TECHNISCHE HOGESCHOOL EINDHOVEN
Afdeling Algemene Wetenschappen
Onderafdeling der Wiskunde

INFORMATICA I
(2e gedeelte)

Prof. Dr. R.J. Lunbeck

Najaarssemester 1978-1979

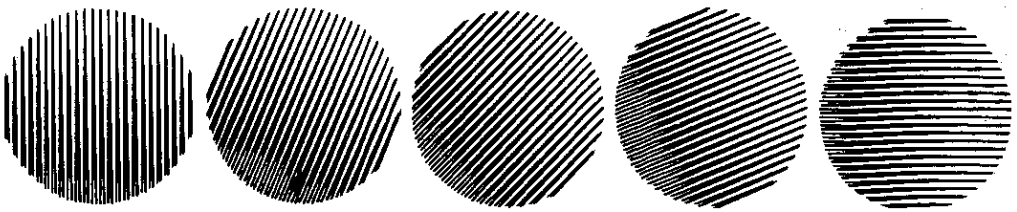
Technische Hogeschool
Eindhoven

Dictaatnummer 2. 271
Prijs f. 2,50

Onderafdeling der Wiskunde en Informatica

Informatica 1

(2de gedeelte)



Inhoudsbeschrijving

INFORMATICA I-2e gedeelte

R.J. Lunbeck

Najaarssemester 1978-1979

JdG, 5 December 2005

TECHNISCHE HOGESCHOOL EINDHOVEN

Onderafdeling der Wiskunde

Informatica I

(2-de gedeelte)

Najaarssemester 1978-1979

2. Datastructuren

Deze paragraaf heeft een tweeledig doel. Enerzijds om Algol 60 (met toevoeging van de while - do constructie) toe te passen op een aantal deels vrij gecompliceerde problemen. Anderzijds dient het ter kennismaking met een aantal z.g. datastructuren. Hieronder wordt informeel verstaan een "aantal gegevens waartussen bepaalde verbanden bestaan". Formeel kan een datastructuur gedefinieerd worden als een paar (K,R), waarin K een verzameling "knopen" is en R een over K gedefinieerde binaire relatie. Hierin is een knoop een grootheid waarvan de waarde bestaat uit een rij van enkelvoudige waarden (mogelijk van verschillend type). Een binaire relatie over K is een deelverzameling van de verzameling van paren (x,y) met $x \in K$, $y \in K$, die aan zekere voorwaarden voldoen. Wordt bij deze paren aan gerichte lijnstukken (pijlen) gedacht, dan is een datastructuur dus door een aantal knopen en pijlen grafisch voor te stellen. Bovendien onderscheiden de verschillende structuren zich door de operaties die er op gedefinieerd zijn. Datastructuren die hier in ogenschouw worden genomen zijn wachtlijnen, kettingen en stapels (als voorbeelden van "lineaire" structuren), dan bomen en tenslotte netwerken. Aangezien deze datastructuren niet gedefinieerd zijn in Algol 60, zullen ze gerepresenteerd moeten worden met de daar wel beschikbare structuren, nl. arrays. De beschrijving van de "logische" manipulatie van datastructuren d.w.z. een beschrijving onafhankelijk van de representatie, wordt in Algol 60 dan ook een algorithme die wel afhankelijk is van de representatie en daarmee vaak vrij gecompliceerd. De genoemde structuren zullen steeds aan de hand van een eenvoudig voorbeeld behandeld worden; vele andere voorbeelden zouden te halen zijn uit de gebieden van de discrete wiskunde, de electrotechniek, de operations research, de database problematiek, enz.

2.1. Wachtlijnstructuur; simulatie

Opgave

Gegeven een loket met daarvoor een wachtrij waar alleen op de volle minuten één klant kan aankomen en (een andere) vertrekken; iedere minuut is de kans op aankomst p. Op grond van de noden van de klant zal de afhandelingstijd, d.w.z. als de klant eenmaal aan de beurt is, een geheel aantal minuten a bedragen, waarbij alle a-waarden tussen 2 en 11 minuten dezelfde kans op voorkomen hebben (homogene verdeling). Op het tijdstip 0 zijn er nog geen klanten. Gevraagd wordt o.a. op opeenvolgende tijdstippen het gemiddeld aantal klanten in de rij, die zich mogelijk zal vormen, en de gemiddelde wachttijd.

Discussie

Voor dit eenvoudige geval zouden langs analytische weg de gevraagde grootheden met behulp van de wachttijd-theorie (theorie der stochastische processen) direct te berekenen zijn. Hier zullen zij daarentegen langs "experimentele weg (en wel) door discrete simulatie" bepaald worden. Aangenomen wordt dat het volgende beschikbaar is:

- een klok, k, die met intervallen van een minuut gedurende K minuten loopt (klanten arriveren en vertrekken steeds op de momenten k - ε);
- een parameterloze real procedure random uit de bibliotheek, die bij iedere aanroep een fractie oplevert, waarvan de mogelijke waarden homogeen willekeurig tussen 0 en 1 verdeeld zijn;
- een variabele, wr, waarmee iedere minuut genoteerd wordt hoeveel mensen in de wachtrij staan (de klant aan het loket niet meetellend);
- een variabele wrg, welke iedere minuut aangeeft hoelang de gemiddelde wachtrij tot dat moment is (alléén gemiddeld over de minuten dat zich tenminste 1 klant in de wachtrij heeft bevonden!);
- variabelen, welke iedere minuut aangeven, hoe lang gemiddeld de wachttijd in de rij, twg, en de afhandelingstijd, tag, is van alle klanten die intussen afgehandeld zijn.

Het werkelijke proces is na te bootsen door bij iedere klokstand k te onderzoeken:

a) of een nieuwe klant gearriveerd is. Dit kan gebeuren door af te spreken dat wanneer een aanroep van random een getal < p oplevert dit correspondeert met de aankomst van een nieuwe klant. Voor deze nieuwe klant moeten dan genoteerd worden:
   . aan, de aankomsttijd: om straks zijn wachttijd te kunnen berekenen,
   . afh, zijn afhandelingstijd: om straks wanneer hij aan de beurt komt, meteen te kunnen vaststellen zijn
   . ver, vertrektijd
   . wac, wachttijd in de rij.
   Deze gegevens moet de klant "met zich meedragen in zijn knoop", totdat hij afgehandeld is aan het loket en zijn gegevens verwerkt zijn in de verschillende gevraagde gemiddelden.
   De afhandelingstijd kan, hetzij meteen bij aankomst, hetzij op het moment dat hij aan de beurt komt, berekend worden met

   ```
   entier(2.0 + (12 - 2) * random)
   ```

   omdat random een fractie tussen 0 en 1 oplevert en de entier bewerking vervolgens één van de getallen 2 t/m 11.

b) of een klant op dat moment het loket zal verlaten. Dit kan vastgesteld worden door in de knoop van de klant aan het loket zijn vertrektijd op te zoeken. Zoals gezegd moeten bij het vertrek verschillende gemiddelden opnieuw vastgesteld kunnen worden. Deze gemiddelden van een grootheid h[i] worden berekend met gh/i waarin gh := gh + h[i] i = 1,2,... als er i klanten afgewerkt zijn (de initialisatie van gh is nul). Het voordeel van de berekening in iedere minuut van een gemiddelde is dat men door het bekijken van gh/i op opeenvolgende tijdstippen een beeld krijgt van het "schommelen" om het uiteindelijke gemiddelde (als controle zou men dan ook de gemiddelde afhandelingstijd kunnen berekenen, die op 6.5 minuten moet uitkomen).
   Het berekenen van de gemiddelde wachtrijlengte vraagt even bijzondere aandacht omdat zoals gezegd nu natuurlijk slechts gemiddeld mag worden over die tijdsintervallen van 1 minuut dat er tenminste 1 klant in de wachtrij staat!

c) of een klant bij het loket geplaatst kan worden. Dit kan natuurlijk alleen als het loket leeg is èn wanneer de wachtrij gevuld is (mogelijk met een net gearriveerde klant). Van de bij het loket geplaatste klant kunnen nu berekend worden:
   . zijn vertrektijd (uit afhandelingstijd en tijdstip van plaatsing bij het loket),
   . zijn wachttijd (uit aankomsttijd en tijdstip van plaatsing bij het loket)

De goede volgorde van deze deelprocessen is b, a en dan c omdat alleen dan een net in een lege wachtrij gearriveerde klant meteen aan het mogelijk net vrijgekomen loket geholpen kan worden.

De bij dit probleem horende datastructuur is een eenvoudige. Hij bestaat uit een aantal knopen k_i (voor iedere gearriveerde klant i), waarin worden opgeslagen de vier grootheden aan, afh, ver en wac (die allen van het type integer zijn; in verband met de "levensduur" van deze grootheden waren minder variabelen ook voldoende geweest doch dit zou het programma onduidelijker maken). Tussen deze knopen bestaat slechts de relatie $\{(k_i,k_{i+1})\}$ of anders gezegd: iedere knoop heeft één voorganger behalve de eerste. Dit is een z.g. lineaire structuur. Met de variabele i (> 0) kan men bijhouden dat klant i af te handelen is, met een variabele g dat in knoop k_g gegevens van de volgende arriverende klant zijn te plaatsen.
De op deze structuur uit te voeren bewerkingen zijn eveneens eenvoudig (omdat de klanten in volgorde van aankomst afgewerkt worden heet dit een First In First Out of FIFO queue), nl.
- voor iedere nieuwe klant is aan het ene einde van de lineaire structuur een knoop toe te voegen met daarin direct te plaatsen de aankomsttijd en de afhandelingstijd,
- voor iedere vertrekkende klant is aan het andere einde van de structuur een knoop af te haken, nadat de daarin opgenomen gegevens verwerkt zijn,
- in de opvolgerknoop van de afgehaakte knoop zijn wachttijd en vertrektijd in te vullen.

Een globale schets van de algorithme is:

```
while k < simulatieperiode K
      do k := k + 1;                              {klok 1 minuut verder}
         if klant klaar then                      {*}
            werk totale wachttijd bij;            {loket klant af}
            werk totale afhandelingstijd bij;
            loket leeg := true; i := i + 1
         fi;
         if nieuwe klant gearriveerd then
            vul zijn aankomsttijd in;
            vul zijn afhandelingstijd in;         {nieuwe klant in wachtrij}
            rijlengte := rijlengte + 1; g := g + 1
         fi;
         if loket leeg and rijlengte ≠ 0 then
            loket leeg := false; rijlengte := rijlengte -1;
            wachttijd := k - aankomsttijd;        {klant naar loket}
            vertrektijd := k + afhandelingstijd
         fi;
         if rijlengte ≠ 0 then
            werk totale wachtrijlengte bij        {**}
         fi;
         write (gewenste grootheden)
      od
```

{*} klant klaar is te constateren met k = vertrektijd in knoop i.
Ga zelf na dat nooit i groter dan g kan worden!

