

Rekursie

Bachelor IT

Sven Mariën

(sven.marien01@ap.be)

Recursie ?

Recursie

Recursie (Latijn: *recurrere*, 'teruglopen') is het optreden van een opeenvolging van constructies waarvan elk afzonderlijk gebaseerd is op een of meer soortgelijke voorgaande constructies. Doorgaans verschilt de volgende constructie in waarde van de voorgaande en is er een beginpunt. Recursieve constructies komen enerzijds in de **taalkunde** voor en anderzijds in de **wiskunde**, **informatica** en **logica**.

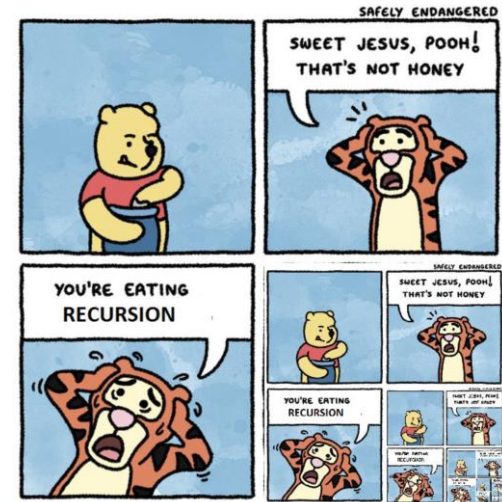
Een speciaal geval van recursiviteit is het **droste-effect**, waarbij een volgende constructie een verkleind beeld is van de voorgaande.



Recurisie bij programmeren

In [computer science](#), **recursion** is a method of solving a problem where the solution depends on solutions to smaller instances of the same problem.^[1] Such problems can generally be solved by [iteration](#), but this needs to identify and index the smaller instances at programming time. Recursion solves such [recursive problems](#) by using [functions](#) that call themselves from within their own code. The approach can be applied to many types of problems, and recursion is one of the central ideas of computer science.^[2]

- Recursie is een probleem kunnen opsplitsen in een eenvoudigere versie van zichzelf
- Bij recursie roept een methode zichzelf aan
- Alternatief voor “iteratie” (bv. **for** lus, **while** lus,..)



Recurisie = eindig

- Een functie die zichzelf aanroept zou in theorie leiden tot een oneindige “loop”
 - In de praktijk => stackoverflow exception
- Daarom dat elke recursieve functie **minstens 1 conditie** moet bevatten die deze recursieve lus doorbreekt.
 - Dit noemt men de “**base case**”

```
public void GoingCrazy (int forHowLong)
{
    GoingCrazy(forHowLong - 1);
}
```

```
public void GoingCrazy (int forHowLong)
{
    if (forHowLong > 0)
        GoingCrazy(forHowLong - 1);
}
```

A recursive function definition has one or more *base cases*, meaning input(s) for which the function produces a result *trivially* (without recurring), and one or more *recursive cases*, meaning input(s) for which the program recurs (calls itself).

Voorbeeld: een biertoren

- Hoeveel blikjes hebben we nodig om een toren te bouwen met basis = 4 blikjes ?

- $Aantal_4 = 4 + 3 + 2 + 1$

- Hoe omzetten naar c# code ?



```
/// <summary>
/// Calculate the total number of beer cans in a tower with the given base
///   I
///  I I
/// I I I
/// I I I I   (<= baseNumber)
/// </summary>
/// <param name="baseNumber">The number of cans at the base of the tower</param>
/// <returns>The total number of cans (including the base)</returns>
1 reference
public int NumberOfCans(int baseNumber)
{
    int Total = 0;
    //iterative solution (with for loop)
    for (int f = 1; f <= baseNumber; f++)
        Total += f;

    return Total;
}
```

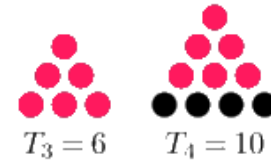
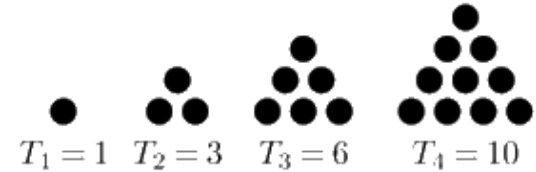
Een biertoren (2)

