ALGORITMIEK

Keuzemodule Wiskunde B/D

Mark de Berg

TU Eindhoven

Voorwoord

Algoritmiek is het gebied binnen de informatica dat zich bezig houdt met het ontwerpen en analyseren van algoritmen en datastructuren. Het is een fascinerend en breed vakgebied dat allerlei raakvlakken heeft met de wiskunde. Deze algoritmiek-module is gericht op scholieren uit de bovenbouw van het VWO en is in eerste instantie ontwikkeld om gebruikt te worden als keuzeonderwerp bij wiskunde B of D. Natuurlijk kan de module ook binnen het informatica-onderwijs gebruikt worden. De module vereist geen voorkennis over informatica of programmeren.

Het is onmogelijk om in een korte module een goed overzicht te geven van de algoritmiek—het bekendste basis-leerboek op dit gebied telt bijvoorbeeld al meer dan 1.000 pagina's. Het doel van de module is vooral om de leerling enig gevoel te geven van de aspecten die een rol spelen bij het ontwerpen van algoritmen. De hoofdstukken 3 en 4 van de module staan min of meer los van elkaar. Mocht er onvoldoende tijd zijn om de hele module te behandelen, dan kan er ook voor gekozen worden om hoofdstuk 3 of hoofdstuk 4 over te slaan.

In het schooljaar 2008/2009 is de module uitgetest in drie 5VWO klassen (wiskunde B) van het Christiaan Huygens College te Eindhoven. Ik wil graag de wiskunde-sectie—en in het bijzonder de docenten Jan Debets, Erik de Rooij en Bertha Scholten—bedanken voor hun enthousiasme om aan dit "experiment" deel te nemen, en voor hun feedback op eerdere versies van de module.

Eindhoven, juli 2009 *Mark de Berg*

Inhoudsopgave

1 Inleiding

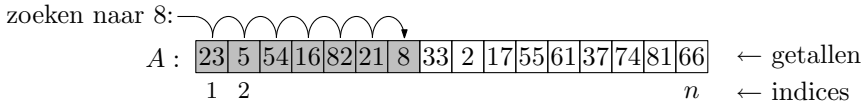
Internet, giromaten, mobiele telefoons, i-pods, computer games, televisies; zonder software zou het allemaal niet bestaan. Het hart van software wordt gevormd door *algoritmen*: precies beschreven stappenplannen om bepaalde taken uit te voeren. Meestal kan een taak op verschillende manieren worden aangepakt, waarbij de ene manier veel efficiënter is dan de andere. Het vinden van het beste algoritme is dan ook vaak een uitdagende puzzel. In deze module leer je een aantal belangrijke aspecten die hierbij een rol spelen.

1.1 Notatie van algoritmen: pseudocode

Een algoritme is een precies beschreven stappenplan om een bepaalde taak uit te voeren. Meestal zal het de bedoeling zijn dat het algoritme uiteindelijk in software wordt omgezet om door een computer uitgevoerd te kunnen worden. Het is daarom belangrijk dat de beschrijving van het algoritme precies genoeg is, zodat een programmeur het zonder problemen kan omzetten in de gewenste programmeertaal. We zullen algoritmen daarom beschrijven in *pseudocode*.

Laten we eens naar een voorbeeld kijken. Stel dat we een algoritme nodig hebben dat in een grote verzameling getallen zoekt naar een gegeven getal. Het opslaan van de verzameling gebeurt in een zogenaamd *array*: een rij getallen, net zoals bijvoorbeeld in een tabel. De getallen zijn genummerd van 1 t/m n. Als we het array A noemen, dan geven we het eerste getal aan met $A[1]$, het tweede getal met $A[2]$, enzovoorts. Het i-de getal staat dus in $A[i]$; de waarde i wordt de *index* van het getal genoemd. De hele verzameling getallen, $A[1], \ldots, A[n]$ wordt kortweg geschreven als $A[1..n]$.

Om te kijken of het getal q voorkomt, kunnen we nu gewoon het array aflopen en bij elk getal controleren of het gelijk is aan q. Zo ja, dan kunnen we stoppen, anders gaan we door met het volgende getal. Als we alle getallen gecontroleerd hebben zonder q gevonden te hebben, dan kunnen we concluderen dat q niet voorkomt. Fig. 1 illustreert dit zoekalgoritme. Om het array A



Figuur 1: Lineair zoeken in een array $A[1..n]$ met $n = 16$ getallen. De getallen die gecontroleerd worden als er naar $q = 8$ wordt gezocht, zijn in grijs aangegeven.

te doorlopen gebruiken we een *hulp-variabele*. Deze variabele, die we hier i noemen, geeft de index aan van het getal waar we op dat moment naar kijken. Het doorlopen van het array gebeurt dus door i telkens met 1 te verhogen. Het toekennen van een waarde aan een variabele wordt genoteerd als met het symbool $\leftarrow$. Als we bijvoorbeeld i de waarde 1 willen geven, dan schrijven we $i \leftarrow 1$ (spreek uit: "i wordt 1"), en als we i met 1 willen verhogen dan schrijven we $i \leftarrow i + 1$ (spreek uit: "i wordt $i + 1$"). In pseudocode ziet ons zoekalgoritme er nu als volgt uit.

Algoritme *LineairZoeken*(A, q)
Invoer: Een array $A[1..n]$ van getallen, en een getal q.
Uitvoer: Een index i zodanig dat $A[i] = q$, of "niet aanwezig" als zo'n index niet bestaat.
1. $i \leftarrow 1$
2. zolang $i \leqslant n$
   doe: als $A[i] = q$
      dan Rapporteer i en stop
      anders $i \leftarrow i + 1$
3. Rapporteer "niet aanwezig"

De invoer van het algoritme wordt tussen haakjes aangegeven, net als bij een wiskundige functie: zoals je $f(x)$ schrijft voor een functie f waar je een getal x in stopt, zo schrijf je *LineairZoeken*(A, q) voor een algoritme *LineairZoeken* waar je als invoer A en q in stopt. Omdat het niet direct duidelijk is wat de invoer voorstelt en wat het resultaat van het algoritme zou moeten zijn, wordt dit aan

het begin van het algoritme vermeld. De beschrijving van het algoritme zelf bestaat uit eenvoudige opdrachten (bijvoorbeeld "$i \leftarrow i+1$" of "Rapporteer i") en tests (bijvoorbeeld "$i \leqslant n$"). Daarnaast zijn er *herhalings-opdrachten* van de vorm

zolang ...
doe: ...

en *keuze-opdrachten* van de vorm

als ...
dan ...
anders ...

Als er bij het anders-gedeelte niets hoeft te gebeuren, dan wordt dit deel weggelaten.

Opgave 1.1.1 Wat is de uitvoer van het volgende algoritme?

Algoritme *EenAlgoritme*(A)
Invoer: Een array $A[1..n]$ met n getallen.
Uitvoer: ??
1. $i \leftarrow 1$; *aantal* $\leftarrow 0$
2. zolang $i \leqslant n$
   doe: als $A[i] > 0$
      dan *aantal* $\leftarrow$ *aantal* $+ 1$
      $i \leftarrow i + 1$
3. Rapporteer *aantal*

Toelichting: Merk op dat de opdracht "$i \leftarrow i + 1$" een beetje ingesprongen is, en recht onder *als* en *dan* staat. Dit geeft aan dat de opdracht bij de herhalings-opdracht hoort. De opdracht "Rapporteer *aantal*" is niet ingesprongen, en wordt dan ook pas uitgevoerd nadat de herhalings-opdracht helemaal afgewerkt is.

Opgave 1.1.2 Wat is de uitvoer van het volgende algoritme? (Let op: strikvraag!)

Algoritme *EenAnderAlgoritme*(A)
Invoer: Een array $A[1..n]$ met n getallen.
Uitvoer: ??
1. $i \leftarrow 1$; *totaal* $\leftarrow 0$
2. zolang $i \leqslant n$
   doe: *totaal* $\leftarrow$ *totaal* $+ A[i]$
3. Rapporteer *totaal*

Opgave 1.1.3 Geef een algoritme dat, gegeven een array $A[1..n]$ met n getallen, de som van alle getallen in A bepaalt.

Opgave 1.1.4 Geef een algoritme dat, gegeven een array $A[1..n]$ met n getallen, het grootste getal in A bepaalt.

Opgave 1.1.5 Geef een algoritme dat, gegeven een array $A[1..n]$ met n getallen en een getal q, bepaalt of er indices i en j zijn (en zo ja, welke) waarvoor geldt $i < j$ en $A[i] + A[j] = q$.

