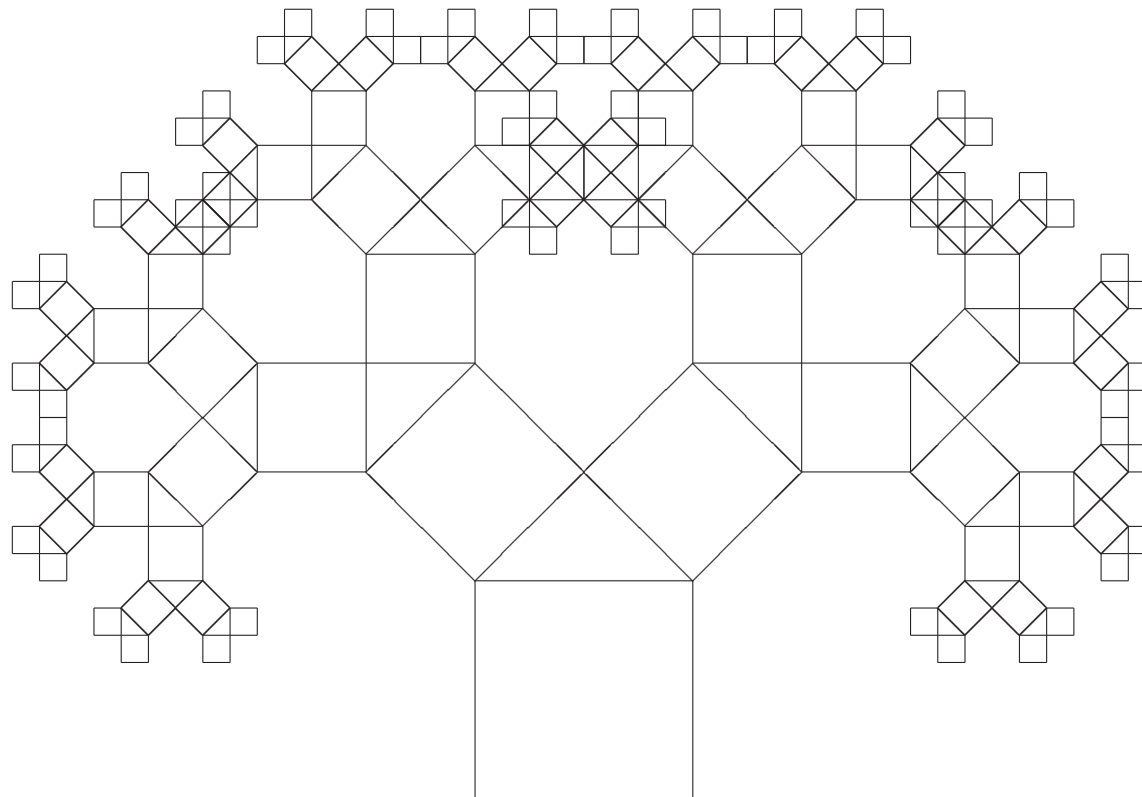

Programmeermethoden

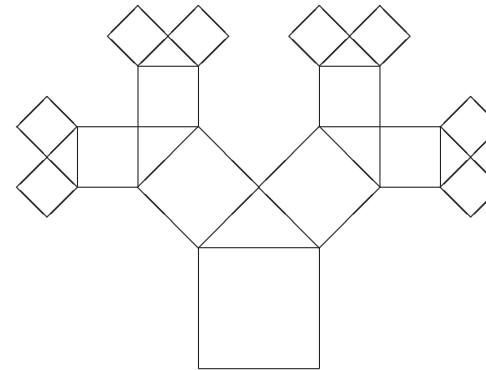
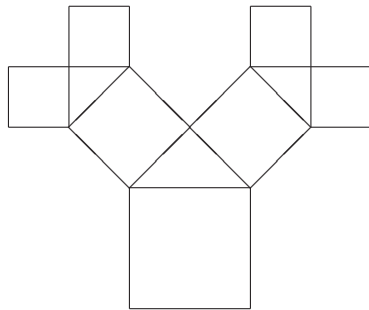
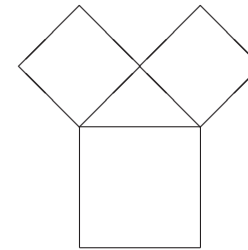
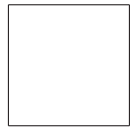
Recurisie

