Algoritmen en datastructuren 3, 2017

Gunnar Brinkmann

27 september 2017

Inhoudsopgave

1 Introductie

De bedoeling van deze tekst is **niet** in plaats van de les te worden gebruikt maar alleen om te helpen de eigen nota's misschien een beetje beter te verstaan als er iets niet echt duidelijk is. Alleen in de les kan je onmiddellijk vragen stellen als je iets niet verstaat en alleen in de oefeningen kan je toetsen of je het geleerde kan toepassen...

2 Algoritmen voor slim gebruik van het geheugen

Een tijdje geleden was er in DA2 een project waar verschillende *AVL-achtige* bomen geïmplementeerd moesten worden. De datastructuur was identiek en de routines die sleutels zochten waren ook dezelfde. Volgens onze ideeën over tijdsverbruik zou dus het aantal bezochte toppen voor een reeks van opzoekoperaties een goede manier zijn om de tijdscomplexiteit te meten. Maar de resultaten waren als volgt (2.000.000 sleutels in de boom en 20.000.000 keer een toevallige sleutel opzoeken):

algoritme	tijd voor opbouwen	aantal bezochte toppen (zoeken)	tijd voor zoeken
1	3.75	426.500.000	33.3
2	8.07	420.300.000	37.6

Hoewel de boom dus beter gebalanceerd was en dezelfde routines de opzoekingen deden, vroeg het meer tijd om de sleutels op te zoeken – er klopte dus iets niet met ons model. Een idee was dat het iets met de cache te maken had. De slechter gebalanceerde boom werd minder herbalanceerd en misschien was de manier waarop het geheugen gealloceerd werd beter omdat toppen die dicht bij elkaar in de boom zitten ook dicht bij elkaar in het geheugen zitten en dus samen in de cache plaats kunnen vinden.
Om dat te testen werden de toppen van de bomen na het toevoegen in het geheugen verplaatst zodat ten minste de toppen die dicht bij de wortel waren allemaal ook dicht bij elkaar zaten. De structuur van de bomen was nog dezelfde (de bomen waren isomorf aan de bomen voor het *verhuizen*). Het resultaat was dat beide algoritmen (nog steeds met dezelfde routines) sneller waren – maar deze keer was de verhouding (ongeveer) zoals verwacht:

algoritme	aantal bezochte toppen (zoeken)	tijd voor zoeken
1	426.500.000	27.7
2	420.300.000	26.8

De modellen die wij gebruikt hebben om te bepalen of een algoritme efficiënt is of niet zijn dus in dit geval niet echt nauwkeurig – en hier gaat het alleen maar om het verschil tussen de cache en het geheugen van de computer. Zodra er zoveel bestanden zijn of de records zo groot zijn dat de boom op de harde schijf wordt bijgehouden, zijn het de leesoperaties op de harde schijf die de snelheid bepalen en niet het aantal vergelijkingen.
De 2-3-bomen die wij in DA2 hebben gezien, gaan al een stukje in deze richting. Als wij het aantal toppen als het aantal leesoperaties zien dan

hebben 2-3-bomen een voordeel in vergelijking met alle binaire zoekbomen omdat de diepte het aantal leesoperaties en daardoor de snelheid bepaalt. Door de grotere vertakking hebben 2-3-bomen een kleinere diepte dan binaire zoekbomen (behalve in het ene geval waar een 2-3-boom een complete binaire boom kan zijn) en dat gaat nu ook onze strategie zijn: de vertakking zo groot mogelijk maken zodat zo weinig mogelijk leesoperaties nodig zijn.
Daarbij gaat het niet alleen om zoekbomen – je hebt ook een vertakking als het bv. om recursie gaat. Ons eerste voorbeeld gaat er één zijn waar de vertakking de recursie beschrijft en niet de datastructuur.

2.1 Hashing

Hashing is een efficiënte manier om sleutels op te slaan en op te zoeken. Het probleem is alleen dat er botsingen kunnen zijn – dan kan het gebeuren dat hashing niet meer efficiënt is. Je kan botsingen voorkomen door een hash-tabel te kiezen die voldoende groot is om de kans op botsingen klein te houden. Het probleem is dat je natuurlijk niet te veel ruimte wil verspillen en dat je voor de geziene manieren om met hashtabellen te werken op voorhand moet vastleggen hoe groot de tabel is.
Het zou dus mooi zijn als je de tabel zou kunnen uitbreiden tijdens het gebruik als je ziet dat er te weinig ruimte is. Het onderwerp van dit deel zijn twee manieren om met hashtabellen te werken die dat op een efficiënte manier toelaten.

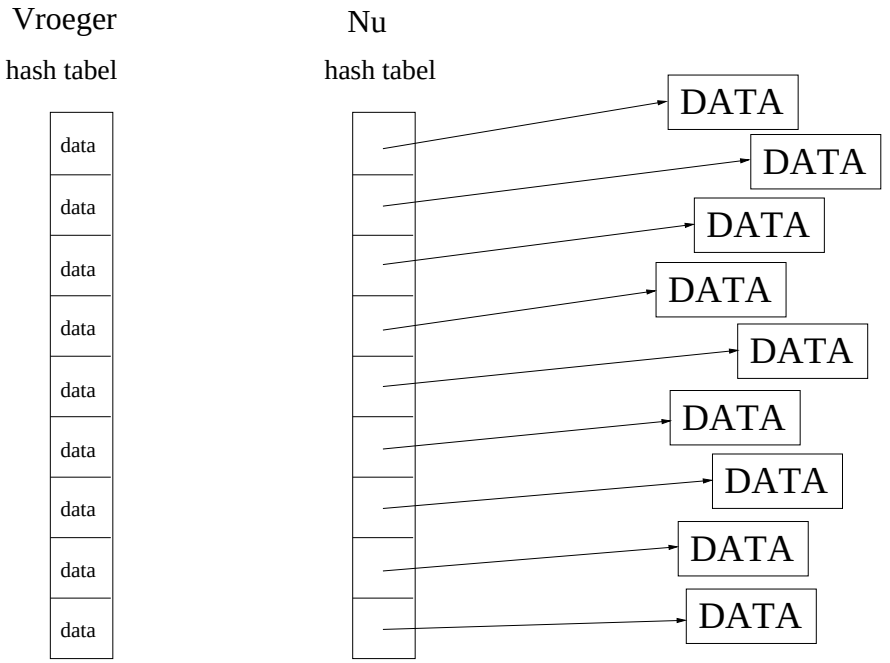
Oefening 1 *Stel dat je closed hashing wil toepassen. Om geen probleem met te veel botsingen te hebben, wordt elke keer dat de laadfactor boven de* 70% *ligt de hashtabel uitgebreid. Wij hebben al in DA2 gezien dat het uitbreiden van een array niet efficiënt kan gebeuren als je elke keer een vast aantal vakken toevoegt maar dat het nodig is de grootte met een getal (in de meeste gevallen* 2*) te vermenigvuldigen. Stel dat je hier (omdat je toch al ruimte verspilt) als factor maar* 1.5 *kiest. Uitbreiden betekent hier wel dat de tabel niet alleen gekopieerd moet worden, maar ook dat voor alle sleutels de hashfunctie opnieuw berekend moet worden.*
Stel voor het volgende dat er geen botsingen zijn:

- *Hoe duur is een toevoegbewerking in het slechtste geval?*
- *Hoe duur is een reeks van n toevoegbewerking op een initieel lege hashtabel met 100 elementen in het slechtste geval? (Herinnering: dat hebben wij ook de geamortiseerde kost van deze reeks van bewerkingen genoemd.)*

In dit deel zijn we er vooral in geïnteresseerd als de records die je wil opslaan heel groot zijn – zo groot dat je ze niet in het geheugen kan houden maar dat je ze op de harde schijf moet plaatsen. Over hoe dat precies gebeurt gaan wij het niet hebben – dat laten wij gewoon over aan de controller van de harde schijf en aan het besturingssysteem. Bovendien hebben jullie daar al iets over gezien. Wij willen het aantal lees/schrijf-operaties op de harde schijf beperken. Een bestand op de harde schijf kan je – anders dan een array werkt in het geheugen – langer maken zonder het te moeten kopiëren (dat zou heel veel leesoperaties vragen). Wij werken hier gewoon met een model waarvan wij stellen dat je een geïndexeerde lijst van *vakjes* hebt zodat je elk vakje in één stap kan lezen en in één stap er iets in schrijven. Bovendien stellen wij dat je de lijst langer kan maken en dat de kosten om er een vakje aan toe te voegen dezelfden zijn als voor een gewone schrijfoperatie. Hoe het besturingssysteem ervoor zorgt dat dat kan, doet er **in dit geval** niet toe.
Een model is nuttig als je gebaseerd op dit model snelle algoritmen ontwikkeld, waarbij *snel* hier de echte tijd in de prktijk betekent – dat is altijd ons doel. Ook al zijn in ons model sommige dingen zeker te gemakkelijk voorgesteld, blijkt het toch een in de praktijk nuttig model.

2.1.1 Extendible hashing

Extendible hashing werkt met een model dat een beetje verschilt van het gewone model waar je de records (de data) direct in een tabel opslaat. Wij werken met een tabel van pointers die naar de echte data wijzen. In een reële toepassing kan deze *echte data* natuurlijk ook maar een deel van de echte data zijn of alleen een identificatienummer dat nog een pointer naar de hele record bevat – dus de echte data alleen maar representeert.
Als je de records in het geheugen kan houden, kan deze manier van werken ook voordelen hebben als je niet met extendible hashing werkt: het zou kunnen dat je moeilijkheden hebt een samenhangend geheugenblok te krijgen om alle records te indexeren. Als je met pointers werkt, is dat natuurlijk geen probleem. Bovendien zou het ook efficiënter zijn de array groter te maken: Wij zouden alleen de pointers moeten kopiëren en niet de (veel grotere) records. Maar zonder verdere wijzigingen zouden wij de data nog altijd moeten rehashen of tenminste lezen en dat is natuurlijk ook vrij duur – en als het over de harde schijf gaat onaanvaardbaar.
Maar nu gaan wij het over het geval hebben waar de records op de harde schijf zitten.
De array met de pointers kan zeker nog in het geheugen gehouden worden, dus hebben wij daarvoor geen leesoperaties op de schijf nodig.
Wij werken met buckets (emmers) waarbij in elke bucket een constant en



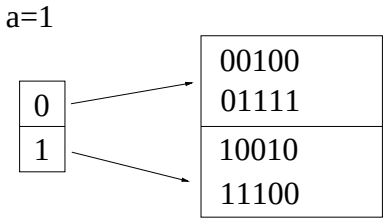
Figuur 1: Een hash tabel en een hash tabel met pointers naar buckets.

vast aantal records terecht kan komen – bv. 5. De reden is dat de emmers blokken voorstellen die je in één stap kan lezen. Het feit dat je dan misschien de gezochte record nog moet zoeken als er meerdere records per emmer zijn, doet er in ons model niet toe: dat gebeurt in het geheugen en kan dus efficiënt gedaan worden. Wij werken met een hash-functie $h()$ waarvan wij stellen dat de waarden tussen 0 en $2^n - 1$ voor een zekere n liggen en dat ze zeker voldoende groot zijn om goed te kunnen hashen – bv. waarden tussen 0 en $2^{32} - 1$. Bovendien stellen wij dat de hashfunctie altijd met hetzelfde aantal n bits voorgesteld wordt – dus misschien ook met nullen in het begin. Wij zullen in onze voorbeelden altijd de hash-waarden in de emmers opslaan. In de realiteit zijn dat natuurlijk de records en niet de hashwaarden, maar zo kan je zien wat er precies gebeurt terwijl artificiële data die wij voor de voorbeelden zouden gebruiken niet echt zou bijdragen tot het verstaan van de principes. Je zou in de realiteit bovendien **ook** nog de hashwaarden kunnen opslaan om die niet te moeten herberekenen, maar omdat nooit veel data opnieuw gehasht moet worden en omdat wij alleen de lees- en schrijfbewerkingen willen minimaliseren, maakt dat geen (groot) verschil.
Arrayindices schrijven wij ook vaak in binaire vorm omdat de methodes de binaire voorstelling van de indices gebruiken. Natuurlijk zijn dat ook gewone getallen – alleen dat ze binair voorgesteld worden om gemakkelijker te

verstaan in welk vakje een record terecht moet komen.

- De grootte van onze pointerarray is altijd een macht van 2 – wij gebruiken de variabele a om dat bij te houden, de grootte is 2^a.

Wij evalueren altijd een zeker aantal – dezelfde a – bits van onze hashfunctie. Inderdaad gebruiken wij dus niet $h()$ om de index van het juiste vakje in onze tabel te vinden maar $(h())|_a$. Wij schrijven altijd $x|_a$ voor het getal dat door de eerste a bits in de binaire voorstelling van x voorgesteld wordt, waarbij wij misschien met nullen in het begin werken om ervoor te zorgen dat de voorstelling van elk getal hetzelfde aantal bits heeft. Als $x = 010011$ in binaire voorstelling (dus 19) dan is $x|_3 = 010$ dus gelijk aan 2. Als wij a bits evalueren, hebben wij 2^a mogelijke waarden $0, \ldots, 2^a - 1$ die de indices van onze (pointer) hash-tabel vormen. *In principe* kunnen wij dus pointers naar 2^a buckets bijhouden.
Een voorbeeld met $a = 1$ zien jullie in Figuur 2



Figuur 2: Extendible hashing met $a = 1$ en twee records per emmer.

Maar inderdaad zullen wij niet alle buckets wijzigen als de pointerarray groter gemaakt wordt. Dat is **heel** belangrijk omdat het veel dure leesoperaties zou vragen en de uitbreidingsstap onaanvaardbaar duur zou maken. Dat betekent dat wij voor sommige buckets niet a bits moeten evalueren maar minder. Wij houden dus voor elke bucket b een getal a_b bij dat zegt hoeveel bits voor deze bucket werden geëvalueerd toen hij aangemaakt werd. Het is duidelijk dat als er meer pointers zijn dan buckets dat sommige pointers naar dezelfde bucket moeten wijzen. Figuur 4 toont een situatie waar er één bucket al werd uitgebreid en één niet. Inderdaad zullen er altijd 2^{a-a_b} pointers naar een bucket b wijzen.

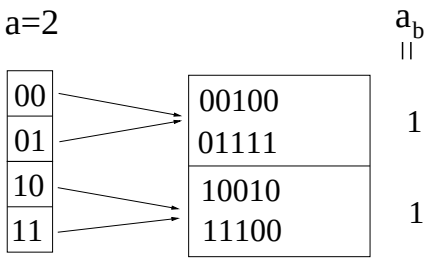
Maar nu dat wij al ongeveer weten hoe de situatie er kan uitzien, moeten wij precies beschrijven hoe extendible hashing werkt:
Je begint met een vaste a. In onze voorbeelden beginnen wij met $a = 1$ omdat het anders vrij moeilijk is om te tekenen, maar in de realiteit zal a zeker ten minste 10 zijn. Wij hebben dan een pointerarray met 2^a pointers

die indices $0, \ldots, 2^a - 1$ hebben en naar 2^a buckets wijzen waarin wij g records kunnen plaatsen. Voor de voorbeelden kiezen wij $g = 2$. Voor elke emmer b is in het begin $a_b = a$. In het begin – waar voor elke emmer geldt dat $a = a_b$ – wijzen dus inderdaad $2^{a-a_b} = 1$ pointers naar elke emmer.

- Toevoegen en opzoeken gebeurt op de volgende manier: je berekent de hashwaarde $h(r)$ van de record r die je wil toevoegen of opzoeken en kijkt in de tabel waar vakje $(h(r))|_a$ naartoe wijst. Daar moet je record r plaatsen of opzoeken. Opzoeken is nooit een probleem maar als je de record wil toevoegen en de emmer is vol is er wel een probleem...

Als de emmer b waar je iets wil toevoegen volzit, willen wij een nieuwe emmer gebruiken en de inhoud verdelen. Maar voor 2 emmers hebben wij ook 2 pointers nodig. Er zijn twee mogelijkheden:

$a_b = a$**:** In dit geval is er maar één pointer die naar deze emmer wijst. Dat is het slechtste geval. Wij moeten dus eerst onze pointerarray uitbreiden om ervoor te zorgen dat ten minste 2 pointers naar deze emmer wijzen. Wij verdubbelen de pointerarray. De variabele a is nu 1 groter (dus $a = a^{oud} + 1$). Als de nieuwe pointerarray $p_n[]$ is en de oude $p_o[]$ dan vullen wij $p_n[]$ met dezelfde pointers als $p_o[]$: voor $0 \leq i < 2^a$ geldt $p_n[i] = p_o[\lfloor i/2 \rfloor]$. Het resultaat voor de eerste keer verdubbelen zien jullie in Figuur 3. Nu wijzen dus twee keer zo veel pointers naar elke emmer als voor het verdubbelen.



Figuur 3: De pointerarray werd verdubbeld en werkt met $a = 2$ maar de emmers werken nog met $a_b = 1$.

Wij hebben er dus voor gezorgd dat wij altijd in de volgende situatie terechtkomen:

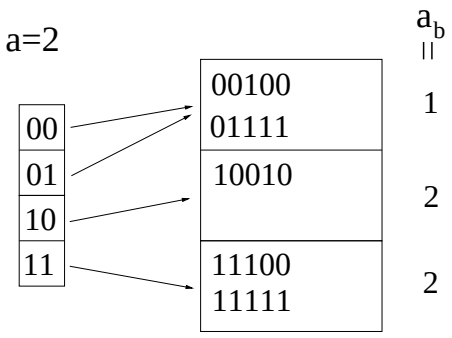
$a_b < a$**:** Dus wijzen nu $2^{a-a_b} \geq 2$ pointers naar de volle emmer b waar wij de nieuwe record willen plaatsen. De eerste a_b bits van de hashwaarde van

elke record zijn gelijk – stel dat het $b_1 \dots b_{a_b}$ zijn. Wij maken nu een nieuwe emmer b' aan en verdelen de records uit b zo dat alle records r met bit nummer $a_b + 1$ gelijk aan 0 (of $2 \cdot (h(r)|_{a_b}) = h(r)|_{a_b+1}$) in de oude emmer b terechtkomen. De eerste $a_b + 1$ bits van de hashwaarden van deze records zijn dus "$b_1 \dots b_{a_b}0$". De anderen (bit nummer $a_b + 1$ gelijk aan 1, dus $2 \cdot (h(r)|_{a_b}) + 1 = h(r)|_{a_b+1}$ en de eerste $a_b + 1$ bits van de hashwaarde zijn "$b_1 \dots b_{a_b}1$") plaatsen wij in de nieuwe emmer b'. Dan plaatsen wij de nieuwe record volgens dezelfde regel (als dat kan...). De pointers in de pointerarray met een index i waarvoor geldt dat $i|_{a_b+1} = 2 \cdot (h(r)|_{a_b}) + 1$ (de eerste $a_b + 1$ bits van de index zijn "$b_1 \dots b_{a_b}1$") moeten nu gewijzigd worden zodat ze naar b' wijzen. De waarden van a_b en $a_{b'}$ worden de oude waarde van a_b plus 1.

Oefening 2

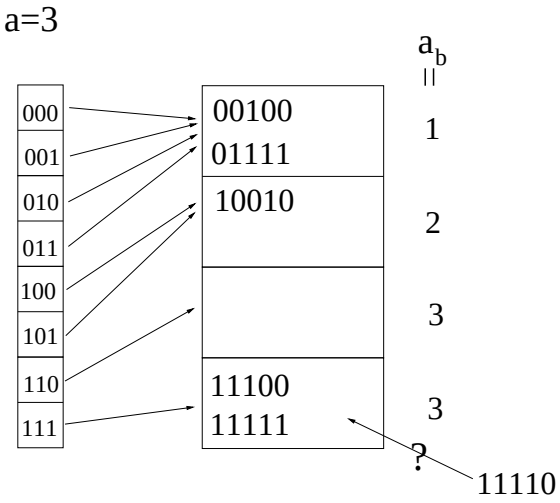
- *Bereken uit a, a_b en $h(r)$ de indices van de pointers die gewijzigd moeten worden. Geef een formule.*
- *Beschrijf expliciet hoe je met bitoperaties de index in de array berekent en hoe je met bitoperaties kan beschrijven welke pointer uit de oude array naar een vakje in de nieuwe gekopieerd moet worden.*

In Figuur 4 zien jullie het resultaat als wij op deze manier 11111 toevoegen: de emmer waarop pointer 11 wees moest herverdeeld worden en omdat $h(r)|_{a_b} = 1$ (alleen het eerste bit wordt geevalueerd van 11111) wordt pointer $2 \cdot 1 + 1 = 3$ gewijzigd en wijst naar de nieuwe emmer.



Figuur 4: Extendible hashing met $a = 2$ maar één emmer werkt nog met $a_b = 1$.

Maar inderdaad kan er een probleem zijn. Wij hebben de oude records in de emmer en de nieuwe record herverdeeld over twee emmers. Maar als wij pech hebben, kan het gebeuren dat bij het herverdelen één emmer leeg blijft en de andere dan nog altijd geen vrije ruimte voor de nieuwe record heeft.



Figuur 5: Extendible hashing met $a = 3$ na het verdubbelen maar toch kan een record nog niet toegevoegd worden.

Dat zou bv. gebeuren als je nu sleutel 11110 toevoegde. Figuur 5 toont de datastructuur na het dubbelen en herverdelen van de volle emmer.
In dit geval moeten wij nog eens verdubbelen en hopen dat het probleem dan opgelost is. Hashing kan natuurlijk altijd problemen veroorzaken als door een slechte hash-functie – of gewoon heel veel pech – de waarden slecht verdeeld zijn. Dit zou ook hier de oorzaak zijn. Met een goede hash-functie en niet te weinig records per emmer zal de load in de praktijk gelijkmatig verdeeld zijn en dus de pointerarray niet overbodig groot zijn. Als in één emmer x records terecht kunnen komen en de hash-functie kent meer dan x keer dezelfde waarde toe, zou het uitbreiden oneindig doorgaan (of precies: het zou doorgaan tot er geen bits meer zijn om te evalueren en zou dan vaststellen dat de methode niet werkt). Een goede hashfunctie is dus heel belangrijk.

Het belangrijke voordeel qua efficiëntie is dat je alleen maar naar de emmer moet kijken die te vol zit. Dat zorgt ervoor dat je misschien met maar één leesoperatie en twee schrijfoperaties kan werken als de grootte van de emmers zo is dat hij in één stap gelezen kan worden. Naar de andere emmers moet je zelfs niet kijken. Het uitbreiden van de pointerarray gebeurt in het geheugen – dat tellen wij dus niet mee als het om lees/schrijf-operaties gaat. Bovendien moet de array natuurlijk **veel** minder vaak gedubbeld worden dan emmers herverdeeld moeten worden – als er m keer een emmer herverdeeld moet worden dan is het aantal dubbeloperaties bij een gelijkmatige verdeling van de hashwaarden van de orde $\log m$. Maar het dubbelen kan ook op een

(tenminste geamortiseerd) efficiënte manier gebeuren. Of precies:

Oefening 3 *Deze oefening is niet gemakkelijk (vraag de assistent je tips te geven als je niet weet hoe je moet beginnen) maar het is wel een nuttige oefening om een beetje DA2 te herhalen en vooral om zichzelf duidelijk te maken hoe extendible hashing werkt:*
Het verdubbelen van de pointerarray lijkt natuurlijk sterk op het verdubbelen van een array, dat jullie in DA2 hebben gezien. Maar er zijn toch verschillen: hier wordt de array direct volgeschreven en bovendien worden de records niet gekopieerd en de samenhang tussen het aantal records en de grootte van de pointerarray is ook niet zo duidelijk.

- *Wat is de geamortiseerde kost van een reeks van n bewerkingen op een initiëel lege extendible hashing tabel als je geen voorwaarden aan de hashfunctie oplegt (behalve natuurlijk dat ze oneindig veel bits genereert – anders werkt het niet voor arbitrair grote n)?*

- *Gegeven $q > 0$. Wat is de geamortiseerde kost van een reeks van n bewerkingen op een initiëel lege extendible hashing tabel als gegarandeerd is dat altijd als er een tabel met p pointers uitgebreid wordt er ook ten minste $p \cdot q$ records zijn? De $O()$ notatie is voldoende maar een bewijs is (natuurlijk) vereist. (Inderdaad zou het voldoende zijn dat voor de datastructuur na n stappen te eisen.)*

Oefening 4 *Stel dat je met extendible hashing ook identieke records – dus zeker met gelijke sleutels – wilt opslaan. Wanneer kan dat en wanneer is er een probleem ? Hoe kan je het probleem oplossen?*

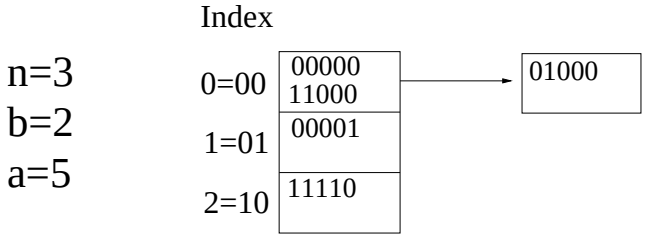
Oefening 5 *Voeg de sleutels 3, 7, 12, 45, 44 en 1 toe aan een lege extendible hashing tabel met twee emmers in het begin. Er kunnen 2 sleutels (records) per emmer geplaatst worden.*
Wij gebruiken een voorstelling van de hashcodes van $h()$ als binaire getallen met 5 bits en wij hebben
$h(3) = 0$ *dus binair* 00000
$h(7) = 31$ *dus binair* 11111
$h(12) = 9$ *dus binair* 01001
$h(45) = 17$ *dus binair* 10001
$h(44) = 2$ *dus binair* 00010
$h(1) = 1$ *dus binair* 00001

2.1.2 Linear hashing

In het geval van extendible hashing kan een enkele uitbreidingsstap *in principe* wel relatief duur zijn – ook al is de geamortiseerde kost (met een goede hashfunctie) heel goed. Het effect van een slechte hashfunctie – of heel veel pech – is dat meerdere keren uitgebreid moet worden of in het ergste geval zelfs de grens van de uitbreidingsmogelijkheden bereikt wordt. Ook het verdubbelen van de pointerarray kan je voor zeer grote arrays niet meer verwaarlozen. Linear hashing gebruikt uitbreidingsstappen waar het slechtste geval goedkoper is. De klemtoon ligt hier op de laadfactor en niet op een te volle emmer. Je moet afhankelijk van de toepassing beslissen of extendible hashing of linear hashing (of gewoon hashing) beter is.
Hier gebruiken wij het voordeel van ons model van de harde schijf dat de tabel zonder te kopiëren uitgebreid kan worden. Het is dus mogelijk een enkele emmer toe te voegen zonder de andere emmers te moeten kopiëren.
De belangrijkste bedoeling is ook hier tijdens het uitbreiden het aantal lees- en schrijf-operaties op de harde schijf te minimaliseren – dus naar zo weinig mogelijk emmers te moeten kijken.
Ook hier zullen wij in de voorbeelden de hash-keys in de emmers plaatsen omdat dat gemakkelijker is om te verstaan. In een toepassing zitten daar natuurlijk de records.

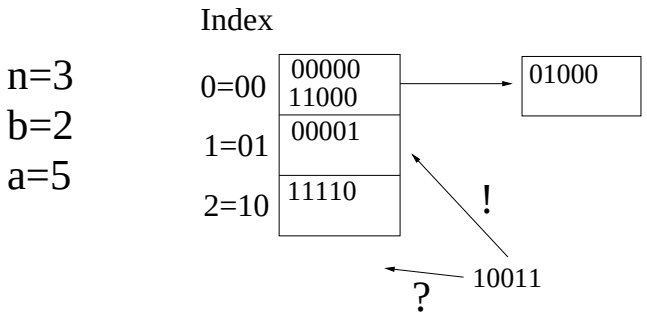
- Je houdt altijd bij hoeveel emmers je hebt (dat noemen wij altijd n) en hoeveel elementen in de array zitten (dat noemen wij a). Het aantal records dat je in één emmer kan plaatsen noemen wij g (voor grootte).

Een groot verschil met extendible hashing is dat wij er hier niet op letten of een emmer te vol zit of niet. Als iets niet meer geplaatst kan worden, werken wij gewoon met overstroomemmers. Natuurlijk maakt dat hashen misschien minder efficiënt – vooral als voor de overstroomemmers een extra leesoperatie nodig is – maar als de hashfunctie goed en de laadfactor $a/(n \cdot g)$ niet te groot is, zullen er normaal niet veel overstroom-emmers zijn. Als wij schrijven dat in een *emmer* gezocht moet worden, bedoelen wij inderdaad de hele emmerketting – dus de emmer samen met zijn overstroomemmers. Ons doel is dus de laadfactor relatief klein te houden om efficiënt hashen te kunnen garanderen.
Wij gebruiken opnieuw de binaire voorstelling van de hash-waarden. Als er n emmers zijn, hebben wij $b(n) = \lceil \log n \rceil$ bits nodig om de indices van de emmers voor te stellen. Hier gebruiken wij de **laatste** b bits van de hashfunctie $h()$ en schrijven $x|^b$ als wij de laatste b bits van de binaire voorstelling van een getal x willen gebruiken. Wij stellen altijd dat $n \geq 2$ en dus $b \geq 1$.



Figuur 6: Een kleine linear hashing tabel.

Wij proberen een record r in emmer $h(r)|^b$ te plaatsen. Het enige probleem dat zou kunnen opduiken is dat $h(r)|^b \geq n$. Alleen als n een macht van twee is, is $b(n) = \log n$. Dan kan dat niet gebeuren – anders wel!
Als je in ons voorbeeld een record met hashwaarde 10011 wil plaatsen, neem je de laatste $b = 2$ bits, dus 11 – het record moet dus in vakje 3 geplaatst worden – maar dat is er niet. In zulke gevallen plaatsen wij het record in vakje $h(r)|^b \mod 2^{b-1} = h(r)|^{b-1}$.



Figuur 7: Het plaatsen van een element waar de eerste keuze voor de index te groot is.

De functie $h_n(r)$ die de index bepaalt waar het element r geplaatst moet worden is dus gegeven door (merk op dat b gewoon een functie van n is):

$$h_n(r) = \begin{cases} h(r)|^b & \text{als } h(r)|^b < n \\ h(r)|^{b-1} & \text{als } h(r)|^b \geq n \end{cases}$$

Op deze manier hebben wij een manier van hashing die ook zonder uitbreiden kan werken en door de overstroomemmers ook met een grote laadfactor nog werkt – hoewel de efficiëntie dan slechter wordt. Het lineaire zoeken in de overstroomemmers zou misschien veel leesoperaties vragen.
Maar nu gaan wij beschrijven hoe wij ervoor zorgen dat de laadfactor $a/(n \cdot g)$ nooit groter wordt dan een bovengrens L die wij op voorhand vastleggen. Het getal $n \cdot g$ is het grootste aantal records dat je kan plaatsen zonder

overstroomemmers. Hoe dichter a dus bij $n \cdot g$ ligt hoe groter de kans dat je overstroomemmers moet gebruiken. Als $a > n \cdot g$ is het zelfs zeker dat je overstroomemmers nodig hebt! Wij zullen dus normaal $L < 1$ kiezen – bv. $L = 0.7$.
Stel nu dat wij een nieuw element toevoegen en daardoor de laadfactor te groot wordt, dus $a/(n \cdot g) > L$. Dan moeten wij onze tabel uitbreiden. Daardoor wordt n groter en de laadfactor kleiner. Wij nemen er altijd maar één nieuw element bij.
Maar dat betekent natuurlijk dat onze hashfunctie $h_n(r)$ verandert! Wij moeten dus bepalen welke records r nu op een foute plaats zitten dus waarvoor geldt $h_{n_{\text{nieuw}}}(r) \neq h_{n_{\text{oud}}}(r)$ of als n voor de oude n staat: $h_{n+1}(r) \neq h_n(r)$. Er zijn twee gevallen: ofwel b verandert ook – ofwel niet.

$b(n) = b(n+1)$**:** Wij schrijven gewoon b voor $b(n) = b(n+1)$. Als je naar de definitie van $h_n(r)$ kijkt, kan een verschil er alleen door veroorzaakt zijn dat verschillende gevallen van de definitie worden toegepast. Dat gebeurt precies voor $h|^b(r) = n = n_{\text{oud}}$. Vroeger was de waarde $n|^{b-1}$ en nu is hij n. De enige records die op een foute plaats kunnen zitten, zijn dus de records in emmer $n|^{b-1}$. Die moeten herverdeeld worden.

$b(n) = b(n+1) - 1$**:** Dus $n = n_{\text{oud}} = 2^{b_{\text{oud}}}$ Wij schrijven b voor b_{oud}. Nu worden $b+1$ bits van $h(r)$ geëvalueerd (dus $h(r)|^{b+1}$ gebruikt). Als de eerste bit 0 is, geldt $h(r)|^{b+1} = h(r)|^b < n < n+1$. Records met deze hashwaarde werden dus vroeger en nu op dezelfde manier geplaatst. Maar als de eerste bit (die voor 2^b staat) 1 is, geldt $h(r)|^{b+1} \geq 2^b = n$. Als $h(r)|^{b+1} > 2^b = n$ dan geldt $h(r)|^{b+1} \geq n+1$ – dus is $h_{n+1}(r) = h(r)|^b = h_n(r)$ (omdat in dit geval altijd geldt dat $h(r)|^b < n$ – zelfs als alle bits 1 zijn.). Records met deze hashwaarde moeten dus niet herverdeeld worden. Het blijft alleen maar $h(r)|^{b+1} = 2^b = n$ dus $h(r)|^b = 0$ om herverdeeld te worden.

Je hoeft in de twee gevallen dus geen verschillende formules te gebruiken – in elk geval (behalve het uitzonderlijke geval $b_{\text{oud}} = 0$ dat je voor echte toepassingen natuurlijk nooit hebt) is de emmer die herverdeeld moet worden emmer $n_{\text{oud}}|^{b_{\text{oud}}-1}$. De twee gevallen die wij besproken hebben, zijn gewoon bewijzen dat deze formule in beide gevallen geldt.
De emmer die herverdeeld wordt, moet niet bijzonder vol zitten – hij kan (in principe) zelfs helemaal leeg zijn!

Oefening 6 *Als je veronderstelt dat inderdaad de hashwaarden goed verdeeld zijn, is dan de kans dat er in de emmerketting die herverdeeld moet worden*

meer dan gemiddeld veel records zitten (de ketting dus langer is) dezelfde als de kans dat er minder dan gemiddeld veel records zitten?
Er is geen expliciete berekening vereist, maar geef wel voldoende uitleg waarom jouw antwoord juist is. Bespreek de gevallen dat n *een macht van* 2 *is – resp. dat niet is – apart.*

Maar belangrijk voor de efficiëntie: maar **één** emmerketting moet herverdeeld worden – en die kan misschien met één enkele leesoperatie ingelezen worden als de hashfunctie en de laadfactor ervoor zorgen dat niet veel overstroomemmers nodig zijn. Als je de schrijfoperaties meetelt zijn er dus maar 2 emmerkettingen die gewijzigd moeten worden.
Een voorbeeld van de manier waarop linear hashing werkt, zien jullie in Figuur 8

Oefening 7 Dit is een bijzonder belangrijke oefening voor linear hashing:
Stel dat je een goede hashfunctie hebt en dat de reeks van hashwaarden goed verdeeld is en elk van de 2^b *mogelijke waarden van* $h()|^b$ *ongeveer dezelfde kans maakt om op te duiken. (Om het iets gemakkelijker te hebben mag je stellen dat het aantal sleutels een macht van twee is die groter dan* 2^b *is en dat elke index* **precies** *even vaak opduikt.) Zitten de emmers dan altijd allemaal even vol?*

Als ja: *bewijs dat.*

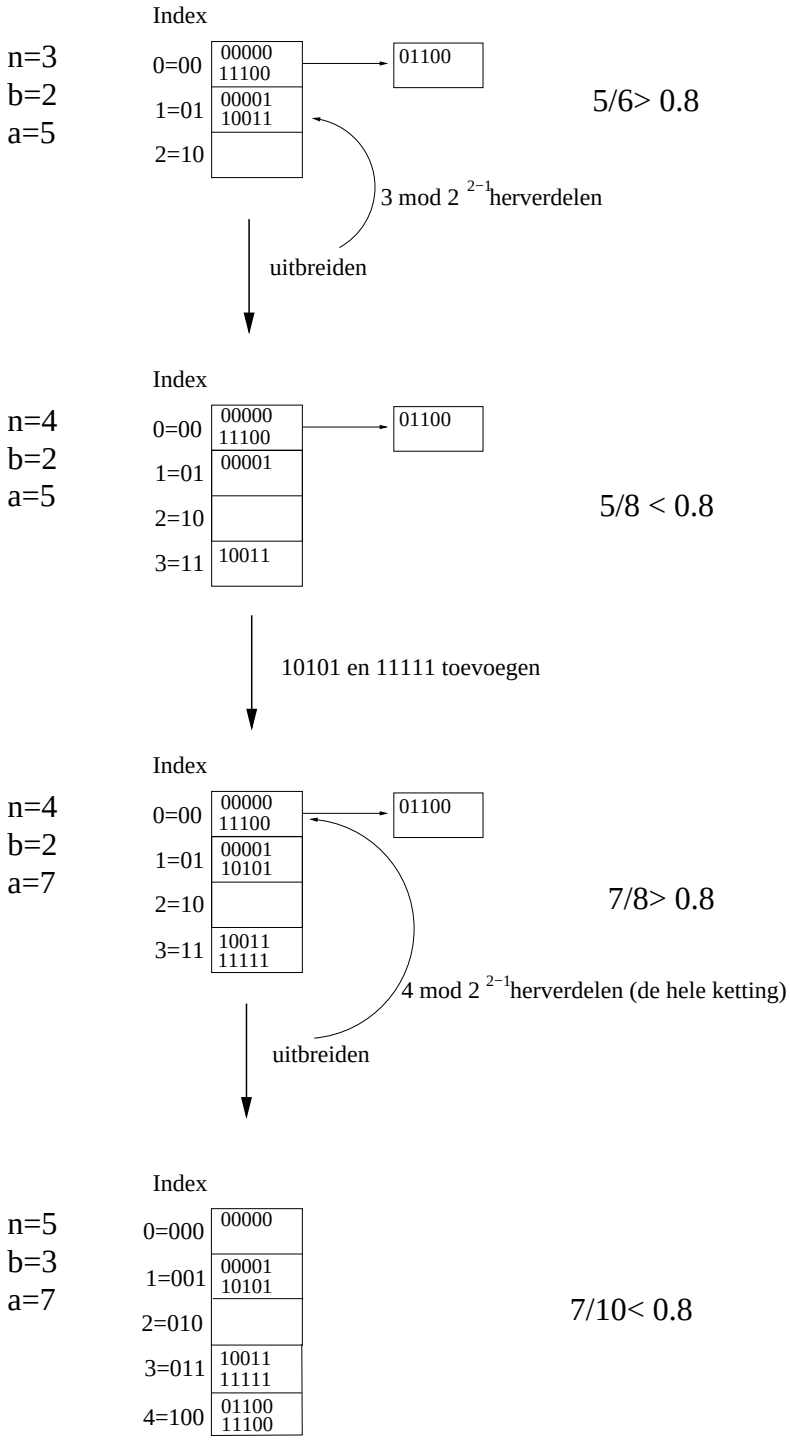
Als niet:
- *Wanneer zijn ze wel even vol?*
- *Welke emmers zijn voller en hoe groot is het verschil?*

Geef uitleg.

Oefening 8 *Geef voldoende voorwaarden op de reeks van sleutels die toegevoegd moeten worden en de hash-functies om te garanderen dat linear hashing voor een reeks van* n *toevoegbewerkingen gemiddeld constante tijd vraagt!*
Geef uitleg!

Oefening 9 *Voeg de sleutels 3, 7, 12, 45, 44, 11 en 1 toe aan een lege linear hashing tabel met twee emmers in het begin. Er kunnen 2 sleutels (records) per emmer geplaatst worden en de tabel wordt uitgebreid zodra de laadfactor groter is dan* 0.8.
De hashcodes zijn als volgt:
$h(3) = 0$ *dus binair* 0
$h(7) = 31$ *dus binair* 11111

$h(12) = 9$ *dus binair* 1001
$h(45) = 17$ *dus binair* 10001
$h(44) = 2$ *dus binair* 10
$h(11) = 8$ *dus binair* 1000
$h(1) = 1$ *dus binair* 1



Figuur 8: Deze linear hashing tabel wordt uitgebreid zodra de laadfactor groter is dan $L = 0.8$.

2.1.3 Bloom filters

Natuurlijk is het mooi als je een hashing tabel kan uitbreiden als je meer geheugen nodig hebt – maar wat als je gewoon te veel geheugen nodig hebt?

Het volgende algoritme werd in 1970 door Burton Bloom gepubliceerd. Toen was het onmogelijk alle juiste spellingen van woorden in het geheugen op te slaan en het algoritme werd gebruikt om ook met minder geheugen een goede spelling checker te implementeren. Intussen is het beschikbare geheugen zeer veel groter dan toen, maar er zijn nog altijd problemen waar het belangrijk is het geheugengebruik zo klein mogelijk te houden. Soms moet je bv ook informatie over een netwerk sturen en dan is de reductie van de hoeveelheid geheugen die deze informatie codeert ook essentieel! Wij zullen wel zien dat de reductie van geheugengebruik een prijs heeft...
Stel dat je met een universum $\{0, \ldots, n-1\}$ werkt. Als je een grote verzameling van elementen uit dit universum moet bijhouden, dan hebben wij al in Algoritmen en Datastructuren 2 gezien hoe je dat kan doen: je werkt met een bitvector en zet bit nummer i op 1 als en slechts als element i in de verzameling zit. Dat vraagt maar 1 bit voor elk element uit het universum. Deze manier van doen werkt natuurlijk ook als de elementen niet $\{0, \ldots, n-1\}$ zijn, maar je wel een unieke index $\{0, \ldots, n-1\}$ aan elk element kan toevoegen.
Eén manier om indices aan objecten toe te kennen is een hashfunctie. Stel dat wij een universum U van objecten hebben. Dit kunnen bv. alle mogelijke combinaties van ten hoogste 20 lettertekens zijn, maar het kunnen ook alle strings zijn – dus een oneindige verzameling. Stel bovendien dat wij een hashfunctie $h : U \to \{0, \ldots, n-1\}$ hebben. Als wij nu voor een verzameling $M \subseteq U$ (bv. alle juist gespelde woorden in de Nederlandse taal) willen toetsen of een zeker element x in M zit, kunnen wij als volgt werken:

- Wij nemen een bitvector $b_0, \ldots, b_{n-1}$ met n bits die in het begin allemaal 0 zijn.
- Dan wordt voor elk element $y \in M$ de bit $b_{h(y)}$ op 1 gezet.
- Als wij willen toetsen of een element $x \in U$ in M zit, dan berekenen wij $h(x)$ en zeggen *ja* als bit $b_{h(x)} = 1$ en anders *nee*.

Als wij dit algoritme toepassen en wij testen een element $x \in M$ dan zal het antwoord *ja* zijn – dat klopt dus. Als wij een element $x \notin M$ testen dan kan het antwoord nee zijn, maar er is ook een kans dat toevallig een ander element dat wel in M zit dezelfde hashwaarde heeft en wij dus het (foute) antwoord

ja krijgen. Er is dus een kans op false positives, dus foute ja-antwoorden. Je zou de antwoorden als *nee* en *misschien* kunnen interpreteren – dan heb je altijd een juist antwoord, maar aan *misschien* heb je vaak niets... Wij zullen zien hoe je de kans op foute ja-antwoorden kleiner kan maken, maar Bloom filters zijn dus zeker niet geschikt voor gevallen waar een fout ja-antwoord niet achteraf nog door een andere test herkend kan worden (wij zullen een voorbeeld zien waar dat wel kan) en dan misschien rampzalige gevolgen heeft. Als het om een spelling checker gaat dan is een programma dat je een beetje helpt zeker al beter dan helemaal geen hulp – daarvoor waren de Bloom filters dus zeker geschikt. Een andere toepassing waarvoor Bloomfilters voorgesteld werden, was het opslaan van een lijst van onvoldoende veilige passwoorden. Ook hier was een fout ja-antwoord niet erg: je zou gewoon een ander passwoord moeten kiezen, terwijl foute nee-antwoorden wel erg zouden kunnen zijn – maar foute nee antwoorden heb je bij Bloom filters niet.
Het idee om de kans op false positives te reduceren is eenvoudig: gebruik niet één hashfunctie $h()$ maar meerdere hashfuncties $h_1(), \ldots, h_k()$ en geef enkel *ja* als antwoord als voor elke hashfunctie $h_i()$ geldt dat $b_{h_i(x)} = 1$.
Daarbij kan je op twee manieren werken als je n bits ter beschikking hebt:

a.) Als n deelbaar is door k gebruik je voor alle hashfuncties aparte bitvectoren van n/k bits. De functie $h_1()$ gebruikt dus bit $0, \ldots, n/k - 1$, de functie $h_2()$ gebruikt bit $n/k, \ldots, 2(n/k) - 1$, etc.

b.) Je gebruikt voor alle hashfuncties dezelfde n bits. Het is dus mogelijk dat dezelfde bit door verschillende hashfuncties gezet wordt.

Wij zullen hier alleen maar de (standaard) manier b.) analyseren. Daarbij zullen we de kans berekenen dat een toevallig gekozen element het antwoord *ja* krijgt. Deze kans gebruiken wij hier als benadering voor de kans op een fout positief antwoord. Voor universa die groot zijn in vergelijking met M is dat zeker een aanvaardbare benadering en voor andere gevallen zou je Bloom filters vermoedelijk toch al niet toepassen...
Op voorhand is daarbij niet duidelijk dat het goed is met meer dan 1 hashfunctie te werken. Als je bv. met extreem veel hashfuncties zou werken, dan zouden bijna alle bits 1 kunnen zijn en je krijgt zeker bijzonder veel foute positieve antwoorden. Is $k = 1$ misschien zelfs optimaal?
Wij gebruiken dat $e^x = \lim_{i\to\infty}(1 + \frac{x}{i})^i$ en dat voor voldoende grote i geldt dat

$$e^x \approx (1 + \frac{x}{i})^i \tag{1}$$

Bovendien veronderstellen wij dat alle hashfuncties de waarden gelijk verdelen, of precies: wij veronderstellen dat als je een toevallig element $x \in U$ kiest de kans dat $h_i(x) = j$ voor alle $1 \leq j \leq n$ gelijk is aan $\frac{1}{n}$ en dat voor elke verzameling M die we beschouwen de kansen voor alle elementen onafhankelijk van elkaar zijn. Dit zijn veronderstellingen die typisch zijn voor hashfuncties en door realistische hashfuncties (en verzamelingen) goed benaderd worden. Wij veronderstellen ook dat de hashfuncties onafhankelijk zijn – dus dat elementen die van de ene hashfunctie een zekere waarde i krijgen door de andere hashfuncties nog altijd gelijkverdeeld zijn.
Als wij één element met één hashfunctie in onze Bloom filter invullen, dan is de kans dat een zekere bit achteraf nog altijd 0 is dus $1 - \frac{1}{n}$. Nadat we k hashfuncties hebben toegepast, is de kans $(1-\frac{1}{n})^k$ en nadat wij k hashfuncties op $m = |M|$ elementen hebben toegepast, is de kans $(1 - \frac{1}{n})^{km}$.
Als wij in vergelijking 1 voor x de term $-\frac{km}{n}$ invullen en voor i de term km (waarvan wij hier veronderstellen dat het voor de gekozen x groot genoeg is), dan krijgen wij als kans dat een bit na het invullen van alle elementen nog altijd 0 is

$$(1 - \frac{1}{n})^{km} \approx e^{\frac{-km}{n}} \tag{2}$$

De kans dat een bit achteraf 1 is, is dus (ongeveer) $(1 - e^{\frac{-km}{n}})$.
Omdat wij veronderstellen dat ook de hashfuncties voldoende onafhankelijk zijn, kunnen wij de kansen op een fout positief antwoord schatten: de kans dat een toevallig gekozen element een positief antwoord krijgt (dus dat alle k bits 1 zijn) is ongeveer $(1 - e^{\frac{-km}{n}})^k$.
Als wij de verzameling M (dus ook $m = |M|$) en het ter beschikking staande geheugen (dus n) als gegeven beschouwen, is de vraag hoe we k moeten kiezen om de kans op foute ja-antwoorden zo klein mogelijk te hebben. Dat is *in principe* heel gemakkelijk: pas jouw kennis uit het middelbaar toe en bereken het minimum van $f(k) = (1-e^{\frac{-km}{n}})^k$ waarbij je n en m als constanten beschouwt.
Met twee trucjes wordt het iets gemakkelijker:
Wij hebben $f(k) = (1 - e^{\frac{-km}{n}})^k = e^{k(\ln(1-e^{(-km)/n}))}$. Als wij schrijven

$$g(k) = k(\ln(1 - e^{(-km)/n})) \tag{3}$$

dan heeft $f(k)$ een minimum als en slechts als $g(k)$ er één heeft (omdat e^x strict monotoon stijgt). Als wij nu ook nog schrijven $p = e^{(-km)/n}$ dan wordt

$$g(k) = g_1(p) = \frac{-n}{m}(\ln p)(\ln(1 - p)). \tag{4}$$

De functie $g(k)$ heeft dus een minimum als $g_1(p(k))$ dat heeft en die heeft een minimum als $g_2(p) = (\ln p)(\ln(1-p))$ een maximum heeft (omdat wij een negatieve multiplicatieve constante weggelaten hebben).
Daarvoor kunnen wij nu onze kennis uit het middelbaar onderwijs toepassen:

$$\frac{\mathrm{d}g_2(p)}{\mathrm{d}p} = \frac{\ln(1-p)}{p} - \frac{\ln p}{1-p} = \frac{((1-p)\ln(1-p)) - (p \ln p)}{p(1-p)} \tag{5}$$

en dat is 0 voor $p = \frac{1}{2}$. Met een beetje meer rekenen kunnen wij ook nog zien dat $p = \frac{1}{2}$ het enige extremum is en in feite een maximum van $g_2(p)$. Met $p = \frac{1}{2} = e^{(-km)/n}$ krijgen wij dat $f(k)$ een minimum heeft voor $k = \frac{n}{m} \ln 2$.
Als m en n gegeven zijn, kiezen we dus het beste $k = \frac{n}{m} \ln 2$ onafhankelijke hashfuncties om een Bloom filter te implementeren. Natuurlijk moet het aantal een geheel getal zijn, zodat wij dit getal moeten afronden, waarbij wij de voorkeur geven aan $\lfloor \frac{n}{m} \ln 2 \rfloor$ omdat minder hashfuncties natuurlijk ook minder tijd vragen.
Als je dus bv. een verzameling met 1.000.000 elementen wilt opslaan en je hebt 1 MB ter beschikking, dus 8.000.000 bit, dan gebruik je het best $\lfloor \frac{8.000.000}{1.000.000} \ln 2 \rfloor = 5$ hashfuncties.

Oefening 10

- *Stel dat je voor een verzameling M met m elementen een Bloom filter wilt maken waar de kans op een fout positief antwoord ten hoogste c is. Natuurlijk werk je met het optimale aantal hashfuncties. Ontwikkel de formule die het aantal bits beschrijft dat je nodig hebt om een Bloom filter te hebben die aan deze eisen voldoet.*

- *Hoeveel bits heb je nodig om een verzameling met $m = 1.000.000$ elementen door middel van een Bloom filter voor te stellen als je ten hoogste een kans van 0.02 (dus 2%) op een fout positief antwoord wil hebben?*

Maar het feit dat $p = \frac{1}{2}$ is ook nog om andere redenen interessant: $p = e^{(-km)/n}$ is de kans dat een bit bij deze optimale keuze achteraf nog 0 is en omdat $p = \frac{1}{2}$, is de kans dat een bit 0 is dus gelijk aan de kans dat de bit 1 is. De structuur van de Bloom filter is dus die van een gelijkverdeelde random string...

Oefening 11 *Veronderstel nu dat je zoals in a.) werkt en de k hashfuncties op disjuncte domeinen met grootte n/k werken. Bereken de kans op foute positieve antwoorden voor gegeven k, n, m en ook de optimale waarde voor het aantal k van hashfuncties dat je moet kiezen om de kans op foute positieve antwoorden zo klein mogelijk te maken als n, m gegeven zijn.*

Oefening 12

- *Als je twee bitmap voorstellingen $b_M, b_{M'}$ van verzamelingen M, M' voor hetzelfde universum hebt, dan kan je door de bitsgewijze* `OR`*-operator ($|$) de verzameling $M \cup M'$ berekenen: $M \cup M' = b_M | b_{M'}$. Klopt dat ook voor Bloom filters $b_M, b_{M'}$ die twee verzamelingen M, M' voorstellen? Of precies: is het resultaat van $b_M | b_{M'}$ altijd de bloomfilter voor $M \cup M'$ als het gebruikte aantal bits en de gebruikte hashfuncties dezelfden zijn? Geef uitleg.*

- *Als je twee bitmap voorstellingen $b_M, b_{M'}$ van verzamelingen M, M' voor hetzelfde universum hebt, dan kan je door de bitsgewijze* `AND`*-operator ($\&$) de verzameling $M \cap M'$ berekenen: $M \cap M' = b_M \& b_{M'}$. Klopt dat ook voor Bloom filters $b_M, b_{M'}$ die twee verzamelingen M, M' voorstellen? Of precies: is het resultaat van $b_M \& b_{M'}$ altijd de bloomfilter voor $M \cap M'$ als je hetzelfde aantal bits en dezelfde hashfuncties neemt? Geef uitleg.*

Oefening 13 *Stel dat je met een Bloom filter met $n = 2^i$ bits werkt voor een zekere $i \in \mathbb{N}, i > 1$. Je stelt vast dat je n te groot hebt gekozen en beter met $n/2$ zou werken, waarvoor het aantal fout positieve antwoorden ook nog aanvaardbaar zou zijn. Beschrijf een manier om een Bloom filter voor $n/2$ te berekenen zonder alle elementen te rehashen. Welke hash functies gebruik je?*

Bloom filters zijn niet alleen nuttig als foute positieve antwoorden niet erg zijn – zoals in het voorbeeld met de slechte passwoorden. Soms is het ook mogelijk achteraf foute positieve antwoorden snel te verbeteren, zoals in het volgende voorbeeld:
In een netwerk heeft een computer A informatie over alle personen die voor een zeker (groot) bedrijf werken en een andere computer B informatie over alle adressen in Europa waar er in de toekomst gebouwd moet worden. Computer B moet nu weten welke personen van het bedrijf op een plaats wonen waar er in de toekomst gebouwd wordt. Eén mogelijkheid zou zijn de hele lijst van plaatsen door te sturen, maar die zou waarschijnlijk te groot zijn. Dus wordt een Bloom filter naar computer A doorgestuurd waarna computer A de vermoedelijk kleine lijst van mensen die op zo'n plaats wonen terugstuurt. De weinige *false positives* – dus personen die niet op een plaats wonen waar er gebouwd moet worden – kunnen dan snel verwijderd worden. In dit voorbeeld wordt dus de klemtoon gelegd op weinig data die over het netwerk gestuurd moet worden, omdat dat vaak een bottleneck is.
Natuurlijk is het een voor de hand liggend idee dat je compressiealgoritmen gebruikt om Bloom filters te comprimeren voordat je ze stuurt – ten slotte is weinig geheugen het hoofddoel van Bloom filters. Jammer genoeg hebben

wij net gezien dat in Bloom filters met een minimale kans op een fout positief antwoord elke bit met kans $p = \frac{1}{2}$ gelijk aan 0 of gelijk aan 1 is – het zijn dus random strings die niet of nauwelijks comprimeerbaar zijn. In 2001 heeft Mitzenbacher iets heel interessants aangetoond: als je jouw Bloom filter niet optimaliseert voor een minimale kans op een fout positief antwoord **voor** het comprimeren, maar **na** het comprimeren, dan kan je soms Bloom filters krijgen die $n' > n$ bits gebruiken en dezelfde kans op een fout positief antwoord hebben als het optimum voor n bits – maar beter comprimeren en na compressie minder dan n bits nodig hebben. Misschien iets om na te lezen...

