

Achtste college algoritmiek

31 maart 2021

Verdeel en Heers

Dynamisch Programmeren

Gegeven n punten $p_1 = (x_1, y_1), \dots, p_n = (x_n, y_n)$.

Gevraagd het/een tweetal punten dat het dichtst bij elkaar ligt. Afstandsmaat: $d(p_i, p_j) = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$.

Brute force algoritme: alle paren (p_i, p_j) (met $i < j$) aflopen en hun onderlinge afstanden $d(p_i, p_j)$ vergelijken.

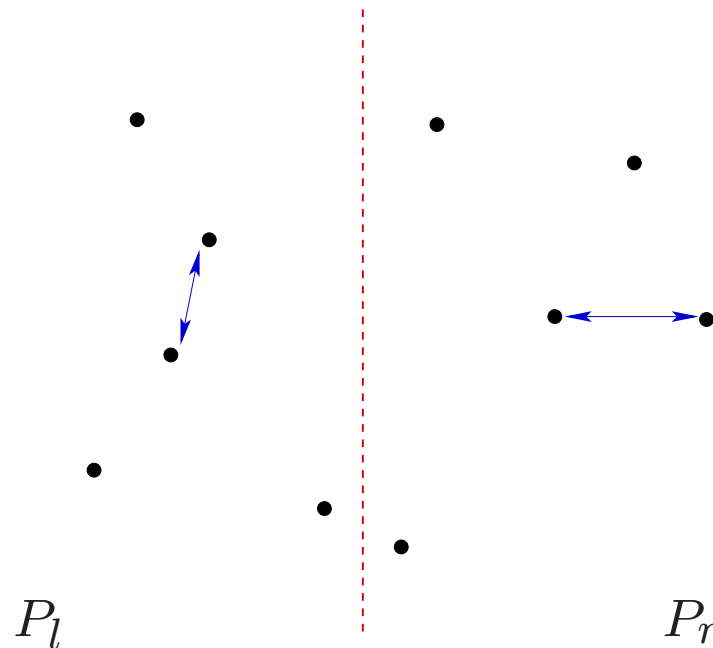
```
dmin := ∞;
for i := 1 to n - 1 do
  for j := i + 1 to n do
    d := (x_i - x_j)2 + (y_i - y_j)2;
    if d < dmin
      dmin := d; k := i; l := j;
    fi // (p_k, p_l) voorlopig closest pair
  od
od
```

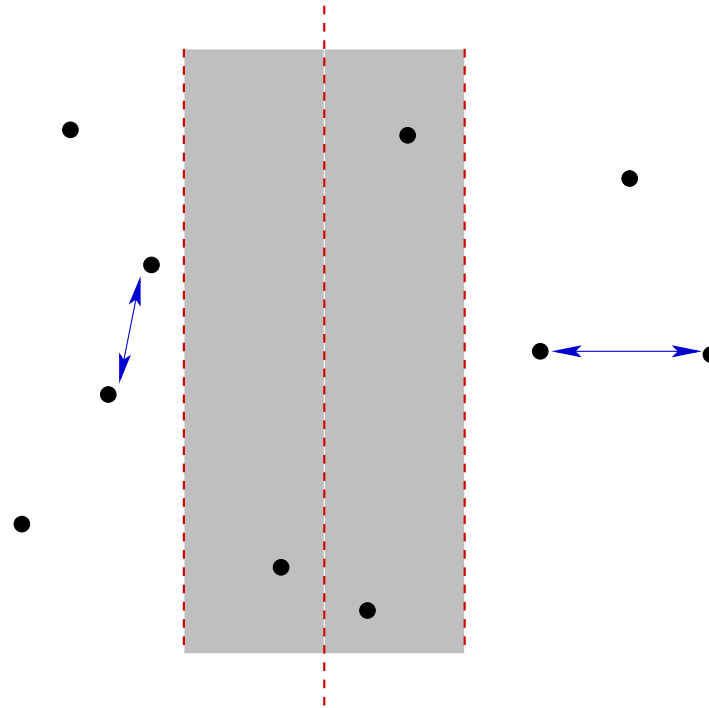
Complexiteit: $\frac{1}{2}n(n - 1) = \Theta(n^2)$

Divide and Conquer:

Verdeel de verzameling van n punten in twee verzamelingen P_l en P_r van elk $\frac{n}{2}$ punten door een geschikte lijn te trekken.

