Algoritmen en datastructuren

Daan Pape
Universiteit Gent

1 juni 2012

1 Inleiding

Opdat men van een algoritme zou kunnen spreken moeten volgende voorwaarden vervult zijn:

- correct
- bestaat uit concrete stappen
- is ondubbelzinnig
- kan in een eindig aantal stappen worden beschreven
- moet eindigen

De correctheid van algoritmen bewijzen is zelf **geen algoritmisch** probleem en men gebruikt bijgevolg **formele wiskundige techtnieken** zoals contradictie, inductie en lusinvarianten. Dit blijkt in de praktijk echter niet steeds mogelijk, men gaat dan over tot het testen van waarschijnlijke inputdata.

Een algoritme is **efficiënt** als het een probleem binnen de vooropgestelde beperkingen en resources oplost. Met de **kost** van een algoritme worden de gebruikte resources bedoelt, meestal wordt dit in termen van 1 resource, zoals tijd, uitgedrukt.

Een **datastructuur** is een voorstelling van gegevens samen met de bewerkingen die op die gegevens toegelaten zijn. Meestal worden deze zodanig gemaakt dat de inhoudt er niet toe doet en er enkel beschreven wordt wat ze doen, ze worden dus **abstract** gedefiniëerd en worden **abstracte datatypes of ADT's** genoemd. Zo wordt in Java de ADT via een interface gespecifiëerd en de datastructuur zelf is dan een klasse.

2 Analyse van algoritmen

Om te vermijden dat men empirische tests moet doen, en dus de algoritmen die men wil vergelijken moet implementeren, doet men een **asymptotische analyse**. Dit is een schattingstechniek waarmee men de efficiëntie van algoritmen voor grootte inputwaarden gaat schatten. Om een **theoretische complexiteitsbenadering** uit te voeren wordt het algoritme opgebroken in **basisbewerkingen**. Dit zijn bewerkingen waarvan de uitvoeringstijd onafhankelijk is van het aantal operandi.

Typisch schrijft men de uitvoeringstijd van een algoritme op als een functie. Zo noteert men de uitvoeringstijd T voor een invoergrootte n als $T(n)$. Men kan deze functies uiteraard plotten maar wat belangrijker is, is de **orde van toename**. Het is deze die toelaat om verschillende algoritmes te vergelijken, zonder ze te implementeren. Veel voorkomende grootte ordes zijn bijvoorbeeld:

- **lineair**: cn
- **logaritmisch**: $\log_2(n)$
- **kwadratisch**: n^2
- **kubisch**: n^3
- **exponentieel**: c^n

Asymptotische analyse, waarbij men naar de snelst stijgende term van de tijdsfunctie kijkt, is enkel nuttig voor grote inputwaarden. Als men van tevoren weet dat de inputwaarden altijd zeer klein zullen zijn kiest men best gewoon voor het simpelste algoritme. Men noteert de orde van toename volgens de Θ-notatie, een kwadratisch stijgende functie wordt genoteerd als $\Theta(n^2)$.

Voor eenzelfde algoritme is de uitvoeringstijd niet steeds gelijk, men gaat uit van 3 gevallen:

1. **best mogelijke uitvoeringstijd** $T_b(n)$: bij sequentieel zoeken bv als resultaat eerste object is
2. **gemiddelde uitvoeringstijd** $T_g(n)$: bij sequentieel zoeken bv als getal middelste object is
3. **slechtst mogelijke uitvoeringstijd** $T_s(n)$: bij sequentieel zoeken bv als getal laatste object is

Meestal wordt de slechtst mogelijke uitvoeringstijd bestudeerd omdat de gemiddelde bestuderen moeilijk is en het slechtste geval toch frequent voorkomt. De gemiddelde tijd is meestal toch niet veel beter dan de slechtst mogelijke tijd.

In wat volgt bespreken we de verschilende asymptotische notaties en hun betekenis. Deze notaties bevatten steeds een functie en geen effectieve data omdat ze enkel op functies toepasbaar is. Men kan dus niet zeggen de ..-notatie van een algoritme, maar wel de ..-notatie van de gemiddelde uitvoeringstijd van dat algoritme.

- **bovengrenzen en O-notatie**: $f(n) = O(g(n))$ indien er constanten $c \in \mathbb{R}_0^+$ en $n_0 \in \mathbb{N}$ bestaan, zodanig dat $0 \leq f(n) \leq cg(n)$ voor alle $n \geq n_0$. Men zegt dan dat $f(n)$ **asymptotisch naar boven toe begrensd** wordt door $g(n)$.
- **ondergrenzen en Ω-notatie**: $f(n) = \Omega(g(n))$ indien er constanten $c \in \mathbb{R}_0^+$ en $n_0 \in \mathbb{N}$ bestaan, zodanig dat $f(n) \geq cg(n) \geq 0$ voor alle $n \geq n_0$. Men zegt dan dat $f(n)$ **asymptotisch naar onder toe begrensd** wordt door $g(n)$.
- **zelfde gedrag en Θ-notatie**: $f(n) = \Theta(g(n))$ als en slechts als $f(n) = O(g(n))$ en $f(n) = \Omega(g(n))$, met andere woorden indien er constanten $c_1, c_2 \in \mathbb{R}_0^+$ en $n_0 \in \mathbb{N}$ bestaan, zodanig dat $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$ voor alle $n \geq n_0$. Men zegt dan dat $f(n)$ en $f(g)$ op een positieve constante na **hetzelfde gedrag op oneindig** vertonen.
- **strikte bovengrenzen en o-notatie**: $f(n) = o(g(n))$ als en slechts als $f(n) = O(g(n))$ en $f(n) \neq \Theta(g(n))$. Waar de O-notatie nog kan betekenen dat de orde van toename tussen beide functies gelijk is wordt deze mogelijkheid bij de o-notatie weggelaten.
- **strikte ondergrenzen en ω-notatie**: $f(n) = \omega(g(n))$ als en slechts als $f(n) = \Omega(g(n))$ en $f(n) \neq \Theta(g(n))$. Waar de Ω-notatie nog kan betekenen dat de orde van toename tussen beide functies gelijk is wordt deze mogelijkheid bij de ω-notatie weggelaten.

Het **vergelijken van het asymptotisch gedrag** van twee functies kan met de limietregel worden gedaan. Daarvoor wordt

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} \tag{1}$$

berekent en volgende uitkomsten zijn dan mogelijk:

- limiet is 0: $f(n) = o(g(n))$
- limiet is $+\infty$: $f(n) = \omega(g(n))$
- limiet is een constante verschillend van 0: $f(n) = \Theta(g(n))$
- limiet bestaat niet: geen conclusie mogelijk

Om gemakkelijk met de O, Ω en Θ notatie te rekenen zijn volgende regels belangrijk. Ze worden voor de Θ-notatie geformuleerd maar zijn ook voor de andere twee notaties geldig:

1. Als $f(n) = \Theta(g(n))$ en $g(n) = \Theta(h(n))$, dan is $f(n) = \Theta(h(n))$
2. Als $f(n) = \Theta(kg(n))$ voor zekere constante $k > 0$, dan is ook $f(n) = \Theta(g(n))$
3. Als $f_1(n) = \Theta(g_1(n))$ en $f_2(n) = \Theta(g_2(n))$, dan geldt dat $f_1(n) + f_2(n) = \Theta(\max(g_1(n), g_2(n)))$

4. Als $f_1(n) = \Theta(g_1(n))$ en $f_2(n) = \Theta(g_2(n))$, dan geldt dat $f_1(n)f_2(n) = \Theta(g_1(n)g_2(n))$

Het is belangrijk te onthouden dat $\log_2 n$ **trager stijgt** dan elke polynomiale functie. Hier werd specifiek gezegt dat grondtal 2 gebruikt wordt in het logaritme. Soms moet van grondtal worden overgegaan. Dit kan met de basisformule:

$$\log_a x = \frac{\log_c x}{\log_c a} \tag{2}$$

Een exponentiële functie stijgt sneller dan elke polynomiale functie, een voorbeeld van een exponentiëel stijgende functie zijn de Fibonacci-getallen. De faculteitsfunctie is een snelstijgende functie met volgende eigenschappen:

(a) $n! = o(n^n)$

(b) $n! = \omega(2^n)$

(c) $\log_2(n!) = \Theta(n \log n)$

We bewijzen nu eigenschap (c) waarvoor we eerst eenbovengrens bewijzen:

$$\begin{aligned} \log_2(n!) &= \log_2 n + \log_2(n-1) + \cdots + \log_2 1 \\ &\leq \log_2 n + \log_2 n + \cdots + \log_2 n \\ &= n \log_2 n \end{aligned}$$

Hieruit volgt dat $\log_2(n!) = O(n \log n)$. Vervolgens bewijzen we een ondergrens:

$$\begin{aligned} \log_2(n!) &= \log_2 n + \log_2(n-1) + \cdots + \log_2 1 \\ &\geq \log_2 n + \log_2(n-1) + \cdots + \log_2(\lceil n/2 \rceil) \\ &\geq \log_2(\lceil n/2 \rceil) + \log_2(\lceil n/2 \rceil) + \cdots + \log_2(\lceil n/2 \rceil) \\ &= \lceil (n+1)/2 \rceil \times \log_2(\lceil n/2 \rceil) \\ &\geq (n/2) \log_2(n/2) \\ &= (n/2) \log_2 n - n/2 \\ &\geq (n \log_2 n)/4 \qquad \text{voor } n \geq 4 \end{aligned}$$

Hieruit volgt dat $\log_2(n!) = \Omega(n \log n)$, waarmee het gestelde is bewezen.

Op basis van voorgaande rekenregels kan men de tijds- en geheugencomplexiteit van algoritmen gaan bepalen. Als voorbeeld bepalen we de uitvoeringstijd van volgend programmafragment:

```
int som = 0;
for (int i = 1; i<=n; i++)
    for(int j = 1; j<=i; j++)
        som++
```

De bewerking `som++` vereist constante tijd c_3. De binnenste lus neemt i stappen dus is de kost $c_3 i$. De buitenste lus heeft n stappen maar de kost per stap is verschillend daar i van 1 tot n loopt. De totale kost van de dubbele lus is dus gegeven door c_3 maal de som van de getallen van 1 tot en met n. Aangezien geldt dat:

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2} \tag{3}$$

is de kost van het programmafragment dus $T(n) = \Theta(n^2)$. Op de zelfde manier worden nu andere programmafragmenten geanalyseerd. De analysetechnieken voor het inschatten van het vereiste geheugen $G(n)$ zijn analoog aan deze voor het inschatten van de vereiste uitvoeringstijd $T(n)$.

Een array is een aaneengesloten blok geheugen wat betekent dat de positie van het i-de array element kan berekend worden indien de index i, het beginadres x, en het aantal bytes b per component gekend is. Het adres van element $a[i]$ wordt dan door $x + b \times i$ gegeven, dit kan in constante tijd $\Theta(1)$.

Bij de vraag of men best een sneller computer koopt of beter het algoritme optimaliseert moet volgende bedenking worden gemaakt. Een computer kan enkel een constante lineaire factor sneller werken. Daar constante factoren verwaarloosd mogen worden bij complexiteitsanalyse zal de snellere computer geen invloed hebben op de verbetering van de probleemgrootte. Het enige verschil zit hem in de absolute inputgrootte die net wat groter zal kunnen zijn.

Er bestaat ook zo iets als **geamortiseerde complexiteitsanalyse**. In plaats van de individuele kost van de bewerkingen op te tellen, beschouwt deze de kost van *een hele sequentie van m bewerkingen* en kent aan iedere individuele bewerking een gedeelte van de totale kost toe. Deze techniek is bruikbaar indien de kost in het slechtste geval voor n bewerkingen kleiner is dan n keer de slechtste geval kost voor één enkele bewerking. Het probleem is dan niet meer de slechtst mogelijke tijd voor één bewerking, maar het *herhaaldelijk* voorkomen ervan. Indien men kan aantonen dat het slechste geval niet herhaaldelijk kan voorkomen kan men een betere tijdsbegrenzing geven die dan de **geamortiseerde tijdsbegrenzing** wordt genoemd. Die zal dan tussen de slechste geval analyse en de gemiddelde geval analyse liggen. De **geamortiseerde kost** garandeert dus de *gemiddelde performantie van elke bewerking in het slechtste geval*.

Op basis van deze notaties en complexiteit kunnen we nu volgende categoriëen van algoritmen en problemen beschouwen:

- **efficiënt algoritme**: de uitvoeringstijd is een polynomiale of trager stijgende functie.
- **handelbaar probleem**: er bestaat een efficiënt algoritme voor.
- **onhandelbaar probleem**: er kan worden bewezen dat er geen efficiënt algoritme voor bestaat.

De algoritmen worden in klassen en soorten onderverdeeld. Een **beslissingsprobleem** is een probleem dat enkel een antwoordt '*ja*' of '*nee*' vereist. Een beslissingsprobleem behoort tot de **klasse P** als het in polynomiale tijd kan worden opgelost, het behoort tot de **klasse NP** als het antwoord in polynomiale tijd kan worden geverifiëerd. Een beslissingsprobleem is **polynomiaal herleidbaar** als men elke instantiatie ervan in polynomiale tijd kan vervormen tot een ander beslissingsprobleem met dezelfde uitkomst. Een beslissingsprobleem is **NP-moeilijk** als elk probleem in de klasse NP polynomiaal herleidbaar is tot dat probleem. Een NP-moeilijk beslissingsprobleem is **NP-compleet (NPC)** als als dit tot de klasse NP behoort.

3 Algoritmen en abstracte datatypes in de Java API

En sorteeralgoritme wordt **stabiel** genoemd als het de volgorde van twee gelijke elementen in de rij bewaart. In java bestaan allereerst sorteer- en zoekmethodes voor arrays van alle primitieve types. De algoritmen hebben een $\Theta(n \log n)$ tijdscomplexiteit en zijn gegarandeerd stabiel. Het gebruik van de `binarySearch` methode vereist dat de rij in natuurlijk stijgende volgorde gesorteerd is.

Ook bepaalde objecten kunnen worden gesorteerd, hiervoor moeten ze wel `Comparable` zijn. De `Comparable` interface definiëerd een natuurlijke ordening. Er moet hiervoor één methode worden geïmplementeerd, de `public int compareTo(T x)` methode. Deze geeft volgende output:

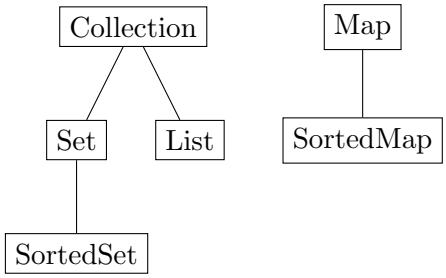
< 0	huidig object kleiner dan opgegeven object
0	huidig object gelijk aan opgegeven object
> 0	huidig object groter dan opgegeven object

Men noemt de ntuurlijke ordening **consistent met** `equals` indien `x1.compareTo(x2) == 0` dezelfde booleaanse waarde als `x2.compareTo(x1) == 0` teruggeeft voor alle `x1`, `x2` objecten van de klasse. Er wordt steeds aangeraden om deze methode consistent te implementeren.