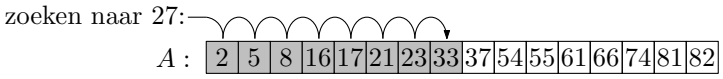
---

2 Efficiëntie van algoritmen

Zoals al eerder opgemerkt zijn er vaak verschillende manieren om een taak uit te voeren. De ene manier kan soms veel efficiënter zijn dan de andere. In dit hoofdstuk zullen we daar een voorbeeld van zien. We zullen ook bekijken hoe we de efficiëntie van een algoritme kunnen onderzoeken zonder het eerst uit te programmeren.

2.1 Binair zoeken

In de vorige sectie hebben we een simpel algoritme gezien, *LineairZoeken*, dat een gegeven getal zoekt in een array met n getallen. Is dit een goed algoritme, of kunnen we het zoeken op een slimmere manier aanpakken? Het antwoord is ja: het kan slimmer. Daarvoor moeten we de getallen wel op een meer gestructureerde manier opslaan; als ze op willekeurige volgorde in het array staan, dan zit er weinig anders op dan ze een voor een te controleren. Het ligt voor de hand



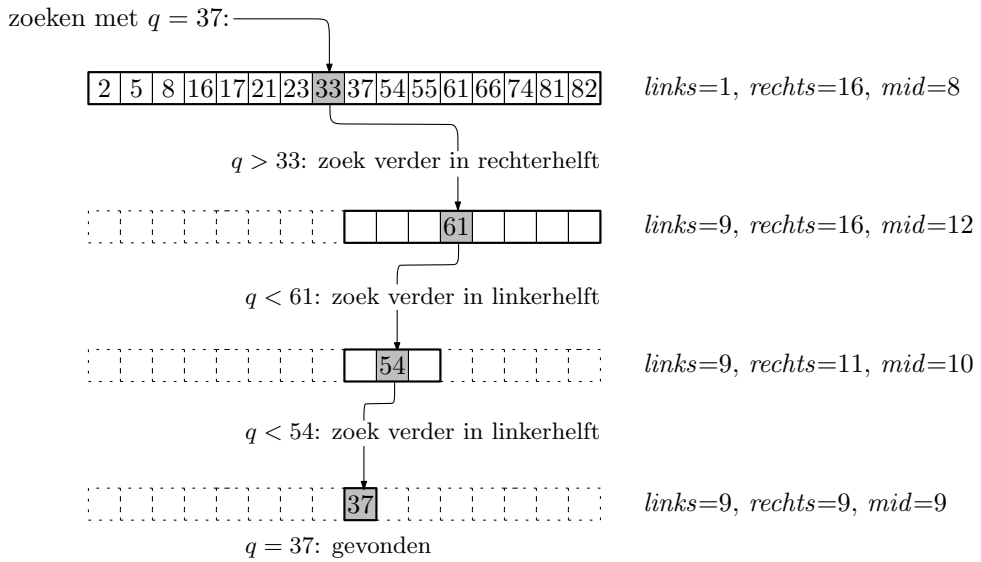
Figuur 2: Lineair zoeken in een gesorteerd array.

de getallen gesorteerd van klein naar groot in A op te slaan, zodat $A[1] \leqslant A[2] \leqslant \cdots \leqslant A[n]$. Als we dan naar een getal q zoeken, dan kunnen we stoppen zodra we een getal groter dan q tegenkomen. Zo kunnen we in Fig. 2 stoppen met zoeken naar 27 als we bij $A[8]$ zijn aangeland, want $A[8] = 33$.

Opgave 2.1.1 Pas algoritme *LineairZoeken* op de hierboven beschreven manier aan.

Deze aanpassing van het algoritme levert soms een versnelling op, maar als we pech hebben dan controleren we nog steeds alle getallen. Gelukkig is dat niet nodig als we het algoritme nog wat slimmer maken. Het idee is als volgt. In plaats van eerst $A[1]$ te bekijken, bekijken we eerst het middelste getal, $A[\frac{n}{2}]$. (We nemen voor het gemak even aan dat n even is, zodat $\frac{n}{2}$ een valide index is.) Als $q = A[\frac{n}{2}]$, dan zijn we gelijk klaar. Stel nu dat $q < A[\frac{n}{2}]$. Dan hoeven we de getallen in $A[\frac{n}{2}+1..n]$ niet meer te controleren. Deze getallen moeten dan immers allemaal groter dan q zijn, aangezien A gesorteerd is van klein naar groot. Met andere woorden, als $q < A[\frac{n}{2}]$ dan kunnen we het zoeken beperken tot het deelarray $A[1..\frac{n}{2}-1]$. Aan de andere kant, als $q > A[\frac{n}{2}]$ dan kunnen we het zoeken beperken tot $A[\frac{n}{2}+1..n]$. In het deelarray waarin we verder zoeken, doen we nu weer hetzelfde: we kijken naar het middelste getal, en zoeken dan verder in de linker- of rechterhelft van dat deelarray, enzovoorts. Het deelarray waarin nog gezocht moet worden, wordt dus steeds kleiner. Het zoekalgoritme is klaar als q gevonden is, of als er geen deelarray meer over is om in verder te zoeken. Fig. 3 illustreert deze strategie, die *binair zoeken* wordt genoemd.

Hieronder staat het algoritme in pseudocode. Het algoritme houdt twee indices bij, *links* en *rechts*. Deze indices geven de linker- en rechtergrens aan van het deelarray waarin we aan het zoeken zijn; de beginwaarden zijn $links = 1$ en $rechts = n$. In het stuk $A[links..rechts]$ willen we nu naar het middelste getal kijken. De index van dit getal, *mid*, is ongeveer $(links + rechts)/2$. We moeten alleen oppassen als $links + rechts$ oneven is, want dan is $(links + rechts)/2$ geen geheel getal en dus geen valide index. Daarom ronden we $(links + rechts)/2$ naar beneden af. Voor het naar beneden afronden van een getal x gebruiken we de notatie $\lfloor x \rfloor$; bijvoorbeeld $\lfloor 4\frac{1}{2} \rfloor = 4$, $\lfloor 8 \rfloor = 8$, en $\lfloor 2\frac{4}{5} \rfloor = 2$. (Het afronden naar boven wordt aangegeven met $\lceil x \rceil$.)



Figuur 3: Binair zoeken. Het deel van het array dat niet meer bekeken hoeft te worden is met stippellijnen getekend.

Algoritme *BinairZoeken*(A, q)
Invoer: Een array $A[1..n]$ met getallen, gesorteerd van klein naar groot, en een getal q.
Uitvoer: Een index i zodanig dat $A[i] = q$, of "niet aanwezig" als zo'n index niet bestaat.
1. *links* $\leftarrow 1$; *rechts* $\leftarrow n$
2. zolang *links* $\leqslant$ *rechts*
   doe: *mid* $\leftarrow \lfloor (links + rechts)/2 \rfloor$
   als $q = A[mid]$
   dan Rapporteer *mid* en stop
   anders als $q < A[mid]$
   dan *rechts* $\leftarrow mid - 1$
   anders *links* $\leftarrow mid + 1$
3. Rapporteer "niet aanwezig"

Zie Fig. 3 voor een voorbeeld van hoe de indices *links*, *rechts*, en *mid* veranderen in de loop van het algoritme, als we in het gegeven array naar het getal $q = 37$ zoeken. Als we naar bijvoorbeeld $q = 36$ hadden gezocht, dan was het algoritme hetzelfde verlopen tot het moment dat $mid = 9$. Na de test $q < A[9]$ had *rechts* dan de waarde 8 gekregen. Hierna was *rechts* $<$ *links* geweest; het algoritme zou dan naar stap 3 zijn gegaan en "niet aanwezig" hebben gerapporteerd.

Opgave 2.1.2 Bekijk het array van Fig. 3. Geef aan hoe de indices *links*, *rechts*, en *mid* veranderen in de loop van het algoritme *BinairZoeken* als naar het getal $q = 16$ wordt gezocht. Beantwoord dezelfde vraag voor $q = 23$ en voor $q = 73$.

Het principe van binair zoeken is dus om eerst in het midden van het array te kijken—vaak op één plek, maar soms ook op bijvoorbeeld de middelste twee plekken—en afhankelijk van wat daar staat in het linker- of rechterdeelarray verder te zoeken. Dit principe kan ook op andere zoekproblemen worden toegepast.

Opgave 2.1.3 Laat $A[1..n]$ een array zijn bestaande uit nullen en enen, waarbij $n \geqslant 2$, $A[1] = 0$ en $A[n] = 1$. (De nullen en enen kunnen verder willekeurig door elkaar staan.)