- Gelijkaardig kunnen we stellen dat:

- $\text{Aantal}_4 = 4 + 3 + 2 + 1$
- $\text{Aantal}_3 = 3 + 2 + 1$
- $\text{Aantal}_2 = 2 + 1$
- $\text{Aantal}_1 = 1$

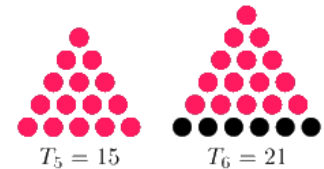
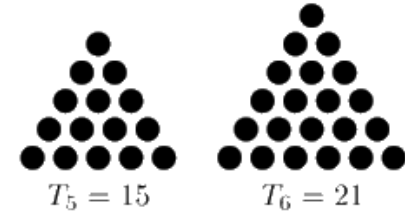
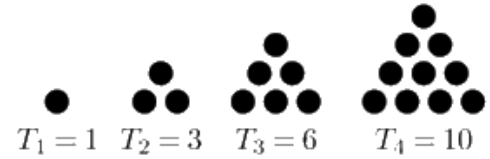
- Oftewel ook:

- $\text{Aantal}_4 = 4 + \text{Aantal}_3$
- $\text{Aantal}_3 = 3 + \text{Aantal}_2$
- $\text{Aantal}_2 = 2 + \text{Aantal}_1$
- $\text{Aantal}_1 = 1$



Een biertoren (3)

- Dus ook:
 - $\text{Aantal}_5 = 5 + \text{Aantal}_4$
 - $\text{Aantal}_6 = 6 + \text{Aantal}_5$
 -
- Algemeen kunnen we stellen dat:
 - Als n = aantal blikjes aan de basis
 - $\text{Aantal}_n = n + \text{Aantal}_{n-1}$

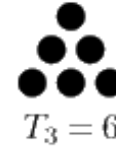


Een biertoren (4)

- Als we dit trachten om te zetten naar c# code:
 - `baseNumber` = aantal blikjes aan de basis
 - $\text{Aantal}_{\text{baseNumber}} = \text{baseNumber} + \text{Aantal}_{\text{baseNumber}-1}$

```
/// <summary>
/// Calculate the total number of beer cans in a tower with the given base
///   I
///  I I
/// I I I
/// I I I I (<= baseNumber)
/// </summary>
/// <param name="baseNumber">The number of cans at the base of the tower</param>
/// <returns>The total number of cans (including the base)</returns>
1reference
public int NumberOfCans(int baseNumber)
{
    int Total = 0;
    Total = baseNumber + NumberOfCans(baseNumber - 1);
    return Total;
}
```


Recurisie = eindig ?



```
public int NumberOfCans(int baseNumber) 3
{
    int Total = 0;
    Total = baseNumber + NumberOfCans(baseNumber - 1);
    return Total;
}
```

```
public int NumberOfCans(int baseNumber) 2
{
    int Total = 0;
    Total = baseNumber + NumberOfCans(baseNumber - 1);
    return Total;
}
```

```
public int NumberOfCans(int baseNumber) 1
{
    int Total = 0;
    Total = baseNumber + NumberOfCans(baseNumber - 1);
    return Total;
}
```

```
public int NumberOfCans(int baseNumber) 0
{
    int Total = 0;
    Total = baseNumber + NumberOfCans(baseNumber - 1);
    return Total;
}
```

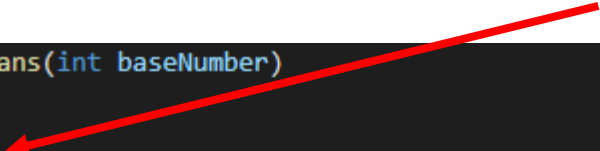
```
public int NumberOfCans(int baseNumber) -1
{
    int Total = 0;
    Total = baseNumber + NumberOfCans(baseNumber - 1);
    return Total;
}
```

....???

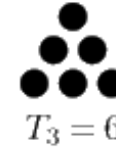
De “base case” ontbreekt !

