SCHETS

VAN EEN HOOFDSTUK

OVER

HET ONTWERPEN VAN ALGORITMEN

door

M. M. Fokkinga
T. H. Twente

12 sept 1974

(herzien): 8 okt 1974

EEN WOORD VOORAF

De voor u liggende tekst is ontstaan door de wens van sommigen (dd. zomer 1974) om bij het Inleidend Informatica - onderwijs bij een Technische Hogeschool in den lande volgens modernere inzichten te werk te gaan. V Aangezien de verzorgers van het onderwijs nogal behoudend waren wat betreft hun ideeen over de methodiek van het programmeren, moest de tekst niet al te revolutionair worden. Derhalve zijn termen als "gestructureerd Programmeren", "asserties" e.d. met grote zorgvuldigheid vermeden. "Bewijssystemen voor de korrektheid van algoritmen" zijn niet behandeld, om de tekst toch maar enigszins aanvaardbaar te laten zijn, en om dezelfde reden zijn "Programmastroomschema's" wel aan bod gekomen.

Graag had ik ook de "semantiek" van de geboezigde algoritmetaal minder operationeel gegeven, al heb ik nog geen idee hoe die uitleg er dan uiteindelijk uit zou hebben gezien.

Bovendien lijkt mij dit hoofdstuk vragen om meer: tesamen met een hoofdstuk "algemeen toepasbare technieken" en een hoofdstuk "Bewijssystemen voor korrektheid van algoritmen" vormt dit een aardige Trilogie , als prerequisiet voor een "cursus Algol".

Een noodzakelijke voorwaarde is dan wel dat alle drie de hoofdstukken, dus ook het voor u liggende, een aanvulling "Oefeningen en Opgaven" krijgen.

Twente, 12 sept. '74

W. Dijkstra

V. Ik heb toen een poging gedaan in ±10 pag's een bijdrage daartoe te leveren. Die 10 pag's zijn wel tot ±30 uitgelopen!

- INHOUD

Bij de herziene versie dd 8/10/74:
Dh len. C. Bron zeer dankbaar voor waardevolle opmerkingen. Die hadden o.a. tot gevolg dat ik me veel heb beperkt in de toegelaten vorm van huiswerkopdrachten. In de eerste versie "zaten er meer addertjes onder het gras" dan ik had gedacht!

MT.

0. INLEIDING

(0.1) Dit hoofdstuk is geen ALGOL-cursus. Het onderwerp van behandeling is (de methodiek van) het ontwerpen van (voor computerverwerking bedoelde) algoritmen. Hoewel er enige keren naar ALGOL verwezen wordt, is kennis daarvan niet noodzakelijk. De verwijzingen dienen slechts om te illustreren dat de programmeertaal alleen maar richting geeft aan het ontwerpproces en de methodiek zelf niet beinvloed.

(0.2) We nemen de volgende definitie aan.
Een algoritme is een eindige rij opdrachten die ieder van een type zijn dat bekend is aan de uitvoerder.

(0.3) Wanneer we de computer een ALGOL programma aanbieden, dan kunnen we zeggen dat de computer de uitvoerder en het programma de algoritme is. De titel van dit hoofdstuk had dus ogenschijnlijk ook "het ontwerpen van programma's" kunnen heten. Maar we hebben voor de algemenere titel gekozen omdat we ons juist niet willen bezighouden met de definitieve tekst in een of ander programmeertaal, maar meer met de globale lijn van de programma's.

1. VOORBEELD

"Goed voorbeeld doet goed volgen"
- wijs gezegde -

(1.1) Om het ontwerpproces duidelijk tot uiting te laten komen zal ik me in de eerste naamval tot u richten. Laat ik als voorbeeld een algoritme ontwerpen om een willekeurige rij objecten te sorteren naar grootte. Vooralsnog veronderstel ik dat een mens, u bijvoorbeeld, de uitvoerder van mijn algoritme is. Dan heb ik geen problemen met het feit dat "de uitvoerder bekend moet zijn met het type van de opdrachten". Maar alengs veronderstel ik u dommer en dommer totdat u nog slechts bekend bent met de opdrachten van de programmeertaal ALGOL. De begrippen die ik in ieder geval bekend veronderstel zijn onderstreept. Namen, om dingen aan te duiden, zijn schuingedrukt.*)

(1.2) Mijn eerste ontwerp is:

begin schrijf *reeks* op een nieuwe bladzij;
sorteer *reeks* naar grootte;
schrijf *reeks* op een nieuwe bladzij;
einde

(1.3) Hierbij dient u te weten dat ik de rij objecten gegeven veronderstel en dat ik die rij objecten *reeks* heb genoemd. Voor goed begrip zal ik dat vooraf vermelden. Om makkelijker naar mijn algoritme te kunnen verwijzen geef ik het de naam *sorteer*. Dus hiermee heb ik de nieuwe grootheid *sorteer* geïntroduceerd:

---

*) In het manuscript: met potlood onderstreept. In de fotocopieen wrsch.: vaag onderstreept.

(1.4)

nieuwe grootheid sorteer(reeks) zijnde een algoritme, namelijk grootheid reeks zijnde een rij van objecten
{ik veronderstel dat u de elementen van reeks kent};
toelichting het effect van sorteer(reeks) is dat de reeks gesorteerd wordt opgeleverd en de oorspronkelijke en gesorteerde rij wordt afgedrukt;
begin schrijf reeks op een nieuwe bladrij;
sorteer reeks naar grootte;
schrijf reeks op een nieuwe bladrij
einde

(1.5)

U kunt moeilijk ontkennen dat de algoritme korrekt is. Want als u de algoritme uitvoert zoals ik de opdrachten heb bedoeld, dan gebeurt er precies wat ik me tot doel had gesteld. Maar misschien snapt u mijn opdrachten nog niet. Daarom zal ik de opdracht sorteer reeks naar grootte specificeren.

(1.6)

Wat ik wil is dat u achtereenvolgens het kleinste, het een-na-kleinste, het twee-na-kleinste ... etc. van de elementen van reeks op hun plaats zet. (Nota bene. Ik had ook kunnen willen dat u van groot naar klein de elementen op hun plaats zou zetten, of op een nog niet bepaalde volgorde! Of ik had kunnen willen dat u steeds iets meer ordening in de rij zou brengen door de grotere elementen van links met kleinere elementen van rechts te verwisselen!)

Omdat de objecten eventueel gelijk in grootte maar toch verschillend (bijv. in kleur) mogen zijn, zeg ik in het vervolg "een minimaal" i.p.v. "het kleinste".

Mijn specificatie wordt dus zoiets als:

(1.7)

zoek een minimaal element van reeks;
zet dat op de eerste plaats {nu staat het 1° elt goed};
zoek een minimaal elt van de rest van reeks;
zet dat op de tweede plaats {nu staat het 1°, 2° elt goed};
⋮

⋮

enzovoort, tot en met de laatste.

(1.8) U zult die stippeltjes misschien nog wel begrijpen, maar in programmeertalen zijn ze zo zeker niet toegestaan. Omdat ik niet weet hoeveel keren ik dat zoekproces moet uitschrijven, formuleer ik het als een herhalingsopdracht, van de vorm

(1.9) zolang ⟨een voorwaarde⟩ doe ⟨een serie opdrachten⟩ herhaaldelijk

Ik introduceer hiervoor een nieuwe grootheid die steeds aangeeft tot en met waar de elementen op hun juiste plaats gezet zijn. Die grootheid, een ~~rangnummer~~ geheel getal, geef ik de naam rgno ! als namelijk de elementen van reeks genummerd zijn van eerste tot en met laatste, dan kan ik herhaaldelijk het juiste element op de plaats na rgno zetten en rgno de waarde van het volgende rangnummer geven. Ik realiseer me dat gedurende de herhaling de eigenschap

{de el'ten van reeks van eerste t/m rgno staan al goed}

invariant blijft. De uitwerking krijgt dus de volgende structuur *):

(1.10)

SORTEER REEKS NAAR GROOTTE :
{ik veronderstel dat reeks genummerd is van eerste t/m laatste}
nieuwe grootheid rgno zijnde een geheel getal {bedoeld als aanduiding t/m waar de el'ten van reeks al goed staan}
initialisatie ;
zolang rgno nog kleiner is dan laatste
doe {de el'ten van reeks van eerste t/m rgno staan al goed}
herhaalde opdrachtserie
{de el'ten van reeks van eerste t/m rgno staan al goed}
herhaaldelijk
{ hier geldt : de el'ten van reeks van eerste t/m laatste staan goed}

hier geldt:

hier geldt:

*) Steeds geef ik de oorspronkelijke opdracht als "titel" (in hoofdletters) aan de uitwerking ervan.