Indien een **niet-natuurlijke** ordening gewenst is wordt een `Comparator` gebruikt om de objecten te vergelijken. Deze interface bevat één methode `compare(T x1, Tx2)` die zich voor de rest exact het zelfde gedraagt als de

`compareTo` methode. Ook hier wordt consistentie aangeraden.

Een **collection** of **container** is een object dat meerdere objecten groepeert. Ze bevatten basisbewerkingen zoals opzoeken, toevoegen en verwijderen van een element, bulkbewerkingen op een reeks elementen en iteratoren. Het **collection-framework** bestaat uit **interfaces die de ADT's** voorstellen, de **implementaties** en **polymorfe (herbruikbare) algoritmen**. In java API ziet het collection-framework er als volgt uit:



Het is de interface `Collection` die alle basisbewerkingen implementeert zo daar zijn `size`, `isEmpty`, `contains`, `add`, `remove`. De `add` en `remove` methodes geven true terug indien de collectie door hun oproep werd gewijzigd. Er worden in alle algemeenheid ook bulkbewerkingen gebruikt zoals:

- `c1.containsAll(c2)` : geeft true terug als `c1` alle elementen van `c2` bevat
- `c1.addAll(c2)` : voegt alle elementen van `c2` to aan `c1` en geeft true terug als de collectie hierdoor werd gewijzigd.
- `c1.removeAll(c2)` : verwijdert uit `c1` alle elementen die ook in `c2` aanwezig zijn en geeft true terug indien de collectie hierdoor werd gewijzigd.
- `c1.retainAll(c2)` : verwijdert uit `c1` alle elementen die niet ook in `c2` zitten en geeft true terug indien de collectie hierdoor werd gewijzigd.
- `c.clear()` : verwijdert alle elementen uit `c`.

Om over de elementen van een collectie te itereren wordt een **iteratorpatroon** gebruikt. In een `for`-lus is de integer het **iteratorobject**. Voor collecties is echter de meer algemene interface `Iterator` nodig. Deze bevat volgende methodes:

- `i.hasNext()` : geeft true terug als er nog verdere elementen zijn.
- `i.next()` : geeft het volgende element terug en verplaatst het iteratorobject naar de volgende positie
- `i.remove()` : verwijdert het element dat door `i.next()` laatst werd teruggegeven en kan dus maar éénmaal worden teruggegeven.

Als laatste bevat een `Collection` en eigenlijk elk `Object` nog twee methodes die de gelijkheid `equals` en hashing `hashCode` uitdrukken. Standaard geeft de equals methode true terug als de pointers gelijk zijn. Als twee objecten gelijk zijn volgens de `equals` methode moet de `hashCode` functie dezelfde hash teruggeven. Het is evenwel niet vereist dat de `hashCode` functie voor twee verschillende objecten een verschillende hash teruggeeft.

Wat volgt is een korte opsomming van de verschillende implementaties en hun eigenschappen:

- **Set**: een set laat geen dubbele elementen toe en bevat geen extra methodes dan die van `Collection`. Men moet goed oppassen met elementen waarbij de equals methode veranderlijk is. Er bestaan twee implementaties:
  1. `HashSet` : slaat de elementen op in een hashtabel en de bewerkingen vragen gemiddeld $\Theta(1)$ tijd.
  2. `TreeSet`: slaat de elementen in een gebalanceere binaire zoekboom op en de bewerkingen vragen in het slechtste geval $\Theta(n \log n)$ tijd.

- **SortedSet** : dit is een `Set` die zijn elementen in stijgende volgorde bijhoudt, dit wordt door de implementatie `TreeSet` verzorgt. Als extra methodes worden bewerkingen gedefiniëerd voor:
  - **eindpunten**: het opvragen van het eerste of laatste element:
    - `s.first()` : geeft het eerste element terug
    - `s.last()` : geeft het laatste element terug
  - **rangeviews**: het uitvoeren van bewerkingen op een gedeelte van de set. Dit zijn dus **geen kopiën** van de originele set maar wel **views**. Als men dus elementen verwijdert uit een subset zullen deze in de volledige set ook verwijdert worden.
    - `s.subSet(x1,x2)` : geeft een set terug met elementen in `[x1,x2[`
    - `s.headSet(x2)` : geeft een set terug met elementen in `[x0,x2[`
    - `s.tailSet(x1)` : geeft een set terug met elementen in `[x1,xn[`
  - **comparator**: toegang tot de comparator indien aanwezig
- **List**: dit is een `Collection` waarin de elementen een positie hebben en daarom wordt het ook wel eens **sequentie** genoemd en duplicaten zijn toegestaan. Een `List` bevat een `ListIterator` welke een uitbreiding is op een gewone `Iterator` . Er zijn twee basisimplementaties:
  - `ArrayList` : slaat de elementen op in een array, het opvragen gebeurt in constante tijd, maar het toevoegen is lineair daar alle verdere elementen moeten worden opgeschoven.
  - `LinkedList` : slaat de elementen op in een geschakelde lijst waardoor het tussenvoegen in constante tijd gebeurt, positionele toegang vraagt echter lineaire tijd.
- **Map** : een map of **dictionary** is een object dat **keys** op **values** afbeeld waarbij geen duplicaten van keys zijn toegestaan, ze bevatten zogenaamde bf key-value pairs welke de **entries** genoemd worden. Een map bevat zogenaamde **collectionviews** welke toelaten de map als een collectie te zien, dit wordt met de methodes `keySet`, `values` en `entrySet` gerealiseerd. Er zijn twee implementaties:
  - `HashMap` : slaat de entries op in een hashtabel en heeft doorgaans beste performantie.
  - `TreeMap` : slaat de entries op in een gebalanceerde binaire zoekboom waardoor een volgorde kan blijven behouden.
- **SortedMap**: deze map houdt zijn entries bij in stijgende volgorde en bevat de zelfde uitbreidingen als een `SortedSet`. De `TreeMap` is de standaardimplementatie van java voor een `SortedMap`.

4 Grafen en bomen

Een graaf $G = (V(G), E(G))$ bestaat uit een eindige verzameling **toppen** $V(G)$ en een verzameling **bogen** $E(G)$ respectievelijk de **toppenverzameling** en de **bogenverzameling**. Een boog definiëerd een verbinding tussen twee toppen, de **eindpunten** van de boog genoemd. Het aantal toppen in de graaf wordt de **orde** van de graaf genoemd en het aantal bogen zijn **grootte**. Een graaf van orde n en grootte m wordt ook wel een (n, m)**-graaf** genoemd. **Adjacente toppen** zijn toppen die verbonden zijn door een boog, **adjacente bogen** zijn bogen met een gemeenschappelijk eindpunt. Als een top v een eindpunt is van boog e, dan noemt men v **incident** met e en omgekeerd. Een graaf wordt **dicht** genoemd als bijna alle bogen aanwezig zijn, een graaf wordt **ijl** genoemd indien dit niet zo is.

Er zijn verschillende soorten grafen:

- **multigraaf**: in deze graaf zijn dezelfde toppen door meer dan één boog verbonden. Bogen met dezelfde eindpunten worden **parallelle bogen** genoemd en een collectie van parallelle bogen wordt een **multiboog** genoemd.

- **pseudograaf**: in deze graaf kunnen toppen met zichzelf verbonden zijn, de boog die dat doet wordt een **zelflus** genoemd, een boog die dat niet doet wordt een **eigenlijke boog** genoemd. Als in een graaf zowel zelflussen als multibogen zijn toegestaan is dit een pseudograaf.
- **simpele graaf**: een graaf die geen zelflussen en geen multibogen bevat, meestal wordt met de term graaf naar een simpele graaf verwezen.
- **gerichte graaf**: een boog doelt op de verbinding tussen twee toppen in beide richtingen. Een **gerichte boog** of **pijl** heeft een **kop** en een **staart** en verbind twee toppen in één richting. Een graaf waarin alle bogen gericht zijn is dan een gerichte graaf. De **onderliggende graaf** verkrijgt men dan door de richtingen de verwijderen.
- **gewogen graaf**: in deze graaf wordt aan elke boog een **gewicht** gehecht.
- **complete graaf**: elk paar toppen is verbonden door een boog. Een complete graaf met n toppen wordt als K_n genoteerd.
- **bipartiete graaf**: in deze graaf kan de toppenverzameling V worden opgesplitst in twee deelverzamelingen V_1 en V_2 zodanig dat elke boog van G een eindpunt inV_1 en in V_2 heeft. Het paar (V_1, V_2) wordt dan de **bipartitie** van G genoemd. De toppen in de **bipartitieverzamelingen** zelf mogen echter niet met elkaar verbonden zijn. Een **complete bipartiete graaf** is een bipartiete graaf waarbij alle mogelijk bogen aanwezig zijn.
- **cykelgraaf**: in deze soort graaf bestaat enkel een cykel, deze kunnen als een regelmatige veelhoek worden getekent. Een cykelgraaf met n knopen wordt als C_n genoteerd.
- **planaire graaf**: een planaire graaf kan getekend worden zonder snijdende bogen.

Men definiëert de **graad van een top (deg(v))** als het aantal **buren** van de top. De buren van een top zijn alle toppen die door een boog verbonden zijn met die top. Een **geïsoleerde top** is een top met graad 0, een top met graad 1 wordt een **eindtop** genoemd. Zij G een graaf met $n > 1$ toppen, dan heeft G minstens één paar toppen waarvan de graden gelijk zijn. De maximale graad van een top (Δn) is de graad van de top met de meeste buren.

Een belangrijke stelling is de **stelling van Euler**: Zij G een graaf met orde n en grootte m, en zij $V(G) = \{v_1, v_2, \ldots, v_n\}$. Dan geldt dat

$$\sum_{i=1}^{n} deg(v_i) = 2m \tag{4}$$

Twee grafen worden **isomorf** genoemd als er een bijectie tussen de knopen bestaat en de knopen op dezelfde manier verbonden zijn. In verband met het doorlopen van een graaf kunnen nu verschillende begrippen worden gedefiniëerd:

- **wandeling**: een rij met toppen en bogen waarover wordt bewogen.
- **gerichte wandeling**: een rij met toppen en pijlen waarover wordt bewogen. Een gerichte wandeling van top x naar top y wordt ook wel een **gerichte x-y-wandeling** genoemd en indien x en y dezelfde zijn een **gesloten wandeling** anders een **open wandeling**.
- **spoor**: een wandeling waarin geen herhalende bogen voorkomen, er bestaat ook een **gericht spoor**.
- **pad**: een wandeling waarin geen herhalende toppen voorkomen, behalve eventueel begin en eindpunt. Er bestaat ook een **gericht pad**.
- **circuit**: een niet-triviaal gesloten spoor.
- **cykel**: een niet-triviaal gesloten pad.
- **Euleriaans spoor of circuit** is een spoor of circuit dat alle bogen bevat.

- **Hamiltoniaans pad of cykel** is een pad of cykel dat alle toppen bevat.

Een **traverseerbare** graaf bevat een Euleriaans spoor, een **Euleriaanse graaf** bevat een Euleriaans circuit. Een top v is **bereikbaar** voor een top u als er een wandeling van u naar v is. Een graaf is **samenhangend** als er voor elk paar toppen een wandeling tussen dat paar toppen bestaat.

Volgende stelling kan worden bewezen: zij G een samenhangende multigraaf dan geldt:

- G is euleriaans als en slechts als de graad van elke top even is.
- G is traverseerbaar als en slechts als G twee toppen met oneven graad heeft.

Een **boom** is een samenhangende graaf die geen cykels bevat, een verzameling bomen is een **woud**.

Grafen worden veel gebruikt om het kleurenprobleem voor te stellen. Hiervoor moeten echter een paar nieuwe begrippen worden ingevoerd:

- **kliek**: dit is een deelverzameling S van de toppen van een graaf waarvoor geldt dat elk paar toppen van S adjacent is in de graaf. Een **maximale kliek** is de grootste kliek in de graaf die geen deel is van een andere kliek in de graaf, het aantal toppen in deze kliek wordt het **kliekgetal** van G genoemd. Dit laatste wordt met $\omega(G)$ genoteerd.
- **kleuring van een graaf**: dit begrip doelt op het toekennen van kleuren aan toppen zodanig dat adjacente toppen een verschillende kleur hebben. Een kleuring die k kleuren gebruikt wordt een **k-kleuring** genoemd. De kleinste waarde k waarvoor er een k-kleuring bestaat voor de graaf G noemt men het **chromatisch getal** van G wat als $\chi(G)$ wordt genoteerd.

Voor een complete graaf geldt dat $\chi(K_n) = n$, of dus dat het aantal toppen gelijk is aan het chromatisch getal. Een andere eigenschap stelt dat een graaf G bipartiet is als en slechts als $\chi(G) = 2$. Voor cykelgrafen moet men het kleurenprobleem opsplitsen:

- voor n even en $n \geq 4$ geldt : $\chi(C_n) = 2$
- voor n oneven en $n \geq 3$ geldt : $\chi(C_n) = 3$

Voor een willekeurige graaf is het bepalen van het chromatisch getal een onhandelbaar probleem. Er bestaan echter wel stellingen en opmerkingen die het inschatten van het chromatisch getal kunnen vergemakkelijken:

- voor $K_n : \chi(K_n) = \Delta(K_n) + 1$
- voor C_n en n oneven : $\chi(C_n) = \Delta(C_n) + 1$
- voor elke graaf: $\chi(G) \leq \Delta(G) + 1$
- als G geen complete graaf/oneven cykel is dan $\chi(G) \leq \Delta(G)$ (**stelling van Brooks**)
- $\omega(G) \leq \chi(G)$
- elke planaire graaf kan gekleurd worden met ten hoogste 4 verschillende kleuren (**vierkleurenstelling**)

In computerprogramma's worden twee soorten voorstellingen gebruikt voor grafen:

- **adjacentiematrixvoorstelling**: Zij G een graaf met toppenverzameling $V(G) = \{v_1, v_2, \ldots, v_n\}$, dan wordt de adjacentiematrix $A = [a_{ij}]$ van G gegeven door de $n \times n$ matrix gedefinieerd door:

$$a_{ij} = \begin{cases} 1 & \text{als } v_i v_j \in E(G) \\ 0 & \text{anders} \end{cases}$$

- **adjacentielijstvoorstelling**: met elke top van G wordt een lijst van toppen die ermee adjacent is bijgehouden.

