Løsnings forslag i java In115, Våren 1999

Oppgave 1a

Input sekvensen er:

```
9, 3, 1, 3, 4, 5, 1, 6, 4, 1, 2
```

Etter sortering av det første, midterste og siste elementet, har vi følgende:

```
2, 3, 1, 3, 4, 1, 1, 6, 4, 1, 9
```

Pivot-elementet er derfor 5, Etter at vi skuler det har vi:

```
2, 3, 1, 3, 4, 1, 1, 6, 4, 5, 9
```

Den første og eneste ombyttingen (swap) i denne iterasjonen er mellom 6 og 4:

```
2, 3, 1, 3, 4, 1, 1, 4, 6, 5, 9
```

Itersjonen avsluttes med ombytting av elementene 6 og pivot-elementet:

```
2, 3, 1, 3, 4, 1, 1, 4, 5, 6, 9
```

Subsekvensen S_1 er da `2, 3, 1, 3, 4, 1, 1, 4` og subsekvensen S_2 er `6, 9`. Siden vi har *cutoff* lik 3, dvs at vi ikke burker Quicksort på sekvenser som har tre eller færre elementer, overlater vi subsekvensen S_2 til siste del av algoritmen, der vi ordner de små subsekvensene ved hjelp av f.eks innstikk sortering. Algoritmen fortsetter da rekursivt på subsekvensen S_1. Første, midterste og siste element er allerede sortert, og pivot elementet er 3. Etter at vi skjuler pivoten, har vi følgende:

```
2, 3, 1, 1, 4, 1, 3, 4
```

Ombytting på 3 og 1 gir:

```
2, 1, 1, 1, 4, 3, 3, 4
```

Ombytting av 4 og 3 gir:

```
2, 1, 1, 1, 3, 4, 3, 4
```

Nå er subsekvensene S_1: `2, 1, 1, 1`, og S_2 :`4, 3, 4`. S_2 har tre elementer og er på grunn av *cutoff* lik 3 gjør vi ikke noe mere med den nå. Vi fortsetter derimot med subsekvensen S_1. Etter sortering av de første, midterste og siste elemetet, har vi:

```
1, 1, 1, 2
```

Pivot elementet er da 1 og partisjoneringen forandrer ikke noe. Resultatet blir da:

```
1, 1, 1, 2, 3, 4, 3, 4, 5, 6, 9
```

Som sorteres ferdig med ett gjennomløp av instikksortering.

Oppgave 1b

Prosessen er akkurat den samme som i 1a (på grunn av *cutoff* mister vi i 1a hele tiden subsekvensen S_2, dvs at Quicksort alltid fortsatte på bare en av sekvensene, akkurat som Quickselect gjør. Etter partisjoneringen av siste sekvens `1, 1, 1, 2`, er S_1 lik `1, 1` og S_2 lik `2`. $k=|S_1|+1$ og derfor er det tredje minste elementet lik pivotelementet, dvs lik 1.

Hvis man tolker oppgaven slik at like store elementer skal ignoreres, vil den enkleste løsningen muligens være følgende:

- Kjør Quickselect som skal finne det minste elementet i sekvensen
- Så fjerner man alle kopier av dette elementet og kjører algoritmen igjen for å finne det neste minste elementet.
- Man fjerner alle kopier av det nest minste elementet, og finner så det tredje minste elementet.

Dette kan selvfølgelig forbedres slik at man bare behøver å kjøre Quickselect en gang.

Oppgave 2

Den naturlige løsningen er å gå gjennom `Mengde`-tabellen i en dobbel løkke. For hver *i*-verdi i den ytre løkka bruker vi en indre løkke løkke til å gå bakover i `Mengde`-tabellen intill vi enten er ved starten av tabellen eller har funnet en vinter med mer snø enn vinter *i*.

```
double Mengde[];
int S[];

public void beregnS1()
{
  // Går gjennom alle i lista
  for (int i=1; i<Mengde.length; i++)
    {
      int j = i-1;
      //Kryss sjekker mot alle de tidligere
      while ( j>=0 && M[j]<=M[i] ) j--;

      if ( j<0 ) S[i] = -1;
      else S[i] = i-j;
    }
}
```

Tidsforbruket blir $O(n^2)$ i verste tilfellet(som er at indre while løkka går fra *i* tilbake til 0 hver gang uten å finne noen vintre som har mere snø den denne.

Oppgave 2b

Ideen er å bruke stack'en slik at vi bare trenger å gå gjennom `Mengde`-tabellen en gang. Det viser set at det ikke er nødvendig eksplisistt å bruke `H`-tabellen angitt i oppgaveteksten, istedet legger vi `H[i]`-verdier direkte på stack'en uten første å putte dem inn i `H`-tabellen.

Vi antar at vi har en klasse `IntStack` med operasjonene: `push, pop, top, isEmpty`.

For hvert steg i forløkka popper vi fra stakc'en inntil stack'en er tom eller `Mengde[stack.top()] > Mengde[i]`, dvs inntil vi har en verdi (årstall) på toppen av stack'en som representerer den forrige vinteren med mer snø enn vinter *i*. Da vet vi at `S[i]` skal være lik vinteren på toppen av stack'en eller -1 dersom stacken er tom. Til slutt dyttes i på stacken.

Poenget er at `H`-verdiene som fjernes fra stack'en trygt kan kastes fordi den nye vinteren som vi legger på stack'en, vinter *i*, har minst like mye snø som vintrene vi fjernet. Vintrene som kommer etter *i* har aldri bruk for de vintrene før *i* som har mindre snø enn vinter *i*.

I slutten av hver runde i forløkke gjelder følgende innvariant: Verdiene som ligger på stack'en er (fra topp til bunn): `i, H[i], H[H[i]], H[H[H[i]]], ...`. Det nederste årstallet på stack'en tilsvarer den vinteren som har hatt mest snø av de vintrene vi har behandlet så langt (altså tom vinter i).

```
double Mengde[];
int S[];

public void beregnS2()
{
  IntStack stack = new IntStack();

  for (int i=0; i<Mengde.length; i++)
    {
      /* Så lenge det er elementer på stacken og det gjeldende elementet
       * er større en det på toppen av stacken, altså høyeste tidligere.
       */
      while( !stack.isEmpty() && Mengde[i] > Mengde[stack.top] )
        stack.pop();
      if (stack.isEmpty()) S[i] = -1;
      else S[i] = i-stack.top();
      stack.push(i);
    }
}
```

Metoden over bruker lineær tid, fordi stack operasjonene tar konstant tid, og hver av de N vintrene legges på stack'en maksimalt 1 gang. Det totale arbeidet vi utfører i de N rundene i for-løkka blir dermed også lineært.

Oppgave 3a

Starten av sekvensen som tilsvarer dobbeltorder traversering er gitt som:
$A_1\, B_1\, D_1\, H_1\, H_2\, D_2\, B_2\, E_1\, E_2\, I_1\, I_2$

Fullføringen av sekvensen blir da:
$A_2\, C_1\, F_1\, F_2\, C_2\, G_1\, J_1\, J_2\, G_2\, K_1\, K_2$

Hvis vi fjerner alle nodene med indeks 2 traverserer vi treet i det som tilsvarer preorder (prefix) traversering. Fjerner vi nodene med indeks 1 får vi inorder (infix) traversering.

Oppgave 3b

Løser dette ved en kombinasjon av prefix og infix traversering av treet.

```
class Node
{
  String navn;
  Node hsub, vsub;
  boolean hsref;
}

public void dobbeltOrder(Node t)
{
  if (t != null)
    {
      // Skrive oss selv ut i prefix
      System.out.print(t.navn + "1 ");
      dobbeltOrder(t.vsub);

      // Skrive oss selv ut i infix
      System.out.print(t.navn + "2 ");
      dobbeltOrder(t.hsub);
    }
}
```

Oppgave 3c

Metoden lar seg ikke implementere hvis vi bruker representasjonen som er gitt fordi vi ikke har mulighet til å komme oppover i treet! Hvis vi f.eks har kallet `neste("D", 2)` skal vi returnere Node "B", men node D kjenner bare til seg selv og noden "H".

Med den modifiserte trestrukturen blir `neste` prosedyren veldig enkel: Dersom d=1, returner vi venstre subtre, hvis et slikt finnes, i motsatt fall returnerer vi node P selv. Dersom d=2 returnerer vi noden i høyre subtre.

```
public Node neste(Node P, int d)
{
  if (d==1)
    {
      if ( P.vsub!=null ) return P.vsub;
      else return P;
    }
  else return P.hsub;
}
```

Oppgave 3d

Det er flere måter å løse oppgaven på. En mulighet er å bruke en stack. Vi antar da at vi har en klasse `NodeStack` med vanlige operasjoner. Vi starter med å opprette en ny stack, og kaller deretter en hjelpemetode `rekSettDO` med roten som parameter. Denne metoden modifiserer treet rekursivt.

Metoden er en modifisering av traverseing en i 3b:

- I stedet for å skrive ut noden første gang, dyttes den på stack'en.
- I stedet for å skrive ut noden andre gang, fjernes den fra stack'en (noden er ferdig behandlet).
- Dersom noden har et høyre barn, settes `hsref` lik `false` og vi modifiserer høyre barn rekursivt.
- Dersom noden ikke har et høyre barn, settes `hsref` lik `true` og vi setter `hsub` lik toppnoden på stack'en (eller null dersom stack'en er tom).

```
NodeStack stack;

public void settDobbeltOrder(Node rot)
{
  stack = new NodeStack();
  rekSettDO(rot);
}

public void rekSettDO(Node t)
{
  if ( t!=null )
    {
      stack.push(t);
      rekSettDO(t.vsub);
      stack.pop();

      if ( t.hsub!=null )
        {
          t.hsref = false;
          rekSettDO(t.hsub);
        }
      else
        {
          t.hsref = true;
          t.hsub = (stack.isEmpty())?null:stack.top();
        }
    }
}
```

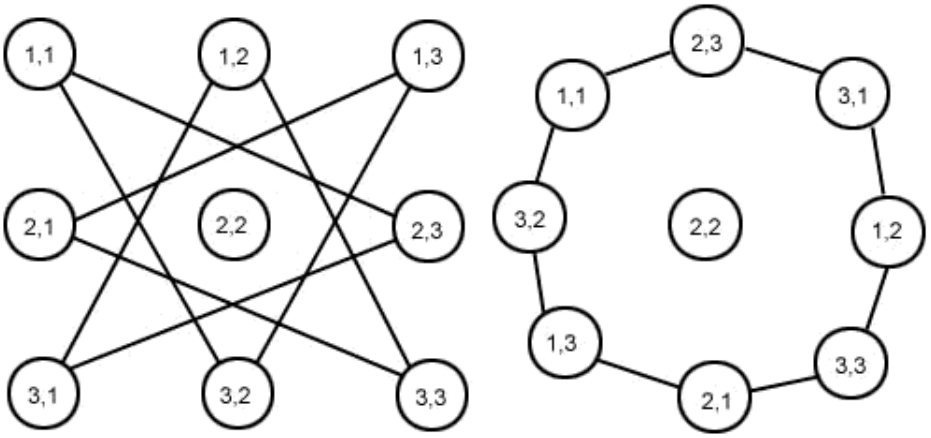
En alterativ løsning baserer seg på at vi sender med neste node som parameter i det rekursive kallet. Det korrekte blir da å sende seg selv som parameter til venstre subtre, og neste-verdien som vi fikk av foreldren som parameter til høyre subtre.

```
public void settDobbeltOrder(Node rot)
{
  rekSettDO2(rot, null);
}

public void rekSettDO2(Node t, Node neste)
{
  if ( t!=null )
    {
      rekSettDO2(t.vsub, t);
      if ( t.hsub!=null )
        {
          t.hsref = false;
          rekSettDO2(t.hsub, neste);
        }
      else
        {
          t.hsref = true;
          t.hsub = neste;
        }
    }
}
```

Oppgave 4a

Det er to måter å tegne grafen på. Det er verdt å merke at hver kant egentlig er en kant i hver retning, altså to kanter per kant. De doble kantene er droppet for at ikke figuren skulle bli for rotete.



Oppgave 4b

Denne oppgaven tilsvarer problemet korteste vei, en til alle, for en urettet graf, som vi løser med ett bredde først søk. Algoritmen blir effektiv ved å bruke en kø.

Den eneste forskjellen fra en standard korteste vei, en til alle, er at vi egentlig gjør en alle til en, men siden en kant fra A til B impliserer at det obså er en kant fra B til A (hvis en springer kan flytte fra A til B, kan en springer også flytte fra B til A), er ikke det noe problem.

Vi starter altså med sluttstillingen og arbeider oss vekk fra sluttfeltet i bredde først manér.

```
class SjakkNode
{
  Felt springer;
  int merke;
  SjakkNode utkanter[] = new SjakkNode[8];
}

class Felt
{
  int xPos, yPos;
  Felt(int x, int y) { xPos = x; yPos = y; }
}

int avstand[][];
Felt besteTrekk[][];
int N; // Brett størrelsen

public void genererTabeller(SjakkNode s)
{
  NodeQueue q;
  SjakkNode sn;

  avstand[s.springer.xPos][s.springer.yPos] = 0;

  q = new NodeQueue();
  q.enqueue(s);

  while( !q.isEmpty() )
    {
      sn = q.dequeue();

      for(int i=0; i<8 && sn.utkanter[i]!=null; i++)
        {
          int x = sn.utkanter[i].springer.xPos;
          int y = sn.utkanter[i].springer.yPos;

          // Dersom vi har en ubehandlet node.
          if( avstand[x][y] == Integer.MAX_VALUE )
            {
              // Oppdatere med foreløpig lengde + 1
              avstand[x][y] =
                avstand[sn.springer.xPos][sn.springer.yPos] + 1;

              // Oppdatere beste trekk med denne noden.
              besteTrekk[x][y] = sn.springer;

              // Legge denne utkanten inn paa koen
              q.enqueue(sn.utkanter[i]);
            }
        }
    }
}
```

Oppgave 4c

Tidsforbruket til algoritmen i punkt b blir $O(N^2)$ fordi hver av de N^2 nodene og hver av de maksimalt $8N^2$ kantene blir behandlet 1 gang.

Oppgave 4d

Ideen er å la metoden selv beregne de inntil 8 utkantene, tilsvarende de inntil 8 mulige springer trekkene fra ett felt.

```
int avstand[][];
Felt besteTrekk[][];
int N; // Brett størrelsen

public void genererTabeller(int u, int v)
{
  FeltQueue q = new FeltQueue();
  Felt f;
  avstand[u][v] = 0;

  q.enqueue(new Felt(u, v));

  while( !q.isEmpty() )
    {
      f = q.dequeue();

      // Trekk 1: To nedover og en til høyre
      if ( f.yPos+2<N && f.xPos+1<N &&
           avstand[f.xPos+1][f.yPos+2] == Integer.MAX_VALUE )
        {
          /* Oppdaterer strekning og vei, samt legger denne noden
           * inn på køen til å bli behandlet senere. */
          avstand[f.xPos+1][f.yPos+2] = avstand[f.xPos][f.yPos] + 1;
          besteTrekk[f.xPos+1][f.yPos+2] = f;
          q.enqueue(new Felt(f.xPos+1, f.yPos+2));
        }

      // Trekk 2: To nedover og en til venstre
      if ( f.yPos+2<N && f.xPos-1>=0 &&
           avstand[f.xPos-1][f.yPos+2] == Integer.MAX_VALUE )
         ...

      // Trekk 3: To oppover og en til høyre
      if ( f.yPos-2>=0 && f.xPos+1<N &&
           avstand[f.xPos+1][f.yPos-2] == Integer.MAX_VALUE )
         ...

      // Trekk 4: To oppover og en til venstre
      if ( f.yPos-2>=0 && f.xPos-1>=0 &&
           avstand[f.xPos-1][f.yPos-2] == Integer.MAX_VALUE )
         ...

      // Trekk 5: En nedover og to til høyre
      if ( f.yPos+1<N && f.xPos+2<N &&
           avstand[f.xPos+2][f.yPos+1] == Integer.MAX_VALUE )
         ...

      // Trekk 6: En oppover og to til høyre
      if ( f.yPos-1>=0 && f.xPos+1<N &&
           avstand[f.xPos+2][f.yPos-1] == Integer.MAX_VALUE )
         ...

      // Trekk 7: En nedover og to til venstre
      if ( f.yPos+1<N && f.xPos-2>=0 &&
```

```
            avstand[f.xPos-2][f.yPos+1] == Integer.MAX_VALUE )
          ...

        // Trekk 8: En oppover og to til venstre
        if ( f.yPos-1>=0 && f.xPos-2>=0 &&
            avstand[f.xPos-2][f.yPos-1] == Integer.MAX_VALUE )
          ...
    }
}
```

Oppgave 4e

En springer kan maksimalt gå til 8 forskjellige felter i ett trekk. For å finne ut hvilket av disse 8 trekkene som er best (= raskeste vei til sluttfeltet), behøver programmet bare slå opp i avstand tabellen for hver av de 8 feltene, og finne den minste verdien. Streng tatt trenger programmet bare å lete inntil den finner et felt med avstand en mindre enn avstanden for feltet som springeren nå står på.

Det er opplagt at dette kun gir konstant økning i tidsforbruket, fordi det er en konstant antall mulige felter å undersøke.