(1.11) Hieruit volgt

a) dat ik initialisatie zó moet specificeren dat de elten vanaf eerste t/m rgno al goed staan wanneer aan de herhaling begonnen wordt. Dit kan ik bijvoorbeeld proberen met de specificatie:

zoek een minimaal element van reeks;
{stel dat op de eerste plaats (nu staat het 10 al goed)};
geef rgno de waarde van eerste.

Maar als laatste < eerste, dwz dat reeks géén elementen bevat en dus zeker géén min. elt, dan is deze specificatie incorrect. Mijn volgende specificatie is gegarandeerd correct:

geef rgno de waarde van eerste − 1.

Immers daarna, dus aan het begin van de herhaling, geldt dat de elten vanaf eerste t/m rgno (dwz géén elten) al goed staan. Bovendien valt de test gelijk al negatief uit indien laatste < eerste en is in dat geval aan de bewering na afloop voldaan.

b) dat ik de herhaalde opdrachtserie kan specificeren als, bijvoorbeeld:

zoek in reeks na rgno een min. elt;
verwissel dat van plaats met het elt op de plaats rgno+1;
geef rgno de waarde van rgno+1

of als:

geef rgno de waarde van rgno+1;
zoek in reeks vanaf rgno een min. elt;
verwissel dat van plaats met het elt op de plaats rgno

of als:

zoek in reeks na rgno een min. elt;
geef rgno de waarde van rgno+1;
verwissel het van plaats met het elt op de plaats rgno

En geen van deze mogelijkheden geniet bij mij voorkeur boven een andere. Ik zal de eerste specificatie kiezen.

geef plnr de waarde van eerste ;
herhaal {de el'ten vanaf eerste tót aan plnr staan al goed}
zoek in reeks vanaf plnr een minimaal element ;
verwissel dat van plaats met el't op plnr ;
geef plnr de waarde van het volgende plaatsnummer
{weer geldt: de el'ten vanaf eerste tót aan plnr staan goed}
totdat plnr = laatste
{nu geldt: de el'ten vanaf eerste tót aan laatste (dus tót en mét) staan al goed}

(1.12) U ziet dat ik ~~in de voorgaatste specificatie al~~ onderdussen het "zet dat op zijn plaats" wat nader heb geformuleerd door
verwissel dat van plaats met ...
Ik concludeer nu dat ik niet zozeer geïnteresseerd ben in de waarde van het minimale element, maar in zijn rangnr. Een verwisseling kan je uitvoeren als je weet wie er verwisseld moeten worden! Daarom kan ik
zoek ... een minimaal element
beter formuleren door
zoek ... het rangnummer van een minimaal element.
Dit zal ik zo dadelijk uitwerken.

(1.13) Omwille van een korte formulering wil ik dat een willekeurig element van reeks aangeduid kan worden met zijn rangnummers Van uw wiskundelessen herinnert u zich dat zoiets gedaan wordt door indicering: t_1, t_2, t_3, ... is het eerste, het tweede, het derde, ... element van de rij t. Omdat het uiteindelijke programma niet met indiceringen gepoust kan worden, schrijf ik het als $t[1]$, $t[2]$, $t[3]$, ... En $t[a \ldots b]$ duidt dan t_a t/m t_b aan.
Het is een belangrijke eigenschap dat op deze manier willekeurig element van reeks bereikbaar is. Daarom geef ik het op bij de veronderstellingen over de grootheid reeks. Verder vind ik dat het verwisselen van twee elementen van reeks zo elementair is, dat ik dat -voorlopig-

niet verder wil uitwerken; zoiets moet maar aan de uitvoerder van zijn algoritme bekend zijn. Dus ook dat neem ik op in de veronderstellingen over de grootheid reeks zijnde een rij van objecten:

(1.14) grootheid reeks *): rij van objecten { ik veronderstel
a) dat willekeurig elt van reeks bereikbaar is met reeks[..], waarin .. het rang-nummer is,
b) dat twee elten van reeks van plaats verwisseld kunnen worden door de operatie verwissel .. met .. }
grootheid eerste, laatste: geheel getal { ik veronderstel dat reeks[eerste ... laatste] gesorteerd moet worden}

(1.15) Nu komt de aangekondigde uitwerking [redacted], nl
zoek in reeks [zgno+1 ... laatste] een min. elt.
Wat ik wil is dat ik één voor één de elementen afloopt en daarbij steeds onthoudt wat het tot dan toe gevonden minimale element is. Daartoe introduceer ik een nieuwe grootheid met de naam zgmin, die bedoeld is als ~~aanduiding~~ rangnummer van het -tijdelijke- minimale element. Dan moet ik zgmin een beginwaarde geven en daarna reeks[zgmin] achtereenvolgens met reeks[zgno], reeks[zgno+1], reeks[zgno+2] .... tot en met de laatste vergelijken en zonodig zijn waarde aanpassen, op de volgende manier:

⋮
als reeks[zgno+--] kleiner is dan reeks[zgmin]
dan geef zgmin de waarde van zgno+--
⋮

(1.16) En alweer, omdat ik de waarde van laatste niet ken en dus niet weet hoeveel keer ik de voorwaardelijke opdrachten (als ... dan ....) moet uitschrijven en het toch heel precies wil formuleren, doe ik het door een herhalingsopdracht. Ik introduceer hiervoor een nieuwe grootheid die steeds aangeeft >zie volg blad 7>

*) In zo'n positie gebruik ik de dubbele punt als afkorting van "zijnde een". Spreek hem ook zo uit !!

tot en met waar er gezocht is naar een min. elt.
Die grootheid geef ik de naam k. Dus zolang k
nog kleiner is dan laatste wil ik herhaaldelijk k
ophogen en daarbij de betrekking

{reeks[rgmin] is min. in reeks[rgno+1...k]}

invariant houden:

(1.17)

ZOEK IN REEKS[RGNO+1 ... LAATSTE] EEN MIN. ELT.:
nieuwe grootheid rgmin: geheel getal {bedoeld als
rangnummer van het te zoeken minimale elt};
nieuwe grootheid k: geheel getal {bedoeld als
aanduiding t/m waar er al gezocht is naar min. elt};
initialisatie;
zolang k nog kleiner is dan laatste
  doe {hier geldt: reeks[rgmin] is min. in reeks[rgno+1...k]}
    herhaalde opdrachtserie
      {hier geldt: reeks[rgmin] is min. in reeks[rgno+1...k]}
  herhaaldelijk
{hier geldt nu: reeks[rgmin] is min. in reeks[rgno+1 ... laatste]}

(1.18)

En uit deze opzet volgt
a) dat ik initialisatie zo moet specificeren dat reeks
reeks[rgmin] al min. is in reeks[rgno+1...k] wanneer aan
de herhaling begonnen wordt. Dit kan ik bijvoorbeeld
~~proberen~~ bereiken met:

geef rgmin de waarde van rgno+1;
geef k de waarde van rgno+1

b) dat ik de herhaalde opdrachtserie moet specificeren als
bijvoorbeeld:

geef k de waarde van k+1;
als reeks[k] kleiner is dan reeks[rgmin]
  dan geef rgmin de waarde van k

maar ook helvolgende is een gegarandeerd goede specificatie:

als reeks[k+1] kleiner is dan reeks[rgmin]
  dan geef rgmin de waarde van k;
geef k de waarde van k+1.

