Inhoudsopgave

Deel II: Algoritmes en datastructuren.

Hoofdstuk 1. Inleiding 2

Hoofdstuk 2. Arrays 5

Hoofdstuk 3. Arrayvarianten: stack en queue.....13

Hoofdstuk 4. Basisalgoritmen.....19

Hoofdstuk 5. Slimme algoritmen.....31

Hoofdstuk 6. Gelinkte lijsten.....54

Hoofdstuk 7. Geordende binaire bomen.....64

Hoofdstuk 8. Sorteren.....78

Hoofdstuk 9. Hashing 94

Deel III: Technologie, historiek en economische aspecten van de informatica.

Hoofdstuk 0. Het idee 101

Hoofdstuk 1. Van analoog naar digitaal.....106

Hoofdstuk 2. De elektronische relais.....111

Hoofdstuk 3. Geheugen.....115

Hoofdstuk 4. Computerarchitectuur.....120

Hoofdstuk 5. Het productieproces.....126

Hoofdstuk 6. Computatietheorie 131

Hoofdstuk 7. Besturingssystemen.....138

Hoofdstuk 8. Internet.....144

Hoofdstuk 9. Artificiële intelligentie.....154

Met dank aan mijn voorganger en inspiratiebron Jacques Tiberghien voor hoofdstukken 2 tot en met 9 (uitgezonderd 5) van deel II.

Met dank aan mijn vrouw Anke voor het nalezen en de steun.

Met dank aan de studenten burgerlijk ingenieur voor de steeds welkome feedback.

Hoeilaart, januari 2019

Hoofdstuk 1

Inleiding datastructuren en algoritmen

De basis van computerprogramma's zijn de datastructuren en de algoritmen.

Datastructuren bepalen hoe de data wordt opgeslagen. Internetbedrijven zoals Google of Facebook beheren miljarden gegevens. Ze moeten die gegevens op een gestructureerde en efficiënte manier opslaan zodat snel de juiste gegevens kunnen opgevraagd worden.

Een probleem definieer je door expliciet te maken wat het einddoel is. Met andere woorden: wat de uitkomst van het programma moet zijn. Bijvoorbeeld: "wat is de snelste weg van de VUB naar de kathedraal van Antwerpen". Hiermee weet je natuurlijk nog niet *hoe* die te vinden. Daarvoor heb je een *algoritme* nodig. Een algoritme beschrijft op een formele wijze hoe men stap-voor-stap tot een oplossing komt. Dit algoritme kan dan uitgevoerd worden door een machine (computer).

1.1. Efficiëntie van datastructuren

Zoals we uitvoerig zullen bespreken in de cursus, bepaalt de manier waarop je gegevens opslaat hoe snel je elementen kan terugvinden, kan toevoegen of verwijderen. We geven hier de conclusies waar je naar kan teruggrijpen bij het einde van de cursus.

De volgende tabel vergelijkt de tijd om de 3 belangrijkste operaties uit te voeren: het opvragen van het i^{de} element (random access genoemd), het terugvinden van een element op naam en het toevoegen of verwijderen van een willekeurig element.

Datastructuur	**Random access (opvragen i^{de} element)**	**Find (via naam)**	**Toevoegen / verwijderen nadat element gevonden is**
Array	O(1) ++	O(n) O(log(n)) als gesorteerd	O(n) -
ArrayList	O(1) ++	O(n) O(log(n)) als gesorteerd	O(n) -
Linked list	O(n) -	O(n) --	O(1) ++ Als we zoeken niet mee tellen
Binaire boom	n.v.t.	O(log(n)) +	O(log(n)) ++
Hashtabel	n.v.t.	O(1) ++	O(1) ++ Zolang aantal elementen < grootte

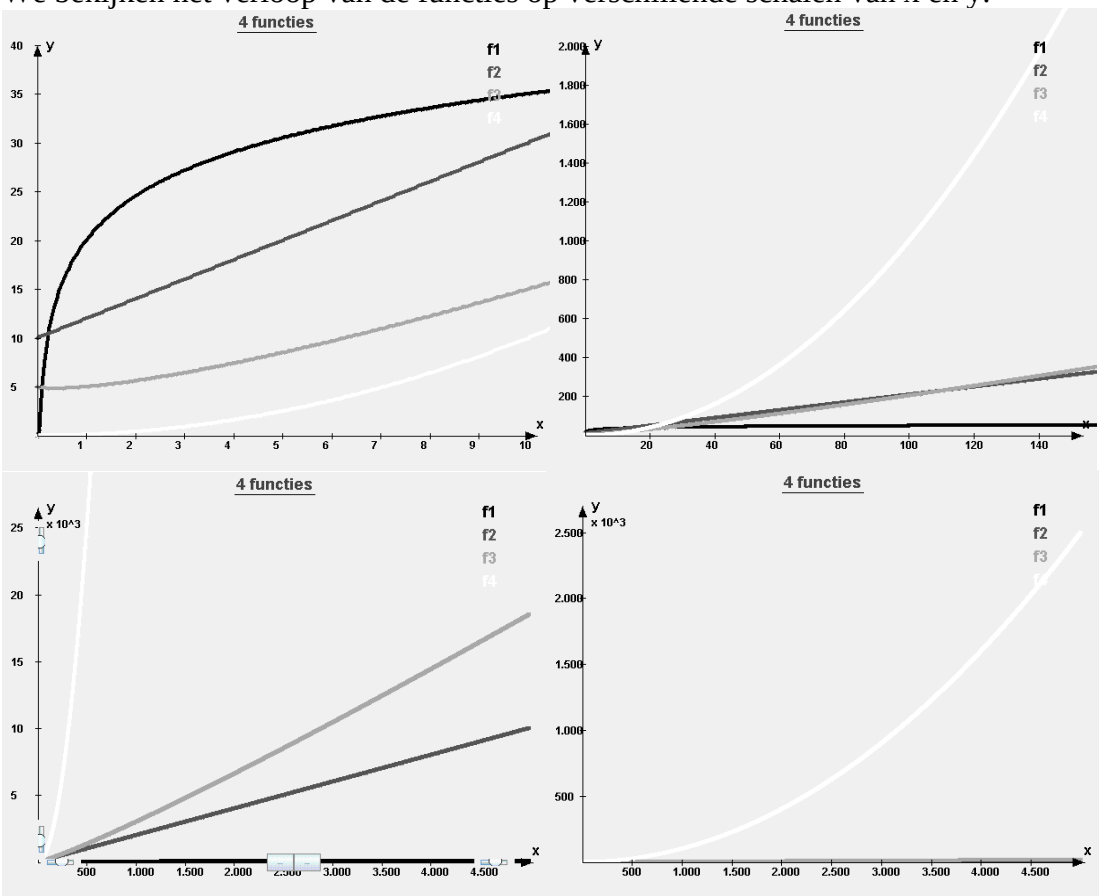
De rekentijd drukken we uit in de grootteorde in functie van *n*, met *n* het aantal elementen die de datastructuur bevat. Dit wordt de big-o notatie genoemd en wordt uitgelegd in de volgende paragraaf. We geven met + en – aan of dit efficiënt is of niet. N.v.t. betekent 'niet van toepassing'.

1.2. De big-o notatie

Rekentijden vergelijken we meestal op een tamelijk 'grove' manier. We zijn vooral geïnteresseerd in hoe de rekentijd evolueert met *n*. Voor datastructuren stelt *n* het aantal elementen voor. O(1) betekent dat de rekentijd een constante is die niet verandert voor grotere n-waarden. Dus zelfs voor een heel grote datastructuur blijft de rekentijd constant. Bij O(n) zal de rekentijd *wel* lineair stijgen. Voor kleine n-waarden kan de rekentijd kleiner zijn dan een O(1)-rekentijd, maar als *n* stijgt zal die snel de constante O(1)-tijd inhalen. En voor heel grote n-waarden zal de rekentijd enorm veel groter worden. Een O(log(n))-rekentijd zit tussen beide in. Deze zal groter worden dan O(1) maar kleiner blijven dan O(n).
Neem volgende functies

- f1(x): $y = 20 + 15*\log_{10}(x)$
- f2(x): $y = 10 + 2*x$
- f3(x): $y = 5 + x* \log_{10}(x)$
- f4(x): $y = 0.1*x^2$

We bekijken het verloop van de functies op verschillende schalen van x en y:



Op de eerste figuur lijkt het alsof f1>f2>f3>f4. Dit is echter enkel voor kleine waarden van x. Bij grotere waarden van x blijkt dat f4 het snelst stijgt, terwijl f1 het traagst, zie tweede figuur. Op de derde figuur is f4 al niet meer te zien, terwijl f1 op de x-as lijkt te liggen. F2 en f3 blijven in mekaars buurt, maar zijn duidelijk veel kleiner dan f4 (laatste figuur).

De belangrijkste conclusie is dat voor heel grote n-waarden:

$$\mathbf{O(1) < O(\log(n)) << O(n) < O(n.\log(n)) << O(n^2)}$$

De rekentijden voor de verschillende grootteordes liggen heel ver uit elkaar. Dat is de reden waarom we vooral in de afhankelijkheid van *n* geïnteresseerd zijn. Twee rekentijden met dezelfde grootteorde zullen 'kort' bij elkaar blijven, bijvoorbeeld een factor 10 verschillen, maar bij een verschillende grootteorde zal de factor blijven oplopen tot duizend, miljoen, ...
Let wel op: O(log(n)) ligt kortbij O(1) en O(n.log(n)) kortbij O(n). Als n = 1 miljard bijvoorbeeld, dan is $\log_2(n) = 30$. Het verschil tussen O(n.log(n)) en O(n) is dus veel kleiner dan tussen $O(n^2)$ en O(n)!
Voor grotere complexiteiten maakt men een verschil tussen polynomiale en exponentiële tijd. Het eerste wordt voorgesteld door een polynomiaal (bvb n^{10}), het tweede door een functie met n in het exponent (bvb 2^n) of een faculteit (n!). Voor grote waarden van n zal een polynomiaal mijlenver onder een exponentiële functie blijven:

$$\mathbf{O(poynomiaal) << O(exponentiële\ functie)}$$

n	n^{10}	2^n
20	$20^{10} \approx 10^{13}$	$2^{20} \approx 10^6$
1000	$1000^{10}=10^{30}$	$10^6.10^{50}=10^{56}$
1000000	10^{60}	$2^{1000000} \approx 10^6.10^{50000}=10^{50006}$

1.3. Catalogiseren van algoritmen

Sommige problemen zijn gemakkelijker op te lossen zijn dan andere. Algoritmen delen we in als volgt:

Type 1: De oplossing kan berekend worden met een formule (analytisch).
Type 2: Je kunt de oplossing gericht zoeken of construeren (rechttoe-rechtaan).
Type 3: Je gaat alle mogelijke actiesequenties af om een oplossing te vinden.
Type 4: Door slimme keuzes (*heuristieken*) te maken, kan je verschillende actiesequenties uitsluiten.
Type 5: Je leert al doende welke de juiste keuzes zijn.

Een voorbeeld van type 1 is het vinden van de nulpunten van een kwadratische vergelijking. Dit kan analytisch: met een formule (via discriminant) die aan de hand van de parameters de nulpunten berekent.

Een voorbeeld van type 2 is het vinden van een nulpunt van een willekeurige continue functie met Newton's algoritme (hoofdstuk 4). Op een gerichte manier gaan we op zoek naar een nulpunt en in de meeste gevallen zullen we die redelijk snel vinden. Op een gelijkaardige manier kunnen we op zoek geen naar een maximum van een functie. Merk op dat deze algoritmen typisch iteratief te werk gaan.

In hoofdstuk 5 gaan we ons richten op problemen waarvoor zulke rechttoe-rechtaan algoritmen niet bestaan (of nog niet gevonden zijn). Dit zijn de algoritmen van het type 3, 4 en 5.

Hoofdstuk 2.

Arrays

Een variabele van het *enkelvoudige type* kan één waarde aannemen. In het geheugen neemt ze één vakje in. De naam van de variabele verwijst dus naar dat éne vakje.
Vaak is het handig om verschillende waarden in het geheugen samen te kunnen zetten en met één naam aan te duiden. Dit gaan we met de *array* doen zoals hieronder getoond:

array

27	30	63	10	27	30	50	63	3	-9

Een vorm die we al kennen is het object. Een object duiden we aan met één naam. Met die naam refereren we naar een ding dat gekarakteriseerd wordt door zijn attributen. Elke attribuut mag een ander soort variabele zijn. Ze hoeven niet van hetzelfde type te zijn. Arrays daarentegen, duiden een collectie van waarden van *hetzelfde type* aan. Een array (letterlijk vertaald: rij) is een lijst van vakjes in het geheugen waarbij in elk vakje eenzelfde soort waarde zit.
Arrays vormen de basis voor de datastructuren die we in de volgende hoofdstukken gaan bespreken.

2.1. Arrays

Het *array*-type laat toe lijsten op te bouwen met elementen die allen van *hetzelfde type* zijn. De lijsten kunnen een willekeurig aantal *dimensies* hebben. De elementen worden geïdentificeerd door middel van een of meerdere *indices*. Hieronder enkele arraybewerkingen:

```
// ==== ARRAYS ====
int[] a; // declaratie van een array variabele
a = new int[10]; // aanmaken van array met grootte 10

a[0] = 10; // waarden toekennen
a[1] = 9;
a[2] = 8;
a[3] = 7;
a[4] = 6;
a[5] = 5;
a[6] = 4;
a[7] = 3;
a[8] = 2;
a[9] = 1;

// vullen van array
for (int i = 0;i < a.length;i++){
    a[i]=2*i; // index van 0 tot length-1
}
// printen van array
for (int i = 0;i < a.length;i++){
    System.out.print(a[i]+" ");
}
```

2.1.1. Aanmaak en gebruik

De zeef van Erathostenes (**package *algoritmen***) kan je implementeren met een array. De grootte van de array (variabele ***N***) vragen we aan de gebruiker. Probeer te achterhalen wat het algoritme doet. Werk het uit voor N=10 bvb.

```
1  import java.util.Scanner;
2
3  public class ZeefVanErathostenes {
4     /** PROGRAMMA */
5     public static void main(String[] args) {
6        System.out.print("Geef range: ");
7        Scanner scanner = new Scanner(System.in);
8        int N = scanner.nextInt();
9
10       // maak array aan
11       boolean[] array = new boolean[N];
12
13       // initialiseer array
14       for(int i=0;i<N;i++)
15          array[i] = true;
16
17       // de zeef
18       for(int i=2;i<N;i++){
19          if (array[i]){
20             int m = i * 2;
21             while(m < N){
22                array[m] = false;
23                m = m + i;
24             }
25          }
26       }
27       // print array
28       System.out.print("Xxx kleiner dan "+N+": ");
29       for(int i=2;i<N;i++)
30          if (array[i])
31             System.out.print(i+", ");
32       System.out.println();
33    }
34 }
```

Een array wordt gedeclareerd en aangemaakt met

```
boolean[] array = new boolean[N];
```

Het gedeelte links van het '=' - teken declareert een variabele van het type *array* opgebouwd uit elementen van het type *boolean*. In het tweede gedeelte maken we een rij van *N booleans* aan: in het geheugen worden *N* geheugenvakjes gereserveerd waarin *booleans* gestopt kunnen worden.

We maken een array dus aan met *new* (net zoals we een object aanmaken). Een array is dus ook een object, een speciaal object weliswaar. Bij de creatie ervan moeten we het aantal elementen meegeven. Hier vragen we de grootte op aan de gebruiker. Deze grootte moet dus bekend zijn bij creatie *en kan niet meer veranderd worden*. Wat dus wel kan met Python-lijsten.

Vervolgens initialiseren we de waarden van de array:

```
for(int i=0;i<N;i++)
   array[i] = true;
```

De variabele *i* wordt gebruikt om de *index* van de array aan te duiden. We lopen over de gehele array en zetten alle waarden op *true*.

Nu starten we de zeef. De verzameling wordt van klein naar groot doorlopen en voor elk getal dat nog op *true* staat in de array, worden al zijn veelvouden verwijderd. We starten met 2 en weten dat dit een priemgetal is. We gebruiken opnieuw dezelfde variabele *i* en zetten die initieel op 2. Vervolgens zetten we alle veelvouden van 2 op *false*. Dit gebeurt met de *while*-loop van lijn 23 en variabele *m*. We nemen de veelvouden kleiner dan *N*. Zo beëindigen we de eerste iteratie van de for-loop. In de volgende iteraties kijken we eerst of we een priemgetal hebben (3: ja; 4: nee;...). Indien ja, dan gaan we weer alle veelvouden van dat getal schrappen.

Vraag: waarom mogen we niet starten met de waarden 0 of 1? Wat zou er in beide gevallen gebeuren?
Nog een mooie vraag: waarom starten we met m = i * 2 en niet gewoon met m = i?
En nog één: wat is het resultaat als we de *if* van lijn 20 wegnemen?
Dit zijn voorbeelden van goede examenvragen!

Tot slot printen we priemgetallen: namelijk de waarden die nog op *true* staan. Merk op dat we weer beginnen met *i*=2! Als je een array aanmaakt, is zijn eerste index altijd 0, maar de twee eerste vakjes hebben we niet nodig. We weten dat dat geen priemgetallen zijn.

2.1.2 Index out of range

Een array heeft een vaste grootte en dus maar een beperkt aantal elementen. Een index van een array moet tussen 0 en *lengte*-1 liggen. Anders geeft Java een fout en 'gooit'(*throw* in Java) een exceptie:

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException:
20
```

Waar slaat dit op terug? '20' is de index die ik vroeg, de array was echter maar 20 groot. Dit gaf deze fout.

2.1.3. Multidimensionale arrays

Net zoals een eendimensionale array kun je *n*-dimensionale arrays aanmaken. Het resultaat is dat je *n* indices moet gebruiken. Een klein voorbeeld:

```
int[][] x = new int[3][5];
x[0][1] = 5;
```

2.1.4. Initialiseren

Bij aanmaak kan je de array ook onmiddellijk opvullen.

```
int[] array = new int[]{1, 2, 3, 4, 5};
```

Je geeft de grootte van de array dan niet mee.
Dit kan ook voor multidimensionale arrays:

```
int[][] x = new int[][]{{1, 2}, {3, 4}, {2, 3}};
```

2.1.5 Speciale array: String

Wat is een tekst? Een aaneenschakeling van letters! Oftewel, in informaticataal: een *array* van *chars*. Je kunt een tekst dus definiëren als:

```
char[] tekst = new char[100];
tekst[0] = 'j';
tekst[1] = 'a';
tekst[2] = 'v';
tekst[3] = 'a';
tekst[4] = '\0';
```

We maken een array van karakters aan en vullen die een voor een met letters. Dit is nogal omslachtig en omdat we veel met teksten werken heeft *Java* een ingebouwd type voor tekst: String. Hetzelfde doen we met:

```
String tekst = "Java";
```

Java zorgt ervoor dat je zonder al te veel poespas gemakkelijk met strings kunt werken.

Andere programmeertalen (zoals C of C++) hebben dit niet. Daar moet je wel degelijk met zo'n array werken. Daarom wil ik nog meegeven dat we in vakje 4 `'\0'`, een nul, hebben gestopt. Dit om aan te duiden dat de tekst afgelopen is. De array is 100 groot, maar we gebruiken enkel de vier eerste vakjes.

2.2. Algoritmes op arrays

Nu we weten wat een array is, kunnen we enkele belangrijke algoritmes bespreken. De code is te vinden in **package *Algoritmen***.

```
1  import java.util.Arrays;
2  import java.util.Random;
3
4  public class ArrayAlgoritmen {
5     /** PROGRAMMA */
6     public static void main(String[] args) {
7        int lengte = 20, maxWaarde = 20;
8        int[] array = randomArray(lengte, maxWaarde);
9            System.out.println("Array: "+Arrays.toString(array));
10       System.out.println("Minimum    : "+minWaarde(array));
11       System.out.println("Totaal     : "+totaal(array));
12    }
13
14    public static int[] randomArray(int length, int maxWaarde){
15       int[] array = new int[length];
16       Random r = new Random();
17       for(int i=0;i<length; i++)
18          array[i] = r.nextInt(maxWaarde);
19       return array;
20    }
21    public static int minWaarde(int[] array){
22       int min = array[0];
23       for(int i=1;i<array.length; i++) // ik begin vanaf 1
24          if (array[i] < min)
25             min = array[i];
26       return min;
27    }
28    public static int totaal(int[] array){
29       int tot = 0;
30       for(int i=0;i<array.length; i++)
31          tot += array[i];
32       return tot;
33    }
```

Eerst vullen we een array van grootte 'lengte' met willekeurig gekozen getallen in `[0, maxwaarde[`. '*Maxwaarde*' zelf kan dus niet worden gekozen. De *Random* klasse laat ons toe willekeurige getallen op te vragen. Op lijn 16 maken we een *Random*-object aan en gebruiken dit op lijn 18. Hier zijn het dus integers, maar je kunt willekeurige waarden van elk type laten kiezen (zie Java documentatie). Voor het printen van de array gebruik ik de *toString*-methode van de *Arrays* klasse. Deze laatste is ook weer een klasse met enkele handige functies. Zo dadelijk gaan we hier nog meer gebruik van maken.
Vervolgens zoeken we het kleinste getal en berekenen we het totaal. Dit mag geen probleem vormen. Voor het minimum starten we met het eerste getal en lopen vervolgens de array af op zoek naar een kleiner getal. Aangezien het in methode *minWaarde* onnodig is om in de *for*-loop het eerste getal opnieuw te checken (weet je waarom?), beginnen we met $i = 1$.
Dit is een beetje ongewoon (meestal begin je vanaf 0) en daarom zet ik er commentaar bij. Dat is een goede gewoonte: *documenteer de niet-triviale code*. Meer hoef je van mij niet te documenteren; als de code voor zich spreekt, hoeft er geen documentatie bij.

Om het totaal te berekenen voegen we elk element toe aan de variabele *tot*. We kunnen ook de tweede vorm van de *for* gebruiken. Dit geeft deze code:

```
int tot = 0;
for(int v: array)
   tot += v;
```

Deze *for* kunnen we echter niet gebruiken om elementen van de array aan te passen (zoals in *RandomArray*) of als we niet alle elementen willen aflopen (zoals in *minWaarde*).

We gaan verder met de functies *gemiddelde*, *aantal* en *isGesorteerd*:

```
1  import java.util.Arrays;
2  import java.util.Random;
3
4  public class ArrayAlgoritmen {
5    /** PROGRAMMA */
6    public static void main(String[] args) {
7      int lengte = 20, maxWaarde = 20;
8      int[] array = randomArray(lengte, maxWaarde);
9      System.out.println("Array      : "+Arrays.toString(array));
10     System.out.println("Minimum    : "+minWaarde(array));
11     System.out.println("Totaal     : "+totaal(array));
12
13     System.out.println("Gemiddelde:"+gemiddelde(array)
14 +"   (ik verwacht hier "+((float)maxWaarde/2)+")");
15
16     System.out.println("Aantal 3's : "+aantal(array, 3)
17 +"   (ik verwacht hier "+((float)lengte/maxWaarde)+")");
18
19     System.out.println("Gesorteerd?  "+isGesorteerd(array));
20     Arrays.sort(array);
21     System.out.println("Gesorteerd?  "+isGesorteerd(array));
22   }
23   public static float gemiddelde(int[] array){
24     int tot = totaal(array);
25     return (float) tot / array.length;
26   }
```

```
27    public static int aantal(int[] array, int waarde){
28       int n=0;
29       for(int i=0;i<array.length; i++)
30          if (array[i] == waarde)
31             n++;
32       return n;
33    }
34    public static boolean isGesorteerd(int[] array){
35       for(int i=1;i<array.length; i++) // ik begin vanaf 1
36          if (array[i] < array[i-1])
37             return false;
38       return true;
39    }
```

Eerst berekenen we het gemiddelde van de array. Omdat het gemiddelde gelijk is aan het totaal gedeeld door het aantal, hergebruiken we de totaalfunctie. Bij het oproepen van de functie bereken ik ook de verwachte waarde voor het gemiddelde (lijn 17) wetende dat de getallen tussen 0 en *maxWaarde* liggen. Omdat we hier een kommagetal kunnen krijgen, zetten we de waarde om naar *float*.
Om te tellen hoeveel keer een waarde in de array voorkomt, gebruiken we een teller die we telkens met 1 ophogen als de waarde gevonden is. We testen de functie voor de waarde '3'. Het verwachte aantal is het aantal elementen gedeeld door het aantal mogelijke waarden (lijn 21).
Vervolgens checken we of de array gesorteerd is. Deze functie zal een boolean teruggeven: waar of fout?
We vergelijken elk element met het vorige element. Hiervoor moeten we wel degelijk vanaf 1 beginnen! Als het element kleiner is dan het vorige, kunnen we direct stoppen en foutief teruggeven. Dit doet de return. De for-lus wordt dus niet beëindigd.
Sorteren bespreken we in een apart hoofdstuk. Hier maken we ons er gemakkelijk vanaf; we sorteren de array met een functie van de *Arrays* klasse.

Tot slot gaan we op zoek naar een waarde in de array, bijvoorbeeld '3'. Hiervoor kun je een sneller algoritme bedenken als je weet dat de array gesorteerd is:

```
1   import java.util.Arrays;
2   import java.util.Random;
3
4   public class ArrayAlgoritmen {
5     /** PROGRAMMA */
6     public static void main(String[] args) {
7        … ZIE HIERBOVEN  …
8
9        System.out.println("Index van 3: "+zoek(array, 3)+"   (aantal
10  zoekiteraties = "+aantalIteraties+", ik verwachtte "+lengte/2+")");
11       System.out.println("Index van 3: "+zoekAlsGesorteerd(array,
12  3)+"   (aantal zoekiteraties = "+aantalIteraties+", ik verwachtte
13  "+Math.log(lengte)/Math.log(2)+")");
14    }
15    public static int aantalIteraties = 0;
16  /** geeft index van eerste voorkomen van waarde. -1 als niet */
17    public static int zoek(int[] array, int waarde){
18       aantalIteraties = 0;
19       for(int i=0;i<array.length; i++){
20          aantalIteraties++;
21          if (array[i] == waarde)
22             return i;
23       }
24       return -1;
```

```
25    }
26    /** geeft index van eerste voorkomen van waarde. -1 als niet */
27    public static int zoekAlsGesorteerd(int[] array, int waarde){
28       // eerst check ik of wel degelijk gesorteerd
29       if (!isGesorteerd(array))
30          return zoek(array, waarde);
31       aantalIteraties = 0;
32       int min = 0, max = array.length;
33       do{
34          aantalIteraties++;
35          int midden = (max + min) / 2;
36          if (waarde == array[midden])
37             return midden; // gevonden!
38          if (waarde < array[midden])
39             max = midden;
40          else // waarde ligt voorbij midden
41             min = midden;
42       } while (min < max);
43       return -1;
44    }
```

Het eerste algoritme loopt de array af van voor naar achter en geeft de index terug als de waarde werd gevonden. Als de *for*-loop afgelopen is zonder dat de waarde tegen is gekomen, geven we -1 terug om dit aan te duiden. Dat is een typische manier om aan te duiden dat iets niet werd gevonden. Een index kan nooit negatief zijn, dus geven we het eerste negatieve getal terug.
Daarnaast wil ik ook bewijzen dat mijn tweede algoritme sneller is. Daarom tel ik het aantal iteraties die mijn functie uitvoert. Dit stop ik in een teller die als een *statische variabele* gedefinieerd is. Een functie kan maar één waarde teruggeven, dus ik kan de teller niet teruggeven. Om toch nog een andere waarde te berekenen, doe ik dat met een statische variabele. Die is dan leesbaar in de *main*. Dit is niet ideaal, maar ook niet zo slecht: de gezochte index is waar het echt om gaat. De teller is een nevenresultaat. Die hoef je niet altijd te weten.

Nu gaan we het geoptimaliseerde algoritme bestuderen. Als je weet dat de array gesorteerd is, waar kijk je dan als eerste naar? Het midden! Dan kijk je of je te ver bent (het gezochte getal is kleiner dan het element in het midden van de array) of dat je nog niet ver genoeg bent (het gezochte getal is groter). In het eerste geval zal het getal tussen het begin en het midden liggen, in het tweede geval tussen het midden en het einde. Zo vernauw je het zoeken, want je hakt het te zoeken gedeelte steeds in twee. Denk aan het zoeken in een telefoonboek. Wat is het verwacht aantal stappen dat je nodig hebt? Log_2 van de arraygrootte *n*.
Afleiding logaritmische wet. We hakken *n* steeds in 2:

n => n/2 => n/4 => ... => 1
Dus: 1*2*2*2*...*2 ≈ n ≈ 2^x (niet exact door afrondingen)
=> aantal iteraties = x = log2 n

Vraag**:** wat gebeurt er als het te zoeken getal het eerste element is? Hoeveel stappen hebben we nodig? Minder dan gemiddeld?

Oefening**:** maak een programma dat het aantal priemgetallen telt na de zeef van Erathostenes.

2.3. De ArrayList

De array toont aan hoe je met computergeheugen omgaat: als je data wilt opslaan, vraag je een geheugengedeelte aan het besturingssysteem (operating system). Dit gebeurt met `new int[10000];`. Het systeem reserveert 10000 vakjes in het geheugen. Die mag je gebruiken voor je array. Met je index loop je naar hartenlust door het geheugen. Maar je mag niet buiten het gereserveerde gedeelte gaan. Een index groter dan de arraygrootte wijst eveneens op een geheugenvakje, maar buiten het gereserveerde gedeelte. Java checkt dit en geeft dit aan als fout. Dat vakje kan immers voor een andere variabele gereserveerd zijn. Merk op dat dit niet in alle programmeertalen gebeurt, in C++ bijvoorbeeld, gebeurt dit niet. Het vergt immers extra rekentijd om dit telkens te checken.

Het is een beperking dat deze gereserveerde grootte vast is; je kunt de arraygrootte niet zomaar aanpassen. Daarvoor zou je een nieuwe array moeten aanmaken en de gegevens van de oorspronkelijke array weer daarin kopiëren. Java biedt een object dat dit allemaal voor jou doet: de *ArrayList*. Bij het aanmaken van een *ArrayList*, wordt een initiële capaciteit gealloceerd. Deze kan dan door de gebruiker worden opgevuld. Zodra hij meer elementen toevoegt dan de capaciteit toelaat, wordt een nieuwe, grotere array gereserveerd en de elementen worden gekopieerd.

Een ArrayList is dikwijls gemakkelijker om mee te werken. De algoritmen voor arrays kunnen we vertalen voor ArrayLists.

Merk op dat deze discussie over geheugenreservatie (ook allocatie genoemd) nuttig is voor de ingenieur. Het is immers goed om te weten wat er op systeemniveau gebeurt. Als ingenieur moet je soms heel efficiënte dingen maken voor speciale apparaten. Dan zul je tot op het systeemniveau af moeten dalen. Voor veel problemen hoéf je dit echter niet te weten; je gebruikt gewoon de *ArrayList*.
Sowieso is het doel van een ingenieursopleiding om inzicht te geven in wat er ‘onder de motorkap’ gebeurt. En dit voor alle disciplines. Met een auto kunnen rijden is niet genoeg, je wilt weten hoe hij werkt.
ArrayList is een voorbeeld van ***encapsulatie***, één van de pijlers van object-georiënteerd programmeren. Je krijgt een object ter beschikking die je gebruikt via zijn methodes, maar je krijgt de interne keuken niet te zien. Die zit lekker verstopt *in* het object. Zoals je al kunt raden, zit in het object een *array* die al je gegevens bijhoudt. Maar je krijgt er geen toegang toe. Deze array is gedefinieerd als *private* (of *protected*), wat dus wil zeggen dat de buitenwereld het attribuut niet te zien krijgt. De methodes in de documentatie zijn wel als *public* gedefinieerd, de gebruiker kan ze wel gebruiken.
Encapsulatie is ook nuttig om de **consistentie** van een datastructuur te behouden. Naast een array houdt een *ArrayList* ook bij hoeveel elementen van de array effectief gebruikt zijn. Dit attribuut (een integer) is private zodat de gebruiker hem niet zelf kan aanpassen. Enkel de *ArrayList* kan dat. Anders zou de gebruiker hem bijvoorbeeld groter kunnen maken dan de grootte van de gereserveerde array, wat een fout zou opleveren als je een te groot element opvraagt.

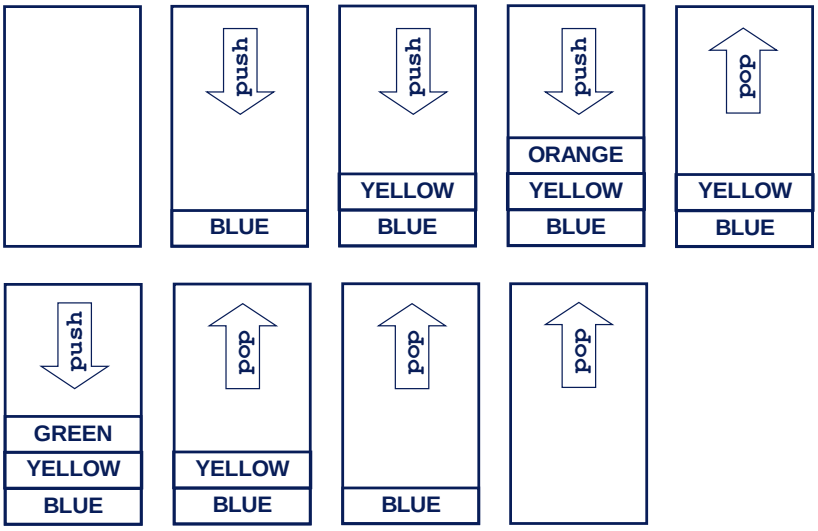
Hoofdstuk 3

Arrayvarianten: stack en queue

In dit hoofdstuk zien we enkele nuttige varianten van de array. Van elke variant maken we een aparte klasse, net als de *ArrayList*. Maar achter de schermen werken we nog steeds met arrays. Omdat we de elementen op een speciale manier willen gebruiken, zullen we er een apart type van maken.

3.1. Een stapel: 'stack'

Neem je bureau met zijn stapels paperassen als voorbeeld. Dat zijn de dingen die je nog moet doen. Regelmatig groeit deze stapel aan. Af en toe wordt hij kleiner, omdat je eraan toe komt om dingen af te werken. We voeren dus twee operaties uit: we leggen iets op de stapel en we nemen er iets af. Het eerste noemen we *push* en het tweede *pop*.
Dit wordt getoond in de volgende figuur waar we een stapel kleuren bijhouden:



Dit wordt ook een *last-in, first-out* (lifo) structuur genoemd. Onder het motto "de laatste zal de eerste zijn".
Nu bouwen we zo'n datastructuur, oftewel een klasse *Stack*. Deze klasse moet dus al zeker de methodes *push()* en *pop()* hebben. Daarnaast is het ook handig om te weten of de *stack* zich in één van de twee extremen bevindt, of hij dus leeg of vol is.

Zo krijgen we de volgende code:

```
import java.nio.BufferOverflowException;
import java.nio.BufferUnderflowException;

public class Stack {
	int[] data;
	int pointer=0; // wijst naar het eerste vrije element

	public Stack(int capacity){
		data = new int[capacity];
	}

	public void push(int element){
		// check of vol
		if (pointer == data.length)
			throw new BufferOverflowException();
		data[pointer] = element;
		pointer++;
	}

	public int pop(){
		// check of leeg
		if (pointer == 0)
			throw new BufferUnderflowException();
		pointer--;
		return data[pointer];
	}
	public boolean isEmpty(){
		return pointer == 0;
	}
	public boolean isFull(){
		return pointer == data.length;
	}
}
```

Intern baseren we ons op een *array* en we vragen de gebruiker de capaciteit van de stack te definiëren. Daarnaast houden we een *pointer* bij die verwijst naar het eerste vrije element. De push en de pop gebruiken die pointer om te schrijven en te lezen naar de array. We moeten wel testen of de array niet vol of leeg is. Dan geven we een exceptie. Hiervoor gebruiken we twee excepties die Java al gemaakt heeft: `BufferOverflowException` en `BufferUnderflowException`. Deze worden gebruikt in soortgelijke problemen.

Opmerking: je kunt de *pointer* ook laten verwijzen naar het bovenste element van de stack, waarbij je met -1 aanduidt dat hij leeg is. Dat is mogelijk en is geen enkel probleem; het is de keuze van de programmeur. Hierdoor wordt de code wel een beetje complexer. Probeer het maar eens.

Je kunt je natuurlijk afvragen waarom we niet met een standaardarray werken als we iets willen doen met een *lifo*-stapel. Zo moeilijk is dat ook niet. Waarom al dat gedoe met de speciale klasse? De reden: het is simpeler en duidelijker om *pop* en *push* op te roepen dan om met pointers te werken. Je laat ook zien wat de bedoeling van de datastructuur is, namelijk een *lifo*-stapel. Bovendien is er zo minder kans op fouten. Je kan de stapel op zich testen. Eenmaal dat je zeker bent dat de code bugvrij is, kun je ze in je programma gebruiken.

Deze datastructuur is nog een mooi voorbeeld van *encapsulatie*. Als gebruiker werk je met de publieke methodes (*public*), maar heb je geen toegang tot de attributen *data* en *pointer*. We zouden beide attributen als *private* kunnen definiëren, maar door ze niet *public* te maken, geven we ook al aan dat ze niet voor de gebruiker zijn (ze zijn dan enkel zichtbaar binnen de *package*).

Zou de gebruiker wèl toegang hebben tot de attributen dan zou de gebruiker de consistentie van het object om zeep kunnen helpen (bewust of per ongeluk). De gebruiker zou data kunnen toevoegen of verwijderen zonder de pointer op de juiste manier te verzetten. Encapsulatie voorkomt dit, het zorgt ervoor dat het object consistent gehouden kan worden, *wat de gebruiker ook doet*. Hij kan *push*'en en *pop*'en zoveel hij wil, het object blijft steeds consistent. Dat is ook een geruststelling voor de gebruiker, je moet de interne keuken van het object niet begrijpen om het te kunnen gebruiken.

Hier een voorbeeld van code met een array die in feite als stack gebruikt wordt (links) en hoe dit herschreven kan worden met een stack (rechts). Het is duidelijk dat het gebruik van een stack object leidt tot elegantere code die duidelijker is en minder tot fouten zal leiden.