{**} voor het berekenen van de gem. wachtrijlengte moet in een teller t bijgehouden worden over hoeveel perioden de wachtrij niet leeg is.

Voor het opstellen van een Algol programma zal de wachtlijn afgebeeld moeten worden op Algol variabelen. Hiervoor kan eenvoudig een array genomen worden omdat hier

- toch alle op te bergen knoopgrootheden integers zijn,
- alle knopen sequentieel afgewerkt moeten worden,
- de eenvoudige relatie impliciet door de sequentiële opvolging van array-rijen gerepresenteerd kan worden.

Als bekend is dat n klanten tijdens de simulatie periode zullen arriveren zou een knopenarray KA[1 : n, 1 : 4] gedeclareerd kunnen worden, waarbij

KA[j,1] de aankomsttijd van klant j is,
KA[j,2] de afhandelingstijd van klant j is,
KA[j,3] de vertrektijd van klant j is,
KA[j,4] de wachttijd van klant j is.

Echter, dit aantal klanten is niet bekend! Moet men dan voor de bovengrens n maar een zeer groot getal invullen? Afgezien van het feit dat misschien die bovengrens verkeerd geschat is, zou dit bovendien kunnen leiden tot een slechte vulling van het array. Immers slechts van de in de wachtrij en bij het loket staande klanten zijn gegevens beschikbaar of nodig. Is de wachtrij betrekkelijk kort, dan wordt slechts een zeer klein deel van het array effectief gebruikt. Deze overweging leidt er toe om de wachtrij "circulair" af te beelden op een array KA[0 : N-1; 1 : 4], d.w.z. dat de gegevens van knoop k ($1 \leq k < K$) afgebeeld worden op KA[k mod N; 1 : 4] waarin N groter gekozen moet worden dan het verwachte aantal klanten in de wachtrij. Zolang $g - i < N$ kunnen er inderdaad geen ongelukken gebeuren met de afbeelding, maar bij iedere toename van g (dus bij arriveren van een nieuwe klant) moet gecontroleerd worden of nog aan deze voorwaarde voldaan is. Zo niet, dan moet wegens gebrek aan ruimte het simulatieproces afgebroken worden (door naar het einde van het programma te springen) nadat een betreffende boodschap is afgedrukt. Een andere consequentie van deze circulaire afbeelding is dat de onderste grens van de eerste index van KA nul kan zijn.

In het uitgeschreven Algolprogramma is ter wille van de doorzichtigheid van het programma voor de tweede index van KA niet een cijfer geschreven maar de naam van de corresponderende probleemgrootheid. Onderzoek waarom, voor de hoofdlus begint, de variabelen KA[i, ver] op nul gezet moeten worden en of deze maatregel voldoende is.

```
begin integer N; read(N);
      begin integer array KA[0 : N - 1, 1 : 4]; real p; boolean loket leeg;
            integer i, aan, afh, ver, wac, g, k, K, wr, t, wrg, twg, tag;
            read(p); read(K);
            aan := 1; afh := 2; ver := 3; wac := 4;
            i := k := wr := t := 0;
            wrg := twg := tag := 0; loket leeg := true;
            while i < N do begin KA[i, ver] := 0; i := i + 1 end;
            i := g := 1;
            while k < K do
                  begin k := k + 1;
                        if k = KA[i mod N, ver]then
                           begin twg := twg + KA[i mod N, wac];
                                 tag := tag + KA[i mod N, afh];
                                 loket leeg := true; i := i + 1
                           end;
                        if random < p then
                           begin if g - i < N
                                 then begin KA[g mod N, aan] := k;
                                            KA[g mod N, afh] := entier(2.0 +10*random);
                                            wr := wr + 1; g := g + 1
                                      end
                                 else begin write (array too small); goto af end
                           end;
                        if loket leeg and wr ≠ 0 then
                           begin loket leeg := false; wr := wr - 1;
                                 KA[i mod N, wac] := k - KA[i mod N, aan];
                                 KA[i mod N, ver] := k + KA[i mod N, afh]
                           end;
                        if wr ≠ 0 then begin t := t + 1;
                                             wrg := wrg + wr
                                       end;
                        write(k, i, g, wr, t, wrg/t, twg/i, tag/i)
                  end
      end
af: end
```

Opmerking

Met discrete simulatie kost het nauwelijks meer moeite om uitkomsten te vinden voor complexer probleemstellingen:

- andere dan de hier gekozen homogene verdeling voor afhandelingstijden,
- een ander dan het hier gekozen gedrag van klanten (niet voor de beurt gaan),
- situaties met meer loketten, ingewikkeld "gedrag" van klanten en loketbedienaars, enz. (het analyseren van dit gedrag kan echter veel tijd vergen!).

Voor discrete simulatie is de taal Simula, een uitbreiding van Algol 60, ontwikkeld.

Discrete simulatie heeft vergeleken met de analytische aanpak de bezwaren dat het afschatten van de benodigde array-ruimte niet altijd eenvoudig is en dat soms de simulatie erg lang voortgezet moet worden voordat stabiele dynamische eindwaarden (met een voldoende nauwkeurigheid) voor de gemiddelden bereikt zijn.

2.2. Kettingstructuren

Deze structuren worden ingevoerd aan de hand van twee opgaven, A en B.

2.2.1. Opgave A.

Op een band staat een monotoon niet-stijgende rij natuurlijke getallen (> 0), afgesloten met een nul. Gevraagd wordt het volgordenummer ("index") van "het op n na grootste getal". Het getal n staat als eerste op de band, maar telt niet mee bij de rij.

Discussie

- De specificatie van de opgave is niet voldoende omdat:
  . het niet zeker is, dat een getal met de gewenste eigenschap voorkomt (als de rij "te kort" is, gezien ook het mogelijk voorkomen van gelijken).
  Te verlangen is nu dat als volgordenummer een nul wordt afgedrukt.
  . door het voorkomen van gelijken meer getallen in aanmerking kunnen komen.
  Te verlangen is dat het volgordenummer van het eerste in aanmerking komende getal (d.w.z. dat getal met de kleinste index) wordt afgedrukt.
  . als het in aanmerking komende getal gevonden is, men nog niet weet wat te doen met de rest van de band. Te verlangen is dat in ieder geval de volledige rij getallen verwerkt wordt, omdat het slordig staat om een berekening af te sluiten, terwijl de bijbehorende getallenband nog niet afgewerkt is.

- Het simpele gegeven, dat de rij getallen monotoon niet-stijgend is, vereenvoudigt de oplossing aanzienlijk:
  . als een in aanmerking komend getal gevonden is, kan men het volgordenummer afdrukken en hoeven de dan volgende getallen alleen nog maar ingelezen te worden.
  . om in aanmerking te komen, moeten zeker n grotere getallen (die niet verschillend hoeven te zijn) geconstateerd zijn. Omdat onder de reeds ingelezen getallen gelijken kunnen voorkomen, zal het gezochte getal meestal meer dan n (en dus niet precies n) grotere voorgangers hebben en zal het ook nooit de laatste van een rij gelijke getallen kunnen zijn.

. Om aan dit geval van meer dan n voorgangers toch niet het volgordenummer nul als uitkomst te verbinden, moet men in de opgave de formulering "op n na grootste getal" vervangen door de formulering "het grootste getal, waarbij nog tenminste n grotere getallen aan te wijzen zijn" (neem bijvoorbeeld n = 2 en de rij 10,9,9,8,... , waarbij de gezochte index 4 is).

Na deze voorbereidende discussie komt men tot de volgende opzet voor een oplossing:

```
while getal ≠ 0
      do if "getal nog niet gevonden" then
            "noteer index laatstgelezen getal";
            if index > n then          { zolang index ≤ n geen oplossing!}
               if getal ≠ vorige getal then {alleen eerste getal van een rij
                        write (index)        gelijken kan in aanmerking komen}
                        "noteer dat getal gevonden is"
               fi
            fi;
            if getal ≠ vorige then vorige := getal fi  {zie laatste commentaar}
         fi;
         read (getal)
      od;
if "getal niet gevonden" then write (0) fi
```

Uitgeschreven in Algol 60 wordt het programma met invoering van de nodige variabelen:

```
begin integer getal, index, n, vorige; boolean found;
      read (n); index := vorige := 0; found := false; read (getal);
      while getal ≠ 0 do
            begin if not found then
                        begin index := index + 1;
                              if index > n then
                                    begin if getal ≠ vorige then
                                                begin found := true;
                                                      write (index)
                                                end
                                    end;
                              if getal ≠ vorige then vorige := getal
                        end;
                  read (getal)
            end;
      if not found then write (0)
end
```

2.2.2. Opgave B

Op een band staat een rij natuurlijke getallen (> 0), afgesloten met een nul. Gevraagd wordt het volgordenummer ("index") van het grootste getal uit die rij, waarbij nog tenminste n grotere getallen aan te wijzen zijn. Komen meer getallen met de gewenste eigenschap voor, dan moet de kleinste van de betreffende indices afgedrukt worden, terwijl een nul moet worden afgedrukt als er geen getal met de gewenste eigenschap is.

Discussie

Het enige verschil met de vorige opgave is dat alleen de woorden "monotoon niet-stijgend" ontbreken, maar deze weglating heeft zoals blijken zal, ingrijpende gevolgen. Omdat de volgorde der getallen willekeurig is, zal men bijv. een lijst kunnen gaan aanleggen van potentieel grotere getallen dan het gezochte getal. Wegens de mogelijke gelijken moet ook hun multipliciteit vastgelegd worden. Evenzo moet van deze getallen hun (kleinste) index bewaard blijven. Deze lijst met 3 getallen op een regel hoeft slechts te groeien tot maximaal n + 1 regels (nl. als alle getallen enkelvoudig zijn). (Als de lijst deze maximale vulling bereikt heeft, kan het gebeuren dat het kleinste getal van de lijst mag vervallen omdat, wegens de eis van n grotere getallen, al vaststaat dat het nooit in aanmerking kan komen.)
Een pas van de band gelezen getal moet met de in de lijst aanwezige getallen vergeleken worden om vast te stellen of dat nieuwe getal in de lijst moet worden opgenomen en zo ja, waar. Om dit vergelijken voor een beetje lange lijst efficiënt uit te kunnen voeren, moet de lijst geordend zijn, welke ordening bereikt kan worden door de getallen in (bijv. afdalende) volgorde op te slaan. Dit heeft het bezwaar, dat bij toevoegen van een nieuw getal (bij wat grote n) flink geschoven moet worden (een proces als in de sorteeralgorithme).