- Even een stapje terugkijken:
 - $Aantal_3 = 3 + Aantal_2$
 - $Aantal_2 = 2 + Aantal_1$
 - $Aantal_1 = 1$ <= base case !

```
public int NumberOfCans(int baseNumber)
{
    int Total = 0;
    if (baseNumber == 1) return 1;           //base case
    Total = baseNumber + NumberOfCans(baseNumber - 1); //recursive part
    return Total;
}
```



Recurisie = eindig !!



=> 6 blikjes nodig voor basis = 3 !

Total = 3+3

```
public int NumberOfCans(int baseNumber) 3
{
    int Total = 0;
    if (baseNumber == 1) return 1;           //base case
    Total = baseNumber + NumberOfCans(baseNumber - 1); //recursive part
    return Total;
}
```

Total = 2+1

```
public int NumberOfCans(int baseNumber) 2
{
    int Total = 0;
    if (baseNumber == 1) return 1;           //base case
    Total = baseNumber + NumberOfCans(baseNumber - 1); //recursive part
    return Total;
}
```

```
public int NumberOfCans(int baseNumber) 1
{
    int Total = 0;
    if (baseNumber == 1) return 1;           //base case
    Total = baseNumber + NumberOfCans(baseNumber - 1); //recursive part
    return Total;
}
```

Iteratief versus recursief

```
public int NumberOfCans(int baseNumber)
{
    int Total = 0;
    //iterative solution (with for loop)
    for (int f = 1; f <= baseNumber; f++)
        Total += f;

    return Total;
}
```

```
1 reference
public int NumberOfCans(int baseNumber)
{
    int Total = 0;
    if (baseNumber == 1) return 1;           //base case
    Total = baseNumber + NumberOfCans(baseNumber - 1); //recursive part
    return Total;
}
```

Wat verwacht je als resultaat voor...

- NumberOfCans(0) ?
- NumberOfCans(-1) ?
- NumberOfCans(-2) ?
- ...
- Bv. dit algoritme werkt enkel voor **baseNumber ≥ 0**
 1. Baken de grenzen van de “input” duidelijk af.
 2. Geef dit aan in je “comment”
 3. Vang op vooraan in de code => “defensive programming”(Merk op: dit staat los van recursie)

```
/// <summary>
/// Calculate the total number of beer cans in a tower with the given base
///   I
///  I I
/// I I I
/// I I I I (<= baseNumber)
/// </summary>
/// <param name="baseNumber">The number of cans at the base of the tower. This number must be  $\geq 0$  </param>
/// <returns>The total number of cans (including the base)</returns>
Reference:
public int NumberOfCans(int baseNumber)
{
    int Total = 0;

    if (baseNumber == 0) return 0;
    if (baseNumber < 0) throw new Exception("baseNumber cannot be smaller than 0");

    if (baseNumber == 1) return 1;           //base case
    Total = baseNumber + NumberOfCans(baseNumber - 1); //recursive part
    return Total;
}
```

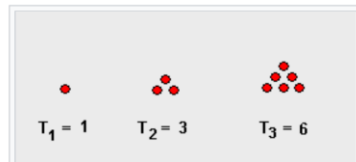
Andere oplossing(en) mogelijk ?

- Is er naast iteratie of recursie nog een andere mogelijke oplossing voor dit “probleem” ?
- Ja, onderzoek leert ons dat:
 - het hier blijkbaar gaat om het zogenaamde “**driehoeksgetal**”
 - een driehoeksgetal (T) kan berekend worden met **deze formule**: $T = n * (n+1) / 2$

Driehoeksgetal

```
/// <summary>
/// Calculate the total number of beer cans
///   I
///  I I
/// I I I
/// I I I I (<= baseNumber)
/// </summary>
/// <param name="baseNumber">The number of cans at the base of the tower. This number must be >= 0 </param>
/// <returns>The total number of cans (including the base)</returns>
1 reference
public int NumberOfCans(int baseNumber)
{
    ...
    return baseNumber * (baseNumber + 1) / 2;
}
```

Een **driehoeksgetal** is een type **veelhoeksgetal**. Een driehoeksgetal kan grafisch worden weergegeven door het aantal stippen in een **gelijkzijdige driehoek**, die gelijkmatig met stippen wordt gevuld. Aangezien bijvoorbeeld drie stippen in de vorm van een gelijkzijdige **driehoek** kunnen worden gelegd, is drie dus een driehoeksgetal.



