MC SYLLABUS 16.1

L. GEURTS

CURSUS PROGRAMMEREN

DEEL 1
DE ELEMENTEN VAN HET PROGRAMMEREN

TWEEDE DRUK

MATHEMATISCH CENTRUM AMSTERDAM 1976

AMS (MOS) onderwerpen classificatie schema (1970): 68-01
ACM-Computing Reviews-categorie: 1.52

ISBN 90 6196 080 0

Eerste druk 1973
Tweede druk 1976 (ongewijzigd)

Inhoud

Voorwoord

Hoewel programmeren bekend staat als een lastig handwerk, wordt er weinig aandacht besteed aan het overdragen van eenmaal verworven vaardigheid op dit gebied. Weliswaar verschijnen er in vaktijdschriften regelmatig publikaties over de manier waarop een programma ontwikkeld zou moeten worden, toch is er geen boek te vinden dat aan de beginner leert hoe hij moet programmeren.
Programmeerkursussen volstaan in het algemeen met het tonen van de elementen van een bepaalde programmeertaal en de eigenaardigheden van een bepaald input-output-systeem, en geven slechts terloops enige wenken die de beginnende programmeur moeten behoeden voor het maken van dezelfde fouten die voorgaande generaties gemaakt hebben.
Terwijl leerling en leraar alle aandacht nodig hebben voor het ontlopen van de voetangels en klemmen van de vereiste notatie en de gehanteerde kodes, blijft de leerling wat het eigenlijke programmeren betreft - het moeilijkste facet - in de kou staan. Het konstrueren van een algoritme lijkt in deze opzet als een aangeboren vaardigheid beschouwd te worden, die hoogstens wat getraind kan worden, maar waarover weinig opvoedends te zeggen valt.
Toch kan iemand die het alleen om het leren van de betreffende programmeertaal gaat - hij is al een bekwaam programmeur in een andere taal - , ook nauwelijks bij deze traditionele vorm van opleiding terecht. Het behandelen van de eigenschappen van de taal wordt voortdurend onderbroken door, in zijn ogen, triviale opmerkingen over de mogelijkheden die de betreffende eigenschappen de programmeur bieden. Ook zal hij weinig te weten komen over de finesses van de taal die hij onder de knie wil krijgen, want deze kunnen niet met vrucht geleerd worden aan beginners - voor wie de kursus dan mede bedoeld is - omdat van hun geheugen, abstraktievermogen en verbeeldingskracht al het uiterste gevergd wordt door de globalere mogelijkheden van de taal.

In deze kursus wordt geprobeerd deze problemen te omzeilen.

In deel I van deze kursus wordt gepoogd de aspirant-programmeur de beginselen van de kunst van het programmeren bij te brengen. Naast enige zeer algmene konstrukties als *while* ... *do* ... en *if* ... *then* ... (*else* ...) en het omzichtig geïntroduceerde begrip "variabele", wordt in dit deel het Nederlands gebruikt voor het formuleren van algoritmen. Grote nadruk wordt gelegd op de manier van programmeren die wel "stapsgewijze verfijning" genoemd wordt ("stepwise refinement" bij Wirth [7], "structured programming" bij Dijkstra [2,8], "programming by action clusters" bij Naur [4]). Bij deze methode wordt, nadat het proces in zeer algemene termen is geformuleerd, door successieve uiteenrafeling van deze termen tenslotte het niveau van gedetailleerdheid bereikt dat door de programmeertaal vereist wordt.
Ook wordt aandacht besteed aan het verifiëren van de korrektheid van algoritmen door het beschouwen van de tekst van de algoritme, in plaats van door "testen".

In deel II van de kursus wordt de programmeertaal ALGOL 60 geïntroduceerd. Omdat de kursist inmiddels geen onbeschreven blad programmeerpapier meer is, kan de taal betrekkelijk recht-toe-recht-aan gepresenteerd worden. Dit gebeurt volgens "top to bottom"-aanpak, aan de hand van het definiërende rapport van ALGOL 60 [1].

Omdat programmeren een vaardigheid is en niet een wetenschap, is het opdoen van ervaring van beslissend belang. De lopende tekst van de kursus wordt hiertoe veelvuldig onderbroken door opgaven, waarvan de uitwerking achterin het boek beschouwd moet worden als een belangrijk deel van de stof, dat voor het volgen van de kursus niet gemist kan worden.

Leo Geurts.

Literatuur:

1. Backus e.a. *Revised report on the algorithmic language ALGOL 60*, Copenhagen, 1962.
2. Dijkstra, E.W. *Notes on structured programming. EDW 249*, T.H. Eindhoven, 1969.
3. Henderson, P. en R. Snowdon An experiment in structured programming, *BIT 12* (1972) *38-53*.
4. Naur, P. Programming by action clusters, *BIT 9* (1969) *250-258*.
5. - - An experiment on program development, *BIT 12* (1972) *347-365*.
6. Wirth, N. Programming and programming languages. *Proc. International Computing Symposium*, Bonn, 1970.
7. - - Program development by stepwise refinement. *CACM 14* (1971) *221-227*.

8. *Structured Programming*, Academic Press (1972):
   Dijkstra, E.W. Notes on structured programming.
   Hoare, C.A.R. Notes on data structuring.
   Dahl, O.-J. en C.A.R. Hoare. Hierarchical program structures.

0. Inleiding

Opdat een computer doet wat we willen, moeten we hem een nauwkeurige taakomschrijving geven. In het dagelijks leven werken we ook wel met taakomschrijvingen; een gerecht wordt bereid volgens een recept, een modelvliegtuig wordt in elkaar gezet aan de hand van een bouwvoorschrift, een zelfbouw-versterker wordt gebouwd volgens een schema, een trui wordt gebreid naar een patroon.
In de wiskunde wordt een voorschrift, bestaande uit een serie elementaire opdrachten, een algoritme genoemd; een algoritme die bedoeld is voor een computer heet een programma. Zo'n programma wordt opgesteld in een voor de computer geschikt notatie-systeem, een programmeertaal (of algoritmische taal).
Aan het probleem van het opstellen van een programma zijn twee aspeckten te onderscheiden:
- het eigenlijke programmeren: het bedenken van de algoritme
- het noteren van de algoritme in een programmeertaal.
Het aspekt van het bedenken en opstellen van algoritmen, los van het gebruik van een bepaalde programmeertaal, is het onderwerp van dit eerste deel van deze cursus.

1. Programmeren in een notedop: zoeken in een woordenboek

1.1. Eerste gebruiksaanwijzing

Een gebruiksaanwijzing bij een woordenboek zou zo kunnen luiden:

> Zoek de eerste bladzijde af en daarna steeds de volgende totdat U het gezochte trefwoord tegenkomt.

Over deze eenvoudige gebruiksaanwijzing valt heel wat op te merken. Wat wel het eerst opvalt is dat de aangegeven methode niet zo slim is: om het woord "zwoerd" te vinden, moeten we bijna het hele woordenboek doorbladeren. Later zullen we een snellere manier bespreken; nu gaan we nader in op deze gebruiksaanwijzing. Er valt namelijk meer op aan te merken: het is niet duide-

lijk wat er precies moet gebeuren nadat we het gezochte woord ook op de laatste bladzijde niet hebben aangetroffen, en er geen volgende bladzijde is. (Natuurlijk is het ons duidelijk dat we dan kunnen ophouden met zoeken, omdat het woord niet in het woordenboek blijkt te staan, maar we moeten er vast aan wennen dat een algoritme ook voor dergelijke gevallen de gang van zaken moet aangeven).
Ook in een ander opzicht is de gebruiksaanwijzing niet volledig: er staat niet hoe je moet vaststellen of het gezochte woord op een bepaalde bladzijde voorkomt. Maar dit soort onvolledigheid is onvermijdelijk. Elke gebruiksaanwijzing kan nu eenmaal alleen beschreven worden in termen die zelf niet meer verklaard worden. (Het lokaliseren van een woord op een bladzijde kan wel nader worden uiteengerafeld, maar dan blijven er ook weer termen over als vergelijk de eerste letter van beide woorden, die niet nader verklaard worden.)
Zo zal ook een programmeur die een algoritme bedenkt, deze voor zichzelf niet direkt tot in de kleinste details specificeren. Hij zal de algoritme eerst opbouwen uit grotere bouwstenen, en zijn ervaring zegt hem dat hij die later zonder grote problemen zal kunnen uitwerken tot in de gedetailleerdheid die de programmeertaal vereist.

1.2. Een betere formulering

Laten we nu de gegeven gebruiksaanwijzing tot een sluitende algoritme herschrijven, en daarbij gebruik maken van bouwstenen als neem de volgende bladzijde, kijk of het woord op deze bladzijde voorkomt en ga na of er nog verdere bladzijden zijn. De kern van het proces zal bestaan uit een herhaalde aktiviteit, namelijk:

> zolang het woord niet op de onderhavige bladzijde staat, en er nog verdere bladzijden zijn, de volgende bladzijde nemen en doorgaan.

De hier gebruikte konstruktie zolang ... , doe ... komt bij het programmeren zeer vaak van pas. Deze konstruktie is zo algemeen, dat hogere programmeertalen daar een eigen notatie voor hebben. Wij zullen de volgende notatie gebruiken:

while ... *do* ...

De komplete algoritme ziet er nu zo uit:

> neem de eerste bladzijde;
> *while* het woord staat niet op deze bladzijde
> en er zijn nog verdere bladzijden *do*
> neem de volgende bladzijde

De algoritme bestaat uit twee stukken. Het eerste is de noodzakelijke voorbereiding voor het zoekproces. Zonder deze initialisatie zou de eerste keer niet duidelijk zijn wat deze bladzijde betekent. We zien hier een andere konventie van ons notatie-systeem: tussen twee stukken van de algoritme die achtereenvolgens moeten worden uitgevoerd, plaatsen we een punt-komma.

1.3. Een stroomschema

In fig. 1 wordt de werking van de algoritme op een andere manier weergegeven. Zo'n schema wordt een flow diagram of stroomschema genoemd.
Als we een woord opzoeken dat in het woordenboek voorkomt, en we zoeken volgens de algoritme, dan hebben we het woord gevonden als we bij STOP belanden.

1.4. Een efficiëntere zoek-algoritme

We hebben nu een gebruiksaanwijzing die wel korrekt is, maar niet zo vlot: om een woord te vinden moeten we gemiddeld het halve woordenboek doorbladeren. Door ervan gebruikt te maken dat de woorden in alfabetische volgorde staan, kunnen we een snellere gebruiksaanwijzing opstellen, die er in grote lijnen zo uitziet:

> Kijk of het woord in de eerste of in de tweede helft van het woordenboek moet voorkomen en ga in die helft verder met halveren, totdat er nog maar een bladzijde over is.

Wat verder uitgewerkt:

> Gebruik bij het opzoeken van de bladzijde waar het gezochte woord staat steeds twee wijsvingers: de linkerwijsvinger wijst steeds naar

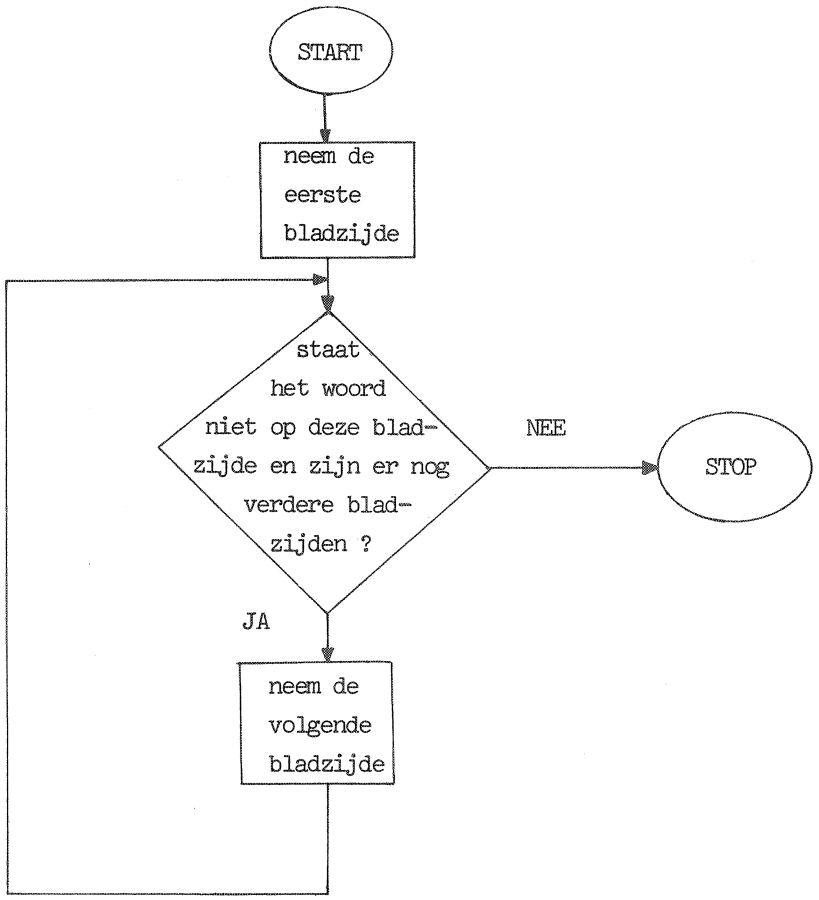


fig. 1 Stroomdiagram van een eenvoudige algoritme voor het opzoeken van een woord in een woordenboek.

een bladzijde die alfabetisch voor de gezochte bladzijde komt, de rechterwijsvinger naar een bladzijde na de gezochte.
Begin met de linkerwijsvinger bij de eerste bladzijde van het woordenboek, en de rechter bij de laatste bladzijde. Open nu het woordenboek ongeveer halverwege met beide duimen. Als de bladzijde daar alfabetisch voor de gezochte komt, verplaats dan de linkerwijsvinger naar die bladzijde, anders de rechterwijsvinger. Zo bevindt zich de gezochte bladzijde nog steeds tussen beide wijsvingers, maar het aantal nog te doorzoeken bladzijden is ongeveer gehalveerd.
Steek nu weer beide duimen halverwege in het pak bladzijden tussen beide wijsvingers, enz. totdat er geen bladzijde meer zit tussen de beide bladzijden waar de wijsvingers naar wijzen. Op dat moment wijst de linker- of de rechterwijsvinger naar de gezochte bladzijde, als het woord althans in het woordenboek voorkomt.

Neem een naam in gedachten en zoek die volgens de gegeven algoritme in een telefoonboek op. Doe dit met een aantal namen, en noteer steeds het benodigde aantal halveringen. Merk op dat dit aantal nagenoeg konstant is.

1.5. Betere formulering

De beschrijving van de algoritme is wat omslachtig, en wordt afgewisseld door uitleg en argumentatie. Als de opsteller van deze gebruiksaanwijzing zijn vinding aan vakbroeders wil bekendmaken via New Directions, vaktijdschrift voor opstellers van gebruiksaanwijzingen, zal hij dat, schrijvend voor ingewijden, wellicht zo doen:

(A1) Onderwerp: een algoritme voor het zoeken van een element *z* in een rij van *N* elementen die van klein naar groot geordend is.

laat *onderwijzer* wijzen naar element nr. *1;*
laat *bovenwijzer* wijzen naar element nr. *N;*
while er is nog minstens een element tussen *onderwijzer* en *bovenwijzer* *do*
begin laat *duim* wijzen naar element ergens halverwege tussen *onderwijzer* en *bovenwijzer;*

```
            if z minstens zo groot als het element waarnaar de
               duim wijst then
            laat onderwijzer wijzen naar het element waarnaar
            de duim wijst else
            laat bovenwijzer wijzen naar het element waarnaar
            de duim wijst
        end
        {Als nu onderwijzer of bovenwijzer naar een element wijst dat ge-
         lijk is aan z, dan is z gevonden, anders komt z in de rij niet voor.}
```

In dit voorbeeld worden enige nieuwe vaktermen gebruikt:

begin ... *end* betekent: het tussenliggende stuk algoritme moet als een eenheid worden opgevat.
In dit geval komt het erop neer dat het hele stuk van *begin* tot *end* herhaald moet worden, zolang aan de voorwaarde is voldaan die tussen *while* en *do* staat.

if ... *then* ... *else* ... betekent: als aan de voorwaarde na *if* voldaan is, doe dan wat tussen *then* en *else* staat (en sla wat na *else* staat over), en doe anders alleen wat na *else* staat (en sla wat tussen *then* en *else* staat over).

Onder ergens halverwege tussen twee elementen zullen we in dit voorbeeld verstaan: precies halverwege als er daar een element ligt, en anders, als we tussen twee elementen zouden uitkomen, dan de plaats van het tweede van die twee elementen. (In het rijtje ABC ligt B dus halverwege A en C, in het rijtje ABCD ligt C halverwege A en D.)

1.6. Korrektheid van de algoritme

Hoe kunnen we er nu zeker van zijn dat deze algoritme het proces beschrijft dat we willen? Door met de hand de algoritme voor enkele voorbeelden uit te voeren, kunnen we een redelijk intuïtief begrip vormen van de algoritme - dat de algoritme korrekt is lijkt dan vanzelfsprekend. Er kan niet nadrukkelijk genoeg op gewezen worden dat dit een misvatting is, en wel een zeer

gevaarlijke. De praktijk van het programmeren is helaas, en volkomen onnodig, de praktijk van het maken van programmeerfouten, die dan op omslachtige en tijdrovende wijze moeten worden opgespoord (in het gunstige geval dat ze bemerkt worden).
De gegeven algoritme bijvoorbeeld, lijkt als twee druppels water op een variant die niet meer korrekt is, waarin nl. de voorwaarde tussen *while* en *do* luidt: *onderwijzer* en *bovenwijzer* wijzen nog niet naar hetzelfde element. Deze variant lijkt op het eerste gezicht wel goed, en in de loop der tijd zal menige programmeur hierin getrapt zijn. Toch is het heel goed mogelijk dergelijke voetangels en klemmen te vermijden, door even de korrektheid van de algoritme na te gaan.
Allereerst zullen we nagaan dat het zoekproces, beschreven door de gegeven algoritme, ooit tot een eind komt. Dat is natuurlijk een eerste vereiste. Het gaat er daarbij om te laten zien dat vroeg of laat niet langer aan de voorwaarde tussen *while* en *do* voldaan zal zijn. Als we kunnen aantonen dat bij iedere uitvoering van het stuk algoritme na *do* het aantal elementen tussen *onderwijzer* en *bovenwijzer* kleiner wordt, dan volgt onmiddellijk dat dit aantal binnen een eindig aantal slagen tot nul zal zijn teruggebracht. Welnu, iedere keer dat het stuk na *do* wordt uitgevoerd, gaat *duim* wijzen naar een element tussen de elementen die worden aangewezen door *onderwijzer* en *bovenwijzer*. Dit is slechts mogelijk, omdat het proces zich er eerst van vergewist heeft dat daar nog minstens een element tussen lag. Nu zijn er twee mogelijkheden: *bovenwijzer* wordt verplaatst naar *duim*, of *onderwijzer* wordt daarheen verplaatst. In beide gevallen is het onmiddellijk duidelijk dat het aantal elementen tussen *onderwijzer* en *bovenwijzer* inderdaad is afgenomen.
Het hierboven gegeven terminatiebewijs is van een zeer gebruikelijk type. We weten nu dat het zoekproces eindigt, maar levert het ook het juiste antwoord? Kunnen we de tussen akkolades geplaatste bewering die achter de algoritme staat ook waarmaken?
Dat kan, en wel op overtuigende en eenvoudige wijze. Beschouw de volgende bewering:

Er geldt tenminste een van de volgende drie gevallen:

1. *z* is kleiner dan element nr. *1*;
2. *z* is groter dan element nr. *N*;
3. *z* is minstens zo groot als het element aangewezen door *onderwijzer* maar hoogstens zo groot als dat aangewezen door *bovenwijzer*.