Een (verder te behandelen) alternatief voor deze lijstmethode is die waarbij men voor iedere ingelezen getalwaarde deze waarde, zijn multipliciteit en zijn (kleinste) index noteert op een schakel of knoop. Deze knopen moet dan in de volgorde van afnemende grootte van de daarop staande waarden aan een ketting geregen worden. Net zoals bij de lijstmethode zullen nu maximaal n + 1 schakels nodig zijn omdat als sommige

getallen aan het begin van de ketting een van 1 verschillende multipliciteit hebben, dit maximum aantal niet eens nodig is.
De ketting zal met 1 schakel beginnend, afhankelijk van de grootte van de getallen op de band, dus kunnen groeien tot n + 1 schakels en zelfs weer korter kunnen worden, als bijvoorbeeld de getallen een hogere multipliciteit krijgen bij het verder inlezen van de band. Men kan zich afvragen of geregeld onderzocht moet worden of de ketting de goede lengte heeft, hetgeen op grond van de multipliciteit in de schakels geconstateerd kan worden, dan wel toelaten, dat de ketting tot maximaal n + 1 schakels groeit en dan niet meer in lengte verandert. Als alle getallen van de band verwerkt zijn, moet men in dat geval één keer met behulp van de multipliciteiten opsporen, welke schakel het gezochte getal bevat. Deze laatste methode lijkt de eenvoudigste, en zal daarom worden gevolgd.
Net zoals de wachtlijn is de datastructuur een lineaire structuur. Hij bestaat nu uit knopen k_i (voor de grootste getalwaarden) waarin worden opgeslagen: w : de betreffende getalwaarde; m : zijn multipliciteit en x : de (kleinste) index van deze getalwaarde. De relatie tussen de knopen is echter ingewikkelder, nl.

$$\{(k_i,k_j) \mid \text{er is precies één } j \text{ waarbij } w_j < w_i \text{ terwijl voor geen } w_k \text{ geldt } w_j < w_k < w_i\} .$$

Deze ingewikkelder relatie brengt met zich mee dat ook de op deze structuur uit te voeren bewerkingen gecompliceerder zijn. Een nieuwe schakel kan namelijk niet zoals bij de wachtlijn achter aan de ketting gehangen worden. Afhankelijk van de grootte van het gelezen getal zal men namelijk één van de volgende handelingen moeten verrichten:

- in één van de bestaande schakels moet slechts de multipliciteit m met 1 opgehoogd worden,
- een nieuwe schakel moet toegevoegd worden:
  - aan het eind van de ketting,
  - ergens in de ketting,
  - in het begin van de ketting,
- er hoeft geen schakel toegevoegd te worden omdat het betreffende getal toch nooit in aanmerking komt (omdat er al n + 1 schakels met grotere getallen in de ketting zitten).

Dit is nog iets te vereenvoudigen door aan het begin van de ketting een extra ("nulde") schakel te hangen en ook de laatste schakel hiermee te

verbinden. Daar de ketting nu gesloten is worden nieuwe schakels dus altijd ergens in de ketting toegevoegd.
Een complicatie ontstaat echter wanneer niet onbeperkt veel schakels toegelaten worden, maar bijvoorbeeld slechts n + 2. Voor een nieuwe schakel moet men als alle schakels al in gebruik zijn genomen, dan de laatste schakel opnieuw gebruiken.
Een globale vorm van de algorithme is:

```
while getal ≠ 0
      do "verwerk getal in ketting"
         "lees (volgend) getal en leg zijn index vast"
      od;
write (gevraagde index)
```

Hierin is "verwerk getal in ketting" verder te detailleren tot:

```
procedure put;
     "zoek de plaats in de ketting op waar het laatst gelezen getal g ver-
      werkt moet worden";
     if g = "waarde in schakel"
          then "verhoog multipliciteit in schakel"
          else if "laatste schakel gepasseerd"
                  then if "er is nog een vrije schakel"
                           then "haak vrije schakel  in  en vul deze"
                           else "niets doen"
                       fi
                  else if "er is nog een vrije schakel"
                           then "haak vrije schakel in en vul deze"
                           else "zoek laatste schakel op en gebruik deze
                                 als nieuwe schakel"
```

Voor het opstellen van een Algol-programma kan de kettingstructuur afgebeeld worden op twee arrays, nl. gg[0: n+1,1:3] met

gg[i,w (=1)] de waarde van een van de band gelezen getal,
gg[i,x (=2)] de (kleinste) index van dat getal,
gg[i,m (=3)] de multipliciteit van dat getal,

(deze grootheden zijn in één array gestopt omdat ze toch allen integers zijn) en een wijzer array W[0:n+1], waarbij W[i] het nummer van de schakel volgend op i is.

(Bij de wachtlijn was array W niet nodig omdat i en i+1 de nummers van twee opeenvolgende schakels waren. Omdat de W[i] ook integers zijn hadden ze bijvoorbeeld ook als vierde component aan gg toegevoegd kunnen worden.) De index i = 0 wordt gebruikt voor de vaste schakel, zodat W[0] het nummer van de schakel met de grootste getalwaarde is. Met deze getalwaarde wordt het ingelezen getal g vergeleken. Is g kleiner dan wordt het vergeleken met het getal in de volgende schakel, enz., totdat eventueel het einde van de ketting is bereikt, waarvoor geldt W[i] = 0. Met andere woorden, het langs de ketting lopen is een proces dat beheerst wordt door

```
while g < gg[i,w] and W[i] ≠ 0 do i := W[i] .
```

Wat de procedure put moet doen is bijgaand in plaatjes weergegeven.
De wijzer j loopt net 1 stap achter i aan, samenhangend met het feit dat men slechts in 1 richting langs de ketting kan gaan. In de uitgeschreven algorithme komen voorts voor de wijzers:

v die wijst naar een nog niet bezette rij in gg en W,
ii en jj die nodig zijn voor het opzoeken van de laatste schakel in de ketting .

De boolean full is ingevoerd om direct te kunnen constateren of er nog vrije schakels zijn. In het programmastuk voor het tussenvoegen van een schakel is de volgorde van vullen van W[jj] en W[j] belangrijk wanneer jj = j. Dan is ook ii = i en W[i] moet dan met 0 gevuld worden; in andere gevallen met i.

In de volgende plaatjes betekent:

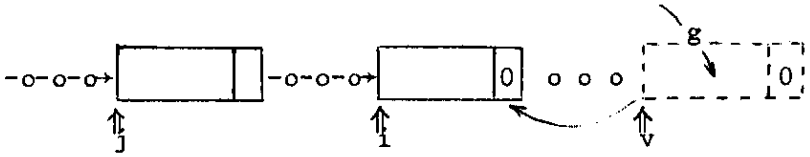
-o-o-o→ : een ketting met daaraan de schakels (gg,W)

⇑ : een wijzer

↷ : een verplaatsing of invulling

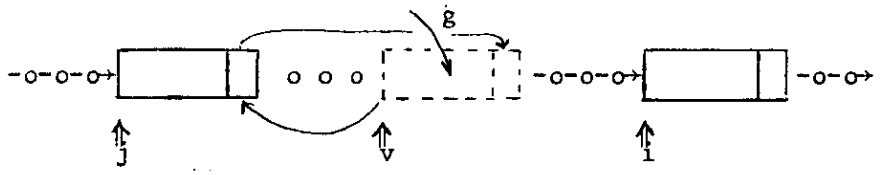
. g = gg[i,w]; men hoeft nu alleen de multipliciteit in gg[i,m] met 1 te verhogen;

. g < gg[i,w] en W[i] = 0; met andere woorden men heeft het einde van de ketting bereikt, maar moet voor g nog een schakel toevoegen en deze schakel het nieuwe einde van de ketting maken. In een plaatje:

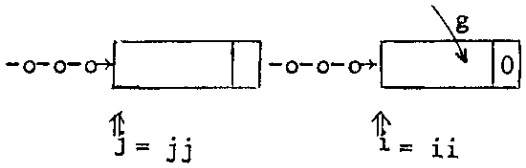


Was het array echter al vol (te constateren met een boolean full), dan zijn er geen vrije schakels en hoeft men niets te doen.

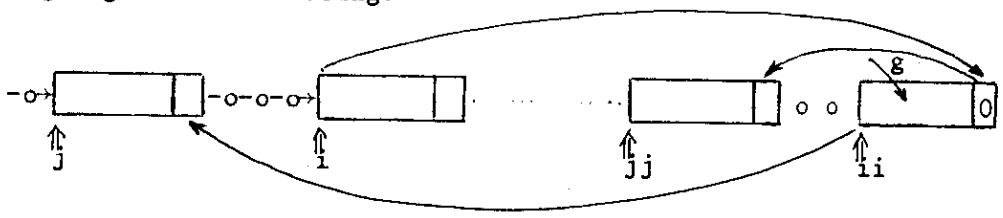
. g > gg[i,w] met andere woorden ergens in de ketting moet een schakel voor g ingelast worden. In een plaatje:



Was echter het array al vol, dan moet òf de laatste schakel opnieuw gevuld worden:



òf de laatste schakel van de ketting moet afgehaakt (en de voorlaatste als nieuwe laatste gemerkt) worden en beschikbaar gesteld voor g ergens in de ketting:



```
procedure put (g); value g; integer g;
        begin integer i,j;
              procedure fill (p,q); value p,q; integer p,q; comment: fill plaatst g en wijzer;
                      begin gg[p,w] := g; gg[p,x] := index; gg[p,m] := 1; W[p] := q end;
              j := 0; i := W[j];
              while g < gg[i,w] and W[i] ≠ 0 do begin j := i; i := W[i] end;
              if g = gg[i,w] then gg[i,m] := gg[i,m] + 1 else
                  if g < gg[i,w] then begin if not full then begin fill (v,0);
                                                                   W[i] := v;
                                                                   v := v + 1
                                                              end
                                          end schakel aan einde bijgeplaatst
                                     else if not full then begin fill (v,i);
                                                                 W[j] := v;
                                                                 v := v + 1
                                                            end
                                          else begin integer ii,jj; ii := i; jj := j;
                                                     while W[ii] ≠ 0 do begin jj := ii; ii :=W[ii]
                                                                        end;
                                                     W[jj] := 0; W[j] := ii;
                                                     fill (ii, if i = ii then 0 else i)
                                               end schakel tussengevoegd;
              if v > n + 1 then full := true
        end;


procedure indexprint; comment: globalen zijn gg, n;
        begin integer i,j,som; j := 0; i := W[j]; som := 0;
              while W[i] ≠ 0 and som < n do
                    begin j := i; i :=W[i] ;    som := som + gg[j,m] end;
              if som ≥ n then print (gg[i,x]) else print (0)
        end;
```

Het volledige programma wordt dan:

```
begin integer (n); read (n);
    begin integer array gg[0:n+1,1:3], W[0:n+1]; integer getal, index, v,w,x,m; boolean full;
          procedure put (g);     } comment: zie het voorgaande
          procedure indexprint;} voor de procedure bodies;
          read (getal); index := 1; v := 1; full := false;w := 1; x := 2; m := 3;
          gg[0,w] := W[0] := 0;
          while getal ≠ 0 do begin put (getal); read (getal); index := index + 1 end;
          indexprint
    end
end
```

Opmerking 1.

