Hoofdstuk 7

Algoritmen voor programma's

7.1 Van kleine instructies naar grote processen

De processen die een computer uitvoert zijn in deze methode al een aantal keren aan de orde geweest. We hebben het gehad over programma's voor beeldbewerking, browsers, editors en routeplanners. In hoofdstuk 3 hebben we de hardware die deze processen uitvoert bekeken. Het centrale deel van de computer is de processor, de CPU. De processor voert eenvoudige instructies uit die achter elkaar in het geheugen van de machine zijn opgeslagen. Het bijzondere daarbij is dat de processor die serie instructies heel precies en heel snel uitvoert. Moderne processoren kunnen tientallen miljoenen instructies per seconde uitvoeren en daarbij wordt geen enkele fout gemaakt.

Zo'n lange reeks computerinstructies is een *computerprogramma* of kortweg *programma*. De instructies die een computer kan uitvoeren, zijn erg primitief. Alle computers kunnen in principe alleen tellen en cijfers onthouden. Als je tot 10 kunt tellen, kun je ook zelf elke computerinstructie uitvoeren. Door de hoge snelheid en de 100% nauwkeurigheid waarmee de computer de instructies uitvoert, kunnen computersystemen dingen doen die voor ons onmogelijk zijn. Een computer kan een erg complexe berekening binnen een uur maken, terwijl een snelle rekenaar in honderd jaar de uitkomst van die berekening nog niet heeft gevonden en daarbij ongetwijfeld heel wat (reken)fouten zal maken. Dat is immers menselijk! Er zijn nog meer verschillen tussen verwerking door mensen en verwerking door een computer. Een mens heeft minstens een dag nodig om een lijvig boek van duizend pagina's te lezen. Elke computer leest de instructies in een programmaboek van duizend pagina's in enkele seconden en verricht tevens de bijbehorende acties. Maar terwijl een mens een boek heel persoonlijk interpreteert, produceren alle computers die hetzelfde programmaboek hebben doorgewerkt, hetzelfde resultaat.

Als we een computer iets voor ons willen laten doen, kan dat alleen als we de opdrachten voor het werk kunnen omzetten in de instructies voor de processor. Dat kan uitsluitend voor routinewerk. Routinewerk levert onafhankelijk van de uitvoerende persoon (of computer) in dezelfde omstandigheden telkens precies hetzelfde resultaat op. Gedetailleerde beschrijvingen van routinewerk noemen we *algoritmen*. Delen van algoritmen noemen we *procedures*. Als we de algoritmen zodanig beschrijven dat een computer ze kan uitvoeren, noemen we ze programma's. De opdrachten in de procedures en de algoritmen zijn dan de instructies voor de computer.
Elke processor kent een zeer beperkt aantal *instructies*. Er zijn verschillen tussen de instructies van verschillende typen processoren, maar in principe gaat het steeds om dezelfde eenvoudige bewerkingen. Je kunt ze in een kort lijstje samenvatten.

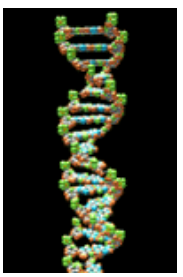
Invoeren	Voer een cijfer in
Onthouden	Onthoud een cijfer
Optellen	Tel twee cijfers op
Uitvoeren	Voer een cijfer uit
Vergelijken	Bepaal of een cijfer groter is dan een ander
Kiezen	Kies een serie instructies
Herhalen	Herhaal een serie instructies

Tabel 1 Lijst met primitieve instructies voor een processor

Je ziet dat het aantal mogelijke instructies nogal beperkt is, hiermee kom je niet verder dan het heen en weer schuiven van gegevens en het maken van eenvoudige berekeningen. Hoe kunnen computers dan toch zulke wonderlijke dingen voor ons doen?

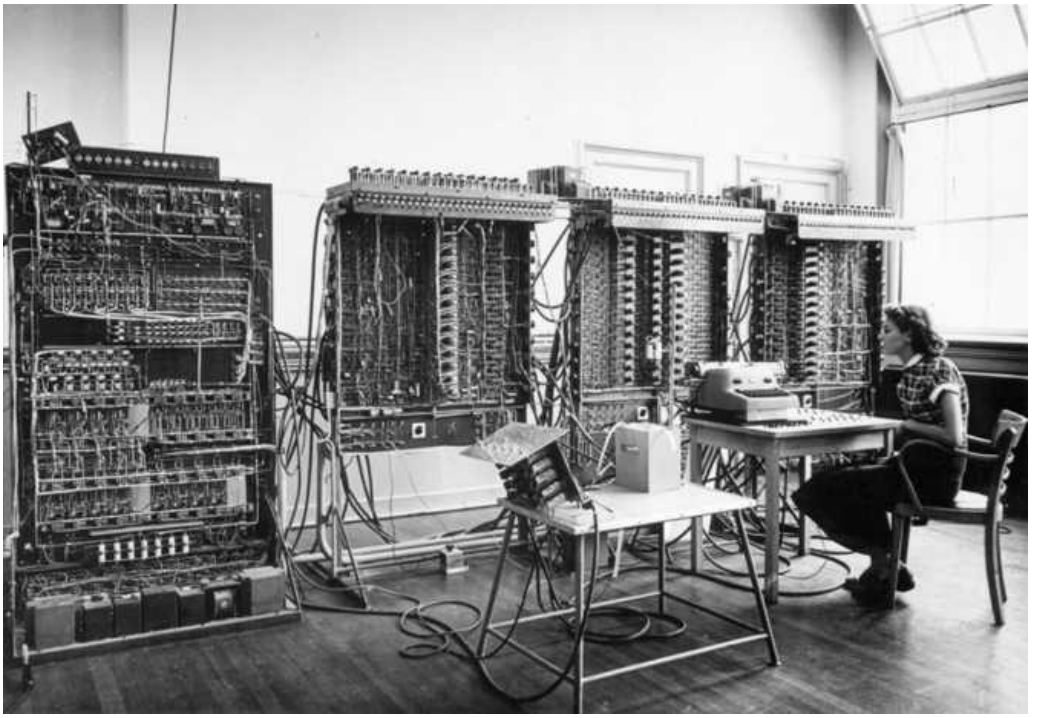
Een wonderlijk model

We kunnen met onze tien cijfers ontelbaar veel getallen noteren en bijvoorbeeld een serie telefoonboeken vullen. Met het alfabet van 26 tekens hebben we in Nederland al meer dan 300.000 woorden gedefinieerd (kijk Van Dale er maar eens op na) en miljoenen boeken vol geschreven. Dat lijkt ook zeer (ver)wonderlijk. Zelfs als we buiten beschouwing laten wat mensen in andere landen met dezelfde 26 letters hebben bedacht en geschreven.
Die wonderlijke prestaties van computers ontstaan door de primitieve instructies in verschillende volgordes achter elkaar te plaatsen en die verschillende reeksen in grote aantallen samen te voegen tot steeds grotere brokken. Dat gebeurt net zoals we regels en zinnen tot alinea's vormen, die op hun beurt in paragrafen tot hoofdstukken worden samengevoegd. Ten slotte vormen de hoofdstukken het boek.