```
...
int[] data = new int[capacity];
int pointer=0;
...

if (pointer == data.length)
    throw new BufferOverflowException();
data[pointer] = element;
pointer++;

...
if (pointer == 0)
    throw new BufferUnderflowException();
pointer--;
element = data[pointer];
...
```

```
Stack stack = new Stack(3);
```

```
stack.push(element);
```

```
element = stack.pop();
```

Tot slot maken we een *fifo*-datastructuur.

3.2. Een wachtrij: 'queue'

Een wachtrij is wat er gebeurt aan een kassa: je sluit achteraan en wacht geduldig op je beurt. Eenmaal voorin de rij gekomen, ben je aan de beurt. *First-in, first out*. Dit wordt dan ook een *fifo-queue* genoemd. Deze worden veel gebruikt in netwerken zoals het internet. Bij het versturen van een mail of consultatie van een website, worden er berichtjes rondgestuurd tussen jou computer en de mail- of webserver. Deze berichtjes zijn geschreven volgens het Internet-Protocol (IP) en heten ip-pakketjes. Ze zwerven over het internet en springen van knooppunt naar knooppunt. In een knooppunt moeten ze de goede richting opgestuurd worden ('routen' genoemd), dit gebeurt door een router. De router handelt de ingekomen berichtjes één voor één af en stopt ze hiervoor in een *fifo-queue*. Ieder pakketje moet geduldig zijn beurt afwachten.

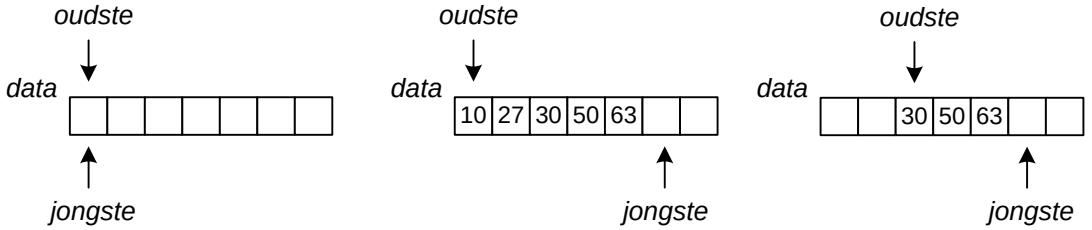
Hier een implementatie:

```
public class FIFOQueue<T>{
	T[] data;
	int oudste=0; // pointer naar oudste element (als niet leeg)
	int jongste=0; // pointer naar de plaats waar het volgend
element moet komen
	boolean full = false; // als queue vol is
	public FIFOQueue(int capacity){
		data = (T[]) new Object[capacity];
	}
	public void add(T waarde){
		if (full)
			throw new BufferOverflowException();
		data[jongste] = waarde;
		jongste++;
		if (jongste == data.length)
			jongste = 0;
		if (jongste == oudste)
			full = true;
	}
	public T get(){
		T waarde = data[oudste];
		data[oudste] = null;
		oudste++;
		if (oudste == data.length)
			oudste = 0;
		if (full)
			full = false;
		return waarde;
	}
	public boolean isEmpty(){
		return !full && jongste == oudste;
	}
	public boolean isFull(){
		return full;
	}
	public int size(){
		if (full)
			return data.length;
		else if (jongste >= oudste)
			return jongste - oudste;
		else
			return data.length - (oudste - jongste);
	}
}
```

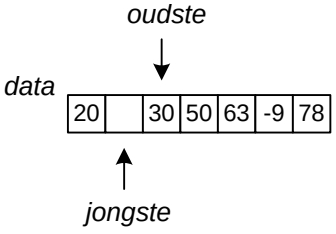
We hebben de klasse *generiek* gemaakt: je kunt ze gebruiken voor eender welk soort objecten. Dit bepaal je bij het aanmaken van een *FIFOQueue* object. Dan geef je het gevraagde type mee, zoals je al deed bij de *ArrayList* en andere klassen. Hoe je dit implementeert, zie je hier. Met de klassenaam geven we <**T**> mee om aan te duiden dat we een *geparameteriseerd type* gaan gebruiken. Tijdens het definiëren weten we het juiste type van **T** nog niet. We werken binnen de klasse met de abstracte en algemene **T**. Bij het aanmaken van een object zal de gebruiker het type meegeven en vervangt Java in de code **T** door het opgegeven type. Je kunt het type dus zien als een 'parameter'.

We slaan de gegevens weer op in een array die een zekere capaciteit heeft. Als er elementen worden toegevoegd, *add()*, slaan we die op aan de achterkant in de array. Als we elementen lezen, *get()*, nemen we die aan de voorkant uit de array. De array wordt dus geleidelijk weer leeg aan de voorkant. Als we een element lezen zouden we natuurlijk alle elementen naar links kunnen opschuiven. Net zoals we bij een kassa naar voren schuifelen. Maar dit is niet de meest efficiënte oplossing. Bij een grote *queue* zal het veel tijd kosten om alle elementen één voor één op te schuiven. Het is dus een betere oplossing om de vakjes aan de voorkant leeg te maken. Hiervoor moeten we twee pointers voorzien, eentje voor de achterkant van de *queue* en eentje voor de voorkant. Deze heb ik respectievelijk *oudste* en *jongste* genoemd.

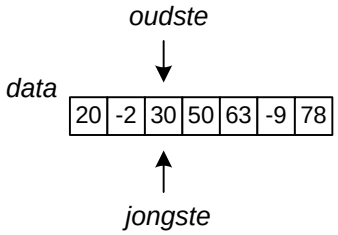
Hier zie je de beginsituatie, de situatie na het toevoegen van 5 elementen en na het lezen van de 2 oudste elementen.



Jongste en oudste en schuiven op naar rechts bij respectievelijk het aanvullen en lezen. Nu zijn er echter twee moeilijkheden die we moeten oplossen. Ten eerste: als we rechts aan het einde van de array komen, gaan we de lege vakjes van de voorkant recycleren. Hiervoor moet onze pointer *jongste* weer naar 0 gaan. Ook *oudste* moet op het einde weer naar 0 gaan.



Een tweede probleem is dat we aan de waarde van *jongste* en *oudste* niet kunnen zien of de array vol of leeg is. In beide gevallen is *jongste gelijk* aan *oudste*. De enige manier om dit bij te houden is in een extra variabele: *full*.



Studente Evelyne Ringoot (2016) heeft me erop gewezen dat het wel degelijk met 2 variabelen op te lossen valt. De 'start' (=oudste) en het 'aantal' elementen bijhouden. De jongste vind je terug met formule '(start+aantal) modulo capaciteit'. 'Aantal' geeft ook aan wanneer de queue vol zit, dan is ze immers gelijk aan de capaciteit. Onze oplossing heeft dan weer als voordeel dat je onmiddellijk het jongste element terugvindt.

Bestudeer de code en merk op hoe we de waarde van *full* up-to-date houden en hoe we ze gebruiken. Bestudeer vooral `isEmpty()`. Ook in *size()* moeten we er gebruik van maken én er rekening mee houden dat *oudste* groter kan zijn dan *jongste*.

Dit zijn weer enkele van de typische hindernissen waar een programmeur mee wordt geconfronteerd. Dit betekent: even zoeken, testen en debuggen.

Dit soort queues zijn ook handig voor het bijhouden van de *n* laatste items. Neem bijvoorbeeld de oproepgeschiedenis van je gsm. Door het beperkte geheugen van je gsm kan slechts een beperkt aantal oproepen worden bijgehouden. Om de gebruiker niet op te zadelen met het opkuisen van de geschiedenis, houdt hij enkel de meest recente bij en veegt de oudste weg. Dit kan gebeuren met een fifo-queue.

3.3. De wachtrij met prioriteit: 'priority queue'

Een veelgebruikte variant is een wachtrij waarbij niet het oudste element wordt teruggegeven, maar het element met de hoogste prioriteit.
Java heeft hiervoor een klasse `PriorityQueue<E>` in zijn bibliotheek. De `poll()`-methode neemt het element met de hoogste prioriteit. De prioriteit van de elementen is gebaseerd op een orderelatie die gedefinieerd is op de elementen. Zoals we gaan zien in 7.2 en 8.5 kan dit in Java via de natuurlijke ordening of via een specifieke ordening.
In 5.2.3 zullen we deze datastructuur gebruiken.

Hoofdstuk 4

Basisalgoritmen

Ingenieurs gebruiken de computer vaak om berekeningen te doen. Hiervoor bestaan gespecialiseerde programma's zoals Matlab, maar dikwijls moeten er ook nieuwe programma's geschreven worden om specifieke numerieke problemen op te lossen. In dit hoofdstuk bespreken we enkele basisalgoritmes en zien we hoe *abstractie* gebruikt kan worden om functies mee te geven als parameter (als functie-objecten). De code van dit hoofdstuk is te vinden in **package *algoritmen***.

4.1. Het zoeken van een nulpunt via 'recursive halving'

Het probleem dat we hier aanpakken, is het vinden van een numerieke oplossing voor een vergelijking van de vorm *f(x) = 0*: vind een waarde van 0 voor welke de functie 0 is. Een nulpunt wordt ook een *wortel* genoemd. Voor een kwadratische vergelijking kunnen we het nulpunt via een formule berekenen aan de hand van de discriminant. Zo'n *analytische oplossing* is niet altijd mogelijk. Neem de volgende complexe functie:

$$F(r)=0.840376\frac{2r+1}{(r+1)^2}-0.221538\frac{8r^2-3r-1}{(r+1)^3}-0.0173983\frac{54r^3-145r^2+12r+3}{(r+1)^4}+0.0214648\times$$

$$\frac{32r^4-203r^3+161r^2-5r-1}{(r+1)^5}+0.0026529\frac{5+30r-2010r^2+5376r^3-2835r^4+250r^5}{(r+1)^6}$$

Hiervoor bestaat er geen analytische oplossing om de wortels te berekenen met behulp van een formule die gebaseerd is op de parameters van de functie.
Een alternatieve methode is het *numeriek* zoeken met een algoritme: **we berekenen *f* voor verschillende waarden van *x* en gaan op zoek naar een nulpunt**. Doordat we de vorm van *f* niet kennen, kan dit een hele opgave zijn.
We zouden alle mogelijke waarden van *x* kunnen aflopen, maar we kunnen moeilijk alle reële getallen afgaan. We kunnen een eindig aantal waarden van *x* testen, bijvoorbeeld alle veelvouden van een gekozen interval. Zo zouden we kunnen proberen uit te vissen of een nulpunt in een interval ligt.
Stel dat het linkerpunt van het interval een negatieve functiewaarde geeft en het rechterpunt een positieve functiewaarde. Dan zijn we zeker dat de functie de *Y*-as kruist. Ten minste, als we te maken hebben met een continue functie. Daar gaan we hier van uit. Maar zelfs als de twee eindpunten van een interval beide een positieve of een negatieve functiewaarde geven, kan de functie toch even een ommetje maken langs de *Y*-as. Bij het numeriek vinden van nulpunten, moeten we dus steeds uitgaan van enkele assumpties. Anders weten we nooit zeker of er nulpunten aanwezig zijn. Zelfs tussen twee x-waarden die dicht bij elkaar liggen en een zeer grote functiewaarde geven, kan een nulpunt liggen.

Het eerste algoritme dat we hier bespreken start van deze premisse: twee punten waarbij de eerste een negatieve functiewaarde geeft en de tweede een positieve. De functiewaarde wordt berekend met de functie *f*. Om verschillende functies te kunnen testen, heb ik er twee in commentaar gezet.

Het algoritme deelt het interval telkens in twee en bepaalt in welke helft het nulpunt ligt. Zo kan de kruising van de *Y*-as op een snelle manier gevonden worden. Hoe kort we de functiewaarde bij de nul willen, bepaalt het aantal iteraties die nodig zijn om deze te vinden.
Hier staat dit ingesteld op '0.01'. Aan het begin van de functie testen we de conditie van de functiewaarden. Als niet wordt voldaan, geven we een exceptie. Hiervoor wordt de `IllegalArgumentException` gebruikt. Dit om aan te geven dat de argumenten (=parameterwaarden) foutief zijn:

```
public class IetsMetFunctie {

	/** PROGRAMMA  */
	public static void main(String[] args) {
		double a = -10, b = 10;

		double xx = vindXX(a, b);
		System.out.println("XX voor deze functie is "+xx);
	}
	public static double f(double x){
//		return 2 *(x - 3);
		return x * x  + 12 *(x - 3);
//		return - x * x * x / 8 + x * x  + 20 *(x - 3);
	}
	public static double vindXX(double a, double b){
		final double PRECISIE = 0.001; // configuratieparameter

		if ( !(f(a) < 0 && f(b) > 0))
			throw new IllegalArgumentException("Geef betere
beginpunten!");

		double m = (a + b) / 2;

		while( Math.abs( f(m)) > PRECISIE){
			System.out.print(m+", ");
			if (f(m) < 0)
				a = m;
			else
				b = m;
			m = (a + b) / 2;
		}
		System.out.println(m);
		return m;
	}
}
```

4.2. Functies als parameters

Een nadeel van de vorige oplossing is het definiëren van de functie in de statische methode *f*. Willen we een andere functie testen, dan moet we deze methode aanpassen. Dit deed ik door enkele functies in commentaar te zetten. Dit is voor een goede code echter onwerkbaar. "Old school" zouden we werken met een parameter als volgt:

```
public class IetsMetFunctie_oldschool {

	/** PROGRAMMA  */
	public static void main(String[] args) {
		double a = -10, b = 10;

		double nulpunt = vindNulpunt(a, b, 1);
		System.out.println("XX voor deze functie is "+nulpunt);
	}
	public static double f(double x, int functie){
		switch (functie) {
		case(1): return 2 *(x - 3);
		case(2): return x * x  + 12 *(x - 3);
		case(3): return - x * x * x / 8 + x * x  + 20 *(x - 3);
		default: throw new IllegalArgumentException("Functie
"+functie+" is geen geldige parameter.");
		}
	}
	public static double vindNulpunt(double a, double b, int functie){
		final double PRECISIE = 0.000001; // configuratieparameter

		if ( !(f(a, functie) < 0 && f(b, functie) > 0))
		   throw new IllegalArgumentException("Betere beginpunten!");

		double m = (a + b) / 2;
		int ctr=0;
		while( Math.abs( f(m, functie)) > PRECISIE){
			System.out.print(m+", ");
			if (f(m, functie) < 0)
				a = m;
			else
				b = m;
			m = (a + b) / 2;
			ctr++;
		}
		System.out.println(m);
		System.out.println("Aantal iteraties  = "+ctr);
		return m;
	}
}
```

Nu kunnen we wel meerdere functies testen, maar moet de originele code nog steeds aangepast worden (hier: de switch) als we nieuwe functies willen testen. We moeten ***de functie zelf*** als parameter kunnen meegeven aan het algoritme. Het algoritme werkt immers voor elke continue functie, het staat los van de functie. Als je de code nu leest, lijkt het alsof het algoritme speciaal geschreven is voor deze specifieke functie gedefinieerd met *f*.

De oplossing is het definiëren van het concept ‘functie’ en dit te gebruiken om functies aan te maken als objecten. Object-georiënteerd programmeren komt er steeds op neer om van alle relevante dingen objecten te maken!
Hiervoor definiëren we eerst wat een functie is. Wat is de essentie van een functie? Dat het een gegeven waarde omzet in een nieuwe waarde. Programmatorisch gebeurt dit met een methode. Met een functie ‘bepaal’ je deze methode. Programmatorisch: je ‘implementeert’ de methode.

Het concept functie wordt gedefinieerd met een *interface*:

```
public interface Functie {
    /** geeft functiewaarde in opgegeven punt */
    double f(double x);
}
```

Voor het abstracte begrip 'functie' bestaat er geen concrete invulling van *f*. Deze methode is dus abstract, zonder implementatie. Omdat de klasse niets concreets heeft, kan het gedefinieerd worden als een interface, wat een compleet abstracte klasse is. Het zou hier niet fout zijn om het als abstracte klasse te definiëren. Dit zou het volgende geven:

```
public abstract class Functie {
    /** geeft functiewaarde in opgegeven punt */
    public abstract double f(double x);
}
```

Merk op dat we de methode nu als expliciet publiek en abstract moeten definiëren. Voor een *interface* hoeft dat niet, want per definitie zijn al zijn methodes abstract en publiek. Deze abstracte klasse zou ook volstaan in wat volgt, maar omdat er in de klasse niets concreets is, maken we hem een interface. Een voordeel van een interface is dat een klasse van meerdere interfaces mag erven, maar slechts van één abstracte klasse. Interfaces zijn dus iets handiger.

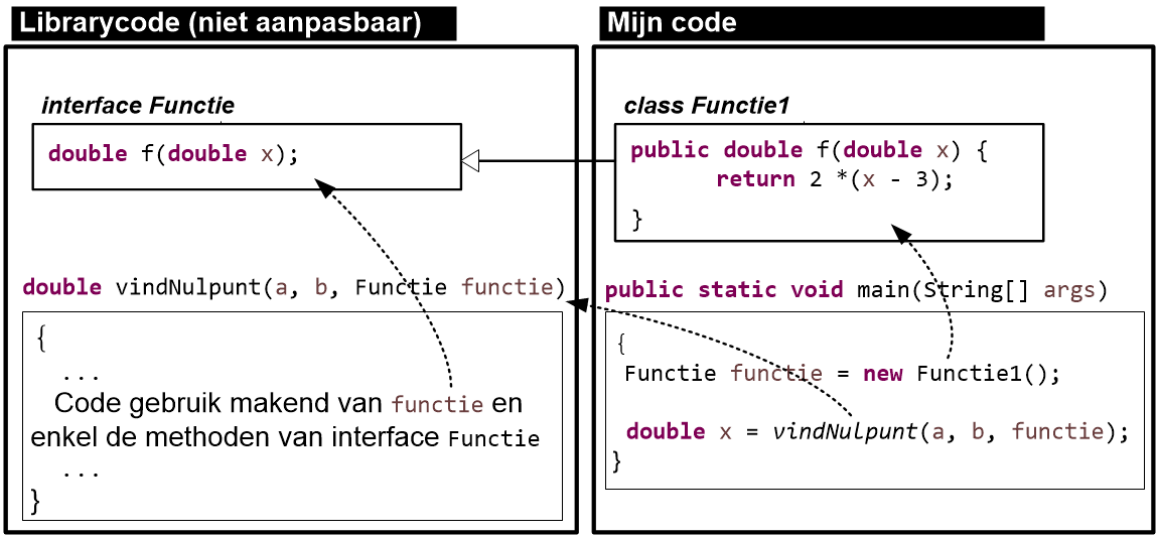
Een concrete klasse voor een functie moet de methode *f* dus implementeren. De drie functies die in de code van 4.1 staan worden nu drie aparte klassen:

```
class Functie1 implements Functie{
      public double f(double x) {
            return 2 *(x - 3);
      }
}
class Functie2 implements Functie{
      public double f(double x) {
            return x * x  + 12 *(x - 3);
      }
}
class Functie3 implements Functie{
      public double f(double x) {
            return - x * x * x / 8 + x * x  + 20 *(x - 3);
      }
}
```

Het algoritme voor het vinden van een nulpunt kan nu aan de slag met het concept functie (de veranderingen staan vetgedrukt en zijn onderlijnd):

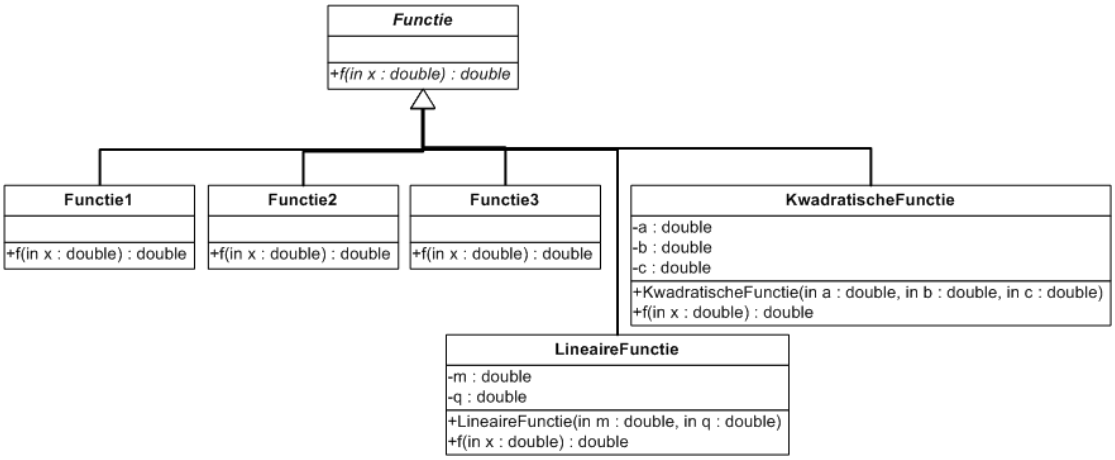
```
public class NulpuntVanFunctie {

	/** PROGRAMMA  */
	public static void main(String[] args) {
	  double a = -10, b = 10;

	    Functie functie = new Functie1();

	    double nulpunt = vindNulpunt(a, b, functie);

	    System.out.println("Nulpunt voor deze functie is "+nulpunt);
	}

	public static double vindNulpunt(double a, double b, Functie
functie){
		if ( !(functie.f(a) < 0 && functie.f(b) > 0))
			throw new IllegalArgumentException("Geef betere
beginpunten!");

		double m = (a + b) / 2;

		while( Math.abs( functie.f(m)) > 0.01){
			System.out.print(m+", ");
			if (functie.f(m) < 0)
				a = m;
			else
				b = m;
			m = (a + b) / 2;
		}
		System.out.println(m);
		return m;
	}
}
```

Het grote verschil is dat de eerste code afhankelijk is van methode *f* die in dezelfde klasse gedefinieerd wordt, terwijl nu *de functie zelf* een parameter is en meegegeven wordt. Deze code kan dus in een bibliotheek (library) gestopt worden, terwijl de functie door de gebruiker apart gedefinieerd wordt en meegegeven bij het oproepen van het algoritme.



Hier zien we dus een toepassing van *abstractie*, de derde pijler van object-georiënteerde programmeertalen.

Nog beter is klassen te maken voor type functies.

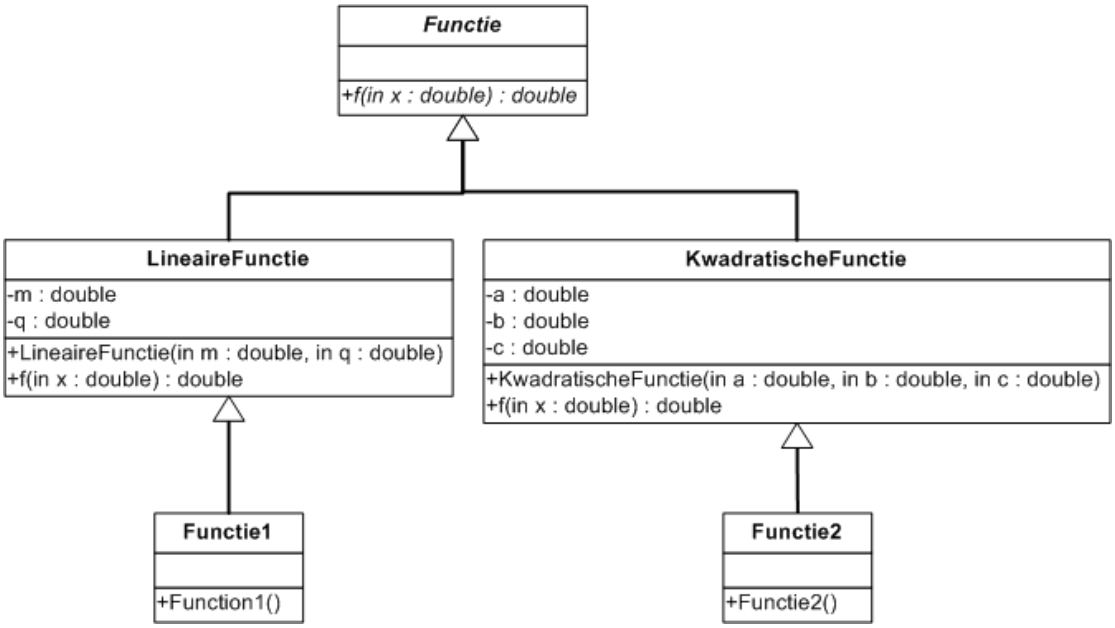



```
class LineaireFunctie implements Functie{
  double m, q;
  public LineaireFunctie(double m, double q){
      this.m = m;
      this.q = q;
  }
  public double f(double x) {
      return m * x + q;
  }
}
class KwadratischeFunctie implements Functie{
  double a, b, c;
  public KwadratischeFunctie(double a, double b, double
c){
      this.a = a;
      this.b = b;
      this.c = c;
  }
  public double f(double x) {
      return a * x * x + b * x + c;
  }
}
```

Het gebruik wordt dan:

```
Functie functie = new KwadratischeFunctie(1, 12, -36);
double nulpunt = vindNulpunt(a, b, functie);
```

Functie 1 en 2 zijn dan in feite subklassen van respectievelijk de lineaire en kwadratische functies:



Dit geeft:

```
class Functie1 extends LineaireFunctie{
      Functie1(){
            super(2, -6);
      }
}
class Functie2 extends KwadratischeFunctie{
  Functie2(){
      super(1, 12, -36);
   }
}
```

De gebruiker moet geen parameters mee geven aan beide klassen, ze zijn vast. De klassen geven deze parameters mee via de superconstructor.

Sinds Java 8 (versienummer 1.8) kan je functies op een eenvoudigere manier meegeven, namelijk als **lambda-expressies**.

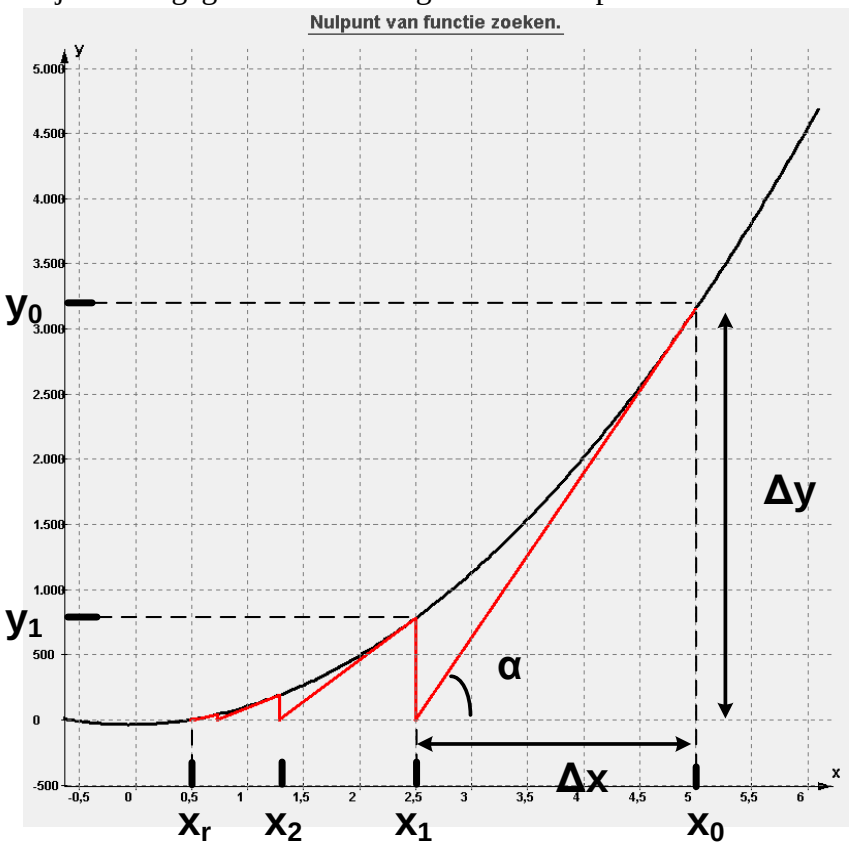
```
double nulpunt = vindNulpunt(a, b, x -> 2*x + x - 3);
```

Je definieert ze met een pijl '**->**': voor de pijl zet je de inputparameters (hier: x), na de pijl zet je de berekening van de output. De Javacompiler checkt dat de gegeven lambda-expressie voldoet aan de gevraagde interface *Functie*, wat hier het geval is!

4.3 Het Newton-algoritme voor het bepalen van nulpunten van een functie

Een tweede algoritme voor een vergelijking van de vorm *f(x) = 0* is het *Newton-algoritme*. Het algoritme vertrekt van een ruwe benadering van een wortel en berekent een betere waarde. Het Newton-algoritme is gebaseerd op een eigenschap die bewezen wordt in het opleidingsonderdeel "Numerieke Algoritmen" (2de Ba.IrW). De oplossingswijze wordt geïllustreerd in de volgende figuur. Het basisidee is om de raaklijn te gebruiken als schatting van het dichtstbij gelegen nulpunt, aangezien de raaklijn de plaatselijke 'mate van verandering' aangeeft. Ze geeft de richting van het

verloop van de functie en kan gebruikt worden voor de schatting van het nulpunt. De raaklijn wordt gegeven door de afgeleide in dat punt.



Neem functie *f(x)* met nulpunt x_r. Indien x_0 een benadering van x_r is en men in x_0 de raaklijn aan de functie *f(x)* tekent, dan is x_1, het snijpunt tussen deze raaklijn en de x-as, een betere benadering van x_r. Door deze eigenschap herhaaldelijk toe te passen, is het dus mogelijk een willekeurig goede benadering van x_r te vinden. De raaklijn wordt gegeven door de *afgeleide* van de functie in het punt. Het tweede punt wordt dan:

$$\frac{\Delta y}{\Delta x} = \frac{\partial f}{\partial x} \Leftrightarrow \Delta x = {}^{\Delta y}\Big/_{\frac{\partial f}{\partial x}} \Leftrightarrow x_1 = x_0 - {}^{\Delta y}\Big/_{\frac{\partial f}{\partial x}}$$

Met de nieuwe schatting van het nulpunt gaan we dit na door $y_1=f(x_1)$ te berekenen. Als y_1 inderdaad pal op of zeer dicht bij nul ligt, hebben we een nulpunt gevonden. Indien niet, berekenen we de volgende benadering op dezelfde manier.
We moeten niet alleen in elk punt de functiewaarde kunnen berekenen, maar ook de waarde van de afgeleide. Deze afgeleide zal ook als eigenschap moeten worden doorgegeven, samen met de functie. Dit lossen we op door een extra methode toe te voegen aan de functie:

```
interface FunctieMetAfgeleide extends Functie {
      /** geeft waarde van de afgeleide in opgegeven punt */
      double afgeleide(double x);
}
```

Deze nieuwe interface heeft twee abstracte methodes: *f()* overgeërfd van *Functie* en *afgeleide()*. Dit is dus een voorbeeld van *overerving*, de tweede pijler van object-georiënteerd programmeren.