Volgende stelling karakteriseert een boom. Zij G een graaf met n toppen, dan zijn volgende uitspraken equivalent:

- G is een boom
- G heeft geen cykels en bevat $n-1$ bogen
- G is samenhangend en bevat $n-1$ bogen
- G is samenhangend en elke boog is een brug
- Elke twee toppen van G zijn precies door één pad verbonden
- G bevat geen cykels en voor elke nieuwe boog e bevat $G+e$ precies één cykel

Een **gerichte boom** is een gerichte graaf waarvan de onderliggende graaf een boom is. Een gewortelde boom T is een gerichte boom die een speciale top r, **de wortel** bevat, zodanig dat er voor elke top v van T een r-v-pad bestaat.

Een k-aire boom is een gewortelde boom waarin elke top ten hoogste k kinderen heeft. Een top zonder kinderen wordt een **blad** genoemd, toppen met kinderen worden **interne toppen** genoemd. Een **binaire boom** is een 2-aire boom waarin één kind als het linkerkind en het ander als het rechterkind wordt beschouwd. Een gewortelde boom T is een **compete k-aire boom** als elke top van T ofwel k kinderen of wel geen kinderen heeft. In een **complete binaire boom** heeft elke top twee kinderen. We geven hieromtrent enkele stellingen:

- Zij in een niet lege binaire boom n_0 het aantal bladeren en n_2 het aantal toppen van graad 2, dan is $n_0 = n_2 + 1$.
- Een complete k-aire boom met i interne toppen heeft in totaal $ki+1$ toppen.
- Een complete binaire boom met i interne toppen heeft $i+1$ bladeren.
- Een binaire boom bevat ten hoogste 2^i toppen van niveau i ($i \geq 0$).
- Zij T een binaire boom van hoogte h met n toppen, dan geldt dat $n \leq 2^{h+1} - 1$.
- Zij T een binaire boom van hoogte h met n toppen, dan geldt dat $h \geq \lceil \log_2 \left(\frac{n+1}{2}\right) \rceil$. De gelijkheid geldt als T een gebalanceerde complete binaire boom is.
- Zij T een ternaire boom van hoogte h en b bladeren, dan is $h \geq \log_3 b$
- Zij T een binaire boom van hoogte h en b bladeren, dan is $h \geq \log_2 b$

We geven van deze laatste stelling het bewijs. We gebruiken inductie op h om de equivalente ongelijkheid

$$b \leq 2^h$$

te bewijzen. Voor $h = 0$ bestaat de boom uit één top, die ook een blad is, m.a.w. $b = 1$ en dus $b \leq 2^h$. Veronderstel nu dat de ongelijkheid voldaan is voor elke binaire boom met hoogte kleiner dan h. We beschouwen nu een boom T van hoogte h met b bladeren. We beschouwen eerst het geval dat de wortel van T slechts 1 kind heeft. Dit is een binaire boom van hoogte $h-1$ die ook b bladeren heeft. Gebruik makend van de inductiehypothese bekomen we $b \leq 2^{h-1}$. Aangezien $2^{h-1} < 2^h$, is in dit geval dus ook $b \leq 2^h$. Vervolgens beschouwen we het geval dat de wortel van T twee kinderen heeft. Zij h_l de hoogte van de linkerdeelboom en h_r de hoogte van de rechterdeelboom; er geldt dat $h_l \leq h-1$ en $h_r \leq h-1$. Zij b_l het aantal bladeren in de linkerdeelboom en b_r het aantal bladeren in de rechterdeelboom. Uit de inductiehypothese volgt dat:

$$b_l \leq 2^{h_l} \text{ en } b_r \leq 2^{h_r}$$

Hieruit bekomen we:

$$b = b_l + b_r \leq 2^{h_l} + 2^{h_r} \leq 2^{h-1} + 2^{h-1} \leq 2^h$$

hetgeen het gestelde bewijst.

5 Recursie

Een algoritme is **recursief** als het **zichzelf oproept** om een gedeelte van het werk uit te voeren. Algemeen bestaat een recursief algoritme uit twee delen:

1. **basisgeval**: een geval simpel genoeg om zonder recursieve oproep te doen, dit eindigt de recursie meestal.
2. **recursief geval**: een oproep naar zichzelf met een probleem kleiner dan het oorspronkelijke.

Het berekenen van een faculteit, de torens van Hanoi en machtsverheffing zijn typische voorbeelden van recursieve algoritmen. Zij het niet dat deze problemen eventueel efficiënter kunnen worden opgelost.

De uitvoeringstijd van een recursief algoritme wordt als een **recurrente betrekking** opgeschreven wat de asymptotische analyse bemoeilijkt. Er zijn drie methodes om de complexiteit te bepalen:

1. **substitutiemethode**: met deze methode wordt een schatting gemaakt van de begrenzing en de correctheid van de schatting wordt met inductie bewezen. Deze kan dus enkel gebruikt worden als de oplossing op zicht kan worden gezien.
2. **iteratiemethode**: de recurrente betrekking wordt iteratief volledig uitgewerkt tot een sommatie waarin enkel n en de initiële waarden optreden. Een **recursieboom** kan hierbij een zeer handig hulpmiddel zijn.
3. **mastermethode**: zij $a \geq 1$ en $b > 1$ constanten, zij $f(n)$ een functie en zij $T(n)$ gedefinieerd over de niet-negatieve gehele getallen door de recurrente betrekking

   $$T(n) = aT(n/b) + f(n) \tag{5}$$

   waarbij n/b te interpreteren is als ofwel $\lfloor n/b \rfloor$ ofwel $\lceil n/b \rceil$. Dan kan $T(n)$ als volgt asymptotisch worden begrensd:

   (a) Als $f(n) = O(n^{\log_b a - \varepsilon})$ voor zekere constante $\varepsilon > 0$, dan is

   $$T(n) = \Theta(n^{\log_b a}) \tag{6}$$

   (b) Als $f(n) = \Theta(n^{\log_b a})$, dan is

   $$T(n) = \Theta(n^{\log_b a} \log n) \tag{7}$$

   (c) Als $f(n) = \Omega(n^{\log_b a + \varepsilon})$ voor een zekere constante $\varepsilon > 0$, en als $af(n/b) \leq cf(n)$ voor zekere constante $c < 1$ en voldoende grote n (regulariteitsvoorwaarde), dan is

   $$T(n) = \Theta(f(n)) \tag{8}$$

   De master methode beschrijft een verdeel-en-heers algoritme. Het probleem van grootte n wordt opgesplitst in a deelproblemen van grootte n/b. Elk deelprobleem wordt opgelost in $T(n/b)$. De kost van het opsplitsen en samenvoegen van de deelresultaten wordt door $f(n)$ beschreven.

In de praktijk worden voor de analyse van algoritmen bepaalde technische details verwaarloosd. Zo wordt verondersteld dat de argumenten altijd natuurlijke getallen zijn, er worden geen randvoorwaarden opgesteld, en men vermeldt enkel het algemene geval omdat andere gevallen enkel een constante factor verschil hebben waardoor ze geen invloed hebben op de orde van toename van het algoritme.

Ookal is recursie een krachtige ontwerptechniek, het is niet steeds aan te raden deze te gebruiken. Als vuistregel geldt dat men recursie moet vermijden als deze een gewone `for`-lus vervangt en er na de recursieve oproep dus niets meer gebeurt. Dit noemt men **staartrecursie**, voorbeelden hiervan zijn het recursief berekenen van een faculteit, het recursief vinden van de grootste gemene deler met Euclides en het recursief berekenen van de fibonacci getallen.

6 Algemene ontwerpstrategieën

Enkele veel gebruikte ontwerpstrategieën zijn de volgende:

- **brute kracht**: een algoritme probeert gewoon alle mogelijkheden uit, meestal is dit inefficiënt maar wel makkelijk te bedenken en toe te passen. Deze benaderingsmethode wordt ook wel eens **exhaustieve benadering** genoemd.
- **tijd/ruimte-tradeoffs**: bij een algoritme moet altijd de afweging uitvoeringstijd versus gebruikt geheugen worden gemaakt. Als men meer geheugen gebruikt, door eerder bekomen informatie op te slaan, zal dat meestal tot een sneller algoritme leiden.
- **verdeel-en-heers**: deze algoritmen bestaan uit een **verdeel-fase**, waarbij het oorspronkelijke probleem opgesplitst wordt in kleinere deelproblemen die recursief opgelost worden, en een **heers-fase**, waarbij de oplossing van voor het oorspronkelijk probleem geconstrueerd wordt uit de oplossingen van de deelproblemen.
- **verminder-en-heers**: waar bij een echt verdeel-en-heers algoritme twee recursieve oproepen vereist zijn is in dit geval maar 1 deelprobleem noodzakelijk. Meestal leidt tot echter tot staartrecursie zoals bij het recursief berekenen van een faculteit.
- **transformeer-en-heers**: bij deze aanpak wordt een probleem vereenvoudigd, veranderd of gereduceerd naar een probleem waarvooral een oplossing gekend is of waarvoor eenvoudig een oplossing kan worden gevonden. Zo kan men een probleem vaak vereenvoudigen door bv. vooraf te sorteren,...

Het zoeken in een rij is een veel besproken probleem en we geven hier een brute kracht en een verdeel-en-heers strategie:

- **sequentiëel zoeken**: de volledige rij wordt doorlopen totdat het gevonden element wordt gevonden. Er zijn dus twee mogelijke situaties waarbij de zoekbewerking stopt: ofwel wordt het element gevonden en teruggegeven, ofwel wordt het einde van de rij gevonden. Dit maakt dat er in de `while`-lus twee stopcondities staan. Dit kan gereduceerd worden naar één stopconditie indien er een **sentinel** (of "schildwacht") wordt gebruikt. Het komt er op neer dat het gezochte element wordt toegevoegd aan het einde van de rij. Daardoor kan de controle of men het element gevonden heeft worden vermeden en moet op het einde maar 1 extra controle gebeuren, namelijk of de vonst binnen de lengte van de oorpronkelijke rij zit.
- **binair zoeken**: in een gesorteerde rij kan binair gezocht worden. Men verdeelt hierbij de rij in twee totdat het element triviaal geselecteerd kan worden.

Om de verschillende soorten ontwerpstrategiën aan te tonen worden in wat volgt enkele belangrijke problemen besproken. We beginnen met het **probleem van de maximale deelrijsom**. Gegeven is een rij $(a_0, a_1, \ldots, a_{n-1})$ van n gehele getallen. Bepaal een deelrij $(a_i, a_{i+1}, \ldots, a_j)$, met $0 \leq i \leq j \leq n-1$ waarvoor de deelrijsom $\sum_{k=i}^{j} a_k$ maximaal is. We geven hiervoor twee mogelijke oplossingen:

- **brute kracht**: Alle mogelijke deelrijsommen worden bekeken en de grootste wordt teruggegeven. In de meest basische vorm levert dit een $\Theta(n^3)$ algoritme op, echter gelet op de gelijkheid $\sum_{k=i}^{j} a_k = a_j + \sum_{k=i}^{j-1} a_k$ kan een $\Theta(n^2)$ brute kracht algoritme worden gegeven (pg. 192).

- **verdeel-en-heers**: we merken op dat de maximale deelrijsom te associëren is met een deelrij die ofwel volledig tot de linkerhelft, volledig over de rechterhelft of in het midden gesitueerd is. De eerste twee gevallen kunnen met recursie worden opgelost, het laaste geval wordt opgelost door in de linkerhelft de deelrij met de maximale som op te sporen die het laaste element bevat en in de rechterhelft deze die het eerste element bevat. Deze twee maximale sommen worden dan opgeteld (pg. 194). Het bekomen algoritme heeft een $\Theta(n \log n)$ tijdscomplexiteit.

- **lineair algoritme**: Gebruik makend van een paar eigenschappen kan het brute kracht algoritme verder worden verbeterd tot een lineair algoritme. Er wordt hierbij van volgende eigenschappen gebruik gemaakt:

  - Wanneer een deelrij $(a_i, \ldots, a_j)$, met $0 \leq i \leq j < n-1$, een negatieve som heeft, dan kan de maximale deelrij niet van de vorm $(a_i, \ldots, a_j, \ldots, a_k)$, met $0 \leq i \leq j < k \leq n-1$ zijn.
  - Voor willekeurige $i \geq 0$, als $a_i, \ldots, a_j$ de eerste deelrij is waarvoor de som negatief wordt, dan is voor elke $i \leq p \leq j$ en elke $q \geq p$, de deelrij $a_p, \ldots, a_q$ ofwel geen maximale deelrij, ofwel een reeds geziene maximale deelrij.

    Bewijs. Voor $p = i$ volgt het gestelde onmiddelijk uit de vorige eigenschap. Voor $p > i$ is de beschouwde deelrij ofwel van de vorm $a_i, \ldots, a_p, \ldots, a_j, \ldots, a_q$ ofwel van de vorm $a_i, \ldots, a_p, \ldots, a_q, \ldots, a_j$. Aangezien j de eerste index is waarvoor de som negatief wordt, is de som van $a_i, \ldots, a_{p-1}$ positief en dus is de som van $a_p, \ldots, a_q$ kleiner dan of gelijk aan de som van $a_i, \ldots, a_q$. In het eerste geval, als $j < q$, dan volgt uit de vorige eigenschap dat de deelrij $a_i, \ldots, a_q$ geen maximale deelrij is. Anders is $a_i, \ldots, a_q$ een reeds geziene deelrij met grotere som.

  Het uiteindelijke algoritme kan nu als volgt worden geformuleerd (pg. 196):

  - Construeer de rij $s_i, i = 0, \ldots, n$, als volgt: $\begin{array}{rcl} s_0 & = & 0, \\ s_i & = & \max(s_{i-1} + a_i, 0), \text{ voor } i = 1, \ldots, n \end{array}$
  - De gezochte som $s = \max(s_i, i = 0, \ldots, n)$.

  De complexiteit is dus $\Theta(n)$ en de rij moet maar één keer worden doorlopen en moet niet in het geheugen worden opgeslaan. Ook is belangrijk op te merken dat dit algoritme op elk moment een oplossing van het probleem van de maximale deelrijsom voor het reeds ingelezen gedeelte van de inputrij heeft. Algoritmen met deze eigenschap noemt men **on-line** algoritmen.

Het **probleem van het dichtste puntenpaar** kan met brute kracht opgelost worden in $\Theta(n^2)$ en met de verdeel en heers techniek in $\Theta(n \log n)$ (pg. 192 - 202).

