# Recursieve functies: de krachtigste constructie van een programmeertaal

Daniel von Asmuth

**Samenvatting**

We leggen uit waarom recursie een belangrijk onderdeel is van veel programmeertalen aan hand van voorbeelden in C en Lisp.

**Waarschuwing 1.** Dit is versie 20240306. De tekst is oaan verandering onderhevig.

## 1 Inleiding

**De meeste programmeertalen kennen recursieve functies; voor functioneel programmeertalen zijn ze zelfs de belangrijkste constructie. Beginners vinden deze elementaire programmeertechniek soms moeilijk, maar met een beetje oefening werkt recursie gemakkelijker dan iteratie en levert overzichtelijke en leesbare code op. Er wordt wel eens gezegd dat recursie minder efficiënt is, maar met de juiste processorarchitectuur en compilerondersteuning valt dat erg mee als de programmeur oplet.**

**In de wiskunde zijn recursieve definities gebruikelijk. Een van de meest bekende is de 'faculteit' functie:**

$$n! = \begin{cases} n=0 \rightarrow 1 \\ n>0 \rightarrow n \times (n-1)! \end{cases}$$

**Deze definitie kan gemakkelijk worden vertaald naar een C functie:**

```
int fac( n)
{ return n == 0 ? 1 : n * fac( n-1); }
```

**We kunnen hetzelfde resultaat bereiken met een iteratieve functie, dat wil zeggen met gebruik van een lus (loop):**

```
int fac( n)
{ int i, p = 1;
  for( i = 1; i <= n; i++) p *= i;
  return p;
}
```

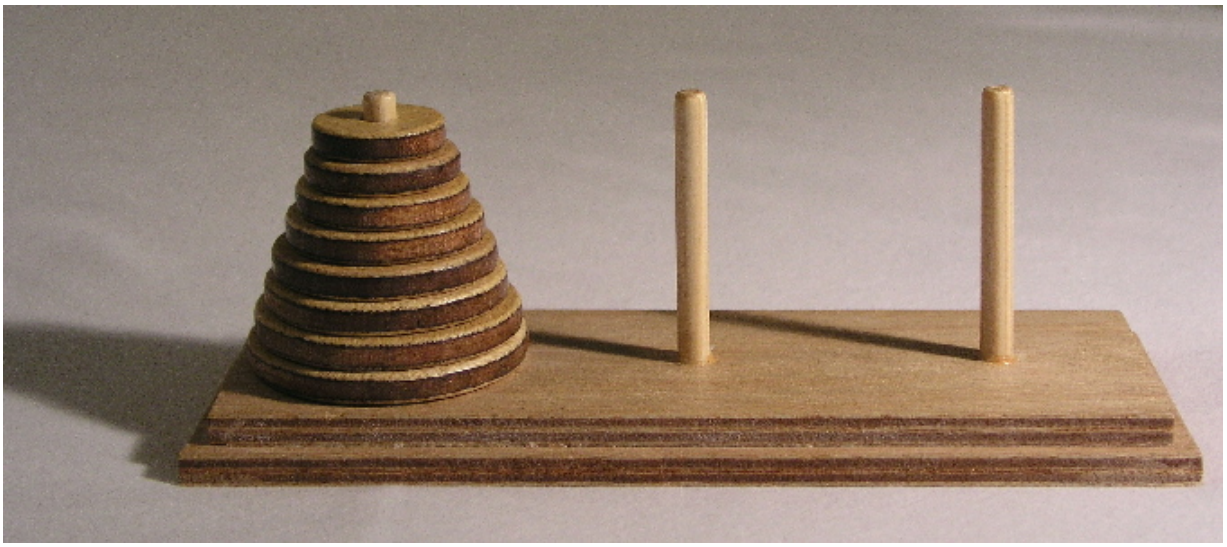
**We noemen een functie *recursief* wanneer ze zichzelf aanroept; een groep functies A, B, en C noemen we *wederzijds recursief* wanneer functie A B aanroept, functie B C aanroept en functie C weer A. Het belangrijkste is om ervoor te zorgen dat een functie zichzelf niet oneindig vaak aanroept. In het voorbeeld hierboven zien we dat de faculteit functie zichzelf aanroept met als argument *n - 1*. De recursie termineert dus wanneer n de waarde 0 bereikt, maar als n $<$ 0 zal dat nooit gebeuren.**

**C en andere talen gebruiken de hardware stack niet enkel om retouradressen van functies op te slaan, maar ook voor de parameters en automatische variabelen (dat zijn lokale variabelen die niet statisch gedeclareerd zijn). Iedere keer dat een functie (methode) of subroutine (procedure) wordt aangeroepen, wordt daarvoor een extra stuk geheugen gereserveerd, dat weer vrijkomt na het verlaten van de functie. Zodoende zijn de parameters en automatische variabelen van meerdere aanroepen van dezelfde functie ook verschillend.**

**Sommige mensen zijn huiverig voor recursie omdat ze teveel geheugen zou vereisen. Aan de ene kant is dat met moderne architecturen niet zo'n probleem meer, aan de andere kant is dat pas een probleem als er grote hoeveelheden data op de stack staan. Daarmee blijft een paar nanoseconden over om de parameters op de stack te kopiëren.**

## 2 De torens van Hanoi

**Er gaat een legende dat in een tempel in de stad Benares in India een aantal hindoe-priesters bezig zijn om een toren van 64 gouden schijven te verplaatsen tussen een drietal naalden van diamant.**



**Op bovenstaande foto (geleend van Wikipedia) zijn de schijven van hout i.p.v. goud en het zijn er ook slechts 8. Het doel is om de hele toren van de linkse naald A te verplaatsen naar de rechtse naald C, waarbij je telkens maar één schijf tegelijk mag verplaatsen. Je mag de middelste naald B gebruiken om schijven te parkeren. Verder geldt de eis dat er nooit een grotere schijf op een kleinere mag rusten.**

## 3 De oplossing

**Procedure**
verplaats toren van bovenste $n$ schijven van naald $A$ via $B$ naar $C$.
**Implementatie**
*als het aantal* n *gelijk is aan 1*
- verplaats dan de bovenste schijf van naald $A$ naar naald $C$

*als het aantal* n *groter is dan 1*
- verplaats bovenste $n - 1$ schijven van naald $A$ via $C$ naar $B$
- verplaats $n^e$ schijf van naald $A$ naar $C$
- verplaats bovenste $n - 1$ schijven van naald B naar $C$ via $A$

## 4 Correctheid: „doet-ie het of doet-ie het niet?"

**Je moet altijd bewijzen dat een programma het probleem daadwerkelijk oplost; anders zou je de software moeten testen voor alle denkbare invoer data.**

**De eerste stap is om je te realiseren dat je maar één schijf tegelijk mag verplaatsen, maar hierboven staat al een procedure om een stapel van $n - 1$ schijven in meerdere stappen te verplaatsen van $A$ via $C$ naar $B$. Het algoritme bevat dus een dubbele recursie. Bij een recursieve functie hoort een bewijs met *inductie*.**

**Stelling 2.** *De procedure hierboven realiseert het gestelde doel.*

**Bewijs.**

**Geval n = 0**

Het algoritme doet niets en dat is correct.

