INF 4130 Oppgavesett 3, 20/09-2011 m/løsningsforslag

Oppgave 1

1.1 Løs oppgave 20.19 (B&P), (a) er vist på forelesningen og kan vel bare repeteres, men løs (b).

(a) er altså løst på forelesningen.

(b) Siden det er to løkker inne i hverandre (se foilene), så blir tiden O(n*m).

1.2 Kjør algoritmen med noen enkle eksempler, f.eks. "algori" og "logari", og med to helt like ord.

```
        l o g a r i
      0 1 2 3 4 5 6
     --------------
  0 | 0 1 2 3 4 5 6 <- Initialisering
a 1 | 1 1 2 3 3 4 5
l 2 | 2 1 2 3 4 4 5
g 3 | 3 2 2 2 3 4 5
o 4 | 4 3 2 3 3 4 5
r 5 | 5 4 3 3 4 3 4
i 6 | 6 5 4 4 4 4 3
     --------------
      ^
      Initialisering
```

```
        l i k e
      0 1 2 3 4
     ----------
  0 | 0 1 2 3 4
l 1 | 1 0 1 2 3
i 2 | 2 1 0 1 2
k 3 | 3 2 1 0 1
e 4 | 4 3 2 1 0
     ----------
```

1.3 Vis at man kan programmere algoritmen slik at man bare tar plass til en kolonne (eller rad), samt en enkel-variabel.

Vi skal altså beregne arrayen som den er satt opp over, og vi antar at den er indeksert $D[0{:}m,0{:}n]$. Vi indekserer den $D[i,j]$, og altså interessert i verdien $D[m,n]$.

Vi beregner en og en kolonne fra venstre, og bruker i programmet en array *DK*[0:*m*] som initialiseres med 0, 1, 2, ..., *m*. Denne arrayen skal underveis i *DK*[0:*i*] inneholde verdier fra kolonne *j*, og i *DK*[*i*+1:*m*] inneholde verdier fra kolonne *j*-1.

Vi må også ha to variable "nyDij" og "forrige" (vi klarer oss altså ikke med *en* variabel, slik oppgaven foreslo). De aktuelle verdiene er uthevet med fet eller kursiv i tabellen over. Programmet blir da slik:

```
for i = 0 to n do { DK[i] = i; }    // Initialisering av DK
forrige = 0;                        // Generelt: verdien av D[i-1, j-1]
for j = 1 to n do {
   DK[0] = j;                       // Initialisering av nullte linje
   for i = 1 to m do {
      nyDij = if P[i] == T[j] then forrige
              else min(DK[i], forrige, DK[i-1]);
      forrige = DK[i];
      DK[i] = nyDij;
   }
}
```

Svaret ligger i DK[m].

Oppgave 2

2.1 Løs oppgave 20.20 (B&P) Hint: Forsøk å gjøre lure forandringer i initialiseringen.

2.2 Kjør algoritmen med noen eksempler, f.eks.: P="hvis" og T="haiehus" med K=2, og "lag" og "varelager" med K=1. (Haiehuset er selvfølgelig der hvor hushaiene bor……)

Trikset er å initialisere D[0, 0:n] (altså nullte linje) med bare nuller (i stedet for 0, 1, 2, ..., n). Det vil bety at når vi begynner å sammenlikne P med T et stykke ut i T, så vil vi starte med editeringsavstand 0. Ellers blir utfyllingen nøyaktig som før.

```
        h a i e h u s
      0 1 2 3 4 5 6 7
     ---------------
  0  | 0 0 0 0 0 0 0 0 <- Ny initialisering
h 1  | 1 0 1 1 1 0 1 1
v 2  | 2 1 1 2 2 1 1 2
i 3  | 3 2 2 2 3 2 2 2
s 4  | 4 3 3 3 3 3 3 2
     ---------------
```

```
          v a r e l a g e r
        0 1 2 3 4 5 6 7 8 9
      ---------------------
    0 | 0 0 0 0 0 0 0 0 0 0
l   1 | 1 1 1 1 1 0 1 1 1 1
a   2 | 2 2 1 2 2 1 0 1 1 1
g   3 | 3 3 2 2 3 2 1 0 1 2
      ---------------------
```

I tillegg til den opplagte løsningen med editeringsavstand = 0, blir det altså to løsninger som slutter hhv. en foran og en bak denne, som begge har ed.avstand = 1.

Oppgave 3

Vi skal løse pengevekslerens problem med dynamisk programmering: Problemet består i å gi igjen K øre med så få mynter som mulig. La $\{v_1, v_2, \ldots, v_n\}$, være myentenes verdier. Vi kan f.eks ha: $v_1 = 25$, $v_2 = 10$, $v_3 = 5$, $v_4 = 1$. Vi antar verdiene er heltallige og at $v_{i-1} < v_i$. Videre antar vi at $v_n = 1$, slik at det alltid finnes en løsning. Og selvfølgelig antar vi at vi har uendelig med vekslepenger av alle aktuelle verdier.

