

Negende college algoritmiëk

17 april 2015

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*)

- 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

**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  $\text{munt}[j]$  het minimale aantal munten zijn dat nodig is om een bedrag van  $j$  eurocent te betalen. We zoeken dus  $\text{munt}[n]$ .

Neem voor het gemak even aan dat de muntsoorten oplopend zijn gesorteerd ( $d_1 < d_2 < \dots < d_m$ ).

Dan geldt:

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

Opmerking: als  $0 < j < d_1$  is bedrag  $j$  niet te maken. Je gebruikt hier dat je weet welke munt de kleinste waarde heeft.

Vul array *munt* van links naar rechts.

```
munt[0] := 0;
for j := 1 to n do
  tmp := ∞;
  i := 1;
  while i ≤ m and di ≤ j do // oplopend gesorteerd
    if 1 + munt[j - di] < tmp then
      tmp := 1 + munt[j - di];
    fi
    i := i + 1;
  od
  munt[j] := tmp;
od
```

Complexiteit MP met DP:  
tijd  $\Theta(m * n)$ ; extra geheugen:  $\Theta(n)$

**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 uit de tabel:

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

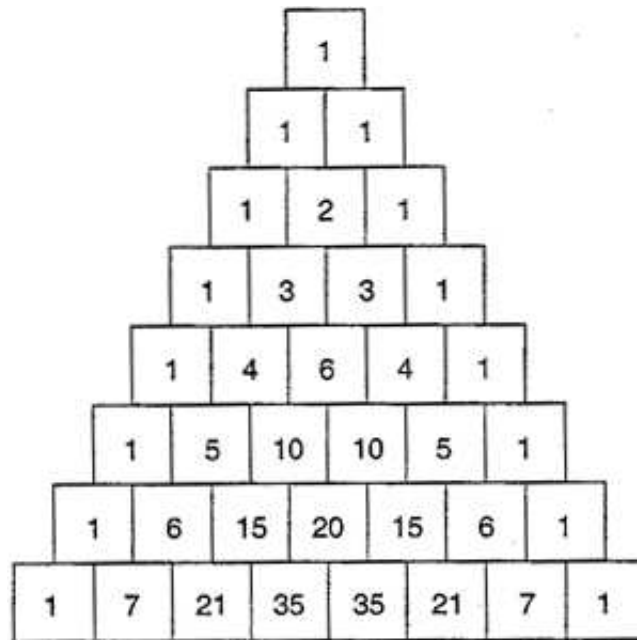
1. Een (eenvoudige) variant is: gegeven een bedrag van  $n$  euro, is dat te betalen met muntsoorten  $d_1, \dots, d_m$ ? Dit kan geheel analoog aan het optimalisatieprobleem worden opgelost met DP. Gebruik een array `munt`, waarbij `munt[j] = True` als het bedrag  $j$  gemaakt kan worden, en anders `False`.
2. Een ander algoritme voor het muntenprobleem:  
betaal  $n$  met  $d_1, \dots, d_m$ ::  
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 het levert niet altijd een optimale (hier: minimale) oplossing (soms ook geen oplossing, terwijl er wel een oplossing is).

Vul boolean array `munt` van links naar rechts.

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

Complexiteit MP met DP:  
tijd  $\Theta(m * n)$ ; extra geheugen:  $\Theta(n)$



Driehoek van  
Pascal



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

Gegeven een pot met  $n$  **verschillende** objecten.

Doe **in één keer** een greep van  $k$  objecten uit de pot. Ofwel, kies  $k$  objecten zonder terugleggen.

Hoeveel mogelijkheden?

En ook:

$$(a + b)^n =$$

$$\binom{n}{0}a^n + \binom{n}{1}a^{n-1}b + \binom{n}{2}a^{n-2}b^2 + \dots + \binom{n}{n-1}ab^{n-1} + \binom{n}{n}b^n$$

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  $\text{bin1}(i,j)$ 's worden echter herhaald berekend.

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

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 geïntialiseerd  
    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];  
    }  
}
```

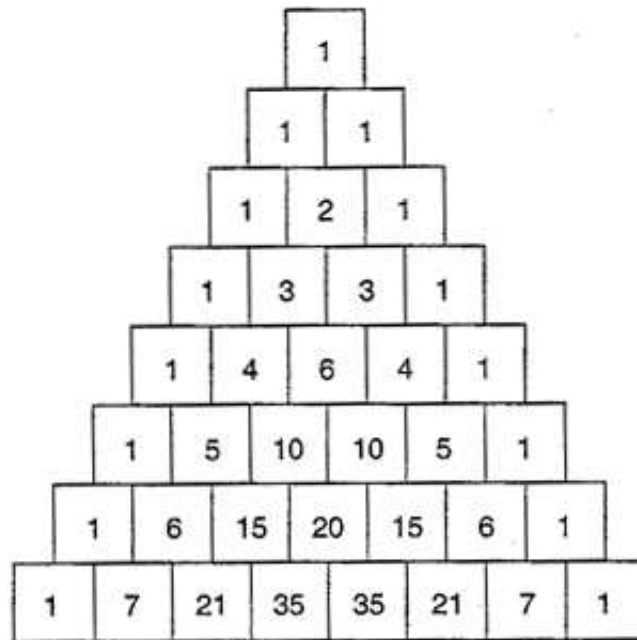
Complexiteit:  $O(n * k)$ ; extra geheugen:  $\Theta(n * k)$

We gebruiken (de linker benedendriehoek van) een globaal, op nul geïntialiseerd 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					$C(n, k)$

Top down:

Het array wordt gevuld in de volgorde zoals voorgeschreven door de recursie; niet alle array-elementen uit de linker benedendriehoek zijn nodig.



Driehoek van  
Pascal



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

We gebruiken (de linker benedendriehoek van) een globaal, op nul geïntialiseerd 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 recurrenente betrekking (recursieve formulering); de hele linker benedendriehoek wordt gevuld.

