Derde college algoritmiek

23 februari 2008

Kannen en Complexiteit

Probleem $\longrightarrow$ Toestand-actie-ruimte

Een toestand-actie-ruimte (toestand-actie-diagram, state transition diagram, toestandsruimte, state space)

- *Bestaat uit* alle mogelijke toestanden en acties
- Begintoestand, eindtoestand(en)
- Een actie veroorzaakt een overgang van de ene (toegelaten) toestand naar een andere
- *Oplossing* van het probleem: een opeenvolging van acties die van de begintoestand naar een eindtoestand leiden

Voorbeeld 4: Kannenprobleem

We hebben twee kannen: een grote met een inhoud van 8 liter, en een kleine met een inhoud van 5 liter. Op de kannen staat geen maatverdeling. Verder hebben we de beschikking over een waterkraan en een afvoer. Bij aanvang zijn beide kannen leeg.

Vraag: Hoe krijgen we precies 4 liter water in een van de twee kannen? En liefst zo snel mogelijk.

We onderscheiden toestanden en zinvolle (!) acties:

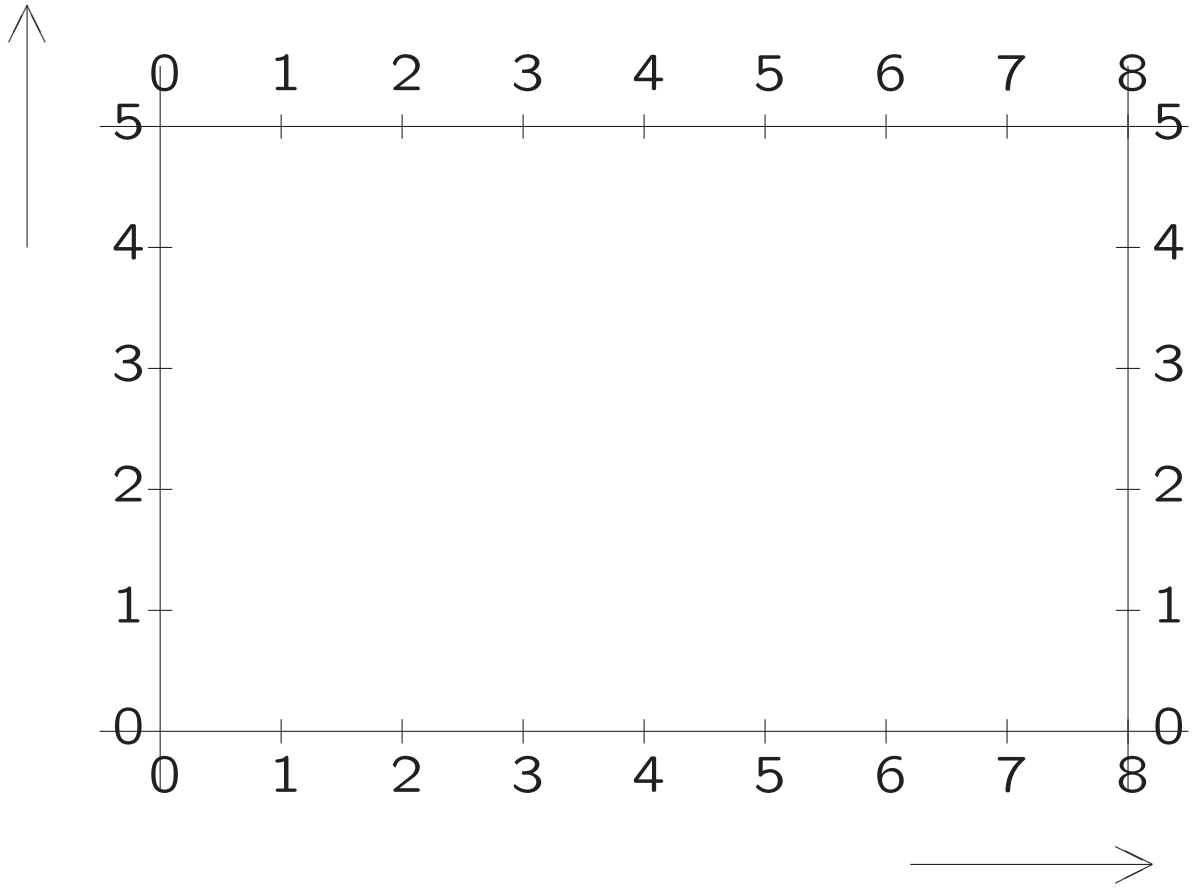
Toestand: Een paar (x, y) met $0 \leq x \leq 8$ en $0 \leq y \leq 5$. Hierin is x de inhoud van de grote kan en y de inhoud van de kleine kan.
Begintoestand: beide kannen leeg, dus (0,0)
Eindtoestanden: alle toestanden met 4 liter in een van beide kannen, dus (4, y) en (x, 4)

