TECHNISCHE HOGESCHOOL EINDHOVEN
Afdeling Algemene Wetenschappen
Onderafdeling der Wiskunde

Herorienteringscursus

"COMPUTERWISKUNDE"

Algorithmen en Rekenmachines

September 1967

COMMISSIE MODERNISERING LEERPLAN WISKUNDE

Herorienteringscursus "Computer Wiskunde"

Algorithmen en rekenmachine

te houden van:

11 tot en met 15 september 1967
8 tot en met 12 januari 1968

aan de:

Technische Hogeschool te Eindhoven
Universiteit te Utrecht en te Amsterdam

INHOUD

Hoofdstuk 1. ALGORITHMEN

§ 0. Gebruik van de computer in vogelvlucht 1
§ 1. Algorithmische taal 2
§ 2. Eerste indruk van de taal 3
§ 3. De variabele 5
§ 4. Getalband 5
§ 5. Grootste getal op band 7
§ 6. Som van rij getallen 9
§ 7. Geïndiceerde variabelen 10
§ 8. Overzicht van ALGOL 11
§ 9. Relatie tot ALGOL 60 13
§ 10. Waarom "stored programme" 13
§ 11. Procedures 14
§ 12. Economie 15
§ 13. Doelstelling van de rest van dit hoofdstuk 16
§ 14. Het beschouwde probleem 16
§ 15. Bepalen of getal in rij voorkomt 17
§ 16. Getallen tussenvoegen in rij 18
§ 17. Sorteren 19
§ 18. Het oorspronkelijke probleem 22

Hoofdstuk 2. NUMERIEKE TOEPASSINGEN

§ 0. Inleiding 23
§ 1. Eindige precisie rekenapparatuur 23
§ 2. Vierkantsvergelijking 25
§ 3. Nauwkeurigheid uitkomst bij onnauwkeurige operanden 26
§ 4. Vierkantsvergelijking II 26
§ 5. Recursie 27
§ 6. Onzekerheidsinterval voor wortels 29
§ 7. Wortels van vergelijkingen door successieve halvering 30
§ 8. Wortels van vergelijkingen met Newton 30
§ 9. Iteratieve processen in het algemeen 33
§ 10. Stopcriterium 34
§ 11. Successieve halvering vergeleken met Newton 36
§ 12. Numerieke kwadratuur 37
§ 13. Trapeziumregel 38
§ 14. Simpson 39
§ 15. Glijdende stap 40
§ 16. Algorithme 42
§ 17. Slotopmerkingen 44

Hoofdstuk 3. NON-NUMERIEKE COMPUTER TOEPASSINGEN

§ 1. Backtracking 46
§ 2. Een klasse van tweepersoonsspelen 51
§ 3. Kortste boom tussen N punten 53

Hoofdstuk 4. OVER DE STRUCTUUR VAN REKENAUTOMATEN

§ 1. Inleiding 58
§ 2. Globaal overzicht van de belangrijkste componenten 58
§ 3. De sequentialisering 61
§ 4. Programmanotatie in 1-adrescode 64
§ 5. De opdrachtcyclus 68
§ 6. Over de functie van een vertaler 69

Enkele karakteristieken van computers anno 1967 73

Hoofdstuk 5. VRAAGSTUKKEN 74 - 84

1. Algorithmen

0. Gebruik van de computer in vogelvlucht

In vogelvlucht schetsen we de entourage van het gebruiken van een computer bij het oplossen van een probleem. Hierbij worden de volgende fasen onderscheiden.

0.1. Men bedenkt een oplossing van het probleem in termen van eindig veel operaties uit het operatie repertoire van de computer. Daartoe moet men dit repertoire kennen. Als men bijv. wenst wortel te trekken, en de machine kan niet worteltrekken, dan zal men het worteltrekken moeten ontleden in operaties die de machine wel kent. Deze oplossing noemt men de algorithme.

0.2. Men vervaardigt voor de in 0.1 bedoelde algorithme een geschrift dat in voor de machine begrijpelijke taal de uit te voeren operaties precies aangeeft. Zo'n geschrift heet een programma.

0.3. Men zet het onder 0.2 bedoelde geschrift om in voor de computer leesbaar schrift. De computer kan nl. nog geen handschrift en nauwelijks schrijfmachineschrift lezen. Dit omzetten geschiedt op schrijfmachineachtige apparaten, die de ingetypte tekst door gaatjes in ponsband of ponskaarten vastleggen (en daarnaast doorgaans gelijktijdig ook nog eens in gewoon schrijfmachineschrift op een gewoon vel papier afdrukken, zulks ten gerieve van de pons-typist of diens opdrachtgever). Dit noemt men het verponsen van het programma.

0.4. Men geeft ponsband of -kaarten af aan het bedienend personeel van de computer. Dit plaatst band of kaarten in het zgn. leesstation van de computer, waar ze door rollers worden voortbewogen langs fotocellen of contactborstels. Hierdoor resulteren de gaatjes in electrische impulsen en neemt de computer kennis van de geponste tekst. Hij gaat echter niet elke operatie onmiddellijk na lezing uitvoeren, maar slaat eerst het hele programma in zijn "geheugen" op (doorgaans op magnetische wijze). Pas als het hele programma in het geheugen staat wordt tot uitvoering overgegaan. Men spreekt daarom van een "stored programme computer".

0.5. Het programma wordt uitgevoerd. Alle operaties worden in de door het programma bepaalde volgorde uitgevoerd (dit is niet noodzakelijk de volgorde waarin ze staan opgeschreven). Ofschoon we pas later op het operatierepertoire van de computer ingaan, zij reeds nu vermeld dat men daarin, naast de vanzelfsprekende rekenoperaties, ook operaties aantreft voor communicatie met de buitenwereld. Aangezien men nl. niet van elke afzonderlijke optelling, aftrekking

etc. de uitkomst zal wensen te vernemen, moet men in het programma aangeven welke uitkomsten dan wel moeten worden meegedeeld, en deze worden dan afgedrukt op een aan de computer gekoppelde "schrijfmachine". Ook kan men aangeven dat, alvorens de verwerking van het programma wordt voortgezet, nieuwe informatie over het leesstation moet worden ingevoerd.

1. Algorithmische taal

Zodra men een, al dan niet menselijke, instantie wil uiteenzetten hoe een bepaald werk moet worden gedaan, dient men over een daartoe geschikt expressiemedium, een taal, zo men wil, te beschikken, waarin men precies kan uitdrukken wat men gedaan wil hebben. De natuurlijke taal schiet hiervoor, naar ieder vrijwel dagelijks ervaart, hopeloos te kort ("ik had je nog zo duidelijk gezegd wat je moest doen").

Wil zo'n expressiemedium aan zijn doel beantwoorden dan zal het zich moeten bedienen van een welomlijnd repertoire van welbegrepen handelingen en welbegrepen afspraken betreffende de volgorde waarin de handelingen uitgevoerd dienen te worden.

De makers van breipatronen hebben zo'n expressiemedium inderdaad gevonden (met als welbegrepen handelingen bijv. één recht en twee averecht); men vergelijke hiermee eens de montage voorschriften voor zelf te monteren huisraad.

Voor het formuleren van rekenwerk zijn ettelijke talen ontwikkeld. Een daarvan is ALGOL 60 (afkorting van algorithmic language 1960). In deze cursus zullen we ons van een vereenvoudigde versie hiervan bedienen die we kortweg met ALGOL zullen aanduiden. In ALGOL 60 gestelde teksten kunnen zonder meer als de in 0.2 bedoelde programma's dienen, waarmede men elke computer voor zich kan laten werken die ALGOL 60 verstaat (zie ook hoofdstuk 4). De taal is echter evenzeer ontworpen als communicatiemiddel betreffende rekenwerk tussen mensen. In verband met internationale samenwerking ter zake leunt de taal wat tegen het engels aan.

Men kan ALGOL 60 en ook ons ALGOL zeer formeel definiëren. Liever zullen we hier echter volgens de "natuurmethode" te werk gaan, d.w.z. meteen volledige teksten in deze taal presenteren en dan zeggen wat ze betekenen.

De hierboven genoemde welbegrepen handelingen zullen we opdrachten noemen. Een algorithme bestaat uit opdrachten. Wanneer men deze in de aangegeven volgorde uitvoert, volvoert men het proces dat door de algorithme wordt beschreven.

2. Eerste indruk van de taal

Aan de hand van een voorbeeld zullen we nu een indruk geven van ALGOL 60. Het voorbeeld zal uiteraard geen diepzinnige algorithme betreffen, evenmin als doorgaans met de eerste zin, die iemand ooit in een vreemde taal uitspreekt, een grootse gedachte tot uitdrukking wordt gebracht. Het voorbeeld moet dan ook niet gezien worden als exemplarisch voor computergebruik. Het heeft slechts ten doel een deel van het handelingenrepertoire te laten zien waarvan men zich moet bedienen; pas als we dat repertoire kennen zullen we zinrijker algorithmen beschouwen.

We wensen van de volgende drie uitdrukkingen de waarden te bepalen voor gegeven positieve waarden van de erin optredende variabelen a, b en c:

$$\frac{b^2+c^2-a^2}{2bc}\ ,\quad \frac{a^2+c^2-b^2}{2ac}\ ,\quad \frac{a^2+b^2-c^2}{2ab} \qquad (2.1)$$

(de gevraagde waarden stellen toevallig de cosinussen der hoeken van een driehoek voor, wanneer de lengten der zijden van deze driehoek voorgesteld worden door a, b en c).

We zullen aannemen dat de lezer (mens of machine) van onze algorithme in staat is de waarden van dergelijke uitdrukkingen te bepalen bij gegeven waarden van a, b en c. Wij zullen echter niet op prijs stellen dat hij dat in dit geval ook zonder meer doet. Immers dan zou hij de waarden van a^2, b^2 en c^2 elk driemaal berekenen, en dat zouden wij niet doen. Onze eigen rekenwijze zou op het volgende neerkomen. We berekenen de waarde van a^2 en kennen deze waarde toe aan een nieuwe variabele, die we bijv. akw noemen. We berekenen vervolgens de waarde van b^2 en kennen die toe aan bkw. Tenslotte berekenen we de waarde van c^2 en kennen die toe aan ckw. De uitdrukkingen in (2.1) vervangen we nu door

$$\frac{bkw+ckw-akw}{2\times b\times c}\ ,\quad \frac{akw+ckw-bkw}{2\times a\times c}\ ,\quad \frac{akw+bkw-ckw}{2\times a\times b} \qquad (2.2)$$

en bepalen de waarden hiervan op de gebruikelijke wijze.

In onze versie van ALGOL 60 zullen we de betreffende algorithme aangeven door de in (2.3) omraamde tekst, waarbij we aan a, b en c de waarden 7.1, 3.15 resp. 4.28 zullen laten geven, en de resultaten op een schrijfmachine laten afdrukken.

```
begin a := 7.1;   b := 3.15;  c := 4.28;                    (1)
      akw := a↑2; bkw := b↑2; ckw := c↑2;                   (2)
      res1 := (bkw + ckw - akw) / (2 × b × c);              (3)
      res2 := (akw + ckw - bkw) / (2 × a × c);              (4)        (2.3)
      res3 := (akw + bkw - ckw) / (2 × a × b);              (5)
      PRINT(res1); PRINT(res2); PRINT(res3)                 (6)
end                                                         (7)
```

Toelichting. De achtereenvolgens uit te voeren opdrachten zijn van elkaar gescheiden door een puntkomma. De opdrachten worden in principe uitgevoerd in de volgorde waarin ze in de tekst voorkomen. De onderstreepte woorden begin en end dienen hier slechts om de tekst te begrenzen. We bespreken nu regel voor regel hoe het door deze algorithme beschreven proces verloopt.

(1) De eerst uit te voeren opdracht is blijkens het zojuist gezegde "a := 7.1", uit te spreken als "a wordt 7.1", en te interpreteren als: "ken aan de variabele a de waarde 7.1 toe". Analoog de tweede en derde opdracht: "b := 3.15" resp. "c := 4.28".

(2) Daarna komt de opdracht "akw := a↑2" : het kwadraat van de waarde van a (dus van 7.1) moet worden berekend (uitkomst 50.41) en aan de variabele akw toegekend. De machtsverheffing wordt hier dus met de aparte operator ↑ aangeduid, in onderscheid met de in de wiskunde gangbare conventie, volgens welke men de exponent op enige afstand boven een denkbeeldige lijn schrijft. Niet alleen voorkomt men met de door ons gevolgde schrijfwijze onduidelijkheden, ook is het bij het verponsen van een tekst (zie 0.3) heel prettig als deze lineair is, d.w.z. bestaand uit een rij symbolen die alle op hetzelfde niveau staan. Analoog de twee volgende opdrachten op regel (2).

(3) De waarde van de uitdrukking "(bkw + ckw - akw) / (2 × b × c)" wordt berekend, met gebruikmaking van de waarden die de erin voorkomende variabelen zojuist hebben gekregen, en deze waarde moet worden toegekend aan de variabele res1 (afkorting voor resultaat 1). Ook deze uitdrukking is weer "lineair" geschreven.
(4), (5) analoog aan (3).

(6) De waarden van de variabelen res1, res2 en res3 worden afgedrukt op de schrijfmachine.

(7) Tenslotte arriveert men bij end en concludeert dat het proces is voltooid.

3. De variabele

Blijkens het voorafgaande dient de variabele alleen om naar een (getal-)waarde te verwijzen, niet naar de wijze waarop deze tot stand is gekomen. Men zou zich dan ook kunnen voorstellen dat er a.h.w. een kaartsysteem wordt aangelegd, voor elke variabele een kaart, en dat men het toekennen van een waarde aan een variabele interpreteert als het schrijven van een getal op de betreffende kaart. Telkens als men in een uit te voeren berekening een variabele tegen komt, leest men op dat moment zijn waarde van de kaart af. Derhalve heeft na het uitvoeren van de opdrachten

```
a := 7;   akw := a↑2;   a := 8;
```

de variabele akw nog steeds de waarde 49.
Welke naam men voor een variabele kiest is voor de algorithme volstrekt irrelevant. Als men in (2.3) overal "akw" vervangt door bijv. "piet" verandert er aan het beschreven rekenproces niets. Analoog bijv. res2. Het bevordert echter wel de "leesbaarheid" van de algorithme voor de menselijke lezer wanneer men in de naam van de variabele iets van de herkomst van zijn waarde tot uitdrukking brengt. Zo verwijst akw duidelijk naar akwadraat.
Zodra men voor variabelen meerletterige namen toelaat is het duidelijk niet meer gewenst de in de algebra geldende conventie te handhaven dat men in producten de maaltekens weglaat. In (2.3), (3) t/m (5), is dan ook met deze conventie gebroken.

4. Getalband

In de algorithme (2.3) dragen de getallen 7.1, 3.15 en 4.28 een zeer incidenteel karakter. Wanneer men (2.1) voor een aantal verschillende tripels waarden voor a, b, c wenst te laten evalueren dient men (2.3) even zo vele malen op te schrijven met alleen (2.3) (1) enigszins gewijzigd.
Het is daarom handig aan te nemen dat men aan de algorithme nog een rij getallen kan hechten, en dat men in de algorithme kan spreken over "het volgende getal uit deze rij". Deze rij getallen noemen wij gewoonlijk "de getalband", omdat men bij computergebruik deze getallen vaak achter elkaar in een ponsband ponst.
In de volgende algorithme wordt hiervan gebruik gemaakt.

```
begin                                                          (1)
weer:  a := READ;     if a = 0 then goto klaar;                (2)
       b := READ;     c := READ;                               (3)
       akw := a↑2;    bkw := b↑2;    ckw := c↑2;               (4)
       res1  := (bkw + ckw - akw) / (2 × b × c);               (5)
       res2  := (akw + ckw - bkw) / (2 × a × c);               (6)
       res3  := (akw + bkw - ckw) / (2 × a × b);               (7)
       PRINT(res1);  PRINT(res2);  PRINT(res3);                (8)
       goto weer;                                              (9)
klaar: end                                                     (10)
```

(4.1)

Zij de bijgevoegde getalband als volgt: + 7.1 + 3.15 + 4.28 + 7.23 + 4.12 + 5.82 + 0
Dan verloopt het proces als volgt:

(2) De aanduiding "weer" zal als straatnaambordje blijken te fungeren, maar resulteert niet in een handeling. De eerste opdracht is dus "a := READ", die als effect heeft dat het eerstvolgende getal van de getalband aan a als waarde wordt toegekend. Dus a krijgt als waarde 7.1. Dit is het invoeren van nieuwe informatie over het leesstation dat we in 0.5 noemden.

De volgende opdracht luidt "if a = 0 then goto klaar". Als a de waarde 0 zou hebben zou de uitvoering van de opdrachten in tekstvolgorde doorbroken worden, en zou de uitvoering van de algorithme hervat worden daar waar de "label" (ofwel het straatnaambordje) "klaar" staat; aangezien deze label bij end staat zou hiermee het proces geëindigd zijn. Als a evenwel niet de waarde 0 heeft (en dat is hier het geval) wordt "goto klaar" niet uitgevoerd, en komt gewoon de in de tekst volgende opdracht aan de beurt dus:

(3) "b := READ", dus b krijgt als waarde 3.15 en vervolgens krijgt c als waarde 4.28.

(4) t/m (8) analoog aan (2.3), (2) t/m (6).

(9) De opdracht "goto weer" heeft nu geen conditie bij zich van het soort if ... then, en wordt zonder meer uitgevoerd. D.w.z. de algorithme wordt hervat bij de label "weer", dus met de opdracht "a := READ". Nu krijgt a de waarde 7.23; dit is weer niet 0, zodat vervolgens b de waarde 4.12 en c de waarde 5.82 krijgt. Er wordt weer gerekend en afgedrukt; "goto weer" wordt weer actief. Daarna echter wordt het getal 0 gelezen, zodat "goto klaar" uitgevoerd wordt. Daarmee is dan het proces geëindigd.

Hieruit is ook duidelijk, hoe men de getalband voor dit programma moet inrichten. Men kan er willekeurig veel drietallen getallen op zetten, met achter het laatste drietal een 0 als teken dat men alle drietallen gehad heeft. Inmiddels impliceert dit, dat geen der drietallen als eerste getal 0 mag hebben. Dit is echter nauwelijks een beperking, aangezien de berekening zinloos wordt zodra a, b of c de waarde 0 heeft.

Een label is een vrij te kiezen naam (zie 3), waarvan de keuze voor de algoritme niet, maar voor de menselijke lezer wel van belang is. Om der wille van deze leesbaarheid schrijven we een label ook steeds naar links uitspringend.

5. Grootste getal op band

Nu we aan de hand van algorithmisch bezien niet zo interessante voorbeelden een indruk hebben gekregen van het opdrachtenrepertoire van de taal, gaan we een algorithme beschouwen die op deze naam meer aanspraak kan maken. Het betreft het bepalen van het grootste van een eindige rij getallen. Het is een algorithme die, zeker als deel van een grotere algorithme, zeer veel gebruikt wordt.

Als men een algorithme moet opstellen ter oplossing van een probleem is het vaak nuttig eens bij zichzelf te rade te gaan hoe men het betreffende probleem nu eigenlijk zelf zou oplossen. Dit is lang niet altijd eenvoudig omdat men na langdurige ervaring allerlei probleempjes a.h.w. automatisch oplost zonder dat men zich realiseert wat men nu precies doet.

In het onderhavige geval zal de uitslag van het beraad wel ongeveer als volgt luiden. We nemen het eerste getal van de rij in gedachten en constateren dat dit getal groter is dan of gelijk is aan het tweede getal van de rij, het derde, etc. tot we een getal vinden waarvoor dit niet meer het geval is. Dan nemen we dat getal in gedachten etc. Enige nadere reflectie leert nog dat het veiliger is niet al te zeer op het eigen geheugen te vertrouwen en het te onthouden getal maar liever op te schrijven.

Volgens dit principe werkt ook de volgende algorithme. Deze bepaalt van een getalband, waarop tenminste één negatief getal staat, het grootste van de getallen die aan het eerste negatieve getal voorafgaan. Gegeven is nog dat het eerste getal op de band niet negatief is.

```
begin   max := READ;                        (1)
lees:   term := READ;                       (2)
        if term < 0 then goto klaar;        (3)
        if term ≤ max then goto lees;       (4)
        max := term; goto lees;             (5)
klaar:  PRINT(max)                          (6)
end                                         (7)
```

(5.1)

Toelichting

(1) max zal als waarde steeds het grootste tot dusverre gelezen getal hebben. Na het eerste gelezen getal is dat dus dit getal.

(2) "lees" is weer een label. "term := READ" kent het tweede getal van de band aan term toe.

(3) Mocht dat tweede getal al negatief zijn, dan wordt volgens afspraak het grootste zoeken gestaakt, en naar regel (6) gegaan, waar de waarde van max, zijnde het eerste getal van de band, wordt afgedrukt. Als het tweede getal echter niet negatief is wordt doorgegaan op regel (4).

(4) Het gekozen getal wordt met het tot dusverre grootste vergeleken. Als het gelezen getal niet groter is wordt verder gegaan op regel (2); als het wel groter is wordt op regel (5) dit pas gelezen getal aan max toegekend, zijnde het tot dusverre grootste getal.

Opgave 1. Ga na wat er gebeurt als men, in strijd met het gegeven, nu toch eens een band aanbiedt waarop het eerste getal negatief is:

a) voor het geval er verder geen enkel getal meer op de band staat;

b) voor het geval er verder geen negatief getal meer op de band staat;

c) voor het geval het tweede getal negatief is;

d) in de overige situaties.

Opgave 2. Wat voor effect heeft het toevoegen aan het eind van regel (1) van de opdracht "if max < 0 then goto klaar;" in de situaties als bedoeld in opgave 1.

Opgave 3. Ga na of het in opgave 2 bedoelde effect hetzelfde is als wanneer men (1), (2) en (3) vervangt door:

```
begin   max := READ; term := max;
lees:   if term < 0 then goto klaar;
        term := READ;
```

6. Som van rij getallen

Een eveneens zeer veelvuldig gebruikte algorithme is die ter bepaling van de som van een eindige rij getallen. We doen dit "zelf" als volgt: we nemen het eerste getal van de rij in gedachten, tellen daar het tweede bij en onthouden dit resultaat, tellen daar het derde bij etc.

Evenzo werkt de volgende algorithme. Op een getalband zal het eerste getal aangeven hoeveel der erop volgende getallen gesommeerd moeten worden.

```
begin   n := READ; i := 1; som := READ;          (1)
herh:   if i = n then goto klaar;                (2)
        som := som + READ;                       (3)
        i := i + 1; goto herh;                   (4)
klaar:  PRINT(som)                               (5)
end                                              (6)
```

(6.1)

Een variant hierop is de volgende algorithme, die van (6.1) slechts in de eerste regel verschilt, en waarvan we daarom alleen de eerste regel geheel geven

```
begin   n := READ; i := 0; som := 0;
etc
```

(6.2)

Deze algorithme komt er in feite op neer dat we eerst het getal 0 in gedachten nemen, en daar vervolgens elk getal van de rij bij optellen.

Opmerking. Het bandlezen wordt hier gestaakt op grond van een parameter die als eerste op de band wordt meegegeven. In 5. gebeurde dit op grond van het aantreffen van een negatief getal op de band. Dit verschil heeft niets te maken met het feit dat er in 5. het grootste moest worden gezocht en in 6. gesommeerd moest worden. We wilden alleen laten zien dat men op verschillende wijzen kan aangeven wanneer het bandlezen gestaakt moet worden.

Opgave 1. Ga na voor welke getalbanden (6.1) en (6.2) een verschillende reactie geven. Welke algorithme is dus algemener?

Opgave 2. Schrijf een algorithme teneinde van de bovengenoemde getalband het grootste te bepalen van het 2^e t/m n^e getal, waarbij n het eerste getal op de band is.

Opgave 3. Schrijf een algorithme die voor de in 5. bedoelde getalband de som bepaalt van de getallen die aan het eerste negatieve getal voorafgaan.

Opgave 4. Schrijf een algorithme die voor een van deze getalbanden niet alleen het grootste maar ook het kleinste der bedoelde getallen bepaalt.

7. Geïndiceerde variabelen

De voorafgaande algorithmen hadden de eigenschap dat ze telkens, na een reeds ten tijde van het schrijven van de algorithme bekend (en bovendien nog klein, nl. 3 of 1) aantal getallen van de band gelezen te hebben, deze meteen konden verwerken en daarna vergeten. Dit vergeten vond plaats doordat men aan de variabelen, waaraan de gelezen getallen als waarde waren toegekend, een nieuwe waarde toekende. Bij het nu volgende probleem verkeren we niet in die situatie.

Op een getalband geeft het eerste getal aan hoeveel getallen er op die band nog volgen. Gevraagd te bepalen, hoeveel verschillende er onder die volgende getallen zijn.

Er zit nu weinig anders op dan tijdens het lezen van de achtereenvolgende getallen van de band alle verschillende tot dan toe gelezen getallen te bewaren. Het natuurlijke hulpmiddel hiervoor is de variabele met index. We zullen de achtereenvolgens gevonden verschillende getallen in deze volgorde toekennen aan de variabelen a_1, a_2, a_3, etc. Als op een zeker moment a_1, ..., a_v de tot dan toe gevonden verschillende getallen zijn, zullen we een hierna te lezen getal achtereenvolgens vergelijken met a_1, a_2, ..., a_v, en als het van al deze verschilt, het als a_{v+1} toevoegen aan de rij.

```
begin   n := READ; a[1] := READ; g := 1; v := 1;          (1)
lees:   if g = n then goto klaar;                         (2)
        term := READ; g := g + 1;                         (3)
        j := 1;                                           (4)
test:   if term = a[j] then goto lees;                    (5)
        j := j + 1; if j ≤ v then goto test;              (6)
        v := v + 1; a[v] := term; goto lees;              (7)
klaar:  PRINT(v)                                          (8)
end                                                       (9)
```

(7.1)

Toelichting

(1) Om dezelfde reden, waarom men bij machtsverheffing de exponent niet op hoger niveau wil schrijven, willen we indices niet verlaagd aangeven. Vandaar dat we i.p.v. a_1 schrijven a[1].