(i) Beredeneer dat er een index i moet bestaan (met $1 \leqslant i < n$) zodanig dat $A[i] = 0$ en $A[i + 1] = 1$. (Met andere woorden, er is een plek in het array waar een 1 direct rechts van een 0 staat.)

(ii) Geef een op binair zoeken gebaseerd algoritme dat zo'n index i vindt.

Opgave 2.1.4 Laat $A[1..n]$ een array met n verschillende getallen zijn waarin de getallen eerst steeds groter worden en dan, nadat het grootste getal is bereikt, steeds kleiner tot het eind van het array. Met andere woorden, er is een index j zodat $A[1..j]$ van klein naar groot gesorteerd is en $A[j..n]$ van groot naar klein gesorteerd is. Geef een op binair zoeken gebaseerd algoritme dat, gegeven zo'n array A, de index j vindt. *NB: Dit is een lastige opgave.*

2.2 Efficiëntie-analyse

In het vorige hoofdstuk hebben we twee verschillende algoritmen gezien om een getal in een array te zoeken, *LineairZoeken* en *BinairZoeken*. Welk algoritme is sneller? Om dit te onderzoeken zouden we beide algoritmen kunnen uitprogrammeren, en dan gaan testen op een computer. Dit kan echter een tijdrovende bezigheid zijn, vooral als de algoritmen wat ingewikkelder worden. Gelukkig is uitprogrammeren en testen meestal niet nodig; door de pseudocode goed te bestuderen kunnen we vaak al veel zeggen over de snelheid van een algoritme.

Laten we eens kijken naar het algoritme *LineairZoeken*.

Algoritme *LineairZoeken*(A, q)
Invoer: Een array $A[1..n]$ met getallen, en een getal q.
Uitvoer: Een index i zodanig dat $A[i] = q$, of "niet aanwezig" als zo'n index niet bestaat.
1. $i \leftarrow 1$
2. zolang $i \leqslant n$
   doe: als $A[i] = q$
   dan Rapporteer i en stop
   anders $i \leftarrow i + 1$
3. Rapporteer "niet aanwezig"

Wat is de *looptijd* van dit algoritme, dat wil zeggen, hoeveel tijd heeft het algoritme nodig om tot een antwoord te komen? Het liefst zouden we de looptijd uitdrukken in seconden, maar op grond van alleen de pseudocode kunnen we daar weinig uitspraken over doen.[1] Daarom zullen we de looptijd van een algoritme uitdrukken in het aantal stappen dat het doet. Dat aantal hangt natuurlijk af van de grootte van de invoer: het zal allicht meer stappen kosten om een getal te zoeken in een array met 1.000.000 getallen dan in een array met 100 getallen. Met andere woorden, de looptijd van een algoritme is een functie van de invoergrootte. Maar zelfs voor een vaste invoergrootte kan de looptijd variëren. Zo kan algoritme *LineairZoeken* geluk hebben en het gezochte getal al in $A[1]$ tegenkomen, of het kan pech hebben en het hele array moeten doorzoeken. We kunnen dus kijken naar de *beste-geval looptijd* van een algoritme, of naar de *gemiddelde looptijd*, of naar de *slechtste-geval looptijd*.

Over het algemeen is de beste-geval looptijd niet zo interessant: deze wordt vaak bepaald door een toevalstreffer, en de kans daarop is meestal klein. De gemiddelde looptijd is nuttiger, maar

[1] Dit zou overigens ook niet goed kunnen als we het algoritme uitprogrammeren en testen: de precieze looptijd van een algoritme hangt namelijk niet alleen af van het algoritme zelf, maar ook bijvoorbeeld van de computer waarop het getest wordt en de gebruikte programmeertaal. Je kunt dus niet veel meer zeggen dan: "Mijn implementatie van het algoritme kost X seconden voor deze specifieke test op een computer van het type Y."

deze is vaak moeilijk te bepalen. Kijk bijvoorbeeld naar *LineairZoeken*. Als het gezochte getal in $A[1]$ staat dan heeft het algoritme één stap nodig, als het in $A[2]$ staat, dan heeft het twee stappen nodig, enzovoorts. Als we naar een getal zoeken waarvan we weten dat het in het array voorkomt, dan is het gemiddelde aantal stappen dus:

$$\frac{1}{n} \cdot (1 + 2 + \cdots + n) \;=\; \frac{1}{n} \sum_{i=1}^{n} i \;=\; \frac{1}{n} \cdot \frac{1}{2} n(n+1) \;=\; \frac{n+1}{2}.$$

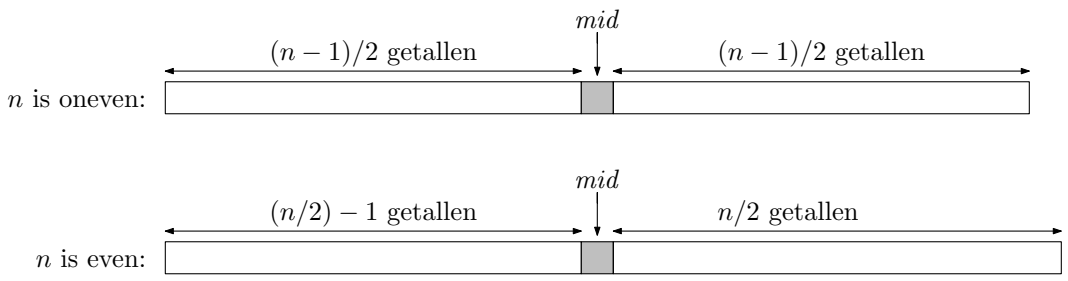
Als we zoeken naar een getal dat niet voorkomt, dan wordt altijd het hele array doorzocht en is het aantal stappen dus n. Maar wat is het gemiddelde als we niet weten of het gezochte getal voorkomt? Dat ligt ergens tussen $(n+1)/2$ en n, afhankelijk van de kans dat het gezochte getal voorkomt. Helaas is dit soort kans-informatie meestal niet beschikbaar, waardoor het bepalen van de gemiddelde looptijd niet goed mogelijk is. Daarom richt men zich bij het analyseren van de looptijd van een algoritme op de slechtste-geval looptijd. Een voordeel hiervan is dat dit een garantie geeft op de looptijd: slechter dan de slechtste-geval looptijd zal het (per definitie) niet worden. Voor *LineairZoeken* worden er in het slechtste geval n stappen gedaan.

Merk op dat de slechtste-geval looptijd van *LineairZoeken* ten hoogste twee keer zo groot is als de gemiddelde looptijd, onafhankelijk van de waarde van n. De beste-geval looptijd daarentegen is $(n+1)/2$ (of meer) keer zo klein als de gemiddelde looptijd. Als n groot is, dan is het verschil tussen de beste-geval looptijd en de gemiddelde looptijd dus erg groot. Dit is nog een reden om ons op de slechtste-geval looptijd te richten: in tegenstelling tot de beste-geval looptijd ligt die meestal vrij dicht bij de gemiddelde looptijd.

Laten we nu de looptijd van *BinairZoeken* eens proberen te analyseren.

Algoritme *BinairZoeken*(A, q)
Invoer: Een array $A[1..n]$ met getallen, gesorteerd van klein naar groot, en een getal q.
Uitvoer: Een index i zodanig dat $A[i] = q$, of "niet aanwezig" als zo'n index niet bestaat.
1. *links* $\leftarrow 1$; *rechts* $\leftarrow n$
2. zolang *links* $\leqslant$ *rechts*
   doe: *mid* $\leftarrow \lfloor (links + rechts)/2 \rfloor$
   als $q = A[mid]$
   dan Rapporteer *mid* en stop
   anders als $q < A[mid]$
   dan *rechts* $\leftarrow mid - 1$
   anders *links* $\leftarrow mid + 1$
3. Rapporteer "niet aanwezig"

Het algoritme kijkt eerst naar $A[mid]$ met $mid = \lfloor (n+1)/2 \rfloor$. Het beste geval treedt op als $q = A[mid]$; dan is het algoritme in één stap klaar. Anders wordt er verder gezocht in $A[1..mid-1]$ of in $A[mid+1..n]$. Zoals te zien in Fig. 4 hebben deze beide deelarrays grootte $(n-1)/2$ als n oneven is. Als n even is, dan heeft een van de beide deelarrays grootte $(n/2) - 1$ en de andere grootte $n/2$. In het slechtste geval zoeken we dus verder in een deelarray ter grootte $n/2$. In de tweede stap kijken we in het midden van dit deelarray, waarna we in het slechtste geval verder



Figuur 4: De grootte van de deelarrays waarin eventueel verder gezocht moet worden.

moeten zoeken in een deelarray ter grootte $n/4$. De grootte van het deelarray waarin we aan het zoeken zijn, neemt dus af volgens de volgende rij:

$$n \;\rightarrow\; n/2 \;\rightarrow\; n/4 \;\rightarrow\; n/8 \;\rightarrow\; \cdots$$

Dit gaat in het slechtste geval door tot het deelarray nog maar één getal bevat en dat getal ook ongelijk aan q blijkt te zijn. Hoeveel stappen duurt dat maximaal? Zoals je in de bovenstaande reeks kan zien, is het grootte van het deelarray na k stappen gereduceerd tot $n/2^k$. Dus na ${}^2\log n$ stappen is de grootte gereduceerd tot 1; immers, per definitie geldt $2^{{}^2\log n} = n$. We concluderen dat *BinairZoeken* in het slechtste geval ${}^2\log n + 1$ stappen nodig heeft om een getal te zoeken in een array met n getallen. Dat is een stuk beter dan de n stappen die *LineairZoeken* in het slechtste geval nodig heeft.