Ketting- (of lijst-) structuren worden veel gebruikt in de informatica en haar toepassingen, zoals bij o.a. formule- en polynoommanipulatie, simulatie, graphics. Er zijn vele varianten naast de hier behandelde enkelvoudige kettingstructuur. Wordt nl. in iedere schakel niet alleen naar de volgende, maar ook naar de vorige schakel verwezen, dan spreekt men van een dubbele kettingstructuur. (Door de programmeringstruc niet afzonderlijk een naar voren en een naar achter wijzende wijzer te gebruiken, maar de som van de twee wijzers, kan men soms een wijzerplaats, ten koste van iets meer rekenwerk, uitsparen. Aan begin en eind van het betreden van de ketting moeten de goede voorzieningen getroffen worden). Zijn begin en eind van de ketting ook met elkaar verbonden dan spreekt men van ringstructuren.

Opmerking 2.

Wat de afbeelding van een ketting op een array betreft, zal men wanneer iedere schakel altijd dezelfde typen bevat, geneigd zijn om een ketting af te beelden op één meerdimensionaal array. Een andere mogelijkheid (die men trouwens meestal toe zal passen wanneer het aantal velden van schakel tot schakel varieert) is om de ketting af te beelden op twee lineaire arrays, A en B. In array A plaatst men dan eerst de velden uit de eerste schakel, dan die uit de tweede schakel, enz., terwijl men in array B bijhoudt waar in array A het eerste veld uit de eerste schakel staat, waar het eerste veld van de tweede schakel, enz.

Opmerking 3.

Een probleem bij ketting- (en ook ingewikkelder data-) structuren is vaak dat men niet altijd van te voren weet hoe lang de benodigde ketting moet zijn. Men krijgt dan problemen

- met de bovengrens van het array waarop de ketting afgebeeld wordt,
- met schakels die van de ketting afgehaakt moeten worden i.v.m. het probleem waarbij de ketting gebruikt wordt.

In sommige systemen worden deze problemen ondervangen door af en toe "niet meer nodige schakels schoon te vegen" en ze weer beschikbaar te stellen als vrije schakels. Een andere, wat meer tijd vergende, maar eenvoudiger te overziene methode maakt gebruik van een "stapel" vrije schakels. Zowel het pakken van een vrije schakel, als het teruggeven van een overbodig geworden schakel gebeurt dan aan de top van de stapel.

2.3. Stapelstructuur; recursie

Opgave

Een bekende puzzle draagt de naam "Torens van Hanoi". Gegeven zijn drie pennen, te nummeren 1, 2 en 3. Op pen 1 liggen n schijven van afnemende diameter, waarbij de onderste schijf de grootste is.De andere pennen zijn aanvankelijk leeg. De bedoeling van het spel is alle schijven over te brengen naar pen 2, daarbij zonodig gebruik makend van pen 3,doch zodanig dat telkens slechts 1 schijf verplaatst wordt en dat tijdens het proces nooit ergens een schijf komt te liggen op een schijf met kleinere diameter.

Discussie

Door proberen ziet men dat voor kleine n de oplossing eenvoudig is aan te geven; bijvoorbeeld (de pijl heeft steeds betrekking op de bovenste schijf):

n = 1	1 → 2
n = 2	1 → 3, 1 → 2, 3 → 2
n = 3	1 → 2, 1 → 3, 2 → 3, 1 → 2, 3 → 1, 3 → 2, 1 → 2

Voor grotere n wordt het al vrijwel onmogelijk om het spel met het minimum aantal"zetten" te spelen. Men kan voor willekeurige n het probleem echter als volgt transformeren:

- verplaats de bovenste n - 1 schijven van 1 naar 3,
- verplaats de onderste schijf van 1 naar 2,
- verplaats de n - 1 schijven van 3 naar 2,

Het probleem van n schijven is daarmee gereduceerd tot het twee keer verplaatsen van n - 1 schijven en de verplaatsing van nog een schijf. Door inductie ziet men ook dat het minimum aantal verplaatsingen gelijk is aan $2^n - 1$. Met deze reductie kan men doorgaan tot het oorspronkelijke probleem gereduceerd is tot een reeks verplaatsingen van telkens 1 schijf.

Het hele proces wordt blijkbaar beheerst door twee handelingen, nl. door (voor P, Q en R is een permutatie van 1, 2, 3 te lezen):

- verplaats-toren (van m schijven van pin P naar pin Q via pin R)
- verplaats-schijf (van pin P naar pin Q)

waarbij verplaats-schijf (in het programma de procedure movedisk)

- een bijzonder geval is van verplaats-toren voor m = 1 en een rechtstreeks verplaatsen van P naar Q zonder van R gebruik te maken,
- uitgeschreven kan worden met: nieuwe regel; write (P); write (-Q)

Na deze voorbereidingen ziet men dat een recursief werkend programma direct op te schrijven is, terwijl de juistheid volgt uit inductie naar het aantal schijven.

```
begin procedure movetower (m,P,Q,R); value m,P,Q,R; integer m,P,Q,R;
                begin if m ≠ 1 then begin movetower (m - 1,P,R,Q);
                                           movedisk (P,Q);
                                           movetower (m - 1,R,Q,P)
                                     end
                                else       movedisk (P,Q)
                end;
      movetower (n,1,2,3)
end
```

Bij deze oplossing is dus gebruik gemaakt van het feit dat in Algol procedures zichzelf mogen aanroepen. In programmeertalen waarin dit niet toegestaan is (bijvoorbeeld Fortran), zal men een kunstgreep moeten gebruiken voor deze recursie. Met name in deze opgave moet men bewerkstelligen dat het proces movetower, dat gekenmerkt wordt door de vier variabelen m, P,Q en R, expliciet vervangen wordt door deelprocessen met lagere waarden van m, enz. Dit gaat dan bijvoorbeeld met het bijgaande programma.
Waar het op neer komt is dat een "stapel"-array ST met bijbehorende stapelindex s wordt ingevoerd, waarbij hier in iedere "stapellaag" (dus voor een bepaalde s-waarde) vier getallen opgeborgen kunnen worden. Een procedure push verhoogt de stapelwijzer en drukt dan vier getallen op de stapel. Een procedure pull haalt vier getallen van de stapel en verlaagt vervolgens de stapelwijzer. Als in een programma slechts push en pull gebruikt worden om met de stapelwijzer te spelen, kan men dus alleen maar de "bovenste stapelplaats" bereiken.

Opmerking. De procedures push en pull moeten "beveiligd" zijn, zodat steeds gecontroleerd wordt of de stapelwijzer nog wel in het toegelaten gebied ligt. Voor N moet men een voldoend groot getal nemen; in dit voorbeeld kan men aantonen dan 2n - 1 stapellagen zeker genoeg zullen zijn. De verder niet uitgewerkte procedure paniek is een standaardfunctie die moet bewerkstelligen dat in deze panieksituaties de programma-uitvoering netjes afgebroken wordt nadat de nodige boodschappen zijn afgedrukt.

```
begin integer m,n,P,Q,R,s; integer array ST[1:N, 1:4];
      comment: zoals reeds eerder vermeld, moet men voor N een groot getal
               invullen, s is de laatst gevulde stapelplaats;
      procedure push (a,b,c,d); value a,b,c,d; integer a,b,c,d;
                begin s := s + 1;
                      if s ≤ N then begin ST[s,1] := a; ST[s,2] := b;
                                          ST[s,3] := c; ST[s,4] := d
                                    end
                               else paniek
                end;
      procedure pull (a,b,c,d); integer a,b,c,d;
                begin if s > 0 then begin a := ST[s,1]; b := ST[s,2];
                                          c := ST[s,3]; d := ST[s,4];
                                          s := s - 1
                                    end
                               else paniek
                end;
      s := 0; read (m);
      push (m,1,2,3);
      while s > 0 do begin pull (n,P,Q,R);
                           if n ≠ 1 then begin push (n - 1,R,Q,P);
                                               push (1,P,Q,R);
                                               push (n - 1,P,R,Q);
                                         end
                                    else movedisk (P,Q)
                     end
end
```

In het eigenlijke programmadeel volgend op de procedure declaraties ziet men dat de stapelwijzer wordt geïnitialiseerd en dat de beginsituatie van de puzzle in de eerste stapellaag gedrukt wordt (voor m moet men een getal (in)-lezen). Wat daarna in de while-lus gebeurt, realiseert men zich het best door het programma voor een kleine n waarde (bijvoorbeeld 3) na te lopen. Men ziet dan hoe de stapelwijzer nogal eens heen en weer gaat. Merk ook op dat de drie opeenvolgende push-bewerkingen in de while-lus in omgekeerde volgorde staan vergeleken met het eerste programma. Dit is echter nodig omdat de laatste push het eerste uit te voeren proces moet bevatten, daar dit het eerste is dat uit de stapel gehaald wordt!

Een soortgelijke stapeltechniek (uiteraard met, afhankelijk van het probleem, andere aantallen getallen in iedere stapellaag) zal men steeds moeten toepassen in programmeertalen waarin geen recursie toegestaan is. In Algol is recursie direct mogelijk omdat het vertaalprogramma dat een Algoltekst omzet in machinetaal, meteen in het machinetaalprogramma voor stapelvoorzieningen zorgt.

Opmerkingen.

1. Is er ook een iteratieve oplossing mogelijk, zonder een stapel te introduceren? Dit kan inderdaad na eerst (door inspectie van de oplossingen voor kleine n en dan door inductie) enkele eigenschappen van de oplossing aangetoond te hebben. (zie W.W. Rouse Ball & H.S.M. Coxeter - Mathematical Recreations and Essays).
2. Het is natuurlijk ook mogelijk om een stapel via een ketting af te beelden, omdat men een stapel op kan vatten als een geordende rij schakels. Aangezien bij een stapel alleen de top van de stapel bereikbaar hoeft te zijn en men niet door een ketting heen wil lopen om het uiteinde te bereiken, ligt het voor de hand om de top van de stapel af te beelden op de eerste schakel (na een vaste schakel) van de ketting, enz.
3. Een stapel is wederom een eenvoudige lineaire structuur, doch nu één waarbij toevoeging en verwijdering van knopen slechts aan één kant plaats vindt. Men noemt een stapel ook wel een L(ast)I(n)F(irst)O(ut) wachtlijn.
4. Een zg. dubbelstapel (decque) is een lineaire structuur waarbij aan beide einden toevoeging en verwijdering van knopen mogelijk is.