De variabele n geeft het aantal te lezen getallen, g het aantal reeds gelezen getallen, v het aantal verschillende daaronder. Het eerste getal wordt aan a[1] toegekend. Het aantal gelezen getallen en het aantal verschillende daaronder worden beide 1 gesteld.

(2) Als het aantal gelezen getallen gelijk is aan het aantal te lezen getallen zijn we klaar.

(3) Na elk gelezen getal wordt de waarde van g met 1 verhoogd.

(4) Voorbereiding voor het vergelijken van term met a[j] voor j = 1 t/m v.

(5) Zodra de waarde van term blijkt samen te vallen met die van een der a[j], wordt het vergelijken gestaakt en wordt verder gegaan op (2).

(7) Op deze regel komt men slechts als het zojuist gelezen getal aan geen der waarden der a[j], dus aan geen der vorige getallen, gelijk was. Dus wordt de waarde van v (het aantal der tot dusverre verschillende) met 1 verhoogd, en wordt het zojuist gelezen getal als waarde toegekend aan a[v] (waarin v uiteraard de zojuist met 1 vermeerderde waarde heeft).

Inmiddels hebben we hier al een vrij ingewikkelde algorithme gegeven. Deze ingewikkeldheid komt voornamelijk daardoor, dat er binnen de grote algorithme op de regels (4), (5) en (6) een vrij zelfstandige kleinere algorithme voorkomt, nl. om te bepalen of een gegeven getal gelijk is aan één uit een gegeven rij getallen. Men kan zich het opstellen van algorithmen vaak zeer vergemakkelijken door na te gaan of men in het uit te voeren proces dergelijke deelalgorithmen kan onderkennen, en te trachten het hele proces als aaneenschakeling van dergelijke deelalgorithmen te zien. We komen op deze zaak nog terug.

8. Overzicht van ALGOL

Zoals reeds eerder is opgemerkt willen we hier geen formele definitie van ALGOL geven. Nu in het voorafgaande vrijwel alles aan de orde is gekomen wat we in deze cursus van ALGOL willen laten zien, willen we echter wel een overzicht geven van de taal en daarmee van het ten dienste staande handelingenrepertoire.

8.1. Een programma is een opeenvolging van opdrachten, van elkaar gescheiden door puntkomma's, en geplaatst tussen begin en end.

8.2. De opdrachten zijn de volgende:

8.2.1. variabele := arithmetische uitdrukking,

8.2.2. goto label,

8.2.3. if conditie then goto label,

8.2.4. PRINT(arithmetische uitdrukking).

8.3. Een arithmetische uitdrukking is een opeenvolging van getallen, variabelen met getalwaarde, haakjes, de operatoren $\uparrow \times / + -$ en van de uitdrukkingen READ, abs(x), sqrt(x), sin(x), cos(x), tan(x), exp(x), ln(x), arcsin(x), arccos(x), arctan(x), sign(x), entier(x). Hierin stelt x steeds een arithmetische uitdrukking voor. Voor zover de namen der functies niet vanzelfspreken: abs(x) = $|x|$; sqrt(x) = $\sqrt{x}$; exp(x) = e^x; sign(x) = 1 voor $x > 0$, 0 voor $x = 0$, - 1 voor $x < 0$; entier(x) is het grootste gehele getal $\leqslant x$.

8.4. Een conditie heeft de gedaante $x < y$, $x \leqslant y$, $x = y$, $x \neq y$, $x \geqslant y$, $x > y$, waarbij x en y steeds arithmetische uitdrukkingen zijn. Ingewikkelder condities kan men uit de voorafgaande maken m.b.v. de logische operatoren $\neg$ (niet) $\wedge$ (en) $\vee$ (of), alsmede van haakjes.

Voorbeeld: $x + 5 < y \times z \wedge (a < b \vee b < c)$.

8.5. Een variabele is een naam, of een naam met tussen vierkante haken een of meer (in dit laatste geval door komma's gescheiden) indices. Deze indices zijn weer arithmetische uitdrukkingen. Voorbeeld: a[i + 1, 2 × j].

8.6. Een label is een naam.

8.7. Een naam is een rijtje letters en evtl. cijfers, beginnend met een letter. Voorbeeld: a17q.

8.8. Een getal heeft de gebruikelijke schrijfwijzen, echter met decimaalpunt i.p.v. komma: 17; 17.38; we staan echter nog aanhechting van een schaalfactor toe: $17.38_{10} - 5$.

8.9. We releveren nog hoe de logische operatoren gedefinieerd zijn:
$\neg$ a is waar als a niet waar is en andersom,
a $\wedge$ b is waar ddan als a en b beide waar zijn,
a $\vee$ b is niet waar ddan als a en b beide niet waar zijn.
De operatoren hebben in deze volgorde prioriteit over elkaar.

Opmerking 1. De gewone spreektaal is in feite al even ongeschikt voor het geven van scherpe definities als voor het precies formuleren van algorithmen. Men is dan ook in de wiskunde en i.h.b. in de logica al geruime tijd geleden overgegaan tot het formaliseren van definities. Om dezelfde reden is ALGOL 60 in de oorspronkelijke publicatie ervan op veel formeler wijze gedefinieerd dan wij dat hier deden.

Opmerking 2. Zoals we reeds eerder opmerkten geeft de in de wiskunde gangbare notatie van exponenten en indices, alsmede het weglaten van maaltekens, kans op misverstanden, zeker wanneer men de formules bijv. met de hand of de schrijfmachine op papier zet; een simpele uitdrukking als $m_2 n$ illustreert dit. Vele wiskundige formules vereisen dan ook naast een goede drukker een goede (= welwillende) verstaander. Desondanks zullen we ons in deze syllabus veroorloven om der wille van prettige leesbaarheid indices toch maar op de in de wiskunde gebruikelijke wijze te schrijven, voor zover zij althans een eenvoudige gedaante hebben.

9. Relatie tot ALGOL 60

Door aan het begin van een in ons ALGOL gesteld programma een verklaring toe te voegen van de namen die erin optreden wordt het een programma in ALGOL 60 , en daarmee uitvoerbaar op een computer die ALGOL 60 verstaat.

Voorts hebben we enkele faciliteiten weggelaten die ALGOL 60 biedt, en dus de omvang van de taal verkleind. Zo kent ALGOL 60 bijv. de opdracht

"if a > b then c := d else a := p - q"

(waarvan de betekenis wel duidelijk zal zijn), die men echter ook m.b.v. de in 8. gegeven opdrachten kan weergeven.

Een meer essentiële omissie zullen we in 11. nog goed maken.

10. Waarom "stored programme"

Uit de gegeven algorithmen is al wel gebleken waar de ware winst bij het werken met computers zit en wat hun kracht is: het min of meer cyclisch doorlopen van programma onderdelen. Hierdoor worden slechts eenmaal opgeschreven opdrachten vele malen uitgevoerd, telkens in iets andere omstandigheden.

Aangezien nu het inlezen van een opdracht van een ponsband vele malen meer tijd kost dan het uitvoeren ervan, terwijl anderzijds in het geheugen staande informatie (w.o. opdrachten) vrijwel zonder tijdverlies toegankelijk is, is het dui-

delijk waarom men het hele programma eerst inleest, en daarna pas tot uitvoering overgaat (zie 0.4).

11. Procedures

Het aantal functies dat blijkens 8. in het repertoire is opgenomen, is slechts beperkt. En wanneer men veelvuldig bijv. een cosinus hyperbolicus of een Besselfunctie in zijn berekeningen wil betrekken, is het prettig als men deze functies aan het repertoire kan toevoegen. Dit is mogelijk op de volgende wijze. Als men aan het begin van zijn programma opneemt:

```
procedure cosh(x);
begin a := exp(x); cosh := (a + 1/a) × 0.5 end
```

(11.1)

dan kan men in willekeurige arithmetische uitdrukkingen net zo over de cosinus hyperbolicus beschikken als over de in 8. genoemde functies. Men mag dan schrijven q := cosh(7 × y) - 3 × cos(u) / cosh(p), met de betekenis die deze schrijfwijze suggereert.

Uiteraard is cosh een willekeurige naam, en had men evengoed de naam piet mogen bezigen. Op deze wijze kan men elke willekeurige functie, ook van meer variabelen, aan de machine bekend maken a.h.w. onder overlegging van een algorithme ervoor. In deze algorithme moet de berekende functiewaarde dan worden toegekend aan de naam van de procedure (hierboven was dit "cosh").

In wezen is de in 8.3 wèl genoemde functie exp op soortgelijke wijze aan de machine bekend gemaakt, maar dan voor alle programma's tegelijk. Van nature kent de machine namelijk de exponentiële functie helemaal niet. Bij deze bekendmaking stond tussen begin en end dan ook een vrij grote algorithme, die essentieel met behulp van een polynoom, een benadering van exp(x) berekent. We merken nog op, dat dientengevolge de berekening van exp(x) (en sin(x), etc.) veel meer tijd kost dan een simpele optelling of vermenigvuldiging. Vandaar ook dat we bij cosh(x) niet geschreven hebben "cosh := (exp(x) + exp(-x)) × 0.5", waardoor per "aanroep" van de cosinus hyperbolicus tweemaal een waarde van de exponentiële functie had moeten worden berekend, i.p.v. de ene die volgens (11.1) nodig is.

Op dezelfde wijze kan men algorithmen een naam geven die men niet als functie in een arithmetische uitdrukking wenst te gebruiken doch als zelfstandige opdracht. Toekenning van een waarde aan de naam van de procedure kan nu achterwege blijven. Bijvoorbeeld, als men aan het begin van zijn programma opneemt

```
procedure wissel(a, b);
begin p := a; a := b; b := p end
```

dan zal de opdracht "wissel(p, jan)" beduiden dat de waarden van p en jan verwisseld moeten worden (zie 15. voor een beter voorbeeld). Aldus kan men het opdrachtenrepertoire van de machine verrijken met opdrachten naar maat, die precies doen wat men wenst.

Deze mogelijkheid om een stuk programma een naam te kunnen geven en te kunnen aanduiden welke van de erin optredende namen voor de buitenwereld van belang zijn, blijkt bijzonder handig te zijn bij het programmeren. Men verdeelt nl. de algorithme in stukken die een duidelijke en goed formuleerbare werking hebben, formuleert deze als procedure, en maakt ze aan de machine bekend. Daarna kan dan het eigenlijke programma bestaan uit een zeer overzichtelijke opeenvolging van opdrachten waardoor deze procedures worden aangeroepen (dus opdrachten van het type wissel(p, jan)).

Rekencentra hebben veelal ettelijke procedures voor de meest uiteenlopende bezigheden kant en klaar liggen ten gerieve van de gebruikers.

12. Economie

In het algorithmisch denken staat centraal de economie. Het is nl. vaak in het geheel niet moeilijk een algorithme te bedenken die de gewenste resultaten oplevert. Men wenst echter bovendien

12.1. Dat het programma bij uitvoering zo weinig mogelijk rekentijd in beslag neemt. Er zijn voorbeelden van problemen waarvoor de meest voor de hand liggende algorithme uitvoering van n! operaties vergt, maar waarvoor een algorithme bedacht kan worden die met n^2 operaties toe kan. Voor n = 30 schelen n! en n^2 een factor 10^{30}.

12.2. Dat de algorithme bij uitvoering zo weinig mogelijk geheugenruimte in beslag neemt, bijv. omdat nog tal van andere algorithmen ook geheugenruimte nodig hebben. Elke variabele neemt een geheugenplaats in beslag. Vandaar dat we in de algorithmen in 5., 6. en 7. niet eerst de gehele getallenband hebben ingelezen en alle getallen aan geïndiceerde variabelen als waarde toegekend, om daarna pas onze eigenlijke werkzaamheden erop aan te vangen.

12.3. Dat het programma redelijk testbaar is. Wanneer men een programma heeft geschreven pleegt dit nl. fouten te bevatten. Om deze te kunnen vinden moet het

programma ingedeeld zijn in duidelijk herkenbare stukken, waarvan men de werking stuk voor stuk kan controleren. Ook in dit opzicht is het werken met procedures zeer aan te bevelen.

12.4. Dat het programma in een minimum van tijd geschreven is.
Deze eisen zijn uiteraard met elkaar in strijd. Toch is het goed ze te stellen. Het is vaak niet nuttig dagen te gaan zitten piekeren om 1 minuut rekentijd of 10 geheugenplaatsen uit te sparen.

13. Doelstelling van de rest van dit hoofdstuk

Met het resterend deel van dit hoofdstuk beogen we een indruk te geven van de wijze waarop men met een algorithme bezig is in zijn streven naar (rekentijd-) efficiency, en hoe deze algorithme daarbij kan evolueren tot iets dat nauwelijks meer lijkt op de algorithme die men in eerste aanleg had bedacht. Tevens willen we hierbij enkele algorithmen presenteren die een algemene bruikbaarheid hebben.

De beschouwingen monden uit in ALGOL-procedures. Deze zijn echter de technische completering van het algorithmisch denken, en de lezer mag ze beschouwen als geschreven te zijn in kleine letters.

14. Het beschouwde probleem

Het probleem dat aan het restant van dit hoofdstuk ten grondslag ligt is dat uit 7., waarbij we dus het aantal verschillende getallen van een band willen bepalen. We zullen hierbij vooral letten op de situatie dat vrijwel alle getallen op de band verschillend zijn.

We beschouwen (7.1) nog eens. Aangezien men de getallen van de band in elk geval zal moeten lezen en tellen, zal er op de regels $\neq$ (4), (5), (6) niet veel te bezuinigen vallen. We richten dus de aandacht op (4), (5), (6). Voor een nieuw van de band gelezen getal moet onderzocht worden of het met de waarde van een der a_1 t/m a_v samenvalt. Als dit niet het geval is worden de regels (5) en (6) v maal doorlopen. Mochten alle getallen op de band verschillend zijn, dan worden de regels (5) en (6) dus in totaal $1 + 2 + 3 + \ldots + (n - 1) \approx \frac{1}{2} n^2$ maal doorlopen, hetgeen dus uitvoering van $\frac{3}{2} n^2$ opdrachten vergt (weliswaar nemen de opdrachten niet allemaal evenveel tijd in beslag, maar we scheren ze eenvoudigheidshalve over een kam). De vraag is dus gewettigd of men niet op een voordeliger manier kan uitvinden of een gegeven getal in een gegeven rij getallen voorkomt.

15. Bepalen of getal in rij voorkomt

Dit probleem vertoont wel gelijkenis met het probleem om uit te zoeken of een gegeven naam in een gegeven kaartsysteem voorkomt. Verkeert dit kaartsysteem in een toestand van volstrekte wanorde, dan kan men niet veel anders doen dan de kaarten af te lezen in de volgorde waarin ze staan. Wanneer het kaartsysteem daarentegen alfabetisch geordend is zal men waarschijnlijk intuitief als volgt handelen: men bekijkt een ongeveer in het midden staande kaart en constateert dat de gezochte naam hierop staat, dan wel in alfabetische zin eraan voorafgaat of erop volgt. In het eerste geval is men klaar. In de laatste twee gevallen beperkt men zich tot de kaarten die aan de uitgekozen kaart voorafgaan resp. erop volgen, en handelt daarmee weer zoals zojuist met het hele kaartsysteem. Als het systeem n kaarten bevat, hoeft men de gegeven kaart slechts met hoogstens entier(^{2}log(n)) + 1 kaarten van het systeem te vergelijken.

Evenzo zal men kunnen nagaan of een gegeven getal in een gegeven geordende rij van n getallen voorkomt door bovenbedoeld halveringsproces toe te passen op hun rangnummers. Ook nu hoeft men het gegeven getal slechts met hoogstens entier(^{2}log(n)) + 1 getallen uit de rij te vergelijken. Aangezien $\lim_{n\to\infty} {}^2\log(n) / n = 0$ steekt dit voor toenemende n steeds gunstiger af bij het vergelijken van het gegeven getal met alle getallen uit de rij. Bijvoorbeeld voor $n = 10^5$ is $^2\log(n) / n \approx 1/6000$.

We geven nu een ALGOL procedure die volgens deze lijnen werkt. Hij doet nog iets meer dan alleen maar vaststellen of het gegeven getal in de rij voorkomt. Zij nl. de rij a_1 t/m a_n strikt monotoon stijgend. Dan kent de procedure aan j een waarde toe als volgt:

als $a_1 \leq$ getal $< a_n$ dan j zodanig dat $a_j \leq$ getal $< a_{j+1}$,

als getal $< a_1$ dan $j = 0$,

als getal $\geq a_n$ dan $j = n$.

Voorts wordt aan "aanwezig" de waarde 1 toegekend als het getal inderdaad in de rij voorkomt, de waarde 0 als dit niet het geval is.

```
procedure zoek(getal, a, n, j, aanwezig);                    (1)
begin b := 0; e := n + 1; aanwezig := 0;                     (2)
halveer:  j := entier((b + e) / 2);                          (3)
          if j = b then goto klaar;                          (4)
          if getal = a_j then goto gevonden;                 (5)
          if getal < a_j then goto links;                    (6)
rechts:   b := j; goto halveer;                              (7)
links:    e := j; goto halveer;                              (8)
gevonden: aanwezig := 1;                                     (9)
klaar:    end                                                (10)
```

(15.1)

Hierin is b steeds 1 kleiner dan de laagste index en e 1 groter dan de hoogste index van het nog te onderzoeken segment van de rij a_i, j geeft steeds ongeveer het midden daarvan aan. Pas als e - b = 1, en er dus niets meer te onderzoeken valt, gebeurt het dat j = b (zie (4)).

16. Getallen tussenvoegen in rij

Nu we hebben gezien dat het goed zoeken is in een geordende rij vragen we ons af hoe we ervoor kunnen zorgen dat de rij a_i geordend is. En als hij eenmaal geordend is moeten we er natuurlijk voor zorgen dat hij geordend blijft na toevoeging van een nieuw element. Dit laatste is zeer eenvoudig te doen:

Stel dat de waarden van a_1 t/m a_v een monotoon stijgende rij vormen, en stel dat hetzij $1 \leq j < v$ en $a_j < \text{getal} < a_{j+1}$, hetzij j = 0 en getal $< a_1$, hetzij j = v en getal $> a_v$. Dan zullen we de waarden van a_{j+1} t/m a_v toekennen aan a_{j+2} t/m a_{v+1}, en de waarden van getal aan a_{j+1}.

Een algorithme hiervoor luidt:

```
    i := v;                                  (1)
A:  if i = j then goto B;                    (2)
    a_{i+1} := a_i; i := i - 1; goto A;      (3)
B:  a_{j+1} := getal;                        (4)
```

(16.1)

(N.B. Men kent eerst de waarde van a_v toe aan a_{v+1}, daarna die van a_{v-1} aan a_v etc.; als men de volgorde $a_{j+2} := a_{j+1}$; $a_{j+3} := a_{j+2}$ etc. zou aanhouden krijgt men een heel ander effect!) Door het toevoegen van elementen aan de rij a_i van het begin af volgens deze algorithme te verrichten, is de rij a_i steeds geordend.

Er is weer analogie met het alfabetisch kaartsysteem, waar het toevoegen van een nieuwe kaart immers ook geschiedt door de j+1-ste t/m laatste kaart naar achteren te schuiven en de nieuwe kaart in het zo ontstane gat tussen de j-de en j+1-ste kaart te plaatsen.

Anders echter dan bij het kaartsysteem, waar het opschuiven der kaarten niets pleegt te kosten, is de verhuisactie in onze getallenrij erg onvoordelig, aangezien hiervoor immers v - j maal de regels (2) en (3) moeten worden doorlopen. In het ergste geval, nl. als voor elk in de rij op te nemen getal geldt j = 0 (d.w.z. de toe te voegen getallen komen in dalende volgorde binnen) zal het opbouwen van de rij a_1 t/m a_v dus in totaal aanleiding geven tot $1 + 2 + 3 + \ldots + (v-1) \approx \frac{1}{2} v^2$ maal doorlopen van de regels (2) en (3). Maar ook als men aanneemt dat de fractie van het aantal reeds in de rij aanwezige getallen, die moet worden opgeschoven voor het toevoegen van een nieuw getal, gemiddeld $\frac{1}{2}$ is, dan nog worden de regels (2) en (3) minstens circa $\frac{1}{4} v^2$ maal doorlopen voor het opbouwen van de rij a_1 t/m a_v.

Wanneer bij ons probleem uit 7. vrijwel alle getallen op de band verschillend zijn, zal een programma dat met de algorithmen (15.1) en (16.1) werkt, dus nog steeds uitvoering van een aantal opdrachten betekenen dat een factor n^2 bevat, en hebben we dus niet essentieel gewonnen t.o.v. (7.1).

17. Sorteren

We vragen ons derhalve af of er een voordeliger manier is om een rij van n getallen naar grootte te ordenen (men noemt dit ook wel sorteren). Dit is bepaald niet alleen van belang voor het probleem uit 7.; bij vele computertoepassingen moet men vaak vele duizenden getallen sorteren. Het hiervoor volgens 16. nodige aantal opdrachten ten bedrage van αn^2, α een constante in de grootte orde van 1, wil men dan wel graag reduceren.

De factor n^2 brengt ons op de gedachte de rij doormidden te delen (stel maar dat n even is) en de beide helften apart te sorteren, ten bedrage van $\frac{1}{4} \alpha n^2$ opdrachten voor elke helft, dus $\frac{1}{2} \alpha n^2$ opdrachten in totaal. Het samenvoegen van de twee gesorteerde helften tot één geordende rij is nl. voordelig uitvoerbaar als volgt:

Laten de samen te voegen trajecten zijn a_p t/m a_{p+h-1} en a_{p+h} t/m a_{p+2h-1}, beide monotoon stijgend (dat de trajecten aan elkaar grenzen en even lang zijn, is echter niet wezenlijk). We bepalen de kleinste van a_p en a_{p+h} en kennen die toe aan het element b_0 van een hulprij b_i. Als bijv. a_{p+h} die kleinste was bepalen we vervolgens de kleinste van a_p en a_{p+h+1} en sturen die naar b_1 etc.

Steeds vergelijken we van beide trajecten de kleinste overgebleven elementen, en kennen de kleinste daarvan toe aan het volgend element van de rij b_i. Zo gaan we door totdat een der trajecten uitgeput is.

Wat hierna gebeurt hangt er vanaf of het linker dan wel het rechter traject uitgeput is. Als het linkertraject uitgeput is, staan de resterende elementen van het rechtertraject al op de juiste plaats, en kunnen we de elementen van de b--rij daar gewoon voor zetten. Als daarentegen het rechtertraject uitgeput is moeten we eerst nog het restant van het linkertraject naar het eind van het rechtertraject verplaatsen, en daarna de b-rij terugschrijven. Zie (17.1) voor een algorithme.

Aangezien elke vergelijkoperatie resulteert in een wegschrijfoperatie naar de b-rij is het duidelijk dat het totale aantal opdrachten, dat moet worden uitgevoerd om de beide trajecten samen te voegen hoogstens $2h\gamma$ is, γ een constante van bescheiden grootte, onafhankelijk van h.

De beide helften te sorteren en samen te voegen kost dus $\frac{1}{2}\alpha n^2 + \gamma n$ opdrachten, en dit is al voor vrij kleine waarden van n voordelig t.o.v. αn^2.

We vatten dan ook de gedachte op bovenstaand proces ook maar te gebruiken om de beide helften te sorteren. Dit komt dan voor beide helften samen op $\frac{1}{4}\alpha n^2 + \gamma n$ opdrachten i.p.v. de term $\frac{1}{2}\alpha n^2$ in de vorige alinea, zodat de totale sortering op deze wijze $\frac{1}{4}\alpha n^2 + 2\gamma n$ opdrachten kost, en we hebben dus zelfs voor vrij kleine n alweer winst geboekt.

Door op deze weg voort te gaan kunnen we steeds snellere processen construeren. Ook zien we al spoedig in wat hier de kern van de zaak is: als de rij in 2^k even lange gesorteerde deelrijtjes is ingedeeld, kan men deze ten koste van in totaal γn opdrachten twee aan twee samenvoegen tot tweemaal zo lange gesorteerde rijtjes. Door dit k maal te herhalen (dus in totaal $k\gamma n$ opdrachten) is dan de hele rij gesorteerd.

Maar dan kunnen we het proces uit 16., waarmee we in bovenstaande gedachtengang de deelrijtjes sorteerden, ook wel geheel missen. Laat nl. de oorspronkelijke te sorteren rij een lengte n hebben die een macht van 2 is, dus $n = 2^k$. Dan kunnen we deze rij opvatten als te zijn ingedeeld in 2^k gesorteerde deelrijtjes ter lengte 1. We kunnen hem dan door een k maal herhaald twee aan twee samenvoegen geheel sorteren, ten koste van $\gamma n \cdot {}^2\log(n)$ opdrachten. Hiermee is dan onze speurtocht naar een sorteerproces dat geen factor n^2 meer heeft in het benodigde aantal te voeren opdrachten, met succes bekroond, althans wanneer de te sorteren rij een lengte 2^k heeft. Men ziet echter gemakkelijk in hoe men

dit proces kan modificeren om het bruikbaar te maken voor rijen van willekeurige lengte (zie opgave 2).

Ten besluite geven we nu twee ALGOL procedures die het hierboven geschetste nauwkeuriger weergeven.

