Negende college algoritmiek

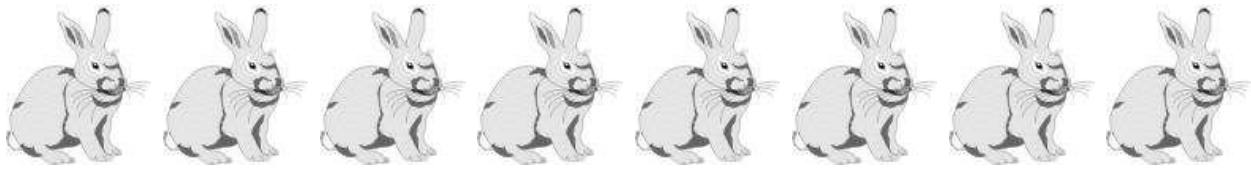
12 april 2022

Dynamisch Programmeren

Definitie Fibonacci-getallen:

$$\text{fib}(n) = \begin{cases} 0 & \text{als } n = 0 \\ 1 & \text{als } n = 1 \\ \text{fib}(n-1) + \text{fib}(n-2) & \text{als } n > 1 \end{cases}$$

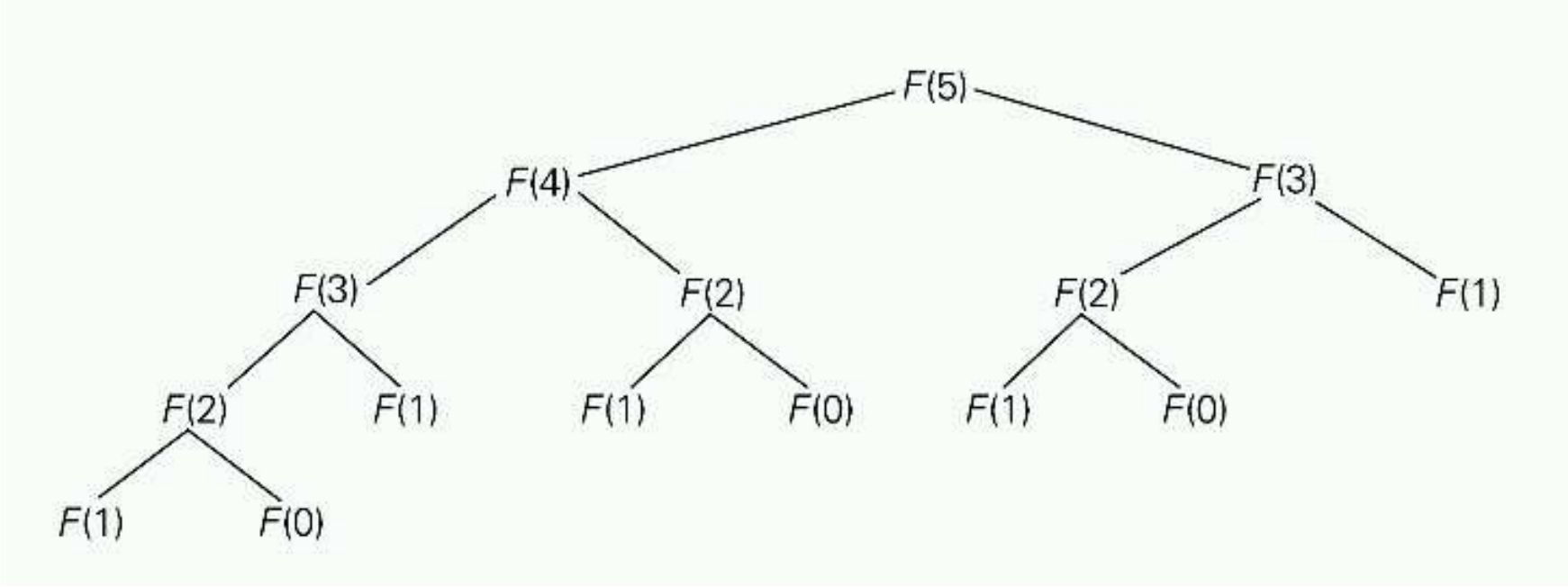
0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584, 4181, 6765, 10946, ...