2.2 Sorteren van grote hoeveelheden data – extern sorteren

Als je meer data moet sorteren dan je in het geheugen kan houden, kunnen sorteeralgoritmen zoals quicksort die in het geheugen heel goed presteren inefficiënt worden omdat er plotseling een verschil is tussen de verschillende manieren waarop je een element moet lezen – als je een element moet lezen dat al in het geheugen staat is dat veel goedkoper als een element te lezen dat nog van de harde schijf gelezen moet worden. In dit deel zullen wij een algoritme zien dat het aantal leesoperaties op de harde schijf minimaliseert. Onze oplossing zal een mergesort algoritme zijn. Maar in plaats van het in elke recursiestap in maar twee delen te splitsen, zullen wij het op elk niveau in meer delen splitsen waarbij wij rekening zullen houden met de grootte van het geheugen en de grootte van de blokken die je in één keer kan lezen.

basisstap: je splitst het hele bestand op zodat je delen $d_1, \ldots, d_n$ hebt die **net** in het geheugen gesorteerd kunnen worden. Deze delen worden efficiënt in het geheugen gesorteerd (bv. door middel van mergesort of quicksort) en teruggeschreven.

Als je bv. 10GB moet sorteren en je kan 500MB in het geheugen sorteren (waarvoor je natuurlijk meer dan 500MB geheugen nodig hebt) zonder te moeten swappen, zou je 20 delen maken en die in het geheugen lezen, sorteren en dan terugschrijven. Zo wordt elk blok met data één keer gelezen en één keer geschreven.

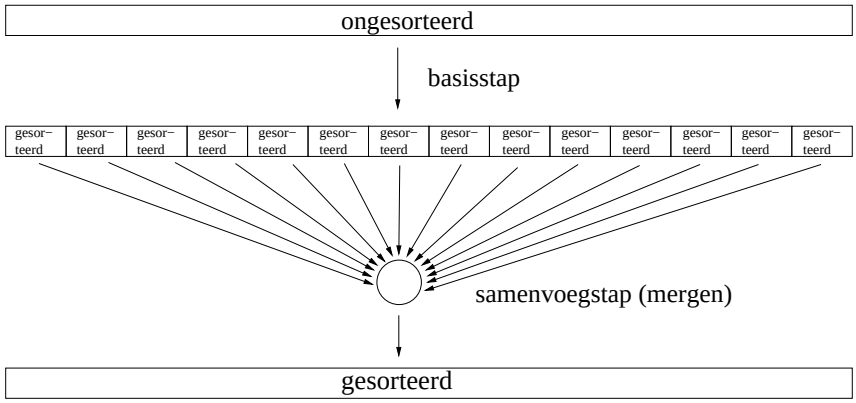
samenvoegstap: Nu moeten de gesorteerde delen nog gemerged worden. Wij lezen van zoveel mogelijk delen het eerste blok in het geheugen. *Zoveel mogelijk* betekent dat wij maar zoveel delen kiezen dat wij voor elk deel één blok in het geheugen kunnen houden. Dan schrijven wij altijd de kleinste sleutel van één van de delen in een outputblok tot dat

volzit en uitgeschreven wordt. Als één van de blokken leeg is, wordt dat door het volgende blok van dat deel vervangen. Dit samenvoegen tot één nieuw gesorteerd deel noemen wij één mergeoperatie.

Als wij t delen tegelijk kunnen mergen en er zijn samen d delen dan passen wij deze mergeoperatie $\lceil d/t \rceil$ keer toe totdat elk deel één keer met andere delen samengemerged werd. Wij hebben dan $\lceil d/t \rceil$ grotere gesorteerde delen.

In deze stap wordt elk blok opnieuw één keer gelezen en één keer geschreven.

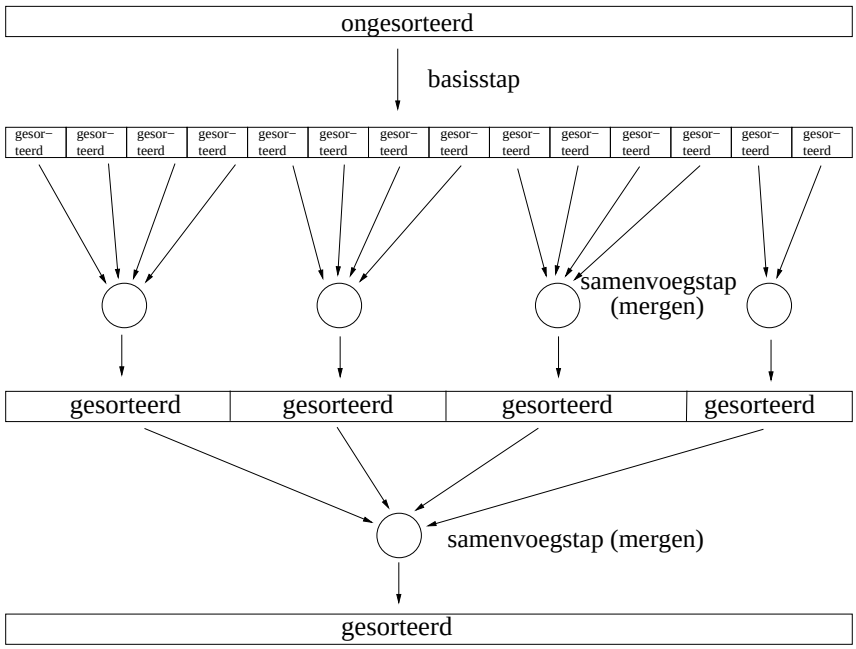
Deze samenvoegstap wordt herhaald tot er maar één bestand overblijft.
Het principe voor één samenvoegstap zien jullie nog eens in Figuur 9 en in Figuur 10 zien jullie een voorbeeld met een heel kleine vertakking van 4. Daardoor zijn er twee samenvoegstappen nodig. Realistische voorbeelden kan je natuurlijk niet tekenen...



Figuur 9: Het principe van extern sorteren.

Stel dat B de blokgrootte is die je in één stap kan lezen en M de grootte van het geheugen dat je voor het opslaan van de delen ter beschikking kan stellen. Dan kan je dus $\frac{M}{B}$ delen per mergeoperatie mergen, zodat het aantal delen voor de volgende samenvoegstap met de factor $\frac{M}{B}$ daalt. Omdat in het begin de delen grootte M mogen hebben, kan je dus in s samenvoegstappen bestanden van grootte $M * (\frac{M}{B})^s$ sorteren.
Als je er opnieuw van uitgaat dat je 500MB ter beschikking hebt en dat bv. een blok van 4KB in één stap gelezen kan worden, zijn dat al 62.5 Terabyte die je kan sorteren met maar één enkele samenvoegstap. Dat is inderdaad heel veel en ook al wordt de hoeveelheid data groter – het geheugen groeit ook en de 500 MB waarvan wij hier gesteld hebben dat die ter beschikking staan, zijn zeker een benedengrens...

Wij kunnen dus met 2 lees- en schrijf-operaties per blok 62.5 TB sorteren.



Figuur 10: Extern sorteren in 2 samenvoegstappen. In dit voorbeeld kunnen maar 4 delen tegelijk gemerged worden.

Wat wij hier gezien hebben is dus gewoon mergesort waar wij aan de basis niet alleen enkele sleutels hebben maar al grotere reeds gesorteerde delen. En het belangrijkste: de vertakking is **veel** groter. De vertakking $\frac{M}{B}$ bepaalt de diepte van de recursieboom en de diepte van de recursieboom bepaalt het aantal lees- en schrijf-operaties.
Wij hebben er niet op gelet wat je bv. moet doen als de records die je wil sorteren niet in één blok passen etc. In dergelijke gevallen zijn lichte wijzigingen nodig.

Oefening 14 *Wij hebben geschreven* „Dan schrijven wij altijd de kleinste sleutel van één van de delen in een outputblok. . . “. *Hoe doe je dat het best op een efficiënte manier? Altijd naar het kleinste element in elk opgeslagen blok kijken zou lineair zijn in het aantal blokken. Kan dat efficiënter? Denk aan DA1 en DA2.*

Oefening 15 *Gegeven de grootte M van het geheugen dat ter beschikking staat en B de blokgrootte die je in één stap kan lezen. Geef een formule voor het aantal leesoperaties (per blok) op de harde schijf dat nodig is om een bestand met n bytes te sorteren.*

Oefening 16 *De bedoeling van deze oefening is (nog) een beetje meer gevoel te krijgen voor welke betekenis de asymptotische analyse heeft en in welke gevallen ze meer of minder belangrijk is.*
Stel dat een lees/schrijf-operatie op de harde schijf kost C_1 per blok heeft en een operatie in het geheugen (vergelijken, verplaatsen, etc.) kost C_2. De motivatie voor de laatste algoritmen die wij hebben gezien was dat $C_1 >> C_2$ – dus mag je dat ook hier stellen.
Je hebt een vast geheugen van 400 *MB ter beschikking en een bestand van n records waarvan elke record een grootte van* 400 *bytes heeft. Een blok heeft grootte* 4 *KB.*
Onderzoek drie varianten van extern sorteren: één keer sorteer je de deelbestanden die net in het geheugen passen met mergesort, één keer met bubblesort en één keer met quicksort. In elk geval pas je achteraf de mergebewerking op zoveel niveau's toe als er noodzakelijk zijn.

- *Zal er een groot verschil in performantie zijn als je deze 3 varianten in de praktijk op een bestand van bv.* 10 *GB (dus* 25000000 *records) toepast?*
- *Bepaal de asymptotische complexiteit van alle drie varianten van extern sorteren.*

2.3 Grote zoekbomen

Waarschuwing in het begin: in de literatuur vind je B-trees en B+-trees soms op verschillende manieren gedefinieerd. De ideeën zijn natuurlijk altijd dezelfde als jullie ze hier zullen zien, maar de details zijn soms een beetje verschillend – dus opgelet als jullie op verschillende plaatsen kijken.
In dit deel zullen wij het nu over een echte vertakking hebben – het gaat over bomen. In zoekbomen is het aantal kinderen van een top altijd één groter dan het aantal sleutels die de top bevat (waarbij de kinderen leeg kunnen zijn). Als wij een boom hebben waar het kleinste aantal niet lege kinderen van een top die geen blad is g is (elke interne top moet dan ten minste $g-1$ sleutels bevatten en stel dat dat ook voor de bladeren geldt) en alle bladeren zitten op dezelfde diepte d dan kunnen in deze boom ten minste $g^{d+1}-1$ sleutels geplaatst worden. Om n sleutels te plaatsen, heb je dus een diepte van ongeveer $\log_g(n) = \frac{1}{\log g}\log n$ nodig. Als de diepte het aantal leesoperaties beschrijft, moeten wij er dus voor zorgen de graad van de toppen zo groot mogelijk te kiezen. En precies dat zullen wij doen: wij proberen een boom met een vertakking te bouwen die zo groot mogelijk is – onder de voorwaarde dat wij elke top nog met een enkele leesoperatie kunnen lezen en de top dus nog in één blok geplaatst kan worden.

2.3.1 B-trees

Wij herhalen in de definitie van een B-tree de ordeningseigenschappen in de toppen van de boom in plaats van gewoon te zeggen dat het een zoekboom is, zodat de definitie beter met die van een B+-tree vergeleken kan worden.

Definitie 1 *Wij stellen hier dat een boom een sleutel maar één keer kan bevatten.*
Een B-tree met (even) grootte n is een boom met volgende eigenschappen:

- *het aantal a van sleutels in de wortel voldoet aan $1 \leq a \leq n$*
- *het aantal a van sleutels in toppen die niet de wortel zijn, voldoet aan $\frac{n}{2} \leq a \leq n$*
- *als in een top t de sleutels $s_1 < s_2 < \cdots < s_a$ zitten dan geldt*
  - *Ofwel is t een blad (en heeft dus geen kinderen) ofwel heeft t precies $a+1$ (niet lege) kinderen $k_1, \ldots, k_{a+1}$ waarvoor met $s_0 = -\infty$ en $s_{a+1} = \infty$ voor alle sleutels s in de deelboom met wortel k_i geldt dat $s_{i-1} < s < s_i$.*
- *alle bladeren zitten op dezelfde afstand van de wortel (dezelfde diepte).*

Het is dus duidelijk een uitbreiding van de definitie van een 2-3-boom en ook de bewerkingen lijken heel sterk op die van een 2-3-boom:

zoeken: Voor elke top zoek je of de sleutel in die top zit of in welk kind je moet doorgaan met het zoeken. Voor grote n doe je dat het best door middel van logaritmisch zoeken.

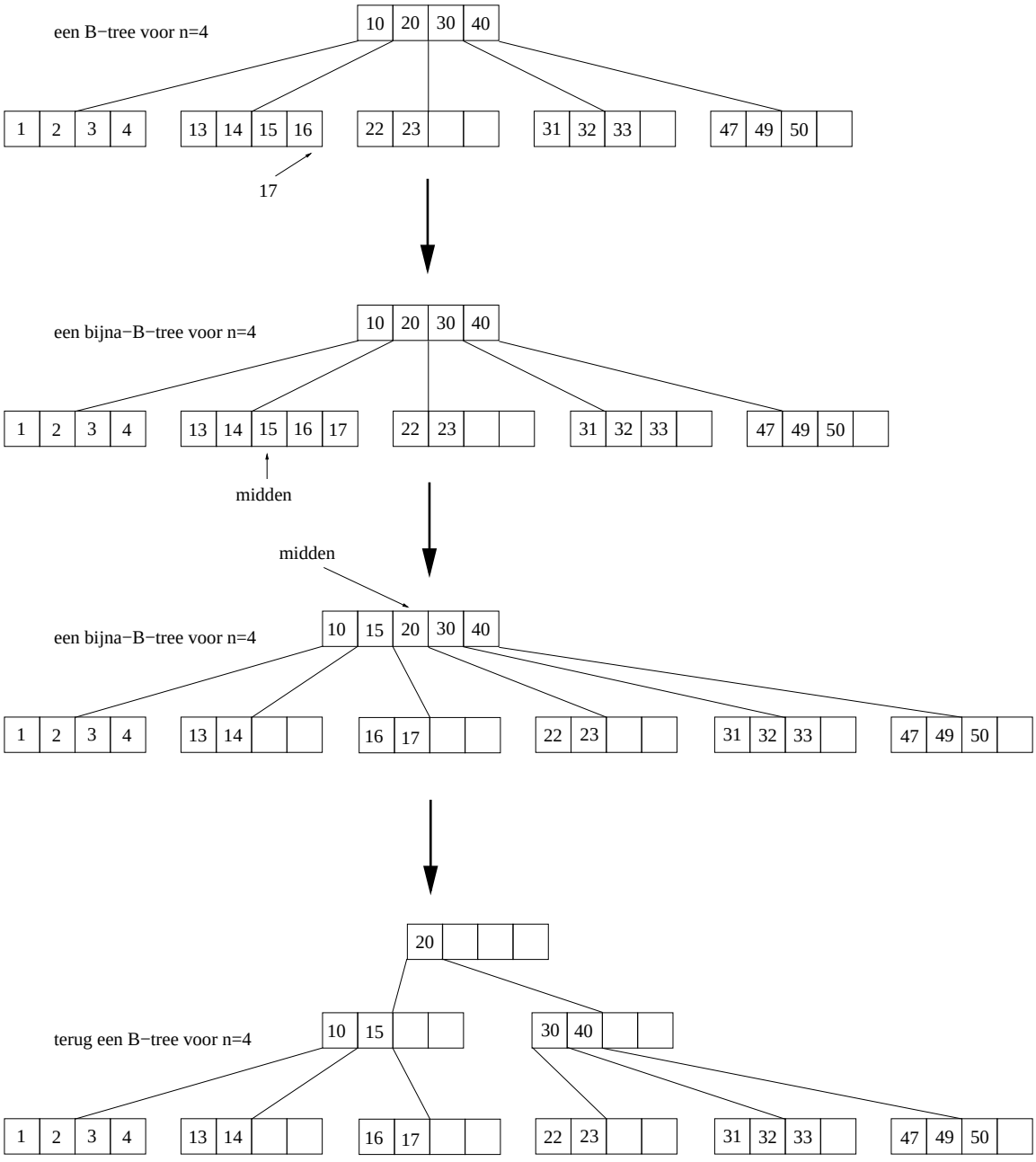
toevoegen: Eerst zoek je het blad waar de sleutel toegevoegd moet worden.

- Als er nog plaats is, voeg je hem gewoon toe (waarbij de sleutels in het blad gesorteerd moeten blijven).
- Anders bevat de top al n sleutels, dus samen met de nieuwe $n+1$ sleutels. Wij werken nu tijdelijk met een boom waar **één** top $n+1$ sleutels bevat (dat noemen wij hier een bijna-B-tree) en die herbalanceren wij dan tot hij opnieuw aan de eigenschappen van een B-tree voldoet.

Het principe is daarbij gemakkelijk om te verstaan: omdat $n+1$ sleutels niet in één top passen, heb je er twee nodig – de ouder zal dus één kind meer hebben. Maar omdat het aantal kinderen van de ouder bepaald wordt door

het aantal sleutels, moet ook één sleutel naar boven schuiven om ervoor te zorgen dat er één kind meer is.

Een voorbeeld voor het herbalanceren zien jullie in Figuur 11.



Figuur 11: Het toevoegen en herbalanceren in een B-tree voor $n = 4$.

Daarbij moet je de bijna-B-tree vooral als een model zien. Jullie moeten in de implementatie niet echt met een top met $n+1$ sleutels werken – jullie kunnen ook gewoon met een B-tree werken waar je voor één top nog een extra sleutel hebt. Het model waar er $n+1$ sleutels in één top zitten, is gewoon gemakkelijker om uit te leggen. Het herbalanceren gebeurt precies als volgt: Je splitst de top met $n+1$ sleutels. De grootste $n/2$ sleutels zitten in de ene nieuwe top en de kleinste $n/2$ sleutels in de andere. Dan heb je nog de middelste sleutel s die ertussen zit. Als er geen ouder is, wordt die de nieuwe wortel met deze twee toppen als kinderen (voor de wortel is het toegelaten dat hij maar 1 sleutel bevat – om het even wat n is). Als er een ouder is wordt s als sleutel aan de ouder toegevoegd en de twee nieuwe toppen als kinderen. Omdat de ouder nu één sleutel meer bevat moet hij ook één kind meer hebben. De positie waar je de sleutel en de kinderen moet toevoegen vind je snel als je nog weet het hoeveelste kind de top was. Op deze manier hebben wij één top die te veel sleutels bevatte verwijderd – maar misschien ook één nieuwe top gemaakt (als de ouder al n sleutels bevatte). Maar die zit nu dichter bij de wortel – als wij het herstellen recursief toepassen, gaat het proces dus zeker stoppen. Wij moeten alleen maar naar toppen langs het toegangspad kijken – dat is dus goedkoop – en bovendien hebben wij nadat een top gesplitst moet worden twee toppen met samen n vrije plaatsen – dus voldoende ruimte voor n toevoegoperaties zonder splitsen.
Samenvattend kan dus gezegd worden dat toevoegen een bewerking is die heel efficiënt is.

Verwijderen:

Net zoals voor 2-3-bomen is verwijderen ook hier relatief ingewikkeld en duur. Als er niet te veel verwijderbewerkingen zijn, is het dus het beste als je met grafstenen werkt: als een sleutel verwijderd moet worden, wordt hij niet echt verwijderd maar gemarkeerd als niet meer bestaande. Je zou hem natuurlijk ook kunnen vervangen door de grootste sleutel in zijn kleiner-kind of de kleinste sleutel in zijn groter-kind en inderdaad alleen sleutels in de bladeren verwijderen. Alleen als die beide ook een grafsteen hebben zou je ook grafstenen in het midden van de boom plaatsen. In een interne top moet de sleutel aanwezig blijven omdat hij tijdens het zoeken en toevoegen ook nog gebruikt wordt voor de *navigatie* in de boom – dus om de juiste sleutels en plaatsen te vinden. In een blad zou je hem ook gewoon kunnen verwijderen, zodat *grafstenen* in principe betekenen dat je hier toppen met minder dan $n/2$ sleutels aanvaardt.

Oefening 17 *Werk precies uit hoe je in een B-tree met grafstenen werkt, dus wanneer je echt grafstenen plaatst of gewoon toelaat dat er minder sleutels zijn, wanneer je een sleutel met een grafsteen door een nieuwe sleutel*

vervangt, etc.

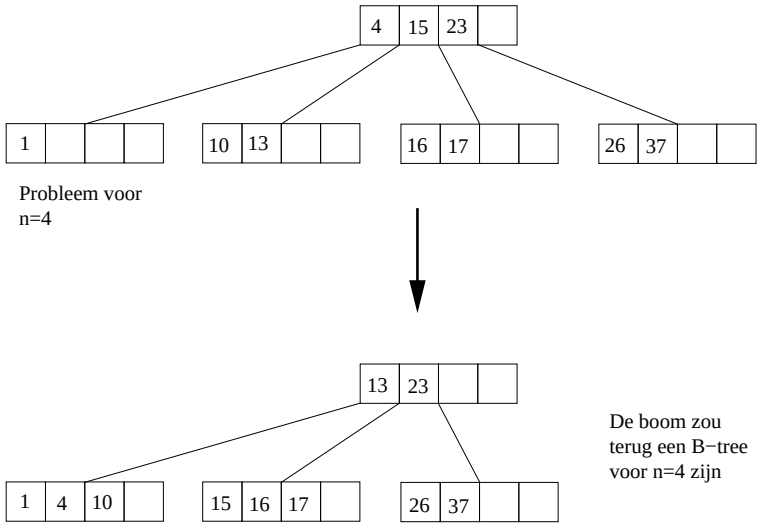
Als het een boom is met **extreem** veel verwijderbewerkingen (zonder tussendoor voldoende toevoegbewerkingen die de grafstenen in de bladeren toch automatisch verwijderen) zou je misschien beter toch niet altijd met grafstenen werken. Wij zullen er ons hier gewoon van overtuigen dat je op een manier die alleen lokale wijzigingen langs een pad van de wortel naar een blad gebruikt de boom **kan** herbalanceren zonder de details uit te werken (maar dat kan je dan wel zelfstandig). Maar dat is vooral omdat wij nieuwsgierig zijn of het kan...

Oefening 18 *Bewijs het volgende lemma:*

Lemma 2 *Gegeven een even getal $n \geq 2$. Dan bestaat er een B-tree met grootte n, ten minste $n/2$ sleutels in de wortel, met s sleutels en diepte 1 als en slechts als $\frac{n^2}{4} + n \leq s \leq n^2 + 2n$.*

Zoals net gezegd kunnen wij er ook hier vanuitgaan dat wij alleen maar sleutels in bladeren verwijderen. Als een sleutel uit een top wordt verwijderd die geen blad is, kunnen wij die vervangen door de kleinste sleutel in de deelboom waarvan het *groterkind* de wortel is of de grootste sleutel in de deelboom waarvan het *kleinerkind* de wortel is. Die zitten zeker in een blad. Als het blad achteraf nog ten minste $\frac{n}{2}$ sleutels bevat, moeten wij niets doen. Anders werken wij met een bijna-B-tree waar deze keer één top één sleutel te weinig bevat.
Stel eerst dat de ouder van de foutieve top niet de wortel is. Dan zitten er in de ouder en de kinderen van de ouder samen ten minste $\frac{n^2}{4} + n - 1$ sleutels en ten hoogste $n^2 + \frac{3}{2}n - 1$ sleutels. Dus kan – behalve in het geval dat er precies $\frac{n^2}{4} + n - 1$ toppen zijn – de deelboom bestaande uit de ouder en zijn kinderen vervangen worden door een andere deelboom die aan de eisen voldoet (deelboomvervangmethode en Lemma 2). Achteraf is de boom opnieuw een B-tree.
Als er precies $\frac{n^2}{4}+n-1$ sleutels inzitten, gaan wij de deelboom vervangen door een deelboom waar de ouder maar $\frac{n}{2}-1$ sleutels heeft. De foutieve top zit dan één stap dichter bij de wortel en door de hersteloperaties herhaaldelijk toe te passen, kunnen wij de boom uiteindelijk weer tot een B-tree maken. Om te bewijzen dat je inderdaad deze iets te kleine deelbomen ook door deelbomen met de fout in de wortel kan vervangen, kunnen wij (iets algemener dan echt nodig) analoog met Lemma 2 aantonen dat er voor $\frac{n^2}{4}+\frac{n}{2}-1 \leq s \leq \frac{n^2}{2}+\frac{n}{2}-1$ sleutels zo'n bomen met $\frac{n}{2} - 1$ sleutels in de wortel bestaan en wij hebben duidelijk $\frac{n^2}{4} + n - 1 \geq \frac{n^2}{4} + \frac{n}{2} - 1$ en $\frac{n^2}{4} + n - 1 \leq \frac{n^2}{2} + \frac{n}{2} - 1$ kan voor $n \geq 1$ ook gemakkelijk bewezen worden. De nodige bomen bestaan dus.

Wat overblijft is het geval dat de ouder de wortel is. Dit is volledig analoog – behalve dat wij in dit geval aan de ene kant ook naar het geval moeten kijken dat er minder dan $\frac{n}{2}$ sleutels in de ouder zitten (in de wortel mag dat) maar er aan de andere kant geen rekening mee moeten houden dat er achteraf ten minste $\frac{n}{2}$ sleutels in de ouder moeten zijn. Het enige geval waar wij de deelboom hier niet door een deelboom met diepte 1 kunnen vervangen, is waar er maar n sleutels zijn. In dit geval nemen wij een boom met maar 1 top en n sleutels.
Natuurlijk zijn de details hier niet echt uitgewerkt maar ik hoop toch dat jullie ervan overtuigd zijn dat de boom hersteld kan worden door alleen maar lokale bewerkingen langs het toegangspad te doen. Jammer genoeg kan ook een lokale bewerking hier veel meer leesoperaties vragen dan wij willen investeren – dit is dus alleen een oplossing als grafstenen een veel te grote overhead zouden betekenen. Als je de verwijderoperatie **echt** wil implementeren moet je natuurlijk goed over de meest efficiënte manier nadenken en dan zal je zeker niet onmiddelijk naar alle kinderen van een top kijken maar eerst of je het probleem niet al met een deel van de kinderen kan oplossen...



Figuur 12: Een voorbeeld voor het vervangen van een deelboom waar één top één sleutel te weinig bevat. De bijna-B-tree waarin dit deel wordt vervangen, zou achteraf terug een B-tree zijn.

Wij hebben B-trees geïntroduceerd om ervoor te zorgen dat de diepte – die in ons model het aantal leesoperaties beschrijft – zo klein mogelijk is. Maar hoe groot is de diepte ten hoogste als je s sleutels hebt?
Het slechtste geval (dus het geval waar voor een gegeven diepte het aantal sleutels minimaal is) is duidelijk als de wortel maar één sleutel bevat en

alle andere toppen maar $\frac{n}{2}$ sleutels. Als de diepte $d \geq 1$ is en schrijven wij $t = \frac{n}{2} + 1$ dan bevat elk van de twee kinderen van de wortel

$$(t-1)(1 + t + t^2 + t^3 + \cdots + t^{d-1}) = t^d - 1$$

sleutels. Samen met de wortel zijn dat $2t^d - 1$ sleutels. In een B-tree van orde n met s sleutels is de diepte $d \leq \log_t \frac{s+1}{2} = (\log \frac{s+1}{2}) / \log(\frac{n}{2} + 1)$.

Hoe kies je n?

Wij willen n zo groot mogelijk kiezen, maar het moet wel nog altijd in één leesbewerking gelezen kunnen worden.
Stel bv. dat wij getallen met 5 bytes willen opslaan en een blok heeft 4096 bytes. Wij hebben ook nog de pointers nodig. Stel dat die 8 bytes nodig hebben. Voor n sleutels per top hebben wij dus $5n + 8(n + 1)$ bytes nodig. Dus kiezen wij een even n zo groot mogelijk maar op een manier dat nog steeds $5n + 8(n + 1) \leq 4096$. Dat geeft $n = 314$.

Als wij in een B-tree met deze parameters 10^{10} getallen moeten plaatsen, hebben wij dus een B-tree met diepte $d \leq (\log \frac{10^{10}+1}{2}) / \log(\frac{314}{2} + 1) < 4.411$. Dus $d = 4$ in het slechtste geval. Dus zijn ten hoogste 5 leesoperaties nodig om elke van de 10^{10} sleutels te vinden!

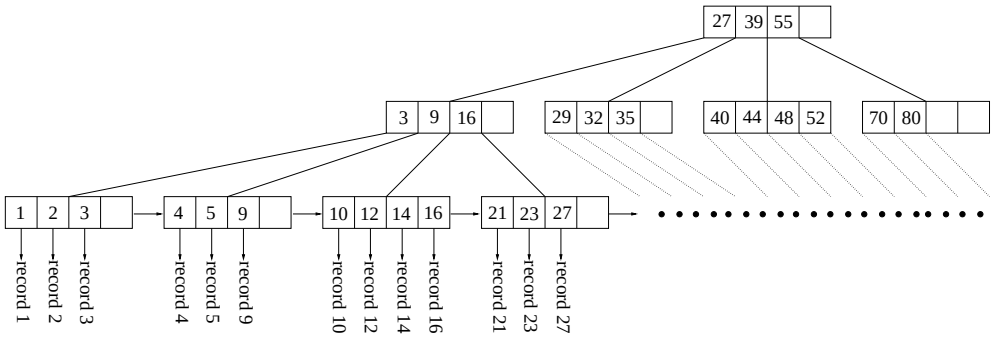
Oefening 19 *Stel dat* 99% *van de sleutels in een B-tree met grootte 300 grafstenen hebben en s het aantal sleutels zonder grafstenen is. Hoeveel slechter is de bovengrens voor de diepte van deze B-tree in vergelijking met een B-tree met grootte 300 waarin alleen maar de s sleutels zonder grafstenen zitten?*

Oefening 20 *Stel dat je een B-tree met grootte n hebt waar in elke top precies n sleutels zitten.*

- *Hoe groot is het aantal sleutels in de bladeren in vergelijking met het aantal sleutels in interne toppen van de boom?*
- *Stel dat er s sleutels in de bladeren zitten (onder deze omstandigheden moet dus $s = n * (n+1)^d$ voor een d zijn). Bewijs een goede bovengrens voor het verschil in diepte tussen deze boom en een boom B' waar alleen maar de sleutels uit de bladeren inzitten. Let op: voor B' kan je natuurlijk niet eisen dat er precies n sleutels in elke top zitten – kijk één keer naar de minimale en één keer naar de maximale diepte.*

2.3.2 B+-trees

Maar hoe realistisch is het dat wij met getallen als *typische* gegevens werken? Normaal is het zeker zo dat de records met de hele data veel groter zijn. De data ter identificatie (naam, rijksregisternummer, etc.) is misschien wel relatief klein, maar bovendien kan de record nog veel meer informatie bevatten (misschien zelfs een foto of andere gegevens die veel geheugen vragen zoals contracten, facturen, etc.). Deze informatie in de records zou de mogelijke vertakking **extreem** beperken. Het zou dus een goed (en voor de hand liggend) idee zijn niet echt de records op te slaan maar gewoon sleutels die nodig zijn om de records te identificeren en een pointer naar een record op de harde schijf – de boom dus gewoon als een index te gebruiken. Dat zou je dus zeker ook in B-trees doen – anders wordt de vertakking veel te klein. Maar B+-trees gaan nog één stap verder: ook de extra pointer vraagt natuurlijk geheugen en beperkt de vertakking. Daarom zullen wij pointers naar de records **alleen** in de bladeren bijhouden. Daar heb je de ruimte voor de pointers naar de kinderen niet nodig (als je in 1 bit of 1 byte bijhoudt dat het een blad is of – nog beter – gewoon bijhoudt wat de diepte van de boom is) en kan je dezelfde ruimte misschien gebruiken voor de pointer naar de records. Dat betekent wel dat elke sleutel in de bladeren aanwezig moet zijn, dus dat er kopieën in de binnenste toppen zitten – en wij dus meer sleutels hebben. Maar door de grote vertakking is de verhouding tussen het aantal bladeren en het aantal interne toppen heel groot.



Figuur 13: Het principe van een B+-tree.

In Figuur 13 zien jullie het principe van een B+-tree. De pointers die van het ene blad naar het andere blad gaan, maken de boom natuurlijk tot iets dat inderdaad geen boom is – de naam is dus een beetje misleidend. De bedoeling is efficiënter van het ene blad naar het volgende te kunnen gaan – bv. als je over alle gegevens wil itereren.

Definitie 3 *Wij stellen hier dat elke sleutel in maar één record voorkomt. Een B+-tree met (even) grootte $n \geq 2$ is een boom met volgende eigenschappen:*

- *het aantal a van sleutels in de wortel voldoet aan $1 \leq a \leq n$*
- *het aantal a van sleutels in toppen die niet de wortel zijn, voldoet aan $\frac{n}{2} \leq a \leq n$*
- *als in een top t sleutels $s_1 < s_2 < \cdots < s_a$ zitten dan geldt*
  - *Ofwel is t een blad (en heeft dus geen kinderen) ofwel heeft t precies $a+1$ kinderen $k_1, \ldots, k_{a+1}$ waarvoor met $s_0 = -\infty$ en $s_{a+1} = \infty$ voor alle sleutels s in de deelboom met wortel k_i geldt: (***let op – gewijzigd in vergelijking met B-trees:***) dat $s_{i-1} < s \leq s_i$.*
- *alle bladeren zitten op dezelfde afstand van de wortel (dezelfde diepte).*
- **nieuw:** *Elke sleutel van een record zit precies één keer in een blad.*
- **nieuw:** *Als t een blad met a sleutels $s_1, \ldots, s_a$ is, bevat t ook a verwijzingen $p_1, \ldots, p_a$ naar records waarbij voor $1 \leq i \leq a$ geldt dat p_i naar de record wijst die sleutel s_i bevat.*

Bovendien is er nog een extra pointer die de eigenschap een boom te zijn verstoort (maar wij noemen het toch al B+-tree):

- **nieuw:** *Elk blad bevat een pointer naar het volgende blad. (De pointer van het laatste blad wijst naar NULL.)*

Het maakt in principe *niet uit of je in plaats van $s_{i-1} < s \leq s_i$ vraagt dat $s_{i-1} \leq s < s_i$. Het resultaat zal een even efficiënte datastructuur zijn en jullie zullen beide definities in de literatuur terugvinden.*
Je zal ook definities vinden waar de pointers tussen de bladeren in beide richtingen gaan – dus zodat het volgende en het vorige blad bereikt kunnen worden.

De bewerkingen lijken heel sterk op die van B-trees. Maar er zijn natuurlijk sommige verschillen die veroorzaakt zijn door het feit dat elke sleutel in een blad moet opduiken.

zoeken: Zoeken is inderdaad het gemakkelijkst. Stel dat je sleutel s zoekt en met s_i vergelijkt. Je moet hier dan zelfs niet testen of $s = s_i$ als het een interne top is omdat je in de gevallen *gelijk* en *kleiner* op dezelfde

manier moet doorgaan – je moet in één van de kinderen zoeken die een index van ten hoogste i hebben. Het juiste kind bepalen kan je ook hier bv. met logaritmisch zoeken.

toevoegen: Als je een sleutel in een blad kan toevoegen zonder dat het volzit, is er geen probleem. Als dat wel volzit moet je splitsen: je splitst de top in twee toppen met een grootte die zo gelijk mogelijk is (dan verschilt het aantal sleutels in de twee toppen ten hoogste met 1). Let op de pointers tussen de bladeren! Nu heeft de ouder één kind te veel en moet hij één sleutel meer krijgen. Je neemt een kopie van de grootste sleutel in de kleinere van de twee delen en voegt die aan de ouder toe. Nu kan je er ook een pointer naar een kind meer plaatsen. Als de ouder niet volzat is het gedaan – anders heb je nu één top met één sleutel te veel. Dan wordt op dezelfde manier doorgegaan als in het geval van B-trees. Zie Figuur 14 voor een voorbeeld.

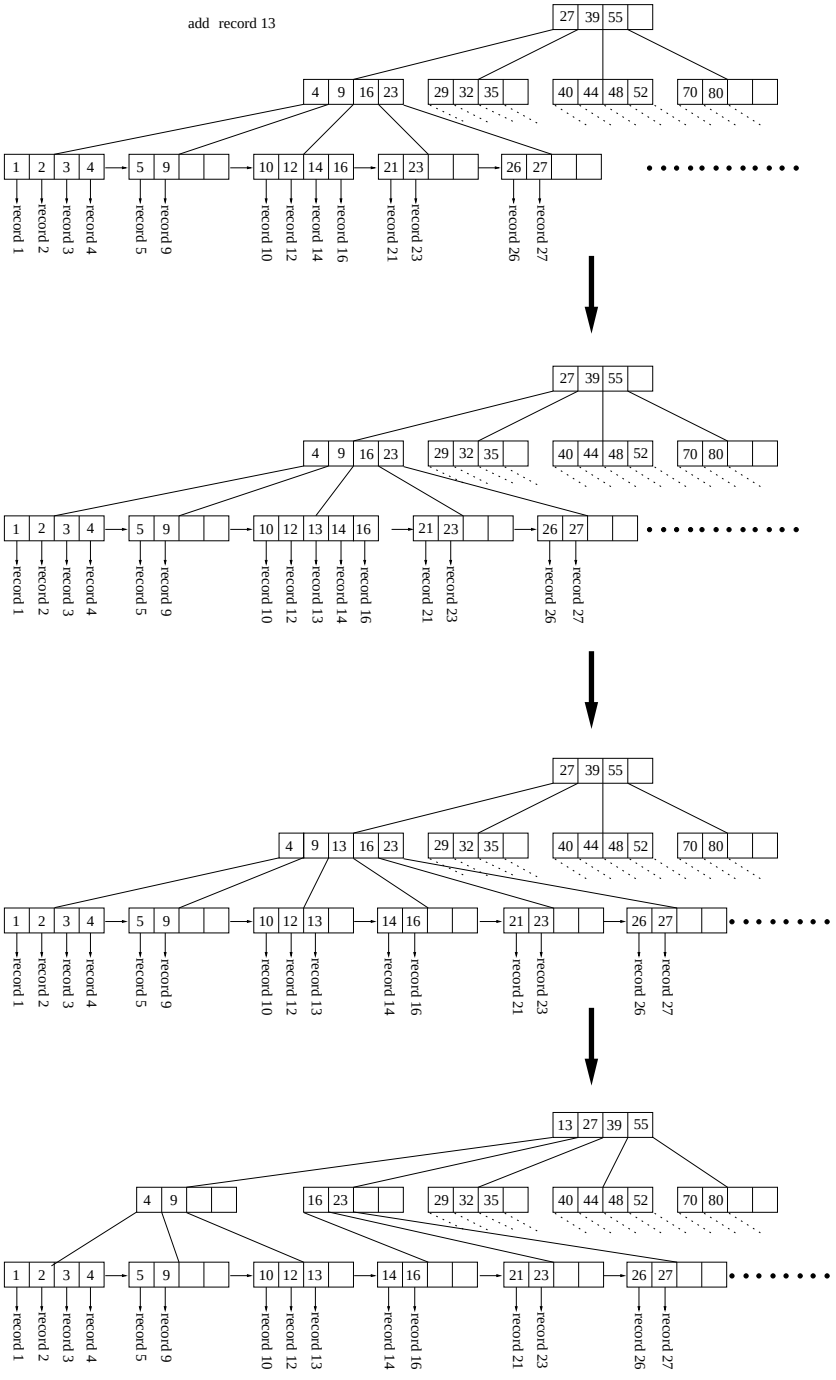
verwijderen: Ingewikkeld – maar jullie hebben nu al voldoende kennis om het verwijderen analoog met B-trees zelf uit te werken. Maar ook hier geldt dat als er niet extreem veel verwijderbewerkingen zijn het gewoon beter is grafstenen te gebruiken.

Oefening 21 *In een top mogen geen twee identieke sleutels zitten. Maar wij kopiëren een sleutel uit een blad en voegen die aan de ouder toe – hoewel het wel kan gebeuren dat een sleutel in een blad en zijn ouder zit. Toon aan dat de gekopieerde sleutel nog niet in de ouder van het blad zit.*

B+-trees kunnen dus even gemakkelijk gebruikt worden als B-trees. Toevoegen en opzoeken kan heel efficiënt geïmplementeerd worden en alleen de verwijderoperatie is een beetje ingewikkeld. Het feit dat de interne toppen in principe alleen maar voor de navigatie in de boom dienen en je kopieën van sleutels hebt, wordt door het feit dat je geen pointers naar records in de interne toppen moet bijhouden meer dan gecompenseerd.
Maar hoe groot is de diepte van een B+-tree met grootte n in het slechtste geval? Het geval waar voor een gegeven diepte het aantal sleutels minimaal is, is ook hier duidelijk als er 1 sleutel in de wortel en $\frac{n}{2}$ sleutels in alle andere toppen zitten. Als de diepte $d \geq 1$ is en schrijven wij $t = \frac{n}{2} + 1$ dan bevat elk van de twee kinderen van de wortel t^{d-1} bladeren dus verwijzingen naar ten minste $(t-1)t^{d-1}$ records. In de hele boom zijn dus ten minste $2(t-1)t^{d-1}$ records opgeslaan, dus – met s het aantal sleutels dat bij records behoort – $s \geq 2(t-1)t^{d-1}$. Als wij dat herschrijven dan geldt voor de diepte $d \leq \log_t s + 1 - \log_t(2(t-1))$ en omdat $2(t-1) \geq t$ geldt $\log_t(2(t-1)) \geq 1$ (maar het is nauwelijks groter) en $d \leq \log_t s = \frac{\log s}{\log(\frac{n}{2}+1)}$.

Als wij dat nu met de grens voor B-trees vergelijken dan zien wij dat dat zelfs voor dezelfde vertakking t naar beneden afgerond meestal dezelfde diepte is en alleen in zeldzame gevallen 1 groter. En als je er nog rekening mee houdt dat de vertakking voor B+-trees groter gekozen kan worden dan wordt het voordeel duidelijk!

Oefening 22 *Stel dat wij 10.000.000.000 records willen opslaan die door een unieke sleutel als een getal voorgesteld kunnen worden. Een blok heeft* 4096 *bytes en wij willen de toppen zo kiezen dat ze in één blok passen.*
Eerst werken wij met een B-tree waar wij de sleutels (5 bytes), de boompointers (8 bytes) en de pointers naar de records (8 bytes) in een top plaatsen. Wat is de diepte van deze boom in het slechtste geval?
Dan werken wij met een B+-tree waar wij de pointers dus maar één keer nodig hebben – als het om interne toppen gaat, zijn het pointers naar toppen en anders naar records. Wat is de maximale diepte van deze boom?



Figuur 14: Een voorbeeld voor het toevoegen in een B+-tree.

3 Algoritmen voor strings

Iedereen van jullie heeft zeker al algoritmen voor strings gebruikt – of het nu *query replace* in een editor is of het zoeken van woorden in een tekst. Maar ook op andere, minder duidelijke plaatsen zijn stringalgoritmen belangrijk – bv. in de bioinformatica waar de strings bv. het DNA voorstellen of in databanken waar de strings een identifier – bv. van een scheikundig molecule kunnen zijn. Wat precies met de strings gedaan wordt – of getest moet worden of twee strings identiek zijn (misschien de gemakkelijkste taak), of of een string deelstring van een andere string is, of of één string een *mutatie* van een andere string kan zijn, etc... verschilt natuurlijk van geval tot geval. Wij zullen hier verschillende algoritmen zien – soms ook verschillende algoritmen voor dezelfde taak. De hoofdreden om deze algoritmen te leren is om de ideeën te leren kennen en ze achteraf zelfstandig misschien in andere omstandigheden te kunnen toepassen. Wij zullen hier – als een soort bewijs dat het nuttig is ideeën te kennen – bv. ook technieken zien die *in principe* geïnspireerd zijn door technieken die jullie al in DA1 en DA2 hebben gezien!

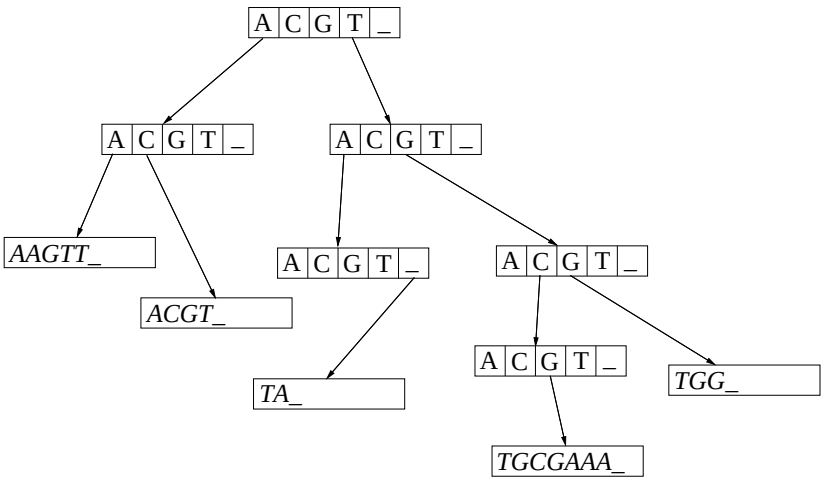
3.1 Tries (prefix trees)

De naam *trie* is afgeleid van het middengedeelte van *re***trie***val*. Vermoedelijk was het de bedoeling het zoals in *retrieval* uit te spreken – dus zoals *tree* – tenslotte is het een boom. Maar intussen spreekt het iedereen zo uit als *to try*.
Een trie – soms ook prefix tree genoemd – is een boom die ook als zoekboom gebruikt kan worden en vooral veel toepassingen heeft als het om strings gaat – bv. in de computationele biologie. Laten we eerst motiveren waarom er nood aan nog een implemetatie van een zoekboom is:
Stel dat wij geen sleutels, maar strings in een zoekboom willen opslaan. Dat kan natuurlijk heel eenvoudig – je moet alleen een functie hebben die twee strings s, s' vergelijkt en zegt of $s < s'$, $s = s'$ of $s > s'$. Er is maar één probleem: tot nu toe hebben wij nooit rekening gehouden met de kost van het vergelijken. Als je n sleutels in een gebalanceerde zoekboom hebt, is de kost voor het opzoeken $O(\log n)$ – maar als je strings vergelijkt komt er nog een factor $m = |s|$ bij, waarbij s de string is die je opzoekt – het wordt dus $O(m \log n)$. Tries combineren nu het vergelijken met het opzoeken door de vertakking in de boom afhankelijk van de lettertekens in de string te maken!
Jullie zullen (een beetje) verschillende datastructuren in de literatuur vinden die allemaal *trie* genoemd worden. Ze implementeren allemaal hetzelfde idee, maar de details zijn soms een beetje anders. Meestal (ook hier) wordt verondersteld dat voor geen twee strings die je wilt opslaan de ene een pre-

fix van de andere is. Deze eigenschap kan je ook afdwingen door een extra `end-of-string` teken te gebruiken. In de programmeertaal **C** wordt bv. elke string door een NULL-teken afgesloten, zodat nooit één string een prefix van een andere kan zijn.
Als wij ook strings willen toelaten, die een prefix van een ander string zijn, moeten wij bij interne toppen een flag bijhouden die zegt of de top bij een woord behoort of niet. In feite komt het als je het implementeert bijna op hetzelfde neer: in het geval waar je prefixen toelaat heb je een flag die `FALSE` (of 0) is of `TRUE` (niet 0) en in het geval waar je een extra `end-of-string`-teken gebruikt, heb je de pointer naar het speciale teken die ofwel `NULL` is ofwel niet `NULL`.
Wij kiezen dus voor de volgende definitie:

Definitie 4 *Gegeven een alfabet A. Een trie is een wortelboom waar elke interne top voor elke letter $a \in A$ een kind k_a heeft, dat ook leeg kan zijn en elk blad een string $s \in A^*$ als label heeft.*
De wortel stelt de lege string $\emptyset$ voor.
Als een top de string s voorstelt en het kind k_a is niet leeg, dan stelt k_a de string s, a voor.
Als een blad b het label s heeft en de string s' voorstelt, dan is s' een prefix van s.
De labels van de bladeren zijn de strings die in de trie zijn opgeslaan.

In deze trie dienen de interne sleutels dus alleen maar voor de navigatie om een sleutel die in een blad is opgeslagen te vinden. Een voorbeeld voor een trie met alfabet $A = \{A, C, G, T, \_\}$ zien jullie in Afbeelding 15.



Figuur 15: Een voorbeeld voor een trie.

Het is duidelijk dat het opzoeken van een string van lengte m in tijd $O(m)$ kan – dus een factor van $\log n$ beter dan in een gebalanceerde zoekboom met n elementen en in feite onafhankelijk van het aantal elementen in de boom! Dat ziet er fantastisch uit, maar jullie als ervaren programmeurs zien al het addertje onder het gras: als het alfabet relatief groot is (bv. alle ASCII tekens), heb je voor elke top vrij veel geheugen nodig – waarbij in de meeste gevallen veel geheugen alleen voor NULL-pointers gebruikt wordt. Bij veel implementaties zijn de kinderen voor elk letterteken dan ook alleen *"impliciet"* aanwezig en is de manier waarop je de pointers in een top opslaat belangrijk – zie bv. het hoofdstuk over *ternary tries*.
Maar laten we eerst kijken hoe je in een trie iets opzoekt, toevoegt en verwijdert. Wij veronderstellen dat de boom altijd ten minste één top heeft. Geïnitialiseerd wordt de boom met één top die de lege string voorstelt.

opzoeken: Stel dat je een string $s = s_0, \ldots, s_k$ wilt opzoeken. Dan ga je – beginnend bij de wortel, dus op diepte $d = 0$ – op diepte d naar kind k_{s_d} totdat

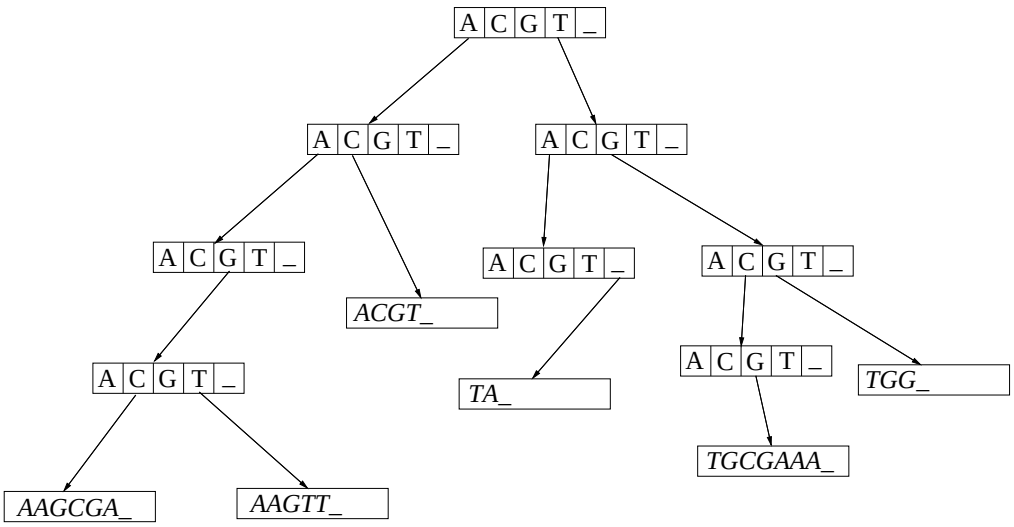
- het kind van een interne top NULL is (string niet gevonden)
- een blad bereikt wordt. Dan vergelijk je de gezochte string met de string die in het blad opgeslagen is.

toevoegen: Stel dat een string $s = s_0, \ldots, s_k$ toegevoegd moet worden. Je begint net zo als bij het opzoeken. Als de string gevonden wordt, hoeft die niet meer toegevoegd te worden. Stel dus dat je in een situatie terechtkomt waar die niet gevonden wordt.

- het kind k_{s_d} van een interne top is NULL: voeg een blad als kind k_{s_d} toe en label het met s.
- een blad b wordt op diepte d bereikt, maar het label $l_0, \ldots, l_m$ is niet gelijk aan $s_0, \ldots, s_k$. Kies $j = \min\{i | l_d = s_d, \ldots, l_{i-1} = s_{i-1}, l_i \neq s_i\}$. Maak dan b een interne top en vanuit b maak een pad dat langs de kinderen $k_{s_d}, \ldots, k_{s_{i-1}}$ gaat en voeg bij de laatste top twee bladeren als kinderen k_{l_i} en k_{s_i} toe met labels $l_0, \ldots, l_m$ resp. $s_0, \ldots, s_k$.

verwijderen: Je zoekt het blad op dat de string voorstelt. Dan verwijder je alle toppen na de laatste top voor het blad die ten minste twee niet lege kinderen heeft, resp. na de wortel als zo'n top niet bestaat. Als alleen nog de wortel over is, vervang je die door een blad dat de lege string voorstelt.

