Achtste college algoritmiek

30 maart 2007

Restant Decrease and conquer
Dynamisch Programmeren

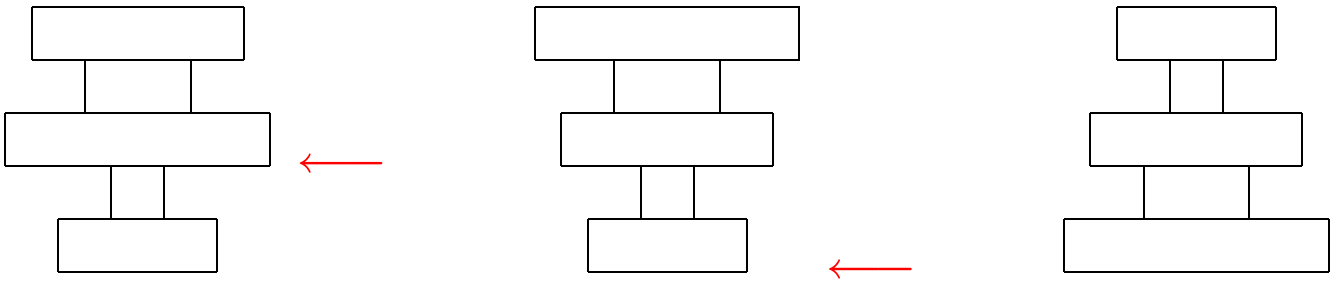
Voorbeelden:

- Algoritme van Euclides.
  Dit is gebaseerd op: ggd(m, n) = ggd(n, m **mod** n)

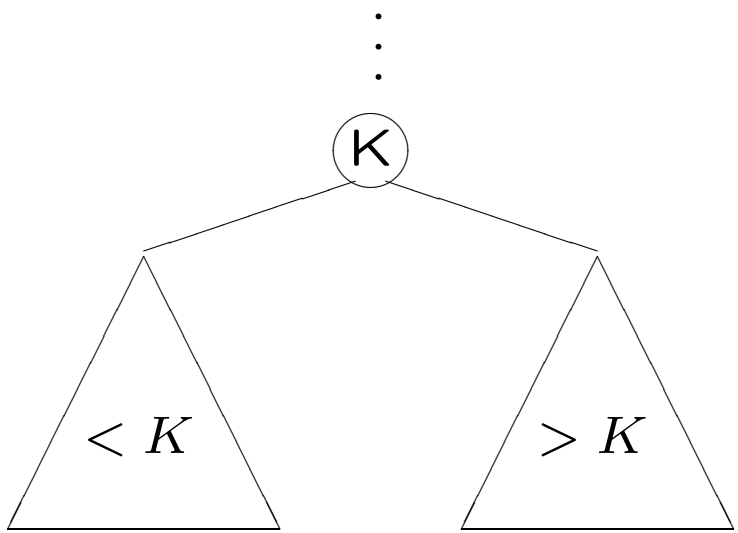
- Flipping pancakes (Levitin, excercise 5.6.11)

- Binaire zoekbomen

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 )
        return root;
     else
        if ( K < 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, $a^n = \overbrace{a * \cdots * a}^{n\times}$

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 $a^n = a^{n-1} * a$

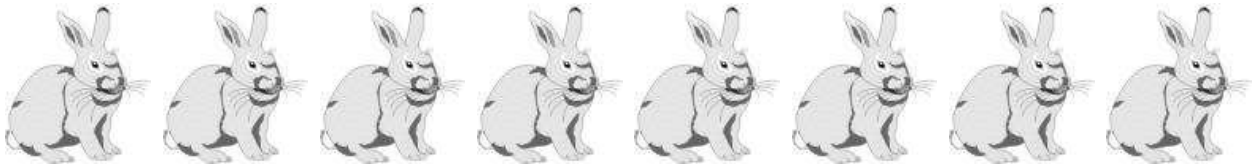
4. Decrease by a constant factor: gebaseerd op

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

Definitie Fibonacci-getallen:

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

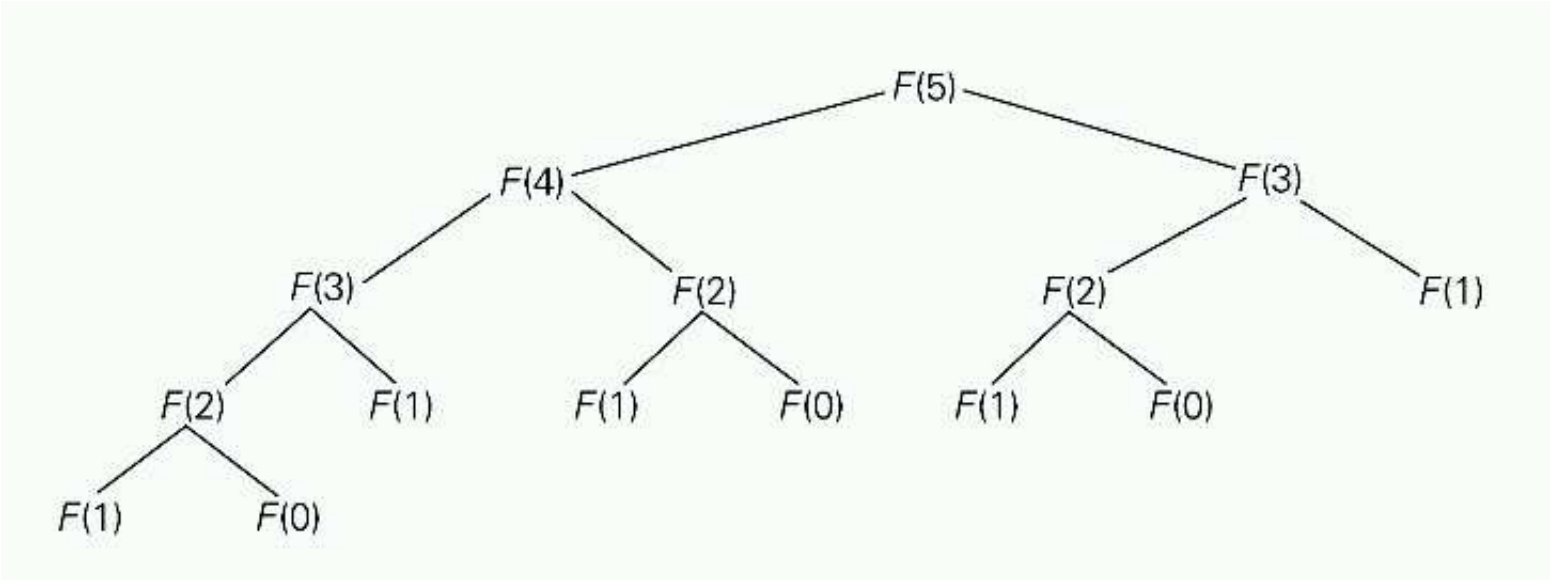
1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584, 4181, 6765, 10946, ...