```
static int nbrIteraties=0;
public static double vindNulpuntMetNewton(FunctieMetAfgeleide
functie, int eersteGok){
	final int MAX_ITERATIES = 150;
	final double PRECISIE = 0.001;

	double nulpunt = eersteGok;
	double fx = functie.f(nulpunt);
	nbrIteraties=0;
	while (Math.abs(fx) > PRECISIE && nbrIteraties <
MAX_ITERATIES){
		double dfx = functie.afgeleide(nulpunt);
		if (dfx == 0)
			throw new RuntimeException("Nulpunt met Newton:
Afgeleide in "+nulpunt+" is nul waardoor Newton faalt.");

		System.out.println("["+nbrIteraties+"] Current =
"+nulpunt+" fx="+fx+" dfx="+dfx+" => new = "+(nulpunt - fx/dfx));

		nulpunt = nulpunt - fx/dfx;
		fx = functie.f(nulpunt);
		nbrIteraties++;
	}
	if (nbrIteraties >= MAX_ITERATIES)
		throw new RuntimeException("Nulpunt met Newton:
convergeert niet, te veel iteraties, beste punt tot nu toe heeft
waarde "+functie.f(nulpunt)+"!");
	System.out.println("Newton nulpunt = "+nulpunt+" met fx="+fx);
	return nulpunt;
}
```

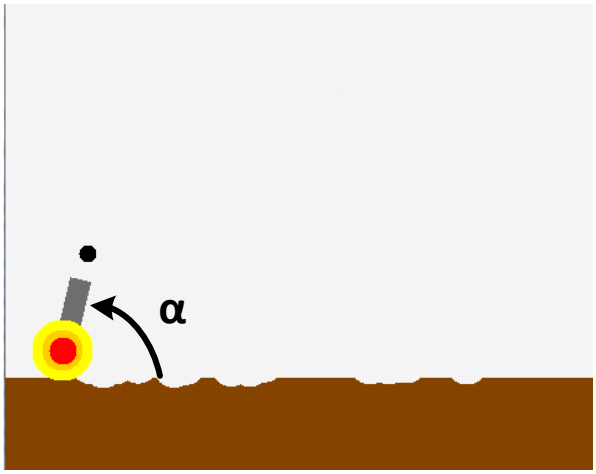
Er kunnen zich twee problemen voordoen. Ten eerste kan het zijn dat we niet korter bij het nulpunt komen. Niets garandeert dat de afgeleide de juiste richting van een nulpunt aangeeft. Indien de functie heel grillig is, kunnen we net verder van een nulpunt terechtkomen. In het ergste geval blijven we zoeken. Om dit te voorkomen, zetten we een maximum op het aantal iteraties die we toelaten. Merk op dat het voor kan komen dat er helemaal geen nulpunt is. Dit wordt ook opgevangen door het beperken van het aantal iteraties.
Een tweede probleem is een afgeleide die nul is. Dit geeft een deling door nul in de formule. Dat is te begrijpen: een afgeleide nul slaat op een raaklijn evenwijdig met de *X*-as, een rechte die de *X*-as nooit zal snijden. In beide gevallen geven we een *RunTimeException*. Merk op dat we in het tweede geval, het algoritme kunnen herstarten met het nemen van een nieuw willekeurig beginpunt.

Merk op dat we de afgeleide kunnen schatten a.d.h.v. twee kort bijgelegen punten x_1 en x_2, en hun functiewaarden:

$$\frac{\Delta y}{\Delta x} \approx \frac{f(x_2) - f(x_1)}{x_2 - x_1}$$

4.4 Discrete simulaties

Om systemen te begrijpen worden deze dikwijls gesimuleerd. Neem het afschieten van een kanonskogel met een bepaalde kracht en onder een bepaalde hoek. Het gewicht van de kogel is een derde parameter.



Alhoewel dat de mechanica gebaseerd is op continue wetten, gaan we ze benaderen door de tijd in kleine tijdstappen (Δt) op te delen (= *discretizeren*). Na elke tijdstap passen we de wetten toe (hier 2-dimensionaal):

$$\Delta v_x = a_x . \Delta t$$
$$\Delta v_x = a_y . \Delta t$$
$$\Delta x = v_x . \Delta t$$
$$\Delta y = v_y . \Delta t$$

Door een versnelling verandert de snelheid van een object, dewelke op zijn beurt de positie aanpast.

De volgende code toont hoe dit gesimuleerd kan worden. We voeren ook wrijving in, deze is proportioneel met de snelheid in het kwadraat, en ook een onzekerheidselement veroorzaakt door bvb de wind.

```
class Kogel{
	double massa;
	double x, y, vx, vy;
	Kogel(double massa){
		this.massa = massa;
	}
}
void simulationLoop(double kogelgewicht, double hoek, double vuurkracht){
	final int TIME_STEP = 1;
	final double G = 9.81, FRICTION=0.0001, UNCERTAINTY=0.1;
	Random rand = new Random();

	Kogel kogel = new Kogel(kogelgewicht);
	kogel.vx = vuurkracht  * Math.cos(hoek) / kogelgewicht;
	kogel.vy = vuurkracht  * Math.sin(hoek) / kogelgewicht;
	int time = 1;
	boolean ended = false;
	while(!ended){
		// forward time
		time += TIME_STEP;

		// update velocity
```

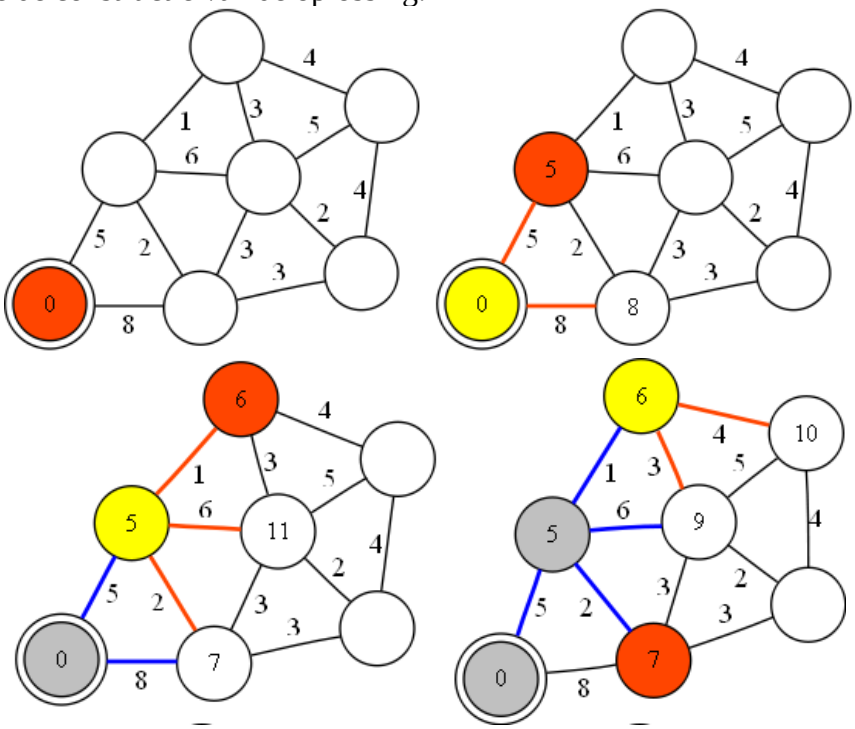
```
            double friction = FRICTION * kogel.vy * kogel.vy;
            kogel.vy += - TIMESTEP * G + (kogel.vy < 0 ? friction : -
friction) + (rand.nextDouble() - 0.5) * UNCERTAINTY;

            friction = FRICTION * kogel.vx * kogel.vx;
            kogel.vx += (kogel.vx < 0 ? friction : -friction) +
(rand.nextDouble() - 0.5) * UNCERTAINTY;

            // update position
            kogel.x += kogel.vx * TIMESTEP;
            kogel.y += kogel.vy * TIMESTEP;

            // check collisions
            if (kogel.y < 0)
                  ended = true;
      }
}
```

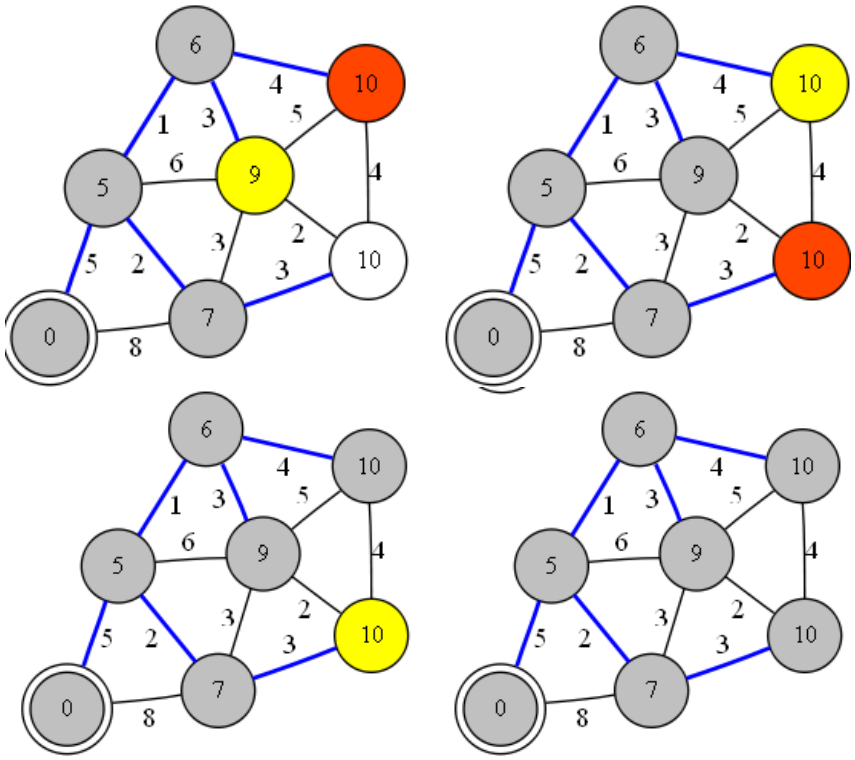
4.6. Het kortste pad met Dijkstra

Het volgende probleem lijkt op het Traveling Salesman Probleem, maar hier bestaat *wel* een oplossing die in lineaire tijd de oplossing kan vinden. Dijkstra's algoritme bepaalt het kortste pad om van één node naar een andere te gaan in een graaf. Elke link heeft een gewicht toegekend gekregen die we kunnen beschouwen als de afstand tussen de 2 nodes of de tijd die het kost om van één node naar de andere te gaan.

Wat is het kortste pad van een node naar node 0 waarbij we de som van de gewichten willen minimaliseren?

Hier zie je de constructie van de oplossing:





Oplossing: vanuit elke node, neem de blauwe link en zo kom je tot bij 0.

Algoritme:

Het algoritme onthoudt per node:
- de tot-dan-toe gevonden afstand naar node 0. Initieel op oneindig gezet.
- de eerstvolgende node die hij moet nemen om naar node 0 te gaan (in blauw aangegeven in graaf).

Afstand van node 0 zet je op 0. De node voeg je toe aan de priority queue.
Doe zolang priority queue niet leeg is:
- Neem eerste element uit priority queue (kortste afstand tot node 0)
- Ga voor deze node al zijn buren af:
  - Tel de afstand tot buur bij je eigen afstand.
  - Kijk of deze afstand beter is dan de tot-dan-toe gevonden afstand van de buur. Indien dit zo is, update de tot-dan-toe gevonden afstand en geef aan dat de buur naar de node moet gaan.
  - Als de buur nog niet in de priority queue is, voeg hem toe.

Hoofdstuk 5

Slimme algoritmen

In dit hoofdstuk bespreken we een reeks slimme algoritmen die ons helpen om bepaalde beslissingen te nemen. Met deze beslissingen willen we een specifiek doel bereiken of een score maximaliseren.

De code van dit hoofdstuk vind je terug in **package *zoeken***.

5.1. Type algoritmen

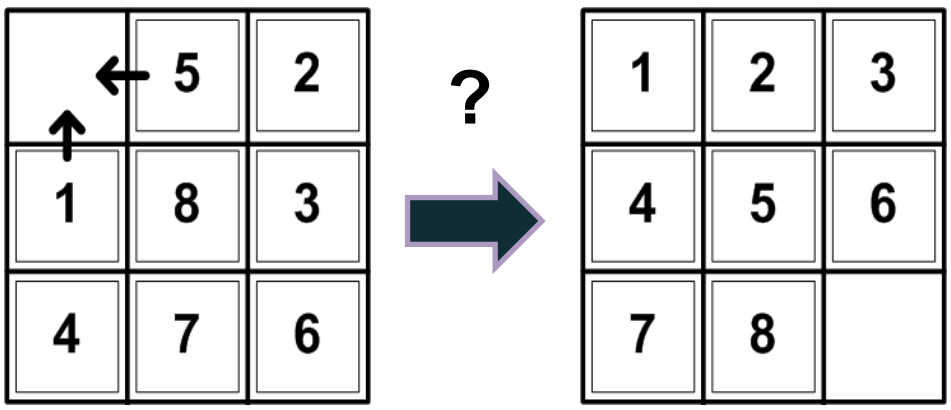
In hoofdstuk 1 deelden we de algoritmen in volgens 5 types:

Type 1: De oplossing kan berekend worden met een formule (analytisch).
Type 2: Je kunt de oplossing gericht zoeken of construeren (rechttoe-rechtaan).
Type 3: Je gaat alle mogelijke actiesequenties af om een oplossing te vinden.
Type 4: Door slimme keuzes (*heuristieken*) te maken, kan je verschillende actiesequenties uitsluiten.
Type 5: Je leert al doende welke de juiste keuzes zijn.

In dit hoofdstuk starten we met algoritmen van Type 3 en gaan dan richting algoritmen van type 4 en 5. Deze algoritmen worden onderzocht in het domein **Artificiële Intelligentie** (AI), dat zich richt op het ontwerpen van intelligente computerprogramma's. Natuurlijk kunnen we hier maar een klein tipje van de sluier oplichten.

5.2. De schuifpuzzel

We starten ons onderzoek met het bestuderen van een bekend probleem: de schuifpuzzel waarbij je de verschillende stukjes in de juiste volgorde moet schuiven.

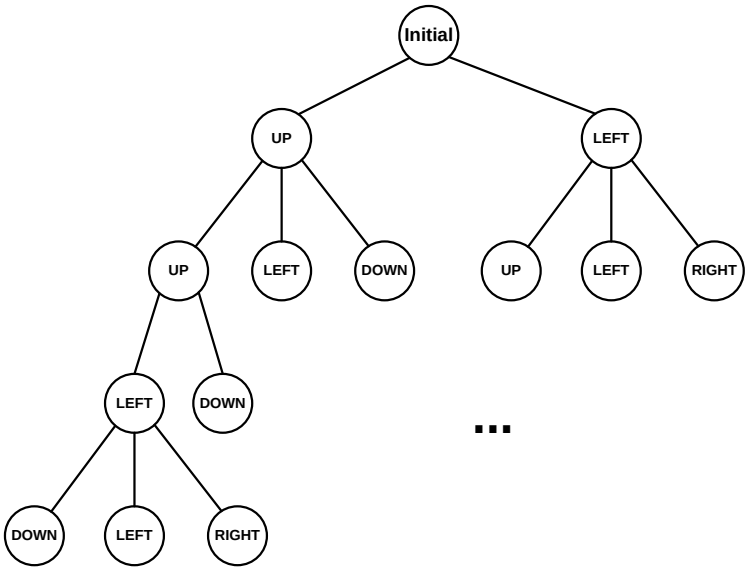


In elke toestand heb je maximaal vier mogelijke acties: UP, RIGHT, DOWN, LEFT. Enerzijds willen we een sequentie van acties die ons tot de geordende configuratie leidt, anderzijds willen we de kortste sequentie die dit bereikt. Welke algoritmes kunnen dit?

5.2.1. De zoekboom

Stel dat we het minimale aantal stappen willen vinden om tot bij de juiste configuratie te komen. Dit kunnen we doen door alle mogelijke actiesequenties af te lopen. De verzameling van ‘alle mogelijke actiesequenties’ noemen we de **zoekruimte**.

Alle mogelijke sequenties kan je voorstellen met een omgekeerde boom. Neem bovenstaande beginsituatie. Als eerste stap heb je twee mogelijkheden (UP en LEFT). Daarna heb je telkens drie mogelijkheden, enzovoorts. Dit geeft de volgende boom:

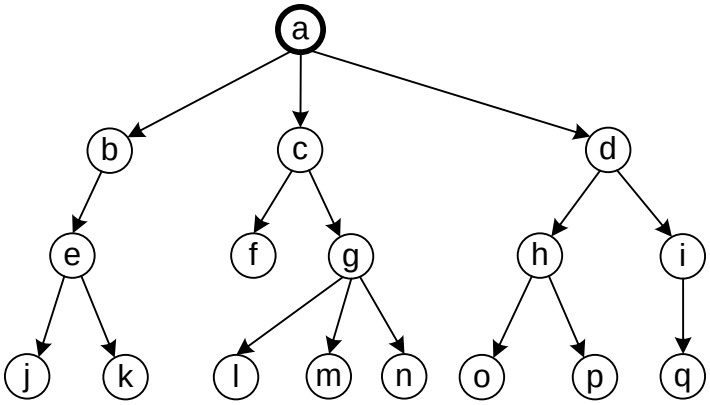


We kunnen deze boom beperken door niet terug te keren: na een UP doen we geen DOWN want dan komen we weer in dezelfde toestand terecht. In vakjargon noemen we dit het ‘snoeien’ van de boom (‘to prune’ in het engels). Door deze boom op te stellen, construeren we de verzameling van alle sequenties. Elk pad in de boom stelt een actiesequentie voor. Hierin kunnen we de sequentie zoeken die het snelst tot de oplossing leidt. Het aantal stappen wordt gegeven door de ‘diepte’ van de boom.
We zullen 6 algoritmen bespreken om dit probleem op te lossen:

1. Depth-first search	*4. Best-first search*
2. Breadth-first search	*5. A-star*
3. Greedy search	*6. Iterative deepening*

5.2.2. Brute-force search: depth-first versus breadth-first

Om de oplossing(en) uit de boom te halen, moeten we de boom doorlopen. Dit kan op twee manieren: **depth-first traversal** en **breadth-first traversal**. Neem de volgende boom:



Depth-first gaat eerst in de diepte. Ze loopt de nodes af als volgt: a-b-e-j-k-c-f-g-l-m-n-d-h-o-p-i-q. ***Breadth-first*** gaat eerst in de breedte: eerst de nodes van diepte 1, dan van diepte 2, enzovoorts. De nodes lopen we af in de volgorde: a-b-c-d-e-f-g-h-i-j-k-l-m-n-o-p-q. Beide zijn mogelijk.

Het voordeel van *depth-first* is dat het in feite kan zonder de zoekboom in een datastructuur te moeten bijhouden. We beginnen met het eerste pad, helemaal links. Als dat niets oplevert, nemen we het tweede pad, enzovoorts. Dit wordt ook **backtracking** genoemd, omdat het algoritme "*incrementally builds candidates to the solutions, and abandons each partial candidate c ('backtracks') as soon as it determines that c cannot possibly be completed to a valid solution*" [wikipedia]. Op `http://en.wikipedia.org/wiki/Backtracking` kan je een mooie animatie zien over het oplossen van een sudoku met backtracking.

Bij *breadth-first* moeten we dit wel, want we creëren eerst alle nodes op een zekere diepte vooraleer we naar de volgende diepte gaan. Hiervoor moeten we alle nodes van de laatste diepte bijhouden. *Breadth-first* is echter logischer als we op zoek gaan naar de kortste actiesequentie. Met *depth-first* zal het linkse deel van de boom eerst helemaal doorlopen worden, terwijl de oplossing zich meer naar rechts kan bevinden op een kleinere diepte. Met *breadth-first* is het ook niet echt nodig om een maximale diepte mee te geven omdat we stap-voor-stap dieper gaan zoeken in de boom zolang er nog geen oplossing gevonden is. Bij *depth-first* moet zeker wel een maximale diepte voor de schuifpuzzel worden meegegeven, omdat het meest-linkse pad anders oneindig lang zou worden.

Dit zijn dus algoritmes van het Type 3: we doorploegen de hele zoekruimte op zoek naar de beste oplossing. Dit wordt ook **brute force search** genoemd, omdat we op pure rekenkracht zoeken, niet echt intelligent dus.

Hier de code voor backtracking. Vanuit de startnode (initiële toestand van de puzzel) gaan we alle mogelijke zetten af en checken of we de oplossing gevonden hebben. Dit

doen we recursief: de nieuwe nodes worden ook nagegaan op dezelfde manier. Merk op dat als we een nieuwe node hebben onderzocht, we de zet ongedaan maken.

```
public static <Zet> boolean backtracking (ZoekNode<Zet> node){
	List<Zet> zetten = node.possibleMoves();

	for(Zet zet: zetten ){
		node.move(zet);
		if (node.isSolution()){
			System.out.println("Oplossing gevonden: "+node+"!");
			return true;
		}
		if (backtracking(node))
			return true; // stop met verder zoeken

		node.undoMove(zet); // keer terug
	}
	return false; // geen children of geen enkele leidde naar oplossing
 }
```

Deze code is *abstract*. We hebben het algoritme op een algemene manier beschreven die *onafhankelijk* is van het probleem. Je komt er niets tegen dat specifiek te maken heeft met het puzzelprobleem. Om die reden definiëren we *ZoekNode* als een interface en is *Zet* een geparametriseerd type (Java generics).

```
public interface ZoekNode<Zet> {
	List<Zet> possibleMoves();
	void move(Zet zet);
	void undoMove(Zet zet);
	boolean isSolution();
	ZoekNode<Zet> clone();
}
```

Om het algoritme voor een specifiek probleem op te lossen moet je de methodes van de interface implementeren. De code voor het oplossen van de puzzel vind je online. Verder moet je het type van *Zet* aangeven. Voor de puzzel zal dit een *Java enum* (enumeratie) zijn met de waarden UP, LEFT, DOWN en RIGHT.

Bovenstaande code legt het principe van backtracking uit, maar is nog niet bruikbaar. De backtracking stopt immers pas als een oplossing bereikt is. Dat wil zeggen dat hij de linkertak van de boom verder gaat uitwerken zonder deze ooit te beëindigen (ga na wat er gebeurt). We gaan dit verbeteren door een maximale diepte in te stellen. Verder willen we ook nog het pad tot de oplossing construeren. Dit doen we door de zetten in een lijst te stoppen bij het beëindigen van de recursie. Als een oplossing gevonden is, worden alle functie-aanroepen afgesloten en voegen we de zet toe aan de lijst. We geven het pad dus terug als output, de onderliggende functie-aanroep checkt of er een pad is teruggegeven en weet zo dat een oplossing gevonden is. In de eerste code werd er een boolean teruggegeven waarmee de backtracking gestopt werd.

```
static int MAXIMALE_DIEPTE = 10;
public static <Zet> List<Zet> backtrackingMetPadEnMaximaleDiepte
(ZoekNode<Zet> node, int max_diepte){
        MAXIMALE_DIEPTE = max_diepte;
        return backtrackingMetPadEnMaximaleDiepte_rec(node, 1);
 }
private static <Zet> List<Zet> backtrackingMetPadEnMaximaleDiepte_rec
(ZoekNode<Zet> node, int diepte){
       List<Zet> zetten = node.possibleMoves();

       for(Zet zet: zetten ){
              node.move(zet);
              if (node.isSolution()){
                     System.out.println("Oplossing gevonden: "+node+"!");
                  // pad wordt aangemaakt bij het terugkeren uit de recursie
                     List<Zet> pad = new ArrayList<Zet>();
                     pad.add(zet);
                     return pad;
              }
              if (diepte < MAXIMALE_DIEPTE){
                     List<Zet> pad =
backtrackingMetPadEnMaximaleDiepte_rec(node, diepte+1);
                     if (pad != null){
                            pad.add(0, zet); // voeg vooraan toe
                            return pad; // stop met verder zoeken
                     }
              }
              node.undoMove(zet); // keer terug
       }
       return null; // geen children of geen enkele leidde naar oplossing
 }
```

De *breadth-first* werkt met dezelfde interface, maar zonder recursie. We houden de nodes bij in de FIFOQueue van hoofdstuk 3 die we initialiseren met de huidige toestand (*startnode*). Vervolgens worden de nodes uit de queue genomen en de mogelijke zetten afgegaan of ze een oplossing opleveren. Merk op dat we nu telkens een kopie van de node moeten maken, want we houden ze apart bij. Bij de backtracking was dit niet nodig, je voerde de zet uit, deed de recursie en zette vervolgens de zet terug. Ga na wat er gebeurt indien we geen kopie zouden maken.

```
public static <Zet> boolean breadthfirst (ZoekNode<Zet> startnode){
   FIFOQueue<ZoekNode<Zet>> openNodes = new FIFOQueue<ZoekNode<Zet>>(1000);
   openNodes.add(startnode);

   while(!openNodes.isEmpty()){
       ZoekNode<Zet> node = openNodes.get();
       List<Zet> zetten = node.possibleMoves();
       for(Zet zet: zetten ){
              ZoekNode<Zet> child = node.clone(); //copy!!
              child.move(zet);
              if (child.isSolution()){
                     System.out.println("Oplossing gevonden!");
                     return true;
              }
              openNodes.add(child);
       }
   }
   return false; // geen oplossing gevonden
 }
```

5.2.3. Gericht zoeken: *greedy search*, *best-first* en *A-star*

De zoekruimte van alle mogelijke oplossingen wordt voor de meeste problemen al snel te groot om alle mogelijkheden te kunnen aflopen. Neem de schuifpuzzel. Als we aannemen dat je gemiddeld twee mogelijke voortzettingen hebt (we sluiten het terugkeren naar de vorige situatie uit) en de optimale oplossingen ongeveer tien stappen telt, dan heb je $2^{10} \approx 1000$ mogelijke sequenties. Dit is voor een mens al niet meer te overzien. Een computerprogramma zou de oplossing nog wel snel weten te vinden. Voor een grotere puzzel wordt dit ook al snel ondoenbaar, want voor een puzzel van 6x6 heb je waarschijnlijk meer dan honderd stappen nodig (je hebt 36 stukjes, maar ieder stukje zal je meermaals moeten bewegen). Dit geeft $2^{100} \approx 10^{30}$ mogelijke sequenties. Dat is voor een computer quasi-onberekenbaar.

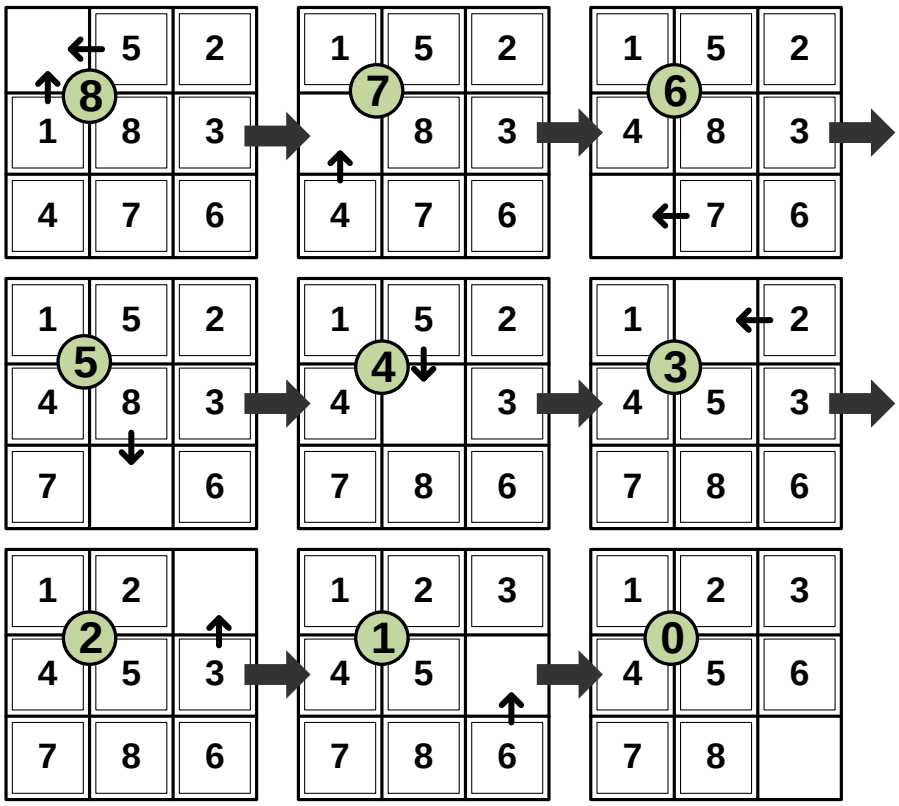
Laat ons op zoek gaan naar oplossingen die voorkomen dat we *alle* mogelijkheden moeten afgaan. Een eerste, wat naïeve oplossing is een actie kiezen die ons dichter bij de eindconfiguratie brengt, zonder vooruit te denken. Je checkt of het verschuiven het stukje dichter bij de eindpositie brengt. De afstand bereken je met de Manhattan-afstand ($|x_1 - x_2| + |y_1 - y_2|$). Deze afstand geeft het aantal verplaatsingen nodig om een stuk tot zijn eindpositie te brengen. Vervolgens som je de afstanden van alle stukjes. Dit noemen we de **score** van de speltoestand. Een score van 0 betekent dat we de eindtoestand bereikt hebben. Om dit algoritme te implementeren, voegen we de methode **float** score(); toe aan de interface *ZoekNode*. De score maakt de som van de Manhattanafstand tot eindtoestand voor alle vakjes.

Bij *greedy search* kies je op een hebzuchtige manier voor de directe winst, je kijkt niet verder naar de toekomst. Je kiest een actie die de score verbetert. Als er meerdere acties zijn die dit doen, kies je de eerste actie. Als geen enkele actie leidt tot een verbetering, stoppen we het zoeken.

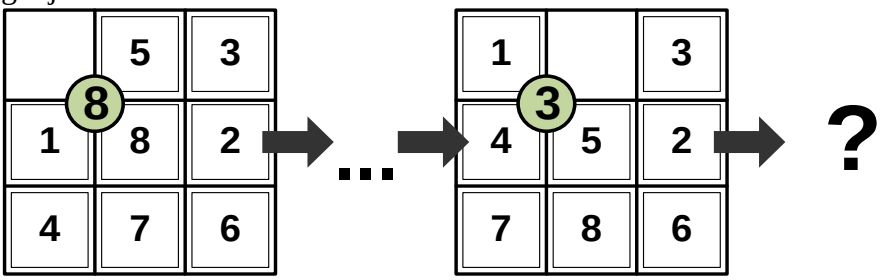
```
public static <Zet> boolean greedySearch (ZoekNode<Zet> node){
      int min_score = node.score();
      boolean moved;
      do {
            moved = false;
            for(Zet zet: node.possibleMoves() ){
                  node.move(zet);
                  if (node.isSolution())
                        return true;
                  if (node.score() < min_score){
                        min_score = node.score();
                        moved = true;
                        break;
                  }
                  node.undoMove(zet); // keer terug
            }
      } while (moved);
      return false;
}
```

De code van de 6 algoritmen vind je in ZoekAlgoritmen.java in package *zoeken*.

Dit zal echter niet altijd leiden tot een oplossing, maar wèl in de opgegeven beginsituatie. Dit zie je hier. We hebben bij elke toestand de score bijgezet.



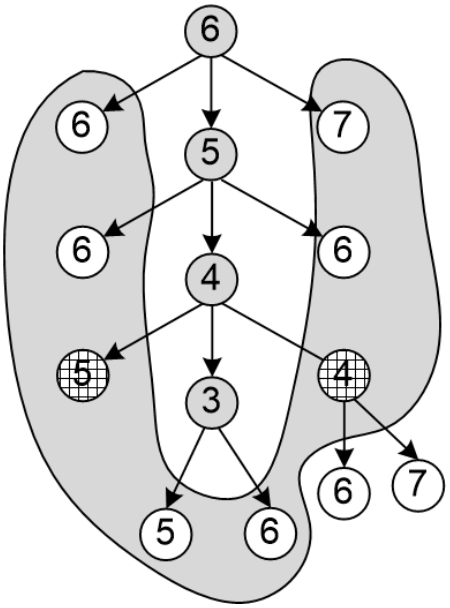
Als we stukje 2 en 3 echter omwisselen in de beginsituatie lukt greedy search niet meer. Het leidt na vijf acties tot een situatie waarin geen onmiddellijke verbetering meer mogelijk is:



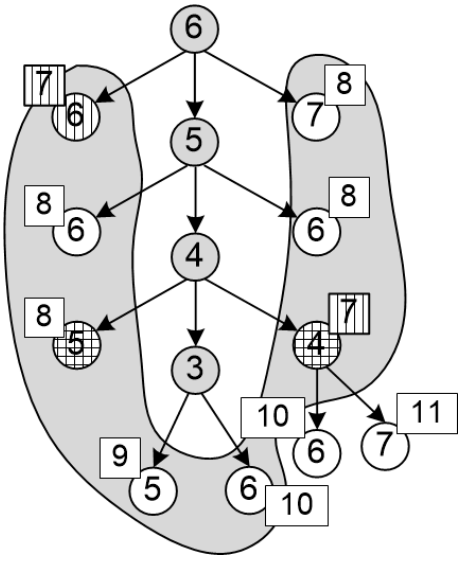
Op de zoekboom van de volgende pagina (onder 5.2.4) zie je dat er 3 zetten zijn die naar een score van 4 leiden. Het bovenstaand algoritme zal dan stoppen. We zullen verder in de toekomst moeten kijken en accepteren dat de score even moet verslechteren vooraleer ze verbetert.

Een idee van student Ruben Pauwels (examen juni 2016) is *greedy search* te combineren met *backtracking*: als je vast zit, keer je terug op je stappen en neemt het tweede-beste pad. Dit noemen we ***best-first search***. Hiervoor moeten we alle voorgaande nodes bijhouden zoals bij *breadth-first*. Het enige wat we vervolgens moeten veranderen is het gebruiken van een PriorityQueue in plaats van een gewone first-in first-out queue. De PriorityQueue zal de elementen sorteren op score. De verschillen zijn in het grijs aangeduid.