Als de string `AAGCGA_` aan onze voorbeeld-trie in Afbeelding 15 wordt toegevoegd, is het resultaat de trie in Afbeelding 16.



Figuur 16: De voorbeeld-trie uitgebreid met een nieuwe string.

Oefening 23 *Voeg* `TGCGAAB_` *aan de trie in Afbeelding 16 toe. Verwijder dan* `TA_`.

Oefening 24 *Stel dat jouw alfabet A alle ASCII-tekens zijn – dus alle tekens van* 8 *bit. Natuurlijk kan je een string van lengte* m *met tekens uit* A *ook interpreteren als een string van lengte* $8 * m$ *met tekens uit* $\{0, 1\}$.
*Wij implementeren de trie door voor de kinderen een array van pointers (*64 *bit) van lengte* $|A| = 256$*, resp. lengte* $|\{0, 1\}| = 2$ *aan te maken.*

- *Stel dat je (bijna) alle strings van lengte* l *wilt opslaan. Heb je meer geheugen nodig als je de trie volgens* A *vertakt of als je een trie maakt die volgens* $\{0, 1\}$ *vertakt.*

- *Stel dat je – in verhouding met het totale aantal strings – relatief weinig strings van lengte* l *wilt opslaan en dat jouw trie veel lange paden zonder vertakking heeft (de paden in de trie die volgens* $\{0, 1\}$ *vertakt, zijn natuurlijk nog langer dan die in de trie die volgens* A *vertakt). Heb je meer geheugen nodig als je de trie volgens* A *vertakt of als je een trie maakt die volgens* $\{0, 1\}$ *vertakt?*

Het grootste probleem bij de implementatie van een trie is dus het opslaan van de pointers naar de kinderen. Als die in de vorm van een array worden opgeslaan, kan je sleutels bijzonder snel opzoeken, maar vooral als er in vergelijking met de lengte van de strings weinig strings opgeslaan zijn (dat noemen wij soms een dunne trie (Engels: sparse)), verspil je in het geval van een groot alfabet heel veel geheugen. Alternatieven zouden bv. hashmaps zijn, die in het geval van veel NULL-pointers geheugen kunnen besparen, maar trager zijn. Je kan dan ook bv. bitmaps toevoegen om vroeg NULL-pointers te detecteren, etc. Wij zullen nu speciale tries zien om dunne tries efficiënt te implementeren.

3.1.1 Patricia trees (tries)

Onze definitie van een trie laat het al toe, geen lange paden direct voor een blad te hebben – die kunnen verwijderd worden. Paden zonder vertakking tussen twee interne toppen kunnen wij volgens de definitie niet gewoon verwijderen. Patricia trees of Patricia tries werden in 1968 door Donald R. Morrison geïntroduceerd. *Patricia* staat voor **P**ractical **A**lgorithm to **R**etrieve **I**nformation **C**oded **i**n **A**lphanumeric.
Het idee is om lange paden zonder vertakking te vermijden en altijd alleen interne toppen met vertakking te hebben door altijd alleen rekening te houden met de sleutels waar wel vertakt wordt. Patricia tries worden soms ook gecomprimeerde tries genoemd. Daarvoor heeft elke interne top een extra variabele `skip` die zegt hoeveel lettertekens zonder vertakking er zijn – die sla je gewoon over. Je zoekt dus altijd alleen maar volgens die tekens waar er verschillende mogelijkheden zijn. Daarbij kan het natuurlijk gebeuren dat op pad naar een blad de string die je zoekt al vroeg een verschil heeft met de string die in het blad opgeslagen is – en je merkt het niet omdat het een teken is dat je overslaat. Als een string in de trie bv. `AAAAABAAAAABC_` is en er zijn alleen maar vertakkingen bij de `B`'s, dan zou het opzoeken van `BBBBBBBBBBBB_` tot het blad gaan dat `AAAAABAAAAABC_` bevat en pas dan zou je merken dat het niet dezelfde string is – ook al verschilt al het eerste letterteken. Belangrijk is natuurlijk dat **als** de string in de trie zit het juiste blad gevonden wordt.
Als pseudocode ziet dat er zo uit:

Algoritme 1 *(Zoek Patricia)*

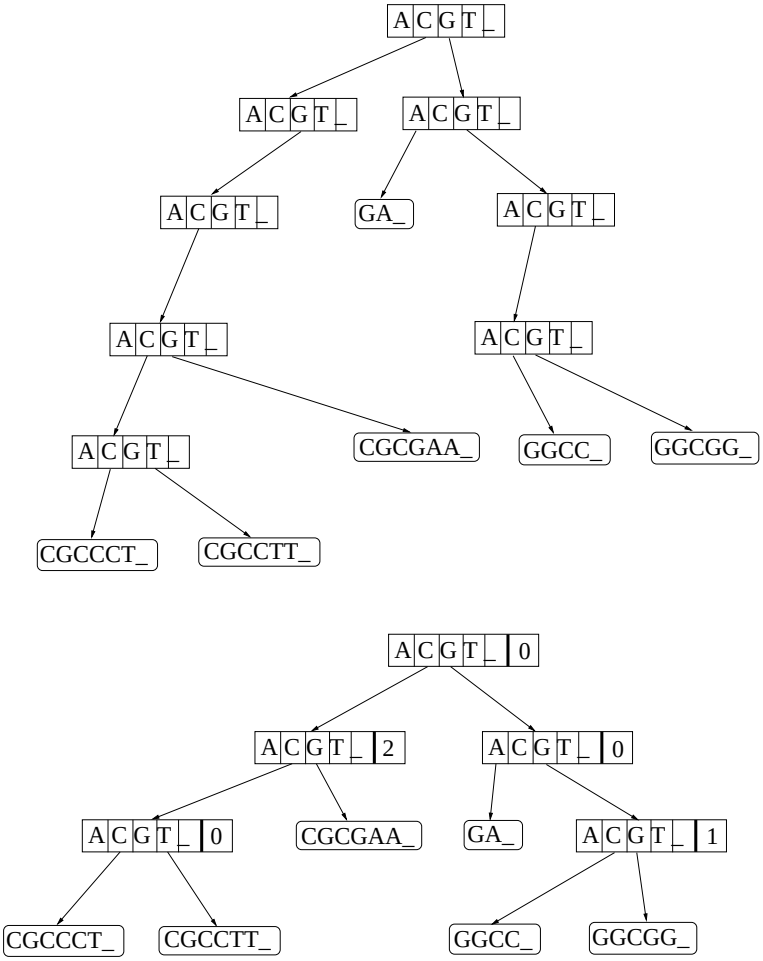
```
zoek_Patricia(s,m,top)
// zoekt de string s[]=s[0],...,s[m] in de Patricia tree
// met wortel top
```

```
{

next=0

while (top geen blad)
{ if (next+top.skip > m) return ‘‘niet gevonden’’
  // de string s is korter dan alle strings die vanaf
  // hier bereikt worden
  top = top.kind[s[next+top.skip]]
  if (top = NULL) return ‘‘niet gevonden’’
  next = next+top.skip+1
}
// nu zit je in een blad
if (top.string = s) return ‘‘gevonden’’
else return ‘‘niet gevonden’’
```

De complexiteit is – net zo als voor gewone tries – $O(m)$ als een string van lengte m opgezocht wordt.
Toevoegen bij een Patricia tree is duidelijk ingewikkelder, omdat je zolang je geen blad bereikt hebt niet weet met welke prefix je bezig bent. Zoals in ons voorbeeld `AAAAABAAAAABC_` kan het gebeuren dat je heel laat merkt dat je al vroeger een vertakking zou hebben moeten introduceren. Je moet dan dus eerst een string vinden om de tekens die je hebt overgeslaan te reconstrueren. Als de klemtoon niet alleen op het efficiënte opzoeken ligt en je ook efficiënt wilt toevoegen, is het misschien zinvol de deelstrings met de overgeslagen tekens bij de interne toppen bij te houden. Dat vraagt wel een beetje extra geheugen, maar je kan dan ook al tijdens het zoeken de prefixen vergelijken. In Figuur 17 zien jullie een trie die één keer op de gewone manier voorgesteld is en één keer als Patricia trie. Hier zien jullie al dat je geheugen kan besparen. In realistisch grote en dunne tries (die je zeker niet meer wilt tekenen) is dat natuurlijk nog veel meer.

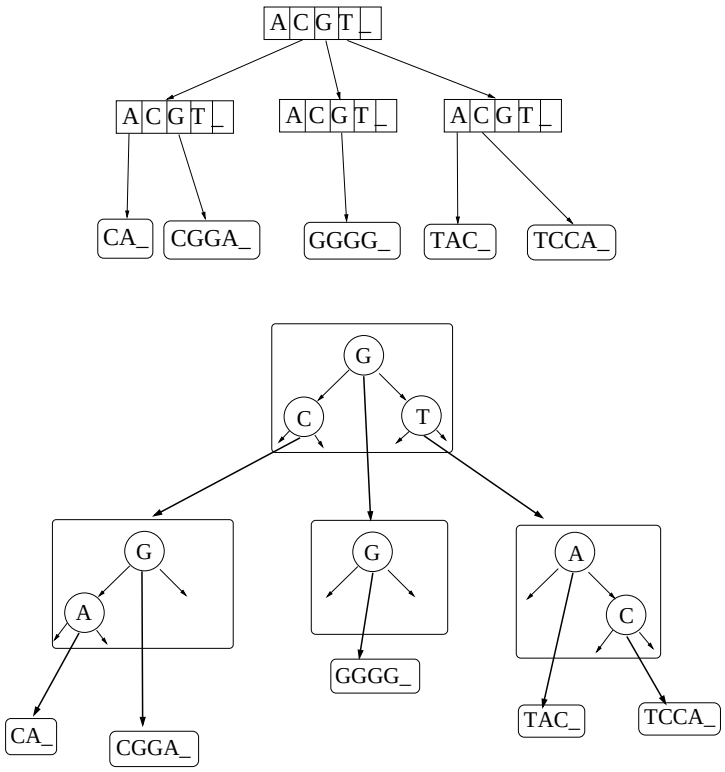
3.1.2 Ternary trees (tries)

Ternary trees of ternary tries – beschreven door J. Bentley and B. Sedgewick in 1988 – kan je op verschillende manieren interpreteren – aan de ene kant als bomen met drie vertakkingen, waarbij de middenvertakking een bijzondere eigenschap heeft: dan wordt overgeschakeld na het volgende teken in de string – en aan de andere kant als een implementatie van het datatype trie waar de kinderen in een boom opgeslagen zijn. Wij beschrijven het op de tweede manier. Als je het implementeert komt het in principe op hetzelfde neer.



Figuur 17: Een trie voorgesteld op de *gewone* manier en als Patricia trie.

In ternary tries wordt het probleem met het geheugen voor de pointers naar de kinderen opgelost door die in kleine boompjes op te slaan. Elke top in de trie heeft dus een pointerboom voor de kinderen. Elke top in de pointerboom heeft naast een sleutel uit het alfabet nog drie pointers: `kleiner`, `gelijk` en `groter` de `kleiner`- en `groter`-pointers gaan naar andere toppen in de pointer-boom. De `gelijk`-pointer gaat naar de volgende top in de trie – dat is dus een pointer naar het kind. Op deze manier geïnterpreteerd zijn de ternaire bomen dus in feite binair: de pointers naar de kinderen zijn gewoon de data die bij de sleutel behoort en die je wilt opslaan. De bomen bevatten dus alleen de lettertekens die ook echt gebruikt worden en omdat het aantal lettertekens dat met die prefix en op die positie gebruikt wordt normaal relatief klein is, wordt de boom bij het toevoegen ook niet geherbalanceerd. In Figuur 18 zien jullie een trie die één keer op de gewone manier voorgesteld



Figuur 18: Een trie voorgesteld op de *gewone* manier en als ternary trie.

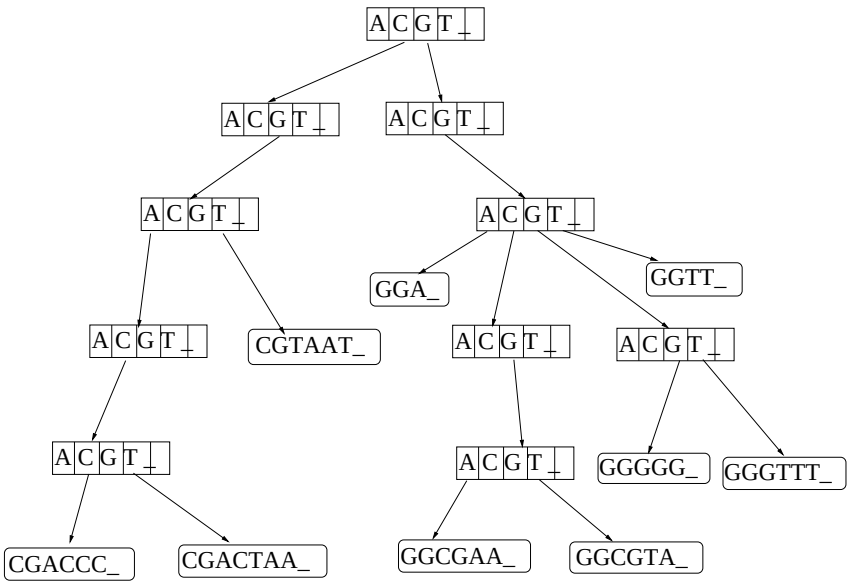
is en één keer als ternary trie. Hier lijkt het er niet op dat ternary tries een voordeel hebben. Als daarentegen het alfabet bv. alle ASCII tekens zou bevatten, zou de gewone manier veel meer ruimte eisen, terwijl de ternary trie niet zou veranderen.

Oefening 25

- *Stel de trie in Figuur 19 voor als Patricia trie.*
- *Stel de trie in Figuur 19 voor als ternary trie.*

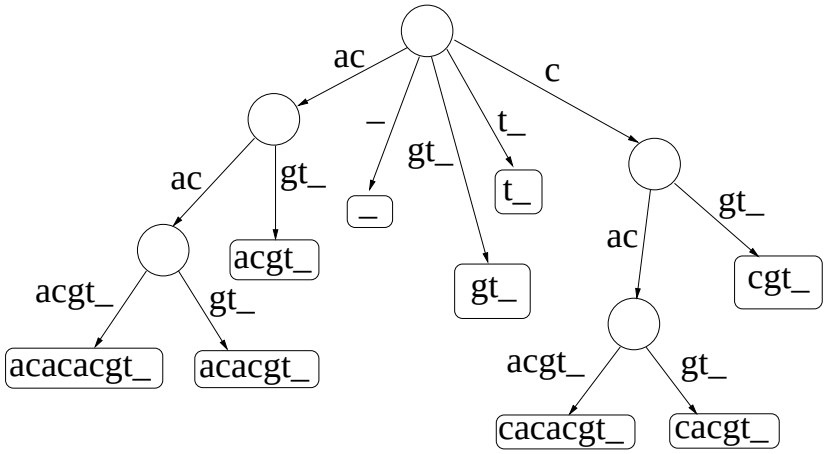
3.1.3 Suffix trees

Suffix trees of underlinesuffix bomen zijn een soort trie dat sommige verrassend snelle algoritmen voor strings mogelijk maakt. Daarbij is het opstellen van de suffix boom een preprocessing dat een heel krachtige datastructuur ter beschikking stelt die dan uitgebuit kan worden – wij zullen daarvoor voorbeelden zien. Anders dan gewone tries die een dynamische datastructuur vormen die bedoeld zijn om ook strings te kunnen toevoegen en verwijderen, wordt een suffix boom dus normaal één keer opgebouwd en dan blijft die zo.



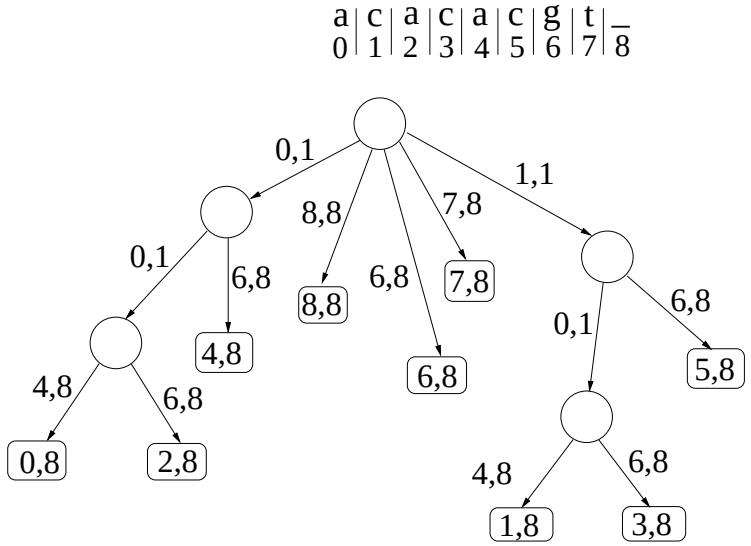
Figuur 19: Een trie voorgesteld op de *gewone* manier.

Stel dat een tekst t gegeven is. Wij stellen ook hier dat die door een uniek teken afgesloten is – wij zullen opnieuw `_` gebruiken. De suffix tree voor t is dan een trie die alle suffixes van t bevat. Voor de tekst `acacacgt_` bevat de trie dus `acacacgt_`, `cacacgt_`, `acacgt_`, `cacgt_`, `acgt_`, `cgt_`, `gt_`, `t_` en `_`. Het soort trie dat wij kiezen, lijkt het meest op een Patricia trie: er zijn geen toppen met maar één kind. Al voor Patricia tries hebben wij gezegd dat je de strings die je overslaat ook bij de toppen kan bijhouden en dat is wat wij hier doen. Je kan deze strings in de top bijhouden waar een boog naartoe gaat of in de boog zelf – dat komt op hetzelfde neer. Als wij de suffix boom tekenen, schrijven wij deze strings samen met het eerste teken dat het verschil maakt altijd aan de bogen – dat is gemakkelijker om te verstaan en dan hoeven wij de tekens van het alfabet ook niet meer in de top te tekenen. Het voorbeeld voor `acacacgt_` zien jullie in Figuur 21.
Als de tekst lengte n heeft, heeft de suffix boom voor de tekst ten hoogste $2n - 1$ toppen en $2n - 2$ bogen. Het aantal toppen en bogen is dus $\Theta(n)$. Jammer genoeg vraagt alleen het opslaan van alle prefixen in de bladeren $\Theta(n^2$ geheugen. In feite is het hier – anders dan in echte Patricia tries – niet nodig de prefixen allemaal op te slaan. Hier gebruiken wij dat het geen arbitraire trie is, maar dat alle opgeslagen strings de suffixen van dezelfde tekst zijn – alle strings die wij opslaan zijn dus deelstrings van deze tekst. Wij kunnen dus gewoon de tekst opslaan (dat eist $\Theta(n)$) en elke keer dat wij een string willen opslaan – om het even of het in een blad is of in een boog –



Figuur 20: Een suffix boom.

slaan wij gewoon de pointers naar het begin en het einde van een kopie van deze string in de tekst op. Dat vraagt constant geheugen voor elke string en omdat het aantal bladeren en bogen maar $\Theta(n)$ is, vraagt deze manier de suffix boom op te slaan ook maar geheugen $\Theta(n)$.



Figuur 21: Een suffix boom waar de strings als pointers naar de tekst zijn opgeslagen.

De voor de hand liggende manier de suffix boom op te bouwen – ook al eist die nu maar $\Theta(n)$ geheugen – vraagt wel tijd $\Theta(n^2)$ voor een tekst van lengte n. Maar het volgende is een heel belangrijk resultaat, dus zullen wij het een

stelling noemen:

Stelling 5 *De suffix boom voor een tekst van lengte n kan in tijd $O(n)$ opgebouwd worden.*

Het eerste algoritme dat in lineaire tijd werkt, is van Weiner (1973). Later heeft Mc Creight (1976) een verbeterd lineair algoritme gegeven dat minder geheugen eist. Het tot nu toe beste algoritme is van Ukkonen (1995). Dit algoritme heeft alle voordelen van de vroegere algoritmen en is beter verstaanbaar – maar wij zullen het hier (ten minste dit jaar) niet zien.

3.2 Exact string matching

In dit hoofdstuk zoeken wij een string $z[]$ in een andere string $t[]$. Dit is dus de typische toepassing van zoekmachines of van een editor die een woord in een tekst zoekt. Wij zullen de string die gezocht wordt altijd z noemen (z voor zoeken) en de lengte van deze string m (de string is dus $z[0]z[1]z[2]\ldots z[m-1]$). De string waarin gezocht wordt (vaak ook de tekst genoemd) heet altijd t (t voor tekst) en de lengte wordt als n genoteerd (deze string is dus $t[0]t[1]\ldots t[n-1]$).
Natuurlijk heeft iedereen onmiddellijk een idee hoe je een string kan zoeken:

Algoritme 2 *(Brute kracht)*

```
strings_gelijk(z,t,start)
// test of de eerste |z| tekens van t -- beginnend met
// positie start -- gelijk zijn aan die van z
// het wordt gesteld dat in t na start nog ten minste even
// veel tekens staan als in s en dat |z| gekend is
{
for (i=0; i< |z|; i++) if (z[i]!=t[start+i]) return FALSE;

return TRUE;
}

is_contained(z,t)
// test of de string z in de string t voorkomt
{
for (i=0; i<= |t|-|z|; i++)
  if strings_gelijk(z,t,i) return TRUE;
```

```
return FALSE;

}
```

Hier worden de strings vanaf $n - m + 1$ startposities vergeleken en elke vergelijking vraagt ten hoogste m stappen, de totale complexiteit is dus $O((n - m + 1) * m)$.

Opmerking 6 *Algoritme 2 toegepast op een tekst van lengte n en een zoekstring van lengte m vraagt in het slechtste geval* $O((n - m + 1) * m)$ *stappen.*

Als wij m als constant beschouwen (normaal zal de zoekstring heel klein zijn in vergelijking met de tekst) is dat $O(n)$ en dus zeker niet slecht. Maar er zijn twee redenen om toch betere algoritmen te zoeken. Ten eerste is het wel een beetje gesjoemeld als wij m verwaarlozen – de zoekstring **kan** in principe ook $\Theta(n)$ zijn – bv. $n/2$ – en dan mag m natuurlijk niet als constant beschouwd worden en wordt de complexiteit in feite $O(n^2)$. En ten tweede is voor praktische doeleinden een grote constante factor voor de complexiteit ook niet echt goed en vooral in DA3 ligt de klemtoon toch sterk op de praktische kant van algoritmen!

Oefening 26 *Als de binnenste lus van algoritme 2 echt m stappen nodig heeft, hebben wij het woord gevonden – dus zal het algoritme stoppen. En als de lus* bijna *m stappen doet, zouden wij verwachten dat misschien de volgende keren dat de functie wordt gebruikt al heel snel een tegenstrijdigheid gevonden wordt. Het is dus niet onmiddellijk duidelijk dat het algoritme echt soms* $\Theta(n^2)$ *stappen nodig heeft. Daarom moet dat aangetoond worden:*
Geef een geparametriseerde reeks van voorbeelden waarvoor dit algoritme in feite $\Theta(n^2)$ *stappen nodig heeft. Natuurlijk moet je ook aantonen dat* $\Theta(n^2)$ *stappen nodig zijn.*

Oefening 27

- *Herschrijf Algoritme 2 zo dat de zoekstring ook één of meerdere wildcards* `?` *mag bevatten. Wij stellen dat* `?` *geen deel uitmaakt van de tekst en dat elk letterteken erdoor gematcht wordt. Voor* `?uizen` *zou dus bv.* `huizen` *maar ook* `buizen` *een match in de tekst zijn.*

  Wat is de complexiteit van jouw algoritme?

- *Geef een algoritme zodat de zoekstring ook één of meerdere wildcards* `*` *mag bevatten waarbij* `*` *alle deelstrings matcht – ook de lege string. Voor* `ver*uizen` *zouden dus bv.* `verhuizen` *en* `vergeet de muizen` *een match in de tekst zijn. Jouw algoritme moet een arbitraire string in de*

tekst teruggeven die de zoekstring matcht als die bestaat en anders de lege string.

De complexiteit van jouw algoritme mag ten hoogste de complexiteit van Algoritme 2 zijn.

In DA1 en DA2 werd gezegd dat één van de bedoelingen van deze lessen is om technieken en ideeën te zien die je achteraf in een andere context kan toepassen om snelle algoritmen te ontwikkelen. Het volgende algoritme is een voorbeeld daarvan.

Het algoritme van Rabin-Karp

Als wij de kost van de functie `strings_gelijk()` in Algoritme 2 constant zouden kunnen maken of de functie alleen maar in heel zeldzame gevallen zouden moeten toepassen, zou de factor m in de complexiteit verdwijnen en wij zouden een complexiteit van $O(n)$ hebben. Het algoritme van Rabin-Karp doet dat in feite niet **gegarandeerd** – de complexiteit van het **slechtste geval** blijft dezelfde – maar in de praktijk is de kost van de met de functie `strings_gelijk()` corresponderende functie wel constant. Misschien doet jullie dat al aan een techniek uit DA1 denken waar het slechtste geval ook geen verbetering ten opzichte van de standaardtechnieken was, maar die in de praktijk wel heel nuttig is: hashing. En het idee van hashing – dat ook al voor Bloom-filters toegepast werd – is wat wij ook hier zullen gebruiken.
Voordat wij de zoekstring en een deelstring van de tekst vergelijken, vergelijken wij eerst de waarde van een hashfunctie van deze strings. Als die niet gelijk is, zijn de strings zeker verschillend en moeten wij ze ook niet expliciet vergelijken. Dat betekent wel dat wij misschien een hashfunctie moeten berekenen voor alle deelstrings van lengte m in de tekst t. Als deze berekeningen even duur of duurder zouden zijn dan het vergelijken van de twee strings zou het natuurlijk geen voordeel zijn. Het trucje is dus dat wij om de hashfunctie voor de string die op positie $i+1$ begint te berekenen de hashwaarde voor de string die op positie i begint gewoon updaten.
Wij zullen als voorbeeld een hashfunctie voor het alfabet $\Sigma = \{0, 1\}$ geven die al door Rabin en Karp werd gesuggereerd – wij werken dus met bitstrings. Maar het is duidelijk (zie oefeningen) dat het voor elk alfabet op een gelijkaardige manier gedaan kan worden.
Kies een natuurlijk getal q (bv. een priemgetal) en interpreteer een bitstring s (in ons geval kan dat ofwel $z[]$ ofwel een deelstring van $t[]$ zijn) gewoon als voorstelling van een getal $g(s)$. De hashwaarde $h(s)$ van de string is dan gewoon $h(s) = g(s) \mod q$. Als de string $s = s_0 s_1 s_2 \dots s_{m-1}$ is dat als formule

$$h(s) = (\sum_{i=0}^{m-1}(s_i 2^{(m-1)-i})) \mod q$$

De som ziet er misschien ingewikkeld uit, maar het is alleen maar het getal dat je krijgt als je de bitstring interpreteert als de voorstelling van een binair getal.
Als wij dus de hashfunctie willen berekenen van de string van lengte m die op positie p in de tekst $t_0 t_1 \dots t_n$ begint, dan is dat

$$h(t_p \dots t_{p+m-1}) = (\sum_{i=0}^{m-1}(t_{p+i} 2^{(m-1)-i})) \mod q$$

Hierbij hebben wij de posities als index geschreven in plaats van als arrayindex (dus t_i in plaats van $t[i]$) omdat dat formules beter leesbaar maakt.
Als wij gebruiken dat
$(a + b) \mod q = (a + (b \mod q)) \mod q$
en
$(a * b) \mod q = (a * (b \mod q)) \mod q$
dan krijgen wij voor de string die op positie $p + 1$ begint:

$$\begin{aligned}
h(t_{p+1} \dots t_{p+m}) &= (\textstyle\sum_{i=0}^{m-1}(t_{p+1+i} 2^{(m-1)-i})) \mod q \\
&= (\textstyle\sum_{i=0}^{m-2}(t_{p+1+i} 2^{(m-1)-i}) + t_{p+m}) \mod q \\
&= (-t_p 2^m + \textstyle\sum_{i=-1}^{m-2}(t_{p+1+i} 2^{(m-1)-i}) + t_{p+m}) \mod q \\
&= (-t_p 2^m + \textstyle\sum_{i=0}^{m-1}(t_{p+i} 2^{m-i}) + t_{p+m}) \mod q \\
&= (2 \textstyle\sum_{i=0}^{m-1}(t_{p+i} 2^{(m-1)-i}) - t_p 2^m + t_{p+m}) \mod q \\
&= ((2 \textstyle\sum_{i=0}^{m-1}(t_{p+i} 2^{(m-1)-i}) \mod q) - t_p 2^m + t_{p+m}) \mod q \\
&= (2h(t_p \dots t_{p+m-1}) - t_p 2^m + t_{p+m}) \mod q
\end{aligned}$$

Deze berekening geeft ons een heel snelle en gemakkelijke manier om de hashwaarde $h(t_{p+1} \dots t_{p+m})$ in constante tijd te berekenen als wij de hashwaarde $h(t_p \dots t_{p+m-1})$ al kennen. Alleen voor het berekenen van de hashwaarde van de zoekstring $z[]$ en van $h(t_0 \dots t_{m-1})$ hebben wij dus tijd $O(m)$ nodig – alle andere hashwaarden die wij nodig hebben, kunnen in constante tijd berekend worden.
De volgende pseudocode beschrijft het algoritme van Rabin-Karp

Algoritme 3 *(Rabin-Karp)*

```
strings_gelijk(z,t,start)
```

```
// test of de eerste |z| tekens van t -- beginnend met positie
// start -- gelijk zijn aan die van z
// het wordt gesteld dat in t na start nog ten minste even veel
// tekens staan als in z en dat |z| gekend is
{
for (i=0; i<|z|; i++) if (z[i]!=t[start+i]) return FALSE;

return TRUE;
}

is_contained_Rabin_Karp(z,t)
// test of de string s in de string t voorkomt
{
bereken h(z)

for (i=0; i<=|t|-|z|; i++)
{ bereken h(t_i...t_{i+m-1})
  // als i>0 kan dat gewoon door de oude waarde van h()
  // te updaten
  if (h(t_i...t_{i+m-1})==h(z))
    { if strings_gelijk(z,t,i) return TRUE; }
}
return FALSE;

}
```

De extra tijd in vergelijking met Algoritme 2 voor het berekenen van de hashfuncties is $O(m)$ voor het berekenen van $h(z)$, $O(m)$ voor het berekenen van $h(t_0...t_{m-1})$ en dan nog $O(n-m)$ voor het updaten van $h()$ – samen dus $O(n)$. De functie `strings_gelijk()` vraagt nog altijd tijd $O(m)$, in het slechtste geval is de complexiteit dus nog altijd $O((n-m+1)*m)$. Maar als wij ervan uitgaan dat de hashfunctie goed is en dus de waarden van de deelstrings ongeveer gelijk zijn verdeeld over de q mogelijkheden, dan wordt de functie maar in $(n-m+1)/q$ gevallen toegepast. De totale tijd die ervoor nodig is, is dus $O(m*(n-m+1)/q)$ en als wij $q \geq m$ kiezen dus $O(n-m+1)$. De buitenste lus vraagt ook tijd $O(n-m+1)$ – alles samen vraagt als wij deze veronderstellingen doen dus een tijd van $O(n)$.

Opmerking 7 *Het algoritme van Rabin Karp vraagt in het slechtste geval (als de waarden van de hashfunctie heel slecht verdeeld zijn) – net zoals het Algoritme 2 –* $O((n-m+1)*m)$ *stappen.*

In de praktijk – en veronderstellend dat de waarden van de hashfunctie gelijkmatig verdeeld zijn en q voldoende groot is – vraagt het algoritme tijd $O(n)$.

Oefening 28 *Je zoekt de string $z =$*`00101` *en past het algoritme van Rabin Karp toe met $q = 13$.*
Geef een voorbeeld van een tekst t van lengte n waarvoor het algoritme bijzonder slecht presteert als je naar z zoekt.

Oefening 29

- *Beschrijf precies hoe je de hashwaarden berekent en update om te garanderen dat je geen problemen met overflow van integers krijgt.*
- *Het argument dat bewijst dat het updaten van de hashwaarde in constante tijd kan gebeuren telt in feite alleen het aantal* integer operaties*. Maar voor arbitrair grote waarden van q passen de hashwaarden niet noodzakelijk in een integer van beperkte grootte. Wat is de invloed op de complexiteit als je voor de hashwaarden de arbitrair grote integers door integers van beperkte grootte (bv. 64 bit) moet simuleren? Zal dat in de praktijk gebeuren? Waarom (of waarom niet)?*
- *Stel dat je met q werkt dat groter is dan de maximale som in de formule – dat komst dus neer op het schrappen van de modulo-berekening. Dan hoef je als de hashwaarden overeenstemmen de strings zelfs niet meer vergelijken omdat ze zeker gelijk zijn. Is dat een goed idee?*

Oefening 30 *Stel dat je als alfabet niet alleen $\Sigma = \{0, 1\}$ hebt maar een algemeen (eindig) alfabet Σ. Definieer voor Σ een hashfunctie die in constante tijd geüpdatet kan worden.*
Denk daarbij erover na, of het belangrijk is dat in de som als basis voor de exponenten een 2 wordt genomen. Werken de argumenten ook met een andere basis – bv een ander priemgetal?

Oefening 31

- *Wijzig het algoritme van Rabin-Karp zo dat de zoekstring ook één wildcard* `?` *mag bevatten. Wij stellen dat* `?` *geen deel uitmaakt van de tekst en dat elk letterteken erdoor gematcht wordt (zoals in oefening 27). Ook hier moet het updaten van de hashfunctie (behalve in het begin) in constante tijd mogelijk zijn.*

- *Wijzig het algoritme van Rabin-Karp zo dat de zoekstring ook één of meerdere wildcards * mag bevatten (zoals in oefening 27).*

Jouw algoritme moet een arbitraire string in de tekst teruggeven die de zoekstring matcht als die bestaat en anders de lege string.
De complexiteit van jouw algoritme mag ten hoogste de complexiteit van het algoritme van Rabin-Karp zijn.

Oefening 32 *Wijzig het algoritme van Rabin-Karp zo dat je efficiënt naar een langste match kan zoeken.*

Het algoritme van Knuth-Morris-Pratt

Als jullie naar het brute kracht algoritme 2 kijken dan komt de factor m van de functie `strings_gelijk()`. Maar deze functie is alleen maar duur als het deel in de tekst waarmee wij het vergelijken in het begin gelijk is aan de string die wij zoeken. In principe duikt het probleem dus altijd voor een deelstring op die zo begint als de zoekstring – die wij dus al op voorhand heel goed kennen. En dat betekent dat met een beetje preprocessing voor deze string zeker een voordeel te halen is. En dat is het principe van het door D. Knuth, J.H. Morris en V. Pratt in 1977 ontwikkelde algoritme.
Het principe wordt duidelijk als je naar een voorbeeld kijkt: Wij zoeken de string `zoek_me` in een tekst. Stel dat je probeert of die in de tekst t vanaf positie i voorkomt. Als al na 1 stap de vergelijking fout teruggeeft – dus $t[i] \neq z$ moet je de volgende keer vanaf positie $i + 1$ zoeken. Die heb je ten slotte nog niet gezien en het is dus mogelijk dat daar een match staat. Maar stel nu bv. dat de posities $0, \ldots, 4$ van de zoekstring nog overeenkomen en de vergelijking pas op positie 5 mislukt. Dat betekent dat vanaf positie i de tekst `zoek_` in de tekst staat maar niet de tekst `zoek_m`. Nu heb je de volgende posities wel al gezien en daaruit kan je een voordeel halen: je weet dat bv. tot positie $t[i + 4]$ geen z voorkomt en je dus de string ten vroegste vanaf positie $i + 5$ terug kan vinden. En je kan dat al op voorhand weten: **altijd** als ik `zoek_me` opzoek en de vergelijking op positie 5 voor de eerste keer mislukt, kan ik de volgende 4 posities als mogelijke startposities overslaan.

Het basisidee:

Ik wil op voorhand voor mijn string een verschuivingstabel $V[]$ opstellen. Deze tabel bevat voor elke positie van de zoekstring het aantal vakjes waarmee ik mijn startpositie in de volgende iteratie zal mogen opschuiven als er de eerste mismatch optreedt op die positie in de zoekstring.

In het geval dat het eerste letterteken maar één keer in de zoekstring $z[]$ opduikt (zoals in `zoek_me`) is de tabel $V[]$ gemakkelijk te berekenen: $V[0] = 1$ en $V[j] = j$ voor $0 < j \leq |s|$. Je weet namelijk dat het eerste letterteken verschilt van de volgende j lettertekens – een vergelijking die vroeger in de tekst start dan j tekens later zou dus al voor het eerste letterteken van $z[]$ een fout opleveren!
Voor andere teksten, als de zoekstring bijvoorbeeld `barbados` is, is het al iets moeilijker
We zullen nu eerst een precieze definitie van $V[]$ ontwikkelen waarmee we verder kunnen werken.
Stel dat wij de posities $t[i+0], \ldots, t[i+j]$ al met $z[0], \ldots, z[j]$ vergeleken hebben en dat $z[j] \neq t[i+j]$ de eerste mismatch is. Hierbij is $j < |z|$ aangezien we anders het zoekwoord al volledig gematcht zouden hebben. Een verdere potentiële match zal dus zeker na positie $j-1$ eindigen aangezien tot deze positie nog geen match gevonden werd. Nieuwe startposities die nog geen mismatch voor positie $j-1$ geven, zijn dus posities s waarbij $z[0] = z[s]$,...,$z[j-1-s] = z[j-1]$. Als zo'n positie s niet bestaat, m.a.w. er is geen interne match binnen de zoekstring tot positie $j-1$, dan is j de eerst mogelijke startpositie waarop een verdere match mogelijk is. Aangezien we de eerst mogelijke startpositie zoeken, nemen we de kleinst mogelijke s die aan de voorwaarden voldoet. Of formeel:

Definitie 8 *Voor een gegeven zoekstring z is de verschuivingstabel $V[]$ gedefinieerd als*
$V[0] = 1$ *en voor* $j > 0$
$V[j] = \min\{s | (0 < s < j$ *en* $z[0] = z[s], \ldots, z[j-1-s] = z[j-1])$
of $s = j\}$

Als je op een positie i met $0 < i < V[j]$ begint dan weet je dat er ten laatste op positie $j-i$ een botsing is en dus de string vanuit deze positie niet gevonden kan worden.
Voor het voorbeeld `barbados` betekent dat

	b	a	r	b	a	d	o	s
positie j	0	1	2	3	4	5	6	7
V[j]	1	1	2	3	3	3	6	7

Wij weten nu hoeveel lettertekens je kan overslaan als je een match tot positie $j-1$ hebt maar geen tot positie j – maar er is nog een observatie die nodig is om ervoor te zorgen dat het algoritme echt lineair is:

Observatie: Stel dat er een match van $z[]$ tot positie $j-1$ in $t[]$ vanaf positie i staat. Dan kan een complete match ten vroegste op positie $i + V[j]$

beginnen (dat hadden wij al). **Maar:** als $V[j] < j$ dan weten wij van de posities $i+V[j], \ldots, i+j-1$ al dat ze overeenkomen met $z[V[j]], \ldots, z[j-1]$ – ze werden al vergeleken. Maar volgens de definitie van $V[]$ komen die dan ook overeen met $z[0], \ldots, z[j-1-V[j]]$. Als wij de string vanaf positie $i+V[j]$ zoeken, moeten wij dus de eerste $j-V[j]$ tekens niet opnieuw vergelijken. Wij moeten pas beginnen op de positie waar de eerste mismatch opdook.

Er kan dus op twee manieren bespaard worden: aan de ene kant wordt `vergelijk_strings()` minder vaak opgeroepen (door sommige startposities over te slaan) en aan de andere kant zijn de oproepen goedkoper omdat niet altijd vanaf het begin vergeleken moet worden.

Stel op dit moment dat wij de tabel $V[]$ efficiënt kunnen berekenen. Dan is het algoritme van Knuth-Morris-Pratt als volgt:

Algoritme 4 *(Knuth-Morris-Pratt)*

```
vergelijk_strings(z,t,start,gelijke_tekens)
// zoekt of de string z in de tekst t vanaf positie start voorkomt.
// Het wordt verondersteld dat de tekens 0...(gelijke_tekens-1)
// in z en t (vanaf start) gelijk zijn.
// Het wordt bovendien verondersteld dat in t na start nog ten
// minste even veel tekens staan als in z en dat |z| gekend is.
// De functie geeft de index van de eerste mismatch terug
{
for (i=gelijke_tekens; i<|z|; i++)
         if (z[i]!=t[start+i]) return i;

return |z|;
}

is_contained_Knuth_Morris_Pratt(z,t)
// test of de string z in de string t voorkomt
{
bereken_V(z)

start=0;
gelijke_tekens=0;

while (start<=|t|-|z|)
{
```

```
  eerste_mismatch=vergelijk_strings(z,t,start,gelijke_tekens);
  if (eerste_mismatch=|z|) return TRUE;

  start=start+V[eerste_mismatch];
  if (eerste_mismatch=0) gelijke_tekens=0; // speciaal geval
    else gelijke_tekens=eerste_mismatch-V[eerste_mismatch];
}

return FALSE;
}
```

Het geval waar de mismatch al op positie 0 opduikt is in zekere zin speciaal omdat daar ook de argumenten waarom wij kunnen opschuiven anders zijn. Het is het enige geval waar wij opschuiven **na** de positie van de eerste mismatch.
Het beste is natuurlijk de werking van het algoritme eens te illustreren aan de hand van een voorbeeld. Wij zoeken de string `barbados` in `barbaarse_barbecue_in_barbados`. De eerste positie die vergeleken moet worden wordt met een v gemarkeerd.

```
v
barbaarse_barbecue_in_barbados
barbados
     5
```

De eerste mismatch is op positie 5. Omdat $V[5] = 3$ kunnen wij met 3 posities opschuiven en moeten de eerste $5-3 = 2$ tekens niet meer vergeleken worden:

```
     v
barbaarse_barbecue_in_barbados
   barbados
     2
```

De eerste mismatch is nu op positie 2. Omdat $V[2] = 2$ kunnen wij met 2 posities opschuiven en moeten de eerste $2-2 = 0$ tekens niet meer vergeleken worden – dat helpt in dit geval dus niet.

```
     v
barbaarse_barbecue_in_barbados
     barbados
     0
```

Wij hebben onmiddellijk een mismatch, wij kunnen dus met $V[0] = 1$ positie opschuiven en moeten 0 posities (speciaal geval) niet vergelijken:

```
      v
barbaarse_barbecue_in_barbados
      barbados
      0
```

Wij hebben opnieuw onmiddellijk een mismatch en kunnen dus met $V[0] = 1$ positie opschuiven. Dat gaat door totdat wij de `b` van `barbecue` bereikt hebben:

```
          v
barbaarse_barbecue_in_barbados
          barbados
              4
```

De eerste mismatch is nu op positie 4. Omdat $V[4] = 3$ kunnen wij met 3 posities opschuiven en moeten de eerste $4-3 = 1$ tekens niet meer vergeleken worden:

```
              v
barbaarse_barbecue_in_barbados
             barbados
              1
```

De eerste mismatch is nu op positie 1. Nu zal dat zo doorgaan en er zal altijd onmiddellijk een mismatch gedetecteerd worden totdat de volgende `b` bereikt wordt – en dan wordt de match ook gevonden:

```
                      v
barbaarse_barbecue_in_barbados
                      barbados
                             8
```

Maar wat is de complexiteit van dit algoritme? Op dit moment stellen wij dat het berekenen van de tabel in tijd $O(m)$ kan gebeuren – dat zullen wij later natuurlijk nog bewijzen – en kijken wij alleen naar de routine die de string in de tekst echt zoekt.

Stelling 9 *Het algoritme van Knuth, Morris en Pratt vraagt tijd $O(n)$ als in een tekst $t[]$ van lengte n een zoekstring $z[]$ van lengte $m \leq n$ gezocht wordt.*

Bewijs: Wij veronderstellen hier dat "`bereken_V()`" in tijd $O(m)$ (dus ook $O(n)$) kan gebeuren. Dat zullen wij later nog zien.

De lus in `is_contained_Knuth_Morris_Pratt()` wordt duidelijk ten hoogste $n - m$ keer doorlopen en de kost in de lus is constant als wij de kost van `vergelijk_strings()` niet meerekenen. Wij moeten dus alleen nog bewijzen dat alle oproepen van
`vergelijk_strings()` samen ten hoogste tijd $O(n)$ vragen.

De complexiteit van `vergelijk_strings()` wordt bepaald door het aantal vergelijkingen die in de routine gebeuren en daar is een belangrijk verschil tussen succesvolle vergelijkingen en vergelijkingen die een mismatch opleveren: je kan hetzelfde teken uit de tekst meerdere keren *zonder succes* vergelijken, maar (dat zullen wij tonen) maar één keer succesvol. Wij tellen hier de vergelijkingen met en zonder succes dus apart. Omdat `vergelijk_strings()` ten hoogste $n - m$ keer wordt opgeroepen en omdat in elke oproep ten hoogste een vergelijking zonder succes gebeurt (bij een mismatch wordt onmiddellijk een waarde teruggegeven) zijn ten hoogste $n - m$, dus $O(n)$ vergelijkingen zonder succes.

Voor de andere oproepen is het nuttig de ontwikkeling te zien van de absolute positie in de tekst die met de string vergeleken wordt. In het algoritme wordt altijd de positie relatief met de startpositie bijgehouden, maar als je naar de absolute positie in de tekst kijkt dan hebben jullie misschien al in het voorbeeld gemerkt dat die nooit kleiner wordt. In feite wordt op het moment dat een teken één keer succesvol vergeleken wordt dit teken nooit meer met de zoekstring vergeleken – en omdat er maar n lettertekens zijn, volgt uit deze observatie natuurlijk dat ook deze bewerkingen samen maar tijd $O(n)$ vragen.

Wij moeten dus alleen nog de observatie bewijzen:

Stel dat `vergelijk_strings()` wordt opgeroepen met de variabelen `start` en `gelijke_tekens`. Dan wordt vergeleken vanaf positie
`start+gelijke_tekens`.

Wij moeten nu bewijzen dat als de volgende oproep met `start'` en `gelijke_tekens'` is dat dan
`start'+gelijke_tekens'`$\geq$
`start+eerste_mismatch`

omdat `start+eerste_mismatch` net de eerste positie **na** de succesvolle vergelijkingen is en dat garandeert dat al succesvol vergeleken tekens

niet nog eens getest worden.

Als `eerste_mismatch`= 0 dan zijn er geen succesvolle vergelijkingen en `gelijke_tekens` en `gelijke_tekens'` zijn beide 0 dus `start'`=`start`+1 – de bewering klopt dus. Anders geldt

```
start'+gelijke_tekens'=
start+V[eerste_mismatch]+eerste_mismatch-V[eerste_mismatch]=
start+eerste_mismatch
```

En precies dat wouden wij bewijzen.

■

In typische toepassingen die jullie kennen, zijn de zoekstrings veel korter dan de tekst – zelfs de brute kracht manier om $V[]$ te berekenen (die in $O(m^2)$ of zelfs $O(m^3)$ draait afhankelijk van hoe je het implementeert) zou dan geen probleem zijn. Maar voor toepassingen waar het verschil tussen m en n niet zo groot is, moet de preprocessing wel efficiënt zijn – en het best in tijd $O(m)$ gebeuren.
Wij zoeken niet voor elke index i van $V[]$ waar de vroegste startpositie is, maar voor elke mogelijke startpositie – beginnend met de kleinste, dus 1 – kijken wij hoe ver we raken zonder mismatch. Voor de nog niet ingevulde indexen i tot en met de eerste mismatch is de startpositie dan de vroegste startpositie, dus de juiste waarde voor $V[i]$. In principe zoeken wij dus alleen maar een match van $z[]$ in zichzelf. Daarvoor kan je **bijna** hetzelfde algoritme toepassen als voor het zoeken van de tekst. Je zoekt de tekst in zichzelf – dus zijn $z[]$ en $t[]$ dezelfde string. Omdat je nu naar matches van verschillende lengten zoekt, moet je wel tot het einde doorgaan. Belangrijk daarbij is dat de waarden van $V[]$ die je gebruikt altijd waarden zijn die je al op voorhand hebt berekend. $V[0] = V[1] = 1$ is voor elk zoekwoord hetzelfde – daarmee kan je dus beginnen.
Om de pseudocode eenvoudiger te houden gaan we ervanuit dat op het einde van $z[]$ nog een speciaal `end_of_line` teken volgt dat verschilt van alle tekens in de string en dat je in $V[]$ één teken meer kan invullen dan de lengte van $z[]$ (ook al gebruik je dat achteraf niet). Op deze manier moeten wij geen testen in `vergelijk_strings()` invullen die garanderen dat je niet over de grenzen van de array leest, wat de pseudocode minder leesbaar zou maken.

Algoritme 5 *(Berekenen van $V[]$)*

```
bereken_V(z)
// bereken de verschuivingstabel van de string z
{
V[0]=1;
V[1]=1;

start=1; // voor 0 zou je gewoon dezelfde string terugvinden
gelijke_tekens=0;

while (start<|z|-1)
{
  eerste_mismatch=vergelijk_strings(z,z,start,gelijke_tekens);

  if (eerste_mismatch=0) V[start+1]=start+1;
    else
     for (i=gelijke_tekens+1; i<=eerste_mismatch; i++)
                                              V[start+i]=start;
    //  de andere tekens werden al ingevuld

  start=start+V[eerste_mismatch]; // deze waarde is
                                  // gegarandeerd al ingevuld

  if (eerste_mismatch=0) gelijke_tekens=0; // speciaal geval
    else gelijke_tekens=eerste_mismatch-V[eerste_mismatch];
}
}
```

Het is duidelijk dat `bereken_V()` $O(m)$ tijdsbegrensd is omdat het extra werk in vergelijking met `is_contained_Knuth_Morris_Pratt()` (het invullen van $V[]$) zeker $O(m)$ tijdsbegrensd is. De `while()` lus loopt wel tot het einde, maar in feite hebben wij bij de analyse van `is_contained_Knuth_Morris_Pratt()` ook gewoon gesteld dat het ten hoogste tot het einde loopt. En het deel dat `bereken_V()` vervangt is hier het invullen van maar 2 getallen in een array – dat is dus zeker constant.
De werking van `bereken_V()` zie je het best opnieuw aan een voorbeeld. Hoe berekent deze functie de tabel voor `barbados`? Wij beginnen:

	b	a	r	b	a	d	o	s
positie j	0	1	2	3	4	5	6	7
V[j]	1	1						

1

```
barbados
 barbados
 0
```

Wij hebben onmiddellijk een mismatch (`eerste_mismatch=0`), kunnen dus $V[1+1] = 2$ invullen en $V[0] = 1$ opschuiven.

	b	a	r	b	a	d	o	s
positie j	0	1	2	3	4	5	6	7
V[j]	1	1	2					

```
  2
barbados
  barbados
  0
```

Wij hebben opnieuw onmiddellijk een mismatch, kunnen dus $V[2+1] = 3$ invullen en $V[0] = 1$ opschuiven.

	b	a	r	b	a	d	o	s
positie j	0	1	2	3	4	5	6	7
V[j]	1	1	2	3				

```
   3
barbados
   barbados
     2
```

Nu hebben wij de eerste mismatch op positie 2. Wij kunnen dus $V[3+1] = 3$ en $V[3+2] = 3$ invullen en $V[2] = 2$ posities opschuiven.

	b	a	r	b	a	d	o	s
positie j	0	1	2	3	4	5	6	7
V[j]	1	1	2	3	3	3		

```
     5
barbados
     barbados
     0
```

Nu krijgen wij nog twee keer onmiddellijk een mismatch en hebben dan

	b	a	r	b	a	d	o	s
positie j	0	1	2	3	4	5	6	7
V[j]	1	1	2	3	3	3	6	7

Oefening 33 *Pas het algoritme van Knuth-Morris-Pratt toe op de zoekstring* `z[]= volvoert` *en de tekst* `t[]= volvo_vol_vogels_volvoert_kunststukje`*. Bereken de verschuivingstabel en pas het zoekalgoritme toe. Werk voldoende tussenstappen uit om te laten zien hoe het algoritme werkt.*

Oefening 34 *Pas het algoritme van Knuth-Morris-Pratt toe op de zoekstring* `z[]= ananas` *en de tekst* `t[]= bananen_in_ra'ananas'_centrum`.
Bereken de verschuivingstabel en pas het zoekalgoritme toe. Werk voldoende tussenstappen uit om te zien hoe het algoritme werkt.

Oefening 35 *Wat moet aan de pseudocode gewijzigd worden om* **alle** *matches van de zoekstring te vinden?*
Werkt het algoritme van Knuth-Morris-Pratt ook in tijd $O(n)$ als in een tekst $t[]$ van lengte n alle voorkomens van een zoekstring $z[]$ van lengte $m \leq n$ gezocht worden?

Oefening 36 *Stel dat $m >> n$. Geef argumenten waarom het algoritme van Knuth-Morris-Pratt zoals beschreven dan niet $O(n)$ tijdsbegrensd is en beschrijf precies hoe je het moet wijzigen dat het dat wel is.*

Oefening 37 *Bewijs: als het algoritme zonder de extra "observatie" wordt geïmplementeerd (dat is hetzelfde alsof je* `gelijke_tekens` *altijd gelijk aan* 0 *zet) zijn er gevallen waarvoor het algoritme $\Theta(n^2)$ is.*

Oefening 38 *Het algoritme houdt alleen rekening met de informatie waar je een match hebt en niet met het letterteken op de mismatch – hoewel dat ook al gelezen werd. Zou het helpen ook daarmee rekening voor de verschuivingstabel te houden?*

Oefening 39 *Wijzig het algoritme van Knuth-Morris-Pratt zo dat het voor twee strings $z[]$ en $t[]$ de positie en de lengte van een langste match van een prefix van $z[]$ in $t[]$ teruggeeft.*
Gezocht is dus een match van $z[0]z[1]\ldots z[i]$ waarvoor i zo groot mogelijk is.