Los beide deelproblemen (recursief) op en laat d de kleinst voorkomende afstand zijn tussen punten van P_l resp. P_r .

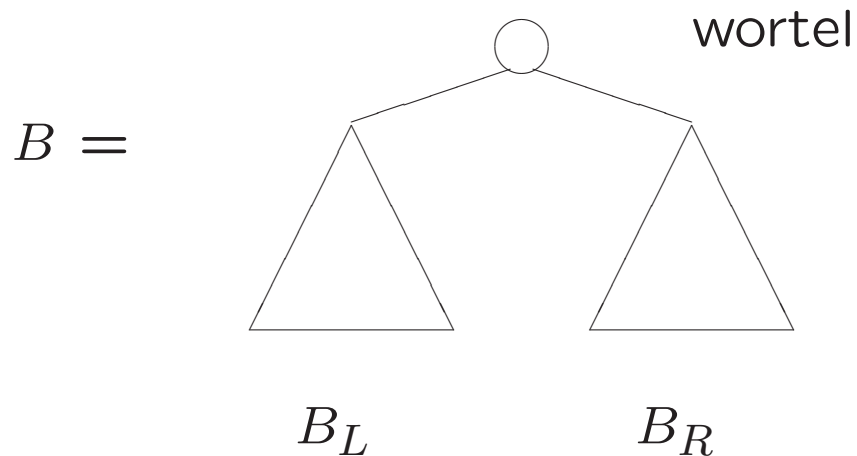




Controleer of er binnen de strip ter breedte $2d$ rondom de scheidlijn tussen P_l en P_r puntenparen (p, p') zijn met $p \in P_l$ en $p' \in P_r$ en onderlinge afstand kleiner dan d .

Hiervoor blijken slechts $O(n)$ puntenparen bekeken te moeten worden. Dit levert een $O(n \lg n)$ algoritme op.

Een binaire boom B wordt **recursief** gedefinieerd als ofwel leeg, ofwel bestaande uit een knoop (de wortel) en twee disjuncte subbomen B_L en B_R die beide ook weer een binaire boom zijn: de **linkersubboom** en de **rechtersubboom**.



Bij (veel) problemen met binaire bomen ligt oplossen via divide & conquer dus voor de hand.

De **hoogte** van een binaire boom is het hoogste nivo dat voorkomt, waarbij de wortel per definitie op nivo 0 zit.

Voor de hoogte van een binaire boom B geldt dus:

$$\text{hoogte}(B) = 1 + \max \{ \text{hoogte}(B_L), \text{hoogte}(B_R) \}$$

Verdeel en heers algoritme in C++:

```
int hoogte( knoop * root ) {  
    if ( root == nullptr )        // lege boom  
        return -1;  
    else  
        return ( 1 + max( hoogte( root->links ), hoogte( root->rechts ) ) );  
} // hoogte
```

DECREASE by one & CONQUER

```

Insertionsort( $A[0 \dots m - 1]$ )::
  if  $m > 1$ 
    Insertionsort( $A[0 \dots m - 2]$ );
    Voeg  $A[m - 1]$  op de juiste plek in;
  fi .

```

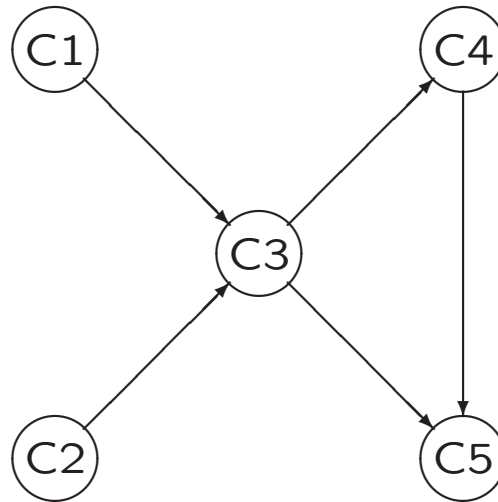
Invoegen van $A[m - 1]$ in het reeds gesorteerde voorstuk $A[0] \dots A[m - 2]$ door van rechts naar links $A[m - 1]$ te vergelijken met $A[i]$. Deze recursieve versie komt overeen met de iteratieve versie zoals bij [Programmeermethoden](#) behandeld (zie ook Levitin):

$$A[0] \leq A[1] \leq \dots \leq A[i] \leq A[i + 1] \leq \dots \leq A[m - 3] \leq A[m - 2] || A[m - 1] \dots$$

