TECHNISCHE HOGESCHOOL EINDHOVEN
Afdeling Algemene Wetenschappen
Onderafdeling der Wiskunde

ALGORITHMEN

naar het college van

Prof. Dr. M. Rem

Opgetekend door

Adrie van Oers en John Peters

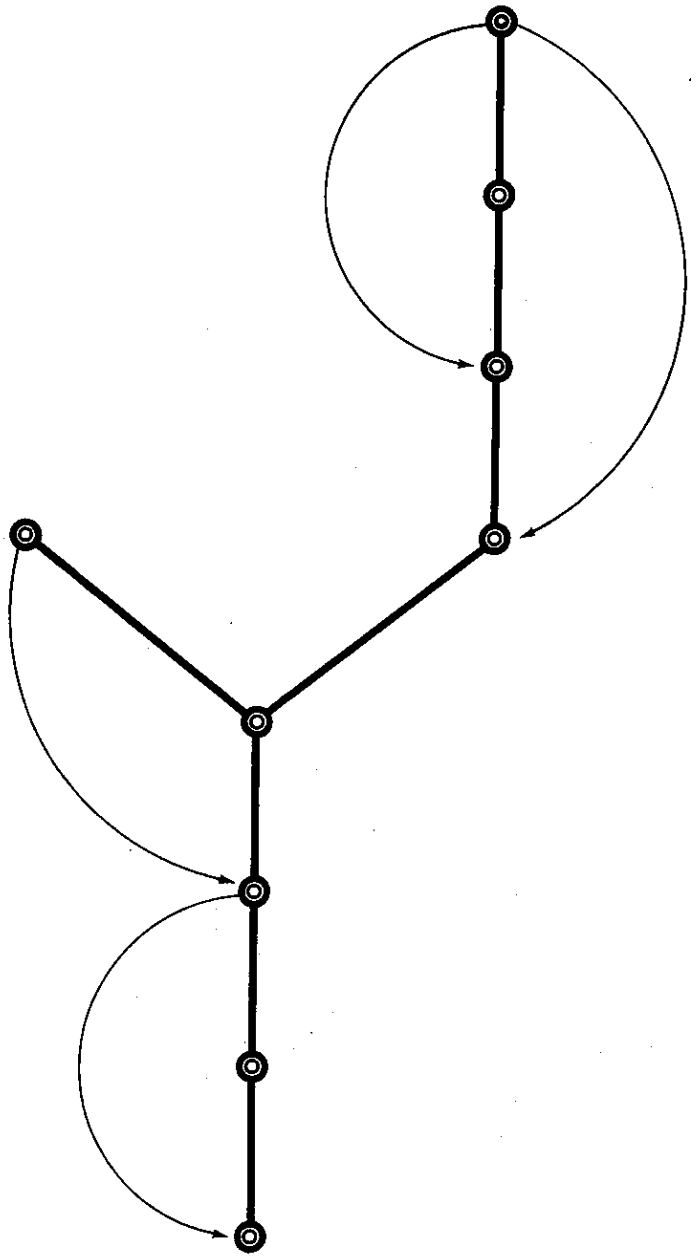
Voorjaarssemester 1981

Technische Hogeschool
Eindhoven
Dictaatnummer 2.292
Prijs f. 3,50

Onderafdeling der Wiskunde en Informatica

Algoritmen

opgetekend door Adrie van Oers en John Peters naar het college van prof.dr. M. Rem
tijdens het voorjaarssemester van 1980



TECHNISCHE HOGESCHOOL EINDHOVEN

Onderafdeling der Wiskunde

ALGORITHMEN

opgetekend door Adrie van Oers en John Peters
naar het college van Prof. dr. M. Rem
tijdens het voorjaarssemester van 1980

voorjaarssemester 1981

INHOUDSOPGAVE

INHOUDSOPGAVE

(vervolg)

1. Inleiding

Dit college heeft tot doel de student vertrouwd te maken met technieken die leiden tot het ontwerpen van efficiënte algorithmen. Deze technieken zullen gedemonstreerd worden aan de hand van o.a. de onderwerpen sorteren, operaties op verzamelingen en berekeningen aan grafen. Er zal ook ingegaan worden op een klasse van problemen die (waarschijnlijk) geen efficiënte oplossing toestaan: die der NP-volledige problemen.

Deze aantekeningen zijn beknopt en daarom waarschijnlijk niet geschikt voor een zelfstandige bestudering. Ze hebben tot doel als referentiemateriaal te dienen bij het college. Van de meeste onderwerpen wordt een literatuurverwijzing gegeven. Vaak kunnen deze verwijzingen gebruikt worden als de student zich verder in het betreffende onderwerp wil verdiepen. Het gehele college is geïnspireerd door [1] , waarin de student naast andere onderwerpen een overvloed aan oefenmateriaal kan vinden.

Voor de notatie van de algorithmen gebruiken we de guarded commands zoals die zijn ingevoerd in [2] , uitgebreid met een eenvoudig proceduremechanisme. De betekenis van een aantal aanduidingen en statements die gebruikt worden volgt hierna.

Voor een array s geldt:

```
s.hib = bovengrens van s
s.lob = ondergrens van s
s.dom = domein van s = s.hib - s.lob + 1
s.high = s(s.hib)
s.low = s(s.lob)
```

---

assignment statements worden geschreven als:

```
x := a                  : variabele x krijgt waarde a
x,y := y + 1, x - 1     : t := x; x := y + 1; y := t - 1
s:(i) = a               : array s krijgt waarde a op plaats i
s:hiext(x)              : s.hib := s.hib + 1; s:(s.hib) = x; s.dom:= s.dom + 1
s:loext(x)              : s.lob := s.lob - 1; s:(s.lob) = x; s.dom:= s.dom + 1
s:hirem                 : s.hib := s.hib - 1; s.dom:= s.dom - 1
s:lorem                 : s.lob := s.lob + 1; s.dom:= s.dom - 1
```

```
x,s:hipop                 : x := s(s.hib); s:hirem
x,s:lopop                 : x := s(s.lob); s:lorem
s:swap(x,y)               : t := s(x); s:(x) = s(y); s:(y) = t
s := (ℓ,s1,s2,...,sn)     : s.lob := ℓ; s.dom := n;
                            s:(ℓ + i - 1) = s ,   1 ≤ i ≤ n
```

Om in een algorithme naar een loop of een statement te kunnen verwijzen gebruiken we de labels L en S.
We voeren de Boolese operatoren Δ (conditional and) en ∇ (conditional or) in.
A Δ B betekent dat B niet geëvalueerd wordt als $\neg$ A geldt,
A ∇ B betekent dat B niet geëvalueerd wordt als A geldt.
We gebruiken de notatie inf om een waarde aan te duiden die groter is dan alle getallen.

We spreken verder af dat logarithmen altijd als grondtal 2 hebben, tenzij anders wordt vermeld ("ℓn" heeft e als grondtal).

De rekentijd van een algorithme zal vaak afhangen van de waarde van de input. Bij het bepalen van de benodigde rekentijd zullen we in het algemeen uitgaan van de worst case, d.w.z. die input waarvoor de uitvoering van de algorithme de meeste tijd vergt. Dit in tegenstelling tot de average case, waarbij de verwachte rekentijd bepaald wordt. Van dit laatste wordt een voorbeeld gegeven in sectie 4.2 van deze aantekeningen. Een voordeel van worst case analyse is dat we geen stochastische aannamen behoeven te maken en dat we een bovengrens voor de benodigde rekentijd krijgen.

Bij berekeningen aan rijen, verzamelingen, etc. zal de rekentijd afhangen van het aantal elementen waaruit die rijen en dergelijke bestaan. We zullen de rekentijd, T(n), uitdrukken als functie, g(n), van het aantal elementen. We letten hierbij slechts op de ordegrootte van de rekentijd: met "een algorithme is O(g(n))" bedoelen we dat er een constante c is zodat geldt $T(n) \leq c \cdot g(n)$ voor alle behalve een eindige verzameling waarden van $n \in \mathbb{N}$. We noemen deze ordegrootte ook wel de "complexiteit" van de algorithme. Wanneer een algorithme $O(n^2)$ is en een andere $O(n^3)$ noemen we de eerste efficiënter dan de tweede. Dat wil niet zeggen dat de benodigde rekentijd van de eerste altijd minder is. Ten eerste wordt de orde bepaald met worst case analyse. Ten tweede kunnen beide termen een verschillende constante hebben. Voor voldoend grote n zal de eerste algorithme echter altijd minder

rekentijd vergen. Zo zien we in de onderstaande tabel dat voor voldoend grote n geldt $n^3 > 10\ n^2 > 100\ n \log n$.

n	n^3	$10\ n^2$	$100\ n \log n$
1	1	10	0
5	125	250	1 161
10	1 000	1 000	3 322
20	8 000	4 000	8 644
50	125 000	25 000	28 219
100	1 000 000	100 000	66 439

2. Enkele voorbeelden

2.1. Vermenigvuldiging

Laten a en b natuurlijke getallen zijn, binair voorgesteld bestaande uit n bits.

We willen nu het produkt a * b berekenen door behalve optellen en aftrekken alleen gebruik te maken van vermenigvuldigen met twee en delen door twee.

Voor de algorithme kiezen we invariant $x * y + z = a * b$.

algorithme

```
x,y,z := a,b,0;
do y ≠ 0 → do even(y) → y,x := y/2, 2 * x od;
           y,z := y - 1, z + x
od
```

Het aantal operaties in deze algorithme is $O(\log b) = O(n)$.

Optellen van twee n-bits integers kost $O(n)$ operaties, evenals deling door twee of vermenigvuldiging met twee.

De algorithme is dus $O(n^2)$.

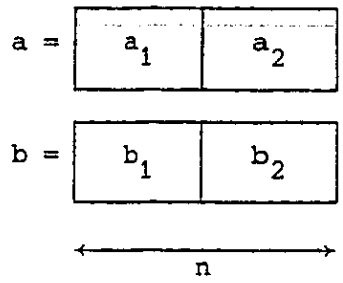
We kunnen a * b ook op een andere manier (wel met dezelfde basisoperaties) bepalen.

Laat n een tweemacht zijn, dan schrijven we a en b als

$$a = a_1 * 2^{n/2} + a_2$$

$$b = b_1 * 2^{n/2} + b_2$$

Binair voorgesteld:



Nu is $a * b = a_1b_1 2^n + (a_1b_2 + a_2b_1) 2^{n/2} + a_2b_2$

$= (a_1 + a_2)(b_1 + b_2) 2^{n/2} - a_1b_1 2^{n/2} - a_2b_2 2^{n/2} + a_1b_1 2^n + a_2b_2$

Laat $u = (a_1 + a_2)(b_1 + b_2)$

$v = a_1b_1$

$w = a_2b_2$

Dan is $a * b = (u - v - w) 2^{n/2} + v2^n + w$.

Als T(n) het aantal operaties is om 2 n-bits getallen door splitsing met elkaar te vermenigvuldigen, dan geldt:

$$T(n) = 3\ T(\frac{n}{2}) + kn$$

$$T(1) = k\ .$$

Hieruit volgt $T(n) = 2k\ n^{\log 3} - 2kn$. Dit is $O(n^{\log 3})$ en dus efficiënter dan de eerste methode.

2.2. Het n-de getal van Fibonacci

De rij van Fibonacci wordt gegeven door

$$F_0 = 0,\ F_1 = 1,\ F_i = F_{i-1} + F_{i-2} \quad \text{voor} \quad i \geq 2\ .$$

Gevraagd wordt om F_n te berekenen.

We kunnen dit doen met de volgende, recursieve, algorithme:

algorithme

```
int proc f(i):
    if i ≤ 1 → f := i
    □  i > 1 → f := f(i - 1) + f(i - 2)
    fi
corp
```

De procedure wordt aangeroepen met f(n).
Als N_i het aantal aanroepen van f is om F_i te bepalen dan geldt:

$$N_0 = 1$$
$$N_1 = 1$$
$$N_i = 1 + N_{i-1} + N_{i-2} \quad \text{voor} \quad i \geq 2 .$$

Hieruit volgt: $N_n = 2F_{n+1} - 1$, dit is ongeveer $O(1,6^n)$.
Dit betekent dat het aantal aanroepen van f om F_n te bepalen exponentieel is in n.
Het bepalen van F_n kan echter in n stappen door gewoon de rij te genereren volgens de definitie. Dit is duidelijk efficienter dan met behulp van de procedure.

Opmerking. F_n kan in log n stappen bepaald worden met behulp van de betrekking

$$F_{2j} = F_j(2F_{j+1} - F_j)$$
$$F_{2j+1} = F_j^2 + F_{j+1}^2 .$$

2.3. Produkt van een rij matrices [3]

De vermenigvuldiging van een p × q-matrix met een q × r-matrix kost pqr vermenigvuldigingen. Omdat matrixvermenigvuldiging associatief is, hangt het totaal aantal vermenigvuldigingen bij het berekenen van het produkt van een aantal matrices, $M_0 * M_1 *...* M_{n-1}$, af van de volgorde waarin dit produkt is uitgerekend.
We willen een programma maken, dat het minimum aantal vermenigvuldigingen bepaalt om $M_0 * M_1 *...* M_{n-1}$ uit te rekenen.

Gegeven is een array r(0 : n). Het aantal rijen van M_i is gelijk aan r(i) en het aantal kolommen van M_i is gelijk aan r(i + 1).

definitie:

a_{ij} = het minimum aantal vermenigvuldigingen om $M_i * M_{i+1} *...* M_j$ uit te rekenen.

Gevraagd wordt dan $a_{0,n-1}$.

We kunnen $M_i * M_{i+1} *...* M_j$ uitrekenen als produkt van twee matrices:

$$(M_i * M_{i+1} *...* M_{k-1}) * (M_k * M_{k+1} *...* M_j) \qquad i < k \leq j .$$

Het minimum aantal vermenigvuldigingen om deze 2 matrices uit te rekenen is resp. $a_{i,k-1}$ en $a_{k,j}$.
Het zijn een r(i) * r(k) matrix en een r(k) * r(j + 1) matrix.
Het produkt van deze twee matrices vergt zelf r(i) * r(k) * r(j + 1) vermenigvuldigen, zodat geldt:

$$a_{ij} = \min(k : i < k \leq j : a_{i,k-1} + a_{k,j} + r(i) * r(k) * r(j + 1))$$

$$a_{ii} = 0 .$$

We kunnen nu de volgende procedure maken

```
int proc a(i,j):
    if i = j → a := 0
    [] i ≠ j → a := min(k : i < k ≤ j : a(i,k-1) + a(k,j) + r(i) * r(k) * r(j + 1))
    fi
corp
```

Hoe vaak procedure a wordt uitgevoerd bij een aanroep a(i,j) hangt af van j - i.
Laat h = j - i en N(h) = aantal maal dat proc a wordt uitgevoerd.

$$N(0) = 1$$

$$N(h) = 1 + 2 \sum_{i=0}^{h-1} N(i)$$

$$N(h+1) = 1 + 2 \sum_{i=0}^{h} N(i) = 1 + 2N(h) + 2 \sum_{i=0}^{h-1} N(i) = 3N(h)$$

zodat $N(h) = 3^h$.

Dit betekent dat bij de aanroep a(0,n - 1) de procedure a 3^n maal wordt uitgevoerd.

In zulke gevallen is het beter om a_{ij} te berekenen met behulp van "dynamic programming", d.w.z. we bepalen a_{ij} <u>eerst dan</u> wanneer <u>alle</u> $a_{i,k-1}$ en $a_{k,j}$ $(i < k \leq j)$ berekend en geadministreerd zijn.

We veronderstellen dat een array a(0 : n - 1, 0 : n - 1) gedeclareerd is en we vullen het array met behulp van de volgende algorithme:

```
      i := 0;
L1:   do i ≠ n → a:(i,i) = 0; i := i + 1 od;
      ℓ := 1;
L2:   do ℓ ≠ n → i := 0;
L3:              do i + ℓ ≠ n → j := i + ℓ; k := i; min := inf;
L4:                 do k ≠ j → k := k + 1;
L5:                             do min > a(i,k-1) + a(k,j) + r(i) * r(k) * r(j+1)
                                → min := a(i,k-1) + a(k,j) + r(i) * r(k) * r(j+1)
                                od
                    od;
                    a:(i,j) = min; i := i + 1
                 od;
        ℓ := ℓ + 1
      od
```

L1 is O(n), L5 wordt 0 of 1 maal uitgevoerd, L4 is dan O(n), L3 is O(n * L4) = $O(n^2)$, L2 is O(n * L3) = $O(n^3)$.
De algorithme is dus $O(n^3)$ en dus veel efficiënter dan de bovenstaande procedure.