De procedure meng voegt de monotoon niet dalende rijen a_p t/m a_{p+h-1} en a_{p+h} t/m a_{p+2h-1} samen tot een niet dalende rij a_p t/m a_{p+2h-1}. Ter vergemakkelijking van het lezen vermelden we nog dat l de telling in het linkertraject verzorgt, r die in het rechtertraject en bb die in de b-rij.

```
procedure meng(a, p, h);                                                    (1)
begin       l := p; lgrens := l + h; r := lgrens; rgrens := r + h; bb := - 1; (2)
vergelijk:  bb := bb + 1; if a[l] > a[r] then goto rechtskleinst;           (3)
linkskleinst: b[bb] := a[l]; l := l + 1; if l < lgrens then goto vergelijk; (4)
            goto bterug;                                                    (5)
rechtskleinst: b[bb] := a[r]; r := r + 1; if r < rgrens then goto vergelijk; (6)
rechtsleeg: a[l+h] := a[l]; l := l + 1; if l < lgrens then goto rechtsleeg; (7)
bterug:     a[p+bb] := b[bb]; bb := bb - 1; if bb ≥ 0 then goto bterug      (8)
end                                                                         (9)
```

We zien hieruit nog dat $\gamma = 8$. (17.1)

Met deze procedure kunnen we nu een compact sorteeralgorithme geven om a_1 t/m a_n, n een 2-macht, te sorteren op de wijze die we eerder aangaven.

```
procedure sorteer(a, n);                                                    (1)
begin       h := 1;                                                         (2)
langer:     p := 1; hh := 2 * h;                                            (3)
evenlang:   meng(a, p, h); p := p + hh; if p < n then goto evenlang;        (4)
            h := hh; if h < n then goto langer                              (5)
end                                                                         (6)
```

(17.2)

Opgave 1. Schrijf een procedure meng(a, p, h, k) die de monotoon stijgende rijen a_p t/m a_{p+h-1} en a_{p+h} t/m $a_{p+h+k-1}$ samenvoegt tot een stijgende rij a_p t/m $a_{p+h+k-1}$.

Opgave 2. Schrijf met behulp van de procedure uit opgave 1 een algorithme om een rij a_1 t/m a_n te sorteren, als n niet noodzakelijk een 2-macht is. Ga na dat dit mogelijk is met het uitvoeren van $\gamma n \cdot \text{entier}(^2\log(n-1) + 1)$ opdrachten.

18. Het oorspronkelijke probleem

We waren in de sorteerproblematiek terechtgekomen omdat we in een geordende rij zo voordelig konden nagaan of een gegeven getal er al in voorkomt.

Nu is voor het sorteerprobleem weliswaar een economische oplossing gevonden, maar het is in het geheel niet duidelijk hoe men onze rij a_i (zie 15.) op economische wijze gesorteerd houdt na elke toevoeging.

Stel echter eens dat we domweg de hele getalband sorteren met de procedure uit 17. opgave 2, ten koste van omstreeks $8n.^2\log(n)$ opdrachten. Men ziet gemakkelijk in dat een algorithme, die vaststelt hoeveel verschillenden er in een geordende rij voorkomen, slechts uitvoering vereist van een aantal opdrachten evenredig met n, hetgeen voor grote n dus klein is t.o.v. $8n.^2\log(n)$, zodat dit laatste bedrag in feite ongeveer de totale hoeveelheid werk aangeeft. Wanneer vrijwel alle getallen op de band verschillend zijn, steekt dit inderdaad zeer gunstig af tegen de $\frac{3}{2}n^2$ opdrachten die (7.1) in dit geval vereist. Voor $n = 10^3, 10^4, 10^5$ verschilt de hoeveelheid rekenwerk een factor 30, 225 resp. 1900.

De algorithme, die we nu uiteindelijk bereikt hebben, gelijkt inderdaad helemaal niet meer op wat ons oorspronkelijk voor de geest stond, zoals al in 13. werd opgemerkt.

Opmerking 1. Men kan laten zien dat de algorithme die ons met (15.1) en (16.1) voor de geest stond zelfs in het geval dat (16.1) in het geheel geen tijd gekost zou hebben (dus dat men de getallen gratis kon opschuiven) toch nauwelijks sneller zou zijn geweest dan de algorithme die we zojuist ontwikkeld hebben, nog steeds onder de aanname dat vrijwel alle getallen op de band verschillend zijn.

Opmerking 2. In het geval dat het aantal verschillenden op de band slechts een kleine fractie is van het totaal, is onze algorithme natuurlijk wel minder efficiënt. Door dan echter in de procedure meng een wijziging aan te brengen kan men de algorithme ook in dit geval veel efficiënter maken. De wijziging behelst dat meng nooit vlak achter elkaar twee gelijke getallen naar de b-rij zal sturen. Wanneer er in totaal k verschillende getallen op de band staan kan men laten zien dat het aantal benodigde opdrachten in totaal omstreeks $\gamma n.^2\log(k)$ is.

2. Numerieke toepassingen

0. Inleiding

De computer wordt vaak gebruikt bij het oplossen van allerlei mathematische problemen, en ALGOL is hierop speciaal gericht. Uiteraard moet men ook voor de oplossing van deze problemen algorithmen bedenken die in eindig veel opdrachten de oplossing berekenen. Voor tal van problemen echter kunnen dergelijke algorithmen niet bestaan, bijv. niet voor de bepaling van $\int_0^x e^{-t^2} dt$ en niet voor de bepaling van de wortels van hogere graadsvergelijkingen. Men mag dan hoogstens verwachten goede benaderingen voor de gewenste waarde te kunnen berekenen. De numerieke analyse is een tak van de wiskunde die er zich o.a. mee bezighoudt benaderingsmethoden op te stellen waarvan de uitkomsten wèl met eindig veel (en liefst zo weinig mogelijk) opdrachten te bepalen zijn. In dit hoofdstuk zullen we enkele van deze methoden laten zien.

Soms echter is de oplossing van een probleem zo triviaal tot een algorithme om te vormen, dat men dit zonder meer doet. Stel dat men (mogelijk als sluitstuk van een langere berekening) moet oplossen de vierkantsvergelijking $ax^2 + bx + c = 0$ met bekende waarden van de coëfficiënten. De door de zgn. "abc-formule" aangegeven oplossing $x_{1,2} = (-b \pm \sqrt{b^2 - 4ac})/2a$ heeft dan zo'n onmiddellijk algorithmiseerbare gedaante.

Het is echter geenszins zeker dat de computer aan de hand van deze door de traditie geheiligde algorithme ook de gewenste resultaten oplevert. En als men voor de oplossing van een probleem een benaderingsmethode heeft bedacht, waarvan men kan aantonen dat de mathematisch juiste uitkomst ervan dicht bij de oplossing van het gegeven probleem ligt, is het alweer niet zeker dat de computer een uitkomst oplevert die dicht bij de oplossing van het gegeven probleem ligt. Ook deze verschijnselen worden in de numerieke analyse bestudeerd teneinde de oorzaken op te sporen, en te vermijden dat op te stellen benaderingsmethoden deze onaangename eigenschap hebben. Ook dit zullen we aan voorbeelden toelichten.

1. Eindige precisie rekenapparatuur

De geschetste misère ontstaat doordat een computer, onder de (practische) aanname dat men hem slechts met een eindige getalband uitrust, principieel slechts in een eindige precisie kan werken. Hij bestaat dan nl. uit eindig veel componenten die elk slechts in eindig veel verschillende toestanden

kunnen verkeren, zodat de hele computer slechts in eindig veel verschillende toestanden kan verkeren. Hij kan dus zeker slechts eindig veel verschillende getallen voorstellen. Een eindige rij getallen, niet alleen bestaand uit 0, is echter nooit gesloten t.o.v. optelling, aftrekking, vermenigvuldiging en deling.

In de dagelijkse rekenpraktijk is de precisie echter veel sterker gelimiteerd dan deze principiële grens noodzakelijk maakt. Om redenen van efficiency kent nl. elke rekenmachine een verzameling van geprivilegeerde getallen met welke hij snel kan optellen, aftrekken, vermenigvuldigen en delen, en waarin hij het resultaat van deze operaties uitdrukt. Mocht het exacte resultaat van zo'n operatie niet tot deze verzameling behoren, dan wordt het afgerond op het dichtstbijzijnde getal van de verzameling. We zullen deze getallen aanduiden als de standaardgetallen. Wanneer men niets bijzonders doet wordt een programma geheel verwerkt met gebruikmaking van deze getallen en van deze snelle operaties, en we zullen dit verder in deze syllabus ook aannemen.

Men kan echter deze "natuurlijke" precisie van de machine zeer ver voorbijstreven door telkens 2 of meer standaardgetallen samen één nieuw getal te laten voorstellen (bijv. 3 getallen van 10 cijfers kunnen een getal van 30 cijfers voorstellen). Het is echter duidelijk dat de geheugenbehoefte dan lineair oploopt met de precisie, en dat de rekentijd nog veel sneller zal toenemen.

De verzameling der standaardgetallen varieert van computer tot computer. De getallen zijn steeds rationaal. Om de gedachten te bepalen geven we twee voorbeelden:

a) de getallen $m \times 10^p$, m een geheel getal tussen -10^8 en 10^8, p een geheel getal tussen -50 en 50;

b) de getallen $m \times 2^p$, m een geheel getal tussen -2^{40} en 2^{40}, p een geheel getal tussen -2048 en 2048 ($2^{40} \approx 10^{12}$, $2^{2048} \approx 10^{616}$).

Ook wanneer men een computer de waarde van een der elementaire functies laat berekenen (bijv. log of sqrt), wordt de uitkomst, indien deze niet tot de verzameling der standaardgetallen behoort, weer afgerond op het dichtstbijzijnde getal van de verzameling. Doorgaans echter zal de in werkelijkheid verkregen waarde verder van de ware verwijderd zijn dan op grond van deze afrondregel te verwachten is. Dit komt doordat deze functies benaderd worden door vrij ingewikkelde uitdrukkingen (zie 11.), bij het evalueren waarvan op

zichzelf al verscheidene afrondfouten gemaakt worden. Voorts vertoont de ware uitkomst van de benaderingsformule natuurlijk op zichzelf al een fout. Deze laatste fout houdt men echter veelal klein t.o.v. de afrondfout.

2. Vierkantsvergelijking

We illustreren het in 1. vermelde aan het klakkeloos gebruik van de abc-formule voor de vierkantsvergelijking.

Van een functie f wenst men de nulwaarde benaderd te bepalen. De functiewaarden zijn bekend in een aantal punten in een omgeving van 0. Hieruit blijkt dat de functie vrijwel lineair verloopt in deze omgeving, en dat de gevraagde nulwaarde ook daarin ligt (zie fig. 1). Weliswaar zou men door lineaire interpolatie al een heel redelijk resultaat hebben verkregen, maar men wenst nog iets beter te doen en bepaalt de kwadratische functie $ax^2 + bx + c$, die op drie der gegeven punten dezelfde waarde aanneemt als de gegeven functie. Nu heeft deze kwadratische functie weliswaar twee nulpunten, maar één daarvan is (op grond van het bijna lineaire gedrag van f) absoluut zeer groot, zodat men de absoluut kleinste wenst te hebben.

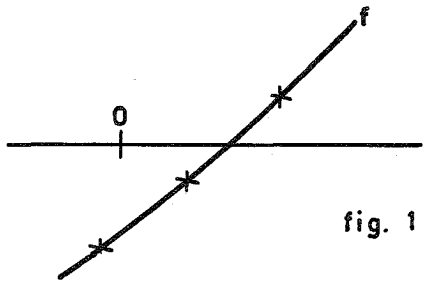


fig. 1

Stel eens dat de vierkantsvergelijking luidt $x^2 + (10^8 - 1)x - 10^8 = 0$, zodat de wortels zijn $\alpha = 1$, $\beta = -10^8$. Wegens $b > 0$ berekent men de absoluut kleinste wortel als $(-b + \sqrt{b^2 - 4ac})/2a$. Blijkbaar geldt $\sqrt{b^2 - 4ac} = 10^8 + 1$. Stel echter dat, tengevolge van hogergenoemde afrondfouten, de door berekening verkregen waarde van deze vierkantswortel 10^8 is, een in relatieve zin toch zeer goed resultaat. Dan krijgt men voor de absoluut kleinste wortel 0.5 in plaats van 1, een relatief zeer slecht resultaat.

Men gaat gemakkelijk na dat bij elke vierkantsvergelijking met twee sterk in grootte orde verschillende wortels bij toepassing van de abc-formule een kleine relatieve fout bij de berekening van $\sqrt{b^2 - 4ac}$ resulteert in een veel grotere relatieve fout in de verkregen absoluut kleinste wortel. Onder relatieve fout is te verstaan: $\frac{\text{verkregen waarde - exacte waarde}}{\text{exacte waarde}}$.

3. Nauwkeurigheid uitkomst bij onnauwkeurige operanden

De in 2. ontmoete situatie illustreert het belang van de (onmiddellijk in te ziene) algemene regel: als optelling of aftrekking van twee getallen, die beide met een kleine relatieve fout belast zijn, een resultaat heeft dat in absolute waarde veel kleiner is dan de beide operanden, dan kan dit resultaat met een zeer grote relatieve fout belast zijn. Als men een algorithme bedenkt, bij uitvoering waarvan afrondfouten gemaakt kunnen worden, moet men met deze regel terdege rekening houden.

Anderzijds ziet men gemakkelijk in dat bij vermenigvuldiging en deling van twee getallen met een lage relatieve fout de uitkomst weer een lage relatieve fout zal hebben. Hetzelfde geldt voor optelling en aftrekking als de uitkomst dezelfde grootte orde heeft als de absoluut grootste der operanden.

Opgave 1. Stel dat men werkt met de standaardgetallen als bedoeld in 1, voorbeeld a). Stel dat de machine voor $-\frac{\pi}{2} < x < \frac{\pi}{2}$ een waarde voor $\sin(x)$ oplevert, die men verkrijgt door de exacte waarde af te ronden op het dichtstbijzijnde standaardgetal (halven worden afgerond naar boven). Welke relatieve fout ongeveer heeft dan de berekende waarde van $\sin(x)$ en van $\sin(x) - x$ voor $x = 10^{-2}, 10^{-4}, 10^{-6}, 10^{-8}$.

Opgave 2. Schrijf een algorithme die, onder de aanname van opgave 1, voor $x \neq 0$, $|x| < \frac{\pi}{2}$, $\frac{\sin(x) - x}{x^3}$ berekent met een relatieve fout van hoogstens 10^{-4}. Aangenomen mag worden dat, behalve in de berekening van $\sin(x)$, geen afrondfouten worden gemaakt.

4. Vierkantsvergelijking II

Uit 3. ziet men dat de in 2. geschetste ramp niet kan optreden bij de bepaling van de absoluut grootste wortel, en dat men deze vindt met een relatieve precisie die vergelijkbaar is met die van $\sqrt{b^2 - 4ac}$. Dit geeft dan echter meteen de remedie voor het probleem, hoe men de absoluut kleinste wortel met een behoorlijke relatieve precisie kan bepalen. Men deelt dan nl. de absoluut grootste wortel op c/a. Zo krijgt men de volgende formules:

$$-\frac{b}{2a}\{1 + \sqrt{1 - 4ac/b^2}\} \quad \text{voor de absoluut grootste wortel} \qquad (4.1)$$

$$-\frac{2c}{b}/\{1 + \sqrt{1 - 4ac/b^2}\} \quad \text{voor de absoluut kleinste wortel} \qquad (4.2)$$

ongeacht de tekens van a, b en c.

Hieruit moge nog blijken dat het voor numeriek behoorlijke resultaten niet altijd aanbevelenswaardig is vierkantswortels uit noemers te verdrijven; zou men dit nl. toepassen op de formule voor de absoluut kleinste wortel, dan krijgt men een uitdrukking, waaraan weer de bovengesignaleerde bezwaren van de klassieke formule kleven.

5. Recursie

Een ander voorbeeld, dat een zeer voor de hand liggende algorithme falikant verkeerde uitkomsten kan geven, is het volgende. Men wenst

$$I_n = \int_0^1 t^n e^{-t} dt \qquad (5.1)$$

voor een aantal waarden van n te kennen, zeg voor n = 0 t/m 100. Uit (5.1) ziet men door partiële integratie dat

$$I_n = nI_{n-1} - e^{-1} \quad , \quad n \geqslant 1 \; , \quad I_0 = 1 - e^{-1} \; . \qquad (5.2)$$

Hiermee kan men de gevraagde waarden voor I_n dus zeer eenvoudig genereren. Als men dit echter met de computer doet blijken de uitkomsten nergens op te lijken. We gaan na hoe dit komt.

Het getal e^{-1} is geen standaardgetal. Bij het werken met de computer zal men dus genoegen moeten nemen met $e^{-1} + \varepsilon$. Werkt men met de standaardgetallen uit 1a, dan geldt wegens $0.1 < e^{-1} < 1$ dat $|\varepsilon| \leqslant \frac{1}{2} . 10^{-8}$. Een tabel leert dat $\varepsilon \approx 10^{-9}$.

Beschouwen we nu de recursie

$$I_n^* = nI_{n-1}^* - e^{-1} - \varepsilon \; , \quad I_0^* = 1 - e^{-1} - \varepsilon \; , \qquad (5.3)$$

dan ziet men gemakkelijk in dat $I_n^* = I_n - p_n \varepsilon$, waarbij

$$p_n = np_{n-1} + 1 \; , \quad p_0 = 1 \; .$$

Dus $p_n > n!$, zodat $|I_n^* - I_n| > n!|\varepsilon|$.

Nu geldt $I_n < \frac{1}{n+1}$, zoals men gemakkelijk uit (5.1) ziet door de factor e^{-t} te vervangen door 1.

Derhalve zal gelden $|I_n^* - I_n| > I_n$ wanneer $(n+1)!\varepsilon > 1$, en dit is bij de gegeven waarde van ε reeds het geval voor n = 12. Het verschil tussen I_{12}^* en I_{12} is dus groter dan I_{12} zelf, en men maakt dus een relatieve fout van meer dan 100% wanneer men de waarde van I_{12}^* neemt i.p.v. I_{12}.

Bovendien kan de machine de recursie (5.3) natuurlijk niet exact doorvoeren, en een bij de k-de stap gemaakte afrondfout ε_k geeft aanleiding tot een fout van $n(n-1)\ldots(k+1)\varepsilon_k$ in I_n^*.

Al met al is de recursie (5.2) dus hoogst <u>instabiel</u>. Algemeen noemt men een recursieschema instabiel wanneer een kleine verstoring in de waarde van een der variabelen een steeds stijgende fout in de volgende variabelen ten gevolge heeft. Inmiddels is dit op zichzelf nog geen ramp; dat wordt het pas als de fouten sneller stijgen dan de variabelen zelf, zoals hier.

Voor deze instabiliteit moet men bij recursieschema's altijd oppassen. Het is overigens wel duidelijk dat men bij het rekenen "met de hand" evenzeer met dit verschijnsel te maken krijgt.

De vraag rijst natuurlijk hoe men de waarden van I_n dan wel in behoorlijke precisie te weten kan komen.

Stel eens dat I_n bekend was. Dan konden we met de recursie

$$I_{n-1} = (e^{-1} + I_n)/n \tag{5.4}$$

de waarde van I_{n-1}, I_{n-2}, ..., I_0 bepalen. We kunnen echter niet hopen I_n ooit exact te kennen, en beschouwen daarom hiernaast de recursie

$$J_{n-1} = (e^{-1} + J_n)/n \quad . \tag{5.5}$$

Stel dat $J_n = I_n + \delta$. Dan ziet men gemakkelijk in dat

$$J_{n-k} = I_{n-k} + \delta/\{n(n-1)\ldots(n-k+1)\} \; .$$

Wegens $I_{n-k} > \frac{1}{e(n-k+1)}$ (zie (5.1)) is de relatieve fout van J_{n-k} t.o.v. I_{n-k} dus kleiner dan

$$e\delta/\{n(n-1)\ldots(n-k+2)\} \; , \tag{5.6}$$

hetgeen bij stijgende k zeer snel afneemt.

Men zou dus kunnen overwegen I_n door numerieke kwadratuur (zie 12. t/m 17.) in behoorlijke benadering te bepalen, en dan terug te gaan met (5.5). Het snelle dalen van (5.6) suggereert echter een eenvoudiger weg. We lichten dit toe aan een voorbeeld.

Stel dat men I_0 t/m I_{100} met een relatieve fout $< 10^{-7}$ wenst te kennen, dan beweren we dat dit lukt door (5.5) toe te passen voor n = 104, 103, 102, ... 1, uitgaande van $J_{104} = 0$. Wegens $I_{104} < \frac{1}{105}$ is dus $|\delta| < \frac{1}{105}$. Dan geeft (5.6)

voor k = 4 de waarde e/(105 × 104 × 103 × 102) en dit is zeker kleiner dan 10^{-7}. Dus is de relatieve fout van J_{100} t.o.v. I_{100} zeker kleiner dan 10^{-7}. In J_{99}, J_{98}, ... is de fout nog veel kleiner.

In dit betoog werd geen rekening gehouden met tussentijds gemaakte afrondfouten en de onnauwkeurigheid waarmee $\frac{1}{e}$ bepaald is. We beschouwen daartoe nog eens (5.5). Aangezien $\frac{1}{e}$ en J_n hetzelfde teken hebben introduceren de afrondfouten bij het berekenen van het rechterlid van (5.5) dus telkens geringe relatieve fouten, waarvan het effect op de volgende J_p echter weer snel uitdempt naarmate p afneemt, tengevolge van het snelle dalen van (5.6).

Deze wat losjes aandoende beschouwingen over de effecten van afrondfouten kan men exact substantiëren, maar daaraan gaan wij hier voorbij.

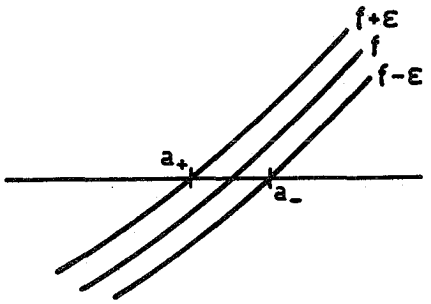
Deze methode van teruglopende recurrenties met vrij willekeurige startwaarde vindt de laatste jaren in de numerieke analyse veel toepassing.

6. Onzekerheidsinterval voor wortels

Wanneer men een algorithme heeft ter berekening van de waarden f(x) van een functie f, die bij exacte uitvoering der arithmetische operaties ook inderdaad f(x) zou opleveren, zal dit bij verwerking op een computer ten gevolge van de afrondfouten gewoonlijk hiervan afwijkende waarden opleveren, zeg f(x) + r(x). Aangezien de waarde van r(x) nogal grillig heen en weer pleegt te springen als functie van x, zegt men wel, naar analogie van verschijnselen in de fysica, dat f ten gevolge van de afrondfouten met (numerieke) ruis belast wordt. In tegenstelling tot de fysische ruis is de numerieke ruis echter volstrekt reproduceerbaar!

Dit heeft gevolgen wanneer men bijv. een nulwaarde van f wenst te bepalen. Veruit de meeste processen hiertoe gebruiken nl. berekende functiewaarden om prognoses te doen omtrent de ligging van de nulwaarde, en concluderen bij een functie, die op een omgeving van de nulwaarde stijgend is, uit het positief zijn der berekende functiewaarde in een punt a dat de nulwaarde links van a ligt (analoog rechts voor negatief). Wanneer t.g.v. afrondfouten de berekende functiewaarde het verkeerde teken heeft gekregen, wordt men zo de verkeerde kant uitgestuurd. De afrondfouten impliceren voor deze processen dus een begrensdheid van de precisie waarmee men de nulwaarde van f kan bepalen.

Zij nu f continu en monotoon stijgend op een zeker interval I, en laat f daar een nulwaarde hebben. Zij $\varepsilon = \max_{x \in I} |r(x)|$. Stel dat f + ε en f - ε ook nul-



waarden hebben op I, en noem deze a_+ respectievelijk a_-. Dan zullen we (a_+, a_-) onzekerheidsinterval voor de nulwaarde van f t.o.v. I noemen. Analoog voor dalende f.

Buiten dit onzekerheidsinterval hebben de berekende functiewaarden in elk geval het juiste teken, daarbinnen weten we het niet. Van een behoorlijk proces om nulwaarden te bepalen zal men verwachten, dat het een waarde oplevert in of althans vlak bij dit onzekerheidsinterval. Meer mag men redelijkerwijs niet verwachten, aangezien immers op grond van de gemaakte aanname omtrent r de grafiek van f + r slechts binnen de "buis" hoeft te verlopen en dus inderdaad een nulwaarde vlak bij of zelfs in a_- of a_+ kan hebben.

7. Wortels van vergelijkingen door successieve halvering

Zij f een continue functie op het segment [a,b], en laat f(a) en f(b) verschillend teken hebben. Gevraagd een nulwaarde van f.

Het eenvoudigste en wellicht ook meest voor de hand liggende proces hiervoor is dat van successieve halvering: zij m het midden van [a,b]; als f(m) hetzelfde teken heeft als f(a) vervang a door m en herhaal; als f(m) hetzelfde teken heeft als f(b) vervang b door m en herhaal; stop zodra de waarde in een der uiteinden 0 is of wanneer de lengte van het interval beneden een zekere grens gedaald is.