Time and space complexity ?

```
public int NumberOfCans(int baseNumber)
{
    int Total = 0;
    //iterative solution (with for loop)
    for (int f = 1; f <= baseNumber; f++)
        Total += f;

    return Total;
}
```

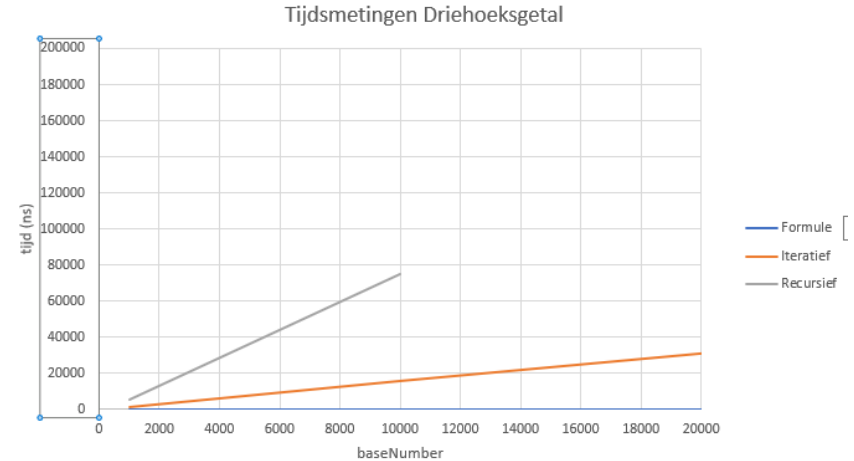
```
1 reference
public int NumberOfCans(int baseNumber)
{
    int Total = 0;
    if (baseNumber == 1) return 1;           //base case
    Total = baseNumber + NumberOfCans(baseNumber - 1); //recursive part
    return Total;
}
```

```
1 reference
public int NumberOfCans(int baseNumber)
{
    return baseNumber * (baseNumber + 1) / 2;
}
```

| Driehoeksgetal berekenen | Time complexity | Space complexity |
|--------------------------|-----------------|------------------|
| Iteratief algoritme | $O(n)$ | $O(1)$ |
| Rekursief algoritme | $O(n)$ | $O(n)$ |
| Algoritme met formule | $O(1)$ | $O(1)$ |

Enkele metingen...

- De metingen bevestigen enigszins de verwachtingen.
- De **formule** versie heeft een **constant** tijdsverloop
- De **iteratieve** en **recursieve** versie een **lineair** tijdsverloop
- De **recursieve** versie is echter **trager** (overhead van het aanroepen van een methode)
- De **recursieve** versie geeft er na een tijdje de brui aan (bij een baseNr = 12043) omwille van **stack overflow** problemen.



```
/// </summary>
/// <param name="baseNumber">The number of cans at the base of the tower. This number must be >= 0 </param>
/// <returns>The total number of cans (including the base)</returns>
2 references
public static int NumberOfCansR(int baseNumber)
{
    int Total = 0;

    if (baseNumber == 0) return 0;
    if (baseNumber < 0) throw new Exception("baseNumber cannot be smaller

    if (baseNumber == 1) return 1; //base case
    Total = baseNumber + NumberOfCansR(baseNumber - 1); //recursive pa
    return Total;
}
/// </summary>
```

baseNumber 7957 is

Exception Unhandled
System.StackOverflowException: "

View Details | Copy Details | Start Live Sha
Exception Settings

Directe versus indirecte recursie

- **Directe** recursie:
 - Een methode roept **zichzelf** aan
- **Indirecte** recursie:
 - Methode A roept methode B aan, die op zijn beurt terug methode A aanroept

```
public static int doOne(int n)
{
    if (n <= 0) {
        System.out.println("Hallo van One! n="+n+" dus ik stop ermee!");
        return 0;
    }
    else
        System.out.println("Hallo van One! n="+n);
        return n = n + doTwo(n-1);
}
```

```
public static int doTwo(int n)
{
    if (n <= 0) {
        System.out.println("Hallo van One! n="+n+" dus ik stop ermee!");
        return 0;
    }
    else
        System.out.println("Hallo van Two! n="+n);
        return n = n + doOne(n-2);
}
```