(119) Ik vind dat de algoritme sorteer al aardig is uitgewerkt. ~~Iemand is uitgewerkt~~. Ik begrijpt toch alles? Het wordt nu tijd me af te vragen wat ik nog verder moet specificeren opdat mijn algoritme geschikt is voor computerverwerking.

(I) INTERMEZZO.

(I.1) Van algoritmen die bedoeld zijn voor computerverwerking moeten we een drietal essentiële kenmerken goed kennen opdat we zinvol kunnen ontwerpen. Deze kenmerken zijn:

-1- samenstellingen van opdrachten (besturingsstructuren)
-2- nieuwe grootheden (datastructuren)
-3- elementaire opdrachten (toekenning, voorwaarden)

(I.2) Ad 1. Besturingsstructuren.

Zo genoemd omdat ze de uitvoerder "sturen" door de programmatekst.

Er zijn drie belangrijke mogelijkheden om opdrachten samen te stellen. De samenstellingen leggen tevens de volgorde vast waarin de opdrachten moeten worden uitgevoerd.

a) sequentiële samenstelling, dmv. de puntkomma.
b) voorwaardelijke opdracht, dmv. de tekst

als < voorwaarde >
    dan < opdracht 1 >
    anders < opdracht 2 >

waarbij de laatste regel eventueel weggelaten mag worden

Iets algemener en soms wat handiger is het onderscheiden van gevallen, dmv. de tekst

ingeval
< voorwaarde 1 > dan < opdracht 1 > ;
⋮
< voorwaarde n > dan < opdracht n >
en anders dan < opdracht n+1 >

... en ook hier hoeft de laatste clausule er niet per se bij. Ter voorkoming van misverstand moet u er wel voor zorgen dat de voorwaarden elkaar uitsluiten.

c) herhalingsopdracht, dmv. de tekst

zolang ⟨voorwaarde⟩
doe
⟨ ⋮ opdrachtenserie ⋮ ⟩
herhaaldelijk

Heel soms kan het gebeuren dat u de herhaling wil onderbreken midden in de tekst van de opdrachtenserie. Dat kan dan door op die plaats
als .... dan beëindig de herhaling
te zetten. Maar dat is niet aanbevelenswaardig omdat het een stuk moeilijker wordt om u zelf van de correctheid te overtuigen van de aan het eind van de tekst van de herhalingsopdracht geformuleerde situatie. (in commentaarvorm {...})

In paragraaf 3 worden deze drie besturingsstructuren nader gemotiveerd.

>zie volg blad2>

(I.3) Ad 2. Datastructuren

Zo genoemd omdat ze dienen voor de opslag van data (= gegevens).

Van fundamenteel belang is de mogelijkheid nieuwe grootheden te introduceren. Om je een goede voorstelling te vormen geven we eerst een idee van wat er bij de uitvoering van een algoritme gebeurt. Daartoe veronderstellen we dat een mens de uitvoerder van de algoritme is en niet de elektronische rekenmachine.

(I.4) De uitvoerder is erg goed in hoofdrekenen. Hij kan geweldig lange expressies zonder hulpmiddelen uitrekenen. Zijn geheugen is minder en daarom heeft hij een haantenbak tot zijn beschikking met haarten van allerlei formaat, die in het begin nog blanco zijn. Op de haarten komen de gegevens te staan die hij moet onthouden. De haarten die in gebruik zijn, zijn voorzien van een opschrift in oostindische ~~de~~ inkt; de gegevens staan er met potlood op geschreven.

Door een opdracht van de vorm

nieuwe grootheid <naam> : <soort>

geef je de uitvoerder opdracht om

a) een nieuwe haart te nemen van een zodanig formaat dat hij van willekeurig object van het type <soort> alle mogelijke gegevens erop zou kunnen schrijven
b) de <naam> er als opschrift op te schrijven - met inkt -
c) de haart in de "geheugen-bak" te zetten.

(I.5) Gebruik je verderop in de algoritme de grootheid <naam>, dan wordt juist die haart gezocht die als opschrift <naam> heeft. (Sommige programmeertalen, ALGOL niet, schrijven voor dat je bij het introduceren van nieuwe grootheden niet alleen het type <soort> moet aangeven, maar ook welke operaties erop mogelijk zijn en wat die operaties doen! In het sorteer-voorbeeld is zoiets, zodaan bij de grootheid reeks, namelijk de veronderstellingen a) en b)).

Één open regel

in commentaarvorm

(I.6) Ad.3. Elementaire opdrachten

Er zijn twee belangrijke mogelijkheden waarin vrijwel alle opdrachten uiteindelijk moeten worden uitgevoerd. Dit zijn de toekenningsopdracht en de test op (on)gelijkheid.

(I.7) a) de toekenningsopdracht heeft de vorm

geef <naam> de waarde van <een uitdrukking>

Dit wordt in vaak geschreven als

<naam> := <een uitdrukking>

en het teken := wordt ~~vaak~~ uitgesproken als "wordt".

Het effect van een dergelijke opdracht kunt u als volgt inzien. Stel u weer voor dat de uitvoerder van de algoritme een kaartenbak voor zich heeft ~~staan~~ met kaarten waarop als opschrift namen staan in oostindische inkt. Bij het uitvoeren van de toekenningsopdracht bepaalt de uitvoerder allereerst de waarde van <een uitdrukking>. ~~Dan~~ Vervolgens zoekt hij de kaart op met opschrift <naam>. Dan gumt hij uit wat er als waarde op de kaart staat genoteerd (als er tenminste al iets op staat). En tenslotte schrijft hij met potlood de zojuist bepaalde nieuwe waarde op de kaart en stopt de kaart weer terug in de bak.

(I.8) Voorbeeld.

Zij rgno het opschrift van een kaart, waarop de waarde 4 -met potlood- staat geschreven. Door uitvoering van de toekenningsopdracht rgno := rgno + 1 vindt het volgende plaats.

- de kaart met naam rgno wordt opgezocht en de waarde wordt gelezen: 4, en de kaart wordt weer teruggezet.
  De waarde van 4+1 wordt bepaald: 5
- De kaart met opschrift rgno wordt opgezocht
- wat erop staat wordt uitgegumd
- 5 wordt er -met potlood- opgeschreven
- de kaart wordt weer teruggezet.

We concluderen dat rgno na afloop de waarde 5 heeft.

(I.9) Omdat de grootheden van waarde kunnen veranderen worden ze ook wel variabelen genoemd. Het nadruk wij u gewezen op het verschil tussen het woordgebruik "variabele" hier en in de wiskunde. Hier heeft een variabele altijd één bepaalde waarde (of misschien nog niet, wanneer hij net is geïntroduceerd). In de wiskunde is een variabele een aanduiding voor een of ander willekeurig element van één bepaalde verzameling.

(I.10) b) de test op (on)gelijkheid heeft de vorm
de waarde van <de ene uitdrukking> is (on)gelijk aan de waarde van <de andere uitdrukking>.
Dit wordt altijd geschreven als
<de ene uitdrukking> = <de andere uitdrukking>, of
<de ene uitdrukking> ≠ <de andere uitdrukking>.

Werken we met getallen en uitdrukkingen die getallen als waarde aannemen, dan kunnen we in de meeste programmeertalen ook het kleiner dan, <, testen en ≤, ≥ etcetera.

De elementaire testen kunnen worden samengesteld door de logische voegwoorden en, of, niet.

(I.11) Natuurlijk geven bovenstaande essentiële kenmerken van algoritmen nog veel vrijheid en zal er in het algemeen het een en ander aangepast moeten worden voordat de uiteindelijke tekst een juist geschreven programma uit de gekozen programmeertaal is. Maar nogmaals, om de uiteindelijke tekst bekommeren we ons bij het ontwerpen niet zo zeer.
EINDE INTERMEZZO.

