Negende college algoritmiek

9 april 2025

Verdeel en Heers
Dynamisch Programmeren

Puzzel 2:

```
  D O N A L D
  G E R A L D
  ----------- +
  R O B E R T
```

Elke letter stelt een cijfer voor $(0, 1, \ldots, 9)$ en verschillende letters corresponderen met verschillende cijfers. Het cijfer 0 komt niet voor als meest linkse cijfer in een getal. (Dus i.h.b. is $D \neq 0$, $G \neq 0$ en $R \neq 0$.)

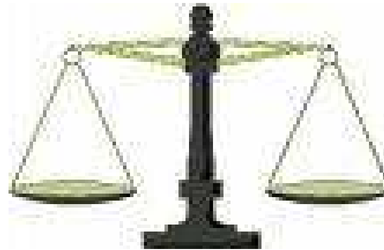
Om het makkelijker te maken is gegeven dat $D = 5$.

Vergelijk ook Levitin, hoofdstuk 3.4, opgave 11.

Is programmeeropdracht 2, deadline 29 april 2025.

Fake coin probleem

Gegeven n identiek uitziende munten. Eén ervan is vals. Bekend is dat de valse munt lichter is dan de andere. Tevens is een balans beschikbaar. Bepaal door weging de valse munt.



Decrease by a constant factor: verdeel de munten in twee stapels van $\lfloor \frac{n}{2} \rfloor$ en —indien n oneven— een losse munt. Als de twee stapels even zwaar zijn (best case) is de losse munt de valse. Zo niet, dan bevindt de valse zich in de lichtste van de twee stapels.

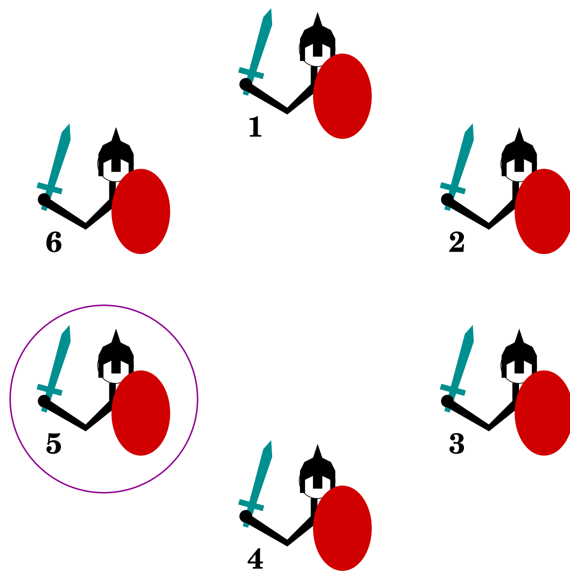
Recurrente betrekking voor het aantal wegingen dat nodig is in de worst case om de valse munt te ontdekken:

$$W(n) = W(\lfloor \tfrac{n}{2} \rfloor) + 1 \text{ als } n > 1,\ W(1) = 0$$

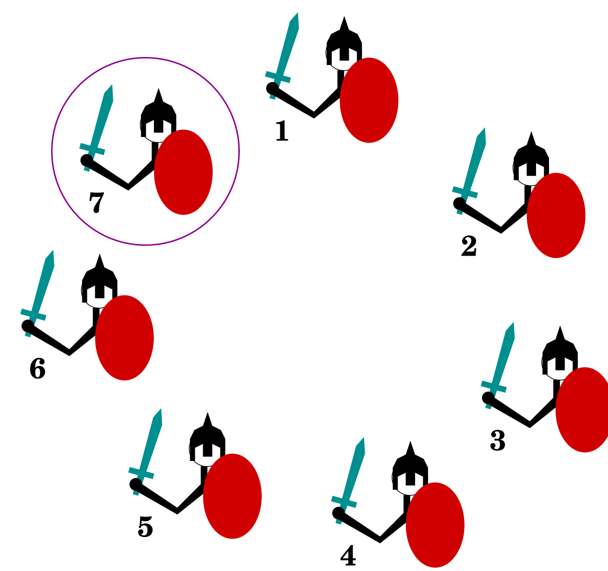
Oplossing: $W(n) = \lfloor \lg n \rfloor$.

Zie Levitin excercise 4.5.10 voor een efficiënter decrease by a constant factor algoritme.

Josephus probleem: gegeven n personen, genummerd 1 t/m n, die in een cirkel staan. Elimineer, te beginnen bij persoon 2, telkens elke tweede persoon, totdat er nog maar één persoon, $J(n)$, over is. Bepaal wie deze overlevende is.