**Geval n = 1**

De schijf wordt van naald A genomen en op naald C gelegd; ook dit geval is correct

**Geval n = 2**

- De kleinere schijf wordt verplaatst van naald $A$ naar $B$
- De grotere schijf wordt verplaatst van naald $A$ naar $C$
- De kleinere schijf wordt verplaatst van naald $B$ naar $C$.

**Inductie: oplossing voor *n + 1* schijven**

- verplaats toren van $n$ schijven van naald $A$ naar $B$
- verplaats (toren van) *1* schijf van naald $A$ naar $C$
- verplaats toren van $n$ schijven van naald $B$ naar $C$ via $A$

Als de procedure door de recursieve functieaanroepen het doel bereikt voor torens van 0 t/m n schijven, dan volgt hieruit dat ze ook voor n +1 schijven correct is. De inductieregel zegt dat de bewering daarom geldig is voor alle (eindige waarden van) n. □

**Nadere bestudering levert op dat de oplossing voor 64 schijven $2^{64}$ - 1 = 18.446.744.073.709.551.615 rekenstappen vergt; binnen het bereik van een snelle computer. De benodigde stackruimte is maximaal 64 niveaus diep.**

# 5 Een iteratieve variant

Elders heb ik bewezen dat de Hanoi functie met $n$ schijven $2^n - 1$ aanroepen vergt. De iteratieve variant van bijlage G gaat daarom uit van een for-lus die die getallen afloopt.

Er blijkt dan een eenvoudig verband te bestaan tussen het volgnummer door de lus en de recursiediepte, zodat de functie get_depth() de laatste kan berekenen. De functie get_bit() extraheert het gewenste bit uit een 64-bits integer.

De Hanoi() functie doet weinig anders anders dan de nummers van de naalden permuteren. Permutaties komen hieronder uitgebreid aan de orde, maar hier zijn de 3! permutaties van A, B,C hard gecodeerd. De eerste recursieve aanroep voert permutatie nr. 2 uit en de tweede permuatie nr. 6. Beide permutaties zijn hun eigen inverse. Alles wat we nodig hebben zijn combinaties van die twee permutaties. De functie fill() maakt twee tabellen van het resultaat van de applicatie van die permutaties en slaag ze op in de arrays apply2 en apply6 om tijd te besparen.

| | |
|---|---|
| 0 0 0 1 | 0 0 0 |
| 0 0 1 0 | 0 0 - |
| 0 0 1 1 | 0 0 1 |
| 0 1 0 0 | 0 - - |
| 0 1 0 1 | 0 1 0 |
| 0 1 1 0 | 0 1 - |
| 0 1 1 1 | 0 1 1 |
| 1 0 0 0 | - - - |
| 1 0 0 1 | 1 0 0 |
| 1 0 1 0 | 1 0 - |
| 1 0 1 1 | 1 0 1 |
| 1 1 0 0 | 1 - - |
| 1 1 0 1 | 1 1 0 |
| 1 1 1 0 | 1 1 0 |
| 1 1 1 1 | 1 1 1 |

**Tabel 1.** De combinaties van permutaties 2 en 6 voor 4 schijven

Dan blijkt er een regelmaat te bestaan in de permutatie - zie tabel 1, waar in de eerste kolom de volgnummers binair zijn weergegeven en in de tweede kolom de permutaties, waarin de 0 staat voor de tweede permutatie en 1 voor de zesde. Je ziet dat de getallen in de tweede kolom precies de eerste drie binaire cijfers van het volgnummer ervoor zijn en het aantal permutaties gelijk is aan $n$ - $d$, waarin $n$ het aantal schijven is en $d$ de recursiediepte. Hiermee kan de iteratieve saigon() functie berekenen hoe vaak ze permutatie 2 of 6 moet uitvoeren of terugrollen, waarna een eenvoudige printopdracht volstaat.

De iteratieve code is iets moeilijker te doorgronden en loopt iets langzamer.

# 6 Probleem: genereer alle permutaties

**Sommige datastructuren zoals binaire bomen zijn van nature geschikt om recursief te doorzoeken, terwijl arrays en linked lists heel geschikt zijn voor for-loops. Voor een iets praktischer probleem beschrijven we een algoritme om een lijst van alle permutaties te genereren van de lijst $\langle 1 \ldots n \rangle$. Daarvoor zijn verschillende iteratieve oplossingen gevonden, maar een recursief algoritme is gemakkelijker te doorgronden.**

# 7 Permuteren in Scheme

**De oplossing is ontleend aan „Functioneel programmeren - een inleiding" van Bird & Wadler en iets uitgebreid in Chicken Scheme uitgeprogrammeerd; de code staat in de bijlage.**

**De functie list1() maakt van een argument $x$ een lijst $\langle x \rangle$. Een lijst is Lisp is een paar van twee elementen, waarvan de laatste gelijk is aan NIL. Oftewel**
(cons (quote 1) (cons (quote 2) (cons (quote 3) (cons (quote 4) NIL)))) $= \langle 1, 2, 3, 4 \rangle$

**De functie map1() is een basale *iterator* die een functie toepast op alle elementen van een lijst.**

```
(define inc (lambda (x) (+ x 1)))
```

**Dan zou (map1 inc quote( 1 2 3 4)) de lijst (2 3 4 5) opleveren.**

**De functie concat() voegt de lijst $x$ samen met $ls$. We noteren dit als**
$\langle 1, 2, 3 \rangle \clubsuit \langle 4, 5 \rangle = \langle 1, 2, 3, 4, 5 \rangle$

**De functie cat1() is een *gecurryde* variant van concat(), gebruikt omdat map1() een functie met één parameter verwacht (een eponiem van Haskell Curry, terwijl het concept al door Gottlob Frege werd gebruikt).**

**De functie interleave() krijgt een atoom $x$ en een lijst $ls$ en zal $x$ op alle mogelijke posities in $ls$ invoegen. Bijvoorbeeld**
(interleave $'x'(1234)$) = ((x 1 2 3 4) (1 x 2 3 4) (1 2 x 3 4) (1 2 3 x 4) (1 2 3 4 x))

**De functie inter1() is een iterator die een lijst maakt door interleave() aan te roepen op elk element van $ls$, bijvoorbeeld**

```
(inter1 'x '((1 2) (3 4))) =
((x 1 2) (1 x 2) (1 2 x) (x 3 4) (3 x 4) (3 4 x))
```

**De functie permut() levert een lijst van alle permutaties van de invoer, zoals**

```
(permut '(1 2 3)) = ((1 2 3) (2 1 3) (2 3 1) (1 3 2) (3 1 2) (3 2 1))
```

