

Negende college algoritmiëk

13 april 2007

Dynamisch Programmeren

- nuttig bij problemen met *overlappende deelproblemen*
- druk een oplossing van het probleem uit in oplossingen van deelproblemen (*recurrente betrekking*)
- 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*)

- 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  $0, 1, 2, \dots, n$ , 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.



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

Laat  $\text{kosten}[i]$  de prijs van de goedkoopste busreis van 0 naar  $i$  voorstellen, langs alle tussenliggende steden (in oplopende volgorde). We zoeken dus  $\text{kosten}[n]$ .

Dan geldt:

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

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).

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

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

## Knapzakprobleem

**Gegeven**  $n$  objecten, met gewicht  $w_1, \dots, w_n$  en waarde  $v_1, \dots, v_n$ , 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  $V[i][j]$  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  $V[n][W]$ . We nemen hier impliciet aan dat  $W$  een positief geheel getal is.

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

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

En we definiëren:

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

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
       $V[i][j] := 0;$ 
    else
      if  $j < w_i$  then
         $V[i][j] := V[i - 1][j];$ 
      else
         $V[i][j] := \max (V[i - 1][j], v_i + V[i - 1][j - w_i]);$ 
    fi fi od
  
```

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

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

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	14	42	42	42	56	56
	3	0	0	0	14	40	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	14	42	42	42	56	56
	3	0	0	0	14	40	40	40	40	54	54	54	54	56	82
	4	0	0	0	14	40	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  $V[n][W]$  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:

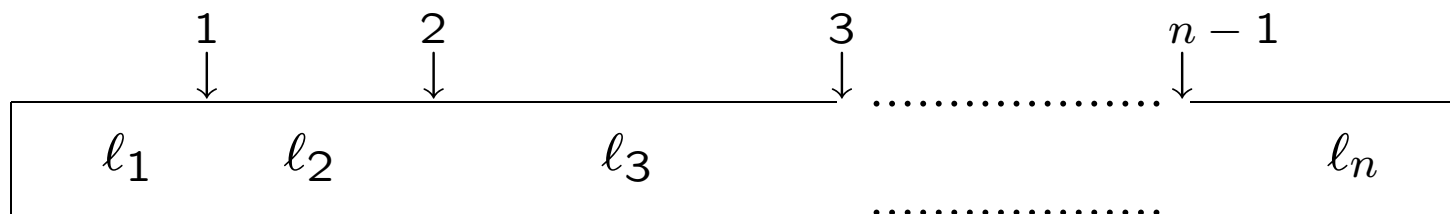
object	gewicht	waarde
1	8	42
2	3	14
3	4	40
4	5	27