(1.20) Na bovenstaande beschrijving, zien we dat ons tot nu toe ontwikkeld sorteeralgoritme al aardig voldoet. Duidelijk zijn de behandelde besturingsstructuren, de introductie van

nieuwe grootheden, de toekenningsopdrachten en de voorwaarden te herkennen. Het is niet moeilijk de tekstgedeelten tot één geheel samen te voegen. Dan blijkt dat alleen het "is kleiner dan" en de soorten "rij van objecten", "geheel getal", en de opdracht "schrijf .. op een nieuwe bladrij" nog verder gespecificeerd zouden moeten worden. De rest is bekend verondersteld, dus onderstreept, of zijn namen, dus schuingedrukt.

~~De bedoeling~~ Wat bekend verondersteld mag worden wordt niet door de ontwerper van de algoritme bepaald, maar door ende programmeertaal die gekozen is voor de uiteindelijke tekst (de zojuist beschreven algemene kenmerken van algoritmen

(1.21) Laat ik nu eens ALGOL-60 nemen als programmeertaal. Ik beschouw nog eens de grootheid reeks, zijnde een rij van objecten {met veronderstellingen a) en b)}. Als objecten kies ik getallen. In ALGOL is er een datastructuur voor rij van getallen, die we met rij van getallen zullen aanduiden. Het is gelukkig mogelijk om bij een rij van getallen willekeurig element te bereiken door indicering (veronderstelling a). Een operatie verwissel .. met .. (veronderstelling b) bestaat niet zonder meer in ALGOL. Die moet ik dus verder detailleren. Het volgende stukje tekst is een algemene beschrijving om de gewenste verwisseling tot stand te brengen:

(1.22)
```
begin nieuwe grootheid z : getal {dus van dezelfde
                soort als de el'ten van reeks, bedoeld als
                tijdelijke opslag bij het verwisselen};
    z := y;
    y := x;
    x := z
einde
```

(1.23) Hierbij dient u te weten dat ik de te verwisselen el'ten gegeven veronderstel en dat ik die el'ten x en y heb ge-

noemd. Voor goed begrip zal ik dat vooraf vermelden. Om makkelijk naar deze tekst te kunnen verwijzen geef ik het de naam verwissel. Dus hiermee heb ik de nieuwe grootheid verwissel zijnde een operatie geïntroduceerd:

(1.24)

```
nieuwe grootheid verwissel (x, y) : operatie , namelijk
grootheid x, y : getal { de te verwisselen elementen } ;
toelichting het effect van verwissel (x,y) is dat y
de oorspronkelijke waarde van x heeft , en x die van y ;
begin nieuwe grootheid z : getal {van dezelfde soort als
      x en y , bedoeld als tijdelijke opslag bij 't verwisselen}
      z := y ;
      y := x ;
      x := z
einde
```

(N.B. wat zou het effect zijn als i.p.v. z := y ; y := x ; x := z het volgende zou worden uitgevoerd: y := x ; x := y ?)

(1.25) Met de aanname dat de elementen van reeks getallen zijn is de voorwaarde reeks[..] is kleiner dan reeks[..] gemakkelijk te specificeren. Dit wordt reeks[..] < reeks[..]. Had ik een andere datasoort dan getallen gekozen, dan zou het "is kleiner dan" anders gespecificeerd moeten worden.

~~Voor de soort plaatsnummer kies ik gehuel getal. Dit ligt vooral voor de hand en is in ALGOL als datastructuur~~ ~~domweg~~

(1.26) Zonder de bele ondwikkeling aan te geven, kan ik ook nog de opdracht schuif .. op een nieuwe bladzij , als een operatie specificeren :

```
nieuwe grootheid schrijf (r, e, l) : operatie , namelijk
grootheid r : rij van getallen ;
grootheid e, l : geheel getal ;
toelichting het effect van schrijf (r, e, l) is dat r [e ... l] op
een nieuwe bladrij wordt afgedrukt ;
begin neem een nieuwe bladrij ;
      nieuwe grootheid i : geheel getal { index voor r } ;
      i := e ;
      zolang i < l
         doe print (r [i]) ;
             neem een nieuwe regel ;
             i := i + 1
      herhaaldelijk
einde
```

(1.27) De gehele tot nu toe ontwikkelde algoritme ziet er als volgt uit :

```
(1) nieuwe grootheid sorteer ( reeks , eerste, laatste) : algoritme , namelijk
(2) grootheid reeks : rij van getallen ;
(3) grootheid eerste , laatste : geheel getal ;
(4) toelichting het effect van sorteer (reeks , eerste , laatste) is dat
(5) reeks [ eerste ... laatste ] gesorteerd wordt en zo wel de oorspronke-
(6) lijke alsook de nieuwe rij op een nieuwe bladrij wordt afgedrukt ;
(7)
(8) begin nieuwe grootheid verwissel ( x, y ) : operatie , namelijk
(9)       grootheid x, y : getal ;
(10)      toelichting het effect is dat y de oorspronkelijke waarde
(11)      van x heeft en x die van y ;
(12)      begin nieuwe grootheid z : getal { van dezelfde soort als
(13)            x en y , en bedoeld als tijdelijke opslag voor een ervan } ;
(14)            z := y ; y := x ; x := z
(15)      einde ;
(16)
```

> zie volg blad z>

```
(17)  nieuwe grootheid schrijf (r, e, l) : operatie , namelijk
(18)  grootheid r : rij van getallen ;
(19)  grootheid e, l : geheel getal ;
(20)  toelichting het effect is dat r[e...l] op een nieuwe bladzij wordt afgedrukt;
(21)  begin neem een nieuwe bladzij ; i := e ;
(22)        zolang i < l doe print(r[i:]) ; neem nieuwe regel ; i := i+1 herh'h
(23)  einde ;
(24)
(25)  SORTEER REEKS NAAR GROOTTE :
(26)  nieuwe grootheid rgno : geheel getal {t/m waar de el'ten al goed staan} ;
(27)  rgno := eerste -1 ;
(28)  zolang  rgno < laatste
(29)    doe { de el'ten reeks[eerste ... rgno] staan al goed}
(30)
(31)        ZOEK IN REEKS[RGNO+1 ... LAATSTE] EEN MIN ELT :
(32)        nieuwe grootheid rgmin : geheel getal { rangno van te zoeken min elt }
(33)        nieuwe grootheid h : geheel getal { rangno t/m waar er gezocht is }
(34)        rgmin := rgno +1 ;
(35)        h := rgno +1 ;
(36)        zolang h < laatste
(37)          doe {reeks [rgmin] is minimaal in reeks[rgno+1... h] }
(38)              h := h+1 ;
(39)              als reeks[ h ] < reeks [rgmin] dan rgmin := h
(40)              {reeks[rgmin] is minimaal in reeks[ rgno+1 ... h] }
(41)        herhaaldelijk
(42)        {reeks [rgmin] is minimaal in reeks[ rgno +1 ... laatste ] } ;
(43)
(44)        verwissel ( reeks [rgmin] , reeks[ rgno+1]) ;
(45)
(46)        rgno := rgno +1
(47)        {de elten reeks[ eerste ... rgno ] staan al goed}
(48)  herhaaldelijk
(49)  {de elten reeks[ eerste ... laatste ] staan al goed }
(50)
(51)  schrijf ( reeks , eerste, laatste)
(52) einde .
```

2. VUISTREGELS BIJ HET ONTWERPEN

(2.1) Omdat het onderwijs is een methodiek voor het ontwerpen van algoritmen op enkele pagina's over te brengen, beperken we ons tot enkele vuistregels.
U wordt met klem gevraagd bij het lezen ervan steeds na te gaan waar en hoe deze vuistregels in het sorteervoorbeeld zijn toegepast.

(2.2) ### 1a. stapsgewijze verfijning

Dit is de methode om het probleem, waarvoor een algoritme geschreven moet worden, te analyseren en deelproblemen te onderkennen. Voor die delen kunt u dan afzonderlijk de algoritme verder uitwerken.
← Deze methode geeft u de mogelijkheid met volle aandacht de oplossing te formuleren voor een deelprobleem onafhankelijk van de andere deelproblemen. Hierdoor laat de korrekte werking van de algoritme zich ook makkelijker inzien, en is de algoritme ook begrijpelijker voor anderen.

In het voorbeeld is het probleem 'sorteer reeks naar grootte' ontleed als 'zoek een minimaal elt' en 'verwissel dat van plaats met..'.