# 8 Analyse

**Hier gebruiken we een algebraïsche notatie naast de Lisp broncode, zoals de $\clubsuit$ operator om lijsten samen te voegen voor de concat() functie. $\#l$ duidt de lengte van een lijst aan en $l_i$ is het i-e element van een lijst: functies die gemakkelijk in Lisp kunnen worden gecodeeerd.**

$x \epsilon l \equiv x = \text{car}(l) \vee x \epsilon \text{cdr}(l)$
$l_2 \epsilon \text{Perm}(l_1) \equiv \#l_1 = \#l_2 \wedge \forall x \epsilon l_2 : x \epsilon l_1$

**Als $x \epsilon l$, dan definiëren we een *sublijst* van $l$ als een lijst $l'$ die alle elementen van l bevat in de zelfde volgorde met uitzondering van x.**

**Hulpstelling 3.**

$\forall i : 0 \leqslant i \leqslant \#l : \text{interleave}(x, l)_{\text{ii}} = x$

**Bewijs.**
**Basis:**
interleave( $x, \langle\rangle$ ) = $\langle\langle x \rangle\rangle$
$\langle\langle x \rangle_0\rangle_0$ = x
interleave($x$, ls)= $\langle\langle x \rangle$ ✣ls $\rangle$✣....
$\langle\langle\langle\langle x \rangle$ ✣ls $\rangle$✣$y\rangle_0\rangle_0$ = x
**Inductie:**
interleave( $x$, ls) = $\langle\langle x \rangle$ ✣ls $\rangle$ ✣map1( cat1(car( ls)), interleave( x, cdr( ls)))
=$\langle\langle x \rangle$✣ ls$\rangle$✣ $\langle$car(ls)✣ interleave($x$, car(cdr(ls))))✣($\langle$car(ls)$\rangle$✣interleave(x, cdr(cdr(ls)))
$\langle\langle$interleave($x$, ls)$\rangle_{n+1}\rangle_{n+1}$ = $\langle\langle$car(ls)✣interleave($x$, car(cdr(ls))))$_{n+1}\rangle_n$
= $\langle\langle$interleave($x$, car(cdr(ls))))$_n\rangle_n$
volgens de inductiehypothese is dat gelijk aan $x$ □

**Hulpstelling 4.**

Alle elementen van interleave(x,l) zijn lijsten die een sublijst gelijk aan $l$ hebben door element $x$ te schrappen.

**Bewijs.**
**Basis:**
interleave( $x, \langle\rangle$ ) = $\langle\langle x \rangle\rangle$
De $x$ schrappen resulteert in $\langle\langle\rangle\rangle$
**Inductie:**
interleave( $x$, ls) = $\langle\langle x \rangle$ ✣ls $\rangle$ ✣map1( cat1(car( ls)), interleave( x, cdr( ls)))
Het eerste element$\langle\langle x \rangle$ ✣ls $\rangle$ voldoet aan de bewering.
Het tweede element is $\langle$car(ls)✣$\langle x \rangle$ ✣ cdr(ls)$\rangle$, wat eveneens voldoet.
Het restant begint met het eerste element van l en bevat volgens de inductiehypothese alle elementen van cdr(x). □

**Hulpstelling 5.**

**Als $l = \langle l_0, l_1, \ldots l_{n-1} \rangle$** een permutatie is van 1..$n$, dan is interleave(n+1, l) een lijst permutaties van 1..$n$+1.

**Bewijs.**
**Basis:**
interleave( 1, $\langle\rangle$ ) = $\langle\langle 1 \rangle\rangle$
**Inductie:**
interleave( $n+1$, ls) = $\langle\langle n+1 \rangle$ ✣ls $\rangle$ ✣map1( cat1(car( ls)), interleave( $n$+1, cdr( ls))) =
$\langle\langle n+1 \rangle$✣ ls$\rangle$✣ $\langle$car(ls)✣ interleave($n+1$, car(cdr(ls))))✣($\langle$car(ls)$\rangle$✣interleave($n$+1,cdr(cdr(ls)))
Het eerste element $\langle\langle n+1 \rangle$ ✣ls $\rangle$ voldoet aan de bewering.
Het tweede element bevat het eerste element van ls en volgens de inductiehypothese (en de hulpstellingen 1 en 2) de rest van ls en ($n$+1). □

**Hulpstelling 6.**

**Als $l = \langle l_0, l_1, \ldots l_{n-1} \rangle$** een permutatie is van 1..$n$, dan is interleave(n+1, l) de lijst van alle permutaties van 1..$n$+1 die sublijst l bevatten.

**Bewijs.**
Dit volgt uit hulpstellingen 1 en 2 en 3, plus de conclusie dat de interleave() functie de waarde $n$+1 op alle mogelijke posities invoegt in $l$. □

**Hulpstelling 7.**

**Als $l = \langle l_0, l_1, ...l_{n\text{-}1}\rangle$ een lijst van alle permutaties van 1..$n$ is, dan zal inter1( $n$+1, l) de lijst van alle permutaties van 1..$n$+1 opleveren.**

**Bewijs.**
**Basis:**
$\text{inter1}(0, \langle\rangle) = \langle\langle\rangle\rangle$
$\text{inter1}(1, \langle\langle\rangle\rangle) = \langle\langle 1\rangle\rangle$
**Inductie:**
$\text{inter1}(n+1, l) = \text{interleave}(n+1, \text{car}(l)) \text{✠} \text{inter1}(n+1, \text{cdr}(l))$
Met hulpstelling 4 volgt dat $\text{interleave}(n+1, l_i)$ de lijst van permutaties van $1...n$ bevat die sublijst $\text{car}(l)$ gemeen hebben.
De inductie leidt tot de conclusie dat $\text{inter1}(n+1, l)$ een lijst zal retourneren die de resultaten van alle interleave() aanroepen combineert.
We wisten al dat de elementen van $\text{interleave}(n+1, l_i)$ alle verschillend zijn. Omdat de elementen van $l_i$ ook alle verschillend zijn, volgt dat alle elementen van $\text{inter1}(n+1, l)$ alle verschillend zijn.
Omdat $l$ alle permutaties van 1..$n$ bevat, volgt met hulpstelling 4 het gevraagde. □

**Stelling 8.**

**De functie permut( ′($n$ ... 2, 1)) retourneert een lijst van alle permutaties van de argument lijst.**

**Bewijs.**
**Basis:**
$\text{permut}(\langle\rangle) = \langle\langle\rangle\rangle$
$\text{permut}(\langle 1\rangle) = \langle\langle 1\rangle\rangle$
**Inductie:**
$\text{permut}(\langle n+1, n, ...., 2, 1\rangle) = \text{inter1}(n+1, \text{permut}\langle n, ...., 2, 1\rangle))$
Volgens de inductiehypothese levert $\text{permut}\langle n, ...., 2, 1\rangle)$ een lijst van alle permutaties van 1..n en volgens hulpstelling 5 zal inter1() de lijst van alle permutaties van 1..$n$+1 produceren.
□