Opgave 2.2.1 Hoeveel stappen heeft *BinairZoeken* in het slechtste geval nodig om een getal te zoeken in een array met 1024 getallen?

Hierboven hebben we beredeneerd dat het aantal stappen dat binair zoeken in het slechtste geval nodig heeft ${}^2\log n$ is. Merk op dat ${}^2\log n + 1$ niet altijd een geheel getal is—dat is het alleen als n een 2-macht is—terwijl het aantal stappen dat natuurlijk wel moet zijn. Hieruit kunnen we concluderen dat het aantal stappen in het slechtste geval gelijk is aan $\lfloor {}^2\log n \rfloor + 1$.

Opgave 2.2.2 Hoeveel stappen heeft *BinairZoeken* in het slechtste geval nodig om een getal te zoeken in een array met 1.000.000 getallen?

Opgave 2.2.3 Hoe groot is het grootste array waarin *BinairZoeken* in maximaal 30 stappen klaar is?

De bovenstaande analyses van *LineairZoeken* en *BinairZoeken* vertellen ons hoeveel stappen deze algoritmes in het slechtste geval nodig hebben om te zoeken in aan array ter lengte n. Hierbij komt een stap overeen met het bekijken van een getal in het array. Voor *LineairZoeken* is dat n, voor *BinairZoeken* $\lfloor {}^2\log n \rfloor$. Eigenlijk is dat niet helemaal eerlijk: regel 2 in *LineairZoeken* is namelijk iets minder werk dan regel 3 in *BinairZoeken*. Dit kleine verschil weegt echter niet op tegen het verschil in het aantal stappen. Bij het analyseren van algoritmen verwaarlozen we dit soort verschillen daarom.

Opgave 2.2.4 Stel dat een stap in *BinairZoeken* drie keer zoveel tijd kost als een stap in *LineairZoeken*. Vergelijk de tijd die *BinairZoeken* in het slechtste geval nodig heeft om in een array met 100.000 getallen met de tijd die *LineairZoeken* in het slechtste geval nodig heeft in zo'n array. Hoeveel keer zo snel is *BinairZoeken*?

Opgave 2.2.5 Het volgende algoritme bepaalt, gegeven een array $A[1..n]$ en een getal q, of A twee getallen bevat waarvan de som precies q is. Analyseer het aantal stappen dat het algoritme in het slechtste geval nodig heeft.

Algoritme *ZoekPaar*(A, q)
Invoer: Een array $A[1..n]$ van n getallen, en een getal q.
Uitvoer: Indices i en j met $i < j$ zodanig dat $A[i] + A[j] = q$, of "niet aanwezig" als zulke indices niet bestaan.
1. $i \leftarrow 1$
2. zolang $i \leqslant n - 1$
   doe: $j \leftarrow i + 1$
   zolang $j \leqslant n$
   doe: als $A[i] + A[j] = q$
   dan Rapporteer i, j en stop
   anders $j \leftarrow j + 1$
   $i \leftarrow i + 1$
3. Rapporteer "niet aanwezig"

Opgave 2.2.6 In de vorige opgaven hebben we een algoritme gezien dat, gegeven een array $A[1..n]$ en een getal q, bepaalt of A twee getallen bevat waarvan de som precies q is. Het algoritme heeft in het slechtste geval $(n - 1)n/2$ stappen nodig. Als het array A gesorteerd is, dan is er een efficiëntere oplossing:

Algoritme *ZoekPaar2*(A, q)
Invoer: Een array $A[1..n]$ met getallen, gesorteerd van klein naar groot, en een getal q.
Uitvoer: Indices i en j met $i < j$ zodanig dat $A[i] + A[j] = q$, of "niet aanwezig" als zulke indices niet bestaan.
1. $i \leftarrow 1$
2. zolang $i \leqslant n - 1$
   doe: $j \leftarrow$ *BinairZoeken*$(A, q - A[i])$
   als $j \neq$ "niet aanwezig" en $j > i$
   dan Rapporteer i, j en stop.
   anders $i \leftarrow i + 1$
3. Rapporteer "niet aanwezig"

Analyseer het aantal stappen dat *ZoekPaar2* in het slechtste geval nodig heeft.

Opgave 2.2.7 Bekijk weer het probleem van de vorige opgave: gegeven een gesorteerd array $A[1..n]$ en een getal q, bepaal of er indices i, j zijn (met $i < j$) zodanig dat $A[i] + A[j] = q$. Geef een algoritme dat dit probleem in n stappen oplost.
Hint: Stel dat $A[1] + A[n] < q$. Welke conclusie kan je daaruit trekken? En welke conclusie kan je trekken als $A[1] + A[n] > q$?

O-notatie

Als maat voor de looptijd van een algoritme hebben we hierboven het aantal stappen gebruikt dat het algoritme (in het slechtste geval) nodig heeft voor een invoer ter lengte n. Maar wat is nu precies een stap? Eigenlijk zouden we misschien het aantal basis-operaties—optellingen, vermenigvuldigingen, vergelijkingen, enzovoorts—-moeten tellen. Door het algoritme net wat anders op te schrijven kan dit aantal echter veranderen. Daarom wordt in de informatica niet het precieze aantal basis-operaties geteld, maar wordt er alleen gekeken naar het verband tussen de invoergrootte, n, en het aantal operaties: is dit een logaritmisch verband (zoals bij binair zoeken), een linear verband (zoals bij lineair zoeken), of bijvoorbeeld een kwadratisch verband. Het is dan niet meer van belang of het aantal basis-operaties ${}^2\log n$ is, of $2 \cdot ({}^2\log n)$, of bijvoorbeeld $5 \cdot ({}^2\log n)$; in al deze gevallen is het verband logaritmisch. Om dit aan te geven wordt de zogenaamde O-notatie gebruikt. Om een logaritmisch verband aan te geven schrijven we dan dat de looptijd $O(\log n)$ is. De O-notatie heeft een precieze wiskundige betekenis, waar we hier verder niet op in zullen gaan.

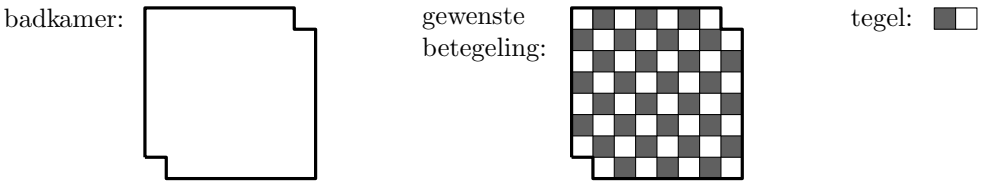
3 Correctheid van algoritmen

Stel dat je een algoritme hebt bedacht om een bepaalde taak uit te voeren. Een van de vragen die je je dan kunt stellen is: hoe *efficiënt* is het algoritme, dat wil zeggen, hoeveel stappen heeft het nodig om de taak uit te voeren? Naar dit aspect hebben in het vorige hoofdstuk gekeken. Een andere vraag, die daar eigenlijk aan voorafgaat, is: is het algoritme *correct*, dat wil zeggen, doet het wat het moet doen?

Van sommige algoritmen is de correctheid duidelijk. Een voorbeeld hiervan is *LinearZoeken*. Dat loopt een voor een alle getallen in het array af, tot het getal gevonden is of alle getallen gecontroleerd zijn. Het moge duidelijk zijn dat dit een correcte strategie is. Overigens hoeft dat nog niet te betekenen dat de pseudocode correct is: als we bijvoorbeeld in regel 2 van *LinearZoeken* per ongeluk $i < n$ schrijven in plaats van $i \leqslant n$, dan zou het algoritme $A[n]$ niet controleren en dus incorrect zijn als $q = A[n]$. Voor andere algoritmen is het zelfs niet duidelijk of de gekozen strategie wel correct is. In dit hoofdstuk zullen we wat dieper ingaan op hoe je kunt bewijzen dat een algoritme correct is. We kijken hierbij vooral naar het gebruik van zogenaamde invarianten.