```
RecKnapzak(i, j) ::
  if ( i = 0 or j = 0 ) then
    V[i][j] := 0;
  else
    if ( wi > j ) then
      V[i][j] := RecKnapzak(i-1, j);
    else
      V[i][j] := max { RecKnapzak(i-1, j),
                       vi + RecKnapzak(i-1, j-wi) };
    fi
  fi
return V[i][j];
```

Vraag: welke van de twee methodes verdient de voorkeur?

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 definitie)
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

Een houtzaagmolen reket voor het in twee stukken zagen van een stam van lengte  $\ell$  precies  $\ell$  euro, ongeacht de plek waar dit moet gebeuren. Na bestudering van de knoesten op een boomstam van lengte  $\ell$  wordt besloten dat deze in achtereenvolgens stukken van lengtes  $\ell_1, \ell_2, \dots, \ell_n$  gezaagd moeten worden. (De hele boomstam heeft dus lengte  $\sum_{i=1}^n \ell_i$ .) De plekken waar gezaagd gaat worden zijn dus van tevoren bekend. Er zijn hier  $n - 1$  zaagplekken.



Merk op dat de **volgorde van zagen** van invloed is op de prijs.

### Voorbeeld

Laat  $n = 4$  en  $l_1 = 6, l_2 = 8, l_3 = 7, l_4 = 2$ . De boomstam heeft dus lengte 23.

- Stel we zagen achtereenvolgens op plek 1, dan plek 2 en dan plek 3. De kosten zijn dan  $23 + 17 + 9 = 49$  euro.
- Stel we zagen achtereenvolgens op plek 3, dan plek 2 en dan plek 1. De kosten zijn dan  $23 + 21 + 14 = 58$  euro.

### Probleem

Bepaal de minimale kosten om de gegeven boomstam in stukken met de opgegeven lengtes  $l_i$  te zagen.

## Deelproblemen

Het probleem brengen we terug tot het bepalen van de minimale kosten  $Z[i][j]$  die moeten worden gemaakt om een stam ter lengte  $L(i, j) = l_i + l_{i+1} + \dots + l_j$  te verzagen tot achtereenvolgens stukken van lengte  $l_i, l_{i+1}, \dots, l_j$ . Alle  $l_i$ , en dus ook alle  $L(i, j)$ , zijn gegeven. Het oorspronkelijke probleem is dan het bepalen van  $Z[1][n]$ . Merk op dat altijd  $1 \leq i \leq j \leq n$ .

## Recurrente betrekking

$$Z[i][j] = \begin{cases} L(i, j) + \min_{i \leq k \leq j-1} \{Z[i][k] + Z[k+1][j]\} & \text{als } i < j \\ Z[i][i] = 0 & \end{cases}$$

De  $Z[i][j]$  op plek  $\#$  wordt berekend uit Z-waarden op de plekken met een  $*$ ; dus uit dezelfde rij en dezelfde kolom.

	j	→										
i	0	.	.	.	.	.	.	.	.	.	.	.
↓	0	0	.	.	.	.	.	.	.	.	.	.
			0	*	*	*	*	*	*	#	.	.
				0	.	.	.	.	.	*	.	.
					0	.	.	.	.	*	.	.
						0	.	.	.	*	.	.

**Invulvolgorde**

De tabel kan bottom up gevuld worden door alle diagonalen  $j = i + d$  af te lopen en per diagonaal bijvoorbeeld van linksboven tot rechtsonder te gaan. Een andere mogelijkheid is de tabel rij voor rij te vullen (van onder naar boven) en per rij (verplicht) van links naar rechts.

```
void vulkosten ( int n ) { // L en Z globaal
    int i, j;
    for (i = 1; i <= n; i++)
        Z[i][i] = 0;
    for (i = n-1; i > 0; i--) {
        for (j = i+1; j <= n; j++ ) {
            min = Z[i][i] + Z[i+1][j];
            for (k=i+1; k<j; k++) {
                if ( Z[i][k] + Z[k+1][j] < min )
                    min = Z[i][k] + Z[k+1][j];
            } // min bevat nu het minimum
            Z[i][j] = L[i][j] + min;
        } // for j
    } for i
} // vulkosten
```

**Gegeven** onbeperkt veel munten van  $d_1, d_2, \dots, d_m$  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).

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

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

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

Het array munt kan rij voor rij, en *moet* dan per rij van links naar rechts worden ingevuld, want om  $\text{munt}[i][j]$  te bepalen heb je het array-element erboven en een array-element links ervan in dezelfde rij nodig.

		j →							
		0	1	.	.	j	.	.	n
i ↓	1								
	2								
	.								
	.								
	.								
	i								
.									
m									

$i, j - d_i$                        $i - 1, j$   
 $i, j$