Oefening 40 *Stel dat je met de wildcard* `?` *wil werken en het algoritme van Knuth-Morris-Pratt gewoon wijzigt door een vergelijking waarin een* `?` *betrokken is altijd te aanvaarden. Toon aan dat het gewijzigde algoritme niet juist werkt.*

Oefening 41 *Wijzig het algoritme van Knuth-Morris-Pratt zo, dat het met wildcards* `*` *in de zoekstring $z[]$ kan werken, maar om het even hoeveel wildcards gebruikt worden nog altijd gegarandeerd in tijd $O(n)$ draait met n de lengte van de tekst $t[]$.*
Bewijs expliciet dat jouw algoritme de gegeven complexiteit heeft.

Het algoritme van Boyer-Moore-Horspool

Het algoritme van Boyer-Moore-Horspool (1980) is een vereenvoudiging van het algoritme van Boyer and Moore (1977). Het algoritme van Boyer and Moore in zijn oorspronkelijke vorm was *bijna* even efficiënt als het algoritme van Knuth-Morris-Pratt en met een kleine wijziging van Galil in 1979 zelfs even efficiënt (dat is: $O(n+m)$ voor elke toepassing). Maar het bijzondere aan het algoritme van Boyer-Moore-Horspool is dat het heel eenvoudig en gemakkelijk toe te passen is. Dat maakt het in de praktijk – waar je ook rekening moet houden met de constanten die in de $O()$-notatie verwaarloosd zijn – bijzonder snel.
Voor ons is het hier interessant te zien dat een eenvoudig idee soms tot een heel goed algoritme kan leiden – ook al is de prestatie in het slechtste geval misschien niet zo goed als die van andere – ingewikkeldere – algoritmen.
Het volgende algoritme is bijna identiek aan het brute kracht algoritme. Het enige verschil is dat de strings niet van positie 0 tot $|z|-1$ maar van $|z|-1$ tot 0 worden vergeleken. Daardoor kan soms vroeger en soms later gestopt worden – een echt voordeel is dat niet.

```
strings_gelijk_invers(z,t,start)
// test of t[start]...t[start+|z|-1] = z[0]...z[|z|-1]
// hierbij wordt vanaf het einde vergeleken

{
for (i=|z|-1; i>=0; i--) if (z[i]!=t[start+i]) return FALSE;

return TRUE;
}


is_contained(z,t)
// test of de string z in de string t voorkomt
{
for (i=0; i<= |t|-|z|; i++)
  if strings_gelijk_invers(z,t,i) return TRUE;

return FALSE;

}
```

Stel nu dat je bijvoorbeeld de string `barbara` in `barbados_zoekt_barbara`

zoekt. De eerste keer dat de lus in `is_contained()` wordt doorlopen vergelijk je

```
barbados_zoekt_barbara
barbara
```

en het eerste teken dat je vergelijkt is het laatste van `barbara`. Je vergelijkt dus `a` met `o`. Daarbij zie je dat de vergelijking niet alleen een fout oplevert – het is zelfs zo dat er in de zoekstring `barbara` helemaal geen `o` zit ...
Het is dus zinloos vanaf een startpositie te vergelijken waar de `o` nog in het deel zit dat vergeleken wordt – de volgende startpositie met kans op succes is dus de positie na de `o`.

```
barbados_zoekt_barbara
       barbara
```

Nu vergelijken wij eerst een `t` met de laatste `a` – en zijn in dezelfde situatie: er is geen `t` in `barbara`, wij kunnen dus opschuiven tot na de `t`.

```
barbados_zoekt_barbara
              barbara
```

Nu vergelijken wij een `r` met de `a` en ook al is dat een mismatch, kunnen wij ons argument hier niet meer toepassen: `barbara` bevat wel degelijk een `r` (zelfs twee). Hoe ver kunnen wij dus opschuiven? Wij weten dat er alleen maar een kans is als onze startpositie zo is dat er een `r` op de positie van de `r` in de tekst terechtkomt. Als je van achteraan telt dan zie je dat je ofwel 1 ofwel 4 posities moet opschuiven. Als je 4 posities opschuift, toets je niet of er een mogelijke match is als je maar 1 positie opschuift. Voor lettertekens uit de zoekstring kiezen wij dus altijd voor het kleinste aantal posities (maar wel ten minste 1 positie) dat je kan opschuiven om een match van het laatste teken te hebben – in ons voorbeeld dus 1:

```
barbados_zoekt_barbara
               barbara
```

Deze ideeën zullen wij nu gebruiken om een verschuivingstabel op te stellen. Een groot verschil met Knuth-Morris-Pratt is daarbij dat je de verschuiving opzoekt gebaseerd op een teken in de tekst – en niet gebaseerd op een positie in de zoekstring zoals bij Knuth-Morris-Pratt. Als wij een zoekstring $z[]$ hebben en de tabel V_H noemen, dan kunnen wij voor een letterteken x definiëren:

$$V_H(x) = \begin{cases} |z| & \text{als } x \text{ niet in } z[] \text{ voorkomt} \\ & \text{of alleen op positie } |z|-1 \\ p-1 & \text{als } |z|-p \text{ de laatste positie voor } |z|-1 \\ & \text{is, waar } x \text{ in } z[] \text{ voorkomt} \end{cases}$$

Deze tabel kan je met de volgende pseudocode gemakkelijk en snel berekenen:

```
vul_VH(z)

{
// eerst voor alle bytes (ASCII tekens) de waarde |z| invullen:
for (i=0; i<256; i++) VH[i]=|z|;

// dan voor tekens die in z[] voorkomen de waarden overschrijven
// de laatste keer dat een teken voorkomt bepaald de waarde:

for (i=0; i<=|z|-2; i++) VH[z[i]]=|z|-i-1;
}
```

Deze tabel kan duidelijk heel gemakkelijk en snel ingevuld worden. Ook in de $O()$-notatie is dat lineair in $m = |z|$, dus $O(m)$.

Voor `barbara` is dat

$x =$	`a`	`b`	`r`
$V_H[x]$	2	3	1

en $V_H[x] = 7$ voor alle andere tekens.

Daarmee kunnen wij nu het algoritme van Boyer-Moore-Horspool beschrijven:

```
is_contained_BMH(z,t)

{
vul_VH(z);

for (i=0; i<= |t|-|z|; )
```

```
  { if strings_gelijk_invers(z,t,i) return TRUE;
    i= i+ VH[t[i+|z|-1]];
  }

return FALSE;

}
```

Omdat wij de werking al voor het zoeken van `barbara` in `barbados_zoekt_barbara` gezien hebben, zullen wij hier niet nog een voorbeeld uitwerken.

Oefening 42 *Pas het algoritme van Boyer-Moore-Horspool toe op de zoekstring* `z[]= ananas` *en de tekst* `t[]= de_spa_in_ra'ananas'`.
Bereken de verschuivingstabel en pas het zoekalgoritme toe. Werk voldoende tussenstappen uit om te zien hoe het algoritme werkt.

Natuurlijk zien jullie onmiddelijk, dat hier – vergelijkbaar met Knuth-Morris-Pratt – informatie die je in `strings_gelijk_invers()` verwerft ook gebruikt zou kunnen worden om soms verder op te schuiven of vergelijkingen te besparen. Dan gaan jullie al in de richting van het (iets ingewikkeldere) algoritme van Boyer-Moore, resp. Boyer-Moore-Galil .

Oefening 43 *In de pseudocode wordt de functie* `strings_gelijk_invers(z,t,i)` *gebruikt. Zou de code ook juist werken als je dat door* `strings_gelijk(z,t,i)` *vervangt?*

Oefening 44 *Gegeven een zoekstring $z[]$ met $|z| = m$ en een tekst $t[]$ met $|t| = n$.*

- *Geef voorbeelden van $z[]$ en $t[]$ (die voor alle $n \geq m > 0$ werken) die voor het algoritme van Boyer-Moore-Horspool het beste geval voorstellen. Wat is de complexiteit van het algoritme van Boyer-Moore-Horspool in dit beste geval ($\Theta()$-notatie).*
- *Geef voorbeelden van $z[]$ en $t[]$ (die voor alle $n \geq m > 0$ werken) die voor het algoritme van Boyer-Moore-Horspool het slechtste geval voorstellen. Wat is de complexiteit van het algoritme van Boyer-Moore-Horspool in dit slechtste geval ($\Theta()$-notatie).*

Het shift-AND algoritme

Het algoritme van Knuth, Morris, and Pratt kan gezien worden als een optimalisatie van het brute kracht algoritme dat wij misschien als eerste zouden

proberen. Het volgende algoritme van R. Baeza-Yates en G. Gonnet is om meerdere redenen interessant: aan de ene kant is al de aanzet een beetje verrassend en ligt de klemtoon hier sterk op de praktijk – waardoor het natuurlijk heel goed bij DA3 past – en aan de andere kant zouden de technieken voor jullie ook een motivatie kunnen zijn in andere toepassingen waar jullie zelf algoritmen moeten ontwerpen eens na te denken of een vertaling naar bitvectoren misschien een snel algoritme zou opleveren! De theoretische slechtste-geval complexiteit is niet alleen $O(n*m)$ – net zoals het brute-kracht algoritme – maar in dit geval heb je zelfs $\Theta(n*m)$ stappen in het beste geval nodig (als je de hele tekst moet doorlopen) terwijl het brute-kracht algoritme in dat geval maar tijd $\Theta(n)$ vraagt. Maar als wij rekening houden met toepassingen waar m relatief klein is, wordt het niet alleen lineair (dat wordt het brute-kracht algoritme ook), maar presteert het zelfs bijzonder goed omdat de constante in de lineaire functie heel klein is! Dit algoritme is vooral een goede keuze als je alle matches zoekt en er zijn er veel in de tekst waarin je zoekt.
In sommige boeken kan je dit algoritme als shift-AND terugvinden en in anderen als shift-OR – de redenen daarvoor zullen jullie in de oefeningen zien.

Definitie 10 *Gegeven een zoekstring $z[]$ van lengte m en een tekst $t[]$ van lengte n.*
Dan definiëren wij de $m \times n$ matrix M als $M[i][j] = 1$ als de prefix uit de tekens t.e.m. positie i van $z[]$ met de deelstring die in $t[]$ op positie j eindigt gematcht kan worden en anders 0. *Of formeel:*

$$M[i][j] = \begin{cases} 1 & \textit{als } z[0] = t[j-i], \ldots, z[i] = t[j] \\ 0 & \textit{anders} \end{cases}$$

Hierbij hebben wij de rijen en kolommen nummers vanaf 0 gegeven in plaats van vanaf 1.

Voor het geval $t[]$ =ananas en $z[]$ =ana krijgen wij

	a	n	a	n	a	s
a	1	0	1	0	1	0
n	0	1	0	1	0	0
a	0	0	1	0	1	0

In de laatste rij staat dus een 1 op positie j ($M[m-1][j] = 1$) als en slechts als een match van het hele zoekwoord $z[]$ op positie j van de tekst eindigt. Deze matrix heeft $n * m$ elementen, die alle n te berekenen en in te vullen vraagt dus in principe tijd $O(m * n)$.
Maar hoe berekenen wij deze matrix?

Wij zullen een dynamisch programmeren algoritme gebruiken dat bovendien – door bitvectoren te gebruiken – parallel werkt: meerdere van de $M[i][j]$ worden tegelijk met hulp van vroegere waarden berekend.
Daarvoor definiëren (en berekenen) wij eerst de binaire karakteristieke vector C_x voor elke letter x in het alfabet. Deze vector heeft lengte m en

$$C_x[i] = \begin{cases} 1 & \text{als } z[i] = x \\ 0 & \text{anders} \end{cases}$$

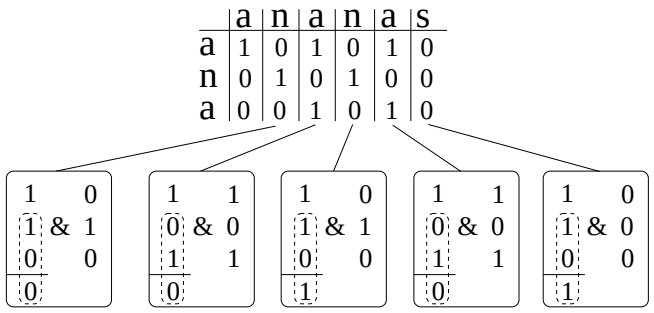
C_x houdt dus bij waar het teken x in $z[]$ staat. In ons voorbeeld $z[]$ =ana krijgen wij $C_a = (101)$, $C_n = (010)$ en $C_x = (000)$ voor alle anderen tekens x in het alfabet.
De eerste kolom is gemakkelijk: $M[i][0] = 0$ voor $1 \leq i < n$ omdat op positie 0 natuurlijk geen match van lengte groter dan 1 kan eindigen. En $M[0][0] = 1$ als en slechts als $z[0] = t[0]$
Stel nu dat wij kolom j al hebben en kolom $j+1$ willen berekenen. $M[0][j+1] = 1$ als en slechts als $t[j+1] = z[0]$ – of equivalent $M[0][j+1] = C_{t[j+1]}[0]$. Voor $i > 0$ hebben wij $M[i][j+1] = 1$ als en slechts als er een match van lengte i (de posities $0, \ldots, i-1$) tot positie j is **en** $t[j+1] = z[i]$. Of anders geschreven: $M[i][j+1] = M[i-1][j]$ & $C_{t[j+1]}[i]$ – en nu zien jullie misschien waar de naam van dit algoritme vandaan komt: als de $M[][j]$ en de karakteristieke vector $C_{t[j+1]}[]$ bitvectoren zijn dan is $M[][j+1] = \text{shift}^1(M[][j])$ & $C_{t[j+1]}[]$ als de shift-operatie shift^1 bit $i-1$ op positie i schuift en het nieuwe bit 0 op 1 zet. Als pseudocode ziet dat er als volgt uit – waarbij wij altijd alleen maar één kolom van de matrix bijhouden – zodra wij de nieuwe kolom berekend hebben, hebben wij de oude ten slotte niet meer nodig:

Algoritme 6 *(Het shift-AND algoritme)*

```
shiftAND(z,t)
{
// test of de string z in de string t voorkomt
// geeft TRUE terug als ja anders FALSE
// alle karakteristieke vectoren C[x][] en ook de vector kolom[]
// zijn bitvectoren -- dus bv. een unsigned long int als m<=64

  bereken_karakteristieke_vectoren(); // dat is straightforward

  if (z[0]=t[0])
```

```
      { if (|z|=1) return TRUE;
       kolom=(1,0,0,...,0) // lengte |z|
      }
    else  kolom=(0,0,0,...,0)

  for (j=1; j<|t|;j++)
  {
    kolom[]= SHIFT^1(kolom[]) & C[t[j]][];
    if (kolom[|z|-1]=1) return TRUE;
  }

  return FALSE;
}
```

Hoe deze operaties tot de tabel voor ons voorbeeld leiden, zien jullie in Figuur 22.



Figuur 22: De stappen van het shift-AND algoritme toegepast op $z[] = ana$ en $t[]$ =ananas (waarbij na de eerste match niet gestopt wordt).

Als je deze operaties algemeen wil implementeren dan vraagt het berekenen van de volgende kolom – zoals verwacht – tijd $O(m)$. Als m naar boven door een constante begrensd is, is dat dus constant – maar ook dat is niet echt een voordeel in vergelijking met andere algoritmen. Maar als de lengte van de zoekstring ten hoogste het aantal bits van een woord in de computer is – en voor heel veel toepassingen mag je zeker stellen dat de zoekstring niet meer dan 64 tekens bevat – dan kan je de bitvectoren in één computerwoord voorstellen en de bitmanipulaties gebruiken (<<,>>,&,|) die de processor kan uitvoeren. Dan zijn de bewerkingen niet alleen *constant* maar in feite *echt heel goedkoop*.
In gevallen waar je met heel lange woorden bezig bent, kan je eerst naar een prefix zoeken die nog in één computerwoord past. De lengte van een compu-

terwoord is op dit moment meestal 64 bits. Als het om *gewone teksten* gaat, zal deze prefix vermoedelijk maar voor een heel klein deel van de mogelijke startposities gematcht kunnen worden. Voor de zeldzame gevallen waar die gematcht wordt, pas je voor de rest dan de brute kracht manier toe om te zien of de rest van de zoekstring hier aansluit.
Een andere mogelijkheid is met bitvectoren te werken die meer dan één computerwoord gebruiken.

Oefening 45 *Geef een algoritme in pseudocode dat de verzameling van alle C_x in tijd $O(m)$ kan berekenen waarbij m de lengte van het gezochte woord $z[]$ is.*

Oefening 46 *Schrijf de C-code die de shift-AND operaties implementeert voor een woordlengte op de computer van WORDSIZE bits.*

- *voor een zoekwoord dat gegarandeerd ten hoogste WORDSIZE letters bevat*
- *voor een zoekwoord met onbeperkte lengte (maar werk hier wel met kolommen van dezelfde lengte als het woord!)*

Oefening 47 *Pas het shift-AND algoritme toe op het zoekwoord $z[]$ =*`zoeken` *en de tekst $t[]$ =*`iets_zoets_zoeken`

Oefening 48 *Wijzig het shift-AND algoritme zo dat het ook met wildcards* `?` *en* `*` *kan werken en nog altijd even goed presteert.*

Oefening 49 *Wij zijn eraan gewend dat* 1 *voor* `TRUE` *staat en* 0 *voor $FALSE$ en zo is het shift-AND algoritme ook beter uit te leggen en te verstaan. Maar nu dat je het algoritme* **hebt** *verstaan: wissel in Definitie 10 de betekenis van* 0 *en* 1*.*

- *Werk het algoritme uit voor deze gewijzigde definitie en geef de pseudocode.*
- *Schrijf de belangrijkste lijn van de C-code voor het algoritme dat het algoritme volgens deze nieuwe definitie implementeert.*
- *Welke versie zou je implementeren? Die met de interpretatie van* 1 *zoals in de tekst of zoals in de oefening? Waarom?*

Oefening 50 *Wijzig Algoritme 6 zo dat de lengte van een langste match beginnend met $z[0]$ terug wordt gegeven.*

shift-AND voor matches met fouten

Maar het shift-AND algoritme laat het ook toe matches nog meer te veralgemenen dan met wildcards: matches met een beperkt aantal mismatches – zonder op voorhand te specifiëren waar die moeten zitten. Wij zullen hier het algoritme zien dat het programma *agrep* te gronde ligt en dat door S. Wu en U. Manber in 1992 ontwikkeld werd. Ook hier ligt de klemtoon op de praktijk – dus het geval waar het toegelaten aantal fouten begrensd en klein is.
Wij zullen het hier op dit moment alleen maar hebben over *mismatches* – dus waar een letterteken met een **ander** letterteken gematcht moet worden. Je kan het ook over *fouten* in de betekenis van ontbrekende lettertekens (in de tekst of de zoekstring) hebben.

Definitie 11 *Gegeven een zoekstring $z[]$ van lengte m, een tekst $t[]$ van lengte n en een (klein) getal k.*
Dan definiëren wij de $m \times n$ matrix M^k als $M^k[i][j] = 1$ als de prefix uit de tekens t.e.m. positie i van $z[]$ met de deelstring die in $t[]$ op positie j eindigt gematcht kan worden waarbij er ten hoogste k mismatches optreden en anders 0. *Of formeel (waarbij wij gebruiken dat* ten hoogste k mismatches *betekent dat er ten minste $i + 1 - k$ goede matches zijn):*

$$M^k[i][j] = \begin{cases} 1 & \text{als er indices } 0 \leq l_0 < l_1, < \cdots < l_{i-k} \leq i \text{ bestaan zo dat} \\ & z[l_0] = t[j - i + l_0], \ldots, z[l_{i-k}] = t[j - i + l_{i-k}] \\ 0 & \text{anders} \end{cases}$$

Ook hierbij hebben wij de rijen en kolommen nummers vanaf 0 gegeven in plaats van vanaf 1.

De matrix $M^0[][]$ is dus de matrix $M[][]$ die wij al kennen – hier laten wij 0 fouten toe en wij weten hoe wij die moeten berekenen.
In de laatste rij van $M^k[][]$ staat dus een 1 op positie j ($M^k[m-1][j] = 1$) als en slechts als een match met ten hoogste k fouten van het hele zoekwoord $z[]$ op positie j van de tekst eindigt. Het idee lijkt dus heel sterk op het gewone shift-AND algoritme en de vraag is alleen maar hoe je $M^k[][]$ het best berekent. De eerste kolom is gemakkelijk voor alle $k > 0$: $M^k[][0] = (1, 0, \ldots, 0)$
Er zijn 2 mogelijkheden waarom voor een $0 < l \leq k$ de waarde van $M^l[i][j] = 1$ is (wat betekent: er zijn ten hoogste l mismatches van een prefix van lengte $i + 1$ met een string die op positie j eindigt). Wij veronderstellen dat $j > 0$ en $l > 0$ omdat wij voor deze waarden al weten hoe we $M^l[][j]$ berekenen.

Bovendien kunnen wij ook stellen dat $i > 0$ omdat $M^l[0][j] = 1$ voor alle $l > 0$.

- als er ten hoogste $l-1$ mismatches van de prefix met lengte i tot positie $j-1$ zijn (dus $M^{l-1}[i-1][j-1] = 1$) dan is het zonder belang of het letterteken op positie j gematcht wordt – er zijn zeker ten hoogste l fouten tot positie j.
- als er precies l mismatches van de prefix met lengte i tot positie $j-1$ zijn (dan is $M^l[i-1][j-1] = 1$) **en** $t[j] = z[i]$ dan zijn er ten hoogste l mismatches tot positie j.

In andere gevallen is er duidelijk geen match met de vereiste eigenschappen en moet $M^l[i][j] = 0$ zijn. Wij moeten de matrices dus in een volgorde berekenen zodat de waarden voor $l-1$ en $j-1$ al bekend zijn.
De kolom $M^l[.][j]$ kunnen wij dus berekenen als

$$M^l[][j] = (1, 0, \ldots, 0) \quad | \quad \text{shift}(M^{l-1}[][j-1]) \quad | \quad (\text{shift}(M^l[][j-1]) \quad \& \quad C[t[j]][])$$

Hierbij kan shift() bv. dezelfde operatie zijn als $\text{shift}^1()$ of het kan een shift zijn die de 0-bit op positie 0 schuift. Door de operatie $(1, 0, \ldots, 0) \quad |$ wordt deze bit toch al op 1 gezet.

Oefening 51 *Werk ook dit algoritme uit met de rol van* 0 *en* 1 *in Definitie 11 gewijzigd.*

Oefening 52 *Schrijf de pseudocode voor dit algoritme op.*

Oefening 53 *Pas het algoritme toe op* $z[]$ =`dank` *en* $t[]$ =`een_dansfeest` *met* 1 *toegelaten mismatch.*

Oefening 54 *Wij kunnen de definitie van* toegelaten fouten *in Definitie 11 ook zo beschrijven, dat er een perfecte match is met een woord dat wij uit het zoekwoord kunnen bouwen als wij ten hoogste k lettertekens door een ander letterteken vervangen. Stel dat het nu ook toegelaten is een letterteken uit de tekst te verwijderen (deletion) en dat dat ook als* 1 fout *telt (als de tekst dus bv.* `ik_ben_peter` *is dan zou dit algoritme voor de zoekstring* `peer` *zeggen dat er een match met 1 fout gevonden is). Geef de* **exacte** *definitie van* $\bar{M}^k[][]$ *voor dit concept en geef het algoritme om* $\bar{M}^k[][]$ *te berekenen.*

3.3 Benaderend string matching

Wij hebben al verschillende stappen in de richting van benaderend string-matching gedaan door wildcards of mismatches toe te laten. In dit deel willen wij het **eerst** over een speciaal geval hebben: het geval waar de rol van $z[]$ en $t[]$ symmetrisch is. Wij hebben hier dus niet één string die typisch kort is en een andere die lang is, maar ze zijn beide van ongeveer dezelfde lengte en wij willen weten *hoe verschillend* ze zijn. Je hebt bv. twee bestanden waarbij het ene bestand een gewijzigde versie van het andere bestand is en wilt weten hoe sterk het bestand gewijzigd werd. Dat wordt bv. door het Unix-commando *diff* berekend (waarbij *diff* ook nog zegt wat de wijzigingen zijn).
Als wij zeggen *hoe verschillend* dan hebben wij het in feite over een maat – of een afstand. Hoe je de definitie van een afstand kiest, hangt er sterk van af wat de context is. In de biologie zijn de strings bv. vaak DNA structuren en de afstand hangt ervan af hoeveel mutaties nodig zijn om van het ene DNA naar het andere te gaan. Maar je kan er ook best rekening mee houden hoe waarschijnlijk de mutaties zijn. Dat geldt natuurlijk ook voor teksten: je kan tellen hoeveel *wijzigingen* nodig zijn om de ene tekst in de andere te veranderen, maar dan is nog altijd niet duidelijk welke soort *wijzigingen* bedoeld zijn. Als de afstand moet beschrijven hoe waarschijnlijk het is dat het woord $t[]$ in feite hetzelfde woord zou moeten zijn als $t'[]$ dan zouden toegelaten wijzigingen bv. het weglaten van een letterteken, het toevoegen van een letterteken of het verwisselen van een letterteken zijn. Maar je zou ook hier waarschijnlijkheden kunnen gebruiken. Misschien is het waarschijnlijker dat lettertekens verwisseld worden die naast elkaar op het toetsenbord staan dan lettertekens die niet naast elkaar staan – of is het waarschijnlijker dat de volgorde van twee lettertekens fout is (bv. *volgrode* in plaats van *volgorde*) dan dat twee lettertekens verwisseld zijn (bv. *vilgorde* in plaats van *volgorde*), etc.
De afstand die wij hier gebruiken is dus gewoon één van vele mogelijke afstanden die allemaal in zekere omstandigheden zinvol kunnen zijn.

Definitie 12 *Voor de editeerafstand zijn 3 bewerkingen toegelaten:*

- *Het vervangen van een letterteken door een ander letterteken. Deze bewerking hangt nauw samen met de mismatches die wij in samenhang met het shift-AND algoritme al besproken hebben.*
- *Het verwijderen van een letterteken. Als het verwijderde letterteken positie i had, houden alle tekens die voor het verwijderen een positie $j < i$ hadden de positie en alle tekens die een positie $j > i$ hadden, hebben achteraf positie $j - 1$.*

- *Het toevoegen van een letterteken. Als het nieuwe letterteken na het toevoegen positie i heeft, houden alle tekens die voor het toevoegen een positie $j < i$ hadden de positie en alle tekens die een positie $j \geq i$ hadden, hebben achteraf positie $j + 1$.*

De editeerafstand $d(t, t')$ tussen twee strings $t[]$,$t'[]$ is het kleinst mogelijke aantal bewerkingen van deze soort toegepast op $t[]$ zodat het resultaat gelijk aan $t'[]$ is.
Deze afstand wordt soms ook de Levenshtein afstand *genoemd.*

Wij zullen dit aan een voorbeeld illustreren. Wij gebruiken de strings $t =$`lessen` en $t' =$`feesten`

```
l e s s e n        (1) vervang l door f

f e s s e n        (2) vervang s op positie 3 door t
0 1 2 3 4 5

f e s t e n        (3) voeg nieuwe e op positie 1 toe
0 1 2 3 4 5

f e e s t e n
0 1 2 3 4 5 6
```

Er bestaat dus een reeks met 3 stappen van $t =$`lessen` naar $t' =$`feesten`. En omdat er geen kortere reeks bestaat (gemakkelijk om te zien), geldt $d(t, t') = 3$.

Opmerking 13 *Voor strings $t[]$,$t'[]$, $t''[]$ geldt:*

(i) $d(t, t') \geq 0$

(ii) $d(t, t') = 0$ *als en slechts als* $t = t'$

(iii) $d(t, t') = d(t', t)$

(iv) $d(t, t'') \leq d(t', t) + d(t', t'')$

(v) $d(t, t') \leq \max\{|t|, |t'|\}$

Eigenschap (v) zegt dat de editeerafstand goed gedefinieerd is (hij kan niet oneindig zijn) en de eigenschappen (i) t.e.m. (iv) rechtvaardigen het over een *afstand* of een *metriek* te spreken.

Bewijs: De meeste punten volgen onmiddellijk uit de definitie. Het enige punt waarover wij misschien een beetje meer moeten nadenken is (iii) en dat punt zullen wij hier expliciet bewijzen:

Als $d(t,t') = 0$ dan is $t = t'$ en dan geldt zeker dat $d(t,t') = d(t',t)$. Maar je kan elke bewerking b zodat b toegepast op t de string t' oplevert omkeren: als de bewerking is dat $t[i]$ vervangen wordt door letterteken x dan is $t'[i] = x$ en kunnen wij als omgekeerde bewerking $t'[i]$ vervangen door $t[i]$. Als een nieuw teken op positie i in $t[]$ wordt toegevoegd dan kunnen wij als omgekeerde bewerking teken $t'[i]$ verwijderen. Als teken $t[i]$ wordt verwijderd dan kunnen wij omgekeerd in t' het teken $t[i]$ op positie i toevoegen.

Stel dus dat $d(t,t') = n \geq 1$. Dan is er een reeks $t = t_0, t_1, \ldots t_{n-1}, t_n = t'$ met $d(t_i, t_{i+1}) = 1$ voor $0 \leq i < n$. Als wij de bewerkingen tussen twee opeenvolgende strings omkeren hebben wij een reeks die t' in t omvormt. Dus $d(t',t) \leq n$. Maar als $d(t',t) < n$ zou op dezelfde manier volgen $d(t,t') < n$ – een tegenstrijdigheid. Dus geldt $d(t',t) = d(t,t')$.

■

Om de editeerafstand efficiënt te berekenen hebben wij nog een lemma nodig:

Lemma 14 *Gegeven $t[]$ en $t'[]$. Dan is er een reeks van $k = d(t,t')$ bewerkingen $b_1, \ldots, b_k$ die $t[]$ naar $t'[]$ wijzigt zodat als i_j de bij bewerking b_j betrokken index is, geldt $i_1 \leq i_2 \leq \cdots \leq i_k$.*

Dit betekent dat wij beginnend aan de linkerkant van de string de string kunnen wijzigen en dan doorgaan naar de rechterkant waarbij wij nooit reeds vroeger bezochte delen opnieuw moeten wijzigen. Dat is natuurlijk heel nuttig voor een algoritme! Het betekent dat wij in feite de prefixen in volgorde van hun lengte wijzigen!

Bewijs: Dit kan je bewijzen door gewoon naar alle combinaties van bewerkingen te kijken waar eerst index i betrokken is en daarna index j waarbij $j < i$. In elk geval kan je ofwel bewijzen dat er een kortere reeks van bewerkingen is – wat een tegenstrijdigheid zou zijn – (bv. als je eerst een teken op positie i toevoegt en het dan door een ander teken vervangt) of dat je de bewerkingen kan vervangen door 2 bewerkingen met indices in volgorde.

Maar dit bewijs bevat noch interessante ideeën noch is het moeilijk – wij zullen het hier dus niet geven.

■

Eén van de doelen van de lessen over algoritmen is ook in te kunnen schatten welke technieken het best toegepast kunnen worden om een probleem op te lossen. Soms heb je dan zo'n gevoel dat het ene probleem op een zekere manier op het andere lijkt en dus misschien dezelfde technieken toegepast kunnen worden. Misschien hebben sommigen van jullie de indruk dat dit probleem een beetje op het probleem lijkt waar wij volgorden van matrices moesten bepalen om zo efficiënt mogelijk te kunnen vermenigvuldigen. Beide hebben een gelijkaardige *lineaire* structuur...
Wij zullen – net zoals voor het matrixvermenigvuldigingsprobleem en het shift-AND algoritme – ook hier dynamisch programmeren toepassen. Wij berekenen in feite niet alleen de editeerafstand tussen de strings $t[]$ en $t'[]$, maar tussen alle prefixen van $t[]$ en $t'[]$. Hier wordt dan heel duidelijk hoe dicht dit algoritme bij het shift-AND algoritme staat. Het is alleen jammer dat hier niet alleen de waarden 0 en 1 kunnen voorkomen die het mogelijk gemaakt hebben de bewerkingen door bitvectoren te paralleliseren. Als $n = |t|$ en $m = |t'|$ dan schrijven wij voor $0 \leq i \leq n$ en $0 \leq j \leq m$ de notatie $d[i][j]$ voor de afstand tussen de prefix van lengte i van $t[]$ (dus $t[0], t[1], \ldots, t[i-1]$) en de prefix van lengte j van $t'[]$ (dus $t'[0], t'[1], \ldots, t'[j-1]$). De notatie duidt er al op dat wij dat in een 2-dimensionale array zullen bijhouden. De waarden $d[i][0] = i$ en $d[0][j] = j$ voor alle i, j zijn duidelijk omdat als één van de strings leeg is alle lettertekens ofwel verwijderd moeten worden ofwel toegevoegd moeten worden. Als wij veronderstellen dat wij op het moment dat wij $d[i][j]$ willen berekenen de waarden van $d[i'][j']$ voor alle tweetallen (i', j') die lexicografisch kleiner zijn dan (i, j) al kennen, dan kunnen wij als volgt argumenteren:

- als $t[i-1] = t'[j-1]$ dan kan het laatste letterteken ongewijzigd blijven en dus $d[i][j] \leq d[i-1][j-1]$. Zie Oefening 56.

Als $t[i-1] \neq t'[j-1]$ dan zijn er 3 bewerkingen die de laatste bewerking in een kortst mogelijke reeks kunnen zijn. Wij gebruiken hier de notatie $t[k]$ om het letterteken te beschrijven dat voor de wijzigingen op positie k staat, maar merk op dat het voor de laatste wijziging door toevoegingen of verwijderingen op een helemaal andere positie kan staan.

- $t[i-1]$ vervangen door $t'[j-1]$. Dan is $d[i][j] = d[i-1][j-1] + 1$.
- $t'[j-1]$ aan de (misschien gewijzigde) $t[0], t[1], ..., t[i-1]$ toevoegen. Dan was de gewijzigde string ervoor $t'[0], t'[1], \ldots, t'[j-2]$ en die werd bereikt door $t[0], t[1], \ldots, t[i-1]$ te wijzigen. Wij hebben dus $d[i][j] = d[i][j-1] + 1$.

- $t[i-1]$ verwijderen. Dan wordt $t[0], t[1], \ldots, t[i-2]$ veranderd naar $t'[0], t'[1], \ldots, t'[j-1]$ en achteraf het overbodige letterteken $t[i-1]$ verwijderd. Dus geldt $d[i][j] = d[i-1][j] + 1$.

Deze observaties kunnen nu gebruikt worden om een algoritme in pseudocode te formuleren. Daarbij zijn sommige dingen voor de betere verstaanbaarheid anders opgeschreven dan in een efficiënt programma – soms weet je bv. al dat een waarde beter is dan de waarde van `best` in de code.

Algoritme 7 *(Dynamisch programmeren om de editeerafstand te berekenen)*

```
editeerafstand(t[],t'[])
{
// het wordt verondersteld dat t,t' beide niet leeg zijn en dat de
// lengte |t|, resp. |t'| is. De editeerafstand tussen t en t' wordt
// teruggegeven.


for (i=0;i<=|t|;i++) d[i][0]=i;
for (j=0;j<=|t'|;j++) d[0][j]=j;

for (i=1;i<=|t|;i++)
{
  for (j=1;j<=|t'|;j++)
      {  // als k lettertekens beschouwd worden heeft het laatste
         // index k-1
         if (t[i-1]=t'[j-1]) wisselkost=0;
            else wisselkost=1;
         d[i][j] = min{d[i-1][j-1]+wisselkost,
                       d[i-1][j]+1, d[i][j-1]+1};
      }
}

return d[|t|][|t'|];
}
```

Als wij de werking van dit algoritme eens beschouwen aan de hand van ons voorbeeld $t[]$ =lessen en $t'[] = feesten$ dan krijgen wij de volgende tabel. De

lijn toont aan hoe de waarden verkregen zijn en ook één van de minst kostelijke mogelijkheden om de string `lessen` te wijzigen om de string `feesten` te verkrijgen.

		f	e	e	s	t	e	n
	0	1	2	3	4	5	6	7
l	1	1	2	3	4	5	6	7
e	2	2	1	2	3	4	5	6
s	3	3	2	2	2	3	4	5
s	4	4	3	3	2	3	4	5
e	5	5	4	3	3	3	3	4
n	6	6	5	4	4	4	4	3

Omdat de tijd voor één doorloop van de binnenste lus duidelijk constant is, hebben wij het volgende resultaat:

Lemma 15 *De editeerafstand tussen twee strings van lengte n en m kan in tijd $O(n * m)$ berekend worden.*

Voor de echte implementatie is het natuurlijk belangrijk geheugen te besparen en misschien niet de hele matrix in het geheugen te houden. Los zeker Oefening 57 op!
Het is niet alleen interessant te weten wat de editafstand is, maar ook de bewerkingen te kennen om de ene string naar de andere te wijzigen. Als je bv. een heel groot bestand hebt die op de twee computers zit dan is het misschien beter als die op de ene computer gewijzigd wordt alleen de nodige bewerkingen naar de andere computer te sturen dan het hele bestand.
De bewerkingen moeten ook niet altijd alleen met lettertekens gebeuren. In het Unix-commando *diff* worden bv. lijnen als characters geïnterpreteerd en de bewerkingen zijn dan verwijderen, toevoegen of vervangen van lijnen... Er zijn dus talrijke variaties en toepassingen van deze algoritmen...

Oefening 55 *Gebruik het beschreven algoritme om de editeerafstand tussen* `t=roestig` *en* `t'=oesters` *te berekenen. Toon de tabel en duidt aan hoe de getallen in de tabel werden berekend. Beschrijf ook de mogelijke kortste reeksen van bewerkingen.*

Oefening 56 *Stel dat* `t[]=t[0],t[1],. . . ,t[i]`, `t'[]=t'[0],t'[1],. . . ,t'[j]` *en* `t[i]=t'[j]`. *Schrijf* t_{i-1} *voor* `t[0],t[1],. . . ,t[i-1]` *en analoog* t'_{j-1} *voor* `t'[0],t'[1],. . . ,t'[j-1]`

Is dan altijd $d(t, t') = d(t_{i-1}, t'_{j-1})$?
Bewijs dat of geef een tegenvoorbeeld.

Oefening 57 *Herschrijf de pseudocode van Algoritme 7.* **Stel** *daarbij dat het antwoord op de vraag in Oefening 56* `ja` *is en let erop zo weinig mogelijk geheugen te gebruiken. De strings kunnen natuurlijk te lang zijn om de hele matrix* `d[][]` *efficiënt in het geheugen te houden!*

Oefening 58 *Bewijs gedetailleerd dat voor de volgende gevallen van twee opeenvolgende bewerkingen b_i, b_{i+1} die vervangen kunnen worden door minder bewerkingen of twee bewerkingen waarvan de tweede een teken met een groter index gebruikt dan de eerste.*

- *b_i verwijdert een teken op positie j, b_{i+1} voegt een teken op positie j toe.*
- *b_i voegt een teken op positie j toe, b_{i+1} verwijdert een teken op positie $j' < j$.*
- *b_i verwijdert een teken op positie j, b_{i+1} vervangt een teken op positie $j' < j$ door een ander teken.*

Bepaal precies de paren van bewerkingen die in een reeks van minimale lengte zoals in Lemma 14 – dus met stijgende indices – gelijke indices moeten hebben en niet door even veel of minder bewerkingen met strict stijgende indices vervangen kunnen worden. Hier is een redenering die niet alle details geeft (dus de indices, etc) voldoende.

Oefening 59 *Definieer een nieuwe afstand $d'()$ door behalve de drie al gekende bewerkingen ook nog een nieuwe bewerking toe te laten: het verwisselen van twee op elkaar volgende lettertekens. Met* `t=lui` *en* `t'=liu` *is dus $d(t, t') = 2$ en $d'(t, t') = 1$. Beschrijf een algoritme om $d'()$ te berekenen in pseudocode en geef uitleg waarom het algoritme juist is.*

Oefening 60 *Wijzig de pseudocode van het algoritme zo dat ook de nodige bewerkingen om $t[]$ naar $t'[]$ te wijzigen worden bijgehouden.*

Oefening 61 *Gegeven 2 woorden $z_1[]$ en $z_2[]$. Het doel is deze twee woorden zo te wijzigen dat ze gelijk zijn. Maar de enige bewerking is dat je lettertekens tot $z_1[]$ en $z_2[]$ mag toevoegen. Beschrijf een efficiënt algoritme dat de minimale lengte van een woord $z'[]$ bepaalt zodat $z_1[]$ en $z_2[]$ omgevormd kunnen worden naar $z'[]$.*

De beste benaderende match

Nu zullen wij het terug over het probleem hebben matches van een korte string in teksten te vinden – alleen dat wij het hier over de beste benaderende match hebben en geen exacte matches. Hoe wij *benaderende matches* in de betekenis van *een gegeven aantal mismatches* efficiënt kunnen berekenen, hebben wij al gezien. Hier zullen wij het er nu over hebben hoe je voor een gegeven string $z[]$ en een gegeven tekst $t[]$ de deelstring van $t[]$ met de kleinste editeerafstand kan vinden.
Het eerste idee zou misschien zijn het algoritme voor de editeerafstand tussen twee strings op alle deelstrings van $t[]$ toe te passen. Maar met $m = |z|$ en $n = |t|$ zijn er $(\sum_{i=0}^{n-1}(n-i))+1 = n(n+1)/2+1$ deelstrings van $t[]$ (op positie i starten $n-i$ mogelijke niet lege strings en dan nog de ene lege string 1 keer geteld). Om $z[]$ met een string van lengte l te vergelijken hebben wij $O(m*l)$ stappen nodig en als wij dat voor alle mogelijke lengten l opsommen, krijgen wij een totale kost van $O(m*n^3)$. En – veronderstellend dat het een goede bovengrens is – betekent dat natuurlijk dat een hierop gebaseerd programma vrij traag zal zijn voor grote teksten.

Oefening 62 *Als je echt naar* **alle** *deelstrings van $t[]$ kijkt, is dat natuurlijk niet bijzonder slim. Je ziet onmiddellijk dat je naar heel lange deelstrings niet moet kijken (waarom?). Maar wat is "*heel lang*"?*
Geef een functie $f()$ zodat voor alle $z[]$ en $t[]$ een deelstring in $t[]$ bestaat met lengte ten hoogste $f(|z|)$ en minimale editeerafstand van $z[]$. Kies $f()$ minimaal – dus zo dat er in feite $z[]$ en $t[]$ bestaan waarin geen kortere beste benaderende matches zijn.

Maar aan de andere kant zien jullie ook dat bij deze manier van doen veel deelresultaten altijd opnieuw berekend worden – de afstand voor dezelfde prefix wordt bv. voor elk woord berekend waarvan het een prefix is. En dan is natuurlijk duidelijk dat het veel sneller kan door opnieuw het principe van dynamisch programmeren toe te passen. Maar omdat er dan nog altijd $O(m*n^2)$ editeerafstanden berekend en opgeslagen worden zal de complexiteit nog altijd ten minste $O(m*n^2)$ zijn.
Wij gaan dus nog 1 stap verder: voor de j-de letter in de tekst (dus $t[j-1]$) en een lengte i van een prefix slaan wij niet de afstanden van $z[0],\ldots,z[i-1]$ van alle deelwoorden van $t[]$ op die op positie $j-1$ eindigen, maar alleen de minimale editeerafstand van een deelwoord dat er eindigt.
Of precies:

$$D[i][j] = \min\{d((z[0],\ldots,z[i-1]),(t[l],\ldots,t[j-1]))|0 \leq l \leq j\}$$

waarbij $(z[0], \ldots, z[-1])$ en $(t[j], \ldots, t[j-1])$ lege woorden zijn. Zie ook hier Oefening 56.
Ook hier zijn de eerste rij en de eerste kolom van $D[][]$ gemakkelijk om te berekenen:
$D[i][0] = i$ voor alle i omdat alle i lettertekens verwijderd moeten worden.
$D[0][j] = 0$ voor alle j omdat de beste match van de lege string natuurlijk met de lege string is en de beste afstand dus 0 is.
Volledig analoog met het berekenen van de editeerafstand tussen twee strings kunnen wij nu voor $i > 0$ en $j > 0$ argumenteren:

$$D[i][j] = \min\{D[i-1][j] + 1, D[i][j-1] + 1, D[i-1][j-1] + g(i,j)\}$$

waarbij $g(i,j) = 0$ als $z[i-1] = t[j-1]$ en anders 1.
Als wij deze tabel ingevuld hebben, moeten wij alleen nog de kleinste waarde in de laatste rij zoeken (dus de kleinste afstand die het hele woord $z[]$ heeft). Dat is de gezochte minimale afstand.

Algoritme 8 *(Dynamisch programmeren om de kleinste editeerafstand van een deelwoord te berekenen)*

```
shortest_editeerafstand_deelwoord(z[],t[])
{
// het wordt verondersteld dat z,t beide niet leeg zijn en dat de
// lengte |z|, resp. |t| is. De kleinste editeerafstand van z met
// een deelwoord van t wordt teruggegeven.


for (i=0;i<=|z|;i++) D[i][0]=i;
for (j=0;j<=|t|;j++) D[0][j]=0;

for (i=1;i<=|z|;i++)
{
  for (j=1;j<=|t|;j++)
      {  // als k lettertekens beschouwd worden, heeft het laatste
         // index k-1
         if (z[i-1]=t[j-1]) wisselkost=0;
            else wisselkost=1;
         D[i][j] = min{D[i-1][j-1]+wisselkost,
                       D[i-1][j]+1, D[i][j-1]+1};
      }
```

```
}

best=oneindig;
for (i=0;i<=|t|;i++) if (D[|z|][i]<best) best=D[|z|][i];
return best;
}
```

Merk op dat dit algoritme bijna identiek is aan Algoritme 7. Alleen de initialisatie van de eerste rij en eerste kolom verschillen en op het einde wordt nog het kleinste getal in de laatste rij gezocht. Wij hebben dus onmiddellijk:

Lemma 16 *De kleinste editeerafstand tussen een string $z[]$ van lengte n en een deelwoord van een string $t[]$ van lengte m kan in tijd $O(n * m)$ berekend worden.*

Maar ook hier is de implementatie en het besparen van geheugen belangrijk. Zie Oefening 68.
De werking van het algoritme voor het voorbeeld $z[]$ =`examen` en $t[]$ =`de_armen_vouwen` kan in de volgende tabel gezien worden.

		d	e	_	a	r	m	e	n	_	v	o	u	w	e	n
	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
e	1	1	0	1	1	1	1	0	1	1	1	1	1	1	0	1
x	2	2	1	1	2	2	2	1	1	2	2	2	2	2	1	1
a	3	3	2	2	1	2	3	2	2	2	3	3	3	3	2	2
m	4	4	3	3	2	2	2	3	3	3	3	4	4	4	3	3
e	5	5	4	4	3	3	3	2	3	4	4	4	5	5	4	4
n	6	6	5	5	4	4	4	3	2	3	4	5	5	6	5	4

Oefening 63 *Pas Algoritme 8 toe op het woord $z[]$ =*`zoeken` *en $t[]$ =*`koekjes_kweken`*. Bouw de tabel op en toon aan wat de beste match is en welke wijzigingen nodig zijn om* `zoeken` *in deze match te wijzigen.*

Oefening 64 *Pas Algoritme 8 toe op het woord $z[]$ =*`test` *en $t[]$ =*`beste_tekst`*. Bouw de tabel op en toon aan wat de beste matches zijn en welke wijzigingen nodig zijn om* `test` *in deze matches te wijzigen.*

Oefening 65 *Beschrijf een algoritme in pseudocode dat als $z[]$, $t[]$ en de tabel die door Algoritme 8 opgebouwd wordt ingevoerd worden de beste match teruggeeft.*

Oefening 66 *Beschrijf een algoritme in pseudocode dat voor een woord $z[]$ van lengte m en een tekst $t[]$ van lengte n alle editeerafstanden van $z[]$ met deelwoorden uit $t[]$ berekent in tijd $O(m*n^2)$. Geef uitleg over de complexiteit van jouw algoritme.*

Oefening 67 *Stel dat de kost van een verwijder-, toevoeg- en wisselbewerking om de ene string in de andere om te vormen niet constant 1 is maar dat elke van de 3 bewerkingen een individuele (positieve) kost heeft. Welke resultaten zijn nog altijd geldig en hoe kan je voor dit probleem een efficiënt algoritme schrijven?*

Oefening 68 *Herschrijf de pseudocode van Algoritme 8.* **Stel** *daarbij dat het antwoord op de vraag in Oefening 56* `ja` *is en let erop zo weinig mogelijk geheugen te gebruiken. Vooral hier kan de tekst natuurlijk veel te lang zijn om de hele matrix* `D[][]` *in het geheugen te houden! Welke van de twee for-lussen is beter de buitenste lus?*

4 Compressiealgoritmen

Stel dat jullie de string
3, 14159265358979323846264338327950288419716939937510582097494459 23
willen comprimeren. Dan zouden jullie misschien zeggen dat dat een benadering van Pi is en dus voor de meeste toepassingen zeker 3, 141592653 voldoende precies is (en gelijk hebben). Wat jullie dan doen, is compressie *met verlies*. Het resultaat van de compressie laat het niet toe de oorspronkelijke data te herstellen. Om te beslissen wat belangrijk is en wat niet moet je weten wat de data voorstelt en hoe hij geïnterpreteerd wordt. Als iemand bv. alleen in elk vijfde cijfer geïnteresseerd was, zou deze manier van compressie heel slecht zijn: je houdt overbodige informatie bij en gooit belangrijke informatie weg. Belangrijke formaten zoals jpg of mp3 zijn ook compressie met verlies: je kan de oorspronkelijke foto of het oorspronkelijke geluid niet reconstrueren. De formaten zijn erop gebaseerd dat geweten is hoe ons zicht en gehoor functioneren. Omdat het voor dergelijke formaten ook vooral belangrijk is te verstaan hoe de waarneming werkt (wat niet ons onderwerp is) zullen wij die niet bespreken maar ons alleen bezig houden met compressie zonder verlies – dus met compressiealgoritmen die het toelaten elke bit van het originele bestand te reconstrueren.
Een eerste vraag zou natuurlijk zijn of er een compressiealgoritme is dat elk bestand kan comprimeren en het antwoord is duidelijk *nee*: Omdat wij elk bestand willen reconstrueren moet een compressiealgoritme een bijectie definiëren tussen de gecomprimeerde bestanden en de bestanden die gecomprimeerd worden. Als twee bestanden op hetzelfde bestand afgebeeld zouden worden, zou je het oorspronkelijke bestand niet kunnen reconstrueren! Maar hoe minder bytes hoe minder mogelijke bestanden – dus kan dat zeker niet. En even gemakkelijk kunnen wij zien dat als er ook maar één bestand is dat echt gecomprimeerd wordt (dus waar het resultaat korter is) dat er dan ook tenminste één bestand moet zijn dat langer wordt.

Oefening 69

- *Bewijs expliciet dat als er een routine voor het comprimeren van bestanden is die een bestand echt comprimeert dat er dan ook ten minste één bestand is dat door deze routine langer gemaakt wordt.*

- *Een een beetje vage uitspraak: elk bestand kan gecomprimeerd worden. Schrijf deze uitspraak op een* **precieze** *manier met kwantoren (voor elk bestand bestaat...) en bewijs de precieze uitspraak of vind een tegenvoorbeeld.*

Maar hoeveel bestanden van een gegeven lengte kan je met een gegeven compressiealgoritme comprimeren? Er zijn

$$\sum_{i=0}^{n} 256^i = \frac{1}{255} 255 \sum_{i=0}^{n} 256^i = \frac{1}{255}(256^{n+1} - 1)$$

bestanden met ten hoogste n bytes. Als wij nu bestanden met $n+1$ bytes willen comprimeren (waarbij wij altijd ten minste één byte willen winnen) dan is het beeld van een gecomprimeerd bestand een bestand met ten hoogste n bytes. Omdat er 256^{n+1} bestanden met $n+1$ bytes maar alleen maar ongeveer $256^{n+1}/255$ bestanden met ten hoogste n bytes zijn, kan dus **ten hoogste** $1/255$de van de bestanden gecomprimeerd worden – en daarvoor moeten wij dan nog *betalen* omdat andere bestanden langer worden! Dat klinkt niet goed...
Maar onze ervaring is helemaal anders: de meeste bestanden waarop wij compressiealgoritmen zoals gzip, zip, bzip2, etc. toepassen, kunnen gecomprimeerd worden – in sommige gevallen zelfs heel sterk. De reden lijkt op de reden waarom metaheuristieken werken: als wij het maximum van een toevallige functie zouden moeten vinden, zouden metaheuristieken nutteloos zijn. Zij werken goed omdat de problemen waarmee mensen meestal bezig zijn structuur vertonen – ze zijn helemaal niet toevallig! En hier is het hetzelfde: de bestanden die wij normaal willen comprimeren zijn niet toevallig. Het zijn bv. tekstbestanden met woorden uit een zekere taal. In dergelijke bestanden komen sommige letters veel vaker voor dan anderen – en dat is een structuur die toevallige bestanden niet tonen. Dergelijke structuren maken het mogelijk bestanden te comprimeren. Alle technieken die wij zullen zien, zullen slecht presteren op bestanden die gewoon toevallige bits bevatten. En wij hebben net getoond dat dat ook niet anders kan!