3.1 Invarianten

Stel dat we de vloer van de badkamer in Fig. 5 willen betegelen zodat er een soort schaakbordpatroon ontstaat. De tegels die we tot onze beschikking hebben zijn rechthoekig, en elke tegel is half wit en half zwart. Is het mogelijk om de badkamer uit de figuur helemaal te betegelen met deze tegels, zonder sommige tegels in tweeën te moeten breken? Een tijdje proberen levert



Figuur 5: De badkamervloer, de gewenste betegeling en de beschikbare tegel.

waarschijnlijk weinig op: wat je ook probeert, je zult ergens vast lopen. Misschien is het dus helemaal niet mogelijk. Maar hoe kunnen we dat zeker weten? Hier komt het begrip invariant ons te hulp.

Stel je voor dat we de badkamer aan het betegelen zijn. Hoe we dat ook doen, als we alleen hele tegels gebruiken dan zal het aantal zwarte vierkanten en het aantal witte vierkanten dat we gelegd hebben telkens even groot zijn. Immers, we beginnen met nul zwarte en witte vierkanten, en bij elke tegel die we leggen komt er precies één zwart en één wit vierkant bij. Dus de gelijkheid

aantal gelegde zwarte vierkanten = aantal gelegde witte vierkanten

is een *invariant* van het betegelen: het is een eigenschap die niet verandert tijdens het proces. Als we de gewenste betegeling goed bekijken, dan zien we dat er minder zwarte vierkanten zijn dan witte. De invariant vertelt ons dus dat we deze situatie nooit kunnen bereiken.

Opgave 3.1.1 Stel dat we in plaats van de rechthoekige tegels uit Fig. 5 de volgende twee L-vormige tegels hebben:  en  .
Kan de badkamer uit Fig. 5 met deze tegels betegeld worden?

Opgave 3.1.2 De EI-taal is een taal met "woorden" die bestaan uit de letter E en I. Een van de woorden in de EI-taal is EI. De andere woorden kunnen uit het woord EI gemaakt worden door het een of meer keer toepassen van de volgende regels:

- *Regel 1:* Als het woord eindigt op een I, mag je er een E achter zetten.
- *Regel 2:* Je mag het woord verdubbelen door het achter zichzelf te plaatsen.
- *Regel 3:* Als er ergens een E staat, mag je die wegstrepen.
- *Regel 4:* Als er ergens drie I's achter elkaar staan, dan mag je deze allemaal wegstrepen. (Het is niet toegestaan maar een of twee van de I's weg te strepen.)

Zo kun je uit EI de woorden EIE (regel 1), EIEI (regel 2), en I (regel 3) maken. Uit bijvoorbeeld EIE kan je daarna EIEEIE, EI, en IE maken, enzovoorts.

(i) Kan het woord EEII gemaakt worden?

(ii) Kan het woord III gemaakt worden?

3.2 Invarianten en algoritmen

De kern van een algoritme wordt meestal gevormd door een of meer herhalings-opdrachten (opdrachten van de vorm zolang ... doe: ...). Invarianten kunnen gebruikt worden om te beargumenteren dat een herhalings-opdracht het gewenste resultaat oplevert.

Stel dat we, gegeven een getal x en een geheel getal $n \geqslant 1$, het getal x^n uit willen rekenen. Een eenvoudige manier om dit te doen is een hulp-variabele z de waarde 1 te geven en daarna n keer met x te vermenigvuldigen. Op deze manier zijn er n stappen nodig om x^n uit te rekenen. Het kan sneller:

Algoritme *Machtsverheffen*(x, n)
Invoer: Een getal x, en een geheel getal $n \geqslant 1$.
Uitvoer: x^n
1. $i \leftarrow n$; $y \leftarrow x$; $z \leftarrow 1$
2. zolang $i > 0$
   doe: als i is even
   dan $y \leftarrow y^2$; $i \leftarrow i/2$
   anders $z \leftarrow z \cdot y$; $i \leftarrow i - 1$
3. Rapporteer z

Dit algoritme ziet er in eerste instantie wat mysterieus uit. Laten we eens kijken hoe de waarden van de variabelen i, y, en z veranderen in de loop van het algoritme:

i	y	z	
10	2	1	$i > 0$ en i is even, dus $y \leftarrow y^2$; $i \leftarrow i/2$
5	4	1	$i > 0$ en i is oneven, dus $z \leftarrow z \cdot y$; $i \leftarrow i - 1$
4	4	4	$i > 0$ en i is even, dus $y \leftarrow y^2$; $i \leftarrow i/2$
2	16	4	$i > 0$ en i is even, dus $y \leftarrow y^2$; $i \leftarrow i/2$
1	256	4	$i > 0$ en i is oneven, dus $z \leftarrow z \cdot y$; $i \leftarrow i - 1$
0	256	1024	$i = 0$, dus ga naar regel 3 en rapporteer 1024.

Voor $x = 2$ en $n = 10$ werkt het algoritme dus. Hoe zit het met het aantal stappen dat *Machtsverheffen* nodig heeft?

Opgave 3.2.1 Beargumenteer dat *Machtsverheffen* ten hoogste $2 \cdot ({}^2\log n)$ stappen nodig heeft, dus dat regel 2 ten hoogste $2 \cdot ({}^2\log n) + 1$ keer wordt uitgevoerd.

Opgave 3.2.1 zegt dat *Machtsverheffen* $2 \cdot ({}^2\log n) + 1$ stappen nodig heeft—een stuk minder dan n stappen (als n groot is). We hebben ook al gezien dat het algoritme voor $x = 2$ en $n = 10$ het juiste resultaat oplevert. Maar is dat eigenlijk altijd het geval? We kunnen het algoritme natuurlijk nog een paar keer uitvoeren, met andere waarden van x en n. Als telkens het juiste resultaat wordt opgeleverd, krijgen we er waarschijnlijk meer vertrouwen in dat *Machtsverheffen* inderdaad correct is. Maar zeker weten doen we het op deze manier niet. Daarvoor zullen op de een of andere manier moeten *bewijzen* dat het algoritme *voor elke waarde* van x en n (met $n > 1$) het juiste resultaat oplevert. Zo'n bewijs zullen we nu gaan geven, met behulp van een invariant.

We beweren dat $z \cdot y^i = x^n$ een invariant van het algoritme is. Preciezer: elke keer dat regel 2 wordt uitgevoerd, geldt $z \cdot y^i = x^n$ voordat de keuze-opdracht "als i is even dan ... anders ..." wordt uitgevoerd, en nadat die is uitgevoerd. Voor de duidelijkheid herhalen we het algoritme nog een keer, met de invariant expliciet aangegeven.

Algoritme *Machtsverheffen*(x, n)
Invoer: Een getal x, en een geheel getal $n \geqslant 1$.
Uitvoer: x^n
1. $i \leftarrow n$; $y \leftarrow x$; $z \leftarrow 1$
2. zolang $i > 0$
   doe: { Invariant: $z \cdot y^i = x^n$ }
   als i is even
   dan $y \leftarrow y^2$; $i \leftarrow i/2$
   anders $z \leftarrow z \cdot y$; $i \leftarrow i - 1$
   { Invariant: $z \cdot y^i = x^n$ }
3. Rapporteer z

Het bewijzen dat $z \cdot y^i = x^n$ inderdaad een invariant is, gaat als volgt:

- De invariant geldt de eerste keer dat regel 2 wordt uitgevoerd. Immers, dan is $i = n$, $y = x$, en $z = 1$, dus inderdaad geldt $z \cdot y^i = x^n$.
- Als de invariant geldt voordat de keuze-opdracht in regel 2 wordt uitgevoerd, dan geldt de invariant ook nadat de keuze-opdracht is uitgevoerd.