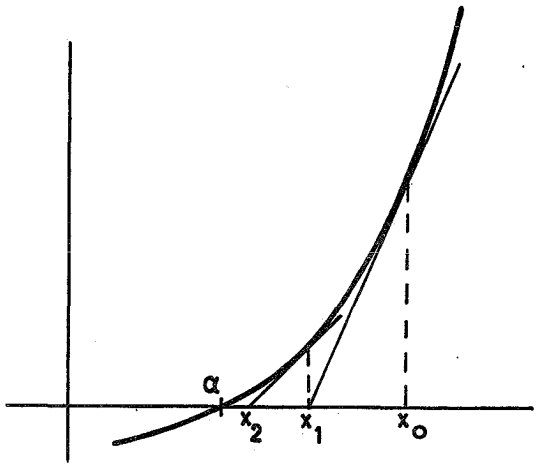
Uiteraard voert het proces tot willekeurig kleine intervallen, die altijd minstens één punt gemeen hebben met het onzekerheidsinterval van een wortel.

Reductie van het oorspronkelijke interval met een factor 10^6 kost blijkbaar het berekenen van 22 waarden van de functie. Wanneer f een ingewikkelde functie is (bijv. een integraal of een slecht convergente reeks) dan kan dit veel tijd kosten, en vraagt men zich af of het niet voordeliger kan. En evenals men een kaart in een alfabetisch kaartsysteem sneller kan vinden dan met halvering mogelijk is wanneer men meer informatie over het namenbestand heeft, kan men nulwaarden voordeliger bepalen wanneer men meer informatie over de functie heeft. De beide problemen zijn trouwens duidelijk verwant.

8. Wortels van vergelijkingen met Newton

Zo'n proces, dat onder omstandigheden sneller verloopt dan halvering, is dat van Newton-Raphson. Het is wel het meest bekende proces om niet-lineaire ver-

gelijkingen op te lossen. Zij op te lossen $f(x) = 0$. Stel dat we al een benaderde oplossing x_0 kennen. Dan wensen we x_1 zo te kiezen dat $f(x_1) = 0$, dus $\Delta f(x) \overset{\text{def}}{=} f(x_1) - f(x_0) = -f(x_0)$. Wegens $\Delta f(x) \approx f'(x)\Delta x$ nemen we voor $\Delta x \overset{\text{def}}{=} x_1 - x_0$ de waarde $-f(x_0)/f'(x_0)$. Dit komt dus neer op (zie figuur) het trekken van de raaklijn aan de grafiek in het punt $(x_0, f(x_0))$, en deze raaklijn snijden met de x-as. Analoog vinden we bij x_1 een x_2 etc.



Formeel luidt dit proces dus

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)} \quad . \tag{8.1}$$

Een proces van de gedaante $x_{i+1} = \varphi(x_i)$ noemt men een <u>iteratief proces</u>.

Enkele eigenschappen van het proces van Newton:

a) Als f in het nulpunt α tweemaal differentieerbaar is en $f'(\alpha) \neq 0$ en x_0 ligt voldoende dicht bij α, dan convergeert de rij x_i naar α.

<u>Bewijs</u>. Er geldt (met ε, η en θ functies die naar 0 gaan als het argument naar α gaat)

$$f(x) = (x-\alpha)f'(\alpha) + \tfrac{1}{2}(x-\alpha)^2(f''(\alpha) + \varepsilon(x))$$

$$f'(x) = f'(\alpha) + (x-\alpha)(f''(\alpha) + \eta(x)) \ .$$

Invullen in $x_{i+1} - \alpha = x_i - \alpha - \dfrac{f(x_i)}{f'(x_i)}$ levert

$$x_{i+1} - \alpha = \left[\frac{f''(\alpha)}{2f'(\alpha)} + \theta(x_i)\right](x_i - \alpha)^2 \quad . \tag{8.2}$$

Zij op een voldoend kleine omgeving $(\alpha-\delta, \alpha+\delta)$ de factor tussen [ ] in absolute waarde $\leq A$ dan geldt $|x_{i+1} - \alpha| \leq A|x_i - \alpha|^2$. Voor alle x_0 uit die omgeving, die bovendien nog voldoen aan $|x_0 - \alpha| < \frac{1}{A}$ is het proces convergent.

Immers $|x_0 - \alpha| = \frac{\rho}{A}$, $\rho < 1$, zodat $|x_i - \alpha| \leq A^{-1}\rho^{2^i}$, en $\rho^{2^i} \to 0$ voor $i \to \infty$.

b) De ongelijkheid (8.2): $|x_{i+1} - \alpha| \leq A|x_i - \alpha|^2$ verklaart de populariteit van de methode. Stel nl. dat A ongeveer 1 is, d.w.z. dat $|f''(\alpha)/2f'(\alpha)|$ ongeveer 1 is. Dan wordt, althans als x_i voldoende dicht bij α ligt, telkens als men i met 1 verhoogt de "fout" $|x_i - \alpha|$ ongeveer gekwadrateerd. D.w.z. als $|x_i - \alpha| \approx 0.1$ dan $|x_{i+1} - \alpha| \approx 0.01$, $|x_{i+2} - \alpha| \approx 0.0001$, $|x_{i+3} - \alpha| \approx 0.00000001$ etc., zodat de convergentie op den duur zeer scherp is. Men spreekt in dit geval van kwadratische convergentie, ter onderscheiding van de situatie waarbij slechts geldt $|x_{i+1} - \alpha| \leq B|x_i - \alpha|$, met $0 < B < 1$, in welk geval men spreekt van lineaire convergentie. Dit is bijv. het geval bij successieve halvering.

Opgave 1. Ga na dat Newton lineaire convergentie geeft voor $f'(\alpha) = 0$ en $f''(\alpha) \neq 0$.

c) Als x_0 ver van α ligt hoeft geen convergentie op te treden (zie fig. 1). In dit verband spreekt men wel van het aantrekkingsgebied van een wortel: een samenhangende verzameling beginwaarden waartoe de wortel behoort en waarvoor het proces naar de wortel convergeert. Vaak voert het Newton proces na enig heen en weer springen toch wel naar het aantrekkingsgebied van een wortel, zij het wellicht niet de dichtstbijzijnde (zie fig. 2).

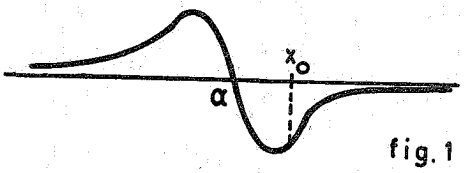


fig. 1

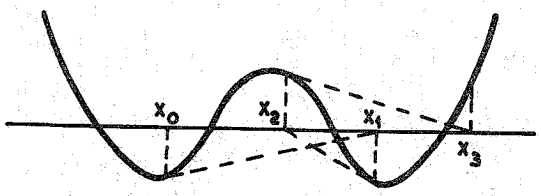


fig. 2

d) Uit het in a) bewezene laat zich niet zonder meer een aantrekkingsgebied voor een wortel afleiden. Het is daarom prettig te weten dat convergentie in elk geval optreedt als de functie convex resp. concaaf is (d.w.z. bol naar beneden resp. naar boven) op een interval dat een nulpunt bevat, wanneer x_0 ook in dat interval ligt en $f(x_0) > 0$ resp < 0 is. De rij x_i convergeert dan zelfs monotoon, zoals men gemakkelijk inziet.

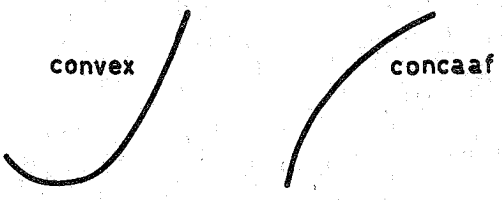


fig. 3

e) Merk op dat de bekende algorithme $x_{i+1} = \frac{1}{2}(x_i + \frac{a}{x_i})$ om $\sqrt{a}$ te bepalen verkregen wordt door Newton's proces toe te passen met $f(x) = x^2 - a$. Wegens d) convergeert het proces voor $x_0 > \sqrt{a}$ en wel monotoon en kwadratisch (dit laatste wegens a)).

Opgave 2. Als men $\sqrt{a}$ niet kent, hoe bepaalt men dan in e) een geschikte startwaarde x voor monotone convergentie.

Opgave 3. Wat wordt in dit proces de relatie tussen $x_{i+1} - \alpha$ en $x_i - \alpha$, aannemende dat in (8.2) de term $\theta(x_i)$ verwaarloosd mag worden.

Opgave 4. Bereken benaderd x_1, x_2, x_3 voor $a = 2$ en $x_0 = 10^6$. Is het geconstateerde gedrag in overeenstemming met wat men bij kwadratische convergentie mag verwachten?

Opgave 5. Zij a een willekeurig complex getal en zij $\alpha^2 = a$. Beschouw de rij uit 8e. Laat zien dat hiervoor geldt

$$\frac{x_{n+1} - \alpha}{x_{n+1} + \alpha} = \left(\frac{x_n - \alpha}{x_n + \alpha}\right)^2 .$$

Toon hiermee aan dat convergentie, zo deze bestaat, kwadratisch is.
Bewijs dat, als x_0 niet op de middelloodlijn ligt van het lijnstuk $(-\alpha, \alpha)$ in het complexe vlak, de rij x_i convergeert naar α (resp. $-\alpha$) wanneer x_0 aan dezelfde kant van bedoelde middelloodlijn ligt als α (resp. $-\alpha$).

Opgave 6. Zij a een complex getal, $a \neq 0$. Men past het proces van Newton-Raphson toe met $f(x) = 1 - \frac{1}{ax}$. Laat zien dat hiervoor geldt

$$x_{n+1} - \frac{1}{a} = a\left(x_n - \frac{1}{a}\right)^2 .$$

Toon hiermee aan dat convergentie, zo deze bestaat, kwadratisch is.
Bewijs dat de rij x_i convergeert naar $\frac{1}{a}$ wanneer x_0 binnen de cirkel in het complexe vlak ligt die door 0 gaat en $\frac{1}{a}$ als middelpunt heeft.
(N.B. Wegens $x_{n+1} = 2x_n - ax_n^2$ heeft men hierin een methode om $\frac{1}{a}$ willekeurig goed te benaderen zonder een deling uit te voeren; in vroeger jaren, toen computers nog geen deling ingebouwd hadden, werd dit proces veel gebruikt.)

9. Iteratieve processen in het algemeen

In feite betekent het oplossen van een vergelijking $f(x) = 0$ m.b.v. een iteratief procédé $x_{i+1} = \varphi(x_i)$ niets anders dan transformatie van de vergelijking $f(x) = 0$ naar de vergelijking $x = \varphi(x)$ (mits φ continu is). Voor deze vergelijking $x = \varphi(x)$ heet het proces $x_{i+1} = \varphi(x_i)$ dan opeens de oplosmethode met successieve substitutie.

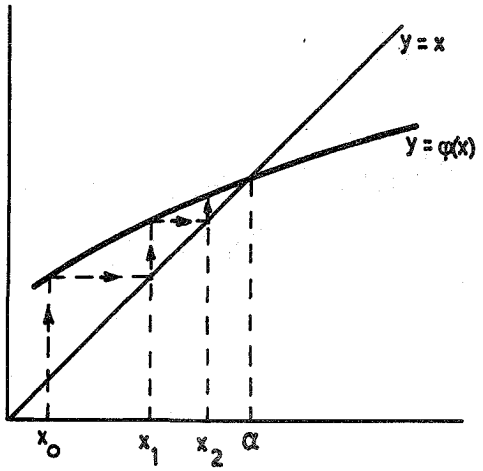
Wanneer φ differentieerbaar is op een symmetrische omgeving van de wortel α, waarop geldt dat $|\varphi'(x)| \leq C < 1$ dan is het proces convergent mits gestart in die omgeving. Er geldt dan immers $x_{i+1} - \alpha = \varphi(x_i) - \alpha = \varphi'(\xi)(x_i - \alpha)$ zodat

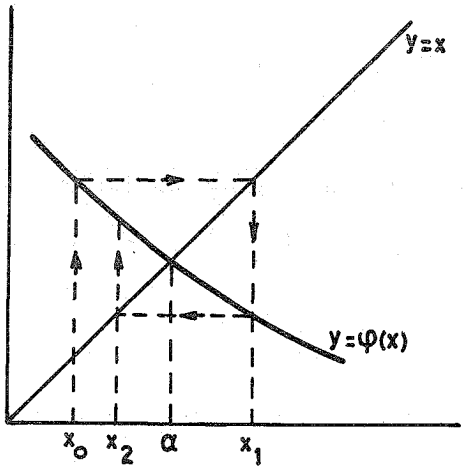
$|x_{i+1} - \alpha| \leq C|x_i - \alpha| \leq C^2|x_{i-1} - \alpha| \ldots \leq C^{i+1}|x_0 - \alpha|$, en $C^{i+1} \to 0$ voor $i \to \infty$.

De convergentie is minstens lineair met factor C (vgl. 8b). Bovendien ziet men, dat wanneer φ' positief is op de hele omgeving, de rij x_i monotoon is, terwijl voor φ' negatief op de hele omgeving de x_i beurtelings links en rechts van α liggen.

Heel snelle convergentie verwacht men uiteraard als C klein is. Bij Newton geldt $\varphi(x) = x - \frac{f(x)}{f'(x)}$ zodat $f(\alpha) = 0 \Rightarrow \varphi'(\alpha) = 0$. Dus inderdaad is C hier heel klein op een omgeving van α.

Een en ander ziet men fraai geïllustreerd in een grafische voorstelling waar men de grafieken van φ en van de identieke functie uitzet.





10. Stopcriterium

Is het vinden van een geschikte startwaarde voor een iteratief proces een probleem (zie 8c)), dit is evenzeer het geval met het vinden van een goed stopcriterium, d.i. een criterium op grond waarvan men de iteratie beëindigt. Beide problemen spelen een essentiële, en soms de moeilijkste, rol bij het bedenken van een algorithme.

Aangezien de bereikbare precisie doorgaans begrensd is (zie 6.), is het vaak de wens met de iteratie door te gaan totdat deze grens bereikt is, m.a.w. men wenst een "zo goed mogelijke" uitkomst te krijgen. In dit geval is het bijzonder prettig als men een proces heeft dat theoretisch strikt monotoon convergeert. Bij praktische uitvoering komt er aan deze monotonie beslist een eind, omdat de computer maar eindig veel verschillende getallen kan voorstellen. Onder ruime voorwaarden echter komt dit einde zeer dicht bij de gewenste "zo goed mogelijke" uitkomst. We lichten dit toe aan Newton's proces.

Dus op te lossen $f(x) = 0$, f convex. Zij $g = f + r$ de door de algorithme voor f in feite opgeleverde functie (zie fig. 1). Zij x_0 een startwaarde voor monotoon dalende convergentie. Men ziet uit de prent dat, tenzij de waarden van $f'(x)$ zeer slecht bepaald zouden zijn (zeg fouten van meer dan 50% zouden vertonen), zodra x_i op enige afstand links van het onzekerheidsinterval terecht komt zal gelden $x_{i+1} > x_i$. Anderzijds

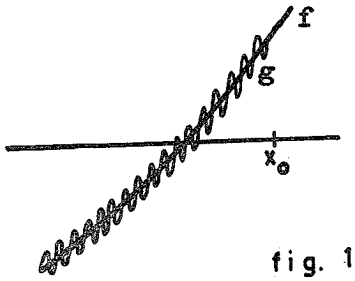


fig. 1

zal onder dezelfde aanname, zolang x_i nog op enige afstand rechts van het onzekerheidsinterval ligt, gelden $x_{i+1} < x_i$. De monotonie zal dus inderdaad in de situatie van fig. 1 eindigen heel dicht in de buurt van het onzekerheidsinterval.

De hier gegeven beschouwing is duidelijk slechts kwalitatief. Met allerlei extra (nog ruime) voorwaarden kan men ook een kwantitatieve beschouwing geven. We gaan hieraan nu voorbij.

We passen het voorafgaande toe om de positieve wortel van de vergelijking $\sin x = \frac{1}{2}x$ met Newton's proces "zo nauwkeurig mogelijk" te bepalen. We nemen $f(x) = \sin(x) - \frac{1}{2}x$. De wortel ligt op het interval $[0,\pi]$, zoals men gemakkelijk nagaat, en op dat interval is f concaaf. Dus gevraagd x_0 zodat $f(x_0) < 0$ (zie 8d)), Klaarblijkelijk voldoet $x_0 = \pi$, en er is monotoon dalende convergentie.
We krijgen dan het volgende programma:

```
begin  x := 3.1415926535;                                   (1)
herh:  xacc := x - (sin(x) - 0.5 x x)/(cos(x) - 0.5);       (2)
       if xacc ⩾ x then goto klaar;                         (3)
       xacc := x; goto herh;                                (4)
klaar: PRINT(x)                                             (5)
end                                                         (6)
```

(10.1)

Na het voorafgaande is hierover niet veel meer te zeggen. We merken nog slechts op dat we hier niet een hele rij variabelen x_i creëren, zoals men wellicht verwachtte, maar het spel spelen met slechts twee variabelen, x en x', waarvan de waarden samenhangen volgens $x' = x - \frac{f(x)}{f'(x)}$. Dit geschiedde natuurlijk om redenen van geheugeneconomie.

Opmerking. Als men de regels (2), (3) en (4) vervangt als volgt:

```
herh: h := (sin(x) - 0.5 x x)/(cos(x) - 0.5);
      if h ⩽ 0 then goto klaar;
      xacc := x - h; goto herh;
```

hetgeen wiskundig equivalent is met de oorspronkelijke algorithme, is het niet meer zeker dat het proces eindigt; het is nl. denkbaar dat de aftrekking x - h als resultaat x oplevert terwijl toch $h > 0$ (als nl. $|h| \ll |x|$).

Opgave 1. Geef een programma dat deze vergelijking oplost met de methode van successieve halvering. Opgehouden mag worden zodra men een interval heeft ter lengte $\leq 10^{-6}$. Hoe vaak zal dit programma moeten halveren?

Opgave 2. Geef een algorithme die met de methode van successieve halvering voor een op [a,b] monotone continue functie, met tegengestelde functiewaarden aan de uiteinden, zo nauwkeurig mogelijk (i.v.m. ruis) het nulpunt bepaalt.

11. Successieve halvering vergeleken met Newton

We zagen dat Newton onder omstandigheden veel sneller convergeert dan successieve halvering. Ook wordt men bij Newton niet verplicht twee functiewaarden van tegengesteld teken te bedenken. Vooral het eerste is vaak erg belangrijk.
Anderzijds is het vinden van twee functiewaarden van tegengesteld teken vaak eenvoudiger dan het vinden van een gegarandeerd goede startwaarde bij Newton. Heeft men bedoelde twee functiewaarden, dan is er bij successieve halvering geen convergentieprobleem meer. Ook het probleem, hoe weer op te houden, is hier veel eenvoudiger: men stelt een tolerantie, en na verloop van tijd wordt die vanzelf bereikt, zelfs als men de tolerantie veel kleiner heeft gemaakt dan het onzekerheidsinterval (in welk geval men er dan wel niet op kan rekenen dat men inderdaad binnen de gewenste afstand van de "ware" wortel komt, maar dan toch de best mogelijke nauwkeurigheid heeft bereikt). Tenslotte vereist successieve halvering niet het berekenen van de waarden der afgeleide. Dit is prettig wanneer bijv. de functie slechts gegeven is door een rekenproces, en men de afgeleide dus in het geheel niet voorhanden heeft. Maar ook wanneer men wél een analytische uitdrukking heeft voor de afgeleide, is deze heel vaak veel ingewikkelder dan die van de functie zelf, en zal dienovereenkomstig meer rekentijd vergen. Daardoor is men bij Newton misschien wel in veel minder stappen klaar, maar deze stappen kunnen per stuk veel meer rekentijd vergen, zodat het helemaal niet zeker is dat Newton ook sneller is, gemeten in rekentijd.

De beide processen hebben derhalve hun eigen merites. Het blijkt mogelijk te zijn door modificatie van deze processen een proces te maken dat de voordelen van beide min of meer in zich verenigt, maar daarop willen we nu niet ingaan.

We willen slechts opmerken dat men vrijwel overal in de numerieke analyse voor de oplossing van een probleem een aantal methoden ter beschikking heeft, elk met hun eigen voordelen betreffende rekentijd, geheugengebruik, stabiliteit, gemakkelijk starten en stoppen etc. Het doen van de juiste keus vereist vaak veel vakkennis en ervaring.

12. Numerieke kwadratuur

Het laatste onderwerp dat we in dit hoofdstuk aan de orde stellen is het benaderd bepalen van de waarde van integralen, de zg. numerieke kwadratuur.

Zij dus gevraagd de waarde van $\int_a^b f(x)dx$ te bepalen, terwijl van f geen primitieve in termen van een redelijk klein aantal elementaire functies bekend is. Een algorithme om deze waarde te benaderen wordt trivialerwijs geleverd door de Riemann-som: zij n een nog nader te bepalen getal. Definieer de rij x_i als volgt: $x_i = a + \frac{i}{n}(b-a)$, $i = 0, 1, \ldots, n$. Dan is de Riemann som

$$I_0(n) = \sum_{i=1}^{n} f(x_i)(x_i - x_{i-1}) = \frac{b-a}{n} \sum_{i=1}^{n} f(x_i) \qquad (12.1)$$

voor voldoend grote n een voldoend goede benadering voor $\int_a^b f(x)dx$.

Hoe groot moet men n kiezen? Zij f continu differentieerbaar op [a,b]. Toepassing van Taylor op $\int_x^{x_i} f(t)dt$ geeft:

$$\int_{x_{i-1}}^{x_i} f(x)dx = f(x_i)(x_i - x_{i-1}) - f'(\eta_i)\frac{(x_i - x_{i-1})^2}{2}$$

met η_i een getal waarvan we slechts weten $x_{i-1} < \eta_i < x_i$. Dus

$$\int_a^b f(x)dx = I_0(n) - f'(\eta) \sum_{i=1}^{n} \frac{(x_i - x_{i-1})^2}{2} = I_0(n) - f'(\eta)\frac{(b-a)^2}{2n} \qquad (12.2)$$

met $a < \eta < b$.

We zullen $I_0(n)$ een numerieke kwadratuurformule noemen; we zullen het verschil tussen $\int_a^b f(x)dx$ en $I_0(n)$ de restterm noemen.

Wanneer men voor $|f'(x)|$, $a \leq x \leq b$, een bovengrens kent, kan men gemakkelijk afleiden voor welke waarde van n de restterm zeker beneden een zekere voorgeschreven waarde zal blijven.

Wanneer we een indruk willen krijgen van de prestaties van deze kwadratuurformule is er natuurlijk geen bezwaar tegen dat we dit doen aan de hand van een integraal, waarvan we de waarde ook wel langs analytische weg kunnen bepalen. We beschouwen nu de situatie $a = 0$, $b = 1$, $f(x) = x$, zodat $\int_a^b f(x)dx = 0.5$.

Wegens $f'(x) = 1$ voor alle x kunnen we nu zonder meer zeggen dat de restterm gelijk is aan $\frac{1}{2n}$. Wenst men dat dit bijv. kleiner is dan 10^{-5}, een niet overdreven te noemen precisie, dan zal men $n = 50000$ moeten nemen. Met name als de functiewaarden zich slechts moeizaam laten berekenen, is dit erg hoog.

13. Trapeziumregel

Kunnen we ook begrijpen hoe het komt dat men voor een bescheiden precisie zoveel werk moet verrichten? Wanneer we naar fig. 1 kijken zien we dat $I_0(n)$ blijkbaar de totale oppervlakte der rechthoekjes voorstelt, en dit is inderdaad geen fraaie benadering van het oppervlak onder de kromme.

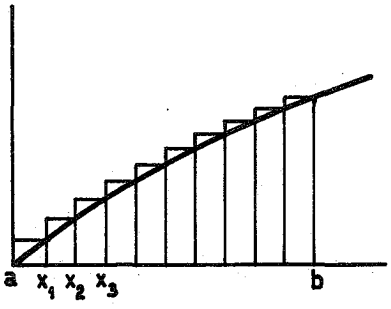


fig. 1

Dit brengt ons op het idee om het oppervlak onder de kromme tussen x_{i-1} en x_i te benaderen door dat van het trapezium $[x_i, f(x_i), f(x_{i-1}), x_{i-1}]$, en dat voor alle i (zie fig. 2). We krijgen dan de benadering $(x_i - x_{i-1})\{f(x_i) + f(x_{i-1})\}/2$ voor $\int_{x_{i-1}}^{x_i} f(x)dx$. Deze benadering noemt

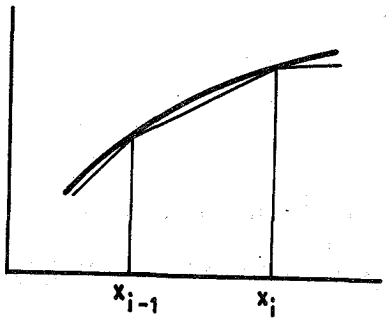


fig. 2

men de trapeziumregel. Herhaalde toepassing levert de herhaalde trapeziumregel

$$I_1(n) = \frac{b-a}{n}\left\{\tfrac{1}{2}f(a) + \sum_{i=1}^{n-1} f(x_i) + \tfrac{1}{2}f(b)\right\} \quad . \qquad (13.1)$$

N.B. Deze uitdrukking verschilt slechts zeer weinig van die voor $I_0(n)$; in de hierin voorkomende som is slechts $f(x_n)$ vervangen door $\frac{1}{2}\{f(x_0) + f(x_n)\}$.

Op een iets minder eenvoudige wijze dan in 12 kan men aantonen

$$\int_{x_{i-1}}^{x_i} f(x)dx = (x_i - x_{i-1})\{f(x_i) + f(x_{i-1})\}/2 - f''(\eta_i)\frac{(x_i - x_{i-1})^3}{12}$$

met $x_{i-1} < \eta_i < x_i$, uiteraard aangenomen dat f tweemaal differentieerbaar is.

Men krijgt zo bij $I_1(n)$ een restterm $-f''(\eta)\frac{(b-a)^3}{12n^2}$ met $a < \eta < b$.