Na de initialisatie van *onderwijzer* en *bovenwijzer*, juist op het ogenblik waar het zoekproces voor de eerste maal de voorwaarde tussen *while* en *do* zou gaan testen, is deze bewering juist. Immers, op dat ogenblik mogen we het derde geval gerust lezen als

3'. *z* is minstens zo groot als element nr. *1*,
maar hoogstens zo groot als element nr. *N*.

Tenminste een van de gevallen 1, 2 en 3' moet gelden, want als 1 noch 2 gelden, volgt onmiddellijk de geldigheid van 3'.

We zien dus dat de bewering geldt wanneer het proces voor de eerste maal bij *while* is aangeland. De volgende stap is om aan te tonen dat als de bewering daar geldt, dat hij dan opnieuw geldt wanneer het proces de eerstvolgende keer daar belandt.

Als de hele bewering juist was omdat we met geval 1 of 2 te doen hadden, dan blijft de bewering natuurlijk juist, want als 1 of 2 eenmaal gelden, dan blijven ze gelden. Het wordt iets ingewikkelder wanneer we met geval 3 te doen hebben. In dat geval weten we dus: *z* is minstens zo groot als het element aangewezen door *onderwijzer* en hoogstens zo groot als het element aangewezen door *bovenwijzer*. Wanneer het proces nu het stuk na *do* gaat uitvoeren komt het bij de test tussen *if* en *then* terecht. Dan zijn er twee mogelijkheden. De eerste is dat *z* minstens zo groot is als het element waarnaar *duim* wijst. Het proces voert dan de opdracht achter *then* uit: *onderwijzer* gaat nu naar datzelfde element wijzen en vanzelfsprekend geldt dan nog steeds geval 3. De andere mogelijkheid is dat *z* niet zo groot is (en dus hoogstens zo groot); door uitvoering van de opdracht na *else* wordt nu *bovenwijzer* naar *duim* gebracht en ook dan blijft geval 3 gelden.

De bewering geldt dus niet alleen de eerste maal, maar ook de tweede en iedere volgende keer dat het proces bij *while* belandt. Waar het nu om gaat is dat de bewering dus ook de laatste keer geldt dat de voorwaarde tussen *while* en *do* getest wordt - de keer waarop het proces ontdekt dat het ten einde is. We mogen daarom bij het aantonen van de korrektheid er gebruik

van maken dat aan het einde van het proces de bewering nog altijd juist is. Verder mogen we er gebruik van maken dat de voorwaarde tussen *while* en *do* de laatste keer niet langer gold, m.a.w., we weten dat er tussen *onderwijzer* en *bovenwijzer* geen element meer ligt.
Neem nu aan dat z in de geordende rij inderdaad voorkomt. (Anders kan het zoekproces natuurlijk z onmogelijk vinden.) Omdat de rij geordend is, zijn de gevallen 1 en 2 meteen uitgesloten. Geval 3 geldt dus. z kan nu niet voor *onderwijzer* of na *bovenwijzer* liggen (alweer vanwege de geordendheid), en ook niet er tussen (want er is niets tussen). Konklusie: de bewering tussen de akkolades is volkomen terecht: of *onderwijzer*, of *bovenwijzer* wijst na afloop naar het gezochte element, als dat tenminste in de rij voorkomt.
Het hier gegeven korrektheidsbewijs laat goed zien hoe dit soort redeneringen verloopt:
- enerzijds overtuigen we ons ervan dat het proces ooit ophoudt;
- anderzijds gaan we na dat het proces, als het ophoudt, het korrekte resultaat levert.

>01> Laat zien dat de inkorrekte variant op de algoritme die hierboven ter sprake kwam, inderdaad fout gaat. Geef een alfabetisch woordenlijstje, en een woord daaruit dat niet gevonden wordt met de foute algoritme, en ook een woord dat wel gevonden wordt.

>02> Laten we ergens halverwege tussen twee elementen nu eens wat losser definiëren, namelijk als: waar dan ook, als het maar tussen beide elementen is. Is de gegeven algoritme dan ook nog korrekt?

>>>> Teken een stroomschema van de gegeven halveringsalgoritme.

2. Een gewichtenprobleem

We bekijken het probleem van een aantal, zeg acht, munten die er identiek uitzien, maar waarvan er een zwaarder is dan de andere, die alle even zwaar zijn. Gegeven is een balans, waarmee bepaald kan worden of een groep munten (een of meer) zwaarder is dan een groep andere munten, of lichter, of even zwaar. Hoe kunnen we nu vaststellen welke de zwaardere munt is, om deze apart te leggen?

2.1. Eenvoudige algoritme: een voor een

Het schema in fig. 2 geeft een methode weer voor het oplossen van dit gewichtenprobleem.
Ervan uitgaande dat er een even aantal munten is (twee of meer), kunnen we deze algoritme zo formuleren:

(A2.1) leg de munten op een hoop;
while er ligt nog geen munt apart *do*
begin neem twee munten van de hoop;
weeg de ene tegen de andere;
if de balans slaat door *then* leg de zwaardere apart
end

We zien hier dat in plaats van *if* ... *then* ... *else* ... ook alleen *if* ... *then* ... kan voorkomen. Dit betekent: als aan de voorwaarde na *if* voldaan is, voer dan de opdracht na *then* uit (en sla deze anders over).

>>>> Voer de algoritme op papier uit voor het geval de vijfde van de acht munten de zwaardere is.

>03> Als we volgens deze algoritme de zwaardere van 8 munten bepalen, hoeveel wegingen zijn er dan
a. minimaal nodig?
b. maximaal nodig?
c. gemiddeld nodig?

Dezelfde vragen als het gaat om 2N munten.

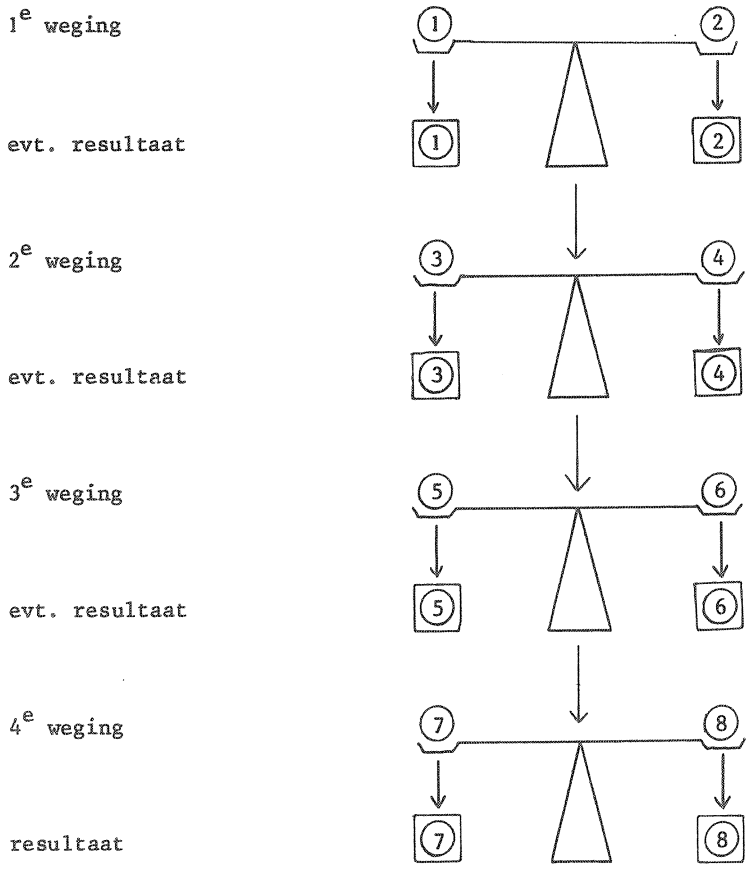


Fig. 2. Schema voor het bepalen van de zwaardere van 8 munten. Een pijl vanaf de linkerschaal van de balans wijst naar het alternatief dat gekozen wordt als de balans naar links doorslaat (idem voor rechts). Een pijl vanaf het midden van de balans wijst naar het alternatief dat gekozen moet worden als beide schalen even zwaar belast blijken.

De algoritme is natuurlijk alleen korrekt als het om een even aantal munten gaat.

>04> Toon de korrektheid aan.

Als we de algoritme zouden gebruiken voor een oneven aantal munten, lopen we het risiko dat de zwaardere munt als laatste alleen overblijft. Dan kan de opdracht neem twee munten van de hoop niet worden uitgevoerd.
We kunnen de algoritme als volgt aan deze moeilijkheid aanpassen:

(A2.2) leg de munten op een hoop;
while er ligt nog geen munt apart *do*
if er ligt nog maar een munt op de hoop *then*
leg die munt apart *else*
begin neem twee munten van de hoop;
weeg de ene tegen de andere;
if de balans slaat door *then* leg de zwaardere apart
end

>>>> Bewijs dat deze algoritme korrekt is voor elk (positief geheel) aantal munten.

2.2. Efficiëntere algoritme: tweedeling

De vorige algoritme is voor grotere aantallen munten niet erg efficiënt. Door gebruik te maken van de mogelijkheid op elk van de schalen van de balans meer dan een munt te wegen, kunnen we komen tot de algoritme uit fig. 3. In woorden:

> Weeg de ene helft van de munten tegen de andere helft,
> en ga met de zwaardere helft door met halveren totdat deze
> nog maar uit een munt bestaat.

Nauwkeuriger:

(A2.3) beschouw de hele groep munten als verdacht;
while de verdachte groep bevat meer dan een munt *do*
begin weeg de ene helft van de verdachte groep tegen de andere;
de zwaardere helft is de verdachte groep
end ;
leg de munt die nu in de verdachte groep zit apart

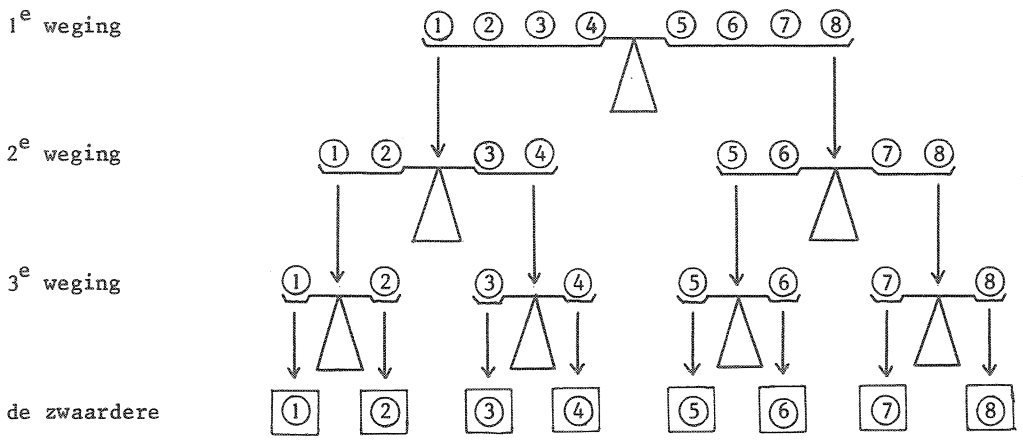


Fig. 3 Schema van een algoritme om van 8 munten de ene zwaardere te vinden.

Het is duidelijk dat de algoritme alleen kan werken als het aantal munten een macht van 2 is, b.v. 8, 16 of 256.

>>>> Voer de algoritme op papier uit voor het geval dat de vijfde van acht munten de zwaardere is.

>05> Werkt de algoritme ook goed met 1 munt (die dan natuurlijk de zwaardere moet zijn)?

>>>> Hoeveel wegingen zijn er nodig bij 8 munten? En bij 2^N munten?

2.3. Variabelen

Bij het uitvoeren van de tweedelingsalgoritme merken we dat met verdachte groep steeds een andere groep munten wordt aangeduid: in het begin alle munten, tenslotte nog maar een. Het hanteren van zo'n naam die nu eens dit, dan weer dat kan aanduiden, is van het grootste belang bij het beschrijven van algoritmen. We hebben dit principe ook al gebruikt bij het beschrijven van het zoekproces door halvering in een woordenboek. Daar verwezen *onderwijzer*, *bovenwijzer* en *duim* steeds naar andere bladzijden. De kracht van dit gebruik van zulke namen is dat ze enerzijds op steeds wisselende dingen betrekking kunnen hebben, maar anderzijds toch een vaste rol spelen: de verdachte groep blijft steeds de gezochte munt bevatten, *bovenwijzer* blijft naar een element verwijzen dat alfabetisch na het gezochte komt, enz.
Ook bij het schrijven van programma's voor een computer staat dit gebruik van namen centraal. Een programmeertaal legt daarbij zijn eigen beperkingen op maar in het algemeen geldt het volgende. De programmeur kan opdracht geven een resultaat onder een bepaalde naam op te bergen. Door in een latere opdracht weer die naam te gebruiken, kan hij bij het proces dat resultaat weer laten bekijken, het eventueel laten kopiëren, enz. Daarbij blijft het resultaat onder die naam bereikbaar, totdat het proces een nieuw resultaat onder die naam opbergt. (Dit gebeurt in het geheugen van de computer.)
We kunnen deze organisatie vergelijken met die van een kaartenbak. Als het resultaat van een bepaalde handeling onder de naam *bovenwijzer* moet worden opgeborgen, dan nemen we een lege kaart, schrijven bovenaan de naam *bovenwijzer* en op de kaart het bedoelde resultaat, b.v. het getal 24. Komt er daarna een opdracht die gebruik wil maken van de huidige waarde van

bovenwijzer, dan zoeken we in de kaartenbak de kaart op met de naam *bovenwijzer*, kopiëren wat er verder op die kaart staat, en werken daarmee verder. Pas op als we een nieuwe opdracht tegenkomen van het type: berg dit resultaat (b.v. het getal 16) op onder de naam *bovenwijzer*, pas dan zal het getal 24 uitgegumd worden en vervangen door 16.
Zo'n kaart nu, waar bovenaan een naam staat en verder een of andere grootheid, zoals een getal, wordt een variabele genoemd. Bovenaan de kaart staat de naam van de variabele, wat verder op de kaart staat heet de waarde van de variabele. In plaats van "de waarde van a is 17" zeggen we ook wel "a is 17", m.a.w., we gebruiken soms de naam van de variabele wanneer we zijn waarde willen aanduiden.
Bij het gebruiken van de waarde van een variabele moeten we er natuurlijk zeker van zijn dat deze variabele al een waarde gekregen heeft. We mogen er niet van uitgaan dat variabelen van nature de waarde nul hebben. In het algemeen kennen programmeertalen ook geen mogelijkheid voor een test als *if* a heeft nog geen waarde *then* ...

2.4. Een notatie

We zullen voor opdrachten die aan een variabele een (nieuwe) waarde toekennen op een speciale manier noteren, met behulp van het teken := (wordt-teken).
Toegepaste in de tweedelingsalgoritme:

(A2.4)
```
verdachte groep := hele groep munten;
while verdachte groep bevat meer dan een munt do
begin weeg de ene helft van verdachte groep tegen de andere;
      verdachte groep := zwaardere helft
end;
leg de munt die nu in de verdachte groep zit apart
```

We zien dat links van := de naam van de variabele staat, rechts zijn nieuwe waarde.

2.5. De korrektheid van de tweedelingsalgoritme

Als we inzien dat deze algoritme eigenlijk dezelfde is als de halverings-

methode bij het zoeken in een woordenboek, dan is het gemakkelijk ons van de korrektheid van deze tweedelingsalgoritme te overtuigen. De bewering waar het in dit geval om gaat is: de *verdachte groep* bevat steeds de zwaardere munt. Het is duidelijk dat deze bewering geldig is als het proces voor het eerst bij *while* belandt, en ook dat de geldigheid niet wordt aangetast door het uitvoeren van het stuk algoritme tussen *begin* en *end*.
Over de eindigheid hoeven we ons ook geen zorgen te maken: de *verdachte groep* wordt elke keer gehalveerd, en moet dus na een eindig aantal slagen uit precies een munt bestaan (omdat met een macht van 2 gestart werd).
Op het moment dat er nog maar een munt verdacht is, moet dit, zoals nu bewezen is, de zwaardere wel zijn.

>06> Een iets verschillend gewichtenprobleem: van een aantal identiek uitziende munten is er een die of zwaarder of lichter is dan de andere munten, die alle even zwaar zijn. Schrijf een tweedelingsalgoritme om met een balans vast te stellen welke de afwijkende munt is, en of deze zwaarder dan wel lichter is dan de andere.
Ga ervan uit dat het aantal munten een macht van 2 is, maar minstens 4. (Van 1 of 2 munten kan immers niet worden vastgesteld of het om een zwaardere of een lichtere munt gaat.)

2.6. Driedeling

Bij het proces dat door de tweedelingsalgoritme werd beschreven, kwam het nooit voor dat de twee groepen munten die tegen elkaar gewogen werden even zwaar bleken. Door van deze mogelijkheid wel gebruik te maken, kunnen we, voor grote aantallen munten, met nog minder wegingen toe.
De algoritme waarop het schema in fig. 4 berust, ziet er zo uit:

(A2.5) *verdachte groep* := hele groep munten;
while de *verdachte groep* bevat meer dan een munt *do*
begin *groep 1* := een derde deel van *verdachte groep*;
groep 2 := ander derde deel van *verdachte groep*;
groep 3 := laatste derde deel van *verdachte groep*;
weeg *groep 1* tegen *groep 2*;

if groep 1 en *groep 2* even zwaar *then verdachte groep := groep 3 else*
if groep 1 zwaarder dan *groep 2 then verdachte groep := groep 1 else*
verdachte groep := groep 2

end;

leg de munt die nu in de *verdachte groep* zit apart

>>>> Voer de algoritme uit met 27 munten, waarvan de achtste de zwaardere is.

Omdat het steeds mogelijk moet zijn de *verdachte groep* in drie gelijke groepen te verdelen, werkt deze algoritme alleen voor een aantal munten dat een macht van 3 is.

>07> Toon de korrektheid aan.

Omdat het bij het in drie groepen verdelen alleen van belang is dat twee van die groepen even groot zijn (zodat ze tegen elkaar gewogen kunnen worden), is het mogelijk de driedelingsmethode ook voor andere aantallen munten te gebruiken:

(A2.6)

verdachte groep := hele groep munten;
while verdachte groep bevat meer dan een munt *do*
begin *groep 1 :=* ongeveer derde deel van *verdachte groep;*
groep 2 := even groot deel van *verdachte groep;*
groep 3 := resterend deel van *verdachte groep;*
weeg *groep 1* tegen *groep 2;*
if groep 1 en *groep 2* even zwaar *then verdachte groep := groep 3 else*
if groep 1 zwaarder dan *groep 2 then verdachte groep := groep 1 else*
verdachte groep := groep 2

end;

leg de munt die nu in de *verdachte groep* zit apart

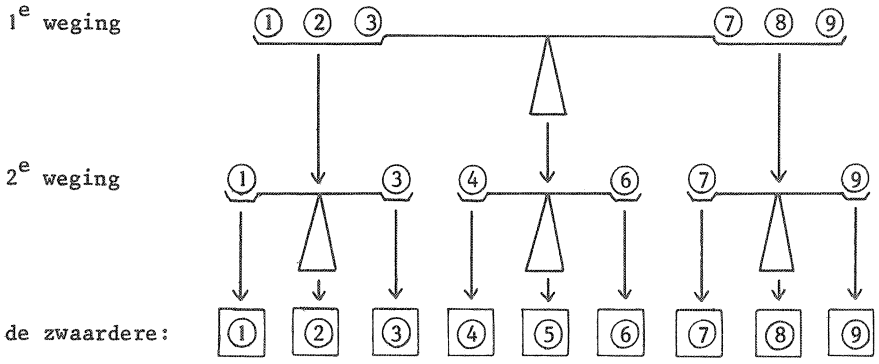


Fig. 4 Schema van een algoritme om van 9 munten de ene zwaardere te vinden.