Josephus 6



Josephus 7

Decrease by a constant factor: maak één doorgang door de cirkel. Er zijn dan nog $\lfloor \frac{n}{2} \rfloor$ overlevenden in de cirkel over, dus hetzelfde probleem maar gehalveerd.

Recurrente betrekking:

$$\begin{cases} J(1) & = & 1 \\ J(2k) & = & 2J(k) - 1 \\ J(2k+1) & = & 2J(k) + 1 \end{cases}$$

Oplossing:

n	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
$J(n)$	1	1	3	1	3	5	7	1	3	5	7	9	11	13	15	1

Variable-size decrease: de reductie in grootte is variabel,
dus kan in elke stap anders zijn

Voorbeelden:

- Algoritme van Euclides
  Dit is gebaseerd op: ggd(m, n) = ggd(n, m **mod** n)
- Flipping pancakes (Levitin, excercise 4.5.12)

- Binaire zoekbomen

Probleem:

Gegeven: twee niet-negatieve gehele getallen m en n (niet beide nul).
Vraag: wat is de grootste gemeenschappelijke deler, genoteerd als ggd(m,n), van m en n?

Voorbeelden:

ggd(60,24) = 12;
ggd(25,0) = 25;
ggd(200, 441) = 1;
ggd(588,495) = 3.

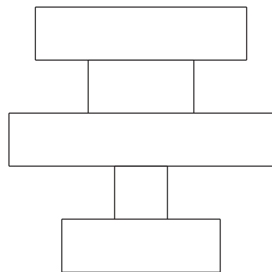
Het algoritme van Euclides is gebaseerd op het herhaald gebruiken van

$$\text{ggd}(m, n) = \text{ggd}(n, m \bmod n),$$

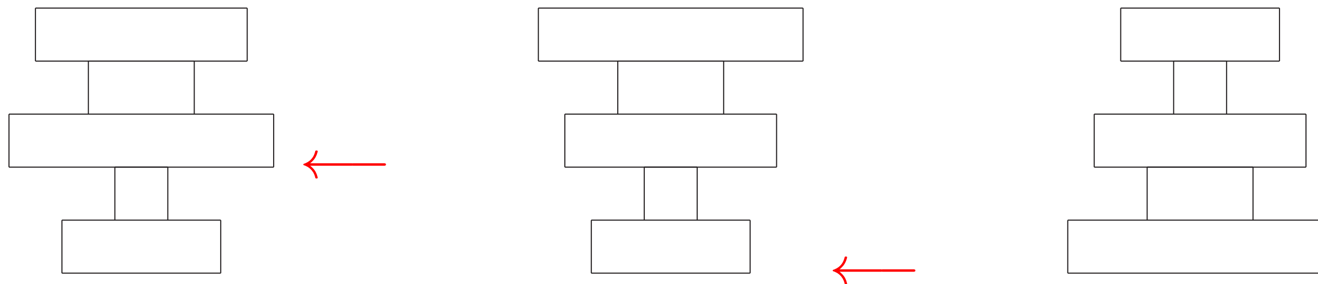
totdat de tweede parameter nul wordt.

Voorbeeld: ggd(60,24) = ggd(24, 12) = ggd(12, 0) = 12

Probleem: Gegeven een stapel van n pannenkoeken, allemaal verschillend in grootte. Verder is alleen een spatel beschikbaar, die je onder een pannenkoek kan schuiven, waarna je de hele stapel daarbovenop in één keer kan omdraaien. De bedoeling is om uiteindelijk alle pannenkoeken bovenop elkaar te krijgen in volgorde van grootte (de grootste onderop).



Probleem: Gegeven een stapel van n pannenkoeken, allemaal verschillend in grootte. Verder is alleen een spatel beschikbaar, die je onder een pannenkoek kan schuiven, waarna je de hele stapel daarbovenop in één keer kan omdraaien. De bedoeling is om uiteindelijk alle pannenkoeken bovenop elkaar te krijgen in volgorde van grootte (de grootste onderop).



BOUNDS FOR SORTING BY PREFIX REVERSAL

William H. GATES
Microsoft, Albuquerque, New Mexico

Christos H. PAPADIMITRIOU*†
Department of Electrical Engineering, University of California, Berkeley, CA 94720, U.S.A.