Voor onze functie $f(x) = x$ is de vooruitgang vergeleken bij $I_0(n)$ zeer opvallend. Voor $n = 1$ krijgt men nl. al meteen de exacte waarde van de integraal, weinig verrassend overigens gezien de herkomst van $I_1(n)$.
Echter geeft ook voor vele andere functies $I_1(n)$ een veel nauwkeuriger resultaat dan $I_0(n)$.

Het is duidelijk dat men wel functies kan bedenken waarvoor bij zekere a, b en n $\max_{a<\eta<b} |f''(\eta)| \cdot \frac{(b-a)^3}{12n^2} > \max_{a<\eta<b} |f'(\eta)| \frac{(b-a)^2}{2n}$. Het is echter ook duidelijk dat bij vaste a en b doch steeds stijgende n het ongelijkteken op den duur omkeert. Immers wordt bij verdubbeling van n het rechterlid slechts gehalveerd, het linkerlid echter gevierendeeld. Voor stijgende n mag men dan ook voor de rij $I_1(n)$ snellere convergentie naar de juiste waarde van de integraal verwachten dan bij $I_0(n)$.

Opgave 1. Ga na welke waarde van n men moet nemen om aan de hand van de gegeven restterm te kunnen garanderen dat $I_0(n)$ resp. $I_1(n)$ de waarden der volgende integralen met een fout kleiner dan 10^{-5} opleveren: $\int_0^{\pi/2} \sin(x)dx$, $\int_0^3 e^{-x^2}dx$, en constateer dat de geringe wijziging van I_1 t.o.v. I_0 wel afbetaalt!

14. Simpson

Na het voorafgaande is het duidelijk hoe men aan een hele klasse van kwadratuurformules kan komen: verdeel het integratieinterval in stukken en vervang op elk stuk de integrand door een eenvoudige functie, die op dat stuk een goede benadering geeft van de integrand, en die men wél analytisch kan integreren. Door de verdeling in deelintervallen steeds fijner te maken krijgt

men zo processen, die naar de ware waarde van de integraal convergeren met een convergentiesnelheid, die men in de hand heeft door de gebruikte soort benadering.

Een heel bekende is nog de <u>regel van Simpson</u>:

$$\int_{x_{i-1}}^{x_{i+1}} f(x)dx \approx \frac{x_{i+1} - x_{i-1}}{6} \{f(x_{i-1}) + 4f(x_i) + f(x_{i+1})\} \qquad (14.1)$$

voor equidistante x_{i-1}, x_i, x_{i+1}. Men krijgt deze benadering door integratie van de kwadratische functie die in x_{i-1}, x_i en x_{i+1} dezelfde waarden aanneemt als f, en men kan laten zien dat de restterm luidt $-f^{(4)}(\eta)\frac{(x_{i+1} - x_{i-1})^5}{2880}$, onder de aanname dat f 4 maal differentieerbaar is.

Herhaalde toepassing levert de <u>herhaalde regel van Simpson</u>:

$$I_2(n) = \frac{b-a}{3n} \{f_0 + 4f_1 + 2f_2 + 4f_3 + \ldots + 4f_{n-1} + f_n\} \qquad (14.2)$$

met n even en $f_i = f(x_i)$, de x_i equidistant, $x_0 = a$, $x_n = b$ en

$$\int_a^b f(x)dx = I_2(n) - f^{(4)}(\eta)\frac{(b-a)^5}{180n^4} \quad . \qquad (14.3)$$

De factor n^4 in de noemer van de restterm geeft natuurlijk weer veel snellere convergentie dan bij de trapeziumregel het geval is.

<u>Opgave 1.</u> Ga na hoeveel functiewaarden men bij de herhaalde regel van Simpson nodig heeft om de waarde van $\int_0^{\pi/2} \sin(x)dx$ met een fout $< 10^{-5}$ te bepalen. Is dit voor $\int_0^3 e^{-x^2}dx$ ook nog prettig na te gaan?

15. <u>Glijdende stap.</u>

Het is in het algemeen helemaal niet eenvoudig een behoorlijke bovengrens voor $|f^{(4)}(x)|$ te vinden, en dus ook niet om een redelijke en toch gegarandeerd goede waarde van n te bepalen. Maar zelfs als men dat kan, dan moet toch gezegd worden dat het benaderen van $\int_a^b f(x)dx$ door $I_2(n)$ doorgaans een inefficiënte zaak is. Immers, wanneer $f^{(4)}$ slechts in sommige delen van het

integratie interval grote waarden aanneemt, op de rest daarentegen slechts bescheiden waarden, dan zou men, door uitsluitend op $\max|f^{(4)}(x)|$ af te gaan, het hele integratie interval in kleine stukjes verdelen, terwijl alleen maar in sommige gebieden een zo fijne verdeling vereist was.

Voor een efficiënte gang van zaken zou men dus op het hele integratie interval $f^{(4)}$ minitieus moeten onderzoeken om de op elke plaats vereiste spatiëring der punten x_i te kunnen vaststellen. Men heeft middelen beraamd om dit door de kwadratuuralgorithme zelf te laten uitzoeken, zij het onder een zekere aanname. Een proces dat dit doet noemt men een proces met glijdende stap. Wij zullen dit nu laten zien.

Zij dus gevraagd $\int_a^b f(x)dx$ benaderd te bepalen, met een fout $\leq \varepsilon$. We zullen $[a,b]$ nu verdelen in deelintervallen, op elk waarvan we nog 3 punten kiezen die met de eindpunten ervan een equidistant rijtje vormen (zie fig. 1). Zij $[c,d]$ zo'n deelinterval. Dan zullen we verlangen dat $\int_c^d$ benaderd wordt met een fout $\leq \frac{d-c}{b-a}\varepsilon$. We beschouwen nu voor $[c,d]$ $I_2(2)$ en $I_2(4)$, die dus $\int_c^d$ benaderen m.b.v. een enkelvoudige toepassing van Simpson resp. m.b.v. Simpson toegepast op de beide helften, en men mag verwachten dat $I_2(4)$ het beste resultaat zal geven. Men zal dus eisen dat $[c,d]$ zo gekozen is dat de restterm voor $I_2(4)$ in modulus kleiner is dan $\frac{d-c}{b-a}\varepsilon$. Onder de aanname dat, wanneer de restterm van $I_2(4)$ aan deze eis voldoet, het interval $[c,d]$ wel zo klein zal zijn dat $f^{(4)}$ daarop niet meer erg sterk varieert, ziet men uit (14.3) dat de restterm van $I_2(4)$ omstreeks 16 maal zo klein zal zijn als die van $I_2(2)$, en dus omstreeks 15 maal zo klein als het verschil van $I_2(4)$ en $I_2(2)$.

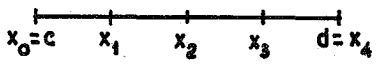


fig. 1

Onder de gemaakte aanname is dus een noodzakelijke voorwaarde opdat $I_2(4)$ een voldoend nauwkeurige benadering van $\int_c^d$ geeft dat

$$\left.\begin{aligned} &|I_2(2) - I_2(4)| \leq 15\,\frac{d-c}{b-a}\,\varepsilon \\ \text{ofwel}\quad &|f(x_0) - 4f(x_1) + 6f(x_2) - 4f(x_3) + f(x_4)| \leq \frac{180}{b-a}\,\varepsilon \quad . \end{aligned}\right\} \qquad (15.1)$$

Veronderstellen we bovendien dat (15.1) niet per ongeluk toch eens vervuld zal zijn terwijl [c,d] zo lang is dat $I_2(4)$ nog geen voldoend goede benadering geeft, dan wordt (15.1) ook een voldoende voorwaarde, en hebben we een <u>criterium</u> om uit te maken dat $\int_c^d$ voldoende nauwkeurig door $I_2(4)$ wordt voorgesteld. Anderzijds geeft een niet al te royaal vervuld zijn van (15.1) aan dat de verdeling met het oog op de te bepalen precisie niet overbodig fijn is.

De gestelde voorwaarden zijn hard, daar het in concrete situaties doorgaans niet doenlijk is met volstrekte zekerheid te verifiëren of ze vervuld zijn. Als afschrikwekkend voorbeeld beschouwe men de functie waarvan fig. 1 de grafiek voorstelt, en die juist in alle in beschouwing genomen punten de waarde 0 heeft, terwijl toch duidelijk $\int_c^d \neq 0$. In de praktijk valt het echter allemaal nogal mee.

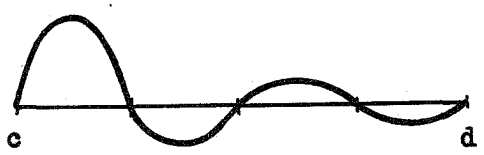


fig. 1

16. <u>Algorithme</u>

Met behulp van het criterium (15.1) kan men nu een redelijk efficiënte algorithme maken ter benadering van integralen.

De kern van deze algorithme is het volgende.

Bereken voor het aan de beurt zijnde deelinterval (in het begin is dit het oorspronkelijke integratie interval) f_0 t/m f_4 en ga na of hiervoor het criterium vervuld is.

Als het criterium niet vervuld is gaan we verder met de linkerhelft van het onderhavige interval. We tekenen echter op een daartoe aangelegde lijst wel aan dat er nog een restant moet worden behandeld; dit aantekenen geschiedt door op de plaats van de lijst, volgend op eventuele vorige aantekeningen, op te schrijven wat middelpunt en rechtereindpunt van dit restant zijn, alsmede de functiewaarden in die punten.

Als het criterium daarentegen wel vervuld is, accepteren we de waarde van $I_2(4)$ voor de integraal over dat interval. Vervolgens beschouwen we als aan de beurt zijnd interval het restant dat het laatst aan de lijst is toegevoegd. Rechtereindpunt en middelpunt met de functiewaarden daarin zijn bekend, echter ook het linkereindpunt en de functiewaarde daarin, zijnde rechtereindpunt

en functiewaarde daarin van het zojuist geaccepteerde interval. Als er na een geaccepteerd deelinterval geen restant meer over is zijn we klaar.

We lichten dit toe aan een voorbeeld. We geven steeds de waarden van x_0 t/m x_4 van het aan de beurt zijnde interval, en geven door onderstreping daarvan aan dat we deze x_i en de bijbehorende functiewaarde op dat moment nog moeten berekenen; de niet onderstreepte x_i en hun functiewaarde zijn op dat moment al bekend van vroeger.

Voor ons voorbeeld nemen we als integratie interval [0,128]. Zie ook fig. 1.

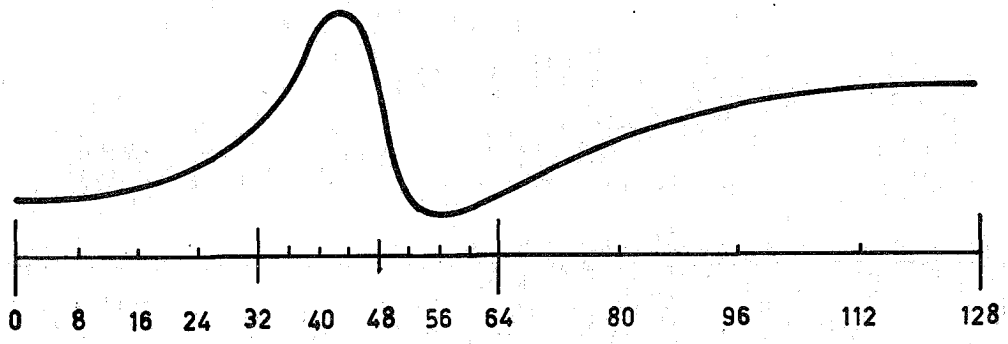


fig. 1

	x_0	x_1	x_2	x_3	x_4	crit	restant	
(1)	0	32	64	96	128	misl	[64,128]	noteer 96, 128 en f-waarden in lijst
(2)	0	16	32	48	64	misl	[32,64]	noteer 48, 64 en f-waarden in lijst
(3)	0	8	16	24	32	lukt		lees laatste informatie uit lijst (zie (2)) af, en veeg ze uit na transport naar (4)
(4)	32	40	48	56	64	misl	[48,64]	noteer 56, 64 en f-waarden in lijst
(5)	32	36	40	44	48	lukt		lees laatste informatie uit lijst (zie (4)) af, en veeg ze uit na transport naar (6)
(6)	48	52	56	60	64	lukt		lees laatste informatie uit lijst (zie (1)) af, en veeg ze uit na transport naar (7)
(7)	64	80	96	112	128	lukt		lijst is leeg.

De informatie die we het laatst aan de restantlijst hebben toegevoegd wordt er dus steeds weer als eerste uitgehaald. Dit LIFO (last in first out) principe geeft bij vele algorithmen aanleiding tot een heel handige en overzich-

telijke manier om informatie op te bergen en weer te voorschijn te halen. Het doet denken aan het opslaan van informatie met behulp van een stapel kaarten, waar men alleen maar kaarten bovenaan mag toevoegen of weghalen. Men zegt dan ook bij algorithmen, die met hun informatie volgens het LIFO principe omspringen, dat ze deze informatie stapelen.

Deze stapeltechniek is eenvoudig in ALGOL te formuleren: de stapel is een rij geïndiceerde variabelen S_1, S_2, S_3, ... met een zg. stapelwijzer i; als men een nieuwe informatieëenheid aan de stapel wil toevoegen, kent men die toe aan S_{i+1}, S_{i+2}, ... S_{i+k}, en verhoogt de waarde van i met k; i wijst dus steeds het laatste bezette element in de S-rij aan.

Een en ander wordt toegepast in de volgende procedure, die een functie f integreert van a naar b met een fout hoogstens ε. Na het voorafgaande is hierbij niet veel meer te vertellen. Vermeld zij nog dat bij de variabele I telkens wordt opgeteld de waarde van $12 \times I_2(4)$ voor geaccepteerde intervallen.

```
procedure INT(a, b, f, eps);
begin    delta := 180 * eps/abs(b - a); I := 0; i := 0;
         x0 := a; x2 := (a + b)/2; x4 := b;
         f0 := f(a); f2 := f(x2); f4 := f(b);
cycle:   x1 := (x0 + x2)/2; x3 := (x2 + x4)/2;
         f1 := f(x1); f3 := f(x3);
         T := f0 - 4 × (f1 + f3) + 6 × f2 + f4;
         if abs(T) ≤ delta then goto vervuld;
         S[i+1] := x3; S[i+2] := f3; S[i+3] := x4; S[i+4] := f4; i := i + 4;
         x4 := x2; f4 := f2; x2 := x1; f2 := f1; goto cycle;
vervuld: I := I + (x4 - x0) × (f0 + 4 × (f1 + f3) + 2 × f2 + f4);
         if i = 0 then goto klaar;
         x0 := x4; f0 := f4; i := i - 4;
         x2 := S[i+1]; f2 := S[i+2]; x4 := S[i+3]; f4 := S[i+4];
         goto cycle;
klaar:   INT := I/12
end
```
(16.1)

17. Slotopmerkingen

Enkele slotopmerkingen over numerieke kwadratuur.

a) Er is reeds gewezen op de moeilijke verifieerbaarheid van de aannamen waarop (16.1) steunt. Niettemin is het moeilijk de algorithme tot deraillement te brengen met andere dan speciaal voor dit doel bedachte integralen, en

blijven bij een oordeelkundig gebruik debacles uit. Hiervoor zijn ettelijke gronden aan te voeren, waaraan we nu echter maar voorbijgaan.

Deze verschijnselen treft men bij vrijwel elke algorithme der numerieke wiskunde aan. Van enig automatisme bij het gebruik van deze algorithmen mag dan ook geen sprake zijn. Men moet input en output der algorithmen kritisch bezien, en oningewijden kunnen zich aan dit gereedschap bezeren.

b) Aangezien we toch al aannamen dat op elk geaccepteerd deelinterval de restterm van $I_2(4)$ omstreeks 15 maal zo klein zou zijn als $I_2(4) - I_2(2)$, kunnen we hopen onze uitkomst nog iets te verbeteren door op de regel met label "vervuld" de tweede factor van de tweede term in het rechterlid te vervangen door f0 + 4 × (f1 + f3) + 2 × f2 + f4 - T/15. Dit is inderdaad een in de praktijk wel gebruikelijke aanvulling.

c) In het officiële ALGOL 60 kan men (16.1) nog aanzienlijk vereenvoudigen, dankzij het feit dat het procedure mechanisme daar de stapeling geheel automatisch kan verrichten. Deze opmerking betreft echter slechts de schrijfwijze van de algorithme; de werking blijft vrijwel eender.

d) Ofschoon de in (16.1) belichaamde algorithme veel gebruikt wordt, zijn er nog heel wat andere, en in vele gevallen efficiëntere, algorithmen voor numerieke kwadratuur.

Opgave 1. Ga na dat voor de (toch niet zo erg gezochte) integraal $\int_{-1}^{2} e^{-au^2} du$,

a een groot positief getal, de hier beschreven algorithme een onzinnig antwoord zal opleveren.

3. Non-numerieke computer toepassingen

Uit de vele non-numerieke toepassingen (men denke bijv. aan vertaalprogramma's, aan verwerking van administratieve gegevens, aan formula manipulation, aan verkeersregeling) kiezen we voor deze bespreking slechts enkele programma's waarbij een groot aantal mogelijkheden systematisch wordt afgetest om de oplossing, of alle oplossingen, of de beste oplossing van het een of andere combinatorische probleem te vinden.

3.1. Backtracking

3.1.1. Laat A een eindige verzameling zijn. We zoeken alle vectoren

$$(a_1 , a_2 , \ldots , a_h) \qquad (h \in \mathrm{Nat},\ a_1 \in A, \ldots , a_h \in A)$$

die aan zekere (hier niet te specificeren) voorwaarden voldoen. ("Nat" stelt de verzameling $\{1,2,3,\ldots\}$ voor). Door $G(a_1 , \ldots , a_h)$ stellen we de uitspraak voor die zegt dat $(a_1 , \ldots , a_h)$ een oplossing is. We behoeven niet alle rijtjes te onderzoeken (het zijn er trouwens oneindig veel) want er is bij elke vector $a_1 , \ldots , a_k$ iets bekend over de mogelijkheden die er voor a_{k+1} nog bestaan om het rijtje tot een oplossing voort te zetten. Deze mogelijkheden voor a_{k+1} vormen een verzameling $C(a_1 , \ldots , a_k)$ die we "candidaten verzameling" noemen. Het is niet gezegd dat alle candidaten kans hebben, alleen dat niet-candidaten geen kans hebben. Om precies te zijn, als $a_{k+1} \notin C(a_1 , \ldots , a_k)$, dan is er geen enkele oplossing met $h \geq k+1$ waarvan de eerste $k+1$ elementen $a_1 , \ldots , a_{k+1}$ zijn.

In $C(a_1 , \ldots , a_k)$ is een volgorde gegeven. Als $C(a_1 , \ldots , a_k)$ niet leeg is, stelt $E(a_1 , \ldots , a_k)$ het eerste element voor. Als $a_{k+1} \in C(a_1 , \ldots , a_k)$, stelt $NK(a_1 , \ldots , a_k , a_{k+1})$ de uitspraak voor dat a_{k+1} nog niet het laatste element van $C(a_1 , \ldots , a_k)$ is. Als nu $NK(a_1 , \ldots , a_k , a_{k+1})$ waar is, stelt $V(a_1 , \ldots , a_k , a_{k+1})$ het element van $C(a_1 , \ldots , a_k)$ voor dat op a_{k+1} volgt. In het geval $k = 0$ schrijven we resp. C, E, $NK(a_1)$, $V(a_1)$; C is de verzameling candidaten voor de eerste plaats, E het eerste element ervan, als $a_1 \in C$ is $NK(a_1)$ de uitspraak dat C nà a_1 nog een element bezit, en $V(a_1)$ het direct op a_1 volgende element.

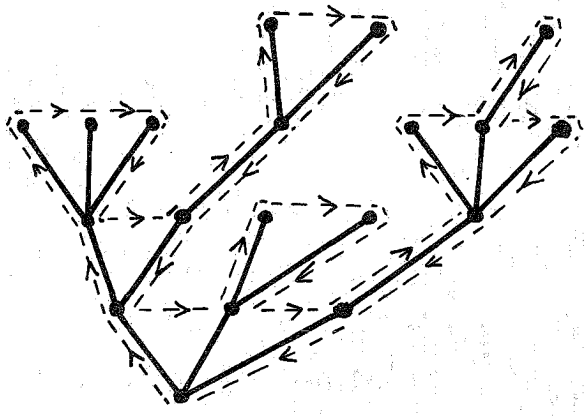
We zullen onderstellen dat een oplossing niet door achterplaatsen van verdere elementen weer een oplossing kan worden. Gemakshalve onderstellen we dat dit

ook in de candidatenverzamelingen tot uitdrukking komt, d.w.z.

$$G(a_1, \ldots, a_k) \quad \text{impliceert} \quad C(a_1, \ldots, a_k) = \emptyset \ .$$

Verder onderstellen we gemakshalve dat als $C(a_1, \ldots, a_k)$ leeg is, ook $C(a_1, \ldots, a_k, a_{k+1}, \ldots, a_m)$ leeg is voor elk stel $a_{k+1}, \ldots, a_m$. Als deze "gemakshalve" gemaakte onderstellingen niet juist mochten zijn, zijn ze door een kleine wijziging in de definitie van $C(a_1, \ldots, a_k)$ juist te maken.

3.1.2. Men kan de situatie visueel maken door een boom te tekenen waarin als punten optreden alle vectoren $a_1, \ldots, a_{k+1}$ met $a_{k+1} \in C(a_1, \ldots, a_k)$. De boom is in een plat vlak getekend, heeft een wortel onderaan (het lege rijtje), de takken stijgen op. De punten op de eerste etage stellen rijtjes met lengte één voor, enz. De toppen (d.z. punten van waaruit geen tak weer naar boven gaat) zijn punten $(a_1, \ldots, a_k)$ met $C(a_1, \ldots, a_k) = \emptyset$. Sommige daarvan kunnen oplossingen zijn. In elk punt $(a_1, \ldots, a_k)$ dat geen top is corresponderen de opstijgende takken met de elementen van $C(a_1, \ldots, a_k)$; de boom is zó getekend dat de volgorde van deze takken (gaande van links naar rechts) overeenkomt met de volgorde in $C(a_1, \ldots, a_k)$.



We kunnen langs alle punten van deze boom lopen door de gestippelde pijlen te volgen. Een opstijgende pijl betekent verlenging van de vector, een afdalende pijl verkorting, en een pijl naar rechts betekent vervanging van het laatste element. Een afdaling wordt nooit direct gevolgd door een opstijging.

De naam "backtracking" hangt samen met de afdalingen, waarbij de vector aan de achterkant wordt afgebroken.

We zullen in het vervolg onderstellen dat de boom eindig is.

3.1.3. De in 3.1.2. beschreven pijlenwandeling wordt in vage termen als volgt beschreven.

1. ingang bij wortel;
2. if je kunt omhoog then begin ga omhoog; goto 2 end;
3. if oplossing bereikt then noteer die;
4. if je kunt naar rechts then begin ga naar rechts; goto 2 end;
5. if je kunt omlaag then begin ga omlaag; goto 4 end;
6. uitgang bij wortel.

Opm. Als de stap omlaag naar de wortel voert wordt dit beschouwd als niet meer omlaag kunnen.

3.1.4. Wat precieser beschrijven we het in termen van de door ons ingevoerde uitdrukkingen. k is een integer die de etage aangeeft, a_1, a_2, ... een array bestaande uit elementen van A.

1. k := 0;
2. if $C(a_1,\ldots,a_k) \neq \emptyset$ then begin $a_{k+1} := E(a_1,\ldots,a_k)$; k := k + 1; goto 2 end;
3. if $G(a_1,\ldots,a_k)$ then noteer oplossing;
4. if $NK(a_1,\ldots,a_k)$ then begin $a_k := V(a_1, \ldots, a_k)$; goto 2 end;
5. if k > 1 then begin k := k - 1; goto 4 end;
6. klaar.

3.1.5. Als voorbeeld het volgende. Zij g een natuurlijk getal. Gevraagd alle oplossingen van

$$k \in Nat,\ a[1] \in Nat,\ a[2] \in Nat,\ \ldots,\ a[k] \in Nat,$$
$$1 \leq a[1] < a[2] < \ldots < a[k],$$
$$a[1] + \ldots + a[k] = g.$$

Voor de verzameling C(a[1], ..., a[k]) zullen we nemen de verzameling van alle $p \in Nat$ met

$$p > a[k], \qquad a[1] + \ldots + a[k] + p \leq g$$

maar men zou ook een kleinere verzameling kunnen nemen, door bijv. de gevallen

met a[1]+...+a[k]+p < g alleen toe te laten als a[1]+...+a[k]+2p+1 ≤ g.

Verder is G(a[1], ..., a[k]) de uitspraak a[1] + ... + a[k] = g,
NK(a[1], ..., a[k]) de uitspraak a[1] + ... + a[k] < g, vervolgens
E(a[1], ..., a[k]) = a[k] + 1, en ook V(a[1], ..., a[k]) = a[k] + 1.

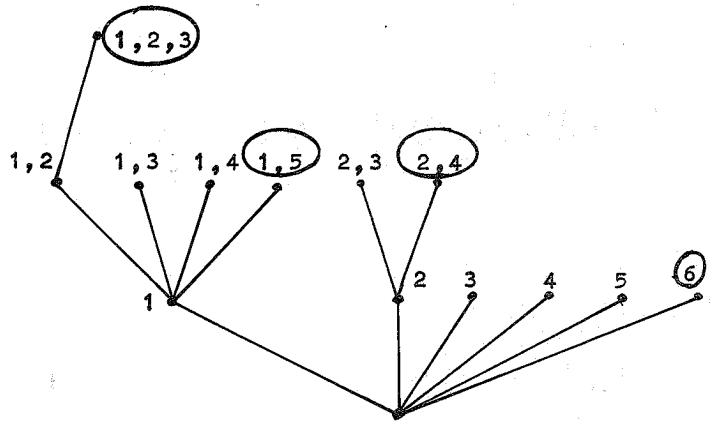
Natuurlijk zal men de som a[1] + ... + a[k] niet telkens opnieuw als som van k termen uitrekenen. Men bepaalt deze som steeds gemakkelijk met behulp van voorafgaande waarden ervan.