Het kan wel gebeuren dat de uitwerking van een deelprobleem om een nadere uitwerking vraagt van een detaillering op een vorig niveau. In het voorbeeld zijn de grootheden eerste en laatste pas geïntroduceerd, nadat daar op het tweede niveau behoefte aan bleek te bestaan.

(2.3) ### 1b. Modulair programmeren

Dit is de methode om een algoritme in op zichzelf staande delen te splitsen die ieder afzonderlijk een gesloten geheel zijn. Wanneer u stapsgewijze verfijning toepast, komt het deels vanzelf tot stand. De modulen, de delen van de algoritme, dient u in ieder geval zo logisch ↓mogelijk (i.p.v. efficient) te

kiezen en voor ieder van de modulen moet u een volledige documentatie en overtuiging van de korrekte werking geven.

(2.4) Operaties, zoals die in het voorbeeld zijn gebruikt, kunnen als modulen worden opgevat. Voor een juiste beschrijving van het effect ervan dient u alle grootheden die "kontakt vormen met de buitenwereld" als parameters aan te geven. Als u dit niet doet, krijgt u borendien moeilijkheden met het aanpassen van de algoritme aan gewijzigde eisen.

Wanneer in operatie schrijf ~~niet~~ alleen e en l tot parameters waren gemaakt, en z niet (maar wel overal in de tekst van schrijf z door reeks was vervangen), dan had u moeilijkheden gekregen wanneer schrijf en sorteer onafhankelijk van elkaar waren ontworpen en in sorteer toevallig een andere naam dan reeks was gekozen voor de te sorteren rij.

(2.5) Wanneer alle grootheden die kontakt vormen met de er omheen liggende tekst tot parameters zijn gemaakt, zijn de modulen veel algemener bruikbaar: zonder enige wijziging kunt u ze ook gebruiken als bouwstenen voor andere algoritmen.

(2.6) ## 2a. Datastructuren kiezen

Als onderdeel van de stapsgewijze specificatie van de algoritme, moeten ook de soorten van de nieuw geintroduceerde grootheden in stappen verfijnd worden. Leg u niet vroegtijdig vast op minder gelukkige, definitieve keuzen!

In het voorbeeld is bijvoorbeeld de grootheid reeks van de soort 'rij van objecten' ~~gegeven~~ geintroduceerd. Uiteindelijk hebben we getallen als objecten gekozen. (Maar even goed had de opdrachtgever halfweg de tijd kunnen voorschrijven dat de objecten zowel een grootte alsook een kleur hadden). Het kan heel wel zijn dat de uiteindelijke programmeertaal verscheidene specificaties toelaat voor grootheden van

het type 'rij van objecten'. Toevallig laat ALGOL^{60} alleen de specificatie array toe, maar andere talen kennen bijvoorbeeld ook list en file. Bij list's en file's kunnen we echter niet willekeurig element door indicering bereiken, maar alleen die welke door zg. wijzers worden aangewezen. En bovendien heeft een file dan nog maar één enkele, beperkte wijzer. Het is dus maar goed dat we ons niet vroegtijdig hebben vastgelegd op files!

(2.7) 2b. Het kiezen van besturingsstructuren

Ook hier geldt: niet te vroeg een keuze maken. In het algemeen zal dit niet gebeuren, zeker niet als u getrapte detaillering nastreeft.

(2.8) Een andere opmerking is wel van belang. Beperk u zelf in uw keuze van besturingsstructuren tot de drie fundamentele: sequentiële samenstelling, voorwaardelijke opdracht of onderscheid naar gevallen en de herhalingsopdracht. In paragraaf 3 wordt een nadere toelichting op deze aanbeveling gegeven.

(2.9) 2c Het kiezen van initialisatie ~~en variantisatie~~

Heel vaak zult u in een algoritme een 'lusstructuur' willen hebben (zie paragraaf 3). Het is van uitermate groot belang dat u zich eerst realiseert wat er in de lus gedaan moet worden ~~en dat uiteindelijk een specificatie~~, vóórdat u zich vastlegt door een niet geschikte en maar al te vaak verkeerd gekozen beginwaarde. Het kiezen van de juiste beginwaarden, de initialisatie, is afhankelijk van de bedoeling van de lus. De initialisatie dient om bij het binnengaan van de lus de tijdens de herhaling invariant blijvende eigenschap te vestigen! Daarom kunt u pas de initialisatie bepalen, nadat u zich gerealiseerd hebt wat de veronderstelde eigenschap a.h. begin van de lus is.

~~(gebrekkig!) niet geneigd dit te doen vóórdat de lus is gespecificeerd.~~

(2.10) 3a Ontwerpbeslissingen expliciet maken

Er is geen enkel probleem dat maar op één manier opgelost zou kunnen worden of dat maar op één manier mooi en goed en logisch opgelost zou kunnen worden. Voortdurend maakt de ontwerper beslissingen om één van de mogelijke voortzettingen te volgen. Omdat de uiteindelijke oplossing met vallen en opstaan wordt gevonden en het dus steeds nodig is terug te komen op een eerder gekozen aanpak, is het heel belangrijk tijdens het ontwerpen dat soort beslissingen heel expliciet te maken en vooral ook de mogelijkheid van alternatieven in te zien en de mogelijke alternatieven te formuleren!

(2.11) Zo had de ontwerper van het sorteervoorbeeld zich ook moeten realiseren dat het zoeken van een minimaal element zowel van links naar rechts, alsook van rechts naar links mogelijk is. Dit heeft wel degelijk invloed op de uiteindelijk opgeleverde gesorteerde rij als de objecten onderling gelijke grootte toch verschillend kunnen zijn, dus bijv. zowel een kleur alsook een afmeting hebben!

(2.12) 3b Zich overtuigen van de korrektheid

Door u zelf steeds weer te overtuigen van de korrekte werking van de algoritme en het programma, bespaart u zich onschatbaar veel kostbare uren. Wanneer u het wel gelóóft, loopt u de kans op het overdoen van al dat ponswerk, al dat wachten op de uitvoer van resultaten, al dat ontdekken en opsporen van fouten, al de pogingen die "eventjes vlug te verbeteren" en dan tóch dat herschrijven van gedeelten van het programma en zelfs het overdoen van het proces van ontwerp vanaf het niveau van

detaillering waarin de fout is gemaakt. Bovendien kan een niet correcte algoritme in sommige gevallen wel de gewenste resultaten opleveren, zodat u werkelijk heel veel moeite zult hebben met het ontdekken en opsporen van fouten aan de hand van de uitvoer van resultaten!

(2.13) Voor een overtuiging van de correctheid is een naleving noodzakelijk van alle in dit hoofdstuk gegeven richtregels!!

(2.14) 4a Documentatie in de tekst formuleren

Het is van groot belang dat er van de algoritme een documentatie is waaruit een ander, de assistent van het praktikum of de gebruiker en u zelf (na jaren) de algoritme kunnt begrijpen. De ideale documentatie is het hele ontwerpproces vast te leggen, maar dit is nogal ondoenlijk.

(2.15) De enige manier om van een algoritme met een grote tekst het gedrag te kunnen begrijpen, is om dit gedrag in te zien op grond van de gedragingen van gedeelten van de tekst. Het is onmogelijk één of meer opdrachten afzonderlijk te beschouwen en toch hun werking in onderlinge samenhang te begrijpen. De manier waarop de mens grote teksten begrijpt, is te abstraheren van gedeelten van de tekst tot welbegrepen gedragingen.

(2.16) In de uiteindelijke versie van de sorteeralgoritme kunnt u het volledige gedrag begrijpen op grond van het feit dat u weet dat het ene tekstgedeelte een minimaal element van de rest vindt en een ander tekstgedeelte van plaats verwisselt zó, dat er weer een element op de juiste plaats staat. En hierbij heeft u geabstraheerd van de feitelijke manier waarop dat minimale element gevonden wordt en hoe die verwisseling plaats vindt!