Acties: vullen, legen en overgieten

- een kan geheel (aan)vullen
- een kan geheel leeggooien
- de ene kan leeggooien in de andere
- van de ene kan in de andere gieten totdat deze vol is

Toestandsruimte

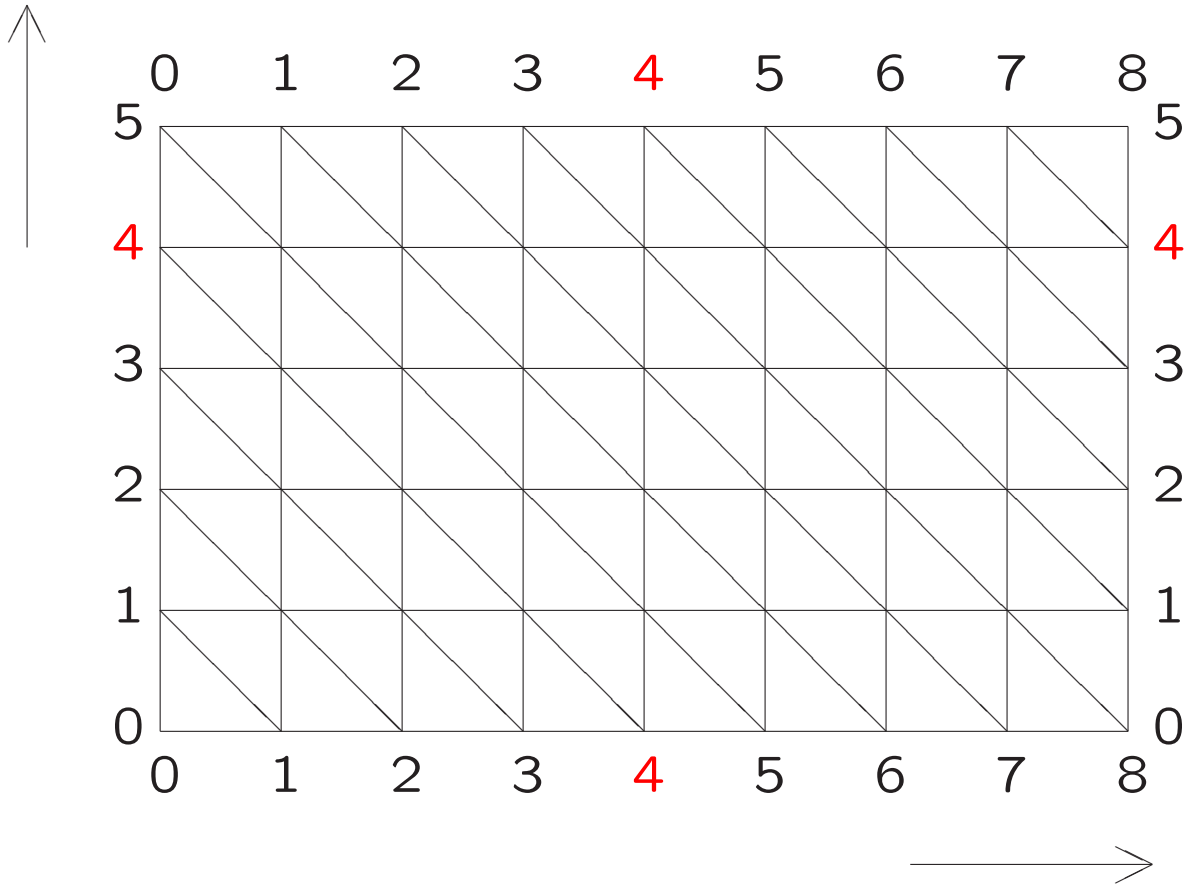
inhoud kleine kan



inhoud grote kan

... met mogelijke acties

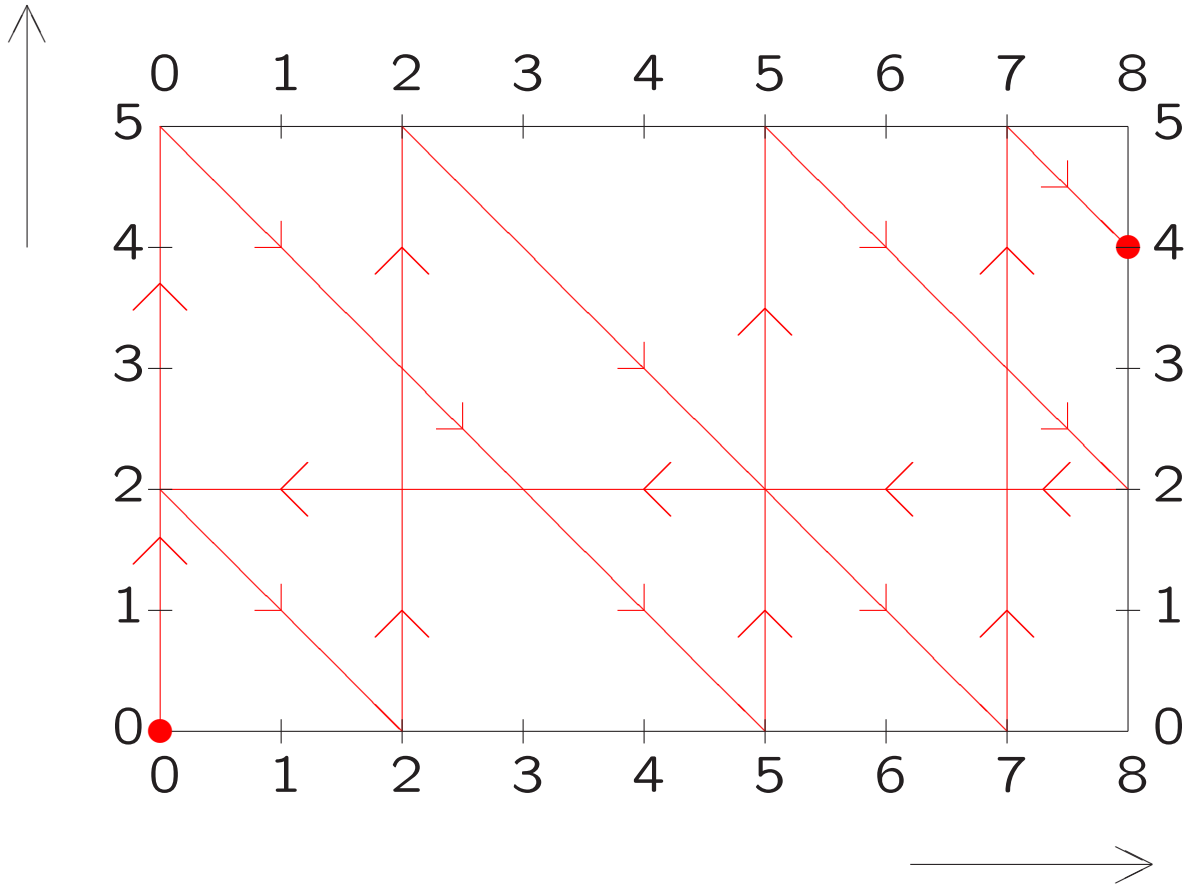
inhoud kleine kan



inhoud grote kan

Oplossing

inhoud kleine kan



inhoud grote kan

De snelste oplossing gebruikt de volgende strategie en zorgt voor 4 liter in de kleine kan. Er is overigens ook een (iets) langere oplossing, die 4 liter in de grote kan achterlaat.

```
Herhaal
    Herhaal
        Vul de kleine kan;
        Giet over in de grote kan;
    totdat de grote kan vol is
    Grote kan leeggooien;
    Giet uit de kleine kan over in de grote kan;
totdat oplossing gevonden
```

Zie verder het college.