2.4. Boomstructuur

Opgave

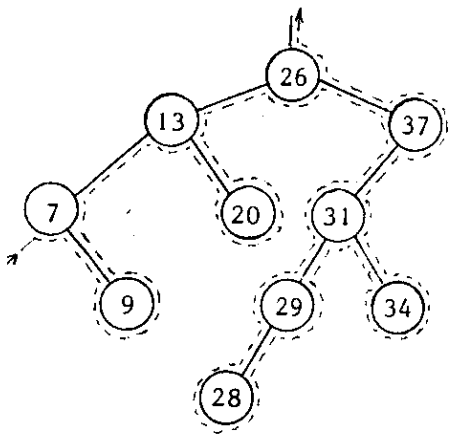
Een rij in te lezen natuurlijke getallen, afgesloten met een nul, moet gesorteerd weer afgedrukt worden.

Discussie

Men zou er in de eerste plaats natuurlijk over kunnen denken om de rij getallen in te lezen in een array, waarvan gehoopt wordt dat het groot genoeg is. De ingelezen getallen kunnen vervolgens in het bezette deel van het array gesorteerd worden met de vroeger behandelde algorithme. Deze methode zal nu echter niet gevolgd worden, doch in plaats daarvan de volgende die toegelicht wordt aan het voorbeeld van de rij getallen

26, 37, 13, 7, 31, 9, 34, 29, 20, 28 .

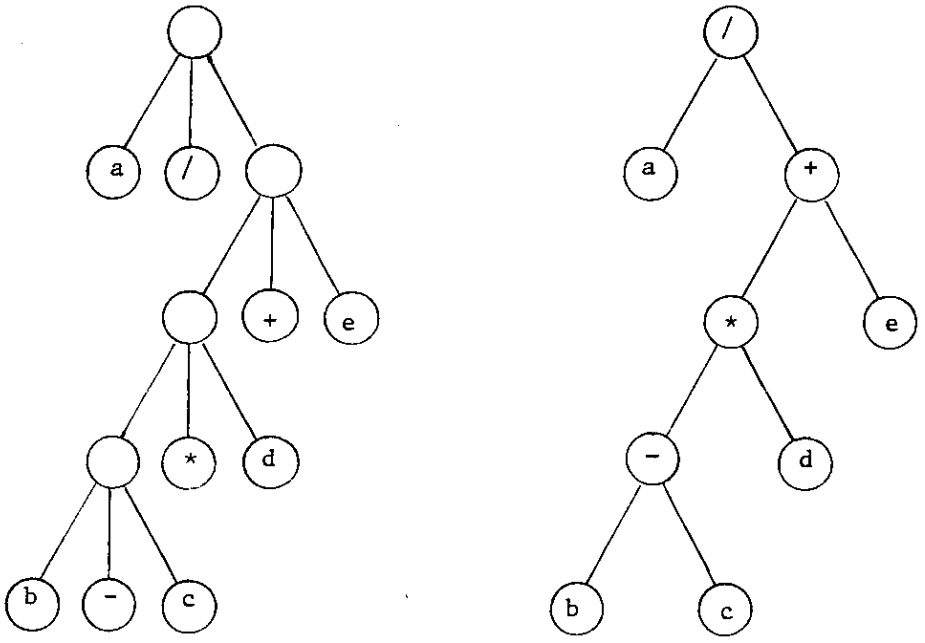
- plaats het eerste getal in een eerste knoop, die verder "wortel" genoemd wordt,
- plaats het volgende getal in een volgende knoop linksonder, resp. rechtsonder de voorgaande knoop wanneer dit volgende getal kleiner, resp. groter dan (of gelijk aan) het eerste getal is,
- herhaal dit plaatsingspatroon, waarbij onder iedere knoop dus ten hoogste twee volgende knopen mogen hangen; dit leidt uiteindelijk tot het hiernaast getekende plaatje,
- "wandelt" men tenslotte met een "links-onder voorkeur" door de gevormde figuur dan ontmoet men de ingelezen getallen gesorteerd naar grootte.



Hetgeen gevormd is, heet een "binaire (omdat aan iedere knoop ten hoogste twee andere knopen hangen) boom". Het is een bijzonder geval van de datastructuur boom (K,R) bestaande uit een verzameling knopen k_i ($i > 0$) en een verzameling relatie R die aan de volgende voorwaarden voldoen:

. er is één uitverkoren knoop, die wortel genoemd wordt,
. met uitzondering van de wortel heeft iedere knoop precies één voorganger, waarmee hij (door een pijl naar die knoop) verbonden is.

Terwijl in de hiervoor behandelde lineaire structuren iedere knoop gekenmerkt was door het hebben van één voorganger (behalve ook de eerste knoop) èn één opvolger, kan een boom dus meer opvolgers per knoop hebben. De eindknopen van een boom (die dus geen opvolgers meer hebben) heten ook wel bladeren, de verbinding (pijl) van een knoop naar zijn opvolger heet ook tak en de verbinding van de wortel naar een willekeurige andere knoop via de takken heet een weg. De lengte van een weg (en het niveau van zijn eindknoop) is het aantal pijlen van die weg. Een boom is geordend als van de opvolgers van iedere knoop de volgorde vastgelegd is. De orde van een knoop is het aantal opvolgers van die knoop, de orde van een boom is het maximum van de orde van zijn knopen.
Bomen gebruikt men in veel vakgebieden om prioriteitenschema's of hierarchische verbanden te representeren (voorbeelden zijn stambomen, organisatiestructuren, classificatieschema's, enz.). In de informatica worden ze o.a. gebruikt om syntaxdiagrammen (zie appendix A) of expressies te representeren. Dit kan op verschillende manieren gebeuren zoals hieronder getekend is voor de expressie a/((b - c) * d + e).



Om in Algol 60 met bomen te kunnen werken moet men de boomstructuur weer afbeelden op een array. Voor het relatiedeel gebruikt men bijvoorbeeld een meer-dimensionaal wijzerarray W met evenveel kolommen als de orde van de boom bedraagt. Voor het knopendeel is een knopenarray KA nodig met evenveel kolommen als er variabelen in de knoop vast te leggen zijn. Aangezien in het volgende zowel wijzers als knoopvariabelen allen integers zijn, worden deze arrays W en KA tot één array verenigd (dat tree genoemd zal worden).
Bij een willekeurige boom kan altijd op éénduidige wijze een corresponderende binaire boom opgegeven worden met behulp van de volgende regel:

neem als linkertak van iedere "vaderknoop" de pijl naar de "oudste zoon" en als rechtertak de pijl naar de "eerstvolgende broer" van die vader.

Om deze reden en omdat binaire bomen het meest gebruikt worden is de verdere discussie beperkt tot binaire bomen. De opslag van de eerste boom in deze paragraaf in een array (met N plaatsen) is daar bijvoorbeeld de volgende:

nr.	getal	l-tak	r-tak
1	26	3	2
2	37	5	
3	13	4	9
4	7		6
5	31	8	7
6	9		
7	34		
8	29	10	
9	20		
10	28		

De manier waarop bomen geïntroduceerd zijn, suggereert een recursieve definitie; nl. een boom is leeg of bestaat uit een wortel en 0, 1 of 2 bomen (de "linker"-, "rechter"-boom). Het "vullen" van een boom is dan het vullen van de wortel als deze nog leeg is, en anders het vullen van de daarvoor in aanmerking komende boom. Verifieer zelf dat deze definitie van vullen precies overeenstemt met hetgeen eerder beschreven is.

Een (recursief werkend) programma om een getal in een boom, afgebeeld op een niet-vol array te plaatsen, zal bijv. de volgende structuur hebben:

```
procedure build (boom)
        begin if "array niet-vol" then
                if "wortel = leeg"
                    then "vul wortel, verklein opslagruimte en
                          onderzoek of array nog steeds niet-vol is"
                    else if getal < wortel
                            then begin if "geen linkertak" then "maak linkertak";
                                       build (linker sub-boom)
                                 end
                            else begin if "geen rechtertak" then "maak rechtertak";
                                       build (rechter sub-boom)
                                 end
        end
```

De juistheid van deze procedure volgt door inductie naar het aantal reeds geplaatste knopen in de boom. Uitgeschreven krijgt men

```
procedure build (n); integer n; comment: array tree, arraywijzer u, getal g en full zijn globalen;
        begin if not full then
                begin if tree[n, 1] = 0
                        then begin tree[n, 1] : = g;
                                   if u = N then full : = true else u : = u + 1
                             end
                        else if g < tree[n, 1]
                               then begin if tree[n, 2] = 0 then tree[n, 2] : = u;
                                          build (tree[n, 2])
                                    end
                               else begin if tree[n, 3] = 0 then tree[n, 3] : = u;
                                          build (tree[n, 3])
                                    end
                end
        end
```

Hierin is u een variabele die aangeeft dat u regels in het array tree zijn ingevuld, full een boolean die true is wanneer array tree volledig ingevuld is, en tenslotte n het nummer van de beschouwde knoop is.

In dit geval is het niet veel moeilijker om een niet-recursief programma op te schrijven. Terwijl bij een recursief programma de aandacht gericht is op het vullen van bomen, moet men bij een niet-recursief de aandacht richten op het feit dat men een lege plaats moet vinden om het getal te plaatsen.

```
procedure build (n); value n; integer n; comment: tree, u, g en full zijn globalen;
        begin if not full then
                    begin boolean placed; placed : = false;
                          while not placed do
                               begin if tree[n, 1] = 0
                                       then begin tree[n, 1] : = g; placed : = true;
                                                  if u = N then full : = true else u : = u + 1
                                            end
                                       else if g < tree[n, 1]
                                              then begin if tree[n, 2] = 0 then tree[n, 2] : = u;
                                                         n : = tree[n, 2]
                                                   end
                                              else begin if tree[n, 3] = 0 then tree[n, 3] : = u;
                                                         n : = tree[n, 3]
                                                   end
                               end
                    end
        end
```

Opmerking:
n moet in de valuelijst geplaatst worden omdat de n in de procedurebody links van een := symbool niet door de actuele parameter vervangen mag worden.