week 11: 21–25 november 2011

<http://www.liacs.nl/home/kosters/pm/>



Boom van Pythagoras



Basisidee:

proces is **recursief** als het naar zichzelf verwijst

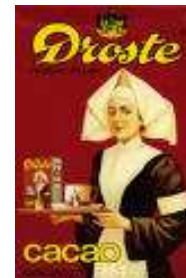
functie is **recursief** als hij zichzelf (in)direct aanroept

Voorbeeld:

leeftijd (nu) = leeftijd (een jaar geleden) + 1

Woordenboek:

Recursie: zie Recursie



Een **recursief/ve** proces/procedure/functie bestaat in het algemeen uit twee delen:

1. één of meer **kleinste** (eenvoudigste) gevallen die **direct oplosbaar** zijn: de **basisgevallen**
2. een algemene methode die een bepaald geval **reduceert** tot één of meer **kleinere** (eenvoudiger) gevallen, waarbij men uiteindelijk op een basisgeval uitkomt

Algemene gedaante van een recursieve functie:

if **basisgeval** **then**

 los op zonder recursie; // **makkelijk**

else

 één of meer **recursieve** eenvoudigere **aanroepen**;

fi

Let op de symbolische notatie in **pseudo-code**.

Probleem:

$P(X)$ = Betaal een bedrag X

Oplossing:

$P(0)$ = doe niks

$P(X)$ = Geef de grootste munt $Y \leq X$

Betaal het bedrag $X-Y$: $P(X-Y)$

Vraag:

Wat kan hier nog fout gaan? betaal 30 met 25/10

Recursieve definitie van n -faculteit ($n!$):

$$\text{fac}(n) = \begin{cases} 1 & \text{als } n = 0 \\ n \times \text{fac}(n - 1) & \text{als } n > 0 \end{cases}$$

Recursieve C++-functie ($n \geq 0$):

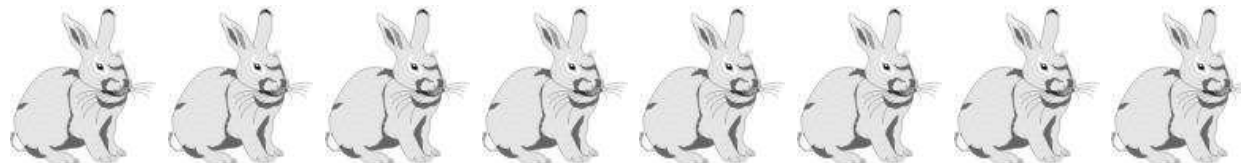
```
long faculteit (int n) {  
    if ( n == 0 ) // basisgeval  
        return 1;  
    else  
        return n * faculteit (n-1); // recursie  
} //faculteit
```

Definitie **Fibonacci-getallen**:

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

Alternatief: $\text{fib}(1) = \text{fib}(2) = 1$.

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
```

Er is hier sprake van een **watervaleffect**:
de aanroep `fib1 (5)` veroorzaakt
14 andere (dubbele) aanroepen.