Complexiteit (= tijdcomplexiteit) van een algoritme:

- = hoeveelheid werk verricht door het algoritme
- hangt meestal af van de grootte van de invoer: hoe groter de invoer, hoe groter de complexiteit
- wordt bepaald door het aantal keer dat de basisoperatie wordt uitgevoerd
- het belangrijkst is de (asymptotische) groei
- wordt vaak uitgedrukt in O-notatie (orde van grootte)
- hangt vaak ook af van het soort invoer: worst case, best case, average case

Voorbeeld 1:

```
// invoer: array a[0···n − 1] bestaande uit n reals
// uitvoer: de grootste waarde
```

```
max := a[0];
for i := 1 to n − 1 do
    if ( a[i] > max ) then ⟵ basisoperatie
        max := a[i];
    fi
od
return max;
```

Complexiteit: $C(n) = n - 1 \in \Theta(n)$

Voorbeeld 2:

```
// invoer: array a[0···n − 1] bestaande uit n reals
// uitvoer: true als alle n waarden verschillen
```

```
for i := 0 to n − 2 do
    for j := i + 1 to n − 1 do
        if ( a[i] = a[j] ) then ⟵ basisoperatie
            return false; fi
    od
od
return true;
```

Best case complexiteit: $B(n) = 1 \in \Theta(1)$

Worst case complexiteit:

$$W(n) = \sum_{i=0}^{n-2} n - 1 - i = \frac{1}{2} n(n-1) \in \Theta(n^2)$$

O-notatie beschrijft het asymptotisch gedrag:

$f \in O(g) \iff \exists c > 0$ en $\exists n_0 \geq 0$ zodat $\forall n > n_0$:

$$f(n) \leq c \cdot g(n)$$

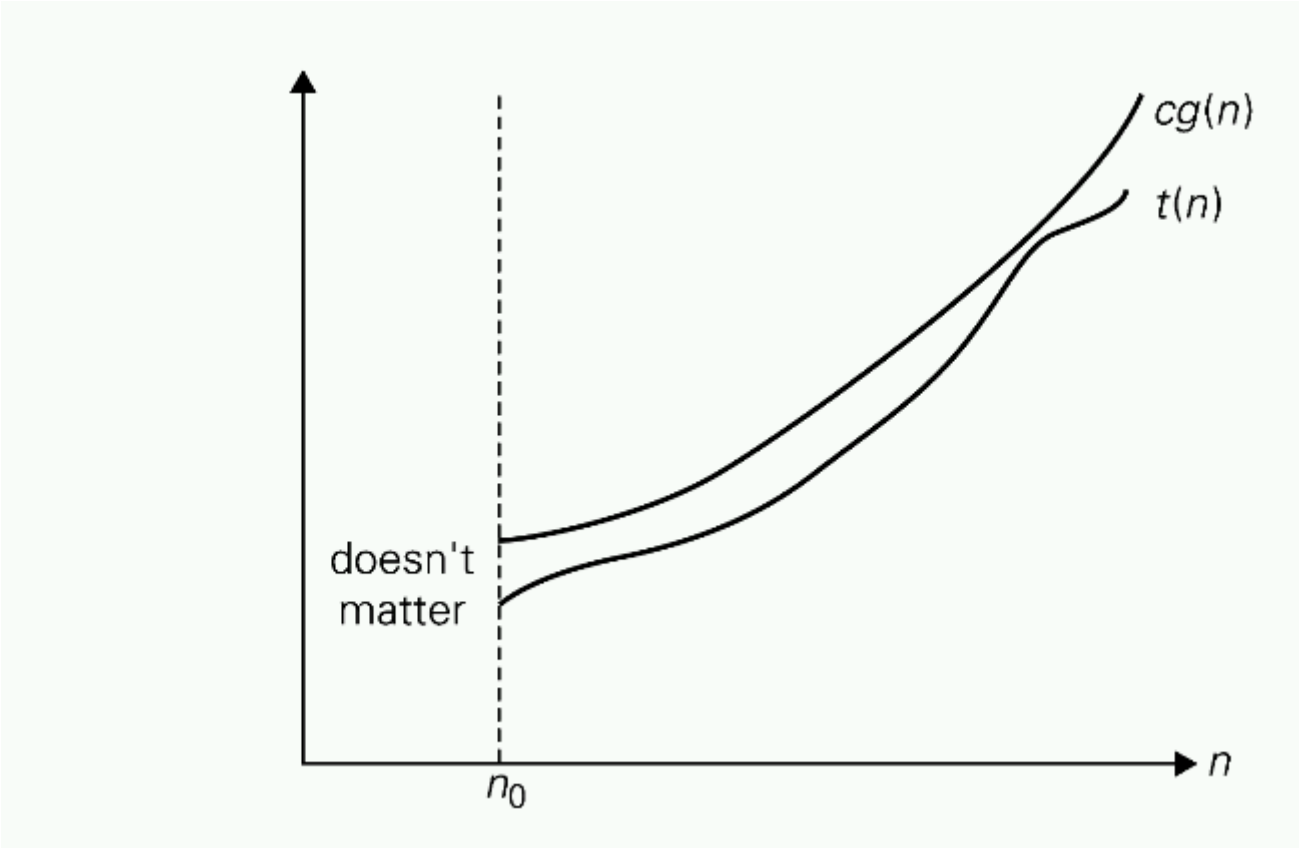
$f \in \Omega(g) \iff \exists c' > 0$ en $\exists n_0 \geq 0$ zodat $\forall n > n_0$:

$$f(n) \geq c' \cdot g(n)$$

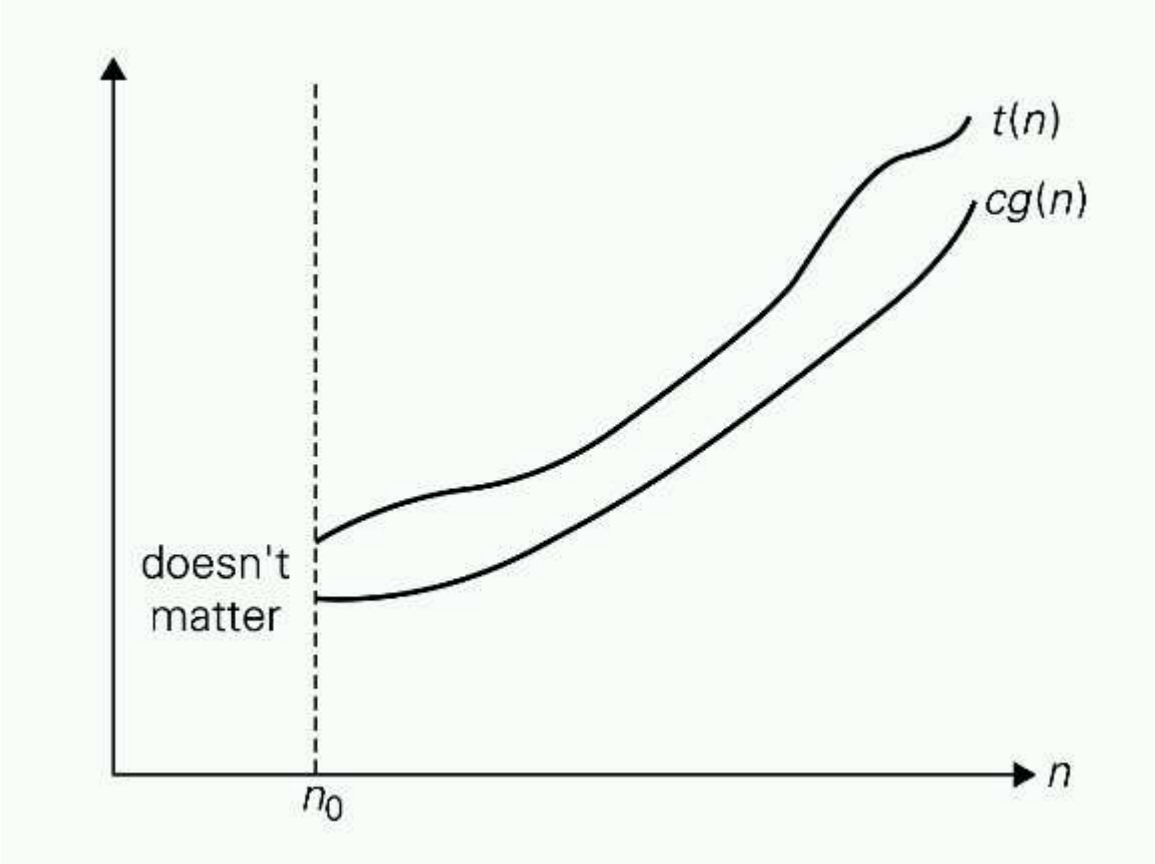
$f \in \Theta(g) \iff \exists c_1, c_2 > 0$ en $\exists n_0 \geq 0$ zodat $\forall n > n_0$:

$$c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$$

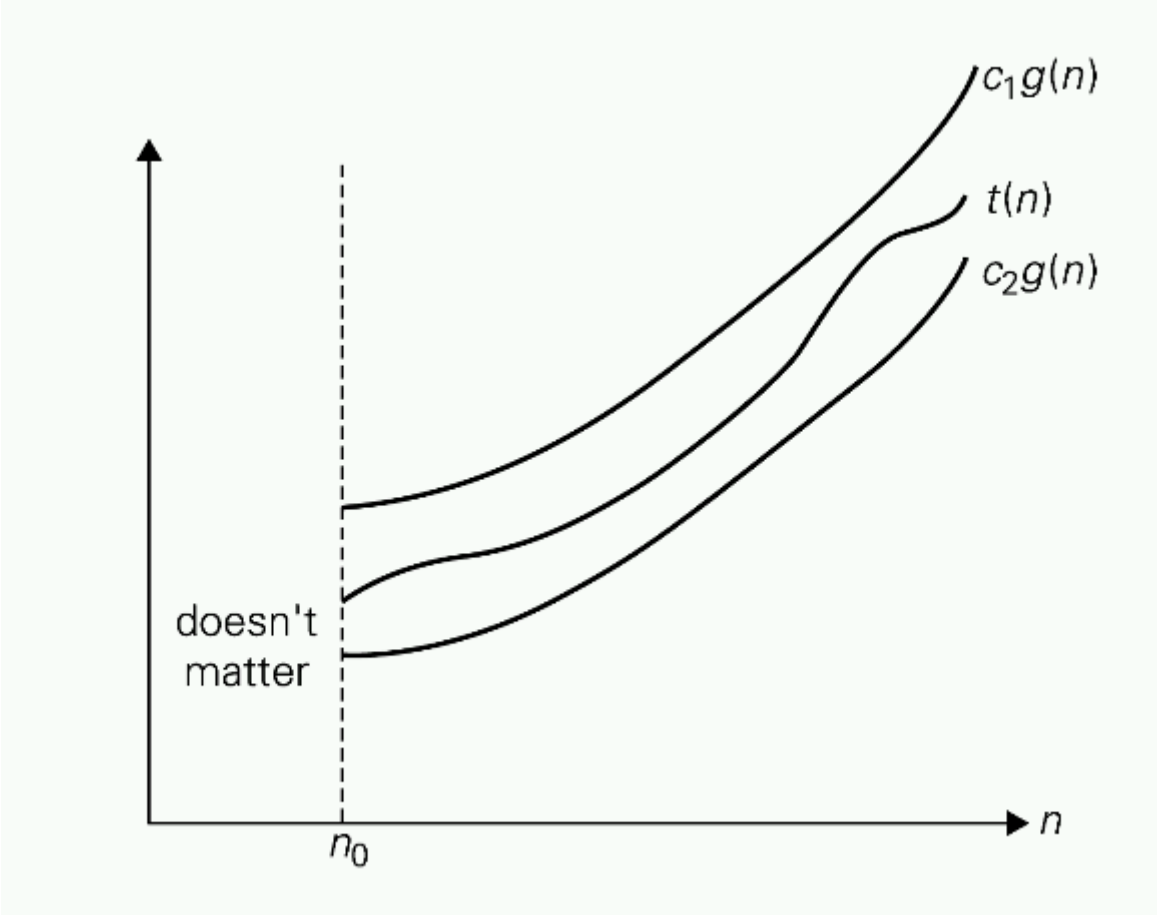
Aangezien f in onze toepassingen de complexiteit van een algoritme voorstelt is hier steeds $f > 0$.