Het is duidelijk dat het in ongeveer drieën delen van een groep van 7 munten neerkomt op: 2, 2, 3; 8 munten: 3, 3, 2; 9 munten: 3, 3, 3; enz.

>>>> Voer de algoritme uit met 8 munten, waarvan de vijfde de zwaardere is.

>08> Stel dat bij het in drieën delen van de *verdachte groep* alleen ervoor gezorgd wordt dat twee van de drie groepen precies even groot zijn (minstens 1), terwijl de grootte van de derde groep willekeurig is (dus 8 munten worden verdeeld als 4, 4, 0 of 3, 3, 2 of 2, 2, 4 of 1, 1, 6). Blijft de korrektheid van de algoritme dan onaangetast? Is er verder nog verschil?

2.7. Vergelijking van de drie algoritmen

We hebben drie algoritmen ter oplossing van het gewichtenprobleem gezien:

- een voor een: elke munt wordt tegen een andere gewogen, totdat de balans doorslaat.
- tweedeling : de *verdachte groep* wordt steeds gehalveerd, totdat er nog maar een munt in zit.
- driedeling : de *verdachte groep* wordt steeds in drieën gedeeld, totdat er nog maar een munt in zit.

De opzet van de een-voor-een-methode is de eenvoudigste. er wordt geen gebruik gemaakt van de mogelijkheid meer dan een munt tegelijk te wegen.

Het idee van de tweedeling gaat iets verder, en de algoritme is ook minder gemakkelijk te lezen. In deze algoritme wordt wel gebruik gemaakt van de mogelijkheid grotere groepen munten te wegen, maar niet van de mogelijkheid dat beide groepen even zwaar zijn.

De driedelingsalgoritme gaat nog een stap verder, en is iets moeilijker te doorzien. Hier worden alle mogelijkheden van de balans gebruikt. Als we het aantal wegingen vergelijken dat elk van de algoritmen gemiddeld nodig heeft, krijgen we het beeld uit fig. 5. (De tweedelingsmethode werkt alleen voor een twee-macht aan munten.)

	aantal munten										
		1	2	3	4	5	6	7	8	16	32	64
methode:	een voor een	0	1	1	$1\frac{1}{2}$	$1\frac{3}{5}$	2	$2\frac{1}{7}$	$2\frac{1}{2}$	$4\frac{1}{2}$	8	$16\frac{1}{2}$
	tweedeling	0	1	?	2	?	?	?	3	4	5	6
	driedeling	0	1	1	$1\frac{1}{2}$	$1\frac{4}{5}$	2	2	2	$2\frac{7}{8}$	$3\frac{5}{16}$	4

fig. 5 Tabel van het aantal wegingen dat gemiddeld nodig is.

We zien dat de een-voor-een-methode voor kleine aantallen munten efficiënter is dan de tweedelingsmethode en even efficiënt of efficiënter (5 munten !) dan de driedelingsmethode, maar dat deze laatste methode bij grotere aantallen munten minder wegingen vereist.
We zien hier de wet van behoud van ellende geïllustreerd: om met minder wegingen toe te kunnen, moeten we een lastigere algoritme schrijven.

3. Toepassing van variabelen

3.1. Een gemiddelde bepalen

We willen een algoritme opstellen voor het bepalen van het gemiddelde aantal ogen dat bij het gooien met een dobbelsteen bovenkomt. We laten daarvoor tien keer gooien, de ogen bij elkaar tellen, en tenslotte dit bedrag door 10 delen. We schrijven de algoritme voor iemand die de opdrachten gaat uitvoeren met een dobbelsteen, een potlood, kladpapier en een vel papier voor het resultaat.

(A3) *aantal gedane worpen := 0;*
totaal geworpen ogen := 0;
while aantal gedane worpen < 10 do
begin worp := aantal ogen bij nieuwe worp*;*
aantal gedane worpen := aantal gedane worpen + 1;
totaal geworpen ogen := totaal geworpen ogen + worp
end;
geef als resultaat de uitkomst van *totaal geworpen ogen / 10*

>09> Wat gebeurt er, in gewoon Nederlands gezegd, met de waarde van het *aantal gedane worpen* in de opdracht
aantal gedane worpen := aantal gedane worpen + 1

>>>> Voer de algoritme uit.

De drie variabelen van de algoritme hebben elk een eigen funktie, die door hun naam al enigszins wordt aangeduid:

aantal gedane worpen bevat, elke keer dat het proces bij *while* belandt, het getal dat aangeeft hoeveel worpen er dan gedaan en geadministreerd zijn;

totaal geworpen ogen geeft, steeds als we bij *while* komen,aan, hoeveel ogen er tot dan toe totaal gegooid zijn;

worp wordt steeds gebruikt om even het aantal ogen van de recentste worp te onthouden.

De hier genoteerde algoritme heeft de gedetailleerdheid van een computerprogramma; het overeenkomstige programma zou er ook bijna hetzelfde kunnen uitzien.

3.2. Output

Omdat bij het gebruik van een computer het geheugen van die computer als kladpapier dienst doet, en we niet zomaar kunnen zien wat er in dat geheugen staat, moeten in een computerprogramma ook opdrachten voorkomen, die ervoor zorgen dat de gewenste resultaten worden afgedrukt. Om ons daaraan al te wennen, zullen we in onze algoritmen gebruik maken van opdrachten als:

schrijf ("uitkomst")

met de betekenis: schrijf het tekstje : uitkomst (of druk het af)

of *schrijf (totaal geworpen ogen / 10)*

met de betekenis:

schrijf de uitkomst van de deling *totaal geworpen ogen / 10* op

Zoals in programmeertalen gebruikelijk plaatsen we datgene wat getypt moet worden tussen haakjes, en als het om een letterlijk tekstje gaat, zetten we dat nog tussen aanhalingstekens. Staan er geen aanhalingstekens, dan staat er een of andere formule, waarvan de uitkomst moet worden getypt (in het simpelste geval een getal of een variabele, waarvan de waarde moet worden getypt).

3.3. Input

Vaak willen we een algoritme niet voor een speciaal geval opstellen (b.v. voor precies 8 munten) maar willen we met alle mogelijkheden rekening houden (b.v. alle gehele positieve aantallen munten). Om dan toch te kunnen refereren naar de waarde van bepaalde grootheden, zullen we opdrachten gebruiken als

aantal munten := input

met de betekenis : geef aantal munten als waarde het volgende getal van de invoergetallen.

De serie invoergetallen kunnen we ons voorstellen als een strook waarop een serie getallen staat, die een voor een aan de beurt komen, te beginnen met het eerste.

Als de invoergetallen zijn: 3 1 -2

dan heeft *a := input; b := input; c := input*

hetzelfde effekt als *a := 3; b := 1; c := -2*

3.4. Vingeroefeningen met variabelen

Omdat variabelen zo belangrijk zijn bij het programmeren, is het goed er enige ervaring mee op te doen.

1. Wat is het resultaat van het stukje programma:

```
a := 13;
schrijf (a)
```

Na uitvoering van de eerste opdracht , is de situatie: 

d.w.z. er is een variabele *a*, die nu het getal 13 als waarde heeft.
Zoals afgesproken is hiervan niets echt zichtbaar. Pas na uitvoering van de tweede opdracht wordt het getal *13* als output geleverd.

2. Wat doet:

```
a := 13; b := 7;
schrijf (a + b)
```

Na uitvoering van de eerste opdracht is de situatie: 

Na uitvoering van de tweede: 

De output luidt dus: *20*

3. `a := 13; b := 7; schrijf (a - b)`
Output: *6*

4. `a := 13; schrijf (a + b)`
Als dit een geheel programma moet voorstellen, is het fout, want de variabele *b* heeft geen waarde gekregen, zodat de tweede opdracht niet kan worden uitgevoerd.

5. `a := 13; a := 14; schrijf (a)`
Na de eerste opdracht: 

Na de tweede: 

Output: *14*

6. `a := 5; b := a; schrijf (a)`
Nadat *a* de waarde 5 heeft gekregen, krijgt *b* diezelfde waarde, maar dit heeft niet tot gevolg dat de waarde van *a* verloren gaat. Output is dus: *5*

7. *a := 7; b := 3;*
a := a + b; b := a - b - b;
a := a + b; b := a - b - b;
schrijf (a); schrijf (b)

We zullen de uitvoering van dit programma stap voor stap volgen. Links de opdrachten, rechts de tussenresultaten op het kladpapier, na uitvoering van de opdracht links.
Denk eraan: in de opdracht staat links de naam van de variabele, rechts de formule die aangeeft welke waarde de variabele moet krijgen.

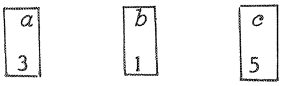
	a	*b*
a := 7	7	?
b := 3	7	3
a := a + b	10	3
b := a - b - b	10	4
a := a + b	14	4
b := a - b - b	14	6

Output: *14* *6*

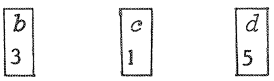
>10> Wat is de output van de volgende series opdrachten:
a. *a := 3; a := 5;* schrijf (*a*)
b. *a := 3; b := a;* schrijf (*a*)
c. *a := 3; b := a;* schrijf (*b*)

>11> We zien dat in voorbeeld 7 de eindwaarden van *a* en *b* het dubbele zijn van hun beginwaarden. Zou dit ook gebeurd zijn voor andere beginwaarden van *a* en *b*?

>12> Bedenk een serie opdrachten om uitgaande van de situatie:



te komen tot de situatie:



De waarde van *a* na afloop doet niet ter zake.
De algoritme moet natuurlijk ook goed gaan als er andere getallen in

a, b en c stonden: die moeten dan in b, c en d komen.

>13> Bedenk een serie opdrachten om de waarden van de variabelen a en b om te wisselen.
Als in de beginsituatie b.v. zou zijn: $\boxed{\begin{matrix}a\\3\end{matrix}}$ $\boxed{\begin{matrix}b\\5\end{matrix}}$ dan moet na de de uitvoering van de serie opdrachten de toestand zijn: $\boxed{\begin{matrix}a\\5\end{matrix}}$ $\boxed{\begin{matrix}b\\3\end{matrix}}$
Aanwijzing: gebruik een extra variabele.

>14> Bedenk ook een algoritme om de getallen in de variabele a en b om te wisselen zonder van een extra variabele gebruik te maken.

>15> Als er getallen staan in de variabelen a en b, wat is dan, in gewoon Nederlands gezegd, het effekt van uitvoering van de opdracht

if $a > b$ *then* *max* := a *else* *max* := b

($a > b$ betekent a is groter dan b)

>16> Schrijf een stukje algoritme dat ervoor zorgt dat de variabele *max* als waarde krijgt de grootste van de waarden van de variabelen a, b en c.

>17> Schrijf een stukje algoritme dat, ervan uitgaande dat er getallen staan in a en b, ervoor zorgt dat het grootste van die twee getallen in a komt, en het andere in b.

>18> Schrijf een stukje algoritme dat de 3 getallen in de variabelen a, b en c zodanig rangschikt, dat na afloop het kleinste in a, het grootste in c en het derde in b staat.

4. Recapitulatie

4.1. Algoritmen

- Een algoritme bestaat uit een (eindig) aantal opdrachten.
- Elke opdracht is van een type dat aan de uitvoerder bekend is.
- De opdrachten worden uitgevoerd in de volgorde waarin ze staan.
- De uitvoering van een algoritme moet eens ophouden.
- Een algoritme is in het algemeen niet alleen geschikt voor het oplossen van een bepaald probleem, maar voor een hele klasse van problemen (b.v. het vinden van de zwaardere munt uit elke muntenverzameling met een zwaardere, het zoeken van een willekeurig woord in een willekeurig woordenboek). Er moeten dan speciale aanwijzingen gegeven worden die bepalen welk probleem uit de klasse aangepakt moet worden (b.v. zoek het woord Zaïre op in de lijst Atlantis, Utopia, Zaïre). Deze gegevens worden invoer of input genoemd.
- Een algoritme levert resultaten af, b.v. via een opdracht als *schrijf (a / b)*. Deze resultaten worden uitvoer of output genoemd.
- Een algoritme bestaat vaak uit een gedeelte dat herhaald moet worden (cyclus of loop), voorafgegaan door enige opdrachten die ervoor zorgen dat de cyclus goed begint (initialisatie), en besloten met enkele opdrachten om de resultaten van de cyclus op te bergen of af te drukken. Binnen een cyclus komen vaak weer stukken voor met eenzelfde opbouw.

4.2. Korrektheid

Bij het bewijzen van de korrektheid van een algoritme gaat het vooral om de loops in de algoritme.

Eindigheid: wijs bij zo'n loop een grootheid aan die door een geheel getal wordt gekarakteriseerd (b.v. aantal munten in de *verdachte groep*), en die bij elke uitvoering van de loop met minstens 1 verlaagd wordt, terwijl de loop niet meer wordt uitgevoerd als deze grootheid een bepaalde waarde (b.v. 0) heeft bereikt.

Korrektheid resultaat: plaats met gelukkige hand enige beweringen op strategische plaatsen in de algoritme. Bewijs dat deze beweringen elke keer waar zullen zijn als het proces op de plaats van die bewering

is aangeland. Bewijs dit door volledige inductie, d.w.z.:

1. bewijs dat de bewering de eerste keer geldt;
2. bewijs dat als de bewering de eerste n keer heeft gegolden, hij ook de $n+1^{ste}$ keer zal gelden.

4.3. Notatie

- In onze notatie worden opeenvolgende opdrachten van elkaar gescheiden door een punt-komma.
- Opdrachten kunnen enkelvoudig of samengesteld zijn.
- Enkelvoudige opdrachten zijn b.v.:

  a := a + 1

  b := input

  schrijf (1 / a)

  weeg de ene helft van de *verdachte groep* tegen de andere helft
- Samengestelde opdrachten kunnen van de volgende soorten zijn:

  while test *do* opdracht

  if test *then* opdracht 1 *else* opdracht 2

  if test *then* opdracht

  begin opdracht 1; opdracht 2; ... opdracht n *end*

  De term opdracht kan hier zowel voor een enkelvoudige als voor een samengestelde opdracht staan. Daardoor zijn konstrukties mogelijk als:

```
if test 1 then
while test 2 do
begin opdracht 1;
      while test 3 do opdracht 2;
      if test 4 then opdracht 3
end
else
begin opdracht 4; opdracht 5; opdracht 6 end
```

5. Van een slordige lijst namen een alfabetische maken

In een notitieboek hebben we in de loop van een aantal jaren de namen van enige duizenden mensen opgeschreven. De namen staan niet alfabetisch, en het is dan ook wel voorgekomen dat een naam die niet snel vindbaar was, nogmaals is opgenomen. We willen nu een alfabetische lijst van deze namen aanleggen, waarin elke naam natuurlijk maar een keer mag voorkomen.
Er zijn verschillende manieren om dit probleem in de praktijk aan te pakken, b.v. alle namen op kaartjes overschrijven en deze alfabetisch sorteren of de hele lijst op ponskaarten zetten en een programma schrijven dat de alfabetische lijst laat afdrukken. We zullen hier een methode gebruiken die met de hand kan worden gevolgd, maar in zijn opzet toch aan die van een programma aansluit.
Het is hierbij van belang eerst het repertoire vast te stellen van de te gebruiken opdrachten. Laat dit repertoire er ongeveer zo uitzien:

- bekijk de eerste naam uit de oude lijst
- variabele := deze naam
- neem de volgende naam uit de oude lijst
- gum deze naam uit
- schrijf op deze plaats van de oude lijst de naam ...
- voeg ... aan de nieuwe lijst toe

De te gebruiken tests:

- is er nog een volgende naam in de oude lijst
- is de oude lijst niet helemaal leeg
- ... is alfabetisch eerder dan de naam ...

5.1. Eerste niveau

Omdat het probleem nogal groot is in verhouding tot het niveau van de elementaire opdrachten, zullen we de algoritme eerst beschrijven in grotere bouwstenen, die dan later kunnen worden uitgesplitst.

```
(A5.1)  while de oude lijst is niet helemaal leeg do
        begin laagste := de alfabetisch eerste uit de oude lijst;
              gum de laagste uit de oude lijst weg;
              if laagste is nog niet aan de nieuwe lijst toegevoegd then
              voeg laagste aan de nieuwe lijst toe
        end
```

We zien dat in de algoritme rekening wordt gehouden met het meermalen voorkomen van een naam.

5.2. Tweede niveau

De ingewikkeldste bouwsteen in deze eerste opzet van de algoritme is:

laagste := de alfabetisch eerste uit de oude lijst

Het is duidelijk dat het gedetailleerde stukje algoritme dat voor deze bouwsteen in de plaats moet komen, neerkomt op het doorzoeken van de gehele oude lijst, van elke naam die daarin nog voorkomt vaststellend of hij soms nog eerder in het alfabet thuis hoort dan enige naam tot dan toe.
Een andere bouwsteen die nog nader gedetailleerd moet worden is: gum de *laagste* uit de oude lijst weg. Hiervoor zijn drie strategieën denkbaar:

1. De oude lijst nog eens doorlopen totdat een naam wordt aangetroffen die identiek is aan *laagste*, en deze dan uitgummen.
2. Bij het zoeken van de laagste steeds bijhouden waar deze laagste wordt aangetroffen, zodat deze later zonder zoeken kan worden uitgegumd.
3. Bij het zoeken van de laagste deze uitgummen zodra hij wordt aangetroffen.

Omdat mogelijkheid 1 veel werk levert (voor de uitvoerder) en mogelijkheid 2 zich niet laat uitdrukken met de opdrachten in ons repertoire (er is geen opdracht om de namen door te nummeren, en ook geen om met die nummers te werken), zullen we proberen mogelijkheid 3 aan te grijpen: het uitgummen integreren in het zoekproces.
Onmiddellijk duikt nu een moeilijkheid op: op het moment dat we de alfabetisch eerste naam van de lijst denken te vinden (b.v. Alfred E. Neuman), weten we niet zeker of niet een alfabetisch eerdere naam zal volgen (b.v. Anthony A. Aardvark). Als we dus de naam Alfred E. Neuman direkt uitgummen, blijkt bij het tegenkomen van de naam Anthony A. Aardvark dat dit niet terecht was, en deze laatste naam juist zou moeten worden uitgegumd.
De oplossing: we gummen elke keer de nieuwe kandidaat-laagste (b.v. Anthony A. Aardvark) uit, en schrijven op die plaats de voorbarig uitgegumde oude kandidaat (Alfred E. Neuman) neer, enzovoort. Als Anthony A. Aardvark tenslotte de alfabetisch allereerste blijkt te zijn, hoeven we ons om het uitgummen niet meer te bekommeren, dat is al gedaan.

Laten we het stukje algoritme

laagste := de alfabetische eerste uit de oude lijst;
gum de *laagste* uit de oude lijst weg

nu eens uitwerken volgens de zojuist ontwikkelde gedachte.

(A5.2) *laagste* := eerste naam van de oude lijst;
gum de eerste naam van de oude lijst weg ;
while er is nog een volgende naam in de oude lijst *do*
begin *deze naam* := volgende naam op de oude lijst;
if *deze naam* komt alfabetisch eerder dan *laagste* *then*
begin gum *deze naam* van de oude lijst weg;
schrijf op deze plaats de *laagste*;
laagste := *deze naam*
end
end
{de laagste is nu ook uitgegumd}

>>>> Voer deze algoritme uit met een klein lijstje namen, om zo enig intuïtief inzicht in de werking ervan te krijgen.

5.3. Derde niveau

In de uitwerking op het tweede niveau komt nog de vrij ingewikkelde bouwsteen voor:

if *deze naam* komt alfabetisch eerder dan *laagste* *then* ...

Hier wordt van de uitvoerder gevergd dat hij niet alleen van twee letters kan bepalen welke eerder komt in het alfabet, maar ook kan nagaan welk van twee woorden alfabetisch (of beter : lexicografisch) het eerste komt. Het is duidelijk dat deze laatste taak omschreven kan worden met behulp van de test: komt deze letter alfabetisch eerder dan die?
Als we de woorden "kanapee" en "kanarie" vergelijken, zien we dat de eerste vier letters gelijk zijn; pas de vijfde letter geeft uitsluitsel: "kanapee" komt lexicografisch voor "kanarie". Een iets andere situatie doet zich voor bij vergelijking van "kanarie" en "kanariegeel". Hier zijn de eerste zeven letters gelijk, maar "kanarie" heeft geen achtste, en is daarom lexicografisch het eerste. In een algoritme om dit vast te stellen moeten dus de

letters van het ene woord van links naar rechts met die van het andere woord vergeleken worden, totdat blijkt:
- ofwel dat de letters ongelijk zijn,
- ofwel dat de letters van beide woorden op zijn,
- ofwel dat de letters van een van beide woorden op zijn.

In plaats van de test

if woord 1 alfabetisch voor *woord 2* *then* ...

kunnen we nu schrijven:

(A5.3)
```
letter 1 := eerste letter van woord 1;
letter 2 := eerste letter van woord 2;
while letter 1 = letter 2
      en zowel woord 1 als woord 2 heeft nog meer letters do
begin letter 1 := volgende letter van woord 1 ;
      letter 2 := volgende letter van woord 2
end;
if letter 1 alfabetisch voor letter 2 of
   woord 1 heeft geen letters meer maar woord 2 wel then ...
```

5.4. Uiteindelijke algoritme

Als we niveau 3 inbouwen in niveau 2, en het resultaat daarvan weer op niveau 1 inpassen, krijgen we dit resultaat:

```
(A5.4)  while de oude lijst is niet helemaal leeg do
        begin laagste := eerste naam van de oude lijst;
              gum de eerste naam van de oude lijst weg;
              while er is nog een volgende naam in de oude lijst do
              begin deze naam := volgende naam op de oude lijst;
                    letter 1 := eerste letter van deze naam;
                    letter 2 := eerste letter van laagste;
                    while letter 1 = letter 2 en
                          deze naam heeft nog meer letters en
                          laagste heeft nog meer letters do
                    begin letter 1 := volgende letter van deze naam;
                          letter 2 := volgende letter van laagste;
                    end;
                    if letter 1 alfabetisch voor letter 2 of
                       deze naam heeft geen letters meer,
                       maar laagste wel then
                    begin gum deze naam van de oude lijst weg;
                          schrijf op deze plaats de laagste;
                          laagste := deze naam
                    end
              end;
              {de laagste is nu ook uitgegumd}
              if de nieuwe lijst is nog leeg of anders
                 laagste verschilt van de laatste naam van de nieuwe lijst
              then voeg laagste aan de nieuwe lijst toe
        end
```

N.B. De laatste test van niveau 1 (A5.1) is in deze uiteindelijke versie nog vervangen door een kombinatie van twee eenvoudiger tests.

6. Verdeel en heers: programmeren door stapsgewijze verfijning

Het vorige hoofdstuk hebben we besloten met een forse algoritme, die de oplossing was voor een niet eenvoudig probleem: aan iemand die eigenlijk alleen het alfabet kent, en kan overschrijven en uitgummen, aan het verstand brengen hoe hij van een lange ongeordende lijst namen (met dubbele erbij) een alfabetisch geordende lijst moet maken.
Wie de algoritme voor het eerst ziet zonder de voorgeschiedenis te kennen zal er moeite mee hebben de beschreven gang van zaken te begrijpen. Nog veel moeilijker heeft iemand het, die zelf zo'n algoritme wil schrijven, maar niet de moeite neemt eerst de voorgaande stadia te doorlopen. Als hij er al in slaagt een algoritme in elkaar te zetten, zal het hem ook moeilijker vallen de korrektheid van zijn algoritme te laten zien.
Het is dan ook van groot belang bij het programmeren de methode van stapsgewijze verfijning te volgen: eerst schrijven we een korte algoritme, opgebouwd uit opdrachten die weliswaar een niveau eenvoudiger zijn dan de opdracht: los het probleem op, maar waarvan toch enige zo grof zijn, dat ze, net als het oorspronkelijke probleem, in fijnere elementen moeten worden gesplitst. Door deze verfijning ver genoeg door te zetten, belanden we tenslotte op het niveau van gedetailleerdheid dat door de programmeertaal vereist wordt. Zo is dan een hiërarchie van algoritmen ontstaan, waar we bovenaan de algemene strategie zien uitkomen, terwijl onderin, als onder een vergrootglas, de details van de uiteindelijke algoritme zichtbaar worden (fig. 5). De belangrijkste vorm van verfijning is het splitsen van een opdracht in een aantal kleinere, onderling gelijkluidende opdrachten. Het voordeel van die gelijkluidendheid is dat maar een opdracht hoeft te worden opgeschreven, in een vorm als: *while* test *do* opdracht.
Dit verdeel- en heersprincipe van stapsgewijze verfijning stelt ons in staat iets omvangrijks toch nog te overzien en aan te pakken.

6.1. Voordelen

De direkte voordelen van deze hiërarchische aanpak zijn tweeërlei:
1. Bij het programmeren hoeven we niet op alles tegelijk te letten.
2. Het aantonen van de korrektheid van de eind-algoritme kan gebeuren door een aantal korte bewijzen op de verschillende niveaus.

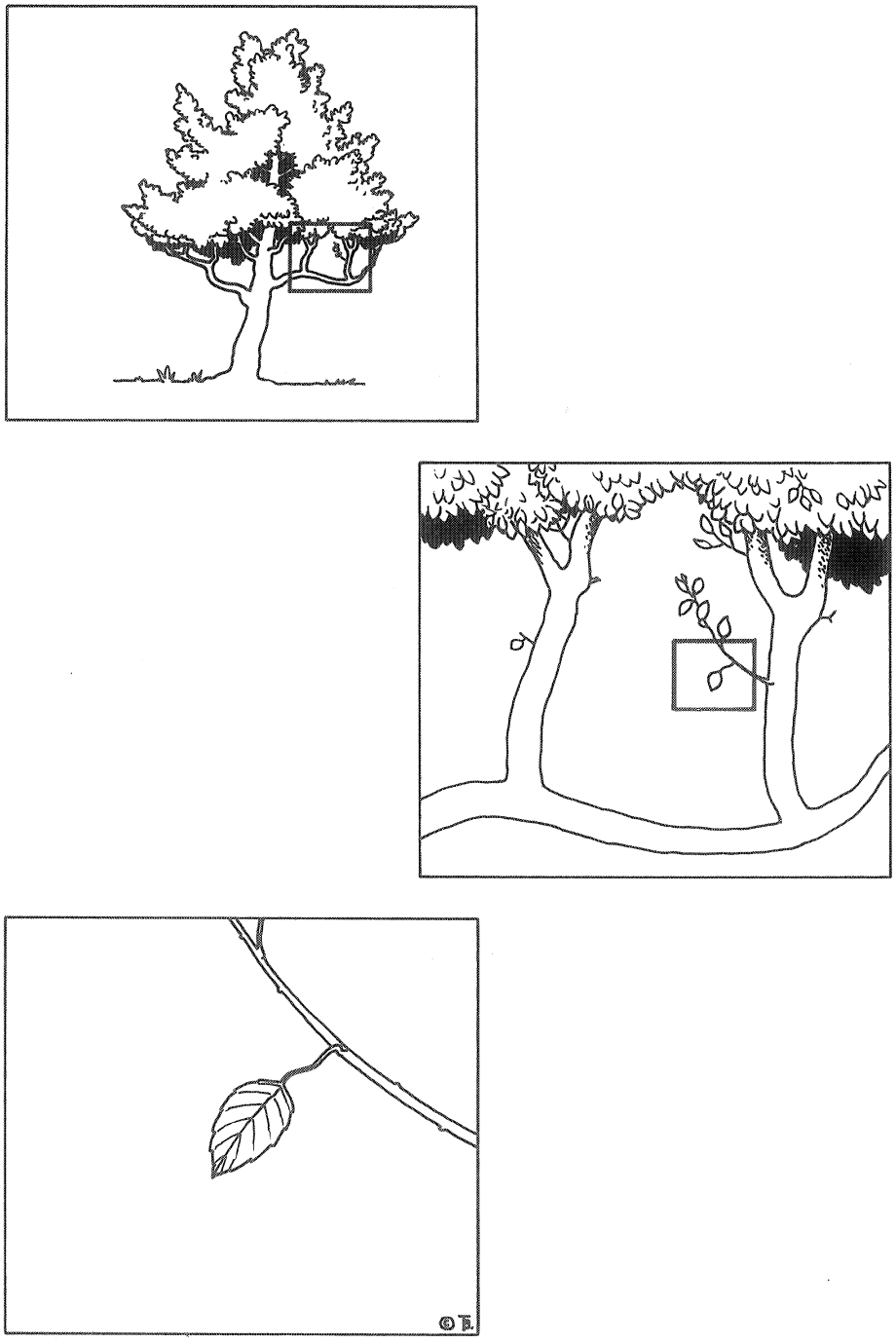


fig. 5 Stapsgewijze verfijning

Toegelicht aan de hand van de algoritme voor het alfabetiseren van een naamregister:

1. Op het eerste niveau zorgden we ervoor dat de namen een voor een, zonder dubbele toe te laten, op de nieuwe lijst kwamen. Op het tweede niveau bekommerden we ons alleen om het vinden van de lexicografisch eerste naam van de oude lijst, en het uitgummen ervan. Op het derde niveau tenslotte, behoefden we alleen nog aan te geven hoe bepaald kan worden of een naam lexicografisch eerder komt dan een andere naam.
2. De eindigheid van de algoritmen kan op de verschillende niveaus apart aangetoond worden; elk van deze drie bewijzen is eenvoudig. Wat de korrektheid van het resultaat betreft: op het eerste niveau tonen we aan dat als de namen in lexicografische volgorde van de oude lijst behaald worden, de nieuwe lijst korrekt en zonder dubbele namen worden opgesteld. Op het tweede niveau kan bewezen worden dat de volgorde waarin de namen afgeleverd worden lexicografisch is, als het derde niveau korrekt werkt, hetgeen dan tenslotte nog aangetoond moet worden.

Nog twee extra voordelen van het programmeren door stapsgewijze verfijning zijn de volgende:

1. Dokumentatie. In de praktijk moeten programma's vaak lang nadat ze geschreven zijn aangepast worden aan nieuwe eisen. De programmeur, of een ander, moet dan, soms jaren later, weer wijs worden uit de tekst voor het programma. Deze tekst op zich is daarvoor in het algemeen niet voldoende, deze moet voorzien zijn van enige vorm van uitleg voor de menselijke lezer. De hiërarchie van steeds gedetailleerdere algoritmen is de ideale dokumentatie voor dit doel, omdat hierin juist die begrippen worden gehanteerd die aan ons menselijk begrip aansluiten.
2. Modulariteit, d.w.z. opgebouwd zijn uit losse vervangbare elementen. Doordat we op ieder niveau bepaalde bouwstenen aanwijzen die verder worden gedetailleerd, zijn we in staat om stukken die aan nieuwe eisen moeten worden aangepast op een of ander niveau aan te wijzen, en alleen dat stuk opnieuw te verfijnen tot het vereiste niveau.
   Ook voor het overnemen van stukken uit een algoritme in een andere, is het nuttig als de oorspronkelijke algoritme modulair was opgebouwd.

6.2. Moeilijkheden

Natuurlijk moeten we bij het verfijnen van de ene opdracht van niveau 1 rekening houden met de uitwerking van andere opdrachten van niveau 1. We hebben dat gezien bij het verfijnen van de opdrachten *laagste* := de alfabetisch eerste uit de oude lijst, en gum de *laagste* uit de oude lijst weg. De uitwerkingen van deze twee handelingen bleken zeer met elkaar verweven te zijn. Het is dan ook verstandig eerst die opdrachten aan te pakken waarvan de uitwerking de grootste invloed lijkt te gaan hebben op de verfijning van andere opdrachten. Ook dan zal het niet altijd te vermijden zijn dat zo'n als eerste uitgewerkte opdracht weer moet worden aangepast onder invloed van de verfijning van andere opdrachten. Ook zal het wel voorkomen, dat we er met het uitwerken van de opdrachten van een bepaald niveau niet komen: soms blijkt bij de detaillering b.v. dat een of andere initialisatie nodig is die op het hogere niveau niet voorzien was.

6.3. Konklusie

Hoewel we dus op de methode van stapsgewijze verfijning niet kunnen blindvaren, en programmeren wel altijd een kwestie van passen en meten zal blijven, geeft deze methode van hiërarchisch programmeren een belangrijk houvast, dat vooral voor forsere algoritmen moeilijk gemist kan worden.

7. Arrays

Bij het uitwerken van het eerste niveau van de algoritme voor het alfabetiseren van een lijst namen (A5.1), moesten we een methode kiezen voor het vinden van de lexicografisch eerste naam, en tevens voor het uitgummen van die naam. We stonden toen voor het probleem hoe we de plaats van de naam die we moesten uitgummen konden onthouden. Daarbij hebben we de mogelijkheid van genummerde namen niet gekozen, omdat we ons een ander opdrachtenrepertoire hadden voorgenomen.
Het is goed die weg alsnog in te slaan, omdat de situatie van een rij genummerde gegevens in de praktijk van het programmeren een grote rol speelt.
In de meeste programmeertalen kan de programmeur naast gewone variabelen ook arrays gebruiken; een array is een rij variabelen, die ieder dezelfde naam, maar een verschillend nummer hebben. De variabelen in de array *a* worden aangeduid met b.v. *a [3]*, *a [4]*, *a [0]* enz. Dit lijkt weinig te verschillen van het gebruik van gewone variabelen als *a3*, *a4* of *a0*, maar een belangrijk voordeel van het gebruik van een array is, dat de afzonderlijke variabelen ook kunnen worden aangeduid door b.v. *a [teller]*. Als *teller* de waarde *3* heeft wordt hiermee de variabele *a [3]* aangegeven. Ook aanduidingen als *a [n - 4]* of zelfs *a [a [0]]* zijn mogelijk. (De uitdrukking tussen de vierkante haken wordt de index genoemd.)
Als de waarden van 100 gewone variabelen bij elkaar moeten worden opgeteld, moeten we dat zo opschrijven:

s := a1 + a2 + a3 + a4 + ... + a100

maar dan met de namen van de andere variabelen i.p.v. de stippeltjes.
Zijn de 100 variabelen opeenvolgend genummerde elementen van een array, dan gaat het ook zo:

(A7.1)
```
teller := 0; s := 0;
while  teller < 100 do
begin teller := teller + 1; s := s + a [teller] end
```

We zien dat *a [teller]* telkens een volgende variabele aanduidt, doordat de waarde van teller steeds met *1* verhoogd wordt.

7.1. *for*-opdracht

Het komt bij het programmeren vaak voor dat een stukje algoritme een zeker aantal keren (zeg 100) moet worden uitgevoerd. In A7.1, en eerder in A3, gingen we zo te werk:

```
teller := 0;
while teller < 100 do
begin teller := teller + 1;
      te herhalen stuk
end
```

In de meeste programmeertalen bestaat een eenvoudige notatie voor zulke gevallen. Wij zullen deze gebruiken:

```
for teller from 1 to 100 do   te herhalen stuk
```

Als we deze schrijfwijze toepassen voor A7.1:

```
s := 0;
for teller from 1 to 100 do s := s + a [teller]
```

Toegepast in A3:

```
totaal geworpen ogen := 0;
for aantal gedane worpen from 1 to 10 do
begin worp := aantal ogen bij nieuwe worp;
      totaal geworpen ogen := totaal geworpen ogen + worp
end;
schrijf (totaal geworpen ogen / 10)
```

In de formulering *from 1 to 100 do*
geeft de *1* de startwaarde voor de teller aan; de *100* geeft aan hoe groot de waarde van de teller maximaal mag zijn: is de waarde van de teller groter, dan wordt het te herhalen stuk niet meer uitgevoerd.

7.2. Vingeroefeningen met arrays en *for*-opdrachten

>19> Van twee arrays *a* en *b* moeten de elementen *1* t/m *100* onderling van waarde verwisseld worden, dus *a* [*1*] met *b* [*1*], *a* [*2*] met *b* [*2*], enz. Schrijf hiervoor een stukje algoritme.

>20> In een array *v* moeten de variabelen *v* [*-10*] t/m *v* [10] achtereenvolgens de getallen *-10, -9, ... , 10* als waarde krijgen.
Schrijf hiervoor een stukje algoritme.

>21> Op een zeker punt in een algoritme moet het kleinste van de getallen in de elementen *1* t/m *100* van de array *g* gezocht worden.
Schrijf een stukje algoritme dat een verfijning is van:
minimum := het kleinste getal in *g* [*1*] t/m *g* [*100*]

>>>> Verander de algoritme uit >19> zo dat ook gezorgd wordt voor
aantal := aantal keren dat dit *minimum* voorkwam

>22> In array *oud* staan *n oud* getallen (in de variabelen *oud* [*1*] t/m *oud* [*n oud*]. Om in een array *nieuw* een verzameling aan te leggen van alle verschillende getallen uit de array *oud*, kunnen we zo te werk gaan:

for oud wijzer from 1 to n oud do
begin if oud [*oud wijzer*] komt nog niet in *nieuw* voor *then*
voeg *oud* [*oud wijzer*] aan *nieuw* toe
end

Verfijn deze algoritme.

>23> In de tweede algoritme van deze cursus (A1, het zoeken van een element in een rij elementen die van klein naar groot geordend is) gebruikten we eigenlijk ook een array, b.v. in de uitdrukking het element waarnaar de *duim* verwijst. In array-notatie: *element* [*duim*].
Herschrijf deze algoritme in de notatie van arrays en variabelen.
Maak gebruik van de aanduiding *midden (a, b)* om een getal ergens halverwege tussen *a* en *b* aan te geven.
Laat de algoritme ook voor output zorgen: het nummer van het gezochte element, of anders het tekstje "niet aanwezig".

>>>> Van de rij getallen 8 4 9 3 5
vormen de getallen 8 6 7 6 ...
de cumulatieve gemiddelden: het derde cumulatieve gemiddelde (7) is

het gemiddelde van de eerste drie getallen, het vierde cumulatieve gemiddelde (6) van de eerste vier, enz.
Schrijf een stuk algoritme dat van n opeenvolgende input-getallen de n opeenvolgende cumulatieve gemiddelden in de array *cum* opbergt.

We zullen nu algoritme A5.1 uitwerken, ervan uitgaande dat de namen op de oorspronkelijke lijst genummerd staan als in een array, n.l. van *1* t/m *N*. We zullen de opdrachten van A5.1 niet in de volgorde uitwerken waarin ze staan, maar beginnen met de invloedrijkste (A), dan de invloedrijkste die nog over is (B), enz.:

```
F
C       while de oude lijst is niet helemaal leeg do
A       begin laagste := de alfabetisch eerste uit de oude lijst;
B             gum de laagste uit de oude lijst weg;
D             if laagste is nog niet aan de nieuwe lijst toegevoegd then
E             voeg laagste aan de nieuwe lijst toe
        end
```

```
A:      plaats := 1;
(A7.2)  for teller from 2 to aantal oude do
        if oud [teller] lexicografisch voor oud [plaats] then
        plaats := teller;
        laagste := oud [plaats]
```

De variabele *aantal oude* moet wel in het begin van de algoritme de waarde *N* krijgen (zie F). De variabele *plaats* wordt gebruikt om de index in de array *oud* te onthouden waar de laagste-tot-nu-toe werd gevonden. De algoritme lijkt verder veel op die voor het vinden van het kleinste van een rij getallen (zie >21>).

```
B:      oud [plaats] := oud [aantal oude];
        aantal oude := aantal oude - 1
```

Door op de positie waar *laagste* gevonden is de laatste naam van de oude lijst te kopiëren, is dit gat gedicht, zodat het proces in latere slagen op dezelfde manier kan verlopen (zonder voorzieningen

om zo'n gat over te slaan). Omdat nu de laatste plaats van de oude lijst niet meer nodig is, wordt *aantal oude* met *1* verlaagd.

C: *aantal oude* > *0*

D: *aantal nieuwe* = *0* of anders
laagste ≠ *nieuw* [*aantal nieuwe*]

De variabele *aantal nieuwe* moet op *0* geïnitialiseerd zijn (zie F), en moet bij elke toevoeging van een nieuwe naam met *1* verhoogd worden (zie E). Deze variabele is nodig opdat het proces weet tot waar de array *nieuw* gevuld is. Er is geen andere manier om daarachter te komen.

E: *aantal nieuwe := aantal nieuwe + 1;*
nieuw [*aantal nieuwe*] := *laagste*

F: *aantal oude := N; aantal nieuwe := 0;*

Deze initialisatie was op het eerste niveau niet voorzien.

Tot een geheel samengesteld:

(A7.3)
```
aantal oude := N; aantal nieuwe := 0;
while aantal oude > 0 do
begin plaats := 1;
      for teller from 2 to aantal oude do
      if oud [teller] lexicografisch voor oud [plaats] then
         plaats := teller;
      laagste := oud [plaats];
      oud [plaats] := oud [aantal oude];
      aantal oude := aantal oude - 1;
      if aantal nieuwe = 0 of anders
         laagste ≠ nieuw [aantal nieuwe] then
      begin aantal nieuwe := aantal nieuwe + 1;
            nieuw [aantal nieuwe] := laagste
      end
end
```

>24> In een array *d* staat in elk van de elementen *d[1]* t/m *d[1000]* een *0* of een *1*. Schrijf een algoritme die bepaalt hoeveel nullen en hoeveel enen er in de array voorkomen, en die deze aantallen als output geeft.

>25> Pas de algoritme van >22> aan voor het geval dat in de array niet nullen en enen staan, maar de getallen *1*, *2*, *3* en *4*.
Maak de verandering op het goede niveau, en werk het veranderde niveau dan verder uit, totdat de gehele algoritme is aangepast.

>26> Verander de algoritme van >25> nu voor het geval dat er niet *4* maar *n* mogelijke getallen in de array staan, n.l. de getallen *1*, *2*, ... *n*.

>27> De rij van Fibonacci ziet er zo uit:

0 1 1 2 3 5 8 13 ...

De rij begint met 0, 1, en verder is elke term gelijk aan de som van de twee voorafgaande.
Een eenvoudige algoritme om de termen 0 t/m 100 uit de rij van Fibonacci te laten uitvoeren is:

bouw de rij op;
voer de rij uit

Uitgewerkt:

```
f[0] := 0; f[1] := 1;

for i from 2 until 100 do f[i] := f[i - 1] + f[i - 2];

for i from 0 until 100 do schrijf (f[i])
```

Het eerste niveau van een andere aanpak ziet er zo uit:

initialiseer;
for *i* *from* *0* *to* *100* *do*
bereken de volgende term en voer deze uit

Werk deze aanpak uit. Wat is het voordeel boven de vorige methode?

7.3. Meer-dimensionale arrays

Als we de algoritme A7.2 willen verfijnen tot het niveau van het vergelijken van letters i.p.v. namen zoals we dat ook in A5.4 gedaan hebben, dan ligt het voor de hand ook de namen op te vatten als arrays, met in elke positie een letter. Daarbij is dan voor elk woord nog een variabele nodig waarin het aantal letters van het woord is aangegeven.
Omdat we met veel namen te maken hebben, zullen we ook veel arrays nodig hebben en, wat belangrijker is, we zouden deze arrays genummerd willen zien (zoals de namen in A7.2 genummerd waren): we hebben behoefte aan een array van arrays.
We kunnen daarvoor gebruik maken van een twee-dimensionale array, dat is een hoeveelheid variabelen met elk twee indices.
We kunnen ons zo'n twee-dimensionale array zo voorstellen:

	1	2	3	4	5	6	7	8	9	10
1	C	O	W	Z	N	O	F	S	K	I
2	A	A	R	D	V	A	R	K		
3	N	E	U	M	A	N				
4										
5										

Als de array *letter* heet, dan heet de variabele waarin de V staat: *letter* [*2, 5*], die met de Z: *letter* [*1, 4*]; in *letter* [*1, 2*] staat een O, in *letter* [*2, 1*] een A, enz. De derde letter van naam nr. *teller* duiden we dus aan met *letter* [*teller, 3*].

>28> Herschrijf A5.3 in deze notatie, en pas deze dan in A7.2 in, op de plaats van de test

if oud [*teller*] lexicografisch voor *oud* [*plaats*] *then* ...

Daarbij wordt de array *oud* twee-dimensionaal. Van elk van de namen in *oud* is tevens het aantal letters gegeven. Soortgelijke konsekwenties zijn er voor *laagste*.

>29> Veronderstel dat de opdracht *a := dobbel (6)* ervoor zorgt dat de variabele *a* als waarde krijgt een van de getallen 1, 2, 3, 4, 5, 6, en wel elke met een gelijke kans. Dat betekent dat na 6000 keer uitvoeren van deze opdracht ongeveer 1000 keer een 1 aan *a* wordt toegekend, 1000 een 2, enz., en dit zonder verband tussen twee opeenvolgende keren. Schrijf een algoritme die 10000 keer dobbelt, en als output levert:

1. hoe vaak het resultaat 1 was, hoeveel 2 enz.
2. hoe vaak na een 1 weer een 1 gedobbeld werd, hoe vaak na een 1 een 2 en zo van elk van de 36 combinaties.

>30> Pas de algoritme uit >29> aan, zodat hij ook werkt voor een dobbelsteen met n kanten: *a := dobbel (n)* levert met gelijke kansen de waarden 1, 2, ..., n. (n kan hierbij elk geheel getal groter dan 1 zijn.)

>31> Schrijf een stuk algoritme dat de variabelen *a* [*k*] t/m *a* [*g*] bekijkt (*k* en *g* zijn variabelen en de waarde van *k* is kleiner dan die van *g*), en dat ervoor zorgt dat de kleinste waarde die in dit stuk array wordt aangetroffen, aan *a* [*k*] wordt toegekend, terwijl de oude waarde van *a* [*k*] elders in het stuk array geplaatst wordt, zodat geen van de oorspronkelijk aanwezige waarden verloren gaat.

>32> Schrijf nu, uitgaande van >31>, een stuk algoritme dat de waarden van alle variabelen *a*[*k*] t/m *a*[*g*] van klein naar groot ordent, zodat na afloop voor elke i ($k \leq i < g$) geldt: *a* [i] $\geq$ *a* [i + 1], terwijl geen van de oorspronkelijke waarden verloren gaat.

Waar nodig zullen we ook arrays van dimensie 3 en hoger gebruiken.

>>>> In de variabelen *b* [*1*] t/m *b* [*1000*] staan getallen 0 en 1. Schrijf een algoritme die berekent hoeveel elk van de 16 verschillende opeenvolgingen van 4 nullen en/of enen voorkomt. Lever ook output.

8. Data structures

Bij het programmeren moeten we ook bedenken op welke manier we de informatie waarmee we werken zullen opbergen: in losse variabelen, in arrays, in combinaties van deze. De keuze van deze data structures hangt af van

- de aard van de informatie die we willen hanteren: een enkelvoudige grootheid zullen we niet in een array opbergen;
- de manier waarop we met die informatie willen omspringen: als we van een aantal gegevens nu eens dit dan weer dat nodig hebben, afhankelijk van de waarde van een bepaalde variabele, dan zullen we die gegevens in een array opbergen;
- de keus die de programmeertaal ons biedt: soms weinig meer dan:
  a. enkelvoudige variabelen, elk slechts geschikt voor waarden van een type: gehele getallen, reële getallen, waarheidswaarden, soms letters,
  b. een- of meer-dimensionale arrays, elk geschikt voor waarden van een type.

8.1. Integer variabelen

Een variabele die alleen een geheel getal als waarde kan hebben, wordt een integer variabele genoemd. Het voordeel van het gebruik van integer boven dat van real variabelen is, dat ze minder geheugenruimte vergen. In de meeste programma's komen heel wat grootheden voor die van nature alleen gehele getallen als waarde kunnen aannemen: tellers, de laagste en de hoogste index waar een array gevuld is, een variabele waarin geturfd wordt hoe vaak iets bepaalds is voorgekomen. Een regelmatig voorkomend gebruik van integer variabelen zien we in het volgende voorbeeld.
In een schaakprogramma moeten de stukken van elkaar onderscheiden worden, en daartoe krijgen de verschillende soorten elk een nummer:

pion	*1*	*toren*	*4*
paard	*2*	*dame*	*5*
loper	*3*	*koning*	*6*

In het programma kan dan een variabele *geslagen stuk* als waarde het kodegetal krijgen dat bij het zojuist geslagen stuk hoort.

Daarna kan in het programma staan:

if geslagen stuk = 1 then ... else
if geslagen stuk = 5 then ...

De variabele *geslagen stuk* is klaarblijkelijk van het type integer. Het is raadzaam om bij het werken met zulke getal-koderingen aan het begin van het programma een aantal opdrachten te plaatsen als:

pion := 1; paard := 2; loper := 3;
toren := 4; dame := 5; koning := 6

Verderop kan het programma daardoor veel leesbaarder worden, en daardoor kunnen fouten voorkomen worden:

if geslagen stuk = pion then ... else
if geslagen stuk = dame then ...

8.2. Real variabelen

Een variabele die een (reëel) getal als waarde kan hebben, wordt een real variabele genoemd. Een real variabele kost in het algemeen evenveel tot tweemaal zoveel geheugenruimte als een integer variabele. Typische reals zijn fysische grootheden als lengte, temperatuur, soortelijk gewicht of resultaten van mathematische bewerkingen als wortel en sinus.

8.3. Boolean variabelen

Zoals in een real of integer variabele de uitkomst van een berekening met getallen kan worden opgeborgen, zo kan in een boolean variabele het resultaat van een test worden bewaard. Omdat een test maar op twee manieren kan uitvallen, kan een boolean variabele alleen de waarde test klopt of test klopt niet krijgen. We schrijven deze waarden als *true* en *false*.
In sommige systemen beslaat een boolean variabele evenveel ruimte als een integer variabele, in andere passen er 20 à 30 boolean variabelen op de plaats van een integer variabele.
Voorbeeld van een (weliswaar onnuttig) gebruik van een boolean variabele:

if a > b then groter := true else groter := false;
if groter then begin w := a; a := b; b := w end

We zien dat in plaats van een test (zoals *a > b*) ook een boolean variabele (zoals *groter*) mag staan: deze boolean heeft als waarde immers de uitkomst

van een test. Hetzelfde verschijnsel zijn we bij andere variabelen ook tegengekomen: in plaats van *a := 3 + 4* kunnen we ook schrijven

```
b := 3 + 4;
a := b
```

Het voorbeeld met de boolean variabele groter kan korter zo geschreven worden:

```
groter := a > b;
if groter then begin w := a; a := b; b := w end
```

Met de opdracht *groter := a > b* is niets vreemds aan de hand: net als in andere toekenningsopdrachten moet de waarde berekend worden van wat rechts van := staat, en het resultaat van die berekening (hier *true* of *false*) moet toegekend worden aan de variabele die links van := genoemd wordt. Met deze waarde wordt verder gewerkt zodra weer de waarde van de variabele gevraagd wordt.

Een notatie: i.p.v. *groter := a > b; klog :=* niet *groter* schrijven we:
groter := a > b; klog := ¬ *groter*

>>>> Herschrijf het voorbeeld zonder boolean variabele.

>33> Wat wordt het resultaat afgeleverd door de volgende algoritmen?

a.
```
p := 3; q := 5;
if p > q then schrijf ("nee") else schrijf ("ja")
```

b.
```
p := 3; q := 5; gr := p > q;
if gr then schrijf ("nee") else schrijf ("ja")
```

c.
```
gr := false;
if gr then schrijf ("nee") else schrijf ("ja")
```

d.
```
p := 3; q := 3;
if p > q of p < q then schrijf ("nee") else schrijf ("ja")
```

e.
```
p := 3; q := 3; gr := p = q;
gr := ¬gr;
if gr then schrijf ("nee") else schrijf ("ja")
```

f.
```
p := 3; q := 3;
if false then schrijf ("nee");
if false of p = q then schrijf ("ja")
```

g.
```
gr := true; p := 3; q := 3;
if ¬gr then schrijf ("nee");
if gr en p = q then schrijf ("ja")
```

Om een nuttig gebruik van een boolean variabele te zien zullen we een algoritme opstellen voor het volgende probleem. In een array *a* [*1*] t/m *a* [*wijzer*] staan getallen. Het getal in de variabele *nieuw* moet toegevoegd worden als het niet in de array *a* voorkomt.
Eerste niveau:

```
initialisatie;
while er zijn nog meer getallen in a en
      nieuw nog niet aangetroffen do
vergelijk nieuw met het volgende getal in a;
if niet gevonden then voeg nieuw toe
```

Tweede niveau (zonder boolean variabele):

```
i := 0;
while i < wijzer en a [i + 1] ≠ nieuw do i := i + 1;
{nu geldt of i ≥ wijzer of a [i + 1] = nieuw,
 beide tegelijk kan niet}
if i ≥ wijzer then
begin wijzer := wijzer + 1;
      a [wijzer] := nieuw
end
```

Tweede niveau (met boolean variabele):

```
i := 0; gevonden := false;
while i < wijzer en ¬ gevonden do
begin i := i + 1;
      if a [i] = nieuw then gevonden := true
end;
{nu geldt of i ≥ wijzer of gevonden}
if ¬ gevonden then
begin wijzer := wijzer + 1;
      a [wijzer] := nieuw
end
```

De versie met boolean variabele is leesbaarder, de korrektheid valt gemakkelijker in te zien, en hij is iets efficiënter: nadat de *while*-loop is afgelopen hoeft alleen nog de waarde van *gevonden* te worden geïnspecteerd om te zien of *nieuw* moet worden toegevoegd. In de eerste versie moet de test $i \geq wijzer$ worden uitgevoerd en dat zal iets meer werk zijn.

Vaak is het gemakkelijk een grootheid die maar twee verschillende waarden kan krijgen in een boolean variabele op te bergen.
In het schaakprogramma zou een integer variabele *zet* gebruikt kunnen worden om bij te houden wie aan zet is: *zet* heeft de waarde *1* als wit aan zet is, en *2* als zwart aan zet is. Als we dan zorgen voor de initialisatie

wit := 1; zwart := 2

en vervolgens *zet := wit* dan kan verderop in het programma staan:

if zet = wit then ... else ...

en om aan te geven dat nu de ander aan zet is:

if zet = wit then zet := zwart else zet := wit

(Liever niet in plaats hiervan het ondoorzichtige : *zet := 3 - zet*).
Als we in plaats van de integer variabele *zet* een boolean variabele *wit aan zet* gebruiken, dan kan vooraan in het programma staan:

wit aan zet := true

en verderop:

if wit aan zet then ... else ...

en om de ander de beurt te geven

wit aan zet := ¬ wit aan zet

(d.w.z. de variabele *wit aan zet* krijgt als nieuwe waarde het tegengestelde van zijn oude waarde).

8.4. Arrays

In plaats van een aantal aparte variabelen gebruiken we arrays, als die variabelen als een serie gelijksoortige grootheden bewerkt moeten worden.
We hebben al nuttig gebruik van arrays gezien, zoals voor een alfabetisch geordende lijst waaraan steeds een nieuwe naam moet worden toegevoegd, nadat deze met alle aanwezige namen vergeleken is en niet aangetroffen.
Een regelmatig voorkomend nuttig gebruik van arrays is wat genoemd kan worden "het koderen van een aantal alternatieven". We zien dit in het volgende

voorbeeld, weer ontleend aan een denkbeeldig schaakprogramma. Nadat een stuk geslagen is, willen we de waarde van dat stuk aftrekken van de totale waarde van de stukken op het bord (een paard is 3 pionnen waard, een toren 5, enz.). Niet zo handig is dan:

```
if geslagen stuk = pion  then totaal := totaal - 1 else
if geslagen stuk = paard then totaal := totaal - 3 else
if geslagen stuk = loper then totaal := totaal - 3 else
if geslagen stuk = toren then totaal := totaal - 5 else
if geslagen stuk = dame  then totaal := totaal - 9
```

Beter is het aan het begin van het programma een array *waarde* te vullen:

	1	2	3	4	5
waarde	*1*	*3*	*3*	*5*	*9*

In plaats van de vijf regels hierboven kan een regel dienst doen:

```
totaal := totaal - waarde [geslagen stuk]
```

Het programma wordt hierdoor leesbaarder, waardoor fouten minder snel gemaakt worden, en, eenmaal gemaakt, eerder opvallen.

8.5. Overbodige arrays

Een heel belangrijke regel bij het kiezen van een data structure is: gebruik geen array als het met enige variabelen even goed lukt. Een goede reden hiervoor is dat het werken met array-elementen een computer meer tijd kost dan het werken met enkelvoudige variabelen. Als in plaats van een grotere array enkele losse variabelen gebruikt kunnen worden, is het voordeel duidelijk: minder geheugengebruik, want de geheugenruimte is beperkt. Een voorbeeld van het gebruik van enkele variabelen i.p.v. een grote array hebben we gezien in <25<, waar twee manieren ter sprake kwamen om termen van de rij van Fibonacci te berekenen. Doordat we met een rij getallen te doen hebben, ligt het misschien voor de hand een array te gebruiken. Maar dat is onnodig, want

1. de output van b.v. de tiende term hoeft niet te wachten op de berekening van b.v. de vijftiende; de output eist dus niet het opslaan in een array;
2. voor het berekenen van een term zijn alleen de twee voorafgaande termen nog nodig; de algoritme kan dus volstaan met twee variabelen om deze twee termen in te onthouden.

Als we in een programma met punten in een plat vlak willen werken, zullen we zo'n punt in het algemeen weergeven door zijn koördinaten in een rechthoekig assenstelsel. De koördinaten van punt A zouden we dan kunnen opbergen in *A [1]* en *A [2]*. Een afstand AB zouden we zo berekenen:

```
AB := wortel ((A [1] - B [1])^2 + (A [2] - B [2])^2)
```

en de omtrek van de driehoek ABC zo:

```
omtrek := wortel ((A [1] - B [1])^2 + (A [2] - B [2])^2) +
          wortel ((B [1] - C [1])^2 + (B [2] - C [2])^2) +
          wortel ((C [1] - A [1])^2 + (C [2] - A [2])^2)
```

Er is in dit voorbeeld geen enkele reden om de beide koördinaten van een punt in een array van twee elementen op te bergen.
Zo gaat het ook, en sneller:

```
omtrek := wortel ((xa - xb)^2 + (ya - yb)^2) +
          wortel ((xb - xc)^2 + (yb - yc)^2) +
          wortel ((xc - xa)^2 + (yc - ya)^2)
```

Als in het programma met een groter aantal punten gewerkt wordt, is het vaak wel handig de x-koördinaten in een array op te bergen, en de y-koördinaten in een andere. De omtrek van een veelhoek gevormd door de punten (x[1], y[1]), (x[2], y[2]), ... , (x[10], y[10]) kan dan zo berekend worden:

```
omtrek := wortel ((x [1] - x [10])^2 + (y [1] - y [10])^2);
for i from 1 to 9 do
omtrek := omtrek + wortel ((x [i + 1] - x [i])^2 + (y [i + 1] - y [i])^2
```

en de koördinaten xz en yz van het zwaartepunt van deze tien punten:

```
xz := 0; yz := 0;
for i from 1 to 10 do
begin xz := xz + x [i]; yz := yz + y [i] end;
xz := xz / 10; yz := yz / 10
```

Een vuistregel voor het vermijden van overbodig gebruik van arrays is:
als de index (of een van de indices) van de array overal in het programma een getal is, en nergens een variabele of een formule, dan is die array (of die index) overbodig.

Dat het nuttig kan zijn arrays van kleine afmetingen te gebruiken hebben we gezien bij het turven van de frekwenties van de verschillende opeenvolgingen van nullen en enen in een rij. Daar is het handig een array van 2 × 2

elementen te gebruiken: *f* [*0, 0*] voor de 0-0-opeenvolgingen, *f* [*0, 1*] voor de 0-1-opeenvolgingen, enz.

8.6. Combinaties van arrays en variabelen

In <28< hebben we gezien dat de lijst namen werd weergegeven door een stelsel van arrays en variabelen: een tweedimensionale array die de letters van de namen bevat, een variabele die het aantal namen aangeeft, en een eendimensionale array die van elke naam het aantal letters bevat.
Kijken we nog eens naar het ontstaan van deze complexe data structure, dan zien we dat hij eigenlijk zo in elkaar zit: een een-dimensionale array van namen, met daarbij een variabele die de lengte aangeeft; elke naam is op zich ook weer een een-dimensionale array van letters, met elk een variabele die het aantal letters aangeeft. Deze konstruktie van een een-dimensionale array met een variabele die aangeeft tot waar de array gevuld is, zullen we in de praktijk vaak tegenkomen, zowel in gevallen waar de gevuldheid van de array tijdens het proces verandert, als in gevallen waar de algoritme geschikt moet zijn voor arrays van welke lengte dan ook.

8.7. Het opbouwen van data structures

Het is duidelijk dat de keuze van de data structures niet onmiddellijk uit de probleemstelling voortvloeit, maar in belangrijke mate wordt gedikteerd door de algoritme. Andersom wordt de algoritme ook weer beïnvloed door de keuze van de data structures. Het is dan ook, vooral bij ingewikkelde algoritmen en data structures, nuttig de data structures te laten meegroeien met de algoritme. We hebben dit zien gebeuren bij het opstellen van de algoritme voor het alfabetiseren van een lijst namen in <28<. Op het niveau waar hele namen met elkaar vergeleken worden, bestond de lijst uit een een-dimensionale array van namen met een variabele die de gevuldheid aangaf; pas op het niveau waar de algoritme met letters ging werken, kwam de tweedimensionale array in het spel.

>34> Gegeven is als invoer de plaats van 10 kruisjes en 10 nulletjes op een 10 × 10 bord, en wel in deze vorm:

5,8 5,7 5,5 5,3 4,4 4,3 3,3 5,2 5,1 5,6
4,10 4,9 5,9 4,8 3,7 2,6 4,7 2,5 4,5 4,6

In dit geval is deze stand bedoeld:

10				0						
9				0	0					
8				0	×					
7			0	0	×					
6		0		0	×					
5		0		0	×					
4				×						
3			×	×	×					
2					×					
1					×					
	1	2	3	4	5	6	7	8	9	10

Schrijf een algoritme die de invoergetallen leest, en nagaat of er vertikale rijen van 5 of meer aaneensluitende kruisjes of van 5 of meer aaneensluitende nulletjes voorkomen. Geef de output in deze vorm:

kolom 4 6 nullen

>35> Dezelfde opgave als >34>, maar nu moet de algoritme geen aaneensluitende rijen opsporen, maar vaststellen in welke kolom of rij het grootste aantal symbolen (kruisjes of nulletjes) van dezelfde soort voorkomt.

De output zou voor het in >34> gegeven voorbeeld moeten luiden:

kolom 5 7 kruisen

In een ander geval zou de output kunnen zijn:

rij 6 4 nullen

Kies de data structures met zorg.

>36> Schrijf een algoritme die van een rij van 1000 invoergetallen (elk getal is of *1* of *0*) bepaalt hoevaak daarin rijen van 1, 2, 3 enz. gelijke getallen voorkomen.

Voorbeeld van zo'n rij (10 i.p.v. 1000 getallen):

0100111100

Output:

1	1	1
2	2	0
3	0	0
4	0	1

De linkerkolom geeft de lengte van de rijen aan, de middelste kolom het aantal keren dat een rij nullen van die lengte voorkomt, de rechter kolom het aantal keren dat een rij enen van die lengte voorkomt.

9. Procedures

Bij het detailleren van een algoritme zal het vaak voorkomen dat op verschillende plaatsen een (vrijwel) gelijke bouwsteen nodig is. Een eenvoudig voorbeeld daarvan hebben we in <18< gezien. De opdracht rangschik *a*, *b* en *c* in volgorde van oplopende waarden werd daar zo uitgewerkt:

(A9.1)
```
if a > b then wissel a en b;
if b > c then wissel b en c;
if a > b then wissel a en b
```

Omdat de opdracht Wissel *p* en *q* verfijnd kan worden tot

(A9.2) `w := p; p := q; q := w`

kan A9.1 zo uitgewerkt worden:

(A9.3)
```
if a > b then begin w := a; a := b; b := w end;
if b > c then begin w := b; b := c; c := w end;
if a > b then begin w := a; a := b; b := w end
```

Eigenlijk hebben we A9.3 niet nodig, omdat A9.1 en A9.2 al alle informatie geven.
Gelukkig beschikt vrijwel elke programmeertaal over de mogelijkheid de lange versie A9.3 te omzeilen, en te volstaan met A9.1, vergezeld van een explicatie in de trant van A9.2. De explicatie wordt een procedure genoemd. In dit geval is het een procedure voor het van klein naar groot rangschikken van de waarden van twee variabelen. Bij het aanroepen van de procedure (d.w.z. het gebruik ervan in A9.1) wordt aangegeven om welke variabelen het gaat, b.v. *b* en *c*. Deze *b* en *c* worden dan de parameters van die aanroep genoemd.
In de eerste aanpak van <34< hebben we de bouwsteen output, die we op niveau 2 drie maal gebruikten, op niveau 3 verfijnd, zodat de gedetailleerde versie in de resulterende algoritme drie maal precies gelijk voorkwam.
Hier zouden we een procedure *output* kunnen introduceren, met als tekst de verfijning van niveau 3.

```
if richting = rij then schrijf ("rij") else schrijf ("kolom");
schrijf (nr); schrijf (max);
if soort = kruis then schrijf ("kruisen") else schrijf ("nullen")
```

In de uiteindelijke tekst van de algoritme kunnen we dan, in plaats van deze drie regels, steeds schrijven: output.
Omdat het drie keer om dezelfde tekst gaat, kunnen we volstaan met een procedure zonder parameters.

9.1. Notatie

In de meeste programmeertalen moet ergens vooraan in het programma van elke te gebruiken procedure de naam en de tekst worden gegeven, en tevens moet daarbij worden aangegeven welke grootheden in de tekst parameters zijn, in de trant, van:

```
procedure wissel (p, q):
begin w := p; p := q; q := w end
```

Verder in de algoritme kan dan een aanroep staan als:

```
wissel (a, b)
```

met de betekenis:

```
w := a; a := b; b := w
```

Eventueel kan binnen de tekst van een procedure weer een procedure aangeroepen worden:

```
procedure sorteer 3 (x, y, z):
begin if x > y then wissel (x, y);
      if y > z then wissel (y, z);
      if x > y then wissel (x, y)
end
```

Hierdoor heeft de aanroep *sorteer 3 (f, g, h)* de betekenis:

```
if f > g then wissel (f, g);
if g > h then wissel (g, h);
if f > g then wissel (f, g)
```

en dit betekent dan weer:

```
if f > g then begin w := f; f := g; g := w end;
if g > h then begin w := g; g := h; h := w end;
if f > g then begin w := f; f := g; g := h end
```

In fig. 6 zien we hoe we ons de volgorde waarin de opdrachten van de verschillende procedures aan de beurt komen, kunnen voorstellen.

>37> Schrijf een procedure *sorteer 2* die in de procedure *sorteer 3* aangeroepen kan worden. Laat *wissel* een bouwsteen zijn van *sorteer 2*. Schrijf daarna stapsgewijs de aanroep *sorteer 3 (i, j, k)* uit, zoals dat hierboven voor de aanroep *sorteer 3 (f, g, h)* gedaan is.

Een opdracht als *schrijf (f[i])* kunnen we ook als aanroep van een procedure beschouwen.

9.2. Functie-procedures

Als vaak in een programma een opdracht voorkomt als

$$a := 2 \times n^3 + 3 \times n^2 - n + 2^n / n^4$$

of $$t[i] := 2 \times i^3 + 3 \times i^2 - i + 2^i / i^4$$

is het handig de gebruikte functie een naam te geven, b.v. f, en deze te gebruiken:

```
a := f (n)
```

en `t[i] := f (i)`

Apart moet dan de betekenis van f worden gegeven, b.v. zo:

```
functie f (y):
f := 2 × y^3 + 3 × y^2 - y + 2^y / y^4
```

Dergelijke functies zijn een soort procedures, die tevens een waarde afleveren. Voorbeelden van functie-procedures die we al gebruikt hebben, zijn

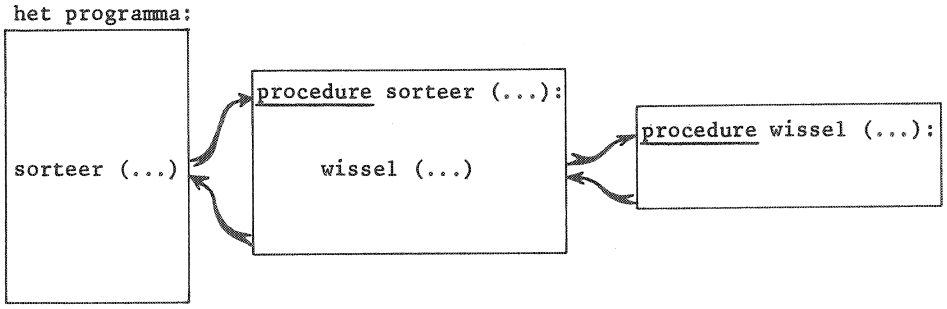


fig. 6 Geneste procedure-aanroepen

wortel (zie 8.5), dobbel (zie >29> en >30>), midden (zie >23>).
In de opdracht *a := input* is *input* te beschouwen als een aanroep van een functie-procedure zonder parameters, die als waarde het volgende invoer-getal levert.

9.3. Recursieve procedures

Soms is het nuttig in de tekst van een procedure de procedure zelf weer aan te roepen. We kunnen ons de gang van zaken dan voorstellen als in fig. 7. Een voorbeeld vinden we in de volgende aanpak van het zoeken in een woordenboek door halveren (vgl. >23>).

```
procedure zoek (onderwijzer, bovenwijzer):
begin if bovenwijzer - onderwijzer > 1 then
      begin duim := midden (onderwijzer, bovenwijzer);
            if z ≥ element [duim] then
            zoek (duim, bovenwijzer) else
            zoek (onderwijzer, duim)
      end else
      if element [onderwijzer] = z then schrijf (onderwijzer) else
      if element [bovenwijzer] = z then schrijf (bovenwijzer) else
      schrijf ("niet aanwezig")
end
```

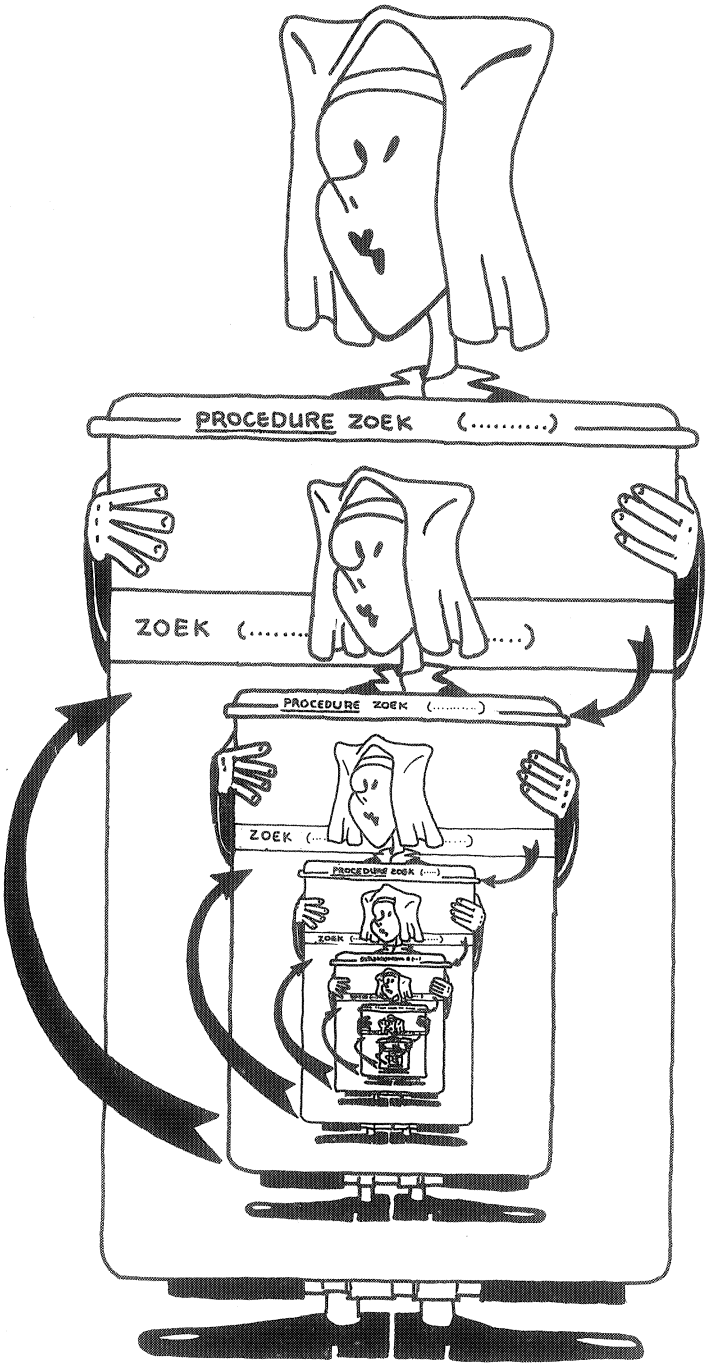


fig. 7 Recursieve procedure-aanroepen

De taak *zoek (onderwijzer, bovenwijzer)* kan beschreven worden als:

> indien er nog een element zit tussen de twee elementen aangewezen door *onderwijzer* en *bovenwijzer*, verklein het interval tussen *onderwijzer* en *bovenwijzer* dan totdat er geen meer tussen zit.

Deze taak wordt door de recursieve (zich zelf aanroepende) procedure *zoek* zo aangepakt:

> indien er nog een element tussen zit, plaats dan de *duim* halverwege en doe nu hetzelfde tussen *duim* en *bovenwijzer* of tussen *onderwijzer* en *duim*.

Omdat de tekst van een procedure als explicatie en niet als onmiddellijk uit te voeren opdracht moet worden beschouwd, is nog de opdracht *zoek (1, N)* nodig om het beschreven mechanisme aan het werk te zetten.

Zoals we aan <23< hebben gezien, is voor dit probleem helemaal geen recursieve procedure nodig. Later zullen we voorbeelden zien van problemen die zonder recursieve procedures zeer moeilijk, en met zeer gemakkelijk kunnen worden aangepakt.

Het is duidelijk dat er in de tekst van een recursieve procedure voor gezorgd moet worden dat niet onder alle omstandigheden weer een aanroep van de procedure zelf uitgevoerd wordt, omdat de uitvoering van een aanroep van zo'n procedure anders nooit zal eindigen.

>38> Schrijf een recursieve functie-procedure *faculteit*, zodat *faculteit (6)* als waarde krijgt 6 × 5 × 4 × 3 × 2 × 1 = 720, enz. Zorg ervoor dat faculteit (0) = 1
Schrijf ook een niet-recursieve procedure *fac* die hetzelfde doet.

9.4. Wanneer een procedure gebruiken?

Het is natuurlijk mogelijk voor elke bouwsteen op het eerste niveau van de hiërarchie van een algoritme een procedure te schrijven, en voor elk van de bouwstenen waaruit zo'n procedure wordt opgebouwd weer een, enz. Wanneer is het nu raadzaam een bouwsteen werkelijk de vorm van een procedure te geven?

1. Als een bouwsteen meermalen gebruikt wordt: dan spaart het schrijfwerk om elk van die keren een korte aanroep van een procedure te schrijven, die

dan een maal gedetailleerd word uitgeschreven.

2. Als te voorzien is dat deze bouwsteen in latere programma's ook gebruikt zal kunnen worden: een procedure kan dan gemakkelijk worden overgenomen, omdat deze door middel van de parameters eenvoudig aan de nieuwe omstandigheden kan worden aangepast. Het is dan ook verstandig in dit geval ruim gebruik te maken van de mogelijkheid grootheden als parameter aan te wijzen. Zo zou de procedure *zoek* uit 9.3 beter nog een extra parameter kunnen hebben, nl. het gezochte element. Weliswaar is dat voor het zoeken van een bepaald element z niet nodig, maar in een nieuw programma moeten wellicht een aantal verschillende elementen x, y en z worden opgezocht. Als de procedure *zoek* dan de extra parameter heeft, kan geschreven worden:

   zoek (1, k, x); zoek (1, k, y); zoek (1, k, z)

Vuistregel: maak alle grootheden die een kontakt vormen met de wereld buiten de procedure tot parameters.

3. Als een bepaalde bouwsteen een aktie beschrijft die voor het begrip een duidelijk afgeronde, eenvoudige betekenis heeft, maar waarvan de gedetailleerde versie er toch vrij ingewikkeld gaat uitzien: in het eigenlijke programma kan dan een korte begrijpelijke term gebruikt worden, b.v. *sorteer (A, n)* in plaats van het ondoorzichtige stuk programma, dat nu de tekst van de procedure *sorteer* vormt.

Uitwerking van de vraagstukken.

<01< Woordenlijst: Atlantis, Utopia, Zaire
Atlantis en Utopia worden niet, maar Zaire wordt wel gevonden.

<02< Bij de lossere definitie blijft de algoritme korrekt. Het korrektheids-bewijs blijft opgaan, want dat maakt alleen gebruik van het feit dat de duim tussen beide wijsvingers belandt.

<03< 8 munten:
a. minimaal 1 weging; b. maximaal 4 wegingen; c. omdat 1, 2, 3 en 4 wegingen elk even waarschijnlijk zijn, is het gemiddelde
(1 + 2 + 3 + 4) / 4 = 2½ weging.
2N munten:
a. 1 weging; b. N wegingen; c. (N + 1)/2 wegingen.

<04< We zullen de juistheid van de beweringen aantonen die tussen akkolades in de tekst van de algoritme zijn gevoegd (h is het aantal munten op de hoop):

leg de munten op een hoop;
{h even, h ≥ 2, zwaardere op de hoop, geen munt apart}
while {(h even, h ≥ 2, zwaardere op de hoop, geen munt apart) of zwaardere ligt apart} er ligt nog geen munt apart *do*
begin {h even, h ≥ 2, zwaardere op de hoop, geen munt apart}
neem twee munten van de hoop;
{(h even, h ≥ 0, zwaardere niet op de hoop, geen munt apart) of (h even, h ≥ 2 en zwaardere op de hoop, geen munt apart)}
weeg de ene helft tegen de andere;
if de balans slaat door *then*
leg de zwaardere apart {zwaardere munt ligt apart}
{else h even, h ≥ 2, zwaardere op de hoop, geen munt apart}
end
{de zwaardere ligt apart}

We zullen met volledige induktie aantonen dat bewering 3 elke keer geldt als het proces deze bewering passeert.

a. de eerste keer geldt bewering 3 omdat dit uit 1 en 2 volgt.

b. geldt 3 eenmaal, dan is gemakkelijk in te zien dat 4 vervolgens ook geldt. Als de balans dan doorslaat zal bewering 5 gelden, zodat daarna het tweede deel van bewering 2 geldt, en bewering 3 nooit meer gepasseerd wordt. Slaat de balans niet door, dan geldt bewering 6, vervolgens het eerste deel van bewering 2, en dan bewering 3 weer.

Konklusie: bewering 3 geldt inderdaad elke keer dat hij gepasseerd wordt.

Volgens de bovengegeven redenering zal elk van die keren ook 4, en dan 5 of 6, en vervolgens 2 weer gelden.

Als we nu het feit dat 2 telkens geldt, kombineren met de wetenschap dat 7 alleen bereikt wordt als niet aan de voorwaarde na *while* voldaan is, dan zien we dat 7 bereikt wordt als niet het eerste deel van 2 geldt, maar het tweede deel: de zwaardere ligt apart. Hetgeen te bewijzen was.

De eindigheid: Nadat de cyclus een zeker aantal keren is uitgevoerd is h = 2 (als het proces niet al eerder beëindigd is). In dat geval zal de balans zeker doorslaan, zal de zwaardere zeker apart gelegd worden, en zal de cyclus dus niet nog eens worden uitgevoerd.

<05< Met 1 munt werkt de algoritme ook goed, want aan de voorwaarde tussen *while* en *do* is de eerste keer al niet voldaan, zodat het stuk tussen *begin* en *end* niet een keer wordt uitgevoerd, en onmiddellijk de laatste opdracht aan de beurt komt.

<06<

We zullen de afwijkende munt apart leggen, en het antwoord op de vraag of deze munt zwaarder is dan de andere munten of lichter zullen we afleveren in de variabele *signalement*.

```
weeg de ene helft van de munten tegen de andere helft;
zware groep := de zwaardere helft;
lichte groep := de lichtere helft;
weeg de ene helft van de zware groep tegen de andere helft;
if een van de helften is zwaarder then
begin signalement := zwaarder; zware groep := de zwaardere helft;
      while de zware groep bevat meer dan een munt do
      begin weeg de ene helft van de zware groep tegen de andere helft;
            zware groep := de zwaardere helft
      end;
      leg de munt die nu in de zware groep zit apart
end else
begin signalement := lichter;
      while de lichte groep bevat meer dan een munt do
      begin weeg de ene helft van de lichte groep tegen de andere helft;
            lichte groep := de lichtere helft
      end;
      leg de munt die nu in de lichte groep zit apart
end
```

We zien dat de beide *begin-end* stukken bijna gelijk zijn. Hiervan gebruik makend kunnen we tot deze formulering komen:

```
weeg de ene helft van de munten tegen de andere helft;
zware groep := de zwaardere helft;
lichte groep := de lichtere helft;
weeg de ene helft van de zware groep tegen de andere helft;
if een van de helften zwaarder then
begin verdachte groep := de zwaardere helft;
      signalement := zwaarder
end else
begin verdachte groep := lichte groep;
      signalement := lichter
end;
```

while de *verdachte groep* bevat meer dan een munt *do*
begin weeg de ene helft van de *verdachte groep* tegen de andere helft;
verdachte groep := de helft die aan het *signalement* beantwoordt
end;
leg de munt die nu in de *verdachte groep* zit apart

<07< Als het proces bij *while* belandt geldt altijd:
de zwaardere munt zit in de verdachte groep.
Bewijs: 1. de eerste keer is dit gegeven;
2. als het eenmaal geldt, dan komt de munt in *groep 1*, *groep 2* of *groep 3*; *groep 1* en *groep 2* zijn even groot, als in een van beide de zwaardere munt zit, dan slaat de balans naar die kant door, en wordt dat de *verdachte groep*, zodat de bewering weer geldt. Slaat de balans niet door, dan moet de zwaardere munt in *groep 3* zitten, dan wordt die de *verdachte groep*, en ook dan geldt de bewering weer.
Als de stelling elke keer geldt, dan ook de laatste keer, als er nog maar een munt in de *verdachte groep* zit. Het proces belandt dan bij de laatste opdracht, en legt deze zwaardere munt apart. Q.E.D.
Eindigheid: de *verdachte groep* wordt altijd in drie groepen gesplitst die elk kleiner zijn dan hijzelf. Een van deze drie wordt telkens de nieuwe *verdachte groep*, zodat de *verdachte groep* bij elke slag minstens een kleiner wordt. Het aantal 1 moet dus in een eindig aantal slagen bereikt worden. (Nul wordt nooit bereikt, omdat in een groep van nul munten nooit de zwaardere kan voorkomen.) Zodra het aantal 1 bereikt is houdt het proces op.

<08< De korrektheid blijft onaangetast, want het bewijs uit <07< geldt voor A2.5, voor A2.6 en voor de versie van A2.6 waarin niet zo goed mogelijk in drieën wordt gedeeld.

<09< De waarde van deze variabele wordt met 1 verhoogd.

<10< a. 5; b. 3; c. 3

<11< Ja, voor elke startwaarde van *a* en *b*.
Als we deze beginwaarden A en B noemen, verloopt het proces zo:

	a	*b*
beginsituatie	A	B
a := a + b;	A + B	B
b := a - b - b;	A + B	A - B
a := a + b;	A + A	A - B
b := a - b - b;	A + A	B + B

<12< *d := c; c := b; b := a*

<13< *w := a; a := b; b := w*

<14< *a := a + b; b = a - b; a := a - b*
Deze methode van omwisselen kost minder variabelen, maar meer rekentijd (behoud van ellende).

<15< De grootste van beide waarden van *a* en *b* wordt aan de variabele *max* toegekend.

<16< Vier manieren:

1. *if a > b then max := a else max := b;*
   if c > max then max := c
2. *max := a;*
   if b > a then max := b;
   if c > max then max := c
3. *max := a;*
   if b > max then max := b;
   if c > max then max := c
4. *if a > b then*
   begin if a > c then max := a else max := c end else
   if b > c then max := b else max := c

<17< Eerste opzet:

if *a* < *b* *then* omwisselen

Uitgewerkt volgens <13<:

if *a* < *b* *then*

begin *w* := *a*; *a* := *b*; *b* := *w* *end*

<18< Naast enkele wat ingewikkeldere algoritmen kunnen we ook de volgende bedenken:

zet *a* en *b* in goede volgorde

zet *b* en *c* in goede volgorde

{*c* heeft nu de grootste waarde}

zet *a* en *b* in goede volgorde

Uitgewerkt:

if *a* > *b* *then* wissel *a* en *b*;

if *b* > *c* *then* wissel *b* en *c*;

if *a* > *b* *then* wissel *a* en *b*

Tenslotte:

if *a* > *b* *then* *begin* *w* := *a*; *a* := *b*; *b* := *w* *end*;

if *b* > *c* *then* *begin* *w* := *b*; *b* := *c*; *c* := *w* *end*;

if *a* > *b* *then* *begin* *w* := *a*; *a* := *b*; *b* := *w* *end*

<19<
```
for i from 1 to 100 do
begin w := a [i]; a [i] := b [i]; b [i] := w end
```

<20<
```
for i from -10 to 10 do v [i] := i
```

<21<
```
waar := 1;
for i from 2 to 100 do
if g [i] < g [waar] then waar := i;
minimum := g [waar]
```

of:

```
waar := 1; minimum := g [waar];
for i from 2 to 100 do
if g [i] < minimum then
begin waar := i; minimum := g [waar] end
```

<22< De verfijning van voeg *oud [oud wijzer]* aan *nieuw* toe kan er zo uitzien:

```
n nieuw := n nieuw + 1;
nieuw [n nieuw] := oud [oud wijzer]
```

Dit maakt deze initialisatie nodig:

```
n nieuw := 0
```

De test:

if oud [oud wijzer] komt nog niet in *nieuw* voor *then* ...

kan zo verfijnd worden:

```
signaal := 0; i := 0;
while i < n nieuw en signaal = 0 do
begin i := i + 1;
      if nieuw [i] = oud [oud wijzer] then signaal := 1
end;
if signaal = 0 then ...
```

Tot een geheel samensteld:

```
n nieuw := 0;
for oud wijzer from 1 to n oud do
begin signaal := 0; i := 0;
      while i < n nieuw en signaal = 0 do
      begin i := i + 1;
            if nieuw [i] = oud [oud wijzer] then signaal := 1
      end;
      if signaal = 0 then
      begin n nieuw := n nieuw + 1;
            nieuw [n nieuw] := oud [oud wijzer]
      end
end
```

```
<23<  onderwijzer := 1; bovenwijzer := N;
      while bovenwijzer - onderwijzer > 1 do
      begin duim := midden (onderwijzer, bovenwijzer);
            if z ≥ element [duim] then onderwijzer := duim
                                  else bovenwijzer := duim
      end;
      if element [onderwijzer] = z then schrijf (onderwijzer) else
      if element [bovenwijzer] = z then schrijf (bovenwijzer) else
      schrijf ("niet aanwezig")
```

```
<24<  nullen := 0; enen := 0;
      for i from 1 to 1000 do
      if d [i] = 0 then nullen := nullen + 1
                   else enen := enen + 1;
      schrijf (nullen); schrijf (enen)
```

of:

```
nullen := 0;
for i from 1 to 1000 do
if d [i] = 0 then nullen := nullen + 1;
schrijf (nullen); schrijf (1000 - nullen)
```

<25< *een := 0; twee := 0; drie := 0; vier := 0;*
for i from 1 to 1000 do
if d [i] = 1 then een := een + 1 else
if d [i] = 2 then twee := twee + 1 else
if d [i] = 3 then drie := drie + 1 else vier := vier + 1;
schrijf (een); schrijf (twee); schrijf (drie); schrijf (vier)

<26< *for t from 1 to n do f [t] := 0;*
for i from 1 to 1000 do f [d [i]] := f [d [i]] + 1;
for t from 1 to n do begin schrijf (t); schrijf (f [t]) end

<27< Deze algoritme moet gedetailleerd worden:

B initialiseer;
for i from 0 to 100 do

A bereken de volgende term en voer deze uit

A: voor de berekening van een term zijn de twee vorige termen nodig; daarvoor zullen we de variabelen *vorige* en *eervorige* gebruiken:

nieuwe := vorige + eervorige; schrijf (nieuwe);
eervorige := vorige; vorige := nieuwe

B: *eervorige := 0; schrijf (eervorige); vorige := 1; schrijf (vorige)*

Hiermee zijn al twee termen uitgevoerd, zodat de regel tussen B en A gaat luiden:
for i from 2 to 100 do

Resultaat:

```
eervorige := 0; schrijf (eervorige); vorige := 1; schrijf (eervorige);
for i from 2 to 100 do
begin nieuwe := vorige + eervorige; schrijf (nieuwe);
      eervorige := vorige; vorige := nieuwe
end
```

Om de regel beginnend met *for* uit de opgave onveranderd te laten, kan deze konstruktie gebruikt worden:

```
for i from 0 to 100 do
if i = 0 then begin eervorige := 0; schrijf (eervorige) end else
if i = 1 then begin vorige := 1; schrijf (vorige) end else
begin nieuwe := vorige + eervorige; schrijf (nieuwe);
      eervorige := vorige; vorige := nieuwe
end
```

of deze konstruktie:

```
eervorige := -1; vorige := 1;
for i from 0 to 100 do
begin nieuwe := eervorige + vorige;
      schrijf (nieuwe);
      eervorige := vorige; vorige := nieuwe
end
```

<28< A 5.3 in de notatie met arrays:

```
letter 1 := woord 1 [1]; letter 2 := woord 2 [1];
i := 1;
while letter 1 = letter 2 en
      i < aantal letters woord 1 en
      i < aantal letters woord 2 do
begin i := i + 1;
      letter 1 := woord 1 [i]; letter 2 := woord 2 [i]
end;
if letter 1 alfabetisch voor letter 2 of
   (i = aantal letters woord 1 en i < aantal letters woord 2) then ...
```

Bij het inpassen van dit stuk algoritme in A 7.2 moeten we *oud* [*teller*] i.p.v. *woord 1* en *oud* [*plaats*] i.p.v. *woord 2* plaatsen.
woord 1 en *woord 2* zijn een-dimensionale arrays van letters, *oud* moet een twee-dimensionale array van letters worden:

aantal letters	*oud*
1	10
2	8
3	6

oud	1	2	3	4	5	6	7	8	9	10
1	C	O	W	Z	N	O	F	S	K	I
2	A	A	R	D	V	A	R	K		
3	N	E	U	M	A	N				

De derde letter van de tweede naam duiden we aan met *oud* [2,3], het aantal letters van deze tweede naam met *aantal letters oud* [2].
De opdracht uit A 7.2:

```
laagste := oud [plaats]
```

moet nu gaan luiden:

```
for l from 1 to aantal letters oud [plaats] do
laagste [l] := oud [plaats, l];
aantal letters laagste := aantal letters oud [plaats]
```

Zo uitgewerkt gaat A 7.2 er zo uitzien:

```
plaats := 1;
for teller from 2 to aantal oude do
begin letter 1 := oud [teller, 1]; letter 2 := oud [plaats, 1];
      i := 1;
      while letter 1 = letter 2 en
            i < aantal letters oud [teller] en
            i < aantal letters oud [plaats] do
      begin i := i + 1;
            letter 1 := oud [teller, i]; letter [2] := oud [plaats, i]
      end;
      if letter 1 alfabetisch voor letter 2 of
         (i = aantal letters oud [teller] en
         i < aantal letters oud [plaats]) then
      plaats := teller
end;
for l from 1 to  aantal letters oud [plaats] do
laagste [l] := oud [plaats, l];
aantal letters [laagste] := aantal letters oud [plaats]
```

<29< Eerste niveau:

```
B  initialisatie;
   for i from 1 to 10000 do
   begin worp := dobbel(6);
A        administreer
   end;
C  output
```

Tweede niveau:

A: naast het administreren van de worp door het verhogen van de waarde van de variabele *f [worp]*, moeten we er ook voor zorgen dat in een variabele *na [vorige worp, worp]* de opeenvolging van twee worpen wordt geadministreerd:

```
f [worp] := f [worp] + 1;
if i > 1 then na [vorige worp, worp] := na [vorige worp, worp] + 1;
vorige worp := worp
```

B:
```
for worp from 1 to 6 do
begin f [worp] := 0;
      for vorige worp from 1 to 6 do
      na [vorige worp, worp] := 0
end
```

C:
```
for worp from 1 to 6 do
begin schrijf (worp); schrijf (f [worp]) end;
for vorige worp from 1 to 6 do
begin schrijf ("na"); schrijf (vorige worp);
      for worp from 1 to 6 do
      begin schrijf (worp); schrijf (na [vorige worp, worp]) end
end
```

Resultaat:

```
for worp from 1 to 6 do
begin f [worp] := 0;
      for vorige worp from 1 to 6 do
      na [vorige worp, worp] := 0
end;
for i from 1 to 10000 do
begin worp := dobbel (6);
      f [worp] := f[worp] + 1;
      if i > 1 then
      na [vorige worp, worp] := na [vorige worp, worp] + 1;
      vorige worp := worp
end;
```

```
for worp from 1 to 6 do
begin schrijf (worp); schrijf (f [worp]) end;
for vorige worp from 1 to 6 do
begin schrijf ("na"); schrijf (vorige worp);
      for worp from 1 to 6 do
      begin schrijf (worp); schrijf (na [vorige worp, worp]) end
end
```

<30< Elke *6* in <29< moet in *n* veranderd worden, en de algoritme kan vooraf-
gegaan worden door de opdracht: *n := input;*

```
<31<  plaats := k; min := a [plaats];
      for i from k + 1 to g do
      if a [i] < min then
      begin plaats := i; min := a [plaats] end;
      a [plaats] := a [k]; a [k] := min
```

<32< *for p from k to g do*
schrijf de kleinste waarde van *a* [*p*] t/m *a* [*g*] in *a* [*p*], en
schrijf *a* [*p*] op de plaats waar dit minimum gevonden was

Gedetailleerd met behulp van <31<:

```
for p from k to g do
begin plaats := p; min := a [plaats];
      for i from p + 1 to g do
      if a [i] < min then
      begin plaats := i; min := a [plaats] end;
      a [plaats] := a [p]; a [p] := min
end
```

<33< Door elk van de algoritmen a t/m g wordt afgedrukt: *ja* .

<34< Eerste methode
niveau 1:
A vul het bord;
for x from 1 to 10 do
B zoek in kolom *x* rijen van 5 of meer, en druk ze af

```
niveau 2:
A: maak het bord leeg;
   for s from 1 to 10 do
   begin lees stand; zet hier een kruis op het bord end;
   for s from 1 to 10 do
   begin lees stand; zet hier een nul op het bord end

B: aantal := 0;
   for y from 1 to 10 do
   if dit veld is leeg then {evt. rij is nu afgelopen}
   begin if aantal ≥ 5 then output;
         aantal := 0
   end
   else {er staat een kruis of nul op dit veld}
   if aantal = 0 then {begin van rij na leeg veld}
   begin soort := wat op dit veld staat;
         aantal := 1
   end
   else {er is al een rij, past deze erbij?}
   if op dit veld staat een teken van de soort then {ja!}
   aantal := aantal + 1
   else {nee, dus einde oude rij, begin nieuwe}
   begin if aantal ≥ 5 then output;
         aantal := 1; soort := deze andere soort
   end;
   {is er een rij van minstens 5 afgelopen aan het eind van de kolom?}
   if aantal ≥ 5 then {ja!} output

niveau 3:

initialisatie:
   leeg := 0; kruis := 1; nul := 2;

maak het bord leeg:
   for a from 1 to 10 do
   for y from 1 to 10 do veld [x, y] := leeg
```

lees stand:

```
x := read; y := read
```

zet hier een kruis op het bord:

```
veld [x, y] := kruis
```

zet hier een nul op het bord:

```
veld [x, y] := nul
```

output:

```
schrijf ("kolom"); schrijf (x); schrijf (aantal);
if soort = kruis then schrijf ("kruisen")
else schrijf ("nullen")
```

Resultaat:

```
leeg := 0; kruis := 1; nul := 2;
for x from 1 to 10 do for y from 1 to 10 do veld [x, y] := leeg;
for s from 1 to 10 do
begin x := input; y := input; veld [x, y] := kruis end;
for s from 1 to 10 do
begin x := input; y := input; veld [x, y] := nul end;
for x from 1 to 10 do
begin aantal := 0;
      for y from 1 to 10 do
      if veld [x, y] = leeg then {evt. rij is nu afgelopen}
      begin if aantal ≥ 5 then
            begin schrijf ("kolom"); schrijf (x); schrijf (aantal);
                  if soort = kruis then schrijf ("kruisen")
                  else schrijf ("nullen")
            end;
            aantal := 0
      end else {er staat een kruis of nul op dit veld}
      if aantal = 0 then {begin van rij na leeg veld}
      begin soort := veld [x, y]; aantal := 1 end
      else {er is al een rij, past deze daarbij?}
      if veld [x, y] = soort then {ja!} aantal := aantal + 1
      else {nee, dus einde oude rij, begin nieuwe}
      begin if aantal ≥ 5 then
            begin schrijf ("kolom"); schrijf (x); schrijf (aantal);
                  if soort = kruis then schrijf ("kruisen")
                  else schrijf ("nullen")
            end;
            aantal := 1; soort := veld [x, y]
      end;
      if aantal ≥ 5 then
      begin schrijf ("kolom"); schrijf (x); schrijf (aantal);
            if soort = kruis then schrijf ("kruisen")
            else schrijf ("nullen")
      end
end
```