Recursieve 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
   Vult het array van klein naar groot (for-loop)

Met array, top down

```
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.

```
fibo[0] = 1;
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).

- 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*)

DP: bottom up $\leftrightarrow$ top down

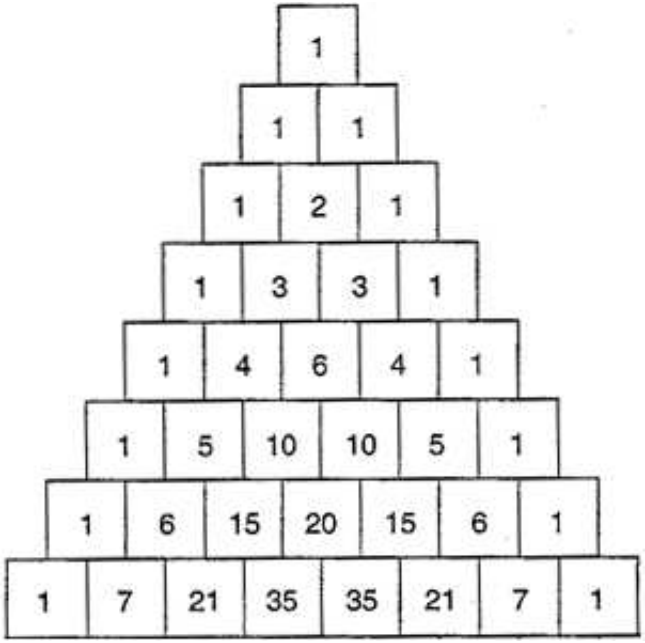
- de bottom up methode is *iteratief*, de top down variant is recursief

- 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



Driehoek van Pascal

$$C(n,k) = \binom{n}{k} = \begin{cases} \binom{n-1}{k-1} + \binom{n-1}{k} & 0 < k < n \\ 1 & k = 0, n \end{cases}$$

Een recursief algoritme ligt voor de hand:

```
int bin1(int n, int k) {
    if ( ( k == 0 ) || ( k == n ) )
        return 1;
    else
        return ( bin1(n-1,k-1) + bin1(n-1,k) );
}
```

Veel van de `bin1(i,j)`'s worden echter herhaald berekend.

Complexiteit: $O(\binom{n}{k})$ (exercise 8.1.6)

Recursief algoritme met een array `C` voor het opslaan van tussenresultaten (dus `C[i][j]` $= \binom{i}{j}$):

```
int bin2(int n, int k) {
// C is globaal (foei) en op nul geinitialiseerd
    if ( C[n][k] != 0 )  // reeds eerder berekend
        return C[n][k];
    else {
        if ( ( k == 0 ) || ( k == n ) ) {
            C[n][k] = 1;
        else
            C[n][k] =  bin2(n-1,k-1) + bin2(n-1,k);
        return C[n][k];
    }
}
```

Complexiteit: $O(n * k)$; extra geheugen: $\Theta(n * k)$

We gebruiken een globaal (op nul geïnitialiseerd) array `C` voor het opslaan van tussenresultaten, dus `C[i][j]` $= \binom{i}{j}$.

	0	1	2	...	$k-1$	k
0	1					
1	1	1				
2	1	2	1			
⋮						
k	1					1
⋮						
$n-1$	1				$C(n-1, k-1)$	$C(n-1, k)$
n	1					$C(n, k)$

Het array wordt rij voor rij gevuld, te beginnen bij rij 0, en per rij van links naar rechts, gebruikmakend van de recurrente betrekking.

```
int bin3(int n, int k) {
    for ( i = 0; i <= n; i++ )
        for ( j = 0; j <= min(i,k); j++ )
            if ( ( j == 0 ) || ( j == i ) )
                C[i][j] = 1;
            else
                C[i][j] = C[i-1][j-1] + C[i-1][j];
    return C[n][k];
}
```

Complexiteit: $\Theta(n * k)$; extra geheugen: $\Theta(n * k)$.
We kunnen hier echter volstaan met een een-dimensionaal array ter lengte k. Er is dus maar $O(k)$ extra geheugen nodig. Zie ook exercise 8.1.4.

- **Werkcollege** programmeeropdracht 2:

  donderdag 5 april 2007 in zaal 306/308

- **Opgaven:**

  zie http://www.liacs.nl/home/graaf/ALGO/algo2007.html

- **Volgend college:**

  vrijdag 13 (!) april 2007