# 9 Recursief permuteren in C

**C geldt als een efficiënte taal. De volgende oplossing maakt dat waar door gebruik van een array. De permutaties worden telkens uitgeprint in plaats van opgeslagen. Een ander verschil is dat de swap() functie telkens twee elementen verwisselt in plaats van lijsten te construeren.**

**De broncode is te vinden in de bijlage. Afgezien van enkele hulpfuncties, doet loop() al het werk met een dubbele recursieve aanroep. Die functie wordt ongeveer 2 * n! maal aangeroepen om n! oplossingen te produceren en is daarmee efficiënt. Voor de analyse gebruiken we de Floyd-Hoare bewijsregels.**

**Hulpstelling 9.** ***(zonder bewijs)***

**Aanroepen van de functie fill() vult 'array' met de waarden 1..n.**

**Notatie 10.** *Als a een array is met a[i] = x en a[j] = y, dan is $a^i_j$ een array waarvan alle elementen gelijk zijn aan die van a, behalve dat $a^i_j[i] = y$ en $a^i_j[i] = x$.*

**Hulpstelling 11.**

**$\{a\}$swap$(i,j)\{a^i_j\}$ mits $1 \leqslant i \leqslant n \wedge 1 \leqslant j \leqslant n$**

**Bewijs.** *We lopen de definitie van swap() van achteren naar voren af*
wp( „a[i]=t", {a[i] = y ∧ a[j] = x}) = {a[i] = y ∧ t = x}
wp( „a[i]=a[j]", {a[i] = y ∧ t = x} = {a[j] = y ∧ t = x}
wp( „t=a[i]", {a[j] = y ∧ t = x} = {a[j] = y ∧ a[i] = x} □

**Gevolg 12.**

**$\{a\}$swap$(i,j)$;swap$(i,j)\{a\}$ mits $1 \leqslant i \leqslant n \wedge 1 \leqslant j \leqslant n$**

**Gevolg 13.**

**$\{a^i_j\}$swap$(i,j)\{a\}$ mits $1 \leqslant i \leqslant n \wedge 1 \leqslant j \leqslant n$**

**Hulpstelling 14.**

**loop(i, j) doet niets voor 0 < j < i**

**Bewijs.** Dit is in te zien door de if-statements na te lopen. □

**Hulpstelling 15.**

**loop(i, j) drukt een permutatie van 1..n af en verder niets voor $i \leqslant 0$**

**Bewijs.** Het array moet van te voren gevuld zijn met een permuatie van 1..n (zie hulpsteling 9). De rest is af te leiden uit de if-statements en de code van de print() functie. □

**Stelling 16.**

**De aanroep van loop(i, j) termineert voor alle waarden van i en j**

**Bewijs.**

- **geval** $i \leqslant 0$
  Zie hulpstelling 15
- **geval** $j \leqslant 0 \wedge i > 0$
  Door de if-statements na te lopen zien we dat er niets gebeurt.
- **geval** $0 < j < i$
  Zie hulpstelling 14
- **geval** $0 < i \leqslant j$ *dit vereist een dubbele inductie*
  - loop( i, j-1);
    dit geeft de recursie loop(i, j) ↪ loop(i, j-1) ↪ .... ↪ loop(i, i), die stopt na (j-1) stappen, waarna de functie retourneert en de volgende stap uitvoert.
  - swap( i, j);
    die functie termineert in 3 stappen.
  - loop( i-1, n);
    dit geeft de recursie loop(i, n) ↪ loop(i-1, n) ↪ .... ↪ loop(0, n), die stopt na n doorgangen, die elk een eindig aantal stappen vergen.
  - swap( i, j);
    die functie termineert in 3 stappen. □

**Gevolg 17.** *De aanroep van loop(n, n) roept swap(i,j) aan met parameters* $1 \leqslant i \leqslant n \wedge 1 \leqslant j \leqslant n$.

**Hulpstelling 18.**

**$\{a\}$loop$(i,j)\{a\}$ mits $1 \leqslant i \leqslant n \wedge 1 \leqslant j \leqslant n$**

Het bewijs vereist weer dubbele inductie. De basis bestaat weer uit die gevallen waarin loop() niets doet of alleen het array uitprint. De volgende stap is een geval waarin de recursie niets doet, zodat er twee aanroepen van swap() overblijven, die volgens gevolg 12 het array niet veranderen. Volgens de inductiehypothese zal na afloop van een reeks van doorgangen waarvan geen het array wijzigt, hetzelfde gelden voor de complete loop.

**Hulpstelling 19.** *loop(1,j) zal, voor* $1 \leqslant j \leqslant n$, j *permutaties van 1..n afdrukken waarin het* $1^e$ *element is verwisseld met het* $1^e$, $2^e$,....$j^e$.

**Bewijs.** *We kunnen de code uitrollen tot*

```
loop( 1, j-1);
swap( 1, j);
print();
swap( 1, j);
```

**Basis**: $j = 1$ *(we lopen de code weer van achter naar voren door)*
$\{a_1^1\}$swap$(1,1)\{a\}$
$\{a_1^1\}$print$(1,1)\{$a *wordt afgedrukt*$\}$
$\{a\}$swap$(1,1)\{a_1^1\}$
$\{a\}$loop$(1,0)\{a\}$ *doet niets*
**Inductie: *j+1***
$\{a_{j+1}^1\}$swap$(1,j+1)\{a\}$
$\{a_{j+1}^1\}$print$(1,1)\{a_{j+1}^1$ *wordt afgedrukt*$\}$
$\{a\}$swap$(1,j+1)\{a_{j+1}^1\}$
$\{a\}$loop$(1,j)\{a\}$ *drukt* $a_j^1$ *af voor 1...j*. □

**Dit is vergelijkbaar met hulpstelling 5. We generaliseren het resultaat enigszins in het volgende lemma. We hebben twee inductie stappen nodig voor de dubbele recursie.**

**Gevolg 20.** *loop(i,j) zal, voor* $1 \leqslant j \leqslant n$, j *permutaties van 1..n afdrukken waarin het* $i^e$ *element is verwisseld met het* $i^e$, $(i+1)^e$,....,$n^e$.

