

Løsningsforslag til kapittel 15 i  
*Introduction to Algorithms* (Cormen et al.)

Til bruk i emnet TDT4120 Algoritmer og Datastrukturer

Åsmund Eldhuset

Inneholder løsninger av følgende oppgaver:

**15.1:** 1 til 5

**15.2:** 2

**15.3:** 4

**15:** 4

# Kapittel 15

## Seksjon 15.1 - Treningsoppgaver

### 15.1-1

Hver stasjon registrerer hvor man kom fra, så derfor kan vi bare spore oss bakover. Men en rekursiv funksjon som starter bakfra og "graver" seg forover, vil til slutt måtte begynne å nøste seg ut igjen. Dette kan vi utnytte ved at vi skriver ut den gjeldende stasjonen *etter* at funksjonen har kalt seg selv rekursivt. Merk at  $l_i[j]$  betyr "linjen man kom fra hvis man står i stasjon  $j$  på linje  $i$ ". Vi holder oss til notasjonen boka bruker, men i et ekte programmeringsspråk ville dette ha vært et todimensjonalt array, og man ville ha skrevet  $l[i][j]$ .

```
PRINT-STATIONS-RECURSIVE(station, line)
  if station = 0
    return
  PRINT-STATIONS-RECURSIVE(station - 1, l_line[station])
  print "line " line ", station " station
```

Denne kan så startes slik (den linjen man kom ut på til slutt, er angitt av  $l^*$ ):

```
PRINT-STATIONS-RECURSIVE(n, l*)
```

### 15.1-2

I følge (15.8) og (15.9) har alltid  $r_1$  og  $r_2$  samme verdi, så da er  $r_i(j) = r_1(j) = r_2(j)$ . Dermed ser vi av (15.9) at

$$r_i(j) = r_1(j+1) + r_2(j+1) = r_i(j+1) + r_i(j+1) = 2r_i(j+1) \quad (1)$$

Når vi har fått oppgitt at  $r_i(j) = 2^{n-j}$ , er det enkelt å bruke substitusjonsmetoden. Vi setter bare dette inn i (15.8) og (1) og ser om det stemmer:

$$r_1(n) = r_2(n) = r_i(n) = 2^{n-n} = 2^0 = 1$$

og

$$\begin{aligned} r_i(j) &= 2r_i(j+1) \\ 2^{n-j} &= 2 \cdot 2^{n-(j+1)} = 2^{1+n-j-1} = 2^{n-j} \end{aligned}$$

### 15.1-3

Hvilket summetegn som står innerst har ingenting å si, så

$$\sum_{i=1}^2 \sum_{j=1}^n r_i(j) = \sum_{j=1}^n \sum_{i=1}^2 r_i(j)$$

Siden  $r_1(j) = r_2(j) = r_i(j) = 2^{n-j}$ , er

$$\sum_{i=1}^2 r_i(j) = 2r_i(j) = 2 \cdot 2^{n-j}$$

Dermed er

$$\sum_{j=1}^n \sum_{i=1}^2 r_i(j) = \sum_{j=1}^n (2 \cdot 2^{n-j}) = 2 \sum_{j=1}^n 2^{n-j}$$

Vi vil nå bytte summasjonsvariabel. Vi lar  $k = n - j$ , og siden  $j$  går fra 1 til  $n$ , vil  $k$  gå fra  $n - 1$  til 0. Hvilken vei vi går når vi summerer, har ikke noe å si, så da er

$$2 \sum_{j=1}^n 2^{n-j} = 2 \sum_{k=0}^{n-1} 2^k = 2(2^n - 1) = 2^{n+1} - 2$$

#### 15.1-4

Du er nødt til å spare på hele  $l$ -tabellen (som består av  $2n$  elementer), for du vet ikke hvilken vei som ender opp med å være best. Derimot kan vi observere at *utregningen av  $f_i[j]$  bare avhenger av  $f_i[j - 1]$*  (pluss  $a$ - og  $t$ -tabellene, men de er vi ikke blitt bedt om å røre). Dermed trenger vi ikke  $f$ -tabellen, men vi kan klare oss med to variabler som angir tidsforbruket frem til den forrige stasjonen på hver av de to linjene. Plassforbruket er dermed  $2n + 2$ .