Oefening 70 *Wij hebben aangetoond dat je met een gegeven compressiealgoritme ten hoogste ongeveer* $1/255$ *van alle bestanden kan comprimeren. Maar dan hebben wij aanvaard dat het misschien een compressie van maar 1 byte is.*

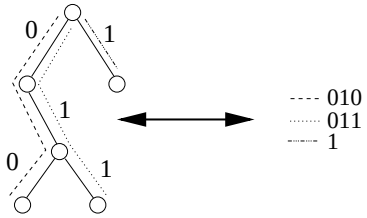
- *Bereken hoe groot het aandeel is van bestanden die door een vast compressiealgoritme met ten minste* 100 *bytes gecomprimeerd kunnen worden.*
- *Geef een bovengrens voor de kans dat een vast compressiealgoritme een toevallig bestand van* $5MB$ *of meer om ten minste de helft kan comprimeren*

Oefening 71 *Wij hebben net gezien dat je maar* $1/255$*ste van alle bestanden kan comprimeren. Stel nu dat elke computer ongelofelijk veel ruimte ter beschikking heeft en op voorhand al grote delen kan opslaan die in bestanden kunnen voorkomen (dus heel grote voorgëimplementeerde woordenboeken – bv. alle bestanden t.e.m. 500 bytes) zodat je op de volgende manier kan comprimeren: je slaat gewoon de index van de onderdelen op en stuurt die door. De index van de woorden kan je natuurlijk nog eens met andere compressiealgoritmen bewerken. Bepaal voor deze manier van comprimeren een bovengrens voor het aantal bestanden dat je kan comprimeren. Resultaten en bewijzen uit de les mogen geciteerd worden en moeten niet herhaald worden.*

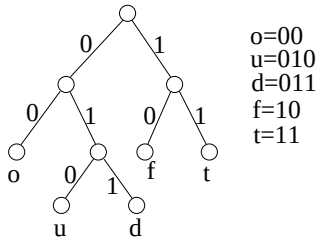
Een eerste aanzet:
Stel dat je niet meer een vaste lengte voor alle codewoorden (bv. bytes) wil hebben, maar variabele lengte van de codewoorden toelaat. Je kan bv. bytes coderen door nullen in het begin weg te laten. Dus bv. `00011111` als `11111`. Dat lijkt al ruimte te besparen, maar het probleem is natuurlijk dat je op deze manier niet meer kan herkennen waar het ene woord stopt en het volgende begint als je ze in een bestand achter elkaar schrijft. Waar zijn bv. de grenzen in `111110110101011010111`?
Eén manier om dit op te lossen is het gebruik van prefix-codes:

Definitie 17 *Een verzameling* M *van woorden (met tekens uit een zeker alfabet) heet een prefix-code als voor elk woord* $w = w_1, w_2, \ldots, w_m$ *geen woord* $w' = w'_1, w'_2, \ldots, w'_n$ *bestaat met* $n > m$ *en* $w_i = w'_i$ *voor alle* $1 \leq i \leq m$*.*

Je kan dus gemakkelijk herkennen wanneer je het einde van een woord hebt bereikt – gewoon omdat als je doorgaat er geen element in de verzameling zit dat door de langere code gecodeerd zou kunnen zijn! In de toekomst zal het alfabet waarover wij het hebben gewoon $\{0, 1\}$ zijn – wij hebben het dus over woorden van bits.
Prefixcodes kan je altijd in de vorm van een boom voorstellen waar de bogen gelabeld zijn met letters uit het alfabet. De codewoorden corresponderen dan met de labellingen langs de paden van de wortel naar de bladeren.
De prefixeigenschap correspondeert dan met het bereiken van een blad: er is geen langer woord met hetzelfde begin omdat het blad anders een kind had (en dan zou het geen blad zijn).
De rijen van nullen en enen staan normaal voor tekens – zoals bv. de ASCII tekens beschreven worden door rijen van nullen en enen met lengte 8. In een boom kunnen wij dat aanduiden door de letters aan de bladeren te schrijven. Wat zou bv. de string 00010011 coderen als wij de code nemen die in Figuur 24 beschreven is?



Figuur 23: Een kleine prefixboom en de bijbehorende codewoorden.



Figuur 24: Een kleine prefixboom, de bijbehorende codewoorden en tekens.

Je begint met het eerste bit en volgt de takken in de boom. Als je een blad bereikt, schrijf je het teken dat bij dit blad behoort op. Dan is één codewoord gedecodeerd en beginnen wij met de volgende bit opnieuw bij de wortel om het volgende teken op dezelfde manier te decoderen. Als je dat toepast tot alle bits gelezen zijn, heb je het hele woord gedecodeerd.
In het voorbeeld lezen wij eerst 00 en hebben het blad o bereikt. Dan beginnen wij opnieuw met de wortel en bereiken het blad u na 010, beginnen opnieuw en bereiken d na 011. De string die er gecodeerd was was dus het woord *oud*.

Oefening 72 *Wat is het juiste antwoord op de vraag wat de string* 100001011 *codeert als wij de code nemen die in Figuur 24 beschreven is?*

Als wij deze manier van coderen toepassen, kunnen de codes voor verschillende tekens ook een verschillende lengte hebben. Als wij een code willen hebben die zo kort mogelijk is, is het dus verstandig korte codes voor tekens te gebruiken die vaak opduiken en de langere codes voor tekens die minder vaak gebruikt worden. Om een voordeel aan deze techniek te hebben, moet je wel bestanden hebben waar de tekens niet *toevallig met gelijke kans* verdeeld zijn, maar die structuur vertonen – waar sommige tekens dus veel vaker opduiken dan anderen.
Als wij een alfabet A hebben en voor $x \in A$ het aantal keren dat x in het bestand zit dat wij willen comprimeren $a(x)$ is en de lengte van zijn code $l(x)$ dan is de lengte van het gecodeerde bestand $\sum_{x \in A}(a(x) * l(x))$ bits. Wij

hebben geen invloed op de functie $a()$ – het bestand is gegeven – maar wij kunnen de codering kiezen en dus invloed op de $l(x)$ uitoefenen.
Ons doel is dus de codering op een manier te kiezen dat $\sum_{x \in A}(a(x) * l(x))$ minimaal is.

4.1 Huffman codering

Huffman codering is een gretige manier om een code te construeren. Maar terwijl gretige algoritmen vaak alleen maar benaderende resultaten opleveren kunnen wij in dit geval aantonen dat het resultaat optimaal is.
Huffman codering werkt op de volgende manier:

- Doorloop eerst het bestand en bepaal voor elk teken x de frequentie $a(x)$.

Nu zullen wij de coderingsboom stap voor stap opbouwen. De boom zal niet uniek bepaald zijn, maar alle bomen die je op deze manier krijgt, zijn even goed. Wij veronderstellen dat er ten minste twee verschillende lettertekens in de tekst staan:

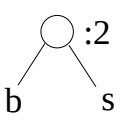
Algoritme 9 Huffman codering:

- *maak een verzameling van bomen aan waarin elke letter met een strikt positieve frequentie een boom met één top vormt.*
- *herhaal de volgende stap tot er nog maar één enkele boom in de verzameling zit:*
  - *Kies de wortel X van een boom met de kleinste frequentie en dan de wortel Y van een boom met de kleinste frequentie van de bomen die niet X zijn. Maak één nieuwe boom Z door van de wortels van X en Y de twee kinderen van één nieuwe wortel te maken. De frequentie $a(Z)$ wordt gedefinieerd als $a(Z) = a(X) + a(Y)$.*

De lettertekens zijn achteraf de bladeren van deze boom.

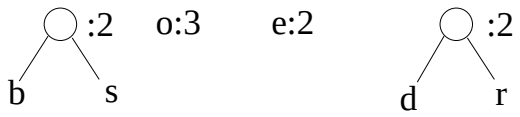
Als voorbeeld zullen wij nu de korte tekst **boosdoeer** coderen waar de lesgever een n is vergeten:

b:1 o:3 e:2 s:1 d:1 r:1

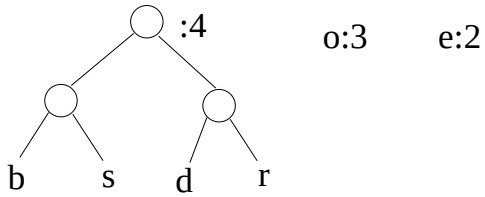
eerste stap: 

o:3 e:2 d:1 r:1

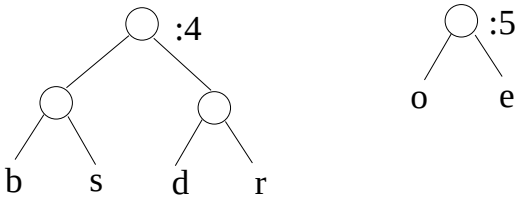
tweede stap:



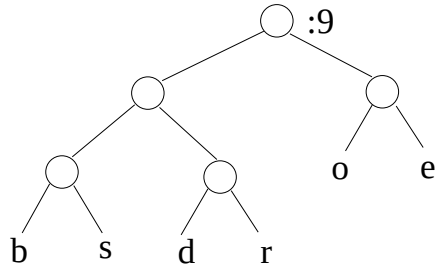
derde stap:
(niet uniek!)



vierde stap:



vijfde stap:



Deze laatste boom die alle lettertekens bevat is de Huffman-boom van de code. De codes van de lettertekens zijn dus:

b: 000

o: 10

e: 11

s: 001

d: 010

r: 011

De code van **boosdoeer** is de Huffman-boom samen met de string 000101000101010111011.

Oefening 73 *Bereken de Huffman codering voor* bananenboom.

Inderdaad is dat een gretig algoritme, en wij hebben gezien dat gretige algoritmen soms alleen benaderingen berekenen – en in sommige gevallen zelfs slechte benaderingen. Maar in dit geval krijgen wij inderdaad de optimale oplossing:

Stelling 18 *Voor een gegeven alfabet $x_1, \ldots, x_k$, $k \geq 2$ en frequenties $a(x_i)$ berekent het Huffman algoritme een prefixcode (boom) T waarvoor*

$$Q(T) = \sum_{i=1}^{k} l(x_i) * a(x_i)$$

minimaal is, waarbij $l(x_i)$ de lengte van het codewoord voor x_i is.

Bewijs: Let op: voor het Huffman algoritme zijn de letters in het alfabet natuurlijk kleine boompjes. Wij zullen het dus altijd over bomen hebben.

Opmerking: In een optimale boom met ten minste 3 toppen heeft elke top ofwel 0 ofwel 2 kinderen. Toppen met maar 1 kind zou je gewoon kunnen verwijderen (en de ouder en het kind met elkaar verbinden of de top alleen maar verwijderen als het de wortel was) en het resultaat zou een betere boom zijn: voor alle bladeren x blijft $l(x)$ gelijk of wordt kleiner. Dat betekent ook dat elke top (behalve de wortel) een broer heeft (een andere top met dezelfde ouder).

Wij gebruiken inductie in de grootte s van het alfabet. Als er maar twee lettertekens zijn, is het resultaat uniek (er is maar één mogelijke optimale boom) – dat is dus triviaal.

Stel nu dat bewezen is dat het Huffman algoritme voor s lettertekens altijd een optimale boom bouwt en dat er nu $s + 1$ lettertekens zijn. Stel bovendien dat het algoritme als de boom T_{s+1} voor $s + 1$ lettertekens opgebouwd wordt eerst de laatste 2 letters in een boompje $T_{x_s,x_{s+1}}$ samenvoegt (anders moeten de letters gewoon hernoemd worden) en er de frequentie $a(T_{x_s,x_{s+1}}) = a(x_s) + a(x_{s+1})$ aan toekent. Deze twee hebben dus de kleinste frequenties (maar ze zijn misschien niet de enigen met deze eigenschap).

Dan bouwt het algoritme een optimale boom T_s voor de verzameling van bomen $x_1, \ldots, x_{s-1}, T_{x_s,x_{s+1}}$ als wij $T_{x_s,x_{s+1}}$ als één letterteken beschouwen. Stel nu dat de boom T_{s+1} die voor $x_1, \ldots, x_{s+1}$ gebouwd wordt – dat is dezelfde boom maar met $T_{x_s,x_{s+1}}$ als boom van diepte 1

in plaats van een enkele top – niet optimaal is maar dat er een betere optimale boom T_{opt} bestaat.

Opmerking: Wij mogen stellen dat x_s, x_{s+1} broers zijn in T_{opt} en op het laagste niveau zitten:

Neem twee broers y,y' op het laagste niveau. Stel dat $y \notin \{x_s, x_{s+1}\}$ en $x \in \{x_s, x_{s+1}\}$, $x \notin \{y, y'\}$. Dan kunnen wij x met y wisselen en krijgen en ten minste even goede boom: Wij hebben $l(y) \geq l(x)$ (omdat y op het laagste niveau zit) en $a(y) \geq a(x)$ (omdat x één van de twee toppen met laagste frequentie is).

Voor de nieuwe boom T'_{opt} na het wisselen zou dus gelden:

$Q(T'_{opt}) = Q(T_{opt}) - ((a(x)l(x)) + a(y)l(y)) + ((a(x)l(y)) + a(y)l(x)) = Q(T_{opt}) - (a(y) - a(x))(l(y) - l(x)) \leq Q(T_{opt})$

De nieuwe boom zou dus ten minste even goed zijn. Wij kunnen dus x_s, x_{s+1} met andere toppen wisselen zodat aan de voorwaarden uit de opmerking voldaan is.

Als wij in deze boom nu x_s, x_{s+1} en de ouder door één top z met frequentie $a(x_s) + a(x_{s+1})$ vervangen dan geldt voor de zo ontstane boom $T_{opt,s}$:

$Q(T_{opt,s}) = Q(T_{opt}) - (a(x_s) + a(x_{s+1})) < Q(T_{s+1}) - (a(x_s) + a(x_{s+1})) = Q(T_s)$

Maar dan zou T_s niet optimaal zijn wat een tegenstrijdigheid met de inductiehypothese is.

■

Je kan een tekst dus coderen door eerst de boom op te slaan die de code beschrijft en daardoor vastlegt hoe je de tekst moet decoderen, en achteraf de code volgens deze boom. Over de manier waarop je de boom het best opslaat, zullen wij het hier niet hebben – compressie is alleen maar nuttig voor grote bestanden en dan is het kleine en voor een gegeven alfabet constante deel voor de boom zonder veel belang.
Wat wij Huffman codering genoemd hebben kan je ook *statische Huffman codering* noemen. Wij gaan ervan uit dat je de frequenties van elk letterteken in het bestand kent. Aan de ene kant betekent dat natuurlijk dat je het bestand twee keer moet lezen – één keer om de frequenties te bepalen en één keer om het te comprimeren. Dat is natuurlijk niet ideaal. Maar soms is het zelfs onmogelijk: als je geen bestand maar een stream van data wil comprimeren dan moet dat *online* gebeuren – je **kan** de stream niet helemaal lezen

en dan nog een keer opvragen. Daarvoor kan je adaptieve Huffman codering gebruiken. Die zullen wij in het volgende hoofdstuk zien.

Oefening 74 *Construeer de Huffman code van de volgende tekst:*

`zaken_nemen_geen_keer`

Oefening 75 *Stel een manier voor om statische Huffman codering voor streams te gebruiken. Wat zijn de voordelen en nadelen in vergelijking met statische Huffman codering als je die op de hele verzameling van data kan toepassen?*

Oefening 76 *Een geautomatiseerd systeem om goederen in een opslagplaats op te slaan kan ook onderafdelingen van het magazijn samenvoegen – maar nooit meer dan 2 tegelijk.*
De tijd die het nodig heeft om deze onderafdelingen samen te vatten is $c * (d_1 + d_2)$ als d_1 het aantal goederen in afdeling 1 is en d_2 het aantal goederen in afdeling 2.
Een groot bedrijf wil nu al zijn n onderafdelingen tot één grote afdeling samenvatten. De hoeveelheden $d_1, \ldots, d_n$ van goederen in de afdelingen zijn gekend.
Geef een $O(n \log n)$ algoritme dat een volgorde bepaalt waarop de onderafdelingen zo snel mogelijk samengelegd kunnen worden.

Oefening 77 *Gegeven een vast compressiealgoritme A. De taak is een bestand van n MByte te vinden dat door het algoritme* **niet** *gecomprimeerd kan worden.*
Geef een algoritme dat zo'n bestand construeert. Welk soort algoritme zou je voorstellen? Stel dat het algoritme A in tijd $O(n)$ werkt. Hoeveel tijd zou je verwachten dat jouw algoritme vraagt – kan het even snel?

Oefening 78 *Bediscussieer de manier waarop je tijdens de Huffman codering altijd de twee boompjes met de minimale frequentie vindt. Welke datastructuren zou je kunnen toepassen?*

Stel dat je op de volgende manier werkt:
Je houdt twee lijsten bij: lijst (a) bevat in het begin de tekens samen met hun frequenties – dat zijn dus ook kleine boompjes. Je moet de lijst in stijgende volgorde van de frequentie sorteren. De lijst (b) gaat de gebouwde boompjes bevatten en is in het begin leeg.
Als je nu moet beslissen welke twee boompjes je moet samenvoegen om de volgende boom te bouwen, kies je 2 keer het kleinste van de elementen die

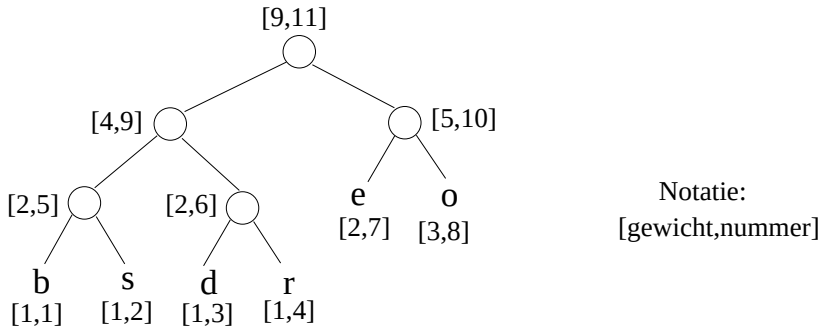
in de lijsten vooraan staan en verwijder je het uit zijn lijst. Dan vorm je daarvan de nieuwe boom en schrijf je hem op het einde van lijst (b).
Toon aan dat dit algoritme correct werkt – dus: dat je op deze manier altijd de 2 bomen met de kleinste frequenties samenvoegt.
Hoeveel tijd vraagt dit algoritme voor n tekens met gegeven frequenties ?

Wij hebben *in zekere zin* bewezen dat *Huffman codering optimaal is*. Maar wat betekent dat precies? Is er geen betere prefixcode om iets op te slaan? Daarvoor moeten wij precies kijken wat de voorwaarden van de stelling waren: één van de voorwaarden was dat de code **vast** is, dat dus dezelfde code voor de hele te coderen tekst gebruikt wordt.

4.1.1 Adaptive (Dynamic) Huffman Coding

Een nadeel van Huffman coding is dat je de tekst twee keer moet lezen – eerst om de boom te berekenen en achteraf nog eens om de tekst te coderen. Het wordt soms ook two-pass Huffman algoritme genoemd. Het streamen van data kan op deze manier duidelijk niet. Het feit dat je de Huffmanboom moet doorsturen wordt soms ook als nadeel genoemd, maar voor grote hoeveelheden data is het aantal bytes dat voor de Huffman boom nodig is natuurlijk verwaarloosbaar.
Adaptive Huffman Coding overloopt de tekst maar één keer en kan dus ook voor streamen gebruikt worden. Als de tekst $t_1, \ldots, t_n$ is dan wordt voor $i > 1$ het teken t_i gecodeerd door middel van een optimale Huffman boom voor de tekst $nng, t_1, \ldots, t_{i-1}$, waarbij `nng` een speciaal teken is dat de mogelijkheid biedt nieuwe tekens toe te voegen.. Dan wordt het gewicht van teken t_i in de boom met 1 verhoogd en de boom als nodig gewijzigd tot een optimale boom voor de tekens $nng, t_1, \ldots, t_i$. De boom is dan klaar om teken t_{i+1} te coderen of te decoderen.
Als de structuur van de te coderen tekst sterk verandert – bv. als een tarbestand met veel verschillende types bestanden gecomprimeerd moet worden – kan deze code dus in in het begin zelfs een efficiëntere codering gebruiken dan de statische Huffman coding die altijd ook rekening houdt met tekens die later vaak voorkomen.
De basis van adaptive Huffman coding werd door Faller (1973) en Gallager (1978) gelegd. Wij zullen het algoritme van Knuth (1985) zien, dat sommige verbeteringen bevat en FGK-algoritme (Faller-Gallager-Knuth) genoemd wordt. In zijn artikel bespreekt Knuth ook variaties, zoals een *sliding window* waarbij niet alleen het gewicht van het nieuwe teken verhoogd wordt, maar ook het gewicht van oude tekens verlaagd wordt, zodat de boom altijd optimaal is voor de laatste s tekens, waarbij s een op voorhand vastgelegd

getal is. Vitter heeft later (1987) een nog beter algoritme voorgesteld dat zeker de moeite is om te bestuderen. Het FGK-algoritme is intuïtief en een goede basis om dan ook naar de verschillen met het Vitter-algoritme te kijken – maar in deze les zullen wij dat niet doen.
Als wij de boom wijzigen, moeten wij natuurlijk zeker zijn dat de nieuwe boom optimaal is. Daarvoor gebruiken wij iets dat wij al bewezen hebben (Stelling 18): een boom die het resultaat van het statische Huffman algoritme kan zijn, is optimaal.
In de volgende afbeelding zien we nog eens ons voorbeeld voor *boosdoeer* (hier gebruiken wij op dit moment nog niet het teken `nng`). Bij elke top staat er $[g, o]$. Daarbij geeft g het gewicht van de top en het tweede nummer is o als de top de wortel van de o-de boom is die door het algoritme gekozen wordt. Wij noemen o het ordenummer van de top. Voor een top t schrijven wij $a(t)$ en $o(t)$ voor het gewicht (het aantal keren dat die voorkomt), resp. het ordenummer van t. Om de boom te kunnen aanpassen, heeft adaptive Huffman coding ook de gewichten van elke top nodig – die zullen wij dus tijdens de uitvoering van het algoritme voor elke top bijhouden – net zoals de ordenummers.



Daarbij zijn drie eigenschappen van een Huffman boom duidelijk:

(a) het gewicht van een interne top is altijd de som van de gewichten van zijn twee kinderen (zo hebben wij het gewicht ten slotte vastgelegd).

(b) als voor twee toppen t, t' geldt dat $o(t) < o(t')$ dan geldt $a(t) \leq a(t')$ (wij kiezen de toe te voegen bomen met gewichten in stijgende volgorde).

(c) als er n bladeren zijn, dan zijn er $2n - 1$ toppen en voor $1 \leq j < n$ zijn de toppen met ordenummers $2j - 1$ en $2j$ broers – ze hebben dezelfde ouder (wij kiezen altijd twee toppen in elke stap en voegen die aan dezelfde ouder toe).

Het mooie is nu dat dat ook omgekeerd juist is: als een boom (met gewichten en ordenummers) deze eigenschappen heeft, dan kan die ook het resultaat van het Huffman algoritme zijn – en is dus optimaal voor de gewichten. Omdat het zo belangrijk is, zullen wij dat als Lemma opschrijven:

Lemma 19 *Stel dat een binaire boom T met $2n-1$ toppen en de volgende eigenschappen gegeven is: van de $2n-1$ toppen zijn n toppen $b_1, \ldots, b_n$ bladeren en elk blad b_i is gelabeld met een letterteken. Alle toppen t – dus alle bladeren en alle interne toppen – hebben paarsgewijs verschillende ordenummers $o(t)$ uit $1, \ldots, 2n-1$ en (niet negatieve) gewichten $a(t)$. Dan geldt:*
Als T aan (a), (b) en (c) voldoet en $l(b_i)$ de diepte van het blad b_i is, dan is T een boom zodat

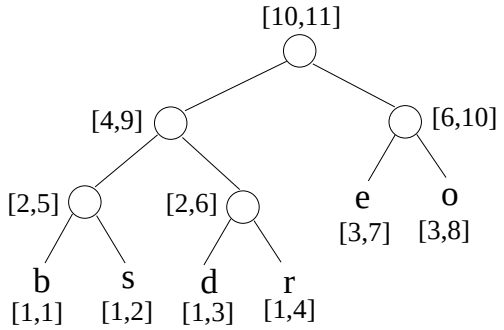
$$Q(T) = \sum_{i=1}^{n} l(b_i) * a(b_i)$$

minimaal is voor alle binaire bomen met bladeren $b_1, \ldots, b_n$.

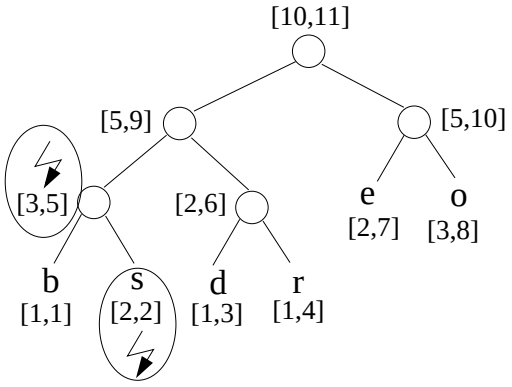
Het is dus in zekere zin *een optimale prefixboom – als je aanvaardt dat toppen met gewicht 0 ook in de boom moeten zitten.*

Oefening 79 *Geef een expliciet bewijs van Lemma 19. De beste manier om dit te doen is te bewijzen dat de boom met het Huffman-algoritme opgebouwd kan worden. Dit kan* "gemakkelijk" *met inductie (zie ook het bewijs van Stelling 18) – maar het moet wel nauwkeurig opgeschreven worden.*

Als wij een nieuw letterteken lezen dat al in de boom zit, wordt het gewicht van dat teken en de toppen op het pad naar de wortel met één groter. Als wij na `boosdoeer` dus bv. nog een `e` lezen, wordt de boom:



Deze boom voldoet aan (a)-(c) en hoeft dus niet gewijzigd te worden – maar dat was geluk! De reden was dat elke top die gewijzigd werd het hoogste ordenummer had van alle toppen met hetzelfde gewicht. Als wij na `boosdoeer` bv. nog een `s` zouden lezen, wordt de boom:

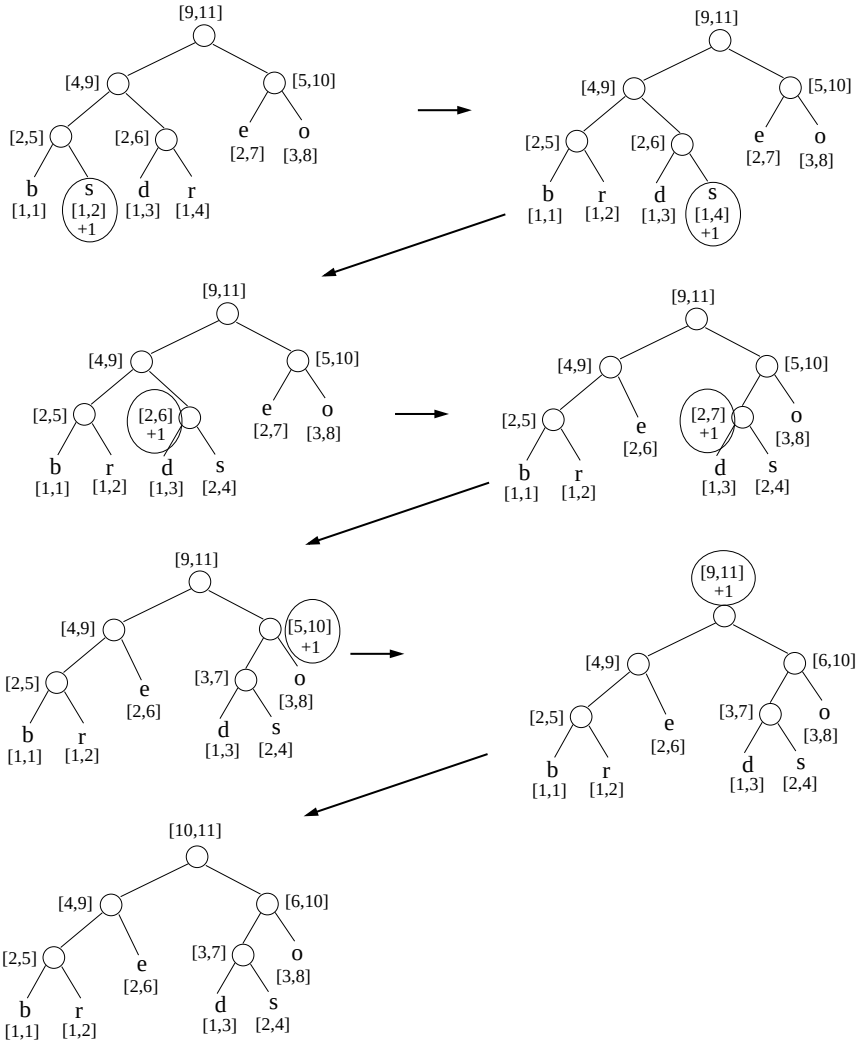


– en die boom voldoet niet aan (b)!
Het idee is nu de situatie waar het wel lukt af te dwingen: als je het gewicht van een top t om 1 groter wilt maken, wissel je die (en de hele boom die eraan hangt) eerst met de top die het grootste ordenummer heeft van alle toppen met gewicht $a(t)$. Je wisselt daarbij de positie in de boom en de ordenummers. Dan verhoog je het gewicht met 1 en ga je naar de nieuwe ouder. Zo ga je door totdat je de wortel hebt bereikt. Of als pseudocode:

Algoritme 10 *Adaptive Huffman coding voor strict positieve gewichten*

```
adjust_Huffman_tree_pos(z)
// z is het volgende teken dat gelezen werd en wij veronderstellen
// hier dat het al in de boom zit

t = de top met label z

while (t is niet de wortel)
{
  t'= de top x met a(x)=a(t) en o(x) maximaal;
  wissel t en t'in de boom
  wissel de ordenummers van t en t'
  verhoog a(t) met 1
  t = (nieuwe) ouder van t
}

// nu is t de wortel
verhoog a(t) met 1

}
```

Hierbij hebben we het alleen over het wijzigen van de boom gehad. Als je een tekst codeert moet je natuurlijk ook altijd het teken uitschrijven – gecodeerd door de boom voor de wijziging op dezelfde manier als bij statisch Huffman coderen.
Het helpt zeker een voorbeeld te zien. Wij passen het algoritme toe op het lezen van een `s` na `boosdoeer`, waarvan wij al gezien hebben, dat het een probleem vormde als de boom niet gewijzigd wordt:



Wij zouden graag zeggen dat de kost voor het wijzigen van de boom net zoals bij het wijzigen van zoekbomen lineair is in de lengte van het pad na de top die gewijzigd wordt – maar doordat de toppen altijd op andere plaatsen in de boom terechtkomen, is dat – ten minste niet onmiddelijk – duidelijk. Het is zelfs zo, dat een top bij het vervangen op een grotere diepte geplaatst kan worden.

Oefening 80 *Geef een voorbeeld van een met ordenummers en gewichten gelabelde boom die aan (a),(b),(c) voldoet en een blad bevat dat als het gewicht met 1 verhoogd wordt door Algoritme 10 in de eerste iteratie op een grotere diepte geplaatst wordt.*

Lemma 20 *Als Algoritme 10 een blad t in de Huffman-boom op diepte d met* 1 *verhoogt, dan wordt de wortel na ten hoogste d iteraties bereikt.*

Bewijs: Stel dat $o(t) = x_0, \ldots, x_k$ de ordenummers van de toppen op het pad van t naar de wortel zijn en $o(t') = x'_0, \ldots, x'_{k'}$ de ordenummers van de wisselende toppen t' die in Algoritme 10 gebruikt worden. Wij willen aantonen dat $k' \leq k$. Wij hebben $x_k = x'_{k'}$ omdat wij in beide gevallen bij de wortel stoppen. Als wij dus voor alle i met $i \leq \min\{k, k'\}$ bewijzen dat $x_i \leq x'_i$ is, dan is – omdat x_k het grootste ordenummer heeft – ten laatste x'_k de wortel, dus $k' \leq k$.

Wij willen dus bewijzen: (*) $x_i \leq x'_i$ voor alle $0 \leq i \leq k, k'$.

Wij bewijzen eerst dat als voor twee (verschillende) toppen u, u' geldt dat $o(u) < o(u')$, de ouders p_u en $p_{u'}$ voldoen aan $a(p_u) \leq a(p_{u'})$. De broer u_b van u heeft ordenummer $o(u) - 1$ of $o(u) + 1$. Omdat $o(u) - 1 < o(u) + 1 \leq o(u')$ geldt dus dat $a(u_b) \leq a(u')$ (b) en (c). Voor de ouder p_u geldt dus $a(p_u) \leq a(u) + a(u')$ (a). Analoog heeft de broer u'_b van u' ten minste gewicht $a(u)$, dus $a(p_{u'}) \geq a(u) + a(u')$.

Nu bewijzen we (*) (en dus het lemma) met inductie. In het begin hebben t en t' hetzelfde gewicht en t' wordt met het grootste ordenummer gekozen – voor $i = 0$ geldt (*) dus. Stel nu dat (*) voor $i - 1 < \min\{k, k'\}$ geldt. Dan is $x_{i-1} \leq x'_{i-1}$ en stel dat de bijhorende toppen $t_x, t_{x'}$ zijn. Dus geldt voor de ouders $a(p_{t_x}) \leq a(p_{t_{x'}})$. Dan wordt t' als de top met gewicht $a(p_{t_{x'}})$ en maximale ordenummer gekozen. Als $a(p_{t_x}) < a(p_{t_{x'}})$, dan geldt zeker dat $x_i = o(p_{t_x}) < o(p_{t_{x'}}) \leq o(t') = x'_i$. Door de keuze van t' als de top met het maximale ordenummer en gewicht $a(p_{t_{x'}})$ geldt ook als de gewichten van p_{t_x} en $p_{t_{x'}}$ gelijk zijn $x_i = o(p_{t_x}) \leq o(t') = x'_i$ – en dat wouden wij bewijzen.

■

Het aantal iteraties van Algoritme 10 is dus de diepte van het letterteken dat gelezen wordt. Omdat het letterteken voor het coderen toch al opgezocht moet worden, zijn dat even veel iteraties als voor het algoritme dat de uit te schrijven code berekent. Maar wat ons echt interesseert, is natuurlijk altijd de tijd. Het is mogelijk het algoritme zo te implementeren dat de tijd voor het wijzigen inderdaad ook $O(d)$ is als het net gelezen teken op diepte d staat. Op

een constante na is het wijzigen van de boom dus even duur als het opzoeken van het letterteken! Daarvoor is wel een slimme boekhouding nodig, zodat je altijd snel t' kan vinden – en de boekhouding ook snel kan updaten. Deze details van het algoritme zijn belangrijk en kunnen best uitgewerkt worden door het algoritme echt te implementeren. In zijn artikel *Dynamic Huffman coding* (Journal of Algorithms 6, 1985, pp.163–180) geeft Knuth alle details die nodig zijn om het te kunnen implementeren – als het je alleen niet lukt.

Op dit moment zijn we er altijd vanuitgegaan dat alle tekens die we tegenkomen vanaf het begin in de boom zitten. Je kan natuurlijk het hele alfabet in het begin in een op voorhand vastgelegde boom stoppen – bv. in een boom met minimale diepte. Omdat de gewichten allemaal 0 zijn, is elke boom optimaal. Inderdaad stelt ook Knuth voor met een dergelijke optimaal gebalanceerde boom te werken – maar niet als deel van de Huffmanboom die gewijzigd wordt, maar als extra boom. Hij stelt voor één extra teken `nng` (nog niet gecodeerd) met gewicht 0 bij te houden dat betekent *"een nieuw teken wordt toegevoegd"*. Als het alfabet alle tekens van k bytes zijn, komt de manier met de optimaal gebalanceerde boom erop neer dat de $8k$ op een gecodeerd `nng`-teken volgende bits het nieuwe teken direct geven. Hij stelt ook voor de boom als er een teken *opgehaald* wordt te wijzigen, maar omdat elk teken in de boom ten hoogste één keer opgehaald wordt, gaat het erbij maar om heel weinige bits die bespaard kunnen worden – wij zullen de details hier dus niet geven. Voor de code die wij voor een nog niet gezien teken uitschrijven gebruiken wij dus de Huffman code van het teken `nng` gevolgd door het teken voorgesteld als een op voorhand vastgelegd aantal bits (bv. één byte).
Wij hebben dan wel nog een nieuwe operatie nodig: de boom uitbreiden met een nieuw teken. Dat doen we door de `nng`-top te vervangen door een boom met 3 toppen: één ouder o met gewicht 1 die de `nng`-top (gewicht 0) en de nieuwe top (gewicht 1) als kinderen heeft. Dan wordt de top o net zo behandeld alsof die er al zat en van gewicht 0 naar gewicht 1 zou veranderd zijn. Behalve in het begin als o al de wortel is, begin je de gewone recursie dus met de ouder van o. De `nng`-top houdt ordenummer 1, de nieuwe top krijgt ordenummer 2 en o ordenummer 3. Voor alle andere toppen moeten de ordenummers met 2 verhoogd worden. Dat kan efficiënt door de ordenummers als $o(t) + b$ op te slaan, waarbij $o(t)$ bij de top hoort en b een *verschuiving* voorstelt die voor alle toppen dezelfde is (en die dan om 2 verhoogd wordt) ofwel door gewoon af te stappen van ordenummers die bij 1 beginnen. In feite zijn de ordenummers vooral belangrijk om het algoritme goed te verstaan – voor de implementatie moet je zelfs niet altijd alle ordenummers van alle toppen bijhouden...

Er zit wel een addertje onder het gras: omdat er een top met gewicht 0 is, zal zijn broer – en alleen maar die – een ouder hebben met hetzelfde gewicht. Als die ouder nu de top met dit gewicht en het grootste ordenummer is, zou het mislopen: je kan de top niet vervangen door een boom waar die zelf inzit. In dit speciaal geval kan je wel gewoon het gewicht van de top met 1 verhogen en dan met de ouder doorgaan zonder dat op het einde (a),(b) of (c) geschonden zijn. De licht gewijzigde pseudocode voor tekens die al in de boom zitten, is dus:

Algoritme 11 *Adaptive Huffman coding voor ten hoogste één gewicht 0*

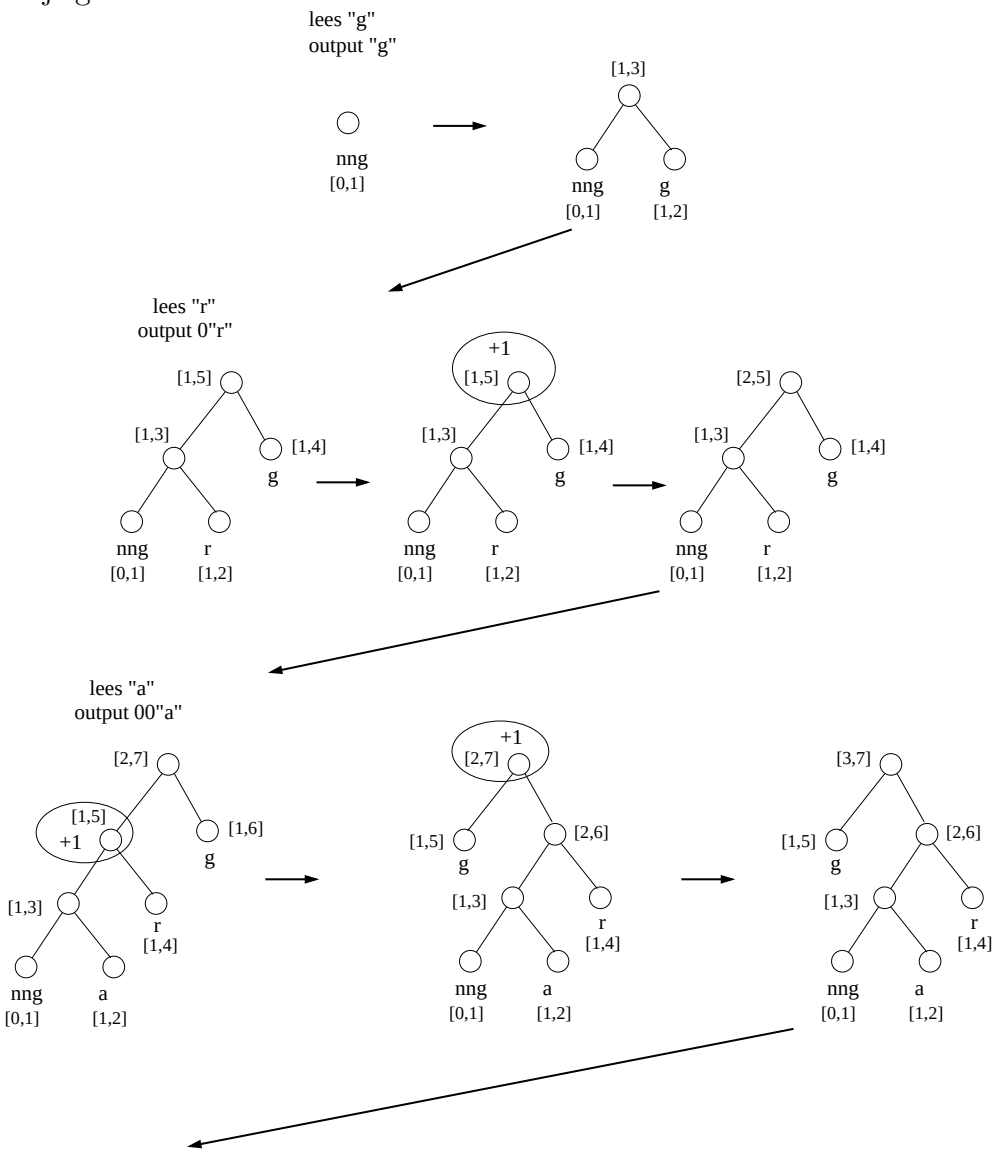
```
adjust_Huffman_tree(z)
// z is het volgende teken dat gelezen werd

als z in de boom zit
    t = de top met label z
anders
    maak de kleine boom met nng, teken z en ouder o
    als o geen ouder heeft: return
    anders t = ouder van o

while (t is niet de wortel)
{
  t'= de top x met a(x)=a(t) en o(x) maximaal;
  als t' niet de ouder is van t
   { wissel t en t'in de boom
     wissel de ordenummers van t en t'
   }
  verhoog a(t) met 1
  t = (nieuwe) ouder van t
}

// nu is t de wortel
verhoog a(t) met 1

}
```

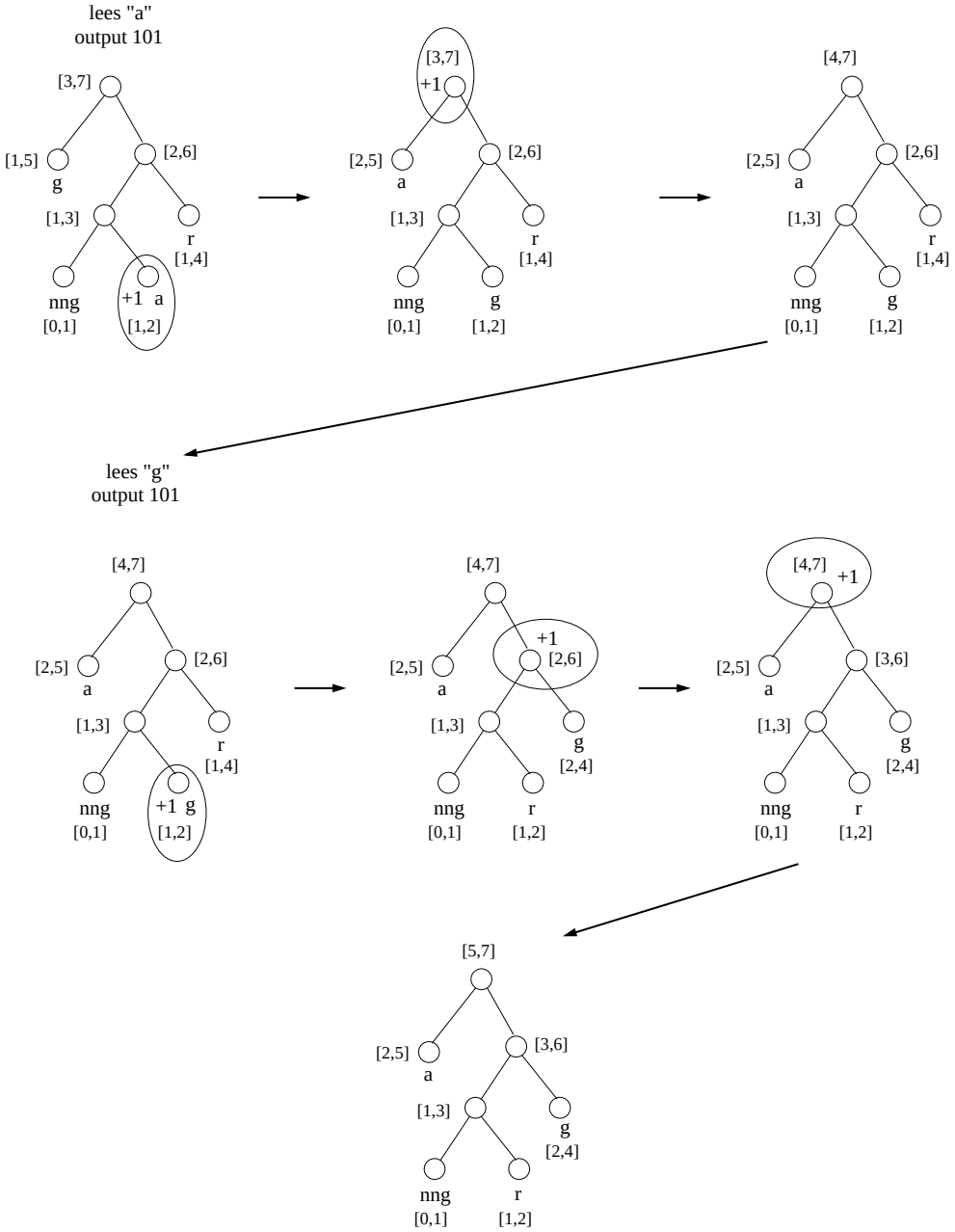
Oefening 81 • *Toon aan dat er in een Huffman boom met maximaal één top met gewicht* 0 *ten hoogste één top* t *zit, zodat de top* t' *met*

hetzelfde gewicht en het grootste ordenummer de wortel is van een boom die t bevat.

- *Toon aan dat de wijziging in de pseudocode ervoor zorgt, dat de boom op het einde aan (a),(b) en (c) voldoet. Je mag gebruiken dat het algoritme voor strikt positieve gewichten juist werkt.*

Maar opnieuw is het het beste een voorbeeld te zien. Wij zullen eens het woord `graag` coderen. Het wordt afgesloten door een `EOF` dat natuurlijk ook doorgestuurd wordt, maar de boom moet daarvoor natuurlijk niet meer gewijzigd worden.





Als wij nu nog een `r` zouden lezen, dan zouden wij het speciale geval tegenkomen, waar wij zonder de wijziging in het algoritme de top met een deelboom zouden moeten wisselen waar die zelf inzit.
Het decoderen gaat nu eenvoudig op dezelfde manier: het eerste teken en de tekens na het `nng`-teken worden direct gelezen (bv. door het aantal op voorhand vastgelegde bits te lezen of zoals door Knuth beschreven) en anders lees je de Huffman code en wijzig je achteraf de boom.

Oefening 82 • *Codeer de tekst* `voordeel`.

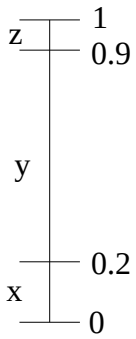
• *Decodeer de door jou opgestelde code om te toetsen of die juist was.*

Oefening 83 *Decodeer de adaptive Huffman code*
"n"0"o"0100"d"001100"a"100111
Tekens tussen aanhalingstekens staan – net zoals in de tekst – voor de 8 bits die dit letterteken voorstellen. Het resultaat is wel niet echt een woord uit de woordenlijst van de Nederlandse taalunie...

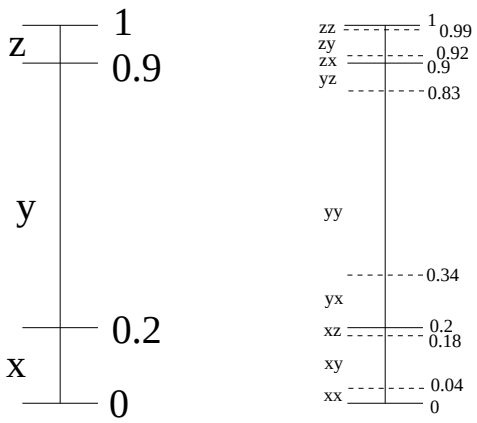
4.2 Arithmetisch coderen

Net zoals Huffman coding buit ook arithmetisch coderen (arithmetic coding) uit dat de lettertekens die gecodeerd worden verschillend vaak voorkomen – of anders gezegd: een verschillend grote kans hebben om voor te komen. Het is niet alleen zo dat arithmetisch coderen de basis is van de algoritmen met de hoogste compressiefactor, het idee van arithmetisch coderen is ook bijzonder interessant, dus moeten wij dat zeker zien. Wij gaan uit van wat in de teksten over arithmetisch coderen altijd een model genoemd wordt. Dit model is niets anders dan een verzameling van tekens (bv. alle bytes) en een functie $p(t)$ die beschrijft hoe groot de kans is dat teken t gebruikt wordt (bv. de relatieve hoeveelheid van dit teken in de Nederlandse taal). Als je het model voor een vaste tekst wilt optimaliseren en het met Huffman coding vergelijkt, is $p(t)$ dus $a(t)/n$, waarbij $a(t)$ het gewicht van teken t is en n de lengte van de tekst. Voor het coderen en decoderen moet (natuurlijk) hetzelfde model gebruikt worden – het moet dus ofwel meegestuurd worden of (gebaseerd op het type tekst) bij het coderen en decoderen gekend zijn. Als je bv. een Nederlandstalige tekst wilt coderen, hoef je niet noodzakelijk de probabiliteiten in deze tekst te kennen, maar zal het model dat de gekende probabiliteiten van lettertekens voor de Nederlandse taal beschrijft een goede benadering zijn. In dit hoofdstuk gaan we uit van een vast model.
Voor de voorbeelden zullen wij altijd met hetzelfde, **heel** eenvoudige, model werken: wij hebben maar drie lettertekens: x, y en z. De probabiliteiten zijn $p(x) = 0.2$, $p(y) = 0.7$ en $p(z) = 0.1$.
Wij gebruiken de notatie $[a, b)$ voor een interval dat bij a gesloten is (a bevat) en bij b open (b niet bevat). Het idee is nu aan elke string s een interval $i(s) = [a, b)$ met $0 \leq a < b \leq 1$ toe te kennen en dat zo te doen dat voor alle vaste n de verzameling van alle intervallen die aan strings s van lengte n toegekend zijn een partitie van het interval $[0, 1)$ vormen – dus:
$\bigcup_{\{s|\ |s|=n\}} i(s) = [0, 1)$ en
$\forall s, s'$ met $|s| = |s'| = n$ en $s \neq s'$ geldt $i(s) \cap i(s') = \emptyset$.

Omdat de intervallen voor een vaste lengte n een partitie vormen, zit dus elk reëel getal r met $0 \leq r < 1$ voor elke n in precies één interval $i_n(r)$ die bij een string $s_n(r)$ van lengte n hoort.
Als er k verschillende lettertekens zijn en de string heeft lengte n, dan zijn er k^n intervalle in de partitie – dat zijn dus heel snel veel te veel om te construeren. Gelukkig is dat geen probleem, omdat ze nooit allemaal geconstrueerd hoeven te worden. Wij moeten altijd alleen maar het interval voor één string construeren – de string die wij willen comprimeren.
Laten we eerst een triviaal voorbeeld zien: de partitie voor strings van lengte 1. Elk teken krijgt een interval dat even lang is als zijn probabiliteit. In ons model definiëren wij voor het eerste teken x (met $p(x) = 0,2$) $i(x) = [0, 0.2)$. Aan het tweede teken y (met $p(y) = 0.7$) kennen wij het volgende interval $i(y) = [0.2, 0.9)$ toe en sloteindelijk aan het derde teken z (met $p(z) = 0.1$) het daarop volgende interval $i(z) = [0.9, 1)$. Omdat alle tekens samen natuurlijk kans 1 hebben, zal op die manier een partitie van het interval $[0, 1)$ gedefinieerd zijn. Als afbeelding ziet dat er zo uit (waarbij uit de afbeelding niet duidelijk wordt, waar de intervallen open en gesloten zijn, maar dat weten wij natuurlijk):



Als wij de partitie voor k tekens al gedefinieerd hebben, dan krijgen wij de partitie voor $k + 1$ tekens door elk interval van de partitie met k tekens net zo onder te verdelen als wij het in het begin met het interval $[0, 1)$ gedaan hebben. Voor 2 tekens ziet ons voorbeeld er dan zo uit:



Nu kunnen wij een string voorstellen door eender welk getal in het interval dat bij de string behoort. Wij stellen getallen natuurlijk binair voor. Omdat wij alleen getallen moeten coderen die in $[0, 1)$ liggen, leggen wij vast dat de bitstring $b_1, \ldots, b_k$ het getal $\sum_{i=1}^{k}(b_i * 2^{-i})$ voorstelt. Het getal 0.5 (binair 1) zit bv. in $i(yy)$ en dus is $s_2(0.5) = yy$. Voor het getal 0.8125 (binair 1101) geldt ook dat $s_2(0.8125) = yy$. Voor 0.9 krijgen wij $s_2(0.9) = zx$, etc. De lengte van een interval is altijd het product van de probabiliteiten van de tekens die erin zitten. Hoe langer een interval is, hoe zekerder je kan zijn dat het een getal bevat dat met relatief weinig bits voorgesteld kan worden. Het kan ook dat een heel kort interval een getal bevat dat toevallig een korte voorstelling heeft in de gekozen (bv. binaire) representatie – bv. het interval $[0.5, 0.5000000000000000000001)$, maar dat is dan toeval. Wij kunnen uit de lengte van een interval wel een bovengrens voor de lengte van onze representatie afleiden:

Lemma 21 *Als een interval $[a, b)$ met $0 \leq a < b \leq 1$ lengte $l < 1$ heeft, dan bevat het een getal waarvan de voorstelling ten hoogste lengte $\lceil \log \frac{1}{l} \rceil$ heeft.*

Bewijs: Als $a = 0$ voldoet de bitstring 0 – stel dus dat $a > 0$. Stel dat $n = \lceil \log \frac{1}{l} \rceil$ en dat $g < a$ maximaal is met de eigenschap dat $g \notin [a, b)$ en dat g met ten hoogste n bits voorgesteld kan worden. Het getal $g + 2^{-n}$ is ofwel 1 of kan ook met maximaal n bits voorgesteld worden. Als het interval geen getal bevat dat met n bits voorgesteld kan worden, geldt $b \leq g + 2^{-n}$ (voor $g + 2^{-n} = 1$ geldt dat ook) – dus $l = b - a < g + 2^{-n} - g = 2^{-n}$, dus $\log l < -n$ en sloteindelijk $\log \frac{1}{l} = -\log l > n$, wat de keuze van n tegenspreekt.