Voorbeeld 2: Towers of Hanoi

- Ook gekend als “The tower of Brahma”
- Is een puzzel waarbij:
 - Alle schijven van de 1^e naar de 3^e stok dienen te worden verplaatst.
- Spelregels:
 - **1 schijf** verplaatsen **per keer**.
 - Een schijf mag **enkel op een stok terecht komen** (dus niet ergens opzij gelegd worden)
 - Een schijf mag enkel **op een grotere schijf** terecht komen



FIGURE 11.5 The Towers of Hanoi puzzle

Oplossing met 3 disks

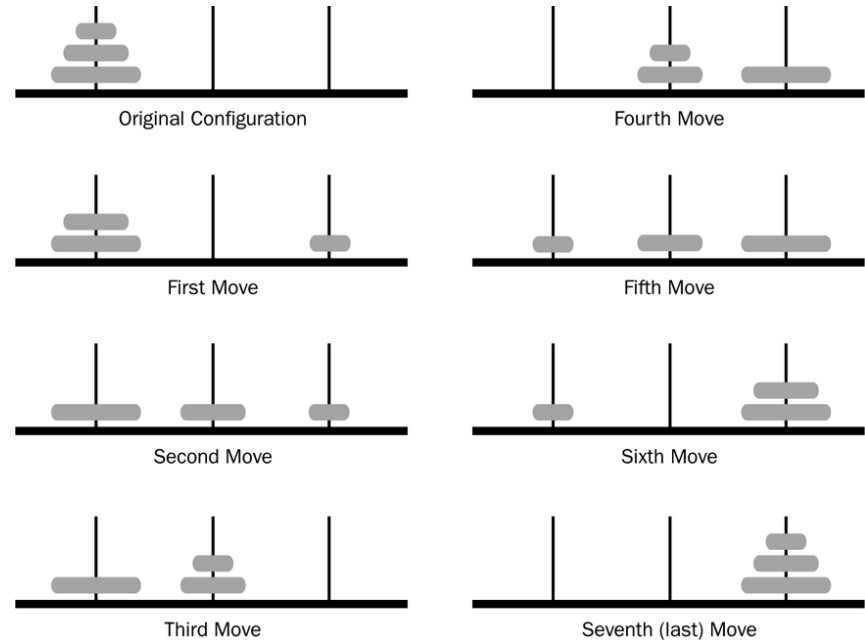
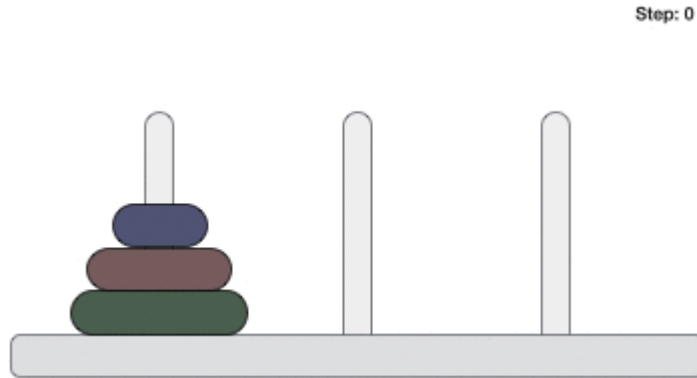
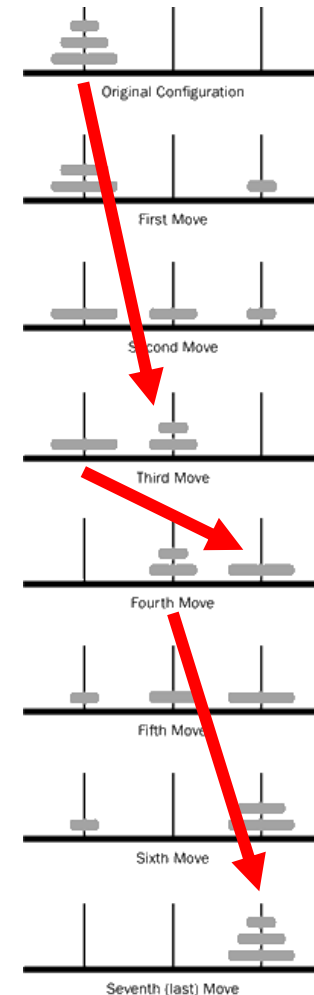


