RA

stichting
mathematisch
centrum

MC

REKENAFDELING

MR 120/70 OKTOBER

RA

G.H.A. KOK

# ALFABETISEREN EN ANDERE SORTEERWERKZAAMHEDEN

2e boerhaavestraat 49 amsterdam

Printed at the Mathematical Centre, 49, 2e Boerhaavestraat, Amsterdam.

The Mathematical Centre, founded the 11-th of February 1946, is a non-profit institution aiming at the promotion of pure mathematics and its applications. It is sponsored by the Netherlands Government through the Netherlands Organization for the Advancement of Pure Research (Z.W.O), by the Municipality of Amsterdam, by the University of Amsterdam, by the Free University at Amsterdam, and by industries.

Inhoud

Inleiding

Als we over sorteren spreken denken we niet op de eerste plaats aan het sorteren van getallen, maar ook aan ingewikkelde orderelaties zoals alfabetiseren. Bij het praktisch uitvoeren van sorteerprocessen willen we de mogelijkheid openen om informatie "mee te nemen" die voor de orderelatie niet relevant is, tevens willen we zo min mogelijk eisen aan het te sorteren materiaal opleggen.

In sectie 1. staat de informatie die de gebruiker van de door ons ontwikkelde sorteerprogramma's nodig heeft. In sectie 2. staan enkele sorteermethoden en de theorie van andere onderdelen van de programma's. Het sorteerprogramma, *Omni-sort* - waar wij de voorkeur aan geven - staat in sectie 3. . In sectie 4. staat een programma om reeds gesorteerde verzamelingen met elkaar te mengen en een variatie op het programma uit 3. . In de programma's wordt wel eens enige kennis over het "MC-ALGOL 60-systeem voor de X8" verondersteld, voornamelijk over de functie van de standaardprocedures "RESYM", "PRINT", "PRINTTEXT" en de "flexowriter-code"; hiervoor verwijzen we naar het rapport MR 81.

De oppervlakkige gebruiker van ons sorteerprogramma kan volstaan met het lezen van sectie 1. . De secties 2. en 3. zijn bedoeld voor lezers die geïnteresseerd zijn in sorteren en voor programmeurs die onze programma's aan een ander ALGOL-systeem willen aanpassen. In sectie 4. staat een speciale toepassing: het efficiënt sorteren van verzamelingen die bestaan uit twee of meer deelverzamelingen die op zich al gesorteerd zijn.

## 1. Praktisch gedeelte.

In deze sectie geven we een beschrijving van onze sorteerprogramma's en enkele voorbeelden. Deze programma's zijn zeer algemeen, kenmerkend is:

1: Het is mogelijk om bij de informatie waarop gesorteerd wordt (de sorteenheid) een hoeveelheid extra informatie mee te nemen (de aanhangeenheid). De lengte van de eenheden is variabel.
2: Het programma is makkelijk aan bepaalde problemen aan te passen.
3: Er kan zonodig gebruik worden gemaakt van een achtergrondgeheugen.
4: Er wordt profijt getrokken van het feit dat in een bepaalde te sorteren hoeveelheid materiaal veel elementen gelijk kunnen zijn.
5: Het programma is vooral geschikt om te alfabetiseren.

We hebben voor dit doel drie programma's geschreven, te weten *Tree-sort*, *Omni-merge* en *Omni-sort*. *Tree-sort* voldoet aan 1, 2, 4 en 5, *Omni-merge* en *Omni-sort* voldoen aan alle vijf de eisen. De meeste namen van procedures in de drie programma's zijn gelijk. De aanpassing aan een bepaald probleem voor een van de drie programma's geeft onmiddellijk de aanpassing aan hetzelfde probleem voor de andere twee programma's. (Er is nog een variant van *Omni-merge*, *Omni-merge 2*, die speciaal gemaakt is voor het mengen van twee gesorteerde verzamelingen. zie 4.).
De verschillen tussen de programma's zitten in de sorteerstrategie. De strategie van *Omni-sort* is een mengvorm van die van *Tree-sort* en *Omni-merge*.

### 1.1. Handleiding sorteerprogramma's

*Tree-sort*, *Omni-merge* en *Omni-sort* bestaan alle drie uit een vast en een probleemafhankelijk deel.
Het constante gedeelte is een pakket procedures die uitmonden in de procedures *Voeg in* en *Lever het gesorteerde af*.

Het probleemafhankelijk deel bestaat uit de declaratie van de procedures *Uitvoer sort* en *Uitvoer aanhang*, een aanroep van de procedure *Initialiseer* uit het constante gedeelte en een stukje programma. In dat stukje programma wordt telkens één sort- en één - mogelijk lege - aanhangeenheid opgeslagen in de arrays *Sort* en *Aanhang*. Vervolgens wordt *Voeg in* aangeroepen, die de inhoud van deze arrays "ergens" opbergt. Daarna wordt *Lever het gesorteerde af* aangeroepen. Deze procedure vult - door middel van een cyclus - in de juiste volgorde de arrays *Sort* en *Aanhang* en roept *Uitvoer sort* en *Uitvoer aanhang* aan. In die procedures is door de programmeur van het probleemafhankelijke deel beschreven wat met de inhoud van *Sort* en *Aanhang* moet gebeuren.

Onderdeel van het constante gedeelte is een Boolean procedure, die staat voor de relatie *A opvolger van of dezelfde als B*. Deze procedure vergelijkt van twee integer arrays A en B de eerste, tweede, derde enz. elementen tot er twee corresponderende elementen zijn die verschillen.
*A opvolger van of dezelfde als B* krijgt dan de waarheidswaarde van de volzin: "Het eerste element van A dat verschilt van een corresponderend element van B is (groter,kleiner) dan dat element, of alle corresponderende elementen zijn gelijk en A heeft niet (minder,meer) elementen dan B". De programmeur van het probleemafhankelijke deel definieert door de manier, waarop hij het array *Sort* vult, de ordening van de verzameling. Of er in de bovenstaande volzin "groter" en "minder", of "kleiner" en "meer" staat wordt door hem beslist, d.m.v. de waarde die hij aan de Boolean *Klein voor Groot* geeft.
Als extra faciliteit is een comprimatiemechanisme in het constante gedeelte opgenomen. Dit dient om symboolrijen, die eventueel gealfabetiseerd moeten worden, compact op te slaan. Door een aanroep *Initialiseer voor compr* wordt een alfabet in een bepaalde volgorde, waardoor het alfabetiseren gedefinieerd is, ingelezen.
Een array, waarvan de elementen tot aan een bepaalde pointer gevuld zijn met de machinerepresentatie van de symbolen uit dat alfabet, kan nu gecomprimeerd worden; de symbolen worden in groepjes per array-element samengetrokken en de pointer wordt aangepast. Dit comprimeren gebeurt zo dat: comprimeren gevolgd door ordenen met *A opvolger van of dezelfde als B* gevolgd door décomprimeren overeenkomt met ordenen volgens de relatie *alfabetisch voor* in het gegeven alfabet.

De procedure *Voeg in* heeft als parameters twee integers die (in deze volgorde) staan voor het aantal gevulde elementen van *Sort* en *Aanhang*. Bij *Omni-merge* moet *Voeg in* vervangen worden door *Meng in*, bij *Omni-merge* door *Meng in 2*. *Meng in 2* heeft een derde parameter: een Boolean die *false* moet zijn bij het laatste element van iedere op zichzelf reeds gesorteerde verzameling, en voor alle andere elementen *true*.

Voor iedere verschillende sorteenheid wordt na het aanroepen van *Lever het gesorteerde af* de procedure *Uitvoer sort* eenmaal aangeroepen. Tijdens deze aanroep heeft de integer *Frekw* een waarde die gelijk is aan het aantal malen dat deze sorteenheid is voorgekomen. De integer *L sort* is gelijk aan het aantal gevulde elementen van *Sort*.
Voor iedere aanhangeenheid wordt *Uitvoer aanhang* aangeroepen. Tijdens *Uitvoer aanhang* is de inhoud van *Sort*, *L sort* en *Frekw* nog die van de bijbehorende aanroep van *Uitvoer sort*. *L aanhang* is het aantal gevulde elementen van *Aanhang*. De volgorde waarin de aanhang eenheden van een bepaalde sorteenheid aan bod komen is altijd de volgorde van inlezen.
Het comprimeren van de sort of de aanhangeenheden kan automatisch gebeuren door via de getallenband *Compr sort* of *Compr aanhang* (of beide) *true* te maken. Indien het comprimeren op deze wijze is gebeurd dan zorgt het constante gedeelte zelf voor het aanroepen van *Initialiseer voor compr* en *Decompr*. Soms is het nodig dat het comprimeren geheel vanuit het probleemafhankelijke deel gebeurt; dan moet de programmeur er voor zorgen dat ook *Initialiseer voor Compr* en *Decompr* wordt aangeroepen.
De procedures *Compr* en *Decompr* hebben beide als parameters het te comprimeren integer array en een integer die het aantal gevulde arrayelementen aangeeft.

Zolang op het MC het file-systeem niet in gebruik genomen is moet in ieder geval de voorkeur worden gegeven aan *Omni-sort*. *Omni-merge 2* is geschikt om snel twee op zich reeds gesorteerde verzamelingen te mengen.

Getallenbandje

Tussen haakjes staat steeds een waarde die op dit moment op het MC bruikbaar en/of beschikbaar is.

1: *L drum* (81 920); Het aantal op de drum gereserveerde plaatsen voor het file-systeem.

2: *L brok* (1024); De lengte van de 8 buffers voor het transport naar de drum (dit getal is optimaal).

3: *L array* (15 000); Opslagruimte in het kerngeheugen.(zo groot mogelijk).

4: *L sort* (afhankelijk van het probleem); Lengte van het array *Sort*. Het is verstandig om *L sort* ruim te kiezen.

5: *L aanhang* (idem); Lengte van het array *Aanhang*.

De volgende drie getallen dienen om een Boolean een waarde te geven. Deze Boolean wordt *true* als het getal 1 is en anders *false*.

6: Een getal voor de Boolean *Klein voor groot*. Als *Klein voor groot false* is dan wordt in de relatie *A opvolger van of dezelfde als B* de voorwaarde "Het eerste element van het array A dat verschilt van een corresponderend element van B is groter dan dat element, of alle corresponderende elementen zijn gelijk en A heeft niet minder elementen dan B", vervangen door "Het eerste element van het array A dat verschilt van een corresponderend element van B is kleiner dan dat element, of alle corresponderende elementen zijn gelijk en A heeft niet meer elementen dan B".

7: Een getal voor de Boolean *Compr sort*; Als *Compr sort true* is dan zorgt het systeem automatisch voor het aanroepen van *Initialiseer voor compr* en de sorteenheid wordt in het constante gedeelte gecomprimeerd en gedécomprimeerd.

8: Een getal voor de Boolean *Compr aanhang*; *Compr aanhang* heeft een zelfde effect als Compr sort, er wordt echter maar een keer geïnitialiseerd.

De volgende 3 punten alleen als er gecomprimeerd wordt.

9: *g* (67 108 863); Het grootste getal dat in een integer voorgesteld kan worden.

10: *L alfabet*; Het aantal symbolen van het alfabet onmiddellijk gevolgd door een overgang op een nieuwe regel. (Minimaal 2)

11: De symbolen van dat alfabet. Hiervoor zijn alle symbolen die met RESYM (MR 81) kunnen worden ingelezen toegestaan.

Samenvatting:

De programmeur van het probleemafhankelijke deel dient dus te zorgen voor:

De declaratie van de procedures *Uitvoer sort* en *Uitvoer aanhang*,

Een stukje programma waarin een of meerdere malen de procedure *Voeg in* of bij *Omni-merge* *Meng in* wordt aangeroepen; dit stukje programma moet beginnen met de aanroep *Initialiseer*.

Een aanroep *Lever het gesorteerde af*.

Vijf extra maal "end".

Het getallenbandje.

## 1.2. Vier voorbeelden

### 1.2.1. Alfabetiseren

We hebben een simpel stukje programma geschreven om te alfabetiseren. *Uitvoer sort* drukt een - gealfabetiseerd - woord af, *Uitvoer aanhang* is dummy. *Deel van woord* kijkt of een symbool een letter of een apostrof is (alle andere symbolen zijn woordscheiders!). Indien een ingelezen symbool het eindsymbool (de "bar") is, wordt *Lever het gesorteerde af* aangeroepen. Voor een methode om een koppelteken gevolgd door een overgang op nieuwe regel in een afgebroken woord te onderdrukken zie 1.2.4..

```
integer i, sym, eindsymbool;

procedure Uitvoer sort;
begin for i:= 1 step 1 until L sort do PRSYM(Sort[i]); NLCR end;

procedure Uitvoer aanhang; ;

Initialiseer;
begin

  Boolean procedure deel van woord(sym); value sym; integer sym;
  deel van woord:= sym > 9 ∧ sym < 63 ∨ sym = 120;

  i:= 0; eindsymbool:= 127;
L: sym:= RESYM; if deel van woord(sym) then goto M else
  begin if sym = eindsymbool then goto K else goto L end;
N: sym:= RESYM; if deel van woord(sym) then
  begin Voeg in(i, 0); i:= 0;
    if sym = eindsymbool then goto K else goto L
  end;
M: i:= i + 1; if sym > 36 ∧ sym < 63 then sym:= sym - 27;
  Sort[i]:= sym; goto N;
K: Lever het gesorteerde af
end
```

1.2.2. Retrograad alfabetiseren.

Zij M een verzameling woorden, dan verstaan we onder retrograad alfabetiseren het volgende: Keer alle woorden om (iedere $W = a_1 \ldots a_n$ wordt $a_n \ldots a_1$), alfabetiseer de verzameling M, en keer nogmaals alle woorden om. Een retrograad woordenboek zou dus kunnen beginnen met "logica" en eindigen met "jazz".
Een extra moeilijkheid bij de verzameling woorden die wij retrograad willen alfabetiseren is dat de symbolen "-", "." en "<" in de woorden voorkomen ("<" als vervanger van de spatie). Toch willen we dat woorden als "scherts-formatie" en "schertsformatie" bij elkaar staan.

Het probleemafhankelijke deel ziet er als volgt uit:

```
integer M, K, punt, koppelteken, kleiner dan, spatie,
eindsymbool, i, l, k, sym;

procedure inverteer(A, l);value l;integer l;integer array A;
begin integer i, a, m; m:=l : 2; for i:=1 step 1 until m do
      begin a:=A[l+1-i]; A[l+1-i]:=A[i]; A[i]:=a end
end inverteer;

Boolean procedure Deel van woord(sym);value sym;integer sym;
Deel van woord:=sym > 9 ∧ sym < 36 ∨ sym=120;

procedure Uitvoer sort; inverteer(Sort, L sort);

procedure Uitvoer aanhang;
begin integer n, sym, stop, i;

      procedure stopsym; if n < L aanhang then
      begin stop:=Aanhang[n] : 1000; sym:=Aanhang[n] -
            stop × 1000; n:=n + 1
      end else stop:=0;
      K:=K + 1; if K=1 then
      begin NEW PAGE; if EVEN(M)=-1 then
            PRINT(M : 2 + 1); M:=M + 1; NLCR
      end;
      n:=1; stopsym;
      for i:=1 step 1 until L sort do
      begin L: if i=stop then
            begin PRSYM(sym); stopsym; goto L end;
            PRSYM(Sort[i])
      end;
      if stop=L sort + 1 then PRSYM(sym); NLCR;
      if K=50 then K:=0
end Uitvoer aanhang;

Initialiseer; punt:=88; koppelteken:=65; kleiner dan:=72;
spatie:=93; eindsymbool:=127;M:=1; K:=0;
L: l:=k:=Aanhang[1]:=0;
S: sym:=RESYM;
if sym=punt ∨ sym=koppelteken ∨ sym=kleiner dan then
begin k:=k + 1; if sym=kleiner dan then sym:=spatie;
      Aanhang[k]:=(1 + 1) × 1000 + sym; goto S
end;
if Deel van woord(sym) then
begin l:=l + 1; Sort[l]:=sym; goto S end else
T: if l > 0 ∧ Sort[l] < 17 then
begin inverteer(Sort, l); Voeg in(l, if k=0 then 1 else k) end;
if sym≠eindsymbool then goto L; Lever het gesorteerde af
```

De Boolean procedure *Deel van woord (sym)* kijkt of *sym* een letter of een apostrof is. De procedure *inverteer (A,l)* verwisselt *A[1]* met *A[l]*, *A[2]* met *A[l-1]* enz.. De sorteenheid wordt opgebouwd uit symbolen, waarvoor *Deel van woord true* is. Bij ieder voorkomen van ".", "-" of "<" voor het $n$-de symbool van de sorteenheid, krijgt een element van de aanhangeenheid de waarde $n$ maal 1000 plus de code van ".", "-" of de spatie. Komen deze symbolen niet in het woord voor dan *L aanhang:= 1* en *Aanhang [1]:= 0*.
Het comprimatiemechanisme is door het inlezen van twee getallen in een zodanige stand gezet, dat het systeem automatisch *Initialiseer voor comprimatie* aanroept en ervoor zorgt dat de sorteenheid gecomprimeerd en gedecomprimeerd wordt. We kunnen dus zonder meer *Voeg in* aanroepen. We gaan door met het invoegen van woorden, totdat we een keer het symbool "|", waarvan de code 127 is, tegenkomen. *Lever het gesorteerde af* roept voor iedere sorteenheid *Uitvoer sort* aan en voor iedere aanhangeenheid *Uitvoer aanhang*. *Uitvoer sort* inverteert het array *sort* weer.

In *Uitvoer aanhang* worden de punten, spaties en koppeltekens en hun plaats in het woord uit de codering teruggevonden. Van het feit, dat de inhoud van het array *Sort* sinds de laatste aanroep van *Uitvoer sort* nog niet veranderd is en het feit dat iedere sorteenheid minstens één aanhangeenheid heeft, maken we dan gebruik door de sorteenheid met tussenvlechting van punten, koppeltekens en spaties in *Uitvoer aanhang* af te drukken. In deze procedure wordt een indeling in genummerde pagina's verzorgd.

1.2.3.

Sorteren van gehele getallen die groter dan de integercapaciteit mogen zijn.

Het probleem hierbij is: hoe we de sorteenheid moeten vullen om de gewenste ordening te krijgen.
Als $a_1, a_2, \ldots, a_l$ de cijfers van het getal
$\pm\, a_1 a_2 \ldots a_l$ zijn dan is een goede sorteenheid kennelijk
$\pm\, l, \pm\, a_1, \pm\, a_2, \ldots, \pm\, a_l$.

Daar we ook van het comprimatiemechanisme gebruik willen maken is de volgorde van de bewerkingen:

1: $k$ wordt 1 als er geen minteken is, anders -1.

2: $a_1,\ldots,a_l$ worden in *Sort* gezet.

3: $k$ wordt $k \times l$.

4: *Sort* wordt gecomprimeerd.

5: Alle elementen van *Sort* schuiven een plaats op en worden negatief als $k$ negatief is.

6: *Sort* $[1]$ wordt $k$.

Bij het afleveren van de gesorteerde sorteenheden doorlopen deze in *Uitvoer sort* een soort omgekeerd proces en worden afgedrukt.

```
integer i, j, l, k, sym, plusteken, minteken, eindsymbool,
spatie;

procedure Uitvoer sort;
begin k:= Sort[1];
  for i:= 2 step 1 until L sort do Sort[i - 1]:= if k > 0 then
  Sort[i] else - Sort[i]; L sort:= L sort - 1; Decompr(Sort, L sort);
  for j:= 1 step 1 until Frekw do
  begin NLCR; if k > 0 then PRSYM(plusteken) else PRSYM(minteken);
    for i:= 1 step 1 until L sort do
    begin PRSYM(Sort[i]);
      if (L sort - i) = ((L sort - i) : 3) × 3 then PRSYM(spatie)
    end
  end
end;

procedure Uitvoer aanhang; ;

Initialiseer; Initialiseer voor compr; plusteken:= 64; minteken:= 65;
eindsymbool:= 127; spatie:= 93;
L: l:= 0; k:= 1;
M: sym:= RESYM; if sym = spatie then sym:= RESYM;
if (sym = plusteken ∨ sym = minteken) ∧ l = 0 then
begin if sym = minteken then k:= - 1; goto M end;
if sym < 10 then
begin l:= l + 1; Sort[l]:= sym; goto M end
else if l > 0 then
begin if l = 1 ∧ Sort[1] = 0 then k:= 1; k:= k × l; Compr(Sort, l);
  for i:= 1 step - 1 until 1 do Sort[i + 1]:= if k > 0 then Sort[i]
  else - Sort[i]; Sort[1]:= k; l:= l + 1; Voeg in(l, 0)
end;
if sym≠eindsymbool then goto L;
Lever het gesorteerde af
```

1.2.4. Het maken van een alfabetische frekwentielijst.

Een alfabetische frekwentielijst van een bepaalde tekst is een lijst, waarvan de woorden gesorteerd zijn op frekwentie én op alfabet. De frekwentie is iets dat we bijvoorbeeld door alfabetiseren kunnen vinden (Iedere keer dat *Uitvoer sort* wordt aangeroepen heeft *Frekw* de waarde van de frekwentie). Omdat we er in het sorteersysteem voor zorgen dat aanhangeenheden van sorteenheden in volgorde van binnenkomst blijven staan, kunnen we zo'n lijst maken door eerst te alfabetiseren en vervolgens te sorteren op frekwentie met de letters van het woord als aanhangeenheid.

De procedures *Uitvoer sort* en *Uitvoer aanhang* bestaan ieder uit twee delen, een voor de uitvoer bij het alfabetiseren en een voor de uitvoer bij het sorteren op frekwentie. De Boolean *alfabetiseren* zegt welk deel genomen moet worden.
*Uitvoer sort* schrijft *Frekw*, gevolgd door *L sort*, gevolgd door de elementen van *Sort* naar *file 4*. (Bij de eerste aanroep van *Uitvoer sort* weten we zeker dat *file 2* en *file 4* niet meer gebruikt worden. Zie voor ons file-systeem 2.2.3.) Als alles naar *file 4* geschreven is, wordt *alfabetiseren false* en krijgt *m* de waarde *Av* (aantal verschillende sorteenheden). Vervolgens wordt vanuit *file 4* telkens een sorteenheid gevuld met de frekwentie en een aanhangeenheid gevuld met het gecomprimeerde woord. *Uitvoer sort* en *Uitvoer aanhang* zorgen in de stand *alfabetiseren* voor het afdrukken van de lijst.

De integer *sym na koppelteken* gebruiken we om een koppelteken gevolgd door een overgang op een nieuwe regel tijdens het inlezen van de tekst over te slaan, tevens worden hoofdletters vervangen door kleine letters.

```
Boolean alfabetiseren;
integer i,j,l,m,twnr,koppelteken,eindsymbool,sym,sym na koppelteken;

procedure Uitvoer sort;
if alfabetiseren then
begin outbin(4,Frekw);outbin(4,L sort);
      for i:=1 step 1 until L sort do outbin(4, Sort[i])
end else begin NLCR;PRINT(Sort[1]);NLCR end;

procedure Uitvoer aanhang;
if alfabetiseren then           else
begin NLCR; Decompr(Aanhang, L aanhang);
      for i:=1 step 1 until L aanhang do PRSYM(Aanhang[i])
end;

Boolean procedure woorddeel(sym);value sym;integer sym;
woorddeel:=sym>9 ∧ sym<36 ∨ sym=120;

Initialiseer; Initialiseer voor compr; alfabetiseren:=true;
twnr:=119; koppelteken:=65; eindsymbool:= 127;

L: l:=0;sym na koppelteken:=-1;
M: sym:=if sym na koppelteken <0 then RESYM else sym na koppelteken;
if woorddeel(sym - 27) then sym:=sym - 27;
if sym=koppelteken then
begin sym na koppelteken:=RESYM;if sym na koppelteken=twnr then
      begin sym na koppelteken:=-1; goto M end
end else sym na koppelteken:=-1;
if woorddeel(sym) then
begin l:=l+1; Sort[l]:=sym;goto M end
else  if l>0 then
begin Compr(Sort, l); Voeg in(1,0) end;
if sym≠eindsymbool then goto L;
Lever het gesorteerde af;m:=Av; Initialiseer;
alfabetiseren:= Klein voor groot:=false;
for j:=1 step 1 until m do
begin Sort[1]:=inbin(4); l:=inbin(4);
      for i:=1 step 1 until l do Aanhang[i]:=inbin(4); Voeg in(1,l)
end; Lever het gesorteerde af
```

1.3. De verwerking van het programma.

Dit proces kunnen we in vier fasen verdelen:

1. De initialisering.
2. De eerste sorteerfase. (Bij *Omni-merge* bestaat deze fase niet.)
   Tijdens deze fase worden de sort- en aanhangeenheden ingelezen, eventueel gecomprimeerd en in het inwendige geheugen opgeslagen. Dat opslaan gaat volgens de methode van een "binaire boom" (zie 2.2.2.), waardoor alle in het geheugen aanwezige eenheden steeds de gewenste volgorde hebben. Als het voor de boom gereserveerde array vol dreigt te raken worden de eenheden in juiste volgorde naar het achtergrondgeheugen (de drum) geschreven en wordt er weer één nieuwe boom opgebouwd. Het grote voordeel van zo'n binaire boom is dat sorteenheden die meer dan eenmaal voorkomen maar een keer onthouden hoeven te worden.
3. De tweede sorteerfase.
   In deze fase worden al deze gesorteerde stukken met elkaar gemengd totdat alle eenheden in de juiste volgorde staan.
4. De uitvoerfase.
   De eenheden worden naar het inwendige geheugen teruggeschreven, eventueel gedécomprimeerd en *Uitvoer sort* en *Uitvoer aanhang* worden aangeroepen.

Tijdens deze vier fasen wordt af en toe wat informatie over de gang van zaken afgedrukt. Indien er iets mis dreigt te gaan wordt een foutmelding gegeven en de verwerking van het programma gestopt.

1.3.1. Sorteergegevens en foutmeldingen.

Tijdens de initialisering wordt afgedrukt:
(*X* staat voor een getal, *W* voor een waarheidswaarde *true* of *false*, *S* voor een symbool).

| | | | |
|---|---|---|---|
| *L drum en L brok* | *X* | *X* | |
| *L array, L sort en L aanhang* | *X* | *X* | *X* |
| *Klein voor groot, Compr sort, Compr aanhang* | *W* | *W* | *W.* |

En indien er gecomprimeerd wordt:

*g (de integercapaciteit) en L alfabet* X X
$S_1 S_2 \cdots S_L$ *alfabet.*

Als foutmelding kan optreden:

1a. *L alfabet is te klein,*

hetgeen zal gebeuren als we daarvoor een getal kleiner dan 2 hebben genomen.

Tijdens de eerste sorteerfase kunnen de volgende foutmeldingen optreden:

2a. *er is niet geïnitialiseerd,*
2b. *er is niet geïnitialiseerd voor comprimatie,*
2c. *lengte sorteenheid is niet positief,*
2d. *lengte aanhangeenheid is negatief,*
2e. *X*
*bovenstaand getal is geen RESYM-code en kan dus niet gecomprimeerd worden,*
2f. *S*
*bovenstaand symbool wordt ter comprimatie aangeboden maar staat niet in het alfabet,*
2g. *De gesimuleerde files zijn vol.*

Het laatste wil zeggen dat de geheugenruimte op de drum onvoldoende is. Na afloop van deze fase wordt afgedrukt:

*totale aantal sort- en aanhangeenheden* X X
*aantal bomen en aantal vrije geheugen-* X X
*plaatsen op de drum.*

Tijdens de tweede sorteerfase wordt geen informatie verstrekt.

Na afloop van de uitvoerfase wordt de volgende informatie gegeven:

*totale aantal sort- en aanhangeenheden* X X
*aantal comparities* X
*aantal verschillende sorteenheden* X.

Een comparitie is een vergelijking tussen twee sorteenheden.

Indien de aantallen sort- en aanhangeenheden tijdens de tweede en vierde fase verschillen (hetgeen zou betekenen dat er in het sorteerprogramma een fout zit!) wordt daarvoor extra gewaarschuwd door het afdrukken van:

*tijdens het sorteren zijn eenheden verdwenen;*

hier controleert het programma dus zichzelf.

1.3.2. Tijdsduur en geheugenruimte.

Een belangrijke invloed op de totale tijdsduur is de tijd die nodig is voor de comparities. Het aantal comparities is o.a. afhankelijk van het totale aantal sorteenheden, het aantal verschillende sorteenheden, de verdeling van de sorteenheden over de verschillende en de volgorde waarin deze aangeboden worden. Dit zijn allemaal zaken die i.h.a. niet van te voren bekend zijn. Een bruikbare schatting is: $2 \times n \times ln(k)$, waarin $n$ het totale aantal en $k$ het aantal verschillende is. Als $n >> k$ (b.v. $n = 20k$) dan kunnen we $1.5 \times n \times ln(k)$ nemen. De tijdsduur van een comparitie is in de orde van 2 milli sec.
Andere belangrijke factoren die de tijdsduur van het proces beïnvloeden zijn: de tijd nodig voor het inlezen, eventueel comprimeren en afdrukken van de eenheden, de tijd voor het plaatsen van de verschillende sort- en van alle aanhangeenheden in de boom, de tijd die nodig is voor het transport vanuit de boom naar de drum.

We geven enkele door ons gevonden resultaten:

| Totale aantal | Aantal verschillende | Soort eenheden | Aantal comparities | Tijd in sec. |
|---|---|---|---|---|
| 64 052 | 3 949 | woorden | 589 440 | 2200 |
| 4 117 | 729 | woorden | 34 487 | 130 |
| 1 517 | 1 429 | woorden | 16 210 | 110 |
| 2 000 | 2 000 | getallen | 24 286 | 181 |
| 800 | 51 | getallen | 4 282 | 12 |

Indien alle sorteenheden verschillend zijn, is het mogelijk de benodigde geheugenruimte op de drum te schatten door de som van de lengte van alle eenheden te schatten en daar 3 maal het aantal sorteenheden en het aantal aanhangeenheden bij op te tellen. (Indien er gecomprimeerd wordt, dan kan men de lengte van één eenheid berekenen m.b.v. het getal in de tweede kolom van de tabel op pagina 26).
Indien niet alle sorteenheden gelijk zijn, dan wordt de benodigde geheugenruimte kleiner (nooit kleiner dan de ruimte die nodig is om de verschillende sort- en alle aanhangeenheden op te slaan). In verband hiermee is het gunstig om *L array* zo groot mogelijk te kiezen.

Het is mogelijk een zeer grove schatting te maken van de benodigde tijd en geheugenruimte, door een gedeelte (zeg p%) te draaien. De schatting is beter naarmate meer aan de volgende voorwaarden is voldaan:
1. De steekproef is representatief voor het hele materiaal.
2. De omvang van de steekproef is zo groot dat precies éénmaal het inwendige geheugen vol is geweest.

Stel dat het totale aantal sorteenheden van de steekproef $n$ is en het aantal verschillende $k$. We nemen nu aan dat:
1. De verhouding $k/n$ voor alle stukken van het materiaal, die van gelijke omvang als de steekproef zijn, gelijk is.
2. Het verschil van de totale tijd en de tijd nodig voor de comparities lineair oploopt met de omvang van de steekproef.
3. De benodigde geheugenruimte op de drum lineair oploopt met de omvang van de steekproef.

Dan schatten we de totale tijd nodig voor het hele materiaal als

$$\frac{100}{p} \times \textit{verschiltijd} + 2 \times \frac{100}{p} \times n \times \ln\left(\frac{100}{p}\, k\right) \times 2 \times 10^{-3} \textit{ sec.}$$

In deze schatting maken we gebruik van de tijdsduur voor een comparitie $2 * 10^{-3}$ sec. Deze tijdsduur geldt voor de X8, op een andere machine zal die waarde waarschijnlijk anders zijn.
(Met verschiltijd bedoelen we de tijd die in 2. genoemd is : voor de berekening daarvan maken we weer gebruik van de - empirisch bepaalde - tijd voor een comparitie; $2 \times 10^{-3}$ sec.).

En we schatten de benodigde geheugenruimte op de drum als

$\frac{100}{p} \times$ *(L drum - Vrije geheugenplaatsen + 2 × L brok)* plaatsen.

Indien hieruit zou blijken dat de geheugenruimte onvoldoende is, dan kunnen we het materiaal in meerdere malen draaien. Als ons materiaal uit woorden die gealfabetiseerd moeten worden bestaat zouden we bijvoorbeeld eerst de woorden, die beginnen met de letters $a \ldots m$, en later de woorden, die met $n \ldots z$ beginnen, kunnen draaien.

## 2. Theoretisch gedeelte.

### 2.1. Alfabetiseren als orderelatie. Comprimeren.

In deze paragraaf wordt een algemene definitie gegeven van alfabetiseren en worden enige methoden om te comprimeren aangegeven.

definitie 2.1.1. Onder een ordening van een verzameling A verstaat men de relatie $\leq$ in A met de volgende eigenschappen:

1. $a \leq b$ of $b \leq a$.
2. $a \leq b$ en $b \leq c \Rightarrow a \leq c$. (transitiviteit)
3. $a \leq a$ voor alle $a$ in $A$. (reflexiviteit)
4. $a \leq b$ en $b \leq a \Rightarrow a = b$. (antisymmetrie)

Als voor alle $a$, $b$ in $A$ geldt $a \leq b$ óf $b \leq a$ dan noemen we $A$ een geordende verzameling.

Als $a \leq b$ en $a \neq b$ dan schrijven we $a < b$.

definitie 2.1.2. Laten $X$ en $Y$ geordende verzamelingen zijn. Een functie $f: X \to Y$ heet monotoon indien $f(a) \leq f(b)$ voor alle $a, b \in X$ met $a \leq b$. De functie $f$ heet isomorfie (orde-isomorfie) als $f$ een bijectie is en zowel $f$ als $f^{-1}$ monotoon zijn. $X$ en $Y$ heten dan isomorf, notatie: $X \simeq Y$.

Stelling 2.1.3. Laten $X$, $Y$ en $Z$ geordende verzamelingen zijn.

$X \simeq Y,\ Y \simeq Z \Rightarrow X \simeq Z$

bewijs: De compositie van de bijectieve afbeelding tussen $X$ en $Y$ en de bijectieve afbeelding tussen $Y$ en $Z$ is een bijectieve afbeelding tussen $X$ en $Z$, waarbij de monotonie behouden blijft.

stelling 2.1.4. Een geordende verzameling $V$ met $n$ elementen kan geschreven worden als $a_1, a_2, \ldots, a_n$ met $a_i < a_j$ als $i$ kleiner is dan $j$. ($n$ is het aantal elementen).

bewijs: volledige inductie naar $n$.

1. Voor $n = 1$ is de stelling triviaal.
2. Neem het gestelde aan voor $n = p$. Zij $p+1$ het aantal elementen uit $V$. Neem een willekeurig element uit $V$ en noem dat $a$. De (geordende) verzameling $V-\{a\}$ die uit $p$ elementen bestaat kan ik volgens de inductie-aanname schrijven als

$$a_1, a_2, \ldots, a_p \text{ met } a_i < a_j \text{ als } i \text{ kleiner is dan } j.$$

Vergelijk $a$ achtereenvolgens $a_1, \ldots, a_p$.
Er zijn twee mogelijkheden:

Ik vind een element $a_i$ met $a < a_i$ óf
$a_i < a$ voor $i = 1, \ldots, p$.

In het eerste geval vernummer ik $a_i, \ldots, a_p$ tot $a_{i+1}, \ldots, a_{p+1}$ en geef $a$ het nummer $i$. In het tweede geval geef ik $a$ het nummer $p+1$.

definitie 2.1.5. Een alfabet is een eindige niet lege verzameling symbolen.

definitie 2.1.6. Een geordend alfabet is een eindige niet lege geordende verzameling symbolen.

Volgens stelling 2.1.4. kunnen wij een geordend alfabet van $n$ symbolen schrijven als $a_1, a_2, \ldots, a_n$. Aan de andere kant kunnen wij een rij symbolen $a_1, a_2, \ldots, a_n$ op voor de hand liggende wijze opvatten als een geordende verzameling en dus als een geordend alfabet. In het vervolg zullen wij onder een alfabet altijd een geordend alfabet verstaan.

definitie 2.1.7. Een woordverzameling over een gegeven alfabet is de verzameling eindige niet lege rijtjes symbolen uit dat alfabet.

Tenzij anders vermeld is in het vervolg A steeds een alfabet met symbolen $a_1, a_2, \ldots, a_n$, $M$ is de woordverzameling over $A$. De woorden van $M$ worden aangegeven als $W_l$, $W_k$ enz.. $M_c$ is de deelverzameling van $M$ die bestaat uit woorden met een lengte die kleiner dan of gelijk aan $c$ is.

Zij $W_l = a_{l_1} \ldots a_{l_p}$. We noemen $p$ de lengte van $W_l$. Bij $W_l$ denken we ons een oneindige rij getallen die bestaat uit $l_1, l_2, \ldots, l_p$ gevolgd door oneindig veel nullen. Deze rij $\{l_i\}_{i=1}^{\infty}$ noemen we de met $W_l$ geassocieerde rij.

definitie 2.1.8. We definieëren een relatie $\leq$ (alfabetisch voor) in $M$ als volgt: Zij $W_l = a_{l_1} \ldots a_{l_p}$ en $W_k = a_{k_1} \ldots a_{k_q}$ beide in $M$. De geassocieerde rij van $W_l$ is $\{l_i\}_{i=1}^{\infty}$ en van $W_k$ $\{k_i\}_{i=1}^{\infty}$.

$W_l < W_k$ wordt gedefinieerd als: er is een getal $t$ met $l_r = k_r$ als $r < t$ en $l_t < k_t$, en $W_l \leq W_k$ als $W_l < W_k$ óf $W_l = W_k$.

stelling 2.1.9. Zij $M$ de woordverzameling op een alfabet $a_1, a_2, \ldots, a_n$ en $\leq$ de in (2.1.8.) gedefinieerde relatie, dan is $M$ een geordende verzameling.

bewijs: Zij $W_l = a_{l_1} \ldots a_{l_p}$, $W_l \in M$. We definiëren een functie $S_n: M \to \mathbb{Q}$ (rationale getallen) als volgt:

$$S_n(W_l) = \frac{l_1}{(n+1)} + \frac{l_2}{(n+1)^2} + \ldots + \frac{l_p}{(n+1)^p}$$

Uit (2.1.8) volgt onmiddellijk: $W_l \leq W_k$ d.e.s.d.a.

$$S_n(W_l) \leq S_n(W_k).$$

De transitiviteit, reflexiviteit en anti-symmetrie gelden dan voor $M$ omdat ze voor $\mathbb{Q}$ gelden.

Op grond van stelling 2.1.9 en stelling 2.1.4 kan een eindige verzameling woorden gealfabetiseerd worden.

stelling 2.1.10. Als twee geordende alfabetten hetzelfde aantal symbolen hebben dan zijn hun woordverzamelingen isomorf.

bewijs: Zij $A$ een alfabet dat bestaat uit de symbolen $a_1,\ldots,a_n$ met woordverzameling $M$, en $A'$ een ander alfabet met symbolen $a_1',\ldots,a_n'$ en woordverzameling $M'$. Als bijectieve afbeelding $f$ nemen we de afbeelding die van een woord $W_l$ uit $M$ alle symbolen $a_i$ vervangt door corresponderende symbolen $a_i'$ uit $A'$.

De ordening blijft behouden daar $W_l$ en $f(W_l)$ dezelfde geassocieerde rij hebben.

stelling 2.1.11. Als $A$ een alfabet is dat uit minstens twee symbolen bestaat en $M$ is de woordverzameling over $A$, dan is $M$ niet isomorf met een deelverzameling van de gehele getallen.

bewijs: Stel dat $M$ wel isomorf met een deelverzameling der gehele getallen is en $f$ is de bijectieve afbeelding. Als $a_i$ en $a_j$, $j > i$, twee symbolen uit $A$ zijn dan bevinden zich tussen de woorden $a_i$ en $a_j$ oneindig veel woorden uit $M$, bijvoorbeeld: $a_ia_i, a_ia_ia_i, a_ia_ia_ia_i, \ldots$ . Tussen $f(a_i)$ en $f(a_j)$ bevinden zich hoogstens $f(a_j) - f(a_i) - 1$ elementen uit $M$.

definitie 2.1.12. Als $A$ een alfabet is met symbolen $a_1,a_2,\ldots,a_n$ en woordverzameling $M$, dan definiëren we $A^c$ als het alfabet dat bestaat uit de symbolen $\alpha_1,\alpha_2,\ldots,\alpha_k$, $k = n^c + n^{c-1} + \ldots + n$, waarbij $\alpha_1,\alpha_2,\ldots,\alpha_k$ alle woorden uit $M_c$ (dat is de deelverzameling van $M$ met woorden ter lengte kleiner of gelijk $c$) zijn en $\alpha_i < \alpha_j$ als $i < j$. $M^c$ is de woordverzameling over $A^c$.
(Het aantal woorden uit $M_c$ is $n^c + n^{c-1} + \ldots + n$ omdat $M_c$ bevat: $n$ woorden ter lengte 1, $n^2$ woorden ter lengte 2, ... en $n^c$ woorden ter lengte $c$.).

stelling 2.1.13. $M$ is isomorf met $M^c$.

bewijs: Zij $W_l \in M$. Voor het gemak zullen we de symbolen van $W_l$ niet indiceren met $l_1,\ldots,l_p$ maar met $l_{1.1},\ldots,l_{1.c},l_{2.1},\ldots,l_{2.c},\ldots,l_{u.1},\ldots,l_{u.v}$. (De lengte van $W_l$ is dus $(u-1)c + v$, $v \leq c$)

Als bijectieve afbeelding nemen we $f: M \to M^c$ die uit $W_l$ van links naar rechts steeds een groepje van $c$ symbolen opvat als één symbool uit $A^c$. Tenslotte zijn er dan nog $v$ symbolen over die tesamen het laatste symbool van $f(W_l)$ vormen. $f^{-1}$ is de afbeelding die een woord uit $M^c$ weer "uiteen laat vallen" in symbolen uit $A$. Zij $W_l = a_{l_{1.1}}\ldots a_{l_{1.c}} a_{l_{2.1}}\ldots a_{l_{u.v}}$ en $W_k = a_{k_{1.1}}\ldots a_{k_{1.c}} a_{k_{2.1}}\ldots a_{k_{d.e}}$. Om te bewijzen dat $f$ en $f^{-1}$ monotoon zijn is het voldoende aan te tonen dat de volgende beweringen equivalent zijn.

1. Er is een getal $t$ met $l_{i.j} = k_{i.j}$ als $(i-1) \times c + j < t$ en $l_{r.w} < k_{r.w}$ waarbij $t = (r-1)c + w$.
2. Er is een getal $s$ met $L_b = K_b$ als $b < s$ en $L_s < K_s$ .
   ($\{l_{i.j}\}$, $\{k_{i.j}\}$, $\{L_i\}$ en $\{K_i\}$ zijn de geassocieerde rijen van resp. $W_l$, $W_k$, $f(W_l)$ en $f(W_k)$)

Uit de definitie van $f$ volgt $r = s$. De equivalentie tussen $l_{i.j} = k_{i.j}$ als $(i-1) \times c + j < t$, en $L_b = K_b$ als $b < s$ is triviaal. Uit $l_{r.i} = k_{r.i}$ voor $i < w$ en $l_{r.w} < k_{r.w}$ volgt dat het r-de symbool van $f(W_l)$ alfabetisch voor of gelijk aan het $r$-de symbool van $f(W_k)$ is en dus $L_r < K_r$. Omgekeerd volgt uit $L_r < K_r$ dat $L_{r.i} = k_{r.i}$ voor $i$ kleiner dan een getal $w$ en $l_{r.w} < k_{r.w}$ .

stelling 2.1.14. Zij $L_c$ de verzameling, waarvan de elementen de $c$-tallen getallen $(l_1,\ldots,l_c)$ met de volgende eigenschappen zijn:

1. $l_1 > 0$
2. als $l_j = 0$ dan $l_{j+1} = 0$ $(1 \leq j < c)$
3. het maximum van $l_1,\ldots,l_c$ is een gegeven getal $n$.
   ($L_c$ is de verzameling van de eerste $c$ getallen van de met woorden uit $M_c$ geassocieerde rijen. $M_c$ is deelverzameling van $M$ over een alfabet van $n$ symbolen. $L_c$ is op voor de hand liggende wijze isomorf met $M_c$).

Zij verder $\sigma^c(l^1,\ldots,l_c)$ een functie van $L_c$ naar de natuurlijke getallen, zodanig dat:

$$\sigma^c(l_1,\ldots,l_c) < \sigma^c(l'_1,\ldots,l'_c) \Leftrightarrow \text{er is een getal } t,$$
$$1 \leq t \leq c, \text{ met } l_r = l'_r \text{ als } r < t \text{ en } l_t < l'_t.$$

Als $\sigma^c$ zo een afbeelding is, dan is de woordverzameling $K^c$ over de naar opklimmende grootte gerangschikte rij getallen $\sigma^c(l_1,\ldots,l_c)$, opgevat als symbolen van een alfabet, isomorf met de woordeverzameling $M$ over een alfabet van $n$ symbolen.

bewijs: $L_c$ heeft $n^c + n^{c-1} + \ldots + n$ elementen er zijn dus $n^c + n^{c-1}+\ldots+n$ symbolen in het alfabet bij $K^c$. Daar $A^c$ ook $n^c + n^{c-1} + \ldots + n$ symbolen heeft is $M^c$ op grond van stelling 2.1.10.. isomorf met $K^c$. Aangezien $M^c$ wegens (2.1.13) isomorf is met $M$ is $K^c$ isomorf met $M$ (stelling 2.1.13).
Opmerking: De stelling geldt ook als het alfabet bij $K^c$ bestaat uit iedere andere permutatie van de getallen $\sigma^c(l_1,\ldots,l_k)$. Om praktische redenen zullen wij deze getallen naar grootte gerangschikt nemen.

Het afbeelden van $M$ via $M^c$ naar $K^c$ zullen wij comprimeren noemen en de reciproke afbeelding décomprimeren.

Om met een rekenmachine te kunnen alfabetiseren moeten we woorden afbeelden op getallen. Omdat afbeelding met behoud van orderelatie van woorden naar gehele getallen niet mogelijk is (2.1.11) en afbeelding naar rationale getallen het gevaar van afronding door de machine inhoudt, zullen we woorden afbeelden op rijtjes gehele getallen. Comprimeren is zo'n afbeelding naar rijtjes getallen - hoewel niet de eenvoudigste - die als prettige eigenschap heeft dat de relatie < voor getallen overeenkomt met de relatie < voor woorden, die in één machinewoord onthouden worden.

De betekenis van comprimeren voor het met een rekenmachine alfabetiseren is:

1. We kunnen $c$ zo groot kiezen dat $\sigma^c(n,n,\ldots,n)$ nog net in een machinewoord onthouden kan worden, hierdoor besparen we geheugenruimte.
2. Het rijtje getallen dat een woord voorstelt is korter dan de woordlengte in $M$ (als $c > 1$), daardoor hoeven we bij het vergelijken van twee van die rijtjes, minder getallen met elkaar te vergelijken.

Voorbeelden van de afbeelding $\sigma^c$.

Opbouwen van een getal in een $(n+1)$-tallig stelsel;

$$\sigma^c(l_1,l_2,\ldots,l_c) = l_1(n+1)^{c-1} + l_2(n+1)^{c-2}+\ldots+l_c.$$

Het is onmiddellijk duidelijk dat deze afbeelding voldoet aan de voorwaarde dat

A: $\sigma^c(l_1,\ldots,l_c) < \sigma^c(l_1',\ldots,l_c') \Leftrightarrow$ er is een $t$, $1 \leq t \leq c$, met $l_r = l_r'$ als $r < t$ en $l_t < l_t'$.

De grootste waarde van $\sigma^c$ is dan $n(n+1)^{c-1} + n(n+1)^{c-2}+\ldots+n = (n+1)^c - 1$, hetgeen meer is dan het aantal elementen uit $L_c$ $n^c + n^{c-1}+\ldots+n$. Deze keuze voor $\sigma^c$ of een afbeelding in een $k$-tallig stelsel waarbij $k$ een tweemacht groter dan $n+1$ is wordt, voor zover mij bekend is altijd bij alfabetiseren gebruikt.

Voor de hand liggend is om voor $\sigma^c$ een aftelling te nemen van $L_c$ zodanig dat steeds aan A voldaan is. De grootste waarde van $\sigma^c$ is dan de kleinst mogelijke, namelijk: $n^c + n^{c-1}+\ldots+ n$; het aantal elementen uit $L_c$. We zullen deze afbeelding de alfabetische aftelling van $M_c$ noemen.

stelling 2.1.15. De alfabetische aftelling van $M_c$ is de functie

$$\sigma^c(l_1,\ldots,l_c) = U_l + \sum_{j=1}^{U_l} (l_j-1)(n^{c-j}+\ldots+n+1),$$

hierin is $U_l$ het kleinste getal $p$, $0 \leq p \leq c-1$, waarvoor geldt $l_{p+1} = 0$ óf als zo'n $p$ niet bestaat $c$.

bewijs: Zij $I = (l_1, l_2, \ldots, l_c)$ een $c$-tal uit $L_c$. Vóór $(l_1, 0, \ldots, 0)$ bevinden zich de $(l_1-1)$ $c$-tallen $(1,0,\ldots,0), (2,0,\ldots,0) \ldots (l_1-1,0,\ldots,0)$ en alle andere $c$-tallen van de vorm $(i, k_2, \ldots, k_c)$ met $i < l_1$. Tussen $(i,0,\ldots,0)$ en $(i+1,0,\ldots,0)$ bevinden zich de $c$-tallen van de vorm $(i, k_2, k_3, \ldots, k_c)$ (niet $k_2 = k_3 \ldots = k_c = 0$) en dat zijn er precies $n^{c-1} + n^{c-2} + \ldots + n$. Het nummer van $(l_1, 0, \ldots, 0)$ is dus $l_1 + (l_1-1)(n^{c-1}+n^{c-1}+n^{c-2}+\ldots+n) = (l_1-1)(n^{c-1}+\ldots + n+1) + 1$.
Op soortgelijke wijze vinden we dat het nummer van $(l_1, l_2, 0, \ldots, 0)$ gelijk is aan $(l_1-1)(n^{c-1}+\ldots+n+1) + (l_2-1)(n^{c-1}+\ldots+n+1)$. Stellen we $p = U_l$ dan mogen we $I$ schrijven als $(l_1, l_2, \ldots, l_p, 0, \ldots, 0)$ en vinden voor het nummer van $I$

$$p + \sum_{j=1}^{p} (l_j-1)(n^{c-j}+\ldots+n+1).$$

Voor $\sigma^c(n,\ldots,n)$ kunnen we bij de alfabetische aftelling als benadering nemen $n^c+n^{c-1}$ en bij het $(n+1)$-tallig stelsel $n^c + cn^{c-1}$. Het quotiënt is dus van de orde $\frac{n+c}{n+1}$. De maximale waarde die we voor $c$ kunnen nemen is ongeveer evenredig met het aantal bits van een machinewoord. Kennelijk wordt de alfabetische aftelling aantrekkelijker naarmate het machinewoord langer is.

We besluiten deze paragraaf met een tabel, waarin de alfabetische aftelling vergeleken wordt met het $(n+1)$-tallig stelsel, voor een rekenmachine met als grootste voorstelbare integer *67 108 863*.

| Aantal symbolen van het alfabet | Aantal symbolen dat per machine-woord onthouden kan worden als de integercapaciteit *67 108 863* is. | |
|---|---|---|
| | Alfabetische aftelling | *(n+1)*-tallig stelsel |
| 2 | 25 | 16 |
| 3 | 16 | 13 |
| 4 | 12 | 11 |
| 5 | 11 | 10 |
| 6 | 9 | 9 |
| 7 | 9 | 8 |
| 8 | 8 | 8 |
| 9 | 8 | 7 |
| 10 | 7 | 7 |
| 11 | 7 | 7 |
| 12 | 7 | 7 |
| 13 | 6 | 6 |
| ⋮ | ⋮ | ⋮ |
| 19 | 6 | 6 |
| 20 | 5 | 5 |
| ⋮ | ⋮ | ⋮ |
| 35 | 5 | 5 |
| 36 | 5 | 4 |
| 37 | 4 | 4 |
| ⋮ | ⋮ | ⋮ |
| 89 | 4 | 4 |
| 90 | 4 | 3 |
| 91 | 3 | 3 |
| ⋮ | ⋮ | ⋮ |
| 128 | 3 | 3 |

## 2.2 Sorteerstrategie.

### 2.2.1 Twee bekende voor getallen gebruikte sorteermethoden.

In 2.1 hebben we laten zien dat we ons belangrijkste probleem -het alfabetiseren- niet kunnen oplossen door afbeelding van ieder woord op een getal, gevolgd door sorteren van die getallen met de rekenmachine. Toch zijn de sorteermethoden die voor getallen gebruikt worden van belang voor het alfabetiseren. We kunnen immers de strategie van zo'n methode gebruiken, terwijl we de relatie > vervangen door de relatie *A opvolger van of dezelfde als B* zoals we die in (1.1) voor rijtjes getallen beschreven hebben.

De eerste methode die we zullen bekijken is de algorithme *Sort-merge*.

Het principe is:
Sorteer een rij getallen door eerst alle opvolgende getallen in de goede volgorde te zetten. Meng vervolgens alle opvolgende paren tot rijtjes van 4 getallen die in goede volgorde staan. Herhaal het proces van het mengen van twee rijtjes opvolgende getallen net zo lang, totdat we alle getallen in goede volgorde hebben staan.
Een bezwaar van *Sort-merge* is: dat we om de rij, die we krijgen bij het mengen van twee deelrijen, te onthouden, een array nodig hebben, waarvan de lengte van de orde van de lengte van de hele rij moet zijn. Een ander bezwaar is het grote aantal verplaatsingen van de getallen. Een methode die deze nadelen niet heeft is de algorithme *Quicksort* van Hoare (ACM 4(1961) 321).

Het principe daarvan is:
Breng het probleem een rij getallen te sorteren terug tot twee problemen, namelijk het sorteren van twee rijen getallen, die samen dezelfde lengte hebben als de oorspronkelijke rij. Dit doen we door een getal uit die rij te kiezen, de eerste deelrij bevat dan alle getallen die groter dan dat getal zijn, de tweede deelrij alle getallen die kleiner dan of gelijk aan dat getal zijn.

Uiteindelijk komt het probleem dan neer op het sorteren van rijen die uit één of twee getallen bestaan en dat geeft geen moeilijkheden. *Quicksort* en *Sort-merge* zijn beide snelle sorteermethoden en vooral *Sort-merge* is eenvoudig te programmeren. Toch zijn hun strategieën voor de problemen die wij op het oog hebben - sorteenheden met aanhang-eenheden, waarbij mogelijk veel sorteenheden identiek zijn - niet erg geschikt.
Ten eerste worden in beide methoden, maar in grotere mate bij *Sort-merge*, elementen zelf van plaats verwisseld. Dit kan, daar de sorteenheden vaak uit meerdere en misschien zelfs uit heel veel array-elementen bestaan, een tijdrovende zaak zijn. Dit nadeel is op te vangen door de echte elementen op te slaan in een array en de verwisseling uit te voeren in een ander array, waarvan de getallen verwijzen naar het eerste array.
Ten tweede zullen *Quicksort* en *Sort-merge* over een rij getallen waaronder veel dezelfde voorkomen ongeveer even lang doen als over een even lange rij verschillende getallen.
In 2.2.2. en 2.2.4. zullen we methoden bespreken die wél profijt trekken van een eventueel groot aantal identieke sorteenheden.

2.2.2. Sorteren met behulp van een binaire boom, de strategie van *Tree-sort*.

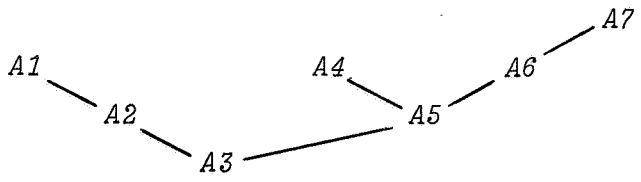
De oudste referentie hierover die ik heb kunnen vinden is naar A.D. Booth (Information and control 3, 327-334 (1960)).
We beginnen met de grammatica van boom.

*P1:* < *boom* > ::= (<*linkertak*><*atoom*><*rechtertak*>)
*P2:* < *linkertak* > ::= <*empty*>|<*boom*>←
*P3:* < *rechtertak* > ::= <*empty*>|→<*boom*>

Als *A1,A2,...,A7* atomen zijn, dan is

$$(((A1) \leftarrow A2) \leftarrow A3 \rightarrow ((A4) \leftarrow A5 \rightarrow (A6 \rightarrow (A7))))$$

een voorbeeld van een boom. Een overzichtelijke manier om een boom te noteren is:




Dit plaatje geeft ons ook een verklaring voor de naam *boom*.

We definiëren het begrip deelboom.

We noemen *boom1* deelboom van *boom2* als er twee - mogelijk lege - symboolrijen *S1* en *S2* bestaan zodat *S1 boom1 S2* dezelfde symbolenrij is als *boom2*. Het begrip grootste deelboom zal geen moeilijkheden geven. Onder een totale boom zullen we een boom, die geen deelboom van een andere boom dan zichzelf is, verstaan.
We gaan de deelbomen en de atomen van een totale boom nummeren in een tweetallig stelsel. De totale boom zelf geven we nummer *1*. Deze boom kunnen we op eenduidige wijze lezen als *(<linkertak><atoom><rechtertak>)*. Het atoom dat in deze produktieregel voorkomt geven we ook nummer *1*.
Zij nu gegeven een deelboom met nummer $i$. Ook deze deelboom kunnen we wegens produktieregel *P1* lezen als *(<linkertak><atoom><rechtertak>)*. Dat atoom geven we weer nummer $i$. Als *linkertak* een deelboom bevat (*linkertak* kan leeg zijn) dan geven we de grootste deelboom van *linkertak* het nummer $2\times i$. Op dezelfde wijze geven we de grootste deelboom van *rechtertak* - indien deze deelboom bestaat - het nummer $2\times i+1$. *Linker-* en *rechtertak* van deelboom $i$ geven we zelf ook het nummer i (Als een bepaalde deelboom het nummer $j$ heeft, dan heeft deze boom opgevat als tak van een andere boom het nummer $j+2$.).
We gaan een boom gebruiken om te sorteren. Neem aan dat we een geordende verzameling $A$ hebben met orde-relatie $<$. Voor *<atoom>* kiezen we elementen van $A$.
De boom wordt zo opgebouwd dat - als $a \in A$ en *deelboom1* en *deelboom2* zijn deelbomen - we $a \rightarrow$ *deelboom1* kunnen interpreteren als "Voor iedere $x \in A$ en $x$ is atoom in *deelboom1* geldt $a < x$" , en *deelboom2* $\leftarrow a$ als "Voor iedere $x \in A$ en $x$ is atoom in *deelboom2* geldt $x < a$" .

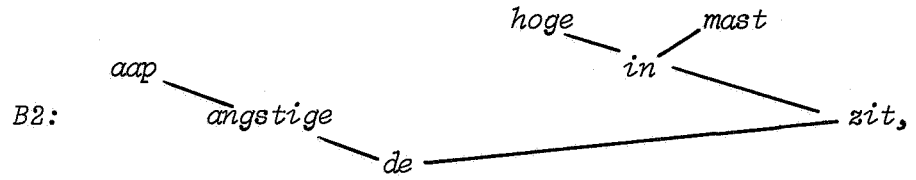
Zij nu $a_1, \ldots, a_n$ een serie trekkingen met teruglegging uit $A$.
We geven een "ALGOL-achtige" beschrijving van het produktieproces van de boom, dat bestaat uit het successief aanbieden van $a_1, a_2, \ldots, a_n$ aan de algorithme *Voeg in* $(a_i)$.
Met *atoomi*, *linkertaki* enz. bedoelen we het atoom of de tak met het boven gedefinieerde nummer $i$.

```
procedure  Voeg in (a_i);
begin
SP1:  if er is nog niets then begin boom := '(a_i)'; goto K end;
      j:= 1;
  L:  if a_i = atoom j then goto K;
      if a_i > atoom j then
      begin
SP2:        if rechtertak j = empty then
            begin rechtertak j:= '→(a_i)'; goto K end;
            j:= 2×j + 1; goto L
      end else
      begin
SP3:        if linkertak j = empty then
            begin linkertak j:= '(a_i)←'; goto K end;
            j:= 2×j; goto L
      end;
  K:
end  Voeg in
```

Als voorbeeld nemen we voor $A$ de verzameling van Nederlandse woorden met als orde-relatie *alfabetisch voor*. Trekking hieruit is de tekst: *de angstige aap zit in de hoge mast.*
De boom die we tijdens het sorteerproces opbouwen wordt.

*B1:* $(((aap) \leftarrow angstige) \leftarrow de \rightarrow (((hoge) \leftarrow in \rightarrow (mast)) \leftarrow zit))$,

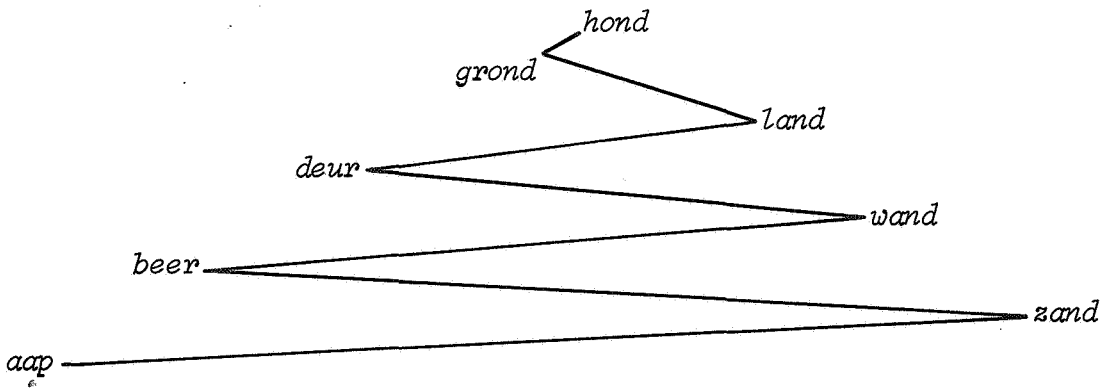
of in "twee-dimensionale" notatie:




We geven een "ALGOL-achtige" beschrijving van een proces waarbij de atomen in gesorteerde volgorde worden afgeleverd.

```
procedure rol op de boom(i); value i; integer i;
begin    if linkertak i ≠ empty then rol op de boom (2×i);
         lever atoom i af;
         if rechtertak i ≠ empty then rol op de boom (2×i+1)
end
```

Een aanroep *rol op de boom (1)* zal alle atomen in gesorteerde volgorde afleveren.
Het bovenstaande is eenvoudig in ALGOL te implementeren. We reserveren voor de boom een array, een atoom bezet een aantal plaatsen in dat array en we bevestigen linker- en rechtertakken aan een atoom door middel van verwijzingen naar andere atomen.
De procedure *rol op de boom(i)* levert praktische problemen. Met een paar wijzigingen kunnen we de bovenstaande beschrijving in de ALGOL-tekst overnemen. Het recursieve karakter van de procedure kan echter moeilijkheden veroorzaken bij bomen van een "pathologische" structuur als:
*(aap → ((beer → ((deur → ((grond →(hond))← land)) ← wand)) ← zand))*,
of in andere notatie:



Als een boom een dergelijke structuur heeft dan is de recursie van de orde van het aantal atomen. Dat betekent dat we kostbare geheugenruimte moeten reserveren ten behoeve van de recursie in dit soort extreme gevallen. Het schrijven van de procedure in een niet recursieve vorm levert geen oplossing omdat we bij het aflopen van de boom toch allerlei verwijzingen zullen moeten onthouden.

Een simpele oplossing verkrijgen we door een wijziging in het karakter van boom. We gaan de rechtertakken die "empty" zijn, gebruiken voor verwijzingen naar een atoom van een lager niveau.
We gaan weer even terug naar ons oude probleem.

*B1* had de gedaante
*(((aap) ← angstige) ← de → (((hoge) ← in → (mast)) ← zit))*.

De atomen staan in alfabetische volgorde van links naar rechts. We kunnen → en ← opvatten als een manier om van een atoom naar een atoom op een hoger niveau te komen. We kunnen dus makkelijk van *de* bij *aap* komen, maar een weg terug hebben we niet. Eigenlijk zouden we iets willen als:

*(((aap) ← angstige) ← de → (((hoge) ← in → (mast)) ← zit))*.

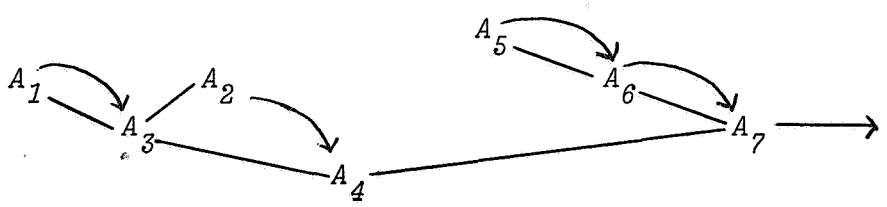
Dit gaan we realiseren. De nieuwe grammatica voor boom wordt:

*P1:* < *boom* > ::= (<*linkertak*><*atoom*><*rechtertak*>)
*P2:* < *linkertak* > ::= <*empty*> | <*boom*>←
*P3:* < *rechtertak* > ::= => | →<*boom*>

Zijn *A1, ..., A7* atomen dan is een voorbeeld van boom

$(((A_1 =>) \leftarrow A_3 \rightarrow (A_2 =>) \leftarrow A_4 \rightarrow (((A_5 =>) \leftarrow A_6 =>) \leftarrow A_7 =>))$

of in "twee-dimensionale" notatie



Het lijkt juist hier iets te zeggen over de wijze waarop de twee-dimensionale notatie uit de lineaire verkregen wordt. Haakjes worden vervangen door de atomen die in de lineaire notatie tussen haakjes staan een regel hoger te schrijven. Verder vervangen we → en ← door strepen naar het laagst geschreven atoom rechts resp. links. => vervangen we door een - gebogen - pijl naar het eerste atoom rechts indien dat aanwezig is, anders door een horizontale pijl naar rechts.

We zullen => interpreteren als "het atoom dat zo dicht mogelijk links van => staat is in de ordening de onmiddellijke voorganger van het atoom dat zo dicht mogelijk rechts staat". Als zo'n atoom rechts ontbreekt dan is het linkse atoom het laatste van de verzameling.

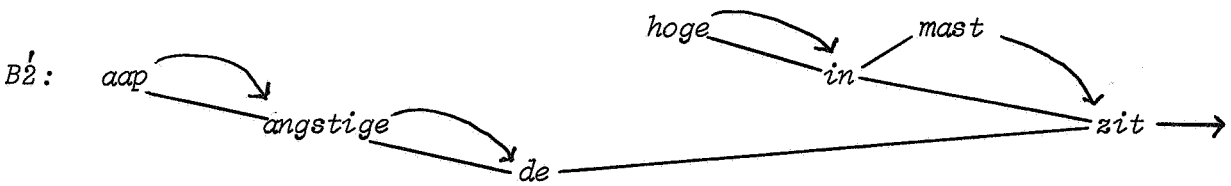
In *Voeg in* ($a_i$) zullen we *SP2* vervangen door:

```
SP2:    if rechtertak j = '=>' then
        begin rechtertak j := '→(a_i =>)'; goto K end;
```

Wij implementeren => als de voorgeschreven verwijzing naar een lager niveau, of in het geval van *A7* als een teken dat we klaar zijn. We kunnen → en => in de implementatie bijvoorbeeld onderscheiden doordat de verwijzing → positief onthouden wordt en de verwijzing ⇒ negatief.

*B1* en *B2* worden in de nieuwe notatie:

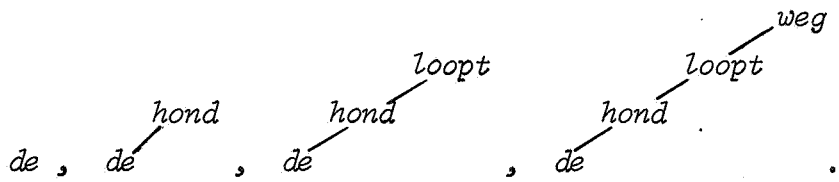
*B1'*: *(((aap =>) ← angstige =>) ← de → (((hoge =>) ← in → (mast =>)) ← zit =>)).*




We geven het analogon van *rol op de boom(i)*. Met *i := '=> i'* zullen we bedoelen "i krijgt het nummer van het atoom waar => naar verwijst".

```
   i:= 1;
L: if linkertak i ≠ empty then begin i:= 2*i; goto L end;
M: if er is een atoom i then lever atoom i af else zijn we klaar;
   if rechtertak i ≠ '=>' then begin i:= 2*i+1; goto L end
   else
   begin i:= '=> i' ; goto M end;
```

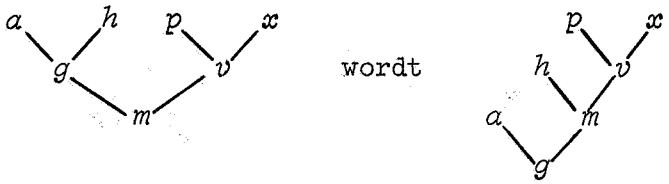
Het begrip atoom in het voorafgaande is niet hetzelfde als een sorteenheid. Een atoom is een sorteenheid plus eventueel een ketting van aanhangeenheden. Ook kunnen we in het atoom een teller voor de frekwentie van de sorteenheid bij houden.
We hebben al eerder opgemerkt dat de sorteermethode met een boom veel lijkt op *Quicksort*; we kunnen beide methoden op elkaar afbeelden. Een bekende moeilijkheid bij *Quicksort* was dat het aantal comparities bij het sorteren van een reeds geordende verzameling van n elementen ongeveer $\frac{1}{2}\ n^2$ is, terwijl het gemiddelde over alle permutaties $2\times n\times ln(n)$ is (zie bijvoorbeeld M.H. van Emden, Informatie jaargang 11 nr.1). Datzelfde probleem doet zich bij de *boom-strategie* ook voor. Stel we willen alfabetiseren de woorden van de zin: *de hond loopt weg*. De boom ontwikkelt zich dan als:




Bij een dergelijke reeds gealfabetiseerde verzameling - en bij alle andere waarbij in de boom geen vertakkingen optreden - is het aantal comparities, dat nodig is voor het plaatsen van het $k$-de atoom, $k-1$.
Bij een boom die zo regelmatig mogelijk vertakt is - dat is een boom, waarvan ieder atoom steeds de mediaan is van alle atomen uit de kleinste deelboom waar het zelf toe behoort - is dat ongeveer $log_2(k-1)$. Voor een verzameling van $n$ elementen zijn dan in het eerste geval
$1 + 2 + \ldots + n-1 = \frac{1}{2}\ n(n-1) \approx \frac{1}{2}\ n^2$ comparities nodig,
en in het tweede geval
$log_2\ 1 + log_2\ 2 + \ldots\ log_2(n-1) = log_2(n-1)! \approx 1.44\times ln(n-1)!$
dat is, als $n$ groot genoeg is, ongeveer gelijk aan $1.44\times n\times ln(n)$. Het gemiddelde over alle permutaties is evenals bij *Quicksort* $2\times n\times ln(n)$.

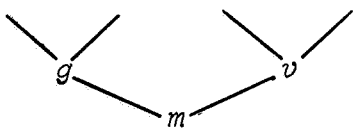
De veronderstelling dat alle permutaties ook in de praktijk bij het sorteren even vaak zullen voorkomen lijkt twijfelachtig en de konsekwenties zouden rampzalig kunnen zijn.
Stel iemand wil 4000 verschillende woorden alfabetiseren. Daarvoor zijn zo'n 80000 comparities nodig. Nu weten wij dat met ons programma op de X8 van het MC een comparitie ongeveer 2 millisec duurt. Het hele alfabetiseren zal dan 160 sec voor de comparities en b.v. 200 sec voor andere werkzaamheden zoals het inlezen en het afdrukken - werkzaamheden waarvoor de tijdsduur lineair met het aantal woorden oploopt - duren.
Daarna ontdekt hij dat hij 40 woorden vergeten heeft en hij alfabetiseert de geordende verzameling plus die 40 woorden.
Het aantal comparities ligt dan in de orde van 8 000 000 en het programma heeft er ongeveer $4\frac{1}{2}$ uur voor nodig.
Aangezien dit soort voorvallen niet zo erg onwaarschijnlijk zijn lijkt het verstandig een beveiliging hiertegen in te bouwen.

Door een aantal verwijzingen te verwisselen is het mogelijk de structuur van een boom te veranderen zonder dat de ordening verloren gaat. *)
b.v.:




Hierbij zakken de atomen $a$ en $g$ ten koste van de $m$, $v$, $p$ en $x$.

Stel we hebben een deelboom waarvan de onderste drie atomen zijn:

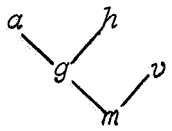



Als de deelboom bij $g$ veel groter is dan die bij $v$, dan is het verstandig om $g$ te laten zakken en de plaats van $m$ te laten innemen.

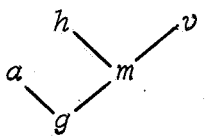
*) Het idee de structuur van de boom tijdens het sorteren te veranderen is afkomstig van M.H. van Emden die voorstelt alle hoogfrekwente atomen "naar onder in de boom te laten zakken".

Dat de deelboom bij $g$ veel groter is dan bij $v$ blijkt b.v. uit het feit dat bij het plaatsen van nieuwe atomen in de boom, deze in veel meer gevallen met $g$ vergeleken zijn dan met $v$. Als we tijdens het opbouw-proces van de boom het aantal comparities tellen en iedere keer als dat aantal een vijfvoud is, we het volgende atoom waarmee het nieuwe atoom dan vergeleken gaat worden een plaats laten zakken, dan is de kans groot dat $g$ na enige tijd de plaats van $m$ inneemt. Natuurlijk zal zo'n verplaatsing niet altijd gunstig zijn, maar gunstige verplaatsingen hebben een grotere waarschijnlijkheid dan ongunstige, en niet vertakte bomen worden wel heel onwaarschijnlijk. Omdat het aantal comparities zo veel groter is, treden in een "slechte" boom veel meer verplaatsingen op dan in een "goede". In plaats van om de vijf comparities, wat een volkomen willekeurige keuze is, laten we om de $2\times ln(k)$ comparities een atoom zakken ($k$ is het aantal verschillende atomen in de boom). In een redelijk vertakte boom gebeurt dan voor ieder nieuw atoom gemiddeld een verplaatsing.

Om een verplaatsing te kunnen uitvoeren zijn nog enkele bijzondere maatregelen nodig zoals het invoeren van $\Longleftarrow$ (volkomen analoog aan $\Longrightarrow$), het is ook zaak er voor te waken dat atomen niet naar zichzelf gaan verwijzen. Een verdere verfijning hebben we aangebracht door er voor te zorgen dat een nieuw atoom nooit door een verplaatsing van een ouder atoom tweemaal met dat atoom vergeleken wordt. Dat laatste gebeurt bijvoorbeeld als we het atoom "$i$" in de volgende boom willen plaatsen




We vergelijken $i$ met $m$ op grond daarvan beslissen we dat we links af moeten en laten dan, vanwege het aantal comparities dat we in het verleden gedaan hebben, het atoom $g$ zakken.



Vervolgens vergelijken we $i$ met $g$ waarna we weten dat we rechtsaf naar $m$ moeten. Om te vermijden dat we een comparitie uitvoeren waarvan we de uitslag al weten, tussen $i$ en $m$, springen we meteen naar $h$.

We hebben deze methode getest op drie soorten verzamelingen getallen. Ten eerste op random getallen, ten tweede op een verzameling getallen in opklimmende volgorde, en ten derde op een verzameling getallen waarvan eerst het kleinste getal wordt gegeven dan het grootste, dan het een na kleinste, het een na grootste etc. . Bij de random getallen had deze methode geen noemenswaardige invloed op het aantal comparities, bij de gesorteerde verzameling bleek het aantal comparities nu ook ongeveer $2 \times n \times ln(n)$ te zijn, en bij de zig-zag verzameling ongeveer $6 \times n \times ln(n)$. De tijd die nodig is voor het uitvoeren van de verplaatsingen is te verwaarlozen bij de tijd nodig voor het hele proces.
Het is verstandig om de hele boom met alles wat eraan vast zit in één array te onthouden. Indien we meerdere arrays gebruiken lopen we het risico dat een array vol is, terwijl in een ander array nog ruimte over is.
We komen in moeilijkheden als dat ene array volraakt. Dat probleem zullen we oplossen door de atomen van de boom als deze vol is in goede volgorde "ergens" te onthouden en een nieuwe boom op te bouwen. Raakt deze boom weer vol dan onthouden we ook de atomen daarvan enz.. Tenslotte mengen we alle gesorteerde deelverzamelingen tot we de oorspronkelijke verzameling in goede volgorde hebben.

2.2.3. Het simuleren van sequential access files.

De sequential access files die wij zullen gebruiken kunnen we ons voorstellen als een rij geheugenplaatsen die aan een zijde bereikbaar is voor het vernietigend lezen en aan de andere zijde voor het schrijven (toevoegen van nieuwe elementen). De lengte van zo'n file is dus variabel. Ons file-systeem is een vereenvoudigde versie van het door J.V.M. van der Grinten, P.J.W. ten Hagen en F.E.J. Kruseman Aretz in NR 7 beschreven systeem. Alle identifiers hebben we gelijk gekozen, zodat we onmiddellijk op dat systeem kunnen overschakelen, zodra dat op het Mathematisch Centrum in gebruik genomen is.
We hebben voor de simulatie de beschikking over de procedure *TO DRUM (A,p)* die de inhoud van het array *A* op de drum opbergt en over *FROM DRUM (A,p)* die *A* weer vult vanaf de drum. In beide gevallen geeft *p* de plaats op de drum van waar af de elementen van *A* komen te staan resp. gestaan hebben. Tijdens het transport wordt doorgerekend, zolang althans *A* niet gebruikt wordt.
We zullen 4 files simuleren. De drum wordt verdeeld in *M* brokken ter lengte *L brok*. Een array *ketenboek*[*1:M*] verwijst naar die brokken. *Keten0* verwijst naar alle vrije brokken (de vrije lijst), *keten 1 t/m 4* verwijst naar de brokken die voor de corresponderende file gereserveerd zijn. De procedure *Geef schakel van (i) aan: (j)* haalt van *keten i* de eerste schakel af en hangt die achter *keten j*. Bij iedere file horen twee arrays ter lengte *L brok*, waarvan beurtelings de een werkarray is en de ander voor transport gebruikt wordt. De procedure *Sla (brok) op voor: (i)* geeft een schakel van de vrije lijst aan *keten i* en slaat de inhoud van het array *brok* op de met die schakel corresponderende plaats op de drum op. De procedure *Geef (brok) vrij voor: (i)* vult het array *brok* vanaf de drum en geeft de corresponderende schakel weer aan de vrije lijst. *outbin(i,kar)* schrijft de integer *kar* in het werk-array van de file i. Als dat vol is wordt het m.b.v. *Sla (brok) op voor: (i)* naar de drum geschreven en gaat het andere array als werkarray dienen. De integer procedure *inbin(i)* krijgt bij iedere aanroep de waarde van een integer die dan aan de beurt is om van de file gelezen te worden.

Voor transport gebruiken we de aanroep *Geef (brok) vrij voor: (i)*. Aangezien de werk- en transportarrays van *inbin* dezelfde arrays zijn als die van *outbin*, is het tijdrovend om die procedures voor dezelfde file snel afwisselend aan te roepen. Lezen gebeurt vernietigend, lezen mag alleen in dezelfde file gevolgd worden door schrijven als de hele inhoud van die file "opgelezen" is.

*outbin* en *inbin* zijn het gezicht van het file-systeem, het zijn de enige procedures hieruit, die we in de tekst van een sorteerprogramma zullen gebruiken.

2.2.4. Sorteren met sequential access files, de strategie van *Omni-merge*.

We hebben een sorteermethode ontworpen die werkt met files. Het principe is nauw verwant aan dat van *Sort-merge*. We geven een beschrijving van de algorithme voor het naar opklimmende grootte rangschikken van de rij getallen $A[1],\ldots,A[n]$.

1: Schrijf de getallen met een oneven index naar file 1 en die met een even index naar file 2.

2: Maak hieruit rijtjes van twee getallen door telkens van file 1 en file 2 een getal lezen en die getallen in goede volgorde naar een andere file te schrijven. Die andere file is voor het eerste, derde, vijfde, ... rijtje file 3 en voor het tweede, vierde, zesde ... rijtje file 4.

3: Maak uit de rijtjes van file 3 en file 4 rijtjes van 4 elementen die naar file 1 en file 3 geschreven worden.

4: Ga zo door tot we op file 1 of file 3 één rij hebben die uit alle getallen bestaat.

In deze vorm is het proces precies hetzelfde als *Sort-merge*. We merken op dat het aantal getallen in het algemeen geen tweemacht is en dat er dus af en toe groepjes van ongelijke lengte gemengd moeten worden.

Het op de vorige pagina beschreven principe hebben we enigszins gewijzigd gebruikt in het programma *Omni-merge*. Het systeem is zodanig aangepast dat de sorteenheden uit meerdere elementen kunnen bestaan en er aanhangeenheden aan een sorteenheid mogen hangen. Tevens zullen we een frekwentieteller bijhouden. Deze teller maakt het mogelijk om, zonder verlies van informatie over de oorspronkelijke verzameling sorteenheden, van twee identieke sorteenheden die we tijdens het mengen tegenkomen er een te laten wegvallen en de ander de aanhangeenheden van de eerste er bij te geven. Dit overdragen van aanhangeenheden gebeurt zo dat: de aanhangeenheid van een eerder aangeboden sorteenheid altijd voor de aanhangeenheid van een later aangeboden identieke sorteenheid staat.

2.2.5. Gemengd sorteren met een binaire boom en met de files: de strategie van *Omni-sort*.

In 2.2.2. zijn we geëindigd met de opmerking dat er problemen zijn als het voor de boom gereserveerde array vol raakt. Een goede oplossing voor die problemen staat in 2.2.4.
Als de boom vol is schrijven we de sorteenheden met hun frekwentieteller en hun aanhangeenheden naar file 1 en we gaan een nieuwe boom opbouwen. Als die vol is schrijven we ook deze naar file 2. De volgende boom gaat weer naar file 1 enz.. Als alles zo verwerkt is, schakelen we het mengsysteem van 2.2.4. in. Ook hier is er voor gezorgd dat de aanhangeenheid van een eerder aangeboden sorteenheid altijd voor de aanhangeenheid van een later aangeboden identieke sorteenheid staat.

# 3. Het programma

## 3.1. Beschrijving

### 3.1.1. file simulatie

integers

*L drum* — Het aantal op de drum voor de files gereserveerde geheugenplaatsen.

*L brok* — De lengte van de brokstukken waarin de drum verdeeld wordt, tevens de lengte van de transportbuffers.

*m* — Het aantal brokstukken.

integer arrays

*ketenboek* — Hiervan verwijzen de elementen naar de brokstukken op de drum

*start keten*, *eind keten*, *lengte keten* — Door het ketenboek lopen vijf ketens: keten 0 is de vrije lijst, de ketens 1 t/m 4 bevatten de brokstukken voor iedere file. Van iedere keten is het begin, het eind en de lengte gegeven door *start keten*, *eind keten* en *lengte keten*.

*brok 1, ..., brok 8* — Voor de $i$-de file zijn twee buffers gereserveerd namelijk *brok (2×i-1)* en *brok(2×i)*, om beurten dient de een als werkarray en de ander als transportbuffer.

*bp, ep* — De lees- en schrijfpointer in het werkarray.

Boolean arrays

*lezen* — *lezen* [$i$] is *true* als de laatste handeling, die in file nr. $i$ is gebeurd, lezen is.

*even* — *even* [$i$] is *true* als op dit moment de evenbuffer van file nr. $i$ werkarray is.

procedures

*procedure* — In deze procedure wordt een formele procedure *pi* aangeroepen met als parameters resp. het werkarray en de transportbuffer van de file nr. $i$.

*uit*

Dient om het programma na het afdrukken van een foutmelding te verlaten.

*geef schakel van(i) aan:(j)*

Dient om de eerste schakel van de *i*-de keten achter de *j*-de keten te hangen. De inhoud van iedere schakel (een schakel is een element van ketenboek) is de index van de volgende schakel of, bij de laatste schakel, 0. De lengte van beide ketens wordt bijgehouden, indien de lengte van de *i*-de keten negatief dreigt te worden - dat kan alleen voorkomen bij de vrije lijst - wordt een foutmelding gegeven. Ook *startketen*[*i*] en *eindketen*[*j*] krijgen hun nieuwe waarde. Deze procedure wordt uitsluitend in de twee volgende procedures aangeroepen.

*sla(brok) op voor:(i)*

Er wordt een schakel van de vrije lijst aan de *i*-de keten gegeven, en de inhoud van *brok* wordt in het met die schakel corresponderende brokstuk op de drum geschreven. Deze procedure wordt uitsluitend in *outbin* aangeroepen.

*geef(brok) vrij voor:(i)*

Indien er voor de *i*-de keten brokken op de drum gereserveerd zijn dan wordt de inhoud van *brok* met het eerste brokstuk gevuld, en de bijbehorende schakel achter de vrije lijst gehangen. Deze procedure wordt uitsluitend in *inbin* aangeroepen.

*outbin(i,kar)*

Dient om kar achteraan de *i*-de file te schrijven. Hierin is de procedure *outbin i* gedeclareerd die dat opslaan beschrijft voor het geval, waarin een formeel array *evenbrok* het werkarray is en *onevenbrok* het transportarray. De aanroep *procedure (i,outbin i)* zorgt ervoor dat *evenbrok* en *onevenbrok* worden vervangen door de juiste buffers.

Indien de laatste handeling in deze file lezen was, dan wordt er geïnitialiseerd om te gaan schrijven, de werk- en transportarrays verwisselen van rol en *outbin* wordt opnieuw aangeroepen met dezelfde parameters.

Indien de laatste handeling schrijven of initialiseren om te schrijven was, dan wordt *kar* in het werkarray geschreven. Indien het werkarray vol is wordt dat opgeslagen, het werk- en het transportarray verwisselen van rol en de schrijfpointer *(ep)* wordt 1.
Als het werkarray niet vol is dan wordt de schrijfpointer opgehoogd.

*inbin (i)*

Deze integer procedure krijgt de waarde van het voorste element van de file. Hierin is een procedure *inbin i*, die een zelfde rol vervult als *outbin i* in *outbin*, gedeclareerd.

Als de laatste handeling in deze file schrijven was dan wordt er geïnitialiseerd om te lezen (d.w.z. *lezen [i] := true)*.
Als de lengte van de keten positief is, dan wordt het transportarray gevuld met het eerste brokstuk van deze file op de drum, het werkarray wordt naar de drum geschreven en vervolgens wordt dit gevuld met het tweede brokstuk van deze file en het werk- en transportarray verwisselen van rol.
Indien de lengte van de keten 0 is, dan staat kennelijk alles wat in deze file geschreven is in het werkarray en daaruit kan nu dus zonder meer gelezen worden. Uiteindelijk wordt *inbin(i)* opnieuw aangeroepen.

Als de laatste handeling in deze file lezen of het initialiseren voor lezen was, dan wordt *inbin* het door de leespointer (*bp*) aangewezen element van het werkarray.
Indien het werkarray nog niet op is wordt *bp* opgehoogd, anders wordt het werkarray gevuld met een nieuw brokstuk van de drum en wordt dus transportarray, het oude transportarray is tijdens het lezen uit het vorige werkarray gevuld met een brokstuk van de drum en wordt nu zelf werkarray.

*initialiseer voor files*

Allerlei identifiers krijgen een startwaarde.
Alle schakels worden aan de vrije lijst gehangen.
Deze procedure wordt onmiddellijk na de declaraties aangeroepen.

## 3.1.2. comprimatie en decomprimatie

Wij gebruiken de in 2.1. beschreven "alfabetische aftelling" voor de comprimatie. Indien we maximaal $c$ symbolen per machinewoord onthouden, dan is de codering voor een groepje van $j$ symbolen $(j \leq c)$:

### 3.1.2.1.

$$(l_1-1)\,(n^{c-1}+\ldots+n+1)+1$$
$$+\,(l_2-1)\,(n^{c-2}+\ldots+n+1)+1$$
$$\vdots$$
$$+\,(l_j-1)\,(n^{c-j}+\ldots+n+1)+1.$$

Hierin is $n$ het aantal verschillende symbolen, er is aangenomen dat de codering van de symbolen van het alfabet loopt van *1 t/m n*.
$l_1,\ldots,l_j$ stelt de codering van ieder symbool voor.

Het groepje dat uit $c$-maal het $n$-de symbool bestaat krijgt dan de grootst mogelijke codering namelijk:

3.1.2.2. $n^c + n^{c-1} + \ldots + n$.

integers

*L alfabet* Speelt de rol van $n$.

*aantal in mw* Speelt de rol van $c$.

*g* Het grootste getal dat in één machinewoord onthouden kan worden.

*max* Krijgt de waarde van 3.2.1.2.

Boolean

*Geinit voor compr* Dient om te controleren of er wel voor de comprimatie geïnitialiseerd is.

integer arrays

*code* Hierin wordt de codering onthouden; omdat in 3.1.2.1. steeds de factor (code-1) voorkomt laten we de code niet van *1* tot $n$, maar van *0* tot *(n-1)* lopen.

*decode* Bevat de décodering van de codering.

*ophoger* Hierin staan de factoren $(n^{c-j}+n^{c-j-1}+ \ldots +n+1)$ uit 3.1.2.1.

procedures

*Initialiseer voor compr* In deze procedure wordt van de band gelezen: *g*, *L alfabet*, en de codering. De décodering wordt tijdens het inlezen van de codering bepaald. Aan de hand van *g* en *L alfabet* wordt *ophoger* gevuld, en *max* en *aantal* in *mw* vastgesteld. De waarde van enkele van deze grootheden wordt afgedrukt. Er wordt een foutmelding gegeven als *L alfabet* $\leq$ *1* (Indien we toestaan *L alfabet = 1*, dan zouden we moeten accepteren dat *aantal* in *mw* en daarmee de lengte van het array *ophoger* gelijk aan *g* zijn). In *geinit voor compr* wordt onthouden dat deze procedure is aangeroepen.

*Compr* — Met deze procedure worden symbolen die staan in $A[1],\ldots, A[l]$ gecomprimeerd en $l$ wordt aangepast. Indien een van de elementen $A[1],\ldots, A[l]$ geen RESYM-code bevat, of een RESYM-code bevat die niet in het alfabet staat dan wordt er een foutmelding gegeven.

*Decompr* — De getallen die in $A[1], \ldots, A[l]$ staan worden opgevat als gecomprimeerde symbolen, en worden gedécomprimeerd waarbij $l$ wordt aangepast. Indien een getal ter décomprimatie wordt aangeboden dat negatief of groter dan *max* is, dan wordt een foutmelding gegeven.

Zowel in *Compr* als in *Decompr* wordt nagegaan of *Initialiseer voor compr* is aangeroepen, zoniet dan wordt een foutmelding gegeven.

3.1.3. Het sorteerproces

integers

*Vrije index* — Wijst de eerste plaats aan in het array *Array* die nog vrij is. (In *Array* wordt de boom opgebouwd).

*L array* — De lengte van *Array*.

*L sort* — Heeft twee functies: op de eerste plaats is het de lengte van het array *Sort* zoals deze lengte van de band af ingelezen wordt, ten tweede is het een pointer die tijdens de uitvoering van de procedures *Uitvoer sort* en *Uitvoer aanhang* aangeeft hoeveel plaatsen van *Sort* gevuld zijn.

*L aanhang* — Analoog aan *L sort*: de lengte van het array *Aanhang*, en een pointer in dat array tijdens de uitvoering van de procedure *Uitvoer aanhang*.

| | |
|---|---|
| *Frekw* | Heeft tijdens de uitvoering van *Uitvoer sort* en *Uitvoer aanhang* een waarde die gelijk is aan de frekwentie van de bijbehorende sorteenheid. |
| *Ct* | Telt het aantal malen dat twee sorteenheden met elkaar vergeleken worden.(comparitie teller) |
| *it* | Telt het aantal comparities sinds de laatste verplaatsing in de boom (zie 2.2.2.). |
| *C* | Bevat de drempel die *it* moet overschrijden voordat een nieuwe verplaatsing in de boom plaats vindt. |
| *Ta1* | Telt het aantal sorteenheden dat aangeboden wordt. |
| *Ta2* | Telt de frekwenties van alle verschillende sorteenheden bij elkaar op. (*Ta2* is dus gelijk aan *Ta1*) |
| *Taa1* | Telt het aantal aanhangeenheden dat aangeboden wordt. |
| *Taa2* | Telt het aantal aanhangeenheden dat afgeleverd wordt (dus *Taa1* is *Taa2*). |
| *Av* | Telt het aantal verschillende sorteenheden. |
| *Oav* | Aantal verschillende sorteenheden dat gevonden was toen de vorige boom naar de files werd geschreven. |
| *Bt* | Telt het aantal bomen dat opgebouwd wordt. |
| *start* | Hierin staat de index van het onderste element van de boom. |
| *Nieuwe index* | Een integer waarin tijdens de aanroep van *Voeg in* de index staat van de volgende sorteenheid in de boom waarmee de nieuwe sorteenheid moet worden vergeleken. Buiten deze procedure heeft *Nieuwe index* de waarde 5 als er in de boom minstens een sorteenheid staat, en anders -1. Hiervan maken we gebruik als we het eerste element in de boom plaatsen. |
| *i* | Een hulpinteger. |
| *Aanhangeind* | Heeft altijd de waarde -1. Iedere sort- of aanhangeenheid bestaat uit een aantal woorden voorafgegaan door een woord waarin dat aantal staat. Een aanhangketting is een aantal aanhangeenheden gevolgd door een woord met als inhoud -1. |

*stukeind* Heeft altijd de waarde -2.
Onder een "stuk" verstaan we een aantal gesorteerde sorteenheden, elke sorteenheid voorafgegaan door zijn frekwentie en gevolgd door zijn aanhangketting. Het stuk wordt afgesloten door een woord met als inhoud -2.

*aanhangeind zonder stopteken*
Heeft altijd de waarde -3; zie *spui eenheden van(i) naar: (j,par).*

Zoals in 2.2.3. beschreven is, gebeurt het mengen met behulp van de files door uit twee files een "stuk" te lezen en deze twee - al mengend tot een stuk - naar een van de twee andere files te schrijven.

*F1* Is het nummer van de eerste van de files waaruit gelezen wordt.

*f1* Is het nummer van de file waarin geschreven wordt.

Booleans

*Klein voor groot* Deze boolean beïnvloedt het sorteercriterium: zie 1. en de procedure *Sort bij.*

*Compr sort* Indien deze boolean *true* is wordt *Sort* automatisch gecomprimeerd en gedécomprimeerd: zie 1.

*Compr aanhang* Analoog.

*Dezelfde* Zie de procedure *Sort bij.*

*Geinit* Wordt gebruikt om na te gaan of er geïnitialiseerd is.

Vervolgens worden enkele van deze grootheden ingelezen, hun waarde wordt met wat begeleidende tekst afgedrukt en een nieuw blok wordt geopend.

integer arrays

*Array* Het array waarin de boom komt te staan.

*Sort* Hierin staan bij het aanroepen van *Voeg in* en *Uitvoer sort* de sorteenheden.

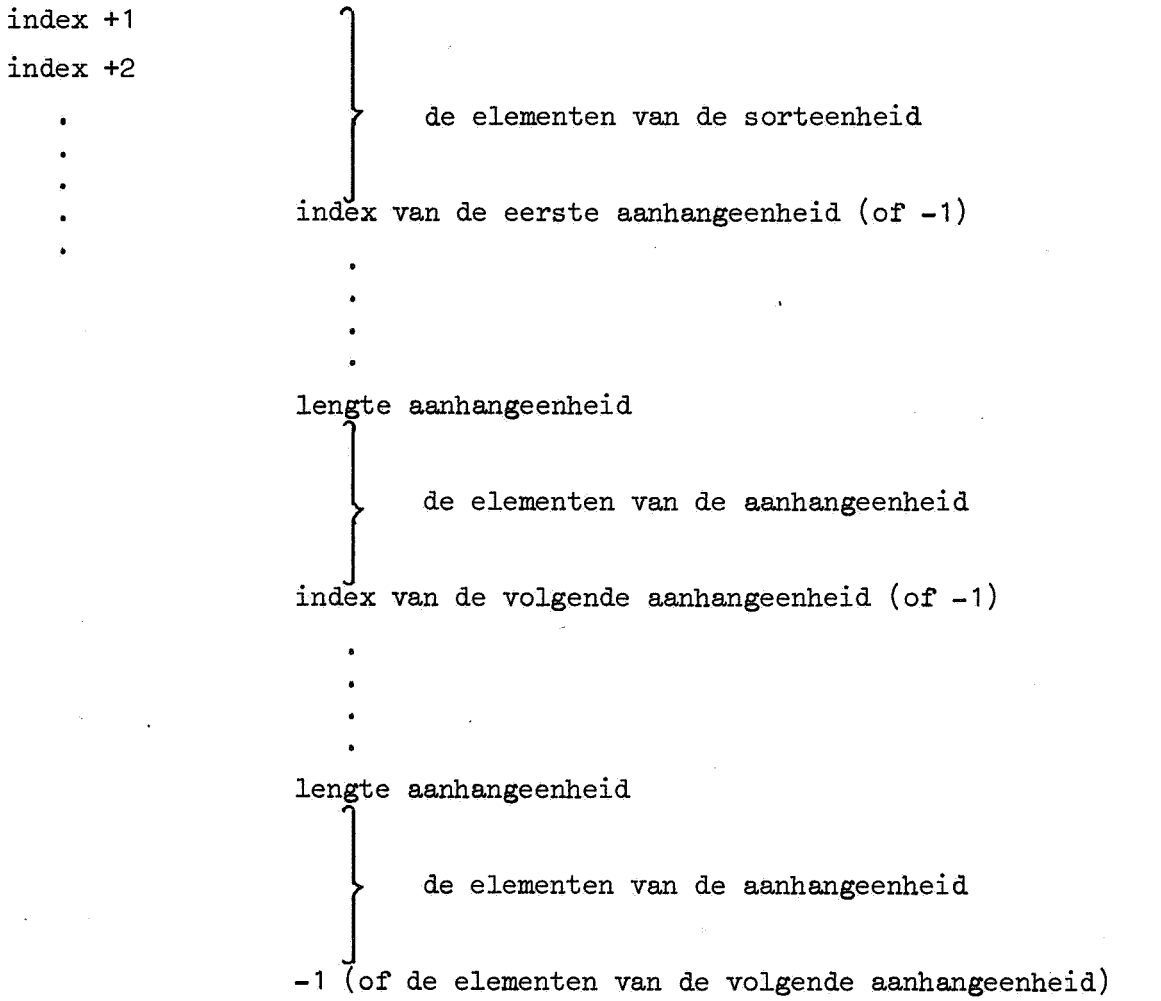
*Aanhang* Analoog.
De plaats *Sort*[*0*] wordt soms gebruikt om de lengte van de sorteenheden te onthouden, bijvoorbeeld ten behoeve van de volgende procedure.

procedures

*Sort bij* (In de declaratie staat: *Sort bij(i1) in:(A1) is dezelfde of opvolger van:(i2,A2)*).
Deze Boolean procedure definieert een orde-relatie tussen een aantal elementen van de arrays *A1* en *A2*. Het aantal daarvoor relevante elementen staat bij *A1*[*i1*] resp. *A2*[*i2*], de relevante elementen staan in de opvolgende plaatsen die beginnen bij *A1*[*i1+1*] resp. *A2*[*i2+1*].
Indien de relevante elementen van beide arrays identiek zijn dan *Sort bij:= Dezelfde:= true*.

Organisatie van *Array*

| array index | inhoud |
|---|---|
| index -4 | *kettingindex* |
| index -3 | index van voorganger als die ontbreekt of bij het kleinste element 0. *linkerindex* |
| index -2 | index van opvolger als die er niet is of bij het grootste element 0. *rechterindex* |
| index -1 | frekwentie van de sorteenheid |
| index | lengte sorteenheid |




Dit is de situatie bij een sorteenheid met twee aanhangeenheden. In de notatie van 2.2.2. zou "index van voorganger" moeten worden vervangen door ←, "index van opvolger" door →, *rechterindex* door => en *linkerindex* door <= .

De verwijzing vanuit (*index -4*) naar het element onder de laatste aanhangeenheid dient om indien er bij deze sorteenheid een nieuwe aanhangeenheid komt, deze makkelijk achter de ketting te kunnen hangen.

*rechterindex* is de index van het laatste element, dat we "op weg omhoog" in deze boom zijn tegengekomen, dat in onze ordening na deze sorteenheid komt, *linkerindex* is het laatste element vóór deze sorteenheid dat we zijn tegengekomen.

Of er in *Array[index -2]* een *rechterindex* of een "index van opvolger" staat, kunnen we zien omdat de laatste positief en de eerste negatief onthouden wordt, hetzelfde geldt voor *linkerindex* en "index van voorganger".

*Voeg in*

Deze procedure verzorgt bovenstaande organisatie. Lokale procedures zijn *Plaats sort* en *Plaats aanhang*. *Voeg in* controleert of er geïnitialiseerd is, of de lengte van de sorteenheid positief en de lengte van de aanhangeenheid niet negatief is, en geeft zonodig foutmeldingen daarvoor.
Indien daarom gevraagd is worden de sort- en de aanhangeenheden gecomprimeerd.
Vervolgens vindt een proces plaats volgens de methode van 2.2.2., tijdens dit proces worden *rechterindex* en *linkerindex* bijgehouden. Na iedere *2ln(Av-0av+10) (=C)* comparities brengen we een verandering in de boom aan.
Indien we een sorteenheid invoegen die al in de boom voorkomt wordt de frekwentie opgehoogd en eventueel *plaats aanhang* aangeroepen.

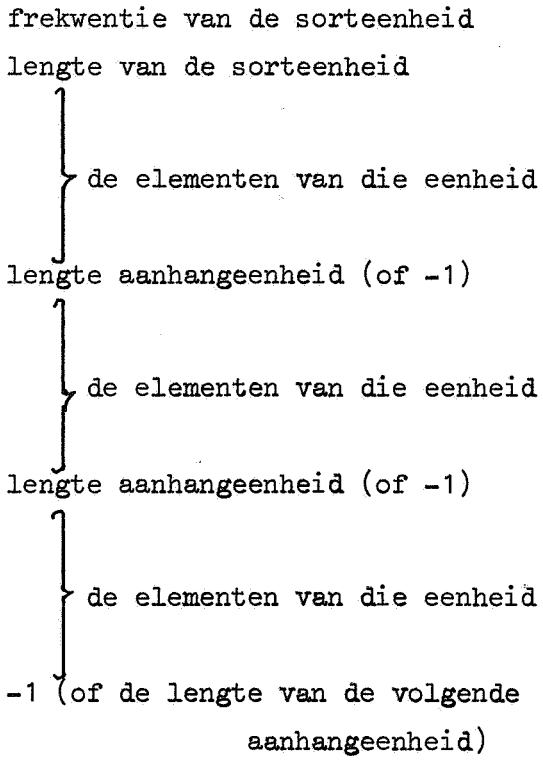
*plaats sort*

Deze procedure plaatst de sorteenheid. Hij begint met te controleren of er in de boom nog ruimte genoeg is voor de sort- en de aanhangeenheid, zoniet dan wordt de boom in gesorteerde volgorde naar de file geschreven, *Array* wordt schoongemaakt en *Voeg in* wordt opnieuw aangeroepen. Indien er een aanhangeenheid is roept *plaats sort* zelf *plaats aanhang* aan.

*plaats aanhang* Plaatst de aanhangeenheid. Omdat deze procedure ook wordt gebruikt om een aanhangeenheid achter een bestaande sorteenheid te hangen, is het nodig dat er opnieuw gekeken wordt of er voldoende plaats in de boom is.

*Naar file* Deze procedure schrijft de in de boom opgeslagen sorteen bijbehorende aanhangeenheden in gesorteerde volgorde naar de file.

De organisatie van de sorteenheid met zijn aanhangheden op de file is:




De eenheden van één boom (i.h.a. sorteenheden met hun aanhangeenheden die in de juiste volgorde staan (zoiets noemen wij een "stuk")) worden afgesloten met het getal -2.

Het afbreken van de boom gebeurt met een proces dat overeenkomt met de laatste versie van *rol op de boom* uit 2.2.2. .
De bomen worden afwisselend naar de eerste en de tweede file geschreven.

*Mengen van de gesorteerde stukken*

Het sorteerproces bestaat uit een aantal scans, waarin stukken van de files *f1* en *f2* worden gelezen, tot een stuk worden gemengd en afwisselend naar file *f3* en *f4* worden geschreven. Aan het eind van iedere scan verwisselen *f1* en *f2* van rol met *f3* en *f4*. Het proces wordt beëindigd wanneer alles in *f1* of *f3* staat.
In deze procedure komen de volgende declaraties voor:

integers

*kar, k, a* Hulpvariabelen

*f1, ..., f4 en fl* Verwisselende nummers van de files.

*as* Aantal stukken dat er bij het begin van een scan nog is.

*nas* Aantal stukken die door het mengen van twee stukken tijdens één scan ontstaan zijn.

*n* Aantal stukken dat tijdens een scan gemengd is. (dus $n=2\times nas$).

integer arrays

*S1, S2* In deze arrays komen de sorteenheden te staan die we met behulp van *Sort bij* met elkaar willen vergelijken. In *S1*[*-1*] en *S2*[*-1*] staat de frekwentie en in *S1*[*0*] en *S2*[*0*] de lengte van die sorteenheden.

procedures

*spui eenheden van (i) naar:(j,par)*

Deze procedure spuit een deel van de $i$-de file naar de $j$-de. Het eerste symbool dat van de $i$-de file gelezen wordt mag niet de frekwentie van een sorteenheid zijn.

De parameter *par* kan de waarden -1 (*aanhangeind*), -2 (*stukeind*) of -3 (*aanhangeind zonder stopteken*) hebben.
In het eerste geval wordt er gespuid tot en met het einde van de eerste ketting van aanhangeenheden, in het tweede tot en met het einde van het eerste stuk en in het derde geval tot het einde van de eerste ketting van aanhangeenheden. (De afsluiter -1 wordt dan niet in de *j*-de file gezet!).

*Meng en spui twee stukken*

In deze procedure worden van de files met nummers *f1* en *f2* een stuk gelezen en al mengend naar de file met nummer *fl* geschreven. Uit beide files wordt een sorteenheid gelezen, deze twee worden met elkaar vergeleken en de sorteenheid die in onze ordening voorganger is wordt naar file nr. *fl* geschreven. Vervolgens wordt de aanhangeenheid van de voorganger naar *fl* geschreven en er wordt uit het stuk van de voorganger een nieuwe sorteenheid ingelezen enz.. Als beide sorteenheden gelijk zijn wordt een van hen naar file nr. *fl* geschreven (de frekwenties zijn bij elkaar opgeteld), gevolgd door de aanhangeenheden van beiden (de aanhangketting uit *f1* eerst). Hulpprocedures zijn de volgende twee procedures:

*spui sort van(f) naar:(S)*

Als *f = f1* dan is *S* altijd het array *S1*, en als *f = f2* dan *S = S2*. Indien in het stuk op file *f* geen sorteenheid meer staat, dan wordt de rest van het stuk van de andere file naar *fl* geschreven.

*spui sort uit(S) naar:(f)*

De tegenhanger van *spui sort van (f) naar:(S).*

Met behulp van bovenstaande procedures vindt dan het *mengen van de gesorteerde stukken* plaats. Na dit proces staat alles in gesorteerde volgorde op file nr. *F1.*

*Uitvoer*

Deze procedure haalt de sort- en de aanhangeenheden van file nr. *F1*, zet ze in de arrays *Sort* dan wel *Aanhang* en geeft *L sort*, *L aanhang* en *Frekw* hun waarde. Indien *Compr sort* of *Compr aanhang true* is wordt het bewuste array gedécomprimeerd. Voor iedere verschillende sorteenheid wordt *Uitvoer sort* en voor iedere aanhangeenheid wordt *Uitvoer aanhang* aangeroepen.

*Lever het gesorteerde af*

Deze procedure roept *Naar file* aan waardoor alles wat op dat moment in de boom staat in goede volgorde naar de file wordt geschreven. Daarna wordt *Mengen van de gesorteerde stukken* en tenslotte *Uitvoer* aangeroepen.
Tussendoor wordt wat informatie over het sorteerproces afgedrukt.

*Initialiseer*

Zoals de identifier zegt zorgt deze procedure voor de initialisatie.

*Initialiseer voor boom*

*Initialiseer* roept zelf *Initialiseer voor boom* aan. Ook als de boom een keer vol is geweest wordt de laatste procedure aangeroepen.

```
begin comment   3.2. Programma-tekst.

                file simulatie ;


integer L drum, L brok, m;
L drum:= READ; L brok:= READ; PRINTTEXT(≮ L drum en L brok ≯);
SPACE(39); PRINT(L drum); PRINT(L brok); m:= L drum : L brok;
begin integer array ketenboek[1:m], start keten, eindketen, lengte
  keten[0:5], brok1, brok2, brok3, brok4, brok5, brok6, brok7,
  brok8[1:L brok], bp, ep[1:4];
  Boolean array lezen, even[1:4];

  procedure procedure(i, pi); value i; procedure pi; integer i;
  if i = 1 then
  begin if even[1] then pi(brok1, brok2) else pi(brok2, brok1) end
  else if i = 2 then
  begin if even[2] then pi(brok3, brok4) else pi(brok4, brok3) end
  else if i = 3 then
  begin if even[3] then pi(brok5, brok6) else pi(brok6, brok5) end
  else if i = 4 then
  begin if even[4] then pi(brok7, brok8) else pi(brok8, brok7) end;

  procedure uit(S); string S;
  begin NLCR; NLCR; PRINTTEXT(S); EXIT end;

  procedure geef schakel van(i)aan: (j); value i, j; integer i, j;
  begin integer schakel;
    lengte keten[i]:= lengte keten[i] - 1;
    if lengte keten[i] < 0 then uit(
    ≮de gesimuleerde files zijn vol≯); schakel:= start keten[i];
    start keten[i]:= ketenboek[schakel]; if start keten[j] = 0 then
    begin start keten[j]:= eind keten[j]:= schakel;
      lengte keten[j]:= 1
    end
    else
    begin lengte keten[j]:= lengte keten[j] + 1;
      eind keten[j]:= ketenboek[eind keten[j]]:= schakel
    end;
    ketenboek[schakel]:= 0
  end geef schakel;

  procedure sla(brok)op voor: (i); value i; integer i;
  integer array brok;
  begin geef schakel van(0)aan: (i);
    TO DRUM(brok, L brok × (eind keten[i] - 1))
  end;

  procedure geef(brok)vrij voor: (i); value i; integer i;
  integer array brok; if lengte keten[i] > 0 then
  begin FROM DRUM(brok, L brok × (start keten[i] - 1));
    geef schakel van(i)aan: (0)
  end;
```

```
procedure outbin(i, kar); value i, kar; integer i, kar;
begin

  procedure outbin i(evenbrok, onevenbrok);
  integer array evenbrok, onevenbrok; if lezen[i] then
  begin lezen[i]:= false; bp[i]:= ep[i]:= 1; even[i]:= ¬even[i];
    outbin(i, kar)
  end
  else
  begin evenbrok[ep[i]]:= kar;
    if ep[i] < L brok then ep[i]:= ep[i] + 1 else
    begin ep[i]:= 1; sla(evenbrok)op voor: (i); even[i]:= ¬even[i]
    end
  end outbin i;

  procedure(i, outbin i)
end out bin;
integer procedure inbin(i); value i; integer i;
begin

  procedure inbin i(evenbrok, onevenbrok);
  integer array evenbrok, onevenbrok; if ¬lezen[i] then
  begin lezen[i]:= true; if lengte keten[i] > 0 then
    begin geef(onevenbrok)vrij voor: (i); sla(evenbrok)op voor: (i);
      geef(evenbrok)vrij voor: (i); even[i]:= ¬even[i]
    end;
    inbin:= inbin(i)
  end
  else
  begin inbin:= evenbrok[bp[i]];
    if bp[i] < L brok then bp[i]:= bp[i] + 1 else
    begin geef(evenbrok)vrij voor: (i); even[i]:= ¬even[i];
      bp[i]:= 1
    end;
  end inbin i;

  procedure(i, inbin i)
end inbin;

procedure initialiseer voor files;
begin integer i;
  for i:= 1 step 1 until 4 do
  begin lengte keten[i]:= start keten[i]:= 0; bp[i]:= ep[i]:= 1;
    lezen[i]:= even[i]:= true
  end;
  lengte keten[0]:= eind keten[0]:= m; start keten[0]:= 1;
  for i:= 1 step 1 until m - 1 do ketenboek[i]:= i + 1;
  ketenboek[m]:= 0
end initialiseer voor files;

initialiseer voor files;
begin comment comprimatie;
```

```
    integer Lalfabet, aantal in mw, g, max;
    Boolean Geinit voor compr;
    integer array code, decode[ - 1:127], ophoger[1:27];

    procedure Initialiseer voor compr;
    begin integer i, k, sym;
      real h, s;
      g:= READ; L alfabet:= READ; Geinit voor compr:= true; NLCR;
      PRINTTEXT(‹ g  en  L alfabet›); SPACE(40); PRINT(g);
      PRINT(L alfabet); if L alfabet ≤ 1 then uit(
      ‹ L alfabet  is  te  klein›);
      for i:= 0 step 1 until 127 do code[i]:= - 1; NLCR; SPACE(72);
      for i:= 1 step 1 until L alfabet do
      begin sym:= RESYM; PRSYM(sym); code[sym]:= i - 1;
        decode[i - 1]:= sym;
      end;
      NLCR; aantal in mw:= 0; s:= 0; h:= 1;
      for h:= h × L alfabet while aantal in mw < 27 do
      begin aantal in mw:= aantal in mw + 1;
        ophoger[aantal in mw]:= s + 1; s:= s + h;
        if s > g then goto K
      end;
    K: max:= ophoger[aantal in mw] - 1;
      aantal in mw:= aantal in mw - 1; PRINTTEXT(
      ‹ aantal symbolen per machinewoord›); SPACE(24);
      PRINT(aantal in mw); NLCR
    end Initialiseer voor compr;


    procedure Compr(A, l); integer l; integer array A; if l > 0 then
    begin integer i, j, k, n, H, C;
      if ¬Geinit voor compr then uit(
      ‹ er is niet Geinitialiseerd voor comprimatie ›); k:= n:= 1;
      for i:= 0 step aantal in mw until l - 1 do
      begin H:= 0;
        for j:= aantal in mw, j - 1 while j > 1 ∧ n ≤ l do
        begin C:= A[n]; if C < 0 ∨ C > 127 then
          begin PRINT(C); NLCR; NLCR; uit(
‹ bovenstaand getal is geen RESYM-code
en kan dus niet gecomprimeerd worden›
            )
          end;
          C:= code[A[n]]; if C = - 1 then
          begin NLCR; NLCR; PRSYM(A[n]); uit(
‹bovenstaand symbool wordt ter comprimatie
aangeboden maar staat niet in het alfabet ›
            )
          end;
          H:= H + C × ophoger[j] + 1; n:= n + 1
        end;
        A[k]:= H; k:= k + 1; if n > l then goto L
      end;
    L: l:= k - 1
    end Compr;


    procedure Decompr(A, l); integer l; integer array A; if l > 0 then
```

```
begin integer H, K, i, j, k;
  integer array A1[1:1];
  k:= 0;
  for i:= 1 step 1 until 1 do A1[i]:= A[i];
  for i:= 1 step 1 until 1 do
  begin H:= A1[i] - 1; if H < 0 ∨ H > max then
    begin NLCR; PRINT(H); NLCR; uit(
      ⊄ bovenstaand getal  kan niet gedecompimeerd worden ⊅)
    end
    else
    for j:= aantal in mw step - 1 until 1 do
    begin K:= H : ophoger[j]; k:= k + 1; A[k]:= decode[K];
      H:= H - K × ophoger[j] - 1; if H < 0 then goto M
    end;
  M:
  end;
  1:= k
end Decompr;

Geinit voor compr:= false;
begin comment hier begint het eigenlijke sorteerprogramma;
  integer Vrije index, L array, L sort, L aanhang, Frekw, Ct, it,
  C, Ta1, Ta2, Taa1, Taa2, Av, Oav, Bt, start, Nieuwe index, i,
  stukeind, aanhangeind, aanhangeind zonder stopteken, F1, f1;
  Boolean Klein voor groot, Compr sort, Compr aanhang, Dezelfde,
  Geinit;
  L array:= READ; L sort:= READ; L aanhang:= READ;
  Klein voor groot:= READ = 1; Compr sort:= READ = 1;
  Compr aanhang:= READ = 1; NLCR; PRINTTEXT(
  ⊄ L  array, L sort  en  L aanhang⊅); SPACE(25); PRINT(L array);
  PRINT(L sort); PRINT(L aanhang); NLCR; PRINTTEXT(
  ⊄ Klein voor groot, Compr sort, Compr aanhang⊅); SPACE(22);
  if Klein voor groot then PRINTTEXT(⊄ true ⊅) else PRINTTEXT(
  ⊄ false ⊅); SPACE(15); if Compr sort then PRINTTEXT(
  ⊄ true ⊅) else PRINTTEXT(⊄ false ⊅); SPACE(15);
  if Compr aanhang then PRINTTEXT(⊄ true ⊅) else PRINTTEXT(
  ⊄ false ⊅); Geinit:= false;
  begin integer array Array[1:L array], Sort[0:L sort],
    Aanhang[1:Laanhang];
    Boolean procedure Sort bij(i1)in:
  (A1)is dezelfde of opvolger van: (i2, A2); value i1, i2;
    integer i1, i2; integer array A1, A2;
    begin integer i, n1, a1, n2, a2, m;
      Ct:= Ct + 1; n1:= A1[i1]; n2:= A2[i2];
      m:= if n1 > n2 then n2 else n1; i:= 0; Dezelfde:= false;
      for i:= i + 1 while i < m do
      begin a1:= A1[i1 + i]; a2:= A2[i2 + i]; if a1 ≠ a2 then
        begin Sort bij:= a1 > a2 = Klein voor groot;
          goto K
        end
      end;
      if n1 = n2 then Sort bij:= Dezelfde:= true else Sort bij:=
      n1 > n2 = Klein voor groot;
```

```
K:
end Sort bij;


procedure Voeg in(Lengte s, Lengte a);
value Lengte s, Lengte a; integer Lengte s, Lengte a;
begin integer rechter index, linker index, ketting index,
  index, oude index, r, or, nr, h;

  procedure plaats sort;
  begin integer i;
    Vrije index:= Vrije index + 4;
    if Vrije index + Lengte s + Lengte a + 4 > L array then
    begin Naar file; Initialiseer voor boom; goto S end
    else Av:= Av + 1; Array[Vrije index]:= Lengte s;
    for i:= 1 step 1 until Lengte s do Array[Vrije index +
    i]:= Sort[i]; if index > 0 then
    begin Array[Vrije index - 2]:= - rechter index;
      Array[Vrije index - 3]:= - linker index;
      Array[index - r]:= Vrije index
    end;
    Array[Vrije index - 1]:= 1;
    ketting index:= Vrije index - 4;
    Vrije index:= Vrije index + Lengte s + 2;
    Array[ketting index]:= if Lengte a > 0 then Vrije index
    + Lengte a + 1 else Vrije index - 1;
    ketting index:= Vrije index - 1;
    if Lengte a > 0 then plaats aanhang else Array[ketting
    index]:= - 1; goto ingevoegd
  end plaats sort;


  procedure plaats aanhang;
  begin integer i;
    if Vrije index + Lengte a + 2 > L array then
    begin Array[index - 1]:= Array[index - 1] - 1; Naar file;
      Initialiseer voor boom; goto S
    end;
    Array[ketting index]:= Vrije index;
    Array[Vrije index]:= Lengte a;
    for i:= 1 step 1 until Lengte a do Array[Vrije index +
    i]:= Aanhang[i];
    Vrije index:= Vrije index + Lengte a + 2;
    Array[Vrije index - 1]:= - 1
  end plaats aanhang;

  if ¬Geinit then uit(‹ er is niet Geinitialiseerd›);
  if Lengte s < 0 then uit(
  ‹Lengte sorteenheid is niet positief›);
  if Lengte a < 0 then uit(
  ‹Lengte aanhangeenheid is negatief›);
  if Compr sort then Compr(Sort, Lengte s);
  if Compr aanhang then Compr(Aanhang, Lengte a);
  Sort[0]:= Lengte s; Ta1:= Ta1 + 1;
  if Lengte a > 0 then Taa1:= Taa1 + 1;
```

```
S: index:= rechter index:= linker index:= r:= or:= nr:=
  oude index:= 0; C:= 2 × ln(Av − Oav + 10);
L: if Nieuwe index < 0 then plaats sort else
  begin it:= it + 1; if it > C then
    begin if Nieuwe index ≠ start then
      begin h:= Array[Nieuwe index − nr];
        Array[index − r]:= if h ≠ − index then h else −
        Nieuwe index; Array[Nieuwe index − nr]:= index;
        if oude index ≠ 0 then Array[oude index − or]:=
        Nieuwe index else start:= Nieuwe index
      end;
      it:= 0
    end;
    if r ≠ 0 then oude index:= index; or:= r;
    index:= Nieuwe index
  end;
  if Sort bij(0)in:
(Sort)is dezelfde of opvolger van: (index, Array) then
  begin if Dezelfde then
    begin Array[index − 1]:= Array[index − 1] + 1;
      if Lengte a > 0 then
      begin ketting index:= Array[index − 4]; plaats aanhang;
        Array[index − 4]:= Vrije index − 1
      end;
      goto ingevoegd
    end
    else r:= 2
  end
  else r:= 3; Nieuwe index:= Array[index − r];
  if Nieuwe index = oude index ∧ Nieuwe index ≠ 0 then
  begin if r = 2 then linker index:= index else rechter
    index:= index; r:= or; index:= oude index;
    oude index:= 0; Nieuwe index:= Array[index − r]
  end;
  if r = 2 then linker index:= index else rechter index:=
  index; nr:= if r = 2 then 3 else 2; goto L;
ingevoegd: Nieuwe index:= start
end Voeg in;

procedure Naar file;
begin integer index, nieuwe index;

  procedure Sort en ketting naar file(index); integer index;
  begin integer n, m, i;
    outbin(f1, Array[index − 1]); n:= Array[index];
    for i:= 0 step 1 until n do outbin(f1, Array[index + i]);
    m:= Array[index + n + 1]; if m = − 1 then goto K;
    n:= Array[m];
  L: for i:= 0 step 1 until n do outbin(f1, Array[m + i]);
    m:= Array[m + n + 1]; if m = − 1 then goto K;
    n:= Array[m]; goto L;
  K: outbin(f1, aanhangeind)
  end Sort en ketting naar file;

  Bt:= Bt + 1; index:= start;
```

```
L: nieuwe index:= Array[index - 3];
  if nieuwe index > 0 then
  begin index:= nieuwe index; goto L end;
M: if index = 0 then goto K; nieuwe index:= Array[index - 2];
  if nieuwe index > 0 then
  begin Sort en ketting naar file(index);
    index:= nieuwe index; goto L
  end
  else
  begin Sort en ketting naar file(index);
    index:= - nieuwe index; goto M
  end;
K: outbin(f1, stukeind); f1:= if f1 = 1 then 2 else 1
end Naar file;

procedure Mengen van de gesorteerde stukken;
begin integer kar, k, a, f1, f2, f3, f4, fl, as, nas, n;
  integer array S1, S2[ - 1:L sort];

  procedure spui eenheden van(i)naar: (j, par);
  value i, j, par; integer i, j, par;
  begin
  L: kar:= inbin(i);
    if kar = aanhangeind ∧ par = aanhangeind zonder
    stopteken then par:= aanhangeind else outbin(j, kar);
    for k:= 1 step 1 until kar do outbin(j, inbin(i));
    if par ≠ kar then
    begin if kar = aanhangeind ∧ par = stukeind then
      begin kar:= inbin(i); outbin(j, kar);
        if par ≠ kar then goto L
      end
      else goto L
    end
  end spui eenheden;


  procedure meng en spui twee stukken;
  begin

    procedure spui sort van(f)naar: (S); value f; integer f;
    integer array S;
    begin S[ - 1]:= inbin(f); if S[ - 1] = stukeind then
      begin if f = f1 then
        begin if ¬Dezelfde then spui sort uit(S2)naar: (f1)
          else
          begin kar:= inbin(f2); outbin(f1, kar) end;
          if kar ≠ stukeind then
        spui eenheden van(f2)naar: (f1, stukeind)
        end
        else
        begin if ¬Dezelfde then spui sort uit(S1)naar: (f1)
          else
          begin kar:= inbin(f1); outbin(f1, kar) end;
          if kar ≠ stukeind then
        spui eenheden van(f1)naar: (f1, stukeind)
```

```
        end;
        goto K
      end
      else Dezelfde:= false; S[0]:= kar:= inbin(f);
      for k:= 1 step 1 until kar do S[k]:= inbin(f)
    end spui sort van;


    procedure spui sort uit(S)naar: (f); value f; integer f;
    integer array S;
    begin outbin(f, S[ - 1]); kar:= S[0];
      for k:= 0 step 1 until kar do outbin(f, S[k])
    end spui sort uit;

  L: spui sort van(f1)naar: (S1); spui sort van(f2)naar: (S2);
  M: if Sort bij(0)in: (S2)is dezelfde of opvolger van: (0,
    S1) then
    begin if Dezelfde then
      begin S1[ - 1]:= S1[ - 1] + S2[ - 1]; Av:= Av - 1;
        spui sort uit(S1)naar: (f1);
        spui eenheden van(f1)naar: (f1, aanhangeind
        zonderstopteken);
        spui eenheden van(f2)naar: (f1, aanhangeind); goto L
      end;
      spui sort uit(S1)naar: (f1);
      spui eenheden van(f1)naar: (f1, aanhangeind);
      spui sort van(f1)naar: (S1); goto M
    end;
    spui sort uit(S2)naar: (f1);
    spui eenheden van(f2)naar: (f1, aanhangeind);
    spui sort van(f2)naar: (S2); goto M;
  K: f1:= if f1 = f3 then f4 else f3
  end meng en spui twee stukken;
  f1:= 1; f2:= 2; f3:= 3; f4:= 4; f1:= 3; as:= Bt;
L: if as = 1 then goto K; n:= nas:= 0;
M: if as - n = 0 then as:= nas else if as - n = 1 then
  begin outbin(f1, inbin(f1));
    spui eenheden van(f1)naar: (f1, stukeind); as:= nas + 1
  end
  else
  begin meng en spui twee stukken; n:= n + 2; nas:= nas + 1;
    goto M
  end;
  a:= f1; f1:= f3; f3:= a; a:= f2; f2:= f4; f4:= a; f1:= f3;
  goto L;
K: F1:= f1
end Mengen van de gesorteerde stukken;

procedure Uitvoer;
begin integer i;
L: Frekw:= inbin(F1); if Frekw < 0 then goto K;
  Ta2:= Ta2 + Frekw; L sort:= inbin(F1);
  for i:= 1 step 1 until L sort do Sort[i]:= inbin(F1);
  if Compr sort then Decompr(Sort, L sort); Uitvoer sort;
M: L aanhang:= inbin(F1); if L aanhang < 0 then goto L;
```

```
            Taa2:= Taa2 + 1;
            for i:= 1 step 1 until L aanhang do Aanhang[i]:= inbin(F1);
            if Compr aanhang then Decompr(Aanhang, L aanhang);
            Uitvoer aanhang; goto M;
          K:
          end Uitvoer;


          procedure Lever het gesorteerde af;
          begin integer i;
            Naar file; NLCR; PRINTTEXT(
            ¢ totale aantal sort- en aanhangeenheden ¢); SPACE(17);
            PRINT(Ta1); PRINT(Taa1); NLCR; PRINTTEXT(
¢ aantal bomen en aantal vrije geheugenplaatsen op de drum¢
            ); PRINT(Bt); PRINT(lengte keten[0] × L brok); NEW PAGE;
            Mengen van de gesorteerde stukken; Uitvoer; NEW PAGE; TAB;
            PRINTTEXT(¢ sorteer gegevens ¢); NLCR; PRINTTEXT(
            ¢ totale aantal sort- en aanhangeenheden ¢); SPACE(17);
            PRINT(Ta2); PRINT(Taa2); NLCR; PRINTTEXT(
            ¢ aantal comparities¢); SPACE(38); PRINT(Ct); NLCR;
            PRINTTEXT(¢ aantal verschillende sorteenheden¢); SPACE(23);
            PRINT(Av); NLCR;
            if Ta1 ≠ Ta2 ∨ Taa1 ≠ Taa2 then PRINTTEXT(
            ¢ tijdens het sorteren zijn eenheden verdwenen ¢); NLCR
          end;


          procedure Initialiseer;
          begin Bt:= Ct:= Ta1:= Ta2:= Taa1:= Taa2:= Av:= 0;
            Geinit:= true; Initialiseer voor boom;
            if Compr sort ∨ Compr aanhang then Initialiseer voor compr;
            aanhangeind:= - 1; aanhangeind zonder stopteken:= - 3;
            stukeind:= - 2; f1:= 1;
          end;


          procedure Initialiseer voor boom;
          begin integer i;
            for i:= 1 step 1 until 5 do Array[i]:= 0;
            Nieuwe index:= - 1; Oav:= Av; Vrije index:= 1; it:= 0;
            start:= 5
          end;

          comment hier komt het probleem-afhankelijke deel te staan;
        end
      end
    end
  end
end
```

4. Het mengen van een aantal reeds gesorteerde verzamelingen.

Een aantal op zich reeds gesorteerde verzamelingen kunnen gemengd worden door het geheel op te vatten als een willekeurige verzameling en deze op de gebruikelijke manier te sorteren. Daarbij verrichten we dan een grote hoeveelheid overbodig werk omdat we geen gebruik maken van de - aan ons bekende - ordening die het materiaal al heeft.

De procedure *Mengen van de gesorteerde stukken* (zie 3.1.) mengt gesorteerde verzamelingen, die volgens een speciale conventie in de files staan, op een efficiënte wijze. We hebben een procedure *Meng in 2 (Lengte s, Lengte a, Volgende is opvolger)* gemaakt, die een sorteenheid met zijn aanhangeenheid volgens deze conventie in de files schrijft. De Boolean *Volgende is opvolger* dient hierbij als teken dat een bepaalde op zich gesorteerde verzameling met deze sort- en aanhangeenheid nog niet afgelopen is.
Men dient zich bij gebruik van *Meng in 2* wel te realiseren dat gelijke sorteenheden die in dezelfde gesorteerde verzameling zitten ieder een aanroep van *Uitvoer sort* veroorzaken. Dit komt omdat er binnen zo'n verzameling geen sorteenheden met elkaar vergeleken worden.

We kunnen een willekeurige verzameling van $n$ elementen ook opvatten als $n$ "gesorteerde" verzamelingen van één element, en deze verzamelingen met bovenstaande methode mengen.
Hiervoor hebben we de procedure *Meng in* geschreven die precies hetzelfde doet als *Meng in 2 (lengte s, lengte a, false)*. Als we in ons sorteerprogramma *Voeg in* door deze procedure vervangen, in de procedure *Lever het gesorteerde af* de aanroep *Naar file* en in *Initialiseer* de aanroep *Initialiseer voor boom* verwijderen, dan hebben we een nieuw sorteerprogramma verkregen waarvan de omvang aanmerkelijk kleiner is. Dit is het eerder genoemde programma *Omni-merge* waarvan we de uiterlijke vorm natuurlijk nog kunnen stroomlijnen door allerlei overbodige declaraties te verwijderen.
(Als we in *Omni-merge* *Meng in* vervangen door *Meng in2* dan krijgen we het programma *Omni-merge 2* voor het mengen van reeds gesorteerde verzamelingen.)

Aan *Omni-merge* zitten twee nadelen die *Omni-sort* verkieselijk maken; ten eerste moeten bij iedere comparitie eenheden van de ene file naar de andere geschreven worden, en ten tweede komt iedere sorteenheid bij het inlezen ook werkelijk op de files te staan. Een voordeel is dat - zolang het totale aantal sorteenheden niet veel groter dan het aantal verschillende is - het aantal comparities wat kleiner is.

Ter vergelijking: van 1328 woorden - 99.5% verschillend - maakten we met *Omni-sort* in 207 sec een retrograde lijst (zie 1.2.2.) en met *Omni-merge* in 678 sec.

```
procedure Meng in(Lengte s, Lengte a); value Lengte s, Lengte a;
integer Lengte s, Lengte a;
begin   integer i;
        if ¬Geinit then uit(‹ er is niet geinitialiseerd›);
        if Lengte s<0 then uit(‹ lengte sorteenheid is niet positief›);
        if Lengte a<0 then uit(‹ lengte aanhangeenheid is negatief›);
        if Compr sort then Compr(Sort, Lengte s);
        if Compr aanhang then Compr(Aanhang, Lengte a);
        Av:=Ta1:=Ta1 + 1;
        outbin(f1, 1); outbin(f1, Lengte s);
        for i:=1 step 1 until Lengte s do outbin(f1, Sort[i]);
        if Lengte a>0 then
        begin Taa1:=Taa1 + 1; outbin(f1, Lengte a);
              for i:=1 step 1 until Lengte a do outbin(f1, Aanhang[i])
        end; outbin(f1, aanhangeind);
        Bt:=Bt + 1; outbin(f1, stukeind);
        f1:=if f1=1 then 2 else 1
end Meng in;

procedure Meng in2(Lengte s, Lengte a, Volgende is opvolger);
value Lengte s, Lengte a, Volgende is opvolger;
integer Lengte s, Lengte a; Boolean Volgende is opvolger;
begin   integer i;
        if ¬Geinit then uit(‹ er is niet geinitialiseerd›);
        if Lengte s<0 then uit(‹ lengte sorteenheid is niet positief›);
        if Lengte a<0 then uit(‹ lengte aanhangeenheid is negatief›);
        if Compr sort then Compr(Sort, Lengte s);
        if Compr aanhang then Compr(Aanhang, Lengte a);
        Av:=Ta1:=Ta1 + 1;
        outbin(f1, 1); outbin(f1, Lengte s);
        for i:=1 step 1 until Lengte s do outbin(f1, Sort[i]);
        if Lengte a>0 then
        begin Taa1:=Taa1 + 1; outbin(f1, Lengte a);
              for i:=1 step 1 until Lengte a do outbin(f1, Aanhang[i])
        end; outbin(f1, aanhangeind);
        if ¬Volgende is opvolger then
        begin Bt:=Bt + 1; outbin(f1, stukeind);
              f1:=if f1=1 then 2 else 1
        end
end Meng in2;
```