Als laatste probleem bekijken we dat van **string-matching** of dus het zoeken van patronen in tekst. Hierbij gaat volgende terminologie gepaard:

- **tekst** T: de tekst waarin het patroon wordt gezocht, met $|T|$ wordt de lengte van de tekst bedoelt, meestal noteren we $n = |T|$.

- **patroon** P: het patroon dat wordt gezocht, met $|P|$ wordt de lengte van het patroon bedoelt, meestal noteren we $m = |P|$.

- **alfabet** Σ: het alfabet waaruit de tekst bestaat. De verzameling van alle woorden op het alfabet Σ wordt als Σ^* genoteerd. Het binaire alfabet wordt dus als $\{0, 1\}^*$ genoteerd.

- **substrings**: substrings worden zoals in python genoteerd, bijvoorbeeld $P[i..j]$ met i inclusief en j exclusief.

Er zijn dus verschillende algoritmen met elk hun complexiteit, volgende zijn de meest bekende algoritmen:

- **brute-kracht-algoritme**: lost het probleem in $O(|P| \times |T|)$ op.
- **algoritme van Rabin-Karp**: dit algoritme maakt gebruik van hashing of **fingerprinting** en heeft een gemiddelde uitvoeringstijd van $O(|P| + |T|)$.
- **algoritme van Knuth-Morris-Pratt** en **algoritme van Boyer-Moore**: beide deze algoritmen lossen het probleem op met een slechtste uitvoeringstijd van $O(|P| + |T|)$ door gebruik te maken van een **verschuivingstabel** die door ene preprocessing van het patroon wordt verkregen. Het implementeren van Boyer-Moore is echter zeer ingewikkeld zodat er nog een extra variant van ontwikkeld is.
- **algoritme van Horspool**: dit algoritme heeft een slechste geval uitvoeringstijd van $O(|P| \times |T|)$ maar werkt n de praktijk zeer snel en is eenvoudig te implementeren.

We bespreken eerst een eenvoudig brute-kracht algoritme. Dit werkt door het patroon te aligneren met de eerste m letters van de tekst en dan van links naar rects het patroon letter per letter te vergelijken met de tekst (pg. 203). In het slechtste geval, P komt niet voor in T, levert dit een uitvoeringstijd van $\Theta(m(n-m+1))$ op. In het beste geval, P is aan het begin van T, levert dit een uitvoeringstijd van $\Theta(m)$ op. In de praktijk is het algoritme beter dan $O(m(n-m+1))$ omdat de binnenste lus een niet-overeenstemmend patroon snel herkent.

In zojuist besproken algoritme werd van links naar rechts vergeleken en bij een mismatch naar rechts opgeschoven. Men kan er ook voor kiezen om van rechts naar links te vergelijken maar bij een mismatch nog steeds naar rechts op te schuiven (pg. 204-207), dit is het idee van **Boyer-Moore**. Het zo bekomen algoritme heeft nog steeds een $O(m(n-m))$ uitvoeringstijd maar er kunnen wel gemakkelijk heuristieken voor gemaakt worden. Deze laatste gaan er meestal voor zorgen dat er meer dan één plaats kan worden opgeschoven. Zo bestaan o.a. de occurence-heuristiek, de match-heuristiek en de **heuristiek van Horspool**.

De heuristiek van Horspool stelt dat bij een mismatch wordt geprobeerd $T[i+m-1]$ te matchen met het meest rechtse optreden van dat karakter in het patroon links van $P[m-1]$. Deze regel garandeert dat het patroon altijd naar rechts verschoven wordt. Om de verschuiving correct te laten gebeuren moet voor elk karakter gekend zijn wat zijn meest rechtse voorkomen in het patroon links van $P[m-1]$ is. Meer formeel:

$$S[x] = \begin{cases} m-1-\max\{i < m-1 | P[i] = x\} & \text{als } x \text{ behoort tot } P[0..m-2] \\ m & \text{anders} \end{cases}$$

Om de verschuiving efficiënt te laten gebeuren wordt deze informatie in een **verschuivingstabel** bijgehouden die vooraf voor het patroon wordt berekend. Wanneer het alfabet Σ vast is loopt het algoritme van Horspool in $O(mn)$. De gebruikte geheugenruimte hangt niet af van P maar van Σ en is $O(|\Sigma|)$. Wat zojuist besproken is moet meer correct een eenvoudig **Boyer-Moore algoritme met bijkomende Horspool heuristiek** genoemd worden.

Het algoritme van **Rabin-Karp** vertrekt ook van het standaard brute-kracht algoritme maar verbetert het op een andere manier. In plaats van $O(m)$ tijd te spenderen aan het controleren of een lang patroon P op een bepaalde positie i voorkomt in de tekst wordt ernaar gestreefd om in $O(1)$ tijd te controleren of we op positie i wel een match kunnen verwachten. Hierdoor kunnen alle behalve 1/m-de van de posities worden geëlimineert en er worden dus maar $(n-m+1)/m$ posities overgehouden waarvoor effectief een $O(m)$ brute-kracht vergelijking moet plaatsvinden. Dit levert gezamelijke uitvoeringstijd $O(m(n-m+1))$ voor de vergelijkingen. Samen met de $O(m)$ tijd voor de voorafgaande testen geeft dit aanleiding tot een totale gemiddelde uitvoeringstijd van $O(n+m)$.

Voor de expliciete controle op positie i moet dus eerst een eenvoudige test worden uitgevoerd die i elimineert. Men doet dit met **fingerprinting** waarbij slechts een aspect van het patroon wordt beschouwd. Zo kan men bijvoorbeeld een pariteitscontrole uitvoeren (pg. 208-209), maar men verwacht dat deze maar de helft van de posities zal elimineren, versnellingsfactor $q = 2$. Het probleem is dat er teveel waarden zijn met een zelfde vingerafdruk. Om een hogere q te bekomen, in het geval van bitstrings, moet de vingerafdrukfunctie aan 3 voorwaarden voldoen:

1. De functie moet bitstrings van lengte m op q verschillende waarden (vingerafdrukken) afbeelden.
2. De functie moet de bitstrings van lengte m gelijkmatig over de q vingerafdrukken verdelen
3. De functie moet in sequentie te berekenen zijn, dit wil zeggen dat bij het wijzigen van 1 bit aan het begin of einde van de string de nieuwe vingerafdruk in $O(1)$ tijd te berekenen is.

Pariteitscontrole voldoet aan deze voorwaarden voor $q = 2$ waardoor een versnellingsfactor 2 wordt bekomen. Een functie die aan de eerste twee voorwaarden voldoet is een **hashfunctie**. Robin en Karp stelden voor bitstrings deze te aanzien als de voorstelling van een natuurlijk getal en de rest bij deling door q te nemen. Zij $s_0 s_1 \dots s_{m-1}$ de m bits, dan berekenen we de hashwaarde als:

$$\sum_{j=0}^{m-1} s_j 2^{m-1-j} \bmod q \tag{9}$$

Deze functie voldoet triviaal aan voorwaarde 1. Of hij voldoet aan voorwaarde b hangt af van de waarde van q maar in de praktijk is dit zo als een q groter dan m wordt gekozen als priemgetal. Ook voldoet de functie aan de derde voorwaarde want $h(s_1 \dots s_m) = s_m + 2(h(s_0 \dots s_{m-1}) - 2^{m-1} s_0) \bmod q$.

7 Gebruik van stapels, wachtlijnen en prioriteitswachtlijnen

Stapels hebben een **LIFO - Last In First Out** structuur en bevatten push, pop en peek bewerkingen. Ze worden zeer veel gebruikt bij het ontwerp van compilers zoals bij het controleren van gebalanceerde symbolen, evalueren van rekenkundige uitdrukkingen (conversie van infix- naar postfix notatie en evaluatie van de postfix) en het oproepen van methodes (call stack).

Compilers controleren dus o.a. of elk symbool correct wordt afgesloten, dus (en), [en], { en },... Ook moet de volgorde geldig zijn zodat (]) bijvoorbeeld niet voorkomt. Dit is typisch een probleem dat met een stack kan worden opgelost. Vertrekkend van een lege stapel worden alle tekens verwerkt, als het teken een open haakje is wordt het op de stapel geplaatst, is het een gesloten haakje dan wordt het vergeleken met dat op de stapel. Als dit geen geldig koppel blijkt te zijn dan is een niet gebalanceerd symbool gevonden, is de stapel leeg dan waren er teveel sluit-symbolen, als men het einde van de tekst bereikt voor de stapel leeg is dan waren er te veel open symbolen. Het is duidelijk dat dat algoritme in $\Theta(n)$ tijd werkt.

Er zijn twee soorten manieren om wiskundige uitdrukkingen te noteren:

- **infix-notatie**: de operator wordt hier tussen de argumenten geschreven en met moet met de prioriteit van de operatoren rekening houden. Ook kan men gebruik maken van haakjes om de prioriteiten te veranderen. Bijvoorbeeld $a + b \times c + (d \times e + f) \times g$.
- **postfix-notatie**: de operator wordt hier na de argumenten geschreven en er zijn geen prioriteitsregels en haakjes die in acht moeten worden genomen. Deze notatie wordt ook wle RPN (of Reverse Polish Notation) genoemd. Bijvoorbeeld $abc \times +de \times f + g \times +$.

De volgorde van de operanden blijft in beide notaties het zelfde, het is alleen de positie van de operatoren die verschilt. Een compiler zal alle uitdrukking eerst naar postfix notatie omzetten omdat deze veel gemakkelijker te evalueren is, dit kan namelijk met een stack gebeuren. Het berekenen, dat in $\Theta(n)$ tijd kan, gaat als volgt:

- De operanden worden op te stapel gezet totdat een operator wordt gelezen.
- Als men een operator tegenkomt haalt men de laatste twee operanden van de stapel en voert de operator uit.
- Men plaatst het bekomen resultaat terug op de stapel.

- Men voert deze stappen uit tot het einde van de uitdrukking is bereikt. Het enige element op de stapel is nu de oplossing.

Het omzetten van de infix naarde postfix notatie kan ook gemakkelijk met een stapel gebeuren. De volgorde van de argumenten is gelijk, dus een argument kan men onmiddelijk uitschrijven. Een operator kan men pas uitschrijven als beide argumenten zijn uitgeschreven, deze worden dus op een stapel bijgehouden, ook de (-symbolen worden op de stapel bijgehouden. Volgende regels worden dus in acht genomen:

- Ontmoet men een)-symbool, dan worden achtereenvolgens alle symbolen tot aan het eerste (-symbool van de stapel gehaald en in de output geplaatst. Het (-symbool zelf wordt gewoon verwijdert.
- Ontmoet men een (-symbool dan wordt dit op de stapel geplaatst.
- Ontmoet men een operator, dan worden eerst achtereenvolgens operatoren van de stapel gehaald en aan de output toegevoegd, tot wanneer de top van de stapel een operator van lagere prioriteit bevat. Aan het (-symbool wordt de laagste prioriteit gegeven zodat dit alleen maar van de stapel kan worden afgehaald nadat een)-symbool werd gezien. Daarna wordt de gelezen operator op de stapel geplaatst.
- Als zo de hele uitdrukking is verwerkt, worden de symbolen die nog of de stack staan er af gehaald en aan de output toegevoegd.

Dit kan in $\Theta(n)$ tijd (pg. 221).

Heel veel talen houden ook het oproepen van methodes bij op een zogenaamde **call-stack**. Men kan dit vergelijken met het balanceren van haakjes. Het oproepen van een methode komt overeen met het openen van een haakje en het terugkeeradres en argumenten worden op de stapel geplaatst. Het verlaten van een methode komt overeen met het sluiten van een haakje en het terugkeeradres en variabelen worden van de stapel gehaald.

We bespreken nu twee algoritmes om een graaf systematisch te doorlopen, **diepte-eerst-doorlopen (DFS - Depth-First-Search)** en **breedte-eerst-doorlopen (BFS - Breadth-first-search)**. Beide algoritmen steunen op het opbouwen van een opspannende boom of opspannend woud. De algoritmes:

1. **diepte-eerst-doorlopen**: dit algoritme gaat uit van volgende gedachten:
   - gegeven: een graaf $G = (V, E)$, starttop $v \in V$
   - Alle toppen worden systematisch doorlopen door steeds zo diep mogelijk verder te gaan. De momenteel bezochte top wordt de **actieve top** genoemd. Aan iedere top wordt een label toegekent, k genoemd, deze labels tellen gewoon de knopen.
   - Als alle toppen adjacent met de actieve top bezocht zijn beschouwen we dit als een **doodlopende top** en er wordt op de stappen teruggekeerd (**backtracking**) naar een top waar wel nog niet-bezochte adjacente knopen zijn.

   Dit algoritme is gemakkelijk recursief te programmeren (pg. 225) en men gebruikt best de adjacentielijstvoorstelling van de graaf. Het label k dat wordt toegekent aan een top v wordt de **diepte-eerst-index** van v genoemd en genoteerd als dfi(v). Als het algoritme klaar is kan men dus aan de dfi(v) nummers zien in welke volgorde de boom werd doorlopen. Op deze manier construeert het algoritme dus een **diepte-eerst-woud** als de graaf opspannend is (bevat alle toppen) en als hij daarbovenop nog eens samenhangend (er is tussen elke twee toppen een pad) is dan bekomt men een **diepte-eerst-boom**.

   Dit kan echter ook met een stapel worden geïmplementeerd (pg. 227). Als we een top voor de eerste keer tegenkomen duwen we hem op de stapel. Als een top doodloopt halen we hem van de stapel en gaan terug naar de vorige op de stapel.

   Uitgaande van de adjacentielijstvoorsteling wordt elke boog hoogstens tweemaal door het algoritme bekeken en voor een graaf G met n toppen en m bogen is de complexiteit dus gegeven door $\Theta(n + m)$.

Moest echter de adjacentiematrixvoorstelling gebruikt worden dan zou het algoritme $\Theta(n^2)$ zijn. DFS wordt onder andere gebruikt om te controleren of een graaf samenhangend, acyclisch of bipartiet is.

2. **breedte-eerst-doorlopen**: dit algoritme gaat uit van volgende gedachten:

   - gegeven: een graaf $G = (V, E)$, starttop $v \in V$ wordt hier ook **wortel** r genoemd.
   - Alle toppen worden systematisch doorlopen door steeds zo breed mogelijk verder te gaan, als alle adjacente toppen bezocht zijn wordt de nieuwe actieve top de minst recent bezochte.