FIGURE 11.6 A solution to the three-disk Towers of Hanoi puzzle

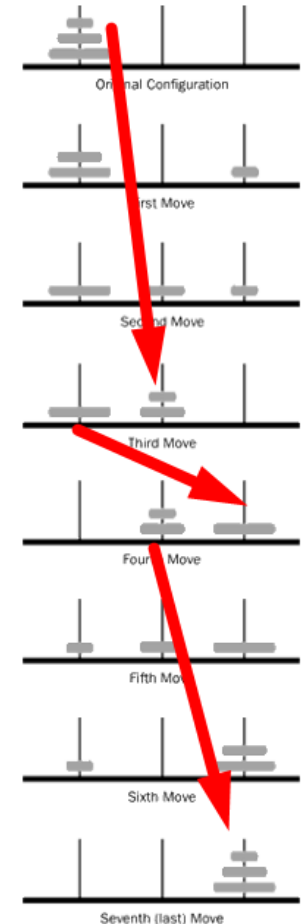
Algemeen principe

- Verplaats de bovenste $n-1$ disks naar de “hulpstok” (**3rd move**)
- Verplaats de grootste vervolgens naar de doelstok (**4th move**)
- Verplaats nadien de overige $n-1$ disks ook naar de doelstok (**last move**)



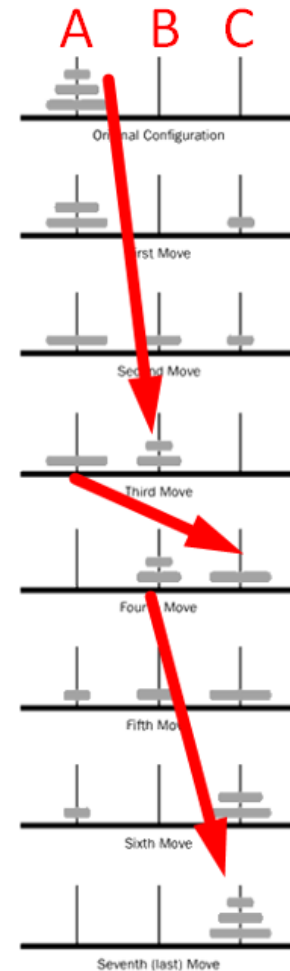
Algemeen principe (2)

- Dit principe geldt voor eender welk totaal aantal van schijven.
- Bv. om 4 schijven te verplaatsen naar de doelstok, moeten we eerste de 3^e bovenliggende verplaatsen naar de hulpstok, enz...
- We hebben het probleem hiermee dus opgesplitst in een eenvoudigere versie van zichzelf => nodig voor recursie



Uitwerking

- Stel:
 - We noemen de torens A, B & C
 - disk = het nummer (grootste is onderaan)
- We maken een methode:
 - VerplaatsToren (diskNr, bronStok, doelStok, hulpStok)
 - Deze methode zal alle stappen uitzoeken die nodig zijn om de disk (en alle bovenliggende) van de bron naar de doelstok te verplaatsen)
- Dan wordt de aanroep ervan:
 - bv. om **disk** nummer **3** te verplaatsen **van stok A naar stok C**
 - => VerplaatsToren (3, A, C, B)



Uitwerking (2)

- VerplaatsToren (diskNr, bronStok, doelStok, hulpStok)

VerplaatsToren (disk, A, C, B) {

- Verplaats de bovenste n-1 disks naar de "hulpstok" (3rd move)
 - VerplaatsToren (disk-1, A, B, C)
- Verplaats de grootste vervolgens naar de doelstok (4rd move)
 - **Doe hier de verplaatsing van disk van A naar C**
- Verplaats nadien de overige n-1 disks ook naar de doelstok (last move)
 - VerplaatsToren (disk-1, B, C, A)

}

- Wat missen we nog ?

