Elfde college algoritmiek

28 april 2026

Dynamisch Programmeren, Gretige Algoritmen

Even praktisch. . .

- Programmeeropdracht 1:
  - deadline met 1 punt aftrek: vanavond, 23.59 uur
  - nakijken. . .
- Programmeeropdracht 2: deadline: maandag 18 mei, 23.59 uur
- woensdag 29 april: practicumbijeenkomst
- maandag 4 mei: werkcollege

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

		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	→											
		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	→											
		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

```
KnapzakTD(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] := KnapzakTD(i-1, j);
            else
                F[i][j] := max { KnapzakTD(i-1, j),
                                 v_i + KnapzakTD(i-1, j-w_i) };
            fi
        fi
        return F[i][j];
    fi
```

Vraag: welke van de twee methodes verdient de voorkeur?

Nodig voor top down

		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

Werkwijze en opmerkingen

1. Druk de (waarde van de) oplossing van het probleem uit in (de waarde van) oplossingen van deelproblemen. Ofwel: stel een recurrente betrekking op
2. Gebruik alleen dynamisch programmeren bij overlappende deelproblemen
3. Definieer een geschikte tabel en ga na wat de berekeningsvolgorde moet zijn
4. Vul aldus bottom up de tabel in (algoritme)
5. Let op geheugenbesparing
6. Pas je algoritme zo aan dat je uit de tabel niet alleen een waarde maar ook de (optimale) oplossing zelf kunt halen
7. Dynamisch programmeren wordt vaak gebruikt voor optimalisatieproblemen

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 $\leq W$).

Voorbeeld:

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

knapzakcapaciteit 12

Gretig algoritme. . .

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 $\leq W$).

Voorbeeld:

object	gewicht	waarde	w/g
1	8	42	5.25
2	3	14	4.67
3	4	40	10
4	5	27	5.4

knapzakcapaciteit 12

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 gewicht w_i totaal gewicht $\leq$ capaciteit W waarde v_i max. totale waarde elk object $\leq$ 1 keer	m munten waarde d_i totale waarde $=$ bedrag n ‘kosten’ 1 min. totale ‘kosten’ 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 munt$[j]$ het minimale aantal munten zijn dat nodig is om een bedrag van j eurocent te betalen. We zoeken dus 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 $\texttt{munt}[j]$ = 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).

Een gretige strategie (recursief geformuleerd):

betaal n met $d_1, \ldots d_m$ (voor het gemak oplopend gesorteerd)::

if $(n = 0)$ klaar;

else geef de grootste munt $d_i \leq n$ (restrictie);

// dan is het nog te betalen bedrag zo klein mogelijk

// en dus heb je zo weinig mogelijk munten

// nodig (hoop je)

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

Bovenstaand algoritme is erg eenvoudig en snel, en levert voor het geval van de gebruikelijke euro-munten (munten van 1, 2, 5, 10, 20, 50, 100, 200 eurocent) het optimale antwoord. Dit is echter niet het geval voor de muntwaarden uit het voorbeeld. (Bovendien gaat dit algoritme ervan uit dat het bedrag te betalen is. Anders is nog een kleine aanpassing nodig.)

- Greed = hebzucht
- Voor oplossen van optimalisatieproblemen
(of loop risico dat niet aan alle eisen voldaan is)
- Oplossing wordt stap voor stap opgebouwd
- In elke stap wordt een gretige keuze gemaakt waarmee de huidige deeloplossing wordt uitgebreid
- Dat wil zeggen: een (locale) keuze die op dat moment de beste lijkt (de grootste directe winst oplevert)
- De vraag is of dat leidt tot een globaal optimale oplossing

De oplossing wordt dus opgebouwd via een serie achtereenvolgende gretige keuzes. Deze keuzes

- zijn consistent met de restricties van het probleem
- zijn locaal optimaal, d.w.z. de best uitziende keuze in die stap
- zijn onherroepelijk: keuzes kunnen niet meer worden teruggedraaid

Een gretig algoritme ziet er dus ruwweg zo uit:

> **while** nog niet alle stappen zijn gedaan **do**
>   doe een keuze die in eerste instantie de grootste
>   winst lijkt op te leveren
> **od**

Uitbreiden van deeloplossingen moet uiteraard wel steeds in overeenstemming met de geldende restricties.

Soms leveren gretige algoritme een optimale oplossing, en soms/vaak niet. In dat geval is de gretige strategie een heuristiek, die bijvoorbeeld leidt tot een goede, maar meestal niet optimale oplossing. Of: de gretige strategie leidt vaak, maar niet altijd, tot een optimale oplossing.

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

Recurrente betrekking met $S(i,j)$. Twee mogelijkheden:

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

2. . . . (komt straks)

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: $S(1,1)$

Rechstreeks 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. . .

Rechstreeks 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:

2. $S(i,j)$ 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

Andere definitie Score:

2. $S(i,j)$ 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:

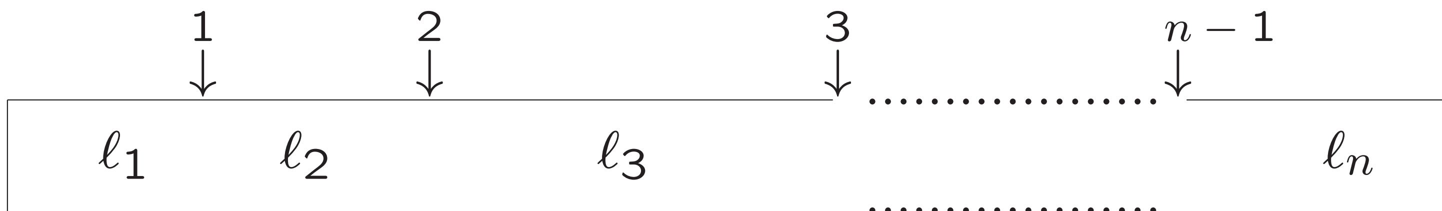
```
S[1][1] = D[1][1];

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

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 stukken i t/m 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

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

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 stukken i t/m 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

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

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	.	.	*	.	.

Invulvolgorde. . .

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	.	.	*	.	.

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
```