In dit model leidt een beperkt aantal primitieve onderdelen door vorming van steeds grotere bouwstenen tot grote ingewikkelde constructies, die onze verwondering opwekken. Het is vergelijkbaar met het model dat in de biotechnologie wordt gebruikt en dat de basis vormt van elke levensvorm. Daarin worden vier zogenaamde 'basen' als primitieve onderdelen in lange ketens aan elkaar gekoppeld tot DNA en daaruit worden de genen en daarmee alle verschijningsvormen van het leven opgebouwd.
Toch bestaat er een belangrijk verschil tussen deze modellen. Het model van de biotechnologie wordt gebruikt om bestaande complexe structuren te analyseren en te registreren hoe de bouwstenen van het leven in elkaar zitten. Met het model dat in de informatica wordt toegepast kunnen met de primitieve instructieset nieuwe toepassingen worden ontwikkeld.

Aan deze voorbeelden zie je dat het mogelijk is om met eenvoudige bouwstenen ingewikkelde zaken te maken. Hetzelfde geldt voor ingewikkelde computerprogramma's. Uiteindelijk bestaan die alleen uit simpele instructies zoals in tabel 1.
In de praktijk hebben we niets aan deze constatering. Grote programma's als Microsoft Word bestaan uit miljoenen instructies. Voor een programmeur is het ondoenlijk om zo'n programma helemaal in machinetaal-instructies te schrijven. Het programma is daarvoor gewoon te groot. Eén foutje in al die miljoenen instructies en het programma werkt niet....
Voor de oplossing van dit probleem duiken we in de geschiedenis. Bij de allereerste computers uit de jaren 40 van de vorige eeuw moesten de instructies nog met de hand in het geheugen van de computer worden ingevoerd. Je moet dit letterlijk nemen. Het geheugen was een schakelbord. Ieder bitje moest met de hand worden ingevoerd, door een schakelaar goed te zetten. Deze computers verwerkten daarom alleen kleine programma's.



Figuur 1 De ARRA, de Automatische Relais Rekenmachine Amsterdam. Dit was in 1952 de eerste computer in Nederland, nog zonder toetsenbord of beeldscherm!

Later werd de methode om een programma in te voeren verbeterd, door het gebruik van ponskaarten en nog later toetsenborden. Het probleem van de eenvoudige instructies was daarmee nog niet opgelost. Dat gebeurde pas toen iemand een briljant idee kreeg:

*"In plaats van een programma in machinetaal in te voeren, zou ik veel liever mijn programma als leesbare tekst willen invoeren. Ik kan zo'n programmeertaal bedenken. De computer kan programma's uitvoeren voor allerlei taken. Als ik nu een programma maak dat in staat is om opdrachten in mijn taal om te zetten in machine-instructies, dan wordt het leven veel eenvoudiger. Ik hoef nog maar één keer een programma in machinetaal te schrijven, het vertaalprogramma. Daarna kan ik ieder volgend programma in mijn eigen taal schrijven. De computer kan het **zelf** omzetten in machinetaal-instructies met behulp van het vertaalprogramma."*

Zo'n vertaalprogramma heet een *compiler*. Sindsdien zijn er veel programmeertalen bedacht en zijn er voor die talen compilers geschreven. In die talen kun je in één keer opdrachten geven die eigenlijk uit veel machinetaalinstructies bestaan.

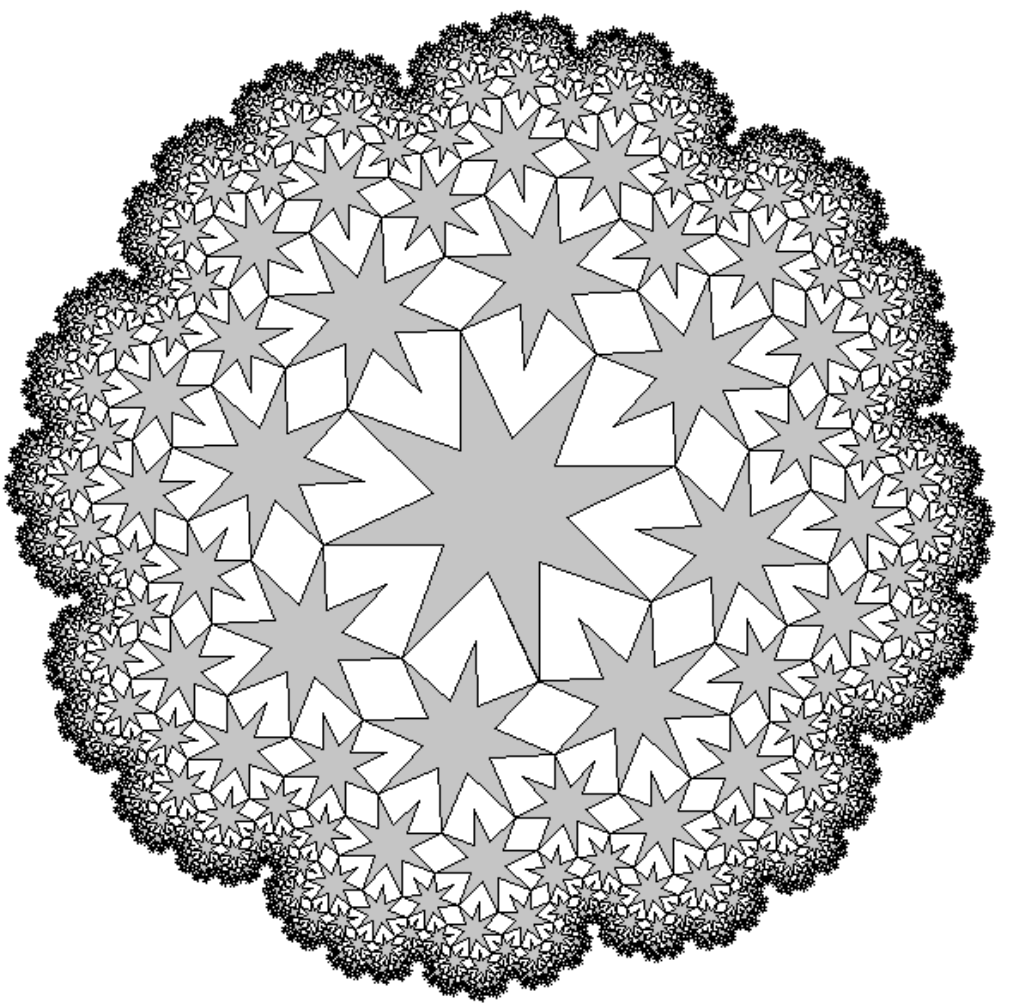
Veel programmeertalen hebben opdrachten waarmee je lijnen kunt tekenen. Meestal kan dat in een paar regels, zoals:

```
setColor(blue);
drawLine(10, 10, 400, 400);
```