Received 18 January 1978
Revised 28 August 1978

For a permutation σ of the integers from 1 to n, let $f(\sigma)$ be the smallest number of prefix reversals that will transform σ to the identity permutation, and let $f(n)$ be the largest such $f(\sigma)$ for all σ in (the symmetric group) S_n. We show that $f(n) \leqslant (5n+5)/3$, and that $f(n) \geqslant 17n/16$ for n a multiple of 16. If, furthermore, each integer is required to participate in an even number of reversed prefixes, the corresponding function $g(n)$ is shown to obey $3n/2-1 \leqslant g(n) \leqslant 2n+3$.

1. Introduction

We introduce our problem by the following quotation from [1]

> The chef in our place is sloppy, and when he prepares a stack of pancakes they come out all different sizes. Therefore, when I deliver them to a customer, on the way to the table I rearrange them (so that the smallest winds up on top, and so on, down to the largest at the bottom) by grabbing several from the top and flipping them over, repeating this (varying the number I flip) as many times as necessary. If there are n pancakes, what is the maximum number of flips (as a function $f(n)$ of n) that I will ever have to use to rearrange them?

In this paper we derive upper and lower bounds for $f(n)$. Certain bounds were already known. For example, consider any stack of pancakes. An *adjacency* in

```
Quicksort(A[l···r])::
// sorteert het (sub)array A[l···r] recursief
// uitvoer: A[l···r] oplopend gesorteerd
    if l < r
        s :=Partitie(A[l···r]); // s het splitspunt
        Quicksort(A[l···s−1]);
        Quicksort(A[s+1···r]);
    fi .
```

Voorbeeld: 5 3 1 9 8 2 4 7

Partitie

Partitie($A[l \cdots r]$) ::

// partitioneert een (sub)array, met $A[l]$ als spil (pivot)

$p := A[l]$;

$i := l; j := r + 1$;

repeat

repeat $i := i + 1$; **until** $i > r$ **or** $A[i] \geq p$;

repeat $j := j - 1$; **until** $A[j] \leq p$;

if $i < j$ **then**

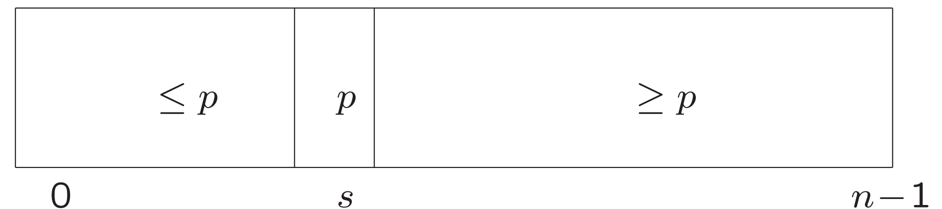
Wissel($A[i], A[j]$);

if

until $i \geq j$;

Wissel($A[l], A[j]$);

return j; .



Bekijk en vergelijk vier verschillende oplossingsmethoden voor het bereke-nen van a^n:

1. Brute force: gebaseerd op de definitie, $a^n = \overbrace{a * \cdots * a}^{n\times}$
2. Divide and conquer: gebaseerd op $a^n = a^{\lfloor \frac{n}{2} \rfloor} * a^{\lceil \frac{n}{2} \rceil}$
3. Decrease by one: gebaseerd op $a^n = a^{n-1} * a$
4. Decrease by a constant factor: . . .

Bekijk en vergelijk vier verschillende oplossingsmethoden voor het bereke-nen van a^n:

1. Brute force: gebaseerd op de definitie, $a^n = \overbrace{a * \cdots * a}^{n\times}$

2. Divide and conquer: gebaseerd op $a^n = a^{\lfloor \frac{n}{2} \rfloor} * a^{\lceil \frac{n}{2} \rceil}$

3. Decrease by one: gebaseerd op $a^n = a^{n-1} * a$

4. Decrease by a constant factor: gebaseerd op

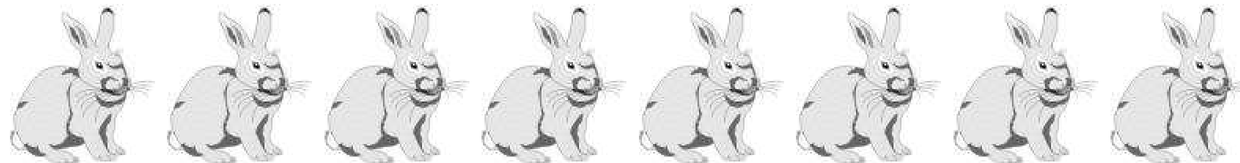
$$a^n = \begin{cases} (a^{\frac{n}{2}})^2 & \text{als } n \text{ even is} \\ (a^{\frac{n-1}{2}})^2 * a & \text{als } n \text{ oneven is} \end{cases}$$

Dynamisch Programmeren

Definitie Fibonacci-getallen:

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

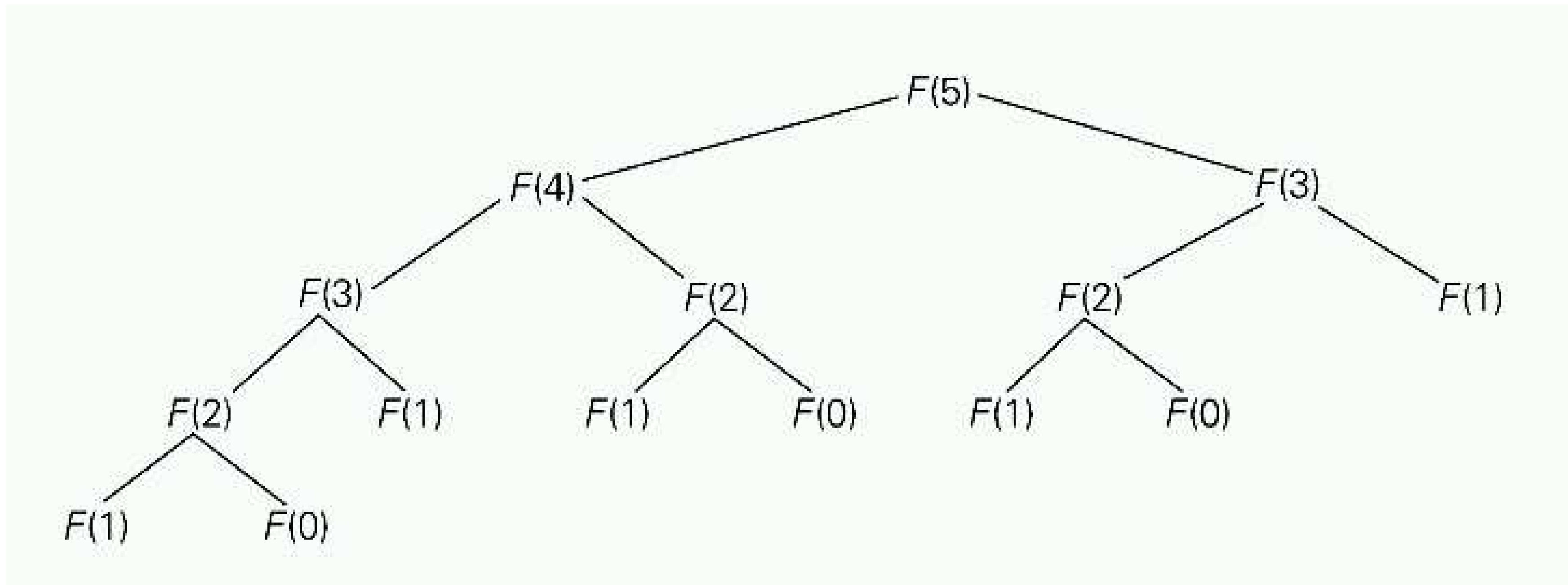
0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584, 4181, 6765, 10946, . . .



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 (*recurrente betrekking*) (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

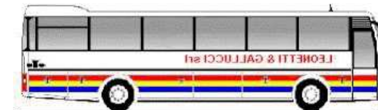
- bottom up lost *alle* deelproblemen op, top down alleen degene die echt nodig zijn voor het oplossen van het oorspronkelijke probleem

- 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:

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

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];
```

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.

Terugvinden reis. . .

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: $S(1,1)$

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:

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                   2
5  4                7   6
3  4  7             10  11  13
1  6  9  6          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

- **Lezen/leren bij dit college:**
  4.4 (geen Russian peasant), slides; 8.inl.;

- **Werkcollege** dynamisch programmeren
  vrijdag 25 april 2025, 11.00–12.45, BW.0.08

- **Practicumbijeenkomst** programmeeropdracht 2:
  dinsdag 15 april 2025, 09.00–10.45, DM.0.09, DM.0.13

- **Volgend college:**
  woensdag 16 of 23 april 2025, 15.15-17.00