3. Sorteren

3.1. Radix sort

Het betreft hier sorteertechnieken die kunnen worden toegepast wanneer de te sorteren objecten uit een begrensde verzameling komen. Als de grootte van die verzameling van dezelfde orde is als het aantal te sorteren objecten wordt de rekentijd lineair in de lengte van de input string.

Een eenvoudig voorbeeld is bucket sort. Gevraagd wordt een rij getallen $a_1,\ldots,a_n$, met $a_i \in \{0,1,\ldots,m-1\}$, naar toenemende grootte te sorteren:

1. Initialiseer m lege buckets, één voor elk element uit {0,1,...,m - 1};
2. Voor alle i: $1 \le i \le n$: plaats a_i in de a_i-de bucket;
3. Concateneer de m buckets: de inhoud van bucket i + 1 wordt achter de inhoud van bucket i geplaatst en zo wordt de gesorteerde rij verkregen.

Stap 1 en stap 3 zijn O(m); stap 2 is O(n).
Als m = O(n) dan is het sorteren O(n).

In het eenvoudige geval dat we slechts getallen sorteren kunnen de buckets worden gerepresenterd door een integer array, waarin het aantal maal dat iedere waarde voorkomt geregistreerd wordt ("turven"). Een dergelijke eenvoudige representatie is niet meer mogelijk als de te sorteren objecten samengesteld zijn en de sortering plaats moet vinden op grond van een bepaalde component in de objecten, zoals blijkt uit de volgende voorbeelden.

We definiëren de lexicografische volgorde van twee strings, s en t, met willekeurige lengte.

definitie:

Als s leeg is, dan is $s \leq t$ voor alle t.
Als s en t beide niet leeg zijn, dan
$s \leq t$ als 1) $first(s) < first(t)$
of 2) $first(s) = first(t) \wedge rest(s) \leq rest(t)$

waarbij first(i) het eerste element is van string i en rest(i) de string is die we krijgen door van string i first(i) weg te laten (analoog de volgorde in een woordenboek, dus 100 < 15).

3.1.1. Sorteren van strings van gelijke lengte

gegeven: n k-tuples, tuple 0 t/m n - 1. De elementen van de tuples komen uit {0,1,...,m - 1} ,
de tuples worden gegeven door een array d(0 : n - 1, 0 : k - 1)
zo dat tuple i = (d(i,0), d(i,1),...,d(i,k - 1)).
gevraagd: de lexicografische volgorde van de tuples.

Als invariant kiezen we:

De volgorde van de tuples is bepaald op grond van de laatste k - j elementen.
Deze volgorde is vastgelegd in een array q en een integer qq:
tuple qq is de eerste in de volgorde,
tuple i ($0 \leq i \leq n - 1$) - heeft als $q(i) \neq n$ tuple q(i) als opvolger
- is als $q(i) = n$ de laatste.

algorithme

```
q := (0); do q.dom ≠ n → q:hiext(q.dom + 1) od;
qq := 0; j := k;
do j ≠ 0 → j := j - 1;
          1. maak m lege buckets;
          2. vul de buckets op grond van het j-de element;
          3. concateneer de gevulde buckets
od
```

1. maak m lege buckets

Hiervoor declareren we een array b(0 : m - 1), een array e(0 : m - 1) en een integer nb,
als b(i) = n dan is bucket i leeg,
als b(i) ≠ n dan is tuple b(i) de eerste in bucket i,
e(i) = n ≡ b(i) = n,
als e(i) ≠ n dan is tuple e(i) de laatste in bucket i en q(e(i)) = n,
het aantal gevulde buckets is gelijk aan nb.

algorithme (1)

```
b,e := (0), (0); do b.dom ≠ m → b:hiext(n); e:hiext(n) od;
nb := 0
```

2. vul de buckets op grond van het j-de element

Tuple t wordt toegevoegd aan bucket d(t,j).
Tuples met gelijk j-de element komen dus in dezelfde bucket terecht. Binnen de bucket houden we zulke tuples gesorteerd (de reeds vastgestelde volgorde op grond van hun "staartstukken"). We bereiken dit door de tuples te inspecteren in de reeds vastgestelde volgorde en ze bij toevoeging aan een bucket achteraan te plaatsen. Voor het representeren van de volgorde binnen een bucket gebruiken we weer het array q.

algorithme (2)

```
do qq ≠ n → t := qq; qq := q(qq); u := d(t,j);
            if b(u) = n → b:(u) := t; nb := nb + 1
            [] b(u) ≠ n → q:(e(u)) = t
            fi;
            q:(t) = n; e:(u) = t
od
```

3. concateneer de gevulde buckets

Laat bucket i de opvolger zijn van bucket ii, beide niet leeg. Om deze twee buckets te concateneren moeten we b(i) de opvolger maken van e(ii), m.a.w. q:(e(ii)) = b(i).

algorithme (3)

```
i := 0; do b(i) = n → i := i + 1 od;
qq := b(i); nc := nb - 1;
do nc ≠ 0 → ii := i; i := i + 1;
            do b(i) = n → i := i + 1 od;
            q:(e(ii)) = b(i);
            nc := nc - 1
od
```

algorithme (1) is O(m),
algorithme (2) is O(n),
algorithme (3) is O(m).

De sorteeralgorithme is dus $O(n + k(2m + n)) = O(k(m + n))$. Als $m = O(n)$, dan is de algorithme lineair in de lengte van de inputtekst.

3.1.2. Sorteren van strings van ongelijke lengte

De strings worden vastgelegd door d(i,j) en $\ell(i)$, $\ell(i)$ is de lengte van string i, $\ell max = max(i : 0 \leq i < n : \ell(i))$, $\ell tot = \sum_{i=0}^{n-1} \ell(i)$ en $d(i,j) \in \{0,1,\ldots,m-1\}$ voor $0 \leq i < n$, $0 \leq j < \ell(i)$.

Voor dit probleem kunnen we gebruik maken van de voorgaande algorithme, door de strings te verlengen met dummy-elementen tot de strings alle de lengte ℓmax hebben. De algorithme zou dan O(ℓmax(m + n)) worden.
We proberen met behulp van twee maatregelen een algorithme te maken die efficiënter is.

1. Bij het bepalen van de volgorde op grond van het j-de element beschouwen we alleen de strings met een lengte groter of gelijk aan j.
2. We laten in de j-de slag alleen die buckets meedoen, die in de j-de slag echt gevuld worden.

Globaal ziet de sorteeralgorithme er dan als volgt uit:

```
0. maak de q-rij en de buckets leeg;
j := ℓmax;
do j ≠ 0 → 1. voeg strings met lengte j aan de q-rij toe;
             j := j - 1;
           2. vul de buckets met strings uit de q-rij o.g.v.
             d(i,j);
           3. concateneer de gevulde buckets
od
```

Deze algorithme is O(ℓtot + m), zodat als m = O(ℓtot) de algorithme O(ℓtot) is. De afleiding van deze ordegrootte laten we zien bij de uitwerking van de punten 1,2 en 3.

Het is duidelijk dat deze algorithme "pre-processing" vereist, d.w.z. een aantal algorithmen als voorspel vereist, die ons de beschikking geven over de informatie die nodig is om de sorteeralgorithme zo te kunnen toepassen. Voor 1. moet bekend zijn welke strings lengte j hebben, en die strings moeten direct aan de q-rij toegevoegd kunnen worden.
Voor 3. moet bekend zijn welke buckets we precies nodig hebben. De algorithmen in het "pre-processing"-deel mogen natuurlijk niet O(ℓtot + m) overschrijven.

Het "pre-processing"-deel

a) Gedeclareerd wordt een array ℓn(1 : ℓmax) en een array ℓq(0 : n - 1), waarin we de gegevens opslaan van ℓmax verzamelingen van strings.
Als ℓn(i) = n dan is er geen string van lengte i.
Als ℓn(i) ≠ n dan heeft string ℓn(i) lengte i.
Als string j lengte i heeft en ℓq(j) ≠ n dan heeft string ℓq(j) ook lengte i.

algorithme

```
      ℓn := (1);
L1:   do ℓn.dom ≠ ℓmax → ℓn:hiext(n) od;
      i := 0; ℓq := (0);
L2:   do i ≠ n → ℓq:hiext(ℓn(ℓ(i)));
                 ℓn:(ℓ(i)) = i;
                 i := i + 1
      od
```

L1 is O(ℓmax), L2 is O(n), dus de algorithme is zeker O(ℓtot).

b) Gedeclareerd wordt een array f(0:ℓtot - 1, 0 : 1), dat gevuld wordt met ℓtot paren (j,d(i,j)), (0 ≤ i < n , 0 ≤ j < ℓ(i)).

algorithme

```
i,k := 0,0;
do i ≠ n → j := 0;
           do j ≠ ℓ(i) → f:(k,0) = j; f:(k,1) = d(i,j);
                         j := j + 1; k := k + 1
           od;
   i := i + 1
od
```

Deze algorithme is O(ℓtot).

Voer nu een dalende radixsort uit op de paren (f(i,0),f(i,1)), $0 \le i < \ell tot$.
De volgorde wordt vastgelegd in een array p(0:ℓtot - 1), de "p-rij" en een integer pp:
p(i) is de opvolger van i, pp is het eerste paar en als p(j) = ℓtot dan is j het laatste paar.
Hiervoor gebruiken we de algorithme voor het sorteren van strings van gelijke lengte. De ordegrootte is O(2(ℓtot + m)) = O(ℓtot + m). In het array f kunnen gelijke paren voorkomen. Gesorteerd staan deze achter elkaar. We willen van gelijke paren er maar één beschouwen. Daartoe wordt de p-rij uitgedund.

algorithme

```
t := pp;
do p(t) ≠ ℓtot → u := p(t);
                  if f(t,0) = f(u,0) ∧ f(t,1) = f(u,1) → p:(t) = p(u)
                  ▯ f(t,0) ≠ f(u,0) ∨ f(t,1) ≠ f(u,1) → t := u
                  fi
od
```

De p-rij wordt eenmaal doorlopen, dus deze algorithme is O(ℓtot).

Na dit voorbereidende werk komt de eigenlijke sorteeralgorithme.

0. maak de q-rij en de buckets leeg

```
b,e := (0),(0);
do b.dom ≠ m → b:hiext(n); e:hiext(n) od;
qq := n
```

1. voeg strings met lengte j aan de q-rij toe:
   Omdat de lege string in lexicografische volgorde vóór iedere andere string komt, worden deze strings aan het begin van de q-rij toegevoegd.

algorithme

```
    t := ℓn(j);
L1: do t ≠ n → q:(t) = qq; qq := t; t := ℓq(t) od
```

2. vul de buckets met strings uit de q-rij o.g.v. d(i,j):
   Dit behoeft geen verder commentaar; het gaat analoog aan het vorige programma. Slechts die strings die in de q-rij zitten worden over de buckets verdeeld.

3. concateneer de gevulde buckets:
   De buckets die echt gevuld worden, kunnen we vinden met behulp van de p-rij. Wanneer bucket i achter bucket ii is gevoegd (q:(e(ii)) = b(i)) wordt bucket i weer leeg gemaakt.

algorithme

```
        i := f(pp,1); pp := p(pp);
L3:     do pp ≠ ℓtot ∆ f(pp,0) = j → ii := f(pp,1); q:(e(ii)) = b(i);
                                     e:(i) = n; b:(i) = n;
                                     i := ii; pp := p(pp)
        od;
        qq := b(i); e:(i) = n; b:(i) = n
```

Deel 0 "maak de q-rij en de buckets leeg" is O(m).
In deel 1 worden strings met lengt j aan de q-rij toegevoegd. Elke string wordt éénmaal toegevoegd, dus L1 wordt in totaal n maal doorlopen.
In deel 2 wordt voor elke i en j $0 \leq i < n$, $0 \leq j < \ell(i)$ een string aan een bucket toegevoegd. Dus wordt in totaal ℓtot maal een string aan een bucket toegevoegd.
In deel 3 beschouwen we alleen buckets die echt gevuld worden. pp doorloopt een deel van de p-rij en neemt iedere keer een andere waarde aan, dus L3 wordt ten hoogste ℓtot, de maximale lengte van de p-rij, maal doorlopen.
Omdat $n \leq \ell tot$ is dus de sorteeralgoritheme O(ℓtot + m).

Opmerking. De algorithme voor het sorteren van strings van gelijke lengte is O(k(n + m)). Door in deze algorithme ook gebruik te maken van een p-rij en voor het vullen van de buckets in een slag alléén die buckets leeg te maken, kunnen we de ordegrootte van de algorithme terugbrengen tot O(kn) met een geheugen gebruik van O(m).

Deze algorithme kunnen we nu ook gebruiken in de pre-processing, die dan O(ℓtot) wordt. Ook de sorteeralgorithme zelf kan in O(ℓtot) door niet meer alle buckets leeg te maken, maar in plaats daarvan aan het begin van deel 2 de vereiste buckets leeg te maken. Dit kan dan weer in deel 3 vervallen. O(m) vinden we zo niet meer terug in de algorithmen maar wel in het geheugengebruik. (Dit onder de aanname dat het declareren van een array een rekentijd vergt die onafhankelijk is van de domeingrootte.)

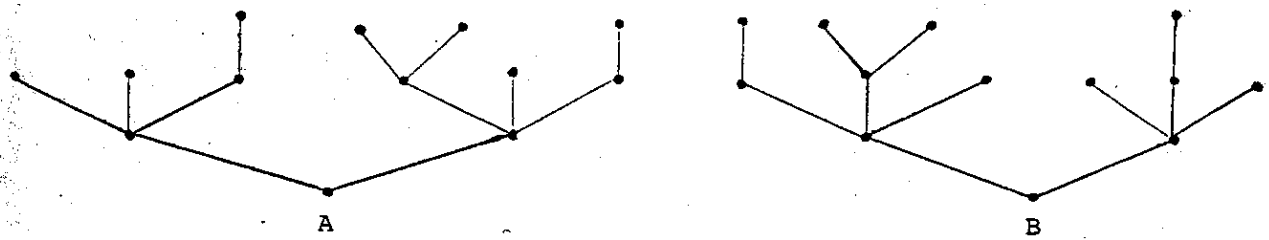
3.1.3. Isomorfie van wortelbomen

We kunnen gebruik maken van de sorteeralgorithme van strings van ongelijke lengte bij het bepalen van de isomorfie van twee wortelbomen. Twee wortelbomen heten isomorf wanneer de ene uit de andere verkregen kan worden door het permuteren van deelbomen van knopen.
We willen een programma maken dat de eventuele isomorfie van twee bomen vaststelt in O(n), waarbij n het aantal knopen van iedere boom is.
In de algorithme worden de knopen van de bomen als volgt gelabeld:

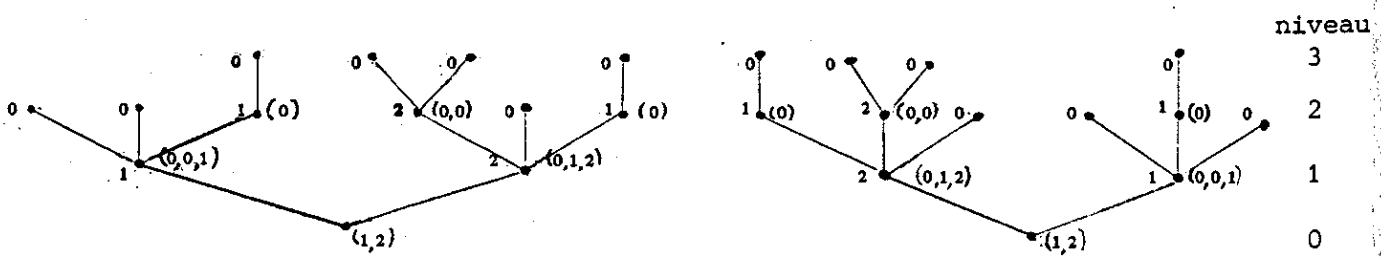
1. de bladen van de bomen krijgen nummer 0,
2. de interne knopen krijgen een geordend tuple, met als elementen de nummers van de zonen (deelbomen),
3. per niveau krijgt ieder verschillend tuple een nummer; de lexicografisch kleinste tuples krijgen nummer 1, etc.

Deze labeling is invariant onder permutaties van deelbomen van dezelfde knoop. De bomen zijn dus slechts dan isomorf, wanneer op elk niveau dezelfde nummers zijn uitgedeeld.

Voorbeeld



Het labelen van de knopen en de bladen gebeurt als volgt:
De tuples staan rechts van de knopen; de nummers links.



A

Op elk niveau zijn dezelfde nummers en tuples uitgedeeld, dus de bomen zijn isomorf.