Deze regels zorgen ervoor dat er een blauwe lijn van schuin over het scherm getekend wordt, van pixel (10, 10) naar pixel (400, 400). Hiervoor zijn zeer veel machinetaal-instructies nodig. De processor moet uitrekenen waar de pixels in het geheugen staan. Daarna moet op die plaatsen de RGB-waarde voor de kleur blauw gezet worden. De instructies die daarvoor nodig zijn staan allemaal in tabel 1. Als programmeur heb je niets meer van doen met deze instructies. De compiler regelt dit voor jou.

7.2 Tekenen met opdrachten

Dankzij hogere programmeertalen en compilers hoeft een programmeur niet meer bezig te zijn met meest basale machinetaalinstructies. Toch blijft een programmeur iemand die met behulp van eenvoudige bouwstenen complexe bouwwerken creëert. Elk computerprogramma gebruikt algoritmen, waarin met behulp van relatief eenvoudige opdrachten complexe taken worden beschreven. Je moet dus heel exact kunnen beschrijven wat er moet gebeuren. Als je onderstaande tekening bekijkt, dan kun je begrijpen dat 'exact beschrijven wat je moet doen' niet altijd makkelijk is.



Figuur 2 Een ingewikkelde tekening

De tekenopdrachten die nodig zijn voor deze tekening staan in het onderstaande programma. Dit ingewikkelde programma hoeft je nu niet te begrijpen, maar je ziet dat de beschrijving verrassend kort is voor zo'n ingewikkelde tekening. Dat komt omdat het algoritme gebruik maakt van allerlei handige structuren. Daarover gaat dit hoofdstuk.

```
import logotekenap.*;

public class SterFractal extends TekenApplet
{
    private InvoerVariabele aantalNivInv;
    private double z,factor;
    private int maxNiv;

    public void initialiseer()
    {   aantalNivInv = new InvoerVariabele("aantal niveaus",0,13,4);
        maakZichtbaar(aantalNivInv);
        z = 50; maxNiv = 4;
        factor = 1/(Math.sqrt(2+Math.sqrt(2)));
    }

    public void tekenprogramma()
    {   penUit();stap(0,-70);penAan();
        ster(z,"oranje");
        for(int i=0 ; i<8 ; i++)
        {   rechts(112.5);
            sterFractal(maxNiv,factor*z,true,"oranje");
            links(112.5);
            penUit();sterStap(z);penAan();
        }
    }
    void ster(double z, String kl)
    {   vulAan(kl);
        for(int i=0 ; i<8 ; i++)
        {   sterStap(z);
        }
        vulUit();
    }
    void sterFractal(int niv ,double z, boolean vol, String kl)
    {   if(niv==0)return;
        ster(z,kl);
        if(niv-1>0)
        {   penUit();sterStap(z);sterStap(z);penAan();
            rechts(112.5);
            if(vol)sterFractal(niv-1,z*factor,true,kl);
            else sterFractal(niv-1,z*factor,false,kl);
            links(112.5);
            penUit();sterStap(z);penAan();
            rechts(112.5);
            if(vol)sterFractal(niv-1,z*factor,true,kl);
            links(112.5);
            penUit();sterStap(z);penAan();
            rechts(112.5);
            if(vol)sterFractal(niv-1,z*factor,false,kl);
            links(112.5);
            for(int i=0 ; i<4 ; i++)
            {   penUit();sterStap(z);penAan();
            }
        }
    }
    void sterStap(double z)
    {   vooruit(z);    rechts(110);
        vooruit(z);    links(155);
    }
    public void invoerVarActie(InvoerVariabele iv)
    {   maxNiv = (int)aantalNivInv.geefWaarde();
        tekenOpnieuw();
    }
}
```

Programma voor de tekening van figuur 2

In dit hoofdstuk laten we dus zien hoe algoritmen worden gemaakt en welke principes en hulpmiddelen hierbij een rol spelen. Daarvoor kijken we naar algoritmen voor het maken van tekeningen met behulp van eenvoudige tekenopdrachten. De opdrachten besturen een denkbeeldige pen over een computerscherm. In het digitaal practicum 'Tekenen in Java' kun je met behulp van deze algoritmen programma's schrijven die deze tekeningen ook daadwerkelijk maken op het computerscherm.

De tekenopdrachten

Laten we eerst maar eens kijken naar de belangrijkste opdrachten waarmee we onze algoritmen gaan bouwen.

penAan(kleur)

Hiermee zet je een denkbeeldige pen aan. Tussen de haakjes kun je aangeven met welke kleur er getekend moet worden.

penUit()

Hiermee haal je de denkbeeldige pen weer van het scherm. Je kunt daarna de pen verplaatsen zonder dat er iets getekend wordt.

vooruit(aantal)

Met deze opdracht beweegt de pen een aantal eenheden vooruit. Hiermee wordt dus echt getekend.

rechts(hoek)

Hiermee veranderen we de tekenrichting van de pen. We laten de tekenrichting van de pen zoveel graden naar rechts (met de wijzers van de klok mee) draaien als er tussen haakjes is opgegeven

links(hoek)

Dit is eenzelfde soort opdracht, maar nu wordt de tekenrichting van de pen naar links gedraaid. Weer over het aantal graden dat tussen de haakjes is opgegeven.

