Negende college algoritmiek

7 april 2016

Dynamisch Programmeren

- nuttig bij problemen met *overlappende deelproblemen*
- druk een oplossing van het probleem uit in oplossingen van deelproblemen (*recursieve formulering*)
- deeloplossingen worden opgeslagen in een *tabel* zodra ze berekend zijn, waardoor elk deelprobleem maar *één keer* hoeft te worden opgelost
- na afloop bevat (of is) de tabel de oplossing van het oorspronkelijke probleem
- DP is van oorsprong een *bottom up* methode: start met de kleine gevallen en combineer hun oplossingen tot oplossingen van steeds grotere gevallen
- er is ook een *top down* variant (*memory function*)

Uit college 8: DP: bottom up $\leftrightarrow$ top down

- de bottom up methode is *iteratief*, de top down variant is recursief

- bottom up lost *alle* deelproblemen op, top down alleen degene die echt nodig zijn voor het oplossen van het oorspronkelijke probleem

- bij beide varianten wordt eenzelfde soort tabel gebruikt

- bij bottom up wordt de tabel in een *bepaalde volgorde* gevuld, bij top down gebeurt dat meer willekeurig

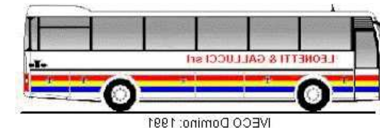
- bij de bottom up manier is vaak een qua geheugengebruik *efficiënter* algoritme af te leiden

We willen een busreis maken langs steden $0, 1, 2, \ldots, n$, in die volgorde. Aangezien meerdere busmaatschappijen op de verschillende (deel)trajecten rijden, zijn de prijzen voor een rit van plaats i naar plaats j (via alle tussenliggende steden) per bus verschillend. Het kan dus voordeliger zijn om in plaats van rechtstreeks met de goedkoopste bus van plaats 0 naar n te reizen, (een paar keer) over te stappen en met een andere bus verder te gaan.

1915: 18 BL

1939: 626 RNL

1959: 309



Laat $\texttt{prijs}[i][j]$, de prijs van het goedkoopste buskaartje rechtstreeks van i naar j, gegeven zijn voor alle $i \leq j$. Het probleem is nu om de prijs van de goedkoopste reis van 0 naar n te vinden.

Laat kosten(n) de prijs van de goedkoopste busreis van 0 naar n voorstellen, langs alle tussenliggende steden (in oplopende volgorde). Dan geldt:

$$\text{kosten}(n) = \begin{cases} 0 & \text{als } n = 0 \\ \min_{0 \leq k < n}(\text{kosten}(k) + \text{prijs}[k][n]) & \text{als } n \geq 1 \end{cases}$$

Voorbeeld:

$$\text{prijs} = \begin{pmatrix} 0 & 5 & 10 & 15 \\ - & 0 & 7 & 13 \\ - & - & 0 & 4 \\ - & - & - & 0 \end{pmatrix}$$

De prijs van de goedkoopste busreis van 0 naar 3 is hier. . .

Laat kosten(n) de prijs van de goedkoopste busreis van 0 naar n voorstellen, langs alle tussenliggende steden (in oplopende volgorde). Dan geldt:

$$\text{kosten}(n) = \begin{cases} 0 & \text{als } n = 0 \\ \min_{0 \leq k < n}(\text{kosten}(k) + \text{prijs}[k][n]) & \text{als } n \geq 1 \end{cases}$$

Voorbeeld:

$$\text{prijs} = \begin{pmatrix} 0 & 5 & 10 & 15 \\ - & 0 & 7 & 13 \\ - & - & 0 & 4 \\ - & - & - & 0 \end{pmatrix}$$

De prijs van de goedkoopste busreis van 0 naar 3 is hier 14 (met tussenstop in plaats 2).