```
public static <Zet> List<Zet> bestFirst(ZoekNode<Zet> startnode) {
  PriorityQueue<ZoekNode<Zet>> openset =new PriorityQueue<ZoekNode<Zet>>();
  openset.add(startnode);

  while(!openset.isEmpty()){
       ZoekNode<Zet> node = openset.poll(); //node met de laagste score

       List<Zet> zetten = node.possibleMoves();
       for(Zet zet: zetten ){
              ZoekNode<Zet> child = node.clone(); //copy!
              child.move(zet);

              if (child.isSolution()){
                 System.out.println("Oplossing gevonden: "+child+"!");
                 return path; // code constructie pad hier weggelaten
              } else
                 openset.add(child);
       }
  }
  return null; // failure to find solution
}
```



De zoekboom hier links getoond is een deel van de totale zoekboom.
In de zoekboom komt greedy search vast te zitten in node 3. De 4 *bezochte* nodes zijn aangeduid in het grijs. De 8 omcirkelde nodes zijn ook *bekeken*: de score van deze moest ook berekend worden om vervolgens de beste te nemen. Er zijn dus 12 nodes bekeken door het algoritme.
Best-first komt in eerste instantie ook bij node 3 uit, maar zal verder gaan met node 4 en vervolgens node 5 (in ruitjes gearceerd).



Een populair algoritme voor dit soort problemen is het ***A-star algoritme***, wat een variant van bovenstaande best-first is. De score bestaat uit 2 delen:

- g-score = afstand vanaf start
- h-score = afstand tot doel = onze score

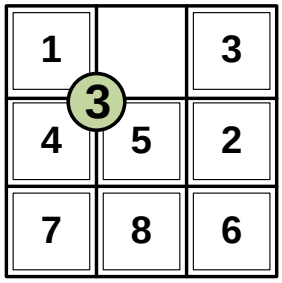
In de zoekboom (links) geven we de a-starscore in de vierkantjes. De g-score is 0 voor de eerste node, 1 voor de volgende nodes…
A-star verschilt dus enkel van best-first in het berekenen van de score. In de linkse zoekboom zal a-star na node 3, node 4/7 en vervolgens node 6/7 (verticaal gearceerd) bezoeken. Het volgt dus een ander pad dan best-first.

5.2.4. Iterative deepening & fitness landscapes

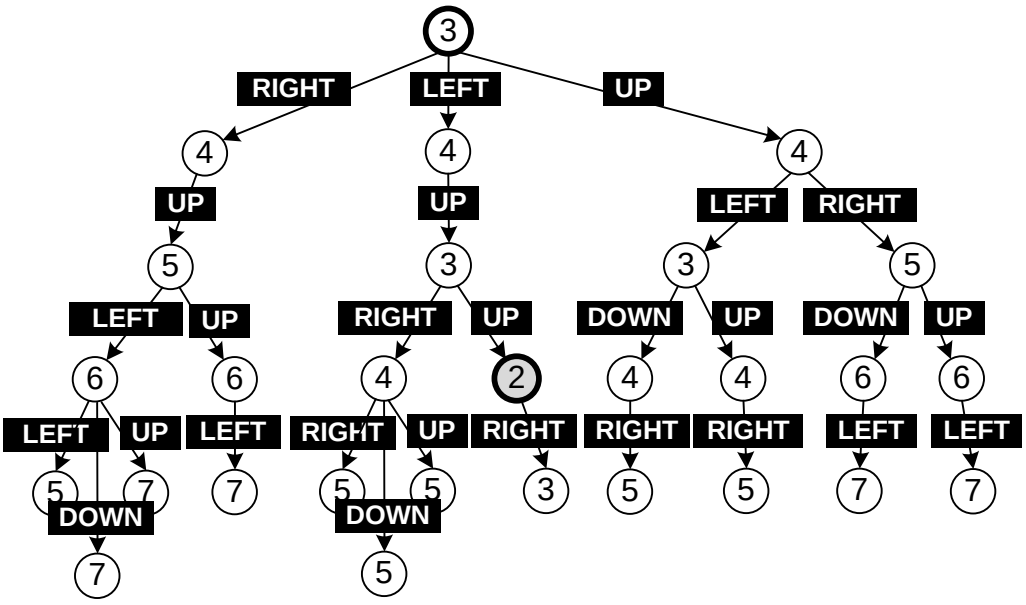
Voor de meeste problemen moet je *een aantal stappen vooruitdenken*, zoals in de schuifpuzzel (of met schaken zoals we later zullen zien).

Om te voorkomen dat we de hele zoekruimte moeten afgaan naar de oplossing, beperken we de zoekruimte door op zoek te gaan naar een oplossing in de toekomst die onze score verbetert. We ontplooien de zoekboom tot een welbepaalde diepte, de **horizon**, en kiezen vervolgens de node met de beste score. Dit doen we iteratief, na elke stap kijken we 1 stap verder en kiezen we het beste pad. Op deze manier proberen we stelselmatig de huidige situatie te verbeteren.

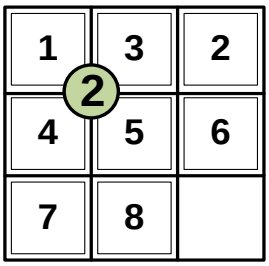
Hoe ver moeten we vooruitkijken? Het is natuurlijk niet de bedoeling dat we de hele zoekboom opstellen. Valt het te bewijzen dat met een bepaalde horizon we de oplossing vinden? Neem de eindsituatie van de vorige paragraaf.



Deze heeft een score van 3 (stukje 2 is op afstand 2 van correcte positie, stukje 6 op afstand 1). Dit is de zoekboom tot diepte 4 vanaf deze positie, met aanduiding van de score:



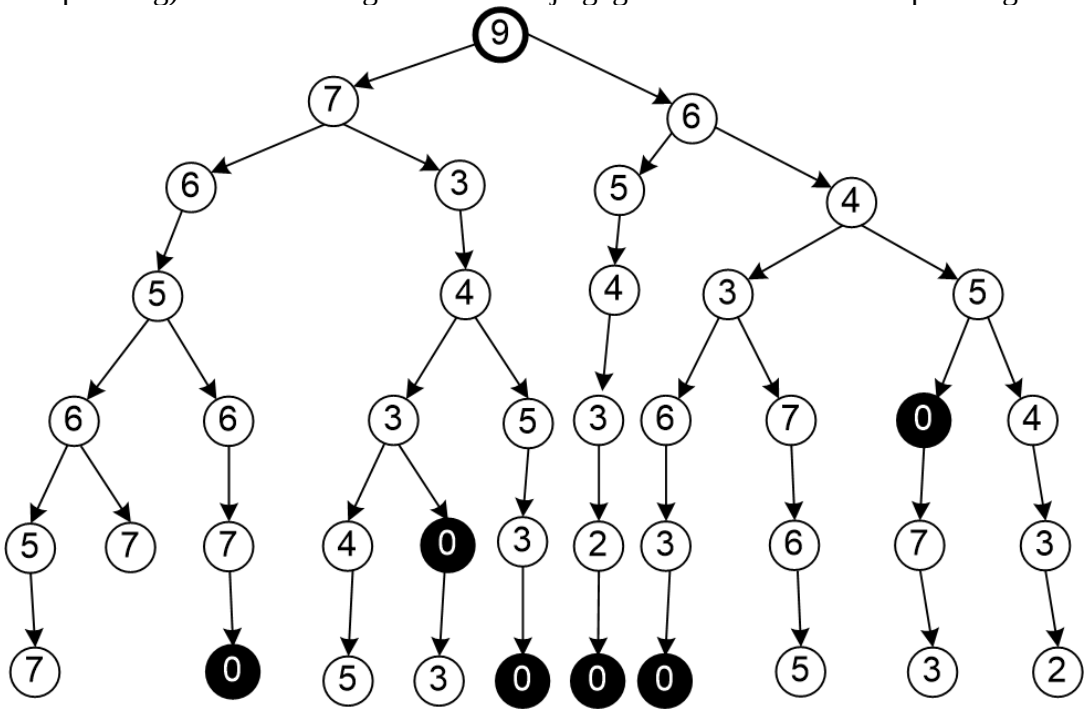
De eindconfiguratie is dus niet haalbaar in 4 stappen. En erger nog, deze boom vertelt ons nog steeds niet wat de volgende stap zal zijn. De best haalbare score in deze boom heeft score 2, maar daarmee belanden we in een situatie die ons in feite nog verder verwijdert van de eindconfiguratie:



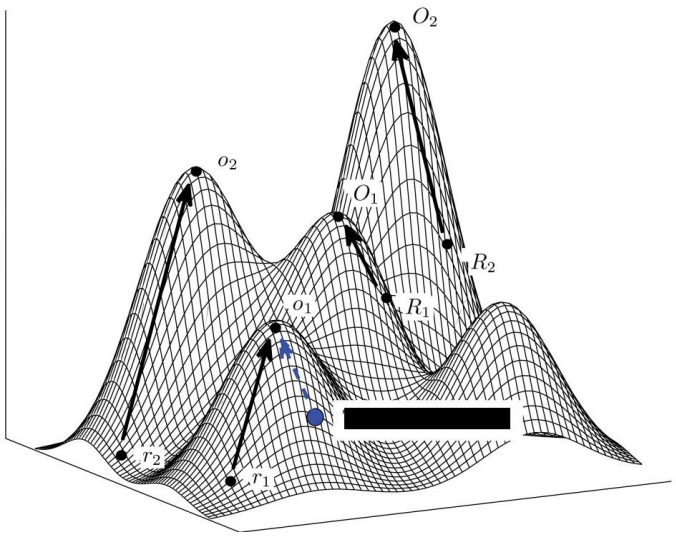
Dit laat duidelijk de beperkingen van deze aanpak zien. We moeten blijkbaar verder in de toekomst kijken…

We hebben een app gemaakt waarmee je de 6 algoritmen kan vergelijken en zelf ook het spel spelen. Te vinden als **SchuifPuzzel.jar** op parallel website (Theorie – Les 7) of als **Schuifpuzzel.java** in **package schuifpuzzel**.

Oefening (ex-examenvraag)
Neem de volgende zoekboom waarbij ook een score voor elke toestand berekend is (al is die niet een perfecte inschatting van de situatie) en we een score 0 willen bereiken. Pas de 6 verschillende zoekalgoritmen toe (neem als horizon 2 voor het desbetreffende algoritme). Toon in detail hoe ze werken (volgorde van stappen). Vergelijk de snelheid van de algoritmen om een oplossing (zwarte node) te vinden en vergelijk ze voor het vinden van de kortste oplossing (minst aantal stappen van root tot oplossing). Met welke algoritmen vind je gegarandeerd de kortste oplossing?

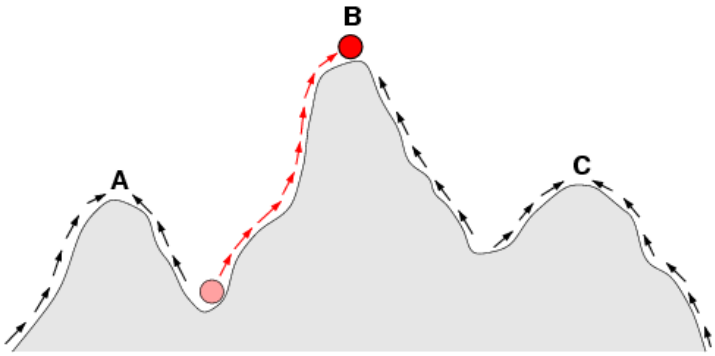


Laat ons nu bestuderen hoe de score verandert in de zoekruimte. Dan verkrijgen we een soort landschap, ook **fitness landscape** genoemd. De hoogte geeft de score aan, het horizontale vlak geeft alle mogelijke toestanden weer. Al moet gezegd worden dat men meestal met een meerdimensionale ruimte te maken heeft.



In de figuur gaat het er om de score te maximaliseren, wat het geval is voor de meeste problemen. Voor onze schuifpuzzel definieerden we een score die je moet minimaliseren. Essentieel is er geen verschil, je moet enkel alles omdraaien.

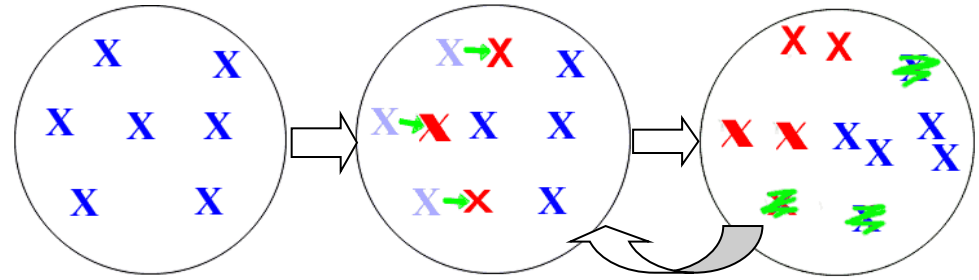
Naburige toestanden hebben een score die dicht bij elkaar ligt. Door de zoekruimte lopen is als het bewandelen van een soort berglandschap. *Greedy search* van vorige paragraaf komt erop neer om vanuit het beginpunt steeds een weg naar boven te kiezen. Het is duidelijk te zien dat je zo al snel op een **lokaal maximum** terecht komt, waarna je geen directe winst meer kunt behalen. Op de volgende figuur zie je duidelijk de punten die leiden tot het globaal maximum B en de punten die leiden tot lokale maxima A en C.



Om van A een situatie met een hogere score te bereiken, moet de horizon van ons algoritme dus groot genoeg zijn.

Ter informatie, de term **fitness landscape** komt uit de biologie. De evolutie van soorten, zoals ontdekt door Charles Darwin, is ook een ‘wandeling’ door een fitness landscape. De *fitheid* van een levend wezen wordt bepaald door zijn aangepastheid aan de omgeving (mogelijkheid om zich te verdedigen tegen vijanden, voorzien van energie en bouwstoffen, voort te planten, enzovoorts). Als onvoldoende aangepast, zal het levend wezen niet overleven, dat heet *natuurlijke selectie*. Door *genetische mutaties* zal de bouw en het gedrag van een soort veranderen en aldus ook zijn fitheid. Aldus ontstaat er een evolutie van soorten en een klim door het fitness landscape, waarbij elke stabiele soort op een bergtopje zit.

Informatici hebben het principe van de biologische evolutie gebruikt voor efficiënte algoritmes, **genetische algoritmes** genaamd. Ze starten met een populatie van willekeurige oplossingen die niet optimaal zijn. Een oplossing wordt een individu genoemd. Oplossingen worden dan willekeurig veranderd (mutaties) en gecombineerd. Vervolgens worden op basis van de score (fitheid) de slechtste oplossingen geëlimineerd (zoals bij natuurlijke selectie).



Iteratief muteert men de individuen en selecteert de beste. Met de hoop dat de optimale oplossing(en) boven komen drijven. Een genetisch algoritme is van het type 4. Door kleine veranderingen toe te passen en goede oplossingen te combineren hopen we te evolueren naar een optimale oplossing zonder dat alle mogelijke oplossingen moeten afgaan. Er is natuurlijk geen garantie op succes.

5.2.5. Verdere verbeteringen van de algoritmes

De oplossingen die we tot hiertoe hebben besproken zijn de typische methodes van computerprogramma's. We hebben echter onze bedenkingen bij de kracht van deze algoritmes. Laat ons eens kijken hoe een mens dit aanpakt.

(a) We lossen de puzzel *rij-per-rij* af. Behalve de laatste twee rijen, die moeten we tegelijkertijd aanpakken. Dit kunnen we implementeren door de score enkel te definiëren op de stukken van de eerste rij. Als die af is (score 0), bereken je de score op de stukken van de volgende rij. Op het einde neem je de score van de 2 laatste rijen.

(b) Een mens probeert van alles uit en *leert* van zijn ervaringen. Het algoritme wordt continu verbeterd. Dit bespreken we in 5.4.

(c) We stellen *regels* op die ons toelaten telkens de juiste keuze te maken.

Kunnen we deze ideeën gebruiken om onze algoritmes te verbeteren?

(b) kunnen we redelijk eenvoudig toepassen: we passen de scoredefinitie aan. In het begin nemen we enkel de stukken van de eerste rij in rekening. Als die rij af is (score 0), brengen we de stukken van de tweede rij in rekening, enzovoorts. Behalve op het einde, dan brengen we de stukken van de laatste twee rijen in rekening. Ik heb het niet getest, maar ik denk dat we zo met *een beperkte horizon* tot de oplossing kunnen komen. (Al ben ik er niet zeker van dat dit steeds de kortste oplossing is.)

Tot slot, ik ben ervan overtuigd dat je regels kunt opstellen die aangeven welke de beste zet is in de gegeven situatie. Dat is wat ieder kind doet. Het probeert door logisch na te denken een aantal regels op te stellen waarmee de oplossing van het probleem geconstrueerd kan worden. Men kan deze regels programmeren onder de vorm van *if(…) { … } else if (…){… } else if (…){… },* enzovoorts. Zo krijgen we een oplossing van *Type 1*. Dit is echter niet voor alle problemen mogelijk.

5.2.6. Implementatie

Op de website vind je uitleg bij de Javacode voor het oplossen van de schuifpuzzel.

We laten zien hoe we deze code **generiek** maken: we definiëren het probleem op een abstracte manier en implementeren de algoritmes op dit abstract probleem. Zodoende zijn de algoritmes toepasbaar op gelijkaardige problemen zonder dat je de oplossingsmethode moet herprogrammeren! Je kan de oplossingen hergebruiken. Verder laten we zien hoe je meerdere oplossingen voor hetzelfde probleem kunt maken en deze op een generieke wijze kunt vergelijken. Deze code maakt gebruik van de 3 pijlers van object-oriëntatie en laat haar ware kracht zien.
Als je dit inziet en kan toepassen, dan heb je de fundamenten om een volleerde programmeur te worden.

Aangezien deze cursus voor de meeste studenten een eerste kennismaking is met Java en object-oriëntatie wordt de bijgevoegde implementatie niet gerekend bij de leerstof. Ze is wel beschikbaar voor de nieuwsgierige student en mag gebruikt worden in het project.

5.2.7. NP-complete problemen en het traveling salesman-probleem

Wetenschappers denken dat er een klasse van problemen bestaat die een *exponentiële zoektijd* vergen voor het vinden van de optimale oplossing. Dit gebeurt als je een groot deel van de zoekboom moet doorzoeken. Als een node in de zoekboom gemiddeld t takken heeft, dan is de grootte van de zoekruimte ongeveer t^d met d de diepte van de boom. Dit is een exponentiële functie (zie 1.2). Alleen extreem snoeien of reduceren van de boom geeft een polynomiale zoektijd. Voor **NP-complete problemen** is dit niet mogelijk.

Eén van de bekendste en belangrijkste NP-complete problemen is het ‘traveling salesman’-probleem. Stel je een vertegenwoordiger voor die de volgende steden wil bezoeken in de meest efficiënte volgorde, d.w.z. de kortste totale afstand.

Er zijn 1.307.674.368.000 mogelijke volgorden om deze vijftien steden af te gaan, namelijk de faculteit van 15. Men denkt dus dat je deze lijst van mogelijkheden niet substantieel kunt beperken; je hebt altijd kans om de optimale oplossing over het hoofd te zien. Je moet praktisch alle mogelijke sequenties nagaan.

Omdat men onmogelijk alle mogelijke volgorden kan aflopen, gebruikt met heuristieken (type 4) zoals het idee van Melissa De Staercke (examen juni 2014): beschouw voor een stad enkel de dichtst-bijzijnde steden. Het volgende pad is gevonden door een genetisch algoritme, het ligt waarschijnlijk heel dicht bij de optimale oplossing: (http://ai4r.rubyforge.org/geneticAlgorithms.html):

Volgend filmpje toont die genetische algoritmen toegepast op het traveling salesman probleem: https://www.youtube.com/watch?v=94p5NUogClM.

Wetenschappers denken - maar het is tot nog toe niet bewezen - dat het onmogelijk is de zoekruimte te beperken tot een polynomiale zoektijd. Het is daarentegen wèl bewezen dat als er voor één van de problemen een snel algoritme bestaat, dat dan *alle* NP-complete op een snelle manier opgelost kunnen worden. Het is bewezen dat alle problemen in feite equivalent zijn.

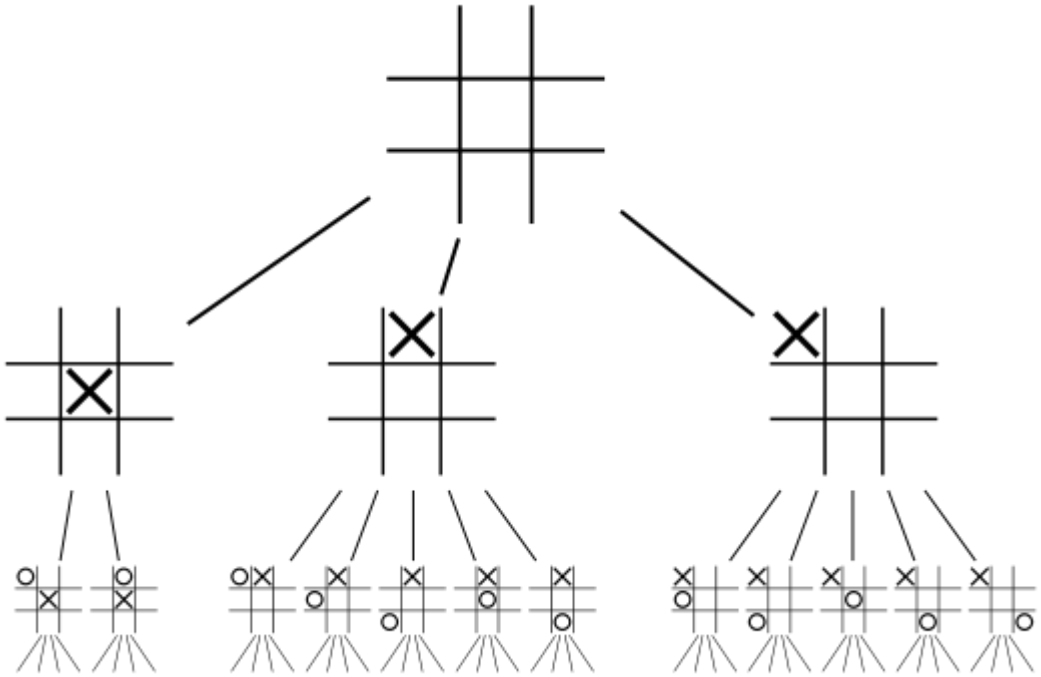
5.3. Spel

Beslissingen stellen zich dikwijls in de context van een ‘spel’. Je speelt tegen een tegenstander, hebt een aantal acties ter beschikking, speelt om beurten en er is een einddoel of eindscore bepaald. Of varianten hierop.

5.3.1. Oxo (‘Tic Tac Toe’ in ’t engels)

De zoekruimte kan ook voorgesteld worden door een *boom*. Neem het welbekende oxo-spel. De grootte van de zoekruimte is de faculteit van 9: 9x8x7x6x5x4x3x2x1 = 362880. De eerste speler heeft 9 vakjes voor zijn eerste zet, de tweede speler heeft dan nog maar 8 mogelijkheden, enzovoorts. Maar als je in rekening brengt dat verschillende volgordes tot eenzelfde situatie kunnen leiden, verkrijg je 255.168 verschillende spelverlopen. Door symmetrieën zijn sommige situaties in feite equivalent. Zo zijn er maar drie verschillende eerste zetten. Als tweede zet zijn er

maar 12 mogelijkheden [wikipedia]. Dat zijn er dus 6 maal minder dan de 72 (9x8) indien we hier geen rekening mee houden.



Merk op dat het eerste niveau van de zoekboom bepaald wordt door de zetten van speler 1, het volgend niveau door speler 2, dan komen weer de zetten van speler 1, enzovoorts.
De hele zoekboom kan door een computer worden opgesteld in redelijke tijd. Dit is bij schaken echter niet meer het geval.

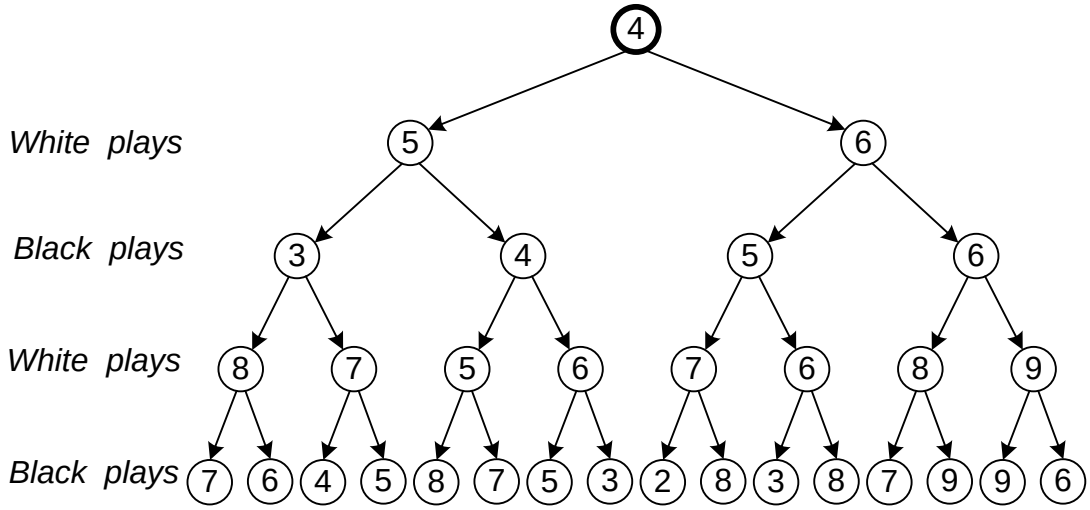
5.3.2. Schaken: minimax

Bij schaken wordt de totale zoekboom al snel te groot. Het is onmogelijk om alle sequenties uit te rekenen tot op het punt van winst (schaakmat). Daarom reken je de sequenties uit tot een bepaald moment in de toekomst en bepaal je dan of je situatie verbeterd is. Bijvoorbeeld doordat je een stuk van de tegenstander hebt weten te pakken. We gaan, zoals in 5.2.4, de zoekboom ontwikkelen tot een zekere diepte (de horizon) en via een score onze keuze bepalen. De kunst is om een goede score of *evaluatiefunctie* te definiëren. Voor schaken is deze meestal opgebouwd uit de volgende elementen:

1. Waarde van de stukken
2. Positie van de koning (of hij veilig staat en goed beschermd is)
3. Controle over het middenveld (de vier middelste vakjes)
4. Positie van de pionnen (bijvoorbeeld of ze aaneensluitend gepositioneerd zijn)
5. Positie van de stukken (bijvoorbeeld, hoeveel de bewegingsvrijheid van elk stuk)

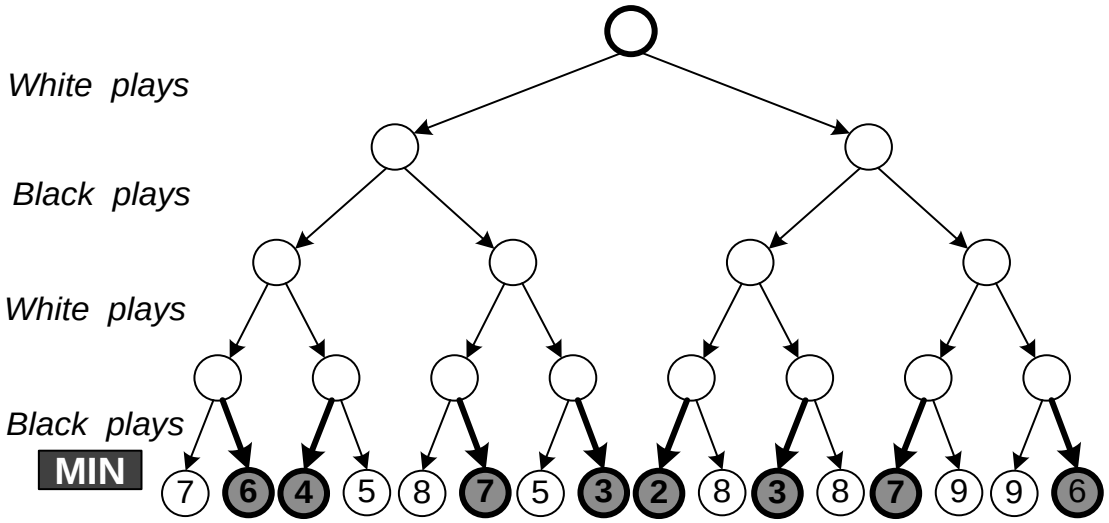
Eenmaal de zoekboom opgesteld en de scores berekend, bepalen we de beste zet door het **minimax**-algoritme.

Stel in een fictief spel dat wit aan zet is en zijn huidige score is vier. Wit heeft twee mogelijke zetten. Vervolgens heeft zwart ook twee mogelijkheden, enzovoorts. Voor het gemak noemen we deze mogelijkheden links en rechts. De volgende omgekeerde boom geeft de scores aan die bij elke zet horen. De score is die van wit, dat wil zeggen dat wit deze score moet maximaliseren, terwijl zwart gaat voor het minimaliseren van de score. Laat het duidelijk zijn dat enkel de score van de eindsituatie van belang is (bij schaken: schaakmat). De score die je toekent aan de tussenliggende situaties geven enkel een *inschatting* van de toestand. In de boom zie je dat de inschatting soms foutief is, zo lijkt wit aan de rechterkant van de boom een goede positie bereikt te hebben, maar zwart heeft in bepaalde gevallen een adequate zet voorhanden om de spelsituatie te doen keren.

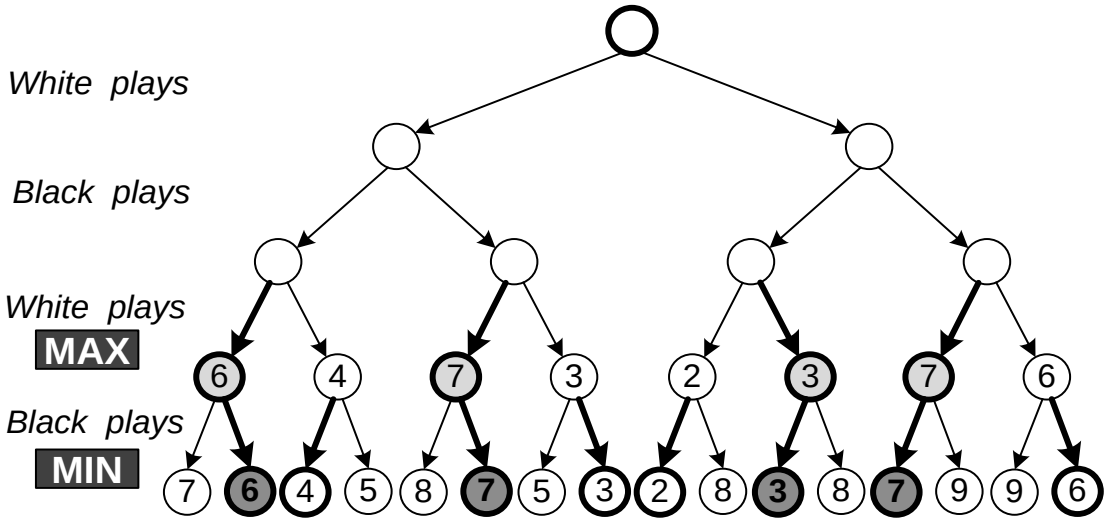


Op het eerste gezicht lijkt dus *rechts* de beste zet met een score van zes. Maar als we verder in de toekomst kijken is dit niet het geval. Als beide spelers maar 1 zet vooruitdenken (een vorm van greedy search): wit kiest voor rechts, dan zwart voor links, vervolgens wit voor links en tot slot zwart voor links. De eindscore voor wit is dan 2, wat de meest ongunstige situatie is.
Stel dat beide spelers 2 zetten vooruitdenken. Wit kiest nog steeds voor rechts, want zwart kiest dan voor links (score van 5). Als wit voor links zou kiezen zou zwart met links een score van 3 behalen. Dan kiest zwart voor links, want daar kan wit slechts 7 behalen. Vervolgens kiest wit voor rechts want zwart kan daar slechts een score van 3 behalen. De eindscore voor wit is dan 3, wat al beter is.

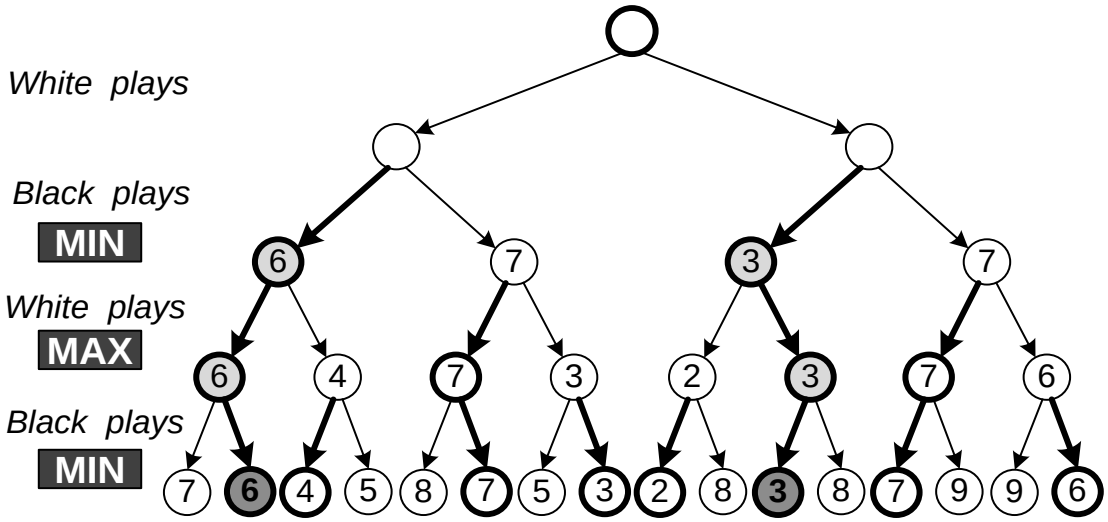
Laat ons nu vier zetten vooruitkijken. Dit lossen we op met het ***minimax***-algoritme. We beginnen onderaan de boom. Enkel score van de laagste nodes is van belang!



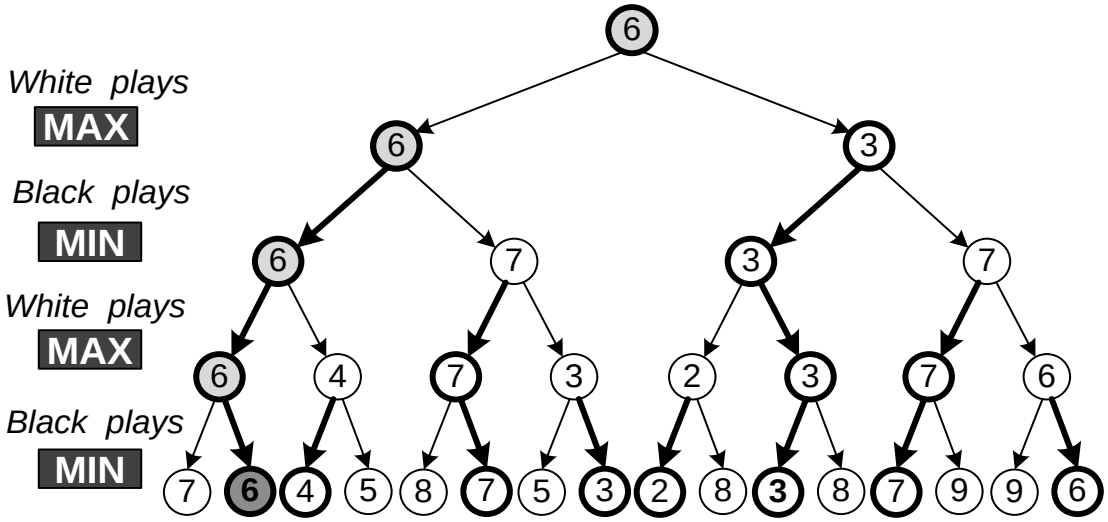
Zwart moet de score minimaliseren, hij zal dus de zet kiezen die naar de laagste score leidt. Hij neemt dus het minimum (**MIN**). Er zijn acht mogelijke scenario's waarin het spel verzeild kan zijn geraakt. Nu evalueren we de witte situatie. Wit wil maximaliseren en zal voor de vier posities (gekregen na de 2^e zet) de zet kiezen die leidt naar de maximale score (**MAX**).



Vanuit de vier beginsituaties (met score 3, 4, 5 & 6) kan wit een score van respectievelijk 6, 7, 3 en 7 krijgen. Vervolgens nemen we de plaats van zwart in (**MIN**).



Zwart kan de mogelijke winst van wit beperken tot 6 en 3. Wit heeft dus de keuze: linksaf slaan geeft een directe score van 5, maar een score van 6 op de lange termijn. Rechtsaf slaan geeft een directe score van 6, maar een veel slechtere score van 3 na vier zetten. Wit kiest dus voor de linkse actie.



Het minimax-algoritme gaat er dus van uit dat elke speler de meest rationele keuze maakt. Een andere keuze van zwart leidt dus telkens tot een slechtere keuze. Maar daar ga je dus niet van uit. Wit kan dus een score van '6' bereiken.
Als wit toch zou denken dat zwart niet al te doortastend is, zou wit voor *rechts* kunnen kiezen en erop gokken om bij één van de 9's rechtsonder in de boom proberen te komen. Dat is natuurlijk een risico, want zwart heeft de mogelijkheid om een score van 3 te bereiken zoals we reeds zagen.

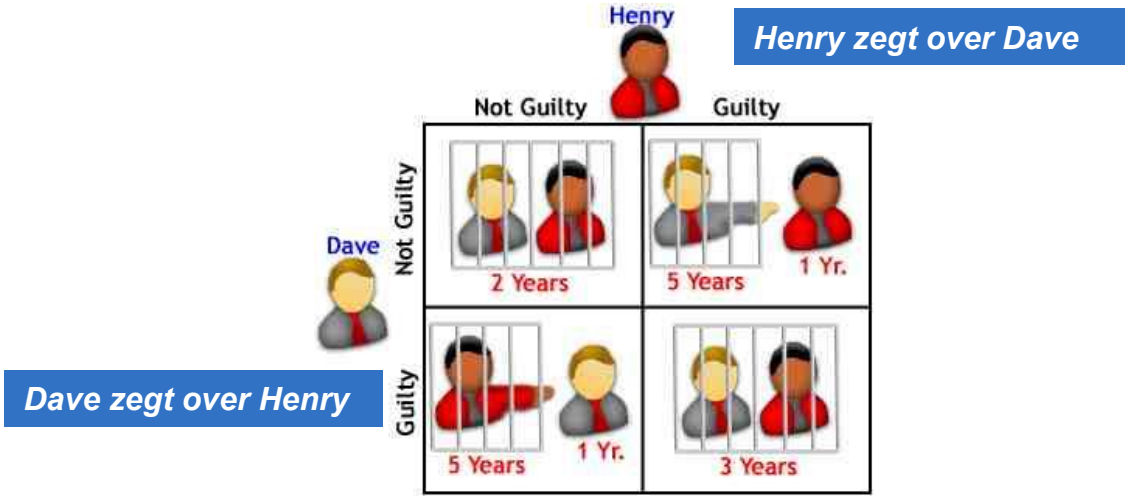
Dit algoritme begint er al 'slim' uit te zien. Moderne schaakprogramma's zijn hierop gebaseerd. Maar dit is niet hoe mensen denken. Het door de computer nabootsen van menselijke intelligentie bevindt zich nog maar in een beginstadium. Ik heb daar wat onderzoek naar gedaan en er een artikel over gepubliceerd: "An Alternative Approach for Playing Complex Games like Chess", *Annual machine learning conference of Belgium and The Netherlands* (BeneLearn 2008), Spa, Belgium 2008. Je kunt de paper vinden op mijn homepage: http://parallel.vub.ac.be/~jan. De studie geeft een mogelijke verklaring zonder een werkbare oplossing aan te bieden.

5.3.4. Non-zero sum game: Prisoner's dillema

Vorige voorbeelden gingen over **zero sum games**, wat de ene wint verliest de andere. In werkelijkheid bestaan er spelen waarin er **win-win** of **loose-loose** situaties bestaan. Daarin is het heel wat minder evident om de beste keuze te berekenen, want het hangt in grote mate af van de strategie van de tegenstander.

We spreken hier over een spel, maar dit kan ook slaan op 'echte' problemen in het leven. In de economie bijvoorbeeld, spreekt men over speltheorie om te begrijpen hoe alle spelers beslissingen nemen en zo het economisch verloop bepalen. Het voorstellen van de economie als 'spel' was een belangrijke doorbraak voor de economische wetenschappen. John Nash behaalde hiervoor de Nobelprijs voor economie in 1994. (Zie ook de film "A Beautiful Mind" uit 2001 met Russel Crowe die John Nash speelt.)

Het welbekende **prisoner's dilemma** illustreert één van de fundamenten van de speltheorie. Het voorbeeld gaat als volgt. Henry en Dave zijn opgepakt voor een misdrijf. De politie is er vrijwel zeker van dat het misdrijf gepleegd is door beide of door één van beide. Maar wie nu precies verantwoordelijk is, heeft de politie niet kunnen achterhalen. De rechter roept hen apart bij zich en vraagt hen *of de ander schuldig is*. Hun antwoord bepaalt welke gevangenisstraf de rechter zal opleggen. Dit is getoond in het volgende schema:



Als ze elkaar vrijpleiten, krijgen ze twee jaar, omdat de rechter toch inschat dat ze waarschijnlijk liegen. Als ze elkaar allebei beschuldigen is de rechter zeker en geeft hun drie jaar. Maar als Henry aangeeft dat Dave de verantwoordelijke is terwijl Dave Henry vrijpleit, dan wordt Dave de hoofdverantwoordelijke geacht en krijgt hij de maximumstraf van vijf jaar. Henry krijgt als medeplichtige slechts één jaar. Hetzelfde geldt in het omgekeerde geval.

Henry en Dave zijn zich bewust van dit schema en moeten een beslissing maken of ze hun kompaan aldanniet aan geven. Het beste voor beide is de ander niet aan te geven. Dan krijgen ze samen vier jaar. Maar als je zeker bent dat de ander jou niet verraadt,

zou je zelf wel verraden, want dan krijg je maar één jaar. Samen heb je dan wel zes jaar, maar je denkt egoïstisch en dus kies je voor verraad.

Dit dilemma komt in veel situaties voor in interactie met een ander, waarin je niet zeker bent dat hij eerlijk is. Het beste is dat je beide eerlijk bent (win-win situatie), maar als je de ander kunt bedriegen, ben je nog beter af. Behalve als hij jou ook bedriegt natuurlijk.

In werkelijkheid ga je in feite iteratief voor deze keuze te staan komen. Als je met dezelfde mensen moet samenwerken, heeft bedriegen niet veel zin, want dan zullen ze je niet meer vertrouwen bij een volgende gelegenheid. Dit wordt het *Iterative Prisoner's Dilemma* genoemd. Je speelt dit spel meermaals met dezelfde tegenspeler, waarbij je beide een keuze maakt en de uiteindelijke negatieve score gegeven wordt door het bovenstaande schema. Over dit 'spel' is al veel inkt gevloeid, zie bijvoorbeeld Wikipedia. Er worden zelfs competities gehouden waarbij twee spelers een match spelen door honderd keer de keuze te maken. Het blijkt dat de eenvoudige strategie "oog-om-oog, tand-om-tand" consequent hoog eindigt in de toernooien. Hier is de beslissingstabel voor deze strategie:

		Volgende zet	
		Samen-werking	**Verraad**
Laatste zet tegenstander	**Verraad**	0	1
	Samen-werking	1	0

5.4. Leren

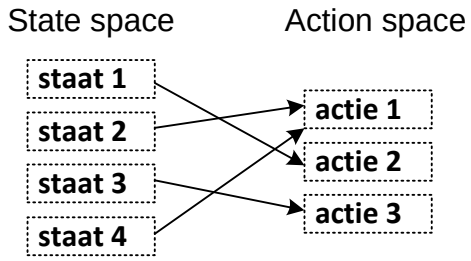
Een belangrijke tak van de Artificiële Intelligentie richt zich op het vinden van oplossingen door het leren uit opgedane ervaringen (**Machine Learning**). Dit kan effectief zijn in problemen waar de zoekruimte te groot is. Maar anderzijds kan het helpen in situaties waarin de juiste oplossing op voorhand niet te bepalen valt. Als je moet samenwerken of concurreren met andere personen/spelers, zoals in spelsituaties maar ook in het echte leven, hangt de optimale strategie af van het gedrag van die andere spelers. Je zult de strategie dus moeten aanpassen aan de medespeler of tegenstrever. Dit kun je doen tijdens het verloop van het spel, je optimaliseert je strategie **dynamisch**. De belangrijkste methode om dit te doen heet **reinforcement learning**.

5.4.1 Reinforcement learning: de theorie

Een probleem wordt gedefinieerd door

- de mogelijke toestanden
- de mogelijke acties
- de beloning/straf

Een strategie is dan een *mapping* van de staten op de acties (bepalen wat je doet in elke situatie):

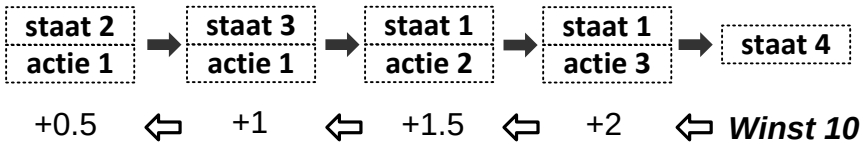


Deze *mapping* gaan we leren door een matrix van gewichten bij te houden voor elke staat-actie combinatie:

	actie 1	actie 2	actie 3
staat 1	4.1	3.8	6.7
staat 2	12.3	1.2	15.6
staat 3	8.7	10.2	0.2
staat 4	2.3	8.7	1.7

Het gewicht geeft de geleerde effectiviteit van elke actie aan voor die bepaalde staat. Initieel worden alle gewichten gelijk gekozen en in de *exploratiefase* worden de acties willekeurig gekozen. De gewichten worden dan aangepast afhankelijk of de keuze leidde tot een beloning of straf.
Vervolgens wordt de actie gekozen naargelang het gewicht. Hoe groter het gewicht, hoe groter de kans dat de actie gekozen wordt (*P(actie) ~ gewicht*).
Tot slot, in de exploitatiefase wordt de actie met het grootste gewicht gekozen. Al is het goed om soms andere acties te kiezen, want het spel kan dynamisch veranderen zodat op een bepaald moment andere acties toch optimaal blijken te zijn.

Neem het volgende voorbeeld waarin na 5 tijdstappen een winst van 10 gegenereerd is:



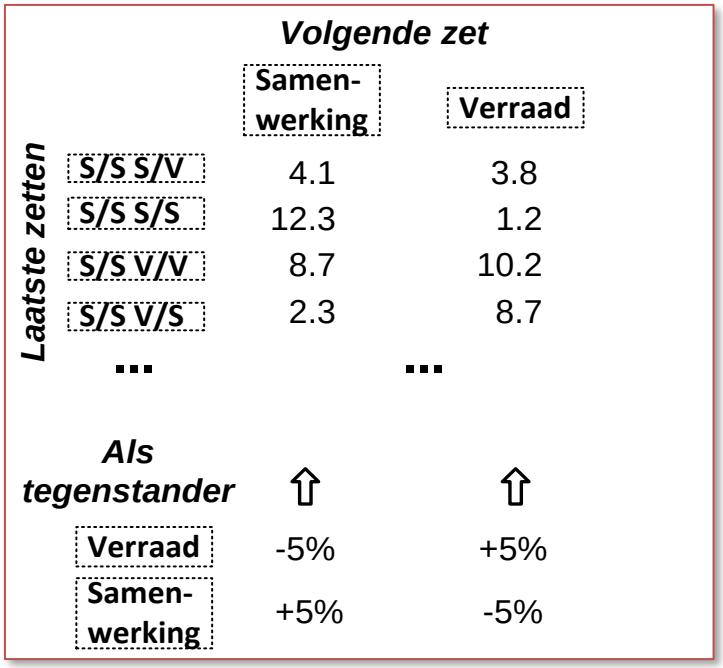
We verhogen de gewichten van de laatste acties. We voeren een ‘forgetting factor’ in: we voeren kleinere gewichtaanpassingen uit voor oudere acties omdat deze mogelijks niet bijdroegen tot de winst. Bij verlies doen we het omgekeerde, we verlagen de gewichten.

5.4.2. Leren toegepast op het iterative prisoner’s dilemma

Je kunt je strategie voor het iterative prisoner’s dilemma laten evolueren aan de hand van *reinforcement learning*. Je neemt bijvoorbeeld de twee laatste rondes als ‘staat’

en voor elk van deze zestien staten (vier maal vier) hou je een gewicht bij voor de twee mogelijke keuzes. Je houdt dus rekening met het verleden (al is het beperkt).

Vervolgens pas je tijdens het spel de gewichten aan als de tegenstander het omgekeerde doet. Door dit toe te passen tijdens je spel, pas je je aan veranderend gedrag van de tegenspeler.



5.4.3. Leren toegepast op blad-steen-schaar

Reinforcement learning is ideaal om *patronen* bij je tegenspeler te vinden zodat je hem kan verslaan. Als hij bepaalde keuzes meer maakt dan andere. Hier ook neem je *n* laatste keuzes in rekening. Stel dat we de laatste zet beschouwen dan bekomen we 27 gewichten:

Laatste zet	Volgende zet: blad	steen	schaar
bl – bl	4.1	3.8	3.6
bl - st	12.3	1.2	1.8
bl - sc	8.7	10.2	3.4
st – bl	2.3	8.7	7.8
st - st	4.1	3.8	3.6
st - sc	12.3	1.2	1.8
sc – bl	8.7	10.2	3.4
sc - st	2.3	8.7	7.8
sc - sc	4.1	3.8	3.6

Nu blijkt echter:

"Mensen kiezen niet toevallig", dat was het vertrekpunt van de onderzoekers van de Chinese universiteit. Winnaars blijven vaak bij hun winnende keuze en als ze verliezen, veranderen ze volgens een cyclisch proces. Als je daarop gaat anticiperen, zou je de winnende formule hebben voor spelletjes zoals blad-steen-schaar.

Met dit in gedacht kunnen we een patroon gebruiken voor de volgende zet en als volgt leren of we moeten veranderen of niet of best een willekeurige zet doen:

		Volgende zet		
		verander	**blijf**	**random**
Laatste zet	**winst**	14.1	3.8	3.6
	verlies	14.3	1.2	1.8
	gelijk	11.7	1.2	8.4

Nu moeten er 3x minder gewichten geleerd worden, wat sneller kan. Voor complexere spelen is het belangrijk zulke patronen te gebruiken omdat het aantal mogelijkheden snel stijgt.

5.4.4. Combineren minimax en leren: optimaliseren van de evaluatiefunctie

We kunnen minimax en leren combineren. Minimax wordt aangedreven door een evaluatiefunctie (5.53.2), maar de evaluatiefunctie zelf laten we evolueren. Meestal is de evaluatiefunctie een combinatie van verschillende factoren, zoals de vijf factoren voor het schaken. We kunnen het gewicht van elk van deze factoren *leren*. Het gewicht geeft aan hoeveel we elke factor meetellen in de globale score en duidt de relevantie van de factor aan. Als dit gewicht na het leren bijvoorbeeld zeer klein geworden is, duidt dit op een irrelevante factor.

5.4.5. Leren: combinatorische explosie van mogelijke situaties

Een nog niet beantwoorde vraag is of we het leeralgoritme kunnen toepassen op onze schuifpuzzel? Als we het algoritme letterlijk nemen, loopt het fout.

Dan moeten we voor elke toestand leren wat de beste actie is. De staat van de puzzel is gegeven door de positie van de acht stukjes. Voor elke staat willen we leren wat de beste actie is. Maar er zijn negen faculteit mogelijke configuraties. Het heeft dus niet veel zin om elke staat te gaan mappen op de acties. Zelfs als dit mogelijk is, gaan we een hele lange leerfase tegemoet om de goede acties te leren. Ook bij schaken zou je dit probleem tegen komen als je tracht alle staten te modelleren.

Wat we moeten doen is op een slimme manier ‘staten’ karakteriseren. Vervolgens moet we voor een klasse van staten de *mapping* naar acties te doen (in de vorm van mappen van hoofdstuk 9). Dit is ook wat de menselijke intelligentie doet: je veralgemeent je regels *staat -> actie*.

Je kunt de *mapping* nu zien als regels: *als de staat eigenschap x heeft, doe dan y*. Voor de schuifpuzzel wil je regels als “in het begin, ga voor de directe winst (optimaliseer de afstand)”. Naar het einde van het spel kun je de situaties zoals die in 5.2.4 proberen te karakteriseren (waar directe optimalisatie faalt) en voor die moeilijke situaties regels maken waarbij je een aantal acties voorziet. Dit lijkt nogal

omslachtig, zeker omdat we al een statische oplossing gevonden hebben in 5.2.5 door aanpassing van de scoredefinitie. Het volgend voorbeeld toont aan hoe we *reinforcement learning* kunnen toepassen op problemen waarin het aantal staten te groot is.

5.4.6. Leren van abstracte regels: Militaire strategie

Neem een militair spel waarin je moet beslissen hoe je je soldaatjes moet positioneren en hoe defensief/aanvallend ze zich op moeten stellen. Je wilt *abstracte regels* opstellen in de vorm van 'verdeel soldaatjes over verdediging en aanval', 'versterk de verdediging als thuisbasis onder vuur ligt', 'maak omtrekkende beweging om de vijand heen', 'stel je verdekt op', 'in de aanval, blijf bij elkaar, maar niet te dicht bij elkaar', … Deze regels kan je niet als dusdanig programmeren. Je moet ze uitdrukken in functie van de posities (coördinaten) van de eigen en vijandige soldaatjes. In feite beschrijf je **patronen** in je regels. Patronen programmeren is niet gemakkelijk.

Hier enkele pogingen om de patronen concreter te maken:

Staat:

- Afstand tot thuisbasis
- Afstand tot de vijf dichtstbijzijnde medesoldaatjes
- Afstand tot vijand
- Afstand vijand tot thuisbasis
- Verhouding soldaatjes aanvallend/verdedigend

Acties:

- Ga over tot verdediging/aanval
- Verwijder u van thuisbasis
- Vergroot/verklein afstand tot medesoldaatjes
- Vergroot/verklein afstand tot vijand
- Maak een omtrekkende beweging
- Exploreer (beweeg in willekeurige richting)

Omdat je de strategie van de tegenstander niet kent, kan je je strategie optimaliseren met *reinforcement learning* door enkele parameters van je strategie aan te passen. Bijvoorbeeld: verhouding tussen aanvallende en verdedigende soldaatjes, aanvalslust, …

5.5. Conclusies

In dit hoofdstuk zijn we gestoten op de grenzen van de huidige computeralgoritmes. *Brute force search* is goed te doen door een computer, het is ook gemakkelijk te implementeren. Voor de toepassing van meer geavanceerde technieken bleek nog veel *engineering* noodzakelijk te zijn; het inzetten van de menselijke intelligentie. De mens moet nog altijd goed nadenken over het design van de algoritmen. Zo is een slimme definitie van de score voor de schuifpuzzel noodzakelijk om af te kunnen stappen van brute force. Bij leren zijn de keuzes zelfs nog belangrijker. Patronen programmeren blijkt helemaal niet simpel. Laat staan dat de computer zelf in staat is deze designbeslissingen te nemen. *Artificiële intelligentie* oftewel een vervanger van het menselijk brein is er nog niet…

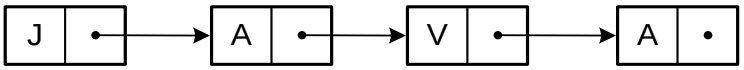
Hoofdstuk 6

Gelinkte lijsten

Arrays zijn niet echt efficiënt als je veelvuldig elementen wilt toevoegen en verwijderen. Stel dat we een gesorteerde lijst van namen bijhouden. Als we een naam willen toevoegen, moet dit op de juiste plek gebeuren. Voor zo'n *insert* moet er een plaatsje vrijgemaakt worden. Als je dit met een array doet, moeten alle elementen erna naar achter worden opgeschoven. Omgekeerd geldt dat als je een element verwijdert, de elementen erna weer naar voor opschuiven om het lege vakje op te vullen.
Een ander voorbeeld is een tekstdocument. Bij het aanmaken en bewerken van zo'n document ben je steeds tekst aan het toevoegen, invoegen, verplaatsen, verwijderen, kopiëren en dergelijke. De tekst ('een rij letters') moet ergens opgeslagen worden. Dit kan met een array, maar zoals gezegd, is dit heel inefficiënt. Hiervoor bestaat een veel flexibelere oplossing: *de gelinkte lijst*. De code van dit hoofdstuk is te vinden in **package *algoritmen***.

6.1. Definitie gelinkte lijst

Het principe van de gelinkte lijst is het bijhouden van de gegevens in een flexibel 'treintje', waarbij elk element in een wagonnetje gestopt wordt. Dit laat toe om wagonnetjes toe te voegen, te verplaatsen of te verwijderen. Bijvoorbeeld "java":



De truc is om aan elk element een verwijzing naar het volgende element toe te voegen. Zoals we zo dadelijk zullen zien, laat dit ons toe de lijst eenvoudig te kunnen manipuleren. De definitie van een wagonnetje, genoemd *Link*, is als volgt:

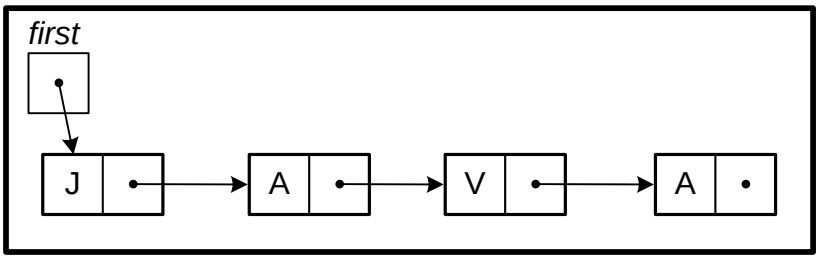
```
class Link<T>{
	T data;
	Link<T> next;

