Negende college algoritmiek

7 april 2011

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*)

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 (de volgorde wordt daar bepaald door de recursie)

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

Binomiaalcoëfficienten

			1				
		1		1			
	1		2		1		
1		3		3		1	
	1		4		6		4		1
1		5		10		10		5		1
	1		6		15		20		15		6		1
1		7		21		35		35		21		7		1

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: $O(\binom{n}{k})$ (exercise 8.1.6)

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: $\Theta(n * k)$

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			
⋮						
k	1					1
⋮						
$n-1$	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 eendimensionaal array ter lengte k. Er is dus maar $O(k)$ extra geheugen nodig. Zie ook exercise 8.1.4.

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 $V[i][j]$ 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):

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

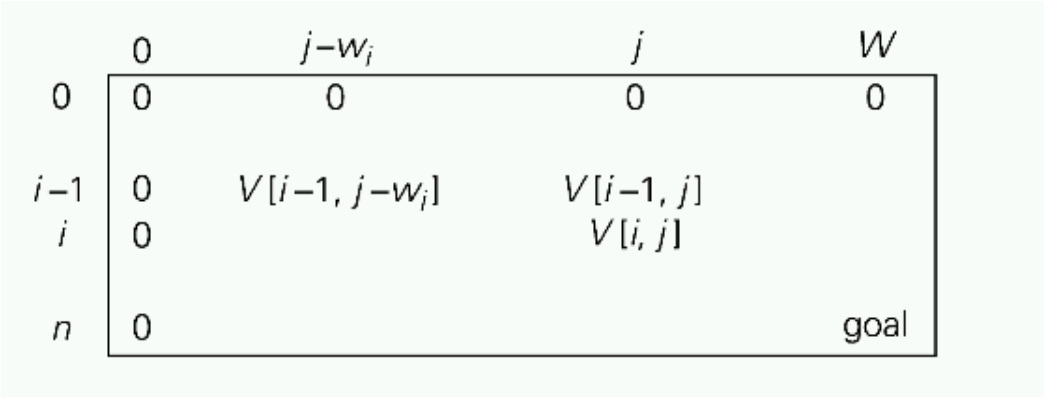
En we definiëren:

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

KZP met DP — vervolg

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 od



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													

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													

		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	?			

KZP met DP — tabel 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	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