Een recursief algoritme:

```
kosten(n)::
    if n=0 then
       return 0;
    else
       temp := prijs[0][n];   // k = 0
       for k := 1 to n-1 do
           hulp := kosten(k) + prijs[k][n];
           if hulp < temp then
               temp := hulp;
           fi
       od
       return temp;
    fi .
```

De recursieve oplossing doet exponentieel veel aanroepen, en er is heel veel overlap tussen de deelproblemen. Oplossing: deeloplossingen opslaan in een geschikt array.

Laat kosten[i] de prijs van de goedkoopste busreis van 0 naar i voorstellen, langs alle tussenliggende steden (in oplopende volgorde). We zoeken dus kosten[n].
Dan geldt:

$$\text{kosten}[i] = \begin{cases} 0 & \text{als } i = 0 \\ \min_{0 \le k < i}(\text{kosten}[k] + \text{prijs}[k][i]) & \text{als } i \ge 1 \end{cases}$$

We gaan het array nu bottom up vullen. Merk op dat om kosten[i] te berekenen, *alle* kleinere waarden kosten[k] met $k < i$ nodig zijn. Die moeten dus al eerder berekend zijn. We moeten het array derhalve van links naar rechts vullen.

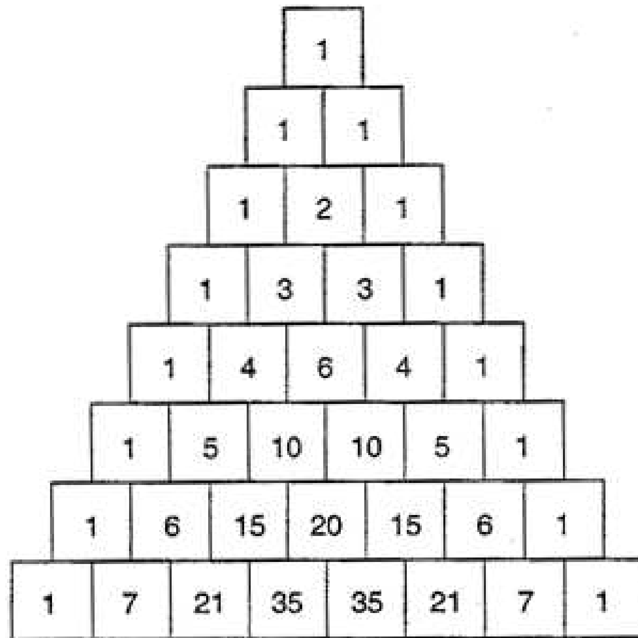
```
kosten[0] := 0;
for i := 1 to n do
    temp := prijs[0][i]; // met 1 bus, zonder overstappen
    for k := 1 to i − 1 do
        hulp := kosten[k]+prijs[k][i];
        if hulp < temp then
            temp := hulp; // goedkoopste tot dusver
        fi
    od
    kosten[i] := temp;
od
return kosten[n];
```

Het algoritme is eenvoudig zo aan te passen dat ook de tussenstops van de goedkoopste reis worden gevonden.

```
kosten[0] := 0; stop[0] := 0;
for i := 1 to n do
    temp := prijs[0][i]; tempstop:= 0; // met 1 bus
    for k := 1 to i − 1 do
        hulp := kosten[k] + prijs[k][i];
        if hulp < temp then
            temp := hulp; // goedkoopste tot dusver
            tempstop := k; // bijbehorende tussenstop
        fi
    od
    kosten[i] := temp; stop[i] := tempstop;
od
return kosten[n];
```

Hierin is stop[i] steeds de laatste tussenstop op de goedkoopste reis van 0 naar i.

Gegeven een pot met n verschillende objecten.
Doe in één keer een greep van k objecten uit de pot.
Hoeveel mogelijkheden?