- => de "base case"
- **Disk 1** kunnen we altijd verplaatsen vermits die altijd bovenaan ligt

VerplaatsToren (disk, A, C, B) {

If (disk == 1)

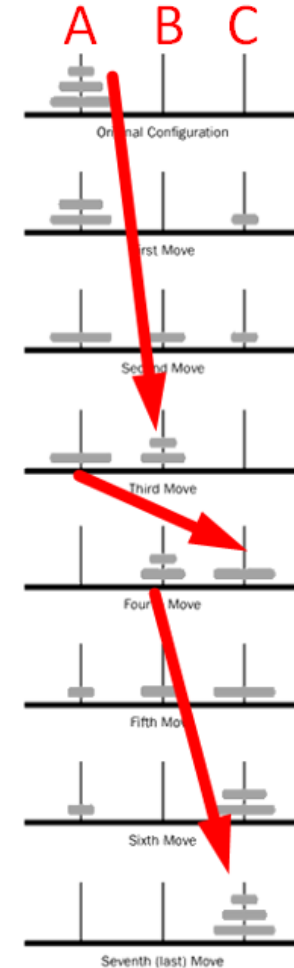
- **Doe hier de verplaatsing van disk van A naar C**

else {

- Verplaats de bovenste n-1 disks naar de "hulpstok" (3rd move)
 - VerplaatsToren (disk-1, A, B, C)
- Verplaats de grootste vervolgens naar de doelstok (4rd move)
 - **Doe hier de verplaatsing van disk van A naar C**
- Verplaats nadien de overige n-1 disks ook naar de doelstok (last move)
 - VerplaatsToren (disk-1, B, C, A)

}

}



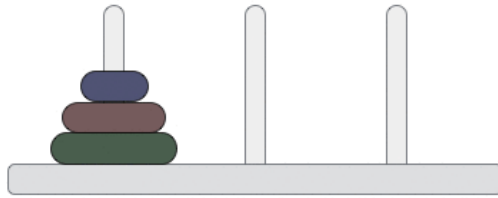
Uitwerking (3)

```
public class Tower {  
    public void startTower(int disks)  
    {  
        doTower(disks, 'A', 'B', 'C');  
    }  
  
    private void doTower(int n, char start, char auxiliary, char end) {  
        // De ring nummering is van klein naar groot  
        // De ring met het kleinste cijfer is ook de kleinste ring  
        // De ring met het grootste cijfer is het breedst  
        if (n == 1) {  
            System.out.println("Ring "+n+" moving from "+ start + " -> "+end);  
        }  
        else{  
            doTower(n-1,start,end,auxiliary);  
            System.out.println("Ring "+n+" moving from "+ start + " -> "+end);  
            doTower(n-1, auxiliary, start , end);  
        }  
    }  
}
```

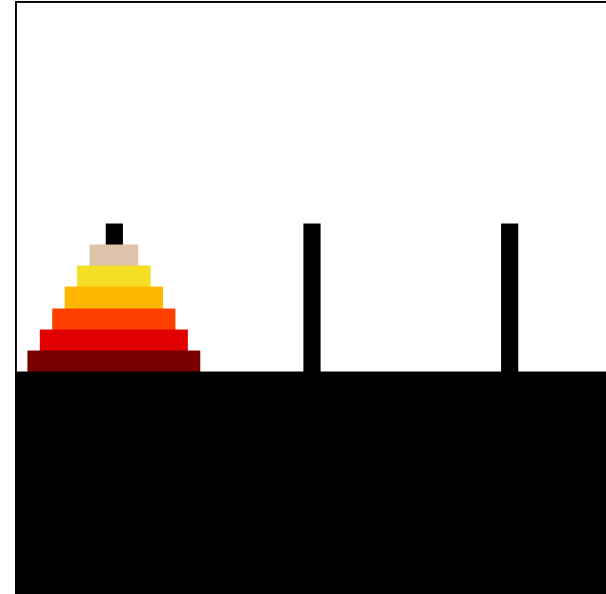
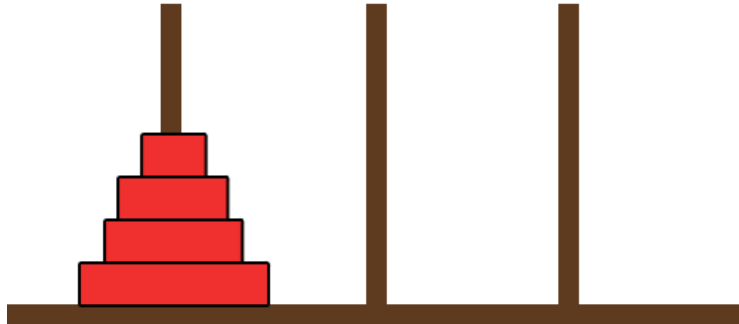
- => Uitwerking tijdens labo met echte “stacks”

Complexity ?

