
  }
```

Det som gjøres i hver av nodene, utenom de rekursive kallene, er av tid *O(1)*. Den totale tiden blir da *O(n)*

Oppgave 3a

Rosa tilsvarer fargekode 3, altså har vi følgende:
Graf 1: (1, 2, 3), (1, 4, 3), (2, 4, 3)
Graf 2: Node 3
Graf 3: (5, 6, 3)
Fargen er da denne er en del av definisjonen. Noden 3 er en graf ut ifra definisjone om at en node er en subgraf med farge f dersom det er en singel node som ikke har kanter med fargen f koblet til seg.

Oppgave 3b

Vi skal her programmere en metode:

```
public void fargeSubGrafer(Node G[], int f, int N)
```

som finner samtlige fargesubgrafer i G. Man kan anta at man i tillegg har en metode:

```
public void skrivKant(Kant k)
```

Nok en gang kommer vi over tilfellet der oppgaven benytter seg av indre klasser, og dette løser vi ved å legge de aktuelle variablene ut på klassenivå.

```
// Definisjon av klassene node og kant
class Node
{
  int merke;
  Kant nabo1;
}

class Kant
```

```
{
  int nr, nabo, farge;
  Kant neste;
}

class FargeSubGrafer
{
  Node G[];
  int f, N;

  /* Går gjennom alle fargenekantene til denne noden, og dersom
   * fargen ikke finnes er dette en Singel node og altså en
   * gyldig subfargegraf. */
  public boolean erSingel(Node n)
  {
    boolean funnet = false;
    Kant hj = n.nabo1;
    while ( hj != null && !funnet )
      {
        if ( hj.farge == f ) funnet = true;
        hj = hj.neste;
      }
    return !funnet;
  }

  /* Bruker dybde først søk for å finne fargegrafene. Merk at
   * at man her like gjerne kunne brukt ett bredde først...
   */
  public dfsFarge(Node u, Node fra)
  {
    Kant hj;
    if ( u.merke == 0 )
      {
        u.merke = 1;
        hj = n.nabo1;
        while (hj != null)
          {
            if ( G[hj.nabo] != fra &&
                 hj.farge == f &&
                 hj.nabo.merke != 2 )
              {
                skrivKant(hj);
                dfsFarge(G[hj.nabo], u);
              }
            hj = hj.neste;
          }
      }
    u.merke = 2;
  }

  public FargeSubGrafer(Node G_[], int f_, int N_)
  {
    G = G_;
    f = f_;
    N = N_;

    int k=0;

    for (int i=0; i<N; i++)
      if ( G[i].merke == 0 )
        {
```

```
            k++;
            System.out.print("Subgraf nr " + k + ": ");
            if ( erSingel(G[i] ) System.out.print(i);
            else dfsFarge(G[i]), null);
          }
    }
}
```

Oppgave 3c, d, e

Dette stoffet er ikke lenger pensum i kurset...

Oppgave 4

Dette problemet med kabel A blir identisk med å løse Traveling Salesman. Det er del av pensum å vite at dette er NP-komplett, og at slike problemer neppe kan løses i polynomisk tid. Dermed er uttalelsen til Ola gal.

Problemet med å legge kabel B slik at minst mulig kabel blir brukt blir imidlertid identisk med å finne et minimalt spenntre mellom husene. For dette problemet finnes to algoritmer, Prim og Kruskal. Om man bruker en proioretskø med kanter i kø bruker disse algoritmene tid $O(E \log n)$. Siden vårt problem har antall kanter som er $O(n^2)$, gir dette tid $O(n^2 \log n)$, og det er mer enn Kari påstår.

Når man er mange kanter er det imidlertid en annen enklere utgave av Prim, en som ikke bruker prioritetskø, men holder nodene i randa i en enkel liste. Denne får tid $O(n^2)$, og ut fra det er det helt rimelig påstand Kari kommer med. Noen tilsvarende utgave av Kruskal (med tid $O(n^2)$) finnes ikke.