	Link(T data){
		this.data = data;
		this.next = null;
	}
	Link(T data, Link<T> next){
		this.data = data;
		this.next = next;
	}
}
```

Een *Link* heeft twee attributen: één om de gegevens bij te houden, de *data*, en één om het volgende wagonnetje aan te duiden, de *link*. Met deze laatste koppelen we de link aan het volgende element. We voorzien twee constructors, één met enkel data als parameter, en één met ook het volgend element. Als er *next* op *null* staat, betekent dit dat er geen volgend element is, met andere woorden: het einde van de lijst. *Null*, ook *null-pointer* genoemd, slaat op geen verwijzing (naar 'niets' oftewel 'null').

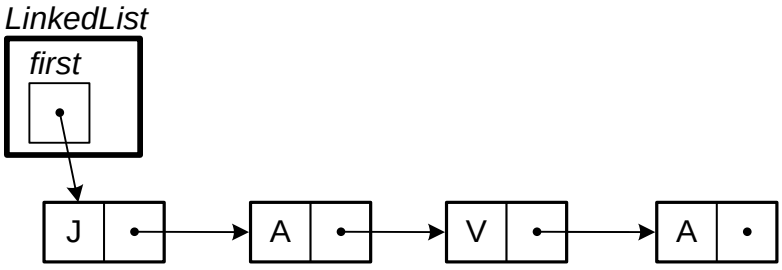
We gaan onze lijst als één geheel beschouwen en stoppen dit in een object van klasse *LinkedList*. Hiervoor moet het object een verwijzing naar het eerste element bijhouden. Dit doen we met *first*. De klassedefinitie wordt dan:

```
public class LinkedList <T> {

    class Link<T>{
          T data;
          Link<T> next;

          Link(T data){
                this.data = data;
                this.next = null;
          }
          Link(T data, Link<T> next){
                this.data = data;
                this.next = next;
          }
    }

    Link<T> first;

    public LinkedList(){
          first = null;
    }
}
```

We hebben de definitie van *Link* in de klasse *LinkedList* gestopt, omdat we *Link* enkel binnen de klasse gaan gebruiken. Dit wordt een **inner class** genoemd, enkel bruikbaar binnen de omhullende klasse. Onze datastructuur ziet er dus als volgt uit:

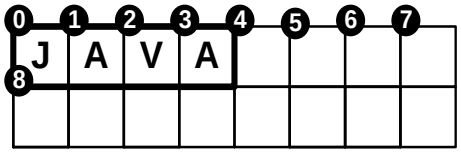


Deze figuur is eigenlijk niet helemaal correct. Als we een *LinkedList*-object aanmaken, krijgen we een object met één attribuut, namelijk *first*. Als we elementen toevoegen met behulp van Link-objecten wordt nieuw geheugen vrijgemaakt, in feite buiten het *LinkedList*-object.

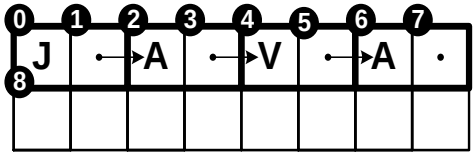


We moeten zelf een referentie bijhouden naar het eerste *Link*-object. Zouden we dit niet doen, dan zitten de objecten wel ergens in het geheugen, maar we weten niet waar! Dit soort referenties wordt ook wel een *pointer* genoemd. Als de lijst leeg is, staat *first* op *null*.

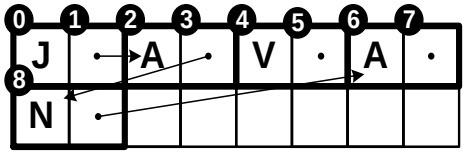
Laat ons nu bestuderen hoe de gegevens in het geheugen van de computer staan. Een array (bvb string) wordt lineair opgeslagen:



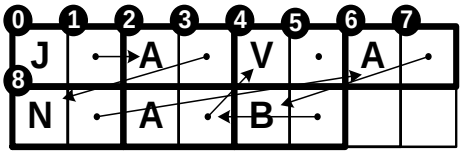
Bij een gelinkte lijst hebben we meer geheugen nodig:



Maar we zijn flexibel en kunnen de ‘V’ vervangen door een ‘N’ bijvoorbeeld:



Het is dus belangrijk te begrijpen dat de elementen niet achter elkaar in het geheugen zullen zitten zoals bij een array. Na wat manipulaties bekomt men bijvoorbeeld:



6.2. Java’s LinkedList

Java biedt een *LinkedList* klasse. Hieronder vind je de voornaamste methoden:

Constructor Summary

LinkedList() Constructs an empty list.
LinkedList(Collection<? extends E> c) Constructs a list containing the elements of the specified collection, in the order they are returned by the collection's iterator.

Method Summary

boolean	**add**(E e) Appends the specified element to the end of this list.
void	**add**(int index, E element) Inserts the specified element at the specified position in this list.

void	**addFirst**(E e) Inserts the specified element at the beginning of this list.
void	**addLast**(E e) Appends the specified element to the end of this list.
void	**clear**() Removes all of the elements from this list.
boolean	**contains**(Object o) Returns `true` if this list contains the specified element.
E	**get**(int index) Returns the element at the specified position in this list.
E	**getFirst**() Returns the first element in this list.
E	**getLast**() Returns the last element in this list.
int	**indexOf**(Object o) Returns the index of the first occurrence of the specified element in this list, or -1 if this list does not contain the element.
boolean	**offer**(E e) Adds the specified element as the tail (last element) of this list.
E	**poll**() Retrieves and removes the head (first element) of this list
E	**pollFirst**() Retrieves and removes the first element of this list, or returns `null` if this list is empty.
E	**pollLast**() Retrieves and removes the last element of this list, or returns `null` if this list is empty.
E	**remove**() Retrieves and removes the head (first element) of this list.
E	**remove**(int index) Removes the element at the specified position in this list.
E	**removeFirst**() Removes and returns the first element from this list.
E	**removeLast**() Removes and returns the last element from this list.
E	**set**(int index, E element) Replaces the element at the specified position in this list with the specified element.
int	**size**() Returns the number of elements in this list.

Als je de methoden van Java's *LinkedList* bekijkt, zie je de basisoperaties. Met deze operaties kun je alle bewerkingen definiëren. Maar je zult ook zien dat je geen toegang hebt tot de interne datastructuur. Je krijgt de 'wagonnetjes' niet te zien!
De methodes laten toe om de rij van elementen te manipuleren zonder dat je de interne keuken te zien krijgt. De wagonnetjes worden door Java gebruikt om de data bij te houden,

zonder dat je daar zelf toegang toe hebt. Dat is maar goed ook, want dat is veel gemakkelijker: je vertelt wat er moet gebeuren en de rest gebeurt achter de schermen.

Op deze belangrijke les komen we nog terug!

6.3. Bewerkingen op gelinkte lijsten

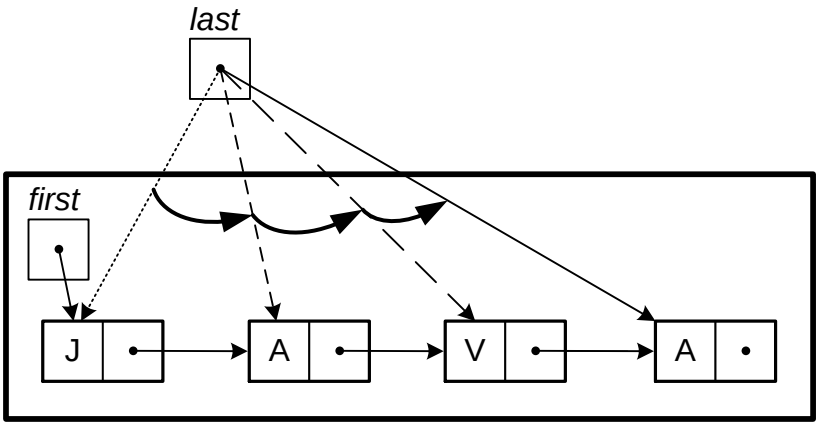
Ondanks dat we over een 'perfecte' *LinkedList* beschikken (die van Java), zullen we toch zelf de operaties op gelinkte lijsten implementeren omdat dit een goede oefening is om jullie programmeervaardigheden aan te scherpen. Vele trucs die je hier aanleert, zullen later nog van pas komen bij het maken van je eigen algoritmen. We bespreken hier de voornaamste operaties die je nodig hebt om met een gelinkte lijst te werken.

6.3.1 Toevoegen van een element aan het einde (*append*).

Dit is code:

```
public void append(T data){
        Link<T> el = new Link<T>(data);
        if (first == null){
                first = el;
        } else {
                // zoek laatste link
                Link<T> last = first;
                while (last.next != null)
                        last = last.next;
                last.next = el;
        }
}
```

Voor het nieuwe element wordt een nieuw *Link*-object aangemaakt. Deze moet aan de laatste wagon gekoppeld worden, wat er op neerkomt dat het laatste element moet wijzen naar dit nieuwe element. Hiervoor moeten we op zoek gaan naar de laatste link. Wel moeten we er eerst rekening mee houden dat de lijst nog leeg kan zijn. In dat geval wordt de nieuwe link *first*. In het andere geval beginnen we bij de eerste link en springen we via alle wagonnetjes tot de laatste. Dit doen we met behulp van een hulpvariabele, *last*.



6.3.2 Printen van lijst

```
public String toString(){
	Link<T> link = first;
	String str = "[";
	boolean eerste = true;
	while (link != null){
		if (eerste)
			eerste = false;
		else
			str+=", ";
		str += link.data;
		link = link.next; // ga naar volgende
	}
	str+="]";
	return str;
}
```

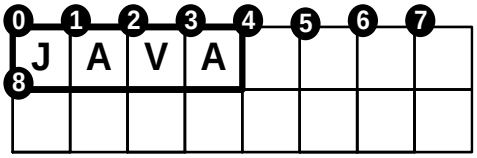
Ook voor het printen gebruiken we een hulpvariabele, *link*, die de gehele lijst afloopt. We testen of *link* niet *null* is. Als de lijst leeg zou zijn, zouden we hier al stoppen. We willen een komma tussen alle elementen. De eerste komma moet dus voor het tweede element komen. Daarvoor gebruiken we de boolean *eerste* die test of we nog met het eerste element bezig zijn.

Vraag**:** wat gebeurt er met de komma's als we het volgende zouden proberen (dus zonder die boolean)?

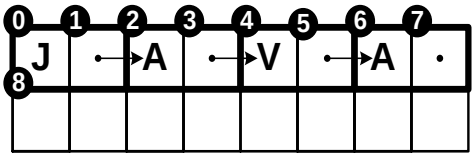
```
	while (link != null){
		str += link.data;
		str+=", ";
		link = link.next; // ga naar volgende
	}
```

6.3.3 Vinden van een bepaald element (op index)

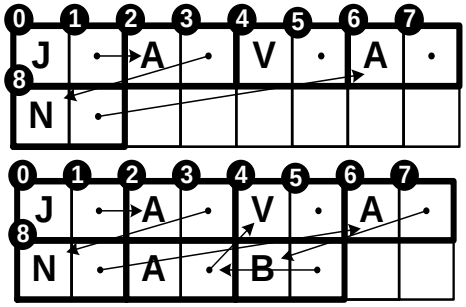
Het is belangrijk te begrijpen dat bij een array, het *i*-de element zich *i* vakjes verder bevindt dan het begin:



Het 1000000e element vinden is het beginadres + 1000000. Voor een gelinkte lijst is dit anders. Idealiter bevinden alle delen zich achter elkaar in het geheugen:



Dit kan zo zijn in het begin (al is dat ook niet gegarandeerd), maar nadat de lijst enkele veranderingen heeft ondergaan, zullen de links zich op willekeurige plaatsen in het geheugen bevinden:



De enige manier om het *i*-de element te vinden is het treintje aflopen... Het 1000000e element vinden kost 1000000 stappen, tegenover slechts 1 optelling voor een array!

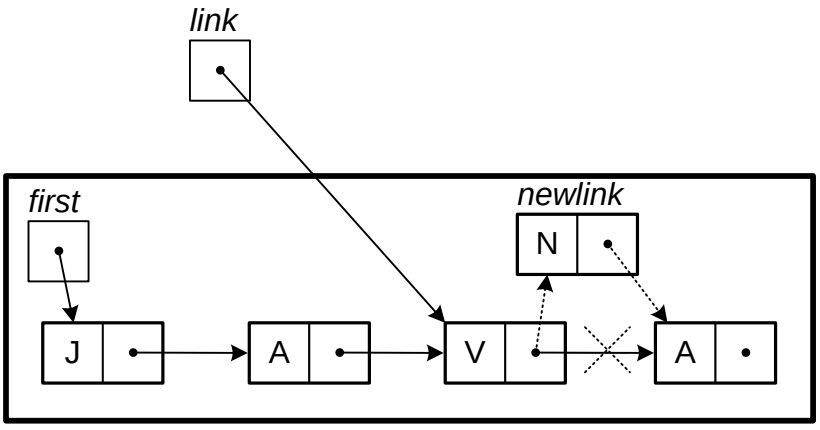
```
public T get(int index){
        int i=0;
        Link<T> link = first;
        while (link != null && i < index){
                link = link.next;
                i++;
        }
        if (link == null)
                return null;
        else
                return link.data;
  }
```

We lopen het treintje weer af en we tellen hoe ver we zitten. We gaan hiermee verder zolang onze teller kleiner is dan het gewenste aantal. We moeten ons er ook van bewust zijn dat er misschien minder elementen zijn, waardoor we dus met een *null* opgezadeld zitten. Dan moeten we ook stoppen, en geven we een *null* terug.

6.3.4 Inlassen van een bepaald element

```
	public void insert(T data, int index){
		Link<T> newlink = new Link<T>(data);
		if (index == 0){
			// wordt eerste element
			newlink.next = first;
			first = newlink;
		} else {
			int i=0;
			Link<T> link = first;
			while (link != null && i < index - 1){
				link = link.next;
				i++;
			}
			if (link == null)
				throw new IllegalArgumentException("Insert at
"+index+" impossible; list has only "+i+" elements.");
			newlink.next = link.next;
			link.next = newlink;
		}
	}
```

Nu gaan we op zoek naar de link vóór de plaats waar het nieuwe element moet komen. In onderstaande figuur wensen we 'n' op de 3e plaats. Let op: we beginnen vanaf nul te tellen, de letter 'v' heeft dus index 2. We moeten ook incalculeren dat de gebruiker een ongeldige index kan meegeven. In dat geval komen we aan het einde van de lijst voor we de juiste link gevonden hebben. In het OK-geval hebben we na de *while*-lus de volgende situatie:



In stippellijn is aangeduid wat er moet gebeuren. De link voor de *insert*-positie moet naar de nieuwe link wijzen en de nieuwe link naar de volgende. Merk op dat je eerst

```
	newlink.next = link.next;
```

en dan pas

```
	link.next = newlink;
```

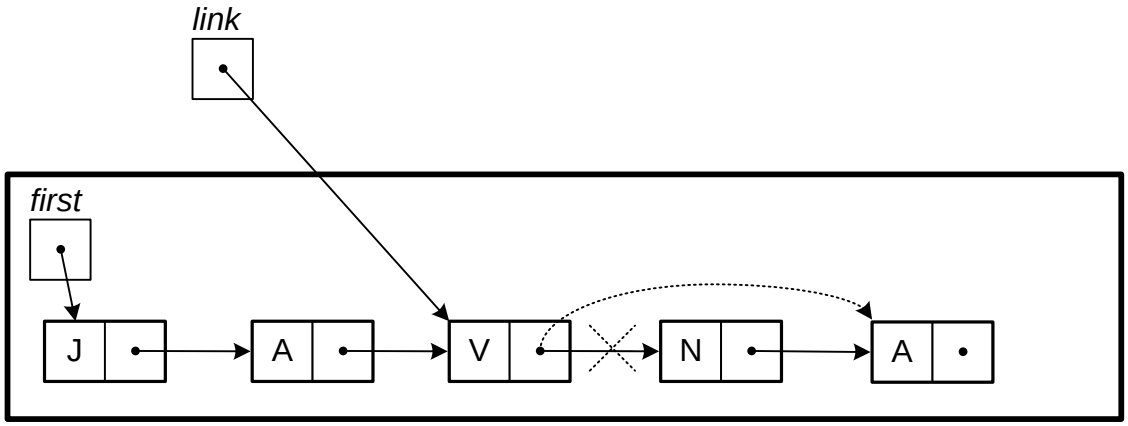
Wat gebeurt er als we beide instructies in omgekeerde volgorde zouden uitvoeren?
Merk bovendien op dat het programma ook werkt als het vierde element (hier de 'a') niet bestaat.

6.3.5 Verwijderen van een record uit een lineaire lijst

Met de `public void remove(int index)` methode moeten we het element met de gegeven index verwijderen.