Tweede methode

niveau 1:

```
   kruis := 1; nul := 2;
   for teken from kruis to nul do
C  begin maak het bord leeg;
B        zet tekens op het bord;
         for x from 1 to 10 do
A        zoek in kolom x rijen van 5 of meer tekens, en druk ze af
   end
```

tweede niveau:

```
A: aantal := 0;
   for y from 1 to 10 do
   if dit veld is bezet then aantal := aantal + 1 else
   if aantal ≥ 5 then
   begin output; aantal := 0 end;
   if aantal ≥ 5 then output

B: for s from 1 to 10 do
   begin x := input; y := input;
         bezet [x, y] := true
   end

C: for x from 1 to 10 do
   for y from 1 to 10 do bezet [x, y] := false
```

derde niveau:

```
output:
   schrijf ("kolom"); schrijf (x); schrijf (aantal);
   if teken = kruis then schrijf ("kruisen")
   else schrijf ("nullen")
```

Resultaat:

```
kruis := 1; nul := 2;
for teken from kruis to nul do
begin for x from 1 to 10 do
      for y from 1 to 10 do bezet [x, y]:= false;
      for s from 1 to 10 do
      begin x := input; y:= input;
            bezet [ x, y ] := true
      end;
      for x from 1 to 10 do
      begin aantal := 0;
            for y from 1 to 10 do
            if bezet [x, y] then aantal := aantal + 1 else
            if aantal ≥ 5 then
            begin schrijf ("kolom"); schrijf (x); schrijf (aantal);
                  if teken = kruis then schrijf ("kruisen")
                  else schrijf ("nullen");
                  aantal := 0
            end
      end;
      if aantal ≥ 5 then
      begin schrijf ("kolom"); schrijf (x); schrijf (aantal);
            if teken = kruis then schrijf ("kruisen")
            else schrijf ("nullen")
      end
end
```

Derde methode

Omdat 10 kruisjes en 10 nulletjes slechts in enkele van de 10 kolommen van het bord een rijtje van vijf kunnen vormen, is het handig om bij het lezen van de positie van de kruisjes en de nulletjes bij te houden hoeveel er in elke kolom staan. Alleen in de kolommen met 5 of meer kruisjes en in die met 5 of meer nulletjes hoeft dan nog gezocht te worden.

niveau 1:

initialiseer;
kruis := 1; nul := 2;
for teken from kruis to nul do
begin maak het bord leeg;
zet tekens op het bord en turf per kolom;
for x from 1 to 10 do
if er staan 5 of meer tekens in kolom *x then*
kijk of er een rij van 5 of meer is, en druk deze af
end

De uitwerking verschilt weinig van die van methode 2:

initialiseer:
for x from 1 to 10 do aantal in kolom [x] := 0

zet tekens op het bord en turf per kolom:
for s from 1 to 10 do
begin x := input; aantal in kolom [x] := aantal in kolom [x] + 1;
y := input; bezet [x, y] := true
end

<35< eerste niveau:

E initialiseer tellingen;
for p from 1 to 10 do
A { *begin* lees stand;
kruisentelling van deze kolom met 1 verhogen;
kruisentelling van deze rij met 1 verhogen
end;
for p from 1 to 10 do
B { *begin* lees stand;
nullentelling van deze kolom met 1 verhogen;
nullentelling van deze rij met 1 verhogen
end;
C zoek hoogste telling;
D output

Tweede niveau:

```
A: x := input;
   kruisen in kolom [x] := kruisen in kolom [x] + 1;
   y := input;
   kruisen in rij [y] := kruisen in rij [y] + 1

B: x := input;
   nullen in kolom [x] := nullen in kolom [x] + 1;
   y := input;
   nullen in rij [y] := nullen in rij [y] + 1

C: kruis := 1; nul := 2; kolom := 1; rij := 2;
   nr := 1; soort := kruis; richting := kolom;
   max := kruisen in kolom [nr];
   for x from 2 to 10 do
   if kruisen in kolom [x] > max then
   begin nr := x; max := kruisen in kolom [nr] end;
   for y from 1 to 10 do
   if kruisen in rij [y] > max then
   begin nr := y; richting := rij; max := kruisen in rij [nr] end;
   for x from 1 to 10 do
   if nullen in kolom [x] > max then
   begin nr := x; richting := kolom; soort := nul;
         max := nullen in kolom [nr]
   end;
   for y from 1 to 10 do
   if nullen in rij [y] > max then
   begin nr := y; richting := rij; soort := nul;
         max := nullen in rij [nr]
   end

D: if richting = rij then schrijf ("rij") else schrijf ("kolom");
   schrijf (nr); schrijf (max);
   if soort = kruis then schrijf ("kruisen") else schrijf ("nullen")

E: for x from 1 to 10 do
   begin kruisen in kolom [x] := 0; kruisen in rij [x] := 0;
         nullen in kolom [x] := 0; nullen in rij [x] := 0
   end
```

Resultaat:

```
for x from 1 to 10 do
begin kruisen in kolom [x] := 0; kruisen in rij [x] := 0;
      nullen in kolom [x] := 0; nullen in rij [x] := 0
end;
for p from 1 to 10 do
begin x := input; kruisen in kolom [x] := kruisen in kolom [x];
      y := input; kruisen in rij [y] := kruisen in rij [y] + 1
end;
for p from 1 to 10 do
begin x := input; nullen in kolom [x] := nullen in kolom [x] + 1;
      y := input; nullen in rij [y] := nullen in rij [y] + 1
end;
kruis := 1; nul := 2; kolom := 1; rij := 2;
nr := 1; soort := kruis; richting := kolom;
max := kruisen in kolom [nr];
for x from 2 to 10 do
if kruisen in kolom [x] > max then
begin nr := max; max := kruisen in kolom [nr] end;
for y from 1 to 10 do
if  kruisen in rij [y] > max then
begin nr := y; richting := rij; max := kruisen in rij [nr] end;
for x from 1 to 10 do
if nullen in kolom [x] > max then
begin nr := x; richting := kolom; soort := nul;
      max := nullen in kolom [nr]
end;
for y from 1 to 10 do
if nullen in rij [y] > max then
begin nr := y; richting := rij; soort := nul;
      max := nullen in rij [nr]
end;
if richting = rij then schrijf ("rij") else schrijf ("kolom");
schrijf (nr); schrijf (max);
if soort = kruis then schrijf ("kruisen") else schrijf ("nullen")
```

Doordat het maximum gezocht moest worden in vier arrays, kost dit stuk van de algoritme nogal veel schrijfwerk. Ook het initialiseren en bijwerken van deze vier arrays is vrij bewerkelijk om op te schrijven. Door gebruik te maken van slechts een array van dimensie 3 kan dit schrijfwerk bekort worden. Als variabele om b.v. het aantal nullen in kolom 4 bij te houden gebruiken we in deze opzet *aantal [nul, kolom, 4]* (waarbij *nul = 2* en *kolom = 1*), etc.

```
kruis := 1; nul := 2; kolom := 1; rij := 2;
for s from kruis to nul do
for r from kolom to rij do
for n from 1 to 10 do aantal [s, r, n] := 0;

for s from kruis to nul do
for r from kruis to rij do
begin n := input; aantal [s, r, n] := aantal [s, r, n] + 1 end;

soort := kruis; richting := kolom; nr := 1;
max := aantal [soort, richting, nr];
for s from kruis to nul do
for r from kolom to rij do
for n from 1 to 10 do
if aantal [s, r, n] > max then
begin soort := s; richting := r; nr := r;
      max := aantal [soort, richting, r]
end;

if richting = rij then schrijf ("rij") else schrijf ("kolom");
schrijf (nr); schrijf (max);
if soort = kruis then schrijf ("kruisen") else schrijf ("nullen")
```

<36< Eerste niveau:

```
A initialiseer de administratie;
  lengte := 1; soort := input;
  for i from 2 to 1000 do
  begin nieuw := input;
        if nieuw  = soort then lengte := lengte + 1 else
```

```
B       begin administreer de afgelopen rij;
              lengte := 1; soort := nieuw
        end
  end;
C administreer de laatste rij;
D output
```

Als de langste rij nullen of enen die voorkomt een lengte *maxlengte* heeft, zal de administratie moeten bestaan uit 2 × *maxlengte* variabelen. Het is duidelijk dat 1 ≤ *maxlengte* ≤ 1000.
Het initialiseren zal nu bestaan uit het plaatsen van de waarde *0* in 2 × 1000 variabelen, maar het lijkt verstandig de output af te breken bij de grootste aangetroffen lengte: *maxlengte* .

Tweede niveau:

```
A: for lengte from 1 to 1000 do
   begin f [0, lengte] := 0; f [1, lengte] := 0 end;
   maxlengte := 1;

B: f [soort, lengte] := f [soort, lengte] + 1;
   if lengte > maxlengte then maxlengte := lengte

C: hetzelfde als B

D: for lengte from 1 to maxlengte do
   begin schrijf (lengte); schrijf (f [0, lengte]);
         schrijf (f [1, lengte])
   end
```

Resultaat:

```
   for lengte from 1 to 1000 do
   begin f [0, lengte] := 0; f [1, lengte] := 0 end;
   maxlengte := 1;
   lengte := 1; soort := input;
   for i from 2 to 1000 do
   begin nieuw := input;
```

```
            if nieuw = soort then lengte := lengte + 1 else
            begin f [soort, lengte] := f [soort, lengte] + 1;
                  if lengte > maxlengte then maxlengte := lengte;
                  lengte := 1; soort := nieuw
            end
      end;
      f [soort, lengte] := f [soort, lengte] + 1;
      if lengte > maxlengte then maxlengte := lengte;
      for lengte from 1 to maxlengte do
      begin schrijf (lengte); schrijf (f [0,lengte]);
            schrijf (f [1, lengte])
      end
```

<37<

```
      procedure sorteer 3 (i, j, k):
      begin sorteer 2 (x, y);
            sorteer 2 (y, z);
            sorteer 2 (x, y)
      end.

      procedure sorteer 2 (a, b):
      if a > b then wissel (a, b)

      procedure wissel (p, q):
      begin w := p; p := q; q := w end
```

De aanroep *sorteer 3 (i, j, k)* betekent nu:

```
sorteer 2 (i, j); sorteer 2 (j, k); sorteer 2 (i, j);
```

en dit betekent:

```
if i > j then wissel (i, j);
if j > k then wissel (j, k);
if i > j then wissel (i, j)
```

en dit betekent tenslotte:

```
if i > j then begin w := i; i := j; j := w end;
if j > k then begin w := j; j := k; k := w end;
if i > j then begin w := i; i := j; j := w end
```

UITGAVEN IN DE SERIE MC SYLLABUS

Onderstaande uitgaven zijn verkrijgbaar bij het Mathematisch Centrum, 2e Boerhaavestraat 49 te Amsterdam-1005, tel. 020-947272.

MCS 1.1 F. GÖBEL & J. VAN DE LUNE, *Leergang Besliskunde, deel 1: Wiskundige basiskennis*, 1965. ISBN 90 6196 014 2.

MCS 1.2 J. HEMELRIJK & J. KRIENS, *Leergang Besliskunde, deel 2: Kansberekening*, 1965. ISBN 90 6196 015 0.

MCS 1.3 J. HEMELRIJK & J. KRIENS, *Leergang Besliskunde, deel 3: Statistiek*, 1966. ISBN 90 6196 016 9.

MCS 1.4 G. DE LEVE & W. MOLENAAR, *Leergang Besliskunde, deel 4: Markovketens, en wachttijden*, 1966. ISBN 90 6196 017 7.

MCS 1.5 J. KRIENS & G. DE LEVE, *Leergang Besliskunde, deel 5: Inleiding tot de mathematische besliskunde*, 1966. ISBN 90 6196 018 5.

MCS 1.6a B. DORHOUT & J. KRIENS, *Leergang Besliskunde, deel 6a: Wiskundige programmering 1*, 1968. ISBN 90 6196 032 0.

MCS 1.7a G. DE LEVE, *Leergang Besliskunde, deel 7a: Dynamische programmering 1*, 1968. ISBN 90 6196 033 9.

MCS 1.7b G. DE LEVE & H.C. TIJMS, *Leergang Besliskunde, deel 7b: Dynamische programmering 2*, 1970. ISBN 90 6196 055 X.

MCS 1.7c G. DE LEVE & H.C. TIJMS, *Leergang Besliskunde, deel 7c: Dynamische programmering 3*, 1971. ISBN 90 6196 066 5.

MCS 1.8 J. KRIENS, F. GÖBEL & W. MOLENAAR, *Leergang Besliskunde, deel 8: Minimaxmethode, netwerkplanning, simulatie*, 1968. ISBN 90 6196 034 7.

MCS 2.1 G.J.R. FÖRCH, P.J. VAN DER HOUWEN & R.P. VAN DE RIET, *Colloquium stabiliteit van differentieschema's, deel 1*, 1967. ISBN 90 6196 023 1.

MCS 2.2 L. DEKKER, T.J. DEKKER, P.J. VAN DER HOUWEN & M.N. SPIJKER, *Colloquium stabiliteit van differentieschema's, deel 2*, 1968. ISBN 90 6196 035 5.

MCS 3.1 H.A. LAUWERIER, *Randwaardeproblemen, deel 1*, 1967. ISBN 90 6196 024 X.

MCS 3.2 H.A. LAUWERIER, *Randwaardeproblemen, deel 2*, 1968. ISBN 90 6196 036 3.

MCS 3.3 H.A. LAUWERIER, *Randwaardeproblemen, deel 3*, 1968. ISBN 90 6196 043 6.

MCS 4 H.A. LAUWERIER, *Representaties van groepen*, 1968. ISBN 90 6196 037 1.

MCS 5 J.H. VAN LINT, J.J. SEIDEL & P.C. BAAYEN, *Colloquium discrete wiskunde*, 1968. ISBN 90 6196 044 4.

MCS 6 K.K. KOKSMA, *Cursus ALGOL 60*, 1969. ISBN 90 6196 045 2.

MCS 7.1 *Colloquium Moderne rekenmachines, deel 1*, 1969. ISBN 90 6196 046 0.

MCS 7.2 *Colloquium Moderne rekenmachines, deel 2*, 1969. ISBN 90 6196 047 9.

MCS 8 H. BAVINCK & J. GRASMAN, *Relaxatietrillingen*, 1969. ISBN 90 6196 056 8.

MCS 9.1 T.M.T. COOLEN, G.J.R. FÖRCH, E.M. DE JAGER & H.G.J. PIJLS, *Elliptische differentiaalvergelijkingen, deel 1*, 1970. ISBN 90 6196 048 7.

MCS 9.2 W.P. VAN DEN BRINK, T.M.T. COOLEN, B. DIJKHUIS, P.P.N. DE GROEN, P.J. VAN DER HOUWEN, E.M. DE JAGER, N.M. TEMME & R.J. DE VOGELAERE, *Colloquium Elliptische differentiaalvergelijkingen, deel 2*, 1970. ISBN 90 6196 049 5.

MCS 10 J. FABIUS & W.R. VAN ZWET, *Grondbegrippen van de waarschijnlijkheidsrekening*, 1970. ISBN 90 6196 057 6.

MCS 11 H. BART, M.A. KAASHOEK, H.G.J. PIJLS, W.J. DE SCHIPPER & J. DE VRIES, *Colloquium Halfalgebra's en positieve operatoren*, 1971. ISBN 90 6196 067 3.

MCS 12 T.J. DEKKER, *Numerieke algebra*, 1971. ISBN 90 6196 068 1.

MCS 13 F.E.J. KRUSEMAN ARETZ, *Programmeren voor rekenautomaten; De MC ALGOL 60 vertaler voor de EL X8*, 1971. ISBN 90 6196 069 x.

MCS 14 H. BAVINCK, W. GAUTSCHI & G.M. WILLEMS, *Colloquium Approximatietheorie*, 1971. ISBN 90 6196 070 3.

MCS 15.1 T.J. DEKKER, P.W. HEMKER & P.J. VAN DER HOUWEN, *Colloquium Stijve differentiaalvergelijkingen, deel 1*, 1972. ISBN 90 6196 078 9.

MCS 15.2 P.A. BEENTJES, K. DEKKER, H.C. HEMKER, S.P.N. VAN KAMPEN & G.M. WILLEMS, *Colloquium Stijve differentiaalvergelijkingen, deel 2*, 1973. ISBN 90 6196 079 7.

MCS 15.3 P.A. BEENTJES, K. DEKKER, P.W. HEMKER & M. VAN VELDHUIZEN, *Colloquium Stijve differentiaalvergelijkingen, deel 3*, 1975. ISBN 90 6196 118 1.

MCS 16.1 L. GEURTS, *Cursus Programmeren, deel 1: De elementen van het programmeren*, 1973. ISBN 90 6196 080 0.

MCS 16.2 L. GEURTS, *Cursus Programmeren, deel 2: De programmeertaal ALGOL 60*, 1973. ISBN 90 6196 087 8.

MCS 17.1 P.S. STOBBE, *Lineaire algebra, deel 1*, 1974. ISBN 90 6196 090 8.

MCS 17.2 P.S. STOBBE, *Lineaire algebra, deel 2*, 1974. ISBN 90 6196 091 6.

MCS 17.3 N.M. TEMME, *Lineaire algebra, deel 3*, 1976. ISBN 90 6196 123 8.

MCS 18 F. VAN DER BLIJ, H. FREUDENTHAL, J.J. DE IONGH, J.J. SEIDEL & A. VAN WIJNGAARDEN, *Een kwart eeuw wiskunde 1946-1971, Syllabus van de Vakantiecursus 1971*, 1974. ISBN 90 6196 092 4.

MCS 19 A. HORDIJK, R. POTHARST & J.TH. RUNNENBURG, *Optimaal stoppen van Markovketens*, 1974. ISBN 90 6196 093 2.

MCS 20 T.M.T. COOLEN, P.W. HEMKER, P.J. VAN DER HOUWEN & E. SLAGT, *ALGOL 60 procedures voor begin- en randwaardeproblemen*, 1976. ISBN 90 6196 094 0.

MCS 21 J.W. DE BAKKER (red.), *Colloquium Programmacorrectheid*, 1975. ISBN 90 6196 103 3.

* MCS 22 R. HELMERS, F.H. RUYMGAART, M.C.A. VAN ZUYLEN & J. OOSTERHOOF, *Asymptotische methoden in de statistiek*, 1976. ISBN 90 6196 104 1.

MCS 23.1 J.W. DE ROEVER (red.), *Colloquium Onderwerpen uit de biomathematica, deel 1*, 1976. ISBN 90 6196 105 X.

* MCS 23.2 J.W. DE ROEVER (red.), *Colloquium Onderwerpen uit de biomathematica, deel 2*, 1976. ISBN 90 6196 115 7.

MCS 24.1 P.J. VAN DER HOUWEN, *Numerieke integratie van differentiaalvergelijkingen, deel 1: Eenstapsmethoden* , 1974. ISBN 90 6196 106 8.

MCS 25 *Colloquium Structuur van Programmeertalen*, 1976. ISBN 90 6196 116 5.

MCS 26.1 N.M. TEMME (red.), *Nonlinear Analysis, volume 1*, 1976. ISBN 90 6196 117 3.

MCS 26.2 N.M. TEMME (red.), *Nonlinear Analysis, volume 2*, 1976. ISBN 90 6196 121 1.

MCS 27 M. BAKKER, P.W. HEMKER, P.J. VAN DER HOUWEN, S.J. POLAK & M. VAN VEELDHUIZEN, *Colloquium Discreteringmethoden*, 1976. ISBN 90 6196 124 6.

* MCS 28 N.M. TEMME (red.), *Nonlinear Diffusion Problems*, 1976. ISBN 90 6196 126 2.

* MCS 29.1 J.C.P. BUS (red.), *Numerieke Programmatuur, deel 1*, 1976. ISBN 90 6196 128 9.

MCS 31 J.H. VAN LINT (red.), *Inleiding in de Coderingstheorie*, 1976. ISBN 90 6196 136 X.

* MCS 32 L. GEURTS (red.), *Colloquium Bedrijfssystemen*, 1976. ISBN 90 6196 137 8.

De met een * gemerkte uitgaven moeten nog verschijnen.