Omdat deze algorithme slechts een voorbeeld is van de toepassing van de sorteringsalgorithme voor strings, geven we allen de globale opbouw van de algorithme.

Laten S_1 en S_2 geordende rijen tuples op niveau ℓ van de bomen A en B zijn.
Laat v_ℓ het aantal knopen op niveau ℓ zijn.
Bepaal voor ieder niveau de verzameling knopen van dat niveau en geef meteen de bladen nummer 0.
Op het hoogste niveau (topniveau) bevinden zich geen tuples, maar alleen bladen.

algorithme

ℓ := topniveau; S1, S2 := $\emptyset$,$\emptyset$;
do S1 = S2 $\wedge$ $\ell \neq 0 \rightarrow$ ℓ := ℓ - 1;

1. geef voor ieder volgend verschillend tuple in S1 een nummer aan de betreffende knooppunten; Idem voor S2;
2. L1 := lijst van knopen op niveau ℓ + 1 van boom A gesorteerd naar nummer; L2 := idem voor boom B;
3. voor iedere knoop in L1 : voeg een nummer toe aan het initieel lege tuple van de vader op niveau ℓ; idem voor L2;
4. S1 := de lexicografisch gesorteerde tuples op niveau ℓ; idem voor S2.

od;
isomorf := (S1 = S2)

De stappen 1,2 en 3 zijn $O(v_{\ell+1})$. Stap 4 is $O(\ell tot + m) = O(v_{\ell+1})$, zodat de orde van deze algorithme gelijk is aan

$$O\left(\sum_{\ell=0}^{topniveau-1} v_{\ell+1}\right) = O(n) \ .$$

3.2. Heap sort

In sectie 3.1 was de sorteertijd van de orde van de lengte van de input wanneer m ook van die orde is. Bij het sorteren van een rij natuurlijke getallen a_0 t/m a_{n-1} kunnen we m gelijk aan het maximum van die rij kiezen. Voor het efficiënt toepassen van bucketsort zal dan het maximum van die rij van dezelfde ordegrootte moeten zijn als het aantal elementen van die rij. Wanneer hier niet aan voldaan is zouden we m kleiner kunnen maken door bijv. de elementen binair voor te stellen (m is dan 2) en ze als strings te beschouwen. De lengte van de invoer is dan $\Sigma(\log a_i) = \log(\Pi a_i)$. De sorteertijd hangt nu niet meer af van het aantal getallen dat we sorteren, maar van de grootte van die getallen. We kunnen de getallen dan beter sorteren door ze met elkaar te gaan vergelijken ($<,\leq,=,\geq,>$).

Een rij met lengte n heeft n! verschillende permutaties. Ieder van deze permutaties kan de gevraage volgorde zijn. Om aan te geven om welke permutatie het gaat zijn $\log(n!)$ bits nodig.
Iedere vergelijking ($<,\leq,=,\geq,>$) die het programma uitvoert levert ten hoogste één bit informatie op, zodat er dus tenminste $\log(n!)$ vergelijkingen nodig zijn. Stirling approximatie geeft: $\log(n!) \approx \log(\sqrt{2\pi n}\ (\frac{n}{e})^n) = n \log n + \log \sqrt{2\pi n} - n \log e$, zodat het sorteren tenminste $O(n \log n)$ vergelijkingen vergt.

Een sorteeralgorithme die een rij sorteert in $O(n \log n)$ is Heap sort [4] . De te sorteren getallen bevinden zich in een array a(1 : n). De enige operatie die uitgevoerd wordt is het verwisselen van twee elementen (swaps), zodat de rij een permutatie van de oorspronkelijke rij blijft.
Het is niet voldoende om alleen swaps tussen buren uit te voeren, want het aantal inversies kan $\binom{n}{2} = \frac{1}{2} n(n-1)$ zijn en iedere swap tussen buren doet het aantal inversies met één dalen.
In plaats daarvan vergelijken we a(p) en a(2p), $1 \leq p \leq n/2$, en verwisselen a(p) en a(2p) wanneer a(p) > a(2p). We vergelijken a(p) ook met a(2p + 1), $1 \leq p < n/2$, en verwisselen a(p) en a(2p + 1) wanneer a(p) > a(2p + 1).

We definiëren nu de heapvoorwaarde:

De rij a(1), a(2),...,a(k) voldoet aan de heapvoorwaarde als

$$\forall p,\ 1 \le p \le k/2 \quad \text{geldt} \quad a(p) \le a(2p)$$

en

$$\forall p,\ 1 \le p < k/2 \quad \text{geldt} \quad a(p) \le a(2p+1)\ .$$

Zo is ieder punt a(p) de wortel van een binaire boom, tree(p), en a(p) is het minimum van tree(p); a(1) is dus het minimum van de rij.
Voor ieder punt p in de boom geldt dat op het wortelpad van p (het pad van p naar de wortel) geen grotere waarde dan a(p) voorkomt.
Als het hele array a tot een heap gemaakt is, gebeurt het sorteren op de volgende manier: zet het minimum achteraan (door een swap) en maak weer een heap van de overige elementen; herhaal dit tot de rij geheel gesorteerd is. (De rij wordt dus naar afnemende grootte gesorteerd).

De algorithme bestaat uit twee stappen:

1. constructie van de heap a(1 : n),
2. opbouw van de gesorteerde rij.

1. Constructie van de heap a(1 : n).

Invariant P: a(1 : k) voldoet aan de heapvoorwaarde.

algorithme

```
k := 1;
do k ≠ n → k := k + 1; herstel P od
```

herstel P:

Invariant Q: er is ten hoogste één element, a(p), uit de rij a(1),...,a(k), initieel a(k), dat grotere elementen op zijn wortelpad kan hebben.

We zorgen ervoor dat gedurende de algorithme "herstel P" steeds aan Q is voldaan. Uit (p = 1 of a(p div 2) ≤ a(p)) ∧ Q volgt dat a(1),a(2),...,a(k) aan de heapvoorwaarde voldoet.

De voorganger van a(p) is a(p div 2); a(p div 2) heeft naast a(p) ook nog een a(qq) als opvolger. Uit Q volgt dat a(p div 2) ≤ a(qq). Als a(p div 2) > a(p) verwisselen we a(p) met a(p div 2).

Nu geldt: a(p div 2) ≤ a(p) en a(p div 2) ≤ a(qq), a(p div 2) is dus het minimum van tree(p div 2). Het enige element uit de rij a(1),a(2),...,a(k) dat grotere elementen op zijn wortelpad kan hebben is a(p div 2), zodat met p := p div 2 weer aan Q is voldaan.

algorithme

```
p,q := k, k div 2;
do p ≠ 1 ∆ a(q) > a(p) → a:swap(p,q); p := q;
                         q := p div 2
od
```

Deze algorithme is O(log k) = O(log n). De algorithme wordt n maal uitgevoerd zodat de ordegrootte van de constructie van de heap a(1 : n) gelijk is aan O(n log n).

2. Opbouw van de gesorteerde rij:
a(1 : n) vormt nu een heap en de rij wordt gesorteerd in monotoon niet-stijgende volgorde.

Invariant P: a(1 : k) is een heap en a(k + 1 : n) bevat in niet-stijgende volgorde de n - k kleinste elementen van de rij.

algorithme

```
k := n;
do k ≠ 1 → a:swap(1,k); k := k - 1;
           maak a(1 : k) tot een heap
od
```

maak a(1 : k) tot een heap.

Invariant Q: er is ten hoogste één element, a(p), in a(1 : k), initieel a(1), dat niet het minimum van tree(p) is.

Als a(p) twee opvolgers heeft en a(1 : k) vormt geen heap, dan wordt a(p) verwisseld met de kleinste van zijn twee opvolgers. Er is dan weer aan Q voldaan.
Als a(p) geen opvolgers meer heeft dan volgt uit Q dat a(1 : k) een heap vormt.

Als a(p) één opvolger heeft (dan is p = ½k) en a(p) > a(k) dan worden deze twee verwisseld.

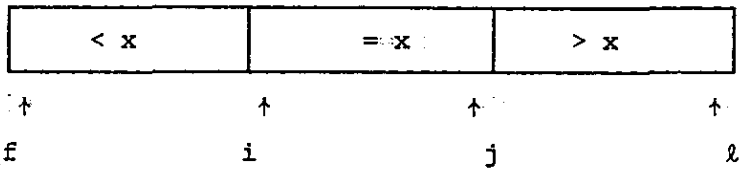
algorithme

```
p,q := 1,2; {q = 2 * p}
do q < k ∆ (a(q) ≤ a(q + 1) ∧ a(p) > a(q)) →
                                    a:swap(p,q); p := q; q := 2 * p
[] q < k ∆ (a(q + 1) ≤ a(q) ∧ a(p) > a(q + 1)) →
                                    a:swap(p,q + 1); p := q + 1; q := 2*p
od
{als q = k dan heeft a(p) maar één opvolger};
do q = k ∆ a(p) > a(q) → a: swap(p,q); p := q; q := 2 * p od
```

De algorithme "maak a(1 : k) tot een heap" is O(log k) = O(log n) en wordt n maal uitgevoerd. De bovenstaande algorithme is dus O(n log n). De algorithme heap sort is dus O(n log n).

3.3. Find

Gegeven is een rij getallen a(1) t/m a(n).
We willen een algorithme maken, die bepaalt welk getal na sorteren op de k^e plaats komt, $1 \leq k \leq n$. De hele rij sorteren kan in O(n log n).
Alle getallen moeten worden bekeken, dus de algorithme is tenminste O(n).
We proberen een algorithme te maken, die O(n) is. De algorithme zal het array zó permuteren dat na afloop a(k) het gevraagde getal is.
Laat x een willekeurige waarde uit a(f : ℓ) zijn, bijv. a(k), en verdeel het array in drie stukken:



wanneer f ≤ k < i dan bevindt de gevraagde waarde zich in a(f,i - 1),
wanneer j < k ≤ ℓ dan bevindt de gevraagde waarde zich in a(j + 1,ℓ),
wanneer i ≤ k ≤ j dan heeft a(k) de gevraagde waarde.

Invariant P: $(1 \le f \le k \le \ell \le n) \wedge (a(p) \le a(q) \le a(r))$, voor alle p,q,r
met $1 \le p < f$, $f \le q \le \ell$ en $\ell < r \le n$.

Wanneer $f = \ell$ dan volgt uit P: $k = f = \ell$ en $a(p) \le a(k) \le a(r)$ voor $1 \le p < k$ en $k < r \le n$

en dan heeft a(k) dus de gevraage waarde.

algorithme [5]

```
f,ℓ := 1,n;
do f ≠ ℓ → x := a(k);
          i,h,j := f,f,ℓ;
          do h ≤ j → if a(h) < x → a:swap(i,h); i,h := i + i, h + 1
                      [] a(h) = x → h := h + 1
                      [] a(h) ≥ x → a:swap(j,h); j := j - 1
                      fi
          od {f ≤ i ≤ ℓ ∧ f ≤ j ≤ ℓ};
          if k < i  → {f ≤ k < i}ℓ := i - 1
          [] k > j → {j < k ≤ ℓ} f := j + 1
          [] i ≤ k ≤ j → f,ℓ := k,k
          fi
od
```

Deze algorithme is worst-case $O(n^2)$ en dus duidelijk niet de gezochte algorithme. Het probleem zit in de keuze van x. Het meest effieciënt zou zijn voor x de mediaan van a(f) t/m a(ℓ) te kiezen. Het bepalen van de mediaan is echter het probleem net, dat we op moeten lossen, zij het met een speciale keuze voor k.
We stellen ons tevreden met een x zodat tenminste ¼ van de rij groter of gelijk is aan x én tenminste ¼ van de rij kleiner of gelijk is aan x.
In elke slag wordt de verzameling getallen waar de gevraagde waarde in kan liggen met minstens ¼ kleiner en dan wordt, zoals we zullen zien, de bovenstaande algorithme O(n).
Zo'n x krijgen we als volgt:
Beschouw het array a(f) t/m a(ℓ). Dat array wordt verdeeld in stukjes ter lengte p (p is oneven). Van elk stuk bepalen we de mediaan.
Van de verzameling medianen bepalen we de mediaan en x krijgt de waarde van die mediaan. De helft van de verzameling medianen is groter of gelijk aan x en wanneer we x vergelijken met de stukjes waar die medianen uitkomen is dus

tenminste ¼ van het array a(f) t/m a(ℓ) groter of gelijk aan x, analoog is minstens ¼ van het array a(f) t/m a(ℓ) kleiner of gelijk aan x.

Laat T(n) = de rekentijd die nodig is om het getal te bepalen dat na sorteren op de k^e plaats komt in een array ter lengte n (worst case).

Het bepalen van een mediaan van een stukje ter lengte p, hangt niet af van n, maar uitsluitend van de keuze van de constante p. We bepalen $\frac{n}{p}$ van dergelijke medianen, dus het bepalen van de verzameling medianen is O(n). Het vaststellen van de mediaan van die verzameling heeft aan rekentijd $T(\frac{n}{p})$ nodig. Het partitioneren rond x is O(n). De rest van de berekening is worst case $T(\frac{3}{4}n)$.

Dus $$T(n) \leq T(\tfrac{n}{p}) + T(\tfrac{3}{4}n) + c \cdot n \ .$$

We kiezen p nu zo dat de vergelijking een lineaire oplossing heeft; p = 3 voldoet niet, p = 5 wel:

$$T(n) \leq \tfrac{1}{5}T(n) + \tfrac{3}{4}T(n) + cn \quad \text{impliceert } T(n) \leq 20\,cn, \text{ dus } T(n) = O(n) \ .$$

Als ℓ - f < 5 kunnen we het array niet opdelen in stukjes ter lengte 5, dan zullen we het array a(f),...,a(ℓ) sorteren. Dit heeft natuurlijk geen invloed op de ordegrootte.
We krijgen dan de volgende algorithme [6].

```
proc sel(k,f,ℓ):
    do ℓ - f ≥ 4 → d := (ℓ - f + 1) div 5;
                   i := 0;
                   do i ≠ d → do a(f + i) > a(f + d + i) → a:swap(f + i,f + d + i)
                               [] a(f + d + i) > a(f + 2d + i) → a:swap(f + d + i,f + 2d + i)
                               [] a(f + 2d + i) > a(f + 3d + i) → a:swap(f + 2d + i,f + 3d + i)
                               [] a(f + 3d + i) > a(f + 4d + i) → a:swap(f + 3d + i,f + 4d + i)
                               od;
                      i := i + 1
                   od {de verzameling medianen is nu a(f + 2d : f + 3d - 1};
                   m := (2f + 5d - 1) div 2;
                   sel(m,f + 2d,f + 3d - 1);
                   x := a(m);
                   < oude tekst: i,h,j := f,f,ℓ
                                 :
                                 :
                                 fi >
    od;
    i := f;
    do i ≠ ℓ → i := i + 1; j,y := i,a(i);
               do j ≠ f ∧ a(j - 1) > y → a:(j)= a(j - 1); j := j - 1 od;
               a:(j) = y
    od
corp
```

Met sel(k,1,n) krijgt a(k) de gevraagde waarde.

4. Operaties op verzamelingen

Laat S en U verzamelingen $S \subset U$.

We beschouwen 3 operaties op de verzameling S, $b \in U$.

1. member (b): wordt true als $b \in S$ en false als $b \notin S$,
2. insert (b): b wordt aan S toegevoegd ($S := S \cup \{b\}$),
3. delete (b): b wordt uit S verwijderd ($S := S \setminus \{b\}$).

4.1. Hashing

Om redenen van efficiency willen we bij de verzamelingsoperaties niet de hele verzameling S beschouwen, maar één bepaalde deelverzameling van S, waartoe b behoort of waaraan b moet worden toegevoegd. Een mogelijkheid is de toepassing van hashing [7].

Hashing: U wordt verdeeld in m disjuncte deelverzamelingen $U_0,\ldots,U_{m-1}$. Er is een funktie h, de zogenaamde hashfunktie, die aan ieder element van U het nummer van zijn bijbehorende deelverzameling toewijst.

$h : U \rightarrow \{0,1,2,\ldots,m-1\}$.

We veronderstellen dat de berekening van h(b) een vaste rekentijd vergt. In plaats van S representeren we nu m verzamelingen S_i, waarbij $S_i = S \cap U_i$.
Bij een operatie op b wordt eerst m.b.v. h(b) de deelverzameling $S_{h(b)}$ bepaald en we beperken dan de operatie tot die deelverzameling.