Driehoek van Pascal

$$C(n,k) = \binom{n}{k} = \begin{cases} \binom{n-1}{k-1} + \binom{n-1}{k} & 0 < k < n \\ 1 & k = 0, n \end{cases}$$

Een recursief algoritme ligt voor de hand:

```
int bin1(int n,int k) {
    if ( ( k == 0 ) || ( k == n ) )
        return 1;
    else
        return ( bin1(n-1,k-1) + bin1(n-1,k) );
}
```

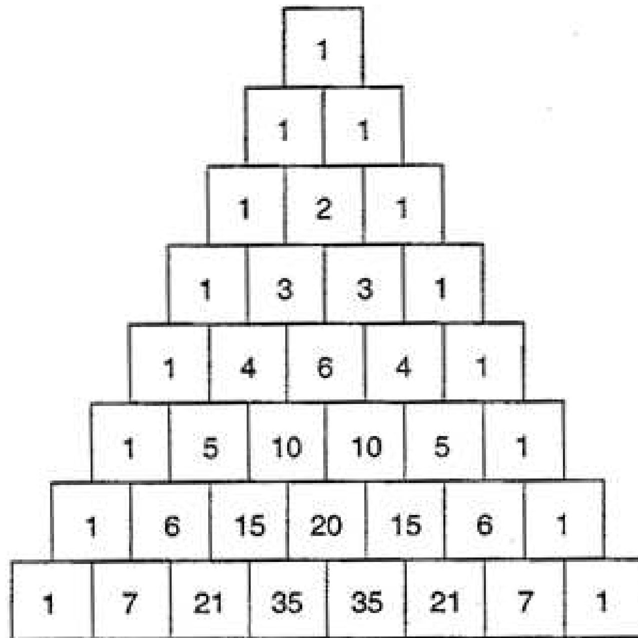
Veel van de `bin1(i,j)`'s worden echter herhaald berekend.

Aanroep: `bin1(n,k)`.
Complexiteit: $O(\binom{n}{k})$

Recursief algoritme met een array `C` voor het opslaan van tussenresultaten (dus `C[i][j]` $= \binom{i}{j}$):

```
int bin2(int n,int k) {
// C is globaal (foei) en op nul geinitialiseerd
    if ( C[n][k] != 0 )  // reeds eerder berekend
        return C[n][k];
    else {
        if ( ( k == 0 ) || ( k == n ) ) {
            C[n][k] = 1;
        else
            C[n][k] =  bin2(n-1,k-1) + bin2(n-1,k);
        return C[n][k];
    }
}
```

Complexiteit: $O(n * k)$; extra geheugen: $\Theta(n * k)$



Driehoek van Pascal

$$C(n,k) = \binom{n}{k} = \begin{cases} \binom{n-1}{k-1} + \binom{n-1}{k} & 0 < k < n \\ 1 & k = 0, n \end{cases}$$

We gebruiken een globaal (op nul geïnitialiseerd) array `C` voor het opslaan van tussenresultaten, dus `C[i][j]` $= \binom{i}{j}$.

	0	1	2	...	$k-1$	k
0	1					
1	1	1				
2	1	2	1			
$\vdots$						
k	1					1
$\vdots$						
$n-1$	1				$C(n-1, k-1)$	$C(n-1, k)$
n	1					$C(n, k)$

Bottom up:
Het array wordt rij voor rij gevuld, te beginnen bij rij 0, en per rij van links naar rechts, gebruikmakend van de recurrente betrekking (recursieve formulering).

```
int bin3(int n,int k) {
    for ( i = 0; i <= n; i++ )
        for ( j = 0; j <= min(i,k); j++ )
            if ( ( j == 0 ) || ( j == i ) )
                C[i][j] = 1;
            else
                C[i][j] = C[i-1][j-1] + C[i-1][j];
    return C[n][k];
}
```

Aanroep: `bin3(n,k)`.
Complexiteit: $\Theta(n * k)$; extra geheugen: $\Theta(n * k)$.
We kunnen hier echter volstaan met een een-dimensionaal array ter lengte k. Er is dus maar $O(k)$ extra geheugen nodig. (Zie ook exercise 8.1.9)

Knapzakprobleem

Gegeven n objecten, met gewicht $w_1, \ldots, w_n$ en waarde $v_1, \ldots, v_n$, en een knapzak met capaciteit W. **Gevraagd**: de meest waardevolle deelverzameling der objecten die in de knapzak past (dus met totaalgewicht $\leq W$). **Aanname**: gewichten zijn integers > 0.