Beide procedures build zijn "beveiligd", want in het begin van de procedures wordt onderzocht of er in het array tree nog ruimte is om een nieuwe knoop toe te voegen. Als getallen die niet meer in de boom kunnen worden ondergebracht, wel worden ingelezen, dan krijgt men het volgende programma

```
begin integer array tree[1 : N, 1 : 3]; comment: voor N moet een groot getal ingevuld;
      integer t, u; boolean full;
      procedure build (n); integer n;
                comment: zie de voorgaande programma's voor de body;
      for t : = 1 step 1 until N do for u : = 1 step 1 until 3 do tree[t, u] : = 0;
      u : = 1; full : = false; read (g);
      while g ≠ 0 do begin build (1); read (g) end
end
```

```
begin procedure printtree 1 (n); integer n ;
                begin if tree[n, 2] ≠ 0 then printtree 1 (tree[n, 2]);      - linkerboom
                      print (tree[n, 1]);                                   - wortel
                      if tree[n, 3] ≠ 0 then printtree 1 (tree[n, 3])       - rechterboom
                end;
      printtree 1 (1)
end
```

Het afdrukken van de ingelezen getallen in de goede volgorde gebeurt (juistheid volgt weer door inductie naar het aantal knopen) met een recursief werkende procedure printtree heel eenvoudig. Door het even te proberen kan men constateren dat het opschrijven van een niet-recursief werkend programma veel meer werk vereist. Men kan dit doen door net zoals bij de torens van Hanoi een stapel met push en pull procedures in te voeren. Deze stapel geeft de mogelijkheid om in de goede volgorde expliciet door het array tree heen te wandelen, daarbij o.a. noterend welke knopen al bezocht maar nog niet afgewerkt zijn. Een knoop die een onafgewerkte linkertak heeft, mag nog niet afgedrukt worden en wordt om hem later te behandelen op de stapel geplaatst. Wordt een knoop afgedrukt, dan moet daarna zijn rechtertak, indien aanwezig, afgedrukt worden. Er zijn dan:

- knoopnummers die nog niet bekeken zijn (ze komen later op de stapel)
- knoopnummers van reeds afgedrukte knopen (ze stonden eens op de stapel, maar nu niet meer)
- knoopnummers van net bereikte knopen (die komen met een + teken op de stapel)
- knoopnummers die al bekeken zijn, maar nog niet verwijderd mogen worden omdat ze nog afgedrukt moeten worden (die komen met een - teken op de stapel).

```
procedure printtree 2; comment; niet recursief werkend;
        begin integer array stapel[1 : M]; integer s, n;
            procedure push (n); integer n;
                    begin if s < M then begin s : = s + 1; stapel[s] : = n end
                                   else paniek
                    end;
            boolean procedure pull (n); integer n;
                   begin if s > 0 then begin n : = stapel[s]; s : = s - 1;
                                                pull : = true
                                       end
                                  else       pull : = false
                   end;
            s : = 0; push (1);
            while pull (n) do
                 begin if tree[abs (n), 2] ≠ 0 and n < 0
                         then begin push (-n); push (tree[n, 2]) end
                         else begin print (tree[abs (n), 1]);
                                    if tree[abs(n),3] ≠ 0 then push (tree[abs(n),3])
                              end
                 end
        end
```

De werking van deze procedure is blijkbaar de volgende:

- met pull (n) wordt het laatstgeplaatste knoopnummer van de stapel gehaald
- heeft deze knoop een linkertak en is het knoopnummer positief, dan wordt
  . dit knoopnummer met een min-teken op de stapel teruggeplaatst (daarmee voorkomend dat het ooit nog eens op de aanwezigheid van een linkertak onderzocht wordt)
  . het "wortelnummer" van de linkerboom op de stapel geplaatst
- heeft deze knoop geen linkertak of was het al eens op een linkertak onderzocht, dan wordt
  . de knoop afgedrukt (het nummer wordt niet meer teruggeplaatst op de stapel!)
  . het "wortelnummer" van een eventuele rechterboom op de stapel geplaatst.

Het aantal (M) stapelplaatsen moet tenminste gelijk zijn aan het aantal knopen, omdat dit nodig is voor het ongunstigste geval van een boom met alleen maar linkertakken.

De procedure printtree 2 is niet de enige oplossing. Een andere is

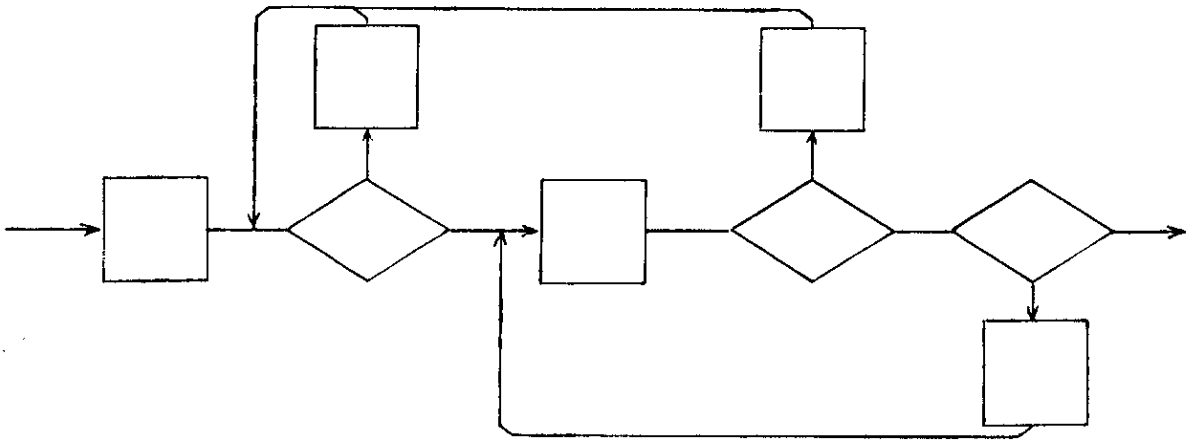
```
procedure printtree 3;
        begin comment declareer net als bij printtree 2 een stapel, de
              variabele n en de push en pull procedures
              n : = 1;
              while tree[n, 2] ≠ 0 do begin push (n); n : = tree[n, 2] end;
              push (n);
              while pull (n) do
                    begin print (tree[n, 1]);
                          if tree[n, 3] ≠ 0 then
                             begin n : = tree[n, 3];
                                   while tree[n, 2] ≠ 0 do
                                         begin push (n); n : = tree[n, 2] end;
                                   push (n)
                             end
                    end
        end
```

In deze procedure wordt bij de wortel (en iedere rechterboom) eerst een zo lang mogelijk pad van linkertakken "op de stapel geplaatst". Breekt deze reeks af (omdat er geen linkertak meer is) dan wordt de inhoud van de laatste knoop afgedrukt en de wortel van een eventuele rechterboom op de stapel geplaatst, enz.

Opmerking. Om het laatste programma korter en "efficienter" te maken zou men in de verleiding kunnen komen om de push en pull procedures uit te schrijven en op te merken dat het opbouwen van een reeks van linkertakken twee keer voorkomt. Men komt dan tot het volgende programma:

```
procedure printtree 4;
        begin integer array stapel[1 : M]; integer s, n;
              s : = 0; n : = 1;
           1: while tree[n, 2] ≠ 0 do begin s : = s + 1; stapel[s] : = n;
                                            n : = tree[n, 2]
                                      end;
           m: print (tree[n, 1]);
              if tree[n, 3] ≠ 0 then begin n : = tree[n, 3]; goto 1 end;
              if s > 0 then begin n : = stapel[s]; s : = s - 1; goto m end
        end;
```

Niet alleen is dat programma nauwelijks te doorgronden, maar ook de juistheid ervan is moeilijk aan te tonen. Verwonderlijk is dit niet, als naar het stroomdiagram van dit programma gekeken wordt:



Het is duidelijk dat men aan deze efficiency verleidingen niet moet toegeven en in talen als Fortran die geen recursie toelaten, expliciet stapelprocedures moet invoeren.

Opmerking 2. Met de gekozen afbeelding van een boom op een array kan men gemakkelijk "in de boom afzakken" (van de wortel naar de bladeren gaan). Dit betekent bij wandelingen door een boom dat knoopnummers op een stapel bewaard moeten worden. Een alternatief hiervoor is het toevoegen van een derde wijzer bij iedere knoop die naar de voorganger wijst.

Opmerking 3. Het eerder gegeven printprogramma om een boom uit te printen is een voorbeeld van een "in-order" wandeling door een boom. Met

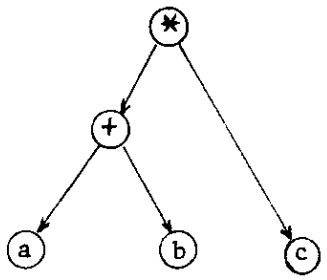
```
print (tree[n,1])
if tree[n,2] ≠ 0 then printtree (tree[n,2])
if tree[n,3] ≠ 0 then printtree (tree[n,3])
```

krijgt men de zg. "pre-order" wandeling, en met

```
if tree[n,2] ≠ 0 then printtree (tree[n,2])
if tree[n,3] ≠ 0 then printtree (tree[n,3])
print (tree[n,1])
```

de "post-order" wandeling (en wel alle drie van links naar rechts).
Past men dit toe op de hiernaast getekende boom, dan krijgt men met:

in-order : (a + b) * c - infix notatie
pre-order : * + abc - prefix of Poolse notatie
post-order: ab + c * - postfix of inverse-Poolse notatie



voor dezelfde expressie (bij de infix notatie moet men bovendien haakjes gebruiken i.v.m. de prioriteiten van operatoren). Bij de prefix notatie gaan per definitie de operatoren vooraf aan de twee operanden, bij de postfix notatie volgen ze op de operanden. Slechts bij de infix notatie zijn haakjes nodig. Zo wordt bijvoorbeeld a + b * c in postfix notatie abs * +.