Wat is nu een goede keuze voor de hashfunktie h? Neem aan dat $U = \mathbb{N}$. h moet de S_i's zo gelijkmatig mogelijk vullen. Veel gebruikt wordt $h(b) = b \underline{\text{mod}}\ m$. 2^p is <u>geen</u> goede keuze voor m. De deling is dan wel eenvoudig, maar dan worden de p minst significante bits genomen en er wordt dus weinig "door elkaar gegooid". Wanneer we voor m een priemgetal kiezen, is h wel een goede hashfunktie.

Op welke manier kunnen we de S_i's representeren?
We behandelen twee representaties: direct chaining en open addressing.

4.1.1. Direct chaining

Bij direct chaining maken we gebruik van een array t(0 : n - 1), $m \leq t(i) \leq n$ en een array s(m : n).
array t(0 : m - 1) wordt wel de "hashtable" genoemd,
array t(m : n - 1) de "linktable" en
array s(m : n - 1) de "storagetable". Deze bevat elementen uit S;
s(n) heeft een aparte betekenis.

De representatie van de S_i's is dan als volgt:

als t(i) = n dan is S_i leeg
t(i) $\neq$ n dan is s(t(i)) het eerste element van S_i.
als s(k) $\in S_i$ dan is als t(k) $\neq$ n, s(t(k)) het volgende element van S_i
als t(k) = n, dan is er geen volgend element.

We kunnen nu direkt beschikken over alle elementen uit een bepaalde S_i.
Die elementen worden gesorteerd opgeslagen, wat de rekentijd ongeveer met een faktor twee verkort.
We definiëren s(n) = inf, zodat ieder S_i inf als grootste element heeft.

We maken gebruik van een boolean array e(m : n) dat aangeeft welke plaatsen in het array s niet worden gebruikt.
Er geldt: voor $m \leq i < n$ e(i) = "s(i) is vrij",
e(n) heeft altijd waarde true.
r = min(i : $m \leq i \leq n$: e(i))

initialisering:

```
t := (0); do t.dom ≠ m → t:hiext(n) od;
s := (m); e := (m);
do s.hib ≠ n → s:hiext(inf); t:hiext(n); e:hiext(true) od;
r := m
```

member(b):

```
p := t(b mod m);
do s(p) < b → p := t(p) od;
member := (s(p) = b)
```

insert(b):

```
{p = t(q)}
q := b mod m; p := t(q);
do s(p) < b → q := p; p := t(q) od;
if s(p) = b → {b ∈ S} skip
[] s(p) > b → if r = n → overflow {er zijn geen vrije plaatsen meer}
               [] r < n → {b moet worden toegevoegd tussen q en p
                           op plaats r}
                          t:(q) = r; t:(r) = p;
                          s:(r) = b; e:(r) = false;
                          do ¬ e(r) → r := r + 1 od
               fi
fi
```

delete(b):

```
q := b mod m; p := t(q);
do s(p) < b → q := p; p := t(q) od;
if s(p) = b → t:(q) = t(p); e:(p) = true;
               do r > p → r := p od
[]  s(p) > b → skip {b ∉ S}
fi
```

Opmerking. We hebben nog een vrijheid in de keuze van de grootte van m. Wanneer m groot is, is het aantal verzamelingen S_i ook groot en bij een goede hashfunktie, het aantal elementen per S_i klein. Dit betekent dat de tijdsduur voor de procedures member, insert en delete kort is. Daarentegen wordt de gebruikte geheugenruimte groot. Het is een afwegen tussen rekentijd en geheugenruimte.

4.1.2. Open addressing

Bij deze methode wordt het array t, dat nogal veel geheugenruimte kost, niet gebruikt. Omdat de elementen nu niet meer in een "geschakelde lijst" staan, is de operatie delete vrijwel onmogelijk. We declareren het array s(0 : m - 1) en het boolean array e(0 : m - 1). Element b moet door middel van de hashfunktie h(b) op plaats s(h(b)) komen. Dat gebeurt slechts als deze plaats vrij is, d.w.z. als e(h(b)) geldt. Anders moet er een standaardmethode zijn om een andere plaats te kiezen. We proberen achtereenvolgens s((h(b) + g(i)) mod m), met g(0) = 0 voor i = 0,1,2,... . Kies bijv. g(i) = i, lineair probing. Die funktie heeft als nadeel dat de waarden in array s de neiging hebben om in groepjes bij elkaar te komen (clustering), waardoor de operaties langzamer worden. Een betere funktie is $g(i) = i^2$, quadratic probing, een voorbeeld van double hashing. Hierbij bestaat wel het gevaar dat een eventuele vrije plaats niet gevonden wordt. Als m een priemgetal is, zal in ieder geval de helft van de array plaatsen geinspecteerd worden voordat de uitgangspositie weer wordt bereikt.

Immers uit $(h(b) + i^2) \bmod m = (h(b) + j^2) \bmod m \qquad 0 < j - i < m$

volgt

$$i^2 \bmod m = j^2 \bmod m$$
$$(j^2 - i^2) \bmod m = 0$$
$$(j - 1)(j + i) \bmod m = 0$$

Omdat m priem is en $0 < j - i < m$ is $j + i = c.m$ en dus $j \geq \frac{m}{2}$.
De eerste herhaling treedt pas na tenminste $\frac{1}{2}$ m keer op.

We initialiseren de arrays s en e als volgt:

```
s,e := (0),(0);
do s.dom ≠ m → s:hiext(0); e:hiext(true) od
```

Voor de operaties member(b) en insert(b) zullen we quadratic probing toepassing: $g(i) = i^2$.

Invariant geldt

$$P0:\ p = (h(b) + i^2) \bmod m \wedge i \geq 0$$

Ter voorkoming van kwadrateringen voeren we een variabele d in en breiden P0 uit met $d = 2i + 1$. P0 is dan invariant onder

```
p := (p + d) mod m; d := d + 2; i := i + 1
```

De variabele i laten we weg uit het programma. We kiezen h(b) = b mod m.

member(b):

```
p := b mod m; d := 1;
do ¬ e(p) ∧ s(p) ≠ b ∧ d ≠ m → p := (p + d) mod m; d := d + 2 od;
member := (¬ e(p) ∧ s(p) = b)
```

insert(b):

```
p := b mod m; d := 1;
do ¬ e(p) ∧ s(p) ≠ b ∧ d ≠ m → p := (p + d) mod m; d := d + 2 od;
if e(p) → s:(p) = b; e:(p) = false
[] ¬ e(p) → if s(p) = b → skip
            [] s(p) ≠ b → overflow
            fi
fi
```

Nadelen van hashing zijn:

- Van te voren moet bekend zijn uit hoeveel elementen de verzameling bestaat.
- De worst case is erg slecht, nl. $O(n^2)$. Als altijd een snelle respons vereist is, moet deze methode niet toegepast worden.
- Bij open addressing is geen delete mogelijk.

- Er wordt door de arrays heengesprongen; dit is bij machines met een virtueel geheugensysteem inefficiënt, omdat zo'n geheugenorganisatie adressen in elkaars buurt verwacht.
- Doordat de elementen van de verzameling door elkaar gegooid worden is het moeilijk om andere operaties dan member, insert en delete uit te voeren. Denk hierbij bijv. aan "de meest in de buurt liggende waarde".

4.2. Binaire bomen

Wanneer naast member, insert en delete ook de operatie min, de operatie die het minimum bepaalt van een verzameling, op S moet worden toegepast, hebben we geen nut van hashing.
Het minimum van S kan in een willekeurige S_i liggen.

Voor deze vier operaties op S introduceren we een binaire boom.

definitie:

1) de lege boom is een binaire boom
2) als k tot de boom behoort, dan heeft k twee binaire deelbomen: L(k) en R(k)
3) iedere knoop k heeft een waarde $v(k) \in S$.

In de boom brengen we een ordening aan:
Laat k een knoop van de boom zijn, dan geldt

als $k_1 \in L(k)$ dan $v(k_1) < v(k)$
als $k_2 \in R(k)$ dan $v(k_2) > v(k)$.

We spreken dan ook wel van een binaire zoekboom.

definitie tree(k):

voor $k \neq -1$: tree(k) is de deelboom met v(k) als wortel, tree(ℓ(k)) als linkerdeelboom, tree(r(k)) als rechterdeelboom
tree(-1) is de lege boom

gegeven: w(wortel van de boom), arrays ℓ, r en v.
operaties:
member(a): er geldt $a \in S \equiv a \in tree(w)$.
invariant: $a \in tree(w) \equiv a \in tree(t)$.

algorithme

```
t := w;
do t≠-1 ∧ a < v(t) → t := ℓ(t)
[] t≠-1 ∧ a > v(t) → t := r(t)
od {t = -1 ∨ a = v(t)};
member := (t≠-1)
```

min:

invariant: min(tree(w)) ≡ min(tree(t))

we nemen aan dat w ≠ -1.

algorithme

```
t := w; {als ℓ(t) niet leeg is, dan bevindt het minimum zich
        in tree(ℓ(t))}
do ℓ(t) ≠ -1 → t := ℓ(t) od;
min := v(t).
```

Het bepalen van het maximum van S gaat analoog: ℓ wordt vervangen door r.

Bij insert(a) en delete(a) krijgen we te maken met twee problemen:

1. Element a kan alleen als blad aan de boom worden toegevoegd, en ook alleen als blad worden verwijderd.
   Wanneer a als interne knoop in de boom voorkomt, moet op de plaats van a een ander element komen. Als de linkerdeelboom van a leeg is, kunnen we a vervangen door zijn rechteropvolger; bij een lege rechter deelboom door zijn linkeropvolger. Anders door bijv. het maximum van zijn linkerdeelboom. Alle punten, zonder a, vormen nu een binaire boom, dus kunnen we a verwijderen.
2. Laat a een blad van de boom zijn, dan is er een k met v(k) = a.
   Wanneer we nu a uit de boom verwijderen wordt v(k) een "vrije" plaats in het array v. We kunnen deze vrije plaatsten bijhouden (extra array) en bij de procedure insert eerst de vrije plaatsen vullen óf we verwisselen v(k) met v.high en korten v met 1 plaats in zodat in v géén vrije plaatsen voorkomen.

We kiezen voor het tweede alternatief.

insert(a):

invariant: $t = w \nabla ((\ell b \wedge t = \ell(s)) \vee (\neg \ell b \wedge t = r(s)))$

algorithme

```
t := w;
do t ≠ - 1 Δ a < v(t) → t,s,ℓb := ℓ(t),t,true
[] t ≠ - 1 Δ a > v(t) → t,s,ℓb := r(t),t,false
od {t = - 1 ∇ a = v(t)};
if t ≠ - 1 → skip {a = v(t), dus a ∈ S}
[] t = - 1 → v:hiext(a); ℓ:hiext(-1); r:hiext(-1);
              if t ≠ w → w := v.hib
              [] t ≠ w → if ℓb → ℓ:(s) = v.hib
                         [] ¬ ℓb → r:(s) = v.hib
                         fi
              fi
fi
```

delete(a):

invariant: $t = w \nabla (\ell b \wedge t = \ell(s) \vee \neg \ell b \wedge t = r(s))$

algorithme

```
t := w;
do t ≠ - 1 Δ a < v(t) → t,s,ℓb := ℓ(t),t,true
[] t ≠ - 1 Δ a > v(t) → t,s,ℓb := r(t),t,false
od;
if t = - 1 → skip {a ∉ S}
[] t ≠ - 1 → if ℓ(t) = - 1 → "vervang t door r(t)"; t1 := t
              [] r(t) = - 1 → "vervang t door ℓ(t)"; t1 := t
              [] ℓ(t) ≠ - 1 ∧ r(t) ≠ - 1 → "laat v(t1) = max(tree(ℓ(t)))";
                                           "vervang v(t) door v(t1)"
              fi;
              "dicht het gat op plaats t1"
fi
```

```
"vervang t door r(t)":
if t = w → w := r(t)
[] t ≠ w → if ℓb → ℓ:(s) = r(t)
           [] ¬ ℓb→r:(s) = r(t)
           fi
fi
```

"vervang t door ℓ(t)": analoog voorgaande algorithme.

```
"laat v(t1) = max(tree(ℓ(t)))":
invariant: t1 = ℓ(t) ∨ t1 = r(s1)

           t1 := ℓ(t);
           do r(t1) ≠ - 1 → t1,s1 := r(t1),t1 od {r(t1) = - 1}

"vervang v(t) door v(t1)":
v:swap(t,t1);
if t1 = ℓ(t) → ℓ:(t) = ℓ(t1)
[] t1 ≠ ℓ(t) → r:(s1) = ℓ(t1)
fi
```

"dicht het gat op plaats t1":

We willen deze niet meer gebruikte waarde liever achteraan hebben staan. Daarvoor moeten we v.high èn zijn voorganger kennen. Deze kunnen we bepalen met de membershiptest, die we gebruikten bij insert en bij delete. Wanneer t1 = v.hib vinden we die niet meer, omdat we de arrays ℓ en r hebben aangepast. Dus moeten we onderscheid maken tussen t1 = v.hib en t1 ≠ v.hib.

algorithme

```
if t1 = v.hib → v:hirem; ℓ:hirem; r:hirem
 [] t1 ≠ v.hib → t := w;
                 do v.high < v(t) → s,t,ℓb := t,ℓ(t),true
                 [] v.high > v(t) → s,t,ℓb := t,r(t),false
                 od;
                 if t = w → w := t1
                 [] t ≠ w → if ℓb → ℓ:(s) = t1
                            [] ¬ ℓb → r:(s) = t1
                            fi
                 fi;
                 v:swap(t1,t); ℓ:swap(t1,t); r:swap(t1,t);
                 v:hirem; ℓ:hirem; r:hirem
fi
```

De complexiteit van een insert operatie is evenredig met het aantal vergelijkingen dat nodig is tussen het toe te voegen element en de elementen van de verzameling.
Bij n insertoperaties is de complexiteit worst case $O(n^2)$, maar expected case blijkt de complexiteit $O(n \log n)$ te zijn, aangenomen dat er een uniforme distributie van de elementen is.
In het onderstaande bewijs zal aangetoond worden dat het verwachte aantal vergelijkingen, $T(n)$, om in een lege verzameling n insert operaties te doen kleiner of gelijk is aan $2\ \ell n(n)$, met $n \geq 1$ en $T(0) = 0$.

Bewijs:

Laten de elementen zijn $a_1, a_2, \ldots, a_n$. Laten $b_1, b_2, \ldots, b_n$ dezelfde getallen zijn, maar nu gesorteerd naar toenemende grootte.
Als $a_1 = b_j$, $1 \leq j \leq n$, dan komt b_j in de wortel van de boom. Omdat dit het eerste element is, was hiervoor geen vergelijking nodig. De elementen $b_1, b_2, \ldots, b_{j-1}$ komen dan in de linkerdeelboom van de wortel. Hiervoor zijn $j - 1 + T(j - 1)$ vergelijkingen nodig. De elementen $b_{j+1}, \ldots, b_n$ komen in de rechterdeelboom van de wortel. Hiervoor zijn $n - j + T(n - j)$ vergelijkingen nodig.

De kans dat $a_1 = b_j$ is 1/n zodat

$$T(n) = \frac{1}{n} \sum_{j=1}^{n} (j - 1 + T(j - 1) + n - j + T(n - j))$$

$$= n - 1 + \frac{2}{n} \sum_{j=0}^{n-1} T(j), \qquad n \geq 1 .$$

$$T(0) = 0 .$$

Met behulp van inductie bewijzen we dat $T(n) \leq 2n\ \ell n(n)$ voor $n \geq 1$.
Als $n = 1$ dan is $T(1) = 0$ en is aan de induktievergelijking voldaan.
Neem aan $T(j) \leq 2j\ \ell n(j)$ voor alle $j : 1 \leq j < n$.
Dan is

$$T(n) \leq n - 1 + \frac{2}{n} \sum_{j=2}^{n-1} 2j\ \ell n(j) = n - 1 + \frac{4}{n} \sum_{j=2}^{n-1} j\ \ell n(j)$$

$$\leq n - 1 + \frac{4}{n} \int_{2}^{n} x \ell n(x) dx \leq n - 1 + 2n\ \ell n(n) - n \leq 2n\ \ell n(n) .$$

De verhouding $2n\ \ell n(n)/n \log(n)$ is ongeveer 1,4. Dit betekent dat het verwachte gedrag ongeveer 40% slechter is dan wat optimaal mogelijk is.