3.1.6. Het programma van 3.1.4. kan nu worden gespecialiseerd voor het probleem van 3.1.5. Teneinde over een kleine beginmoeilijkheid heen te komen introduceren we een overtollige a[0] die bij de aanvang de waarde 0 krijgt en maar één keer wordt geraadpleegd, nl. de eerste keer dat "two" wordt uitgevoerd.
Het volgende programma is in ALGOL 60 geschreven met uitzondering van PRINT (a[1], ..., a[k]) en PRINTTEXT (≠ klaar ≠).

```
    begin integer g; g := read;
        begin integer k, s; integer array a[0 : g];
one:        k := 0; s := 0; a[0] := 0;
two:        if s + a[k] < g then
                begin a[k + 1] := a[k] + 1; k := k + 1;
                    s := s + a[k]; goto two end;
three:      if s = g then PRINT (a[1], ..., a[k]);
four:       if s < g then begin a[k] := a[k] + 1; s := s + 1;
                                goto two end;
five:       if k > 1 then begin s := s - a[k];
                                k := k - 1; goto four end;
six:        PRINTTEXT (≠ klaar ≠)
        end
    end
```

Als men goed nagaat wat er gebeurt blijkt dat hier en daar wat overbodig werk wordt verricht en dat het programma door middel van kleine wijzigingen kan worden verbeterd. We laten zulks achterwege omdat het ons hier slechts om het wezen van de methode te doen was.

3.1.7. Ter illustratie geven we voor het geval g = 6 de boom aan, alsmede een overzicht van wat er allemaal gebeurt bij het uitvoeren van het programma uit 3.1.6. In elke regel van deze lijst staat een label aangegeven en daarachter de veranderingen die in de waarden van k, s, a[1], a[2], a[3] zijn opgetreden na uitvoering van de bij dat label behorende opdracht.



Uitvoering van het programma 3.1.6 met g = 6

	label	k	s	a[1]	a[2]	a[3]	print		label	k	s	a[1]	a[2]	a[3]	print
1	one	0	0					23	three						
2	two	1	1	1				24	four		6		4		
3	two	2	3		2			25	two						
4	two	3	6			3		26	three						2,4
5	two							27	four						
6	three						1,2,3	28	five	1	2				
7	four							29	four		3	3			
8	five	2	3					30	two						
9	four		4		3			31	three						
10	two							32	four		4	4			
11	three							33	two						
12	four		5		4			34	three						
13	two							35	four		5	5			
14	three							36	two						
15	four		6		5			36	three						
16	two							38	four		6	6			
17	three						1,5	39	two						
18	four							40	three						6
19	five	1	1					41	four						
20	four		2	2				42	five						
21	two	2	5		3			43	six						klaar
22	two														

3.1.8. Een belangrijk facet van backtracking is dat het geheugen van de machine niet de gehele boom te onthouden krijgt: de computer behoeft alleen te onthouden het punt dat onderzocht wordt, het rijtje punten dat er ligt tussen dat punt en de wortel, en de regels volgens welke de boom is opgebouwd.

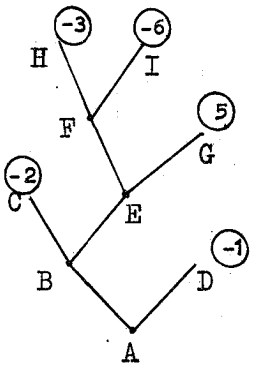
3.2. Een klasse van tweepersoonsspelen

Iets wat niet precies onder de eerder beschreven backtracking valt maar een soortgelijke wandeling op een boom meebrengt, is de hier volgende analyse van een algemene klasse van tweepersoonsspelen.

3.2.1. Gegeven is een boom, die we ons weer in een plat vlak voorstellen met een onderaan liggende wortel en oprijzende takken. De punten van waaruit geen takken meer naar boven gaan noemen we toppen. Bij elke top staat een bedrag vermeld.

Jan en Piet spelen het volgende spel. Er is één fiche, die in de aanvang bij de wortel ligt. Jan en Piet zetten om de beurt; Jan begint. Een zet bestaat uit het naar boven schuiven van de fiche naar één van de direct erboven liggende punten, naar keuze van de speler. De speler die een top bereikt beeindigt daarmee het spel, en krijgt het daar vermelde bedrag door de ander uitgekeerd. Gevraagd wordt welk bedrag Jan minstens in de wacht kan slepen.

3.2.2. We laten een spelanalyse zien aan de hand van het volgende voorbeeld.

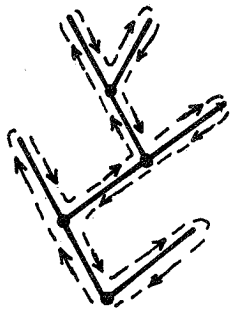


De uitkeringen staan er in kringetjes bijgeschreven. De waarde voor de beginspeler kan als volgt worden bepaald. Oorspronkelijk zijn alleen de toppen geëvalueerd. Vervolgens evalueert men al die punten van waaruit elke zet tot een reeds geëvalueerd punt leidt (hier alleen F), en waardeert zo'n punt met het tegengestelde van het maximum van de direct erboven liggende punten (F

krijgt dus de waarde - max(-3, -6) = 3. Dit is kennelijk de waarde van het spel voor de speler die naar F toe schuift (het voor hem ongunstigste geval is dat zijn tegenstander vervolgens naar H zet). We kunnen dus doen alsof F een top is met uitkering 3, en de evaluaties voortzetten. In derde instantie krijgt E de waarde - max(3,5) = -5, in vierde instantie krijgt B de waarde - max(-2,-5) = 2, in vijfde instantie krijgt A de waarde - max(2,-1) = - 2. Dit is de waarde voor de speler die naar A toe zou schuiven (zo dit mogelijk ware); de waarde voor Jan is dus + 2.

3.2.3. We willen een algorithme hebben die de waarde van het spel bepaalt zonder de computer met het onthouden van de gehele boom te belasten. We stellen ons ook niet voor dat de boom expliciet bekend is, maar dat wel regels bekend zijn volgens welke de boom is opgebouwd en volgens welke de uitkeringen zijn berekend.

De berekening van de waarde van het spel lukt met de volgende algorithme waarbij de boom op andere wijze wordt doorlopen dan in 3.1.2 werd beschreven, nl. als volgt



Ter vereenvoudiging van ons programma onderstellen we dat de "uitkering" ook is gedefinieerd in de punten die geen top zijn; in die punten wordt er de waarde ∞ aan toegekend. Verder voeren we een integer k in die bij een te beschouwen punt het aantal etages boven de wortel aangeeft, en een array w[0], w[1], ..., waarvan de elementen voorlopige spelwaarden in zekere punten aangeven.

Men kan zich aan voorbeelden gemakkelijk overtuigen dat de volgende algorithme de spelwaarde bepaalt. Het leveren van een formeel bewijs dat de algorithe altijd doet wat men verlangt, is (zoals trouwens bij vele algorithmen) veel minder eenvoudig.

```
1.  k := 0; w[0] := ∞;
2.  if geen stap mogelijk then goto 7;
3.  doe stap;
4.  if deze stap was opwaarts then begin k := k+1; w[k] := uitkering end
5.  else begin k := k-1; w[k] := min[w[k], - w[k+1]] end;
6.  goto 2;
7.  waarde voor Jan := - w[0].
```

3.3. Kortste boom tussen N punten

3.3.1. Het probleem

Twee punten kan men door een enkele punt-punt-verbinding aan elkaar verbinden, drie punten kan men door twee punt-punt-verbindingen tot een samenhangend geheel maken, N punten kan men door N-1 punt-punt-verbindingen tot een samenhangend geheel verbinden. Een dergelijk stel van N-1 punt-punt-verbindingen (ook wel "takken" genaamd) heet "een boom tussen deze N punten". Cayley heeft voor het eerst bewezen, dat het aantal mogelijke bomen tussen N punten gelijk is aan

$$N^{N-2} .$$

Als nu van elke der N * (N-1)/2 mogelijke takken de lengte gegeven is en men de lengte van een boom definieert als de som van de lengtes der takken, waaruit hij is opgebouwd, dan kan men zich afvragen, welke boom de kortste is. Wij nemen gemakshalve aan, dat alle gegeven lengtes verschillend zijn. Het zal blijken dat hieruit volgt dat de kortste boom eenduidig bepaald is. Wanneer aan dit gegeven niet is voldaan kan de kortste boom toch nog wel eenduidig bepaald zijn. Als de kortste boom niet eenduidig bepaald is, leidt het door ons te beschrijven proces toch nog tot één van de kortste bomen.

Opm. De punten hoeven niet in een plat vlak met een euclidische metriek te liggen. De gegeven taklengten hoeven zelfs niet aan de driehoeksongelijkheid te voldoen.

Wij achten de punten genummerd van 1 t/m N; wij achten de lengten der takken gegeven door een afstandentabel, die als

```
array afstand[1 : N, 1 : N]
```

gearrangeerd is en wel zo, dat voor $1 \leq i, j \leq N$

afstand[i,j] = lengte van de tak tussen punten i en j.

De waarden van de elementen van het array "afstand" moeten voldoen aan

afstand[i,j] = afstand[j,i] .

Het gevraagde antwoord is een boom van N - 1 takken, elke tak is bepaald door de nummers van zijn eindpunten: het gevraagde antwoord bestaat dus uit een (ongeordende) verzameling (ongeordende) nummerparen. Wij zullen de takken afleveren, genummerd in de volgorde, waarin is vastgesteld, dat ze deel uitmaken van de kortste boom; tevens zullen we de nummerparen zelf geordend afleveren: aan elke tak van de boom zal onze algorithme een begin en een eind toekennen. Wij stellen ons voor het antwoord af te leveren in twee arrays

"array van [1 : N - 1]" en "array naar [1 : N - 1]"

zodat "van[k]" en "naar[k]" begin- resp. eindpunt van de k-de tak aangeven.

Opm. Een dergelijke "overordening" is een normaal verschijnsel, zowel in het feitenmateriaal als in de voorgeschreven volgorde, waarin overigens verwisselbare acties plaatsvinden. Een en ander houdt ten nauwste verband met de lineaire ordening der geheugenplaatsen en het sequentiele karakter van automaten.

3.3.2. De oplossingsmethode

De volgende stelling stelt ons in staat om een kortste boom tak voor tak op te bouwen.

De stelling luidt, dat de kortste punt-punt-verbinding die van enig punt in een deelboom van een kortste boom leidt naar enig punt buiten de deelboom, eveneens deel uitmaakt van die kortste boom. (Een deelboom van een boom B is een verzameling van k der N punten, met $1 \leq k < N$, met een boom tussen die k punten, gevormd uit takken van B.)

Beschouwt men nl. een boom op de N gegeven punten, waarvan genoemde deelboom eveneens een deelboom is, maar waarin de verbinding in kwestie niet als tak voorkomt, dan kan deze geen kortste boom zijn. Immers, door de verbinding in kwestie toe te voegen ontstaat een cyclus met twee aan de deelboom gehechte takken, de tak in kwestie en een andere, die blijkens de veronderstelling langer is dan de tak in kwestie. Door de andere tak weg te nemen, wordt de cyclus opengebroken, maar wordt de totale samenhang niet verbroken. Er is dus weer een boom ontstaan, maar omdat de toegevoegde tak in kwestie korter

was dan de verwijderde andere tak, kan de oorspronkelijk beschouwde boom geen kortste zijn geweest.

Als de deelboom p punten bevat, dan identificeert genoemde stelling de toe te voegen tak als de kortste uit een verzemeling van p * (N-p) takken, nl. alle takken van enig punt in de deelboom naar enig punt daarbuiten.

Administratief kan dit zoekproces versneld worden door voor elk der N-p punten bij te houden de kortste tak naar de deelboom. Enerzijds is het gezochte minimum de kortste van deze N-p "kortste takken naar de deelboom", anderzijds is bij verbinding van een nieuw punt aan de deelboom door toevoeging van de p-de tak daaraan voor de resterende N-p-1 punten erbuiten gemakkelijk opnieuw "de kortste tak naar de deelboom" vast te stellen. Voor elk van de resterende N-p-1 punten is dit nl. of dezelfde tak als voorheen, dan wel de tak, die zo'n punt verbindt met het zojuist aan de deelboom toegevoegde nieuwe punt, en wel de kortste van de twee.

Rest ons te beslissen, hoe we voor algemene waarde van p aangeven, welke takken behoren tot de inmiddels gevormde deelboom en welke N-p takken "kortste takken naar de deelboom" zijn. We spreken af, dat de takken met eindpunten

"van[i]" en "naar[i]"

voor $i \leq p$ zullen behoren tot de deelboom en voor $i > p$ zullen behoren tot de verzameling "kortste takken naar de deelboom". Deze conventie impliceert, dat bij toevoeging van een nieuwe tak in de regel in beide arrays een verwisseling moet worden uitgevoerd.
Als we het beginpunt eenmaal hebben gekozen, leidt herhaalde toepassing van onze stelling langs éénduidige weg tot een kortste boom, Daar dit beginpunt in elke boom voorkomt zien we nu dat er slechts één kortste boom is.

3.3.3. Het programma

Het programma werkt onder de veronderstelling van de existentie van (zie 3.3.1.)

"array afstand[1 : N, 1 : N]", bevattende de afstandentabel;

"array van [1 : N-1]" en "array naar [1 : N-1]" als werkruimte tijdens het proces en als plaats voor het eindantwoord;

verder de individuele variabelen:

"k" ter telling van het aantal takken, dat zich in de deelboom bevindt (dus k = p - 1, als p het aantal punten is);

"j" ter sequentiële aftasting van de N - k - 1 "kortste takken naar de deelboom";

"min" en "h" als hulpgrootheden bij het zoekproces naar de kortste uit de verzameling "kortste takken naar de deelboom" en bij de verwisseling;

"nieuw" ter identificatie van het laatst in de deelboom opgenomen punt.

De variabele "N" fungeert in dit programma als constante.

De overgang van k naar k + 1 wordt bewerkt in drie stappen:

1) Uit de resterende N - k - 1 takken wordt de kortste gezocht (nr.h).
2) Deze wordt aan de deelboom toegevoegd en k wordt met 1 verhoogd.
3) De lijst van N - k - 1 kortste takken naar de deelboom wordt het het oog op de laatste toevoeging bijgewerkt.

Uit deze opsomming blijkt, dat de toevoeging van de laatste tak (van k = N - 2 naar k = N - 1) een lege stap is.

Het programma begint met de toestand te scheppen in overeenstemming met k = 0, dwz. een deelboom zonder takken, bevattende slechts een enkel punt. Deze keuze is geheel vrij, we kiezen hiervoor punt N.

Wat de feitelijke tekst betreft, merken we nog het volgende op.

In de eerste drie regels wordt de initialisatie verzorgd; in de eerste vijf regels vanaf label "herhaal" wordt stap 1 verzorgd; in de daaropvolgende zes regels stap 2 en in de volgende drie regels stap 3.

In het programma komen enkele "for statements" voor; een "for statement" als

```
for j := 1 step 1 until N do < statement >;
```

is, wat het effect betreft, equivalent met het volgende stukje programma:

```
        j := 1
test:   if j > N then goto out;
        < statement >;
        j := j + 1;
        goto test;
out:
```

Behalve van het "for statement" is ook in andere opzichten gebruik gemaakt van ALGOL 60.

```
        k := 0;
        for j := 1 step 1 until N-1 do begin van [j] := N; naar [j] := j end;
        goto test;

herhaal: h := k+1; min := afstand [van [h], naar [h] ];
        for j := k+2 step 1 until N-1 do
            begin if min > afstand [van [j], naar [j] ]
                  then begin h := j ; min := afstand [van [h], naar [h] ] end
            end ;

        k := k+1;
        if k ≠ h then begin min := van [k]; van [k] := van [h]; van [h] := min;
                            min := naar [k]; naar [k] := naar [h];
                            naar [h] := min
                      end;
        nieuw := naar [k];
        for j := k+1 step 1 until N-1 do
        if afstand [nieuw, naar [j] ] <
        afstand [van [j], naar [j] ] then van [j] := nieuw;

   test: if k < N-2 then goto herhaal;
```

errata 61, 63, 71

4. Over de structuur van rekenautomaten

4.1. Inleiding

Om een inzicht te krijgen in de structuur van rekenautomaten zullen wij beginnen met een grove analyse van het soort taken, waarvoor dit soort apparatuur gesteld is.

Om het ons niet te moeilijk te maken, denken we maar even speciaal aan "rekenopgaven" zoals b.v. het inverteren van een matrix van 100 bij 100.

Het eerste dat bij een dergelijke opgave opvalt is dat hij de rekenautomaat confronteert met twee soorten "veelheid"
1) in een dergelijk rekenproces moet een grote hoeveelheid getallenmateriaal beheerd worden: voor het vastleggen van een matrix van 100 bij 100 heeft men al 10000 getallen nodig
2) in een dergelijk rekenproces moet heel veel gedaan worden: voor de inversie van een matrix van 100 bij 100 zijn circa 1000000 vermenigvuldigingen nodig.

De constructie van apparatuur, die tegen deze beide soorten van "veelheid" is opgewassen behoort ongetwijfeld tot een van de grote triomfen der techniek uit de laatste decennia. Deze constructie is mogelijk geweest door te werken volgens het aloude principe "Verdeel en heers", en de rekenautomaat op te bouwen uit twee duidelijk gescheiden gedeelten, een passief gedeelte en een actief gedeelte, elk met de specifieke taak om tegen een van de veelheidsaspecten opgewassen te zijn.

4.2. Globaal overzicht van de belangrijkste componenten

Het actieve gedeelte - het zg. rekenorgaan - is het gedeelte, waarin het werk verzet wordt, waarin optellingen, aftrekkingen, vermenigvuldigingen, delingen, vergelijkingen, etc. worden uitgevoerd. Het is de specifieke taak van het rekenorgaan om zoveel mogelijk werk te verzetten: bij de constructie ervan wordt dan ook de meeste nadruk gelegd op de snelheid waarmee de diverse bewerkingen worden uitgevoerd. Het is heel duidelijk niet de taak van het rekenorgaan om "groot" te zijn: op elk moment zijn als regel dan ook maar heel weinig getallen - als operand voor of uitkomst van bewerkingen - in de directe activiteit van het rekenorgaan betrokken.

Het passieve gedeelte - het zg. geheugen - heeft heel specifiek de taak om groot te zijn, om grote hoeveelheden getallenmateriaal te kunnen herbergen: het fungeert bv. als "ijskast" voor al die tussenresultaten die op dat moment

niet in de directe activiteit van het rekenorgaan betrokken zijn. Het getallenmateriaal is overigens niet de enige informatiehoeveelheid, die in het geheugen moet worden opgeslagen: daarnaast hebben we nl. het programma, dat vastlegt, welke sequens van bewerkingen dan wel door het rekenorgaan op dit getallenmateriaal moet worden uitgevoerd. Voor de duur van het rekenproces zal het geheugen ook het programma moeten herbergen: zoals op elk moment slechts een fractie van het getallenmateriaal in actieve bewerking is, zo doet op elk moment ook slechts een fractie van het programma in het rekenorgaan zijn sturende invloed gelden.

De zojuist geschilderde taakverdeling tussen het actieve rekenorgaan en het passieve geheugen impliceert natuurlijk wel, dat er tussen beide onderdelen een intensief informatieverkeer plaats vindt en wel in beiderlei richting. Van het rekenorgaan naar geheugen vindt informatietransport plaats, iedere keer dat een zojuist gevormd tussenresultaat met het oog op latere relevantie moet worden opgeslagen, in de omgekeerde richting vindt informatietransport plaats voorzover het in het geheugen opgeslagen programma zich moet doen gelden door het rekenorgaan de volgende bewerking voor te schrijven en voorzover zolang in het geheugen opgeslagen tussenresultaten weer als operand van een bewerking moeten optreden.

Enige reflectie leert ons, dat hiermee de kous nog niet helemaal af is. Als ik bv. 20 vragen met "ja" of "nee" moet beantwoorden (in de trant van "Bent U getrouwd?" "Bent U Nederlander?" "Bent U in een ziekenfonds opgenomen?" etc.) dan is het niet voldoende als ik mijn antwoord geef in de vorm van 12 ja's en 8 nee's (bij wijze elk op een los velletje geschreven): elk antwoord is immers slechts interpreteerbaar, als bekend is, bij welke vraag het behoort! In de normale enquêtepraktijk beantwoordt men die vragen dan ook allen op eenzelfde formulier en de plaats op het formulier, waar men "ja" dan wel "nee" invult bepaalt steeds, welke vraag men hiermee dan wel beantwoordt. De vragen hadden ook genummerd kunnen zijn (bv. v/m 0 t/m 19) en onderaan het formulier vindt men dan de te retourneren antwoordstrook met 20 vakjes, eveneens genummerd v/m 0 t/m 19, waarin men de bijbehorende vragen kan beantwoorden. Op de antwoordstrook worden de onderscheiden antwoorden door deze nummers geïdentificeerd (als de huizen in een straat!): zulke identificerende nummers heten "adressen".

Het geheugen van een rekenautomaat herbergt de getallen niet als knikkers in een zak, het is veeleer analoog aan de antwoordstrook. Het is onderverdeeld in genummerde vakjes (gewoonlijk "geheugenplaatsen" genoemd), elk geïdentifi-

ceerd door zijn rangnummer, zijn "adres". Elk informatietransport van en naar het geheugen geschiedt immer onder opgave van het adres van de geheugenplaats, die in dit transport betrokken is. Het is de functie van de zg. "selectie" om in het geheugen het contact te leggen met de door het adres aangewezen geheugenplaats en het informatietransport in de gewenste richting te doen verlopen. Om een en ander te onderstrepen spreekt men daarom soms expliciet van "een adresseerbaar geheugen".

Opmerking 1. Op een antwoordstrook kan men zich grote en kleine vakjes voorstellen, in het geheugen van een rekenautomaat hebben de geheugenplaatsen altijd uniform dezelfde, eindige capaciteit, gegeven door het eindige aantal verschillende toestanden, waarin de geheugenplaats zich kan bevinden. Zou dit = 2 zijn, dan kan zo'n vakje gebruikt worden om een van de antwoorden "ja" of "nee" te onthouden, is het aantal bv. 2000000000, dan kunnen we zo'n vakje bv. gebruiken om een van de gehele getallen v/m - 1000000000 t/m 999999999 te herbergen.

Opmerking 2. Het informatietransport naar het geheugen wordt ook wel "schrijven op een geheugenplaats" genoemd: de geheugenplaats gaat daarmee over in een nieuwe toestand, de aanvankelijk hier vindbare informatie is daarmee verloren gegaan. Het informatietransport uit het geheugen wordt ook wel "lezen van een geheugenplaats" genoemd; de toestand, waarin die geheugenplaats zich bevindt gaat daarbij niet verloren, d.w.z. de hier aanvankelijk vindbare informatie blijft onveranderd voor latere referentie beschikbaar.

Opmerking 3. Bij de taakverdeling over een actief en een passief gedeelte hebben we het voorgesteld, alsof het actieve gedeelte - het rekenorgaan - alleen maar snel en het passieve gedeelte - het geheugen - alleen maar groot hoefde te zijn. Dit laatste is alleen juist, zolang het geheugen geen informatie hoeft op te nemen of af te geven, d.w.z. zolang de selectie er niet op ingrijpt. Van het duo "geheugen met selectie" is de snelheid, waarmee informatie van of naar een geheugenplaats getransporteerd kan worden een wezenlijke eigenschap, omdat hierdoor een bovengrens gesteld is aan de intensiteit van het informatieverkeer, dat tussen dit duo en zijn omgeving kan plaats vinden. Het is gebruikelijk om kortweg over "de snelheid van een geheugen" te spreken: men bedoelt dan altijd een of ander snelheidsaspect van dit duo.

Voordat we een en ander zullen vertellen over de technische realiseringen, gaan we de rekentaak eerst wat gedetailleerder analyseren. De bespreking van de zg. randapparatuur, d.w.z. de apparatuur om informatie de machine in en uit te krijgen, zullen we eveneens uitstellen.

4.3. De sequentialisering

In deze sectie bespreken we de gevolgen van het feit, dat het rekenorgaan "snel maar klein" is. De kleinheid van het rekenorgaan komt daarin tot uiting, dat op elk moment het rekenorgaan slechts actief bezig is met een enkele bewerking (zoals optellen, aftrekken, etc.) en dus slechts operanden en resultaat van een enkele bewerking hoeft te herbergen. Dit heeft tot gevolg, dat het rekenorgaan steeds slechts een enkele bewerking tegelijkertijd uitvoert, zodat alle rekenkundige bewerkingen, die nodig zijn, na elkaar uitgevoerd moeten worden, ook al zouden ze, logisch gezien, simultaan kunnen plaats vinden. Een en ander is natuurlijk een extra pressie om de individuele rekenkundige bewerkingen dan wel zo snel mogelijk te doen plaats vinden. (Volledigheidshalve vermelden we, dat er wel pogingen gedaan zijn om hogere totale rekensnelheden te halen door rekenorganen te construeren, die meer bewerkingen tegelijkertijd kunnen uitvoeren, maar in hun huidige vorm zijn deze pogingen niet al te overtuigend. Wij zullen ons in het volgende daarom beperken tot het strikt sequentiële rekenorgaan.)

