ALGORITMES & DATASTUCTUREN

Joost Visser (jwjh.visser@avans.nl)

Academie voor Deeltijd

Les opnemen

Deze les wordt opgenomen, heeft hier iemand bezwaar tegen?

Les van vandaag

1. **Vakintroductie**
2. Wat zijn algoritmes en datastructuren?
3. Voorbeeld 1: Linear en binair zoeken
4. Tijds- en geheugencomplexiteit

Les van vandaag

1. **Vakintroductie**
    1. Even voorstellen
    2. Leeruitkomsten
    3. Toetscriteria
    4. Bronnen
2. Wat zijn algoritmes en datastructuren?
3. Voorbeeld 1: Linear en binair zoeken
4. Tijds- en geheugencomplexiteit
5. Voorbeeld 2: Sorteren

1. Vakintroductie

Even voorstellen

- *Naam:* Joost Visser
- *Leeftijd:* 25 jaar
- *Studie:* Computer Science master aan de TU/e.
- *Werkervaring*: Full-time algoritme developer bij SmartQare.
- *Specialisaties:* AI en algoritmes.

Af en toe heb ik meegedaan aan Europese algoritmewedstrijden (in Zweden en Engeland). Eén keer zelfs eerste van Nederland geworden.

Contact? → Kan via Teams, Email of WhatsApp.

Vragen tijdens de les? → Stel ze in de Teamschat

1. Vakintroductie

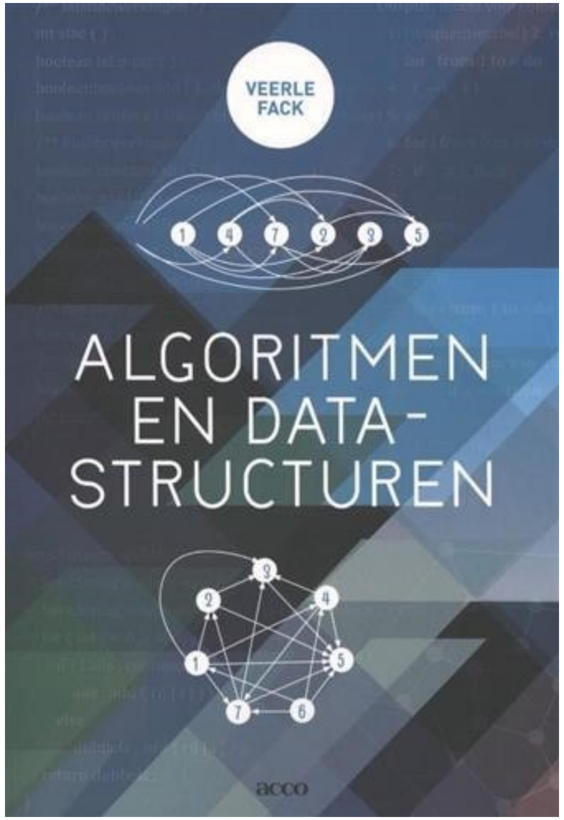
Leeruitkomsten

1. Je kan het juiste datastructuur kiezen bij specifieke situaties.
   - Deze komen uit het Collections Framework in Java.
2. Je bepaalt de rekencomplexiteit en geheugencomplexiteit van een algoritme.
   - Bijvoorbeeld: de algoritme heeft een *worst-case* uitvoeringstijd van $O(n \log n)$.
3. Je kan je eigen datastructuur ontwikkelen met generieke typen.
   - `AnimalList<Platypus> platty = new AnimalList<>();`
4. Je begrijpt standaard sorteeralgoritmes, binaire bomen en recursie.

1. Vakintroductie

Bronnen

1. Deze slides
2. [LinkedIn Learning: Introductions to Data Structures & Algorithms]
3. [TutorialPoint]
4. Algoritmen en Datastructuren, geschreven door Veerle Fack
5. Internet! Stackexchange
6. [Kattis]



1. Vakintroductie

Beoordeling: Toetscriteria

1. De meest voorkomende constructies van programmeertalen begrijpen en kunnen toepassen.
2. Bepalen van rekencomplexiteit en geheugencomplexiteit van algoritmes.
3. Begrijpen en toepassen van recursie.
4. Abstracte datastructuren en algoritmen kunnen kiezen en toepassen in praktische situaties (array, list, stack, queue, set, map, tree, hashing)
5. Generieke datatypen en methodes begrijpen en toepassen

1. Vakintroductie

Toets

Je wordt volledig beoordeeld aan de hand van de toets.

De toets zal uit twee delen bestaan:

1. Theorievragen (50%)
2. Praktijkopdracht (50%)

De toets van vorig jaar staat online als voorbeeld.

In tegenstelling tot vorig jaar kan de toets dit jaar **alleen in Java** worden gemaakt.

De exacte details van de toets worden in les 4 behandeld.