4.3. Gebalanceerde bomen

Om te voorkomen dat bomen "scheefgroeien" moeten ze na een insert of delete operatie weer in balans gebracht worden wanneer ze niet meer aan een bepaald balanscriterium voldoen.
We zouden kunnen eisen dat voor iedere knoop het verschil tussen het aantal knopen in de linker en rechter deelbomen maximaal één is. Deze eis is erg streng en betekent dat na bijna iedere insert of delete de boom geheel gewijzigd zou moeten worden.
Daarom kiezen we voor het volgende, zwakkere, criterium:
Voor iedere knoop mogen de hoogten van de linker en de rechter deelbomen maximaal één verschillen.
Bomen die aan deze eis voldoen heten AVL-trees [Adelson-Velskii & Landis].

4.3.1. AVL-trees [8]

We definiëren de hoogte van een boom:
hoogte (tree(-1)) = 0,
voor k ≥ 0: hoogte (tree(k)) = 1 + max (hoogte (tree(ℓ(k))), hoogte (tree(r(k)))).

Een AVL-tree met hoogte h heeft tenminste $F_{h+2} - 1$ elementen, waarin F_i het i-de getal van Fibonacci is.
Voor h = 0 en h = 1 is het eenvoudig in te zien dat deze bewering waar is.
Als h ≥ 2 dan bestaat de boom uit een wortel en een deelboom met hoogte h - 1 en een deelboom met hoogte minsten h - 2.
Het aantal elementen is dan minstens $1 + F_{h+1} - 1 + F_h - 1 = F_{h+2} - 1$.

Met behulp van de formule $F_h = \frac{1}{5}\sqrt{5}(\frac{1+\sqrt{5}}{2})^h - \frac{1}{5}\sqrt{5}(\frac{1-\sqrt{5}}{2})^h$ kunnen we een schatting krijgen voor het aantal elementen in de boom. Dit is

$O((\frac{1+\sqrt{5}}{2})^h) \approx O(1,6^h)$.
Een boom met hoogte h bevat dus tenminste $1,6^h$ elementen. De hoogte van een boom met n elementen is dan O(log n).

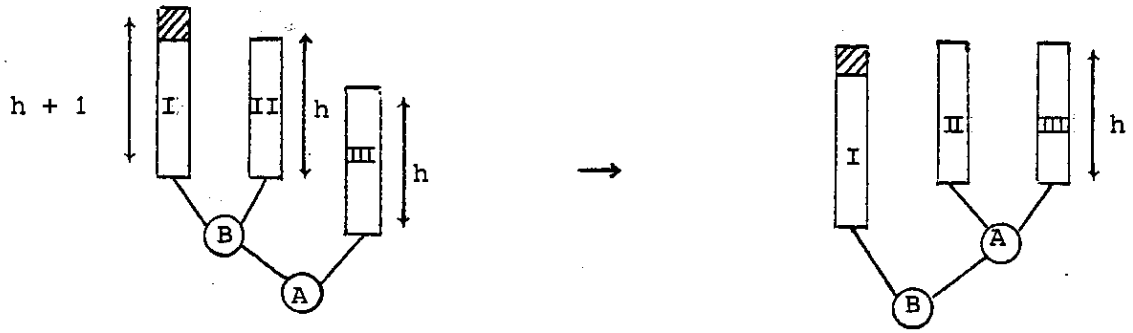
Balanceren van bomen

We willen bijv. na een insert operatie de boom zonodig opnieuw balanceren.
Daartoe houden we voor iedere knoop in de boom een balansfactor bij:
"de overmaat van de rechterdeelboom". Deze factor mag de waarden 1,0,-1 hebben. Administreer het pad van de wortel naar het toegevoegde element en beschouw de punten op dat pad in de richting van de wortel.
Stel dat een zeker punt A een balansfactor -2 heeft, een overmaat links dus.
Als de rechterdeelboom van A hoogte h heeft, dan heeft de linkerdeelboom hoogte h + 2.
Laat B de wortel zijn van deze linkerdeelboom.

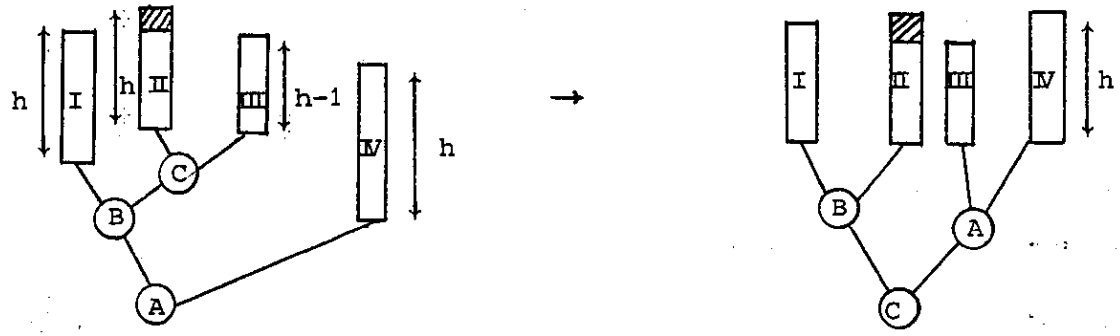
Er zijn nu twee mogelijkheden:

1. het nieuwe element is terecht gekomen in de linkerdeelboom van B
2. het nieuwe element is terecht gekomen in de rechterdeelboom van B.

1. Aanvankelijk had de linkerdeelboom met B als wortel hoogte h, zodat nu geldt: hoogte (tree(ℓ(B))) = h + 1 en hoogte (tree(r(B))) = h. Kies nu B als wortel van de boom en zorg dat de ordening goed blijft.



2. Laat C de wortel zijn van tree(r(B)). Kies nu C als wortel van de boom en A en B als wortels van de deelbomen van C. Zorg dat de ordening goed blijft (B < C < A).



In beide gevallen wordt de hoogte van de boom weer gelijk aan die van de oorspronkelijke boom. Zo'n operatie hoeft dus bij een insert ten hoogste éénmaal uitgevoerd te worden. Voor de delete voeren we analoge operaties uit, maar de hoogte van de boom blijft nu niet gelijk, zodat de operatie mogelijk meer keren moet worden uitgevoerd, maar het blijft O(log n).

4.3.2. B-trees

Terwijl knopen van binaire bomen één element bevatten, representeren de knopen van een B-tree [9] een aantal elementen.

De definitie van een B-tree van orde k, $k \geq 1$, luidt:

1. Iedere knoop bevat ten hoogste 2k elementen.
2. Iedere knoop, behalve de wortel, bevat tenminste k elementen; de wortel bevat tenminste 1 element.
3. Een interne knoop met m elementen heeft m + 1 niet-lege deelbomen, geordend op de gebruikelijke wijze.
4. Alle bladen bevinden zich op hetzelfde niveau.

Als k = 1 wordt de boom ook wel een BB-tree of 2-3 tree genoemd.

B-trees zijn oorspronkelijk ontworpen voor het representeren van verzamelingen op achtergrondgeheugen. Voor een inspectie is dan in het algemeen geheugentransport noodzakelijk. Hoewel een operatie op de verzameling O(log n) is kost het veel wachttijd. De totale wachttijd kan verkort worden door iedere geheugenpagina die naar het hoofdgeheugen getransporteerd wordt zo goed mogelijk te benutten. Hiertoe wordt k zo groot mogelijk gekozen, zo groot dat de elementen van een knoop nog net op een geheugenpagina passen. Door het vergroten van k wordt de hoogte van de boom gereduceerd. We zullen nu de operaties member, insert en delete beschouwen in een B-tree van orde k. Voor een boom van n elementen zijn deze operaties, inclusief eventueel herbalanceren, weer O(log n).

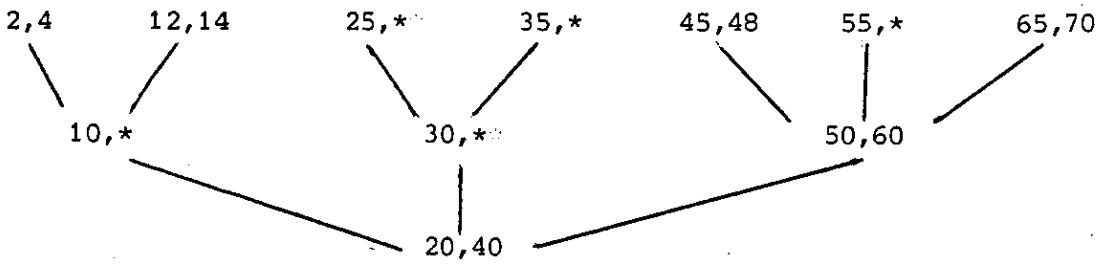
member:

Deze is in principe hetzelfde als bij de binaire boom, echter nu is er in ieder knoop geen keuze uit twee deelbomen, maar uit m + 1 deelbomen voor een knoop die m elementen bevat. Hier kan met binaire search een keuze gemaakt worden.

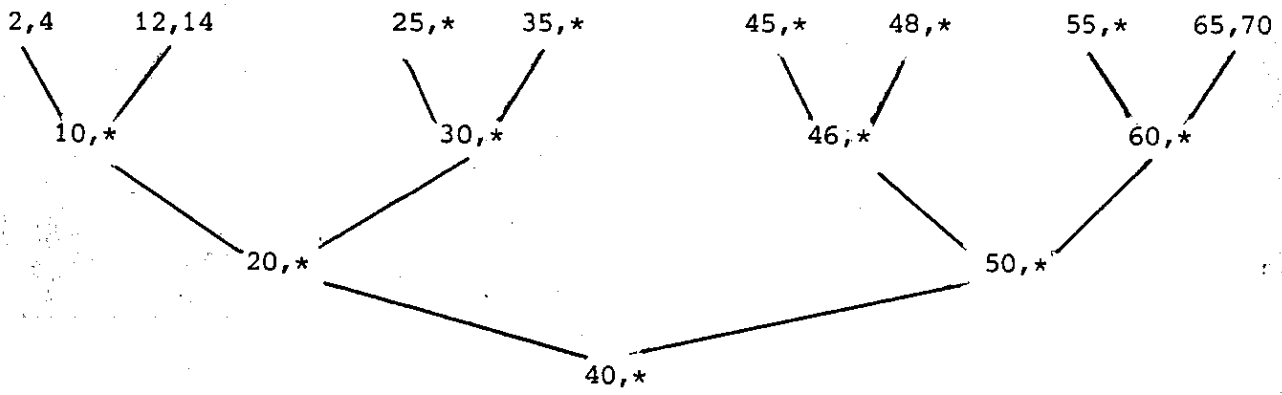
insert:

Net als bij de binaire boom komt het in te voegen element altijd in een blad. Als het aantal elementen in een blad kleiner is dan 2k kan het nieuwe gewoon worden toegevoegd. Anders wordt het blad, dat nu 2k + 1 elementen bevat, verdeeld in 2 bladen met elk k elementen en het middelste element voegen we toe aan de voorgangerknoop. Deze knoop kan nu echter ook 2k + 1 elementen bevatten en dan wordt dit splitsproces herhaald. Als de wortel 2k + 1 elementen zou gaan bevatten zal, omdat deze geen voorganger heeft,

de hoogte van de boom moeten toenemen. De oude wortel wordt gesplitst in 2 interne knopen met elk k elementen en er wordt een nieuwe wortel gemaakt met 1 element.
Voorbeeld van insert in B-tree van orde k = 1. De sterretjes stellen vrije plaatsen voor.



Insert "46" resulteert in de volgende boom:



delete:

Indien een element dat niet in een blad zit verwijderd moet worden, moet dit eerst verwisseld worden met een element dat wel in een blad zit. Dit kan altijd aangezien voor iedere interne knoop geldt dat zijn direct grotere en direct kleinere in een blad zitten. Verder gebeurt alles in omgekeerde volgorde als bij de insert. (Als we in het voorafgaande voorbeeld "46" weer verwijderen ontstaat zo de oude boom weer.)

5. Berekeningen aan grafen

Een graaf is een eindige verzameling punten (Eng. vertices) met daarop gedefinieerd een binaire relatie. Paren elementen die aan de relatie voldoen noemen we takken. We maken onderscheid tussen gerichte grafen en ongerichte grafen. Bij een ongerichte graaf is de binaire relatie symmetrisch, de takken worden dan ook wel zijden genoemd (Eng. edges). Bij een gerichte graaf worden de takken ook wel pijlen genoemd (Eng. arcs).
Uit de definitie volgt al dat er tussen twee punten hoogstens één tak mag bestaan. Verder zijn loops, dit zijn verbindingen van een punt met zichzelf, niet geoorloofd. (De relatie is antireflexief.) In een gerichte graaf is y de opvolger van x en x de voorganger van y als er een pijl van x naar y is. Bij een ongerichte graaf spreken we van buurpunten als twee punten door een zijde verbonden zijn. We definiëren een pad als een aaneenschakeling van takken, waarbij ieder intern punt hoogstens eenmaal voorkomt. Een ongerichte graaf heet samenhangend indien ieder tweetal punten verbonden is door een pad.

5.1. Constructie van een palmboom

Zij G een samenhangende ongerichte graaf met N punten (genummerd 0 t/m N-1). G is gegeven d.m.v. twee arrays: b(0 : N) en s. De buurpunten van punt i worden gegeven door $\{s(j) \mid b(i) \leq j < b(i+1)\}$. Iedere tak is nu twee maal geadministreerd, dus het totaal aantal takken is s.dom / 2.
Voor een palmboom (Eng. depth first spanning tree) geldt:

A1. het is een uit-boom met punt 0 als wortel,
A2. de punten 0,...,N - 1 zijn de knooppunten van de boom,
A3. de boom heeft slechts pijlen tussen buurpunten van G,
A4. de takken van G verbinden slechts punten, waarbij het ene punt in de boom op het wortelpad van het andere ligt.

(Een uit-boom is een wortelboom waarin de pijlen gericht zijn vanuit de wortel.)

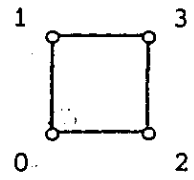
We representeren de palmboom door van iedere tak van G aan te geven wat zijn status is in de palmboom. We maken daartoe gebruik van een array te, met te.dom = s.dom en te(i) $\in \{-1,0,1\}$: zij b(i) $\leq$ j < b(i + 1) dan geldt:

te(j) = 1 : de palm heeft een pijl van i naar s(j),
te(j) = -1 : de palm heeft een pijl van s(j) naar i,
te(j) = 0 : de palm heeft geen pijl tussen i en s(j).
(G heeft wel een tak tussen i en s(j),
zo'n tak wordt een back edge genoemd).

Zij D een deelgraaf van G. D bestaat dan uit een aantal punten van G met de bijbehorende takken.
Als invariant kiezen we: D is een deelgraaf van G en van D is een palmboom bepaald.
De algorithme is dan als volgt:

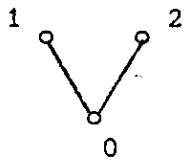
D := {0};
do D ≠ G → voeg aan D één punt uit G\D toe;
herstel invariant
od

Uit het volgende voorbeeld blijkt dat D niet willekeurig gekozen mag worden.

G: 

D bevat initieel alleen punt 0.

Aan D worden achtereenvolgens de punten 1 en 2 toegevoegd, de palm is dan



Aan D wordt nu punt 3 toegevoegd. We krijgen nu óf een pijl van 2 naar 3 en daarmee een back edge tussen 3 en 1 óf een pijl van 1 naar 3 en daarmee een back edge tussen 3 en 2. Beide gevallen zijn in tegenspraak met A4.

De invariant voor de algorithme is dus niet sterk genoeg. De nieuwe invariant wordt:

D is een deelgraaf van G; van D is een palmboom bepaald; in de palm van D is er een punt a zodanig dat alle verbindingen tussen D en G\D verbindingen zijn met punten op het wortelpad van a (het wortelpad is inclusief a en de wortel).

Voor de uitbreiding van D bepalen we het punt p op het wortelpad van a, dat het dichtst bij a ligt, mogelijk a zelf, en dat verbonden is met een punt uit G\D. Noem dit punt uit G\D b en voeg b toe aan D. De palm wordt uitgebreid met een pijl van p naar b. De nieuwe D is een deelgraaf van G, de back edges vanuit b zijn verbindingen tussen b en punten op het wortelpad van p; dus de uitgebreide boom is een palmboom.
Alleen punten op het wortelpad van b hebben verbindingen met G\D. Voor het nieuwe punt a kiezen we b, zodat weer aan de invariant is voldaan.
Voor de algorithme maakt het niet uit welk buurpunt b van p we kiezen, maar om de administratie eenvoudig te houden beschouwen we de buurpunten van p in volgorde van voorkomen in het array s(b(p) : b(p + 1) - 1).
Voor de administratie gebruiken we naast het array te nog drie arrays:

- een array t waarin we het pad van de wortel naar a administreren:
  t.lob = 0, t(0) = 0, t.high = a,
- een array c, waarvoor geldt dat voor elk punt t(i) zijn buurpunten in G\D een deelverzameling vormen van {s(j) | c(i) ≤ j < b(t(i) + 1)},
  c.dom = t.dom,
- een boolean array d(0 : n - 1) waarvoor geldt d(i) ≡ i ∈ D.