  Om dit te laten zien moeten we de twee gevallen in de keuze-opdracht apart bekijken. Als i even is, dan wordt $y \leftarrow y^2$; $i \leftarrow i/2$ uitgevoerd, terwijl z niet verandert. Omdat $(y^2)^{i/2} = y^i$ geldt na deze veranderingen van y en i nog steeds dat $z \cdot y^i = x^n$

  Als i oneven is, dan wordt $z \leftarrow z \cdot y$; $i \leftarrow i - 1$ uitgevoerd, terwijl y niet verandert. Omdat $(z \cdot y) \cdot y^{i-1} = y^i$ geldt na deze veranderingen van z en i ook nog steeds dat $z \cdot y^i = x^n$

Dus de invariant geldt in het begin, en blijft daarna steeds gelden. Met behulp van de invariant kunnen we nu laten zien dat het algoritme het gewenste resultaat oplevert:

- Merk op dat de herhalings-opdracht eindigt als $i = 0$. (Dit punt wordt inderdaad bereikt, want i wordt minstens 1 kleiner elke keer de regel 2 wordt uitgevoerd.) Als we $i = 0$ invullen in de invariant, dan krijgen we $z \cdot y^0 = x^n$. Dus $z = x^n$ na afloop van regel 2. We concluderen dat het algoritme inderdaad correct is.

Opgave 3.2.2 Is *Machtsverheffen* ook correct voor $n = 0$?

Opgave 3.2.3 Het volgende algoritme rekent x^n uit op de al eerder geschetste manier, namelijk door een hulp-variabele z de waarde 1 te geven en daarna n keer met x te vermenigvuldigen.

Algoritme *SimpelMachtsverheffen*(x, n)
Invoer: Een getal x, en een geheel getal $n \geqslant 1$.
Uitvoer: x^n
1. $z \leftarrow 1$; $i \leftarrow 1$
2. zolang $i \leqslant n$
   doe: { Invariant: ?? }
   $z \leftarrow z \cdot x$; $i \leftarrow i + 1$
   { Invariant: ?? }
3. Rapporteer z

Beargumenteer met behulp van een invariant dat dit algoritme correct is. Dus: (i) geef de invariant, (ii) laat zien dat de invariant geldt op het moment dat de herhalings-opdracht voor het eerst wordt uitgevoerd, (iii) laat zien dat de invariant behouden blijft telkens als de herhalings-opdracht wordt uitgevoerd, en (iv) laat zien dat de correctheid van het eindresultaat volgt uit het feit dat de invariant na afloop van de herhalings-opdracht geldt.

Opgave 3.2.4 Ook voor de zoekalgoritmen in het vorige hoofdstuk kunnen we de correctheid met behulp van een invariant bewijzen. Voor *LineairZoeken* op pagina 1 kan dat met de invariant: *Het gezochte getal q staat niet in $A[j]$, voor $1 \leqslant j < i$.* Wat is de invariant voor het algoritme *BinairZoeken* op pagina 4?

Foute software

Je hebt het vast wel eens meegemaakt: de computer "hangt" en is alleen weer aan de praat te krijgen door 'm opnieuw op te starten. Ook in computer games gebeuren er soms dingen die niet echt de bedoeling lijken. Blijkbaar valt het niet mee om foutloze software te schrijven. Bij een game is het niet zo'n ramp als er eens iets mis gaat, maar helaas gaan er ook in cruciale gevallen soms dingen verkeerd. Zo overleden in de jaren tachtig tenminste vijf patiënten omdat zij te veel straling toegediend kregen door een fout in de software van een röntgen-appararaat, en explodeerde op 4 juni 1996 de Ariane 5 raket als gevolg van een software-fout. Het op een wiskundige manier correct bewijzen van algoritmen is dus zeker geen verspilde moeite.

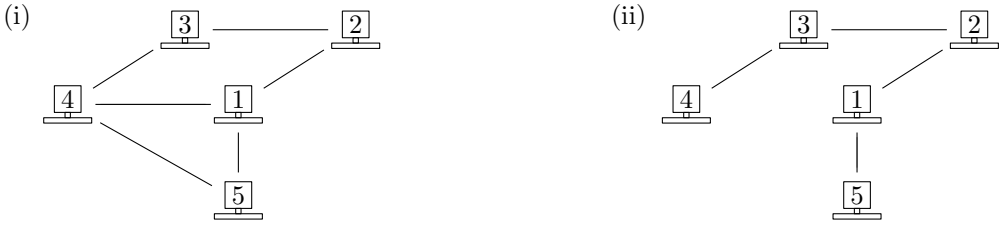
4 Makkelijke en moeilijke problemen

In de vorige hoofdstukken hebben we gezien dat er vaak verschillende algoritmen zijn om een bepaalde taak uit te voeren. Het ene algoritme kan daarbij veel efficiënter zijn dan het andere. Het loont daarom de moeite om niet direct tevreden te zijn met de meest voor de hand liggende aanpak; door nog wat verder te puzzelen kan vaak een efficiënter algoritme gevonden worden. Misschien heb je het idee dat dit niet echt nodig is, omdat computers tegenwoordig zo snel zijn dat het niet veel uitmaakt of een algoritme efficiënt is. In dit hoofdstuk zullen we zien dat dat niet waar is: we zullen twee problemen bekijken waarbij de voor de hand liggende aanpak veel en veel te traag is.

De twee problemen die we zullen bestuderen gaan allebei over *netwerken*.[2] Netwerken—wegennetwerken, computernetwerken, sociale netwerken, enzovoorts—spelen een belangrijke rol in het dagelijks leven. Het eerste probleem gaat over het aanleggen van goedkope netwerken, het tweede over het vinden van een korte rondrit in bestaande netwerken. Hoewel de problemen veel op elkaar lijken, blijken ze toch van een heel andere moeilijkheidsgraad te zijn: voor het vinden van het goedkoopste netwerk bestaat er een efficiënt algoritme, voor het vinden van de kortste rondrit is er geen efficiënt algoritme bekend. Het vermoeden bestaat zelfs dat er geen efficiënt algoritme *kan* bestaan voor het vinden van de kortste rondrit.

4.1 Het goedkoopste samenhangende netwerk

Stel je voor dat we een aantal computers met elkaar willen verbinden in een netwerk. We kunnen een rechtstreekse verbinding aanleggen tussen elk paar computers, maar dat is duur. Het is ook niet nodig, want een indirecte verbinding tussen twee computers volstaat om ze te kunnen laten communiceren. Het is daarom voldoende als het netwerk *samenhangend* is, d.w.z. als er een pad is in het netwerk tussen elk paar computers. In Fig. 6 zie je twee voorbeelden van een samenhangend netwerk.



Figuur 6: Twee samenhangende netwerken van vijf computers.

Bij het aanleggen van het netwerk zullen sommige verbindingen duurder zijn dan andere, bijvoorbeeld omdat de afstand tussen de computers groot is. De vraag is dus welke verbindingen we moeten aanleggen om een zo goedkoop mogelijk samenhangend netwerk te krijgen. Het is duidelijk dat netwerk (i) in Fig. 6 niet optimaal is, want er kunnen verbindingen worden weggelaten zonder dat de samenhang van het netwerk verbroken wordt. Of netwerk (ii) optimaal is hangt af van de kosten van de verschillende verbindingen. We kunnen deze kosten specificeren met behulp van een kostentabel:

	computer 1	computer 2	computer 3	computer 4	computer 5
computer 1	–	33	24	34	15
computer 2	33	–	35	47	41
computer 3	24	35	–	31	39
computer 4	34	47	31	–	40
computer 5	15	41	39	40	–

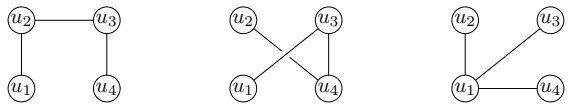
[2]In de wiskunde en de informatica worden netwerken meestal *grafen* genoemd, en de verbindingen tussen de knopen in het netwerk *kanten*.

Uit deze kostentabel kunnen we afleiden dat het voordelig is om de verbinding tussen computers 1 en 2 (kosten: 33) te vervangen door de verbinding tussen 1 en 3 (kosten: 24). De totale netwerk kosten van het netwerk zijn dan 34+35+24+15=108. Zou er een nog goedkoper netwerk zijn? Dat is moeilijk te bepalen, zeker als het aan te leggen netwerk nog wat groter zou zijn. We willen daarom een algoritme ontwikkelen dat het goedkoopste samenhangende netwerk berekent.

In een meer abstracte vorm ziet het probleem waar we een algoritme voor willen ontwikkelen er dus als volgt uit. We hebben een verzameling van n *knopen*—in het bovenstaande voorbeeld waren de knopen computers—waarop we een zo goedkoop mogelijk samenhangend netwerk willen berekenen. De knopen geven we aan met $u_1, \ldots, u_n$, en de verbinding tussen knopen u_i en u_j geven we aan met (u_i, u_j). De kosten van de verbindingen staan in een kostentabel $K[1..n, 1..n]$, waarbij $K[i, j]$ de kosten van de verbinding (u_i, u_j) zijn. De invoer van het algoritme is de kostentabel, en de gevraagde uitvoer is een lijst van de verbindingen in een samenhangend netwerk waarvan de kosten minimaal zijn.