Voorbeeld:

object	gewicht	waarde
1	8	42
2	3	14
3	4	40
4	5	27

knapzakcapaciteit 12

Laat $F[i][j]$ de waarde zijn van de meest waardevolle deelverzameling van de eerste i $(1 \leq i \leq n)$ objecten, die in een knapzak met capaciteit j $(1 \leq j \leq W)$ past. We zoeken dus $F[n][W]$. We nemen hier impliciet aan dat W een positief geheel getal is.

Laat $F[i][j]$ de waarde zijn van de meest waardevolle deelverzameling van de eerste i ($1 \leq i \leq n$) objecten, die in een knapzak met capaciteit j ($1 \leq j \leq W$) past. We zoeken dus $F[n][W]$. We nemen hier impliciet aan dat W een positief geheel getal is.

Dan geldt (want object i zit er wel of niet in):

$$F[i][j] = \begin{cases} \max\{F[i-1][j], v_i + F[i-1][j-w_i]\} & \text{als } j \geq w_i \\ F[i-1][j] & \text{als } j < w_i \end{cases}$$

En we definiëren:

$$F[0][j] = 0 \text{ voor } j \geq 0 \text{ en } F[i][0] = 0 \text{ voor } i \geq 0$$

We kunnen het array bijvoorbeeld rij voor rij (en per rij v.l.n.r.) vullen.

for $i := 0$ **to** n **do**
  for $j := 0$ **to** W **do**
    if $i = 0$ **or** $j = 0$ **then**
      $F[i][j] := 0$;
    else
      if $j < w_i$ **then**
        $F[i][j] := F[i-1][j]$;
      else
        $F[i][j] :=$ max $(F[i-1][j], v_i + F[i-1][j-w_i])$;
fi fi od od

		0	$j - w_i$	j	W
	0	0	0	0	0
	$i-1$	0	$F[i-1][j-w_i]$	$F[i-1][j]$	
w_i, v_i	i	0		$F[i][j]$	
	n	0			goal

Complexiteit: $\Theta(n * W)$; extra geheugen: $\Theta(n * W)$

KZP met DP — tabel vullen

Voor het voorbeeld wordt de tabel als volgt gevuld:

		j	→											
		0	1	2	3	4	5	6	7	8	9	10	11	12
i	0	0	0	0	0	0	0	0	0	0	0	0	0	0
↓	1	0	0	0	0	0	0	0	0	42	42	42	42	42
	2	0	0	0	14	14	14	14	14	42	42	42	56	56
	3	0	0	0	14	40	40	40	54	54	54	54	?	
	4													

Voor het voorbeeld wordt de tabel als volgt gevuld:

		j	→											
		0	1	2	3	4	5	6	7	8	9	10	11	12
i	0	0	0	0	0	0	0	0	0	0	0	0	0	0
↓	1	0	0	0	0	0	0	0	0	42	42	42	42	42
	2	0	0	0	14	14	14	14	14	42	42	42	56	56
	3	0	0	0	14	40	40	40	54	54	54	54	?	
	4													

		j	→											
		0	1	2	3	4	5	6	7	8	9	10	11	12
i	0	0	0	0	0	0	0	0	0	0	0	0	0	0
↓	1	0	0	0	0	0	0	0	0	42	42	42	42	42
	2	0	0	0	14	14	14	14	14	42	42	42	56	56
	3	0	0	0	14	40	40	40	54	54	54	54	56	82
	4	0	0	0	14	40	40	40	54	54	?			

		j	→											
		0	1	2	3	4	5	6	7	8	9	10	11	12
i	0	0	0	0	0	0	0	0	0	0	0	0	0	0
↓	1	0	0	0	0	0	0	0	0	42	42	42	42	42
	2	0	0	0	14	14	14	14	14	42	42	42	56	56
	3	0	0	0	14	40	40	40	54	54	54	54	56	82
	4	0	0	0	14	40	40	40	54	54	67	67	67	82

Dus de gevraagde optimale waarde is 82.