Om de codering te vereenvoudigen kiezen we punt a zodanig dat c.high < b(t.high + 1). Dan is er vanuit a nog een verbinding met G\D of een back edge naar een punt op het wortelpad, die nog niet geadministreerd is in array te.

```
algorithme
L1:     d := (0,true); do d.dom ≠ N → d:hiext(false) od;
        t := (0,0); c := (0,b(0));
L2:     te := (s.lob); do te.dom ≠ s.dom → te:hiext(0) od;
L3:     do t.dom > 0 → j,sj := c.high, s(c.high);
                       c:(c.hib) = c.high + 1;
                       if ¬ d(sj) → d:(sj) = true; t:hiext(sj);
                                    c:hiext(b(sj)); te:(j) = 1
                       [] d(sj) → if t.dom > 1 ∧ sj = t(t.hib - 1) → te:(j) = 1
                                  [] t.dom = 1 ∨ sj ≠ t(t.hib - 1) → te:(j) = 0
                                  fi
                       fi;
                       do t.dom > 0 ∧ c.high ≥ b(t.high + 1) → t:hirem; c:hirem od
        od
```

L1 is O(# punten), L2 is O(s.dom) = O(# takken).
Elke keer dat L3 wordt doorlopen, wordt aan één plaats van het array te een waarde toegekend, dus L3 is O(te.dom) = O(# takken).
De algorithme is O(# punten + # takken) = O(# takken).

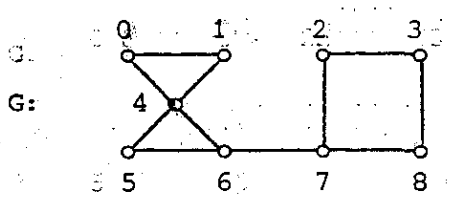
5.2. Biconnected components

Het belang van het maken van palmbomen is onderkend door R.E. Tarjan [10]. Wanneer we aan de algorithme voor het maken van palmbomen statements (geen loops) toevoegen verandert de ordegrootte niet. Tarjan maakte daarvan gebruik door op deze manier problemen met betrekking tot grafen op te lossen in O(# takken), waarvan niet bekend was dat ze in O(# takken) opgelost konden worden.
Als voorbeeld behandelen we de bepaling van de biconnected components van een ongerichte graaf.

Een ongerichte graaf is biconnected als voor iedere drie onderling verschillende punten a,b en c geldt dat er een pad is tussen a en b, dat niet door c gaat. Wanneer de graaf meer dan twee punten bevat is de definitie equivalent met: *ieder tweetal punten ligt op een cycle.*
De definitie is dan ook equivalent met: ieder tweetal punten is verbonden door twee paden, die geen interne punten gemeen hebben.

Een biconnected component is een deelgraaf, die biconnected is, en die niet bevat is in een grotere deelgraaf die biconnected is.
Punten die in twee biconnected components liggen heten articulation points of cut points.
Twee biconnected components hebben ten hoogste één punt gemeen.

Voorbeeld

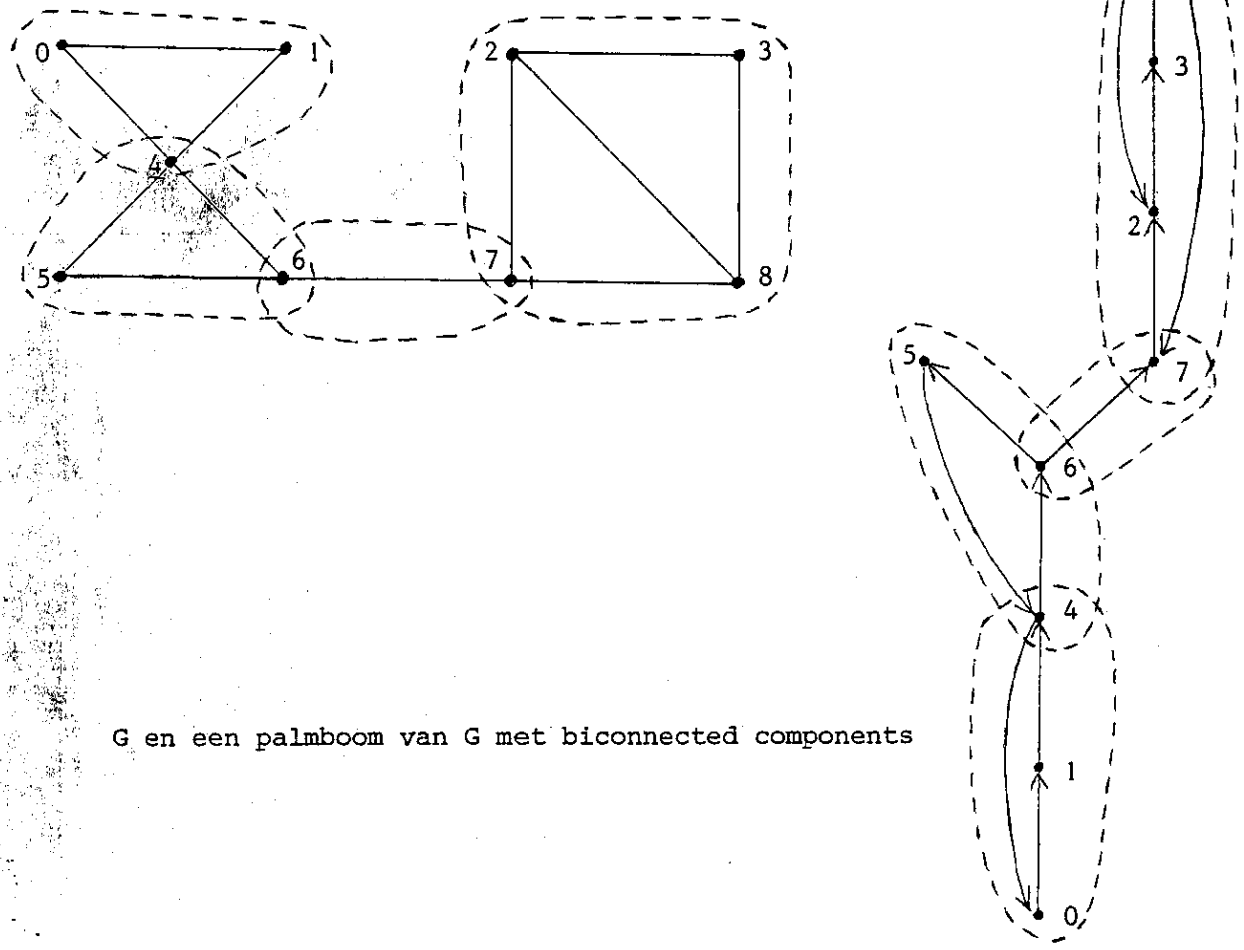


G is niet biconnected (bijvoorbeeld elk pad tussen de punten 4 en 7 gaat door punt 6). De biconnected components van G zijn:
{0,1,4} , {4,5,6} , {6,7} , {2,3,7,8} .
De articulation points zijn: 4,6 en 7.
Wanneer er tevens een tak zou bestaan tussen de punten 1 en 2 was G wel biconnected en waren er geen articulation points.

Omdat we gebruik willen maken van een palmboom van G definiëren we eerst een biconnected component van een graaf aan de hand van zijn palmboom:

definitie:

- de hele boom bestaat uit één of meer biconnected components
- als voor een punt a geldt dat het een opvolger b heeft, zo dat er geen back edge bestaat tussen enig punt in tree(b) en een punt ≠ a op het wortelpad van a, dan vormt {a} ∪ tree(b) één of meer biconnected componenten.

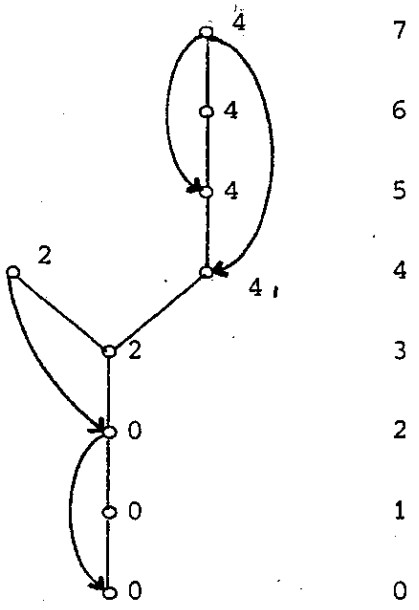


G en een palmboom van G met biconnected components

Gevraagd wordt een programma te maken dat de maximale grootte bepaalt van enige biconnected compononent van een samenhangende ongerichte graaf G.
We zullen uitgaan van het programma voor het construeren van een palmboom. Hieraan worden statements toegevoegd zodanig dat de ordegrootte van het programma niet verandert.
Voor ieder punt a van de palmboom definiëren we de waarden d(a) en ℓ(a).

d(a) is de afstand van a tot de wortel:
d(0) = 0; als i $\bullet\!\!\rightarrow\!\!\bullet$ j dan is d(j) = d(i) + 1.
ℓ(a) = min({d(a)} $\cup$ {ℓ(b) | a $\bullet\!\!\rightarrow\!\!\bullet$ b} $\cup$ {d(j) | $\exists$ back edge(a,j)})

Uit de definitie volgt direct dat ℓ(0) = 0 en als i $\bullet\!\!\rightarrow\!\!\bullet$ j dan ℓ(i) $\leq$ ℓ(j).
De waarde ℓ(a) is de d-waarde van het punt dat het dichtst bij de wortel ligt en dat door middel van een pad van nul of één back edge met tree(a) verbonden is.
Het punt a is een articulation point d.e.s.d. als a een opvolger (in de palmboom) b heeft met ℓ(b) $\geq$ d(a).



ℓ-waarden d-waarden

In de algorithme behouden de arrays t en c dezelfde betekenis als bij de constructie van een palmboom. Omdat we niet meer geïnteresseerd zijn in de palmboom zelf gebruiken we array te niet meer.
Het boolean array d wordt vervangen door een integer array d, dat de d-waarden bevat ($d(i) \neq N \equiv i \in D$). In een array ℓ administreren we de ℓ-waarden. Per tak $t(i) \circ\!\!-\!\!\!\rightarrow\!\circ\ t(i+1)$ is er een waarde bc(i). Dit is de grootte (# takken) van de biconnected component binnen D waar die tak toe behoort. De grootte van een grootste biconnected component binnen D wordt vastgelegd in de variabele mb, die dus na afloop het antwoord bevat.

Bij uitbreiding van D met punt sj wordt de d-waarde bepaald, de ℓ-waarde daarmee geinitialiseerd (S1) en een nieuwe biconnected component van één tak toegevoegd (S2).

De ℓ-waarde van een punt is het minimum van drie verzamelingen. Deze driedeling vinden we in de algorithme terug:

1. De initialisering van de ℓ-waarde (S1).
2. Bij het constateren van een back edge tussen t.high en sj wordt de ℓ.high zo mogelijk verlaagd tot d(sj) (L1).
3. Wanneer t.high (laat dat $t(i+1)$ zijn) uit t verwijderd wordt (S3) en $\ell(i+1)$ dus definitief is, wordt $\ell(i)$ zo mogelijk verlaagd tot $\ell(i+1)$ (L2).

In het laatste geval wordt verder nagegaan of $\ell(i+1) \geq d(i)$ zodat t(i) een articulation point is. Zo ja, dan wordt mb zo mogelijk verlaagd tot bc(i) (L3). Zo nee, dan worden de biconnected components waar de takken $t(i) \circ\!\!-\!\!\!\rightarrow\!\circ\ t(i+1)$ en $t(i-1) \circ\!\!-\!\!\!\rightarrow\!\circ\ t(i)$ toe behoren, samengevoegd (S4).

```
algorithme

d := (0,0); do d.dom ≠ N → d:hiext(N) od;
t := (0,0); c := (0,b(0)); ℓ := (0,0); bc := (0); mb := 0;
do t.dom > 0 → j,sj := c.high, s(c.high); c:(c.hib) = c.high + 1;
                if d(sj) = N → t:hiext(sj); c:hiext(b(sj));
S1:                            d:(sj) = t.hib; ℓ:hiext(t.hib);
S2:                            bc:hiext(1)
                [] d(sj) ≠ N → if t.dom > 1 ∧ sj = t(t.hib - 1) → skip
                               [] t.dom = 1 ∨ sj ≠ t(t.hib - 1) →
L1:                               do ℓ.high > d(sj) → ℓ:(ℓ.hib) = d(sj) od
                               fi
                fi;
                do t.dom > 0 ∧ c.high ≥ b(t.high + 1) →
S3:                           t:hirem; c:hirem; ℓℓ,ℓ:hipop;
L2:                           if t.dom ≥ 1 → do ℓ.high > ℓℓ → ℓ:(ℓ.hib) = ℓℓ od;
                                                bb,bc:hipop;
                                                if ℓℓ ≥ d(t.high) →
L3:                                                 do bb > mb → mb := bb od
                                                [] ℓℓ < d(t.high) →
S4:                                                 bc:(bc.hib) = bc.high + bb
                                                fi
                              [] t.dom = 0 → skip
                              fi
                od
od
```

5.3. Transitieve afsluiting

We behandelen de algorithme van Warshall[11], voor het vaststellen van de transitieve afsluiting van een binaire relatie.

Zij G een gerichte graaf. We representeren de graaf in een boolean array c(1 : n,1 : n), en definiëren de binaire relatie als

$$c(i,j) \equiv G \text{ heeft een pijl } i \circ\!\!\rightarrow\!\!\circ\ j\ .$$

Bij de transitieve afsluiting van deze relatie moeten we de pijl i $\circ\!\!\rightarrow\!\!\circ$ j aan de graaf toevoegen, wanneer er een niet triviaal pad is van i naar j (een triviaal pad is een pad van nul takken).
Als c(i,k) en c(k,j) de waarden true hebben, dan moet ook c(i,j) de waarde true krijgen.
We kunnen als invariant kiezen: $1 \leq k \leq$ langste pad $\wedge$ alle paden ter lengte $\leq k$ zijn vastgesteld.
Deze invariant is waar voor k = 1. Wanneer we k met één verhogen is het niet gemakkelijk om de invariant te herstellen.
Eenvoudiger is:

$$c(i,j) \equiv G \text{ heeft een niet-triviaal pad van } i \text{ naar } j \text{ met interne punten uit } \{\ell \mid 1 \leq \ell \leq k\} .$$

Wanneer we nu k met één verhogen hoeven we alleen maar de eventuele pijlen i $\circ\!\!\rightarrow\!\!\circ$ k en k $\circ\!\!\rightarrow\!\!\circ$ j te beschouwen ($1 \leq i \leq n$, $1 \leq j \leq n$).
Met andere woorden:

$$\forall i,j : c{:}(i,j) = c(i,j) \vee (c(i,k) \wedge c(k,j)) .$$

algorithme

```
k := 0;
do k ≠ n → k := k + 1; i := 0;
        do i ≠ n → i := i + 1;
                if c(i,k) → j := 0;
                        do j ≠ n → j := j + 1;
                                c:(i,j) = c(i,j) ∨ c(k,j)
                        od
                [] ¬ c(i,k) → skip
                fi
        od
od
```

Deze algorithme is $O(n^3)$.

Laten nu de pijlen van G lengten hebben. G is gegeven m.b.v. een integer array c(1 : n, 1 : n), met c(i,j) = als G een pijl i $\circ\!\!\rightarrow\!\!\circ$ j heeft de lengte daarvan en anders inf. We moeten een programma maken dat array c zó wijzigt dat geldt:

c(i,j) = als G een niet-triviaal pad van i naar j heeft,
de lengte van het kortste pad, en anders inf..

De lengte van een pad is de som van de lengten van de pijlen van dat pad.

Hiertoe moeten we in het voorgaande

$$c{:}(i,j) = c(i,j) \vee (c(i,k) \wedge c(k,j))$$

wijzigen in

$$c{:}(i,j) = \min(c(i,j),\ c(i,k) + c(k,j)) \ .$$

5.4. Schorr-Waite markeringsalgorithme [12]

We bekijken een gerichte graaf G met N punten, genummerd 0 t/m N - 1. We veronderstellen dat elk punt ten hoogste twee opvolgers heeft. We introduceren een extra punt -1. We geven elk punt precies twee opvolgers door zonodig het punt -1 (één- of tweemaal) als opvolger toe te voegen. De opvolgers van i worden gegeven door $\ell(i)$ en r(i) (N.B. ook $\ell(-1) = r(-1) = -1$).