   Ook hier wordt best de adjacentielijstvoorstelling gebruikt. Het algoritme wordt standaard geïmplementeerd door middel van een **wachtlijn** (FIFO). Als een top voor de eerste keer bezocht wordt, wordt hij achteraan de wachtlijn toegevoegd. De huidige actieve top is deze vooraan in de wachtlijn, van zodra zijn buren bezocht zijn wordt hij uit de wachtrij verwijderd. Als de wachtlijn leeg is, en nog niet alle toppen zijn bezocht, dan selecteren we een niet bezocht top, kennen een label toe en voegen hem toe in de wachtlijn. Zo vindt dit algoritme een **breedte-eerst-woud** of een **breedte-eerst-boom**. Elke boog wordt in de adjacentielijstvoorstelling hoogstens tweemaal bekeken en voor een graaf G met n toppen en m bogen is de complexiteit dus gegeven door $\Theta(n + m)$.

   Ook dit algoritme kan gebruikt worden om te controleren of een graaf samenhangend, acyclisch of bipartiet is. Het kan echter, indien goed wordt gelabeld, ook het kortste (ongewogen) pad van starttop v naar alle andere toppen bepalen.

We bespreken kort het simuleren met de computer. Als grote hoeveelheden moeten worden gesimuleerd is dit soms te complex om uit te rekenen, men bootst het gedrag dan na met de computer. Meestal simuleert men **gebeurtenissen** en hoe de verdeling op verschillende tijdstippen voorkomt. Veelal gebruikt men hiervoor wachtlijnen of prioriteitswachtlijnen. Om dit te doen zijn er twee manieren:

1. **discrete tijdsgestuurde simulatie**: elke **kloktik** verplaatst de simulatie naar een volgend tijdspunt. De duur van de simulatie hangt dus niet af van de inputgrootte maar van het aantal kloktikken.

2. **gebeurtenisgestuurde simulatie**: de **simulatieklok** wordt steeds naar de volgende gebeurtenis verplaatst. Op deze manier is de simulatie afhankelijk van de inputgrootte.

8 Brute kracht en recursie

Niet alle algoritmen kunnen rechtlijnig worden beschreven en soms moet men gewoon alle mogelijkheden uitproberen via brute kracht (**exhaustieve zoekmethode**) en willekeurig diep genestelde lussen. Dit laatste kan met recursie worden gesimuleerd. **Backtracking-algoritmen** zijn exhaustieve zoekalgoritmen waarbij wordt teruggegaan op stappen indien men ziet dat deze niet tot een oplossing leiden. Backtracking algoritmen kunnen door middel van een **toestandsboom** worden gevisualiseerd. De wortel van de boom stelt de toestand aan het begin van het algoritme voor, de toppen op niveau 2 de mogelijke keuzes in de tweede stap,... Een top is **veelbelovend** als hij tot een oplossing kan leiden, anders is hij **doodlopend**.

Bij het generen van permutaties wordt gevraagd alle mogelijke configuraties van elementen van die rij uit te schrijven. Het basisidee is om elk element eens achteraan te plaatsen, elk overblijvend element eens op de voorlaatste plaats, etc. Voor permutaties van een beperkte lengte gaat dit met genestelde for-lussen en plaats en herstelbewerkingen (pg. 236-237). Aangezien voor rijen van willekeurige lengte een willekeurig aantal for-lussen zou moeten genesteld worden dringt een andere schrijfwijze zich op.
Door het toepassen van een **verminder-en-heers** techniek, de probleemgrootte wordt telkens met een constante 1 verminderd, kan dit gemakkelijk recursief worden opgelost. Men plaatst dan steeds een element achteraan en roept voor de overige elementen de methode opnieuw op (pg. 238).

In het **acht-koninginnen-probleem** wordt gevraagd om acht koninginnen op een schaakbord (8×8) te plaatsen zonder dat ze elkaar bedreigen. Een koningin bedreigt stukken in dezelfde rij, dezelfde kolom en dezelfde diagonaal. Een backtracking algoritme is hier het meest geschikt voor. Volgende gedachten zijn achterliggend:

- Er moet juist 1 koningin per kolom en per rij geplaatst worden omdat ze elkaar anders bedrijgen.
- We kunnen de koninginnen dus kolom per kolom proberen te plaatsen en passen dus een verminder-en-heers strategie toe.
- Als de koninginnen nog niet allemaal geplaatst zijn, maar er toch geen mogelijke plaats meer is keren we terug op de stappen en proberen we een andere mogelijkheid (backtracking).
- We implementeren via recursie (pg. 240).

De representatie van het schaakbord en de koninginnen hebben grote invloed op de snelheid van het algoritme. Zo is de voordehandliggende tweedimensionale array niet geschikt omdat het moeilijk is om de diagonalen te controleren. Een mogelijke keuze wordt gegeven op pagina 242. In het algemeen zijn backtracking algoritmen niet efficiënt en neemt de tijdsduur typisch exponentieel toe.

9 Gretige algoritmen

Gretige algoritmen worden typisch gebruikt in optimalisatieproblemen, waarbij een beste oplossing moet worden gevonden. Gretige algoritmen werken in fasen en in elke fase wordt de tot dan toe beste oplossing gekozen, er wordt dus een lokaal optimale keuze gemaakt, vandaar de naam. Als het algoritme eindigt hoopt men dat de dan lokale optimale oplosing ook de globale optimale is. Indien dit niet zo is, werd een **suboptimale** of benaderende oplosing gevonden. Daarom worden gretige algoritmen ook soms voor benaderende waarden ingezet.

Een planningsprobleem wordt typisch op een gretige manier opgelost. Gegeven zijn taken $j_1, j_2, \ldots, j_n$ met gekende uitvoeringstijden $t_1, t_2, \ldots, t_n$. Gevraagd is de volgorde om de taken uit te voeren zodat de gemiddelde eindtijd zo klein mogelijk is, bijvoorbeeld de job scheduler in een besturingssysteem. Het gretige algoritme voor dit probleem bestaat er gewoon in de taken volgens stijgende uitvoertijd te sorteren (pg. 248). Inderdaad zij $j_{i_1}, j_{i_2}, \ldots, j_{i_n}$ de volgorde van de taken in het schema dan eindigt de eerste taak op t_{i_1}, de tweede op $t_{i_1} + t_{i_2}$,... De totale kost C is dus gelijk aan:

$$C = \sum_{k=1}^{n}(n-k+1)t_{i_k} \text{ of dus } C = (n+1)\sum_{k=1}^{n} t_{i_k} - \sum_{k=1}^{n} k t_{i_k} \tag{10}$$

IN deze laatste voorstelling beïnvloed $\sum_{k=1}^{n} k t_{i_k}$ de totale kost. Stel dat er in een ordening een $x > y$ bestaat waarvoor $t_{i_x} < t_{i_y}$, dan is gemakkelijk in te zien dat een omwisseling van j_{i_x} en j_{i_y} de sommatie verroot, het geen de totale kost C vermindert. Dit wil zeggen dat elke volgorde van de taken waarin de uitvoeringstijden niet monotoon niet-dalend zijn geen optimale oplossing kan zijn.

Het wisselgeldprobleem gaat als volgt. Gegeven een munteenheid met muntstukken met waarden $c_1, c_2, \ldots, c_n$, wat is het minimale aantal stukken nodig om een bedrag k te vormen. Een gretige benadering is om steeds het grootst mogelijke muntstuk c toe te voegen. Dit is gretig want er wordt in elke fase een beslissing genomen die er goed uit ziet zonder aandacht te besteden aan de gevolgen in de toekomst. Er zijn echter voorbeelden te vinden waarvoor dit niet steeds correct is (pg. 250-251).

Voor het opstellen van een **minimale-kost opspannende boom** zijn twee algoritmes:

1. **MST-algoritme van Kruskal**: het idee bestaat erin telkens de kleinste boog toe te voegen op voorwaarde dat de graaf nog steeds acyclisch is. Hierbij is het handig om te beginnen met het sorteren van de bogen van G zodanig dat $w(e_1) \leq w(e_2) \leq \cdots \leq w(e_m)$, in pseudocode op pg. 252.

We bewijzen nu dat dit altijd optimaal zal zijn. Zij G een samenhangende gewogen graaf met n toppen. Zij T de deelgraaf die door het algoritme van Kruskal bekomen werd. Het is onmiddelijk duidelijk dat T een opspannende boom van G is. Stel dat de bogen $E(T) = \{e_1, e_2, \ldots, e_{n-1}\}$ zo gelabeld zijn dat $w(e_1) \leq w(e_2) \leq \cdots \leq w(e_{n-1})$. Dan is het gewicht van T gegeven door:

$$w(T) = \sum_{i=1}^{n-1} w(e_i) \tag{11}$$

Het bewijs gebeurt uit het ongerijmde. We veronderstellen dat T geen minimale-kost opspannende boom is. Uit de opspannende bomen met minimale kost van G selecteren we er één die een maximaal aantal bogen met T gemeen heeft; we noemen deze boom H. Wegens de veronderstelling zijn de bomen H en T niet identiek, zodat T ministens één boog heeft die niet tot H behoort. Zij $e_i (1 \leq i \leq n-1)$ de eerste boog van T die niet tot H behoort, en definieer $G_0 = H + e_i$. Dan heeft G_0 precies één cykel C. Aangezien T geen cykels heeft, is er een boog e_0 van C die niet tot T behoort. De graaf $T_0 = G_0 - e_0$ is ook een opspannende boom van G, en er geldt dat

$$w(T_0) = w(H) + w(e_i) - w(e_0) \tag{12}$$

Aangezien $w(H) \leq w(T_0)$, volgt hieruit dat $w(e_0) \leq w(e_i)$. Door het algoritme van Kruskal is e_i een boog met minimaal gewicht zodanig dat $\langle \{e_1, e_2, \ldots, e_{i-1}\} \cup \{e_i\} \rangle$ acyclisch is. Echter, $\langle \{e_1, e_2, \ldots, e_{i-1}, e_0\} \rangle$ is een deelgraaf van H en is dus acyclisch, zodat $w(e_i) = w(e_0)$. Dus $w(T_0) = w(H)$, hetgeen impliceert dat ook T_0 een minimale-kost opspannende boom van G is. Maar T_0 heeft meer bogen gemeenschappelijk met T dan H, wat in tegenstrijd is met eerdere veronderstelling.

Het sorteren van de bogen kan in $\Theta(m \log n)$ tijd, maar het controleren of een boog toevoegen een cykel zou veroorzaken is moeilijker. Om dit efficiënt bij te houden worden de componenten van de boom in opbouw bijgehouden op een zodanige manier dat snel kan worden gecontroleerd of twee toppen tot dezelfde component behoren. Dan geldt dat:

- Bogen tussen toppen in verschillende componenten zijn toegelaten
- Bogen tussen toppen in dezelfde component zijn niet toegelaten.

Door voor elke top bij te houden wat de kleinste top in zijn component is kan de controle nu in $\Theta(1)$ tijd gebeuren, het bijhouden en controleren neemt wel $\Theta(n^2)$ tijd. Dit maakt dat de totale kost $\Theta(m \log n + n^2)$ is (pg. 253-254).

2. **MST-algoritme van Prim**: er wordt willekeurig een top gekozen uit de graaf G en toegevoegd aan de nieuwe boom T. Vervolgens wordt T in elke stap uitgebreid door eer een bog aan toe te voegen die op volgende manier is gekozen: een boog van minimaal gewicht die een top in T verbindt met een top die niet tot T behoort. Het algoritme stopt als alle toppen van G ook tot T behoren. Er kan op dezelfde manier als vorig bewijs bewezen worden dat dit correct is. Een eenvoudige implementatie heeft kost $\Theta(nm)$ maar met gebruik van een prioriteitswachtlijn kan men de kost reduceren tot $\Theta(m \log n)$.

Het meest bekende algoritme voor het bepalen van het pad met de minimale **gewogen afstand** in een graaf is het algoritme van **Dijkstra**. De afstand $d(s,t)$ in een graaf is het gewicht van het kortste pad. Het algoritme is enkel geldig voor grafen met niet-negatieve booggewichten. Het algoritme geeft aan elke top v een label $\ell(v) = d(u_0, v)$. Initieel is $\ell(u_0) = 0$ en alle andere labels zijn ∞. De toppen kijrgen dan in iedere stap een voorlopig label wat een bovengrens is voor de afstand, ook in iedere stap wordt de top met het kleinste voorlopig label definitief gelabeld. De werking en pseudocode staan op pagina's 258-259 beschreven.

We bewijzen nu dat dit steeds de optimale oplossing voortbrengt. Zij $G = (V, E)$ een gewogen graaf met n toppen. Het algoritme van Dijkstra bepaalt nu de afstand van een vaste top van G. Met andere woorden na afloop van het algoritme is

$$\ell(v) = d(u_0, v) \text{ voor alle } v \in V \tag{13}$$

Bovendien, wanneer $\ell(v) \neq \infty$ en $v \neq u_0$, dan is

$$Q : u_0 = t_0, t_1, t_2, \ldots, t_k = v \tag{14}$$

een kortste $u_0 - v$-pad, waarbij $t_{i-1} = p(t_i)$, voor $i = 1, 2, \ldots, k$.

Wij bewijzen eerst (13). Het volstaat om dit aan te tonen in het geval dat G een samenhangende graaf is. Immers, wanneer G niet samenhangend is, dan zijn de toppen van G die niet door een pad met u_0 zijn verbonden, precies die toppen die als label ∞ hebben bij het beëindigen van het algoritme. Het bewijs gebeurt door inductie op i. We tonen aan dat, nadat $u_i (0 \leq i \leq n-1)$ bepaald is, er geldt dat

$$\ell(v) = d(u_0, v) \text{ voor alle } v \in S_i = \{u_0, u_1, \ldots, u_i\} \tag{15}$$

Het is onmiddelijk duidelijk dat dit geldig is voor $i = 0$. Veronderstel vervolgens dat (15) geldig is voor een zekere $i, 0 \leq i < n-1$; we tonen aan dat (15) ook geldt voor $i+1$. Het volstaat om aan te tonen dat $\ell(u_{i+1}) = d(u_0, u_{i+1})$. Uit de werking van het algoritme weten we dat $u_{i+1} = \min\{\ell(v) | v \in \overline{S_i}\}$. Dus

$$\begin{aligned} \ell(u_{i+1}) &= \min\{\ell(v) | v \in \overline{S_i}\} \\ &= \min\{\ell(u) + w(uv) | u \in S_i, v \in \overline{S_i}, uv \in E(G)\} \\ &= \min\{d(u_0, u) + w(uv) | u \in S_i, v \in \overline{S_i}, uv \in E(G)\} \end{aligned}$$

waarbij de laatste gelijkheid volgt uit de inductiehypothese. Het minimum in de laatste gelijkheid treedt op voor $v = u_{i+1}$, zodat dus wegens $d(u_0, y) = d(u_0, x) + w(xy)$ geldt dat $\ell(u_{i+1} = d(u_0, u_{i+1})$. Om (14) te bewijzen gaan we als volgt te werk. Zijk k het aantal bogen op het pad van v naar u_0 via de voorgangers p, en noem $t_k = v$. Na afloop van het algoritme is $\ell(v) = \ell(t_{k-1}) + w(t_{k-1}v)$, voor een zekere top t_{k-1} waarvoor $p(v) = t_{k-1}$ en

$$d(u_0, v) = d(u_0, t_{k-1}) + w(t_{k1}v)$$

Dit feit impliceert dat t_{k-1} de op een na laatste top is op een kortste u_0-v-pad. Op deze manier werder werkend construeren we ene kortste u_0-v-pad

$$P : u_0 = t_0, t_1, \ldots, t_{k-1}, t_k = v$$

waarbij $t_{i-1} = p(t_i)$, voor $i = 1, 2, \ldots, k$.