**Hulpstelling 21.** loop( i, n) *zal, voor* $1 \leqslant i \leqslant n$, (n - i +1) *maal* loop( i-1, n) *aanroepen, nadat het* $i^e$ *element is verwisseld met het* $i^e$, $i+1^e$,....$n^e$ *en de permutaties zijn afgedrukt.*

**Bewijs.**
**Basis**: *loop( i, i)*
$\{a_i^i\}$swap$(i,i)\{a\}$
$\{a_i^i\}$loop$(i-1,n)\{a_i^i\}$ *(loop(i-1, n) wordt* $1^e$ *keer aangeroepen)*
$\{a\}$swap$(i,i)\{a_i^i\}$
$\{a\}$loop$(i,i-1)\{a\}$ *(doet niets)*
**Inductie:** *loop( i, j+1)*
$\{a_{j+1}^i\}$swap$(i,j+1)\{a\}$
$\{a_{j+1}^i\}$loop$(i-1,n)\{a_{j+1}^i\}$ *(loop(i-1, n) wordt (j-1+1)*$^e$ *keer aangeroepen)*
$\{a\}$swap$(i,j+1)\{a_{j+1}^i\}$
$\{a\}$loop$(i,j)\{a\}$ *(drukt* $a_j^i$ *af voor i...j; zie hulpstelling 20)* □

**Hulpstelling 22.** *De aanroep* loop( i, n) *zal voor* $1 \leqslant i \leqslant n$, *n!/(n-i)! permutaties afdrukken met*
$a_1^1$, $a_2^1$, $a_3^1$,...., $a_n^1$
$b_2^2$, $b_3^2$, $b_4^2$,...., $b_n^2$ *voor b in elke a in bovenstaande lijst*
.....
$c_i^i$, $c_{i+1}^i$, $c_{i+2}^i$,...., $c_n^i$ *voor c in elke permutatie in de voorafgaande lijst*

**Bewijs.**
**Basis:** *loop( 1, n)*
drukt volgens hulpstelling 19 $n$ permutaties af met $a_1^1$, $a_2^1$, $a_3^1$,...., $a_n^1$

**Inductie:** *loop( i+1, n)*
$\{a_n^{i+1}\}\text{swap}(i+1,n)\{a\}$ *(herstel uitgangssituatie)*
$\{a_n^{i+1}\}\text{loop}(i,n)\{a_n^{i+1}\}$ *($a_n^{i+1}$ permuteren op posities a[i] t/m a[n] volgens inductiehypothese)*
$\{a\}\text{swap}(i+1,n)\{a_n^{i+1}\}$ *(genereren $a_n^{i+1}$)*
$\{a\}\text{loop}(i+1,n-1)\{a\}$ *(genereren & afdrukken $a_{i+1}^{i+1}, a_{i+2}^{i+1}$,...., $a_{n-1}^{i+1}$ volgens lemma 20)* □

**Stelling 23.** *De aanroep van loop( n, n) zal alle permutaties van 1...n afdrukken.*

**Dit algoritme is gebaseerd op het factoriale getallenstelsel, waarin het n^e cijfer wordt genoteerd op basis van grondtal *n*, zodat $c_n c_{n-1}...c_2 c_1 c_0 = c_n \times (n+1)! + c_{n-1} \times n! ... + c_2 \times 3! + c_1 \times 2! + c_0 \times 1!$ (een technisch detail is dat we tellen vanaf 1 i.p.v. 0). De tabel hieronder toont in de linker kolom de volgnummers en in de rechter de permutaties. Zie bijlage F voor de broncode.**

| | |
|---|---|
| 0 0 0 0 | **4** 1 2 3 |
| 0 0 1 0 | **4** 2 1 3 |
| 0 1 0 0 | **4** 3 1 2 |
| 0 1 1 0 | **4** 3 2 1 |
| 0 2 0 0 | **4** 1 3 2 |
| 0 2 1 0 | **4** 2 3 1 |
| 1 0 0 0 | 3 **4** 2 1 |
| 1 0 1 0 | 3 **4** 1 2 |
| 1 1 0 0 | 2 **4** 1 3 |
| 1 1 1 0 | 1 **4** 2 3 |
| 1 2 0 0 | 2 **4** 3 1 |
| 1 2 1 0 | 1 **4** 3 2 |
| 2 0 0 0 | 3 1 **4** 2 |
| 2 0 1 0 | 3 2 **4** 1 |
| 2 1 0 0 | 2 3 **4** 1 |
| 2 1 1 0 | 1 3 **4** 2 |
| 2 2 0 0 | 2 1 **4** 3 |
| 2 2 1 0 | 1 2 **4** 3 |
| 3 0 0 0 | 3 1 2 **4** |
| 3 0 1 0 | 3 2 1 **4** |
| 3 1 0 0 | 2 3 1 **4** |
| 3 1 1 0 | 1 3 2 **4** |
| 3 2 0 0 | 2 1 3 **4** |
| 3 2 1 0 | 1 2 3 **4** |

**Tabel 2.** Factoriaal tellen

# 10 Intermezzo: variadische arrays

**BASIC geldt als een verouderde taal en het leren ervan wordt doorgaans afgeraden. Het nieuwe feature dat ik hier wil voorstellen is echter al bekend in een taal, namelijk Visual Basic, waar het 'dynamic arrays' genoemd wordt. Die term wordt echter in andere talen gebruikt om een array aan te duiden waarvan de lengte niet met de declaratie is vastgelegd, maar wordt bepaald tijdens de uitvoering van het programma. De term is verwant aan de variadische functies en macro's van C.**

**Een array heet dan variadisch als je bij de declaratie ten hoogste het type van de elementen hoeft te declareren. Tijdens de uitvoer moet het programma dan geheugen reserveren door het aantal dimensies op te gegeven en de lengte van elke dimensie. Het resultaat is dan een soort rechthoekig blok. In C zou je een variadische functie valloc() kunnen schrijven die het aantal dimensies en de lengtes als parameters ontvangt en onder water malloc() aanroept; een beter idee is een functie zijn die het aantal dimensies meekrijgt, plus een 1-dimensionaal array van lengtes.**

**Om elementen van een 4-dimensionaal array te lezen of schrijven zou je een notatie als var[1,2,3,4] kunnen gebruiken, maar als het aantal dimensies niet vantevoren vaststaat is het beter om iets als var[ind] te gebruiken waarin ind een 1-dimensionaal array van indexen is. Verder wil je een functie hebben die vertelt wat de dimensies van een variadisch array zijn.**

# 11 Intermezzo: variadische lussen

**Om een vier-dimensionaal array te verwerken kun je recursie gebruiken, maar een geneste for-loop is doorgaans sneller.**

```
for( i1 = 0; i1 < n1; i1++)
  for( i2 = 0; i2 < n2; i2++)
    for( i3 = 0; i3 < n3; i3++)
      for( i4 = 0; i4 < n4; i4++)
        var[i1][i2][i3][i4] = .....
```

**Om een variadisch array te verwerken heb je een vergelijkbare constructie nodig, waarin het aantal loops overeenkomt met de dimensie van het array. In een Pascal-achtige syntaxis zou je iets krijgen als**

```
for ind = low to high do
        var[ind] := ...;
```

**Hierin zouden 'low' en 'high' 1-dimensionale arrays zijn met de iteratie-grenzen, 'ind' een array met de loop indexen en 'var' een variadisch array, of de indexen worden meegegeven aan een variadische functie. Voor de compiler maker is het niet zo moeilijk om een dergelijke iterator te implementeren.**

```
BOOL    next( int ind[], int lim[])
{
  int   i;

  for( i = N; i > 0 && ind[i] >= lim[i]; i--)
    ind[i] = 1;
  if( i <= 0)
    return FALSE;
  else
  { ind[i]++;
    return TRUE;
  }
}
```