(2.17) Daarom is de beste manier van documentatie vóór ieder tekstgedeelte als commentaar de veronderstellingen te schrijven waarvan de correcte werking van dat gedeelte van afhankelijk is en ná dát tekstgedeelte de na uitvoering ontstane situatie. Deze twee commentaren geven tesamen het effect of het gedrag aan van dat tekstgedeelte.

Bovendien is het formuleren van de doelstellingen in de tekst (4c) een onontbeerlijk hulpmiddel voor het begrijpen ervan.

(2.18) 4b Correctheidsindicatie in de tekst formuleren

De documentatie is tevens de manier om zelf overtuigd te raken van de correcte werking of om de veronderstellingen te ontdekken die anders stilzwijgend zouden zijn aangenomen.

(2.19) Het kan moeilijk zijn geschikte eigenschappen te formuleren die de correctheid volledig garanderen. Maar wanneer u zich tijdens het ontwerpproces goed realiseert wat u wil doen, lukt het altijd wel een formulering te vinden die een redelijke indicatie is voor de correcte werking (zie ook 4c). Bovendien levert het zoeken naar zo'n formulering vaak een betere, logischer versie op van uw eerste intuitieve aanpak!!

(2.20) Soms kunt u volstaan met het betitelen van een programmadeel waaruit duidelijk blijkt wat er in dat deel gebeurt. Maar een waarschuwing is wel op zijn plaats. Een titel (of label) is veel vrijblijvender dan een correctheidsindicatie. In het voorbeeld is een deel van de algoritme 'ZOEK EEN MINIMAAL ELEMENT' gevonden. Maar hiervan is de correctheidsindicatie {reeks[rgmin] is minimaal element in reeks[rgnoH ... k]} een veel sterkere formulering!

IN REEKS[RGNOH ... LAATSTE

(2.21) **4c Doelstellingen in de tekst formuleren**

Een onmisbaar hulpmiddel voor het begrijpen van de tekst is de doelstellingen te formuleren waarmee nieuwe grootheden worden geïntroduceerd. Wanneer er grootheden worden geïntroduceerd zonder dat u daaraan een zinvolle betekenis kunt toekennen, is de kans erg groot dat u fouten maakt ten gevolge van een onjuist gebruik van die grootheden.

(2.22) De doelstellingen zijn vrijwel altijd betrekkingen tussen de grootheden van de algoritme, waaraan de grootheid voldoet op cruciale punten in de tekst.

Het zijn dan ook juist de bedoelde eigenschappen van de gebruikte grootheden, waaruit de correctheids-indicaties zijn af te leiden: bij ieder tekstgedeelte is de correctheidsindicatie een samenvoeging van de doelstellingen van de daar gebruikte grootheden.

(2.23) Soms hoeft u de bedoelde eigenschappen niet volledig te formuleren, maar kan de naam van de grootheid al voldoende zijn om de bedoeling te begrijpen. Maar denk niet te gauw dat de naam de inhoud dekt.

In het sorteervoorbeeld is de grootheid rgmin geïntroduceerd. Uit de naam leest u al gauw zoiets als "plaats van het minimale element". Dit is nog lang niet wat u begrijpt uit de formulering "rangnr van het te zoeken minimale element". Zeker niet als u zich tevens bewust bent van de bedoeling van h: "rangnr t/m waar er gezocht is naar het minimale element". Een nog betere aanduiding van de betekenis van rgmin zou zijn geweest: "rangnr van het tijdens het zoekproces gevonden tot dan toe minimale element". En deze woordrijke formuleringen worden heel kernachtig weergegeven door op de cruciale punten in de tekst door "reeks[rgmin] is minimaal element in reeks[rgnr+1 .. h]" !!

(2.24) 5 Blokstructuur

Het ontleden van een probleem in zijn ~~logische~~ componenten moet zo overdacht mogelijk gebeuren. Dat wat ~~logisch~~ niet bij elkaar hoort, wordt in afzonderlijke stappen tijdens het ontwerpproces verfijnd. Dat wat ~~logisch~~ bij elkaar hoort, komt in een module. Nieuwe grootheden worden pas geïntroduceerd op het niveau van detaillering waar ze logischerwijze pas bestaan!

(2.25) Een dieper niveau van detaillering wordt in de tekst aangeduid door een verder inspringen naar rechts. Hierdoor komt de zg. blokstructuur van het programma tot uiting duidelijk. Het voorkomt de fouten dat u een grootheid wil gebruiken op een hoger detailleringsniveau dan waar die grootheid is geïntroduceerd. De grootheden "bestaan" slechts in die teksten die eenzelfde of verdere inspringing naar rechts hebben als de plaats van introductie. Niet alleen geldt dit voor hun "logisch bestaan", maar ook voor hun "werkelijk bestaan" voor de uitvoerder. Dit wordt lokaliteit genoemd.

(2.26) Dit laatste kunt u zich als volgt voorstellen. Haal u weer het beeld voor ogen dat in het INTERMEZZO in de vorige paragraaf geschetst is. De uitvoerder van de algoritme heeft als geheugen een kaartenbak met onbeperkt veel blanco kaarten tot zijn beschikking.

- Wanneer hij tijdens het uitvoeren van de algoritme een verder naar rechts inspringend tekstgedeelte bereikt, dan maakt hij vóóraan in de kaartenbak een nieuwe afdeling. De kaarten van grootheden die nu nieuw worden geïntroduceerd, zet hij in de voorste afdeling. Bij het opzoeken van kaarten zoekt hij eerst de nieuwe afdeling af en daarna, in volgorde, de vorige.
  (Als u een nieuwe grootheid introduceert met een al bestaande naam, dan is er voor de uitvoerder

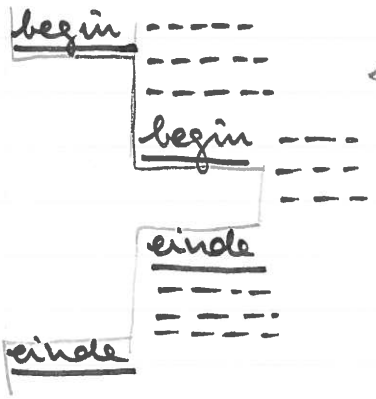
dus géén misverstand mogelijk: hij neemt altijd de jongst mogelijke).

- Wanneer hij tijdens het uitvoeren in een minder ingesprongen tekstgedeelte komt, dan verscheurt hij alle kaarten van de voorste afdeling in zijn geheugenbak. Die grootheden bestaan niet meer!

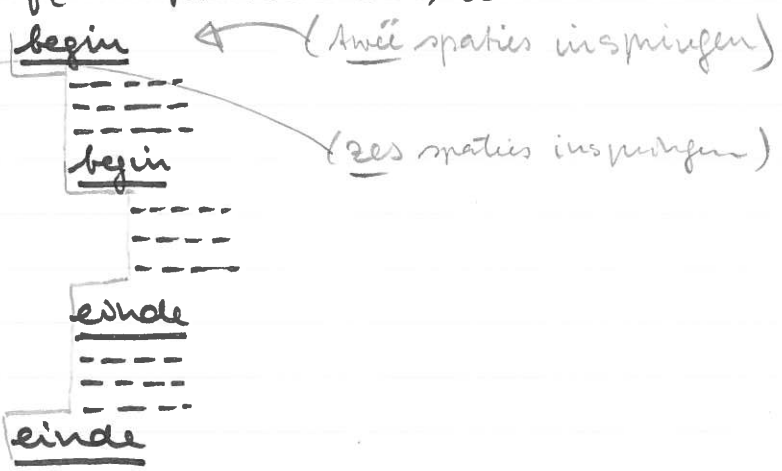
Er moet wel benadrukt worden dat grootheden met eenzelfde naam wél geïntroduceerd mogen worden in verschillende Blokken, maar niet in eenzelfde Blok. Dan raakt de uitvoerder wel in verwarring.

(2.27) In plaats van louter inspringingen te gebruiken in de tekst, geeft men met begin en einde aan waar een nieuw blok van de blokstructuur begint en eindigt. Wij zijn voor gewaarschuwd omdanks het gebruik van de aanduidingen begin en einde echt en juist in de geschreven tekst te blijven in- en terugspringen! Dit houdt een goede aan en is zelfs onmisbaar voor de overzichtelijkheid van het programma.