```
1       public void remove(int index){
2             if (index == 0){
3                   if (first == null)
4                         throw new
5   IllegalArgumentException("Removal of first element impossible;
6   list has no elements.");
7                   first = first.next;
8             } else {
9                   int i=0;
10                  Link<T> link = first;
11                  while (link != null && i < index - 1){
12                        link = link.next;
13                        i++;
14                  }
15                  if (link == null || link.next == null)
16                        throw new
17  IllegalArgumentException("Removal of element "+index+" impossible;
18  list has only "+(i+1)+" elements.");
19                  link.next = link.next.next;
                  }
        }
```

We gaan eerst weer op zoek naar de link vóór het te verwijderen element.



Om het element te verwijderen, moet het element ervoor verwijzen naar het element erachter. Dit gebeurt met `link.next = link.next.next;`
Merk op dat het element met de gegeven index nu wel moet bestaan (die willen we immers verwijderen). Daarvoor moeten we testen op: `(link == null || link.next == null)`

6.3.6 Oefeningen

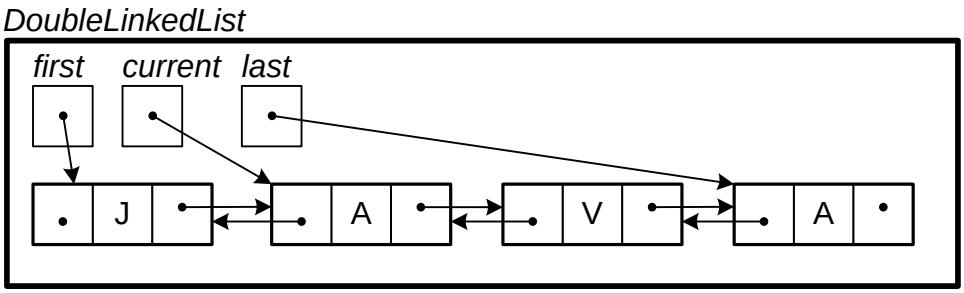
- Implementeer een methode voor het verwijderen van alle elementen tussen i en j (j niet meegerekend).
- Implementeer een methode om twee elementen op posities i en j te verwisselen.
- Implementeer een methode die het aantal elementen van de gelinkte lijst telt.
- Implementeer een methode die een letter toevoegt op zijn alfabetische plaats (veronderstellend dat alle bestaande elementen reeds in volgorde staan)

6.4. De performantie van een gelinkte lijst

De array is de moeder van alle lijsten. Om een lijst van *n* elementen op te slaan, gebruikt de array *n* geheugenplaatsen. Voor een gelinkte lijst moet je de geheugenplaatsen voor de pointers erbij tellen. In het geval van een gelinkte lijst van karakters betekent dit dubbel zoveel geheugenplaatsen.
Zoals geargumenteerd, is een gelinkte lijst flexibeler. Het toevoegen of verwijderen van elementen verloopt gemakkelijker dan bij arrays, waar je alle elementen zult moeten opschuiven.
Maar een gelinkte lijst heeft ook nadelen. Als je een bepaald element zoekt, moet de hele lijst afgelopen worden; je moet dan vanaf de eerste link via alle tussenliggende links gaan. Dit kan voor lange lijsten heel rekenintensief zijn. Hier is een array dan weer in het voordeel, wil je het *n*de element, dan kan de array onmiddellijk het juiste geheugenadres berekenen, namelijk *startadres + n*. Met een array kan elk element zonder poespas worden gevonden. Bij een gelinkte lijst moet je hem doorlopen, zelfs als hij gesorteerd is!
Daar gaan we iets aan doen in hoofdstuk 7.

6.5. Double Linked List en 'current'-pointer

Een kleine uitbreiding van de gewone gelinkte lijst is de dubbelgelinkte lijst. Hier zal elke node ook een pointer naar het vorige element bijhouden (previous). Zo kunnen we niet enkel van voor naar achter door de lijst lopen, maar ook van achter naar voor. Hiervoor houden we ook een pointer naar het laatste element bij. Als je ziet dat Java's LinkedList klasse toelaat om het laatste element te manipuleren, dan weet je dat hij inderdaad een pointer naar het laatste element zal bijhouden:



Hoofdstuk 7

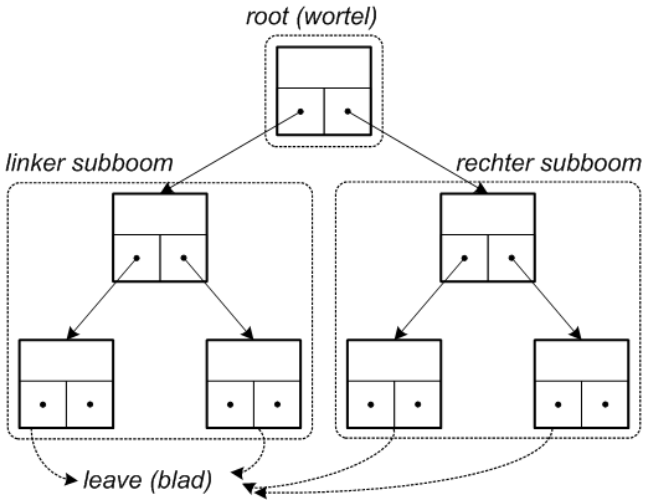
Geordende binaire bomen

Eerder bespraken we hoe gelinkte lijsten een zeer flexibele structuur geven. Het zoeken in een gelinkte lijst was echter niet optimaal, aangezien je enkel de lijst van links naar rechts kunt doorlopen en alle elementen moet passeren. We zagen ook dat ‘binair’ zoeken het snelste was in een geordende lijst: je kijkt halverwege of je element links of rechts zit en gaat verder met dat deel. Aldus kun je telkens de helft van de elementen elimineren. De zoektijd wordt dan $\log_2 n$. Merk op dat hetzelfde idee van halveren terug te vinden is bij de snelle sorteeralgoritmen.
Een binaire boom is gebaseerd op beide principes, het combineert de flexibiliteit van een gelinkte lijst met binaire zoeksnelheid. Een toepassing is dus een woordenboek of elke lijst die je gesorteerd wilt bijhouden. Boomstructuren zijn we reeds tegengekomen in hoofdstuk 5 als zoekbomen. Daar gaat het niet om gesorteerde data, maar om de structuur van de gegevens zelf die vragen om een ‘boom’.

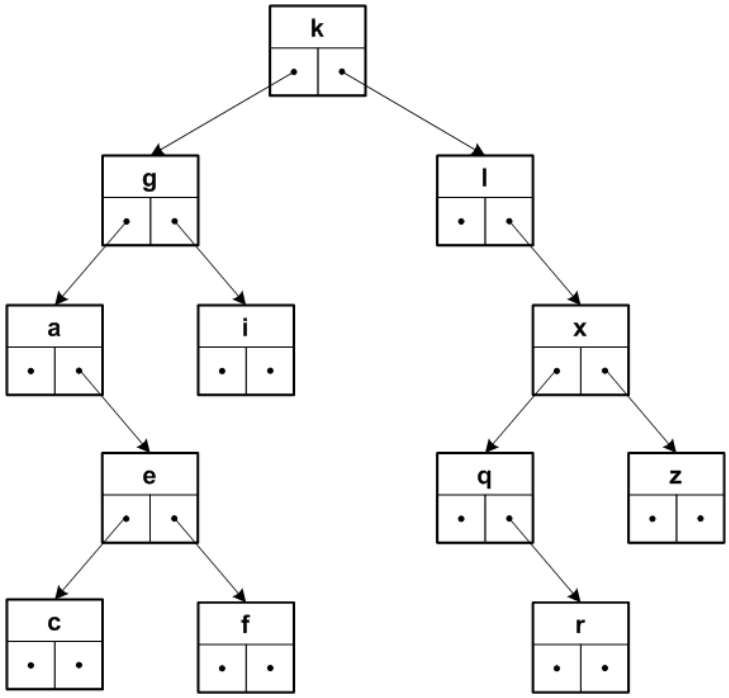
De code van dit hoofdstuk is terug te vinden in de **package *algoritmen***.

7.1 Definitie

De basiselementen van een binaire boom zijn weergegeven in de figuur. Het centrale element langs waar men toegang krijgt tot de boom, noemt men de **root** (**wortel**). De twee bomen waarnaar de twee pointers van de wortel wijzen, noemt men respectievelijk "linkersubboom" en "rechtersubboom". De subbomen zijn uiteraard zelf ook gewone bomen met dezelfde eigenschappen als de boom waarvan ze deel uitmaken. Elementen helemaal onderaan in een subboom die geen subboom bevatten, noemt men "bladeren" (*leaves)*. Deze plantkundige definities kunnen eigenaardig lijken; de verbeeldingrijke lezer mag eruit afleiden dat informatici ondersteboven leven.



Men spreekt van een **geordende binaire boom** wanneer alle elementen een "sleutelveld" bevatten voor welke een orderelatie kan worden gedefinieerd en wanneer elk element de wortel is van een boom met kleinere elementen in de linkersubboom en grotere elementen in de rechtersubboom. De figuur toont een dergelijke boom waarin de sleutels letters van het alfabet zijn.



Geordende binaire bomen worden veel aangewend voor het opbouwen en bijhouden van gegevens die men veelvuldig en snel wil raadplegen (woordenboeken), want de zoektijd in een dergelijke boom is logaritmisch voor een gebalanceerde boom, zoals we later zullen zien. Dit is het gevolg van het halveren van de overblijvende zoekruimte bij het bezoeken van elk element. Het kan dus een traditioneel woordenboek zijn maar ook de lijst van de studenten van een universiteit of de catalogus van een winkel.

7.2 De objecten van een geordende binaire boom

De basisbouwsteen is de *node*.

```
class Node<T>{
      T data;
      Node<T> left, right;

      Node(T data){
            this.data = data;
            this.left = null;
            this.right = null;
      }
}
```

Met behulp van deze generieke node kunnen we een binaire boom bouwen die elementen van het type *T* opslaat. Het parametriseren van een datatype (*generics* in Java) bespraken we reeds onder *ArrayList* en *FifoQueue*. De boom moet enkel de root kennen. Vanuit de root heeft ze toegang tot alle andere nodes.

```
public class BinaryTree<T> {
	Node<T> root;
	Comparator<T> comparator;
	public BinaryTree(Comparator<T> comparator){
		this.comparator = comparator;
		root = null;
	}
```

We verwachten van de gebruiker ook een *Comparator*. Deze zal ons vertellen hoe de elementen te ordenen. Het *comparator*-object zal gebruikt worden om twee objecten te vergelijken. *Comparator* is een *interface* met twee methodes gedefinieerd voor een type **T**:

java.util

Interface Comparator<T>

Method Summary	
int	**compare**(T o1, T o2) Compares its two arguments for order. Returns a negative integer, zero, or a positive integer as the first argument is less than, equal to, or greater than the second.
boolean	**equals**(Object obj) Indicates whether some other object is "equal to" this comparator.

De *compare*-methode vergelijkt twee objecten en geeft een negatieve waarde als het eerste object kleiner is dan het tweede, 0 als ze gelijk zijn en een positieve waarde als het tweede object kleiner is. De *equals*-methode mag je negeren. We komen hier nog op terug in het volgend hoofdstuk.

We moeten met de constructor een *comparator*-object meegeven, een object van een klasse die de *Comparator*-klasse implementeert. We definiëren dus eerst een eigen *Comparator*-klasse, als volgt:

```
class IntegerComparator implements Comparator<Integer> {
	@Override
	public int compare(Integer val1, Integer val2) {
		return val1 - val2;
	}
}
```

Het aanmaken van de boom wordt dan:

```
	BinaryTree<Integer> tree = new BinaryTree<Integer>(new
IntegerComparator());
```

Nog een opmerking: '`@Override`' geeft aan dat we de methode overschrijven of implementeren. Dit noemen we een **annotatie** en is een vorm van documenteren.

Merk op dat dit laatste nog korter kan geschreven met behulp van een ***anonymous class*** (ook ***instant class*** genoemd):

```
BinaryTree<Integer> tree =
   new BinaryTree<Integer>(
      new Comparator<Integer>() {
         @Override
         public int compare(Integer val1, Integer val2) {
            return val1 - val2;
         }
       }
  );
```

We creëren een object van de gevraagde interface met *new* en implementeren tezelfdertijd de gevraagde methodes van de interface. Zo moeten we de klasse `IntegerComparator` niet apart definiëren, wat onnodig is omdat we deze klasse nergens anders nodig hebben. Vlotte Javaprogrammeurs maken hier regelmatig gebruik van.

Nog een voorbeeld van een comparator, nu voor het ordenen van studenten:

```
class StudentComparator implements Comparator<Student>{
  @Override
  public int compare(Student student1, Student student2) {
    int ordeNaam = student1.naam.compareTo(student2.naam);
    if (ordeNaam == 0) // als beide dezelfde naam
     return student1.voornaam.compareTo(student2.voornaam);
    else
     return ordeNaam;
   }
}
```

7.3 Het zoeken van een element in een geordende binaire boom

Het zoeken gebeurt ook recursief. We maken gebruik van de comparator om enerzijds tijdens het dalen langs de goede kant van de boom af te dalen, en anderzijds om te checken of het element gevonden werd (*compare* geeft dan 0).

```
      public boolean contains(T object){
            return find(root, object);
      }
      private boolean find(Node<T> current, T object){
        if (current == null)
            return false;
        else if (comparator.compare(current.data, object) == 0)
            return true;
        else if (comparator.compare(object, current.data) < 0)
            return find(current.left, object);
        else
            return find(current.right, object);
```

```
        }
```

De recursieve methode *find* is een vorm van *staartrecursie*: er wordt maar één recursieve oproep uitgevoerd. Daarom kan deze ook zonder recursie geschreven kan worden, maar met een *while*:

```
public boolean containsWithoutRecursion(T object){
      Node<T> current = root;

      while  (current != null){
            if (comparator.compare(current.data, object) == 0)
                  return true; // gevonden!
            else if (comparator.compare(object, current.data) < 0)
                  current = current.left;
            else
                  current = current.right;
      }
      return false; // niet gevonden
}
```

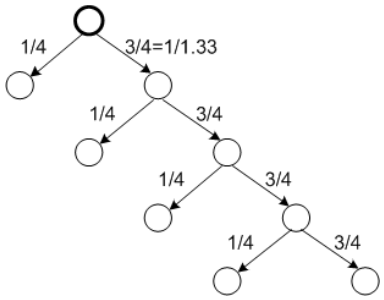
7.4 De zoektijd

Het zoeken van een element in een ongesorteerde lijst (bvb array) van n elementen is gemiddeld $n/2$. Voor een gesorteerde lijst of binaire boom is dit $\log_2 n$. Je deelt het aantal elementen steeds door 2: n => $n/2$ => $n/4$ … => 4 => 2 => 1. Dus: $1.2.2.2 \ldots 2 = 2^x = n$ met x het aantal zoekstappen.

Dus:

$$aantalZoekstappen = \log_2 n$$

Wat gebeurt er bij een ongebalanceerde boom waarbij de nodes niet evenredig verdeeld zijn? Stel dat bij elke node 1/4 van de nodes links liggen en 3/4 rechts:



Als de te zoeken node links ligt zullen we hem sneller vinden dan als hij rechts ligt. In het beste geval ligt de node helemaal links, waarbij we n steeds door 4 delen. Dit geeft:

$$aantalZoekstappen = \log_4 n$$

In het slechtste geval moeten we de boom rechts aflopen en blijven er telkens 3/4=1/1.33 van de nodes over. Dit geeft

$$aantalZoekstappen = \log_{1.33} n$$

In het algemene geval ligt het aantal zoekstappen ergens tussen beide. Het ene is de *best case*, de andere de *worst case*.
Laat ons beide uitersten vergelijken met de zoektijd van een gebalanceerde boom:

$$stappenRatio = \frac{\log_d n}{\log_2 n} = \frac{1}{\log_2 d}$$

Want:

$$\log_d n = \frac{\log_2 n}{\log_2 d}$$

Dit geeft:

$$stappenRatio(best\ case) = \frac{1}{\log_2 4} = 1/2$$

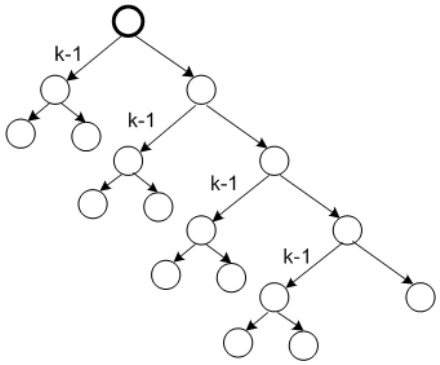
$$stappenRatio(worst\ case) = \frac{1}{\log_2 1.33} = \frac{1}{0.411} = 2.43$$

Om het effect van deze resultaten goed te begrijpen berekenen we ze voor verschillende grootte-ordes (k=100000):

	$\log_2 n$	$\log_4 n$	$\log_{1.33} n$	$\frac{n}{2k} + \log_2 k$	$n/2$
10	3,3	1,7	8,1	16,6	5
100	6,6	3,3	16,1	16,6	50
1000	10,0	5,0	24,2	16,6	500
10000	13,3	6,6	32,3	16,7	5000
100000	16,6	8,3	40,0	17,1	50000
1000000	19,9	10,0	48,4	21,6	500000
10000000	23,3	11,6	56,5	66,6	5000000
100000000	26,6	13,3	64,6	516,6	50000000
1000000000	29,9	14,9	72,7	5016,6	500000000

Het is duidelijk dat een logaritmische zoektijd drastisch sneller is dan een lineaire zoektijd (laatste kolom). Zelfs als de boom niet helemaal gebalanceerd is. Merk op dat de *average case* van de ongebalanceerde boom ergens tussen de *best case* en *worst case* in zit, maar slechter is dan bij een gebalanceerde boom. Dit kan je zien doordat het gemiddelde van $\log_4 n$ en $\log_{1.33} n$ groter is dan $\log_2 n$ (al heb ik niet aangetoond dat de gemiddelde zoektijd gelijk is aan het gemiddelde van beide uitersten).

Neem nu een boom met in elke node (k-1) nodes links en alle overige nodes rechts. We nemen ook aan de linkersubboom gebalanceerd is.



Als *k* << *n*, is de kans groot dat de te zoeken node rechts ligt. Het uiterste rechtse pad is *n*/*k* lang. Om een node te vinden zullen gemiddeld stappen *n*/2*k* afleggen op dit uiterst rechtse pad om vervolgens links te zoeken in een gebalanceerde subboom. Dus:

$$aantalZoekstappen = \frac{n}{2k} + \log_2(k-1) \approx \frac{n}{2k}$$

Het aantal stappen is dan wel *k* kleiner dan voor een gewone lineaire lijst, maar door het lineaire effect is ze voor grote waarden van *n* veel groter dan met een logaritmische wet. Zie maar in de voorlaatste kolom.

De belangrijke conclusie is dus dat we een logaritmische wet moeten trachten te bekomen, ook als deze niet perfect is. De imperfectie maakt niet zoveel uit. Daarom spreken we over *O(n)* en *O(log₂n)*. De orde van *n* is veel belangrijker dan de factor.

7.5 Het opbouwen van een geordende binaire boom

Het toevoegen van een element in de boom, moet met de nodige zorg gebeuren:

```
public void add(T object){
      Node<T> newNode = new Node<T>(object);
      if (root == null)
            root = newNode;
      else
            add(root, newNode);
}
private void add(Node<T> current, Node<T> newNode){
 if (comparator.compare(newNode.data, current.data) < 0){
            // moet links komen
            if (current.left == null)
                  current.left = newNode;
            else
                  add(current.left, newNode);
      } else {
            // rechts
            if (current.right == null)
                  current.right = newNode;
            else
                  add(current.right, newNode);
      }
}
```

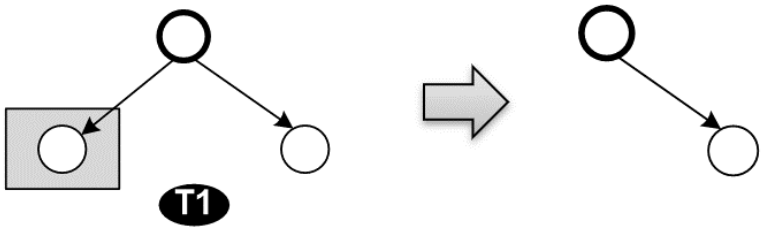
Initieel begint men met een lege boom, *root* staat op null. Het eerst-toe-te-voegen element moet aan de root toegekend worden. Verdere elementen gaan we via een recursief procédé toevoegen. We moeten immers de correcte positie zoeken. Tijdens het afdalen, vergelijken we het element met de node en bepalen we of het element in het linker- of rechterdeel moet komen. Merk op dat de recursieve *add*-methode *private* gedefinieerd is omdat die enkel binnen de klasse gebruikt zal worden.

Merk vervolgens op dat de volgorde van toevoegen, de uiteindelijke boom bepalen.
Oefening: hier de 3 mogelijke volgordes van toevoegen. Bepaal de uiteindelijke boom voor elke volgorde.

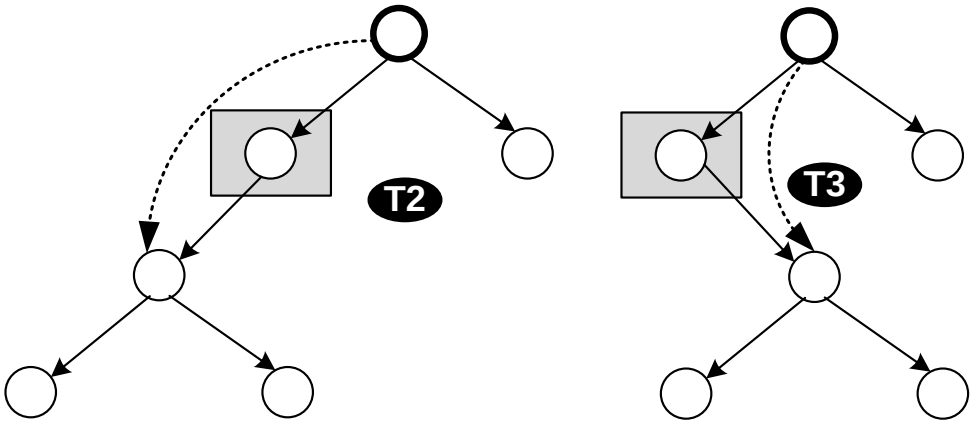
- k, l, g, i, x, q, r, a, e, f, c, z
- a, c, e, f, g, i, k, l, q, r, x, z
- i, z, q, l, k, f, e, g, c, x, a, r

7.6 Het verwijderen van een element uit een geordende binaire boom

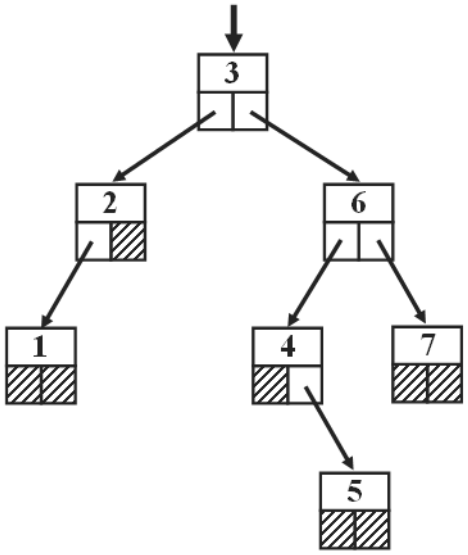
Het is soms nodig om uit een bestaande geordende boom een element te verwijderen, zonder de eigenschappen van de overblijvende boom aan te tasten. Dit is bijvoorbeeld het geval wanneer men een woord uit een woordenboek wenst te schrappen. Afhankelijk van waar het element zich in de boom bevindt, zal het verwijderen gemakkelijk of iets moeilijker zijn.
(T1) De node heeft geen subbomen en kan dus eenvoudig verwijderd worden.



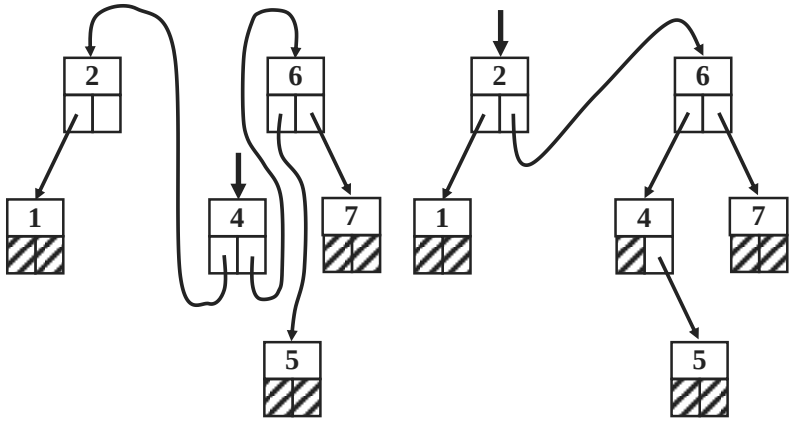
De te verwijderen node is aangeduid met de rechthoek.
(T2 & T3) De node heeft 1 subboom.



(T4 & T5 & T6) De node heeft 2 subbomen. Bijvoorbeeld, het verwijderen van ‘3’in volgende boom:



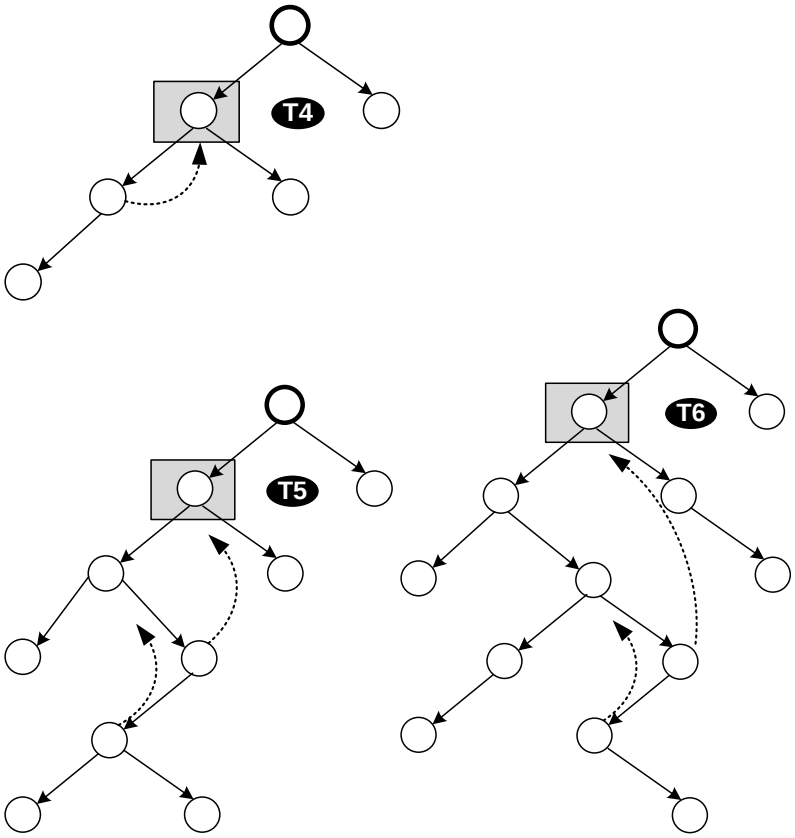
Wanneer het te verwijderen element een linker- én een rechtersubboom bezit, gaat het om het verwijderen van de wortel van een subboom. *Er moet dan een nieuwe wortel gekozen worden tussen de overblijvende nodes*. Omdat, per definitie, de wortel een sleutelwaarde heeft die kleiner is dan al de elementen van de rechtersubboom en groter dan deze van de linkersubboom, zijn het element met de *grootste sleutel uit de linkersubboom* of het element met de *kleinste sleutel uit de rechtersubboom* de beste kandidaten voor de rol van nieuwe wortel. Op deze manier blijven de herschikkingen aan de boom minimaal èn blijft de boom correct (weet je waarom?).
De figuur toont de twee verschillende manieren waarop de vorige boom kan worden herschikt na het verwijderen van element 3: ofwel 4 ofwel 2 worden gepromoveerd.



Hier wordt er geopteerd voor het *meest rechtse element van de linkersubboom*.

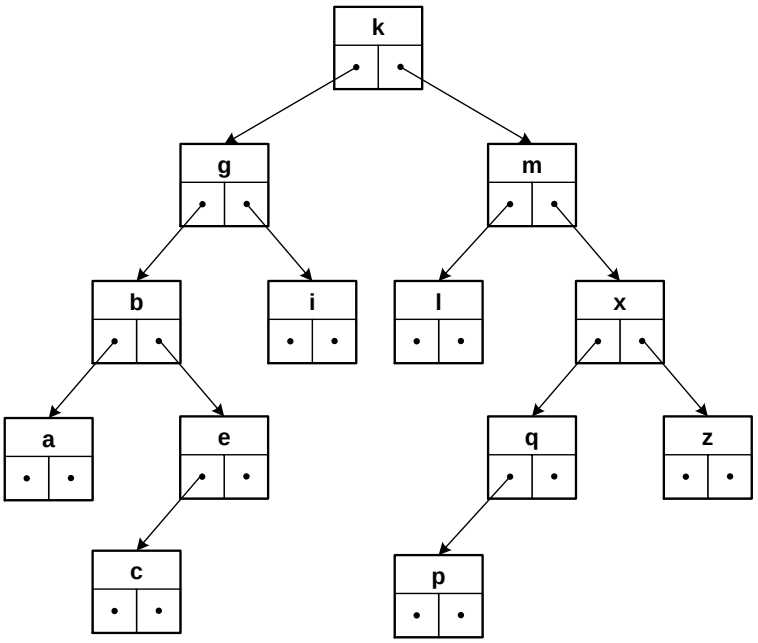
Hieronder het volledige algoritme.
Eerst checken we of de *root* het gezochte element is. Zo ja, dan moet deze verwijderd worden en moeten we van de linker- en rechtersubboom een nieuwe boom maken. Dit doen we met *createSubtree ()*. Op deze methode komen we dadelijk nog terug. Via *findNodeAndRemove()* dalen we in de boom tot we het element vinden of tot we merken dat het element niet aanwezig is. In dat laatste geval komen we op een *null* uit, net als bij het zoeken van een element dat we in het vorige deel hebben gezien. We keren onverrichterzake terug en geven *false* terug. Als we het element vinden als linker- of rechterelement, zullen we dit element vervangen door de subbomen van de elementen. Die we ook aanmaken met *createSubtree ()*. Deze methode maakt één boom van beide subbomen. Als één van beide leeg is, is het gemakkelijk. Dan kan de andere teruggegeven worden.
Merk op dat we met `findNodeAndRemove` één node hoger stoppen, net zoals bij gelinkte lijsten (6.3.5). Dit is nodig omdat we de `left` of de `right` van deze node moeten vervangen door een nieuwe. We testen dus of de `left` of de `right` overeenkomt met de te verwijderen node: `comparator.compare(current.left.data, object)` en `comparator.compare(current.right.data, object)`.

Dit zijn de 3 mogelijke situaties:



```
public boolean remove(T object){
      if (comparator.compare(root.data, object)== 0){
            root = createSubtree(root);
            return true;
      } else
            return findNodeAndRemove(root, object);
}
private boolean findNodeAndRemove(Node<T> current, T object){
      if (current == null)
            return false;
      if (current.left != null &&
comparator.compare(current.left.data, object)== 0){
            current.left = createSubtree(current.left);
            return true;
      } else if (current.right != null
                  && comparator.compare(current.right.data, object)== 0){
            current.right = createSubtree(current.right);
            return true;
      } else if (comparator.compare(object, current.data) < 0)
            return findNodeAndRemove(current.left, object);
      else
            return findNodeAndRemove(current.right, object);
}
/** create single tree from child nodes of node to be removed */
private Node<T> createSubtree(Node<T> nodeToBeRemoved){
      if (nodeToBeRemoved.left == null){
T1 T3       //right is the only subtree that we should consider
            return nodeToBeRemoved.right;
      } else if (nodeToBeRemoved.right == null){
T2          //left is the only subtree that we should consider
            return current.left;
      } else {
            // promote rightmost node of left subtree
            if (nodeToBeRemoved.left.right == null){
T4                nodeToBeRemoved.left.right = nodeToBeRemoved.right;
                  return nodeToBeRemoved.left;
            } else {
     Node<T> rightMostNode = pickRightMostNode(nodeToBeRemoved.left);
                  // he is new root of subtree
T5 T6             rightMostNode.right = nodeToBeRemoved.right;
                  rightMostNode.left = nodeToBeRemoved.left;
                  return rightMostNode;
            }
      }
}
private Node<T> pickRightMostNode(Node<T> p){
      // we expect current.right not to be null
      if (p.right.right != null){
T5          return pickRightMostNode(p.right);
      } else {
            Node<T> rightMostNode = p.right;
T6          p.right = rightMostNode.left;
            rightMostNode.left = null;
            return rightMostNode;
      }
}
```

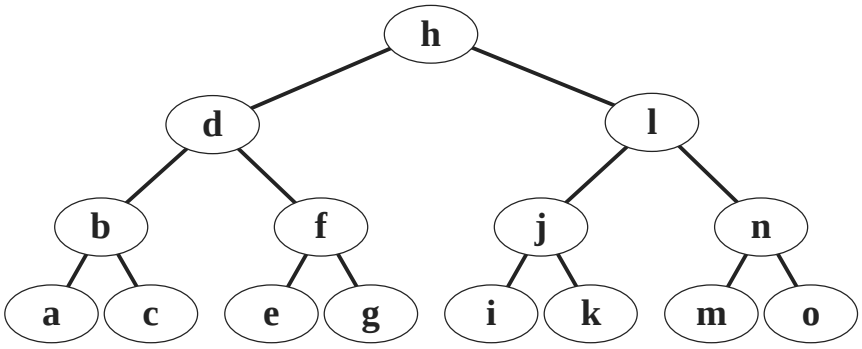
Oefening: Uit de volgende boom verwijderen we eerst element ‘l’, dan ‘q’, daarna ‘g’, vervolgens ‘x’ en tot slot verwijderen we ‘k’ met het algoritme van de cursus. Leg stap-voor-stap uit wat er gebeurt, teken de boom na elke verwijdering.



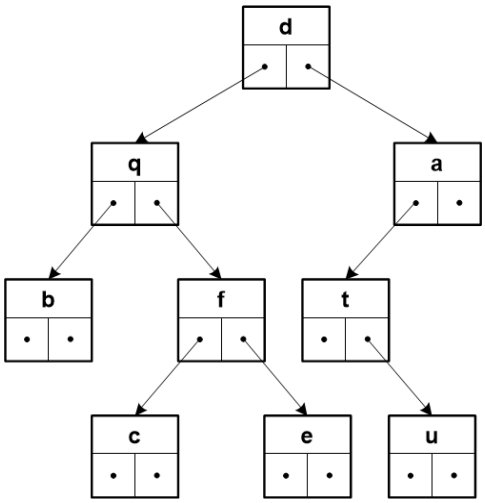
Opmerking: het is duidelijk dat bovenstaande manier niet de enige manier is om een nieuwe boom te maken na het verwijderen van de node. Paulien Vanheusden kwam op het examen met een alternatief (juni 2017): de rechtersubboom van de verwijderde node plak je rechts onder de meest rechtse node. De linkernode van de verwijderde node komt een stapje hoger. Als je in bovenstaande boom ‘g’ verwijdert, hang je ‘i’ rechts aan ‘e’ en ‘b’ hang je links aan ‘k’. Paulien kreeg haar punten omdat ze kon bewijzen dat dit een correct-gebalanceerde boom oplevert, maar meestal een minder goed-gebalanceerde boom zal opleveren dan de methode van de cursus. Lukt dit jou ook?

7.7. Het doorlopen van binaire bomen

Vaak willen we iets doen met alle elementen van de boom, zoals ze afprinten. De vraag is dan in welke volgorde we de elementen willen benaderen. Vaak worden drie specifieke volgordes aangewend die alle drie belangrijke toepassingen hebben. Ze worden respectievelijk '*in order'*, '*in preorder'* en '*in postorder'* genoemd.



PreOrder : ***print – links - rechts*** **: "hdbacfegljiknmo"**
InOrder : ***links – print - rechts*** **: "abcdefghijklmno"**
PostOrder : ***links – rechts - print*****: "acbegfdikjmonlh"**



Oefening: print de boom in de 3 volgordes

Ze kunnen alle drie eenvoudig verwezenlijkt worden door middel van recursieve procedures. De figuur toont een boom en de uitgeschreven versies van dezelfde boom volgens de drie bovenvermelde methodes. Het is vanzelfsprekend ook nog mogelijk van rechts naar links eerder dan van links naar rechts te werken, dat vergt een kleine aanpassing aan de algoritmen. Hier de drie methodes die de boom volgens de drie bovenvermelde manieren doorlopen en de nodes printen:

```
public void preOrder(Node<T> node){
      if (node != null){
            System.out.print(node.data+", ");
            preOrder(node.left);
            preOrder(node.right);
      }
}
```

```
public void inOrder(Node<T> node){
      if (node != null){
            inOrder(node.left);
            System.out.print(node.data+", ");
            inOrder(node.right);
      }
}
```

```
public void postOrder(Node<T> node){
      if (node != null){
            postOrder(node.left);
            postOrder(node.right);
            System.out.print(node.data+", ");
      }
}
```

Initieel worden deze procedures opgeroepen met als actuele parameter een pointer naar de wortel van de boom. De procedure verifieert dan eerst of de boom niet leeg is en indien nodig, begint hij aan de behandeling. De procedure *PreOrder* bv. zal eerst de wortel van de boom printen en daarna, recursief, zichzelf oproepen met eerst een pointer naar de linkersubboom en daarna een naar de rechtersubboom.

Hier printen we de data van de node, maar dit kan vervangen worden door een andere actie die we met de data willen doen. Backtracking (zie hoofdstuk 5) is een voorbeeld. Een tweede toepassing is om het aantal kinderen van alle nodes te berekenen. Hiervoor moeten we de boom in *PostOrder* doorlopen: het aantal nodes van linker- en rechtersubboom tellen. Hier de code:

```
      public int aantalKinderen(Node<T> node){
            if (node == null)
                  return 0;
            int n = aantalKinderen(node.left);
            n += aantalKinderen(node.right);
            System.out.println("Node "+node.data+" heeft "+n+" kinderen");
            return n + 1;
      }
```

Merk op dat deze recursieve algoritmen een vorm van *boomrecursie* zijn: er zijn meerdere recursieve oproepen. Deze kan men niet op een eenvoudige manier met een *while* implementeren. Bij de mergesort zien we hoe een stack/stapel te gebruiken waarin men de elementen/nodes bijhoudt die nog niet afgehandeld zijn.

Hoofdstuk 8

Sorteren

In het hoofdstuk over arrays hebben we laten zien dat het zoeken van een welbepaald element veel sneller kan gebeuren in een gesorteerde array. Dikwijls willen we elementen bijhouden in een zekere volgorde. In dit hoofdstuk zullen algoritmen ontwikkeld worden om arrays te ordenen volgens stijgende of dalende waarden van een bepaald sleutelveld. Het sleutelveld bepaalt de gegevens volgens welke de elementen gesorteerd zullen worden. Voor een array van personen zal dit naam en voornaam zijn. Voor een array van studenten kan dit het rolnummer zijn, maar ze kan ook gesorteerd worden op naam. Dit is afhankelijk van de toepassing. We moeten er enkel voor zorgen dat de relationele operatoren (<, = en >) op het sleutelveld toegepast kunnen worden.

Zoals aangetoond zal worden, kan de performantie van de verschillende algoritmen voor grote arrays erg verschillen. Aangezien ordenen een van de meest frequent voorkomende operaties in programma's is, loont het dus de moeite veel aandacht te besteden aan het samenstellen van een bibliotheek van procedures om optimaal te kunnen ordenen.

De code van dit hoofdstuk is te vinden in de klasse ***Sorting*** van de package ***Algoritmen***.

8.1. Selectionsort

Het **selectionsort** algoritme is een eerste intuïtieve, maar een wat naïeve manier om te sorteren. Het idee is om eerst op zoek te gaan naar het kleinste element, dan naar het op twee na kleinste, etcetera. Het kleinste element zetten we op de eerste plaats, het op een na kleinste op de tweede plaats, enzovoorts. Als we het element op zijn plaats zetten, zouden we logischerwijs de andere elementen een plaatsje naar achteren schuiven. Maar dit zou echter heel wat computertijd kosten. Het is een slimmer idee om de elementen te verwisselen: we zoeken naar het kleinste element en verwisselen dat met het eerste element.
De volgende tabel toont tenslotte de opeenvolgende toestanden (een volledige rij per toestand) van een array met integers die met behulp van het *selectionsort* algoritme geordend wordt. Het reeds geordende deel wordt in vet gedrukt terwijl het kleinste element van het overblijvende deel in schuinschrift voorgesteld is. In de eerste stap worden de eerste en nulde elementen verwisseld.

Step	A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]	A[8]	A[9]	A[10]
0	51	*03*	24	86	45	30	27	63	96	50	10
1	**03**	51	24	86	45	30	27	63	96	50	*10*
2	**03**	**10**	*24*	86	45	30	27	63	96	50	51
3	**03**	**10**	**24**	86	45	30	*27*	63	96	50	51
4	**03**	**10**	**24**	**27**	45	*30*	86	63	96	50	51
5	**03**	**10**	**24**	**27**	**30**	*45*	86	63	96	50	51
6	**03**	**10**	**24**	**27**	**30**	**45**	86	63	96	*50*	51
7	**03**	**10**	**24**	**27**	**30**	**45**	**50**	63	96	86	*51*
8	**03**	**10**	**24**	**27**	**30**	**45**	**50**	**51**	96	86	*63*
9	**03**	**10**	**24**	**27**	**30**	**45**	**50**	**51**	**63**	*86*	96
10	**03**	**10**	**24**	**27**	**30**	**45**	**50**	**51**	**63**	**86**	*96*
11	**03**	**10**	**24**	**27**	**30**	**45**	**50**	**51**	**63**	**86**	**96**

Het verwisselen is geïmplementeerd met de *swap*-functie. We kunnen de functie wat optimaliseren door te controleren of we een element niet met zichzelf aan het verwisselen zijn. Zoals je wellicht al wel weet, kan je een *swap* niet anders implementeren dan via een tussenvariabele (hier: *tmp*). Het zoeken van het kleinste element gebeurt met *indexMinimumVanaf*. Deze functie gaat op zoek naar het kleinste element vanaf een zekere index. Het geeft niet de kleinste waarde terug, maar zijn index; de plaats van de kleinste waarde in de array. De *swap* gebeurt immers op basis van indices!

```
	public static void selectionSort(int[] array){
		aantalVergelijkingen = 0;
		aantalKopies = 0;
		// we selecteren telkens het kleinste element
		for(int i = 0; i< array.length-1;i++){ // laatste is niet nodig
			int minIndex = indexMinimumVanaf(array, i);
			swap(array, i, minIndex);
			if (PRINT_TUSSEN_RESULTATEN)
			   System.out.println(" > ["+i+"] "+Arrays.toString(array));
		}
	}
	public static int indexMinimumVanaf(int[] array, int vanaf){
		int min = array[vanaf];
		int minIndex = vanaf;
		for(int i=vanaf+1;i<array.length; i++){ // vanaf + 1
			if (array[i] < min){
				min = array[i];
				minIndex = i;
			}
			aantalVergelijkingen++;
		}
		return minIndex;
	}
	/** swaps elements i and j from the array  */
	private static void swap(int[] array, int i, int j){
		if (i != j){
			int tmp = array[i];
			array[i] = array[j];
			array[j] = tmp;
			aantalKopies+=3;
		}
	}
```

Tijdens elke iteratie moeten we in het overblijvende deel steeds het kleinste element vinden. Wanneer *n* het aantal elementen in de array is, zal het aantal vergelijkingen tijdens de opeenvolgende iteraties gelijk zijn aan:

$$(n-1)+(n-2)+...+1=\sum_{i=1}^{n-1} n-i=\sum_{i=1}^{n-1} i=(n-1).(\frac{1+(n-1)}{2})=\frac{n^2-n}{2}=O(n^2)$$

Hieruit blijkt dat, voor grote arrays, het aantal vergelijkingen ongeveer evenredig is met het kwadraat van het aantal elementen *n* gedeeld door twee. Het algoritme is van orde *n* kwadraat: het aantal bewerkingen stijgt kwadratisch in *n*.

We kunnen dus besluiten dat *selectionsort* problematisch wordt voor het ordenen van grote arrays. Om de lezer hiervan te overtuigen, volstaat het een array met 10^6 elementen te beschouwen, en aan te nemen dat een vergelijking ongeveer 10^{-6} s duurt. Enkel rekening houdend met de vergelijkingen, stelden we al vast dat het ordenen 5.10^5 seconden of ongeveer vijf dagen in beslag zal nemen. Gelukkig bestaan er snellere algoritmen.
Omdat het aantal uitwisselingen (kopies) nooit groter kan zijn dan *n*, is dit zonder reële invloed op de performantie van het algoritme.

8.2. Bubblesort

Een variant van de *selectionsort* is de **bubblesort**. Alleen al vanwege zijn naam wil ik hem jullie niet onthouden. Bij *selectionsort* begonnen we steeds achteraan de lijst en liepen we de lijst naar voren door op zoek naar het kleinste element. Om een kleinste element te vinden, moeten we het tot-dan-toe-kleinste element bijhouden (in de methode *indexMinimumVanaf* onthouden we deze met de variabele *min*). In het uitgewerkte voorbeeld is het laatste element 10 en dit blijft lang ons kleinste element, tot we 3 tegen komen op positie 1. Maar omdat 10 klein is zal 10 waarschijnlijk van voor in de lijst moeten komen. Dus misschien is het een goed idee om 10 mee naar voren te nemen?

Dat is het idee van *bubblesort*: we nemen het tot-dan-toe-kleinste element mee naar voren en we laten het omhoog 'bubbelen'. Dit zien we gebeuren op het voorbeeld. Het element 10 wordt al snel gepromoveerd tot plaats 2.

Step	A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]	A[8]	A[9]	A[10]
0	51	03	24	86	45	30	27	63	96	50	10
1	***03***	51	*10*	24	86	45	30	27	63	96	50
2	**03**	***10***	51	24	*27*	86	45	30	*50*	63	96
3	**03**	**10**	***24***	51	27	*30*	86	45	50	63	96
4	**03**	**10**	**24**	***27***	51	30	*45*	86	50	63	96
5	**03**	**10**	**24**	**27**	***30***	51	*45*	*50*	86	63	96
6	**03**	**10**	**24**	**27**	**30**	***45***	51	50	63	86	96
7	**03**	**10**	**24**	**27**	**30**	**45**	***50***	51	63	86	96
8	**03**	**10**	**24**	**27**	**30**	**45**	**50**	**51**	63	86	96

Vetgedrukt zijn de elementen waarvan we zeker weten dat ze op hun plaats zijn (die testen we dan ook niet meer). Toch zie je dat ook de andere kleine elementen naar boven bubbelen en de grote elementen naar beneden. Zo zie je hoe 86 aan een langzaam maar zekere weg naar beneden vertrokken is vanaf stap 1. In stap 7 moeten we nog maar twee elementen van plaats verwisselen, 50 en 51. In stap 8 is er nog enige onzekerheid over de laatste 3 elementen, maar die blijken op hun plaats te staan (*sorted* geeft dit aan). We mogen dus stoppen.

```
public static void bubbleSort(int[] array){
         aantalVergelijkingen = 0;
         aantalKopies = 0;
         boolean sorted;
         int i=0;
         if (PRINT_TUSSEN_RESULTATEN)
         System.out.println(" > ["+i+"] "+Arrays.toString(array));
         do {
            sorted = true;
            for(int j=array.length-1;j>i;j--){
                  if (array[j] < array[j-1]){
                        swap(array, j, j-1);
                        sorted = false;
                  }
                  aantalVergelijkingen++;
            }
            i++; // het i'de element staat op zijn plaats
            if (PRINT_TUSSEN_RESULTATEN)
               System.out.println(" > ["+i+"] "+Arrays.toString(array));
         } while(!sorted);
      }
```

Bubblesort zal dus dikwijls sneller stoppen dan de *selectionsort*. Maar hij maakt veel meer kopieën. Bij de *selectionsort* doen we maar één swap per iteraties, terwijl we er hier meerdere doen.
Zowel het aantal vergelijkingen als het aantal kopieën is $O(n^2)$.
Dit kan gelukkig veel sneller. Laten we kijken naar de echt optimale algoritmen.

8.3. Quicksort

Het snelst bekende algoritme om een lijst van elementen te sorteren, bestaat erin de lijst te verdelen in twee delen, zodanig dat alle elementen van het ene deel kleiner zijn dan die van het tweede deel. (Voor deze operatie zullen uiteraard een aantal uitwisselingen tussen elementen van de twee delen noodzakelijk zijn.) Dezelfde verdeelprocedure zal daarna recursief op beide delen toegepast worden, op voorwaarde dat ze meer dan één element bevatten. Want zodra er maar één element overblijft, is het deel gesorteerd. De procedure hebben we in onze implementatie `partition` genoemd.
We wensen twee gelijke delen, hiervoor moeten we de mediaan (het middelste element) kennen. Met de mediaan, die we gebruiken als **pivot**, kunnen we de elementen verdelen in kleinere en grotere elementen. Maar de mediaan kennen we niet. Om deze te vinden zouden we de lijst moeten *sorteren*… De enige oplossing is een gok doen naar de mediaan. Hier kiezen we het meest rechtste element, met de hoop we met dit element 2 min-of-meer gelijke delen krijgen. Maar je had ook een ander pivot-element kunnen kiezen.
De volgende tabel geeft de opeenvolgende stappen van het ordenen van een kleine array weer. Het deel van de array dat opgesplitst wordt, is **vetgedrukt** met grijze achtergrond, terwijl het pivot element waarop de splitsing berust in *schuinschrift* en onderlijnd staat. Na de splitsing krijg je een deel links van de pivot met elementen kleiner dan de pivot en een deel rechts met elementen groter dan de pivot. Beide delen staan in lichtgrijs aangeduid. Merk de volgorde van afhandelen op gegeven door de recursie.

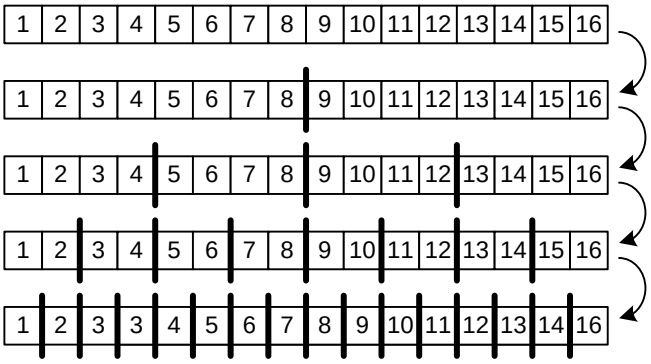
Step	A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]	A[8]	A[9]	A[10]
1	**51**	**03**	**24**	**86**	**45**	**30**	**27**	**63**	**96**	**50**	***10***
	3	*10*	24	86	45	30	27	63	96	50	51
2	3	10	**24**	**86**	**45**	**30**	**27**	**63**	**96**	**50**	***51***
	3	10	24	50	45	30	27	*51*	96	86	63
3	3	10	**24**	**50**	**45**	**30**	***27***	51	96	86	63
	3	10	24	*27*	45	30	50	51	96	86	63
4	3	10	24	27	**45**	**30**	***50***	51	96	86	63
	3	10	24	27	45	30	*50*	51	96	86	63
5	3	10	24	27	**45**	***30***	50	51	96	86	63
	3	10	24	27	*30*	45	50	51	96	86	63
6	3	10	24	27	30	45	50	51	**96**	**86**	***63***
	3	10	24	27	30	45	50	51	*63*	86	96
7	3	10	24	27	30	45	50	51	63	**86**	***96***
	3	10	24	27	30	45	50	51	63	86	*96*

```
1     public static void quickSort(int[] array){
2        aantalVergelijkingen=0;
3        aantalKopies = 0;
4        quicksort(array, 0, array.length - 1);
5     }
6     private static void quicksort(int[] array, int left, int right) {
7        if (right <= left) return;
8        int i = partition(array, left, right);
9        if (PRINT_TUSSEN_RESULTATEN)    System.out.println(" > ["+left+" | "
10                                 +i+" | "+right+"] "+Arrays.toString(array));
11
12        quicksort(array, left, i-1);
13        quicksort(array, i+1, right);
14    }
15    private static int partition(int[] a, int left, int right) {
16          // a[right] is ons pivot-element
17        int i = left;
18        int j = right - 1;
19        while (true) {
20            while (a[i] < a[right]){ // vind links element > pivot
21                i++;
22                aantalVergelijkingen++;
23            }
24            while (a[right] < a[j]){// vind rechts element < pivot
25                aantalVergelijkingen++;
26                if (j == left)  // ga niet buiten array
27                      break;
28                j--;
29            }
30            if (i >= j) // tests of indexen mekaar hebben gekruisd
31                break;
32            swap(a, i, j);  // verwissel beide elementen
33            i++;
34            j--;
35        }
36        swap(a, i, right);  // verwissel met pivot
37        return i;
38    }
```

Vragen

- Lijn 12 & 13: waarom staat er niet *i* in 1 van beide? Zou dit fout zijn?
- Wat als we lijnen 12 en 13 omdraaien?
- Lijn 26: de conditie zorgt ervoor dat we niet buiten de array komen. Waarom niet?
- Wanneer is *i* gelijk aan *j* in de test van lijn 30?

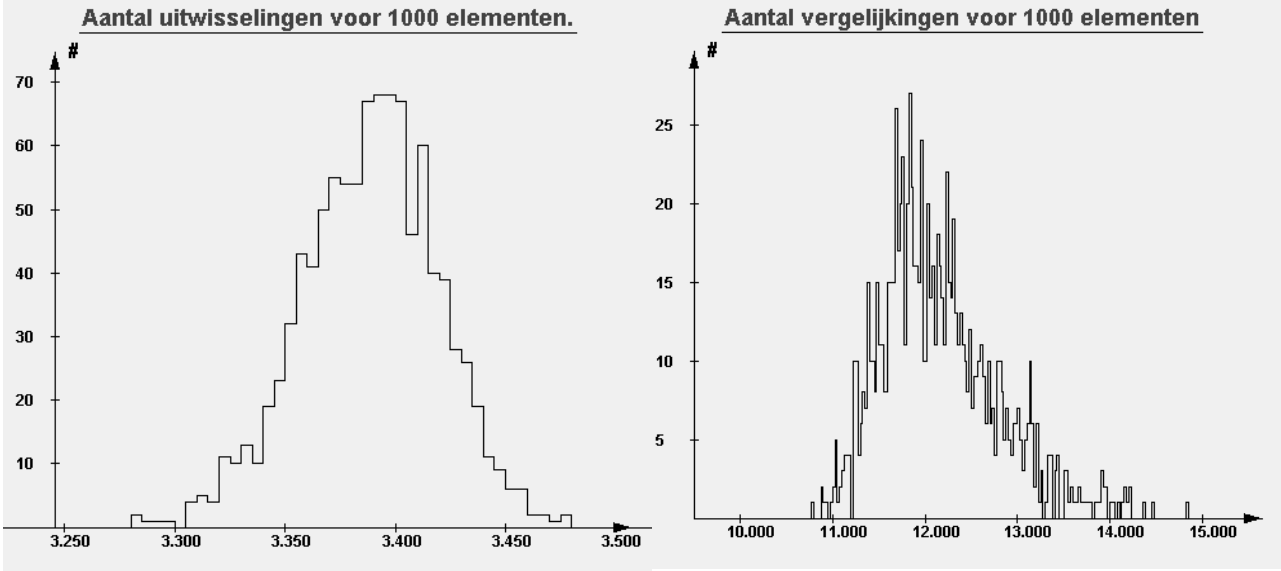
Het opsplitsen van een array met *n* elementen vergt *n* vergelijkingen en, gemiddeld, *n*/4 uitwisselingen. Het verder opsplitsen van het geheel van de reeds opgesplitste arrays vergt ook globaal *n* vergelijkingen, ten minste als het totale aantal elementen constant blijft. Het minimaal aantal keren dat elke array opgesplitst moet worden alvorens er, per array, nog maar één element overblijft is $\lceil log_2 n$. Ten minste als, na *k* opsplitsingen, de gemiddelde lengte van de resulterende arrays gelijk is aan $n/2^k$. Het $\lceil$-symbool duidt aan op het naar boven afronden.



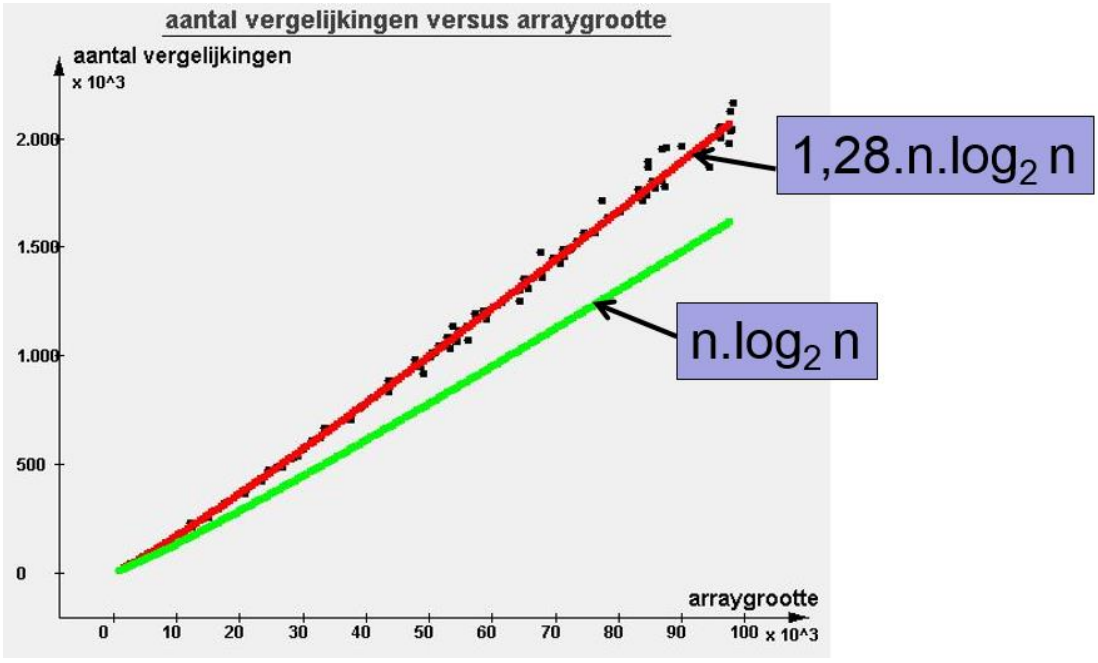
Dit zie je in dit schema. We hebben hier vier stappen nodig. Tijdens elke stap doen we *n* vergelijkingen. Let op, het recursieve algoritme zal eerst de linkse delen sorteren en zo naar rechts opschuiven. Zoals we lieten zien in de tabel. Dit schema is nuttig om duidelijk te maken hoeveel stappen we nodig hebben.

Het totaal aantal vergelijkingen nodig om een array met *n* elementen te ordenen met het quicksort-algoritme is dus $\boldsymbol{n \lceil log_2 n}$ terwijl het aantal uitwisselingen nog vier maal kleiner zal zijn. Hieruit volgt dat quicksort bijzonder goed geschikt is voor grote tabellen. Ter illustratie kunnen we berekenen dat de tabel met 10^6 elementen waarvan sprake is in paragraaf 6.1, in vijf dagen geordend kon worden met behulp van selectionsort die in 20 seconden ($\log_2 10^6 \approx 20$) geordend kan worden op dezelfde computer met quicksort.

De snelheid van quicksort hangt af van de kwaliteit van de pivot. Ideaal is het de mediaan. Als de pivot enorm van de mediaan afwijkt, zal de lijst niet in 2 gelijke delen gesplitst worden en wijken we af van de logaritmische wet. Zoals we al zagen bij de zoektijd van binaire bomen (7.4) is kunnen we een logaritmische wet behouden als we 2 delen bekomen die *ongeveer* gelijkwaardig zijn. Dezelfde analyse kunnen we hier toepassen. Het is bewezen dat de zoektijd gemiddeld 1,39. $\boldsymbol{n\ log_2 n}$ zal zijn (zie Wikipedia). Ik heb dit experimenteel getest door willekeurige arrays van 1000 elementen te sorteren. Het aantal uitwisselingen en aantal vergelijkingen is te zien in de volgende frequentietabel:



Het gemiddeld aantal vergelijkingen is dus ongeveer 12000. *n.log2 n* = 9965 en 1,39*.n.log2 n* = 13851. We doen het dus wat beter dan de theoretische 1,39-wet. Ook heb ik arrays van verschillende groottes getest. Het aantal vergelijkingen staat geplot in de volgende grafiek:



Als we een curve fitten op de gegevens (ook regressie genoemd) geeft dit de functie:

aantal vergelijkingen=1,28*.n.log2 n*.

Uit 7.4 leerden we ook dat we de logaritmische wet niet bekomen als 1 van beider delen na splitsing een constante grootte heeft. Dit zal voor de quicksort ook een afwijking van de logaritmische wet geven en resulteren in een kwadratische wet. Dit noemen we de *worst case* situatie.

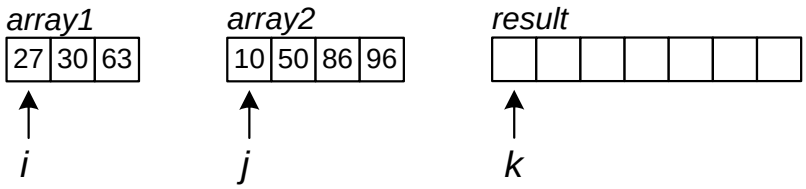
Oefening: ga na dat dit ondermeer gebeurt als de initiële array gesorteerd is (of in omgekeerde volgorde). Ga ook na wat dit geeft voor de andere sorteeralgoritmen.

8.4. Mergesort

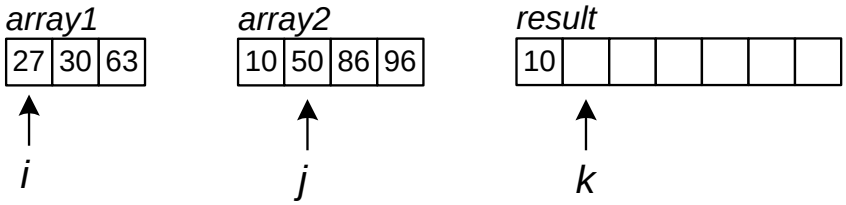
Het **mergesort**-algoritme is het laatste sorteeralgoritme dat we zullen bespreken. Het is even snel als *quicksort* maar heeft een groot nadeel waardoor het niet zoveel wordt gebruikt. Het basisidee is om twee gesorteerde arrays samen te voegen of te '*mergen*', vandaar dus de naam. We werken dat idee eerst uit.

8.4.1 Samenvoegen van twee gesorteerde lijsten.

Stel, we hebben twee arrays die al gesorteerd zijn. Hoe maken we hier op een snelle manier één gesorteerde array van?
Neem het volgende voorbeeld. We starten met twee arrays, één van 3 groot en één met grootte 4. We maken een nieuwe array met grootte 7 om het resultaat in op te slaan.



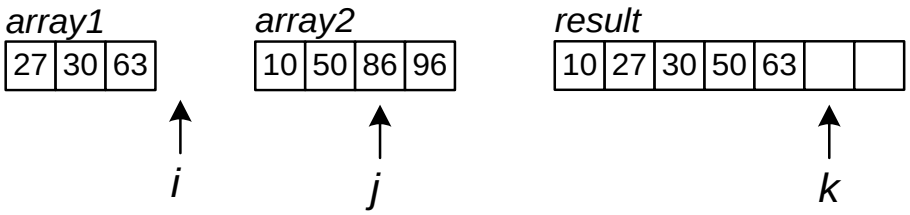
We starten met drie indexen voor elke array, initieel op nul. Het eerste element van het resultaat moet het kleinste element zijn. Omdat we weten dat beide *input-arrays* gesorteerd zijn, moet dit het eerste element van een van beide zijn. We vergelijken beide en merken dat *array2* het kleinste element heeft gegeven door index *j*. Dit komt dus in het eerste vakje van *result*. Dan schuiven we beide indexen *j* en *k* op en krijgen we het volgende.



Het tweede-kleinste element is nu weer het i^{de} of j^{de} element van respectievelijk de eerste of tweede array. Hier is dit de eerste array. Zo lopen we beide arrays af en vullen we het resultaat aan. Dit is beschreven door de eerste *while*-loop van volgende methode:

```
	private static int[] merge(int[] array1, int[] array2){
		int[] result = new int[array1.length + array2.length];
		int i=0, j=0; // i is om voor array1, j voor array2
		int k=0; // k is voor de result array
		while(i < array1.length && j <array2.length){
			if (array1[i] < array2[j]){
				result[k] = array1[i];
				i++;
			} else {
				result[k] = array2[j];
				j++;
			}
			k++;
			aantalVergelijkingen++;
			aantalKopies++;
		}
		while(i < array1.length){
			result[k] = array1[i];
			i++;
			k++;
			aantalKopies++;
		}
		while(j <array2.length){
			result[k] = array2[j];
			j++;
			k++;
			aantalKopies++;
		}
		return result;
}
```

Op een bepaald ogenblik krijgen we de volgende situatie:



We hebben het laatste element van *array1* gekopieerd. Nu mogen we niet meer verder gaan met de eerste *while*, omdat het i^{de} element van *array1* niet bestaat. Logisch ook, we weten dat we alle elementen van de eerste array gehad hebben. Nu moeten we enkel nog de overige elementen van de tweede array kopiëren. Dit gebeurt in het tweede deel van de code. We voorzien twee *while*-loops, de eerste voor het geval dat er nog elementen in *array1* zitten, de tweede voor de elementen van *array2*.
Het mergen van twee gesorteerde arrays kan dus zeer snel. Met een totale lengte van *n* moeten we *n* kopieën maken en maximaal *n* vergelijkingen. Meestal minder dan *n* vergelijkingen, afhankelijk van hoeveel 'geluk' we hebben.

Vraag**:** In welke gevallen moeten we de minste vergelijkingen maken? Hoeveel zijn dat er?

Maar dit mergen heeft een belangrijk nadeel, het kan niet '*in place'* gebeuren, binnen dezelfde array. Je hebt een nieuwe array nodig om je elementen naartoe te kopiëren.

8.4.2 Mergesort recursief.

We kunnen nu twee gesorteerde lijsten samenvoegen tot één gesorteerde lijst. Hoe maken we hier nu een algoritme van voor het sorteren van een volledig-ongesorteerde array? We snijden de array in stukjes tot we arrays hebben van grootte 1. En een array met enkel één element is steeds gesorteerd! Dat is slim, want nu kunnen we onze bovenstaande *merge* loslaten op twee arrays van één element; de sorteervoorwaarde is immers vervuld. Zo krijgen we een gesorteerde array van twee elementen. Die combineren we dan weer met een andere gesorteerde array van twee elementen. En zo voegen we alle stukjes weer samen tot ... één gesorteerde array.

We implementeren dit op twee manieren. Hier tonen we de eerste, recursieve versie die gebruik maakt van bovenstaande *merge*-functie:

```
public static int[] mergeSortRecursief(int[] array){
	// deel array op in twee
	int halveLengte = array.length/2;
	int[] arr1 = Arrays.copyOfRange(array, 0, halveLengte);
	int[] arr2 = Arrays.copyOfRange(array, halveLengte, array.length);

	// sorteer beide delen
	if (arr1.length > 1)
		arr1 = mergeSortRecursief(arr1); // arr1 vervangen door gesorteerde
	if (arr2.length > 1)
		arr2 = mergeSortRecursief(arr2);

	// nu beide onderdelen gesorteerd zijn, kunnen we ze 'mergen'
	return merge(arr1, arr2);
}
```

We hakken de ongesorteerde inputarray in twee. We maken twee nieuwe arrays met elk de helft van de elementen. De Javaklasse *Arrays* biedt een functie om dit te doen. Als deze deelarrays groter zijn dan één, vragen we recursief aan *mergeSort* om deze te sorteren. Als dit is gebeurd, en we hebben twee gesorteerde deelarrays, kunnen we onze *merge*-functie hierop loslaten.
Wat gebeurt er nu recursief? *MergeSort* wordt twee keer opgeroepen met telkens de helft van de array. Het algoritme zal die weer in twee snijden, net zolang tot het stukje maar één element heeft. Dan stopt de recursie. Dan keren we terug uit de recursie en *mergen* de deelarrays. Dit gebeurt dus eerst op twee stukjes van grootte 1 (meest linkse), dan nog twee stukjes van grootte 1 (rechts van de vorige) en dan worden beide stukjes van grootte 2 samengevoegd tot 1 stuk van 4 elementen. Als je het éénmaal door hebt, is recursie fantastisch… Programmeernerds kicken erop.
Vraag: hoeveel arrays maken we aan bij een array van grootte *n*? En hoeveel keer wordt de recursieve functie opgeroepen?

8.4.3 Natuurlijke mergesort.

Hier een tweede oplossing die gebruik maakt van de *FIFOQueue* die we aanmaakten in Hoofdstuk 3. We gaan alle nog-te-mergen stukjes bijhouden in de queue. Want wat gebeurt er bij de recursieve oplossing? De array wordt opgedeeld in stukjes van uiteindelijk slechts grootte 1. Elk element belandt op het einde van de opsplitsing in een array van grootte 1 en zal vanaf dan samengevoegd worden met andere arrays om tot de volledige, gesorteerde array te komen. We kunnen dus ook onmiddellijk de array opdelen in stukjes van lengte 1, in de queue stoppen en dan het samenvoegen aanvatten. Bovendien kunnen we dan een optimalisatie uitvoeren: als er stukjes voorkomen die reeds gesorteerd zijn, gaan we deze niet verder opdelen.

Het voorbeeld van 11 getallen geeft zo 7 gesorteerde delen (van elkaar gescheiden met ‘|’):

0	51 | 3 24 86 | 45 | 30 | 27 63 96 | 50 | 10 |
1	45 | 30 | 27 63 96 | 50 | 10 | 3 24 51 86 |
2	27 63 96 | 50 | 10 | 3 24 51 86 | 30 45 |
3	10 | 3 24 51 86 | 30 45 | 27 50 63 96 |
4	30 45 | 27 50 63 96 | 3 10 24 51 86 |
5	3 10 24 51 86 | 27 30 45 50 63 96 |
6	3 10 24 27 30 45 50 51 63 86 96 |

Die 7 delen houden we bij in de queue. We nemen dan de twee eerste en smelten deze samen. Hier: {51} en {3, 24, 86}. Het samengesmolten deel voegen we weer toe aan de queue en die komt vanachter. Zie situatie na stap 1.

Vervolgens wordt {45} en {30} gemerged, dan {27 63 96} en {50}, enzovoorts.
Hier een implementatie van dit algoritme, met twee delen: fase 1 voor het opdelen en fase 2 voor het sorteren. Fase 2 wordt uitgevoerd zolang er meer dan één array is in de queue. Met één array hebben we de uiteindelijk gesorteerde array.