Opmerkingen:

1. Je kunt volstaan met een eendimensionaal hulparray; deze moet dan wel v.r.n.l. worden gevuld.

2. Uit de tweedimensionale tabel kun je de/een optimale deelverzameling zelf ook terugvinden.

De (maar in het algemeen: een) meest waardevolle deelverzameling vinden we terug door te beginnen bij $F[n][W]$ en van daaruit terug te redeneren.

		j	→											
		0	1	2	3	4	5	6	7	8	9	10	11	12
i	0	0	0	0	0	0	0	0	0	0	0	0	0	0
↓	1	0	0	0	0	0	0	0	0	42	42	42	42	42
	2	0	0	0	14	14	14	14	14	42	42	42	56	56
	3	0	0	0	14	40	40	40	54	54	54	54	56	82
	4	0	0	0	14	40	40	40	54	54	67	67	67	82

4 niet, 3 wel, 2 niet, 1 wel, dus $\{1, 3\}$ is de optimale deelverzameling.

Ter herinnering:

object	gewicht	waarde
1	8	42
2	3	14
3	4	40
4	5	27

```
RecKnapzak(i, j) :: // F[i][j] == -1: nog niet berekend
    if ( F[i][j] >= 0 ) then return F[i][j];
    else
        if ( i = 0 or j = 0 ) then F[i][j] := 0;
        else
            if ( j < w_i ) then
                F[i][j] := RecKnapzak(i-1, j);
            else
                F[i][j] := max { RecKnapzak(i-1, j),
                             v_i + RecKnapzak(i-1, j-w_i) };
            fi
        fi
        return F[i][j];
    fi
```

Vraag: welke van de twee methodes verdient de voorkeur?

Werkwijze en opmerkingen

1. Druk de (waarde van de) oplossing van het probleem uit in (de waarde van) oplossingen van deelproblemen

2. Stel een recurrente betrekking op (recursieve formulering)

3. Gebruik alleen dynamisch programmeren bij overlappende deelproblemen

4. Definieer een geschikte tabel en ga na wat de berekeningsvolgorde moet zijn

5. Vul aldus bottom up de tabel in (algoritme)

6. Let op geheugenbesparing

7. Pas je algoritme zo aan dat je uit de tabel niet alleen een waarde maar ook de (optimale) oplossing zelf kunt halen

8. Dynamisch programmeren wordt vaak gebruikt voor optimalisatieproblemen

Gegeven onbeperkt veel munten van $d_1, d_2, \ldots d_m$ eurocent, en een te betalen bedrag van n ($n \geq 0$) eurocent. Alle d_i zijn > 0 en verschillend. **Gevraagd**: het minimale aantal munten dat nodig is om het bedrag van n eurocent te betalen.

Voorbeeld:

type munt	waarde
1	1
2	4
3	6

te betalen bedrag: 8

Vier manieren om te betalen: 6 + 1 + 1; 4 + 4; 4 + 1 + 1 + 1 + 1; 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1. Dus het gevraagde minimale aantal is: 2 (twee munten van 4 cent).

KZP	MP
n objecten	m munten
gewicht w_i	waarde d_i
totaal gewicht $\leq$ capaciteit W	totale waarde $=$ bedrag n
waarde v_i	'kosten' 1
max. totale waarde	min. totale 'kosten'
elk object $\leq$ 1 keer	munt mag meer keer

Laat $\texttt{munt}[i][j]$ het minimale aantal munten zijn dat nodig is om een bedrag van j eurocent te betalen, wanneer alleen munten van $d_1, d_2, \ldots, d_i$ $(i \geq 1)$ worden gebruikt. We zoeken dus $\texttt{munt}[m][n]$.