```
for  $i := 1$  to  $m$  do  
    munt[ $i$ ][0] := 0; od  
for  $j := 1$  to  $d_1 - 1$  do  
    munt[1][ $j$ ] :=  $\infty$ ; od  
for  $j := d_1$  to  $n$  do  
    munt[1][ $j$ ] :=  $1 + \text{munt}[1][j - d_1]$ ; od  
  
for  $i := 1$  to  $m$  do  
    for  $j := 1$  to  $d_i - 1$  do  
        munt[ $i$ ][ $j$ ] := munt[ $i - 1$ ][ $j$ ];  
    od  
    for  $j := d_i$  to  $n$  do  
        munt[ $i$ ][ $j$ ] := minimum {munt[ $i - 1$ ][ $j$ ], munt[ $i$ ][ $j - d_i$ ]};  
    od  
od
```

Voor het voorbeeld wordt de tabel als volgt gevuld:

			j →									
			0	1	2	3	4	5	6	7	8	
i	1	$d_1 = 1$	0	1	2	3	4	5	6	7	8	
↓	2	$d_2 = 4$	0	1	2	3	1	2	3	4	2	
	3	$d_3 = 6$	0	1	2	3	?					

Voor het voorbeeld wordt de tabel als volgt gevuld:

			j →								
			0	1	2	3	4	5	6	7	8
i	1	$d_1 = 1$	0	1	2	3	4	5	6	7	8
↓	2	$d_2 = 4$	0	1	2	3	1	2	3	4	2
	3	$d_3 = 6$	0	1	2	3	?				

			j →								
			0	1	2	3	4	5	6	7	8
i	1	$d_1 = 1$	0	1	2	3	4	5	6	7	8
↓	2	$d_2 = 4$	0	1	2	3	1	2	3	4	2
	3	$d_3 = 6$	0	1	2	3	1	2	?		

Voor het voorbeeld is de tabel als volgt gevuld:

			j →									
			0	1	2	3	4	5	6	7	8	
i	1	$d_1 = 1$	0	1	2	3	4	5	6	7	8	
↓	2	$d_2 = 4$	0	1	2	3	1	2	3	4	2	
	3	$d_3 = 6$	0	1	2	3	1	2	1	2	?	

			j →									
			0	1	2	3	4	5	6	7	8	
i	1	$d_1 = 1$	0	1	2	3	4	5	6	7	8	
↓	2	$d_2 = 4$	0	1	2	3	1	2	3	4	2	
	3	$d_3 = 6$	0	1	2	3	1	2	1	2	2	

1. Je kunt ook hier weer volstaan met een eendimensionaal hulpparray; deze moet dan wel **v.r.n.l.** worden gevuld.
2. Uit de tweedimensionale tabel kun je de/een optimale deelverzameling terugvinden.

			j →								
			0	1	2	3	4	5	6	7	8
i ↓	1	$d_1 = 1$	0	1	2	3	4	5	6	7	8
	2	$d_2 = 4$	0	1	2	3	1	2	3	4	2
	3	$d_3 = 6$	0	1	2	3	1	2	1	2	2

$d_3$  niet,  $d_2$  wel,  $d_2$  wel,  $d_1$  niet, dus de minimale oplossing bestaat uit 2 munten van 4 eurocent.

1. Een (eenvoudige) variatie is: gegeven een bedrag van  $n$  euro, is dat te betalen met muntsoorten  $d_1, \dots, d_m$ ? Dit kan geheel analoog aan het optimalisatieprobleem worden opgelost met DP. Gebruik een array `mnt`, waarbij `mnt[i][j] = True` als het bedrag  $j$  gemaakt kan worden met  $d_1, \dots, d_i$ , en anders `False`.
2. Een ander algoritme voor het muntenprobleem:  
betaal  $n$  met  $d_1, \dots, d_m$ ::  
geef de grootste munt  $d_i \leq n$ ;  
betaal  $n - d_i$  met  $d_1, \dots, d_i$

Dit is een zogenaamd **gretig algoritme**. Bovenstaand algoritme is erg snel, maar werkt helaas niet altijd.

- donderdag 19 april 2007 in zaal 312

- **Opgaven:**

zie <http://www.liacs.nl/home/graaf/ALGO/algo2007.html>

- **Volgend college:**

vrijdag 20 april 2007