1. Vakintroductie

Lesplan

Les	Datum	Topic	Inhoud
1	4 mei 2021	Introductie	Inleiding, algoritme complexiteit, data-structuren, Linear zoeken, binair zoeken, Sorteren 1
2	11 mei 2021	Datastructuren 1	Abstracte datatypen, Array, List, Set, Map
3	18 mei 2021	Datastructuren 2	Stack, Heap, Deque, Priority Queue, Generieke datatypen
4	25 mei 2021	Algoritmes	Sorteren 2, recursie, binaire bomen, proeftoets

Vragen?

Les van vandaag

1. Vakintroductie
2. **Algoritmes en datastructuren**
    1. Wat is een algoritme?
    2. Wat is een datastructuur?
    3. Java datastructuren
3. Voorbeeld 1: Linear en binair zoeken
4. Tijds- en geheugencomplexiteit
5. Voorbeeld 2: Sorteren

2. Algoritmes en datastructuren

Wat is een algoritme?

Een algoritme is een recept voor het oplossen van een probleem, stap voor stap.

Verzin een algoritme voor de volgende problemen:

1. Gegeven twee nummers, tel ze bij elkaar op.
2. Zit de volgende getal in een gegeven lijst?
3. Wat is de snelste route van mijn huis naar Avans?

Vaak wil je iets van data tijdelijk opslaan voor je calculaties. Hiervoor kan je datastructuren gebruiken; een structuur om je data op te slaan.

2. Algoritmes en datastructuren

Wat is een datastructuur?

Vaak wil je iets van data tijdelijk opslaan voor je calculaties. Hiervoor kan je datastructuren gebruiken; een structuur om je data op te slaan.

Het simpelste is bijvoorbeeld om een lijst met getallen bij te houden, ook wel een array genoemd.

```
int [] inputNumbers = new int[20];
```

Echter zijn er ook andere mogelijkheden om een lijst met getallen op te slaan, zoals in een ArrayList, HashSet of zelfs een Binary Search tree.

Welke je kiest hangt af ondere andere af van (1) hoeveel ruimte je hebt, (2) hoe snel je bepaalde acties wilt doen en (3) welke acties je wilt doen.

2. Algoritmes en datastructuren

Wat is een datastructuur?

Vaak wil je iets van data tijdelijk opslaan voor je calculaties. Hiervoor kan je datastructuren gebruiken; een structuur om je data op te slaan.

Het simpelste is bijvoorbeeld om een lijst met getallen bij te houden, ook wel een array genoemd.

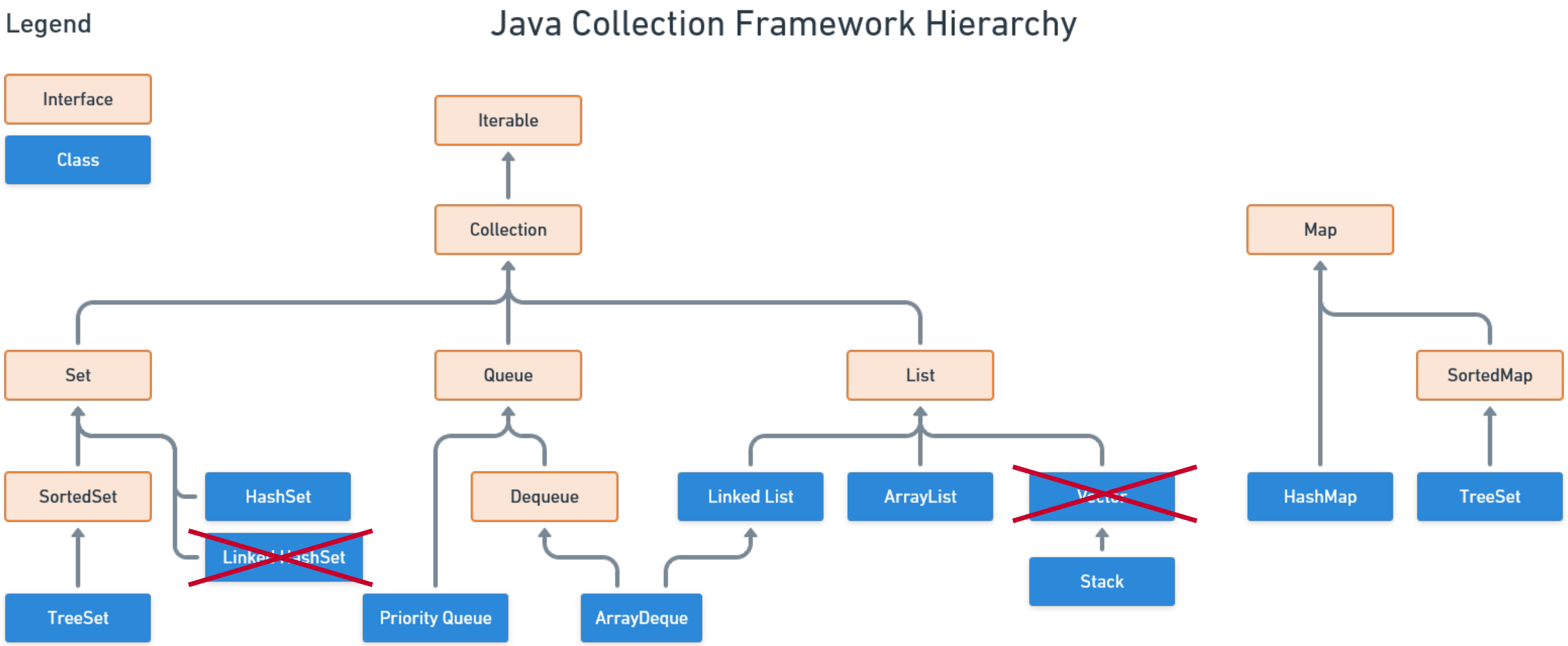
```
int [] inputNumbers = new int[20];
```