Dan geldt (want d_i wordt wel of niet gebruikt):

$$\texttt{munt}[i][j] = \begin{cases} \ldots & \text{als } i > 1, j \geq d_i \\ \ldots & \text{als } i > 1, 0 < j < d_i \\ \ldots & \text{als } i = 1, 0 < j < d_1 \\ \ldots & \text{als } i = 1, j \geq d_1 \\ \ldots & \text{als } i \geq 1, j = 0 \end{cases}$$

Laat $\texttt{munt}[i][j]$ het minimale aantal munten zijn dat nodig is om een bedrag van j eurocent te betalen, wanneer alleen munten van $d_1, d_2, \ldots, d_i$ $(i \geq 1)$ worden gebruikt. We zoeken dus $\texttt{munt}[m][n]$.

Dan geldt (want d_i wordt wel of niet gebruikt):

$$\texttt{munt}[i][j] = \begin{cases} \text{min } \{\texttt{munt}[i-1][j], 1 + \texttt{munt}[i][j-d_i]\} & \text{als } i > 1, j \geq d_i \\ \texttt{munt}[i-1][j] & \text{als } i > 1, 0 < j < d_i \\ \infty & \text{als } i = 1, 0 < j < d_1 \\ 1 + \texttt{munt}[1][j-d_1] & \text{als } i = 1, j \geq d_1 \\ 0 & \text{als } i \geq 1, j = 0 \end{cases}$$

Iets andere formulering recurrente betrekkingen:

$$F[i][j] = \begin{cases} \max\{F[i-1][j], v_i + F[i-1][j-w_i]\} & \text{als } i \geq 1, j \geq w_i \\ F[i-1][j] & \text{als } i \geq 1, j < w_i \\ 0 & \text{als } i = 0, j > 0 \\ 0 & \text{als } i \geq 0, j = 0 \end{cases}$$

$$\texttt{munt}[i][j] = \begin{cases} \min\ \{\texttt{munt}[i-1][j], 1 + \texttt{munt}[i][j-d_i]\} & \text{als } i \geq 1, j \geq d_i \\ \texttt{munt}[i-1][j] & \text{als } i \geq 1, j < d_i \\ \infty & \text{als } i = 0, j > 0 \\ 0 & \text{als } i \geq 0, j = 0 \end{cases}$$

Complexiteit MP met 2-d DP (vgl. KZP):
tijd $\Theta(m * n)$; extra geheugen: $\Theta(m * n)$
Of met eendimensionaal hulparray (v.l.n.r. vullen): $\Theta(n)$

Het kan eenvoudiger, want

KZP	MP
n objecten	m munten
gewicht w_i	waarde d_i
totaal gewicht $\leq$ capaciteit W	totale waarde $=$ bedrag n
waarde v_i	'kosten' 1
max. totale waarde	min. totale 'kosten'
elk object $\leq$ 1 keer	munt mag meer keer

Bij muntenprobleem dus geen noodzaak om bij te houden welke munten we al gebruikt hebben.

Laat $\texttt{munt}[j]$ het minimale aantal munten zijn dat nodig is om een bedrag van j eurocent te betalen. We zoeken dus $\texttt{munt}[n]$.
Neem voor het gemak even aan dat de muntsoorten oplopend zijn gesorteerd ($d_1 < d_2 < \ldots < d_m$).