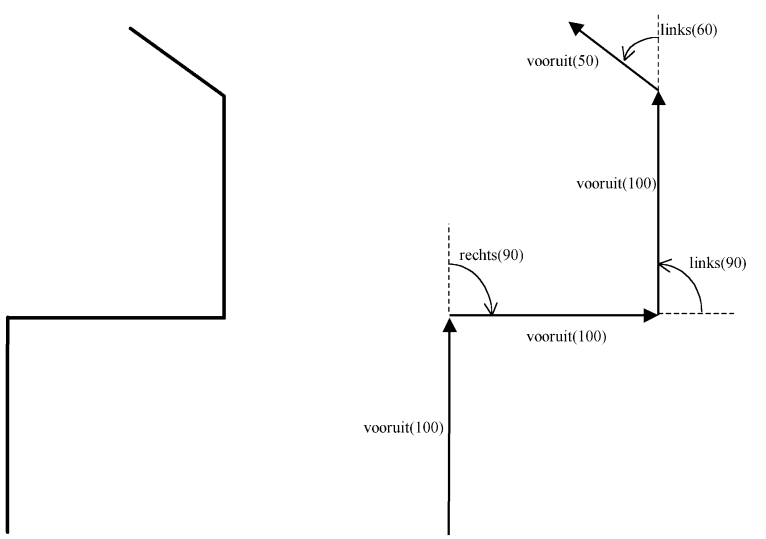
vulAan(kleur)

Deze opdracht kun je gebruiken om vlakken in te kleuren.

vulUit()

Deze opdracht gebruik je in combinatie met vulAan.

In figuur 3 staat een eenvoudige tekening die je met deze opdrachten kunt maken.



Figuur 3 Een eenvoudige lijntekening

Het bijbehorende algoritme (beschrijving met behulp van de opdrachten) ziet er zo uit:

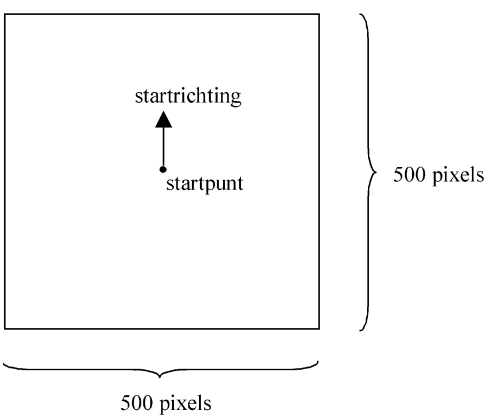
```
penAan(zwart)
vooruit(100)
rechts(90)
vooruit(100)
links(100)
vooruit(100)
links(60)
vooruit(50)
penUit()
```

Algoritme 1

In de rechterkant van figuur 3 kun je precies volgen wat de opdrachten doen.

Het tekenblad

Bij het maken van algoritmen voor een tekening ga je uit van een denkbeeldig tekenblad (zie figuur 4). In het digitaal practicum 'Tekenen in Java' waarin je de tekenprogramma's gaat maken is dit tekenblad uiteraard het beeldscherm.



Figuur 4 Tekenblad

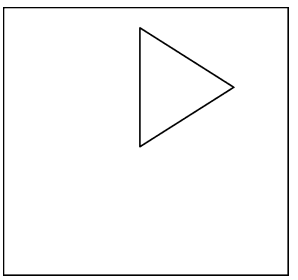
Het tekenblad telt 500 bij 500 pixels. Het startpunt de pen is altijd het midden van het blad en de pen is zo georiënteerd dat er in de richting van de bovenkant van het blad getekend wordt. Natuurlijk kun je binnen je algoritme opdrachten geven waardoor het tekenen niet in het midden begint en ook de tekenrichting kun je in je programma wijzigen. Toch is het noodzakelijk dat je weet welke uitgangspositie de pen aan het begin heeft en hoe groot het werkblad is.

Vulopdrachten

Figuren bestaan niet alleen uit lijnen maar ook uit vlakken. Vooral als die vlakken gekleurd zijn kun je aardige effecten bereiken. Daarvoor heb je een paar extra opdrachten nodig: vulAan() en vulUit(). De driehoek van figuur 5 kun je tekenen door de opdrachtenreeks:

```
vooruit(100)
rechts(120)
vooruit(100)
rechts(120)
vooruit(100)
rechts(120)
```

Algoritme 2

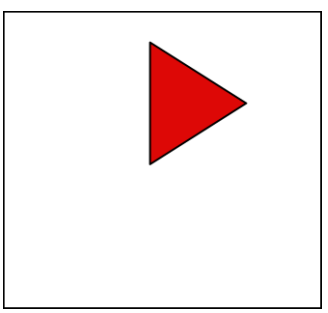


Figuur 5 Lijntekening

Als we de driehoek een kleur willen geven, moet je de reeks opdrachten met twee opdrachten uitbreiden. Je krijgt dan de tekening van figuur 6.

```
vulAan("rood")
vooruit(100)
rechts(120)
vooruit(100)
rechts(120)
vooruit(100)
rechts(120)
vulUit()
```

Algoritme 3



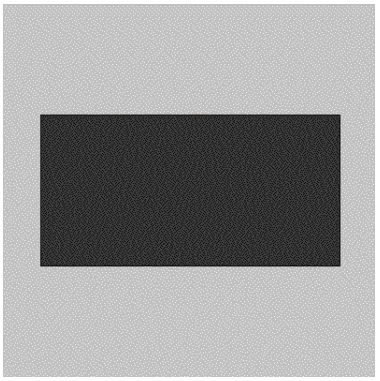
Figuur 6 Lijntekening met gekleurd vlak

7.3 Variabelen gebruiken in een algoritme

Als je een algoritme schrijft voor een tekening, dan is het soms handig om met variabelen te werken. Het kan zijn dat je de tekening iets andere afmetingen wilt geven en dan is het vervelend om het hele algoritme te herschrijven. Als je variabelen gebruikt, dan kun je dat voorkomen Bekijk het volgende algoritme dat een zwarte rechthoek tekent.

```
vulAan("zwart")
vooruit(200)
rechts(90)
vooruit(400)
rechts(90)
vooruit(200)
rechts(90)
vooruit(400)
rechts(90)
vulUit()
```

Algoritme 4



Figuur 7 Rechthoek

In het algoritme is de breedte van de rechthoek 400 en de hoogte 200. Stel nu dat we de rechthoek een andere breedte en hoogte willen geven. Er zit dan niets anders op dan in de code van het programma alle getallen voor de hoogte en de breedte te veranderen. Vooral bij ingewikkelde tekeningen heb je dan grote kans fouten te maken.
Een veel betere oplossing is de hoogte en de breedte voorlopig open te laten. Je doet dit door er *variabelen* van te maken. Dat begrip komt uit de wiskunde. Denk maar aan een vergelijking met meer variabelen. In de vergelijking 2x –3y = 5 zijn x en y de variabelen. Dat is bij programmeren eigenlijk net zo. Bekijk het volgende programma maar. Daarin heten de variabelen breedte en hoogte.