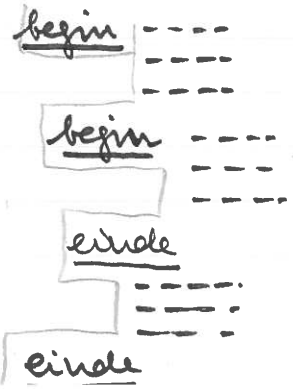
Dus zo:



of (met pijltjeskaarten) zo:



En liever niet zo, omdat het verschrijven onoverzichtelijk is:



(zoals gebeurt in Coll. dict. "Inl. Inf." onderafd. W, T.H.D)

3. OVER BESTURINGSSTRUCTUREN

(Deze paragraaf is alleen bedoeld voor hen die slechte leermeesters zijn, hebben of hebben gehad)

(3.1) De kracht van de electronische rekenmachines ligt in de herhaling. Enerzijds wordt een programma geschreven om vele malen verwerkt te worden; het is geschreven voor een hele klasse van gelijksoortige problemen (alleen de parameters verschillen van waarde). Anderzijds worden binnen een programmaverwerking verscheidene opdrachten meermalen uitgevoerd. Was dit laatste niet het geval, dan zouden we de berekeningen misschien wel beter zelf kunnen uitvoeren in plaats van alle opdrachten (voor een verwerking) voor iedere uitvoering afzonderlijk op te schrijven.

(3.2) Daarom is er behoefte aan een besturingsstructuur die het meermalen uitvoeren van één of meer neergeschreven opdracht mogelijk maakt. De sprongopdracht (Algol: goto) is er zo een. De herhalingsopdracht is er een beperkte versie van.

Hieronder volgt een motivatie waarom nou juist de herhalingsopdracht is aanbevolen.

(3.3) Het aantonen van de korrektheid van een algoritme is iets wat onmogelijk met 100% zekerheid door uittesten of proefdraaien gedaan kan worden. Bovendien kost dat enorm veel tijd en mankracht. Daarom is er behoefte aan een overtuigende berekening van de korrektheid. De redeneervormen die ons daartoe ter beschikking staan zijn de redeneervormen zoals we die tegenkomen in de wiskunde. Dat zijn

- lineair redeneren
- onderscheiden van gevallen
- inductie (volledige inductie, natuurlijke inductie)

Willen we eenzelfde overtuiging hebben van de

korrektheid van een algoritme, als de wiskundigen hebben van de korrektheid van hun "bewijzen", dan hunnen we alleen bovenstaande redeneervormen gebruiken.

(3.4) Welnu, de aanbevolen besturingsstructuren stemmen precies overeen met die redeneervormen, namelijk
- sequentiele samenstelling
- voorwaardelijke opdracht en onderscheiding van gevallen
- herhalingsopdracht.

(3.5) Een aldus geschreven algoritme kan niet alleen gelezen worden als een methode om een oplossing te vinden van het gestelde probleem, maar tevens als een bewijs dat wát er gevonden wordt inderdaad een oplossing is !! Daarom kan de structuur van een algoritme voldoende gevoelsmatige overtuigingskracht geven van de (in)korrekte werking, zonder dat het volledig korrektheids-bewijs is geformaliseerd.

(3.6) Bij het beredeneren, het ontwerpen, van een algoritme zal de ontwerper weinig behoefte blijken te hebben aan andere besturingsstructuren. Om een overtuiging te krijgen van de korrektheid van zijn algoritme zal hij geen andere besturingsstructuur mogen gebruiken.

(3.7) Ten overvloede wijs ik erop dat het nonsens is zonder meer te beweren dat bijv. het sorteerprobleem wezenlijk een "lusstructuur" zou hebben. Integendeel, ieder mens die een rij objecten moet sorteren zal hoogstwaarschijnlijk zonder een "lusstructuur" te werk gaan. En wat dacht u van een muntenscheider die in één de guldens, dwartjes, dubbeltjes, stuivers en centen naar grootte sorteert?

Als we slechts een klein aantal elementaire handelingen mogen verrichten, dan is het wel zo dat er handelingen meermalen moeten worden uitgevoerd.

Ook dan hoeft dit niet in een lusstructuur te gebeuren. Haar ... als IK aan U een serie opdrachten moet geven om met behulp van een klein stel handelingen een rij objecten naar grootte te sorteren, en ik wil overtuigd zijn dat de rij inderdaad naar grootte gesorteerd wordt, dán hoef ik al gauw tot een formulering waarin herhaaldelijk dezelfde bewerking gedaan moet worden.

(3.8) De herhalingsopdracht stemt ook helemaal overeen met de manier waarop wij algoritmen begrijpen. Zoals al gezegd abstraheren wij van een tekstgedeelte en beseffen dan alleen het "netto effect" ervan. Op grond van dat effect kunnen we de eromheen liggende tekst begrijpen en daarvan weer abstraheren. Enzovoort, totdat de hele algoritme begrepen is.

(3.9) Met besturingsstructuren zoals de onbeperkte sprongopdracht heeft een tekstgedeelte waarnaar binnen gesprongen wordt niet een afzonderlijk gedrag: het is in wisselwerking met het gedrag van de tekstgedeelten vanwaar er heen gesprongen wordt. We kunnen slechts van beide tekstgedeelten tegelijkertijd abstraheren en niet eerst van ieder afzonderlijk. Wanneer er veel lus-in-lus gesprongen wordt, kunnen we van geen enkel tekstgedeelte afzonderlijk abstraheren, maar moeten we het gehele programma in één keer begrijpen — of niet!! —.

4. VAN ALGORITME ONTWERP NAAR (ALGOL) PROGRAMMA

(4.1) Deze stap verschilt maar in drie facetten van alle voorgaande stappen in de stapsgewijze ontwikkeling van de algoritme. De mogelijkheden van de taal (ALGOL), de eindige precisie van de computer en efficientieoverwegingen gaan een grotere rol spelen.

(4.2) 1. De mogelijkheden van de taal

Het kan zijn dat de structuur van de algoritme iets gewijzigd moet worden omdat de bedachte constructies niet in de programmeertaal aanwezig zijn. Dit geldt zowel voor (a) de besturingsstructuur als ook (b) voor de datastructuur, en (c) ook andere taaleigenaardigheden kunnen de aandacht vragen.

(4.3) Ad a. Met de vertaling van de gebruikte besturingsstructuur in ALGOL zult u niet veel moeite hebben. De sequentiele samenstelling blijft een punt-komma. De voorwaardelijke opdracht wordt if .. then .. else .. . Het onderscheiden van gevallen kan met een geneste if-then-else constructie worden aangegeven. De herhalingsopdracht kunt u beschrijven met de for statement

for loos := loos while <voorwaarde> do .... .

Hierin is loos een lege variabele, die vanwege de taaleigenaardigheden ~~niet~~ van ALGOL-60 niet mag ontbreken. Wanneer de herhalingsopdracht een voorspelbaar aantal keren wordt doorlopen, kunt u hem beschrijven met

for teller := <beginwaarde> step 1 until <eindwaarde> do ....

(4.4) Ad b De datastructuren zullen in het algemeen meer aan-

> zie volg bladz>

passing behoeven. Voor de specificaties van nieuwe grootheden laat ALGOL alleen de typen real, integer en boolean toe als enkelvoudige structuur en array (van een van de voorgaande typen) als samengestelde structuur. De intuitieve soorten 'gehele getallen', 'reele getallen' en 'waarheidswaarde' komen overeen met de enkelvoudige datastructuren. De (meerdimensionale) 'rij' met array.