2.5. Backtracking

Veel problemen hebben het karakter van het zoeken van combinaties van elementen (van een eindige verzameling) die, wanneer ze aan bepaalde voorwaarden voldoen, een "oplossing" van het probleem vormen. Het eerst vormen van alle combinaties en dan onderzoeken welke voldoen, zou veelal een te tijdrovende zaak zijn. Wanneer tussen de elementen van de verzameling een zekere ordening bestaat (of te definiëren is), is het als regel verstandiger om de gezochte combinaties geordend en systematisch op te bouwen. Meestal brengt nl. de probleemstelling met zich mee dat bij het stap voor stap opbouwen van een combinatie, beginnend met bijvoorbeeld een lege combinatie, bepaalde keuzen buiten beschouwing kunnen worden gelaten, of dat blijkt dat een ingeslagen weg tot opbouw van een combinatie niet tot het goede resultaat kan leiden. (Soms zijn de problemen van een optimum-kosten type, in welk geval bijvoorbeeld een betere oplossing al bekend is tijdens het zoeken van een nieuwe oplossing. Verder zoeken vanuit het bereikte punt heeft bij deze "branch and bound" problemen dan geen zin meer.) Vanuit dit bereikte punt gaat men dan weer systematisch terug (backtracking) totdat een bepaald punt bereikt is waar mogelijk een alternatieve opbouw mogelijk is, daarbij systematisch gebruikmakend van de ordening in de beschikbare elementen. Bij iedere opbouw moet men zo snel mogelijk naar een goed eindpunt proberen door te stoten, doch daarbij wel noteren op welke reeds gevonden punten later nog een alternatieve weg ingeslagen kan worden.
Zeer schematisch ziet deze oplossingsmethode er dus als volgt uit:

begin "begin met een lege combinatie"
"zolang geen oplossing gevonden is moet men de combinatie stap voor stap uitbreiden overeenkomstig de eisen van het probleem, daarbij de voorkeur gevend aan de elementen met de laagste ordening"
"heeft men een oplossing of een doodlopende combinatie bereikt, ga dan systematisch langs de afgelegde weg terug en probeer achtereenvolgens bij de daarbij in aanmerking komende punten weer met de alternatieve, geordende elementen een weg te vinden"
end

Deze vage schets wordt nu aan de hand van bekende opgaven toegelicht.

2.5.1. Knapzakprobleem

Gegeven zijn n artikelen, waarvan het gewicht g_i (> 0) is en de (bijvoorbeeld calorische) waarde w_i (> 0). Gevraagd wordt een knapzak te vullen met een combinatie van deze artikelen, zodanig dat een zeker totaalgewicht G niet overschreden wordt en dat de totale waarde W van deze artikelen maximaal is.

Zonder beperking kan aangenomen worden dat de artikelen zodanig genummerd zijn dat

$$w_1/g_1 \geq w_2/g_2 \geq \ldots w_n/g_n$$

m.a.w. de meest waardevolle artikelen per gewichtseenheid staan voorop. Een aanleiding tot deze ordening is natuurlijk dat gehoopt kan worden dat een combinatie van de meest waardevolle artikelen de optimale vulling geeft.
Met n artikelen zijn 2^n combinaties te maken (voor n = 10 dus al 1024), zodat de gedachte om gewicht en waarde van al deze combinaties te berekenen voor realistische waarden van n gauw verlaten zal worden. Van veel van deze combinaties zal trouwens in een vroeger stadium van samenstelling al vaststaan dat het totaalgewicht overschreden wordt, zodat ze niet meer verder vervolgd hoeven te worden t.a.v. toevoeging van andere artikelen. Gedacht kan dan worden aan de volgende systematische oplossingsmethode. Heeft men de knapzak gevuld met een zekere combinatie van de eerste (m-1)-artikelen (bijv. te beschrijven met een boolean array, initieel met false gevuld), waarbij m-1 ≤ n en vulgewicht < G, dan kunnen er twee volgende situaties (dus na bekijken van m artikelen) zijn:

- als $vulg + g_m \leq G$, voeg dan artikel m toe, zodat de toestand (m-1,vulg,vulw) overgaat in $(m, vulg + g_m, vulw + w_m)$
- artikel m wordt niet toegevoegd en de toestand (m-1,vulg,vulw) gaat over in (m,vulg,vulw).

Voor m = n+1 is een eindsituatie bereikt, terwijl een gemeenschappelijke beginsituatie gegeven wordt door (0,0,0). Uitgaande van deze beginsituatie krijgt men zodoende een binaire "toestandenboom", waarin sommige oplossingswegen echter slechts aan één kant vervolgd hoeven te worden omdat daarvoor $delg \equiv G - vulg < g_m$ is. Deze methode is dus al voordeliger dan de methode van het berekenen van alle 2^n combinaties.

Een verdere beperking op het aantal te beschouwen gevallen is echter mogelijk door ook te letten op de maximaal mogelijke toename van vulw in een

bepaalde knoop (m-1,vulg,vulw). Deze is namelijk

$$\text{delw(delg,m)} = \begin{cases} 0 \text{ als } m = n \text{ of } vulg = G \\ \text{entier } (delg * w_m/g_m + 0.05) \end{cases}$$

(de laatste betrekking wegens de ordening van w/g; de constante 0.05 is voor de afronding toegevoegd). Als dus vulw + delw in die knoop kleiner is dan de waarde W van een al eerder gevonden (maar mogelijk niet optimale) knapzakvulling heeft het geen zin meer om opvolgers van die toestand te bepalen.
Na het voorgaande zal duidelijk zijn dat het streven moet zijn om van de toestandenboom niet meer dan het strikt nodige te construeren. Tijdens de constructie van een weg naar een eindblad, moeten in bijv. boolean array art[1:n] de toegevoegde artikelen genoteerd worden. Wordt hiermee een betere oplossing gevonden dan moet de vorige notering van geselecteerde artikelen in boolean array vul[1:n] overschreven worden.
Voor het vullen van het array art[1:n] is een procedure pak nodig. Hierin zit als basisoperatie dat wanneer de knapzak gevuld is met een combinatie van de eerste i-1 artikelen met een totaalgewicht vulg en een totale waarde vulw, zo nodig en mogelijk het artikel i wordt toegevoegd. Nodig is het namelijk volgens de voorgaande discussie alleen wanneer $i \leq n$ en $vulw + delw \geq W$ is (oorspronkelijk is alleen een ondergrens van W bekend, nl. 0 of - met iets meer rekenen - de max. waarde van w waarvoor de bijbehorende waarde van g kleiner dan G is). Om deze basisoperatie gemakkelijk te programmeren worden als formele parameters van pak alleen i, vulg en vulw gebruikt. De procedure pak doet nog meer; zoekt nl. successievelijk mogelijke vullingen van de knapzak (dan is $i > n$) en registreert de vulling met de hoogste waarde W in een boolean array vul[1:n]. De variabelen W, art en vul worden echter als globale grootheden gehanteerd in de procedure body.

Voor een toestand (i-1,vulg,vulw) die wel potentieel goede aanvullingsmogelijkheden heeft moeten dus de opvolgertoestanden gemaakt worden, nl. $(i,vulg + g_i,vulw + w_i)$ als dat kan i.v.m. de gewichtsbeperking en (i,vulg,vulw) wat altijd kan (artikel i wordt niet toegevoegd). In het eerste geval moet genoteerd worden met art_i := true dat artikel i toegevoegd is, maar meteen daarna kan met pak $(i+1,vulg + g_i,vulw + w_i)$ een verdere vulling ter hand worden genomen. Aangezien deze verdere vulling

later een doodlopende weg kan blijken te zijn, moet direct na pak (i=1,vulg + g_i,vulw + w_i) het artikel i weer uit de knapzak verwijderd worden met art_i := false. De structuur van de procedure pak is dus de volgende.

```
if "laatste artikel nog niet geprobeerd"
    then if "potentieel betere vulling mogelijk"
            then if "gewicht wordt niet overschreden"
                    then "voeg artikel i toe";
                         "probeer verder te vullen met art.i in het
                          assortiment"
                         "verwijder artikel i"
                 fi;
                 "probeer verder te vullen zonder art.i in het assortiment"
         fi
    else "noteer gevonden oplossing als deze beter is dan vorige"
fi
```

Bijgaand zijn zowel de procedures pak en delw als het volledige programma uitgeschreven in Algol 60. Veiligheidshalve wordt in het programma gecontroleerd of de w/g waarden geordend zijn (wat zou er kunnen gebeuren als dit niet het geval is?). Is dat niet het geval dan zorgt een verder niet uitgewerkte procedure paniek er voor dat een betreffende boodschap wordt afgedrukt en dat de programmaverwerking afgebroken wordt. Het zal duidelijk zijn dat deze recursieve procedure pak er een is van het type "depth-first" en dat de "bounds" hier gevormd worden door de gewichtsbeperking en door de groeimogelijkheden voor vulw.
Het opstellen van een niet-recursief werkend programma wordt weer ingewikkeld omdat nu expliciet een stapel moet worden ingevoerd, waarop men dan de bereikte en nog niet afgewerkte toestandsknopen moet registreren.

```
begin integer n; read (n); if n > 0 then
      begin integer array g,w[1:n]; boolean array vul[1:n];
            integer k,G,W;


            procedure pak (i,vulg,vulw); value i,vulg,vulw; integer i,vulg,vulw;
               begin integer delg,j; boolean array art[1:n];
                     integer procedure delw(i,delg); value i,delg; integer i,delg;
                             delw := if i = n then 0
                                          else entier (delg * w[i]/g[i] + 0.05);
                     delg := G - vulg;
                     if i ≤ n
                        then begin if vulw + delw (i,delg) ≥ W then
                                          begin if g[i] ≤ delg then begin art[i] := true
                                                                          pak (i+1,vulg + g[i],vulw + w[i]) ;
                                                                          art[i] := false
                                                                    end;
                                                pak (i+1,vulg,vulw)
                                          end
                                   end
                        else if vulw > W then begin W := vulw;
                                                    for j := 1 step 1 until n do vul[j] := art[j]
                                              end
               end;
            for k := 1 step 1 until n do begin read (g[k]; read w[k]; art[k] := false end;
            read (G); W := 0;
            for k := 1 step 1 until n-1 do begin if w[k]/g[k] < w[k+1]/g[k+1] then paniek end;
            pak (1,0,0);
            write (W); for k := 1 step 1 until n do write (if vul[k] then k else "spatie")
      end
end
```

Voor een niet-recursieve oplossing kan men hier het array art[0:n] tevens als stapel gebruiken, daarin weer met false, resp. true noterend of het betreffende artikel i ($1 \leq i \leq n$) in de knapzak geplaatst is (de uitbreiding met art_0 = true weerspiegelt dat alleen art_0 in de knapzak betekent dat de knapzak leeg is). Zoals bij de recursieve oplossing zal men nu proberen om de knapzak te vullen met de opeenvolgend meest waardevolle artikelen. Dit zal kunnen ophouden:

. omdat het laatste artikel voor plaatsing in de knapzak is bekeken ($i \geq n$)
. omdat het laatst toegevoegde artikel een gewichtsoverschrijding met zich meebrengt (vulg > G),
. omdat het laatst toegevoegde artikel tot een situatie leidt waarbij vulw + delw < W.