For pengesystemet beskrevet over vil en grådig strategi virke. (Og for interesserte kan det jo være en passelig utfordring å bevise nettopp det.) MEN hvis vi fjerner femøringen, havner vi i trøbbel: da vil en grådig algoritme som skal gi igjen 30 øre gi en 25-øring og fem 1-øringer, seks mynter totalt, mens det optimale er å gi tre 10-øringer. Dynamisk programmering finner en optimal løsning for alle pengesystemer, også de hvor en grådig algoritme feiler.

La $C[j]$ være det antall mynter i en løsning som gir igjen j øre. Hvis en løsning bruker en mynt med verdi v_i, vil $C[j] = C[j - v_i] + 1$.

a) Sett opp en rekursiv formel for verdien av en optimal løsning.

Vi må prøve alle mulige mynter for å finne hvilken som er best. (Vi kan ikke bare ta den største, som i de fleste pengesystemer.) Formelen blir altså:

$$C[j] = \begin{cases} 0 & \text{hvis } j = 0 \\ \min\limits_{1 \le i < n} C[j - v_i] + 1 & \text{hvis } j \ge 1 \end{cases}$$

b) Skriv (pseudo)kode for en algoritme basert på formelen i a).

```
Veksl(K, V, n)        // K er summen, V[i:n] myntverdiene
{
   C[0] = 0
   for j = 1 to K do
   {
      C[j] = MAXINT                                        // uendelig
      for i = 1 to n do
         if j >= V[i] and C[j - V[i]] + 1 < C[j] then     // unngår index
            C[j] = C[j - V[i]] + 1                        // under null
   }
   return C
}
```

Merk at vi her bare finner antall mynter… Ønsker vi å vite hvilke, må vi bruke en ekstra tabell *CV* og sette *CV* [*j*] = *V* [*i*] samtidig som vi oppdaterer *C* [*j*] i den innerste FOR-løkka.

c) Kjør algoritmen med $v_1 = 30$, $v_2 = 24$, $v_3 = 12$, $v_4 = 6$, $v_5 = 3$, $v_6 = 1$, og K = 48. (Så gæernt var det faktisk i England for ikke lenge siden!) [Grådig-løsningen er for øvrig 30+12+6.]

Vi setter opp tabellen *C* slik den utvikler seg stegvis under kjøring (for de første stegene):

0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**…**
0	1																		
0	1	2																	
0	1	2	1																
0	1	2	1	2															
0	1	2	1	2	3														
0	1	2	1	2	3	1													

Det interessante (svaret) er det det vil stå i ruten for 48. Vi kan ta to tjuefirere (toshilling eller florin). Vi får en fra C[24], og en til i C[48].

d) Hva blir kjøretiden til algoritmen?

Kjøretiden avhenger som alltid av FOR-løkkene vi går i. (Her er det litt mer FOR-løkke enn tabell, ettersom vi går igjennom alle mynter for hver pengesum.) Kjøretiden blir $O(Kn)$, eller riktigere $\Theta(Kn)$.

Oppgave 4 (om man har tid)

Se på det å bruke memoisering, altså at man har en tabell som forvanlig dyn. progr., men at man skriver algoritmen rekursivt ut fra utfyllings-formlen. Trikset er da at hvert kall da først slår opp i tabellen, og om svaret er beregnet tidligere ligger det her og kan bare leveres. Om det ikke er beregnet gjøres det nå, og man både setter ned og leverer svaret.

a) Skriv en slik algoritme for spøsmålet i oppgave 20.19.

Arrayen *D*[0:*m*,0:*n*] er akkurat som i oppgave 20.19, og den kan greiest initilariseres som det gjøres der (men dette kan i stedet lett bygges inn i den rekursive funksjonen). Resten av arryen initialiseres til -1 (siden 0 har signifikant betydning). Den rekursive funksjonen kan passelig legges inne i funksjonen EditDistance (selv om det vel ikke går i Java), slik at den har direkte tilgang til *D*, *T* og *P*.

```
function EdDist(i,j): int { // Den må kalles utenfra med (m,n)
    if D[i,j] >= 0 then {return D[i,j];}
    if P[i] == T[j] then {
        D[i,j] = EdDist[i-1,j-1]
    } else {
        D[i,j] = min(EdDist[i-1,j],EdDist[i-1,j-1],EdDist[i,j-1]);
    }
    return D[i,j];
}
```

Merk at rekursjonen alltid stopper på grunn av initialiseringen.

b) Kjør den pr hånd på noen enkle eksempler (f.eks. de angitt i oppgave 20.19 over), og se hvor mange verdier du slipper å beregne.

Bergningen starter altså i D[6,6], og følgende verdier blir beregnet (der det er tomt vil det stå en urørt -1).

```
        l o g a r i
      0 1 2 3 4 5 6
     --------------
  0 | 0 1 2 3 4 5 6
a 1 | 1 1 2 3 3
l 2 | 2 1 2 3 4
g 3 | 3 2 2 2 3
o 4 | 4 3 2 3 3
r 5 | 5         3
i 6 | 6           3
     --------------
```

[slutt]