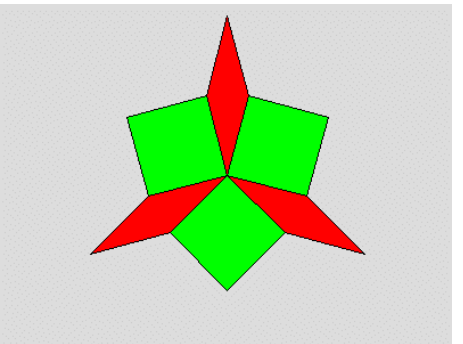
```
hoogte = 200
breedte = 400

vulAan("zwart")
vooruit(hoogte)
rechts(90)
vooruit(breedte)
rechts(90)
vooruit(hoogte)
rechts(90)
vooruit(breedte)
rechts(90)
vulUit()
```

Algoritme 5

Het voordeel van dit algoritme is duidelijk. Wil je de rechthoek andere afmetingen geven, dan hoef je alleen de getallen achter *hoogte*= en achter *breedte*= te veranderen. Je kent dan aan de variabelen breedte en hoogte andere waarden toe en de tekenopdrachten kunnen hetzelfde blijven.

7.4 Deeltaken bij Java

De algoritmen die we tot nu toe bekeken bestaan uit een rij opdrachten die na elkaar worden uitgevoerd. Dat is een belangrijk kenmerk van bijna alle algoritmen. Daarnaast zagen we dat het handig kan zijn om met variabelen te werken. Maar er zijn nog andere manieren die helpen bij het efficient en overzichtelijk beschrijven van algoritmen.
In figuur 8 zie je een figuur die opgebouwd is uit vierkanten en ruiten. Natuurlijk is het mogelijk om alle opdrachten voor het tekenen van de vierkanten en de ruiten achter elkaar te zetten, maar het kan efficiënter door deeltaken te beschrijven. Maak een deeltaak om een vierkant te tekenen en noem deze 'Maak vierkant'. Maak ook een deeltaak om een ruit te tekenen: 'Maak ruit'.



Figuur 8 Ruiten en vierkanten

Het algoritme kan zo worden opgeschreven:

```
Maak vierkant
rechts(30)
Maak ruit
rechts(90)
Maak vierkant
rechts(30)
Maak ruit
rechts(90)
Maak vierkant
rechts(30)
Maak ruit

Maak vierkant
vulAan("groen")
vooruit(120)
links(90)
vooruit(120)
links(90)
vooruit(120)
links(90)
vooruit(120)
links(90)
vulUit()
Maak ruit
vulAan("rood")
vooruit(120)
links(30)
vooruit(120)
links(150)
vooruit(120)
links(30)
vooruit(120)
links(150)
vulUit()
```

Algoritme 6

Een deeltaak is dus eigenlijk een zelfgemaakte opdracht die een vooraf opgegeven serie basisopdrachten uitvoert. De voordelen zijn duidelijk. Het algoritme is op deze manier korter en overzichtelijker beschreven.

7.5 Herhalingen gebruiken

Bij het beschrijven van een algoritme voor het tekenen van figuren moet je vaak een opdracht of een reeks van opdrachten herhalen. Bijvoorbeeld bij het tekenen van een vierkant met een zijde van 100 zien we vier keer dezelfde twee opdrachten:

```
vooruit(100)
rechts(90)
vooruit(100)
rechts(90)
vooruit(100)
rechts(90)
vooruit(100)
rechts(90)
```

Algoritme 7

Door gebruik te maken van herhalingen kunnen we dat korter opschrijven:

```
Herhaal 4 keer
{  vooruit(100)
   rechts(90)
}
```

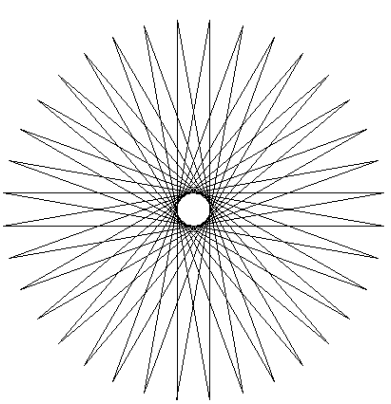
Algoritme 8

Een constructie zoals deze noemen we een *herhalingsopdracht* of *herhalingslus*.

Hieronder zie een algoritme voor een 36-puntige ster. Je kunt zien hoe krachtig de herhalingsopdracht is. Er zijn slechts enkele regels nodig voor de beschrijving van het algoritme.

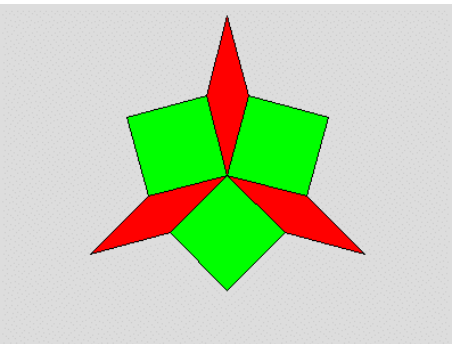
```
Herhaal 36 keer
{  vooruit(400)
   rechts(170)
}
```

Algoritme 9



Figuur 9 36-ster

Eerder zagen we een algoritme voor een tekening met drie vierkanten en drie ruiten. Met behulp van herhalingen kunnen we dit algoritme korter schrijven.



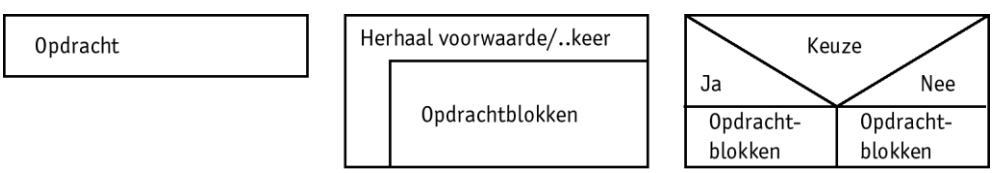
Figuur 10 Ruiten en vierkanten

```
Herhaal 3 keer
{   Maak vierkant
    rechts(30)
    Maak ruit
    rechts(90)
}

Maak vierkant
vulAan("groen")
Herhaal 4 keer
{   vooruit(120)
    links(90)
}
vulUit()

Maak ruit
vulAan("rood")
Herhaal 2 keer
{   vooruit(120)
    links(30)
    vooruit(120)
    links(150)
}
vulUit()
```

