

Achtste college algoritmiek

11 april 2008

Restant Decrease and conquer
Dynamisch Programmeren

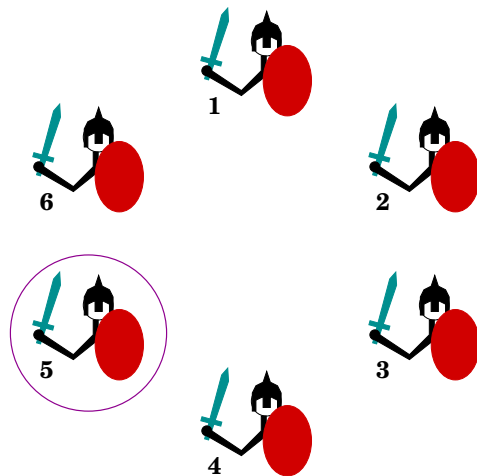
Voorbeelden (variable size / by a constant factor):

- Algoritme van Euclides
Dit is gebaseerd op: $\text{ggd}(m, n) = \text{ggd}(n, m \bmod n)$
- Flavius Josephus (decrease by a constant factor)
- Flipping pancakes (Levitin, exercise 5.6.11)

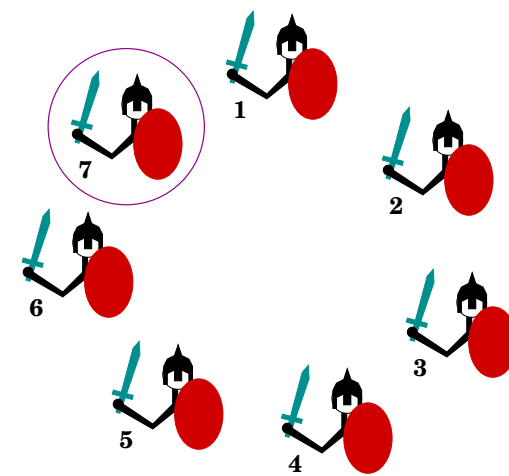


- Binaire zoekbomen

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.

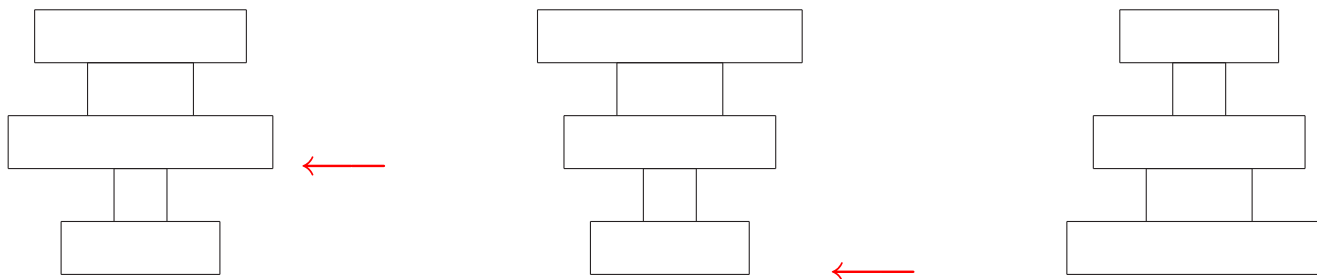
Recurrente betrekking:

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

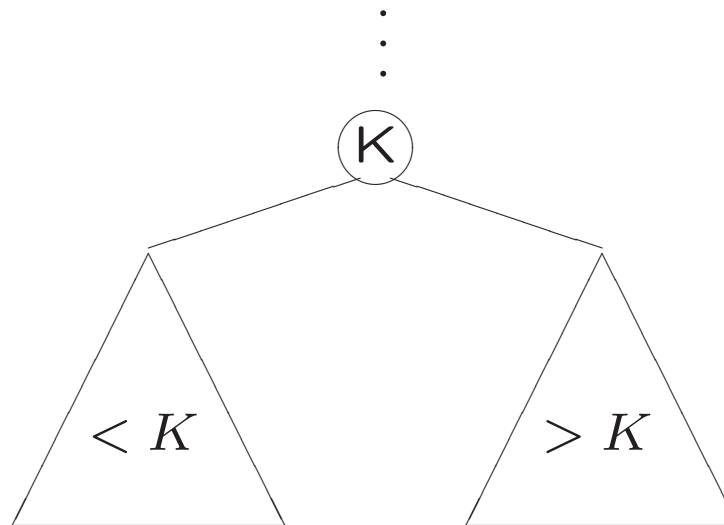
Oplossing:

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

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).