Een rij $e_0, e_1, \ldots, e_m$ heet een pad van i naar j, indien

$$m \geq 0,\ e_0 = i,\ e_m = j \text{ en } \forall_k : 0 \leq k < m{:}\ e_{k+1} = \ell(e_k) \vee e_{k+1} = r(e_k)$$

We schrijven $i \overset{*}{\rightarrow} j$ als er een pad van i naar j bestaat.

Gegeven wordt een punt b, $0 \leq b < N$. We definiëren: $D = \{i \mid b \overset{*}{\rightarrow} i\}$.
Het doel van de markeringsalgorithme is vast te stellen welke punten tot D behoren. Daartoe introduceren we een boolean array m(-1 : N), initieel m(i) = false en de uitvoering leidt tot $\forall_i : -1 \leq i < N : m(i) \equiv (i \in D)$.

Voor dit probleem is er een triviale recursieve algorithme:

```
proc mark(p):
    if ¬ m(p) → m:(p) = true; mark(ℓ(p)); mark(r(p))
    [] m(p) → skip
    fi
corp
```

De procedure wordt aangeroepen met mark(b).

Deze algorithme kunnen we ook niet-recursief noteren

```
p,s := b,(1);
do ¬ m(p) → m:(p) = true; s:hiext(p); p := ℓ(p)
[] m(p) ∧ s.dom ≠ 0 → p,s:hipop; p := r(p)
od
```

We zien hier dat de elementen van D de een na de ander aan de beurt komen en gaandeweg gemarkeerd worden.

s bevat een pad van b naar p.
Er geldt:

1. alle punten van s zijn gemarkeerd,
2. een niet gemarkeerd punt van D ligt op een pad van ongemarkeerde punten waarvan het begin ofwel p ofwel een r(s(i)) is.

De idee van Schorr-Waite is geen extra array s te gebruiken, maar alleen een integer variabele q en wijzigingen aan te brengen in de arrays ℓ en r, zodanig dat uit q, ℓ en r het array s te construeren is. Verder dienen de wijzigingen in ℓ en r zodanig te zijn dat ze na afloop van het programma weer hun oorspronkelijke waarde hebben.

Indien we p := ℓ(p) vervangen door p,q := ℓ(p), p weten we dat ℓ(q) = p en kunnen we ℓ(q) gebruiken om bijv. de voorganger van q op het pad vanaf b te registreren; bij die voorvanger van q hebben we nu ook weer ruimte over om daarvan de voorganger te registreren, etc. Op deze manier kunnen we het hele pad van b naar p registreren.

Omdat de waarden in ℓ en r gewijzigd worden geven we hun beginwaarden een naam. De beginwaarde is $\forall i: -1 \leq i < N : \ell(i) = L_i \wedge r(i) = R_i$. Ook spreken we over de voorganger van i. Noem deze P_i. In het algemeen heeft i diverse voorgangers, maar we beperken ons alleen tot punten op het pad s van b naar p en op dat pad heeft elk punt precies één voorganger.

Met de algorithme die hierna volgt, wordt elk punt uit D driemaal gepasseerd. Elke keer dat punt i gepasseerd wordt, worden ℓ en r gewijzigd. We moeten registreren hoevaak i is gepasseerd. Hiervoor kunnen we array m gebruiken. Daartoe maken we m een integer array, initieel m(i) = 0 en de uitvoering leidt tot

$$\forall_{i \in G} \; [m(i) = 0 \Leftrightarrow i \notin D \wedge m(i) = 3 \Leftrightarrow i \in D]$$

Om de beschrijving van de algorithme te vereenvoudigen voeren we een extra wortel vb in en vb $\notin \{-1,0,1,\ldots,N-1\}$, $\ell(vb) = r(vb) = b$ (vb is een voorganger van b) en $m(vb) = 2$.

We geven nu de algorithme en bespreken daarna de correctheid.

```
p,q := b,vb;
do p ≠ vb → m(p) := m(p) + 1;
            if m(p) = 3 ∨ m(ℓ(p)) = 0 → p,ℓ(p),r(p),q := ℓ(p),r(p),q,p
            ▯ m(p) ≠ 3 ∧ m(ℓ(p)) ≠ 0 → ℓ(p),r(p),q := r(p),q,ℓ(p)
            fi
od
```

In het navolgende schrijven we expressies van de vorm $[a,b,c](i)$, waarin de rij a,b,c een array ter lengte 3 voorstelt met ondergrens 0 en i als index.

Voorbeeld: $[a,b,c](0) = a$ en $[a,b,c](2) = c$.

Als invariant kiezen we:

Invariant I 1. Voor elke i, $-1 \leq i \leq N-1$, geldt

a) $0 \leq m(i) \leq 3$,

b) $\ell(i) = [L_i,R_i,P_i,L_i](m(i))$,

c) $r(i) = [R_i,P_i,L_i,R_i](m(i))$.

2. $0 \leq m(p) \leq 2 \wedge q = [P_p,L_p,R_p](m(p))$.

3. Er is een pad $s = e_0,e_1,\ldots,e_k, e_{k+1}$ in de oorspronkelijke graaf met $k \geq -1, e_0 = vb, e_{k+1} = p$ en $\forall j : 0 \leq j \leq k : 1 \leq m(e_j) \leq 2 \wedge e_{j+1} = [P_{e_j},L_{e_j},R_{e_j}](m(e_j)) \wedge P_{e_{j+1}} = e_j$.

4. $\forall_{i \notin s}[m(i) = 0 \vee m(i) = 3]$.

Dat p,q := b,vb de invariant waarmaakt is eenvoudig na te gaan. We zullen de invariantie slechts informeel verifiëren.

Met p' noteren we de waarde van p voor de uitvoering van de herhaalde statements die we bekijken onder de veronderstelling $I \wedge p' \neq vb$ en waarvan we moeten laten zien dat na afloop I geldt.

Omdat de uitvoering van de herhaalde statements de arrays slechts in p' wijzigt hoeven we niet naar andere punten te kijken.

Uit I2 volgt $0 \leq m(p') \leq 2$. Elke slag van de cyclus omvat $m(p') := m(p') + 1$ en $\ell(p')$, $r(p') := r(p),q$ hetgeen de waarden door $\ell(p')$ en $r(p')$ "roteert" op de vereiste wijze, dus I1 geldt na afloop.

Om in te zien dat I2 en I3 invariant zijn moeten we vijf gevallen onderscheiden, afhankelijk van de waarden van $m(p')$ en $m(\ell(p'))$ aan het begin van een herhaling:

voor uitvoering	$\ell(p')$ voor uitvoering	s na afloop	p na afloop	m(p)
$m(p')=0, m(\ell(p'))=0$	$L_{p'}$	$e_0,\ldots,e_k,p',p$	$L_{p'}$	0
$m(p')=1, m(\ell(p'))=0$	$R_{p'}$	$e_0,\ldots,e_k,p',p$	$R_{p'}$	0
$m(p')=0, m(\ell(p'))\neq 0$	$L_{p'}$	$e_0,\ldots,e_k,p'$	p'	1
$m(p')=1, m(\ell(p'))\neq 0$	$R_{p'}$	$e_0,\ldots,e_k,p'$	p'	2
$m(p')=2$	$P_{p'}(=e_k)$	$e_0,\ldots,e_k$	e_k	$m(e_k)\neq 3$

In de eerste twee gevallen wordt q gelijk aan p' dus aan I2 en I3 is voldaan.

In de volgende twee gevallen blijft p onveranderd. Aan het pad verandert niets, dus I3 geldt. Door $m(p') := m(p') + 1$ en $q := \ell(p')$ geldt $q = [P_p,L_p,R_p](m(p))$ en dus ook I2.
Tenslotte $m(p') = 2$; s wordt nu korter. I3 blijft nu zeker gelden. De waarde van $m(e_k)$ wordt niet gewijzigd, zodat na afloop $1 \leq m(p) \leq 2$, hetgeen samen met $q := p'$ leidt tot $q = [P_p,L_p,R_p](m(p))$, waaruit I2 volgt.

Alleen in dit laatste geval kan er iets veranderen aan I4. Het punt dat van het pad verdwijnt heeft markering 3, dus I4 blijft gelden.

Conclusie: I blijft invariant.

definieer nu $t := 3 * \#G - \sum_{i=-1}^{N-1} m(i)$

$t \geq 0$ en t wordt met elke stap met 1 verlaagd, dus de algorithme is eindig.
Na beëindiging geldt: p = vb, dus er behoren geen punten tot s, dus alle punten uit G hebben markering 0 of 3 (I4).
Punten uit $G \setminus D$ komen nooit aan de beurt, dus $\forall i \in G \setminus D\ [m(i) = 0]$.
Stel dat $i \in D$ en $m(i) = 0$ geldt.
Er is een pad van b naar i, $e_0,e_1,\ldots,e_m$, $e_0 = b$ en $e_m = i$.
$m(b) = 3$ dus er is een j met $m(e_j) = 3$ en $m(e_{j+1}) = 0$.
Omdat $m(e_j) = 3$ zijn zowel de rechter als de linker opvolger van e_j tijdens de algorithme bekeken. De markering van beide opvolgers is $\neq 0$, maar e_{j+1}

is één van deze twee opvolgers. Dit is een tegenspraak,
dus

$$\forall i \in D\ [m(i) = 3]\ .$$

Nog enkele opmerkingen:

- Indien we de initialisering vervangen door p,q := b,b en p ≠ vb vervangen door m(p) ≠ 3 hebben we de extra wortel niet nodig.
- Het is mogelijk de algorithme aan te passen aan de situatie waarin een punt meer dan twee uitgaande takken heeft. De m-waarde loopt dan op tot 1 + aantal uitgaande takken.
- Het "roteren" is vaker te gebruiken, bijv. bij het tellen van bladeren in een boom. Ga na dat voor het tellen van het aantal bladeren van een binaire boom het array m niet nodig is. We kunnen volstaan met de variabelen p en q en een extra variabele t voor het aantal bladen.

6. Discrete Fouriertransformatie

De rij f(0),...,f(N - 1) van complexe getallen wordt bij de discrete Fouriertransformatie naar een andere rij complexe getallen getransformeerd, F(0),...,F(N - 1), zo dat geldt:

$$F(u) = \sum_{v=0}^{N-1} f(v) \cdot w_N^{uv} \ , \text{ waarbij } w_N = e^{2\pi i/N}\ .$$

Het transformeren van de rij behelst het bepalen van N waarden F(u), terwijl voor elke F(u) N vermenigvuldigingen nodig zijn. Op deze manier is de transformatie $O(N^2)$.

Er bestaat een algorithme die de transformatie in O(N log N) uitvoert: de Fast Fourier Transform, bedacht door Cooley en Tukey [13].
Laat N een tweemacht zijn (dit kan eventueel gerealiseerd worden door de rij aan te vullen met nullen).
Uit $w_N = e^{2\pi i/N}$ volgt:

$$(w_N)^{N/2} = e^{\pi i} = -1$$
$$(w_N)^{N} = 1$$
$$(w_N)^{2} = W_{N/2}$$

Voor $u = 0,1,\ldots,N/2 - 1$ geldt:

$$F(2u) = \sum_{v=0}^{N-1} f(v) \cdot w_N^{2uv} = \sum_{v=0}^{N-1} f(v) \cdot w_{N/2}^{uv}$$

$$= \sum_{v=0}^{N/2-1} f(v) \cdot w_{N/2}^{uv} + \sum_{v=0}^{N/2-1} f(v + N/2) \cdot w_{N/2}^{u(v+N/2)}$$

$$= \sum_{v=0}^{N/2-1} f(v) \cdot w_{N/2}^{uv} + \sum_{v=0}^{N/2-1} f(v + N/2) \cdot w_{N/2}^{uv} \quad (w_{N/2}^{u\cdot\frac{N}{2}} = (w_{N/2}^{\frac{N}{2}})^u = 1)$$

$$= \sum_{v=0}^{N/2-1} (f(v) + f(v + N/2)) \cdot w_{N/2}^{uv}$$

$$F(2u + 1) = \sum_{v=0}^{N-1} f(v) \cdot w_N^{(2u+1)v} = \sum_{v=0}^{N-1} f(v) \cdot w_{N/2}^{uv} \cdot w_N^{v}$$

$$= \sum_{v=0}^{N/2-1} f(v) \cdot w_{N/2}^{uv} \cdot w_N^{v} + \sum_{v=0}^{N/2-1} f(v + N/2) \cdot w_{N/2}^{u(v+N/2)} \cdot w_N^{v+N/2}$$

$$= \sum_{v=0}^{N/2-1} f(v) \cdot w_{N/2}^{uv} \cdot w_N^{v} - \sum_{v=0}^{N/2-1} f(v + N/2) \cdot w_{N/2}^{uv} \cdot w_N^{v}$$

$$= \sum_{v=0}^{N/2-1} (f(v) - f(v + N/2)) \cdot w_{N/2}^{uv} \cdot w_N^{v}$$

De oorspronkelijke Fouriertransformatie is nu veranderd in twee nieuwe Fouriertransformaties over rijen ter lengte N/2.

We maken nu een procedure fft(p,q), die de Fouriertransformatie uitvoert op de rij f(p), f(p + 1),...,f(p + q - 1), waarbij de lengte q van de rij een macht van twee is:

$$F(p + u) = \sum_{v=0}^{q-1} f(p + v) w_q^{uv} . \qquad (u = 0,1,\ldots,q - 1) .$$

Als q = 1 dan is F(p) = f(p). Als q > 1 wordt de rij gesplitst in twee deelrijen, een met elementen met even indices en een met elementen met oneven indices.
Hierdoor verandert wel de volgorde van de elementen.

Voor de elementen met even index wordt de transformatie uitgevoerd op de nieuwe rij op de plaatsen p, p + 1,...,p + $\frac{q}{2}$ - 1 , voor de elementen met oneven index op de plaatsen p + q/2, p + q/2 + 1,..., p + q - 1. Voor v = 0,1,...,q/2 - 1 krijgen we deze twee rijen als volgt:

$$f{:}(p + v) = f(p + v) + f(p + \frac{q}{2} + v)$$

$$f{:}(p + \frac{q}{2} + v) = (f(p + v) - f(p + \frac{q}{2} + v)) * w_q^v .$$

Voor het berekenen van de nieuwe waarden hebben we alleen de in de voorgaande slag berekende waarden nodig.

algorithme

```
proc fft(p,q):
    if q = 1 → skip
    [] q > 1 → q := q/2; v := 0;
                do v ≠ q → z1,z2 := f(p + v),f(p + q + v);
                           f:(p + v) = z1 + z2;
                           f:(p + q + v) = (z1 - z2) * exp(πiv/q);
                           v := v + 1
                od;
                fft(p,q); fft(p + q,q)
    fi
corp
```

Als q = 1 wordt de procedure wel aangeroepen, maar er wordt niets berekend. Indien procedure calls duur zijn kunnen we de helft van de calls uitsparen door de q-test te verplaatsen.

algorithme

```
          proc fft(p,q):
S1:          q := q div 2; v := 0;
             do v ≠ q →
             .
             .
             .
             od;
S2:          if q ≤ 1 → skip
             [] q > 1 → fft(p,q); fft(p + q,q)
             fi
          corp
```

We hebben in twee statements (S1,S2) wijzigingen aangebracht, omdat de eerste maal de procedure aangeroepen kan worden met q = 1.
De benodigde rekentijd is:

$$T(1) = 0$$
$$T(q) = C \cdot q + 2T(q/2) = C \cdot q \cdot \log q \ .$$

De procedure wordt aangeroepen met fft(0,N).

Na afloop staat de rij F(u) nog niet in de goede volgorde. Alle elementen met index deelbaar door 2 staan in de eerste helft van de rij. Alle elementen met index deelbaar door 4 staan in het eerste kwart van de rij. Hierdoor ontstaat het vermoeden dat de plaats van F(u) de bitgespiegelde is van u.
We bewijzen dit als volgt:

$R_t(u)$, $0 \leq u < 2^t$, is de bitgespiegelde van u wanneer u gecodeerd is in t bits.

Definitie:

$$R_t(0) = 0 \ .$$
$$\left.\begin{aligned} R_{t+1}(2u) &= R_t(u) \\ R_{t+1}(2u+1) &= R_t(u) + 2^t \end{aligned}\right\} \ 0 \leq u < 2^t \ .$$

Stelling:

Na afloop van het programma geldt: $F(u) = f(R_{\log N}(u))$.