# 12 Iteratief permuteren in C

**Iteratieve code kan worden omgezet in recursieve code en omgekeerd. Dat kan op een vrij rechtlijnige manier door voor elke functieaanroep de argumenten op een zelfgemaakte stack te pushen en na terugkeer weer te poppen.**

**De dubbele recursie van bijlage B kan worden vervangen door een variadische for-loop. Hier valt de iterator iets ingewikkelder uit door dat er extra stappen met nullen zijn ingevoegd, zie tabel 2. Een andere iterator telt in aflopende volgorde. De output van deze iteratoren wordt gebruikt als index voor twee arrays van tellers die cyclisch van 0 t/m N tellen. Die tellers bepalen vervolgens welke elementen van het array *perm* worden verwisseld. De iteratieve code in bijlage C genereert dan de zelfde reeks verwisselingen als het recursieve origineel. Ze is minder efficiënt en begrijpelijk en het schrijven verliep met moeite.**

| | |
|---|---|
| 1000 | 1013 |
| 1100 | 1023 |
| 1110 | 1123 |
| 1111 | 2123 |
| 1112 | 3123 |
| 1113 | 0123 |
| 1114 | 1123 |
| 1120 | 1223 |
| 1121 | 2223 |
| 1122 | 3223 |
| 1123 | 0223 |
| 1124 | 1223 |
| 1130 | 1323 |
| 1131 | 2323 |
| 1132 | 3323 |
| 1133 | 0323 |
| 1134 | 1323 |
| 1200 | 1333 |
| 1210 | 1133 |
| 1211 | 2133 |
| 1212 | 3133 |
| 1213 | 0133 |
| 1214 | 1133 |
| 1220 | 1233 |
| 1221 | 2233 |
| 1222 | 3233 |
| 1223 | 0233 |
| 1224 | 1233 |
| 1230 | 1333 |
| 1231 | 2333 |
| 1232 | 3333 |
| 1233 | 0333 |
| 1234 | 1333 |

**Tabel 3.** Het tellen van de indexen

**In de vakliteratuur zijn nog meer oplossingen voor dit probleem bekend; de meeste zijn iteratief.**

# 13 Het Plain Changes algoritme

**De beschrijving van het Plain Changes algoritme is ontleend aan deel 4 van 'The Art of Computer Programming' van Donald Knuth, maar het werd al in 1668 gepubliceerd als een patroon om 5 verschillende kerkklokken om beurten te luiden. Zie ook Wikipedia, https://en.wikipedia.org/wiki/Change_ringing. Nog een praktische toepassing van permuteren is het recept voor vlechtbrood uit de tv-serie 'Heel Holland bakt', https://heelhollandbakt.omroepmax.nl/recepten/details/vlechtbrood-met-bakvideo/.**

**Het basisidee is om een permutatie van 1....$n-1$ te nemen en daarin het $n^e$ element op alle mogelijke plaatsen in te voegen (u merkt dat dit een recursieve definitie is en dat de andere algoritmen hetzelfde idee volgen). Voor n = 4 krijg je dan**

| | | | | | |
|---|---|---|---|---|---|
| 1234 | 1324 | 3124 | 3214 | 2314 | 2134 |
| 1243 | 1342 | 3142 | 3241 | 2341 | 2143 |
| 1423 | 1432 | 3412 | 3421 | 2431 | 2413 |
| 4123 | 4132 | 4312 | 4321 | 4231 | 4213 |

**Tabel 4.**

**We kunnen permutaties weergeven in de vorm van een *inversie tabel* $c_1c_2....c_n$ waarin $c_j$ staat voor het aantal elementen rechts van getal $j$, die kleiner zijn dan $j$.**

| | | | | | |
|---|---|---|---|---|---|
| 0000 | 0010 | 0020 | 0120 | 0110 | 0100 |
| 0001 | 0011 | 0021 | 0121 | 0111 | 0101 |
| 0002 | 0012 | 0022 | 0122 | 0112 | 0102 |
| 0003 | 0013 | 0023 | 0123 | 0113 | 0103 |

**Tabel 5.**

**Het algoritme drukt alle permutaties van een rij $a_1a_2....a_n$ verschillende getallen af door herhaaldelijk twee naburige elementen te verwisselen. Het maakt gebruik van een hulprij $c_1c_2....c_n$ die de inversies in een gegeven permutatie representeren waarvoor**

$$0 \leq c_j < j \text{ voor alle } 1 \leq j \leq n \tag{1}$$

**Procedure**

I. *Initialisatie*
Zet $c_j \leftarrow 0$ en $o_j \leftarrow 1$ voor alle $1 \leq j \leq n$

II. *Verwerken*
Druk permutatie $a_1a_2....a_n$ af

III. *Verandering voorbereiden*
Zet $j \leftarrow n$ en $s \leftarrow 0$ voor alle $1 \leq j \leq n$
*De volgende stappen bepalen de coördinaat* j *waarvoor* $c_j$ *gaat veranderen, die voldoet aan (1); variabele* s *is het aantal indices* $k > j$ *waarvoor* $c_k = k - 1$

IV. *Klaar voor verandering?*
Zet $q \leftarrow c_j + o_j$
Als q < 0, ga verder met stap VII.
Als q = j, ga verder met stap VI.

V. *Verandering* $c - c_j + s$
Verwissel $a_{j-c_j+s} \leftrightarrow a_{j-q+s}$
Zet $c_j \leftarrow q$
Ga verder met stap II

VI. *Ophogen* s
Als j = 1, beëindig het programma,
Zo niet, zet $s \leftarrow s+1$

VII. *Richting omkeren*
Zet $o_j \leftarrow -o_j$ en $j \leftarrow j-1$
Ga verder met stap IV.

**De uitgewerkte broncode is te vinden in bijlage D. Vermoedelijk is de recursieve methode van B.R. Heap de meest efficiënte, omdat ze per berekende permutatie slechts één verwisseling nodig heeft en geen hulptabellen gebruikt, maar dat ze correct werkt is moeilijker in te zien; zie bijlage E voor de code.**

# 14 Resultaten

**Op mijn laptop blijken de programma's in bijlagen B en D ongeveer 15 minuten te kosten, tegen 21 minuten voor dat van bijlage C, bij N = 12. Recursie is dus niet langzamer dan iteratie, ook al vinden er twee swaps plaats per keer en hier was de omzetting van recursie naar een equivalente iteratieve vorm duidelijk in het nadeel. Ook Heap's algoritme kwam op dezelfde rekentijd uit. De LISP code houdt het totale resultaat in geheugen, zodat de PC niet verder kwam dan N = 10 (10! = 3628800 sublijsten), wat ruim 1 minuut kostte; voor N =9 was LISP ongeveer 10 maal langzamer dan C.**

## 15 Conclusie

**Recursie is een nuttige programmeertechniek die overzichtelijke code oplevert. C programmeurs moeten opletten dat in sommige embedded omgevingen slechts een kleine hoeveelheid geheugen beschikbaar is voor de stack. In theorie zijn recursie en iteratie uitwisselbaar, maar het kost moeite om een recursieve functie om te zetten in een iteratieve.**