Dan geldt:

$$\texttt{munt}[j] = \begin{cases} \min_{d_i \leq j} \{1 + \texttt{munt}[j - d_i]\} & \text{als } j \geq d_1 \\ \infty & \text{als } 0 < j < d_1 \\ 0 & \text{als } j = 0 \end{cases}$$

Vul array `munt` van links naar rechts.

```
munt[0] = 0;
for j := 1 to n do
    tmp := ∞;
    i := 1;
    while i ≤ m and d_i ≤ j do
        if 1 + munt[j − d_i] < tmp then
            tmp := 1 + munt[j − d_i];
        fi
    od
    munt[j] := tmp;
od
```

Complexiteit MP met 1-d DP:
tijd $\Theta(m * n)$; extra geheugen: $\Theta(n)$
Net als MP met 2-d DP (met eendimensionaal array)

Voorbeeld:

type munt	waarde
1	1
2	4
3	6

te betalen bedrag: 8

j	0	1	2	3	4	5	6	7	8
munt[j]	0	1	2	3	1	2	1	2	?

Voorbeeld:

type munt	waarde
1	1
2	4
3	6

te betalen bedrag: 8

j	0	1	2	3	4	5	6	7	8
munt[j]	0	1	2	3	1	2	1	2	?

j	0	1	2	3	4	5	6	7	8
munt[j]	0	1	2	3	1	2	1	2	2

Vind benodigde munten terug in tabel:

j	0	1	2	3	4	5	6	7	8
munt[j]	0	1	2	3	1	2	1	2	2

1. Een (eenvoudige) variatie is: gegeven een bedrag van n euro, is dat te betalen met muntsoorten $d_1, \ldots d_m$? Dit kan geheel analoog aan het optimalisatieprobleem worden opgelost met DP. Gebruik een array `munt`, waarbij `munt`$[j]$ = True als het bedrag j gemaakt kan worden, en anders False.

Vul array `munt` van links naar rechts.

```
munt[0] = 0;
for j := 1 to n do
    tmp := ∞;
    i := 1;
    while i ≤ m and d_i ≤ j do
      if 1 + munt[j − d_i] < tmp then
        tmp := 1 + munt[j − d_i];
      fi
    od
    munt[j] := tmp;
od
```

Complexiteit MP met 1-d DP:
tijd $\Theta(m * n)$; extra geheugen: $\Theta(n)$
Net als MP met 2-d DP (met eendimensionaal array)

Vul array `munt` van links naar rechts.

```
munt[0] = 0;
for j := 1 to n do
    tmp := false;
    i := 1;
    while i ≤ m and d_i ≤ j and not tmp do
        if munt[j − d_i] then
            tmp := true;
        fi
    od
    munt[j] := tmp;
od
```

Complexiteit MP met 1-d DP:

tijd $\Theta(m * n)$; extra geheugen: $\Theta(n)$

Net als MP met 2-d DP (met eendimensionaal array)

1. Een (eenvoudige) variatie is: gegeven een bedrag van n euro, is dat te betalen met muntsoorten $d_1, \ldots d_m$? Dit kan geheel analoog aan het optimalisatieprobleem worden opgelost met DP. Gebruik een array `munt`, waarbij `munt`$[j]$ = True als het bedrag j gemaakt kan worden, en anders False.

2. Een ander algoritme voor het muntenprobleem:

    betaal n met $d_1, \ldots d_m$::

    geef de grootste munt $d_i \leq n$;

    betaal $n - d_i$ met $d_1, \ldots d_i$

   Dit is een zogenaamd gretig algoritme. Bovenstaand algoritme is erg snel, maar het levert niet altijd een optimale oplossing (soms ook geen oplossing, terwijl er wel een oplossing is).

- **Lezen/leren bij dit college:**
  Scan binomiaal coëfficiënten; sheets; paragraaf 8.2;
  voorbeeld 2 in paragraaf 8.1

- **Werkcollege:**
  donderdag 7 april 2016, 13:45–15:30, in zaal Bezuidenhout

- daarna: assistentie opdracht 2, in zaal Paleistuin

- **Opgaven:**
  zie `http://www.liacs.leidenuniv.nl/~vlietrvan1/algoritmiek/`

- **Volgend college:**
  donderdag 14 april 2016