$t \in O(g)$: $t(n)$ groeit niet sneller dan $g(n)$



$t \in \Omega(g)$: $t(n)$ groeit minstens zo snel als $g(n)$



$t \in \Theta(g)$: $t(n)$ groeit even snel als $g(n)$

N	10	50	100	300	1000
$\log_2 N$	3	5	6	8	9
$5N$	50	250	500	1500	5000
$N \cdot \log_2 N$	33	282	665	2469	9966
N^2	100	2500	10.000	90.000	7 cijfers
N^3	1000	125.000	7 cijfers	8 cijfers	10 cijfers
2^N	1024	16 cijfers	31 cijfers	91 cijfers	302 cijfers
$N!$	7 cijfers	65 cijfers	161 cijfers	623 cijfers	onvoorstelbaar
N^N	11 cijfers	85 cijfers	201 cijfers	744 cijfers	onvoorstelbaar

Ter vergelijking:

het aantal protonen in het heelal is een getal met 79 cijfers
het aantal microseconden sinds de Big Bang heeft 24 cijfers

Het is duidelijk uit de definitie dat geldt:

$$f \in \Theta(g) \iff f \in \Omega(g) \text{ en } f \in O(g)$$

Bovendien: $f \in O(g) \iff g \in \Omega(f)$

Stelling:

(1) $$0 < \lim_{n\to\infty} \frac{f(n)}{g(n)} < \infty \iff f \in \Theta(g)$$

(2) $$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0 \iff f \in O(g), \text{ maar } f \notin \Theta(g)$$

(3) $$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty \iff g \in O(f) \text{ maar } g \notin \Theta(f)$$

(1) $n(n-2) \in O(n^3)$; $n(n-2) \in O(n^2)$; $n(n-2) \in \Omega(n^2)$

(2) $n \in O(n^2)$, maar NIET $n \in \Theta(n^2)$

(3) $2^n \in O(3^n)$, maar NIET $2^n \in \Theta(3^n)$

(4) $(n^2+1)^{10} \in \Theta(n^{20})$

(5) $\sqrt{10n^2+7n+3} \in \Theta(n)$

(6) $2n \log_2(n+2)^2 + (n+2)^2 \log_2(n/2) \in \Theta(n^2 \log n)$

(7) $2^{n+1} + 3^{n-1} \in \Theta(3^n)$

(8) $\lfloor \log_2 n \rfloor \in \Theta(\log_2 n)$; $\log_{10} n \in \Theta(\log_2 n)$

(9) $2^n \in O(n!)$, maar NIET $2^n \in \Theta(n!)$

De volgende naamgeving wordt meestal gehanteerd:

1	constant
$\log n$	logaritmisch
n	lineair
$n \log n$	n-log-n
n^2	kwadratisch
n^α $(\alpha > 1)$	polynomiaal
2^n	exponentieel
$n!$, n^n, ...	superexponentieel

Voorbeeld 3:

```
// invoer: array a[0···n − 1] bestaande uit n reals en
//         een reeel getal k
// uitvoer: index i waarvoor a[i] = k; −1 als deze niet
//          bestaat
```

```
i := 0                                   ⟵ basisoperatie
while ( i < n and a[i] ≠ k ) do
      i := 1 + 1; od
if ( i < n )
      return i; fi
return −1;
```

best case/worst case/average case: ···

Recursief algoritme voor de berekening van $n!$:

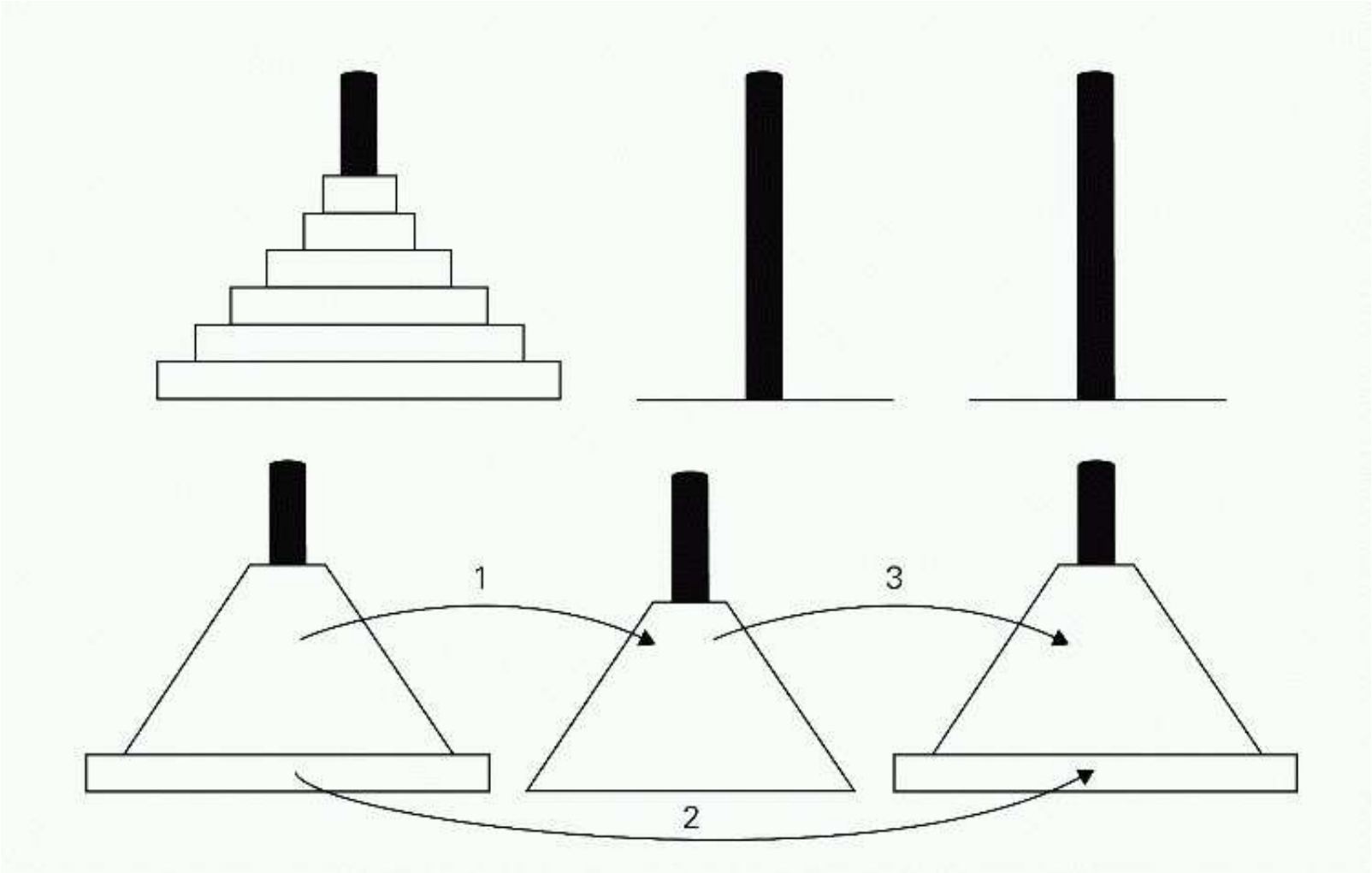
```
int faculteit (int n ) { // gebruikt: n! = n · (n − 1)!
  if ( n == 0 )
     return 1;
  else          ↓ ←———————— basisoperatie
     return n*faculteit(n-1);
}  // faculteit
```

$M(n)$ = aantal vermenigvuldigingen (= aantal recursieve aanroepen - 1) voldoet aan de recurrente betrekking:

$$\begin{cases} M(0) = 0 \\ M(n) = M(n-1) + 1 \quad \text{voor } n > 0 \end{cases}$$

Oplossing: $M(n) = n \longrightarrow$ complexiteit `faculteit` is $\Theta(n)$.

Oplossing Hanoi algemeen



Recursieve oplossing van de Torens van Hanoi

Een recursief algoritme voor het probleem van de Torens van Hanoi (zie Programmeermethoden):

```
// zet toren van n stuks (optimaal) van a naar b via c
// print de zetten
void zet (int n, int a, int b, int c) {
  if ( n > 0 ) {
    zet (n-1, a, c, b);
    cout << "zet van " << a << "naar " << b << endl;
    zet (n-1, c, b, a);
  }  // if
}  // zet
```

- n (het aantal schijven) is een maat voor de grootte van de invoer
- het verzetten van een schijf is de basisoperatie

Laat $M(n)$ = aantal zetten (= aantal recursieve aanroepen - 1), dan voldoet $M(n)$ aan de recurrente betrekking:

$$\begin{cases} M(0) = 0 \\ M(n) = 2M(n-1) + 1 \quad \text{voor } n > 0 \end{cases}$$

Oplossing (zie college):

$M(n) = 2^n - 1 \longrightarrow$ complexiteit `zet` is $\Theta(2^n)$.

- **Werkcollege:**

  donderdag 28 februari 2008 in zaal 174

- **Opgaven:**

  zie http://www.liacs.nl/home/graaf/ALGO/algo2008.html

- **Volgend college:**

  vrijdag 29 februari 2008