Top down met array

```
RecKnapzak(i, j) :: // V[i][j] == -1: nog niet berekend
    if ( V[i][j] >= 0 ) then return V[i][j];
    else
        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];
    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

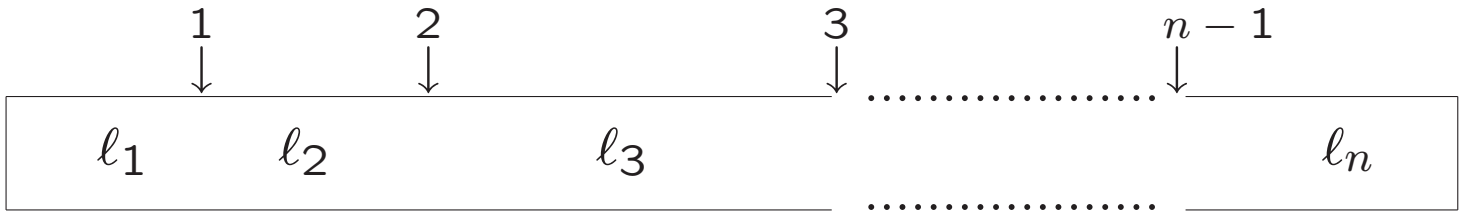
5. Vul aldus bottom up de tabel in (algoritme)

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

Een houtzaagmolen rekent voor het in twee stukken zagen van een stam van lengte ℓ precies ℓ euro, ongeacht de plek waar dit moet gebeuren. Na bestudering van de knoesten op een boomstam van lengte ℓ wordt besloten dat deze in achtereenvolgens (v.l.n.r. gezien) stukken van lengtes $\ell_1, \ell_2, \ldots, \ell_n$ gezaagd moet 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 $\ell_1 = 6, \ell_2 = 8, \ell_3 = 7, \ell_4 = 2$. 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 (zaagplekken dus bekend).

Deelproblemen

Het probleem brengen we terug tot het bepalen van de minimale kosten $Z[i][j]$ die moeten worden gemaakt om de (deel)stam (van zaagplek $i-1$ tot zaagplek j) ter lengte $L(i,j) = \ell_i + \ell_{i+1} + \cdots + \ell_j$ te verzagen tot achtereenvolgens stukken van lengte $\ell_i, \ell_{i+1}, \ldots, \ell_j$. Alle ℓ_i, en dus ook alle $L(i,j)$ en alle zaagplekken, 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 $d_1, d_2, \ldots d_m$ 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).

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):

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

MP met DP: tabel

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

		j $\rightarrow$								
		0	1	.	.	.	j	.	.	n
i	1									
$\downarrow$	2									
	.									
	.									
	.						$i-1,j$			
	i		$i,j-d_i$				i,j			
	.									
	m									

for $i := 1$ **to** m **do**
    $\texttt{munt}[i][0] := 0$; **od**
for $j := 1$ **to** $d_1 - 1$ **do**
    $\texttt{munt}[1][j] := \infty$; **od**
for $j := d_1$ **to** n **do**
    $\texttt{munt}[1][j] := 1 + \texttt{munt}[1][j - d_1]$; **od**

for $i := 1$ **to** m **do**
    for $j := 1$ **to** $d_i - 1$ **do**
        $\texttt{munt}[i][j] := \texttt{munt}[i - 1][j]$;
    od
    for $j := d_i$ **to** n **do**
        $\texttt{munt}[i][j] :=$ minimum $\{\texttt{munt}[i - 1][j], 1 + \texttt{munt}[i][j - d_i]\}$;
    od
od

De complexiteit van dit algoritme is $\Theta(m * n)$; extra gebruikt geheugen voor het opslaan van de tussenresultaten is eveneens $\Theta(m * n)$.

Voor het voorbeeld wordt de tabel als volgt gevuld:

			j	$\rightarrow$							
			0	1	2	3	4	5	6	7	8
i	1	$d_1 = 1$	0	1	2	3	4	5	6	7	8
$\downarrow$	2	$d_2 = 4$	0	1	2	3	1	2	3	4	2
	3	$d_3 = 6$	0	1	2	3	?				

$\texttt{munt}[3][4] = \texttt{munt}[2][4]$, want $d_3 = 6 > 4$.

$\texttt{munt}[3][6] = \min\{\texttt{munt}[2][6], \texttt{munt}[3][0] + 1\} = 1$.

$\texttt{munt}[3][8] = \min\{\texttt{munt}[2][8], \texttt{munt}[3][2] + 1\} = 2$.

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 hulparray; deze moet dan wel v.l.n.r. 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 $d_1, \ldots d_m$? Dit kan geheel analoog aan het optimalisatieprobleem worden opgelost met DP. Gebruik een array `munt`, waarbij `munt`$[i][j]$ = True als het bedrag j gemaakt kan worden met $d_1, \ldots d_i$, en anders False.

2. Een ander algoritme voor het muntenprobleem:

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

   geef de grootste munt $d_i \leq n$;

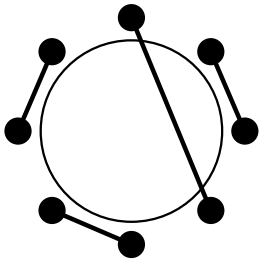
   betaal $n - d_i$ met $d_1, \ldots d_i$

   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).

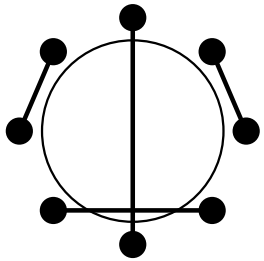
Dit kan ook met DP ....

We hebben een kring van n studenten, waarvan de studierichting bekend is. Iedereen moet precies één andere student de hand schudden. Dit handen schudden moet zodanig gebeuren dat geen enkel tweetal armen elkaar kruist.

Voorbeeld met acht personen:



Geen kruisende armen:
toegestaan



Kruisende armen:
niet toegestaan

De bedoeling is om het aantal paren studenten met dezelfde studierichting dat elkaar een hand geeft te maximaliseren.

- **Lezen bij dit college:**

  Paragraaf 8.1; sheets; paragraaf 8.4

- **Werkcollege:**

  donderdag 7 april 2011, 13:45–15:30, in zaal Archipel

- **Opgaven:**

  zie http://www.liacs.nl/home/rvvliet/algoritmiek

- **Volgend college:**

  donderdag 14 april 2011