Recursieve C++-functie:

```
long fib1 (int n) {
   if ( ( n==0 ) || ( n == 1) )
      return n;  // gaat goed!
   else
      return ( fib1 (n-1) + fib1 (n-2) );
} // fib1
```

Watervaleffect



Voor $n = 5$ worden sommige recursieve aanroepen meerdere malen gedaan. Voor grotere waarden van n wordt dit watervaleffect steeds groter. Dit komt doordat deelproblemen elkaar overlappen.

Oplossing: gebruik een array om tussenresultaten op te slaan, en los op die manier elk deelprobleem precies één keer op.

Dit kan op twee manieren:

1. Top down: memory function
   Combineert recursie met het gebruik van een array

2. Bottom up: het klassieke dynamisch programmeren (DP)
   Vult het array van klein naar groot (for-loop)

```
const int MAX = 45;
long memo[MAX];   // helemaal op -1 initialiseren

long fib2 (int n) {  // recursie met array !
    if ( memo[n] > -1 )  // al eerder berekend
       return memo[n];
    else {
       if ( ( n==0 ) || ( n == 1) )
          memo[n] = n;  // gaat goed!
       else
          memo[n] = fib2 (n-1) + fib2 (n-2);
       return memo[n];
    } // else
 } // fib2
```

Dynamisch programmeren: gebruikt ook een array voor het opslaan van tussenresultaten, maar werkt bottom up. Gebruikt de recurrente betrekking waaraan de Fibonacci-getallen voldoen.

```
fibo[0] = 0;
fibo[1] = 1;
for (i=2; i<=n; i++) {
    fibo[i] = fibo[i-1] + fibo[i-2];
}
return fibo[n];
```

Je hebt overigens niet het hele array nodig, maar je kunt volstaan met 3 (of zelfs 2) variabelen. Zo krijg je de bekende iteratieve oplossing (zie ook Programmeermethoden).

Met drie variabelen.

```
fibo0 = 0;
fibo1 = 1;
i = 1;
while (i<n) {
  tmp = fibo0;
  fibo0 = fibo1;
  fibo1 = tmp + fibo0;
  i++;
}
return fibo1;
```

tmp	fibo0	fibo1	i
	0	1	1
0	1	1	2
1	1	2	3
1	2	3	4
2	3	5	5
3	5	8	6
5	8	13	7
8	13	21	8
...	...	...	...

Werkt voor. . .

- nuttig bij problemen met *overlappende deelproblemen*

- druk een oplossing van het probleem uit in oplossingen van deelproblemen (*recursieve formulering*) (vgl. verdeel en heers)

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

Laat `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.

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. . .

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).

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

kosten(n) $= \ldots$

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

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

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 .
```

Complexiteit / aantal aanroepen. . .

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 .
```

Complexiteit / aantal aanroepen: 2^{n-1} als $n \geq 1$

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:

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

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.

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 .
```

Complexiteit / aantal aanroepen: 2^{n-1} als $n \geq 1$

```
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.

Tentamen, juni 2013

Gegeven een driehoek bestaande uit n rijen met positieve getallen, waarbij de bovenste rij 1 getal bevat, de rij daaronder 2 getallen, tot en met n getallen op de onderste rij, als in het plaatje hieronder, met $n = 4$:

```
      2
    5   4
  3   4   7
1   6   9   6
```

Wat is de grootste som die je kunt krijgen door vanuit de top naar beneden te lopen, waarbij je vanaf een positie in de driehoek alleen naar de twee posities er schuin onder kunt stappen?

De driehoek kan worden gerepresenteerd als een $n \times n$ array $D[1 \ldots n][1 \ldots n]$, waarvan alleen de linkeronderdriehoek gevuld is, als in het plaatje hieronder.

```
2
5  4
3  4  7
1  6  9  6
```

Vanuit $D[i][j]$ kun je in één stap $D[i+1][j]$ en $D[i+1][j+1]$ bereiken.

Brute force / Exhaustive search. . .

Complexiteit / aantal aanroepen. . .