Wij zullen ter illustratie laten zien, hoe de berekening van willekeurig ingewikkelde algebraïsche uitdrukkingen zich op systematische wijze laat sequentialiseren, Heel duidelijk zal hierbij naar voren komen, dat het beroep dat hierbij op het geheugen gedaan wordt, van deze tijdsrangschikking der arithmetische bewerkingen afhangt. (Immers: het geheugen fungeert zoals gezegd als ijskast, waarin men tussenresultaten die gevormd werden lang voordat ze gebruikt worden, voor de tijdsduur kan "conserveren".)

Wij zullen een en ander eerst laten zien aan de hand van een welgekozen voorbeeld, nl. de arithmetische uitdrukking

"$(a \times b - c \times d)/(c \times e - d \times (a + f \times g))$"

Om de berekening straks op de voet te volgen, zullen we de volgende waarden van de variabelen aannemen:

$a = 7 \quad b = 17 \quad c = 4 \quad d = 5 \quad e = 32 \quad f = 3 \quad g = 6$

Wij nemen nu aan, dat de arithmetische bewerkingen op getallen gedefinieerd zijn (m.a.w.: "dat het rekenorgaan kan rekenen"), en we zullen analyseren, wat er achtereenvolgens gebeuren moet, opdat voor elke bewerking beide operanden als getal ter beschikking komen. Zodra van een operator beide operanden als getal ter beschikking zijn, zullen we de door de operator aangegeven betrekking ook onmiddellijk laten plaats vinden.

Verder zullen we de volgorde van links naar rechts zoveel mogelijk aanhouden, in het bijzonder van elke operator eerst de linkeroperand en dan de rechteroperand bepalen.

Van buiten naar binnen lezend presenteert de expressie zich in eerste instantie als een quotient, dat we slechts berekenen kunnen, als eerst deeltal en deler berekend zijn. We richten onze aandacht volgens afspraak eerst op het deeltal

$$a \times b - c \times d$$

dat zich presenteert als een verschil, dat echter pas gevormd kan worden als beide termen berekend zijn. We richten onze aandacht volgens afspraak eerst op de linkerterm

$$a \times b$$

die zich presenteert als product. Ook dit product is niet zonder meer berekenbaar, we kunnen immers niet "a" met "b" vermenigvuldigen, we kunnen slechts de waarde van a met de waarde van b vermenigvuldigen. Weer richten we onze aandacht in volgorde van links naar rechts op de factoren en we krijgen in opeenvolging de drie handelingen:
"pak de waarde van a; pak de waarde van b; vermenigvuldig"

Wij zullen deze opeenvolging, waar het ons voorlopig toch alleen maar gaat om de volgorde, verkort aangeven door

"a b ×"

waarmee de eerste term berekend is.
Hierna wordt de tweede term op dezelfde wijze berekend, nl.

"c d ×"

waarop het moment gekomen is dat de aftrekking kan worden uitgevoerd. De totale handelingssequens, die leidt tot de vorming van de numerieke waarde van het deeltal laat zich dus representeren door

"a b × c d × -" ,

de berekening van de hele expressie impliceert op deze wijze een tijdsopeenvolging van handelingen, die door

"a b × c d × + c e × d a f g × + × - /"

wordt weergegeven.

Opmerking. In tegenstelling tot de "infixnotatie", waar de operatoren tussen de operanden gezet worden, heet de bovengebezigde notatie "postfix" omdat de operatoren achter hun bijbehorende operanden gezet worden. Deze notatie heet ook wel "reversed Polish" ter onderscheiding van de praefixnotatie, waar men de operatoren vóór de operanden noteert, en die wel "Polish notation" genoemd wordt ter ere van de Poolse wiskundige Luckasciewisz.

Voorbeelden: infix a + b c
postfix a b c × +
praefix + a × b c

(De praefixnotatie laat zich lezen als "de som van a en het product van b en c").

Wij keren terug tot de expressie, zoals deze in postfixnotatie gegeven is. Omdat deze notatie zo mooi aangeeft, wat er in volgorde gebeuren moet, rijst natuurlijk onmiddellijk de vraag, of we niet juist aan de hand van de representatie in postfixnotatie de waarde van de expressie prettig zouden kunnen uitrekenen.
Dit kan inderdaad met behulp van een zg. "stapel". Bij een lineair geordende verzameling van elementen waarbij slechts aan één kant elementen toegevoegd of afgenomen worden, spreekt men van een stapel: deze kant heet "de top", de andere kant heet "de bodem".

Het recept om aan de hand van een expressie in ~~praef~~post fixnotatie de waarde te berekenen luidt nu als volgt. Men verwerkt de expressie in volgorde van links naar rechts, eenheid voor eenheid (waarbij een "eenheid" is óf de aanduiding van een variabele of die van een operator). Is de eenheid de aanduiding van een variabele, dan wordt de numerieke waarde van de variabele aan de stapel toegevoegd, is de eenheid de aanduiding van een (binaire) operator, dan wordt de operator op de twee topelementen van de stapel uitgevoerd, die van de stapel worden afgenomen en door de uitkomst vervangen worden. In de hieronder volgende uitwerking geven we regel voor regel het stapelbeeld (bodemzijde links); helemaal vooraan de regel staat de expressie-eenheid vermeld, die dit stapelbeeld gegenereerd heeft.

a : 7
b : 7 17
× : 119
c : 119 4
d : 119 4 5
× : 119 20

```
- :  99
c :  99   4
e :  99   4    32
x :  99   128
d :  99   128  5
a :  99   128  5    7
f :  99   128  5    7   3
g :  99   128  5    7   3   6
x :  99   128  5    7   18
+ :  99   128  5    25
x :  99   128  125
- :  99   3
/ :  33
```

Waarmee inderdaad correctelijk de waarde van de expressie bepaald is (en aan de stapel is toegevoegd).

Opmerking. In de postfixnotatie moet men verschillende symbolen hebben voor het binaire minteken (als in "a - b") dat duidt op aftrekking en het unaire minteken (als in "(- a x b + q)") dat slechts duidt op tekenwisseling. De unaire operator opereert slechts op het topelement van de stapel. Het worteltekenis een andere voor de hand liggende unaire operator.

Het bewijs, dat elke arithmetische expressie in postfixnotatie geschreven kan worden en dat het beschreven rekenvoorschrift aan de hand van de postfixnotatie altijd correct de waarde aan de stapel toevoegt, laten wij gaarne aan de twijfelende lezer over.

4.4. Programmanotatie in 1-adrescode

In de postfixnotatie hebben we een methode gevonden niet alleen om een arithmetische expressie vast te leggen, maar zelfs om voor te schrijven hoe elke willekeurige arithmetische expressie metterdaad moet worden gebruikt: wij kunnen een dergelijke expressie in postfixnotatie beschouwen als een stukje programma. Het voldoet aan de eis, dat deze beschrijving gescandeerd is in "programma eenheden" (nl. aanduidingen van variabelen, dan wel van arithmetische operaties) waaraan steeds een duidelijke actie beantwoordt en dat deze programma eenheden stuk voor stuk in behandeling genomen dienen te worden. De technische naam voor de programma eenheden, waarin het programma "verknipt" wordt, luidt "opdrachten", het programma doet steeds slechts opdracht voor opdracht zijn invloed gelden.

In rekenmachines dient de berekening van arithmetische expressies uiteindelijk herleid te worden tot een successieve uitvoering van opdrachten uit het voor deze rekenmachine karakteristieke opdrachten repertoire. Er bestaan inderdaad enige typen rekenmachines (zg. "machines met ingebouwde stapel") waarbij het opdrachten repertoire zodanig is, dat de postfixnotatie op de voet kan worden gevolgd. Dergelijke opdrachten repertoires bevatten dan q.q. twee heel verschillende soorten operaties, nl. die, welke wel naar een geheugenplaats verwijzen maar geen gereken ten gevolge hebben (aangeduid met "pak de waarde van") en die, die wel gereken ten gevolge hebben, maar niet naar een geheugenplaats verwijzen (aangeduid met de overeenkomstige arithmetische operator). Wij zullen nu laten zien, hoe de berekening van een arithmetische expressie ook kan worden uitgedrukt door een opdrachtensequens, waarin elke opdracht naar een geheugenplaats verwijst.

Wij beschouwen hiertoe een machine met een expliciet benoembaar register in het rekenorgaan (wij zullen dit met "R" aanduiden) en een opdrachten repertoire, waarin elke opdracht zowel een operatie als een geheugenplaats aanwijst. De geheugenplaatsen zullen gebruikt worden om variabelen en tussenresultaten te onthouden, wij zullen ze met kleine letters (a, b, c, ..., t) eventueel gevolgd door een rangnummer, aanduiden. De expressie uit ons voorbeeld is dan programmeerbaar met behulp van opdrachten uit het volgende repertoire

R := a	a := R
R := -a	a := -R
R := R + a	a := a + R
R := R - a	a := a - R
R := a - R	a := R - a
R := R × a	a := R × a
R := R/a	a := a/R
R := a/R	a := R/a

In de linkerkolom staan de opdrachten, waarbij onder raadpleging van de inhoud van een geheugenplaats de inhoud van het R-register herdefinieerd wordt, in de rechterkolom staat het inverse stel, waarin de rol van register en geheugenplaats verwisseld zijn.

Wij zullen met behulp van opdrachten uit bovengegeven repertoire de expressie uit het vorige voorbeeld beschrijven, en wel in twee versies: in de linker versie zullen we de volgorde van de postfixnotatie getrouwelijk volgen. De geheugenplaatsen, nodig om tussenresultaten te stapelen moeten nu expliciet worden aangegeven ("t1", "t2", etc.).

```
R := a
R := R × b
t1 := R
R := c
R := R × d
t1 := t1 - R
R := c
R := R × e
t2 := R

R := d
t3 := R
R := a
t4 := R
R := f
R := R × g
R := R + t4
R := R × t3

R := t2 - R
R := t1/R
```

of als

"(f × g + a) × d"

```
R := f
R := R × g
R := R + a
R := R × d
```

Wij merken hierbij nog het volgende op:
de feitelijke programmalengte is in deze representatie - in tegenstelling tot de postfixnotatie - dus wel afhankelijk van volgordeverwisselingen en tevens: de feitelijke programmalengte (en de mogelijkheid door verwisseling hierop te bekorten) hangt critisch af van welke opdrachten precies in het repertoire voorkomen.

Het opdrachten repertoire uit vele machines bevat niet ten onrechte uit de rechterkolom alleen de bovenste twee, d.w.z. de zuivere transporten: "a := R" en "a := -R". Wat nu als

"t1 := t1 - R"

gecodeerd is, wordt dan

```
"R := R - t1
t1 := -R" .
```

In de linkerkolom ontbreken vaak

"R := a - R"

en - wat nog hinderlijker is -

"R := a/R"

Zo wordt de programmering van de berekening van de expressie

a/(b + c)

met laatstgenoemde opdracht:

```
"R := b
 R := R + c
 R := a/R"
```

en zonder

```
"R := b
 R := R + c
 t1 := R
 R := a
 R := R/t1"
```

Tot slot van deze sectie een enkel woord over hoe een programma nu in het geheugen van de rekenautomaat wordt opgeborgen. Wij recapituleren, dat men zich het geheugen kan voorstellen als een groot aantal consecutief genummerde vakjes en dat elk vakje "een getal" kan herbergen. Hoe kunnen we dit geheugen nu gebruiken om een stuk programma te herbergen?

Om een programma te representeren moeten we twee dingen kunnen vastleggen: de individuele opdrachten waaruit het programma is opgebouwd en de volgorde, waarin ze uitgevoerd moeten worden.

De individuele opdrachten zijn representeerbaar, omdat deze wel op getallen kunnen worden afgebeeld. Immers: aan het begin van deze sectie hebben wij 16 verschillende opdrachtentypes gegeven, zoals die in het opdrachten repertoire van een rekenautomaat voor zouden kunnen komen. We kunnen ons alle voorkomende types (doorgaans enkele honderden) genummerd voorstellen: dit nummer noemen we "het opdrachtnummer", het legt het type van de opdracht vast. Daarnaast moet worden aangegeven op welke variabele de opdracht betrekking heeft: door aan elke variabele een geheugenplaats toe te kennen, kan men de erin te betrekken variabele identificeren door het adres van deze geheugenplaats, d.w.z. eveneens door een nummer. De beide componenten van de opdracht, nl. type en variabele zijn daarmee beide door een cijferrij vastgelegd. Men kan de hele opdracht eenduidig vastleggen, door deze twee cijferrijen achter elkaar te plaatsen, waarmee een manier gegeven is om de inhoud van een geheugenplaats een opdracht te laten representeren.

Naast de individuele opdrachten hebben we ook nog de plicht om hun volgorde te representeren; hier komt het feit, dat het geheugen uit genummerde vakjes bestaat, ons te hulp: de geheugenplaatsen in een geheugen hebben een natuurlijke volgorde, nl. de volgorde van oplopend adres! Opdrachten die achter elkaar uitgevoerd moeten worden, worden zoveel mogelijk op opeenvolgende geheugenplaatsen opgeborgen. De machine moet de opdrachten uitvoeren in de volgorde, waarin ze in het geheugen staan, incidentele uitzonderingen daargelaten, nl. de zg. sprongopdrachten, waarmee deze strikte koppeling tussen lexicografische volgorde (in het geheugen) en dynamische volgorde (in de tijd) eventueel verbroken kan worden, bv. om een rijtje opdrachten over te kunnen slaan (denk aan de if clause!).

4.5. De opdrachtcyclus

Naast registers voor numerieke tussenresultaten (expliciet hebben we er onder de naam "R" een ten tonele gevoerd) bevat het actieve gedeelte van de rekenautomaat als regel tenminste nog twee andere registers:
1) de opdrachtteller (met de capaciteit van een adres), bevattende het adres van de geheugenplaats, welke de volgende uit te voeren opdracht bevat;
2) het opdrachtregister (met de capaciteit van een heel woord), bevattende de opdracht, die nu onder uitvoering is.

Beschouwen wij het hele geheugen als één groot lineair array M, met de geheugenvakjes als elementen, dan kunnen we globaal de werking beschrijven als volgt

```
"terug: opdrachtregister := M[opdrachtteller];
        opdrachtteller := opdrachtteller + 1;
        uitvoering (opdrachtregister);
        goto terug"
```

In de eerste regel is aangegeven, dat onder controle van de opdrachtteller het opdrachtregister uit het geheugen gevuld wordt met de volgende opdracht die aan de beurt is; in de tweede regel wordt de opdrachtteller met 1 verhoogd, anticiperend, dat straks de opdracht uit de in het geheugen volgende geheugenplaats aan bod zal komen. In de derde regel is (heel globaal!) aangeduid dat de eigenlijke logica van de machine vanuit het opdrachtregister gestuurd wordt, dat dus per definitie "de heersende opdracht" bevat. Na voltooiing van de uitvoering van de heersende opdracht herhaalt de cyclus zich.

Opmerking 1. De bijzondere eigenschap van een sprongopdracht is, dat hij op de opdrachtteller kan opereren. Als bv. de functie van een opdracht is om

onder omstandigheden de opdrachtteller met een zeker bedrag (zeg "N") te verhogen, dan betekent dit dynamisch, dat onder die omstandigheden de volgende "N" opdrachten worden overgeslagen (denk weer aan de if clause!).

Opmerking 2. De uitvoering van de opdracht impliceert bijna altijd een tweede geheugencontact, nl. met de variabele, die in de opdracht is aangewezen.

4.6. Over de functie van een vertaler

In het voorafgaande hebben we in vogelvlucht gezien, hoe een rekenautomaat zijn werkzaamheden opdracht voor opdracht uitvoert en hoe hij dit slechts doen kan, mits door een programma in zijn geheugen dit rekenproces dan ook inderdaad opdracht voor opdracht beschreven wordt.

Dit betekent dan wel, dat in het geheugen van de rekenautomaat het programma op een heel andere manier gerepresenteerd wordt dan door de ALGOL-tekst, zoals die aanvankelijk door de programmeur is aangeboden. Om U een indruk te geven hoe aan de hand van een aangeboden ALGOL-tekst er uiteindelijk een machineprogramma in het geheugen terecht komt zodat de rekenmachine de gevraagde resultaten kan gaan berekenen, zullen wij - wederom in grote stappen - een aangeboden ALGOL-tekst op zijn weg vervolgen.

Eigenlijk wil de programmeur zijn programma, d.w.z. een rij "basic symbols" aanbieden. Hij kan dit niet, hij kan slechts een beschreven vel papier aanbieden. De afspraak is, dat hij (in de regel van links naar rechts en de regels van boven naar beneden) voor elk basic symbol een voldoend karakteristieke krabbel op papier zet - wat dus bv. impliceert dat hij voldoend onderscheidbare krabbels gebruikt voor het cijfer 0 en de kleine en grote letter O, voor het cijfer 1, de hoofdletter I en de kleine letter l, voor het maalteken en de letter x etc.

Een dergelijk manuscript is echter weinig geschikt voor automatische verwerking: bij de interpretatie van de "krabbels" wordt teveel een beroep gedaan op de goede wil van de lezer. Daarom wordt het manuscript een keer overgetikt, maar bv. op een zg. ponsende schrijfmachine. Dit is een schrijfmachine, waaraan een bandponser is gekoppeld zodat bij elke aangeslagen toets een voor deze toets karakteristieke configuratie van wel- en niet-gaatjes geponst wordt. Het resultaat is dat uit deze band eenduidig het paginabeeld is af te leiden, dat al overtikkend is ontstaan. En als bij dit overtikken geen lees-, tik- of ponsfouten zijn opgetreden (en dat hopen we nu maar) dan is uit de geproduceerde ponsband op eenduidige manier de door de programmeur bedoelde

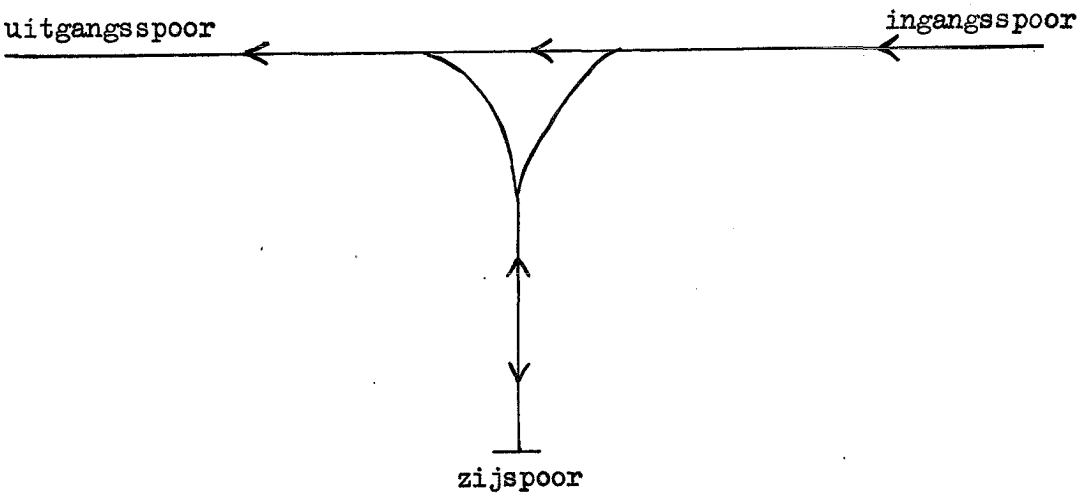
rij "basic symbols" af te leiden.
Dit is logisch al een vrij ingewikkeld proces. Ten eerste legt bij veel ponsende schrijfmachines het bandbeeld wel eenduidig het paginabeeld vast, maar niet andersom, nl. doordat de hoofdlettertoets en de kleine-lettertoets wel een ponsing, maar op zichzelf nog geen zichtbaar resultaat geven, doordat inspringen zowel door tabulatie als met spaties bewerkstelligd kan worden, doordat bij niet transporterende toetsen (onderstreping) elke aanslag wel een ponsing geeft, terwijl op het paginabeeld enkele onderstreping niet van "dubbele" is te onderscheiden. De eerste opgave is dus om uit het bandbeeld het paginabeeld te destilleren, maar het paginabeeld is, afgezien van layout, niet eenduidig door de rij basic symbols bepaald, zodra men voor (sommige) basic symbols om practische redenen meer dan 1 representatie wil toestaan (bv. naast "Boolean" ook "boolean", naast "go to" ook "goto" en "go to"). De herleiding van paginabeeld tot "rij van basic symbols" is dus ook niet triviaal. Tot zover over de vrij ingewikkelde manier waarop de ponsband het aangeboden ALGOL-programma vastlegt.

Mits onze rekenautomaat nu maar beschikt over ponsbandlezers, die hem in staat stellen om in ponsbanden vastgelegde informatie in zich op te nemen, dan kunnen dergelijke ponsbanden met ALGOL-programma's verder automatisch verwerkt worden.

Het proces, dat uit het bandbeeld het paginabeeld destilleert, uit het paginabeeld de rij basic symbols destilleert, het daardoor gerepresenteerde ALGOL-programma interpreteert en in het geheugen een equivalent programma in machinecode opbouwt heet "vertaling", is automatiseerbaar en kan door de rekenautomaat zelf worden uitgevoerd (mits dit proces door een passend programma, de zg. "vertaler", voor de automaat beschreven is).

De automatische verwerking van een ALGOL-programma bestaat dan uit twee gedeelten: de "vertaalfase", waarin een equivalent programma in machinecode opgebouwd wordt en daarna de "executiefase", waarin het feitelijke rekenwerk plaats vindt aan de hand van het in de vertaalfase geproduceerde machinecode programma. Uit de vertaalfase zullen wij ter illustratie een klein detail lichten, nl. de overgang van infixnotatie met haakjes tot postfixnotatie.

Het proces, dat in feite een permutatie opgave is, laat zich inderdaad als rangeeropgave beschrijven, en wel op een spoor als onder getekend.



Rechts op het ingangsspoor komt (tussen haakjes) de expressie in infixnotatie eenheid voor eenheid binnenrijden, links wordt via het uitgangsspoor de expressie eenheid voor eenheid in postfixnotatie geproduceerd.
De ~~rangregels~~ rangeer zijn nu als volgt

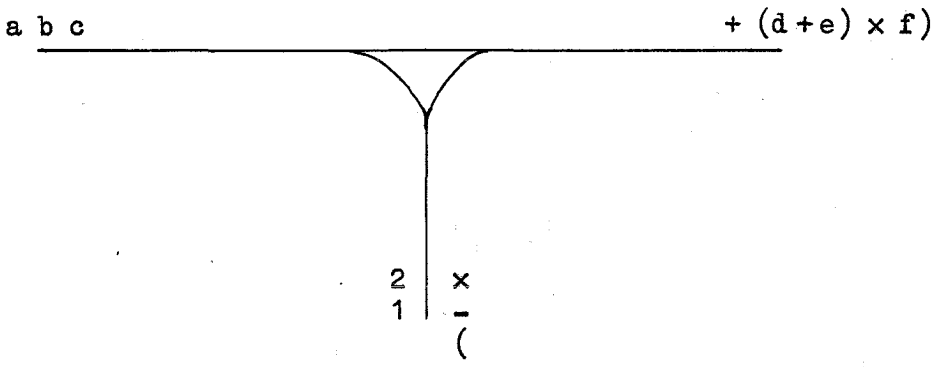
1) Een binnenkomende variabele wordt onmiddellijk rechtdoorgestuurd over het uitgangsspoor;
2) een binnenkomend openingshaakje wordt zonder meer het zijspoor opgerangeerd;
3) een operator krijgt om te beginnen een prioriteitsgetal toegekend en wel volgens de tabel:

↑	3
× en /	2
+ en -	1

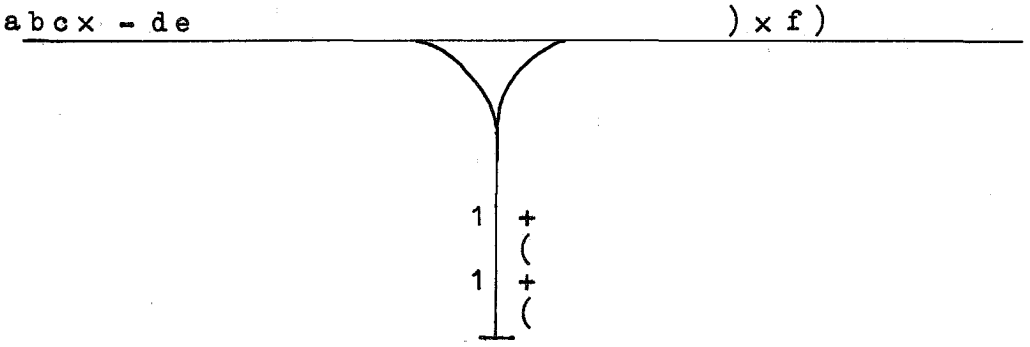
   en zal uiteindelijk het zijspoor opgerangeerd worden; voordien worden, indien aanwezig, echter alle operatoren boven in het zijspoor met gelijke prioriteit of hoger dan de nieuweling via het uitgangsspoor weggerangeerd;
4) een binnenkomend sluitingshaakje heeft tot gevolg dat, indien aanwezig, de operatoren boven in het zijspoor tot aan het openingshaakje via het uitgangsspoor worden weggerangeerd, waarna het binnenkomende sluitingshaakje en het openingshaakje (nu) boven in het zijspoor elkaar annihileren.