Een eenvoudige implementatie van het algoritme leidt tot een $\Theta(n^2)$ complexiteit. Men kan echter door gebruik van een prioriteitswachtlijn om de voorlopig gelabelde toppen bij te houden het algoritme reduceren tot een $\Theta(m \log n)$ algoritme.

Het **handelsreizigerprobleem (TSP)** bestaat er in om een hamiltioniaanse (bevat alle toppen) cykel te zoeken met het kleinste gewicht, veelal wordt verondersteld dat de graaf compleet is, zodat er zeker hamiltoniaanse cykels bestaan. Het eenvoudig exact exhaustief algoritme berekent alle permutaties van n toppen en houdt de rondreis met de kleinste kost bij. Dit is echter een **onhandelbaar** probleem en daarom zijn benaderende algoritmen, die **een korte** rondreis teruggeven, noodzakelijk.

Deze benaderende algoritmen rekenen er op dat de gewichtsfunctie in de graaf voldoet aan de driehoeksongelijkheid. In realiteit is dit meestal ook zo en het probleem wordt dan het **euclidische handelsreizigersprobleem** genoemd. Er zijn twee manieren:

1. **gretige methode** (pg. 262): geeft een benaderende oplossing met performantiegarantie. De **performantie** van de benadering duidt aan hoe goed of slecht een bekomen benaderende oplossing is in vergelijking met de optimale oplossing. Hoe lager de performantiegarantie hoe beter.

2. **benadering met MST/DFS** (pg. 265): geeft performantiegarantie 2, dit wil zeggen dat het bekomen resultaat maximaal 2 maal groter is als het optimale resultaat.

10 Sorteeralgoritmen

Een belangrijke tool bij sorteeralgoritmen zijn **beslissingsbomen**, de interne toppen stellen de uitgevoerde vergelijkingen voor en de bladeren stellen de mogelijke outputs van het algoritme voor. Als men een sorteeralgoritme ontwerpt maakt men best dat dit **generisch is**, dit wil zeggen dat het een algemeen algoritme is die gelijk welke objecten kan sorteren zolang er een ordening op die objecten gedefinieerd is.

Het begrip **inversie** van een rij getallen $(a_1, \ldots, a_n)$ is elk geordend paar (i, j) waarvoor geldt dat $i < j$ mar $a_i > a_j$. Een gesorteerde rij heeft dus geen inversies. Dit begrip is belangrijk om de gemiddelde uitvoeringstijd van een sorteeralgoritme te bepalen omdat men een formule heeft voor het gemiddeld aantal inversies. Dit laatste kan wel enkel als er *geen dubbels* in de rij zitten en we kunnen de inputrij steeds als een permutatie van de natuurlijke getallen beschouwen met een gelijke kans op elke permutatie. We bewijzen nu dat het gemiddelde aantal inversies in een rij van n verschillende getallen gegeven is door:

$$n(n-1)/4 \tag{16}$$

Bewijs. Voor een rij A noemen we A_r de rij in omgekeerde volgorde. Beschouw twee willekeurige getallen (x, y) in de rij waarvoor $y > x$. Dit paar correspondeert met een inversie in ofwel A ofwel A_r. Het totale aantal dergelijke paren voor een rij A (en zijn omgekeerde A_r) is gegeven door $n(n-1)/2$. Het aantal inversies in een gemiddelde lijst is dus de helft hiervan, of $n(n-1)/4$).

We bespreken nu de verschillende sorteeralgoritmen:

- **bubblesort**: elementen worden steeds per twee bekeken, indien nodig worden ze omgewisseld. Op deze manier gaat het grootste element naar achter in de rij. Nu begint het algoritme opnieuw en het tweede grootste element komt op de voorlaatste plaats. Dit gaat door tot de hele rij gesorteerd is. We bewijzen nu dat bubblesort een $T(n) = \Theta(n^2)$ uitvoeringstijd heeft.

  Bewijs. Het aantal vergelijkingen $C(n) = n(n-1)/2$ is hetzelfde voor alle mogelijke rijen van lengte n. Het aantal verwisselingen S_n is afhankelijk van de inputrij en kan in het beste geval $S_b(n) = 0$ zijn of in het slechtste geval $S_s(n) = n(n-1)/2$ zijn. De totale complexiteit is dus $T(n) = \Theta(n^2)$

- **selectionsort**: het grootste element in de rij wordt bepaald en achteraan geplaatst, vervolgens wordt het tweede grootste element in de rij bepaald en achteraan geplaatst, enzovoort. We bewijzen nu dat sorteren door selectie een tijdscomplexiteit van $T(n) = \Theta(n^2)$ heeft.

  Bewijs. Het aantal vergelijkingen is gegeven door $C(n) = n(n-1)/2$, want in elke stap van de dubbele for-lus gebeurt een vergelijking. Het aantal verwisselingen is hoogstens $S_s(n) = n-1$, hetgeen onmiddelijk duidelijk is uit de implementatie: de verwisseloperatie staat in de buitenste lus en kan dus hoogstens $n-1$ keer worden uitgevoerd. Het kan gebeuren dat er geen enkele verwisseling nodig is, nl. als de gegeven rij al gesorteerd is, m.a.w. $S_b(n) = 0$. De totale tijdscomplexiteit is dus $T(n) = \Theta(n^2)$.

- **insertionsort**: In de begintoestand is het eerste element, op zichzelf beschouwd, gesorteerd. De basisbewerking is het rangschikken van de elementen op de posities 1 tot en met $i, 2 < i < n$. Daarbij wordt verondersteld dat de elementen op posities 1 tot en met i reeds gesorteerd zijn en het nieuwe element dus op de juiste plaats kan worden tussengevoegd. We bewijzen dat sorteren door tussenvoegen een slechste-geval uitvoeringstijd $T_s(n) = \Theta(n^2)$ en en beste-geval uitvoeringstijd van $T_b(n) = \Theta(n)$ heeft.

  Bewijs. In het slechtste geval is het aantal stappen uitgevoerd door de dubbele for-lus gegeven door $n(n-1)/2$. Elke stap komt overeen met een vergelijking en een verwisseling, dus de uitvoeringstijd in het slechtste geval is $\Theta(n^2)$. Als echter de rij al gesorteerd was voor het algoritme is de uitvoeringstijd $\Theta(n)$ aangezien de binnenste lus dan steeds faalt. Dit is het best mogelijke geval want de buitenste lus heeft

steeds $n-1$ stappen.

We bewijzen nu ook dat sorteren door tussenvoegen een gemiddelde uitvoeringstijd $T_g(n) = \Theta(n^2)$ heeft. Merk op dat het aantal inversies in de te sorteren rij precies gelijk is aan het aantal keer dat de verwisselopdracht (a_j, a_{j-1}) uitgevoerd wordt. Dus als ker k inversies zijn bij de start van het algoritme, dan moeten er k verwisselingen gebeuren. Aangezien er verder $\Theta(n)$ ander werk nodig is in het algoritme, is de uitvoeringstijd gegeven door $\Theta(k+n)$. Het gemiddeld aantal inversies is $\Theta(n^2)$ waaruit de gemiddelde kwadratische tijd volgt.

De complexiteit van deze algoritmen kunnen dus in volgende tabel worden samengevat:

	# vergelijkingen		# verwisselingen	
Sorteeralgoritme	$C_s(n)$	$C_b(n)$	$S_s(n)$	$S_b(n)$
bubblesort	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$	0
selectionsort	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n)$	0
insertionsort	$\Theta(n^2)$	$\Theta(n)$	$\Theta(n^2)$	0

Alle bovenstaande algoritmen hebben een gemiddelde uitvoeringstijd van $\Theta(n^2)$. Een belangrijke reden waarom deze zo traag zijn is omdat ze *enkel naast elkaar liggende of aangrenzende* elementen met elkaar vergelijken en verwisselen. Dit is een cruciale factor zoals bijkt uit volgende stelling. Om het even welk algoritme dat sorteert door het vergelijken en verwisselen van aangrenzende elementen, vereist gemiddeld $\Omega(n^2)$ uitvoeringstijd.

Bewijs. Het gemiddelde aantal inversies bij het begin van het algoritme is $n(n-1)/4$. Elke verwisseling vermindert het aantal inversies met precies één, zodat dus $\Omega(n^2)$ verwisselingen vereist zijn.

Om een sneller algoritme te ontwrpen moeten dus elementen die ver uit elkaar liggen worden verwisseld en moet meer dan 1 inversie per verwisseling worden geëlimineerd. **Mergesort** maakt gebruik van recursie om dit te doen en maakt gebruik van volgend stramien:

- verdeel een rij van n elementen in twee deelrijen van $\lceil n/2 \rceil$ of $\lfloor n/2 \rfloor$ elementen.
- sorteer de twee kortere deelrijen op dezelfde recursieve manier.
- voeg de gesorteerde deelrijen samen tot één gesorteerde rij ("merge").

We bewijzen nu dat mergesort een tijdscomplexiteit van $T(n) = \Theta(n \log n)$ heeft. Het mergesort algoritme is een recursief algoritme die het probleem van grootte n opsplitst in twee deelproblemen van grootte $n/2$. De complexiteit wordt dus gegeven door de recurrente betrekking:

$$T(n) = 2T(n/2) + f(n)$$

De master-methode zegt dan dat indien $f(n) = \Theta(n)$ de complexiteit $T(n) = \Theta(n \log n)$ is. We moeten dus bewijzen dat het mergen van twee gesorteerde rijen in $\Theta(n)$ tijd kan. Het standaard algoritme neemt twee inputrijen A en B en maakt een outputrij C aan. In elke rij wordt een huidige positie bijgehouden, die start bij het eerste element van de rij. Het kleinste van de twee huidige elementen in A en B wordt gekopieerd in de volgende component van C en de huidige posities worden waar nodig aangepast. Wanneer een van beide inputrijen volledig verwerkt is, wordt de rest van de andere inputrij naar C gekopieerd. Dit heeft dus altijd een $\Theta(n)$ uitvoeringstijd. We hebben dus bewezen dat de slechste-geval, gemiddelde-geval en beste-geval uitvoeringstijd altijd $\Theta(n \log n)$ tijd kost daar het mergen altijd $\Theta(n)$ tijd kost.

Quicksort is het snelste sorteeralgoritme dat we kennen met een gemiddelde uitvoeringstijd van $\Theta(n \log n)$ en een slechtste uitvoeringstijd van $\Theta(n^2)$. Er zijn echter manieren om ervoor te zorgen dat dit slechste geval bijna nooit voorkomt. Het quicksort algoritme gaat als volgt:

- Kies om het even welk element in de rij. Dit element wordt nu de **spil** of **pivot** genoemd.

- Partitioneeer de overblijvende elementen als volgt, de linkerdeelrij bestaat uit elementen kleiner of gelijk aan de spil, de rechterdeelrij bestaat uit elementen groter of gelijk aan de spil.
- Sorteer beide deelrijen recursief op de zelfde manier.
- De gesorteerde rij bestaat uit de linker deelrij, de spil en de rechter deelrij.

In het basisgeval moet ook de mogelijkheid voor een lege rij worden voorzien omdat dit bij de partitionering kan optreden. Vanwege deze vele recursieve oproepen is het beter om voor een rij van slechts enkele elementen een simpel algoritme zoals insertionsort te gebruiken omdat dit sneller zal zijn. Experimenten hebben aangetoond dat elke waarde tussen 5 en 20 elementen een goede drempelwaarde is om een simpel algoritme toe te passen.

Net zoals bij mergesort lost dit algoritme de twee deelproblemen recursief op, maar anders is dat de deelproblemen niet steeds van gelijke grootte zijn. Onder andere de keuze van de spil is belangrijk. Zo is het algoritme niet duidelijk over wat moet gedaan worden bij duplicaten van de spil, het is enkel vereist dat het duplicaat in één van de twee deelrijen gaat. Dit moet dus zo efficiënt mogelijk gebeuren. Een voordeel boven merge-sort is dat de partitioneringsstap veel sneller kan gebeuren. Er zijn drie mogelijke gevallen die kunnen voorkomen:

1. **best mogelijke partitionering**: de rij wordt telkens gehalveerd wat gelijk is aan de merge-sort situtatie. We bekomen dus recurrente betrekking $T_b(n) = 2T_b(n/2) + \Theta(n)$ waaruit met de master methode kan besloten worden dat $T_b(n) = \Theta(n \log n)$.
2. **slechtst mogelijke partitionering**: de linker of rechter deelrij is telkens leeg en we bekomen dan de recurrente betrekking $T_s(n) = T_s(n-1) + \Theta(n)$ wat door middel van de iteratiemethode duidt op $T_s(n) = \Theta(n^2)$.
3. **gebalanceerde partitionering**: zonder bewijs wordt de uitvoeringstijd gegegevn door $T_g(n) = \Theta(n \log n)$, zelfs al is de verdeling bv. 99-1.

Het is dus belangrijk om er bij de implementatie voor te zorgen dat het slechtste geval wordt vermeden. Ook moet men voor speciale inputs zoals reeds gesorteerde rijen en rijen van gelijke elementen oppassen. De keuze van de spil is belangrijk en we kunnen volgende gevallen onderscheiden:

- **eerste of laatste element uit de rij**: problemen bij reeds gesorteerde input
- **middelste element uit de rij**: goed voor reeds gesorteerde input en de kans op kwadratische uitvoeringstijd voor een willekeurge rij is klein. Dit is echter een passieve keuze omdat niet actief gezocht wordt naar de beste spil.
- **mediaan van drie**: een goede keuze voor de spil zou de mediaan zijn (het $\lceil n/2 \rceil$-de kleinste getal in een groep van n getallen). Echter kost het berekenen van de mediaan veel tijd waardoor dit te traag zou zijn. Daarom schat men de mediaan door die van een deelgroep te berekenen, hoe groter de deelgroep hoe naukeuriger maar hoe trager. In de praktijk wordt dikwijls de mediaan van het eerste, middelste en laatste element berekend. Bij een reeds gesorteerde rij komt dit dus neer op het middelste element wat de beste keuze is.
- **random element uit de rij**: dit wordt soms als alternatief gebruikt.