Bewijs (met volledige inductie):

Voor N = 1 is de stelling waar. Neem aan dat de stelling waar is voor $N = 2^t$. Kies nu $N = 2^{t+1}$. F(2u) komt terecht in het eerste deel van de rij. Met inductieveronderstelling komt F(2u) op plaats $R_t(u) = R_{t+1}(2u) = R_{\log N}(2u)$. F(2u + 1) komt terecht in het tweede gedeelte van de rij, dat begint met index 2^t, met inductieveronderstelling op plaats $2^t + R_t(u) = R_{t+1}(2u + 1) = R_{\log N}(2u + 1)$.

We willen een algorithme maken, die de F-waarden in de juiste volgorde zet. Als $i < R_{\log N}(i)$ dan wisselen we f(i) en $f(R_{\log N}(i))$. We doen dit met een procedure rev(u,r,t). Bij aanroep geldt dat $r = R_t(u)$. Het effect van de aanroep is dat f(i) en $f(R_{\log N}(i))$ verwisseld worden voor alle i die voldoen aan

1. $0 \leq i < N$
2. $i < R_{\log N}(i)$
3. $(i * \overline{2^t}) \text{ div } N = u$.

De laatste voorwaarde betekent: alle i waarvan het bitpatroon in log N bits uit dat van u(t bits) te verkrijgen is door er log N - t bits achter te zetten.

algorithme

```
proc rev(u,r,t):
    if 2^t < N → rev(2u,r,t + 1); rev(2u + 1,r + 2^t, t + 1)
    ▯ 2^t = N → if u < r → f:swap(u,r)
                ▯ u ≥ r → skip
                fi
    fi
corp
```

De aanroep is rev(0,0,0).

Om het berekenen van 2^t te vermijden vervangen we 2^t door q, en dus t + 1 door 2q

algorithme

```
proc rev(u,r,q):
    if q < N → rev(2u,r,2q); rev(2u + 1,r + q,2q)
    [] q = N → if u < r → f:swap(u,r)
               [] u ≥ r → skip
               fi
    fi
corp
```

De totale aanroep is nu: fft(0,N); rev(0,0,1). De benodigde rekentijd van rev is lineair in N:

$T(N) = C,$

$T(q) = 2T(2q),$

zo dat geldt: $T(1) = 2T(2) = 4T(4) = \ldots = N \cdot T(N) = C \cdot N$

Opmerking:

Er geldt

$$f(v) = \frac{1}{N} \sum_{u=0}^{N-1} F(u) \cdot w_N^{-uv} ,$$

zodat voor de inverse Fouriertransformatie een analoge algorithme van O(N log N) kan worden toegepast. Dit stelt ons in staat voor de convolutie van twee rijen ter lengte N een algorithme van O(N log N) te maken, in plaats van de triviale $O(N^2)$ algorithme.

De convolutie van twee rijen $(a_0,\ldots,a_{n-1})$ en $(b_0,\ldots,b_{n-1})$ is een rij $(c_0,\ldots,c_{2n-1})$ met

$$c_i = \sum_{j+k=i} a_j b_k .$$

Dus

$$c_0 = a_0 b_0, \quad c_1 = a_0 b_1 + a_1 b_0, \quad c_2 = a_0 b_2 + a_1 b_1 + a_2 b_0 .$$

Vul a en b met n nullen aan tot rijen ter lengte 2n. Dan geldt dat de convolutie van de oorspronkelijke a en b gelijk is aan de inverse Fouriertransformatie van het elementsgewijze product van de Fouriergetransformeerden van de (uitgebreide)a en b.

7. NP-volledige problemen

Het volgende probleem staat bekend als het partitieprobleem. Gegeven is een rij $a_1,\ldots,a_n$ van natuurlijke getallen. Gevraagd wordt een programma te schrijven dat bepaalt of er een verzameling $V \subset \{1,\ldots,n\}$ bestaat zodanig dat

$$\sum_{i\in V} a_i = \sum_{i\notin V} a_i \ .$$

Er is uiteraard een $O(2^n)$ algorithme, nl. de algorithme die alle mogelijke keuzen voor V beschouwt. Deze algorithme is exponentieel in n. Er is geen algorithme bekend die polynomisch is in n. Het is zelfs onbekend of voor dit probleem een polynomische oplossing mogelijk is. Het partitieprobleem is niet het enige probleem waarover een dergelijke onzekerheid bestaat, het behoort tot een grote klasse van "waarschijnlijk inherent-exponentiële" problemen: die der NP-volledige problemen [14].

7.1. De klassen P en NP

Als we voor een probleem slechts een exponentiële oplossing hebben is het probleem weliswaar opgelost, maar de oplossing is alleen bruikbaar voor kleine problemen. Het grote verschil tussen exponentieel en polynomisch blijkt bijv. als we in de tabel van p. 3 een kolom voor 2^n toevoegen:

n	2^n
1	2
5	32
10	1 024
20	$\sim 10^6$
50	$\sim 10^{15}$
100	$\sim 10^{30}$

En 10^{30} μs is 3.10^{16} jaar.

We kunnen programmeerproblemen indelen in klassen naar hun complexiteit:

P is de klasse der problemen die in een polynomische rekentijd opgelost kunnen worden.

NP is de klasse der problemen die met een orakel in een polynomische rekentijd opgelost kunnen worden. ("NP" staat voor nondeterministisch polynomisch).

Een orakel is een (hypothetisch) mechanisme dat in de aanwezigheid van nondeterminisme er voor zorgt dat steeds de "beste" keuze wordt gedaan. Teneinde aan het begrip "beste" inhoud te kunnen geven dient de verzameling der mogelijke antwoorden (eindtoestanden) volledig geordend te zijn. Een nondeterministische algorithme heeft voor een gegeven input (begintoestand) een aantal mogelijke antwoorden. In een berekening met een orakel wordt het nondeterminisme zó opgelost dat onder de mogelijke antwoorden de minimale (in de bovengenoemde ordening) wordt bereikt.

Deze definitie wijkt in zoverre af van de gebruikelijke, dat de klasse NP doorgaans beperkt wordt tot problemen die een boolean waarde als antwoord hebben. Als in de ordening het antwoord true vóór false komt, is onze definitie een generalisering van de gebruikelijke.

Het idee van het orakel is in de klasse NP die problemen te vangen die wel een polynomische oplossing zouden hebben als we maar steeds "de goede beslissing namen". Met een orakel kunnen we een speciaal blad vinden in een binaire boom van hoogte n in n stappen (inspecties van knooppunten), alhoewel de boom ongeveer 2^n bladen kan hebben.

Het is duidelijk dat $P \subset NP$. De interessante, nog onopgeloste, vraag is of $P = NP$. Zijn er problemen die wel tot NP en niet tot P behoren?
De verwachting is dat $P \neq NP$, omdat er een deelverzameling van NP is, die der NP-volledige problemen, waarvoor nog niemand door de jaren heen erin is geslaagd een polynomische oplossing te vinden. Nog een voorbeeld, waarvan we zullen aantonen dat het NP-volledig is, is het probleem of een gegeven ongerichte graaf een kliek (= deelgraaf waarin alle punten buurpunten zijn) van k elementen heeft.

Niet tot NP behoort bijv. het probleem alle maximale klieken van een ongerichte graaf te bepalen. (Een kliek is maximaal als hij geen deelgraaf van een andere kliek is.) Dit probleem kan zelfs met een orakel niet in een polynomische tijd opgelost worden omdat het aantal maximale klieken exponentieel kan zijn, en dus de tijdsduur voor het genereren ook.

7.2. NP-volledigheid

Met het begrip NP-volledigheid willen we onder de klasse NP de moeilijkste problemen karakteriseren.

Een probleem L is NP-volledig als L tot NP behoort en ieder probleem in NP polynomisch transformeerbaar is tot L.

Een probleem L_0 heet polynomisch transformeerbaar tot het probleem L als er een polynomische algorithme bestaat die input strings w_0 verandert in input strings w zodanig dat iedere oplossing (antwoord van een algorithme) van probleem L voor input w ook een oplossing is van L_0 voor input w_0.

Voorbeeld: We hebben een algorithme voor het partitieprobleem. De input is een rij natuurlijke getallen, het antwoord een boolean waarde. We willen het probleem "Heeft G een kliek van k elementen?" oplossen door polynomische transformatie. De input voor dat probleem is een codering van de graaf G en een waarde k. Wij schrijven nu een polynomische algorithme die die input (G en k) converteert tot een rij A van natuurlijke getallen, zodanig dat G een kliek van k elementen heeft als en slechts als A partitioneerbaar is. De oplossing van het kliekenprobleem bestaat dan uit twee stappen. Eerst de conversie, maar die vergt slechts een polynomische tijd, en dan de algorithme voor het partitieprobleem. Als die laatste algorithme polynomisch zou zijn dan hadden we nu ook een polynomische oplossing voor het kliekenprobleem.

Als we voor één NP-volledig probleem een polynomische algorithme vinden, dan hebben we daarmee, omdat ieder probleem in NP er polynomisch naar getransformeerd kan worden, polynomische oplossingen voor alle problemen in NP.

Het vinden van het eerste NP-volledige probleem is natuurlijk het moeilijkst. Zodra we weten dat L NP-volledig is kunnen we laten zien dat een ander probleem L_0 ook NP-volledig is door L polynomisch te transformeren tot L_0 (let op de richting van de transformatie), dan is nl. ieder probleem in NP via L tot L_0 te transformeren. (De compositie van twee polynomische transformaties is weer polynomisch.)

Cook vond in 1971 het eerste NP-volledige probleem, het "satisfiability problem" [15]:

Stel vast of voor een gegeven logische formule (uitgedrukt in "∧", "¬", haakjes en variabelen) er een toekenning van waarheidswaarden (true,false) aan de variabelen bestaat zodanig dat de hele formule de waarde true heeft.

Voorbeeld: a ∧ ¬ a is niet bevredigbaar, a ∧ b wel.

We gaan hier niet in op het bewijs dat het satisfiability problem NP-volledig is.

7.3. Enkele voorbeelden

Het satisfiability problem is ook NP-volledig als we ons beperken tot logische formules in conjuctieve normaalvorm (conjuctie van disjuncties van literals, waarin literals al dan niet ontkende variabelen zijn).

Voorbeeld: De formule

$$(x1 \vee x2) \wedge (x2 \vee \neg x1 \vee x3)$$

is in conjunctieve normaalvorm. De formule

$$x1 \wedge x2 \vee x3$$

niet.

Het satisfiability problem voor conjuctieve normaalvorm is polynomisch te transformeren tot het kliekenprobleem "Heeft G een kliek van k elementen?":

Zij $F = F_1 \wedge F_2 \wedge \ldots \wedge F_q$ met $F_i = x_{i1} \vee x_{i2} \vee \ldots \vee x_{ik_i}$,

waarin x_{ij} een al dan niet ontkende variabele is.

We transformeren F tot een graaf G met $\sum_{i=1}^{q} k_i$ punten, een punt per literal in F. Laat [i,j] het aan de literal x_{ij} toegevoegde punt voorstellen $(1 \leq i \leq q,\ 1 \leq j \leq k_i)$.

G heeft een tak tussen [i,j] en [k,ℓ] als $i \neq k$ en x_{ij} en $x_{k\ell}$ zijn niet v en ¬ v van eenzelfde variabele v. Dan geldt dat G een kliek van q elementen heeft d.e.s.d. als F bevredigbaar is. (Ga na.)

De bovenstaande conversie vergt slechts een polynomische rekentijd. Het kliekenprogramma is in NP: Neem een nondeterministisch programma dat een willekeurige deelgraaf van k elementen selecteert en de waarde true als antwoord geeft d.e.s.d. wanneer die deelgraaf een kliek is. Laat in de ordening de mogelijke antwoorden (true,false) true aan false voorafgaan. Dan vergt de vaststelling of G een kliek ter grootte k heeft met een orakel een polynomische rekentijd. Het kliekenprobleem is dus NP-volledig.

Het satisfiability problem voor conjuctieve normaalvorm is polynomisch te transformeren tot 3-satisfiability. (Als iedere factor uit ten hoogste k termen mag bestaan noemen we het probleem "k-satisfiability".) Hiertoe veranderen we iedere factor

$$x_1 \vee x_2 \vee \ldots \vee x_k , \quad k \geq 4$$

in de volgende conjunctie van disjuncties:

$$\begin{aligned}
&(x_1 \vee x_2 \vee y_1) \wedge \\
&(x_3 \vee \neg y_1 \vee y_2) \wedge \\
&(x_4 \vee \neg y2 \vee y_3) \wedge \\
&\qquad \vdots \\
&(x_{k-2} \vee \neg y_{k-4} \vee y_{k-3}) \wedge \\
&(x_{k-1} \vee x_k \vee \neg y_{k-3}) \,.
\end{aligned}$$

Hierin zijn $y_1, y_2, \ldots, y_{k-3}$ nieuwe variabelen. (Ga na dat de bovenstaande conjuctie bevredigbaar is d.e.s.d. als de oorspronkelijke factor bevredigbaar is.) Het probleem 3-satisfiability behoort duidelijk tot NP en is dus NP-volledig.

Opmerking: Voor 2-satisfiability bestaat een algorithme die lineair is in de lengte van de formule.

We noemen nog twee andere NP-volledige problemen: puntbedekking en cyclebedekking van een graaf. Een zeer uitgebreide lijst van NP-volledige problemen is te vinden in [14].

Een puntbedekking van een ongerichte graaf G is een verzameling S van punten van G zodanig dat iedere tak van G een punt van S als eindpunt heeft. Het puntbedekkingsprobleem is de vraag of voor gegeven G en k G een puntbedek-

king van k elementen heeft. De NP-volledigheid van dit probleem wordt aangetoond door het kliekenprobleem te transformeren met behulp van de eigenschap:

> S is een kliek in G d.e.s.d. als V \ S een puntbedekking van het complement van G is.

(V is de verzameling punten van G. Het complement van G is een graaf met dezelfde punten als G, maar met takken juist daar waar G geen takken heeft.)

Een cyclebedekking van een gerichte graaf D is een verzameling S van punten in D zodanig dat ieder cyclisch pad in D een punt van S bevat. Het cyclebedekkingsprobleem is weer de vraag of G een cyclebedekking van k elementen heeft. We transformeren het puntbedekkingsprobleem in een cyclebedekkingsprobleem door in de ongerichte graaf G van het puntbedekkingsprobleem iedere tak door een cyclisch pad van twee tegenovergestelde pijlen te vervangen. Als we de zo ontstane gerichte graaf D noemen geldt:

> S is een puntbedekking van G d.e.s.d. als S een cyclebedekking van D is.

8. Literatuurverwijzingen

[1] A.V. Aho, J.E. Hopcroft & J.D. Ullman; The Design and Analysis of Computer Algorithms. Addison-Wesley, 1974.

[2] E.W. Dijkstra; A Discipline of Programming. Prentice-Hall, 1976.

[3] Y. Muraoka & D.J. Kuck; On the time required for a sequence of matrix products. Comm. ACM 16, 1, 1973, pp. 22-26.

[4] R.W. Floyd; Algorithm 245: treesort 3. Comm. ACM 7, 12, 1964, p. 701.

[5] C.A.R. Hoare; Proof of a program FIND. Comm. ACM 14, 1, 1971, pp. 39-45.

[6] M. Blum, R.W. Floyd, V.R. Pratt, R.L. Rivest & R.E. Tarjan; Time bounds for selection. J. Computer and System Sciences 7,4,1972, pp. 448-461.

[7] R. Morris; Scatter storage techniques. Comm. ACM 11, 1, 1968, pp. 35-44.

[8] G.M. Adelson-Velskii & Y.M. Landis; An algorithm for the organization of information. Soviet Math. Dokl. 3, 1962, pp. 1259-1262.

[9] R. Bayer & E. McCreight; Organization and maintenance of large ordered indices. Acta Informatica 1, 3, 1972, pp. 173-189.

[10] R.E. Tarjan; Depth first search and linear graph algorithms. SIAM J. Computing 1, 2, 1972, pp. 146-160.

[11] S. Warshall; A theorem on Boolean matrices. J. ACM 9, 1, 1962, pp. 11-12.

[12] D. Gries; The Schorr-Waite graph marking algorithm. Acta Informatica, 1979, pp. 223-232.

[13] J.M. Cooley & J.W. Tukey; An algorithm for the machine calculation of complex Fourier series. Math. Comp. 19, 1965, pp. 297-301.

[14] M.R. Garey & D.S. Johnson; Computers and Intractability: A Guide to the Theory of NP-Completeness. Freeman, 1979.

[15] S.A. Cook; The complexity of theorem proving procedures. Proc. 3rd Annual ACM Symposium on Theory of Computing, 1971, pp. 151-158.