De driehoek kan worden gerepresenteerd als een $n \times n$ array $D[1 \ldots n][1 \ldots n]$, waarvan alleen de linkeronderdriehoek gevuld is, als in het plaatje hieronder.

```
2
5  4
3  4  7
1  6  9  6
```

Vanuit $D[i][j]$ kun je in één stap $D[i+1][j]$ en $D[i+1][j+1]$ bereiken.

Brute force / Exhaustive search: alle paden aflopen

Complexiteit / aantal aanroepen: $\Omega(2^{n-1})$

```
2
5  4
3  4  7
1  6  9  6
```

Recursieve formulering met $S(i, j)$. Twee mogelijkheden:

- $S(i, j)$ is maximale som die je kunt bereiken door vanuit positie (i, j) naar beneden te lopen

- $S(i, j)$ is maximale som die je kunt bereiken door vanuit positie $(1, 1)$ naar positie (i, j) te lopen

Voorkeur. . .

```
2
5   4
3   4   7
1   6   9   6
```

$S(i, j)$ is maximale som die je kunt bereiken door vanuit positie (i, j) naar beneden te lopen

$$S(i, j) = \ldots$$

```
2
5   4
3   4   7
1   6   9   6
```

$S(i,j)$ is maximale som die je kunt bereiken door vanuit positie (i,j) naar beneden te lopen

$$S(i,j) = \begin{cases} D[i][j] & \text{als } i = n \text{ en } 1 \leq j \leq i \\ D[i][j] + \max\{S(i+1,j), S(i+1,j+1)\} & \text{als } i < n \text{ en } 1 \leq j \leq i \end{cases}$$

Gevraagd: $S(1,1)$

Recursief:

```
int S (int n, int i, int j) {
  if (i==n)
    return D[i][j];
  else
    return D[i][j] + max (S(n,i+1,j), S(n,i+1,j+1));
}
```

Complexiteit / aantal aanroepen...

Recursief:

```
int S (int n, int i, int j) {
  if (i==n)
    return D[i][j];
  else
    return D[i][j] + max (S(n,i+1,j), S(n,i+1,j+1));
}
```

Complexiteit / aantal aanroepen: $\Theta(2^n)$

Bottom up DP:

```
for (j=1;j<=n;j++)
  S[n][j] = D[n][j];

for (i=n-1;i>=1;i--)
  for (j=1;j<=i;j++)
    S[i][j] = D[i][j] + max (S[i+1][j], S[i+1][j+1]);

return S[1][1];
```

Ons voorbeeld...

```
2
5   4
3   4   7
1   6   9   6
```

Bottom up DP:

```
for (j=1;j<=n;j++)
  S[n][j] = D[n][j];

for (i=n-1;i>=1;i--)
  for (j=1;j<=i;j++)
    S[i][j] = D[i][j] + max (S[i+1][j], S[i+1][j+1]);

return S[1][1];
```

Ons voorbeeld

```
2                       22
5   4                   18  20
3   4   7                9  13  16
1   6   9   6            1   6   9   6
```

Pad terugvinden. . .

Bottom up DP:

```
for (j=1;j<=n;j++)
  S[n][j] = D[n][j];

for (i=n-1;i>=1;i--)
  for (j=1;j<=i;j++)
    S[i][j] = D[i][j] + max (S[i+1][j], S[i+1][j+1]);

return S[1][1];
```

Tijdcomplexiteit. . .

Ruimtecomplexiteit. . .

Bottom up DP:

```
for (j=1;j<=n;j++)
  S[n][j] = D[n][j];

for (i=n-1;i>=1;i--)
  for (j=1;j<=i;j++)
    S[i][j] = D[i][j] + max (S[i+1][j], S[i+1][j+1]);

return S[1][1];
```

Tijdcomplexiteit: $\Theta(n^2)$

Ruimtecomplexiteit: $\Theta(n^2)$ met 2D array
$\Theta(n)$ met 1D array

Andere definitie Score:

$S(i, j)$ is maximale som die je kunt bereiken door vanuit positie $(1, 1)$ naar positie (i, j) te lopen

Ons voorbeeld. . .