Algoritme 10

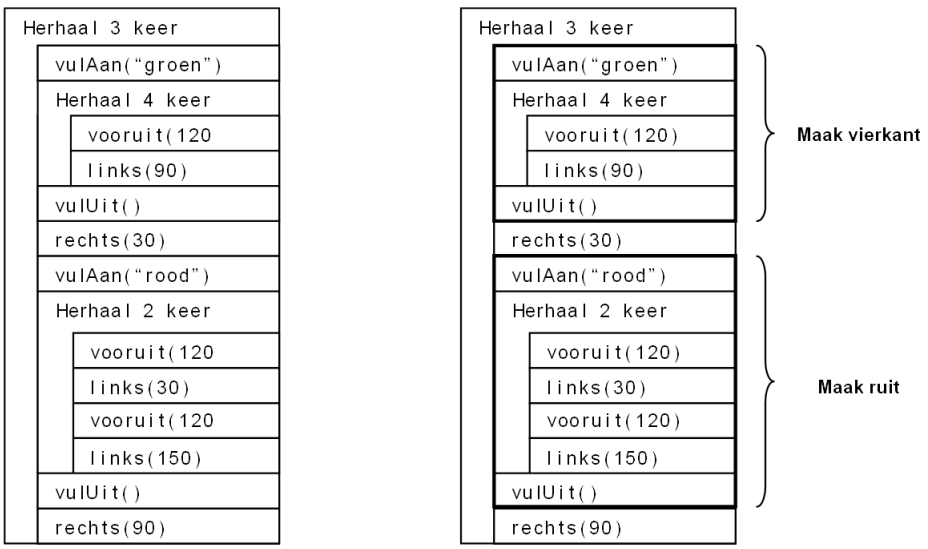
7.6 Structuurdiagrammen

In de loop van de tijd zijn er verschillende middelen gezocht om algoritmen op een overzichtelijke manier te kunnen beschrijven. Zeker als er veel herhalingen herhalingen en deeltaken worden gebruikt en als het algoritme omvangrijk is, dan ontstaat de behoefte aan een duidelijke structuur. *Structuurdiagrammen* zijn een middel om dat te bereiken. Deze diagrammen bestaan uit blokken waarin geschreven wordt welke handeling moet worden uitgevoerd. In figuur 11 zie je de gebruikte blokken.



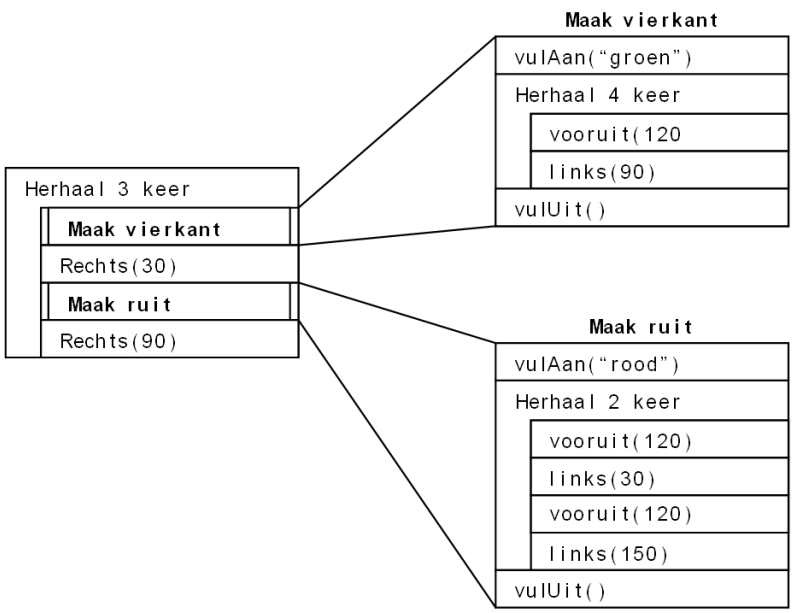
Figuur 11 Basisblokken van een structuurdiagram

We kijken nogmaals naar het algoritme voor de tekening met de ruiten en vierkanten. Als we dit algoritme beschrijven met behulp van structuurdiagrammen, dan ziet dat er zo uit:



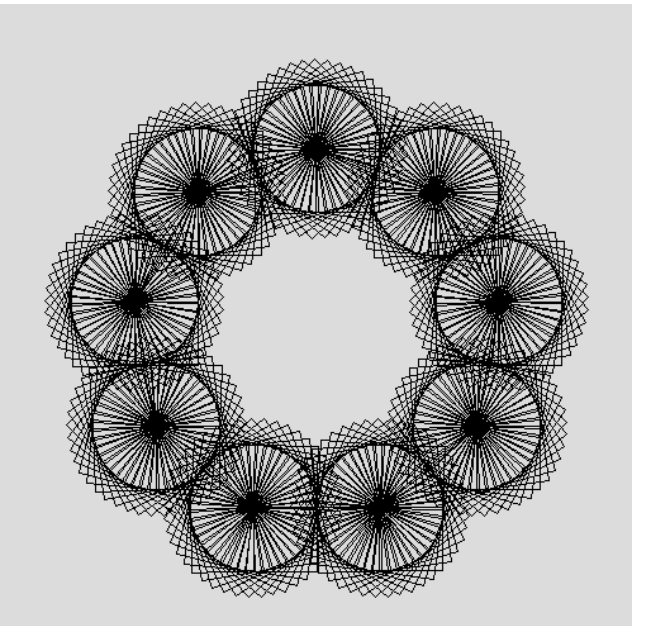
Figuur 12 Structuurdiagram voor de tekening met ruiten en vierkanten

Je ziet de deeltaken "Maak vierkant" en "Maak ruit" als blokken in het diagram. Bij grote structuurdiagrammen worden die blokken vaak uit het hoofddiagram gehaald en apart weergegeven. Het hoofddiagram wordt op die manier kleiner en de structuur blijft duidelijker. Je krijgt dan het diagram in figuur 13.



Figuur 13 Structuurdiagram voor de tekening met ruiten en vierkanten met apart weergegeven deeltaken

Tenslotte nog een algoritme van een ingewikkeldere tekening waarin deeltaken en herhalingsopdrachten handig worden gecombineerd om een krans te tekenen.