Een **binaire zoekboom** is een binaire boom waarbij voor elke knoop geldt dat de waarde in die knoop groter is dan alle waarden in zijn linkersubboom, en kleiner dan alle waarden in zijn rechtersubboom.



Bij het zoeken naar een waarde in een gewone binaire boom (bijv. WLR) moeten in het slechtste geval alle n knopen bekeken worden. Zoeken in een binaire zoekboom is i.h.a. efficiënter: in het slechtste geval worden $h + 1$ knopen bekeken, met h de hoogte van de boom.

```
knoop* zoeken(knoop* root, int getal) {
    if ( root == null )          // lege boom
        return null;
    else
        if ( root->info == getal )        // gevonden!
            return root;
        else
            if ( getal < root->info )
                return zoeken(root->links, getal);
            else
                return zoeken(root->rechts, getal);
} // zoeken
```

Bekijk en vergelijk vier verschillende oplossingsmethoden voor het berekenen van a^n :

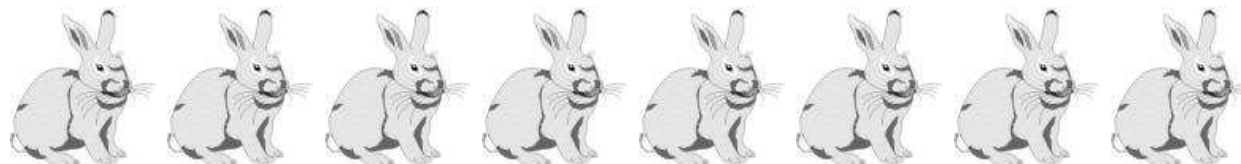
1. **Brute force**: gebaseerd op de definitie, $a^n = \overbrace{a * \dots * a}^{n \times}$
2. **Divide and conquer**: gebaseerd op $a^n = a^{\lfloor \frac{n}{2} \rfloor} * a^{\lceil \frac{n}{2} \rceil}$
3. **Decrease by one**: gebaseerd op $a^n = a^{n-1} * a$
4. **Decrease by a constant factor**: gebaseerd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

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

Definitie Fibonacci-getallen:

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

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

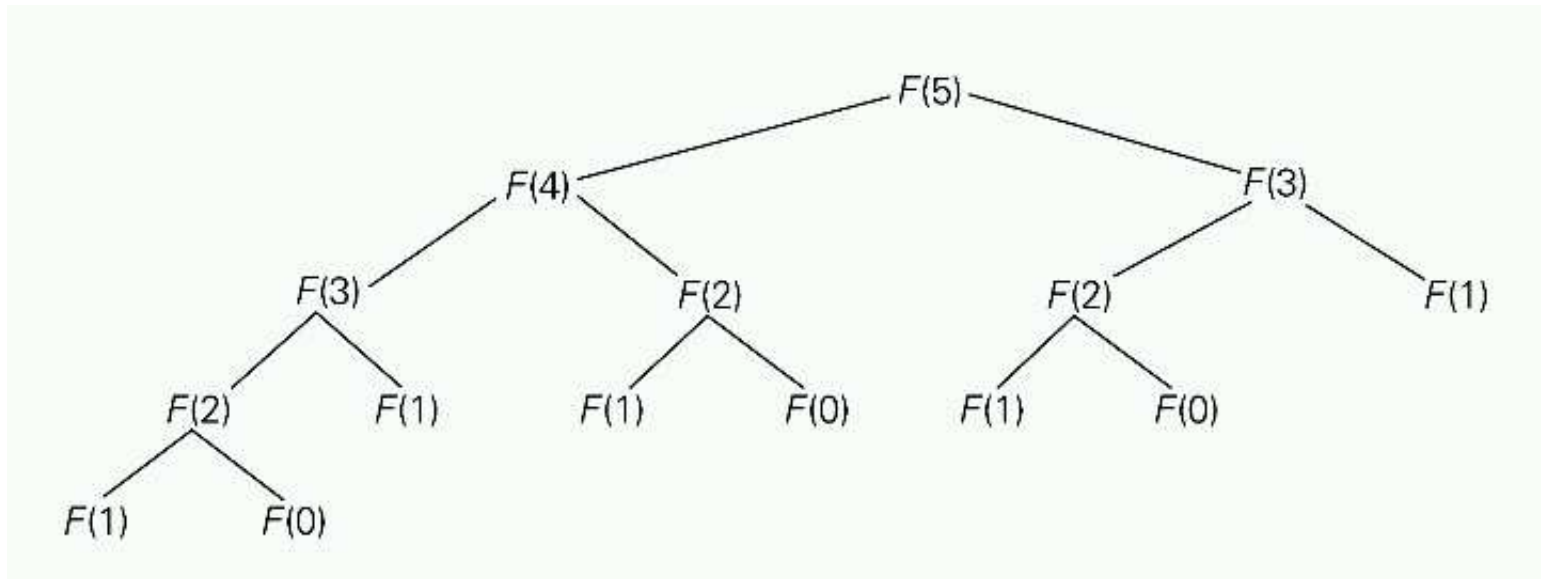


Rekursieve C++-functie:

```
long fib1 (int n) {  
    if ( ( n==0 ) || ( n == 1 ) )  
        return 1;  
    else  
        return ( fib1 (n-1) + fib1 (n-2) );  
} // fib1
```

Watervaleffect





Voor $n = 5$ worden sommige recursieven 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 = 10;
long fib2 (int n) { // recursie met array !
    long static memo[MAX] = {0}; // eenmalig op 0
    if ( n >= MAX ) // helaas
        return fib2 (n-1) + fib2 (n-2);
    else
        if ( memo[n] > 0 ) // al eerder berekend
            return memo[n];
        else {
            if ( ( n==0 ) || ( n == 1 ) )
                memo[n] = 1;
            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.

```
fibonacci[0] = 1;
fibonacci[1] = 1;
for (i=2; i<=n; i++) {
    fibonacci[i] = fibonacci[i-1] + fibonacci[i-2];
}
return fibonacci[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](#)).

- nuttig bij problemen met *overlappende deelproblemen*
- druk een oplossing van het probleem uit in oplossingen van deelproblemen (*recurrente betrekking*)
- 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 $0, 1, 2, \dots, n$, 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 $\text{prijs}[i][j]$, de prijs van het goedkoopste buskaartje van i naar j , gegeven zijn voor alle $i \leq j$. Het probleem is nu om de prijs van de goedkoopste reis te vinden.

Laat $\text{kosten}(n)$ de prijs van de goedkoopste busreis van 0 naar n voorstellen, langs alle tussenliggende steden (in oplopende volgorde). Dan geldt:

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

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).

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 .
```

De recursieve oplossing doet exponentieel veel aanroepen, en er is heel veel overlap tussen de deelproblemen. Oplossing: deeloplossingen opslaan in een geschikt array.

Laat $\text{kosten}[i]$ de prijs van de goedkoopste busreis van 0 naar i voorstellen, langs alle tussenliggende steden (in oplopende volgorde). We zoeken dus $\text{kosten}[n]$.

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 $\text{kosten}[i]$ te berekenen, *alle* kleinere waarden $\text{kosten}[k]$ 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
        kost := kosten[ $k$ ] + prijs[ $k$ ][ $i$ ];
        if kost < temp then
            temp := kost; // 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]; stop[i] := 0; // met 1 bus
    for k := 1 to i - 1 do
        kost := kosten[k] + prijs[k][i];
        if kost < temp then
            temp := kost; // goedkoopste tot dusver
            stop[i] := k; // bijbehorende tussenstop
        fi
    od
    kosten[i] := temp;
od
return kosten[n];
```

Hierin is $stop[i]$ steeds de laatste tussenstop op de goedkoopste reis van 0 naar i .

- **Werkcollege** programmeeropdracht 2:
donderdag 17 april 2008 in zaal 306/308

- **Opgaven:**
www.liacs.nl/home/graaf/ALGO/algo2008.html

- **Volgend college:**
vrijdag 18 april 2008 in **zaal 402**