```
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:  $\text{bin3}(n, k)$ .

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

Je kunt hier echter volstaan met een eendimensionaal array ter lengte  $k$  als je alleen  $\binom{n}{k}$  wilt weten. Er is dus maar  $O(k)$  extra geheugen nodig. Zie ook exercise 8.1.9.

## 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  $F[i][j]$  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.

Opmerking. De bijbehorende optimale deelverzameling kunnen we later uit de tabel afleiden.

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.

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

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

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:  $\Theta(n * W)$

Voor het voorbeeld wordt de tabel als volgt gevuld:

		$j \rightarrow$													
		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:

		$j \rightarrow$													
		0	1	2	3	4	5	6	7	8	9	10	11	12	
$i \downarrow$	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														

		$j \rightarrow$													
		0	1	2	3	4	5	6	7	8	9	10	11	12	
$i \downarrow$	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	?			

		$j \rightarrow$													
		0	1	2	3	4	5	6	7	8	9	10	11	12	
$i \downarrow$	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 ( wi > j ) then
        F[i][j] := RecKnapzak(i-1, j);
      else
        F[i][j] := max { RecKnapzak(i-1, j),
                          vi + RecKnapzak(i-1, j-wi) };
      fi
    fi
  return F[i][j];
fi .
```

Vraag: welke van de twee methodes verdient de voorkeur?

KZP lijkt erg op MP. Bij KZP gebruiken we echter een tweedimensionaal array en bij MP is een eendimensionaal array voldoende.

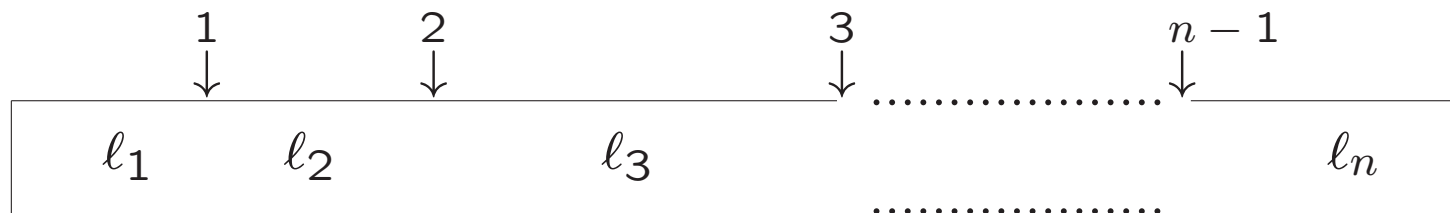
KZP	MP
$n$ objecten gewicht $w_i$ waarde $v_i$ totaal gewicht $\leq$ capaciteit $W$ max. totale waarde elk object $\leq 1$ keer	$m$ munten waarde $d_i$ 'kosten' 1 totale waarde = bedrag $n$ min. totale 'kosten' munt mag meer keer

Verklaring: bij het muntenprobleem is er geen noodzaak om bij te houden welke munten we al gebruikt hebben, zoals bij het knapzakprobleem wel het geval is.

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. Je kunt nu nog kiezen tussen DP met recursie of bottom up DP (iteratief). In wat volgt gaan we bottom up te werk.
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 (indien gewenst)
7. Pas je algoritme zo aan dat je uit de tabel niet alleen een waarde maar ook de (optimale) oplossing zelf kunt halen. Dit kan achteraf uit de tabel of via een geschikt hulparray dat tijdens het vullen van de tabel wordt opgebouwd.

1. Dynamisch programmeren is alleen echt zinvol als sprake is van overlappende deelproblemen
2. 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 (v.l.n.r. gezien) stukken van lengtes  $\ell_1, \ell_2, \dots, \ell_n$  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  $l_1 = 6, l_2 = 8, l_3 = 7, l_4 = 2$ . 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 (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 (met zaagplekken  $i$  tot en met  $j - 1$ ) 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  $L(i, j)$  en alle zaagplekken, zijn gegeven. Het oorspronkelijke probleem is dan het bepalen van  $Z[1][n]$ . Merk op dat altijd  $1 \leq i \leq j \leq n$ . Verder  $Z[i][i] = 0$ ; er hoeft niet gezaagd te worden.

## Recurrente betrekking

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

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

	$j$	$\rightarrow$										
$i$	0	.	.	.	.	.	.	.	.	.	.	.
$\downarrow$	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: de tabel rij voor rij vullen (móét van onder naar boven in het plaatje) 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
```

- **Lezen/leren bij dit college:**  
Sheets, paragrafen boek: 8 incl., 2.5, 8.1 (vb. 2), 8.2  
Scan binomiaalcoëfficiënten;
- **Werkcollege:** donderdag 23 april 2015 in B2
- **Deadline** programmeeropdracht 2:  
dinsdag 28 april 2015, 10:00 uur
- **Extra vragenuren (zaal 302/304):**  
donderdag 23 april, 15:30 – 17:00 uur  
vrijdag 24 april, 15:30 – 17:00 uur
- **Opgaven en programmeeropdrachten:**  
zie <http://liacs.leidenuniv.nl/~graafjmde/ALGO/>
- **Volgend college:**  
vrijdag 24 april 2015, 11:15–13:00, zaal B2.