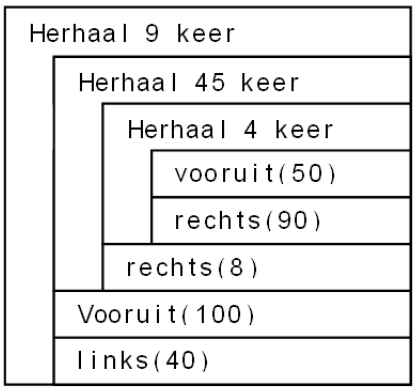
Figuur 14 Krans

```
Herhaal 9 keer:
{   Maak rozet
    vooruit(100)
    links(40)
}

Maak rozet
Herhaal 45 keer
{   Maak vierkant
    rechts(8)
}

Maak vierkant
Herhaal 4 keer
{   vooruit(50)
    rechts(90)
}
```

Algoritme 11

In dit algoritme worden negen rozetten getekend. Elke rozet bestaat uit 45 vierkanten. Elk vierkant is ten opzichte van het vorige 8 graden naar rechts gedraaid. Het algoritme tekent een negenhoek met op elk hoekpunt een rozet.

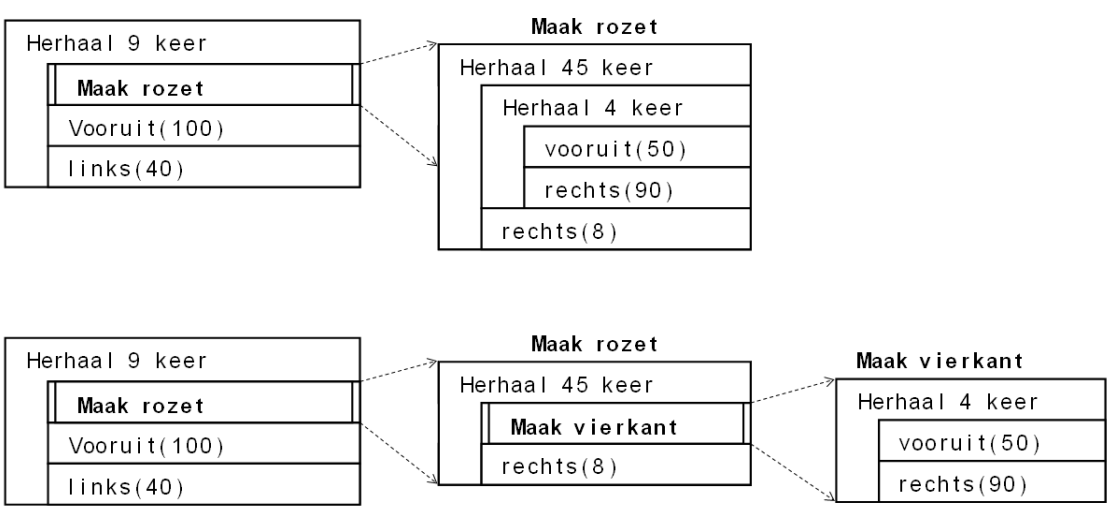
Een compleet structuurdiagram van dit algoritme zie je hieronder.



Figuur 15 Structuurdiagram krans van 9 rozetten

In het structuurschema is te zien dat hier sprake is van drie geneste herhalingslussen. Dat wil zeggen dat binnen de ene herhalingslus de andere wordt opgeroepen. De buitenste lus is bestemd voor het tekenen van 9 rozetten. Om die te kunnen tekenen wordt binnen deze lus een tweede lus aangeroepen voor het tekenen van 45 vierkanten. De laatste herhalingslus tekent een viertal lijnstukken die samen een vierkant vormen.
Omdat het hier om een negenhoek gaat is de hoek waarover gedraaid moet worden om bij het beginpunt van de volgende rozet te komen gelijk aan 40 graden (360 : 9). Bovendien moet de pen daarvoor over een afstand van 100 eenheden in de tekenrichting (= 2x de zijde van een vierkant) verplaatst worden. Bij elke rozet hebben de vierkanten het middelpunt van de rozet als gemeenschappelijk hoekpunt. Omdat er 45 vierkanten per rozet getekend worden, is de hoek waarover gedraaid wordt na het tekenen van een vierkant gelijk aan 8 graden (360 : 45).

Je kunt de deeltaken voor het tekenen van een rozet en een vierkant als blokken in de diagram herkennen en benoemen. Daarmee kun je diagram ook opsplitsen in drie kleinere diagrammen. Hieronder is dat gedaan



Figuur 16 Structuurdiagram krans van 9 rozetten met deeltaken

7.7 Taken met keuzeopdrachten

Tot nu ging onze aandacht naar taken voor tekenopdrachten en hoe deze taken op een efficiënte en overzichtelijke manier konden worden beschreven in een algoritme. Maar er zijn natuurlijk taken te bedenken voor de computer die niets met tekenen te maken hebben. In deze paragraaf bekijken we taken waarin keuzes een rol spelen. Ook voor deze taken formuleren we algoritmen en structuurdiagrammen.

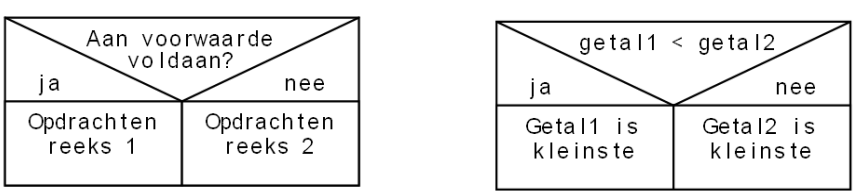
We bekijken de taak waarbij we een rij met getallen moeten sorteren op grootte. Als de rij niet te lang is, dan is dat voor mensen een eenvoudige opdracht. Echter als je nauwkeurig en stapsgewijs wilt beschrijven hoe dat moet (als je een algoritme wilt maken), dan komt daar nog wel wat bij kijken.
We beginnen met een eenvoudigere deeltaak: vind het kleinste getal in een rij met getallen. Als je deze taak nauwkeurig gaat beschrijven in woorden, dan krijg je zoiets als:

Bekijk het eerste en tweede getal en bepaal de kleinste. Als het eerste getal het kleinst is, vergelijk dan het eerste getal met het derde en bepaal weer de kleinste. Als het tweede getal kleiner is dan het eerste, vergelijk dan het tweede getal met het derde en bepaal welke het kleinst is, daarna....