```
public static int[] naturalMergeSort(int[] array){
	FIFOQueue<int[]> queue = new FIFOQueue<int[]>(array.length);

	// 1. we kappen de originele array eerst in gesorteerde stukjes
	int start=0;
	for(int i=1;i<array.length;i++){
		if (array[i] < array[i-1]){
			queue.add(Arrays.copyOfRange(array, start, i));
			start = i;
		}
	}
	queue.add(Arrays.copyOfRange(array, start, array.length)); // het laatste stukje

	// 2. en sorteren we alle stukjes
	aantalVergelijkingen=0;
	aantalKopies = 0;
	while(queue.size() > 1){
		int[] arr1 = queue.get();
		int[] arr2 = queue.get();
		int[] arr3 = merge(arr1, arr2);
		queue.add(arr3);
	}
	return queue.get();
}
```

Stel dat we een array van letters willen sorteren, bijvoorbeeld de letters van het toetsenbord, dan krijgen we onderstaand schema. In bovenstaande code moeten we `int[]` vervangen door `char[].`

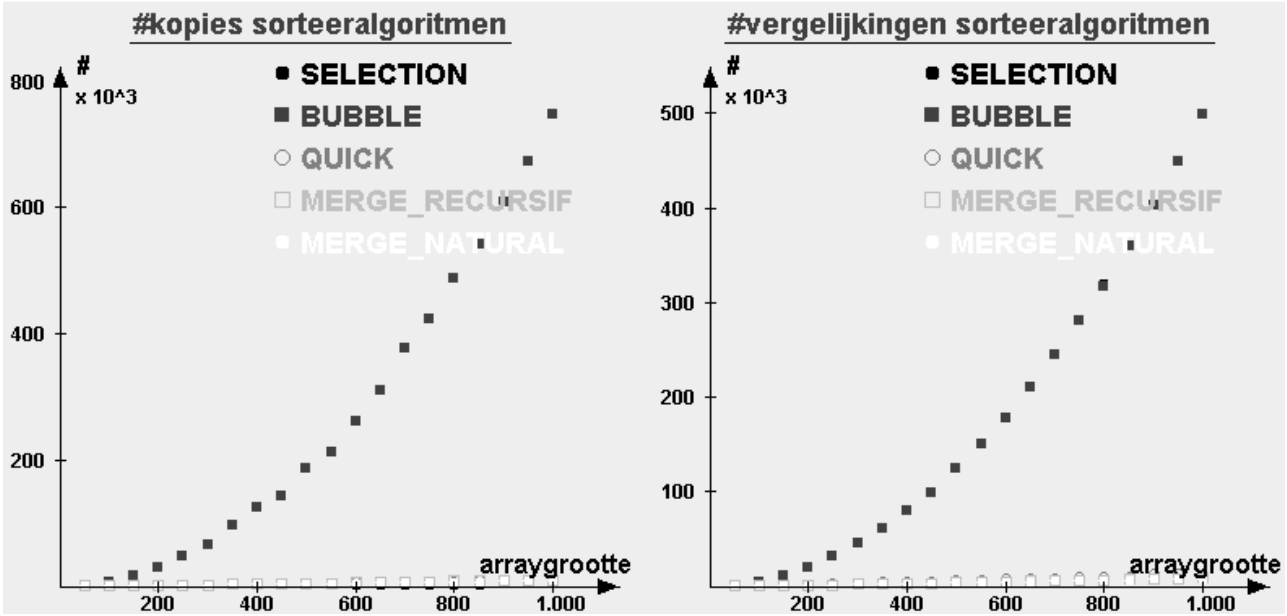
0	**qw|erty|u|iop|as|dfghjklz|x|cv|bn|m|**
1	**u|iop|as|dfghjklz|x|cv|bn|m|eqrtwy|**
2	**as|dfghjklz|x|cv|bn|m|eqrtwy|iopu|**
3	**x|cv|bn|m|eqrtwy|iopu|adfghjklsz|**
4	**bn|m|eqrtwy|iopu|adfghjklsz|cvx|**
5	**eqrtwy|iopu|adfghjklsz|cvx|bmn|**
6	**adfghjklsz|cvx|bmn|eiopqrtuwy|**
7	**bmn|eiopqrtuwy|acdfghjklsvxz|**
8	**acdfghjklsvxz|beimnopqrtuwy|**
9	**abcdefghijklmnopqrstuvwxyz|**

Waarom gebruiken we een *fifo-queue* en geen *stack*? Omdat we eerst de kleinste subarrays willen mergen en de gemergde arrays er achteraan weer bij willen stoppen. Die mergen we dan pas als we klaar zijn met de kleinsten.
Vraag: Teken hetzelfde mergeschema bij gebruik van een stack (*pop* en *push* ipv *get* en *add*).

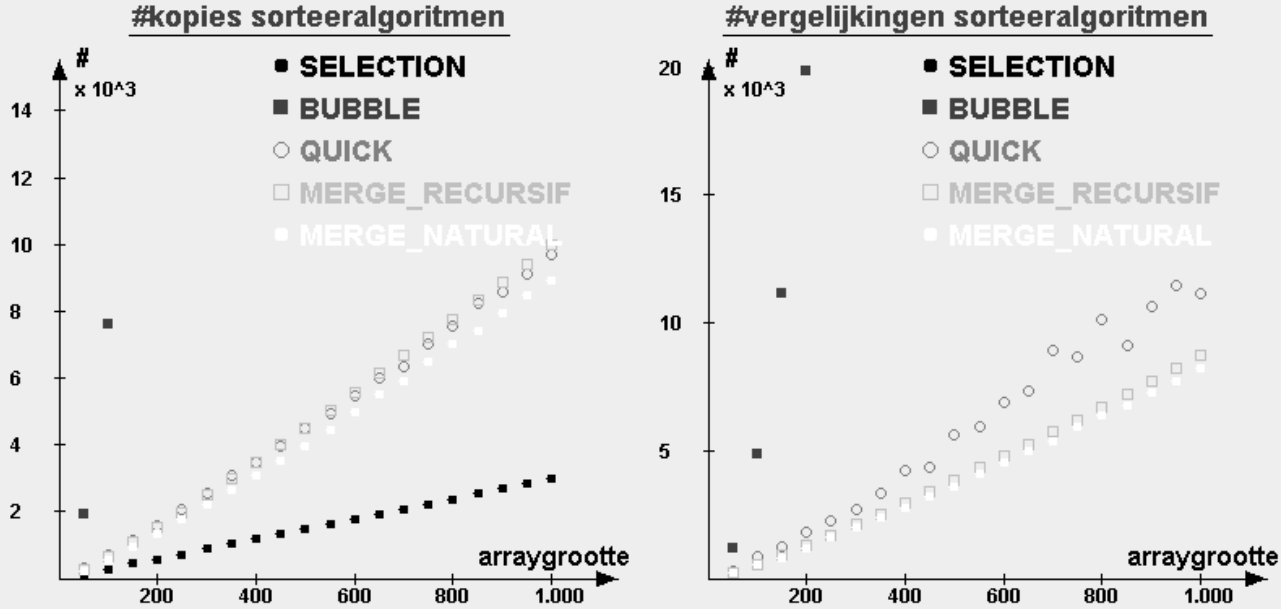
Dit algoritme toont ***hoe je boomrecursie zonder recursie kan programmeren***: je houdt alle deeltjes bij met een queue. Dit is in feite wat er in de computer ook gebeurt bij recursie: alle actieve functie-aanroepen worden op een stack bijgehouden (de program call stack).

8.4.4 Performantie van de 5 algoritmen.

Onderstaande grafieken tonen duidelijk de verschillende grootteordes voor de performantie qua aantal vergelijkingen en aantal kopies.



In de eerste grafiek is enkel de bubble erboven uitsteekt, in de tweede de bubble en selectionsort. De selection maakt weinig kopies, enkel een swap op het einde. Dat is dan ook lineair. Laat ons inzoomen op de onderste curves:



Quicksort heeft iets meer vergelijkingen nodig dan beide merge sorts, maar voor het aantal kopies is de natuurlijke merge de ‘winnaar’. De eerste curve voor selectionsort is mooi lineair.

8.5. Generiek sorteren

De welbekende *Arrays*-klasse biedt een sorteeralgoritme voor arrays van elk van de primitieve types. Voor elk type bestaat een tweede versie waarbij je een range meegeeft.

Method Summary	
`static void`	**sort**`(byte[] a)` Sorts the specified array of bytes into ascending numerical order.
`static void`	**sort**`(byte[] a, int fromIndex, int toIndex)` Sorts the specified range of the specified array of bytes into ascending numerical order.
`static void`	**sort**`(char[] a)` Sorts the specified array of chars into ascending numerical order.
`static void`	**sort**`(char[] a, int fromIndex, int toIndex)` Sorts the specified range of the specified array of chars into ascending numerical order.
`static void`	**sort**`(double[] a)` Sorts the specified array of doubles into ascending numerical order.
`static void`	**sort**`(double[] a, int fromIndex, int toIndex)` Sorts the specified range of the specified array of doubles into ascending numerical order.
`static void`	**sort**`(float[] a)` Sorts the specified array of floats into ascending numerical order.
`static void`	**sort**`(float[] a, int fromIndex, int toIndex)` Sorts the specified range of the specified array of floats into ascending numerical order.
`static void`	**sort**`(int[] a)` Sorts the specified array of ints into ascending numerical order.

static void	**sort**(int[] a, int fromIndex, int toIndex) Sorts the specified range of the specified array of ints into ascending numerical order.
static void	**sort**(long[] a) Sorts the specified array of longs into ascending numerical order.
static void	**sort**(long[] a, int fromIndex, int toIndex) Sorts the specified range of the specified array of longs into ascending numerical order.
static void	**sort**(Object[] a) Sorts the specified array of objects into ascending order, according to the *natural ordering* of its elements.
static void	**sort**(Object[] a, int fromIndex, int toIndex) Sorts the specified range of the specified array of objects into ascending order, according to the *natural ordering* of its elements.
static void	**sort**(T[] a, Comparator<? super T> c) Sorts the specified array of objects according to the order induced by the specified comparator.
static void	**sort**(T[] a, int fromIndex, int toIndex, Comparator<? super T> c) Sorts the specified range of the specified array of objects according to the order induced by the specified comparator.

Maar wat nu met arrays van *objecten*? Zie de vier laatste functies. Een object kan immers vanalles zijn. Hoe weten we volgens welke sleutel deze geordend moeten zijn? Hoe kan Java de orderelatie bepalen? Hiervoor moet 'iemand' haar vertellen welke objecten groter zijn dan een ander object. Voor elk koppel moet Java kunnen nagaan welke groter of gelijk is. Dit kan op twee manieren.

De eerste manier is het vastleggen van de *natural ordering*. Deze wordt gebruikt in **sort**(Object[] a). Hiervoor moet elk object de *Comparable* interface implementeren. Deze interface is als volgt gedefinieerd:

java.lang

Interface Comparable

Method Summary

int	**compareTo**(Object o) Compares this object with the specified object for order. Returns a negative integer, zero, or a positive integer as this object is less than, equal to, or greater than the specified object.

Er is dus één methode die we voor elk object moeten implementeren. Dit object, O_1, kan dan worden vergeleken met elk ander object, O_2. De methode geeft -1 terug als $O_2 > O_1$, 0 als beide objecten gelijk zijn en +1 als $O_2 < O_1$. Volgens de documentatie mag het eender welk negatief of positief getal zijn dat je teruggeeft, maar typisch zullen we -1 of +1 teruggeven. Er wordt vanuit gegaan dat de *comparator* consistent is. Bijvoorbeeld, als blijkt dat $O_2 < O_1$ en $O_3 < O_2$, dan moet ook $O_3 < O_1$. Dan noemen we het een orderelatie.

De *compareTo()* implementeer je dus voor je objecten. Dan kan je objecten onderling gaan vergelijken. Dit wordt de **natural ordering** genoemd. Toegepast op onze student-objecten van hoofdstuk 1, definiëren we de natuurlijke orde op naam en voornaam. Merk op dat de methode *compareTo* toegepast op *naam* en *voornaam* is gedefinieerd in de String klasse. Deze methode kan je dus gebruiken om woorden te vergelijken.

```
public class Student implements Comparable<Student>{
	String voornaam, naam;
	int rolnummer, score;
	Vak[] vakken;
	int[] punten;

	Student(String voornaam, String naam, int rolnummer){
		this.voornaam=voornaam;
		this.naam=naam;
		this.rolnummer=rolnummer;
		vakken = new Vak[4];
	}
	@Override
	public int compareTo(Student student2) {
		int ordeNaam = this.naam.compareTo(student2.naam); // dit is de
compareTo van String
		if (ordeNaam == 0) // beide dezelfde naam
			return this.voornaam.compareTo(student2.voornaam);
		else
			return ordeNaam;
	}
}
```

Studenten met dezelfde naam sorteren we op voornaam. Nu kunnen we sorteren:

```
Arrays.sort(studenten);
```

Maar willen we de studenten enkel maar op naam sorteren? Nee, er zijn gevallen waar we op een andere *sleutel* (*key* in het engels) willen sorteren. Stel dat we bijvoorbeeld op de score willen sorteren. Hiervoor dient de tweede oplossing, gegeven met de functie `sort(T[] a, Comparator<? super T> c)`. Je geeft een *Comparator* mee. Het *comparator*-object zal gebruikt worden om de objecten te vergelijken. Met het object definiëren we op welke sleutel we willen sorteren. Comparator is een interface met twee methodes gedefinieerd voor een type **T**:

java.util

Interface Comparator<T>

Method Summary	
int	**compare**(T o1, T o2) Compares its two arguments for order. Returns a negative integer, zero, or a positive integer as the first argument is less than, equal to, or greater than the second.
boolean	**equals**(Object obj) Indicates whether some other object is "equal to" this comparator.

De *compare*-methode is gelijkaardig aan de *compareTo*-methode van *Comparable*. De *equals*-methode is hier een beetje misleidend. Deze methode is *niet* om **T**-objecten te vergelijken (Student-objecten), maar om twee comparator-objecten te vergelijken. Dit laat je toe om te controleren of twee comparators hetzelfde doen. Dit zullen we niet echt nodig hebben. De *equals*-methode is trouwens voor elke klasse al gedefinieerd, je hoeft deze dus niet te implementeren.
Om Student-objecten op score te sorteren, implementeren we onze eigen *Comparator*:

```
class ComparatorOnScore implements Comparator<Student>{
	@Override
	public int compare(Student student1, Student student2) {
		return student1.score - student2.score;
	}
}
```

Sorteren gebeurt nu als volgt, met ons eigen comparator-object:

```
Arrays.sort(studenten, new ComparatorOnScore());
```

Tot slot: in hoofdstuk 2 zagen we de *ArrayList* klasse als een handige klasse om lijsten bij te houden, als alternatief voor arrays. De *Collections* klasse geeft twee sorteerfuncties lijsten, óók gebaseerd op de natuurlijke volgorde (gedefinieerd met *Comparable*) of met een eigen *Comparator*.

Collections class:

static void	**sort**(List list) Sorts the specified list into ascending order, according to the *natural ordering* of its elements.
static void	**sort**(List list, Comparator c) Sorts the specified list according to the order induced by the specified comparator.

Hoofdstuk 9

Hashing

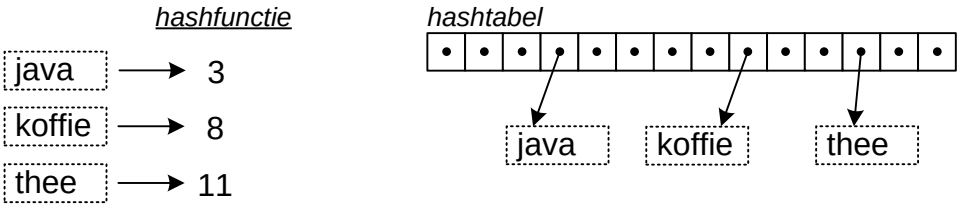
Het zoeken in een verzameling van één object is in deze cursus al verschillende malen aan bod gekomen. In hoofdstuk 2 werd uitgelegd hoe men een object kan zoeken in een array van objecten. Indien in de array de records in een willekeurige volgorde voorkomen, is de zoektijd evenredig met *n*, het aantal elementen. Daarentegen, wanneer de elementen volgens sleutelwaarde geordend zijn, is de zoektijd veel korter. (In de orde van $\log_2 n$.) Voor een array moet zijn afmeting op voorhand worden gedeclareerd. Dat is, gezien de omvang van de verzameling objecten, een vervelende beperking.
In hoofdstuk 7 werden zoekbomen (binaire bomen) besproken. Dergelijke bomen worden dynamisch opgebouwd. Het aantal elementen moet dus niet a priori bekend zijn, omdat in deze bomen de gegevens ook geordend zijn, is de zoektijd in de orde van $\log_2 n$. De combinatie van dynamisch geheugen en van een logaritmische zoektijd verklaart waarom bomen de meest gebruikte techniek zijn voor het zoeken van een record in grote verzamelingen gegevens.
In dit hoofdstuk wordt een derde zoekmethode uitgelegd. Deze is meestal nog veel sneller dan het gebruik van bomen, maar hij vereist een betrouwbare bovenlimiet voor het aantal objecten dat in de lijst terecht zal komen. Het aantal moet a priori bekend zijn, wat het belangrijkste nadeel is van de methode. Deze methode heet **hashing**. Het is een belangrijke, veelgebruikte methode.

9.1. Hashfuncties

Mathematisch gezien is een zoekalgoritme *een functie die voor elke sleutelwaarde een index of een geheugenadres berekent* (of zoekt). Als we zoeken in een niet-geordende array bepaalt de functie het adres door opeenvolgend alle waarden van de index te proberen. In het geval van bomen gebeurt dit door een keten pointers te volgen. Omdat we in feite mogen kiezen waar we een element in een lijst plaatsen, kunnen we dit laten berekenen door een functie die we ook gebruiken om het element later terug te vinden. We gaan dus functies bedenken die met elk mogelijke sleutelwaarde een index in een array associëren. Met **sleutel** bedoelen we de eigenschap(pen) van de objecten die we gebruiken voor het ordenen en terugvinden van de objecten. Dit is bijvoorbeeld je naam en voornaam.

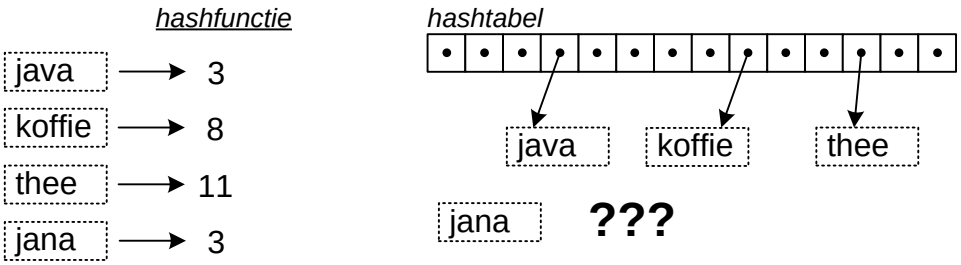
We gaan proberen een methode op te stellen om op een directe manier de plaats van een object in een array te bepalen. Dit doen we door een functie op te stellen die de index geeft aan de hand van de sleutel.

De functie die, vertrekkend van een sleutel een indexwaarde bepaalt, noemt men een **hashfunctie**. Het interessante aan dergelijke functies is dat de tijd die nodig is om de waarde van de index te bepalen onafhankelijk is van *n*, wat wil zeggen dat de zoektijd niet stijgt met de omvang van de zoekruimte. We zeggen dat het algoritme van O(0) is. Dit is aantrekkelijk voor zeer grote gegevensbanken. De zoektijd hangt enkel af van de tijd om de functie uit te rekenen. De functie mag dan ook niet te ingewikkeld zijn.



In dit eerste voorbeeldje slaan we drie strings op. De waarde van de hashfunctie is voor elke string gegeven. Rechts zien we de plaats waar de strings opgeslagen worden in de array.

Elke mogelijke sleutelwaarde stemt overeen met een unieke indexwaarde, maar niets kan garanderen dat er geen twee verschillende sleutels tot dezelfde indexwaarde zullen leiden. Wanneer dit het geval is, spreekt men van een ***botsing***. Uiteraard is het wenselijk dat de gebruikte sleutels zo min mogelijk aanleiding tot botsingen geven. Stel dat "jana" dezelfde hashwaarde geeft als "java", namelijk 3. Dit geeft een probleem omdat het vakje met index 3 al is ingenomen. Oplossingen hiervoor bespreken we in de volgende sectie.



Laat ons eerst een hashfunctie uitwerken.
We kunnen de eerste letter nemen van elk woord ('a' = 0; 'b' = 1; ...). Dan zal de index tussen 0 en 25 liggen. Als we de eerste 2 letters van elk woord volgens de volgende formule:

```
hashsom(woord) = index(1e letter) * 26 + index(2e letter)
```

Dan krijgen we een index tussen 0 ("aa") en $25*26+25=26^2-1$ ("zz"):

- 'aa' geeft 0
- 'ab' geeft 1
- 'ba' geeft 26
- 'zz' geeft $25*26+25=26^2-1$

Merk op dat we de eerste letter vermenigvuldigen met een factor zodat die de meeste impact heeft. Alle woorden die beginnen met dezelfde waarde zitten dus samen. De array moet dus een grootte van 26^2 hebben. Groter heeft geen zin, want het einde van array wordt dan niet gebruikt. Meestal wordt een maximale hashwaarde genomen die groter is dan de arraygrootte en vervolgens neemt men de modulo:

```
hashwaarde = hashsom modulo arraygrootte.
```

De modulo zorgt ervoor dat de index binnen de grootte van de array valt.

Voor alfabetische sleutels zal men meestal de ASCII-code van de karakters nemen (zie deel III einde hoofdstuk 1). Elk karakter (niet alleen de 26 letters van het alfabet,

maar ook leestekens en cijfers) wordt zo met 7 bits beschreven. Voor een woord plak je alle bits achter elkaar en neemt vervolgens de modulo.

In het voorbeeld komt de botsing van “java” en “jana” doordat we enkel de eerste twee letters van het woord gebruikt hebben om de hashwaarde te bepalen. Het is dus beter dat *alle* karakters van de sleutel een invloed hebben op de waarde van de maalt index. Dit kan als volgt: ga alle letters af en bereken de totale waarde:

```
hashsom(woord) += index(ide letter) * 26i
```

Er kan wel een probleem optreden. Laat ons de hashwaarde met bvb 8 bits uitdrukken en stel dat de arraygrootte n een macht van 2 is. Dan zijn enkel de $\log_2 n$ minst-betekenisvolle bits van de numerieke sleutel bepalend zijn voor de indexwaarde. Immers, door de *MOD* $\log_2 n$ operatie zullen alle bits groter dan n niet meetellen in de hashwaarde. Stel dat onze sleutels bestaan uit 8 bits (waarden tussen 0 en 255) en dat n=16. Dan: $\log_2 n$ is 4 wat betekent dat enkel de 4 rechtse bits gebruikt worden. De hashwaarden 0000110, 01010110, 10000110 en 11110110 geven allen dezelfde index, namelijk 0110, want de modulo van een macht van 2 resulteert in het ‘afkappen’ van linkse bits. De vier meest-betekenisvolle bits worden dus niet gebruikt! Dit is het geval in ons voorbeeld. **Het is dus beter om *n* een priemgetal te nemen.**

In het algemeen wensen we te bekomen dat de hashwaarde een kans op botsing geeft evenredig met de vullingsgraad van de array. Als de array voor p% gevuld, moet de kans op botsing p% zijn. In ons geval is dat niet, voor sommige beginletters zijn er meer woorden dan voor andere, omdat de beginletter de regio van de array bepaalt. Die vakken zullen dus meer gevuld zijn.

Enkel wanneer de volledige verzameling sleutels a priori bekend is, kunnen we in plaats van de modulo-hashfunctie, andere, complexere functies aanwenden die het voordeel hebben geen botsingen te veroorzaken (**perfect hashing**). Dit wordt bijvoorbeeld gebuikt in een compiler om de gereserveerde woorden (zoals *class*, *public*, *int*, …) snel te herkennen.

Merk op dat Java automatisch een hashfunctie definieert voor elke klasse. Deze staat gedefinieerd in de **Object**-klasse als:

```
int hashCode()
Returns a hash code value for the object.
```

Ze neemt de waarden van alle attributen in rekening. Omdat alle Javaklassen automatisch een subklasse zijn van Object, hebben alle klassen deze methode. Je kan ze overschrijven zodat je eigen, betere hashfunctie gebruikt wordt.

Examenvraag juni 2016: Leg uit hoe een hashtabel werkt. Wat is een goede hashfunctie? Stel een goede hashfunctie op voor het opslaan van alle Belgische adressen (met straat, huisnummer, postcode en gemeente).

i. Hoe worden botsingen opgelost? Wat zijn de nadelen van hashing?

Oplossing.

- De maximale hashwaarde moet minstens 10 miljoen zijn (het aantal adressen van België), maar mag gerust meer zijn gezien de modulo.
- Als grootte van de array nemen we een priemgetal groter dan 10 miljoen.
- We wensen verschillende hashwaarden voor ongelijke adressen, dus gebruiken we alle onderdelen van het adres in de berekening van de hashwaarde.

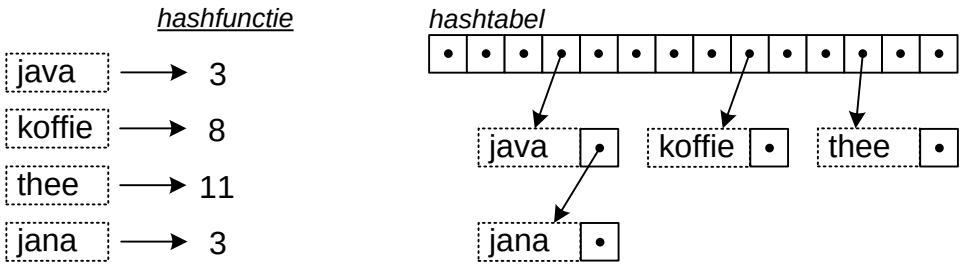
Goede hashfunctie (stringcode hebben we hierboven gedefinieerd):
100*stringcode(straat)+10*huisnummer+postbus+10^5*postcode

Het laatste deel van de vraag behandelen we in de volgende sectie.

9.2. Oplossen van botsingen

Enkel bij perfect hashing komen geen botsingen voor, maar in het algemeen moeten we van uit gaan dat meerdere records op hetzelfde adres opgeslagen moeten worden.

Dit kan door de elementen op te slaan in gelinkte lijsten (Hoofstuk 6)! In die gelinkte lijsten plaats je dan alle objecten waarvan de sleutel naar eenzelfde index leidt.



Door in elke link expliciet de sleutel te vermelden kan je in de gelinkte lijst zoeken naar het gewenste record. Maar dergelijke gelinkte lijsten met expliciete pointers zijn niet bevorderlijk voor de snelheid omdat elke toegang tot gegevens voorafgegaan wordt door minstens één toegang tot pointers. Om die reden wordt er gekozen voor een tweede oplossing waarin geen pointers nodig zijn.

Een tweede oplossing is het berekenen van een tweede hashwaarde met de hoop dat dit vakje nog vrij is. Meestal werkt men met *lineaire* of *kwadratische open adressering*. Hierbij worden de gegevensrecords rechtstreeks in de geïndexeerde array geplaatst, maar, omdat er zich een botsing voordoet, wordt een alternatieve index berekend. De indices die opeenvolgend geprobeerd worden, zijn bij lineaire open adressering:

$h_i = (hash(sleutel) + i)\ MOD\ n$

en bij kwadratische open adressering:

$h_i = (hash(sleutel) + i^2)\ MOD\ n$

waarbij h_0 de initiële index is, *hash(sleutel)* de normale hashfunctie en h_i de index die geprobeerd wordt na de *i*-de botsing. Ter illustratie toont de volgende tabel wat er gebeurt in een initieel lege hashing tabel met 1001 plaatsen waarin opeenvolgend de sleutels 4215, 1212, 7220 en 8219 worden opgenomen. De gebruikte indices worden zowel voor lineaire als voor kwadratische open adressering getoond. *Geschrapte indexwaardes komen overeen met de al bezette plaatsen*.

Key	h0 = Hash(Key)	$h_1 = h_0 + 1$	$h_2 = h_0 + 2$	$h_3 = h_0 + 3$ (lineair)	$h_2 = h_0 + 2^2$ (kwadratisch)
4215	211				
1212	~~211~~	212			
7220	213				
8219	~~211~~	~~212~~	~~213~~	214	215

Bij lineaire open adressering hebben de bezette plaatsen de neiging om zich in blokken te concentreren. Dit heeft tot gevolg dat het gemiddeld aantal pogingen per zoekoperatie vaak groter is dan bij een kwadratische open adressering. Anderzijds, als de gegevens opgeslagen zijn op een schijfgeheugen is het qua toegangstijd juist voordelig wanneer er opeenvolgende zoekoperaties dichtbij elkaar plaatsvinden. Daarom kan lineaire open adressering aantrekkelijk zijn op schijf terwijl het kwadratische beter is in het centrale geheugen.

9.3. Voor- en nadelen van hashing

Het voornaamste voordeel van hashing is zonder twijfel de snelheid. Het vinden van een element is onafhankelijk van de grootte van de lijst! Het uitrekenen van de hashfunctie neemt natuurlijk ook tijd in beslag en moet dus beperkt worden.

Maar hashing heeft ook enkele belangrijke nadelen. Het belangrijkste nadeel is het statische karakter van de datastructuur. De omvang van de tabel waarin de gegevens worden opgeslagen, moet op voorhand bekend zijn en kan onmogelijk worden uitgebreid zonder alle adressen opnieuw te berekenen!
Een tweede nadeel is het feit dat de gegevens in een willekeurige orde opgeslagen zijn. Om een bruikbare lijst van de opgeslagen gegevens uit te printen, moeten alle gegevens worden geordend.
Een laatste nadeel is de enorme moeilijkheid waarmee gegevens uit een hashtabel kunnen worden verwijderd. Dit obstakel wordt eenvoudig omzeild door de objecten die moeten worden verwijderd te markeren met een booleaanse vlag.

In de *Object*-klasse van Java vinden we de volgende functie terug:

```
public int hashCode();
```

Deze functie geeft de hashcode van het object terug. En omdat *Object* de oermoeder is van alle klasses – alle klasses erven automatisch van deze klasse - krijgt elk object standaard een hashcode toegewezen. Handig, je kan dus steeds een hashtabel gebruiken. Niets garandeert echter dat het een goede hashcode is. Soms is het beter om zelf een code op te stellen. Je *overschrijft* dan bovenstaande methode. In de **package *datastructuren*** vind je de **klasse *Hashing*** die voorbeelden toont van de default Java hashcode voor verschillende gegevens.

9.4. Map- en Setimplementaties

We hebben de definitie van een Map al geïntroduceerd. De Map-interface definieert hoe je met een Map werkt, maar voorziet geen implementatie. Het uitwerken van de achterliggende datastructuur en de operaties kan op verschillende manieren. Hiervan bespreken we er twee. Dit is een belangrijke les: enerzijds definieer je de *interface* met het object; de dingen die je met het object moet kunnen doen. Anderzijds voorzie je een *implementatie*. Zo'n implementatie kan op verschillende manieren gebeuren. Maar als gebruiker trek je je daar niets van aan omdat je enkel via de interface met het object communiceert.
Omdat je via de *key* de *value* opzoekt, moet het vinden van een *key* snel gebeuren. Dit hebben we in de vorige hoofdstukken besproken. Twee flexibele en snelle manieren zijn de binaire boom en de hashtabel. Beide geven een oplossing. Deze zijn dan ook allebei in Java voorzien: *TreeMap* en *HashMap*. De eerste slaat de keys geordend op in een binaire boom terwijl de tweede een hashtabel gebruikt. Voor dit laatste gebruikt java de *hashCode()*-methode als hashfunctie (zie einde 9.1).

Definiëren doen we zo:

```
Map<String, String> map = new TreeMap<String, String>();
```

Of

```
Map<String, String> map = new HashMap<String, String>();
```

We declareren de variabele als zijnde een Map:

```
Map<String, String> map;
```

maar we maken een TreeMap- of een HashMap-object aan:

```
map = new TreeMap<String, String>();
```

of

```
map = new HashMap<String, String>();
```

We moeten kiezen voor een implementatie, maar in het gebruik van het object werken we met de methodes gedefinieerd door de Map.
De Map definieert als het ware het 'servicecontract'. Ze bepaalt de 'services' die een map-object aanbiedt. In de documentatie staat eenduidig wat het object zal doen. Elke implementatie zal beloven hieraan te voldoen. Als gebruiker kun je nu op beide oren slapen.

Bij de implementatie van een verzameling (set) is het snel terugvinden van elementen ook belangrijk. Men moet immers nagaan of een element reeds aanwezig is. Hiervoor kan eveneens een hashtabel of binaire boom gebruikt worden (HashSet of HashTree).

9.5. Hashfuncties in authenticatie en security.

Hashfuncties worden ook voor andere doeleinden gebruikt. Bijvoorbeeld om na te gaan of een tekst (of file) veranderd is. Je neemt de hashwaarde van de tekst en houdt die bij. Op een later tijdstip kan je nagaan of het nog steeds over dezelfde tekst gaat door na te gaan of zijn hashwaarde veranderd is. Hiervoor is het belangrijk dat alle elementen van de tekst meetellen in de hashfunctie.
Het kan natuurlijk dat een aantal veranderingen toch dezelfde hashwaarde opleveren. Als we willen voorkomen dat iemand dit opzettelijk kan doen, worden complexe hashfuncties (gebaseerd op grote priemgetallen!) genomen voor dewelke het heel moeilijk te achterhalen valt welke veranderingen moeten gebeuren om dezelfde hashwaarde te bekomen. Deze techniek wordt gebruikt voor digitaal ondertekende documenten.

Een andere toepassing is het opslaan en verifiëren van paswoorden. We willen niet dat een paswoord achterhaald kan worden, en dus wordt het liever niet letterlijk opgeslagen. Dit kan met zogenaamde *trapdoorfuncties*. Dat zijn functies die in één richting snel uit te rekenen, maar niet in de andere richting (de inverse). Complexe hashfuncties voldoen aan die eigenschap. Een paswoord wordt aan de trapdoorfunctie gegeven en de uitkomst (een soort hashwaarde) wordt opgeslagen, niet het paswoord zelf. Als we een paswoord willen verifiëren, wordt de ingegeven waarde ook door de functie gehaald en vergeleken met de opgeslagen waarde. Merk op dat het ingegeven paswoord in een veilige omgeving moet zitten zodat ze niet te lezen is door hackers. Zo zal de communicatie geëncrypteerd worden vooraleer te worden doorgestuurd over het internet. Een website die begint met ‘https’ in plaats van ‘http’ gebruikt geëncrypteerde communicatie die niet te onderscheppen valt.