■

Alle compressiemethoden hebben hun voor- en nadelen – dus gevallen waar ze bijzonder geschikt (of niet zo goed) geschikt zijn. Wij moeten ook – soms

meer en soms minder – rekening houden met de tijd die nodig is om iets te comprimeren. Wij zullen dus zeker niet zeggen, dat arithmetisch coderen in elke opzicht beter is dan Huffman coderen – maar wij kunnen hier al zien, dat het soms verrassend korte codes toelaat. Inderdaad kan bewezen worden dat als lettertekens met de gegeven probabiliteiten onafhankelijk verdeeld zijn (het volgende letterteken dus niet afhangt van het vorige), de lengte van de code (in zekere zin) zelfs optimaal is! Dit is sterker dan Stelling 18, waar wij bewijzen dat het een optimale **prefixcode** is, waar dus duidelijke voorwaarden zijn aan het soort code. Om de optimaliteit te bewijzen, zouden wij de entropie moeten introduceren – maar omdat de klemtoon van AD3 meer toepassings- en ideeën-gericht is, zullen wij dat hier niet doen. Een gespecialiseerde les over compressie waar al deze dingen aan bod kunnen komen, zou zeker zeer interessant zijn...
Huffman codes zouden voor elke string van lengte n natuurlijk ten minste n bits nodig hebben – de string `yyyxzxxzyyyxyyyyxyyyzxyxyyyyxxyyyyyyxyyy` dus ten minste 40 bits (in feite zelfs meer). Als je bv. in ons voorbeeld het interval voor deze string wilt berekenen, dan is dat **ongeveer** $[0.49999999999999639178, 0.50000000000000310862)$.
In dit interval ligt $0.5 = 2^{-1}$ – de string kan dus met 1 bit voorgesteld worden: $s_{40}(0.5) = yyyxzxxzyyyxyyyyxyyyzxyxyyyyxxyyyyyyxyyy$. Inderdaad codeert 0.5 niet alleen de strings $s_1(0.5) = y$, $s_2(0.5) = yy$, ..., $s_{40}(0.5) = yyyxzxxzyyyxyyyyxyyyzxyxyyyyxxyyyyyyxyyy$, maar is $s_n(r)$ voor alle $0 \leq r < 1$ en alle $n \in \mathbb{N}$ gedefinieerd. Door onze manier om de partitie te definiëren geldt ook dat als $k < k'$ de string $s_k(r)$ de prefix van lengte k van $s_{k'}(r)$ is.
Wij zullen nu nog gedetailleerd beschrijven hoe het algoritme werkt en hoe wij het interval voor een string *in principe* berekenen. Dus: gegeven een verzameling $A = \{t_1, \ldots, t_k\}$ van tekens en een probabiliteitsfunctie $p : A \to \mathbb{R}$, zodat de som van alle probabiliteiten 1 is. Wij definiëren dan voor $1 \leq i \leq k$ de functie $f : A \to \mathbb{R}$ met $f(t_i) = 0 + \sum_{j<i} p(t_j)$ (dus het *startpunt* van het interval $i(t_i)$).
Stel nu dat een string $s[1], \ldots, s[n]$ met tekens uit A gegeven is. Dan berekenen wij het interval $i(s[])$ als volgt:

Algoritme 12 *Berekening van het interval voor arithmetisch coderen*

```
get_interval(s[],n)
{
links=0.0;
lengte=1.0;
```

```
for (i=1; i<=n; i++)
{ links = links+(f(s[i])*lengte);
  lengte= lengte * p(s[i]);
}

return: [links, links+lengte);
}
```

Het is dus – **in principe** – heel eenvoudig om te programmeren. Als je het interval hebt, is ook een getal om het voor te stellen eenvoudig te vinden – het is gewoon logaritmisch zoeken, maar je kan het ook zo zien, dat wij het interval van beneden benaderen – totdat wij er voor de eerste keer inzitten. Wij letten alleen op, dat wij nooit een te groot getal kiezen:

Algoritme 13 *Berekening van het getal (de bitstring) dat (die) een interval voorstelt*

```
get_bitstring(l,u)
// output een bitstring die het interval [l,u) voorstelt
{
getal=0.0;
next=0.5;
found=FALSE;

while (found=FALSE)
  {
  if ((getal+next)< b)
     { output 1;
       getal=getal+next;
       if (getal>=a) found=TRUE;
      }
  else output 0;
  next=next/2;
  }

}
```

Inderdaad berekent dit algoritme niet alleen **een** bitstring die een getal in het interval voorstelt, maar een kortst mogelijke bitstring met deze eigenschap!

Oefening 84 *Bewijs dat Algoritme 13 een kortst mogelijke voorstelling van een getal in het interval berekent.*

Oefening 85 *Gegeven het alfabet* $A = \{a, c, g, t\}$ *met tekens in deze volgorde en probabiliteiten* $p(a) = 0.3$*,* $p(c) = 0.2$*,* $p(g) = 0.4$ *en* $p(t) = 0.1$*. Bereken de kortste bitstring die de string* `cag` *voorstelt.*

Oefening 86 *Gegeven: een vast model* A*,* $p()$*, een bitstring* $b[]$ *en een getal* $n \in \mathbb{N}$ *dat de lengte van de string geeft. Geef de pseudocode van een algoritme dat de gecodeerde string berekent. Ga in jouw pseudocode ervanuit dat reële getallen met oneindige precisie berekend worden. Misschien helpt het daarbij het interval soms te herschalen...*

Oefening 87 *Gegeven het alfabet* $A = \{a, c, g, t\}$ *met tekens in deze volgorde en probabiliteiten* $p(a) = 0.3$*,* $p(c) = 0.2$*,* $p(g) = 0.4$ *en* $p(t) = 0.1$*.*
Geef de string van lengte 3 *die door de bitstring* `0010011` *gecodeerd wordt.*

Coderen en decoderen is *in principe* heel eenvoudig, maar als je het algoritme efficiënt en nauwkeurig wilt implementeren, is er een heel nest addertjes onder het gras:

(a) Omdat elk getal in feite een oneindige string codeert, moeten wij behalve de bitstring ook het getal n doorsturen dat zegt hoe lang de prefix is die gecodeerd wordt. Een alternatief is een extra *einde*-teken mee te coderen. In dat geval zou het decoderen gestopt worden zodra de prefix met op het einde het einde-teken berekend is.

(b) Zelfs in ons klein voorbeeldje met weinig tekens en grote probabiliteiten worden de intervallen al snel zo klein, dat bij gewone datatypes al niet te verwaarlozen afrondingsfouten gebeuren. Als je bv. alle tekens met 8 bits wilt voorstellen, heb je al voor 7 tekens niet voldoende bits in het IEEE 754 double-precision binary floating-point formaat om een verschil tussen alle intervallen te kunnen maken – om het even wat de probabiliteiten zijn. Je moet dus met arbitrary precision datatypes werken die **extreem** lang worden (wat niet echt snel is...).

(c) Als je een string wilt coderen, zou je het liefst niet eerst het getal berekenen en pas achteraf de bitstring, maar beter parallel met het berekenen van de eerste in elkaar geschakelde intervallen al de eerste bits van de code uitvoeren. Als bv. het eerste interval $[0.6, 0.71)$ is, zou je onmiddelijk 1 als eerste bit kunnen uitvoren. Jammer genoeg kan dat niet altijd: als jouw interval altijd rond de 0.5 schommelt, kan het zelfs zijn

dat pas het laatste teken beslist of je een getal groter dan of gelijk aan 0.5 (eerste bit 1) of kleiner dan 0.5 (eerste bit 0) moet kiezen.

Daarbij is (a) natuurlijk niet echt een probleem. Als je n tekens wil coderen, zijn de $\log n$ tekens zeker even weinig een probleem als een extra teken waaraan je een heel kleine probabiliteit kan toekennen omdat het toch maar één keer opduikt. Voor (b) en (c) werden verschillende oplossingen ontwikkeld – maar helemaal opgelost kunnen die problemen niet worden.
Er zijn oplossingen die – op kost van een klein verlies aan efficiëntie – met een vaste precisie kunnen werken door continu de intervallen te herschalen. Daarbij worden de echte intervallen maar benaderd, maar als bij het coderen en decoderen dezelfde afrondingen gebeuren, werkt het algoritme nog altijd juist. Inderdaad kan je dan ook met integers werken – maar dat is natuurlijk niet verrassend als het met een vaste precisie werkt. De details van deze oplossingen zijn echt mooi en jullie zouden die best eens in een boek of online (er is een goed leesbare en gedetailleerde beschrijving van Amir Said, HP Laboratories) opzoeken. Hier is er niet de tijd voor – en als jullie later niet in een richting gaan waar jullie dat nodig hebben, zijn dat te veel details en is het voldoende het schitterende idee van arithmetisch coderen te kennen.

PPM – Prediction by Partial Matching

Bij PPM wordt *in principe* gewoon arithmetisch coderen toegepast, maar de probabiliteiten, die in de verschillende stappen gebruikt worden, zijn niet meer gebaseerd op één model, maar er worden afhankelijk van de context (dat is: de vorige tekens) bij het coderen **en** decoderen in verschillende stappen verschillende modellen toegepast. Als de laatste vier tekens in een Nederlandstalige tekst bv. `_thu` zijn, dan is de kans dat het volgende teken een `i` is zeer groot – veel groter dan zonder de context van de vorige lettertekens. Je hebt dus op het einde grotere intervallen en dus kortere voorstellingen van de intervallen als je in dit geval een speciaal model afhankelijk van de laatste vier tekens gebruikt.
Ook hier heb je een hele boel problemen waarover veel artikels gepubliceerd werden:

- Met hoeveel vroegere tekens houdt jouw model het best rekening?
- Hoe sla je de modellen het best op en hoe zoek je efficiënt in deze datastructuur?
- Hoe houdt je rekening met probabiliteiten die 0 zijn – arithmetisch coderen kan daarmee niet werken. Als je bv. met een model voor de

Nederlandse taal werkt, waar er misschien geen woord is dat met `thur` begint, kan het toch dat er een naam is die zo begint of gewoon het Engelse *thursday* gebruikt wordt – en dan moet compressie ook werken.

- Wat doe je als je geen voorkennis hebt over het soort data dat gecomprimeerd wordt en je dus niet weet welk model je moet toepassen?

Het hoofdidee, dat wij net geschetst hebben, werd in verschillende algoritmes geïmplementeerd: PPMC, PPM*, BOA, RK.... Deze algoritmen hebben twee dingen gemeen:

(a) Ze comprimeren **heel** goed. PPM-gebaseerde algoritmen hebben de beste compressiefactoren die er zijn en zijn vooral als het om teksten gaat – waar de modellen op voorhand berekend kunnen worden – en als snelheid minder belangrijk is de beste keuze!

(b) Ze zijn vrij traag, zodat voor de meeste toepassingen toch al die algoritmen gebruikt worden, die wij in de volgende hoofdstukken zullen zien, ook al is de compressiefactor iets slechter.

Wij hebben met Huffman coding en arithmetic coding manieren die rekening houden met de relatieve hoeveelheden (of met andere woorden kansen) van de tekens in de tekst, maar die algoritmen hebben nog altijd de voorwaarde dat het alfabet op voorhand vastligt. Stel bv. dat in de tekst het letterteken "o" altijd gevolgd wordt door "e". Dan zou het toch verstandig zijn niet "o" en "e" voor de code te gebruiken maar "oe"! En zelfs in sommige gevallen waar er "o", "e" en "oe" in de code zitten, zou het kunnen, dat het nuttig is alle drie te coderen. PPM gaat al in die richting door de kansen afhankelijk van de laatste tekens te kiezen, maar toch zijn het kansen van één teken uit een vast alfabet.
Je kan dus het alfabet uitbreiden door kleine deelstrings als lettertekens te beschouwen. In het extreme geval zou je natuurlijk de hele tekst als één letterteken kunnen beschouwen en de code is alleen maar – bv. – "1". Maar hoe kan je dan aan de decoder mededelen waarvor de 1 staat? De ruimte die daarvoor nodig is, is – anders dan bij Huffman coding, waar ook extra informatie gestuurd wordt – zeker niet verwaarloosbaar...
Basisideeën om algoritmen met een betere compressie dan Huffman codering te ontwikkelen die toch al heel snel zijn (vooral: sneller dan PPM), zijn

- Gebruik een groter alfabet dan alleen maar lettertekens – gebruik ook strings!

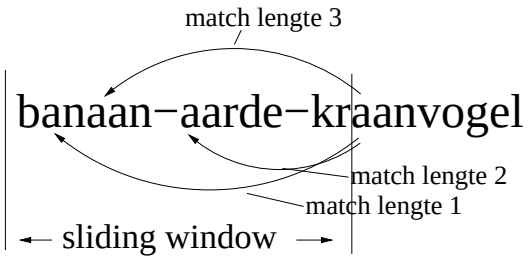
- Gebruik een woordenboek dat dynamisch kan veranderen.

Oefening 88 *Stel dat je niet één groot bestand moet sorteren, maar n al bestaande gesorteerde bestanden $b_1, \ldots, b_n$ tot één groot gesorteerd bestand moet samenvoegen. Voor $1 \leq i \leq n$ bevat bestand b_i l_i sleutels en je kan de bestanden niet tegelijk mergen maar je kan altijd alleen maar 2 bestanden mergen tot een nieuw bestand. Als de oude bestanden l_i en l_j sleutels bevatten, bevat het nieuwe bestand dus l_i+l_j sleutels en de kost van deze mergebewerking is $l_i + l_j$.*

- *Geef een voorbeeld dat toont dat de kost om alle bestanden te mergen afhankelijk is van de volgorde waarop je de bestanden mergt.*
- *Beschrijf een efficiënt algoritme om de optimale (goedkoopste) volgorde te bepalen.*
- *Wat is de complexiteit van dit algoritme?*
- *Toon aan dat jouw algoritme inderdaad de optimale volgorde vindt.*

4.3 LZ77

(Abraham Lempel en Jacob Ziv 1977)
Het LZ77 algoritme is de basis van sommige algoritmen die in de praktijk heel goed presteren en vaak gebruikt worden – zoals bv. zip en gzip.
Het idee: Het idee is dat je de tekst zelf als jouw woordenboek gebruikt. Een woordenboek is natuurlijk alleen maar nuttig als het ook bij de context past (bv. zou een Nederlands woordenboek nauwelijks nuttig zijn voor een Engelse tekst), maar de tekst als woordenboek past natuurlijk ideaal bij zijn eigen context! Dus: aan de ene kant heb je de tekst toch al nodig en aan de andere kant heb je dan ook automatisch de woorden die je nodig hebt. Natuurlijk kan je niet de hele tekst bijhouden (als het bv. een stream is) – dus gebruik je altijd alleen maar een eindig deel van de tekst – bv. de laatste 32 KB die je gezien hebt. Dit eindige deel noemen wij het *sliding window*. Dit sliding window is in het begin leeg en bevat tijdens het algoritme altijd de laatste G bytes waarbij G een getal is dat je op voorhand vastlegt. Zo heb je wel minder woorden ter beschikking maar als de tekst qua structuur verandert heb je de actuele woorden toch al ter beschikking! In de code houd je dan altijd bij waar het gecodeerde woord in het sliding window begint en hoe lang het is. Wat *het woord* is waarnaar je verwijst hangt ervan af wat de langste match in het sliding window is. Maar je moet er natuurlijk ook voor zorgen dat in het geval dat er helemaal geen match in het sliding window is het volgende letterteken gecodeerd kan worden.
Het algoritme werkt als volgt:



neem langste match: positie 3 lengte 3

Algoritme 14 LZ77:
Stel dat het bestand dat gecodeerd moet worden de rij van lettertekens $x_0, x_1 \ldots$ is. Als wij het over een positie in het bestand hebben dan bedoelen wij de index. Analoog gebruiken wij positie voor de lettertekens die op dat moment in het sliding window staan – daar beginnen wij met het eerste teken in het sliding window (om het even wat de index van dit teken in het bestand is) en geven die het nummer 0.

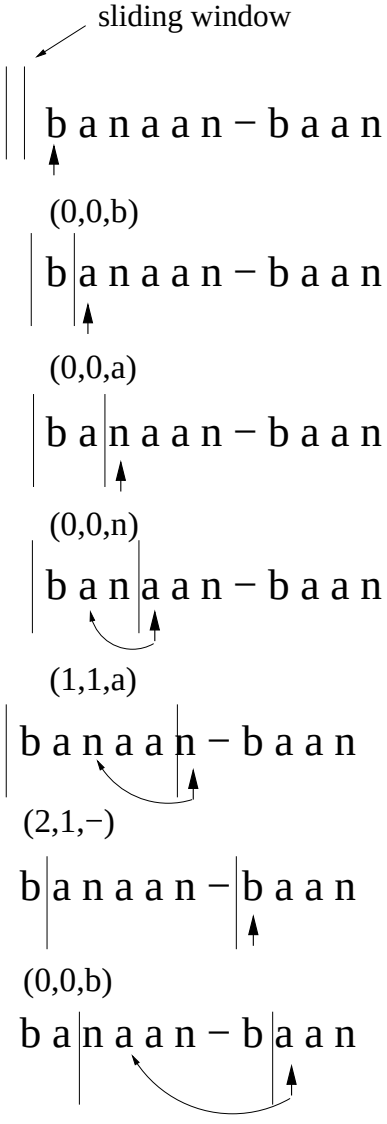
- *Houd altijd een sliding window van de laatste G bytes bij waarbij G een op voorhand vastgelegd getal is.*
- *Als de tekst vanaf x_k gecodeerd moet worden dan wordt de langste match gezocht voor een woord dat op positie k begint. De match moet met een letterteken in het sliding window beginnen – maar het einde van de match mag er wel buiten liggen.*

  Stel dat de index in het sliding window van het begin p is, de lengte l en het letterteken dat op de langste match volgt x. Als er geen match is, neem gewoon $p = 0$ maar omdat $l = 0$ doet het er niet echt toe – ook andere waarden voor p zouden dan werken.

  Dan wordt het 3-tal (p, l, x) als code geschreven.
- *Het volgende teken vanaf waar gecodeerd moet worden, is op positie $k + l + 1$ in het bestand.*

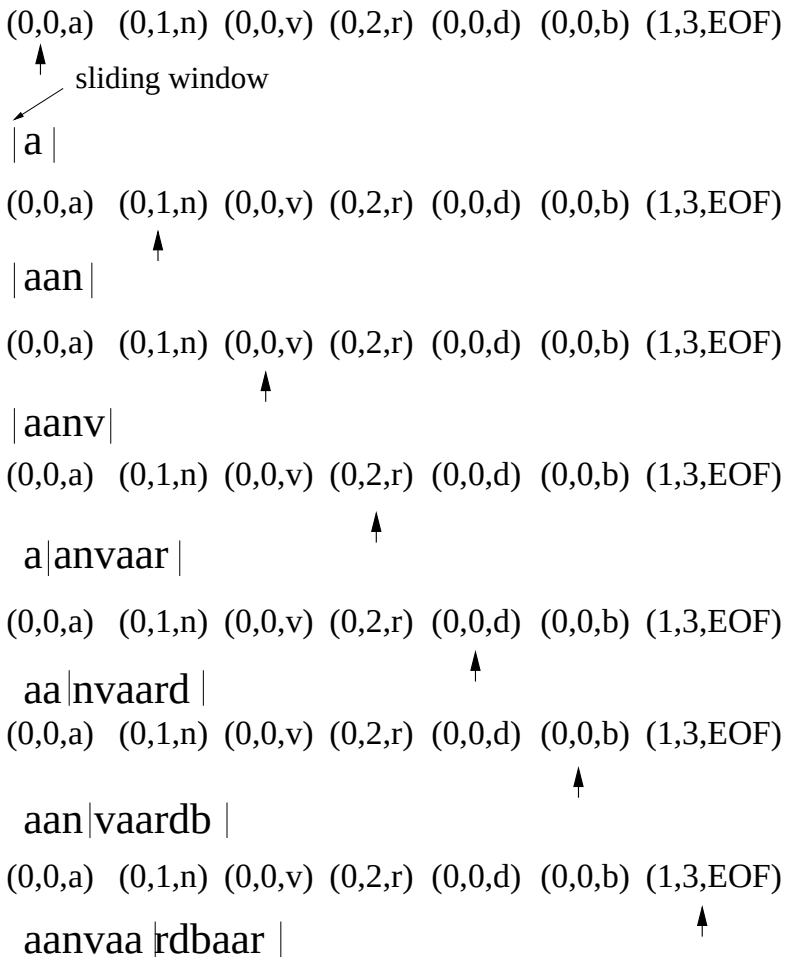
Daarbij kan je natuurlijk de positie op verschillende manieren beschrijven – bv. door het aantal tekens vanaf het begin van het sliding window of vanaf het einde van het sliding window, etc.
Je kan ook getallen en lettertekens als aparte lettertekens voor een alfabet beschouwen en dan ofwel de positie en de lengte opslaan ofwel het niet gevonden teken (dus (p, l) ofwel x). Omdat het verschil tussen getallen en lettertekens herkend kan worden is dat geen probleem. Of je kan alleen maar het letterteken schrijven als de lengte 0 is ofwel...

Al deze manieren van doen zijn mogelijk en verwezenlijken het idee van LZ77. Wij zullen nu een voorbeeld zien waar wij de codering met 3-tallen gebruiken. In de les gebruiken wij alleen deze codering waarbij we de positie vanaf de linkerkant tellen en het eerste teken de positie 0 krijgt.
Stel dat de *tekst* banaan-baan gecodeerd moet worden en dat de grootte van het sliding window 6 is (om iets te tonen moet het natuurlijk zo klein zijn. . .)



Deze output kan dan nog Huffman gecodeerd worden zodat er bv. rekening mee gehouden kan worden dat sommige lengten vaker voorkomen dan andere. Daarbij heb je verschillende mogelijkheden: je kan de output als een reeks van bytes zien – dus ook de lengten gewoon als strings van cijfers. Maar de

symbolen in jouw Huffman-boom zouden ook de hele drietallen kunnen zijn. Je kan ook niet alleen één Huffman boom gebruiken maar bv. twee – één voor de getallen die posities en lengten beschrijven en één voor de lettertekens – of zelfs drie – één voor de posities, één voor de lengten en één voor de lettertekens. Omdat je elke keer als je een nieuw symbool wil lezen, weet of het een lengte, een positie of een letterteken moet zijn, weet je welke boom je moet gebruiken. In het geval van de interpretatie van de output als gewone tekst van bytes heeft dat zeker voordelen. Je kan dan bv. dezelfde korte bitstring voor verschillende tekens gebruiken in de verschillende omstandigheden. In het geval dat je de drietallen als tekens beschouwt, heb je dan wel kortere codes voor elk teken – maar er zijn ook twee of drie codes die gelezen moeten worden terwijl anders een heel drietal door één code beschreven wordt. Het is dus niet op voorhand duidelijk wat in dit geval beter is. Een voordeel zou misschien kunnen zijn dat meer rekening wordt gehouden met de individuele hoeveelheden van de verschillende symbolen in het drietal. Terwijl zekere posities of lengtes misschien vaker voorkomen dan andere, is dat misschien minder het geval als zich hele drietallen moeten herhalen. Of dat echt zo is en voldoende winst oplevert om meer codes te rechtvaardigen is een goede programmeeroefening! De programma's *DEFLATE* en *gzip* gebruiken bv. twee bomen: één voor de posities van de matches en één voor de rest (lengtes en het volgende teken) (waarbij nog verschillende optimalisaties aan bod komen) – dat is een mogelijke oplossing die goede resultaten oplevert.
Tijdens het decoderen wordt hetzelfde sliding window opgebouwd – altijd als een code naar het woordenboek verwijst staat het gezochte woord dus ook ter beschikking.
Ook dit kan het gemakkelijkst met een voorbeeld duidelijk gemaakt worden. Wij stellen ook hier dat de lengte van het sliding window maar 6 is:



Natuurlijk zijn deze voorbeelden weinig overtuigend – de codes zijn normaal veel langer dan het door ons gecodeerde woord en ook het sliding window maakt natuurlijk weinig kans overeenkomsten te vinden als het zo klein is! Maar realistische voorbeelden kan je natuurlijk niet met de hand uitwerken...

Oefening 89 *Codeer de teksten*

- `swiss_miss_missing`
- `een_benen_been`
- `ananas-kanaal`

met LZ77 (zonder het resultaat achteraf nog met het Huffman algoritme te coderen). Gebruik een sliding window van grootte 8*. Denk eraan dat matches in de sliding window moeten beginnen maar niet noodzakelijk eindigen!*

Oefening 90 *In dit voorbeeld hebben wij de code achteraf niet nog Huffman gecodeerd. Decodeer de volgende twee codes met een sliding window met grootte 6:*

- $(0, 0, e)(0, 1, n)(0, 0, d)(1, 2, b)(0, 4, EOF)$
- *Als je het volgende woord decodeert gebeurt er iets raars – maar dat kan:*

  $(0, 0, b)(0, 0, l)(0, 0, a)(0, 9, EOF)$

Oefening 91 *Welk algoritme voor string matching zou je gebruiken om een langste match te vinden. Geef redenen voor jouw keuze.*

Oefening 92

- *Comprimeer de volgende tekst met het LZ77 algoritme en pas achteraf Huffman met twee bomen toe. Gebruik een sliding window dat duidelijk groter is dan deze kleine tekst.*

  ```
  abracadabra_simsalabim
  ```

- *decodeer achteraf jouw code om te zien of je inderdaad hetzelfde resultaat krijgt.*

In de praktijk presteren op LZ77 gebaseerde algoritmen heel goed qua compressiefactor. Maar een probleem is de tijdscomplexiteit als je met grote sliding windows werkt. Natuurlijk is het zo dat hoe groter het sliding window is, hoe meer tijd het vinden van de langste match vraagt. Een sliding window heeft geen structuur – zoals een boom – die het je gemakkelijker maakt om te zoeken. Een dergelijke structuur opbouwen is ook niet zo eenvoudig omdat de sliding window altijd verandert...
Het decoderen is altijd efficiënt – om het even hoe groot het sliding window is. Als je de posities op die manier van de voorbeelden beschrijft, moet het sliding window tijdens het decoderen even groot zijn als tijdens het coderen. Als je de posities vanaf de rechterkant van het sliding window gebruikt, moet het niet even groot zijn maar ten minste even groot.

4.4 LZW

(Abraham Lempel, Jacob Ziv en Terry Welch 1984)
LZW is gebaseerd op het LZ78 algoritme dat Lempel en Ziv in 1978 voorstelden. Het grote verschil met LZ77 (en de reden waarom wij het hier voorstellen – wij willen tenslotte vooral nieuwe ideeën zien) is dat hier een

echt woordenboek wordt gebruikt en niet – zoals in LZ77 – de tekst zelf als woordenboek.
Natuurlijk is het woordenboek ook hier gebaseerd op de actuele tekst om alleen maar *nuttige* woorden in het woordenboek te hebben. Hoe groter het aantal woorden hoe langer de codes om de woorden te beschrijven – het is dus belangrijk zo weinig mogelijk overbodige woorden in het boek te hebben. De klemtoon ligt hier ook op de mogelijkheid efficiënt in het woordenboek te kunnen zoeken.
Coderen: Eerst zullen wij het principe uitleggen voordat wij de details beschrijven:
Je wil teksten van bytes coderen in codewoorden van een op voorhand vastgelegde lengte – meestal 12 bits. Je kan dus getallen $0 \ldots 4095$ voorstellen. Het woordenboek zal maximaal 4096 woorden bevatten die door deze nummers worden voorgesteld. In het begin vul je de *gewone bytes* als woorden in. Zij krijgen als nummer het getal dat ze voorstellen – dat is dus $0, \ldots, 255$. De rest van het woordenboek zal nu dynamisch opgebouwd worden. Daarbij voeg je elke keer dat je een nieuw woord tegenkomt – dat is dan een woord dat 1 letterteken langer is dan de tot nu toe langste deelstring van dit woord in het woordenboek – dit woord toe aan jouw woordenboek. Dat blijf je doen totdat het woordenboek volzit. Je gebruikt de code van een woord dus pas de tweede keer dat je het woord tegenkomt – de eerste keer gebruik je het woord min het laatste teken en voeg je het woord toe aan het woordenboek. Maar natuurlijk is dat alleen maar een informele beschrijving. De details zie je in de volgende pseudocode:

```
voeg eerst de lettertekens toe aan jouw woordenboek
en geef de nummers vanaf 0 in volgorde

//w is ons woord
w= de lege string

while (w!=EOF)
  {
   k= volgende teken
   if (wk al in woordenboek)
        w=wk
   else
        { output code van w
          voeg wk toe aan het woordenboek
          geef wk de volgende code
```

```
          w=''k''
        }
  }
```

Een voorbeeld toont hoe het algoritme werkt:

b a n a a n a a n v a l

↑ in boek

b a n a a n a a n v a l

↑ niet in boek output code(b)=98

voeg ba toe aan woordenboek met nummer 256

b a n a a n a a n v a l

↑ niet in boek output code(a)=97

voeg an toe aan woordenboek met nummer 257

b a n a a n a a n v a l

↑ niet in boek output code(n)=110

voeg na toe aan woordenboek met nummer 258

b a n a a n a a n v a l

↑ niet in boek output code(a)=97

voeg aa toe aan woordenboek met nummer 259

b a n a a n a a n v a l

in boek ↑

b a n a a n a a n v a l

niet in boek ↑ output code(an)=257

voeg ana toe aan woordenboek met nummer 260

b a n a a n a a n v a l

in boek ↑

b a n a a n a a n v a l

niet in boek ↑ output code(aa)=259

voeg aan toe aan woordenboek met nummer 261

b a n a a n a a n v a l

niet in boek ↑ etc.....

Als het woordenboek volzit kan je verschillende dingen doen:

- begin gewoon opnieuw met een nieuw woordenboek
- ga door met dit woordenboek tot het inefficiënt blijkt te zijn (weinig lange matches)
- verwijder het woord met meer dan 1 letterteken uit het woordenboek dat je het langst niet meer gebruikt hebt (als code of als prefix van een code).

Al deze manieren van werken worden toegepast (GIF, unix compress, British Telecom Standard) en jullie hebben zeker ook zelf interessante ideeën wat je zou kunnen doen...

Decoderen:
Tijdens het decoderen wordt hetzelfde woordenboek opgebouwd als tijdens het coderen. Ook hier worden eerst de lettertekens aan het woordenboek toegevoegd. Elke keer dat een code gelezen wordt, zit die al in het woordenboek. Om dat te garanderen wordt tijdens het coderen een woord pas door zijn code beschreven als wij het woord voor de tweede keer tegenkomen. Dan wordt dit woord samen met het eerste letterteken van de volgende code aan het woordenboek toegevoegd. En hier hebben wij een probleem: Het volgende woord kan inderdaad de tweede keer zijn dat wij het woord tegenkomen dat wij net aan het woordenboek willen toevoegen.

Voorbeeld: Codeer **aaaaaaaa...**.
De code begint met 97 voor **a** en **aa** wordt aan het woordenboek toegevoegd (code 256). Dan wordt 256 als code geschreven en **aaa** toegevoegd (code 257) etc.

Als wij nu decoderen, schrijven wij eerst **a** en dan willen wij **a** gevolgd door het eerste letterteken van het woord met code 256 aan het woordenboek toevoegen – maar nummer 256 willen wij net op dit moment bepalen....

Er is dus een probleem als (en slechts als) een woord onmiddellijk nadat het aan het woordenboek wordt toegevoegd als code wordt geschreven. Dan wordt het tijdens het decoderen niet teruggevonden. Maar wij weten al hoe het woord begint: omdat in dit geval het vorige woord alle lettertekens behalve het laatste bevat, begint het nieuwe woord met hetzelfde teken. En wij hebben alleen het eerste teken nodig om het nieuwe woord te bepalen dat toegevoegd moet worden.
Ook hier zie je de details het best in pseudocode waarbij `[woord,teken]` voor de string staat die met de string `woord` begint en daarop het teken `teken` volgt.

```
voeg eerst de lettertekens toe aan jouw woordenboek
en geef de nummers vanaf 0 in volgorde

// eerst de eerste code lezen

c=gelezen code // een getal
w=woord(c) // het woord met deze nummer in het woordenboek

output w

while (nog een code om te lezen)
   {
     c= volgende code
     if (c in woordenboek)
        volgend_woord=woord(c)
     else
        volgend_woord= [w,w[0]]
        // de string w met het teken w[0] achteraan

     voeg [w,volgend_woord[0]] toe aan het woordenboek
                               en geef het volgende nummer

     w=volgend_woord
     output w
   }
```

Ook hier helpt een voorbeeld om te zien hoe het werkt:

97 110 256 258 115

woord= a tekst=a | nog voor de lus

97 110 256 258 115

woord= a volgend woord= n
woordenboek: 256=an tekst=an

97 110 256 258 115

woord= n volgend woord= an
woordenboek: 257=na tekst=anan

97 110 256 258 115
woord= an volgend woord= ana (258 nog niet in woordenboek!)
woordenboek: 258=ana tekst=ananana

97 110 256 258 115

woord= ana volgend woord= s
woordenboek: 259=anas tekst=anananas

Over het feit of LZW of LZ77 beter comprimeren vind je verschillende uitspraken. Daarbij zijn zeker vooral de precieze implementaties belangrijk en natuurlijk de toepassingen. Maar één voordeel heeft LZW zeker: het coderen kan op een efficiëntere manier gebeuren dan voor LZ77 omdat het dure opzoeken van een langst mogelijke deelstring niet moet gebeuren. Het zoeken of een deelstring al in het woordenboek zit, kan op een veel efficiëntere manier gebeuren (bv. met tries – bomen waar de bogen labels hebben zoals in de Huffman bomen).

Oefening 93 *Codeer de teksten*

- `swiss_miss_missing`
- `een_beenen_been` *(met spelfout werkt het iets beter . . .)*
- `ananas-kanaal`

met LZW.

Oefening 94 *Decodeer de volgende met LZW gecodeerde tekst. De nieuwe woorden in het woordenboek beginnen met nummer 256. Geef ook de woorden met code ten minste 256 in het woordenboek. Wees niet verrast als de tekst niet echt zinnig is... Lettertekens worden hier als tekens gegeven en niet als hun code (om de ASCII code niet te hoeven opzoeken).*

```
a 256 b c a b 257 260
```

Oefening 95 *Welke datastructuren zou je gebruiken om jouw woordenboek voor het LZW algoritme bij te houden? Bespreek het woordenboek dat je bij het coderen gebruikt en het woordenboek dat je bij het decoderen gebruikt apart.*

4.5 De Burrows-Wheeler transformatie.

De Burrows-Wheeler transformatie (Michael Burrows en David Wheeler) werd in een artikel van 1994 gepubliceerd, maar gaat terug op onderzoek dat Wheeler al in 1983 deed en toen (nog) niet publiceerde.
De Burrows-Wheeler transformatie is de basis van het programma *bzip2* dat in de meeste gevallen heel goede resultaten oplevert (beter dan bv. *gzip* of *zip*). Maar voor ons is het vooral interessant omdat het een fundamenteel nieuw idee bevat: waarom niet de tekst wijzigen (transformeren) zodat hij achteraf beter gecomprimeerd kan worden? Natuurlijk is het belangrijk dat de wijziging achteraf ongedaan gemaakt kan worden om de originele tekst terug te krijgen.
Wij zullen het algoritme uitleggen aan de voorbeeldtekst **geen_loon**. De Burrows-Wheeler transformatie gebruikt de tekst niet sequentieel, maar begint al met heel grote stukken van de tekst. Misschien gebruikt het de hele tekst, maar ten minste voldoende grote stukken (blokken). Hoe groter de blokken zijn hoe beter de compressie – maar vooral de compressie is ook duurder voor grotere blokken. In onze kleine voorbeelden zal de tekst natuurlijk helemaal in één blok getransformeerd kunnen worden.
Eerst zullen wij de tekst zo interpreteren alsof het geen string is, maar een cykel van karakters, dus alsof na het laatste letterteken terug het eerste letterteken komt. Stel dat de lengte van de tekst n is. Dan kan je van deze *cyclische tekst* n strings van lengte n krijgen door op de n verschillende posities te beginnen. In ons voorbeeld is $n = 9$ en wij krijgen

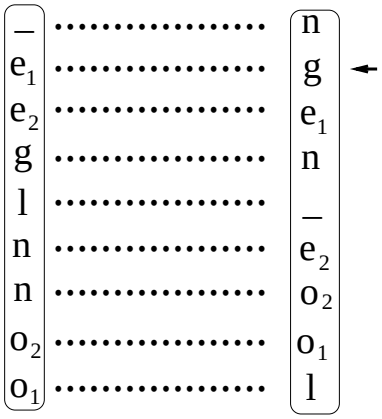
g	e	e	n	_	l	o	o	n
e	e	n	_	l	o	o	n	g
e	n	_	l	o	o	n	g	e
n	_	l	o	o	n	g	e	e
_	l	o	o	n	g	e	e	n
l	o	o	n	g	e	e	n	_
o	o	n	g	e	e	n	_	l
o	n	g	e	e	n	_	l	o
n	g	e	e	n	_	l	o	o

Hierbij toont een pijltje wat het eerste teken van de tekst in de laatste kolom staat. Wij zien dat in elke rij en elke kolom de hele tekst staat. Als jullie het echt willen implementeren zouden jullie natuurlijk het best niet echt n kopieën van de tekst gebruiken, maar een enkele kopie en de verschillende strings gewoon voorstellen als pointers naar het beginkarakter!
Nu sorteren wij de rijen. Daarbij gebruiken wij de waarden van de bytes (een blank is kleiner dan de lettertekens), maar omdat onze voorbeelden voor compressie met gewone teksten met kleine lettertekens werken, komt dat hier overeen met de lexicografische volgorde. Dan krijgen wij

_	l	o	o	n	g	e	e	n
e_1	e	n	_	l	o	o	n	g
e_2	n	_	l	o	o	n	g	e_1
g	e	e	n	_	l	o	o	n
l	o	o	n	g	e	e	n	_
n	_	l	o	o	n	g	e	e_2
n	g	e	e	n	_	l	o	o_2
o_2	n	g	e	e	n	_	l	o_1
o_1	o	n	g	e	e	n	_	l

Daarbij hebben wij een index bij de o en bij de e in de eerste en laatste kolom geschreven om te kunnen zien *welk van de identieke lettertekens (bytes) het is*. Maar dat is alleen maar ter illustratie en niet deel van de methode.
Wat wij nu **eerst** stellen is dat alle rijen van deze matrix verschillend zijn. Als dat niet zo was, kan je dat gemakkelijk zien omdat dan in de gesorteerde matrix identieke rijen achter elkaar moeten staan. In dit geval is de hele tekst een herhaling van een deelstring en je kan het best de deelstring opslaan (misschien nog gecomprimeerd) en hoe vaak je hem moet herhalen. Een

andere oplossing zou zijn dat je op het einde van de originele string een teken toevoegt dat nog niet in de string zit en na de inverse transformatie opnieuw verwijderd moet worden. Dat kan bv. een teken zijn dat kleiner is dan alle tekens in de string (als dat kan). Op deze manier kan je op voorhand garanderen dat alle rijen verschillend zijn!
De getransformeerde tekst is nu een kolom van deze matrix. Voor compressiedoeleinden zou het natuurlijk ideaal zijn de eerste kolom te gebruiken – in een lange tekst zouden daar in het begin alleen maar a's staan, dan alleen maar b's, etc – je zou dus alleen maar de hoeveelheden moeten bijhouden. Jammer genoeg is er geen manier om uit deze kolom de originele (cyclische) tekst te reconstrueren. Maar als het om een *echte* en niet om een toevallige tekst gaat, heeft ook de laatste kolom nog voordelen van het sorteren: ook hier is de verdeling van de lettertekens niet toevallig omdat de lettertekens de prefixen van de lettertekens in de eerste kolom zijn. En wat verrassend is: terwijl je uit de eerste kolom de cyclische tekst niet terug kan krijgen (zonder dure extra informatie op te slaan) kan je dat uit de laatste kolom wel! Je moet alleen nog bijhouden waar het eerste letterteken $s[0]$ van de string $s[]$ staat om ook de niet cyclische string te reconstrueren.
Een voordeel is dat als wij de laatse kolom hebben, wij de eerste kolom gemakkelijk kunnen verkrijgen: het zijn dezelfde lettertekens als in de laatste kolom en wij moeten die gewoon sorteren! Daarbij houden wij natuurlijk geen rekening met de indices (tenslotte zijn die er niet echt), maar stel voor het moment dat je ook de indices kent. Wij hebben dus



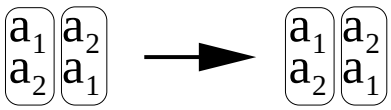
Het transformatiealgoritme kan dus als volgt geschetst worden:

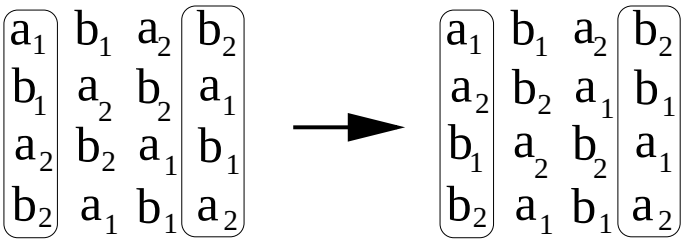
- interpreteer de tekst $s[0], s[1], \ldots, s[n-1]$ als cyclisch en vorm alle n strings $s_i[]$ met $0 \leq i < n$ die je krijgt door op positie i te beginnen.

- sorteer deze n strings in stijgende lexicografische volgorde. Wij schrijven $p_s(i)$ voor de string op positie i na het sorteren – $p_s(0)$ is dus de kleinste string.
- output het nummer i van de rij die met $s[0]$ eindigt – dus de rij i met $p_s(i)[n-1] = s[0]$. Dat is de rij waar het letterteken waarmee de originele string begint op de laatste positie staat. Output dan een blank.
- output de lettertekens $p_s(0)[n-1], \ldots, p_s(n-1)[n-1]$ op de laatste posities in deze volgorde.

Het eerste teken van de originele tekst staat in ons geval in rij 1 als wij met 0 beginnen te tellen. De Burrows-Wheeler getransformeerde van **geen_loon** zou dus bv. **(1_ngen_eool)** zijn als wij vastleggen dat het getal eerst komt en de tekst na de eerste blank staat. Dit is één mogelijke manier van doen, maar je zou ook eerst een integer van vaste lengte voor de positie kunnen gebruiken.
Maar nu is het gemakkelijk de cyclische tekst opnieuw op te bouwen. Wij moeten voor elk letterteken alleen het volgende letterteken kennen – maar na het letterteken dat in de laatste kolom in rij i staat komt het letterteken dat in de eerste kolom in rij i staat – en dat kennen wij!
Als wij dus met het eerste teken van de tekst beginnen – dus het teken nummer 1 van **ngen_eool** dan is dat **g** en in de eerste kolom staat op positie 1 $\mathbf{e}_1$ – het volgende teken is dus $\mathbf{e}_1$. In de laatste kolom staat $\mathbf{e}_1$ in rij 2 en in rij 2 in de eerste kolom staat $\mathbf{e}_2$. Dat vinden wij terug in rij 5 in de laatste kolom, etc... Als wij zo doorgaan krijgen wij de oorspronkelijke tekst **geen_loon**.

Er blijft maar één probleem: toen wij opgezocht hebben welk van de e's aan de rechterkant de **e** is die naar de **g** komt, hebben wij de indices gebruikt – maar die zijn er in het echt niet – daar zijn het alleen maar a's en o's!
Als jullie naar de volgorde kijken waarop de indices opduiken dan is het in de eerste kolom eerst $\mathbf{e}_1$ en dan $\mathbf{e}_2$ – en voor de tweede kolom ook. Voor de o's is het eerst $\mathbf{o}_2$ en dan $\mathbf{o}_1$ in de eerste kolom – en in de tweede kolom ook. En dat is inderdaad geen toeval: wij kunnen identieke lettertekens in de eerste en laatste kolom aan elkaar toekennen in de volgorde waarin ze opduiken!
Om dat te bewijzen stel dat de string $s[] = s[0] \ldots s[n-1]$ is en dat $s[i] = s[j]$ waarbij $s[i]$ in de gesorteerde matrix in de eerste kolom voor $s[j]$ komt. Wij zouden nu graag bewijzen dat dan ook in de laatste kolom $s[i]$ voor $s[j]$ komt, maar jammer genoeg is dat niet altijd zo:





Maar als wij stellen dat de rijen allemaal verschillend zijn, kunnen wij gemakkelijk bewijzen dat de volgorden in de eerste en laatste kolom dezelfde zijn:
Stel dat $s[i] = s[j]$ en dat $s[i]$ in de gesorteerde matrix in de eerste kolom voor $s[j]$ komt – dus
$(s[i], s[i+1] \dots s[n-1]s[0] \dots s[i-1]) < (s[j], s[j+1] \dots s[n-1]s[0] \dots s[j-1])$.
Omdat $s[i] = s[j]$ geldt dus
$(s[i+1] \dots s[n-1]s[0] \dots s[i-1]) < (s[j+1] \dots s[n-1]s[0] \dots s[j-1])$ en dus ook
$(s[i+1] \dots s[n-1]s[0] \dots s[i-1]s[i]) < (s[j+1] \dots s[n-1]s[0] \dots s[j-1]s[j])$.
Maar dat betekent precies dat in de laatste kolom de rij waarin $s[i]$ staat voor de rij komt waarin $s[j]$ staat.
Maar precies dat wouden wij bewijzen en dus is onze methode om lettertekens met elkaar te identificeren (indices toe te kennen) onder deze omstandigheden juist.

Het terugtransformatiealgoritme kan dus als volgt geschetst worden:
Stel dat de code $(p, c[0], \dots, c[n-1])$ is.

- kopieer $c[0], \dots, c[n-1]$ en sorteer de tekens. Noem de nieuwe string $e[]$. In $e[i]$ staat dus het letterteken dat in de originele tekst na het teken $c[i]$ komt.

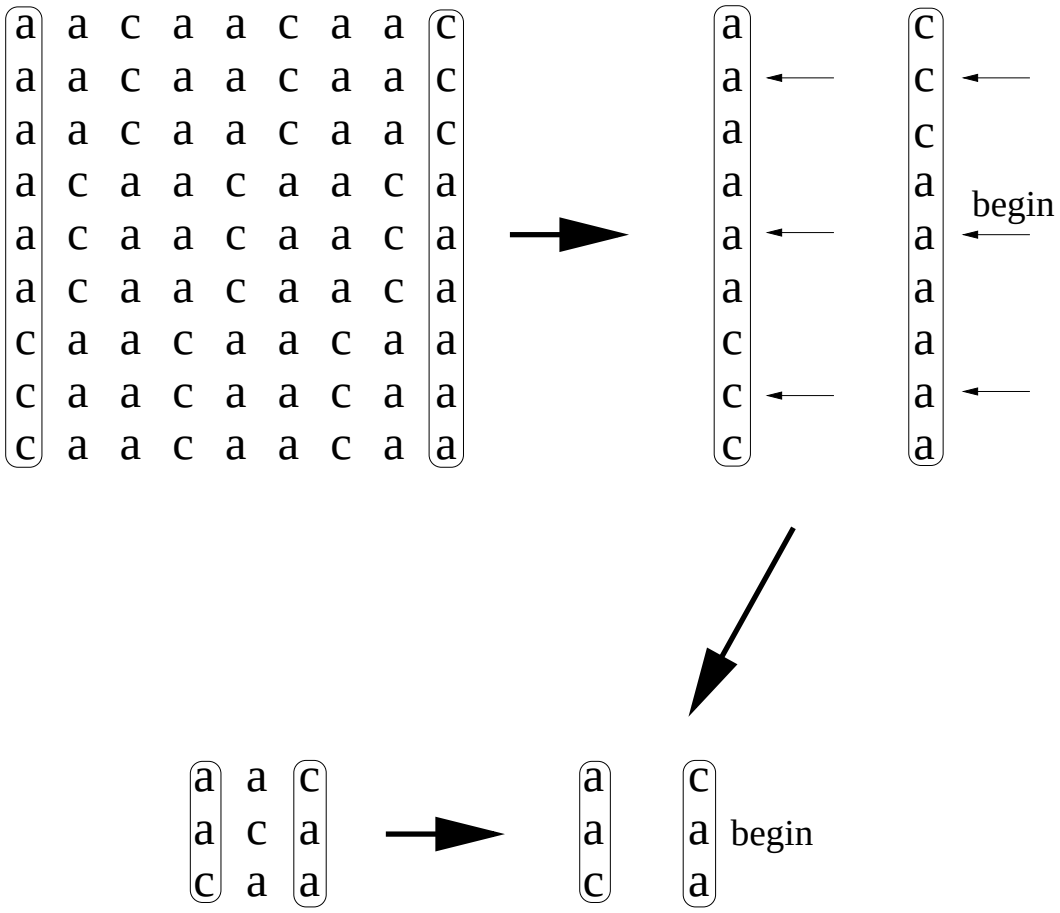
- voor elk letterteken x dat k keer in $c[]$ en dus ook k keer in $e[]$ zit, ken indices $1, \dots, k$ toe aan de posities van x in $c[]$ in de volgorde van de string. Doe hetzelfde voor $e[]$. Wij schrijven x_k voor het letterteken x met index k. Elk geïndexeerd letterteken zit dus één keer in $c[]$ en één keer in $e[]$.

- begin met $x_j = c[p]$ dan herhaal n keer:

- output het letterteken x (zonder index).
- bepaal het nummer i zodat $c[i] = x_j$.
- kies $x_j = e[i]$ als volgend geïndexeerd letterteken.

Jullie zullen op het internet ook beschrijvingen vinden waar jullie de lus niet n *keer* moeten herhalen, maar *totdat je terug bent bij het eerste letterteken.* Maar wij zullen zien dat het voordelen heeft het zo te doen...

Wat zou er gebeuren als wij het algoritme zouden toepassen op een string die een herhaling van deelstrings is – dus waar twee of meerdere rijen identiek zijn?
Stel dat de string $t[0], \ldots, t[k-1]$ niet tot identieke rijen leidt maar de hele string $s[]$ wel, omdat hij uit l herhalingen van $t[]$ bestaat. Dus $s[] = t[0], \ldots, t[k-1], t[0], \ldots, t[k-1], \ldots, t[0], \ldots, t[k-1]$. Als wij op deze string het algoritme toepassen, hebben wij elke rij precies l keer achter elkaar. De hele matrix is dus een opeenvolging van k blokken waarvan elke blok l identieke rijen bevat. Rijen in verschillende blokken zijn verschillend. Maar voor de transformatie doet dat er natuurlijk niet toe – wij kunnen gewoon de getransformeerde tekst construeren.
Nu kijken wij naar het toekennen van indices in $c[]$ (en analoog in $e[]$). Tijdens de terugtransformatie zal een teken in $e[]$ of $c[]$ dat op positie $i < l$ in zijn eigen blok zit zeker een index j krijgen zodat $i = j (\mathrm{mod}\, l)$: in zijn blok is het zeker het i-de en in de blokken daarvoor was het identieke letterteken ofwel niet op de eerste (of laatste) positie aanwezig ofwel l keer – dus zal de index j in elk geval aan $i = j (\mathrm{mod}\, l)$ voldoen.
Maar als ons beginteken een index b heeft dan zien wij dat in het hele algoritme **nooit** een letterteken met een index i gebruikt wordt die niet aan $i (\mathrm{mod}\, l) = b (\mathrm{mod}\, l)$ voldoet. Het algoritme gebruikt dus alleen maar één rij van elk blok. Maar als wij naar de eerste en laatste lettertekens van alleen deze rijen kijken dan zien wij dat dit dezelfde zijn als bij het toepassen van het algoritme alleen op $t[0], \ldots, t[k-1]$. Een voorbeeld ervan zien wij in Figuur 4.5. Inderdaad zullen wij dus na k stappen terug zijn met ons beginteken – maar omdat wij even veel stappen doen als er lettertekens in de getransformeerde tekst zitten – dus $l * k$, zullen wij doorgaan en de tekst $t[0], \ldots, t[k-1]$ l keer herhalen. De originele tekst wordt dus inderdaad ook in dit geval juist teruggetransformeerd.
Het lijkt natuurlijk niet echt efficiënt als wij het grootste deel van de getransformeerde tekst helemaal niet gebruiken – en ook niet nodig hebben. Maar wij zullen zien dat in dit geval de volgende stappen ook bijzonder goed comprimeren.



Figuur 25:

Wat hebben wij tot nu toe gewonnen? De getransformeerde tekst heeft even veel lettertekens. En als wij nu het Huffman algoritme zouden toepassen, hebben wij nog altijd niets gewonnen: de relatieve aantallen lettertekens zijn dezelfde als in de originele tekst – de code zou dus even lang zijn (in feite zelfs iets langer omdat het begin van de tekst ook opgeslaan moet worden).