Ook voor het partitioneren van de rij zijn verschillende strategëen mogelijk. We bespreken er twee:

- **basispartitionering**: Deze strategie bestaat uit drie stappen en gaat er vanuit dat alle elementen verschillend zijn:
  1. Het spilelement wordt achteraan geplaatst door het te verwisselen met het laatste element.
  2. Alle elementen kleiner dan de spil worden naar links verschoven, alle elementen groter dan de spil naar rechts. Dit gebeurt door de linkerdeelrij voorwaarts te doorlopen en te zoeken naar elementen groter dan de spil en door de rechterdeelrij achterwaarts te doorlopen en te zoeken naar elementen kleiner dan de spil. Deze twee gevonden elementen worden dan verwisselt. Dit wordt herhaald totdat de twee 'huidige posities' elkaar passeren.

3. De spil wordt verwisselt met het eerste element van de rechter deelrij.

   Er wordt dus geen extra geheugen gebruikt en dit gaat in de praktijk zeer snel, echter moet opgelet worden met gelijke sleutels, de lussen moeten zo stoppen dat sleutels gelijk aan de spil gelijk worden verdeeld over de deelrijen. Het beste is om onnodige verwisselingen te laten doen en zowel links als rechts te laten stoppen bij een element gelijk aan de spil.

- **mediaan-van-drie partitionering**: dit optimaliseerd het basisgeval. De beste manier om de mediaan van drie te bepalen is om die drie elementen te sorteren. Op deze manier is het eerste element al kleiner dan de spil en het laatste groter. Daarom kunnen we de spil nu met het voorlaatste element verwisselen. Ook kan het doorlopen van de rijen nu respectievelijk bij het tweede en het derdelaatste element starten. Ook zullen de lussen nu makkelijk stoppen wegens de zekere **sentinels**.

Sommige sorteeralgoritmen maken gebruik van voorkennis omtrent de input en kunnen dus niet zomaar alles sorteren en zijn ook niet op vergelijkingen gebaseerd. We geven er twee:

- **countingsort**: dit kan worden gebruikt voor het sorteren van een rij $(a_1, \ldots, a_n)$ waarbij de sleutels a_i gehele getallen zijn uit een interval $[0, k]$ voor een kleine k. Het algoritme telt hoeveel keer elke waarde voorkomt en berekent op basis daarvan de positie van elk element uit. Deze berekening gebeurt door het optellen van de aantallen. Daarna kan door éénmaal de rij te doorlopen elk element op zijn positie worden gezet (pg. 306-307). Dit algoritme is stabiel. We bewijzen nu dat de uitvoeringstijd $T(n) = \Theta(n)$ als $k = O(n)$.

  Bewijs. Elke lus is ofwel $\Theta(n)$ ofwel $\Theta(k)$. De uitvoeringstijd van het algoritme is dus $T(n) = \Theta(n + k)$. Wanneer $k \leq n$, of meer algemeen, wanneer k een functie van n is waarvoor $k = O(n)$, dan wordt k verwaarloosbaar tegenover n en krijgen we dus een lineaire uitvoeringstijd $T(n) = \Theta(n)$ voor het algoritme.

- **Radixsort**: dit kan worden gebruikt voor het sorteren van een rij $(a_1, \ldots, a_n)$ van positieve gehele getallen die een beperkt aantal cijfers hebben. Om deze rij te sorteren wordt eerst een stabiel sorteeralgoritme zoals countingsort gebruikt om de getallen op hun laatste cijfer te sorteren, vervolgens op hun voorlaatste cijfer, ..., tot aan het eerste cijfer. We bewijzen dat de uitvoeringstijd van radixsort $T(n) = \Theta(n)$ is als het aantal cijfers m begrensd is tot een constante.

  Bewijs. De uitvoeringstijd van het sorteren in elke stap van de for lus is $\Theta(n + 9) = \Theta(n)$. Aangezien de for lus m stappen heef, is de totale complexiteit dus $T(n) = \Theta(mn)$. We stellen echter $\Theta(m) = \Theta(1)$ waardoor de totale uitvoeringstijd gelijk is aan $T(n) = \Theta(n)$.

Een **selectieprobleem** bestaat in het vinden van het k-de kleinste element in een rij van n geordende objecten. Speciale gevallen zijn het grootste en kleinste element die elk in lineaire tijd kunnen worden gevonden. Een eenvoudig algoritme voor het vinden van een willekeurig k-de kleinste element is het sorteren van de rij en dan het element te kiezen. De tijdscomplexiteit hangt dan af van het gekozen sorteeralgoritme en kan dus minimaal $\Theta(n \log n)$ zijn.
Het kan echter in gemiddeld $\Theta(n)$ tijd indien we van partitioneren gebruik maken (pg. 301). Dit partitioneren kan in lineaire tijd. Zij ℓ het aantal element uit de linker deelrij, wanneer $\ell = k - 1$ dan is de spil precies het gevraagde k-de kleinste element. Waneer $\ell \geq k$, dan moet verder worden gezocht nar het element dat in de rechterdeelrij het $(k - \ell - 1)$-de kleinste is.

Om ondergrenzen voor bepaalde soorten algoritmen gebaseerd op vergelijkingen, ze maken dus geen gebruik van andere informatie over de elementen dan die bekomen uit het vergelijken ervan, te bewewijzen worden beslissingsbomen gebruikt. Een besslissingsboom is een binaire of ternaire gewortelde boom die een algoritme met een reeks vergelijkingen voorstelt. De interne toppen zijnde vergelijkingen en de bladeren zijn de mogelijke outputs van een algoritme (pg. 303-304). Een sorteeralgoritme wordt als volgt door een binaire beslissingsboom T voorgesteld:

- Interne top is vergelijking van twee elementen van de vorm $a_i < a_j$.
- Linkertak wordt gevolgd als de vergelijking waar is anders wordt de rechtertak gevolgd.
- De bladeren zijn gesorteerde rijen.

Nu kan men gemakelijk het verband tussen het aantal stappen in een algoritme en een beslissingsboom inzien:

- De reeks vergelijkingen door een algoritme is het pad van de wortel naar het blad.
- Het slechste geval voor het algoritme is het langste dergelijk pad dus $C(n) = h(T)$ of de slechste uitvoeringstijd is gelijk aan de hoogte van de boom.

We bewijzen nu volgende stelling. Zij $C(n)$ het aantal vergelijkingen nodig om n elementen te sorteren met een sorteeralgoritme gebaseerd op vergelijkingen. Dan is $C(n) = \Omega(n \log n)$.

Bewijs. We moeten dus bewijzen dat de hoogte van de beslissingsboom gelijk is aan $n \log n$. Aangezien n verschillende elementen op $n!$ manieren kunnen worden gerangschikt, moet T minstens $n!$ bladeren hebben. Een binaire boom met m bladeren heeft minstens een hoogte van $\log_2 m$ dus $h \geq \log_2(n!)$. Er geldt ook dat $\log_2(n!) = \Omega(n \log n)$ en dus verkrijgen we dat $C(n) = h \geq \log_2(n!) = \Omega(n \log n)$ hetgeen het gestelde bewijst.

11 Implementatie van stapels en wachtlijnen

Een `stack` wordt door middel van `arrays` geïmplementeerd omdat ze eenvoudig en efficiënt is. Hierbij geldt wel de beperking van een maximaal elementen en ongebruikte ruimte. Dit wordt echter opgelost met **dynamische arrays** die vergroten als er meer ruimte nodig is. Alle bewerkingen kunnen dus in $\Theta(1)$ tijd worden voltooid. Enkel de `push` bewerking kan $\Theta(n)$ tijd vragen als de achterliggende rij moet worden verdubbeld. Echter zal zo'n uitgebreide `push` bewerking voorafgegaan zijn door minstens $n/2$ `push`-bewerkingen die geen uitbreiding vereisen waardoor deze een geamortiseerde uitvoeringstijd van $\Theta(1)$ heeft (pg. 131).

Ook voor een `queue` wordt een `array` gebruikt. Een efficiënte implementatie beschouwd de `array` als circulair. Daardoor kunnen de `enqueue` en `dequeue` bewerkingen met dynamische rijen in geamortiseerde constante tijd kunnen (pg. 315-316).

12 Binaire hopen en heapsort

Een veel gebruikte voorstelling voor een prioriteitswachtlijn is een **binaire hoop** omdat deze in alle gevallen een $O(\log n)$ bewerkingstijd garandeert en eenvoudig met een `array` kan worden geïmplementeerd. Een binaire hoop bezit twee grote eigenschappen:

1. **structuureigenschap: links-complete binaire bomen**: daar de enige dynamische structuur die aanleiding geeft tot logaritmische basisbewerkingen een boom is, is het voordehandliggend dat een binaire hoop door een boom wordt voorgesteld. Om al slechtst mogelijke tijd de logaritmische te kunnen garanderen moet deze boom gebalanceerd zijn. Een **links-complete binaire boom** is een binaire boom waarvan alle niveau's volledig gevuld zijn, behalve eventueel het laatste, dat van links naar rechts gevuld is. Enkele eigenschappen zijn de volgende:
   - De hoogte van zo'n links-complete binaire boom met n toppen is $\lfloor \log_2 n \rfloor$.
   - Een links-complete binaire boom heeft een **impliciete representatie** aan de hand van een `array` (pg. 321).
2. **ordeningseigenschap**: een **prioriteitsboom** of **binaire hoop** is een links-complete binaire boom met als bijkomende ordeningseigenschap dat voor elke top X met ouder P de sleutel in P kleiner is dan of gelijk is aan de sleutel in X.

We bekijken nu enkele basisbewerkingen:

1. `findMin`: dit element kan in de wortel gevonden worden en kan dus in constante tijd worden teruggegeven.

2. `add`: na deze bewerking moet de ordeningseigenschap van de boom worden hersteld. Allereerst moet een top worden toegevoegd en dit kan enkel in de eerstvolgende vrije positie van de array. Indien de ordening nog klopt stopt de berwerking, anders worden de ouder en de nieuwe top omgewisseld. Dit omwisselen gaat door tot het nieuwe element op de juiste plaats staat en men noemt dit **opwaartse percolatie** (pg. 325).

3. `removeMin`: na deze bewerking moet de ordeningseigenschap van de boom worden hersteld. Als men dit element verwijdert, verwijdert men dus de wortel. Uit de structuureigenschap volgt dan dat het laatste element uit de array moet worden verplaatst. Meestal kan men dat laatste element niet op de plaats van de wortel zetten zonder de ordeningseigenschap te verstoren. Om dit op te lossen laten we de vrije positie verschuiven naar een plaats waar ze wel ken worden ingevuld door het laatste element. Dit noemt men **neerwaartse percolatie**. Het kleinste van de twee kinderen wordt bepaald en dit wisselt met de vrije positie (pg. 326).

Omdat het herstellen van de ordeningseigenschap relatief duur is kan een bewerking `toss` worden ingevoerd. Deze plaatst een nieuw element aan het einde van de array zonder herstellen. Dit is handig als meerdere elementen worden toegevoegd zonder dat er ondertussen elementen worden verwijderd of opgevraagd. Er kan nu een lineaire bewerking `fixHeap` worden gedefiniëerd die de boom in lineaire tijd herstelt. Dit zeer simpel algoritme (pg. 329-331) voert een neerwaartse percolatie door voor de toppen maar dan van de grootste naar de kleinste top. Dit is correct omdat alle nakomelingen van een top i al verwerkt zullen zij nop het moment dat de top i verwerkt wordt.

De binaire hoop ADT geeft aanleiding tot het **heapsort** algoritme welke als volgt werkt:

1. voeg de elementen to aan de binaire hoop door het oproepen van de `toss`-bewerking.

2. roep de bewerking `fixHeap` op

3. doe n oproepen van `removeMin`

De elementen worden nu in gesorteerde volgorde teruggegeven. De eerste twee stappen vragen lineaire tijd en elke oproep van `removeMin` vraagt $O(\log n)$ tijd. Als we stap 3 dus n keer oproepen bekomen we een sorteeralgoritme met $O(n \log n)$ als slechtst mogelijke uitvoeringstijd. Ook in het geheugen kan dit efficiënt gebeuren door elk element die met `removeMin` wordt verwijderd op de achterste vrije plaats te zetten. Ook kan omgekeerd worden gesorteerd als men een **maximum binaire hoop** gebruikt waarbij de wortel het grootste element is. Een implementatie staat op pagina 335.

13 Hashtabellen

Hashtabellen laten toe om door een verlies aan ordeningsinformatie een zeer snelle, gemiddeld constante, toegangstijd tot de elementen te garanderen. De elementen zijn hierbij benoemd en hebben een naam. Indien de elementen kleine niet-negatieve getallen tussen 0 en 65535 zijn kan een array worden gebruikt om de basisbewerkingen zoals opvragen, verwijderen en toevoegen te implementeren. Dit kan op volgende manier. Er wordt een `array` geïnitialiseerd van lengte 65535 met een waarde 0 in elke component. Volgende zijn dan de bewerkingen:

- **toevoegen**: het toevoegen van een element i gebeurt door het incrementeren van $a[i]$. Hierbij stelt $a[i]$ dus het aantal keer dat element i werd toegevoegd voor.

- **controleren**: om te controleren of een element i in de hashtabel zit moet gekeken worden of $a[i]$ niet 0 is.

- **verwijderen**: het verwijderen van een element i gebeurt door te controleren of het in de tabel zit en indien dit zo is het decrementeren van $a[i]$.

Echter wil men dit mogelijk maken voor elk willekeurig object. Opdat dit zou kunnen moet dat object dus een goede **hashfunctie** bevatten die een klein geheel getal teruggeeft. De definitie van een hashfunctie laat echter toe dat twee verschillende objecten de zelfde hash hebben, dit noemt men een botsing. Er zijn twee manieren waarop men hiermee omgaat:

1. **gesloten hashing**: men zoekt in de tabel zelf een andere plaats voor het element, dit kan weer op twee manieren gebeuren:

   (a) **lineaire peiling**: als een botsing plaatsvind dan wordt vanaf de ingenomen plaats verdergezocht tot de eerste vrije plaats. De tabel wordt hierbij als een circulaire structuur beschouwd. Ookal kan het toevoegen zo lang duren, als de tabel goed vol zit bijvoorbeeld, toch kan men gemiddeld verwachten dat maar de halve tabel zal moeten worden afgelopen om een element te vinden. Om de constante tijd waarnaar we streven te halen is dus een relatief lege tabel noodzakelijk. Het zoeken gaat dan volgens de zelfde manier. Elementen verwijderen wordt echter moeilijk in dit systeem omdat het deel kan uitmaken van een sequentieketting. Daarom worden elementen **lui verwijderd**, ze worden als verwijderd gemarkeerd, maar niet echt verwijderd.