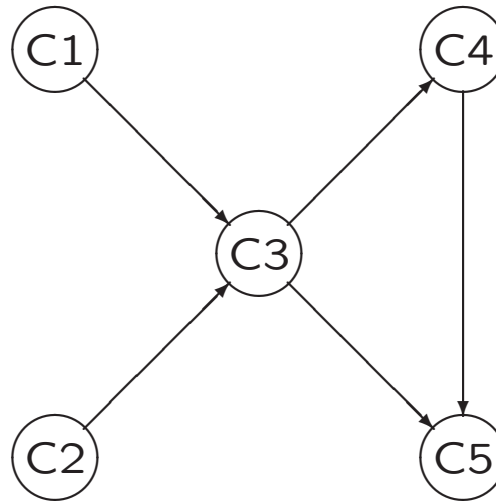
kleiner of gelijk $A[m - 1]$ \uparrow groter dan $A[m - 1]$

hier invoegen

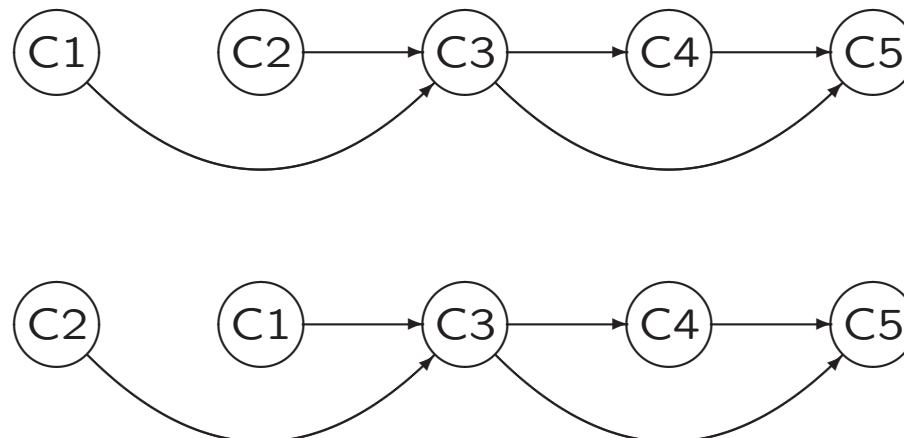
Gerichte grafen, b.v. met benodigde voorkennis voor cursussen:



Gerichte grafen, b.v. met benodigde voorkennis voor cursussen:



Mogelijke volgordes



Gerichte grafen

- eis aan graaf
- algoritme
- implementatie
- efficiëntie
- ander algoritme
- efficiëntie
- effect algoritmes als graaf cyclisch is

Gerichte grafen

- eis aan graaf: acyclisch
- algoritme: source-removal
- implementatie: aantal inkomende takken per knoop bijhouden
- efficiëntie: $\Theta(n + m)$
- ander algoritme: DFS
- efficiëntie: $\Theta(n + m)$
- effect algoritmes als graaf cyclisch is: verschillend

Zie paragraaf 4.2 van boek + antwoorden werkcollege 7/8.

Binair zoeken is een voorbeeld van decrease and conquer, **decrease by a constant factor**. Hier de iteratieve versie.

// invoer: oplopend gesorteerd array $A[0..n - 1]$, te zoeken waarde K
 // uitvoer: positie van K in A (-1 als K niet in A zit)

$l := 0; r := n - 1;$

while $l \leq r$ **do**

$m := \lfloor \frac{l+r}{2} \rfloor;$

if $K = A[m]$ **then** // gevonden

return $m;$

else if $K < A[m]$ **then** // links verder zoeken

$r = m - 1;$

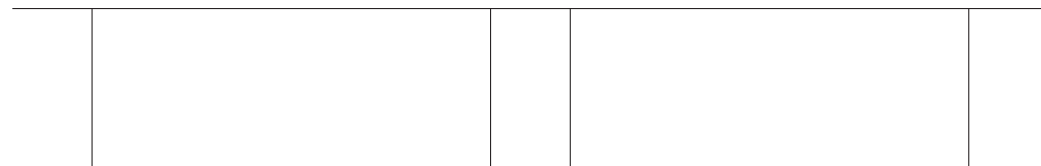
else // rechts verder zoeken

$l = m + 1;$ **fi**

fi

od

return $-1;$

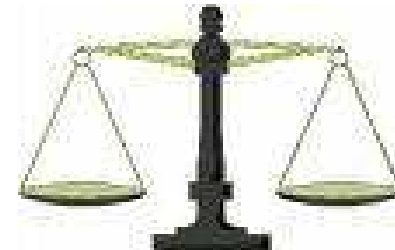


l m r

altijd *links óf rechts* verdergaan

Fake coin probleem

Gegeven n identiek uitziende munten. Eén ervan is vals. Bekend is dat de valse munt lichter is dan de andere. Tevens is een balans beschikbaar. Bepaal door weging de valse munt.



Decrease by a constant factor: verdeel de munten in twee stapels van $\lfloor \frac{n}{2} \rfloor$ en —indien n oneven— een losse munt. Als de twee stapels even zwaar zijn (best case) is de losse munt de valse. Zo niet, dan bevindt de valse zich in de lichtste van de twee stapels.

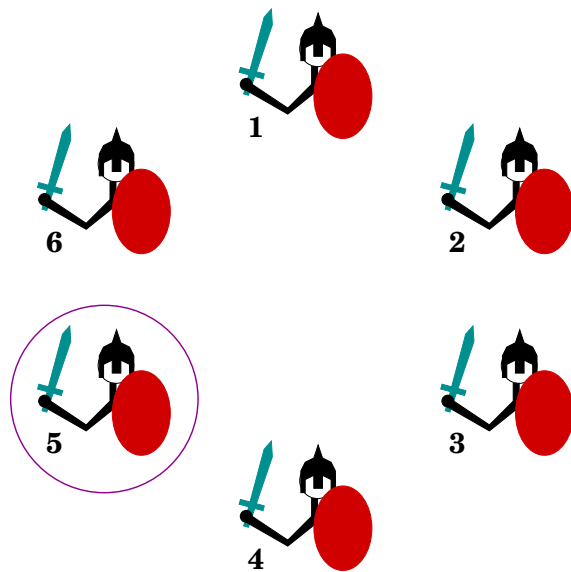
Recurrente betrekking voor het aantal wegingen dat nodig is in de **worst case** om de valse munt te ontdekken:

$$W(n) = W(\lfloor \frac{n}{2} \rfloor) + 1 \text{ als } n > 1, W(1) = 0$$

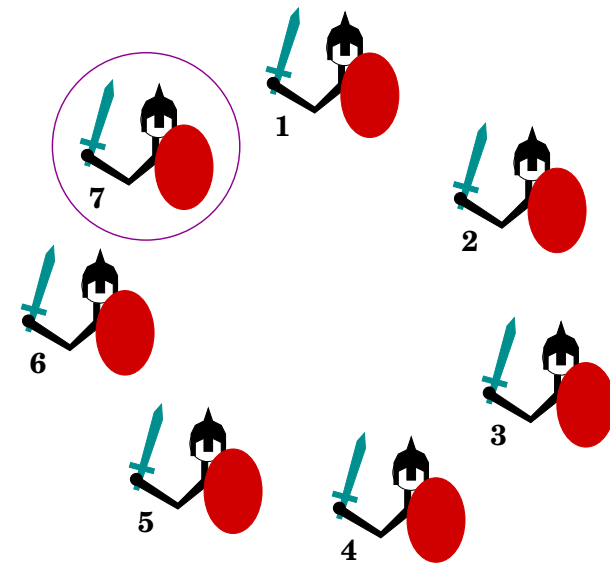
Oplossing: $W(n) = \lfloor \lg n \rfloor$.

Zie Levitin exercise 4.5.10 voor een efficiënter decrease by a constant factor algoritme.

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

Variable-size decrease: de reductie in grootte is variabel,
dus kan in elke stap anders zijn

Voorbeelden:

- Algoritme van Euclides
Dit is gebaseerd op: $\text{ggd}(m, n) = \text{ggd}(n, m \bmod n)$
- Flipping pancakes (Levitin, exercise 4.5.12)



- Binaire zoekbomen

Probleem:

Gegeven: twee niet-negatieve gehele getallen m en n (niet beide nul).

Vraag: wat is de grootste gemeenschappelijke deler, genoteerd als $\text{ggd}(m,n)$, van m en n ?

Voorbeelden:

$$\text{ggd}(60,24) = 12;$$

$$\text{ggd}(25,0) = 25;$$

$$\text{ggd}(200, 441) = 1;$$

$$\text{ggd}(588,495) = 3.$$

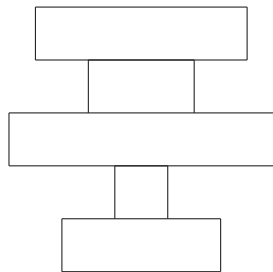
Het algoritme van **Euclides** is gebaseerd op het herhaald gebruiken van

$$\text{ggd}(m, n) = \text{ggd}(n, m \bmod n),$$

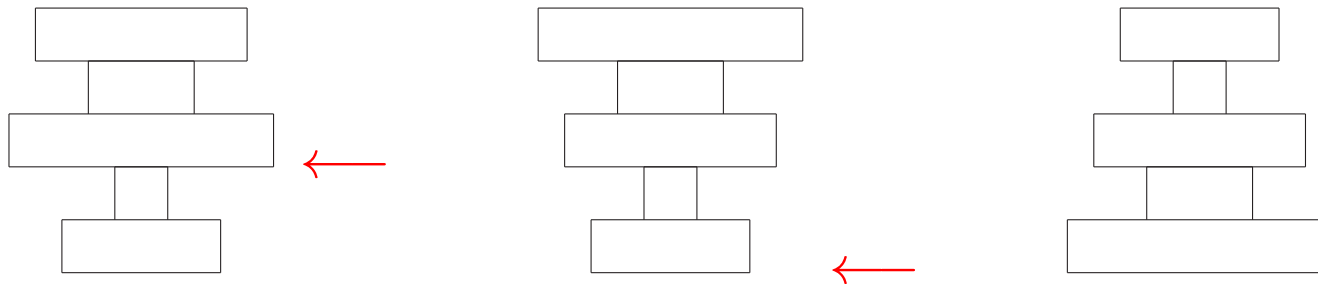
totdat de tweede parameter nul wordt.

Voorbeeld: $\text{ggd}(60,24) = \text{ggd}(24, 12) = \text{ggd}(12, 0) = 12$

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).



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).



BOUNDS FOR SORTING BY PREFIX REVERSAL

William H. GATES

Microsoft, Albuquerque, New Mexico

Christos H. PAPADIMITRIOU*†

Department of Electrical Engineering, University of California, Berkeley, CA 94720, U.S.A.

Received 18 January 1978

Revised 28 August 1978

For a permutation σ of the integers from 1 to n , let $f(\sigma)$ be the smallest number of prefix reversals that will transform σ to the identity permutation, and let $f(n)$ be the largest such $f(\sigma)$ for all σ in (the symmetric group) S_n . We show that $f(n) \leq (5n+5)/3$, and that $f(n) \geq 17n/16$ for n a multiple of 16. If, furthermore, each integer is required to participate in an even number of reversed prefixes, the corresponding function $g(n)$ is shown to obey $3n/2 - 1 \leq g(n) \leq 2n + 3$.

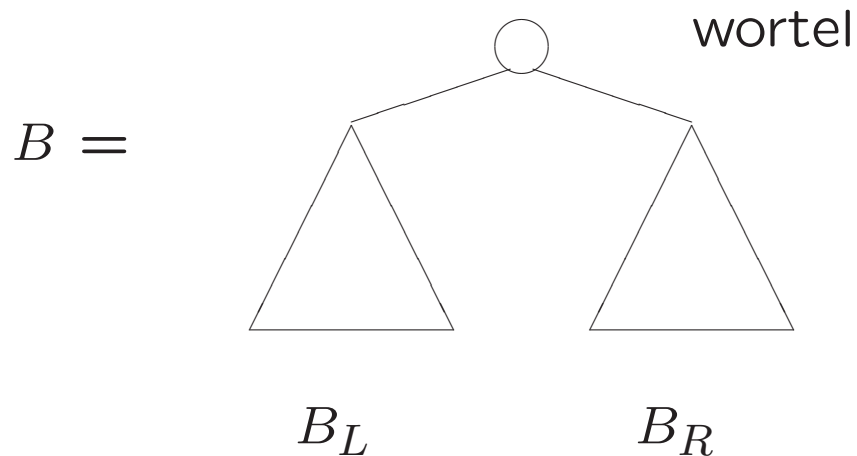
1. Introduction

We introduce our problem by the following quotation from [1]

The chef in our place is sloppy, and when he prepares a stack of pancakes they come out all different sizes. Therefore, when I deliver them to a customer, on the way to the table I rearrange them (so that the smallest winds up on top, and so on, down to the largest at the bottom) by grabbing several from the top and flipping them over, repeating this (varying the number I flip) as many times as necessary. If there are n pancakes, what is the maximum number of flips (as a function $f(n)$ of n) that I will ever have to use to rearrange them?

In this paper we derive upper and lower bounds for $f(n)$. Certain bounds were already known. For example, consider any stack of pancakes. An *adjacency* in

Een binaire boom B wordt **recursief** gedefinieerd als ofwel leeg, ofwel bestaande uit een knoop (de wortel) en twee disjuncte subbomen B_L en B_R die beide ook weer een binaire boom zijn: de **linkersubboom** en de **rechtersubboom**.



Bij (veel) problemen met binaire bomen ligt oplossen via divide & conquer dus voor de hand.

De **hoogte** van een binaire boom is het hoogste nivo dat voorkomt, waarbij de wortel per definitie op nivo 0 zit.

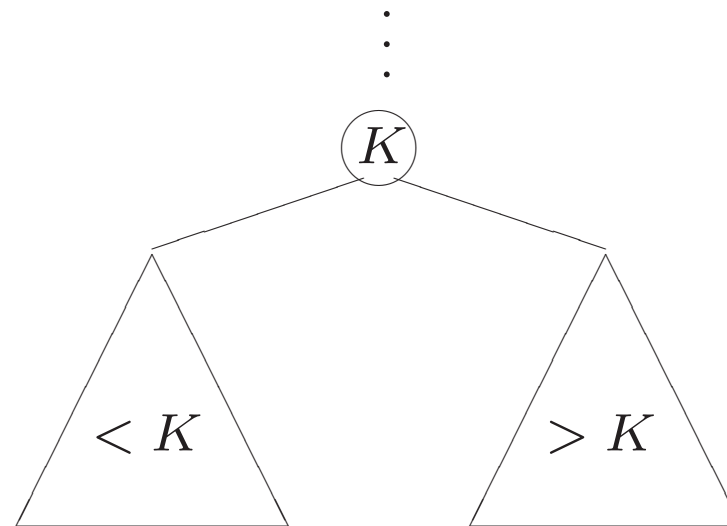
Voor de hoogte van een binaire boom B geldt dus:

$$\text{hoogte}(B) = 1 + \max \{ \text{hoogte}(B_L), \text{hoogte}(B_R) \}$$

Verdeel en heers algoritme in C++:

```
int hoogte( knoop * root ) {
    if ( root == nullptr )        // lege boom
        return -1;
    else
        return ( 1 + max( hoogte( root->links ), hoogte( root->rechts ) ) );
} // hoogte
```

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 == nullptr )        // lege boom
        return nullptr;
    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**: . . .

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

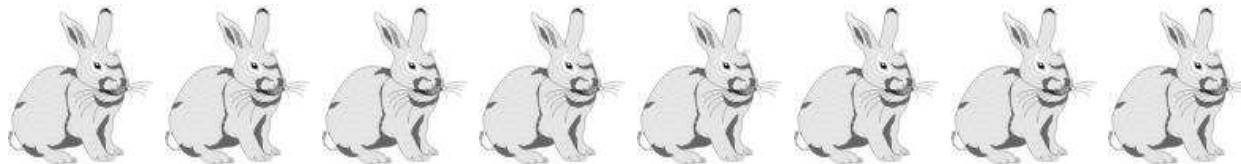
4. **Decrease by a constant factor**: gebaseerd op

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

Definitie Fibonacci-getallen:

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

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597,
2584, 4181, 6765, 10946, ...

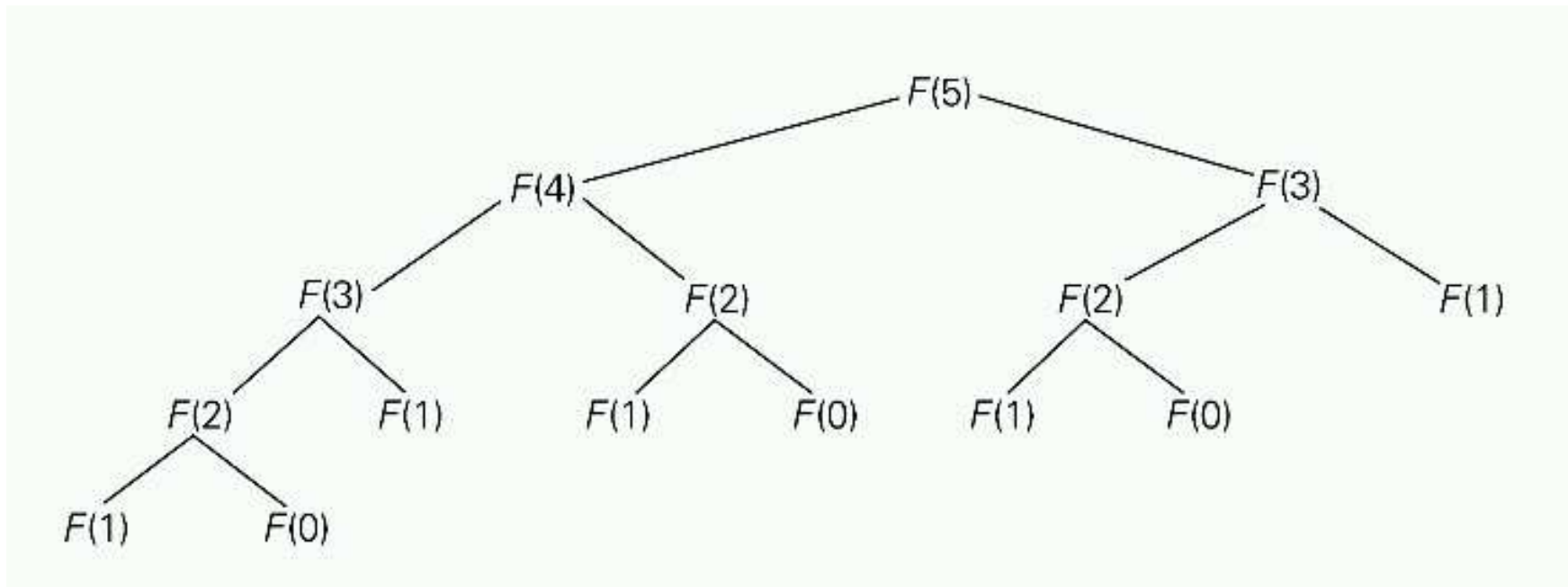


Recursieve C++-functie:

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

Watervaleffect





Voor $n = 5$ worden sommige recursieve 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 = 45;
long memo[MAX]; // helemaal op -1 initialiseren

long fib2 (int n) { // recursie met array !
    if ( memo[n] > -1 ) // al eerder berekend
        return memo[n];
    else {
        if ( ( n==0 ) || ( n == 1 ) )
            memo[n] = n; // gaat goed!
        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] = 0;
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**).

Met drie variabelen.

```
fib0 = 0;
fib1 = 1;
i = 1;
while (i<n) {
    tmp = fib0;
    fib0 = fib1;
    fib1 = tmp + fib0;
    i++;
}
return fib1;
```

tmp	fib0	fib1	<i>i</i>
	0	1	1
0	1	1	2
1	1	2	3
1	2	3	4
2	3	5	5
3	5	8	6
5	8	13	7
8	13	21	8
...

Werkt voor...

- nuttig bij problemen met *overlappende deelproblemen*
- druk een oplossing van het probleem uit in oplossingen van deelproblemen (*recursieve formulering*)
- 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

- **Lezen/leren bij dit college:**
Paragraaf 5.5 (geen convex hull), 5.3, 4.2, 4.4 (geen Russian peasant), 4.5 (incl. + Binary Search Tree), 8.incl., slides
- **Werkcollege** verdeel en heers
donderdag 1 april 2021, 14.15–15.00 (weblecture)
- **Interactief vragenuur:**
donderdag 1 april 2021, 15.15–16.00
- **Opgaven:**
zie <http://www.liacs.leidenuniv.nl/~vlietrvan1/algoritmiek/>
- **Practicumbijeenkomst programmeeropdracht 1/2:**
dinsdag 6 april 2021, 10.30-12.15, Kaltura liveroom
- **Volgend college:**
woensdag 7 april 2021, 14.15–16.00, weblecture