Wat is dan het voordeel?
De volgorde van de strings is lexicografisch. De meeste invloed daarbij heeft natuurlijk het eerste letterteken in elke rij – daar hebben wij een mooie ordening. Maar het laatste letterteken is een prefix van het eerste. In een toevallige tekst betekent dat niet veel omdat elk teken dezelfde kans heeft een prefix te zijn. Maar wij weten al dat je toevallige data niet kan comprimeren. In *echte* teksten (en ook andere bestanden) zijn dergelijke prefixen niet toevallig. De strings die bv. met *aa* beginnen, zullen allemaal achter

elkaar staan. Maar het is zeker niet zo dat elk letterteken dezelfde kans heeft een prefix van *aa* te zijn. Als jullie bv. in het groene boekje kijken dan zijn er veel woorden die met **b**aa of **l**aa beginnen, duidelijk minder met **f**aa maar geen enkel woord begint met **c**aa. Dat zegt natuurlijk niet alles, omdat natuurlijk de prefixen niet altijd op het begin van een woord staan, maar het geeft al een idee. En als jullie naar langere deelstrings kijken, wordt dat nog duidelijker (bv. de prefixen van *aan* of nog langere strings...). Wij kunnen dus hopen dat er in de rijen waar *aa* de volgorde bepaalt op het einde veel b's of l's bij elkaar staan. Nog duidelijker wordt het als jullie naar langere strings kijken. Alle strings die met *eze* beginnen, zullen achter elkaar staan – en de meeste ervan zullen zeker op de laatste positie een **d** hebben. In onze code zullen dus veel d's op elkaar volgen of ten minste heel dicht bij elkaar staan. De letters zijn dus beter gegroepeerd. Wij hebben dus een compressiealgoritme nodig waarvoor dat een voordeel is – adaptive Huffman coding zou bv. al beter presteren!
Maar Burrows en Wheeler stellen voor deze eigenschap van de getransformeerde tekst als volgt te gebruiken:

De *move to front* methode

Je legt vast hoe je jouw alfabet met getallen codeert – je houdt de tekens bij in een lijst en elk letterteken x krijgt als code $c(x)$ zijn positie in de lijst. In de realiteit zouden dat bv. alle ASCII codes zijn – omdat je niet op voorhand weet welke lettertekens opduiken en welke niet moet je een **vaste** lijst hebben met **alle** mogelijke lettertekens! In onze voorbeelden willen wij natuurlijk niet met zo'n lange lijst werken, wij kiezen dus een kortere, maar het is belangrijk om te onthouden dat dat alleen voor de voorbeelden is en in het algemeen niet kan! Hier staat het teken "_" voor een blank, dat komt dus (code 32) voor de cijfers.
Voorbeeld: De lijst: (_,1,a,b,e,f,g,n,o,l). Dus is $c(\_) = 0$, $c(a) = 2,\ldots,c(l) = 9$ etc.
Als jullie nu een string coderen, dan schrijven jullie voor elk letterteken x zijn code $c(x)$ **en** plaatsen dit letterteken dan in het begin van de lijst – het heeft **achteraf** dus code 0.
Als jullie met de lijst hierboven dus **(1_ngen_eool)** willen coderen dan werkt dat als volgt:

de lijst is (_,1,a,b,e,f,g,n,o,l): en **(1_ngen_eool)** moet gecodeerd worden.

lees:**1**: 1 schrijven, lijst nu (1,_,a,b,e,f,g,n,o,l)
lees:_: 1 schrijven, lijst nu (_,1,a,b,e,f,g,n,o,l)

lees:**n**: 7 schrijven, lijst nu (n,␣,1,a,b,e,f,g,o,l)
lees:**g**: 7 schrijven, lijst nu (g,n,␣,1,a,b,e,f,o,l)
lees:**e**: 6 schrijven, lijst nu (e,g,n,␣,1,a,b,f,o,l)
lees:**n**: 2 schrijven, lijst nu (n,e,g,␣,1,a,b,f,o,l)
lees:␣: 3 schrijven, lijst nu (␣,n,e,g,1,a,b,f,o,l)
lees:**e**: 2 schrijven, lijst nu (e,␣,n,g,1,a,b,f,o,l)
lees:**o**: 8 schrijven, lijst nu (o,e,␣,n,g,1,a,b,f,l)
lees:**o**: 0 schrijven, lijst nu (o,e,␣,n,g,1,a,b,f,l)
lees:**l**: 9 schrijven, lijst nu (l,o,e,␣,n,g,1,a,b,f)

Dit toont wel het principe, maar het kleine voorbeeld is niet voldoende om aan te tonen dat het nuttig is. Maar omdat in de Burrows-Wheeler getransformeerde tekst heel vaak herhalingen van letters voorkomen – of ten hoogste met kleine onderbrekingen, zullen in de code zeer veel nullen en kleine getallen opduiken.
Als je het decodeert moet je natuurlijk met dezelfde lijst beginnen en die dan ook op dezelfde manier veranderen:

de lijst is:(␣,1,a,b,e,f,g,n,o,l) en **1,1,7,7,6,2,3,2,8,0,9** moet gedecodeerd worden.
lees:**1**: 1 schrijven, lijst nu (1,␣,a,b,e,f,g,n,o,l)
lees:**1**: ␣ schrijven, lijst nu (␣,1,a,b,e,f,g,n,o,l)
lees:**7**: n schrijven, lijst nu (n,␣,1,a,b,e,f,g,o,l)
lees:**7**: g schrijven, lijst nu (g,n,␣,1,a,b,e,f,o,l)
lees:**6**: e schrijven, lijst nu (e,g,n,␣,1,a,b,f,o,l)
lees:**2**: n schrijven, lijst nu (n,e,g,␣,1,a,b,f,o,l)
lees:**3**: ␣ schrijven, lijst nu (␣,n,e,g,1,a,b,f,o,l)
lees:**2**: e schrijven, lijst nu (e,␣,n,g,1,a,b,f,o,l)
lees:**8**: o schrijven, lijst nu (o,e,␣,n,g,1,a,b,f,l)
lees:**0**: o schrijven, lijst nu (o,e,␣,n,g,1,a,b,f,l)
lees:**9**: l schrijven, lijst nu (l,o,e,␣,n,g,1,a,b,f)

Na de move to front transformatie is de tekst nog altijd niet korter – je vervangt gewoon bytes met een teken door andere bytes die de positie beschrijven – maar die zijn even veel en even groot. De transformatie door de laatste kolom van de gesorteerde matrix te nemen en die dan move to front te coderen zijn dus alleen maar wijzigingen om een beter compromeerbare tekst te maken – en dat is het volledig nieuwe aan deze aanzet!
Maar omdat er **heel veel** nullen en kleine getallen inzitten is de met de move to front methode gecodeerde Burrows-Wheeler getransformeerde tekst nu bijzonder geschikt om (statisch) Huffman gecodeerd te worden – en dat

stellen Burrows en Wheeler inderdaad ook als één van de mogelijkheden voor.

Oefening 96 *Codeer de tekst* **boom_gaat** *door eerst de Burrows-Wheeler transformatie toe te passen en dan de move to front codering. De lijst voor de codering is (_,0,1,2,3,4,5,6,7,8,9,a,b,g,m,n,o,t).*

Oefening 97 *Decodeer de volgende code: (0,11,10,10,7,0,6,1,0,9,11,4,6,1) De code is de move to front code van een Burrows wheeler getransformeerde tekst waarbij het move to front algoritme met de lijst (7,a,e,i,o,u,b,c,n,t,w,_) begonnen is. De 7 is hier natuurlijk een gewoon letterteken en geen getal! Let erop dat de volgorde hier dus niet de ASCII volgorde is – maar als je voor het coderen en decoderen dezelfde volgorde gebruikt, doet dat er niet toe!*

Oefening 98 *Als wij de eerste kolom als getransformeerde tekst nemen, kunnen wij de oorspronkelijke tekst niet reconstrueren. Maar hoe zit het met de tweede kolom. Werk een algoritme voor de tweede kolom uit of geef een redenering waarom de tweede kolom als getransformeerde tekst niet geschikt is.*

Oefening 99 *Zou het algoritme werken als je in plaats van de strings in stijgende volgorde te sorteren de strings in dalende volgorde zou sorteren? Zouden er wijzigingen nodig zijn? Kijk bv. naar de redenering dat de volgorde van lettertekens in de eerste en laatse kolom dezelfden zijn. Bewijs dat dat ook met dalende volgorde het geval is of geef een tegenvoorbeeld.*

Oefening 100 *Geef een string $x\_t_0 \dots t_{n-1}$ waarbij x de voorstelling van een getal x' is met $0 \leq x' < n$ en $t_0, \dots, t_{n-1}$ lettertekens. Maar deze string mag niet het resultaat van een Burrows-Wheeler transformatie kunnen zijn. Geef uitleg.*

Oefening 101 *Werkt het volgende algoritme dat op het idee van de Burrows-Wheeler transformatie gebaseerd is juist? Bewijs dat of geef een tegenvoorbeeld.*

Neem de tekst $t = t_0, \dots, t_{n-1}$ en voeg het teken $t_n = \infty$ toe. Daarbij betekent ∞ dat het een teken is dat groter is dan welk teken ook in het alfabet. Schrijf dan s_i voor de string $t_i, \dots, t_n$ en sorteer de strings $s_0, \dots, s_n$ in lexicografische volgorde. Als nu de string s_k op positie j in deze volgorde staat dan is t_{k-1} het j-de teken van de getransformeerde tekst (met $t_{-1} = t_n$).
De terugtransformatie begint met het teken ∞ – maar zonder het te schrijven – en gaat dan door zoals in de les gezien totdat n tekens geschreven zijn.

5 Parallelle algoritmen

Parallelle algoritmen zijn algoritmen die meerdere processoren gebruiken die samenwerken om één probleem op te lossen. Het voordeel is duidelijk en het principe van parallellisme wordt al lang gebruikt: als bv. een huis wordt gebouwd dan zou (bijna) niemand proberen dat helemaal alleen te doen – het aantal werkuren is zo groot dat de mens die het huis wil bouwen misschien al overleden is als het huis eindelijk klaar is... Daarom werken meerdere mensen tegelijk. Daardoor wordt het aantal werkuren niet kleiner (misschien zelfs groter) maar de tijd tussen begin van de bouw en het einde wordt veel kleiner. Dit is ook het doel van parallelle algoritmen. Het is duidelijk dat de totale verbruikte rekentijd niet kleiner kan worden (je kan de taken van de verschillende processoren ook na elkaar op één processor laten draaien) maar de *wachttijd* tussen het begin van de berekening en het einde kan wel duidelijk korter worden.
Maar het voorbeeld met ploegen die samen werken toont ook twee problemen: soms moet je goed op de volgorde letten (bv. de muren niet schilderen als er nog kabels in de muren gelegd moeten worden) en je kan ook niet arbitrair veel werkers gebruiken. Het helpt bv. niet als 10 personen proberen een enkele schakelaar voor het licht vast te maken. Het gemakkelijkst is het taken aan verschillende werknemers toe te kennen als de taken helemaal niets met elkaar te maken hebben (bv. de kelder schilderen en tegelijk pannen op het dak plaatsen).
Hetzelfde geldt voor parallelle algoritmen: als deeltaken niets met elkaar te maken hebben, is het veel gemakkelijker ze op verschillende processoren te laten draaien dan in gevallen waar de taken sterk van elkaar afhankelijk zijn.

Een probleem bij de beslissing welke onderwerpen voor dit deel van de les gekozen worden, was dat het heel moeilijk is om te voorspellen wat in de toekomst **echt** relevant zal zijn. En dat geldt vooral voor een les waar de klemtoon vooral op een realistische kijk op de zaak ligt. Rond 1980 leek het (voor een tijdje) dat de kloksnelheid van de computers niet meer zo snel groeide en in de tijd na 1980 werd verwacht dat *parallel computing* de toekomst zou zijn. Er werden veel verschillende modellen voor parallelle computers ontworpen. Verschillende architecturen bestonden zelfs als hardware – bv. de bekende Cray supercomputers (vanaf 1976). Maar de transputers waar interprocessorcommunicatie een belangrijk deel van het ontwerp was waren bijzonder interessant. Deze ontwerpen besteedden veel tijd en geld aan het ontwerp van een efficiënt netwerk voor de interprocessorcommunicatie.
Maar het "probleem" was vermoedelijk dat door de extreem grote aantallen van kleine computers er veel meer geld aan de ontwikkeling van CPU's

besteed kon worden die geoptimaliseerd zijn om zelfstandig te werken. Daardoor verdwenen de *echte* parallelle computers langzaam en in plaats daarvan ontstonden clusters van stand-alone computers die door netwerken met elkaar verbonden waren die – in vergelijking met de kloksnelheid – zeer traag zijn. Deze ontwikkeling kan je – natuurlijk – ook in de opleiding terugvinden, waarin vroeger een hele les *parallelle algoritmen* gegeven werd die later gereduceerd werd op een relatief klein deel van DA 3 – en nu is er zelfs één les over parallele software en één les over parallele hardware.
Intussen is het opnieuw zo dat de kloksnelheden niet meer zo snel groeien – maar de processoren worden door het feit dat per clockcycle meer en ingewikkeldere bewerkingen plaats kunnen vinden wel nog altijd sneller. Toch lijkt parallellisme opnieuw belangrijker te worden (wat in toekomst misschien ook tot een groter aandeel van parallelle algoritmen in de les of zelfs een eigen les kan leiden). Bijzonder interessant zijn daarbij de multicore computers die het idee van een shared memory verwezenlijken en ook in grote hoeveelheden verkocht worden. Er gebeurt ook veel onderzoek over de mogelijkheid de grafische kaarten – die massief parallel zijn – voor berekeningen te gebruiken.
De vraag is dus in welke richting de ontwikkeling deze keer gaat. Parallelle algoritmen hebben het normaal niet over 2, 4 of 8 processoren, maar over zoiets als bv. $O(n)$ als n de grootte van het probleem is. De vraag is dus of er een ontwikkeling in de richting van multicore processoren met duidelijk meer cores zal zijn of of de ontwikkeling zal stoppen en opnieuw in een andere richting gaan...
Misschien zeggen jullie nu dat er toch heel veel clusters zijn waarop *parallel computing* wordt toegepast – bv. voor voorspellingen van het weer en ook *scientific computing*. Maar in die gevallen wordt eerder iets toegepast dat *distributed computing* genoemd wordt. Het verschil is dat in *distributed computing* de communicatie tussen de verschillende delen *relatief* klein is. In een cluster gebeurt de communicatie via een intern netwerk, maar in sommige gevallen van distributed computing gebeurt de communicatie van de computers die de deeloplossingen berekenen zelfs over het internet. De communicatie tussen verschillende rekennodes is dus **duidelijk** trager dan de communicatie van een CPU met zijn lokaal geheugen. Voor dergelijke algoritmen is de precieze vorm van het netwerk dan ook minder belangrijk omdat de communicatie tussen de delen klein is.
De klemtoon van deze les ligt wel op realistische toepassingen eerder dan theoretische resultaten, maar toch zullen jullie in dit deel van allebei iets zien...

Notatie: Wij schrijven $t_p(n)$ voor de tijd die een gegeven parallel algoritme dat p processoren voor een taak met invoerlengte n gebruikt ten hoogste

nodig heeft. Dat is dus niet de som van de tijden van de enkele processoren, maar gewoon de tijd die verstreken is tussen het begin van de berekeningen en het einde. Wij schrijven $t^s(n)$ voor de tijd die het snelste (gekende) sequentiële algoritme voor deze input nodig heeft. Het is onmiddellijk duidelijk dat als een parallel algoritme in tijd $O(t_p(n))$ draait dat dan $O(p * t_p(n))$ een bovengrens is voor $t^s(n)$ omdat door achter elkaar uitvoeren van de stappen van de p processoren het parallelle algoritme ook serieel gesimuleerd kan worden, maar de simulatie kan natuurlijk wel een zekere (constante) overhead vragen.
Het product van de tijd en het aantal processoren wordt soms ook de kost $K(n)$ van het algoritme genoemd. Wij weten door de mogelijkheid van simulatie dat dit product (op een constante na) optimaal voor seriële algoritmen is en wij noemen een parallel algoritme kost optimaal als $K(n) = O(t(n))$ waarbij $t(n)$ de tijd voor het beste seriële algoritme is.
Wij zullen nog een ander concept zien: de werk complexiteit van een algoritme. Als de scheduler op een parallelle computer slim genoeg is dan kan hij in sommige gevallen herkennen dat sommige processoren na de eerste stappen niet meer gebruikt worden. Dat is niet in elk geval gemakkelijk, maar in principe kan de programmeur zelf in sommige gevallen misschien een `exit()` in de code voor een processor i schrijven. Dan kunnen deze processoren voor andere doeleinden gebruikt worden. Het is dus niet in elk geval zinvol zo te rekenen alsof alle ooit gebruikte processoren de hele tijd bezig waren als het om de kost van een algoritme gaat. De *kost* die de tijd van de processoren alleen telt als deze processoren ook stappen voor het algoritme uitvoeren wordt de *werk complexiteit* genoemd. Of precies: het werk $W(n)$ dat een parallel algoritme gebruikt voor een invoer met lengte n is gedefinieerd als

$W(n) = \sum_{i=1}^{p} t(i, n)$

waarbij processoren $1, \ldots, p$ aan de taak werken en $t(i, n)$ het aantal stappen (de tijd) is dat processor i maximaal doet voor een invoer met lengte n. Zoals het hier gedefinieerd is, is het niet noodzakelijk zo dat de tijd die voor verschillende processoren gerekend wordt altijd voor dezelfde invoer maximaal is. Maar in de gevallen die wij hier zullen zien is dat wel zo...
Het is duidelijk dat een algoritme dat werk $W(n)$ vraagt op een seriële machine in tijd $O(W(n))$ gesimuleerd kan worden en analoog met *kost optimaal* definiëren wij dat een parallel algoritme werk optimaal is als $W(n)) = O(t(n))$ waarbij $t(n)$ de tijd voor het beste seriële algoritme is.

5.1 Branch and bound met verdeelde processoren

Als het om problemen gaat met een heel goedkope asymptotische complexiteit (bv. $O(n)$ of $O(n \log n)$ met een niet te grote constante voor de $O()$) dan moet je al extreem grote hoeveelheden van data hebben om ervoor te zorgen dat meer dan één processor nodig is om het probleem in aanvaardbare tijd af te werken. Bovendien is de bottleneck dan misschien zelfs eerder het lezen en schrijven van de data. Dat gebeurt dan het best ook verdeeld.
Maar de duurste problemen die jullie in Datastructuren en Algoritmen 2 hebben gezien, zijn NP-complete problemen waarvoor de voor de hand liggende oplossing vaak branch and bound algoritmen zijn (hoewel andere aanzetten – zoals bv. met integer programming – soms heel succesvol zijn). Hier zullen wij het erover hebben hoe je een branch and bound algoritme met een eenvoudig trucje efficiënt op verschillende machines kan verdelen.
In een abstracte vorm kan je zo'n branch and bound algoritme ongeveer als volgt schrijven (waarbij in dit voorbeeld een minimum wordt gezocht):

Algoritme 15 *(Branch and bound)*

```
beste_waarde= +oneindig; // of een gemakkelijk berekenbare
                         // bovengrens

b_and_b(configuratie)
// test de volgende configuratie in de recursieboom
{

if (is_endconfiguratie(configuratie)) // alle keuzes gemaakt
{ if (waarde(configuratie)<beste_waarde)
                    beste_waarde=waarde(configuratie);
  return;
}

else
{
  benedengrens=bepaal_benedengrens(configuratie);
  // dat is de look-ahead van branch and bound die vaak moeilijk
  // is om te vinden maar belangrijk voor de performantie. Hier
  // wordt een benedengrens voor de waarde van een mogelijke
  // eindconfiguratie bepaald die uit deze configuratie voorkomt.

  if (benedengrens<beste_waarde)
```

```
  // verbetering nog mogelijk
  {
  for (elke mogelijke directe opvolgconfiguratie)
                      b_and_b(opvolgconfiguratie);
  }
}// end else
return;
}// end b_and_b
```

Algoritme 15 wordt dan opgestart met de beginconfiguratie (de wortel van de recursieboom) en op het einde staat de beste waarde in `beste_waarde`.
Om deze abstracte beschrijving iets beter te verstaan, kunnen wij bv. naar het inpakprobleem kijken, waar gewichten $g_1 \leq 1, \ldots, g_n \leq 1$ op een manier op vrachtwagens met capaciteit 1 geplaatst moeten worden dat het totale aantal vrachtwagens minimaal is. De *beginconfiguratie* is dan bv. de situatie waar alleen gewicht g_1 op vrachtwagen 1 geplaatst is (dat mogen wij zeker stellen), een opvolgconfiguratie van een configuratie waar $g_1, \ldots, g_k$ geplaatst zijn is een configuratie waar $g_1, \ldots, g_{k+1}$ geplaatst zijn en een eindconfiguratie is een configuratie waar $g_1, \ldots, g_n$ geplaatst zijn. Een gemakkelijk berekenbare bovengrens is bv. n. De waarde van een configuratie is het aantal gebruikte vrachtwagens en `bepaal_benedengrens(configuratie)` is bv. de waarde van `configuratie` plus de som van de nog niet geplaatste gewichten die op geen vrachtwagen meer passen die al bestaat (een slechte benedengrens, maar het gaat hier alleen om het principe).
Als jullie naar de verschillende takken van de recursieboom kijken dan valt op dat de enige samenhang de waarde `beste_waarde` is.
Eén aanzet zou dus bv. zijn alle configuraties $c_1, \ldots, c_k$ op een zeker niveau – dat is de afstand x van de wortel – van de recursieboom op te slaan en die in parallel als beginconfiguraties te gebruiken. Elke keer dat een betere waarde voor `beste_waarde` wordt gevonden, wordt die aan alle computers doorgestuurd. In de meeste gevallen zal het zo zijn als bij het inpakprobleem – dus dat `beste_waarde` in vergelijking met het aantal toppen in de recursieboom niet vaak verbeterd kan worden. Bij het inpakprobleem kan `beste_waarde` ten hoogste $n-1$ keer verbeterd worden terwijl de recursieboom exponentieel groot kan zijn! De communicatie tussen computers die verschillende takken afwerken is dus **miniem** en het is zelfs geen probleem de betere waarden via het internet te sturen!
Dat ziet er al niet slecht uit – maar er zijn twee problemen:

a.) In de meeste gevallen zullen de takken allesbehalve even groot zijn. Sommige takken zullen zeer snel als slecht herkenbaar zijn (de bounding

criteria zijn in deze delen heel efficiënt) en dus nauwelijks werk vragen en sommige takken zullen bijna de hele tijd vragen. Als wij bv. 100 computers ter beschikking hebben en de taak op deze manier in honderd delen splitsen dan is het mogelijk dat 99 delen na weinige seconden gedaan zijn en dat één van de taken weken duurt...

b.) De verschillende startconfiguraties moeten berekend en op een manier opgeslagen worden dat de programma's vanuit dit punt kunnen beginnen. Dat is niet echt moeilijk, maar wij zullen zien dat het vaak overbodig werk is.

Probleem a.) kan gedeeltelijk opgelost worden door de afstand van de wortel van de recursieboom duidelijk groter te kiezen. Daardoor krijgen wij veel meer delen dan wij computers hebben en hoewel ook deze delen zeker verschillend veel tijd nodig hebben, kunnen wij hopen dat door het feit dat elke computer meerdere delen moet afwerken het verschil tussen de tijd die de computer nodig heeft die het langst bezig is (dat is onze wachttijd) en die het minst bezig is niet te groot wordt. Ten slotte kunnen computers die heel goedkope delen opstarten ook vroeger met het volgende deel beginnen en werken daardoor misschien meer delen af. Een probleem is natuurlijk dat wij misschien extreem veel delen hebben...
Branch and bound algoritmen zijn natuurlijk alleen duur als de bomen heel groot zijn. Dat betekent meestal niet dat de bomen heel diep zijn, maar dat er een grote vertakking is. Of met andere woorden: dat er op de lage niveaus (dus dicht bij de wortel) in vergelijking met de hele boom verwaarloosbaar weinig toppen zijn.
Om probleem b.) op te lossen zullen wij het gedeelte tot het gekozen niveau gewoon elke keer opnieuw doorlopen. Omdat het relatief klein is, verliezen wij op deze manier niet veel tijd. Het aantal gevallen mag natuurlijk niet te groot zijn! Om het aantal gevallen klein te houden en aan de andere kant een goede verdeling te krijgen, kiezen wij het level waar wij splitsen relatief diep in de boom (dus veel verschillende toppen om te splitsen), maar vatten wij toppen uit verschillende delen van de zoekboom samen. Of precies:

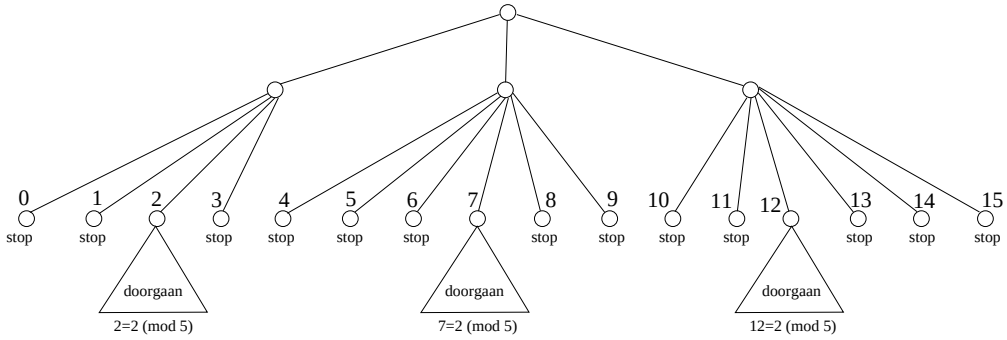
- Kies een diepte d om te splitsen en een aantal k van delen waarin het probleem gesplitst moet worden. Het aantal toppen van de recursieboom op diepte d moet zeer veel groter zijn dan k. De verschillende delen waarin de recursieboom opgesplitst wordt, kunnen nu beschreven worden door de indices $0, 1, \ldots, k-1$.

- Houd een teller bij die – beginnend met 0 – nummers toekent aan de toppen op diepte d in de volgorde waarop ze bezocht worden.

- Als deel nummer i moet afgewerkt worden en je hebt een top met nummer m op diepte d bereikt dan ga door naar de volgende diepte als en slechts als $i = m \pmod{k}$. Anders backtrack.

- Als de delen $0, 1, \ldots, k-1$ afgewerkt zijn (dat kan in parallel gebeuren) is de hele recursieboom doorzocht. Communicatie tussen de delen is alleen het mededelen van nieuwe bovengrenzen.

Deze manier om het doorzoeken van een recursieboom op te splitsen in deeltaken zien jullie in Figuur 26. Diepte 2 en maar 16 toppen op de diepte is niet echt realistisch (het moeten er **veel** meer zijn), maar realistische gevallen kan je lang niet meer tekenen...
Maar: je moet erop letten dat in het gedeelte dat in elk deel wordt doorlopen – dus de niveaus ten hoogste de diepte waar gesplitst wordt – niet gesnoeid wordt of altijd met dezelfde waarde voor de grens gesnoeid wordt – anders is de nummering van de toppen op het splitlevel verschillend in delen die snel klaar zijn en delen die laat klaar zijn en misschien al met een betere grens kunnen werken.
Maar let op: het argument dat computers die gemakkelijke delen draaien ook sneller een nieuw deel opstarten geldt hier niet meer – wij moeten gewoon hopen dat door het feit dat elk deel onderdelen bevat ook de moeilijke gevallen (de takken met grote complexiteit) goed verdeeld zijn.



Figuur 26: De recursieboom voor het geval dat op diepte $d = 2$ gesplitst wordt en deel nummer 2 voor $k = 5$ afgewerkt wordt. Voor toppen met een nummer dat gelijk is aan 2 (mod 5) wordt doorgegaan en voor de anderen gestopt.

Inderdaad is het niet echt nodig dat je atijd op dezelfde diepte splitst. Soms is het beter in sommige takken op een grotere diepte te splitsen dan in andere. De bedoeling is gewoon dat je een verzameling van toppen in de boom hebt die een relatief klein gedeelte bij de wortel splitst van een groot gedeelte en

waar je elke van deze splitstoppen vanuit het gedeelte dat altijd doorlopen wordt kan bereiken. Altijd op dezelfde diepte te splitsen is gewoon één van heel veel mogelijkheden die hier als voorbeeld werd gebruikt.
Over de tijd die nodig is om alle delen af te werken in vergelijking met de tijd als je het in één deel laat draaien, kan niet veel bewezen worden. Omdat de volgorde waarin de boom wordt doorlopen sterk verschilt van de volgorde waarin het in één deel wordt doorlopen – en dus de verschillende takken met andere waarden voor de bovengrens worden doorlopen, kan de som van de tijden theoretisch zelfs arbitrair veel kleiner zijn dan de tijd als je het in één deel laat draaien. Dat is geen strijdigheid met de observatie dat $t^s(n)$ in feite $O(p * t_p(n))$ is omdat $t^s(n)$ betrekking heeft op een optimaal algoritme. – en in dit geval zou een optimaal algoritme de takken in een andere volgorde doorlopen.
Aan de andere kant kan het theoretisch ook gebeuren dat elk deel bijna even veel tijd vraagt als de hele taak...
In de praktijk zal deze manier van opsplitsen een goede manier zijn om dingen in het parallel op te starten waarbij de som van de tijden ongeveer even groot is als de tijd als het serieel wordt opgestart.

Oefening 102 *Ons argument dat een serieel algoritme altijd ten hoogste p keer trager is dan een parallel algoritme was dat een serieel algoritme de parallelle berekeningen kan simuleren. Beschrijf* **expliciet** *wat dat voor dit verdeeld algoritme betekent. Hoe werkt dit serieel algoritme voor een gegeven p? Je mag veronderstellen dat p ook het aantal delen is – dus alle delen in het parallel worden opgestart.*

Oefening 103 *Voor het volgende mag je stellen dat als een optimum gevonden wordt het als resultaat heeft dat alle nog lopende delen onmiddellijk kunnen stoppen omdat de snoeiecriteria merken dat deze waarde niet verbeterd kan worden. Deze veronderstelling is in veel gevallen inderdaad realistisch (bv. als een langste cykel in een graaf gezocht wordt en een Hamiltoniaanse cykel wordt gevonden).*
Stel bovendien dat alle delen tegelijk worden opgestart.

- *Gegeven een constante $C \geq 1$. Schets een scenario (precies: de structuur van een recursieboom) waar voor een branch and bound algoritme dat op de geziene manier in delen wordt opgesplitst de som van de tijden C keer kleiner is dan de tijd als het in één deel wordt opgestart.*

- *Gegeven een constante $C \geq 1$. Schets een scenario (precies: de structuur van een recursieboom) waar voor een branch and bound algoritme*

dat op de geziene manier in delen wordt opgesplitst de som van de tijden C keer groter is dan de tijd als het in één deel wordt opgestart.

Oefening 104 *Herschrijf de pseudocode in Algoritme 15 zo dat het branch and bound algoritme op de net geziene manier opgesplitst wordt.*

Oefening 105 *Gegeven gewichten $g_1 \leq 1, \ldots, g_n \leq 1$. Gezocht is het minimale aantal vrachtwagens met capaciteit 1 dat nodig is om deze gewichten te vervoeren.*

- *Schrijf de pseudocode – met de nodige delen om het algoritme met de modulo optie te kunnen verdelen – voor een branch and bound algoritme dat de gewichten $g_1, \ldots, g_n$ in deze volgorde op elke mogelijke manier plaatst en de waarde b van een beste oplossing tot nu toe altijd bijhoudt.*

  Als bounding criterium gebruik het volgende: als de som k van alle gewichten die nog niet geplaatst zijn en op geen van de al gebruikte a vrachtwagens past, voldoet aan $a + k > b - 1$, kan je snoeien. Je mag veronderstellen dat een functie `get_k()` *al bestaat die de waarde van k teruggeeft.*

- *Geef ook een andere manier* splitstoppen *te bepalen dan altijd op dezelfde diepte te splitsen. Houdt er bv. rekening mee hoeveel vrachtwagens al gebruikt zijn.*

Oefening 106 *Stel dat je pas nadat al een groot deel van de taken afgewerkt is, merkt dat één deel toch veel langer draait dan alle anderen. Geef een oplossing hoe je deze taak verder kan opsplitsen zonder de andere delen die al afgewerkt zijn opnieuw op te starten.*

Oefening 107 *Hetzelfde principe kan ook gebruikt worden voor het doorzoeken van recursiebomen waar geen optimum gezocht wordt, maar waar output op verschillende levels van de zoekboom gebeurt. Waarop moet je hier letten om nog altijd dezelfde verzameling van outputs te hebben en hoe los je het probleem op?*

5.2 Op weg met MPI

Terwijl de rest van het hoofdstuk over parallele algoritmen vrij theoretisch is en (nog) niet praktisch toepasbaar, zullen wij het hier over iets hebben dat wel heel belangrijk is voor toepassingen.

De bedoeling van dit hoofdstuk is niet, een echte inleiding te geven in MPI. De bedoeling is jullie op weg te helpen, MPI in C-programma's te gebruiken – de rest van de weg kunnen jullie dan gemakkelijk zelf vinden. MPI is ook geen algoritme, zodat een uitgebreide inleiding hier ook niet op de juiste plaats zou zijn. MPI (Message Passing Interface) is een bibliotheek van functies die jullie kunnen gebruiken om verschillende processen met elkaar te laten communiceren. MPI bestaat niet alleen voor C, maar onze voorbeelden hier zullen allemaal voor C zijn. Veel van de theoretische modellen die wij later zullen zien, kunnen *in principe* met MPI geïmplementeerd worden – maar in de meeste gevallen niet met de in de theoretische modellen veronderstelde complexiteit. Het grote voordeel van MPI is, dat het een standaard is. De MPI-1 standaard werd al in 1994 vastgelegd en de biblioteek staat ondertussen voor alle belangrijke platformen ter beschikking. Een met MPI geparallelliseerd algoritme is dus portable als je buiten de MPI-functies geen syteemspecifieke dingen doet: jullie kunnen het algoritme op een laptop ontwikkelen en zonder wijzigingen bv. op een linux-cluster laten draaien of op een machine met een groot aantal cores. Het is de taak van de MPI-ontwikkelaars de topologie van het netwerk optimaal te gebruiken
Ook al zijn de programma's portable – de manier waarop de programma's **opgestart** moeten worden, kan verschillen. Die hangt er ook van af, of bv. rekening moet gehouden worden met vrije processoren, queues, etc. In deze lesnota's wordt de manier gegeven die je op één linux computer kan gebruiken (nadat je MPI geïnstalleerd hebt).

5.2.1 MPI opstarten

Maar het beste is als wij onmiddelijk beginnen. Kijk naar het volgende eenvoudige programma:

Programma 1

```
// programma simpleworld.c
#include <mpi.h>
#include <stdio.h>
#include <unistd.h>

int main (int argc, char* argv[])
{
  MPI_Init(&argc, &argv);      /* starts MPI */
  sleep(10);                    /* body */
  printf( "Hello world!\n");    /* body */
  MPI_Finalize();               /* closes MPI */
```

```
}
```

Belangrijk is hier de `#include <mpi.h>` in de header die ervoor zorgt dat alle MPI-functies gekend en goed gedefinieerd zijn. Elk MPI-programma start bovendien met `MPI_Init(&argc, &argv)` en stopt met `MPI_Finalize()`. Dit programma kan je nu met `mpicc -o simpleworld simpleworld.c` compileren. Als jullie dat nu parallel met 4 processen willen laten draaien, dan kan dat met `mpirun -n 4 simpleworld`. Daarbij staat `-n x` ervoor dat je `x` processen wilt opstarten. Deze processen worden afhankelijk van waar je werkt, op dezelfde computer opgestart of op verschillende andere computers. Als je het op een alleenstaande Linux-computer opstart, dan geeft de `sleep(10)` je voldoende tijd om te zien dat er daadwerkelijk 4 processen `simpleworld` aan het draaien zijn. Na 10 seconden zie je dan het resultaat: 4 lijnen met `Hello world!`. In oude implementaties van MPI kan het gebeuren dat niet alle processen *stdout* en *stderr* kunnen gebruiken, maar "meestal" zullen *stdout* en *stderr* naar de *stdout* en *stderr* van het proces *mpirun* op de masternode omgeleid worden. Ook al is de kans klein dat jullie ooit een implementatie tegenkomen die dat niet doet: zelfs in de MPI 3 standaard is niet vastgelegd dat elke geldige implementatie dat **moet** doen. Wie gegarandeerd volledig portable programma's wil schrijven, stuurt best alle data naar het root proces (welk proces dat is, zullen wij nog zien) en laat dat de output doen.
Bijna alle MPI-functies geven een (int-) errorwaarde terug – ook `MPI_init` en `MPI_Finalize`. Hier willen wij het programma zo eenvoudig mogelijk houden, maar een goed programma zou deze waarde natuurlijk het best toetsen, bv met

```
resultaat = MPI_Init (&argc, &argv);
if (resultaat != MPI_SUCCESS).....
```

waarbij `MPI_SUCCESS` een MPI-constante is die aangeeft dat er geen fout opgetreden is. Het testen van errorcodes is belangrijk voor een vaak gebruikt programma, maar hier zou het gewoon afleiden van de hoofdzaak. De programma's hier zijn ten slotte alleen maar bedoeld om een beetje te verstaan wat er gebeurt.
Dat is al een begin, maar natuurlijk niet echt boeiend. Er is bv. nog helemaal geen communicatie en alle processen doen hetzelfde. Om ervoor te zorgen dat verschillende processen verschillende dingen kunnen doen, moeten de processen een soort id hebben op basis waarvan ze kunnen beslissen wat ze moeten doen. Een eerste aanzet zou kunnen zijn met `getpid()` de proces id te gebruiken, maar als de processen op verschillende machines draaien,

zouden die zelfs voor verschillende processen identiek kunnen zijn en bovendien zouden we niet op voorhand weten, welke nummers toegewezen worden. Dat deugt dus niet.
MPI gebruikt het concept van een communicator, die een groep van processen voorstelt die met elkaar kunnen communiceren. Een communicator is als datatype door middel van `typedef` gedefinieerd. Het datatype heet `MPI_Comm` en een belangrijke **constante** met dit datatype is `MPI_COMM_WORLD` – de constante groep die alle processen bevat die in het begin worden opgestart. Inderdaad kan het aantal processen op een dynamische manier groeien. Als dat gebeurt, beschrijft de constante `MPI_COMM_WORLD` niet meer alle processen en kunnen de verschillende nieuwe processen zelfs disjuncte verzamelingen als *world* beschrijven. In deze mini-inleiding zal het aantal processen altijd constant blijven.
Belangrijke functies voor een dergelijke communicator die jullie zeker nodig zullen hebben, zijn:

```
MPI_Comm_rank (MPI_Comm communicator, int *index);
/* bepaal de index van dit proces in "communicator"*/

MPI_Comm_size (MPI_Comm communicator, int *grootte);
/* bepaal het aantal processen in "communicator"*/
```

Ook deze functies geven een errorcode terug. Daarom moeten de adressen van de integer variabelen waar de opgevraagde waarden naartoe geschreven moeten worden, meegegeven worden. De index in een communicator is een goede id voor het proces die we kunnen gebruiken om te beslissen wat een proces doet. Als er k processen in een communicator zijn, dan zijn de indexen $0, \ldots, k-1$. De index 0 in `MPI_COMM_WORLD` is een bijzonder proces dat ook het root proces wordt genoemd. In `Open_MPI` en andere implementaties volgens MPI 2 standaard wordt de `stdin` van `mpirun` aan dit proces doorgegeven en alle andere processen hebben `/dev/null` als `stdin`. Wij zullen data altijd van `stdin` lezen (bv. met `fread()` of `scanf()`) maar voor de duidelijkheid in het programma er expliciet voor zorgen dat alleen proces 0 probeert te lezen.
Met deze functies kunnen wij ons programma al een beetje interessanter maken:

Programma 2

```
#include <mpi.h>
#include <stdio.h>
```

```
void zeg_niets()
{ printf("Ik zeg niet wie ik ben!\n"); }

void zeg_iets(int mijn_id, int totaal)
{  printf("Ik ben proces nummer %d van totaal %d !\n",mijn_id,totaal); }


int main (int argc, char* argv[])
{
  int mijn_id, grootte, i;

  MPI_Init (&argc, &argv);                        /* starts MPI */
  MPI_Comm_rank (MPI_COMM_WORLD, &mijn_id);       /* vraag mijn nummer op */
  MPI_Comm_size (MPI_COMM_WORLD, &grootte);       /* hoeveel processen zijn er? */
  if (mijn_id%2) zeg_niets();
  else zeg_iets(mijn_id,grootte);
  MPI_Finalize();
}
```

Als wij dat compileren en met `mpirun -n 6` laten draaien, **kan** de output bv als volgt zijn:

```
Ik ben proces nummer 0 van totaal 6 !
Ik zeg niet wie ik ben!
Ik zeg niet wie ik ben!
Ik ben proces nummer 4 van totaal 6 !
Ik zeg niet wie ik ben!
Ik ben proces nummer 2 van totaal 6 !
```

Alle processen draaien tegelijk en welk proces eerst zijn resultaat doorgeeft, is niet vastgelegd. De verschillende MPI-processen hebben geen synchronisatie (als die niet door zekere communicaties wordt opgelegd) en draaien onafhankelijk van elkaar. Inderdaad kan de output van een proces zelfs onderbroken zijn door andere output. Een goed programma zou er dus het best zelf voor zorgen dat de output door één proces gesynchroniseerd wordt. Maar dat kunnen wij op dit ogenblik nog niet. Wij kunnen wel al verschillende processen verschillende dingen laten doen – gebaseerd op hun index in een communicator.
Maar toch al is er nog geen communicatie...

Oefening 108 *Schrijf een MPI-programma waar elk proces een lijst van identificaties van de andere processen uitvoert met wie het via de* `MPI_COMM_WORLD` *communicator kan communiceren.*

MPI datatype	C datatype
MPI_CHAR	char
MPI_SHORT	short int
MPI_INT	int
MPI_LONG	long int
MPI_LONG_LONG_INT	long long int
MPI_SIGNED_CHAR	char
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_UNSIGNED_LONG_LONG	unsigned long long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double

Tabel 1: Sommige elementaire datatypes en de bijbehorende C datatypes.

5.2.2 Berichten sturen en ontvangen

De meest eenvoudige vorm van communicatie is de communicatie tussen twee vastgelegde processen (vaak point-to-point communicatie genoemd). Daarover zullen wij het dus eerst hebben.
Een bericht heeft twee belangrijke delen: de enveloppe en de inhoud. De enveloppe bevat informatie over waar het bericht naartoe gaat, hoe groot die is, etc. en de inhoud is dat wat je echt wilt sturen. In principe is de inhoud altijd een array, maar MPI geeft je ook de mogelijkheid afgeleide datatypes te definiëren – net zoals structs in C. Wij zullen alleen met de elementaire datatypes werken, waarvan we de belangrijkste in tabel 1 geven.
Een typische inhoud van een bericht is dus bv. een array van ints. Belangrijk is daarbij dat de inhoud tijdens het sturen soms *vertaald* moet worden. Als je bv. op een cluster werkt waar zowel Little-Endian als Big-Endian machines zijn en je stuurt een int, dan wil je dat de machine die het bericht ontvangt hetzelfde getal ontvangt – MPI zal de volgorde zo wijzigen dat de int op de andere machine hetzelfde getal voorstelt. Moeilijker wordt het als de datatypes verschillende groottes hebben – dan moet MPI soms afronden. Dat zijn allemaal problemen die MPI voor je oplost – omdat je niet op voorhand weet op welk soort machine welk proces draait, **kan** je de conversies niet zelf doen.
Een manier voor het sturen van point-to-point communicatie is de functie

```
int MPI_Send(void *buf, int aantal, MPI_Datatype type,
             int ontvanger, int kenmerk, MPI_Comm comm)
```

De inhoud van ons bericht is beschreven door de 3 variabelen `buf`, `aantal` en `type`, waarbij je `aantal`, en `type` als deel van de enveloppe kan rekenen, omdat het de inhoud beschrijft maar niet de inhoud is. In deze functie is `*buf` een pointer naar de array die de inhoud bevat die we willen sturen, `aantal` het aantal elementen en `type` het datatype van de elementen van de array.
De variabele `ontvanger` is het nummer van het proces in de communicator `comm` die het bericht moet ontvangen. Als laatste deel van de enveloppe heb je nog de variabele `kenmerk` die je als een identificatie van het bericht kan zien. Als een proces p_0 bv. meerdere berichten aan hetzelfde proces p_1 stuurt, dan kunnen die door de kenmerken nog herkend worden en de ontvanger kan precies die berichten ophalen die hij op dat moment nodig heeft.
Aan de andere kant moet het bericht ook ontvangen worden. Daarvoor heb je de functie

```
int MPI_Recv(void *buf, int aantal, MPI_Datatype type, int zender,
             int kenmerk, MPI_Comm comm, MPI_Status *status)
```

Als er een bericht is van het proces met nummer `zender` in de communicator `comm` met identificatie `kenmerk`, dan worden `aantal` items van type `type` naar `buf` geschreven. Het is **jouw** taak om het programma zo te schrijven dat `aantal` en `type` overeenkomen met de variabelen in de functie `MPI_Send()` die het bericht met deze identificatie heeft gestuurd. Als dat niet zo is, gebeurt er een fout, die soms door MPI herkend kan worden (bv. *truncated message* – te weinig bytes gelezen) maar niet altijd.
In de struct status heb je altijd `MPI_SOURCE` – het nummer van het proces dat het bericht heeft gestuurd, `MPI_TAG` – de identificatie van het bericht en `MPI_ERROR` – de error status. Op het eerste gezicht lijkt een dergelijke struct overbodig omdat het alleen maar informatie bevat die we al kennen, maar je kan ook wildcards gebruiken voor `zender` (`MPI_ANY_SOURCE`) en `kenmerk` (`MPI_ANY_TAG`) en dan kan je met de status-struct deze informatie uitvissen. Een wildcard `MPI_ANY_DEST` bestaat **niet**. Als je een bericht stuurt, moet dus duidelijk zijn waar dat naartoe gaat.
De werking kan bv. in het volgende programma gezien worden:

Programma 3

```
#include <mpi.h>
#include <stdio.h>
```

```
#include<stdlib.h>

void proc(int id)
{
  int number, i;
  MPI_Status status;

  if (id==1)
  { i=0;
    do
     {
     MPI_Recv((void *)&number, 1, MPI_INT,0, i, MPI_COMM_WORLD,&status);
     fprintf(stderr,"Process %d received number %d with tag %d \n",
                     id,number,status.MPI_TAG);
     i= (i+2)%5;
     }
     while (i!=0);
  }
  else fprintf(stderr,"Process %d has nothing to do.\n",id);
  return;
}

void proc0(int id)
{
  int i,j;

  for (i=0;i<5;i++)
    { j=3*i;
      MPI_Send((void *)&j, 1, MPI_INT,1, i, MPI_COMM_WORLD);
      fprintf(stderr,"Process %d sent number %d\n",id,j);
    }
  return;
}

int main (int argc, char* argv[])
{
  int myrank;

  MPI_Init (&argc, &argv);
  MPI_Comm_rank (MPI_COMM_WORLD, &myrank);
```

```
  if (myrank==0) proc0(myrank);
  else proc(myrank);

  MPI_Finalize();
  return 0;
}
```

De rare vorm waarop de do-while lus in dit voorbeeld wordt doorlopen, is om te zien dat de berichten inderdaad niet op een FIFO of LIFO manier aan een queue voor een proces worden toegevoegd, maar op basis van de identificatie opgehaald kunnen worden. Als je dit programma met 3 processen opstart, is de output dus bv. :

```
Process 2 has nothing to do.
Process 0 sent number 0
Process 0 sent number 3
Process 0 sent number 6
Process 0 sent number 9
Process 0 sent number 12
Process 1 received number 0 with tag 0
Process 1 received number 6 with tag 2
Process 1 received number 12 with tag 4
Process 1 received number 3 with tag 1
Process 1 received number 9 with tag 3
Process 3 has nothing to do.
```

De volgorde van de output van de verschillende processen kan verschillend zijn (bv. `Process 2 has nothing to do.` hoeft niet het eerste bericht te zijn, maar kan ook later komen), maar voor hetzelfde proces zal de volgorde van berichten altijd dezelfde zijn. Een proces kan dus meerdere berichten hebben die wachten om opgehaald te worden.

Oefening 109 *Schrijf een programma voor 3 processen dat als volgt werkt: De processen 1 en 2 genereren toevallige getallen met* `random()`*, dat geïnitialiseerd wordt met* `srandom(time(0)+process_id)`*. Reeksen van oneven getallen worden in een array bijgehouden en zodra een even getal opduikt, geldt de reeks als afgesloten. Als de reeks niet leeg is, wordt hij naar proces 0 gestuurd. Als de reeks van getallen dus* $4, 1, 6, 8, 5, 7, 9, 8, 11 \ldots$ *is, wordt dus eerst een array met alleen maar* 1 *doorgestuurd, dan een array met* $5, 7, 9$*, etc.*
Proces 0 *leest afwisselend de berichten van proces* 1 *en* 2 *en houdt voor beide processen 1 en 2 bij wat de som van de lengten van de doorgestuurde reeksen*

is en wat de langste reeks was. Zodra één van de processen in totaal 1.000.000 *getallen heeft doorgestuurd, schrijft proces 1 de langste reeks van dit proces uit en het programma stopt.*
Let op: proces 0 weet op voorhand niet hoe lang de reeks van oneven getallen is, je moet er dus voor zorgen dat hij weet hoeveel hij moet lezen. Als proces 0 merkt dat het resultaat gevonden is, gaan de andere twee processen nog altijd door. Zorg ervoor dat die ook stoppen.

Beide functies – `MPI_Send()` en `MPI_Recv()` moeten eerst *afgewerkt* zijn voordat het programma dat ze gebruikt kan doorgaan – ze blokkeren dus de voortgang van het programma en kunnen tot een deadlock leiden. Het is dus belangrijk precies te weten wat *afgewerkt* in dit geval betekent, dus **wanneer** een oproep van deze functies *afgewerkt* is. Inderdaad kan ook Programma 3 in principe in een deadlock terechtkomen – zie Oefening 110.
Wat gebeurt er bv. als je het volgende programma met 3 processen opstart?

Programma 4

```
#include <mpi.h>
#include <stdio.h>

int main (int argc, char* argv[])
{
  int mijn_id, grootte, i, message[1000], message2[1000];
  MPI_Status status;

  MPI_Init (&argc, &argv);      /* starts MPI */
  MPI_Comm_rank (MPI_COMM_WORLD, &mijn_id);
  MPI_Comm_size (MPI_COMM_WORLD, &grootte);

  for (i=0;i<1000;i++) message[i]=mijn_id;

  MPI_Send((void *)message, 1000, MPI_INT, (mijn_id+1)%grootte,
           1,MPI_COMM_WORLD);
  MPI_Recv((void *)message2, 1000, MPI_INT,MPI_ANY_SOURCE,
           MPI_ANY_TAG, MPI_COMM_WORLD,&status);

  fprintf(stderr,"Process %d received list of numbers %d with tag %d \n",
          mijn_id,message2[0],status.MPI_TAG);

  MPI_Finalize();
  return 0;
```

```
}
```

Het resultaat op mijn computer (met 3 processen) was:

```
Process 0 received list of numbers 2 with tag 1
Process 1 received list of numbers 0 with tag 1
Process 2 received list of numbers 1 with tag 1
```

Alles lijkt dus te draaien zo als gehoopt, maar op een andere computer kan het gebeuren dat het programma in een deadlock raakt. Zelfs op een computer waar het lijkt te werken, kan het gebeuren dat het niet meer draait als je meer data stuurt dan maar 1000 integers. De reden daarvoor is de manier waarop het versturen werkt. Voor het ontvangen (`MPI_Recv()`) is het duidelijk wat er gebeurt: het programma wacht totdat het opgevraagde bericht er is, leest het gegeven aantal items (als dat kan – anders is er een fout) en dan is de functie afgewerkt en het programma kan doorgaan.
Voor het sturen (`MPI_Send()`) is dat niet zo duidelijk. Wanneer is de functie klaar – dus wanneer is een bericht *gestuurd*? MPI kan (en zal) dat op meerdere manieren doen. MPI kan het bericht in een eigen buffer schrijven, wat heel snel kan gebeuren. Zodra MPI het bericht in zijn eigen buffer heeft, is `MPI_Send()` gedaan en kan het programma dat iets wil sturen doorgaan. Maar als MPI niet voldoende eigen buffer heeft, kan het ook wachten totdat het proces dat het bericht wil ontvangen dat afhaalt en dan de inhoud van de array zonder bufferen van het ene proces naar het andere schrijven. Dan kan het dus gebeuren dat een proces heel lang op een andere computer moet wachten – en zelfs erger: je kan een deadlock krijgen. Als niet gebufferd wordt – als designbeslissing van de speciale MPI-implementatie of omdat er te veel data gestuurd wordt – zullen in Programma 4 alle processen tegelijk beginnen te schrijven en zolang de andere processen het bericht niet ophalen, zal geen van de processen doorgaan. Maar omdat **alle** processen wachten, zal geen proces iets ophalen en ontstaat er een deadlock...

Oefening 110 *Beschrijf waar en hoe Programma 3 in principe in een deadlock terecht kan komen.*

Oefening 111 *Schrijf een MPI-programma dat dezelfde berichten aan dezelfde processen stuurt als Programma 4, maar waar er verzekerd is dat geen deadlock kan ontstaan.*
Bespreek ook (informeel) de tijd die Programma 4 (als er geen deadlock is) en jouw programma nodig hebben (zowel de tijd per proces als ook de totale tijd vanaf het opstarten totdat alle processen klaar zijn).

Oefening 112 *Implementeer een – wel niet erg snelle – manier om* $\sum_{i=1}^{n} i$ *te berekenen:*
Je werkt met $n+1$ *processen. Proces* n *stuurt zijn* id *aan proces* $n-1$*. De processen* $k \in \{1, \ldots n-1\}$ *ontvangen een getal van proces* $k+1$*, tellen hun* id *erbij op en sturen het resultaat naar proces* $k-1$*. Ten slotte ontvangt proces* 0 *een getal van proces* 1 *en schrijft die als resultaat.*

Oefening 113 *Schrijf een MPI-programma voor mergesort. Proces 0 leest een reeks van getallen van* `stdin` *in en slaat die in een array op. De getallen staan binair in een bestand dat je door middel van een pipe als* `stdin` *van* `mpirun` *gebruikt. Als je k processen hebt, wordt de array eerst in k delen van ongeveer gelijke grootte opgedeeld en worden die aan de verschillende processen doorgestuurd (proces 0 houdt natuurlijk één). De verschillende processen sorteren de delen en sturen die door naar andere processen om te mergen, zodat elk proces ten hoogste 2 delen heeft om te mergen. Als ten slotte nog maar 1 proces bezig is (zorg ervoor dat dat proces 0 is), merget die zijn twee delen en schrijft het resultaat naar* stdout.