Je kunt je voorstellen dat dit een heel onoverzichtelijk verhaal wordt bij een rij van tien getallen. Laten we kijken hoe we dat overzichtelijker kunnen maken met structuurdiagrammen.
Wat meteen opvalt is het gebruik van woorden ‘als’ en ‘dan’. We maken eigenlijk voortdurend keuzes op grond van voorwaarden. We zien steeds een patroon, dat we een *keuzeopdracht* noemen:

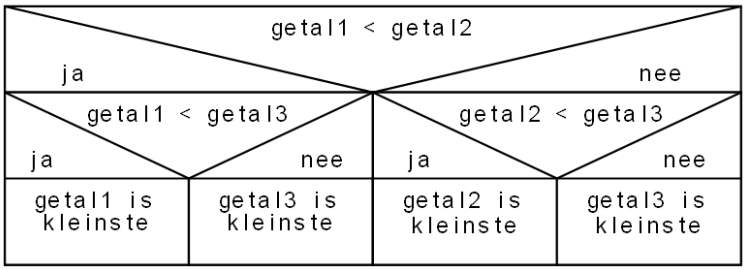
```
ALS   [voorwaarde]
   DAN   [voer deze opdracht uit]
   ANDERS  [voer andere opdracht uit]
```

Dit patroon geven we zo weer in een structuurdiagram:



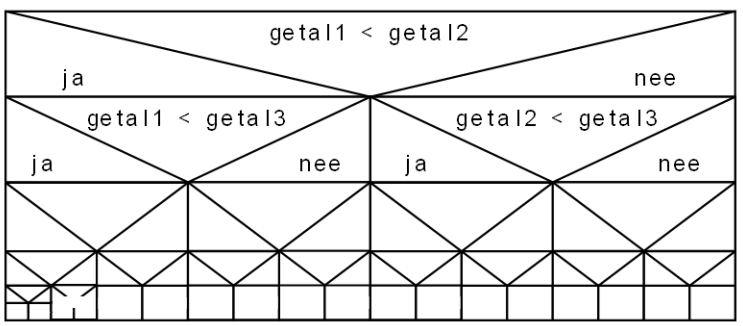
Figuur 17 Structuurdiagram Keuzeopdracht

Rechts in bovenstaande figuur zie je het structuurdiagram voor het bepalen van de kleinste van twee getallen. Hieronder zie je het uitgebreide structuurdiagram voor dezelfde opdracht met drie getallen.



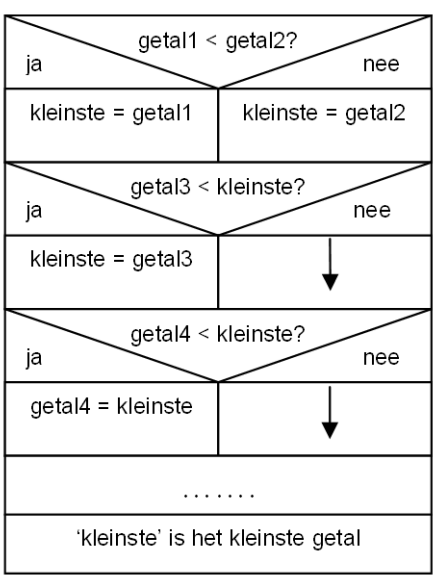
Figuur 18 Structuurdiagram Kleinste van 3 getallen

Zo'n structuurdiagram geeft meer overzicht en houvast dan de beschrijving waarmee we begonnen. Toch lukt het op deze manier niet bij tien getallen. Dat wordt al snel duidelijk als we het diagram proberen uit te breiden naar zes getallen.



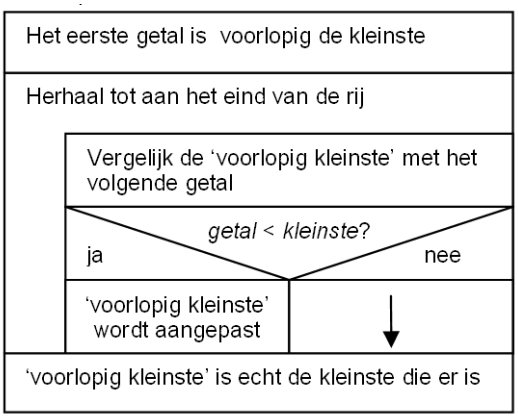
Figuur 19 Structuurdiagram Kleinste van 6 getallen

Als we het iets anders aanpakken, dan is het mogelijk om een structuurdiagram te maken zonder geneste keuzeblokken (keuzeblokken die gevat zijn in andere andere keuzeblokken). We maken dan gebruik van een variable '*kleinste*' die we gebruiken om het (voorlopig) kleinste getal te onthouden. Het structuurdiagram ziet er dan zo uit:



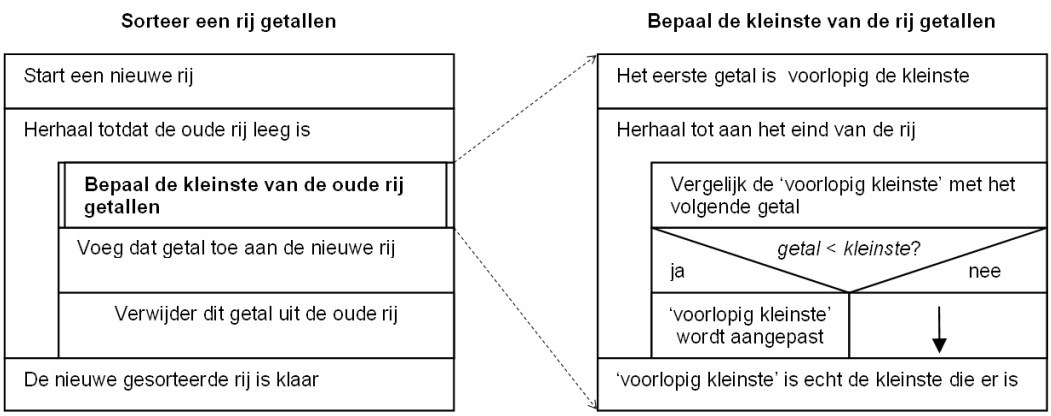
Figuur 20 Structuurdiagram kleinste bepalen

We kunnen natuurlijk ook nog gebruik maken van herhalingen zodat het diagram nog kleiner wordt:



Figuur 21 Structuurdiagram Keuze met herhalingen

Nu gaan we terug naar de oorspronkelijk opdracht, het sorteren van een rij getallen van klein naar groot. We hebben beschreven hoe we het kleinste getal in een rij bepalen. Sorteren kan nu door de kleinste weg te halen en van de rest weer de kleinste te bepalen. Als we al de verkregen 'kleinsten' in een nieuwe rij zetten, dan bevat die rij alle getallen van de oude rij, gesorteerd van klein naar groot. In een structuurdiagram ziet dat er zo uit:



Figuur 22 Structuurdiagram getallen sorteren