We nemen vanaf nu aan dat alle kosten positief zijn. Dat betekent dat een netwerk nooit optimaal kan zijn als een verbinding weggelaten kan worden zonder de samenhang te verliezen. Een samenhangend netwerk zonder dergelijke overbodige verbindingen wordt ook wel een *opspannende boom* genoemd.

Opgave 4.1.1 Hieronder staan drie opspannende bomen op een verzameling van vier knopen getekend:



Teken alle opspannende bomen op de vier knopen. Hoeveel bomen zijn het in totaal? Beredeneer dat dit het juiste aantal is.

Het probleem dat we willen oplossen is het berekenen van de opspannende boom met minimale kosten, oftewel de *minimale opspannende boom*. Een voor de hand liggende aanpak is om van alle mogelijke opspannende bomen de kosten te berekenen.

Opgave 4.1.2 Stel dat we een algoritme hebben dat alle opspannende bomen op n knopen kan genereren. We willen dit algoritme gebruiken om een minimale opspannende boom te vinden. We hebben daartoe een bijzonder snelle computer tot onze beschikking, die 1.000.000.000 opspannende bomen per seconde kan doorrekenen.

Het is bekend dat het aantal opspannende bomen op n knopen gelijk is aan n^{n-2}. Hoeveel tijd heeft het algoritme nodig voor het doorrekenen van alle opspannende bomen voor een verzameling van 10 computers? En voor 20 computers?

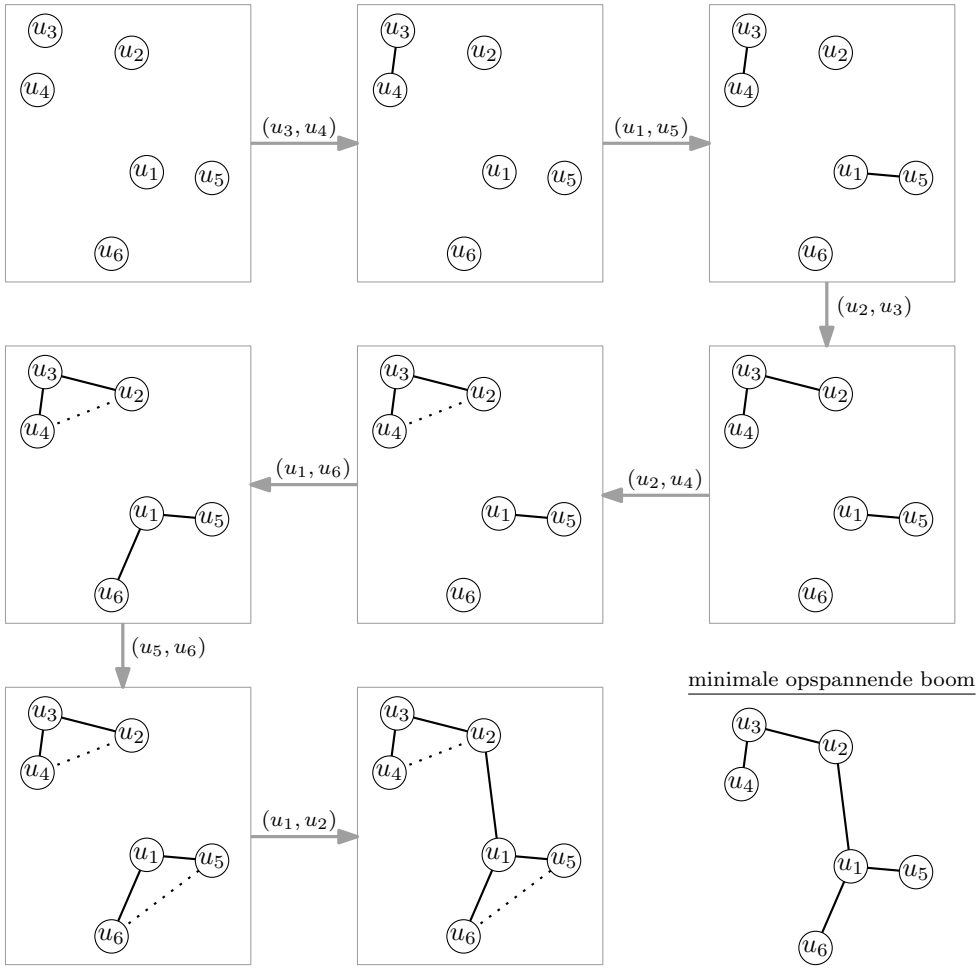
Uit de vorige opgave blijkt dat het domweg doorrekenen van alle mogelijke opspannende bomen ondoenlijk is, tenzij het aantal knopen erg klein is. We zullen slimmer moeten zijn.

Ons nieuwe algoritme is gebaseerd op het volgende idee. Omdat we een zo goedkoop mogelijke boom willen, gaan we alle potentiële verbindingen bekijken op volgorde van hun kosten: eerst de goedkoopste verbinding, dan de een-na-goedkoopste, en zo verder. Elke keer als we een verbinding (u_i, u_j) bekijken, gaan we na of die verbinding nog nodig is. Dit is het geval als u_i en u_j elkaar nog niet kunnen bereiken d.m.v. de eerder toegevoegde verbindingen. Is de verbinding (u_i, u_j) nodig, dan voegen we hem toe aan het netwerk.

Fig. 7 laat zien hoe het algoritme werkt als het wordt uitgevoerd op zes knopen met de volgende kostentabel. (De onderste helft van de tabel is weggelaten, zodat de verbindingen er niet dubbel in staan.)

	u_1	u_2	u_3	u_4	u_5	u_6
u_1	–	47	52	49	20	31
u_2		–	28	31	51	60
u_3			–	17	57	62
u_4				–	55	56
u_5					–	44
u_6						–

Eerst wordt de goedkoopste verbinding, (u_3, u_4), bekeken. Knoop u_3 en u_4 kunnen elkaar nog niet bereiken, dus (u_3, u_4) wordt toegevoegd aan het netwerk. Daarna wordt de een-na-goedkoopste verbinding, (u_1, u_5), bekeken. Ook u_1 en u_5 kunnen elkaar nog niet bereiken, dus (u_1, u_5) wordt toegevoegd. Dan wordt (u_2, u_3) bekeken en toegevoegd. Verbindingen (u_2, u_4) en (u_1, u_6) hebben allebei kosten 31. Het algoritme behandelt verbindingen met gelijke kosten in willekeurige volgorde; in dit geval wordt (u_2, u_4) eerst behandeld. Omdat u_2 en u_4 elkaar al kunnen bereiken, namelijk via u_3, wordt (u_2, u_4) niet toegevoegd. Daarna wordt (u_1, u_6) wel toegevoegd, (u_5, u_6) niet, en (u_1, u_2) weer wel. Op dat moment is het hele netwerk samenhangend en kan het algoritme termineren.



Figuur 7: Het algoritme van Kruskal aan het werk. Niet toegevoegde verbindingen worden gestippeld weergegeven.

Het bovenstaande algoritme wordt het algoritme van Kruskal genoemd, naar de Amerikaan Joseph Kruskal die het algoritme in 1956 publiceerde. In pseudocode ziet Kruskal's algoritme er als volgt uit.

Algoritme *MinimaleOpspannendeBoom*($K[1..n, 1..n]$)
Invoer: Een kostentabel $K[1..n, 1..n]$, waarbij $K[i, j]$ de kosten van de verbinding (u_i, u_j) aangeeft.
Uitvoer: Een een minimale opspannende boom B op de knopen $u_1, \ldots, u_n$.
1. Sorteer de verbindingen op kosten (van lage naar hoge kosten).
2. $B \leftarrow$een netwerk met knopen $u_1, \ldots, u_n$ en zonder verbindingen
3. zolang B nog niet samenhangend is
   doe: Neem de volgende verbinding (u_i, u_j) uit de gesorteerde lijst.
      als u_i en u_j elkaar nog niet kunnen bereiken
      dan voeg (u_i, u_j) toe aan B
4. Rapporteer B.

Opgave 4.1.3 Voer Kruskal's algoritme uit op het voorbeeld van de vijf computers.

Opgave 4.1.4 Beredeneer dat het niet kan gebeuren dat het netwerk B nog steeds niet samenhangend is nadat alle verbindingen bekeken zijn.

Uit de vorige opgave blijkt dat het netwerk B dat Kruskal's algoritme oplevert samenhangend is. Er kunnen ook geen verbindingen worden weggelaten uit B zonder de samenhang te verliezen. B is dus een opspannende boom. Maar is B ook een *minimale* opspannende boom? Het antwoord hierop is ja: Kruskal's algoritme levert altijd een minimale opspannende boom op. Dit kan bewezen worden met behulp van een invariant, maar omdat het bewijs vrij ingewikkeld is zullen we het hier niet behandelen.