Als voorbeeld zullen wij laten zien

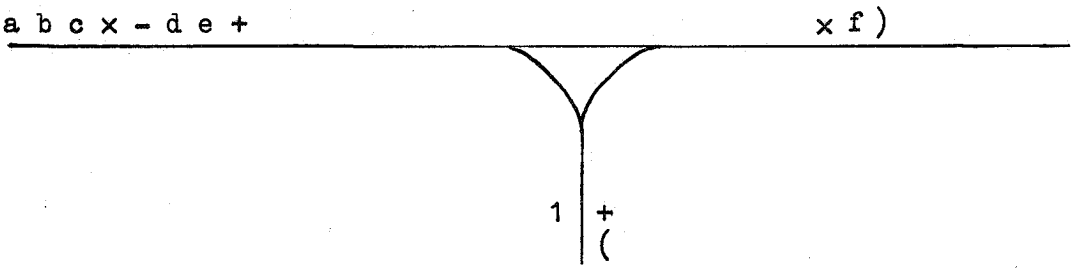
$$(a - b \times c + (d + e) \times f)$$



op dit moment, nu de "+" met prioriteitsgetal =1 aankomt wordt de * en de - uit het zijspoor weggerangeerd. Even later hebben we



wat tot



aanleiding geeft. De "x" gaat het zijspoor op zonder de "+" te verdrijven, zodat de postfixnotatie nog met

f x +

wordt aangevuld.

Opgave. Ga na dat het unaire - teken (dat in ⊖ vertaald worde) gedetecteerd wordt als het - teken in de oorspronkelijke expressie niet na een variabele of een sluitingshaak komt.

Opgave. Ga na, dat het unaire minteken op het zijspoor zowel prioriteitsgetal =1 als =2 mag hebben. Wat is het verschil?

Enkele karakteristieken van computers anno 1967

Aantal opdrachten dat per seconde wordt uitgevoerd: 10^3 (langzame computer) - 10^6 (zeer snelle computer).

Snelheid ponsband lezer: 200 - 2000 tekens per seconde.

Snelheid ponskaart lezer: 4 - 20 kaarten per seconde.

Snelheid regeldrukker (dit is de schrijfmachine uit 0.5): 20 regels per seconde.

Omvang onmiddellijk toegankelijk geheugen: 5000 - 500.000 getallen van 8 - 12 decimalen.

Omvang hulpgeheugen: 4 - 20 kasten met 10^6 getallen per kast. Een kast bevat een magneetband (looptijd van de band van begin tot eind 5 minuten) of magnetiseerbare schijven (wachttijd voor een op te bergen of af te lezen getal 0.1 seconde). Magneetband en schijven zijn in enkele minuten te vervangen voor andere.

5. Vraagstukken

Bij de vraagstukken is het de bedoeling, tenzij anders vermeld, dat de relevante gegevens op een band staan en ingelezen worden en de verlangde resultaten worden geprint.
Het is de bedoeling dat men bij het oplossen van deze vraagstukken een passend gebruik maakt van programmaonderdelen en procedures uit reeds eerder gemaakte vraagstukken en uit de syllabus.
Bij de vraagstukken uit groep A ligt de nadruk meer op het programmeren, bij de vraagstukken uit de groepen B en C meer op de analyse van het probleem.
Bij de vraagstukken uit groep C, die van een numeriek karakter zijn, ligt de kern van de te gebruiken algorithme meestal wel vast (Newton-Raphson, trapeziumregel en dergelijke), maar toepassing hiervan vraagt vaak grote zorgvuldigheid. Het maken van schetsjes van de optredende functies wordt aanbevolen.

A1. Schrijf een programma, dat n! berekent bij een gegeven natuurlijk getal n.

A2. Schrijf een programma, dat $\sum_{n=1}^{50} n^3$ berekent.

A3. Schrijf een programma, dat van de reeks $1 - \frac{1}{2} + \frac{1}{3} - \frac{1}{4} \ldots$ de som van de eerste 100 termen berekent.

A4. Schrijf een programma, dat bij gegeven waarden van n en x $\sum_{k=0}^{n} \frac{x^k}{k!}$ berekent.

Modificeer vervolgens het programma, zodat ook de uitkomsten van de partiële sommen worden geprint.

A5. Op een band staat het natuurlijke getal n, gevolgd door n reële getallen. Schrijf een programma, dat het minimum van deze n getallen bepaalt en telt hoeveel getallen gelijk zijn aan dit minimum.

A6. Schrijf een programma dat het aantal verschillende waarden bepaalt in een monotoon niet-dalende rij a_i, $1 \leq i \leq n$.

A7. Schrijf een programma, dat van n verschillende reële getallen op een band de waarde van het op één na kleinste bepaalt.

A8. Schrijf een programma, dat onderzoekt of een gegeven n x n matrix symmetrisch is. In het geval van symmetrie wordt het getal 1 geprint, in het andere geval het getal 0.

A9. Gegeven is het volgende programma:

```
begin  i := 0; j := 0; n := 0;
   L : i := i + 1; j := j + 1; n := n + 1;
       if i = 69 then i := 0;
       if j = 159 then j := 0;
       if i + j ≠ 0 then goto L;
       PRINT(n)
end
```

Ga na of dit programma eindigt. Zo neen, waarom niet. Zo ja, welke waarde van n wordt dan geprint.

A10. Gegeven is het volgende stukje programma (aan de variabelen a resp. b zijn reeds positieve gehele waarden toegekend):

```
      x := a; y := b;
L:    if x > y then x := x - y else y := y - x;
      if x ≠ y then goto L; GGD := x   .
```

Analyseer of dit stukje programma voor elk tweetal natuurlijke getallen a en b metterdaad aan de variabele GGD de waarde van de grootste gemene deler van a en b toekent.

A11. a) Schrijf een procedure rest (a, b) , die de rest bepaalt bij deling van a door b en als waarde toekent aan de naam rest; a en b zijn natuurlijke getallen.

b) Schrijf met behulp van de procedure rest een programma, dat de g.g.d. en het k.g.v. bepaalt van twee gegeven natuurlijke getallen.

A12. Schrijf een programma, dat voor een gegeven rij getallen a_1 tot en met a_n de waarde van $\max_{i,j}(a_i - a_j)$ berekent.

A13. Onder een stijgende run ter lengte k in een rij getallen a_1 tot en met a_n, waarvan geen twee opeenvolgende getallen gelijk zijn, verstaan we een monotoon stijgende deelrij a_i, a_{i+1}, ..., a_{i+k}, die men noch aan de voorkant, noch aan de achterkant kan verlengen zonder de monotonie te verstoren. Een dalende run wordt analoog gedefinieerd.

a) Schrijf een programma, dat het aantal stijgende en het aantal dalende runs op een getalband telt.

b) Schrijf een programma, dat bij gegeven m, het aantal stijgende en dalende runs telt ter lengte i, $i = 1$, $i = 2$, ..., $i = m$, $i > m$.

N.B. De aantallen runs zijn van belang bij het toetsen van statistische onafhankelijkheid in tijdreeksen.

A14. Schrijf een programma, dat een rij van n getallen naar opklimmende grootte sorteert en wel als volgt: Vergelijk a_i met a_{i+1} voor opvolgende waarden van i en verwissel de getallen als $a_i > a_{i+1}$. Herhaal dit proces, tot de rij gesorteerd is.
Kunt U iets zeggen over het aantal benodigde operaties? Wanneer is deze wijze van sorteren efficiënt?

A15. a) Zij f een veelterm, gedefinieerd door $f(x) = a_0x^n + a_1x^{n-1} + \ldots + a_n$. Schrijf een programma dat voor een gegeven waarde van p de waarde van f(p) berekent m.b.v. het Horner schema, d.w.z. als volgt: zij $b_0 = a_0$, zij $b_j = a_j + b_{j-1} \times p$ voor $1 \leq j \leq n$, dan geldt $f(p) = b_n$.

b) Ga na dat het Horner achema wordt weergegeven door de volgende schrijfwijze als "geneste vermenigvuldigingen" van het polynoom (we nemen n = 4): $(((a_0p + a_1)p + a_2)p + a_3)p + a_4$.

c) Merk op dat geldt $f(x) = (x - p)(b_0x^{n-1} + b_1x^{n-2} + \ldots + b_{n-1}) + b_n$, zodat het Horner schema nauw-verwant is aan de reststelling.

A16. a) Schrijf een programma, dat de coëfficiënten berekent van het polynoom $\sum_{k=0}^{n} a_k x^{n-k}$, als gegeven zijn de n reële wortels x_1 tot en met x_n; stel $a_0 = 1$.

b) Ga na, dat men met het programma onder a) de binomiaalcoëfficiënten $\binom{n}{k}$ voor k = 0, ..., n kan berekenen.

c) Kunt U een efficiënter programma bedenken om de rij $\binom{n}{k}$ voor k = 0, ..., n te berekenen? Schrijf hiervoor een programma.

A17. a) Schrijf de volgende expressies in postfixnotatie:

1) ((a × x + b) × x + c) × x + d

2) d + x × (c + x × (b + a × x))

3) a × x × x × x × x + b × x × x × x + c × x + d

b) Schrijf voor bovenstaande expressies een programma in machinecode met behulp van de instructies uit hoofdstuk 4.

A18. Schrijf een programma dat de coëfficiënten c_n van $P(x) \cdot Q(x)$ berekent; hierbij is

$$P(x) = \sum_{i=0}^{p} a_i x^i \quad \text{en} \quad Q(x) = \sum_{i=0}^{q} b_i x^i \ .$$

A19. Schrijf een programma, dat de coëfficiënten bepaalt van het quotiëntpolynoom bij deling van $f(x)$ door $g(x)$; hierbij is $f(x) = \sum_{i=0}^{n} a_i x^{n-i}$ en $g(x) = x^2 - px - q$.

A20. Schrijf een programma, dat van een gegeven $n \times n$ matrix de rijen zodanig verwisselt, dat voor de nieuwe matrix geldt:

$$|a_{jj}| \geqslant |a_{ij}| \quad \text{voor} \quad j = 1, 2, \ldots, n \quad \text{en} \quad i \geqslant j \ .$$

A21. Gegeven is het volgende lineair onafhankelijke stelsel vergelijkingen:

$$\begin{pmatrix} a_{11} & \cdots & a_{1n} \\ & a_{22} & \vdots \\ \bigcirc & & a_{nn} \end{pmatrix} \begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} b_1 \\ \vdots \\ b_n \end{pmatrix} .$$

Schrijf een programma, dat de onbekenden $x_1, \ldots, x_n$ berekent.

A22. a) Schrijf een programma dat met behulp van Gauss-eliminatie het stelsel lineaire vergelijkingen $\sum_{j=1}^{n} a_{ij} x_j = b_i$, $i = 1, \ldots, n$, oplost. Neem hiertoe aan dat tijdens het eliminatieproces op een gegeven moment het stelsel zodanig is getransformeerd, dat de matrix ervan een gedaante heeft als nevenstaand. De volgende stap van het proces bestaat dan uit het nulmaken van de benedendiagonaalelementen van de volgende (in dit geval 4e) kolom. Men neme bij deze reductie aan dat delers niet nul zijn. Het aldus verkregen "driehoekige" stelsel vergelijkingen kan men daarna op voor de hand liggende wijze oplossen (zie A21).

x	x	x	x	x	x	x
0	x	x	x	x	x	x
0	0	x	x	x	x	x
0	0	0	x	x	x	x
0	0	0	x	x	x	x
0	0	0	x	x	x	x
0	0	0	x	x	x	x

b) Met het optreden van een deler nul (of, wat in de numerieke praktijk even erg is, een zeer kleine deler) moet men natuurlijk rekening houden. Vandaar

dat men in de gebruikelijke algorithmen er de voorkeur aan geeft tijdens het proces de vergelijkingen zodanig te verwisselen, dat het getal, dat als deler zal optreden, zo groot mogelijk is (we gaan hier geheel voorbij aan het al dan niet optimaal zijn van deze handelwijze; dit is trouwens een moeilijk probleem).
Wijzig Uw programma dienovereenkomstig.

A23. Een vereniging heeft de stand van contributiebetalingen op een ponsband staan (de zgn. bestandsband). Deze ponsband bevat een aantal groepen van drie getallen. Elke groep bestaat uit:
lidmaatschapsnummer, jaar van geboorte, laatste jaar van betaling der contributie.
Het einde van de band wordt aangegeven door het getal -1.
Deze band wordt maandelijks gewijzigd in verband met gedane betalingen.
Daartoe wordt elke maand een mutatieband gemaakt met de volgende informatie:

jaar, aantal mutaties, mutaties .

Elke mutatie bestaat uit twee getallen: het lidmaatschapsnummer en het laatste jaar van betaling.
De informatie op de bestands- en mutatieband staat in volgorde van oplopend lidmaatschapsnummer.
De contributie bedraagt per jaar:
f 10,-- als op 1 januari van dat jaar geldt leeftijd $\leqslant$ 20
f 15,-- " " " " " " " " 21 $\leqslant$ leeftijd $\leqslant$ 30
f 25,-- " " " " " " " " 31 $\leqslant$ leeftijd.
Schrijf een programma, dat het volgende doet:
1) de mutatieband lezen en opbergen in het geheugen;
2) de oude bestands-band groepsgewijs lezen;
3) een nieuwe bestands-band maken;
4) voor een lid met achterstand in de betaling printen: lidmaatschapsnummer en achterstallige contributie. (Men heeft achterstand wanneer men de contributie van het vorige jaar niet heeft betaald.)

B1. a) Gegeven n getallenparen (x_1,y_1), ..., (x_n,y_n), op te vatten als een tabel voor de functie $y = f(x)$, die reeds in het geheugen staan. De rij x_1, ..., x_n is monotoon stijgend. Schrijf een programma, dat bij gegeven x in het interval $[x_1,x_n]$ door lineaire interpolatie $f(x)$ berekent.

b) Wanneer gegeven is, dat de getallen x_1 t/m x_n aequidistant zijn, kunt U dan een efficiënter programma bedenken?

B2. Op een schaakbord staan een zwarte koning, een wit paard en een witte koningin.
Schrijf een programma, dat uitmaakt of de koning schaak staat bij een gegeven stand der stukken.
Men bedenke zelf, hoe men schaakbord en stukken in ALGOL representeert. De gegeven stand wordt van de band gelezen. Indien schaak, wordt een 1 geprint, anders een 0.

B3. Schrijf een programma, dat van een gegeven natuurlijk getal de binaire representatie bepaalt.

B4. Bedenk een algorithme, die bij gegeven getallen a en n (n een natuurlijk getal) de macht a^n berekent met een aantal arithmetische operaties (niet ↑) dat kleiner is dan c.log(n), c een constante onafhankelijk van n.

B5. Gegeven een rij getallen x_1 t/m x_n. Schrijf een efficiënt programma dat de matrix (a_{ij}) berekent, waarbij $a_{ij} = \sum_{k=0}^{n} x_k^{i+j}$.

(N.B. Deze matrix komt voor bij kleinste kwadraten aanpassingen met polynomen.)

B6. a) Schrijf een programma, dat bij een gegeven natuurlijk getal n alle paren natuurlijke getallen (x,y) bepaalt, die voldoen aan $x^2 + y^2 = n$.

b) Schrijf een programma, dat bij een gegeven natuurlijk getal n op efficiente wijze het aantal roosterpunten telt, dat binnen of op de cirkel $x^2 + y^2 = n$ ligt.

B7. Schrijf een programma dat van een gegeven natuurlijk getal vaststelt of het een priemgetal is.
Toelichting: Als men van een natuurlijk getal wil vaststellen of het priem is zonder een priemgetaltafel te gebruiken, kan men natuurlijk delen door de achtereenvolgende natuurlijke getallen. Kunt U de rij getallen waardoor U deelt, zó uitdunnen, dat zo'n proces drie keer zo weinig rekentijd vraagt?

Ga na, dat in het licht van de priemgetalstelling (aantal priemgetallen $< n$ voor grote $n \sim \frac{n}{\log(n)}$) voor b.v. $n < 10^6$ dit proces redelijk efficiënt genoemd mag worden ten opzichte van het werken met een priemgetaltafel in het geheugen.
Wat vindt U van pogingen tot verdere verhoging der efficiency door de rij nog verder uit te dunnen?

B8. Schrijf een programma, dat de rij priemgetallen kleiner dan een gegeven natuurlijk getal n berekent. Gebruik hierbij de zeefmethode van Eratosthenes.

B9. Gegeven zijn twee rijen getallen $x_1, \ldots, x_m$ en $y_1, \ldots, y_n$, die allen onderling verschillend zijn en reeds in het geheugen staan. Schrijf een programma ter berekening van het getal $W = \sum_{k=1}^{n} w_k$, w_k = aantal getallen x_i, dat groter is dan y_k, $k = 1, \ldots, n$.

N.B. Dit probleem is afkomstig uit de statistiek: de tweesteekproeventoets van Wilcoxon. De getallen n en m kunnen aanzienlijk in grootte verschillen.

B10. Schrijf een programma, dat bepaalt op hoeveel verschillende manieren men een bedrag van n cent kan betalen met munten van 1, 5, 10, 25, 100 en 250 cent.

B11. Gegeven n positieve getallen $g_1, \ldots, g_n$, die gewichten voorstellen.

a) Schrijf een programma, dat een tabel print van alle mogelijke verschillende gewichten, die men hieruit kan samenstellen door combinaties op één schaal van de balans te zetten.

b) Hetzelfde, maar nu mogen gewichten op beide schalen staan.

B12. Gegeven n lijnen in een plat vlak en een punt P, dat op geen van die lijnen ligt.
Gevraagd een programma, dat de hoekpunten berekent van de minimale convexe veelhoek, waarin P ligt. Indien het oppervlak van die veelhoek oneindig groot is, dient bovendien iets geprint te worden waaruit dit blijkt.

B13. Gevraagd wordt een programma te schrijven, dat telt hoeveel maal het lidwoord "de" en hoeveel maal het lidwoord "het" in een gegeven tekst voorkomt. De tekst is op een band geponst. Men beschikt over een procedure NEXT SYMBOL en door de statement x := NEXT SYMBOL wordt aan x een getalwaarde toegekend, die correspondeert met het volgende symbool op de band. Deze correspondentie is als volgt:

symbool	numerieke waarde
0, 1, ..., 9	0, 1, ..., 9
a, b, ..., z	101, 102, ..., 126
A, B, ..., Z	201, 202, ..., 226
leestekens en spatie	$\geqslant 300$
einde band	-1.

C1. Schrijf een programma, dat met behulp van de iteratie formule van Newton $\sqrt{a}$ berekent voor een gegeven positieve waarde van a.
Het iteratieproces wordt beëindigt als $|x_{n+1} - x_n| < \text{eps}$ (voor gegeven eps).
Maak het programma zo, dat als een gebruiker een te kleine eps geeft, het programma toch eindigt.

C2. Schrijf een programma, dat met behulp van de iteratie formule van Newton bij gegeven waarde van $a \neq 0$ de reële wortel van de vergelijking

$$x^3 - x^2 + x - a = 0$$

met een relatieve nauwkeurigheid van 6 decimalen berekent.

C3. Schrijf een programma, dat bij een gegeven positieve waarde van de parameter p door successieve halvering in 3 decimalen nauwkeurig het snijpunt berekent van de krommen gegeven door:

$$f(x) = \frac{p}{1+x^2} \quad \text{en} \quad g(x) = \log x \ .$$

C4. a) Ga na, dat onderstaande procedure voor een gegeven polynoom $p(x) = \sum_{k=0}^{n} a_k x^{n-k}$ bij een gegeven x de waarde van de Newton-Raphson correctie berekent.

```
procedure NERACOR (x) ;
begin  w := a[0]; wacc := a[0]; k := 1;
cycle: w := w × x + a[k]; wacc := wacc × x + w; k := k + 1;
       if k < n then goto cycle;
       NERACOR := (w × x + a[k])/wacc
end
```

b) Gegeven is een veelterm met uitsluitend reële wortels. Men wenst hiervan de grootste en kleinste wortel te bepalen. Overtuig U ervan door een convex-

heidsargument dat voor startwaarden groter dan de grootste en kleiner dan de kleinste wortel het Newton-Raphson proces op eenvoudige wijze kan worden voortgezet tot de grootst mogelijke nauwkeurigheid is bereikt.
Kunt U in het bijzondere geval dat alle wortels ≥ 0 zijn, uit de coëfficiënten van de vergelijking op eenvoudige wijze een bovengrens voor de wortels afleiden? Dan kunt U het ook wanneer de wortels alleen maar reëel zijn door het polynoom te beschouwen waarvan de wortels de kwadraten zijn van die van het oorspronkelijke polynoom.
Schrijf een programma.
N.B. Men zou nu alle wortels van het polynoom kunnen vinden door met behulp van de reststelling een gevonden wortel te verwijderen, aldus de graad verlagend, en het proces te herhalen. Onder omstandigheden kunnen ten gevolge van de afrondfouten later gevonden wortels evenwel met een zeer grote fout belast zijn.

C5. Bedenk een algorithme, die met Newton-Raphson de wortel van $\frac{p}{x} = \log x$ berekent voor $p = 0.1(0.1)2.0$, in zo groot mogelijke nauwkeurigheid.
Zorg ervoor, dat de startwaarden convergentie garanderen en voordelig gekozen worden.
Hoe stopt U de iteraties?

C6. Gevraagd wordt de beide wortels te berekenen van $\alpha x = \log x$ voor $\alpha = 0.1(0.01)0.36$ ($e^{-1} = 0.36787....$). Onderzoek de geschiktheid hiervoor van de iteratie formule $x_{i+1} = \frac{1}{\alpha} \log(x_i)$ en de inverse hiervan.
Kies de startwaarden zuinig en ga met itereren door tot een zo hoog mogelijke nauwkeurigheid bereikt is. Ga na hoe de convergentiesnelheid afhangt van α.

C7. a) Als U de kleinste wortel van $x = 10 \log(x)$ berekent met de iteratie formule $x_{i+1} = e^{x_i/10}$, hoe zoudt U dan starten en hoe zoudt U de iteraties beeindigen als U het antwoord in maximale nauwkeurigheid wenst te kennen?
Als we aannemen, dat er exact gerekend wordt, hoe zoudt U dan het stopcriterium kiezen om het resultaat met een fout, die kleiner is dan een gegeven $\varepsilon > 0$, te verkrijgen? Kan het misgaan, als er niet exact gerekend wordt?

b) Dezelfde vragen voor de iteratie $x_{i+1} = \frac{1}{10} e^{-x_i}$ ter oplossing van de vergelijking $x = \frac{1}{10} e^{-x}$.

C8. Gegeven n getallen $x_1, \ldots, x_n$, onderling verschillend. Schrijf een programma

dat met behulp van halvering alle extrema berekent van het polynoom $\prod_{i=1}^{n}(x-x_i)$ in zo groot mogelijke precisie. (Men beschouwe de afgeleide.)

C9. Zij $f(x)$ continu op $0 \leqslant x \leqslant 1$.
De rij $T_0, T_1, T_2, \ldots$ is als volgt gedefinieerd:

$$T_0 = \tfrac{1}{2}\{f(0) + f(1)\}$$

$$T_n = \tfrac{1}{2}\{T_{n-1} + M_n\} \quad \text{met} \quad M_n = \left\{f\left(\frac{1}{2^n}\right) + f\left(\frac{3}{2^n}\right) + \ldots + f\left(\frac{2^n-1}{2^n}\right)\right\} \cdot 2^{-n+1}$$

voor $n \geqslant 1$.
Maak voor het geval $f(x) = \dfrac{\sqrt{x+2}}{1+\sqrt{x+1}}$ een programma om T_n te berekenen voor zodanige n dat $|T_n - T_{n-1}| < \text{eps}$ is. De tolerantie eps wordt van de band gelezen. Print eps en T_n.

Zij $I = \displaystyle\int_0^1 \frac{\sqrt{x+2}}{1+\sqrt{x+1}}\,dx$.

Bewijs dat geldt $I = I_n = \dfrac{\varphi_n}{12.2^{3n}}$ met $\varphi_n = \displaystyle\sum_{i=0}^{2^n-1} f''(\xi_{i,n})$, $\dfrac{i}{2^n} \leqslant \xi_{i,n} \leqslant \dfrac{i+1}{2^n}$.

Bewijs dat $\lim\limits_{n\to\infty} \dfrac{\varphi_n}{2^n} = f'(1) - f'(0) \neq 0$.

Bewijs dat $\lim\limits_{n\to\infty} \dfrac{I - T_n}{T_n - T_{n-1}} = \dfrac{1}{3}$.

Geef op grond hiervan een schatting van $I - T_n$ in termen van eps.

C10. Soms moet men functies numeriek integreren, waarvan de afgeleide goedkoop te verkrijgen is (b.v. e^{-x^2}). We construeren een kwadratuurformule voor zo'm geval.
Kies de constanten in

$$\int_a^b f(t)dt \approx c_1 f(a) + c_2 f'(a) + d_1 f(b) + d_2 f'(b)$$

zó, dat polynomia van zo hoog mogelijke graad nog exact geïntegreerd worden. Als de restterm de gedaante $\alpha f^{(k)}(\xi)$ heeft, vind dan α en k. Welk voordeel heeft deze formule als men het interval [a,b] in stukken splitst en op elk der stukken de formule toepast?

C11. Schrijf een programma dat $\displaystyle\int_0^1 t^n e^{-t} dt$ berekent voor gegeven n, door de inte-

grand in een reeks te ontwikkelen, term voor term te integreren, en de aldus verkregen reeks te sommeren.
Als men de sommatie voortzet tot aan de eerste term, die de tot dan toe verkregen som niet meer wijzigt, mag men dan verwachten de waarde van de integraal in redelijke precisie berekend te hebben?

C12. Schrijf een programma, dat de integralen $I_n = \int_1^\infty e^{-t} t^n dt$ berekent voor $n = 0(1)100$ met behulp van een recursie. Onderzoek de door U gebruikte recursie op stabiliteit.