5.2.3 Berichten sturen en ontvangen – zonder de voortgang te blokkeren

Ook als de communicatie goed georganiseerd is, zit je bij de methode zoals wij die tot nu toe gezien hebben met een probleem: soms weet je gewoon niet of er een bericht voor je is! Een typisch geval zijn onze verdeelde branch and bound algoritmen: daar weet een proces dat er een betere grens van een ander proces *zou kunnen zijn* – maar misschien is die er ook niet, en als die er niet is, kan de ontvanger zonder problemen doorgaan. Als je voor een dergelijke toepassing `MPI_Recv()` zou gebruiken, zou je niet alleen tijd verspillen als er geen bericht klaarligt – het zou zelfs kunnen dat het proces nooit stopt omdat er geen bericht komt.
Voor dergelijke toepassingen heb je de functies `MPI_Isend()` en `MPI_Irecv()` waarbij de "I" voor *initiate* staat. Beide functies starten een communicatieoperatie en keren terug nog voordat die afgewerkt is. Om later naar deze opdracht te kunnen refereren, geven ze een MPI handle terug. De manier waarop de functies opgeroepen worden is als volgt:

```
int MPI_Isend(void *buf, int aantal, MPI_Datatype type,
              int ontvanger, int kenmerk, MPI_Comm comm,
              MPI_REQUEST *opdrachtnummer)

int MPI_Irecv(void *buf, int aantal, MPI_Datatype type, int zender,
              int kenmerk, MPI_Comm comm,
```

```
                MPI_REQUEST *opdrachtnummer)
```

De variabele `opdrachtnummer` is hier de plaats waar de MPI handle wordt naartoegeschreven. De functies lijken – op de MPI handle na – sterk op de blokkerende functies. In `MPI_Irecv()` ontbreekt wel de variabele `status`, maar die kan natuurlijk pas ingevuld worden als de ontvangsoperatie afgewerkt is.
Of een operatie klaar is, kan je testen met de functie `MPI_Test()` die de volgende parameters nodig heeft:

```
int MPI_Test(MPI_Request *opdracht, int *flag, MPI_Status *status)
```

Als `flag` *true* is, is de opdracht met handle **opdracht* klaar – anders niet. Als de opdracht klaar is, staat in `status` wat je ook na een `MPI_Recv()` in `status` zou vinden . Na een `MPI_Isend()`-opdracht **kan** in `status` de errorcode van de `MPI_Isend()`-operatie staan.
De werking kan in het volgende programma gezien worden:

Programma 5

```
#include <mpi.h>
#include <stdio.h>
#include<stdlib.h>
#include <unistd.h>

void proc(int id)
{
  int number, gedaan,seconden=0;
  MPI_Status status;
  MPI_Request opdracht;

  MPI_Irecv((void *)&number, 1, MPI_INT,MPI_ANY_SOURCE, MPI_ANY_TAG,
            MPI_COMM_WORLD,&opdracht);

  for (gedaan=0; !gedaan; )
    { MPI_Test(&opdracht,&gedaan,&status);
      if (gedaan)
         fprintf(stderr,"Proces %d heeft bericht van %d ontvangen \n",
                 id,status.MPI_SOURCE);
      else { sleep(5); seconden+=5;
        fprintf(stderr,"Proces %d heeft  %d seconden gewacht!\n",id, seconden);
      }
```

```
    }
  return;
}

void proc0(int id, int total)
{
  int number=1, gedaan;
  MPI_Status status;
  MPI_Request opdracht;

  if (total>1)
    { sleep(13);
      MPI_Isend((void *)&number, 1, MPI_INT,1, 0, MPI_COMM_WORLD,&opdracht);
      MPI_Test(&opdracht,&gedaan,&status);
      if (gedaan) fprintf(stderr,"Proces %d heeft nummer %d onmiddelijk gestuurd!\n
                          id,number);
    }
  return;
}

int main (int argc, char* argv[])
{
  int myrank, size;

  MPI_Init (&argc, &argv);
  MPI_Comm_rank (MPI_COMM_WORLD, &myrank);
  MPI_Comm_size (MPI_COMM_WORLD, &size);

  if (myrank==0) proc0(myrank,size);
  else proc(myrank);

  MPI_Finalize();
  return 0;
}
```

Opgestart met 3 processen zou de output bv. als volgt kunnen zijn:

```
Proces 1 heeft  5 seconden gewacht!
Proces 2 heeft  5 seconden gewacht!
Proces 1 heeft  10 seconden gewacht!
Proces 2 heeft  10 seconden gewacht!
```

```
Proces 0 heeft nummer 1 onmiddelijk gestuurd!
Proces 1 heeft  15 seconden gewacht!
Proces 2 heeft  15 seconden gewacht!
Proces 1 heeft bericht van 0 ontvangen
Proces 2 heeft  20 seconden gewacht!
Proces 2 heeft  25 seconden gewacht!
.
.
.
```

Proces 1 en 2 initialiseren een ontvangstopdracht en gaan – zoals de output toont – door met hun werk als het bericht er nog niet is. In dit geval zou proces 2 oneindig doorgaan omdat er natuurlijk geen bericht komt...
Belangrijk is dat een proces dat `MPI_Isend()` gebruikt de variabele waarin de inhoud is opgeslaan niet wijzigt voordat gedetecteerd is dat het versturen klaar is. Pas als `MPI_Test()` zegt dat het proces klaar is mag die gewijzigd worden – anders kan een fout optreden.

De `MPI_I...()` routines hebben duidelijk voordelen tegenover de blokkerende routines: Ze kunnen alles wat de blokkerende routines kunnen en meer. Bovendien heb je gegarandeerd geen deadlock. Aan de andere kant zijn programma's met de niet blokkerende routines moeilijker te lezen en te verstaan, omdat het effect van het sturen en ontvangen niet zo duidelijk lokaal afgebakken is – bv. dezelfde ontvangstoperatie kan in verschillende functies getoetst moeten worden.

5.2.4 Verder met MPI?

Dit waren slechts de beginselen van MPI, maar met deze weinige basisoperaties kan je al zinvolle en nuttige programma's schrijven. De gehele functionaliteit van MPI is natuurlijk veel groter, maar nu jullie op weg zijn, kunnen jullie zeker zelf verder gaan. Andere nuttige functies (die je wel met de al geziene functies ook zelf kan implementeren) zijn bv. `MPI_Bcast()` (*broadcasting*), `MPI_Gather()` (*gathering*) of `MPI_Scatter()` (*scattering*).

5.3 De modellen voor parallel computing

In dit deel zullen wij nu sommige theoretische modellen voor computers zien die – dat moet wel toegegeven worden – op dit moment (nog?) niet als hardware verwezenlijkt zijn. Maar zoals al in het begin gezegd, is niet te voorspellen of de multicore processoren de parallelle algoritmen niet nieuw

leven inblazen en net dit deel voor jullie in toekomst bijzonder nuttig zal zijn. . . Zeker is dat er veel nieuwe ideeën voor parallelle algoritmen ontwikkeld werden. En ideeën zijn natuurlijk altijd interessant!
De modellen waarover wij het hier hebben, zijn wiskundige modellen voor parallelle computers. Terwijl er voor de gewone computer een door iedereen aanvaard model bestaat – het RAM-model – bestaan voor parallelle computers meerdere verschillende modellen en sommige algoritmen zijn alleen bedoeld voor één model en werken niet op andere modellen zonder belangrijke wijzigingen.
Wij bespreken alleen maar heel kort modellen waar de processoren deel uitmaken van een netwerk dat beschrijft welke processoren met elkaar kunnen communiceren en welke dat niet kunnen. Dergelijke modellen waren een tijdje geleden heel populair en in de vorm van transputers bestonden die zelfs als hardware. Transputers konden zelfs zo geconfigureerd worden dat je een netwerk hebt dat speciaal voor het algoritme dat erop moet draaien geschikt is. Maar er was ook veel onderzoek over netwerken met goede algemene eigenschappen – bv. de *hypercubes* – en over het simuleren van algemene netwerken op specifieke netwerken.
Wij zullen het hier vooral over variaties van het eenvoudigste – en misschien belangrijkste – model hebben. Het is de meest directe veralgemening van het RAM-model en heet **P**arallel **R**andom **A**ccess **M**achine – PRAM.
In dit model heb je gewone processoren die je bv. door een nummer kan identificeren – dus bv. processoren $p_1, \ldots, p_k$. Elke processor heeft zijn eigen programma (dat heet *Multiple Instruction*). Sommige algoritmen gebruiken ook het *Single Instruction* model waar alle processoren hetzelfde programma moeten gebruiken, maar wij veronderstellen gewoon het algemenere concept van Multiple Instruction waar de programma's van de processoren gelijk mogen zijn maar het niet vereist is. De processoren zijn bovendien gesynchroniseerd – dat betekent dat je op verschillende punten van een programma mag veronderstellen dat de processoren van dit punt tegelijk vertrekken. Als je bv. een lus in de pseudocode hebt

```
for i=1 to 20 pardo { do_iets(i) }
```

(met `pardo` voor *"do in parallel"*) dan betekent dat dat voor $1 \leq i \leq 20$ processor p_i de taak `do_iets(i)` afwerkt en pas als alle processoren klaar zijn doorgegaan wordt – ook als de taak voor verschillende i langer duurt dan voor anderen. Als verschillende processoren dezelfde lijnen in een dergelijke lus afwerken wordt normaal ook verondersteld dat ze op hetzelfde moment met de lijnen beginnen – dat er dus ook synchronisatie binnen in de lus is. In ons geval zullen dat altijd maar sommige lijnen zijn en het zal duidelijk zijn waar de synchronisatie moet gebeuren.

Als in de pseudocode zoiets staat als

```
for i=1 to log n pardo { do_iets(i) }
```

dan betekent dat niet dat er een centrale processor is die de andere processoren bestuurt maar dat de individuele processoren – die hun eigen nummer kennen – alleen dan iets doen als hun nummer ten hoogste $\log n$ is. De volgende lijn is dus bv. deel van de individuele programma's van de processoren – waarbij i het nummer van de processor is en wij ervoor moeten zorgen dat de processor n ook kent:

```
if (i<= log n) { do_iets(i) }
```

De processoren hebben een gemeenschappelijk geheugen dat ze allemaal kunnen gebruiken om in constante tijd te lezen en te schrijven. Je kan ook veronderstellen dat elke processor bovendien nog lokaal geheugen heeft maar dat komt natuurlijk op hetzelfde neer als globaal geheugen waarvan een deel alleen door de ene processor gebruikt wordt. Maar als meerdere processoren hetzelfde geheugen gebruiken is er natuurlijk een probleem: wat als meerdere processoren **tegelijk** iets willen lezen of schrijven? Daarvoor bestaan verschillende modellen. Eén model verbiedt gelijktijdig te lezen/schrijven – dat heet Exclusive Read resp. Exclusive Write en wordt met ER resp. EW afgekort. Een ander model laat het toe gelijktijdig te lezen/schrijven – dat heet dan Concurrent Read resp. Concurrent Write en wordt CR resp. CW afgekort.
Maar voor CW is er nog een probleem: wat gebeurt als meerdere processoren tegelijk *verschillende* waarden in een variabele willen schrijven? Wij leggen vast dat het dan toevallig is welke waarde in de variabele terechtkomt. (Maar er bestaan ook modellen die dat anders vastleggen...) Het algoritme moet dus correct werken om het even welke processor erin slaagt zijn waarde te schrijven. In het geval van ER of EW is het de verantwoordelijkheid van het algoritme ervoor te zorgen dat nooit verschillende processoren dezelfde variabele tegelijk willen lezen of er naartoe willen schrijven. Een algoritme dat dat niet garandeert is gewoon ongeldig!
Wij hebben dus 4 verschillende modellen van PRAMs: EREW-PRAM, ERCW-PRAM, CREW-PRAM en CRCW-PRAM.
Maar in alle modellen moet gegarandeerd zijn dat in dezelfde stap nooit één processor een variabele leest en één processor in dezelfde variabele schrijft. Het probleem kunnen jullie in Algoritme 16 zien.

Algoritme 16

x=0; a=0;

for i=1 to 2 pardo
if (i=1) x=1; else a=x;

De stappen `x=1` en `a=x` worden tegelijk uitgevoerd. De vraag is dus of in deze *stap* op het moment dat `x` wordt gelezen de waarde al gewijzigd is of niet... Natuurlijk kan je de modellen ook aanpassen door toe te laten dat gelijktijdig gelezen en geschreven *mag* worden maar te eisen dat het resultaat van het algoritme daarvan niet mag afhangen. In onze algoritmen zullen wij er gewoon altijd op letten dat nooit tegelijk (dat is: in dezelfde *stap* – waarbij *stap* natuurlijk niet echt precies gedefinieerd is) uit een variabele gelezen en in dezelfde variabele geschreven wordt.
Maar misschien is het het beste nu eens naar voorbeeldalgoritmen te kijken. Het eerste voorbeeld kopieert een waarde `a` zodat deze waarde achteraf `n` keer in een array `copy[]` staat. In feite kan de waarde zelfs in meer vakken staan – precies $2^{\lceil \log n \rceil}$. Dit aantal wordt ook in de array `aantal[]` teruggegeven – in $2^{\lceil \log n \rceil}$ kopieën. Het algoritme is geschikt voor een EREW-PRAM en draait in tijd $O(\log n)$. Wij schrijven $[n/2] = 2^{\lceil \log(n/2) \rceil}$:

Algoritme 17 *Waarden kopiëren in een EREW-PRAM*

```
copywaarden(a,n,copy[],aantal[])

{
  for j=1 to [n/2] pardo  n_array[j]= 0;
   // een array om n voor alle n processoren tegelijk leesbaar te maken

  copy[1]=a;
  n_array[1]=n;
  aantal[1]=1; // een hulparray dat zegt hoeveel kopie\"en er al zijn

  for (i=1; i<n; i=2*i) // deze lijn is heel informeel. Voor de precieze
                        // vorm kijk naar het programma per processor
 { for j=1 to i pardo
   { copy[j+aantal[j]]=copy[j];
     n_array[j+aantal[j]]=n_array[j];
     aantal[j]=aantal[j+aantal[j]]=2*aantal[j];
   }
 }
// na deze lus staat a ten minste in copy[1]...copy[n]
}
```

Deze keer zullen wij het voor de duidelijkheid nog eens anders schrijven. Het volgende Algoritme 18 geeft je het programma voor processor i. Het wordt opgestart voor $2^{\lceil \log(n/2) \rceil}$ processoren en elke processor kent zijn eigen nummer "`nummer`".

Algoritme 18 *Waarden kopiëren in een EREW-PRAM – programma voor processor* `nummer`

```
copywaarden(a,n,copy[],aantal[])

{
  n_array[nummer]= 0;

  if (nummer=1)
   { copy[1]=a;
     n_array[1]=n;
     aantal[1]=1;
   }

  for (i[nummer]=1; (n_array[nummer]= 0) || (i[nummer]<n_array[nummer]);
                                              i[nummer]=2*i[nummer])
// aan elke van de volgende binnenste lussen wordt
// tegelijk begonnen -- wij hebben dus een globale synchronisatie
 { if (nummer <= i[nummer])
   { copy[nummer+aantal[nummer]]=copy[nummer];
     n_array[nummer+aantal[nummer]]=n_array[nummer];
     aantal[nummer]=aantal[nummer+aantal[nummer]]=2*aantal[nummer];
   }
 }
// na deze lus staat a ten minste in copy[1]...copy[n]
}
```

Als n kopieën gemaakt moeten worden, vraagt dit algoritme tijd $O(\log n)$ en $2^{\lceil \log(n/2) \rceil}$ processoren – dus $O(n)$ processoren.
Belangrijk is dus te zien dat ook al kunnen EREW machines niet alle waarden tegelijk lezen, je die wel in tijd $O(\log n)$ kan verdelen. Dus kan een ER machine een CR machine met een verliesfactor van ten hoogste $O(\log n)$ (waarbij n het aantal processoren is) simuleren door waarden te verdelen. Maar dat is niet zo triviaal als het misschien lijkt: in één stap kunnen n

processoren n verschillende variabelen wijzigen. Als die waarden dan allemaal verdeeld moeten worden, heb je extra processoren nodig om dat in tijd $O(\log n)$ te doen. Bovendien moet je ook garanderen dat geen twee van de extra processoren dezelfde variabele willen wijzigen...
Een serieel algoritme vraagt tijd $O(n)$. Het product uit tijd en het aantal processoren is dus voor het parallelle algoritme $O(n \log n)$ en voor het seriële $O(n)$. Dat betekent dat het algoritme niet kost optimaal is.

Verbetering van de kost performantie

Wij zullen hier een trucje zien dat je soms kan gebruiken om de kost van een algoritme te verbeteren zonder de – asymptotische – tijd slechter te maken. Typisch voor gevallen waar dit trucje lukt is dat er maar één stap is waar alle processoren nodig zijn (een constant aantal stappen zou ook OK zijn).
Eerst passen wij een licht gewijzigde versie van `copywaarden()` toe. Daarbij worden niet n kopieën gemaakt maar alleen $n/\log n$ kopieën met $O(n/\log n)$ processoren. De tijd die hiervoor nodig is, is $O(\log(n/\log n)) = O(\log n - \log\log n)) = O(\log n)$
Achteraf hebben wij ten minste $n/\log n$ kopieën van $a, n, \ldots$. Dan kopieert elke van de ongeveer $n/(2 * \log n)$ processoren de waarden nog $2(\log n - 1)$ keer. Dat vraagt nog eens tijd $O(\log n)$ en achteraf hebben wij ten minste n kopieën, waarbij wij $O(n/\log n)$ processoren gebruikt hebben en de tijd was $O(\log n)$. De kost is dus $O((n/\log n) * \log n)) = O(n)$ – wat dus identiek is aan het bestmogelijke seriële algoritme. Het algoritme is dus kost optimaal.

Oefening 114 *Geef de pseudocode voor de processor met nummer* `nummer` *en het kost optimale algoritme.*

Parallel lookup

Het volgende voorbeeld bepaalt in constante tijd of een getal a in een array aanwezig is:

Algoritme 19 *Parallel Lookup in CRCW-PRAM*

```
is_present(a, lijst[])
// geeft TRUE terug als a in lijst[] -- anders FALSE

contained=FALSE

for i=1 to |lijst| pardo
     { if (lijst[i]=a) contained=TRUE; }
```

```
return contained;
```

Dit algoritme werkt alleen maar als je een CRCW-PRAM als model hebt. Alle `|lijst|` processoren moeten `a` tegelijk lezen en als er meerdere kopieën van `a` in `lijst` zitten, moeten ook meerdere processoren tegelijk schrijven. Deze pseudocode zegt duidelijk wat de `|lijst|` processoren in het parallel moeten doen – maar welke processor doet bv. `contained=FALSE`? Dit is een deel dat van een arbitraire processor uitgevoerd kan worden – dus bv. processor p_1. Achteraf moeten alle processoren synchroon met hun deel in de lus beginnen en dan moet – nadat alle processoren klaar zijn – een arbitraire processor (dus bv. opnieuw p_1) het commando `return contained` uitvoeren. Ook toekomstige pseudocode moet zo geïnterpreteerd worden.

Algoritme 20 *Parallel Lookup in EREW-PRAM*

```
is_present(a, lijst)
{
// geeft TRUE terug als a in lijst[] -- anders FALSE
// n is de lengte van de lijst

copywaarden(a,n,waarde[],aantal[])

// ook n is nu gekend in aantal[]
for i=1 to n pardo
     { contained[i]=FALSE;
       if (lijst[i]=waarde[i]) contained[i]=TRUE; }

for j=1 to n/2 pardo
   { while (aantal[j]>1) // het aantal samen te vatten waarden
     aantal[j]=aantal[j]/2;
     if (j<= aantal[j])
     {
       contained[j]= contained[j] || contained[j+aantal[j]];
     }
   }
// na deze lus staat TRUE in contained[1] als er ten minste 1
// keer ergens TRUE stond

return contained[1];
}
```

Je ziet dus dat het CR-model duidelijke voordelen heeft in vergelijking met een ER-model. Maar of dat in hardware ook zo geïmplementeerd kan worden is iets anders...

Oefening 115 *Beschrijf een algoritme dat test of een getal p een priemgetal is. Het resultaat moet p zijn als p een priemgetal is en een niet triviale deler als p geen priemgetal is.*

- *Beschrijf een CRCW-PRAM algoritme dat in tijd $O(1)$ werkt.*
- *Beschrijf een EREW-PRAM algoritme dat in tijd $O(\log p)$ werkt.*

Wat is de inputlengte en hoeveel processoren gebruik je (als functie van de **inputlengte***)?*
Hoeveel keer sneller is dit algoritme dan het priemzeef van Erathostenes op een seriële machine?

Oefening 116 *Beschrijf een algoritme voor een CRCW-machine dat in constante tijd het minimum van een verzameling van n getallen kan berekenen. Hoeveel processoren gebruik je?*
Tip: *Gebruik dat een getal het minimum is als alle mogelijke vergelijkingen met de andere getallen opleveren dat het kleiner is.*

5.4 Semigroep algoritmen

Wij zullen nu een algoritme voor een EREW-machine beschrijven dat de som van n getallen in tijd $O(\log n)$ kan berekenen.
Wij bespreken dat relatief abstract omdat wij zo in principe voor vele *verschillende* gevallen tegelijk een algoritme geven – bv. voor de operatoren "`*`", "`minimum`", "`maximum`", etc. Al deze operatoren hebben de eigenschap dat ze samen met de verzameling waarop ze opereren een semigroep vormen.
Een semigroep is een verzameling samen met een associatieve binaire bewerking – dus een bewerking met de eigenschap dat (als wij de bewerking als $\oplus$ schrijven) $(a \oplus b) \oplus c = a \oplus (b \oplus c)$. Als bv. de verzameling de verzameling van gehele getallen is en de bewerking "`+`" dan voldoet dat zeker aan deze eigenschap. In feite heeft "`+`" zelfs nog de mooie eigenschappen dat de bewerking commutatief is en dat er een neutraal element en een invers element bestaan – maar dat hebben wij hier niet nodig. De voorbeelden zullen alleen maar voor verzamelingen van getallen zijn, maar als de operator $\oplus$ bv. met matrices werkt, werken de algoritmen *in principe* op dezelfde manier.
De pseudocode voor deze algoritmen kan je als volgt schrijven:

Algoritme 21 *Semigroep som in een EREW-PRAM*
//Invoer: array a[] met n= 2^k getallen beginnend met index 1.
//Resultaat: a[1]$\oplus\cdots\oplus$a[n]

```
semigroep_plus(a[],n)

{ copywaarden(n,n/2,n_array[],aantal[]);
  // nu kan elke processor die we gebruiken een kopie van n lezen

  for j=1 to log n do
   {
     for i=1 to n/2^j pardo
        { buffer[i]=a[2*i-1]⊕a[2*i];
          a[i]=buffer[i]; }

   }

  return a[1];
}
```

Oefening 117 *Wijzig Algoritme 21 zo dat het ook voor getallen n werkt die geen macht van 2 zijn.*

Dit algoritme vraagt $O(n)$ processoren en tijd $O(\log n)$. Omdat een serieel algoritme tijd $O(n)$ vraagt is dit parallel algoritme dus niet kost optimaal – maar het is niet moeilijk om het zo te wijzigen dat het kost-optimaal is:

Oefening 118 *Wijzig Algoritme 21 zo dat het de semigroep som in tijd $O(\log n)$ op een kost optimale manier berekent.*

Als wij nu de werk complexiteit van Algoritme 21 berekenen dan zien wij dat elk deel van het algoritme inderdaad werk optimaal is: eerst wordt `copywaarden()` gebruikt en als wij daarvoor de kost optimale versie gebruiken dan werkt dat zeker met werk complexiteit $O(n)$.
Maar de for lus in `semigroep_plus()` vraagt werk

$$W(n) = c_1 + \sum_{j=1}^{\log n}(c_2 * n/2^j) = O(n)$$

met constanten c_1 en c_2.

Dus vraagt ook dit deel van het Algoritme 21 werk $O(n)$ en is dus werk optimaal.
Maar: Of deze analyse echt klopt, hangt er wel vanaf hoe de code van elke processor er precies uitziet. Als de processoren in de lus altijd toetsen of zij nog aan `n/`2^j voldoen, heeft elke van de processoren in alle `log n` keren dat de buitenste lus wordt doorlopen werk. Dan is het werk $O(n \log n)$. Maar inderdaad kan een processor zodra hij niet meer aan deze voorwaarde voldoet helemaal stoppen – en dan is het werk in dit deel $O(n)$.
Als je het algoritme als een geheel ziet, moet je de $O(n)$ processoren die de beschreven versie gebruikt al vanaf het begin hebben – dus ook al voor copywaarden. Je moet de code voor die processoren dus zo schrijven dat zij het copywaarden gedeelte gewoon niet afwerken (een `if` gebaseerd op hun nummer dat dat deel overslaat) – en er dus ook geen werk doen.

Oefening 119 *(Heel gemakkelijk – gewoon om te zien of de tekst helemaal verstaan is)*
De volgende bewering heeft twee richtingen. Voor elke richting geef een bewijs of een tegenvoorbeeld:
Een parallel algoritme is werk optimaal als en slechts als het kost optimaal is.

Oefening 120 *Geef een $O(\log n)$ tijdsbegrensd parallel EREW algoritme voor het berekenen van $a[0] - a[1] + a[2] - a[3] + \cdots + a[n-2] - a[n-1]$ voor een gegeven invoerarray $a[]$ met $n = 2^k \geq 2$ getallen beginnend met index 0.*

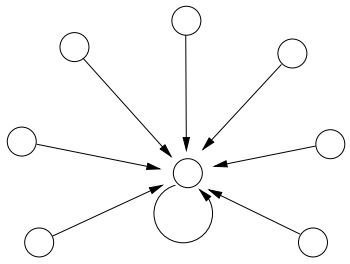
Oefening 121 *Geef operatoren (en misschien een preprocessing dat constante tijd op een EREW-machine vraagt) zodat je het* parallel lookup probleem *voor een vast getal a (dat je dus in de code van elke processor kan schrijven) ook als een semigroep probleem kan beschouwen.*

5.5 Eén parallel grafenalgoritme

In dit deel zullen wij een parallel CRCW algoritme zien dat de samenhangscomponenten in een graaf bepaalt. Dat is aan de ene kant om te zien hoe parallelle algoritmen op iets ingewikkeldere structuren dan alleen maar arrays van getallen werken en aan de andere kant om ook een algoritme te zien dat een beetje (maar niet veel) ingewikkelder is.
Seriële algoritmen voor het berekenen van samenhangscomponenten werken meestal met Depth-First search of Breadth-First search. Daarbij wordt aan elke top een getal toegekend zodat toppen die hetzelfde getal toegekend krijgen – het identificatienummer van de component – in dezelfde component zitten.

Hier willen wij ook dergelijke identificatienummers toekennen – maar dat in tijd $O(\log |V|)$ als de graaf $G = (V, E)$ is.
Wij gebruiken de union-find datastructuur die wij al in Datastructuren en Algoritmen II als voorbeeld hebben gezien om equivalentieklassen (dus ook samenhangscomponenten) efficiënt te berekenen en te updaten. Maar wij zullen de structuur hier opnieuw beschrijven.

Definitie 22 *Een gerichte graaf $D = (V, E)$ (met loops) heet een gerichte wortelboom als elke top één uitgaande boog heeft en er een $v \in V$ (deze top noemen wij dan ook de wortel van de boom) bestaat waar de uitgaande boog een lus is en waar vanuit elke top een gericht pad naar v bestaat.*
Een gerichte wortelboom heet ster als de afstand van elke top tot de wortel ten hoogste 1 is.


Een ster

Opmerking 23 *Stel dat in een CRCW PRAM een array* `opv[]` *van lengte n een gerichte graaf voorstelt waarin de enige cykels loops zijn en waarin uit elke top precies één boog vertrekt. Voor $1 \leq i \leq n$ is* `opv[i]` *de eindtop van de unieke boog die in i vertrekt.*
Dan kan een CRCW PRAM met n processoren in constante tijd beslissen welke top in een component ligt die een ster is – of precies: in constante tijd een array `ster[]` *invullen zodat* `ster[v]=TRUE` *als v in een ster ligt en anders* `FALSE`*.*

Bewijs: Het algoritme dat dit doet, werkt als volgt:

Algoritme 22 *Bepalen welke toppen in sterren liggen*

```
// Het programma voor processor p
// In het begin: opv[] bevat de opvolgers
// Op het einde: ster[v] is juist ingevuld voor elke top v


void beleg_ster_array(ster[])
{
```

```
ster[p]=TRUE;

if (opv[p] != opv[opv[p]])
  { ster[p]=FALSE;
    ster[opv[opv[p]]]=FALSE;
  } // stap 1

if (ster[opv[p]]=FALSE) ster[p]=FALSE; // stap 2
}
```

Dat dit algoritme in constante tijd werkt, is duidelijk. Het is ook duidelijk dat het hier kan gebeuren dat meerdere processoren dezelfde variabele willen lezen of daarin willen schrijven.

Dat dit algoritme juist werkt, kan als volgt gezien worden:

Wij noemen een top die zijn eigen opvolger is een looptop. Een top in een verzameling van wortelbomen maakt deel uit van een componente die een ster is als en slechts als hij aan één van de volgende voorwaarden voldoet:

a.) hij is een looptop en er is geen gericht pad van lengte 2 naar deze top

b.) hij is geen looptop, maar zijn opvolger voldoet aan a.)

Dat is misschien duidelijk, maar het is ook een goede oefening dat expliciet aan de hand van de definitie te bewijzen!

Na stap 1 staat voor elke looptop en elke top op een afstand van 2 of meer van een looptop de juiste waarde in `ster[]`. Een looptop die niet in een ster zit wordt door de processor p_v van een top v op afstand 2 als `FALSE` gemarkeerd en toppen w op afstand 2 of meer van een looptop worden door hun eigen processor p_w als `FALSE` gemarkeerd.

De enige toppen waarvoor de waarde op dit moment dus nog fout kan zijn, zijn toppen op afstand 1 van een looptop. Maar die maken natuurlijk precies dan deel uit van een ster als de looptop waar hun boog naartoe leidt deel uitmaakt van een ster – deze toppen worden dus in stap 2 juist gemarkeerd.

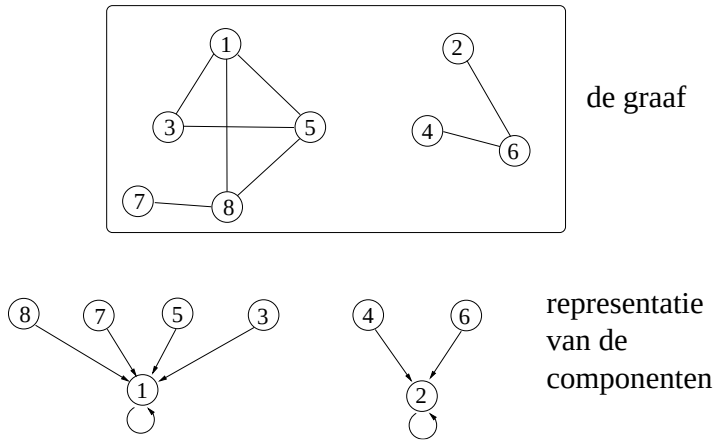
■

Oefening 122 *Wordt de array* `ster[]` *ook door het volgende algoritme juist ingevuld?*

Algoritme 23 `void beleg_ster_array(ster[])`

```
{
ster[p]=TRUE;

if (opv[p] != opv[opv[p]])
  { ster[p]=FALSE;
    ster[opv[p]]=FALSE;
    ster[opv[opv[p]]]=FALSE;
  }

}
```

De componenten worden voorgesteld door sterren – elke top wijst naar een top in de component – misschien zichzelf, misschien ook een andere top. Maar toppen in dezelfde component wijzen naar dezelfde top. Het nummer van de component waarin een top v zit, is dan gewoon de top waarnaar v wijst.



Figuur 27: Een graaf en hoe de samenhangscomponenten voorgesteld kunnen worden. Belangrijk is dat alle toppen in één component naar dezelfde top wijzen – het nummer dat de component voorstelt. Het is niet belangrijk welke van de toppen in de component dat is.

Het idee is nu dat in het begin alle toppen naar zichzelf wijzen. Deze singletons zijn de kleinstmogelijke wortelbomen. Die worden dan uitgebreid tot grotere wortelbomen en tegelijk wordt path compression (vergelijk datastructuren voor verzamelingen in DA2) toegepast. Ten slotte correspondeert elke component tot maar 1 wortelboom – en die is een ster of wordt tot een ster gecomprimeerd.

Het samenvoegen van de wortelbomen gebeurt in parallel. Voor elke boog $\{v, w\}$ gebruiken wij een processor $P_{\{v,w\}}$ die probeert de wortelboom waarin v zit en die waarin w zit samen te voegen (dit *samenvoegen* wordt ook hooking genoemd. Als de boom waarin v zit een ster is en die wordt aan die waarin w zit gehangen dan is de precieze *hooking* operatie `opv[opv[v]]=opv[w]`.
Maar: wij moeten opletten dat wij op deze manier geen gerichte cykels opbouwen door bv. tegelijk wortelboom B_1 aan wortelboom B_2 te voegen en omgekeerd! Bovendien moet de wortelboom met v een ster zijn omdat die anders door deze operatie uit elkaar kan vallen als `opv[v]` niet de wortel is. Wij geven nu eerst de pseudocode en bespreken die dan in stappen. Om niet altijd met buffers te moeten werken die garanderen dat er in één lijn niet dezelfde variabele gelezen en geschreven wordt, spreken wij af, dat leesoperaties in één lijn altijd de waarde lezen van voor de lijn. Dat kan je dan ook met buffers schrijven, maar daardoor wordt de code alleen maar minder leesbaar.

Algoritme 24 *Parallel samenhangscomponenten berekenen*

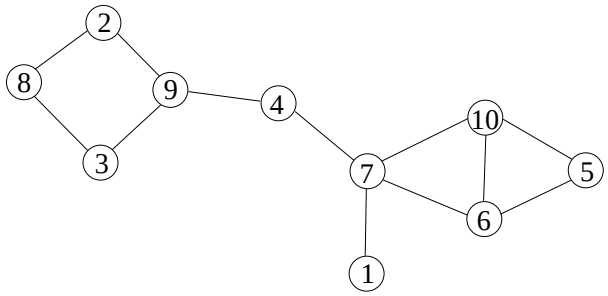
```
// Invoer: een graaf G=(V,E) als lijst van toppen en bogen
// Op het einde: voor elke top v bevat opv[v] een nummer dat
// zijn component kenmerkt.

// voor elke top v hebben wij een processor P_v en voor elke
// boog {v,w}  een processor P_{v,w}
// wij schrijven  hook v->w voor opv[opv[v]]=opv[w]

mark_components(G) {
 for all vertices v pardo opv[v]=v;
 // hier worden singletons gebouwd

 for all edges {v,w} pardo
  { if (v>w) hook v->w; else hook w->v;  //(a)
    if (v is singleton) hook v->w;  //(b)
    if (w is singleton) hook w->v;  //(b)
  }

 still_change=TRUE;
 while (still_change)
  {
   beleg_ster_array(ster[])
   for all edges {v,w} pardo
    { if (ster[v] AND (opv[v]>opv[w])) hook v->w;   //(c)
      if (ster[w] AND (opv[w]>opv[v])) hook w->v; } //(c)
```

```
   beleg_ster_array(ster[]); // updaten
   for all edges {v,w} pardo
    { if (opv[v]≠opv[w])
        { if (ster[v]) hook v->w;   //(d)
          if (ster[w]) hook w->v; } //(d)
    }
   // kijken of er veranderingen zijn en path compression:
   still_change=FALSE;
   for all vertices v pardo
    { if (opv[v]≠opv[opv[v]])
       { opv[v]=opv[opv[v]]; still_change=TRUE; } //(e)
    }
  }
}
```

Oefening 123 *In Algoritme 24 is niet precies beschreven hoe* `(v is singleton)` *in constante tijd beslist kan worden. Werk dit deel van de code expliciet uit. Herschrijf de hele parallelle lus die dit deel bevat.*

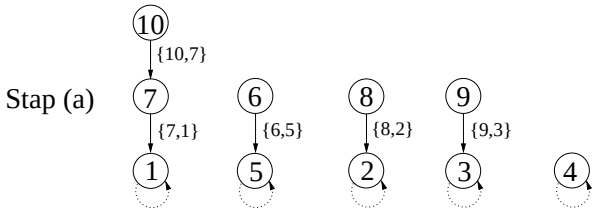
Nu zullen wij eerst naar een voorbeeld kijken hoe dit algoritme op een graaf werkt. In de tekeningen zijn de gewijzigde bogen altijd doorgaande lijnen terwijl oude bogen in de datastructuur met punten zijn voorgesteld. Natuurlijk is de voorgestelde ontwikkeling niet uniek. Als meerdere processoren in dezelfde variabele willen schrijven hangt het er vanaf welke processor erin slaagt te schrijven. Daarom staat in dergelijke gevallen aan de nieuwe bogen welke processor deze boog ingevuld heeft.

De graaf is:

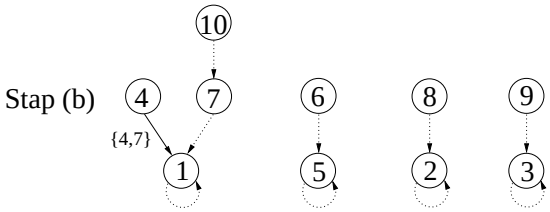


Eerst bouw je singletons:

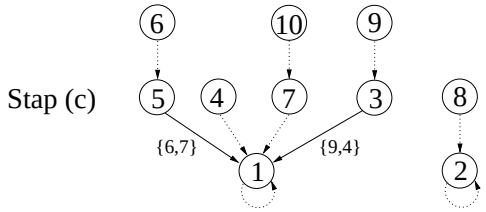
Dan bouw je de eerste wortelbomen in stap (a)



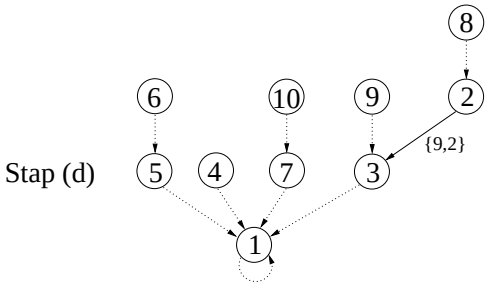
en hook je in stap (b) de overgebleven singletons aan bestaande wortelbomen:



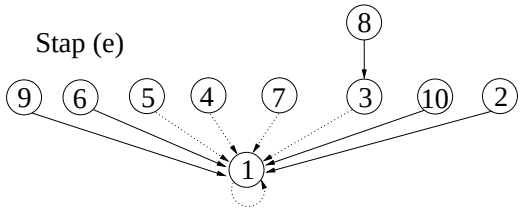
Nu begint de lus die sterren aan wortelbomen hookt en dan path compression toepast. Eerst worden in stap (c) sterren aan bestaande wortelbomen gehooked



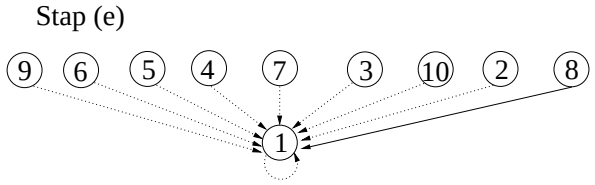
en dan in stap (d) nog eens overgebleven sterren:



Op het einde van de lus wordt path compression toegepast:



Er zijn nog wijzigingen, dus wordt de lus nog eens doorlopen. Er zijn geen sterren meer die nog gehooked kunnen worden, maar er is wel nog path compression mogelijk:



In de volgende doorloop van de lus zijn er dan helemaal geen wijzigingen meer en het programma stopt.

Om te bewijzen dat het algoritme juist werkt, moeten wij dus bewijzen dat er op het einde alleen maar sterren zijn en dat twee toppen dan en slechts dan op het einde tot dezelfde ster behoren als ze tot dezelfde component van de graaf behoren.
Dat bewijzen wij door verschillende kleine tussenstappen

Opmerking 24 *Op elk moment vormt de datastructuur in het algoritme een woud van gerichte wortelbomen.*

Bewijs: In het begin zijn het allemaal singletons – daarvoor klopt de opmerking dus.

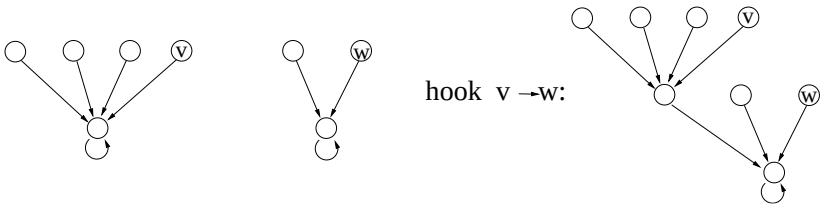
In stap (a) kunnen ook geen cykels toegevoegd worden omdat alleen grotere toppen naar kleinere wijzen.

In stap (b) wordt alleen dan een boog vanuit v toegevoegd als alle buren van v groter zijn dan v (anders zou v nu een andere opvolger hebben) **en** al naar een andere top wijzen (de enige reden dat de poging van de corresponderende boog $\{w, v\}$ niet slaagde, kan zijn dat de processor van een andere boog $\{w, v'\}$ succesvol was). De buren zijn dus allemaal geen singletons meer – er wordt dus geen boog naar v toegevoegd en de datastructuur blijft dus een woud.

Na stap (b) zijn ook alleen nog die toppen singletons die ook in de graaf geïsoleerd zijn – alle andere toppen zijn deel van een wortelboom met ten minste 2 toppen.

De enige mogelijkheid dat in stap (c) een cykel geïntroduceerd wordt, is dat de bogen waarvan de processoren nieuwe hook-operaties uitvoeren componenten in een cykel aan elkaar voegen die allemaal sterren zijn (elke component moet een uitgaande boog hebben). Maar omdat het sterren zijn, worden in deze cykel altijd de wortels vergeleken als het om `opv[v]` en `opv[w]` gaat – en dus moet er één ster zijn met top v die gehooked wordt en waarbij `opv[v]<opv[w]` – een tegenstrijdigheid. Dus worden in deze stap ook geen cykels gevormd.

In stap (c) worden geen nieuwe sterren gevormd omdat als één ster op een wortelboom gehooked wordt het resultaat in het begin een diepte van ten minste 2 heeft – zelfs dan als het allebei sterren zijn (zie Afbeelding 28).



Figuur 28: Een ster met top v wordt op een andere ster met top w gehooked.

Stap (d) heeft dus alleen sterren die ook voor stap (c) al sterren waren. Analoog met stap (b) worden ze nu alleen maar gehooked op componenten die geen sterren zijn waardoor opnieuw geen cykel kan ontstaan.

De enige stap waar `opv[]` dan nog gewijzigd wordt is stap (e). Maar het is duidelijk dat daar de paden alleen maar korter gemaakt worden en geen cykels kunnen ontstaan.

■

Opmerking 25 *Als een wortelboom voor de compressiestap (e) een diepte van $d > 1$ heeft dan heeft hij achteraf een diepte van ten hoogste $\lfloor (2d)/3 \rfloor$*

Bewijs: Met inductie kan je gemakkelijk bewijzen dat een top op even afstand d van de wortel voor de compressie achteraf een afstand van $d' = d/2$ heeft. Als de afstand voor de compressie oneven is, is hij achteraf $d' = (d+1)/2$. De maximale waarde van d'/d is dan voor $d = 3$ en dan is $d' = 4/2 = (2/3)d$.

■

Opmerking 26 *Als het algoritme stopt, zijn alle componenten sterren en twee toppen maken deel uit van dezelfde ster als en slechts als ze tot dezelfde component behoren.*

Bewijs: Zolang er nog componenten zijn die geen sterren zijn, wordt doorgegaan met de compressie en dus zijn er ook wijzigingen en het algoritme gaat door. Dat nooit toppen uit verschillende componenten in dezelfde wortelboom terechtkomen is ook duidelijk. Stel nu dat het algoritme klaar is en dat er nog twee toppen zijn die tot dezelfde component behoren maar in verschillende wortelbomen zitten – dus in verschillende sterren. Deze toppen kunnen zo gekozen worden dat er een boog $\{v, w\}$ is die beide toppen bevat (waarom?). Wij mogen stellen dat `opv[v]>opv[w]` – maar dan worden de wortelbomen van v en w ofwel aan elkaar gehooked ofwel aan andere wortelbomen – maar het algoritme is zeker nog niet afgelopen – een tegenstrijdigheid.

■

Wij weten dus dat het algoritme juist werkt en moeten alleen nog de complexiteit bepalen:

Lemma 27 *De tijdscomplexiteit van Algoritme 24 opgestart op een graaf $G = (V, E)$ is $O(\log |V|)$ met $|V| + |E|$ processoren.*

Bewijs: Tot de `while (still_change)` lus is zeker maar constante tijd nodig en het is ook duidelijk dat één doorloop van de lus constante tijd vraagt. Wat nog bewezen moet worden is dat de lus maar $O(\log |V|)$ keren wordt doorlopen.

Daarvoor definiëren wij een wortelboom in de datastructuur als *actief* (in een doorloop van de lus) als hij in de lus op een andere wortelboom gehooked wordt, een andere wortelboom op deze gehooked wordt of als w gecomprimeerd wordt. Anders heet de boom *passief*. Een boom is passief als en slechts als het een ster is en er is geen boog van een top in deze ster naar een top in een andere wortelboom: als een wortelboom geen ster is wordt hij zeker gecomprimeerd en als hij een ster is met buren in andere wortelbomen dan wordt hij zeker in stap (c) of (d) gehooked. Maar dat betekent ook dat als een wortelboom één keer passief is hij dat zeker ook blijft!

Nu kijken wij naar de volgende waarde van de datastructuur D:

$$W(D) = \sum_{\{T \in D | T \text{ is actief}\}} d(T)$$

waarbij $d(T)$ de diepte van de wortelboom T is. Het is belangrijk te zien dat omdat een wortel nooit aan een blad geplakt wordt maar de pointer naar een top op afstand ten minste 1 van een blad wijst, de diepte van een verzameling wortelbomen die op elkaar gehooked wordt ten hoogste de som van de enkele diepten voor het hooken kan zijn. Tijdens de hooking operaties kan $W(D)$ dus niet groeien. Inderdaad kan $W(D)$ dus alleen maar kleiner worden en als path compression wordt toegepast of componenten pasief worden zal dat ook zeker gebeuren.

Natuurlijk geldt altijd $W(D) \leq |V|$ en als $W(D_t)$ de waarde op het einde van de t-de doorloop van de lus is dan geldt met Opmerking 25 $W(D_{t+1}) \leq (2/3)W(D_t)$. Stel dat t het nummer van de op één na laatste doorloop is. Dan geldt $W(D_t) \geq 1$ omdat het algoritme anders zou stoppen (geen wijzigingen). Als D_0 de datastructuur voor de eerste doorloop van de lus is dan geldt met $W(D_0) \leq |V|$:

$$W(D_t) \leq (\frac{2}{3})^t W(D_0) \quad \Rightarrow \quad 1 \leq (\frac{2}{3})^t |V| \quad \Rightarrow \quad t \leq \frac{1}{\log(3/2)} \log |V|$$

Dus wordt de lus inderdaad maar $O(\log |V|)$ keren doorlopen.

■

Oefening 124 *Wijzig het algoritme zo, dat het aantal componenten van een graaf bepaald en uitgevoerd wordt.*

Oefening 125 *Wijzig het algoritme zo, dat de nummers van de componenten niet arbitraire toppennummers zijn maar* $1, \ldots, k$ *als er* k *componenten zijn. (Deze oefening is niet zo gemakkelijk.)*

Oefening 126 *Bewijs* **expliciet** *dat twee toppen die in Algoritme 24 in dezelfde wortelboom terechtkomen ook tot dezelfde component van de graaf behoren.*

Oefening 127 *Gegeven een verzameling* M *van elementen (in DA2 hebben wij dat altijd een* universum *genoemd) en een verzameling* R *van relaties tussen deze elementen. Beschrijf een parallel CRCW algoritme voor een PRAM dat de door* R *gegenereerde equivalentieklassen in tijd* $O(\log |M|)$ *berekent.*

Oefening 128 *Bespreek de wijzigingen die in Algoritme 24 nodig zijn zodat het ook op een EREW PRAM kan werken. Wat is de complexiteit van jouw gewijzigd algoritme?*

5.6 Pipelining

Ten slotte zullen wij het nog kort over een netwerkmodel hebben – alleen om een dergelijk model ook eens gezien te hebben. In dit model werken wij met gewone processoren die allemaal hun eigen geheugen hebben. Als in de pseudocode dus een variabele gebruikt wordt dan is die lokaal – dus verschillend voor verschillende processoren. Maar ze kunnen ook data doorsturen naar andere processoren waarmee ze verbonden zijn. Wij gebruiken bevelen zoals send(x,i) of receive(y,j) om aan te duiden dat wij de inhoud van een variabele naar processor i sturen of een boodschap van processor j in variabele y willen opslaan. Als op processor j het bevel `send(x,i)` gebruikt wordt, moet er wel een verbinding zijn tussen de processoren i en j – anders moet ervoor gezorgd worden dat de data met hulp van andere processoren doorgestuurd wordt – wat natuurlijk duurder is. In ons voorbeeld is het voldoende als voor n processoren $1, \ldots, n$ processor i voor $1 \leq i \leq n-1$ een lees/schrijf verbinding heeft met processor $i+1$. Het netwerk is hier dus heel eenvoudig – het is een pad.
Zoals de naam pipelining al doet vermoeden geven deze algoritmen data door langs een ketting van processoren. En ook al is het algoritme ontworpen voor een netwerk van processoren kan het op een PRAM natuurlijk gemakkelijk gesimuleerd worden. In feite kunnen deze pipelining algoritmen vooral op een multicore processor met linux gebruik makend van pipes bijzonder gemakkelijk geïmplementeerd worden.
Een kenmerk dat aantoont dat het principe van pipelining toepasbaar is, is een soort genestelde structuur van de problemen. Het principe kan het best op voorbeelden gedemonstreerd worden:
Het eerste voorbeeld dat wij zullen zien, is het sorteren van getallen. Invoer zijn n getallen (die van processor 1 worden gelezen) en uitvoer zijn deze n getallen in dalende volgorde die van processor 1 worden geschreven. Als wij het grootste van de n getallen verwijderen dan is het tweede grootste het grootste van de overblijvende getallen – en algemeen is het i-de grootste het grootste van de verzameling van getallen waaruit de grootste $i-1$ getallen al verwijderd zijn.
Dit kunnen wij als volgt implementeren: processor 1 leest alle getallen, houdt het grootste getal bij en stuurt de rest naar processor 2. Algemeen houdt processor i ($2 \leq i \leq n$) het grootste van de getallen die het van processor $i-1$ krijgt bij en stuurt de rest naar processor $i+1$. Als wij `receive(.,0)` interpreteren als *lees het volgende getal uit de invoer* en `send(.,0)` als *schrijf het volgende getal naar de uitvoer* en het einde van de invoer door een `EOF` gekenmerkt is, kunnen wij dit algoritme opschrijven als

Algoritme 25 *Sorteren met een pipeline*

```
 Het programma voor processor p
//Invoer: een reeks van n getallen
//Uitvoer: dezelfde reeks in dalende volgorde

{
receive(best,p-1);

if (best!=EOF)
  {
    receive(temp,p-1);
    while (temp != EOF)
      { if (temp>best)
           { send(best,p+1); best=temp; }
        else send(temp,p+1);
        receive(temp,p-1);
      }
    // nu zijn alle getallen gelezen
    send(EOF,p+1);

    send(best,p-1);
    // het grootste van de ontvangen getallen wordt eerst gestuurd
    receive(temp2,p+1);
    while (temp2 != EOF)
      {
        send(temp2,p-1);
        receive(temp2,p+1);
      }
  }
send(EOF,p-1);
}
```

Op het einde worden dus de getallen aan de vorige processor teruggestuurd – en omdat altijd het grootste getal eerst wordt gestuurd, kan je gemakkelijk met inductie bewijzen dat de getallen in dalende volgorde uitgevoerd worden.

Wij zullen nu nog een tweede voorbeeld zien waarvan jullie het seriële algoritme al kennen: het priemzeef van Eratosthenes.
Invoer zijn hier alle getallen $2, 3, \ldots, n$ en uitvoer zijn alle priemgetallen tussen 2 en n. De genestelde structuur kan hier zo beschreven worden dat een getal x een priemgetal is als en slechts als het door geen priemgetal kleiner

dan x deelbaar is. De i-de processor in de rij krijgt allen maar getallen die niet door de eerste $i-1$ priemgetallen deelbaar zijn en schrijft alleen die, die niet door de eerste i priemgetallen deelbaar zijn – verwijdert dus die getallen die door *zijn* priemgetal deelbaar zijn. Daarbij is *zijn priemgetal* het kleinste getal dat deze processor krijgt – en omdat de getallen in volgorde worden doorgegeven is het dus het eerste getal. Als wij opnieuw processor nummer 0 met invoer en uitvoer identificeren dan is het programma voor processor p:

Algoritme 26 *Priemzeef met een pipeline*

```
 //Het programma voor processor p
//Invoer: de getallen 2,...,n
//Uitvoer: de priemgetallen 2 <= p <= n in stijgende volgorde

{
receive(getal,p-1);
temp=getal;

while (temp != EOF)
 { if (temp modulo getal !=0) send(temp, p+1);
   receive(temp,p-1);
 }
// nu zijn alle getallen gelezen

send(EOF, p+1);

if (getal!=EOF)
{ send(getal,p-1);
  while (receive(temp,p+1)!=EOF) send(temp,p-1);
}
send(EOF,p-1);
}
```

Ook hier kan je door middel van inductie gemakkelijk bewijzen dat dit algoritme juist werkt.

Pipelining is in feite meer dan alleen maar toepasbaar op dit eenvoudige netwerkmodel. Het is een techniek die ook voor de ontwikkeling van algoritmen voor PRAMs toegepast kan worden als een taak mooi in deeltaken opgesplitst kan worden die na elkaar uitgevoerd moeten worden. In feite zijn op een PRAM veel dingen nog gemakkelijker omdat je natuurlijk een gemeenschappelijk geheugen kan gebruiken. Maar dit opsplitsen in deeltaken die **na** elkaar uitgevoerd moeten worden heeft natuurlijk al iets dat *inherent serieel*

is en soms is de speedup niet zo groot als je zou hopen of kan op een PRAM ook door andere middelen verkregen worden.

Oefening 129 *Gegeven een functie $f : \mathbb{N}\times\mathbb{N} \to \mathbb{N}$ waarvan je mag veronderstellen dat ze al geïmplementeerd is en op een enkele processor in constante tijd berekend kan worden. Invoer is nu een reeks $a_1, \ldots, a_n, \_, b_1, \ldots, b_n$ en het doel is nu een functie*

$$\sum_{i=1}^{n}\sum_{j=1}^{n} f(a_i, b_j)$$

te berekenen. Dat moet dus de uitvoer zijn.
Geef de pseudocode voor een pipeline algoritme dat dit resultaat in lineaire tijd berekent. Invoer en uitvoer gebeuren zoals in de voorbeelden vanuit processor 1.

Oefening 130 *Beschrijf een algoritme voor een CRCW-PRAM met n^2 processoren dat n verschillende getallen in een array $a[]$ in tijd $O(\log n)$ in dalende volgorde in de array $s[]$ kan schrijven.*

Index