Step: 0



Step 0 of 15



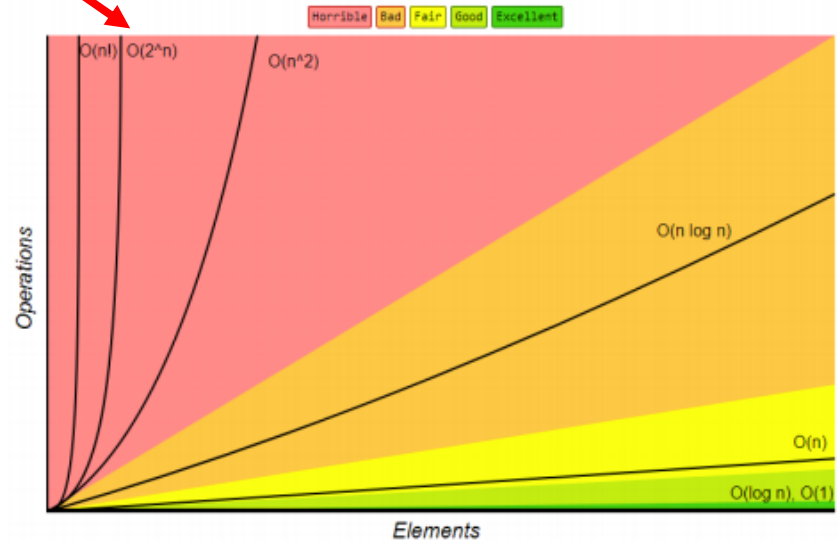
Aantal “moves” nodig ?

| Aantal disks | Minimum aantal “moves” |
|--------------|------------------------|
| 1 | 1 |
| 2 | 3 |
| 3 | 7 |
| 4 | 15 |
| 5 | 31 |
| ... | |
| 10 | 1023 |
| 20 | 1.048.575 |
| 50 | 1.125.899.906.842.623 |
| ... | |

- Aantal moves stijgt spectaculair..
- Om de puzzel op te lossen met N disks zijn er minimum
 $\Rightarrow 2^N - 1$ moves nodig
- Time complexity is **exponentieel**

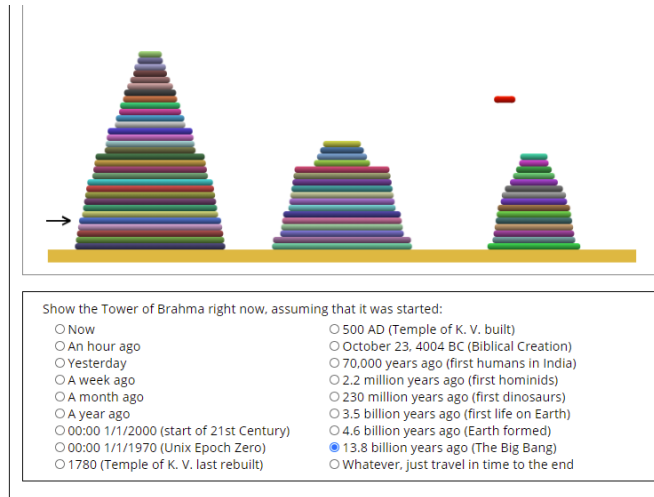
Towers of Hanoi / Complexity

- Time complexity = $O(2^n)$
- Space complexity = $O(n)$



The Tower of Brahma and the End of the Universe

- 64 gouden disks op 3 diamanten staven...
- Lees meer hierover :
<http://ianparberry.com/TowersOfHanoi/index64.html>



Recursieve of iteratieve oplossing ?

- Nadeel van recursie:
 - is trager:
 - Bij recursie zit een extra (tijds)kost voor het aanroepen van een functie
 - verbruikt meer geheugen.
 - Lokale variabelen die telkens opnieuw moeten worden aangemaakt.
- Voordeel van recursie:
 - Recursie is (soms) eleganter en de oplossing is (dikwijls) korter.

Hangt dus van het probleem af of recursie de betere oplossing is.