De pseudocode hierboven is nog niet zo gedetailleerd dat die gemakkelijk omgezet kan worden in een echte programmeertaal. Het sorteren is bijvoorbeeld een probleem op zich, en het is ook niet direct duidelijk hoe we in stap 3 kunnen testen of u_i en u_j elkaar al kunnen bereiken. Als dit op een goede manier wordt gedaan, dan kost Kruskal's algoritme ongeveer $n^2 \cdot ({}^2\log n)$ stappen.

Opgave 4.1.5 Stel dat we Kruskal's algoritme uitvoeren op een computer die 1.000.000 stappen per second kan uitvoeren, en dat het algoritme $n^2 \cdot ({}^2\log n)$ stappen doet. Hoeveel tijd heeft het algoritme nodig voor 20 knopen? En voor 1.000 knopen?

Vergelijk je antwoord met het antwoord van opgave 4.1.2. Merk op dat de computer uit opgave 4.1.2 wel 1.000.000.000 netwerken per seconde kon doorrekenen, en dus veel en veel sneller is.

We concluderen dat we door een iets slimmer algoritme te gebruiken een enorme winst hebben geboekt: het domweg doorrekenen van alle opspannende bomen is zelfs voor bijvoorbeeld 20 knopen ondoenbaar, terwijl Kruskal's algoritme ook voor bijvoorbeeld 1.000 knopen nog prima te doen is. Het probleem dat we in de volgende sectie bekijken zal minder eenvoudig blijken.

4.2 De kortste rondrit door een netwerk

Stel dat je een verkoper bent die klanten door heel Nederland moet bezoeken. De vraag is in welke volgorde je de klanten het beste kunt bezoeken. Als je bijvoorbeeld in Eindhoven woont en klanten in Leeuwarden, Breda en Groningen hebt, dan is de route Eindhoven–Groningen–Breda–Leeuwarden–Eindhoven niet handig: Eindhoven–Breda–Groningen–Leeuwarden–Eindhoven zal een stuk minder reistijd geven. Het berekenen van de kortste rondrit door een aantal steden wordt het *handelsreizigers-probleem* genoemd. (In het Engels: *Traveling Salesman Problem*, of kortweg *TSP*.)

De invoer van het handelsreizigers-probleem is een verzameling van n steden, en een afstandstabel tussen de steden. De gevraagde uitvoer is een rondrit door de steden (d.w.z. een route die alle steden bezoekt en daarna weer in het beginpunt uitkomt) van minimale lengte. In het voorbeeld hierboven zou de afstandstabel er als volgt uit zien:

	Eindhoven	Groningen	Breda	Leeuwarden
Eindhoven	–	235	57	232
Groningen	235	–	250	58
Breda	57	250	–	228
Leeuwarden	232	58	228	–

Met behulp van de afstandentabel kunnen we berekenen dat de route Eindhoven–Groningen–Breda–Leeuwarden–Eindhoven in totaal 945 km is, terwijl de route Eindhoven–Breda–Groningen–Leeuwarden–Eindhoven maar 597 km is. De route Eindhoven–Breda–Leeuwarden–Groningen–Eindhoven is zelfs nog korter, namelijk 578 km; dit is hier de kortst mogelijke rondrit, en dus de gevraagde uitvoer.

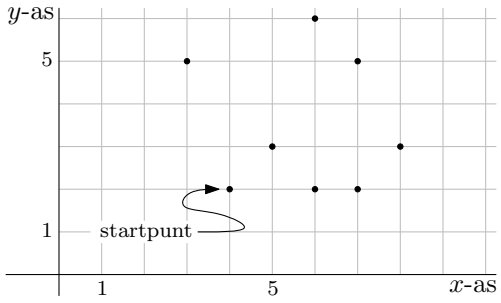
Opgave 4.2.1 Stel dat we een rondrit willen vinden door n steden, waarbij het startpunt van de rondrit (dit is een van de n steden) vastligt. Wat is het aantal verschillende rondritten?

Neem nu aan dat $n = 20$. Hoeveel tijd is er nodig om van alle rondritten de lengte te berekenen, als er 1.000.000.000 rondritten per seconde doorgerekend kunnen worden?

We zitten dus met hetzelfde probleem als bij het berekenen van de minimale opspannende boom: alle mogelijke oplossingen doorrekenen is niet haalbaar, tenzij het netwerk erg klein is.

Laten we eens proberen of we een sneller algoritme kunnen bedenken. Een voor de hand liggend idee is om vanaf het startpunt van de rondrit telkens naar de dichtstbijzijnde stad te gaan die nog niet bezocht is. In het bovenstaande voorbeeld gaan we dan vanaf het startpunt, Eindhoven, eerst naar Breda. Vanaf Breda is Leeuwarden de dichtstbijzijnde stad die nog niet bezocht. Vanaf Leeuwarden blijft dan alleen nog Groningen over, waarna alle steden bezocht zijn en we naar Eindhoven terugkeren. Dit levert dus de rondrit Eindhoven–Breda–Leeuwarden–Groningen–Eindhoven, die inderdaad optimaal is. Maar zou deze strategie altijd een kortste rondrit opleveren? Het antwoord is, helaas, nee.

Opgave 4.2.2 Stel dat we de steden representeren door punten in het platte vlak, en dat de afstand tussen twee steden gelijk is aan de afstand tussen de punten. Hieronder zie je een voorbeeld van een dergelijke invoer.



(i) Voer de hierboven beschreven strategie uit op de gegeven verzameling punten.

(ii) Wordt voor een gegeven verzameling punten altijd dezelfde rondrit gevonden, of hangt dit van het startpunt af?

(iii) Geef een voorbeeld van een verzameling punten waarvoor de strategie geen kortste rondrit oplevert.

(iv) Bewijs dat een optimale rondrit op een verzameling punten in het platte vlak zichzelf nooit kan kruisen.

De bovenstaande strategie is dus weliswaar efficiënt—het is niet moeilijk om een bijbehorend algoritme te schrijven dat in n^2 stappen werkt—maar levert niet altijd een optimaal resultaat. Is er misschien een andere strategie, die ook efficiënt is, maar wel altijd een kortste rondrit oplevert? Het antwoord hierop is niet bekend: men vermoedt dat er geen algoritme kan bestaan dat een kortste rondrit snel kan uitrekenen voor elke verzameling van n steden. Preciezer gezegd: men vermoedt dat er geen algoritme kan bestaan waarvan het aantal stappen polynomiaal is in n (dus bijvoorbeeld n^2 of n^3). Maar niemand heeft kunnen bewijzen dat dit inderdaad onmogelijk is. Gelukkig zijn er wel algoritmen bedacht die gegarandeerd een rondrit opleveren die bijna optimaal is. Zo zijn er bijvoorbeeld algoritmen waarvan bewezen kan worden dat ze een rondrit geven die maximaal 1% langer is dan een optimale rondrit. In de praktijk zijn dit soort *benaderingsalgoritmen* meestal goed genoeg.

P≠NP?

Het handelsreizigersprobleem is niet het enige probleem waarvan het onbekend is of er een algoritme voor bestaat dat in polynomiale tijd werkt; er is een hele klasse van dat soort problemen, die allemaal nauw met elkaar samenhangen. Deze klasse van problemen wordt met NP aangeduid. De klasse van problemen die in polynomiale tijd kunnen worden opgelost wordt met P aangeduid. Men vermoedt dat de problemen uit de NP-klasse niet in polynomiale tijd kunnen worden opgelost, dus dat P≠NP, maar niemand heeft dit nog kunnen bewijzen. Het P≠NP-vraagstuk is het beroemdste probleem uit de informatica; op het oplossen staat een prijs van 1 miljoen dollar.

Het Halting Problem

Stel je voor dat we van een algoritme *Alg* willen bepalen of het eindigt of in een oneindige herhaling terecht komt. Dit probleem wordt het *halting problem* genoemd. Als *Alg* erg ingewikkeld is, is het niet eenvoudig in te zien of het eindigt. Het zou dan handig zijn om een algoritme te hebben dat als invoer het algoritme *Alg* heeft, en dan bepaalt of *Alg* eindigt. Helaas is dit onmogelijk: al in 1936 bewees de Brit Alan Turing dat er geen algoritme kan bestaan dat het *halting problem* voor elk invoer-algoritme *Alg* in eindige tijd oplost. Met andere woorden: elk algoritme voor het *halting problem* zal zelf soms ook in een oneindige herhaling komen!