```
const int MAX = 100;
long memo[MAX]; // globaal, initialiseer met 0-en!
// recursie met array
long fib2 (int n) {
    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
```

Iteratief: opsommen tot je bij $\text{fib}(n)$ bent:

```
long fib3 (int n) {
    long eerste = 1, tweede = 1, hulp;
    int teller;
    for ( teller = 2; teller <= n; teller++ ) {
        // nu geldt: eerste == fib (teller-2) en
        // tweede == fib (teller-1) ("invariant")
        hulp = tweede;
        tweede = eerste + tweede;
        eerste = hulp;
    }//for
    return tweede;
}//fib3
```

Deze versie is erg geschikt voor “grote getallen”.

Gesloten formule (nauwelijks te berekenen!):

$$\text{fib}(n) = \frac{1}{\sqrt{5}} \left(\left(\frac{1 + \sqrt{5}}{2} \right)^{n+1} - \left(\frac{1 - \sqrt{5}}{2} \right)^{n+1} \right)$$

↑

klein in absolute waarde

Met matrices:

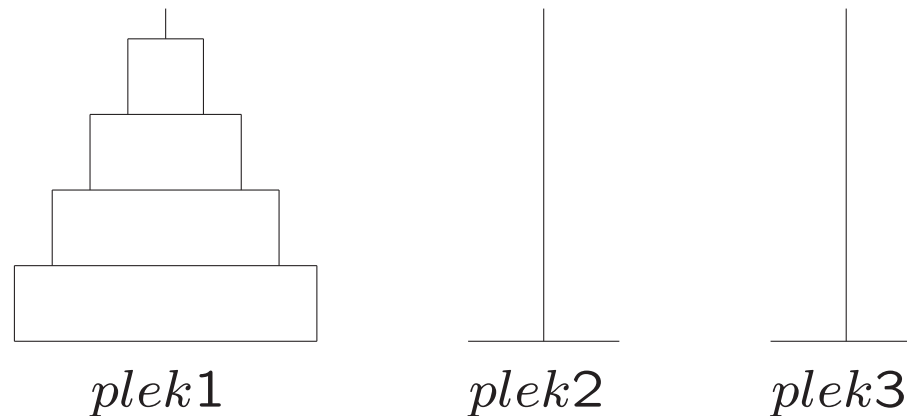
$$\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^n = \begin{pmatrix} \text{fib}(n) & \text{fib}(n-1) \\ \text{fib}(n-1) & \text{fib}(n-2) \end{pmatrix}$$

Gegeven: n ($n \geq 1$) schijven met gat in het midden, alle verschillend in grootte, en 3 palen

Beginsituatie: alle schijven liggen boven op elkaar om één paal, en de andere 2 palen zijn leeg

Restrictie: een grotere schijf ligt nooit op een kleinere

Voorbeeld: beginsituatie voor $n = 4$

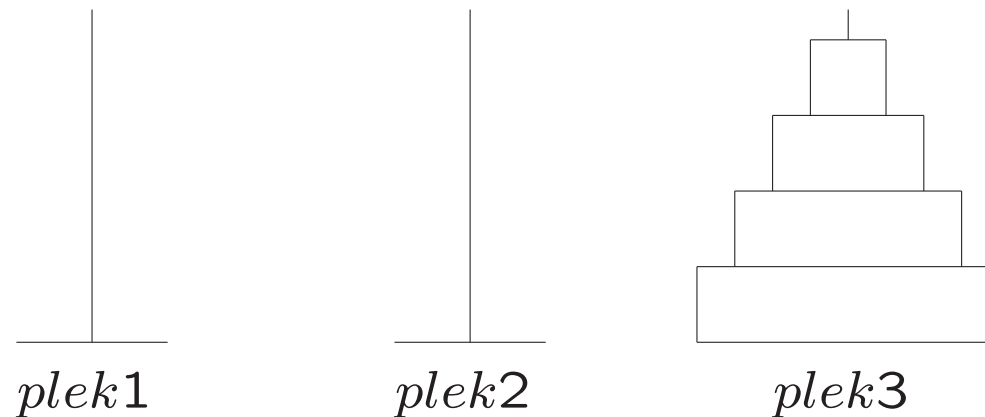


Doel: breng de hele toren naar een van de lege palen

Acties: per keer mag je één schijf verplaatsen (de bovenste van een stapel), en deze bovenop een andere stapel leggen

Restrictie: er mogen alleen kleinere schijven op grotere worden gelegd

Voorbeeld: eindsituatie voor $n = 4$



Oplossing:

n schijven zo efficiënt mogelijk van start naar doel verplaatsen =

eerst de bovenste $n - 1$ schijven zo efficiënt mogelijk van start naar hulp verplaatsen,

dan de grote schijf van start naar doel,

en tenslotte de $n - 1$ schijven zo efficiënt mogelijk van hulp naar doel verplaatsen

Dat is recursie! Wat is het basisgeval?

Oplossing:

n schijven zo efficiënt mogelijk van start naar doel verplaatsen **via hulp** =

eerst de bovenste $n - 1$ schijven zo efficiënt mogelijk van start naar hulp verplaatsen **via doel**,

dan de grote schijf van start naar doel,

en tenslotte de $n - 1$ schijven zo efficiënt mogelijk van hulp naar doel verplaatsen **via start**

Dat is recursie! Wat is het basisgeval?

Algoritme:

```
// Torens van Hanoi: recursief
// 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
```

Het aantal zetten is in totaal $2^n - 1$.

Aanroep: `zet (aantal,1,3,2);`

Hierbij is `aantal` het gewenste aantal schijven.

Probleem: Zoek een waarde in een *gesorteerd* array A met n elementen

Oplossing: **binair zoeken**

- * Kijk of het middelste element het gezochte element is.
- * Zo niet, bepaal dan op grond van vergelijken met dat middelste element of verder (recursie!) gezocht moet worden in de linker helft óf in de rechter helft van het array en herhaal dit.
- * Stop zodra het element gevonden is, dan wel het te onderzoeken array leeg is.
- * Als het aantal elementen even is: kies één van de twee middelste.

```
// Geeft index met A[index] = getal, als getal voorkomt;
// zo niet: resultaat wordt -1.

int binairzoeken (int A[ ], int n, int getal) {
    int links = 0, rechts = n-1; // zoek tussen links en rechts
    int midden;

    while ( links <= rechts ) {
        midden = ( links + rechts ) / 2;
        if ( getal == A[midden] )
            return midden; // of gevonden = true etc.
        else if ( getal > A[midden] )
            links = midden + 1;
        else
            rechts = midden - 1;
    }//while

    return -1;
}//binairzoeken
```

Binair zoeken: recursief

```
int binairzoeken (int A[ ], int n, int links, int rechts, int getal) {
    int midden;
    if ( links > rechts )           // basisgeval: leeg interval
        return -1;                  // dus stop; niet aanwezig
    else {                          // nu echt zoeken
        midden = ( links + rechts ) / 2;
        if ( getal == A[midden] )   // gevonden!
            return midden;
        else                        // verder zoeken: recursieve aanroepen
            if ( getal > A[midden] ) // rechts hetzelfde doen
                return binairzoeken (A,n,midden+1,rechts,getal);
            else                    // links hetzelfde doen
                return binairzoeken (A,n,links,midden-1,getal);
    } //else echt zoeken
} //binairzoeken
```

Aanroep: `iets = binairzoeken (A,n,0,n-1,getal);`

```
sorteer (rij) =  
  if ( rij heeft meer dan 1 element ) then  
    verdeel rij in linkerrij en rechterrij;  
    sorteer (linkerrij);  
    sorteer (rechterrij);  
    combineer (linkerrij, rechterrij);  
fi
```

⇓ (zie elders)

Mergesort: $O(n \lg n)$

Quicksort: $O(n \lg n)$

Insertion sort: $O(n^2)$

n = aantal elementen van de rij, $\lg n = {}^2 \log n = \log_2 n$

```
void print1 (int    a) { // by value
    if ( a > 0 ) {
        a--;
        print1 (a);
        cout << a << ", ";
    } //if
} //print1
```

Nu doen we:

```
getal = 3; print1 (getal); cout << getal << endl;
```

Dat levert: 0, 1, 2, 3

```
void print2 (int & a) { // by reference
    if ( a > 0 ) {
        a--;
        print2 (a);
        cout << a << ", ";
    } //if
} //print2
```

Nu doen we:

```
getal = 3; print2 (getal); cout << getal << endl;
```

Dat levert: 0, 0, 0, 0

```
void print3 (int & a) { // by reference
    if ( a > 0 ) {
        a--;
        print3 (a);
        cout << a << ", ";
        a++; // en a weer terugzetten
    } //if
} //print3
```

Nu doen we:

```
getal = 3; print3 (getal); cout << getal << endl;
```

Dat levert: 0, 1, 2, 3

Recursie wordt ook vaak gebruikt bij het programmeren van spellen als Schaken, Go en ... Boter, kaas en eieren.

We willen het **aantal vervolgpactijen** $S.Aantal()$ weten vanuit een gegeven stand (= positie) S :

```
S.Aantal ( ) ::  
  Teller ← 0;  
  if  $S$  is eindstand then  
    return 1;  
  fi  
  for alle mogelijke zetten  $i$  do  
    S.DoeZet ( $i$ );  
    Teller ← Teller + S.Aantal ( );  
    S.OntDoeZet ( $i$ );  
  od  
  return Teller;
```

Bij deze oplossing is ervoor gekozen de Stand S niet “kapot” te maken, vandaar de aanroep $OntDoeZet(i)$. Gebruik makend van de eigenschap dat recursieve aanroepen S niet verstoren, doet de buitenste aanroep dat nu ook niet.

Je kunt ook, voor iedere i opnieuw, de zet doen in een kopie van S , zodat je S nooit vernielt.

Overigens: er zijn 255.168 verschillende *partijen* Boter, kaas en eieren. En je hebt dan meteen het hele spel door-gerekend (zie later).

Opgave 3 van het tentamen van 7 april 2007:

Gegeven is een n bij n (met n een $\text{const} \geq 2$) array `kost: int kost[n][n];`, gevuld met gehele getallen ≥ 0 . Een getal `kost[i][j] > 0` stelt de kosten voor om rechtstreeks van i naar j te reizen ($0 \leq i, j < n$ en $i \neq j$), en `kost[i][j] = 0` betekent dat er geen directe verbinding is van i naar j . Er geldt `kost[i][i] = 0` voor $0 \leq i < n$; verder is vanuit iedere plaats minstens één directe reis mogelijk. Als je rechtstreeks van i naar j kunt reizen, kun je ook direct van j naar i ; maar misschien verschillen de kosten wel!

- a.** Schrijf een C++-functie `double gem (kost, i)` die bepaalt wat de gemiddelde kosten zijn voor een directe reis vanuit i ($0 \leq i < n$). Denk eraan te middelen over de “niet-nullen”.
- b.** Schrijf een functie `verschil (kost)` die het grootste absolute verschil uitrekent dat je kunt hebben tussen directe reizen van i naar j en j naar i (tussen heen- en terugreizen dus), voor willekeurige i en j . Onderzoek hiertoe alle paren (i, j) .
- c.** Schrijf een C++-functie `hoeveel (kost, i)` die bepaalt hoeveel plaatsen je vanuit een zekere i ($0 \leq i < n$) kunt bereiken, inclusief i zelf, waarbij je zo vaak je wilt mag “overstappen”. Hint: gebruik een array `bool D[n]`, initieel gevuld met `false`'s (alleen `D[i] = true`), waar in `D[j]` moet komen of je j kunt bereiken.

- maak de vierde programmeeropgave — de deadline is op 9 december 2011
- lees Savitch Hoofdstuk 13
- lees dictaat Hoofdstuk 3.10, 4.2.2, 4.2.7
- maak opgaven 51/56 uit het opgavendictaat
- <http://www.liacs.nl/home/kosters/pm/>