2			
5	4		
3	4	7	
1	6	9	6

2			
7	6		
10	11	13	
11	17	22	19

```
2                          2
5   4                      7   6
3   4   7                 10  11  13
1   6   9   6             11  17  22  19
```

Bottom up DP:

```
S[1][1] = D[1][1];

for (i=2;i<=n;i++)
{ S[i][1] = D[i][1] + S[i-1][1];
  for (j=2;j<i;j++)
    S[i][j] = D[i][j] + max (S[i-1][j-1], S[i-1][j]);
  S[i][i] = D[i][i] + S[i-1][i-1];
}

return max (S[n][1],S[n][2],...,S[n][n]);
```

Ten slotte:

- Negatieve getallen
- Minimale som i.p.v. maximale som

Geen probleem

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

Deelproblemen. . .

Laat $F[i][j]$ de waarde zijn van de meest waardevolle deelverzameling van de eerste i ($0 \leq i \leq n$) objecten, die in een knapzak met capaciteit j ($0 \leq j \leq W$) past. We zoeken dus $F[n][W]$. We nemen hier impliciet aan dat W een positief geheel getal is.

$$F[i][j] = \ldots$$

Laat $F[i][j]$ de waarde zijn van de meest waardevolle deelverzameling van de eerste i ($0 \leq i \leq n$) objecten, die in een knapzak met capaciteit j ($0 \leq j \leq W$) 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):

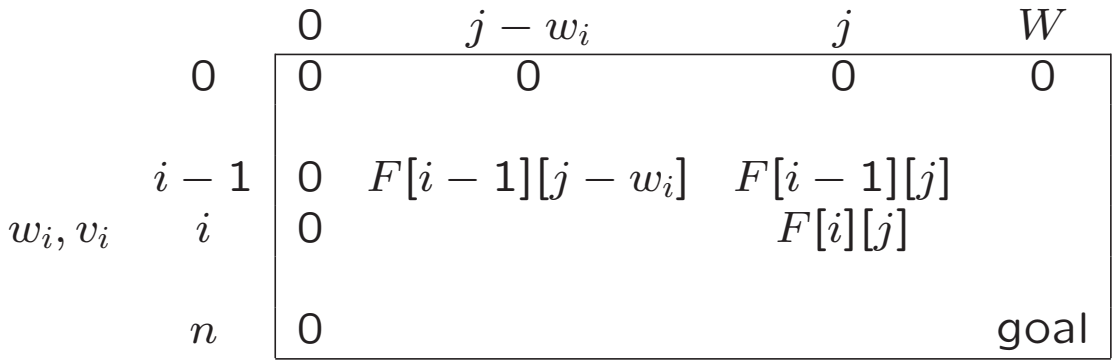
$$F[i][j] = \begin{cases} \max\{F[i-1][j], v_i + F[i-1][j-w_i]\} & \text{als } j \geq w_i \\ F[i-1][j] & \text{als } j < w_i \end{cases}$$

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 $j := 0$ **to** W **do**
  $F[0][j] := 0$;
for $i := 1$ **to** n **do**
  $F[i][0] := 0$;
for $i := 1$ **to** n **do**
  for $j := 1$ **to** W **do**
    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 od od



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													

Ter herinnering:

i	w_i	v_i
1	8	42
2	3	14
3	4	40
4	5	27

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	?			

Ter herinnering:

i	w_i	v_i
1	8	42
2	3	14
3	4	40
4	5	27

		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	$\rightarrow$											
		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
$\downarrow$	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?

		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

Ter herinnering:

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

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):

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

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 bekeken 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:

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

Voorbeeld:

type munt	waarde
1	4
2	6

te betalen bedrag: 8

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
        i++;
    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
        i++;
    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] := true;
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
        i++;
    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:**
  slides; 8.inl., paragraaf 8.2;
  voorbeeld 2 in paragraaf 8.1

- **Werkcollege** dynamisch programmeren
  vanmiddag, 14.15–16.00, zalen 401 / 402 Snellius

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

- **Practicumbijeenkomst** programmeeropdracht 1 / 2:
  woensdag 13 april 2022, 14.15–16.00, computerzalen Snellius

- **Volgend college:**
  dinsdag 19 april 2022, 11.15–13.00