**Uit de analyse blijkt dat recursieve permutatieprogramma's in Lisp en C in feite hetzelfde werken. Functionele programma's zijn wat minder snel op een conventionele computer, maar met wat oefening gemakkelijk te schrijven en verifiëren.**

## 16 Literatuur

Sedgewick, R. Permutation Gerneration Methods. *Computing Surveys,*Vol 9, No 2, Juni 1977, 137 - 164

***Dit artikel is geschreven met GNU TeXmacs.***

# 17 Bijlage A: permutatie programma in Scheme

```
(define list1
  (lambda (x) (cons x '())))

(define map1
  (lambda (p ls)
    (cond ((null? ls)       '())
          (#t               (cons (p (car ls)) (map1 p (cdr ls))))
    )))

(define concat
  (lambda (x ls)
    (cond ((null? x)        ls)
          ((atom? x)        (cons x (concat ls '())))
          (#t               (cons (car x) (concat (cdr x) ls)))
     )))

(define cat1
  (lambda (y)
    (lambda (z) (concat y z))
  ))

(define interleave
  (lambda (x ls)
    (cond ((null? ls)       (list1 (list1 x)))
          ((null? (car ls)) (list1 (list1 x)))
          (#t               (concat (list1 (concat x ls))
                              (map1 (cat1 (car ls)) (interleave x (cdr ls)))))
    )))

(define inter1
  (lambda (x ls)
    (cond ((null? ls)    '())
          (#t            (concat (interleave x (car ls)) (inter1 x (cdr ls))))
    )))

(define permut
  (lambda (ls)
    (cond ((null? ls)    (list1 '()))
          (#t            (inter1 (car ls) (permut (cdr ls))))
    )))
```

# 18 Bijlage B: recursief permutatie programma in C

```c
#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>

#define  N              4
#define  swap(a,b)      swape( &perm[a], &perm[b])

unsigned char   perm[ N];

void    fill( void)
{
  int           i;

  for( i = 0; i < N; i++)
    perm[ i] = i + 1;
}

void    print( void)
{
  int           i;

  for( i = 0; i < N; i++)
    printf( "%01d ", perm[ i]);
  printf( "\n");
}

void    swape( unsigned char *a, unsigned char *b)
{
  unsigned char c;

  c = *a;
  *a = *b;
  *b = c;
}

void    loop( int i, int j)
{
  if( i < N
  {
    if( j < N - i)
    {
      swap( i , N - j - 1);
      loop( i+1, 0);
      swap( i , N - j - 1);
      loop( i, j+1);
    }
  }
  else
    print();
}

int
main( void)
{
  fill();
```

```
  loop( 0,0);
  return 0;
}
```

## 19 Bijlage C: iteratief permutatie programma in C

```
/* Een programma om alle permutaties van 1 .. N te genereren                */
#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>
#define N 4

void    print( int array[])             /* drukt de inhoud van het array af */
{ int   i;

  for( i = 1; i <= N; i++)
    printf( "%01d␣", array[ i]);
  printf( "\n");
}

void    fill_ind( int array[])          /* initialisatie routines           */
{ int   i;

  for( i = 1; i <= N+1; i++)
    array[ i] = 0;
  array[ 0] = N;
}

void    fill_ind1( int array[])
{ int   i;

  array[0] = 1;
  for( i = 1; i <= N+1; i++)
    array[ i] = i-1;
}

void    fill_max2( int array[])
{ int   i;

  array[0] = 1;
  for( i = 1; i <= N+1; i++)
    array[ i] = i;
}

void    fill_max( int array[])
{ int   i;

  array[0] = array[1] = 1;

  for( i = 1; i < N+1; i++)
    array[ i+1] = i;
}

void    swap( int *a, int *b)           /* verwissel twee elementen         */
{ int   c;
```

```
  c = *a;
  *a = *b;
  *b = c;
}

int     next( int ind[], int lim[])     /* iterator                          */
{ int   i, j;

  for( i = 1; i <= N+1 && ind[i] > 0; i++);
  if( i <= N)
  { ind[i]++;
    ind[i+1]=0;
  }
  else
  { for( i = N + 1; i > 0 && ind[i] >= lim[i]; i--)
    { ind[i] = 0;
    }
    if( i <= N+1
    { ind[i]++;
      for( j = i+1; j <= N+1; j++)
      { ind[j] = 0;
      }
    }
  }
  return i;
}

int     prev( int ind[], int lim[])     /* iterator in aflopende volgorde   */
{ int   i, j;

  for( i = N; i > 0  && ind[i] <= 1; i--);
  if( i >= 0)
  { if( ind[i+1] > 1)
    {  ind[i]--;
    }
    else
    { if( ind[i+1] == 0)
      { for( j = i + 1; j <= N; j++)
        { ind[j] = lim[j];
        }
        ind[i]--;
        i = N;
      }
      else
      { for( j = N + 1; j > i + 1 && ind[j] != 1; j--);
        ind[j] = 0;
        i = j - 1;
      }
    }
  }
  return i;
}

void    loop( void)                     /* buitenste lus                     */
{ int   lim[ N + 2];
```

```
  int   lim2[ N + 2];
  int   ind[ N + 2];
  int   ind1[ N + 2];
  int   ind2[ N + 2];
  int   ind3[ N + 2];
  int   stack[ N + 2];
  int   perm[ N + 1];                   /* nulde element wordt niet gebruikt*/
  int   h, i, j, k, l, m;

  fill_max2( perm);
  fill_ind( ind);
  fill_max( lim);
  fill_max2( lim2);
  fill_ind1( ind1);
  fill_max2( ind2);
  fill_max2( ind3);
  fill_ind( stack);

  ind3[ N]++;
  stack[0] = 0;
  m = 0;
  while(( l = next( ind, lim2)) > 0)
  { k =  N + 1 - l;
    ind1[k] = ind1[k] + 1;
    if( ind1[k] > N)
      ind1[k] = k;

    if( ind1[k] == N)                   /* push */
    { stack[++m] = k;
    }
    if((k == 0 && ind1[k] == N))
    { print( perm);
      for( h = 0; h <= N; h++)
        if( stack[m] == h)
        { --m;                          /* pop */
          i = prev( ind3, lim2);
          j =  N + 1 - i;

          swap( &perm[j], &perm[ind2[j]]);

          ind2[j] = ind2[j] + 1;
          if( ind2[j] > N)
            ind2[j] = j;
        }
        else
          break;
    }
    else
    { if( k != 0 )
      { swap( &perm[k], &perm[ind1[k]]);
      }
    }
  }
}

int     main( void)                   /* het programma begint hier        */
```

```
{ loop();
  return 0;
}
```