   Dit systeem brengt echter het probleem van **primaire clustering** met zich mee. We definiëren hiervoor eerst het begrip **laadfactor** λ. Dit is de verhouding van het aantal ingenomen posities tot de grootte van de tabel. De laadfactor varieert tussen 0 (lege tabel) en 1 (volle tabel). Om de performantie van lineaire peiling te schatten worden twee veronderstellingen gemaakt:

   i. De hashtabel is groot. Dit is eigen aan de hashtabel structuur.
   ii. Elke peiling van de hashtabel is onafhankelijk aan de vorige waarbij men dus stelt dat elke keer een positie wordt geraadpleegd de kans dat ze ingenomen is door λ wordt gegeven, onafhankelijk van de vorige peilingen.

   Deze tweede veronderstelling is uiteraard niet correct voor lineaire peiling omdat er zich groepen van ingenomen posities (**clusters**) vormen, zelfs wanneer de tabel nog dun bezet is. Voor een sleutel met een hashwaarde binnen een groep zijn meerdere peilingen nodig en deze sleutel draagt bij tot het vergroten van de groep. Dit fenomeen noemt men **primaire clustering**. We geven nu volgende stellingen zonder bewijs:

   - In de veronderstelling van onafhankelijke peilingen, is het gemiddeld aantal peilingen dat nodig is bij een toevoeging gegeven door $1/(1-\lambda)$.
   - Het gemiddelde aantal peilingen bij een succesvolle zoekbewerking wordt gegeven door $-\ln((1-\lambda)/\lambda)$.
   - Met het principe van lineaire peiling bedraagt het gemiddeld aantal peilingen bij een toevoeging of bij een niet-succesvolle zoekbewerking ongeveer $\frac{1}{2}(1+\frac{1}{(1-\lambda)^2})$ en bij een succesvolle $\frac{1}{2}(1+\frac{1}{1-\lambda})$.

   Men gebruikt best geen linaire peiling meer als de tabel voor meer dan 50% gevuld zal worden.

   (b) **kwadratische peiling**: met deze techniek word het probleem van primaire clustering opgelost. Er wordt gebruik gemaakt van de functie $f(i) = i^2$ om botsingen op te lossen. Meer concreet als de hashfunctie waarde h als uitkomst heeft dan zal men achtereenvolgens posities $h+1^2$, $h+2^2$, ..., $h+i^2$ uitproberen, ook hier word de tabel als circulair object beschouwd. Zonder bewijs stellen we dat met deze methode de grootte van de tabel best een priemgetal is om te garanderen dat een element steeds kan worden toegevoegd als laadfactor niet meer dan 0.5 is. Bovendien wordt dan geen enkele positie meerdere keren uitgeprobeerd in het zoekproces.

   Hoewel dit het probleem van primaire clustering oplost treedt er nu **secundaire clustering** op. Dit wil zeggen dat dezelfde hashwaarden dezelfde sequantie van alternatieve posities uitprobeert. Ook hier kunnen we de onafhankelijkheid van opeenvolgende peilingen dus niet veronderstellen. Experimenteel geeft het toch al betere resultaten dan lineaire peiling. Er bestaan technieken voor het

elimineren van secundaire clustering zoals **dubbel hashing** waarbij een tweede hashfunctie h_2 wordt gebruikt bij een botsing.

Er wordt dan gepeild op posities $h_1(x)$,$h_1(x) + h_2(x)$, ...,$h_1(x) + 2h_2(x)$, enzovoort. Deze tweede hashfunctie moet goed gekozen zijn en bijvoorbeeld nooit 0 teruggeven en garanderen dat alle posities kunnen worden gepeild. Meestal is een functie $h_2(x) = R - (x \bmod R)$ met R een priemgetal kleiner dan de tabelgrootte een goede functie. Theoretisch is dit heel interessant maar veelal wordt kwadratische peiling gebruikt omdat het simpeler te implementeren valt.

2. **open hashing**: bij open hashing of **hashing met gescheiden ketening** wordt bij een botsing geen alternatieve positie in de tabel zelf gezocht, maar worden alle elementen die op de zelfde positie hashen aan een lijst toegevoegd. De hashtabel houdt dan een `array` van lijsten bij en de hashfunctie bepaald in welke lijst een element moet worden toegevoegd of gezocht. Hoewel het zoeken in deze lijsten lineair is, is het achterliggende idee dat deze lijsten zo kort zijn dat dit als constant beschouwd kan worden.

   Zonder bewijs stellen we dat bij open hashing het verwachte aantal peilingen bij een succesvolle zoekbewerking gegeven wordt door $1 + \lambda/2$. Het verwachte aantal peilingen bij een niet succesvolle zoekbewerking is λ. De laadfactor speelt een minder grote rol en is zelfs niet meer tot 1 begrensd. Een grotere laadfactor leidt wel tot langere rijen en dus ene langere uitvoeringstijd. Er is wel minder rehashing nodig en dit is belangrijk als dynamische array uitbreiding niet mogelijk is.

In het boek op pagina's 345-350 staat een implementatie van gesloten hashing. Ook hier worden dynamische tabellen gebruikt om de laadfactor zo laag mogelijk te houden. Speciaal is dat de tabelgrootte niet zomaar verdubbeld wordt daar men een priemgetal als grootte wil. Daarom wordt het eerstvolgende priemgetal na de verdubbeling gebruikt. Een priemtest neemt $O(\sqrt{n})$ tijd en er moet meestal maar $O(\log n)$ tijd gespendeerd worden om het volgende priemgetal te vinden. De oude elementen mogen niet zomaar worden gekopieerd, er moet namelijk **gerehashed** worden. Dit komt omdat een nieuwe tabel een nieuwe hashfunctie met zich meebrengt. Het is mogelijk om kwadratische hashing zonder vermenigvuldiging te implementeren en ook de modulobewerking kan sterk worden geoptimaliseerd, op pagina 350 staat beschreven hoe dit kan.

Bij het ontwerp van een hashfunctie wordt naar een zo gelijkmatig mogelijke verdeling van de sleutels gestreefd. Indien de sleutels gehele getallen zijn is de modulo-operator zeer geschikt. De tabelgrootte speelt echter wel een grote rol en men gebruikt best een priemgetal. Naast de vereiste dat een hashfunctie de sleutels moet verspreiden moet hij ook zeer snel te berekenen zijn. Op pagina 353 wordt een mogelijk hashfunctie voor strings besproken.

14 Geschakelde lijsten

Een **geschakelde lijst** is een datastructuur die een implementatie levert van het ADT `List`. Een `array` is niet altijd efficiënt aangezien de data in een blok in het geheugen zit. Om een element aan het begin toe te voegen, of verwijderen, tussenvoegen, ... moeten steeds een heel pak andere elementen worden opgeschoven. Enkel het opvragen van een element gaat hierdoor zeer snel. Een geschakelde lijst stapt af van de aaneengesloten structuur waardoor alles wat sneller kan, behalve de toegang tot een element wordt iets trager. De elementen worden niet in één blok opgeslaan waardoor vergroten en verkleinen en tussenvoegen veel sneller kan.

Een **enkelvoudig geschakelde lijst** zal elke component, naast zijn eigen datavelden, ook een referentie naar de volgende component laten bijhouden. De laatste component heeft dan een `null` referentie. Het opzoeken gaat dan door vanaf de eerste te vertrekken, het toevoegen gebeurt met een hulpelement die de toe te voegen waarde krijgt en het verwiijderen gebeurt door de referentie naar het volgende element te laten wijzen (pg. 357). Het toevoegen en verwijderen kan hier in constante tijd gebeuren.

De beschreven methodes werken enkel in het algemene geval. Voor het toevoegen van een element mag de lijst niet leeg zijn en het eerste element kan op deze manier niet worden verwijderd. Het gebruik van een **ankercomponent** vermijdt dat deze speciale gevallen moeten worden opgevangen. Dit is een bijkomend element in de lijst die geen data bevat maar ervoor zorgt dat altijd geldt dat elke component een voorganger heeft.

Niet alle belangrijke basisbewerkingen worden door zo'n enkelvoudig geschakelde lijst goed opgevangen. Zo kan men moeilijk naar het laatste element gaan of de lijst in omgekeerde volgorde bijhouden. Dit wordt met een **dubbelgeschakelde lijst** opgelost. Elk element bevat nu een referentie naar het vorige en volgende en er zijn ankercomponenten vooraan en achteraan in de lijst. De algoritmes worden hierdoor ook wat aangepast omdat meer referenties moeten worden bijgewerkt (pg. 360-361).

Een populaire variant van geschakelde lijsten is een **circulaire geschakelde lijst**. Hierbij heeft de laatst component een referentie naar de eerste component. Er zijn dus geen ankercomponenten noodzakelijk maar het speciaal geval 'lege lijst' moet wel in acht worden genomen. Ook wordt een referentie naar de eerste component van een circulaire lijst bijgehouden. Men kan er voor kiezen deze dubbel geschakeld te maken of wel met een ankerelement te werken om geen speciaal 'lege lijst' geval te hebben.

Indien nodig kan men een **gesorteerde geschakelde lijst** maken waarbij een element steeds op de juiste plaats wordt toegevoegd. Details van een implementatie van een geschakelde lijst worden gegeven op pagina's 363-368.

15 Binaire zoekbomen

Een **binaire zoekboom** levert een implementatie van de ADT's `SortedSet` en `SortedMap`. Het is een binaire boom die voldoet aan de **ordeningseigenschap** dat voor elke top X uit de boom alle sleutels in de linkerdeelboom kleiner zijn dan de sleutel van X, terwijl alle sleutels in de rechterdeelboom groter zijn dan de sleutel van X. Deze eigenschap vereist dat alle sleutels verschillend zijn. Een simpele java implementatie wordt op pagina 373 gegeven.

Een zoekbewerking begint bij de wortel en kan gemakkelijk afdalen door de opgelegde ordening. Ook het zoeken van de kleinste of grootste sleutel is geen probleem. Voor het vinden van de kleinste sleutel moet steeds naar links worden afgedaald en voor de grootste steeds naar rechts. Het toevoegen kan eenvoudig weg geschieden waar het zoeken faalde en zorgt dus steeds voor het toevoegen van een blad aan de boom. Het verwijderen is echter moeilijk omdat dit aanleiding kan geven tot het loskoppelen van een gedeelte van de boom. Dit gedeelte moet dan terug worden aaneengeschakeld met behoudt van de ordeningseigenschap, ook willen we de boom niet te diep maken. We onderscheiden volgende gevallen:

- **verwijderde top is een blad**: er is geen enkele formaliteit vereist.
- **verwijderde top heeft één kind**: dat kind kan in de ouder van de verwijderde top worden aangeschakeld.
- **verwijderen top heeft twee kinderen**: een veelgebruikte strategie hier is het vervangen van de sleutel van deze top door de kleinste sleutel uit zijn rechterdeelboom en vervolgens die lege top uit de boom te verwijderen. Deze tweede verwijdering is eenvoudig omdat het gaat over een top met hoogstens

De kost van elke basisbewerking is dus proportioneel met het aantal toppen dat bezocht word gedurende de bewerking. Bij een perfect gebalanceerde boom is de toegangskost logaritmisch en bij een volledig ongebalanceerde boom is er lineaire toegangskost. Men kan bewijzen dat het gemiddelde geval 38% slechter is dan het beste geval.

Volgens het contract van een `SortedSet` moet een iterator de elementen in gesorteerde volgorde teruggeven. Voor een binaire zoekboom kan dit gemakkelijk door een **inorde-doorlopen** van de boom. Hier moeten we echter niet de hele orde in één keer teruggeven, maar enkel de opvolger voor elk element moet kunnen worden gegeven. De opvolger is het element volgend op x indien de boom inorde werd doorlopen. We onderscheiden twee gevallen:

1. Wanneer de huidige top een rechterkind heeft dan is zijn opvolger de kleinste sleutel uit zijn rechterdeelboom.

2. Wanneer de huidige top geen rechterkind heeft dan moeten we vanaf deze top in de boom naar de wortel opklimmen, tot aan de dichtstbijzijnde voorouder waar deh zoekpad van de wortel naar de huidige top naar links ging. Een mogelijke implementatie wordt op pagina 380 gegeven.

Er zijn verschillende manieren voor het implementeren van **gebalanceerde binaire zoekbomen**. Dit zijn binaire zoekbomen waaraan een bijkomende balanceringsvoorwaarde opgelegd is die eenvoudig te onderhouden moet zijn en ervoor zorgen dat de diepte van de boom $O(\log n)$ blijft. Dit maakt dat deze bomen complexer zijn en dat het toevoegen en verwijderen van een element duurder is dan bij een standaard binaire zoekboom. Er is echter wel een snellere toegangstijd en grotere beschermingsfactor tegen extreme gevallen.

Een **AVL-boom** is een binaire zoekboom met de bijkomende balanceringseigenschap dat, voor elke top uit de boom, de hoogte van de linker- en rechter-deelboom hoogstens één verschillen. De hoogte van een lege boom is per definitie -1. Dit maakt dat de balanceringseigenschap relatief makkelijk te onderhouden is en dat hij sterk genoeg is om logaritmische toegangstijd te verzekeren, m.a.w. de diepte van een AVL-boom met n toppen is $O(\log n)$. Er kan worden bewezen dat een AVL-boom van hoogte h tenminste $F_{h+3} - 1$ toppen heeft, waarbij F_i het i-de Fibonacci-getal is.

We bespreken nu de stappen nodig om een top toe te voegen. We merken vooreerst op dat enkel de balans van de toppen op het pad van de toegevoegde top naar de wortel verstoord kan zijn. Het zal dus blijken dat het herstellen van de balans voor de eerste (diepste) ongebalanceerde top op dit pad volstaat voor het herbalanceren van de hele boom. Noemen we X de top die moet worden geherbalanceerd dan zijn er 4 mogelijkheden die een imbalans kunnen veroorzaken:

1. Het toevoegen van een top aan de linkerdeelboom van het linkerkind van X

2. Het toevoegen van een top aan de rechterdeelboom van het linkerkind van X

3. Het toevoegen van een top aan de linkerdeelboom van het rechterkind van X

4. Het toevoegen van een top aan de rechterdeelboom van het rechterkind van X

De 'links-links' en 'rechts-rechts' van een imbalans gevallen zijn op te lossen door een **enkelvoudige rotatie**, de andere twee gevallen moeten door een complexere **dubbele rotatie** opgelost worden. Deze twee bewerkingen zijn fundamenteel in gelijk welke balanceringsalgoritmen. Deze twee rotaties worden met figuren op pagina's 385-388 besproken.