In alle drie gevallen moet men het laatst toegevoegde artikel, zeg m, uit de knapzak verwijderen (art[m] := false), vulg en vulw dienovereenkomstig aanpassen en het proces vervolgen met artikel m+1. Bovendien kan op dit moment ook de waarde van W zo mogelijk verhoogd worden. Wanneer kortheidshalve false door 0 en true door 1 wordt voorgesteld brengt dit de volgende transformaties met zich mee:

```
1,1,..... 1,1,0, ..... 0 (laatste vulling mis)
            ↓
1,1,..... 1,0,0, ..... 0 (dus gecorrigeerd)
- - - - - - - - - - - - - - - - - - -
- - - - - - - - - - - - - - - - - - -
1,1,..... 1,0,1,..... 1,0,..... 0 (nu weer laatste vulling mis
                      ↓
1,1,..... 1,0,1,..... 0,..... 0 (dus gecorrigeerd)
```

Men ziet zodoende dat "van rechts komend zich steeds meer nullen tussen de enen indringen" ("de meest rechtse een wordt telkens door een nul vervangen") en dat dit proces afloopt zodra er alleen nog maar nullen staan. Dit correspondeert met het uit de knapzak halen van het laatst toegevoegde artikel en het dan weer verder proberen vullen met de minder waardevolle artikelen. Gebruik makend van dezelfde variabelen als bij de recursieve oplossing krijgt men voor het hoofddeel van de algorithme

```
W := vulw := vulg := i := 0; ready := false; art[0] := true;
while not ready
      do while i < n and vulw + delw ≥ W and vulg ≤ G
               do  i := i+1; art[i] := true;
                   vulw := vulw + w[i]; vulg := vulg + g[i]
               od;
         W := max(if vulg > G then vulw - w[i] else  vulw fi,W);
         while not art[i] do i := i-1 od;
         ready := i = 0;
         if not ready then art[i] := false;
                           vulw := vulw - w[i]; vulg := vulg - g[i]
         fi
      od
```

Het inlezen van de gegevens en het noteren van de beste oplossing zijn hier dus niet uitgeschreven.

2.5.2. Opgave

Gegeven is een cirkel met daarop 8 posities. Gevraagd wordt iedere positie met een 0 of een 1 te vullen en wel zodanig dat de 8 groepen van telkens 3 opeenvolgende posities de binaire representatie van de getallen 1 t/m 7 weergeven (geïnspireerd door een publicatie van N.G.de Bruijn, Proc. Kon. Akad. 49 (1946) 958; TH-rapport 75-WSK-06; zie ook CACM 18 (1975) 651).

Discussie

A priori weet men niet of deze opgave een oplossing heeft en zo ja, hoeveel. Een weinig efficiënte oplossingsmethode zou zijn om alle $2^8 = 256$ combinaties van nullen en enen te maken en die stuk voor stuk op de gewenste eigenschappen te onderzoeken.
Het is duidelijk dat als er een oplossing is, dat dan ergens drie nullen naast elkaar staan; nummer nu deze posities 0, 1 en 2. Evenzo moeten posities 3 en 7 een 1 bevatten, doch over de posities 4 t/m 6 is op het eerste gezicht weinig te zeggen (voor dit kleine probleem kan men door proberen nu wel de oplossingen vrij snel vinden, doch dat is niet het geval voor een probleem met bijvoorbeeld 64 posities, waarbij de getallen 0 t/m 63 gevormd moeten worden). Zijn er meer oplossingen dan wil men die lexicografisch geordend hebben, d.w.z. een oplossing met 00010..... moet eerder komen dan een oplossing met 00011..... .
Het feit dat men niet weet of er een oplossing is en dat bij meer oplossingen deze geordend opgeleverd moeten worden, leidt er natuurlijk toe om de oplossing door systematisch zoeken op te sporen.
Zij nu aangenomen dat op een bepaald moment (i-1) bits gevonden zijn, dan vormen de laatste 3 bits hiervan een getal, verder n te noemen. De eis van ordening betekent dat men als volgende bit eerst een 0 moet proberen en dan pas een 1. De eis dat alle getallen van 0 t/m 7 gevormd moeten worden, betekent dat de twee laatste bits van n met de nieuwe i^e bit niet een getal mogen vormen dat men al gemaakt heeft bij de vorming van (i-1) bits, m.a.w. m := (2*n) mod 8 en m+1 mogen niet al voorgekomen zijn. Het ligt dus voor de hand om voor de oplossing een boolean array nr [0:7] bij te houden en daarin achtereenvolgens "af te strepen" (op false zetten) welke getallen reeds gevormd zijn.
Als men een "goede" nieuwe bit gevonden heeft, dan moet men - tenzij nu 8 goede bits geconstateerd zijn - dezelfde methode opnieuw toepassen voor de $(i+1)^e$ bit en met een nieuw uitgangsgetal n. Dit suggereert een recursieve opbouwprocedure met zoiets als

```
procedure build (i,n);
                    - - - - - - - - - - - - - - - - - -
                    - - - - - - - - - - - - - - - - - -
    m := (2*n) mod 8;
    if nr [m] then begin bit[i] := 0;
                         nr[m] := false;
                         build (i+1, m)
                   end;
    if nr [m+1] then begin bit [i] := 1;
                           nr [m+1] := false;
                           build (i+1, m+1)
                     end
```

Wat alleen mis kan gaan met deze aanpak is dat zelfs al lukt het om op een correcte wijze van i naar i+1 bits te gaan, succes nog niet verzekerd is omdat dit uiteindelijk wel eens een "doodlopende" weg zou kunnen zijn. Bij het aflopen van deze doodlopende weg heeft men echter achter zich met nr[m] := false getallen "afgestreept". Dit moet men ongedaan maken, wat heel eenvoudig gaat door direct achter de recursieve aanroep build (i+1,*) te schrijven nr[*] := true.

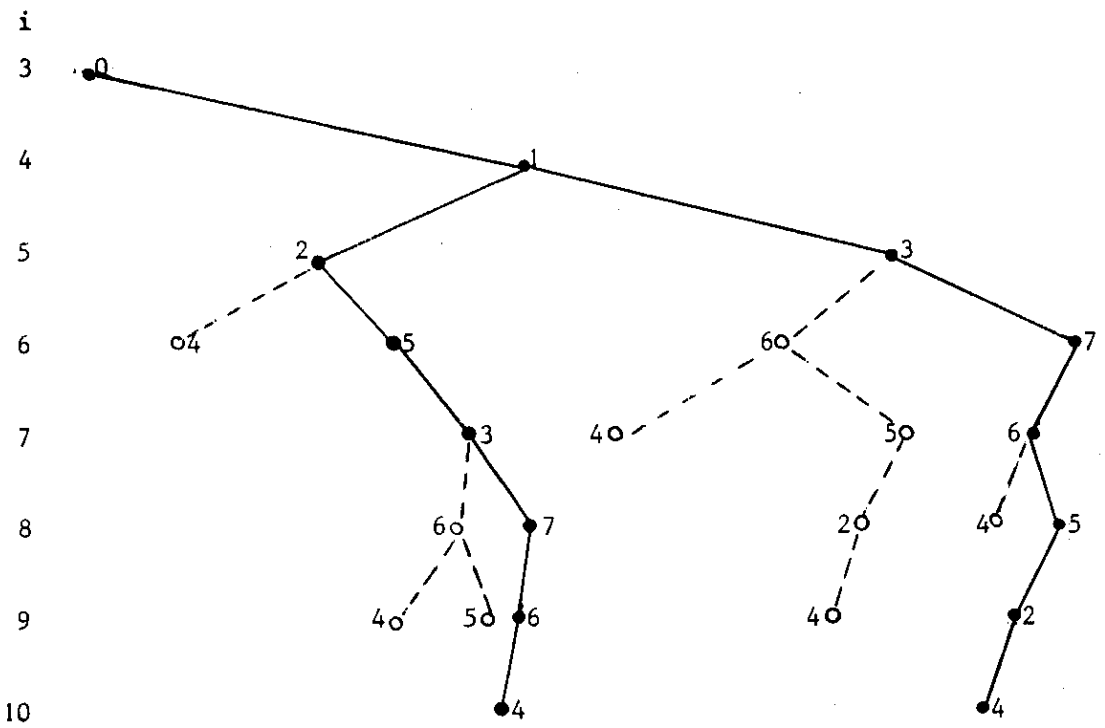
Om het programma nu een definitieve vorm te geven moet men nog een datastructuur voor de gezochte bits vinden en de recursie goed laten beginnen en eindigen. Voor de gezochte bits allicht een lineair array bit [0:9], met 10 (en niet met 8) elementen omdat men na het invullen van de 8^e bit de combinatie met de 0^e en 1^e bit (die men dus gedupliceerd denkt op de toegevoegde plaatsen) nog moet onderzoeken. Als men de eerste 3 plaatsen in bit met nullen vult, moet gehoopt worden dat het recursieve build proces de rest van bit array kan vullen zonder in het array nr een getal te dupliceren. Lukt dit, dan heeft men een oplossing gevonden en het proces zal met een nog niet afgehandelde if voortgezet worden. Men komt zo tot het bijgaande programma (merk op dat de conditie i < 8 or b = 0 samenhangt met de bijzondere behandeling van de laatste 2 plaatsen in het array bit).

Onder het programma is met behulp van tripels p, q, r (q en r corresponderen met parameters i en n van de procedure build, p is de waarde die bit [q], met q uit het vorige tripel, gekregen heeft) aangegeven wat het programma achtereenvolgens uitvoert. Grafisch is dit bovendien nog hieronder geschetst: getrokken zijn tussen de knopen de effectief doorlopen takken, gestippeld zijn de ingeslagen doodlopende wegen. Bij de knopen is ook aangegeven welke nr bij het bereiken van de betreffende knoop afgestreept is. Op te merken is dat door de gevolgde methode van de theoretisch mogelijke 254 takken in feite slechts 23 doorlopen zijn (waarvan 10 voor niets). Dit illustreert de kracht van het systematisch zoeken.

Opmerking.

Zoals o.a. de Bruijn aangetoond heeft, is voor een cirkel met 2^n posities het aantal verschillende oplossingen

$$2^{2^{n-1}} - n$$



```
begin integer i;
      integer array bit [0 : 9]; boolean array nr [0 : 7];
      procedure build (i,n); value i,n; integer i,n;
                begin if i < 10
                      then begin integer b, m;  for b := 0,1 do
                                 begin if i < 8 or b = 0 then
                                       begin m := (2 * n) mod 8 + b;
                                             if nr [m] then
                                                begin bit [i] := b;
                                                      nr [m] := false;
                                                      build (i + 1, m);
                                                      nr [m] := true
                                                end
                                       end
                                 end
                           end
                      else begin  integer j; nlcr;
                                 for j := 1 step 1 until 7 do write (bit [j])
                           end
                end;
      for i := 1 step 1 until 7 do nr [i] := true; nr [0] := false;
      for i := 0 step 1 until 2 do bit [i] := 0;
      build (3,0)
end
```