# 20 Bijlage D. Iteratief plain changes algoritme in C

```
#include <stdio.h>
#define  N              4
#define  swap(a,b)      swape( &perm[a], &perm[b])

signed char     perm[ N+1];
signed char     invr[ N+1];
signed char     dirc[ N+1];

void    print( signed char perm[])      /* een permutatie afdrukken         */
{
  int           i;

  for( i = 1; i <= N; i++)
    printf( "%01d␣", perm[ i]);
  printf( "\n");
}

void    swape( signed char *a, signed char *b) /* elementen verwisselen     */
{
  signed char c;

  c = *a;
  *a = *b;
  *b = c;
}

void    init( int n)                    /* arrays initialiseren             */
{
  int           i;

  for( i = 1; i <= n; i++)
  { perm[i] = i; invr[i] = 0; dirc[i] = 1;
  }
}

void    loop( int n)                    /* genereer de permutaties          */
{
  int           j, q, s;

  init( n);                             /* P1. Initialise                   */
  while( 1)
  {
    print( perm);                       /* P2. Visit permutation            */
    j = n; s = 0;                       /* P3. Prepare for change           */
    while( 1)
    {
      q = invr[j] + dirc[j];            /* P4. Ready to change?             */
      if( q >= 0)
```

```
      { if( q != j)
        { swap( j - invr[j] + s, j - q + s); /* P5. Change and continue     */
          invr[j] = q;
          break;
        }
        else
        { if( j == 1)                  /* P6. Increase S or terminate      */
            return;
          else
            s += 1;
        }
      }
      dirc[j] = -dirc[j];              /* P7. Switch direction             */
      j -= 1;
    }
  }
}

int     main( void)                    /* Hoofdprogramma                   */
{
  loop( N);
  return 0;
}
```

## 21 Bijlage E. Permutatie algoritme volgens Heap in C.

```
#include <stdio.h>

#define  N              4
#define  swap(a,b)      swape( &perm[a], &perm[b])

unsigned char   perm[ N+1];

void    fill( void)
{
  int           i;

  for( i = 1; i <= N; i++)
    perm[ i] = i;
}

void    print( void)
{
  int           i;

  for( i = 1; i i <= N; i++)
    printf( "%01d␣", perm[ i]);
  printf( "\n");
}

void    swape( unsigned char *a, unsigned char *b)
{
  unsigned char c;
```

```
  c = *a;
  *a = *b;
  *b = c;
}

void    loop( int i)
{
  int           j;

  if( i == 1)
    print();
  for( j  = 1; j <= i; j++)
  {
    loop( i - 1);
    swap( i % 2 ? 1 : j, i);
  }
}

int    main( void)
{
  fill();
  loop( N);
  return 0;
}
```

## 22 Bijlage F. Factoriaal tellen in C

```
#include <stdio.h>
#define n 4

int array[ n + 1];                      /* nulde element is ongebruikt      */

void print( void)                       /* drukt inhoud van het array af    */
{
  int i;

  for( i = 1; i <= n; i++)
    printf( "%01d␣", array[ i]);
  printf( "\n");
}

void loop1( int i, int j)               /* recursief                        */
{
  if( i <= n)
  {
    if( j <= n - i)
    {
      array[ i] = j;
      loop1( i + 1, 0);
      loop1( i, j + 1);
    }
  }
  else
    print();
```

```
}

void loop2( void)                       /* iteratief                       */
{
  int i1, i2, i3, i4;

  for( i1 = 0; i1 < 4; i1++)
  {
    array[1] = i1;
    for( i2 = 0; i2 < 3; i2++)
    {
      array[2] = i2;
      for( i3 = 0; i3 < 2; i3++)
      {
        array[3] = i3;
        for( i4 = 0; i4 < 1; i4++)
        {
          array[4] = i4;
          print();
        }
      }
    }
  }
}

int main( void)                         /* hoofdprogrammma                 */
{
  loop1( 1, 1);
  return 0;
}
```

# 23 Bijlage G. Saigon (Iteratieve versie Torens van Hanoi

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

typedef  int                    BOOL;
#ifndef  TRUE
#define  TRUE                   1
#endif
#ifndef  FALSE
#define  FALSE                  0
#endif
#ifdef   linux
#include <stdint.h>
#define  FMT                    "%lu. "
#endif
#ifdef   _WIN32
#include <wtypes.h>
#define  FMT                    "%llu. "
typedef  ULONG64                uint64_t;
#endif
#define  N              3
```

```
char            perms[6][N]     = { {1,2,3},
                                    {1,3,2},
                                    {2,1,3},
                                    {2,3,1},
                                    {3,1,2},
                                    {3,2,1}
                                  };
char            stack[N]        = {1,2,3};
char            apply2[6];
char            apply6[6];

/* retourneert het i-e bit van x */
BOOL
get_bit( const uint64_t x, const int i)
{
  uint64_t      y               =  (uint64_t) 1 << i;

  return (x & y) == 0 ? FALSE : TRUE;
}

/* retourneert de recursiediepte */
int
get_depth( uint64_t i)
{ int           j;

  for( j = 0; j < 64; j++)
    if( get_bit( i, j) == 1)
      break;
  return j + 1;
}

/* retourneert het nummer van een permutatie */
int
search( char van, char naar)
{
  int           i;

  for( i = 0; i < 6; i++)
    if( perms[i][0] == van && perms[i][1] == naar)
      break;
  return i + 1;
}

/* permuteert een permutatie -- niet zi snel */
int
apply( int i)
{
  char          out[N];

  out[0] = stack[(int) perms[i-1][0] - 1];
  out[1] = stack[(int) perms[i-1][1] - 1];
  out[2] = stack[(int) perms[i-1][2] - 1];
  memcpy( stack, out, sizeof( out));
  return search( stack[0], stack[1]);
}
```

```
/* vult de arrays apply2 en apply6 */
void
fill( void)
{
  int           i;

  for( i = 1; i <= 6; i++)
  { stack[0] = 1, stack[1] = 2, stack[2] = 3;
    apply( i);
    apply2[i-1] = apply( 2);
    stack[0] = 1, stack[1] = 2, stack[2] = 3;
    apply( i);
    apply6[i-1] = apply( 6);
  }
}

/* loopt alle stappen af */
void
saigon( int depth)
{
  uint64_t      i;
  int           d, prev_d, j, perm;

  d = 0; perm = 1;
  for( i = 1; i < (uint64_t) 1 << depth; i++)
  { prev_d = d, d = depth - get_depth( i);
    if( d > prev_d)
      for( j = prev_d + 1; j <= d; j++)
        perm = get_bit( i, depth - j) ? apply6[ perm-1] : apply2[ perm-1];
    else
      for( j = prev_d; j > d; j--)
        perm = get_bit( i - 1, depth - j) ? apply6[ perm-1] : apply2[ perm-1];

    printf( FMT, i);
    printf( "Verplaats␣schijf␣van␣%c␣naar␣%c.\n",
            perms[perm-1][0] + '@', perms[perm-1][1] + '@');
  }
}

/* het hoofdprogramma met aantal schijven als parameter -- default 5 */
int main( int argc, char **argv)
{
  int           n               = 5;

  if( argc == 2)
    n = atoi( argv[1]);

  fill();
  saigon( n);
  return 0;
}
```