Derde college algoritmiek

17 februari 2021

Complexiteit

Toestand-actie-ruimte

Niet de bedoeling: globale (member-)variabele `teller`:

```
int binaireboom::aantalbladeren_p(knoop *ingang) {

  if (ingang != nullptr){
    if ((ingang->links == nullptr) && (ingang->rechts == nullptr))
      teller ++;
    else {
      aantalbladeren_p(ingang->links);
      aantalbladeren_p(ingang->rechts);
    }
  }

  return teller;

}//binaireboom::aantalbladeren_p
```

Wel de bedoeling: gebruik returnwaarde:

```
int binaireboom::aantalbladeren_p(knoop *ingang) {

  if (ingang == nullptr)
    return 0;
  if ((ingang->links == nullptr) && (ingang->rechts == nullptr))
    return 1;

  return aantalbladeren_p(ingang->links)
         + aantalbladeren_p(ingang->rechts);

}//binaireboom::aantalbladeren_p
```

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

Groei

algoritme	$C(n)$	$n = 10$	$n = 100$	$n = 1000$
algo1	$10n$	100	1000	10.000
algo2	$100n$	1000	10.000	100.000
algo3	$5n^2$	500	50.000	5.000.000

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: ...

Worst case complexiteit: ...

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 $\forall n > n_0$:

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

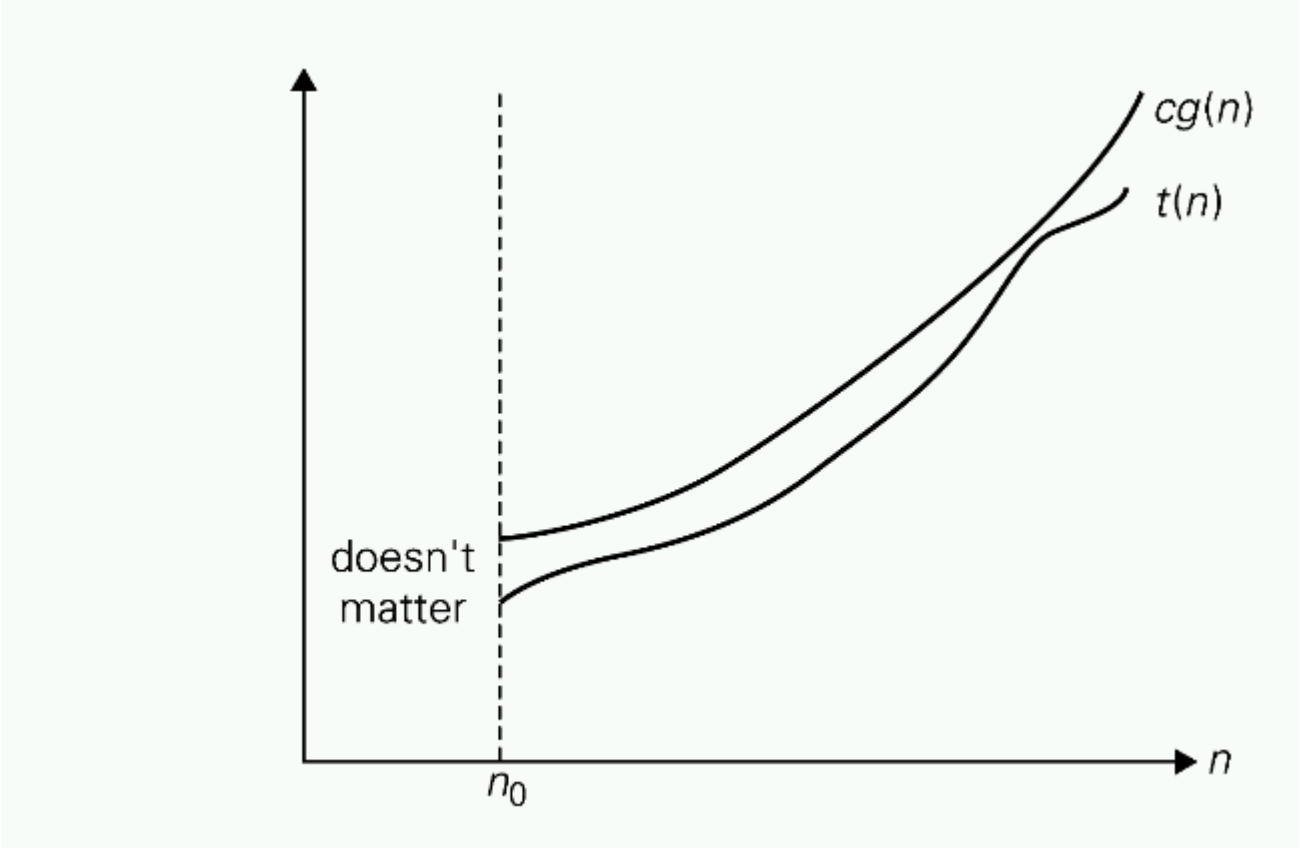
$f \in \Omega(g) \iff \exists c' > 0$ en $\exists n_0 \geq 0$ zodat $\forall n > n_0$:

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

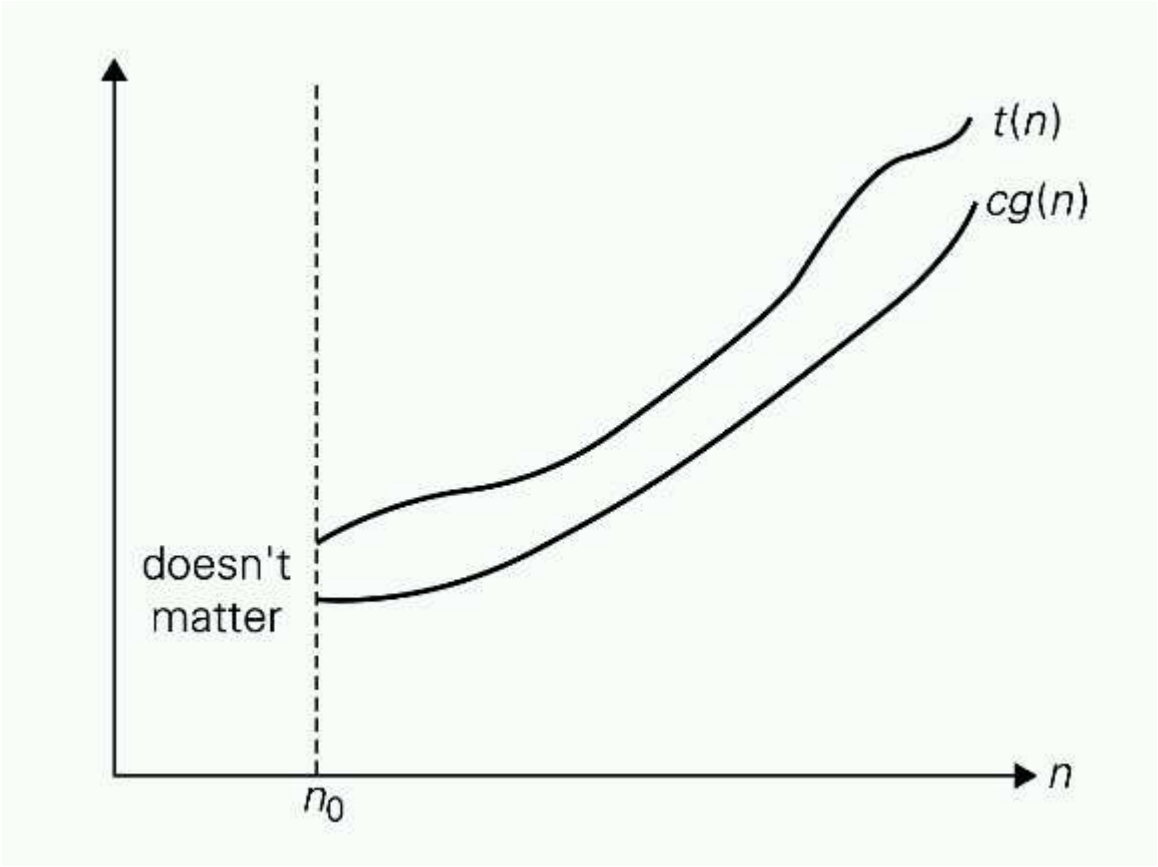
$f \in \Theta(g) \iff \exists c_1, c_2 > 0$ en $\exists n_0 \geq 0$ zodat $\forall n > n_0$:

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

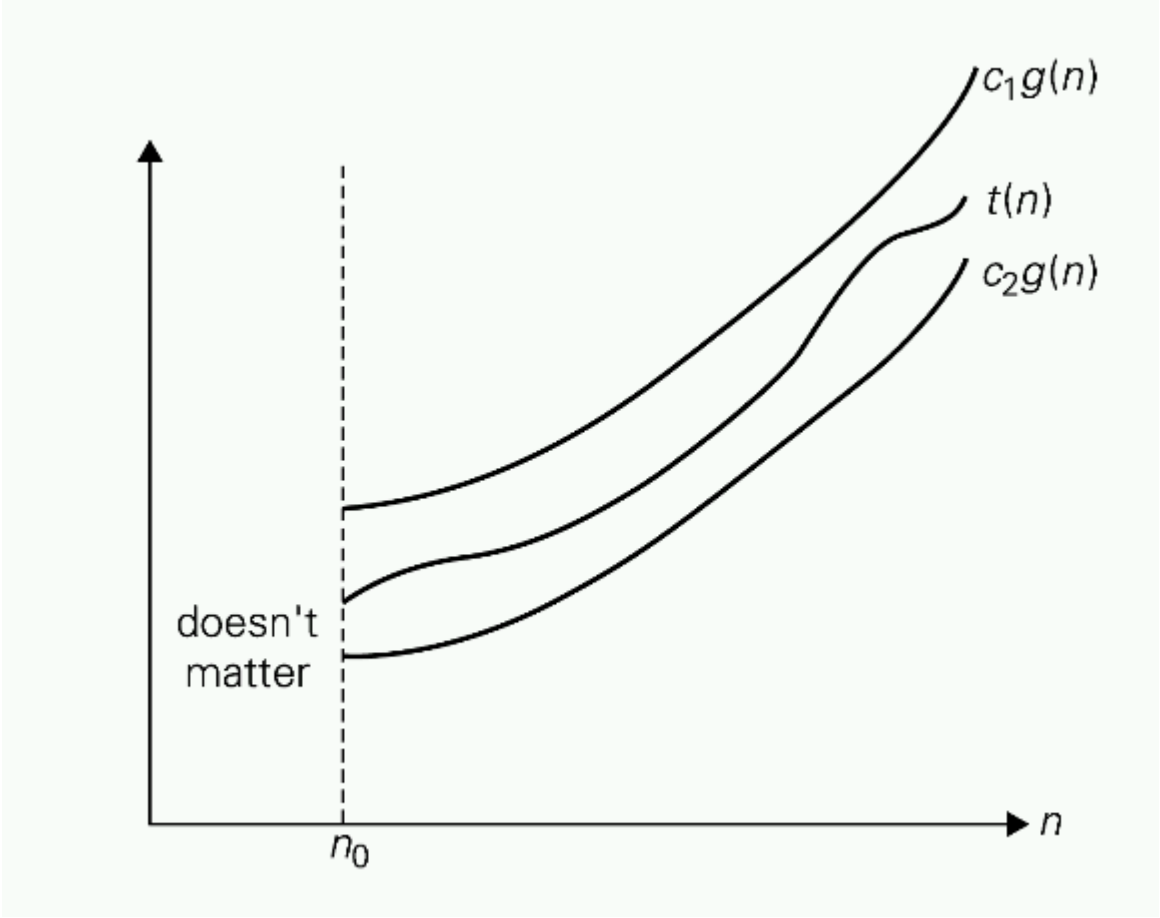
Aangezien zowel f als g in onze toepassingen de complexiteit van een algoritme voorstelt is hier steeds $f > 0$ en $g > 0$.



$t \in O(g)$: $t(n)$ 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 $g(n)$

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

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

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

(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** $a[i] \neq k$) **do**
    $i := i + 1$; **od**
if ($i < n$)
    return i; **fi**
return -1;

best case/worst case/average case: $\cdots$

Recursief algoritme voor de berekening van $n!$:

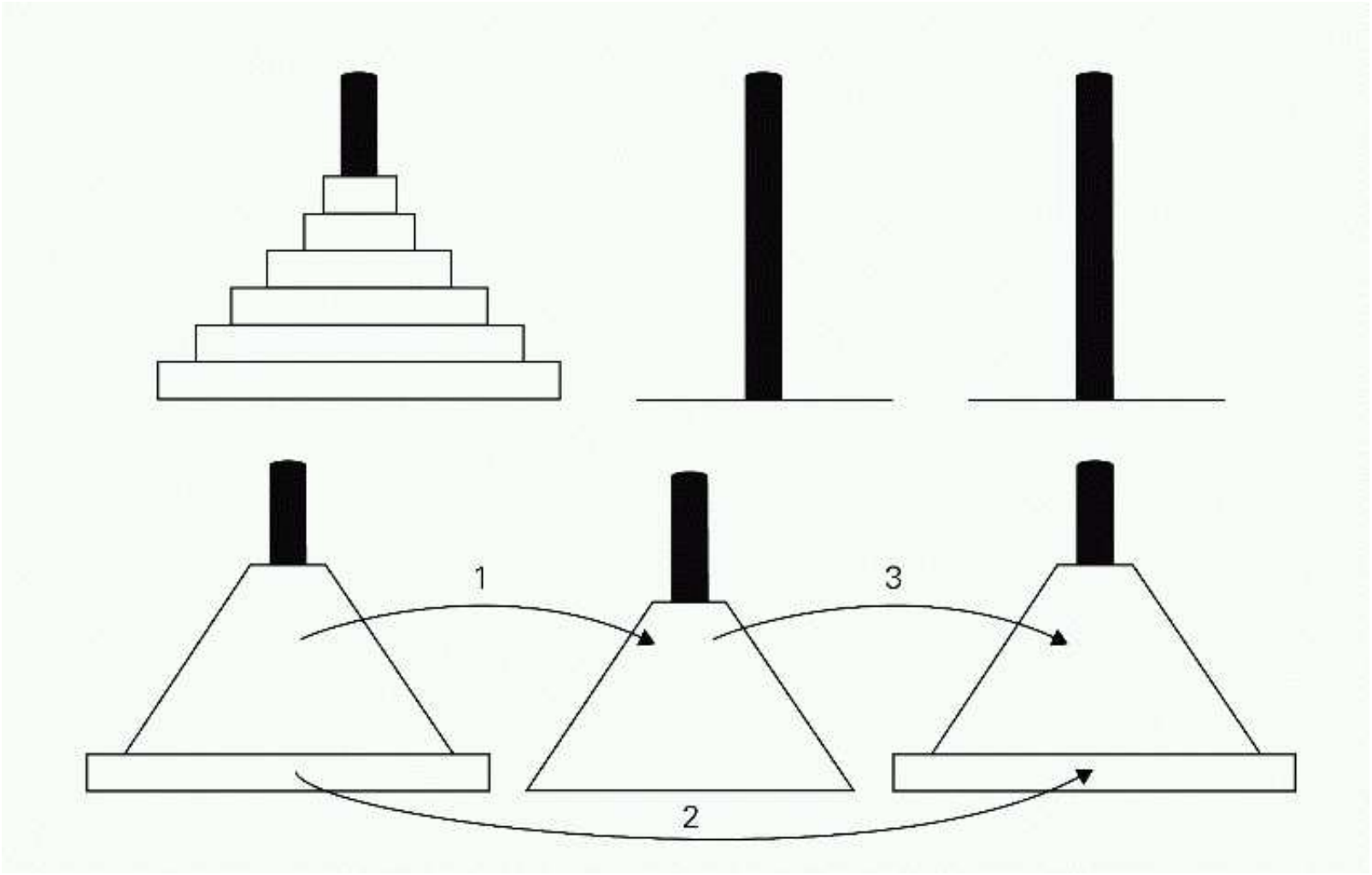
```
int faculteit ( int n ) { // gebruikt: n! = n · (n − 1)!
  if ( n == 0 )
     return 1;
  else          ↓ ←———————— basisoperatie
     return n*faculteit(n-1);
}  // faculteit
```

$M(n)$ = 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, 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 $\Theta(2^n)$.

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 1: Old world puzzle (Ex. 1.2.1.)

We hebben een kool, een geit, een wolf en een boer. Deze moeten met een bootje van de ene kant van de rivier naar de andere. In het bootje kan alleen de boer met één ander iets. Als de boer er niet bij is zal de wolf de geit opeten en de geit de kool. De boer is de enige die de boot kan "besturen".

Vraag: Hoe kan alles naar de andere oever verplaatst worden?

Merk op: de boot ligt altijd aan de oever waar de boer zich bevindt.

De oplossing is een kortste pad van de begintoestand naar de eindtoestand: hier zijn er twee, bij beide moet de boer 7 keer de rivier oversteken.

Merk op: de boot ligt altijd aan de oever waar de boer zich bevindt.

De oplossing is een kortste pad van de begintoestand naar de eindtoestand: hier zijn er twee, bij beide moet de boer 7 keer de rivier oversteken.

Een leuke variant op dit probleem is het volgende:

We hebben drie professoren en drie studenten. Deze moeten allemaal met een bootje van de ene kant van de rivier naar de andere. In het bootje kunnen hooguit twee personen. Op beide oevers mogen de professoren niet in de meerderheid zijn, anders worden de studenten nerveus.

Vraag: Hoe kan iedereen naar de andere oever verplaatst worden?

Merk op dat in tegenstelling tot het boer-wolf-geit-kool-probleem hier iedereen de boot kan "besturen". Er moet nu dus in een toestand worden aangegeven waar de boot ligt.

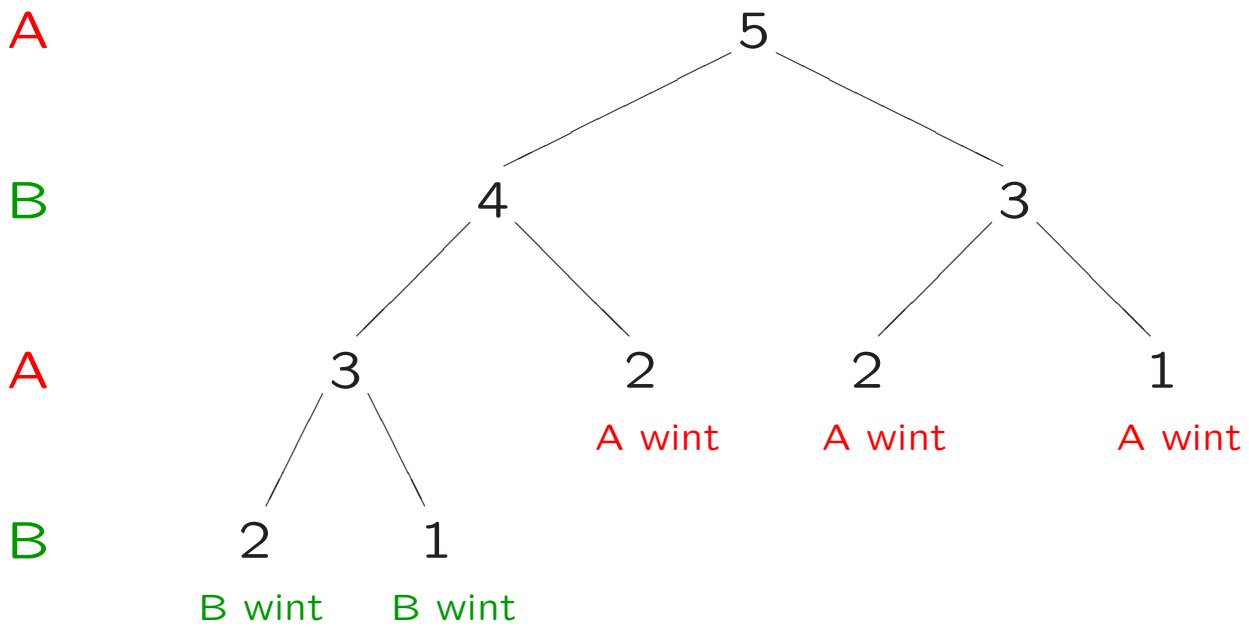
Er zijn 4 verschillende oplossingen, elk met 11 keer overvaren.

Voorbeeld 2: NIM

- We beginnen met één stapel van n lucifers (begintoestand)
- Er zijn twee spelers: A en B
- De spelers pakken om de beurt 1 of 2 lucifers (acties)
- Het spel is afgelopen als er geen lucifers meer op de stapel liggen (eindtoestand)
- De speler die de laatste lucifer(s) pakt heeft gewonnen

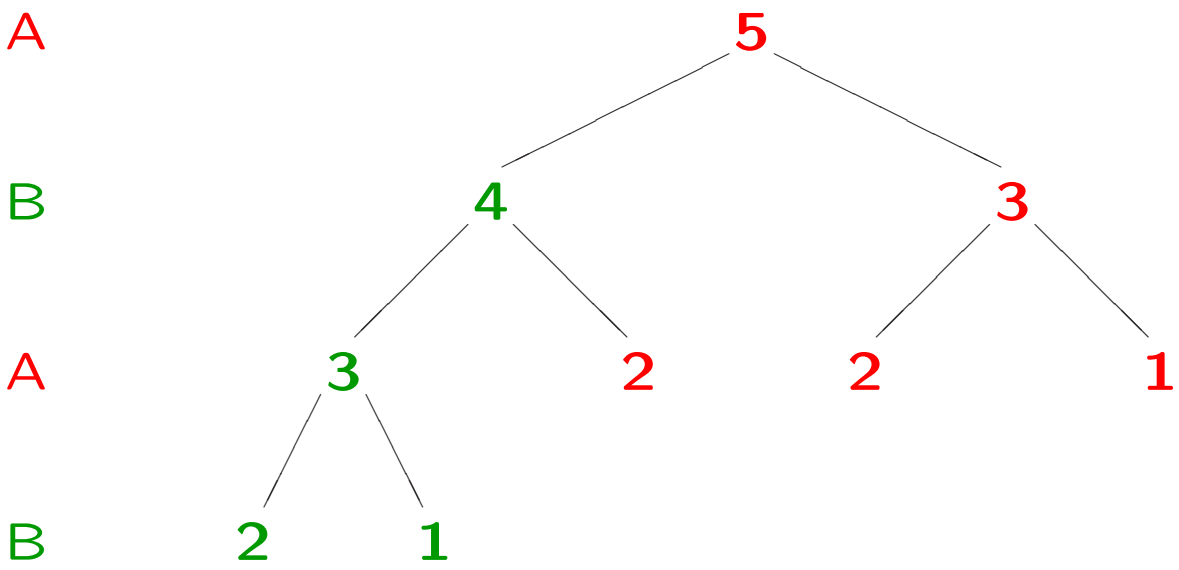
Gevraagd: is het spel winnend voor degene die begint?

toestand-actie-ruimte (spelboom) $n = 5$

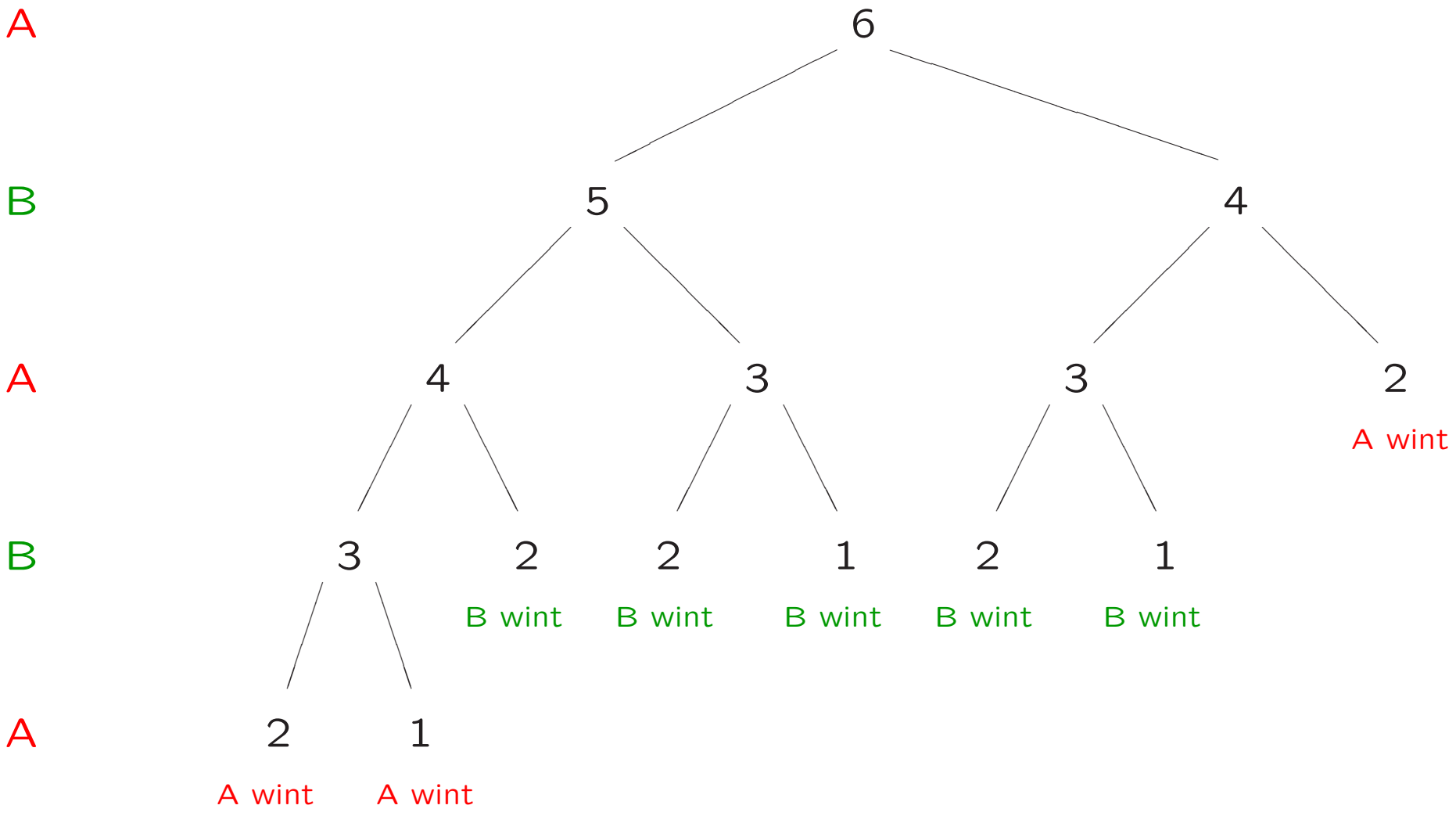


Winnend voor A.

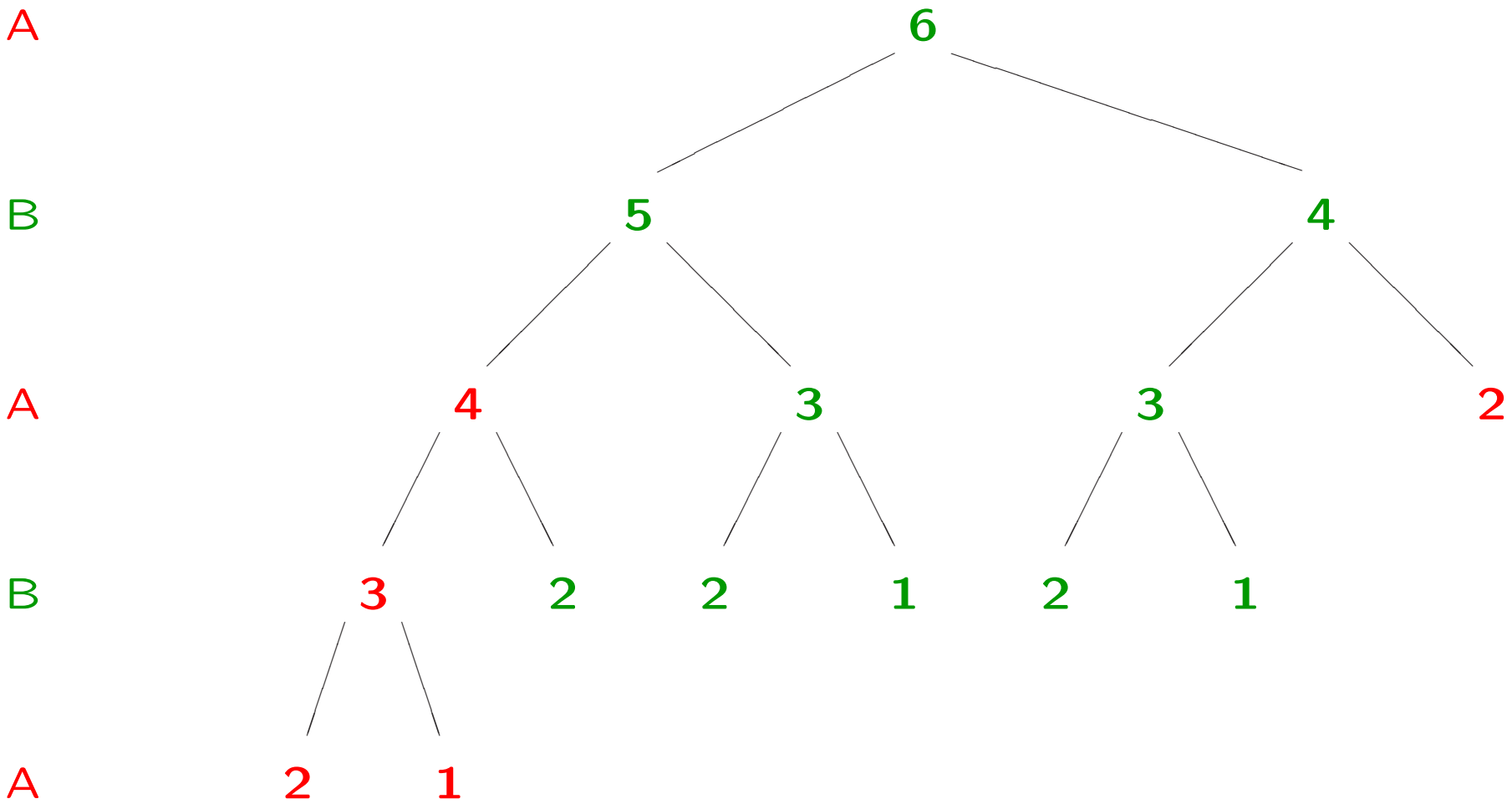
Winnende zet: 2 lucifers wegnemen



toestand-actie-ruimte (spelboom) $n = 6$



A kan niet winnen (bij perfect spel van B)



Een stand is winnend voor degene die aan de beurt is
als een van de directe (= in 1 zet te bereiken) vervolgstanden
NIET winnend is voor de tegenstander.

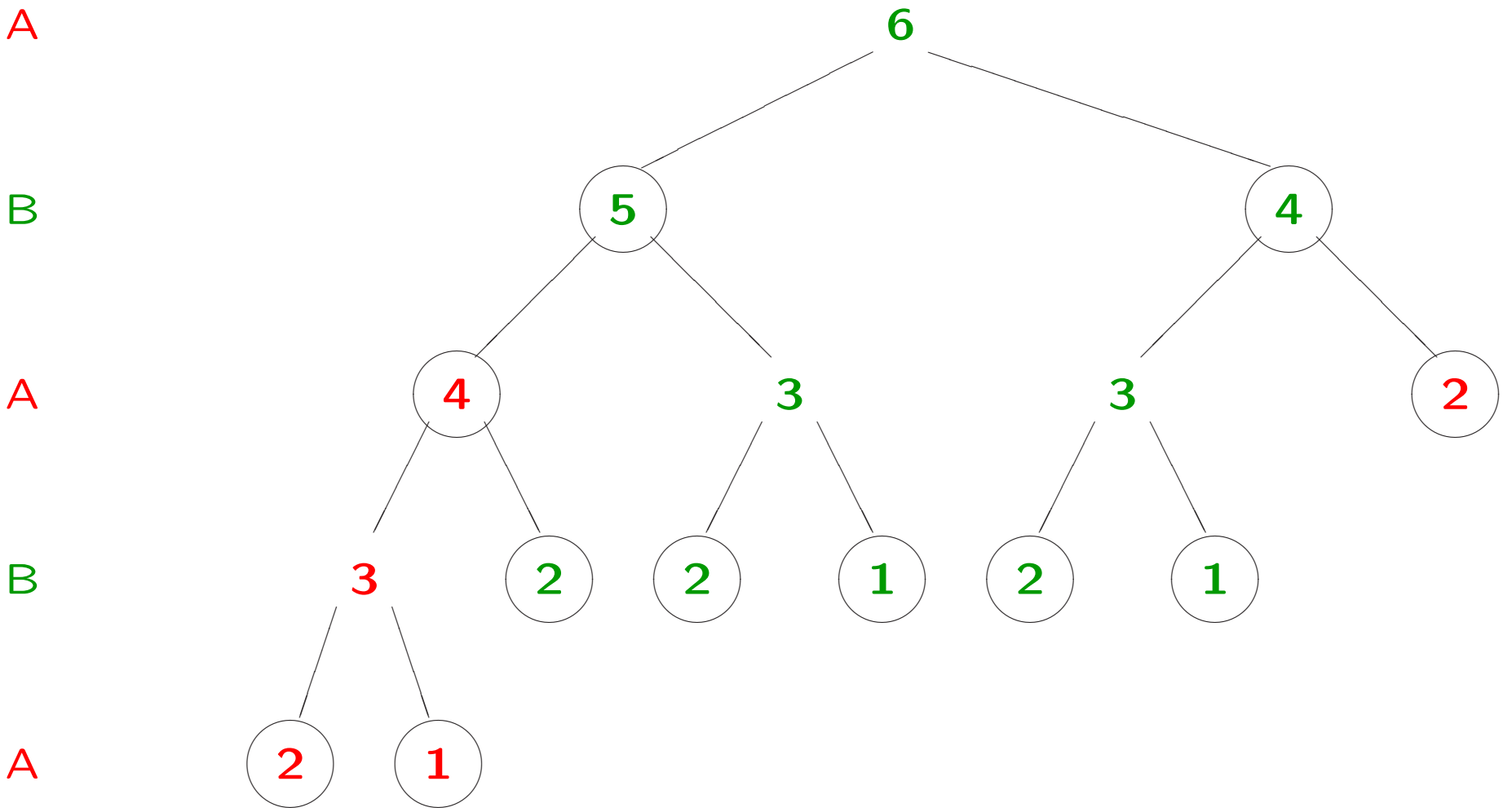
Een algoritme dat bepaalt of een stand winnend is, ziet er dus ruwweg zo uit:

Loop alle mogelijke directe vervolgstanden af:

- kijk of je er een tegenkomt die not winnend is voor de tegenstander: recursie
- zo ja, dan is de oorspronkelijke stand winnend (en heb je meteen een winnende zet) en hoef je niet verder te kijken
- zo nee, dan de volgende vervolgstand proberen

Als alle vervolgstanden zijn geweest is de oorspronkelijke stand niet winnend.

A kan niet winnen (bij perfect spel van B)



```
winnend(stand)::

    if eindstand(stand) then
        //  makkelijk; bijv return false;
    else
        for alle mogelijke zetten i do
            kopie := stand;
            doezet(kopie,i);
            if not winnend(kopie) then
                return true;
            fi
        od
        return false;
    fi
```

Zie ook Programmeermethoden (college over recursie)

Gebruik een kopie en doe daarin de zetten:

```
bool nimwinst (int stand) {
    int lucifer, kopie;
    if ( stand == 0 )     // de tegenstander heeft zojuist
        return false;     // de laatste lucifers gepakt
    else {           // directe vervolgstanden aflopen
        for ( lucifer = 1; lucifer <= 2; lucifer++ ) {
            kopie = stand;  // maak een kopie
            kopie -= lucifer; // doe een zet in de kopie
            if ( !nimwinst (kopie) )
                return true;
        }
        return false;
    }  // else
}
```

Met terugzetten:

```
bool nimwinst (int stand) {
    int lucifer;
    if ( stand == 0 )     // de tegenstander heeft zojuist
        return false;     // de laatste lucifers gepakt
    else {           // directe vervolgstanden aflopen
        for ( lucifer = 1; lucifer <= 2; lucifer++ ) {
            stand -= lucifer; // doe een zet
            if ( !nimwinst (stand) ) {
                stand += lucifer; // terugzetten
                return true;
            }
            stand += lucifer; // terugzetten
        }
        return false;
    }  // else
}
```

Met terugzetten:

```
winnend(stand)::

    if eindstand(stand) then
        //  makkelijk; bijv return false;
    else
        for alle mogelijke zetten i do
            doezet(stand,i);
            if not winnend(stand) then
                undoezet(stand,i);
                return true;
            fi
            undoezet(stand,i);
        od
        return false;
    fi
```

Zie ook Programmeermethoden (college over recursie)

Voorbeeld 3: Torens van Hanoi

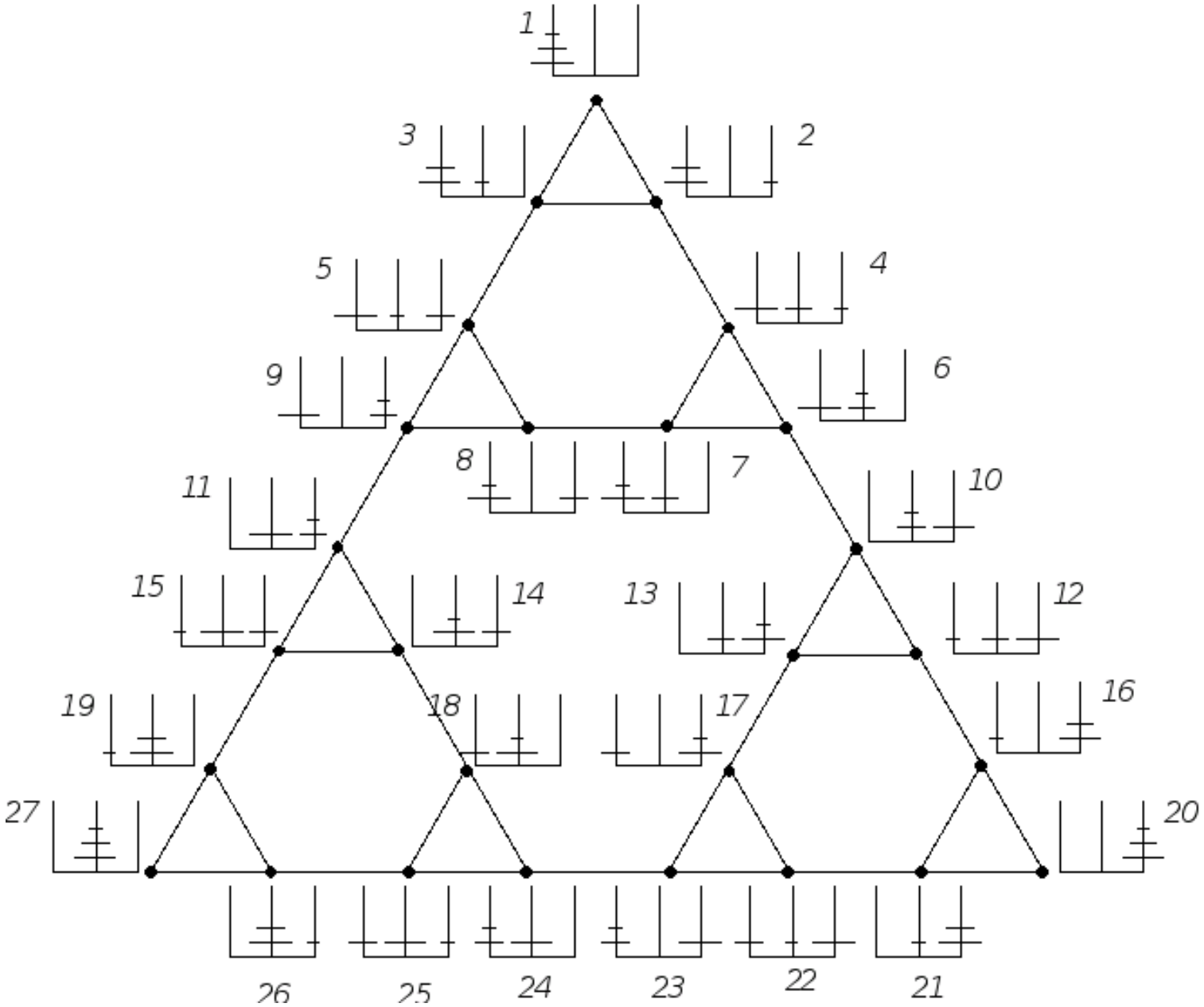
Gegeven n ($n \geq 1$) schijven, alle verschillend in grootte, en 3 palen. In de beginsituatie liggen alle schijven boven op elkaar om één paal, waarbij er geen grotere schijf op een kleinere ligt. De andere 2 palen zijn leeg.

Opdracht: Breng de hele toren (zo snel mogelijk) naar een van de lege palen door het een voor een verplaatsen van schijven van de ene paal naar de andere.

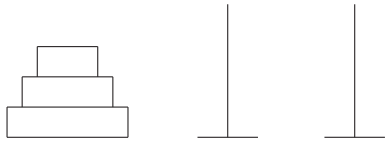
Regels:

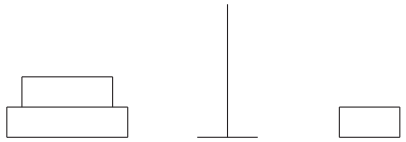
- Alleen de *bovenste* schijf van een stapel kan verzet worden
- deze mag alleen *bovenop* een andere stapel gelegd worden.
- *Restrictie*: er mag nooit een grotere schijf op een kleinere gelegd worden.

Een **toestand** is in dit geval een verdeling van de schijven over de palen, waarbij (als gevolg van de restrictie) geen grotere schijf op een kleinere ligt. Een **actie** is het verplaatsen van een schijf volgens de spelregels.

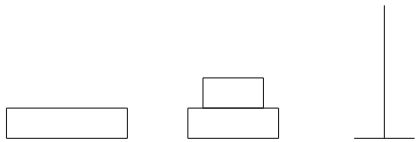

1
3
2
5
4
9
6
8
7
11
10
15
14
13
12
19
18
17
16
27
20
26
25
24
23
22
21

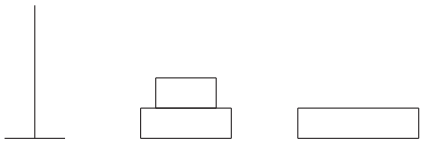
Optimale oplossing voor $n = 3$.

1. 

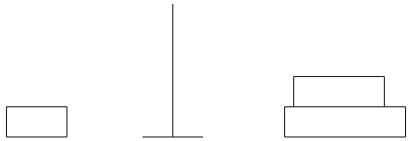
2. 

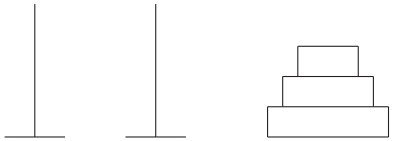
3. 

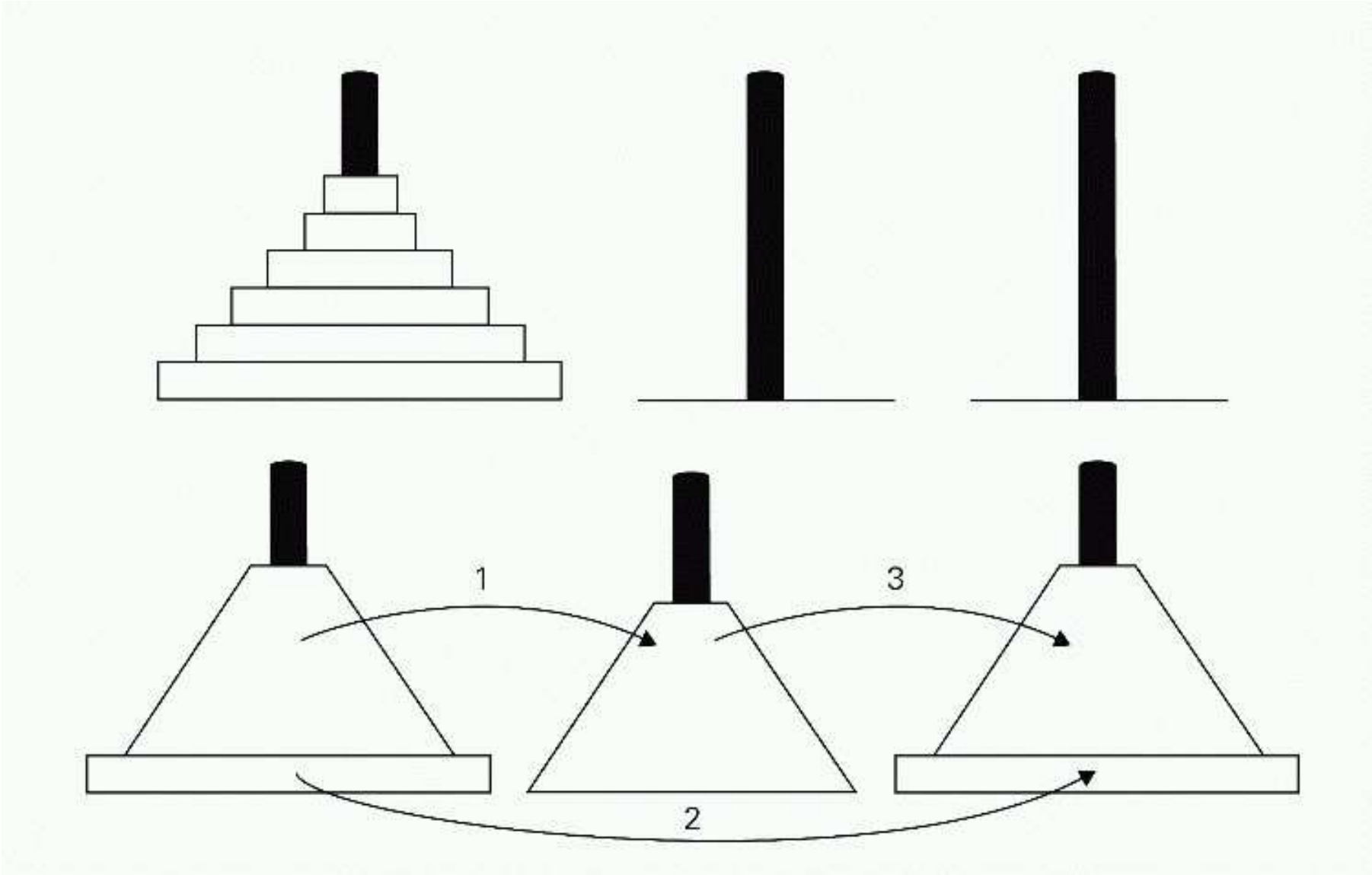
4. 

5. 

6. 

7. 

8. 



Recursieve oplossing van de Torens van Hanoi

- **Lezen/leren bij dit college:**
  Paragraaf 2.1–3
  Paragraaf 6.6 (subparagraaf 'Reduction to Graph Problems')
  Paragraaf 4.5 (subparagraaf 'The Game of Nim')

- **Werkcollege:**
  donderdag 18 februari 2021, 14.15–15.00 (weblecture)

- **Interactief vragenuur:**
  donderdag 18 februari, 15.15–16.00

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

- **Volgend college:**
  woensdag 24 februari 2021, 14.15–16.00, weblecture

- Het onderwerp Complexiteit en recursie (Levitin 2.4 en slides 18–21) slaan we over. Hier komen we in het tweede jaar Informatica nog op terug. Zie ook het tweedejaars Informatica-college Complexiteit.