(4.5) Matrices, een veel voorkomende soort in algoritmen voor wetenschappelijk rekenwerk, zullen bijvoorbeeld als tweedimensionale array's gespecificeerd moeten worden. Graph en kleur zijn soorten die in combinatorische problemen vaak een opduiken en zij zullen bijv. als array en integer gespecificeerd kunnen worden. In het sorteervoorbeeld hadden de objecten een afmeting en kleur kunnen hebben. Een eenvoudige specificatie van 'rij van objecten' was dan een tweedimensionaal

real array [eerste : laatste, 1 : 2],

zodat de introductie van de nieuwe grootheid reeks zou luiden:

real array reeks [eerste : laatste, 1 : 2]

waarbij reeks [i, 1] als waarde de afmeting v.h. object heeft en reeks [i, 2] als waarde een codering v.d. kleur. Met deze specificatie moeten we natuurlijk wel het 'is kleiner dan' aanpassen, e.d.

(6) Ad c Van de andere facetten noemen we hier de blokstructuur en de parameteroverdracht. Over het al of niet aanwezig zijn van de blokstructuur kunnen we kort zijn: ALGOL heeft 't. En de bijzonderheden van het parametermechanisme zijn zo taal-afhankelijk, dat een uiteenzetting daarover niet thuishoort bij een hoofdstuk over het ontwerpen van algoritmen.

(4.7) 2. De eindige precisie van de computer

Hebben we in een algoritme grootheden die als waarde reele getallen aannemen, dan moeten we ons realiseren dat reele getallen in de computer niet exact te represen-

teren zijn.

(4.8) a. Terwijl in het ideale geval een test op gelijkheid positief zou uitvallen, kan het in werkelijkheid een negatief resultaat geven, omdat er gedurende het rekenproces voortdurend afrondfouten zijn gemaakt. De test op gelijkheid zal bijv. vertaald moeten worden als een test op een absoluut verschil kleiner dan 0,00001.

(4.9) b. Niet alleen heeft de eindige precisie een invloed op de vertaling, ook (het ontwerpen van) de algoritme in zijn geheel zal aangepast moeten worden om een schrikbarende en fatale fouten voortplanting en - aangroeiing te voorkomen. ~~Ten gevolge van een deling door een weinig van mek verschillend getal (zullen de resultaten uiterst onbetrouwbaar worden.~~ Ten gevolge van afrondfouten zullen bij het resultaat bij een aftrekking van twee weinig verschillende getallen uiterst onnauwkeurig worden. Overwegingen van deze aard spelen nauwelijks een rol bij combinatorische problemen (sorteervoorbeeld), maar des te meer bij Numerieke Wiskunde.

(4.10) 3. Efficientieoverwegingen.

Voor de professionele programmeurs die programma's schrijven voor veelvuldig gebruik wegen efficientieoverwegingen heel zwaar. Het lezerspubliek van dit hoofdstuk wordt niet tot de categorie professionele programmeurs gerekend. Natuurlijk moet u met efficientieoverwegingen rekening houden, maar niet tot in het absurde toe! Ook bij het ontwerpproces zelf speelt dit, omdat de algoritme anders inherent inefficient kan worden.

(4.11) De efficientie kan zowel de tijd betreffen die voor de verwerking nodig is, alsook de geheugenruimte die in beslag wordt genomen.

(4.12) a. Het kan zijn dat u "omwille van de efficientie"

sommige grootheden voor verscheidene functies zou willen gebruiken. Dit is een zéér gevaarlijke methode omdat de logische structuur van de algoritme erdoor vorrat aangetast. Het is dan ook onverenigbaar met de vuistregel om de doelstellingen in de tekst te formuleren. Bovendien leveren dit soort gehemsteldheden geen noemenswaardig efficienter algoritme op.

(4.13) Nogmaals, wanneer u van een grootheid geen bedoeling kunt formuleren, dan loopt u grote kans een fout te maken door hem _onbewust_ verkeerd te gebruiken.

(4.14) b. Dat het aanroepen van een arrayelement A[3] meer tijd kost dan het aanroepen van een integer a3, hoeft u zich helemaal niet of pas in allerlaatste instantie te realiseren. En u moet maar heel snel vergeten dat een boolean minder geheugenruimte inn kan nemen dan een integer, ook al neemt die slechts de waarden 0 en 1 aan. En ook de kennis dat het aanroepen van een boolean dan juist meer tijd kan kosten dan het aanroepen van een integer, is niet de moeite van het onthouden waard.

(4.15) Laat uw programma vóór alles leesbaar blijven, daar komt u véél meer mee gebaat.

(4.16) c. Voor herhalingsopdrachten zijn wél zinvolle efficientie overwegingen te maken. U moet vooral geen overbodige dingen herhaaldelijk laten uitvoeren. Zeker bij geneste herhalingsopdrachten moet u proberen constante bewerkingen uit de binnengelegen lussen te halen en ze éénmaal uit te laten voeren (in de lus) erbuiten. *)

(4.17) d. En bij meerdimensionale array's neemt de in beslag genomen geheugenruimte snel toe: een array van 100 by 100 by 100 bezit al 1.000.000 elementen, en dat is heel wat!

*) beschouwingen waardoor een herhalingsopdracht één keer minder vaak doorlopen (zoals in het sorteervoorbeeld: we kunnen zetten "zeno < laatste - 1" in (28)),

(4.18) e. Bij efficientiebeschouwingen probeert u een schatting te geven van de tijd ~~(of het aantal bewerkingen)~~ die nodig is voor de uitvoering van de algoritme. Die tijd drukt u dan uit in het aantal bewerkingen van de verschillende soorten die worden gebruikt. Wanneer u dan de tijd kent van ieder van die bewerkingen – in de betreffende programmeertaal en op de betreffende computer – dan heeft u een redelijke "efficientiemaat" voor de algoritme.

(4.19) In het sorteervoorbeeld wordt de buitenlus $n-1$ keer herhaald, als de waarden van eerste en laatste resp. 1 en n zijn. De binnenlus dus $n-1, n-2, \ldots, 1$ keer. Derhalve zijn er bij uitvoering van de algoritme maximaal:

$n-1$ verwisselingen

$1 + 3\cdot(n-1) + 2\cdot\sum_{i=1}^{n-1}(n-i) = n^2+2n+1$ toekenningen, en

$\begin{cases} 1\cdot\sum_{i=1}^{n-1}(n-i) = n^2-n & \text{arrayvergelijkingen} \\ (n+1)+1\cdot\sum_{i=1}^{n-1}(n-i) = n^2+1 & \text{getalvergelijkingen} \end{cases}$

De overeenkomstige getallen voor het sorteervoorbeeld van het collegedictaat "Inl. Inf" (ouafel. W, T.H.D) luiden:

maximaal:

$1\cdot\sum_{i=1}^{n-1}(n-i) = \frac{1}{2}n(n-1)$ verwisselingen (!)

$2\cdot(n-1) + 2\cdot\sum_{i=1}^{n-1}(n-i) = n^2+n$ toekenningen en

$\begin{cases} 1\cdot\sum_{i=1}^{n-1}(n-i) = n^2-n & \text{arrayvergelijkingen} \\ 2(n-1)+1\cdot\sum_{i=1}^{n-1}(n-i) = n^2-2 & \text{andere vergelijkingen} \end{cases}$

Behalve deze maxima, zijn de "gemiddelden" natuurlijk ook van belang, en de situaties waarin de maxima worden bereikt (in ons voorbeeld bij de rij $2,3,4,\ldots,n,1$ en in het andere voorbeeld bij de rij $n,n-1,n-2,\ldots,1$).

(4.20) Tot slot nog een opmerking over programmastroomschema's.

De techniek van programmastroomschema's kunnt u aanwenden om een overzicht in plaatjesvorm te krijgen van uw uiteindelijke programma. Maar zo'n overzicht is van weinig waarde als niet in beginsel

- ieder niveau van detaillering, en
- de verscheidene logische componenten

te herkennen zijn. als afzonderlijke blokken bijvoorbeeld die met stippellijnen zijn aangegeven en het opschrift hebben van het deelprobleem zoals dat in de stapsgewijze ontwikkeling van uw ontwerp te voorschijn kwam.

(4.21) Door het gebruik van de drie aanbevolen besturingsstructuren, krijgt het programmastroomschema een mooie gestroomlijnde vorm. Het is geheel opgebouwd uit de volgende vormen: