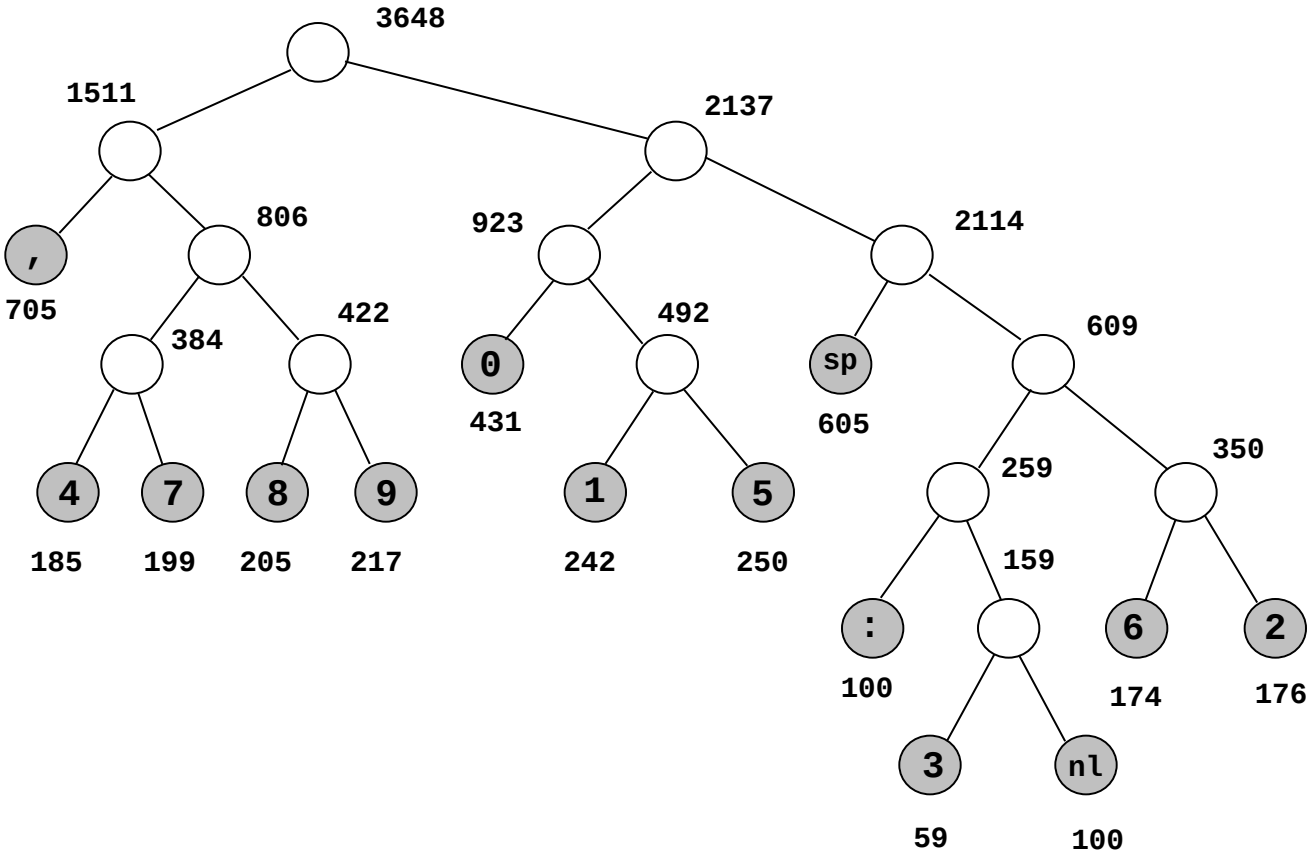
Opgaveløsninger
(sæt 8)

Opgave 26: 12.1 (2 point)



Opgave 27: Simulering (4 point, ikke-obligatorisk)

```
import simulation.event.*;
import java.util.*;

public class ModemSim extends Simulation {
    int freeModems;
    double avgCallLen;
    long freqOfCalls;
    Random r;
    int userNum = 0;

    ModemSim(int modems, double avgLen, long callIntrvl) {
        freeModems = modems;
        avgCallLen = avgLen;
        freqOfCalls = callIntrvl;
        r = new Random();
        new DialIn().schedule(0);
        runSimulation(20);
    }

    class DialIn extends Event {
        public void actions() {
            System.out.print("User " + userNum +
                             " dials at time " + time() + " ");
            if (freeModems > 0) {
                freeModems--;
                long howLong = r.poisson(avgCallLen);
                System.out.println("and connects for " +
                                   howLong + " minutes");
                new HangUp(userNum).schedule(time() + howLong);
            } else
                System.out.println("but gets busy signal");
            userNum++;
            new DialIn().schedule(time() + freqOfCalls);
        }
    }


    class HangUp extends Event {
        int userNum;
        HangUp(int user) { userNum = user; }

        public void actions() {
            freeModems++;
            System.out.println("User " + userNum +
                               " hangs up at time " + time());
        }
    }

    public static void main(String[] args) {
        new ModemSim(3, 5.0, 1);
    }
}
```

Opgave 28: Møntproblemet (4 point, ikke-obligatorisk)

Antag at tilstandene i problemet (stillingerne) er repræsenteret ved objekter af klassen `State`. Vi kan da finde en kortest mulig løsning af problemet på følgende måde. Først indsættes starttilstanden i en tom kø. Så længe køen ikke er tom, eller en løsningstilstand ikke er fundet, udtages den første tilstand fra køen, hvorefter alle tilstande, der kan fremkomme ved udførelse af et lovligt træk i stillingen, indsættes bagest i køen.

En foreløbig udgave af algoritmen i hovedprogrammet ser således ud:

```
State first, last;
first = last = new State();
while (first != null) {
    for (int coin = 0; coin <= 3; coin++)
        for (int dir = -1; dir <= 1; dir += 2)
            if (first.isLegalMove(coin, dir)) {
                State suc = new State(first, coin, dir);
                if (suc.isSolution()) {
                    suc.print();
                    return;
                }
                last = last.next = suc;
             }
    first = first.next
}
System.out.println("No solution");
```

Køens første og sidste tilstand udpeges ved referencerne `first` og `last`. Hver tilstand antages at have en reference `next`, der peger på den efterfølgende tilstand i køen (`null`, hvis tilstanden står sidst).

En træk angives ved et talpar (`coin`, `dir`), hvor `coin` identificerer mønten (`0` = 20-krone, `1` = 10-krone, `2` = 1-krone og `3` = 25-øre), og `dir` angiver trækkets retning (`-1` = til venstre, `1` = til højre).

Hvis et træk er lovligt dannes den stilling, `suc`, der fremkommer ved at udføre trækket i den aktuelle stilling, `first`. Hvis `suc` er en løsningstilstand, udskrives løsningen ved at kalde dens `print`-metode; ellers indsættes `suc` bagest i køen.

Tilbage er nu blot at specificere klassen `State`, således at det er muligt (1) at afgøre, om et givet træk (`coin`, `dir`) er lovligt i en tilstand (`isLegalMove`), og (2) at udskrive tilstanden og dens forgængertilstande (`print`).

Nedenfor ses en foreløbig udgave af klassen:

```
class State {
    int[] square = {0, 1, 2, 3};
    State next, prev;

    State() {}

    State(State prev, int coin, int dir) {
        square = (int[]) prev.square.clone();
        square[coin] += dir;
        this.prev = prev;
    }

    boolean isLegalMove(int coin, int dir) {
        int from = square[coin], to = from + dir;
        if (to < 0 || to > 4)
            return false;
        for (int c = coin + 1; c <= 3; c++)
            if (square[c] == from || square[c] == to)
                return false;
        return true;
    }

    boolean isSolution() {
        return square[0] == 4 && square[1] == 3 &&
               square[2] == 2 && square[3] == 1;
    }

    void print() {
        if (prev != null)
            prev.print();
        for (int coin = 0; coin <= 3; coin++)
            System.out.print(square[coin] + " ");
        System.out.println();
    }
}
```

En tilstand indeholder i arrayet `square` en angivelse af på hvilke felter, de 4 mønter ligger. F.eks. ligger mønt nummer `0` (20-kronen) på feltet `square[0]` .
Referencen `next` peger på det efterfølgende element i køen.

Referencen `prev` peger på den umiddelbart foregående tilstand (dvs. tilstanden før udførelsen af trækket, der førte til den pågældende stilling). Ved hjælp af denne reference er det muligt at udskrive løsningsvejen fra starttilstand til sluttilstand. Metoden `print` udskriver (ved brug af rekursion) tilstandene i rækkefølge fra start til slut.

Klassen indeholder to konstruktører. Den parameterløse konstruktør benyttes til at skabe en starttilstand, mens den anden benyttes til at skabe den tilstand den fremkommer ved at udføre et træk (`coin`,`dir`) i en eksisterende tilstand, `prev`.

Metoden `isLegalMove` kontrollerer om et givet træk er lovligt, mens metoden `isSolution` undersøger, om der er tale om en løsningstilstand.

Det her udviklede program er principielt i stand til at løse den stillede opgave. Men hvis man kører programmet, vil man opdage, at det, selv efter en rum tid, ikke standser med et resultat. Eventuelt udskrives en meddelelse om, at programmet ikke har mere disponibel lagerplads. Årsagen er, at der skal skabes mange tilstande, før løsningen findes.

Men det er let at indse, at mange af disse tilstande vil være identiske. Der er derfor behov for en metode til at kontrollere, om en identisk tilstand tidligere er blevet skabt. En simpel metode til at opnå dette er følgende. Nummerer tilstandene, således at enhver tilstand har sit eget unikke nummer. Lad en tabel indeholde en angivelse af, hvilke numre, der er brugt (dvs. hvilke tilstande, der tidlige er skabt). At afgøre om en tilstand er skabt tidligere vil således kunne afgøres ved et simpelt opslag i tabellen.

En simpel nummerering af tilstandene kan foretages ud fra arrayet `square`. Vi forsyner klassen `State` med følgende metode, der beregner en tilstands unikke nummer.

```
int number() {
    int sum = 0, factor = 1;
    for (int coin = 0; coin <= 3; coin++, factor *= 5)
        sum += factor * square[coin];
    return sum;
}
```

Tabellen kan realiseres simpelt ved hjælp af Javas klasse `BitSet`. I hovedprogrammet oprettes et `BitSet` på følgende måde.

```
BitSet seen = new BitSet();
```

Derudover erstattes koden

```
State suc = new State(first,coin,dir);
if (suc.isSolution()) {
    suc.print();
    return;
}
last = last.next = suc;
```

i hovedprogrammet med

```
State suc = new State(first,coin,dir);
if (!seen.get(suc.number())) {
    if (suc.isSolution()) {
        suc.print();
        return;
    }
    last = last.next = suc;
    seen.set(suc.number());
}
```

Med disse få tilføjelser bestemmer programmet meget hurtigt en løsning bestående af 22 træk.

Udskriften bliver følgende:

```
0 1 2 3
0 1 2 4
0 1 3 4
0 2 3 4
1 2 3 4
1 1 3 4
1 0 3 4
2 0 3 4
2 1 3 4
2 1 2 4
2 1 1 4
3 1 1 4
3 1 0 4
3 2 0 4
3 2 1 4
3 2 1 3
3 2 1 2
4 2 1 2
4 2 1 1
4 3 1 1
4 3 1 0
4 3 2 0
4 3 2 1
```

På en 400 MHz Macintosh PowerBook G3 blev køretiden målt til 0.03 sekunder.

En komplet udskrift af hele programmet følger nedenfor.

```
import java.util.BitSet;

class State {
    int[] square = {0, 1, 2, 3};
    State next, prev;

    State() {}

    State(State prev, int coin, int dir) {
        square = (int[]) prev.square.clone();
        square[coin] += dir;
        this.prev = prev;
    }

    int number() {
        int sum = 0, factor = 1;
        for (int coin = 0; coin <= 3; coin++, factor *= 5)
            sum += factor * square[coin];
        return sum;
    }

    boolean isLegalMove(int coin, int dir) {
        int from = square[coin], to = from + dir;
        if (to < 0 || to > 4)
            return false;
        for (int c = coin + 1; c <= 3; c++)
            if (square[c] == from || square[c] == to)
                return false;
        return true;
    }

    boolean isSolution() {
        return square[0] == 4 && square[1] == 3 &&
               square[2] == 2 && square[3] == 1;
    }

    void print() {
        if (prev != null)
            prev.print();
        for (int coin = 0; coin <= 3; coin++)
            System.out.print(square[coin] + " ");
        System.out.println();
    }
}
```

```
public class CoinPuzzle {
    public static void main(String[] args) {
        BitSet seen = new BitSet();
        State first, last;
        first = last = new State();
        while (first != null) {
            for (int coin = 0; coin <= 3; coin++)
                for (int dir = -1; dir <= 1; dir += 2)
                    if (first.isLegalMove(coin, dir)) {
                        State suc = new State(first, coin, dir);
                        if (!seen.get(suc.number())) {
                            if (suc.isSolution()) {
                                suc.print();
                                return;
                            }
                            last = last.next = suc;
                            seen.set(suc.number());
                        }
                    }
             first = first.next;
        }
        System.out.println("No solution");
    }
}
```