*Dette er en svært vanlig plassoptimalisering i dynamisk programmering*, for ofte avhenger utregningen av nye verdier kun av noen få av de tidligere verdiene.

#### 15.1-5

Av linjene 4-8 i FASTEST-WAY ser vi at hvis  $l_1[j] = 2$ , kan det bare ha skjedd hvis

$$\begin{aligned} f_1[j - 1] + a_{1,j} &> f_2[j - 1] + t_{2,j-1} + a_{1,j} \\ f_1[j - 1] &> f_2[j - 1] + t_{2,j-1}. \end{aligned} \tag{2}$$

Tilsvarende ser vi av linjene 9-13 at hvis  $l_2[j] = 1$ , må

$$\begin{aligned} f_2[j - 1] + a_{2,j} &> f_1[j - 1] + t_{1,j-1} + a_{2,j} \\ f_2[j - 1] &> f_1[j - 1] + t_{1,j-1}. \end{aligned} \tag{3}$$

Innsetting av (3) i (2) gir

$$f_1[j - 1] > f_2[j - 1] + t_{2,j-1} > f_1[j - 1] + t_{1,j-1} + t_{2,j-1}$$

og dermed

$$0 > t_{1,j-1} + t_{2,j-1}$$

som åpenbart er umulig siden ingen  $t$ -verdier er negative. Altså kan ikke  $l_1[j] = 2$  og  $l_2[j] = 1$  for noen  $j$ .

## Seksjon 15.2 - Treningsoppgaver

### 15.2-2

MATRIX-CHAIN-MULTIPLY( $A, s, i, j$ )

```

if  $i + 1 = j$ 
    return  $A_i \times A_j$ 
return MATRIX-CHAIN-MULTIPLY( $A, s, i, s[i, j]$ )
    × MATRIX-CHAIN-MULTIPLY( $A, s, s[i, j] + 1, j$ )

```

## Seksjon 15.3 - Treningsoppgaver

### 15.3-4

På side 327 er sammenhengen mellom de forskjellige stasjonene forklart. Vi ser at korteste vei gjennom stasjon  $S_{1,j-1}$  og korteste vei gjennom stasjon  $S_{2,j-1}$  begge kan inngå som en del av korteste vei gjennom både stasjon  $S_{1,j}$  og  $S_{2,j}$ . Dermed har vi overlappende delproblemer.

## Kapittel 15 - Oppgaver

### 15-4

For hver person kan enten den personen inviteres eller ikke. Hvis personen inviteres, kan ingen av hans direkte “undersåtter” inviteres. Hvis personen ikke inviteres, kan man velge om hver enkelt “undersått” skal inviteres eller ikke. Vi kan definere delproblemene som “maksimal convivality som kan oppnås ved å invitere person  $x$  sine undersåtter, gitt hvorvidt person  $x$  er invitert eller ikke”.

I koden tenker vi oss at **person** er en node i treet (hver node

```
MAX-CONVIVALITY(person, isInvited)
  if results[person, isInvited]  $\neq$  nil
    return results[person, isInvited]
  currentChild  $\leftarrow$  leftChild[person]
  convivality  $\leftarrow$  0
  while currentChild  $\neq$  nil
    if isInvited
      convivality  $\leftarrow$  convivality + MAX-CONVIVALITY(currentChild, false)
    else
      convivality  $\leftarrow$  convivality + max(MAX-CONVIVALITY(currentChild, false),
        MAX-CONVIVALITY(currentChild, true))
      currentChild = nextSibling[currentChild]
  results[person, isInvited]  $\leftarrow$  convivality
  return convivality
```