In principe kan elke datastructuur de volgende CRUD acties doen:

1. Create → Data maken en toevoegen
2. Read → Data uitlezen
3. Update → Data aanpassen
4. Delete → Data verwijderen

2. Algoritmes en datastructuren

Java datastructuren



Les van vandaag

1. Vakintroductie
2. Wat zijn algoritmes en datastructuren?
3. **Voorbeeld 1: Zoeken**
    1. Wat is het zoekprobleem?
    2. Linear zoeken
    3. Binair zoeken
4. Tijds- en geheugencomplexiteit
5. Voorbeeld 2: Sorteren

3. Voorbeeld 1: Zoeken

Wat is het zoekprobleem?

Zoekprobleem: vind een waarde, zoeksleutel, in een verzameling gegevens

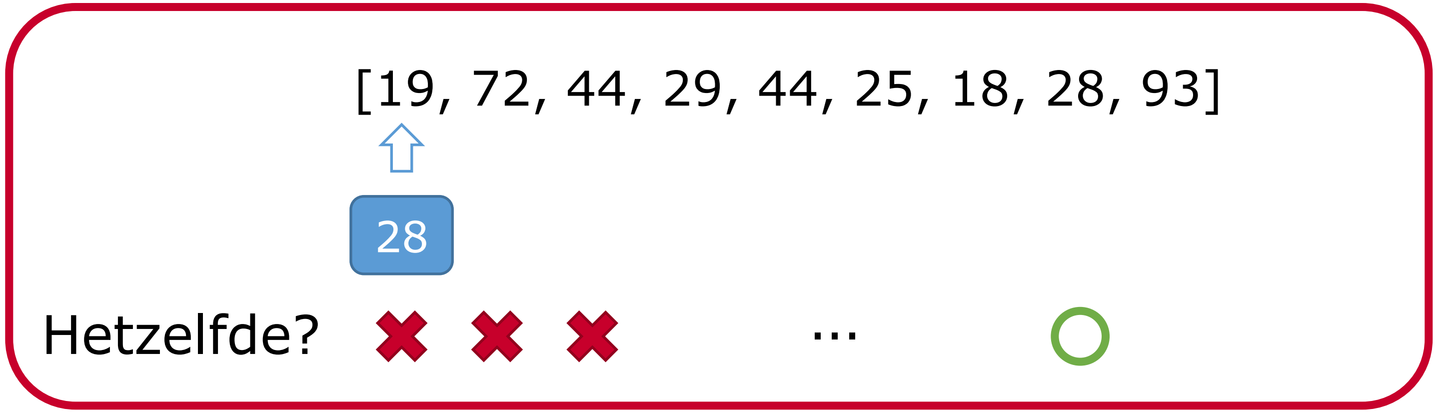
Voorbeelden:

1. Vind 28 in [19, 72, 44, 29, 44, 25, 18, 28, 93]
2. Vind 72 in [18, 19, 25, 28, 29, 44, 44, 72, 93]
3. Vind Henk in [“Piet”, “Jan”, “Katja”, “Annabel”, “Frans”, “Lieke”]

3. Voorbeeld 1: Zoeken

Linear zoeken

1. Vind 28 in [19, 72, 44, 29, 44, 25, 18, 28, 93]



Hoe *snel* is dit algoritme?

… hoe meet je de snelheid van een algoritme?

3. Voorbeeld 1: Zoeken

Linear zoeken

1. Vind 28 in [19, 72, 44, 29, 44, 25, 18, 28, 93]

Hoe *snel* is dit algoritme?

… hoe meet je de snelheid van een algoritme?

1. In de *worst-case* scenario, hoe snel is het algoritme?
2. Neem aan dat één comparison ‘1’ tijdseenheid kost.

$$\boxed{T = 9}$$

3. Wat nou als de lijst een aritraire lengte n heeft?

$$\boxed{T(n) = n}$$

3. Voorbeeld 1: Zoeken

Linear zoeken

1. Vind 28 in [19, 72, 44, 29, 44, 25, 18, 28, 93]

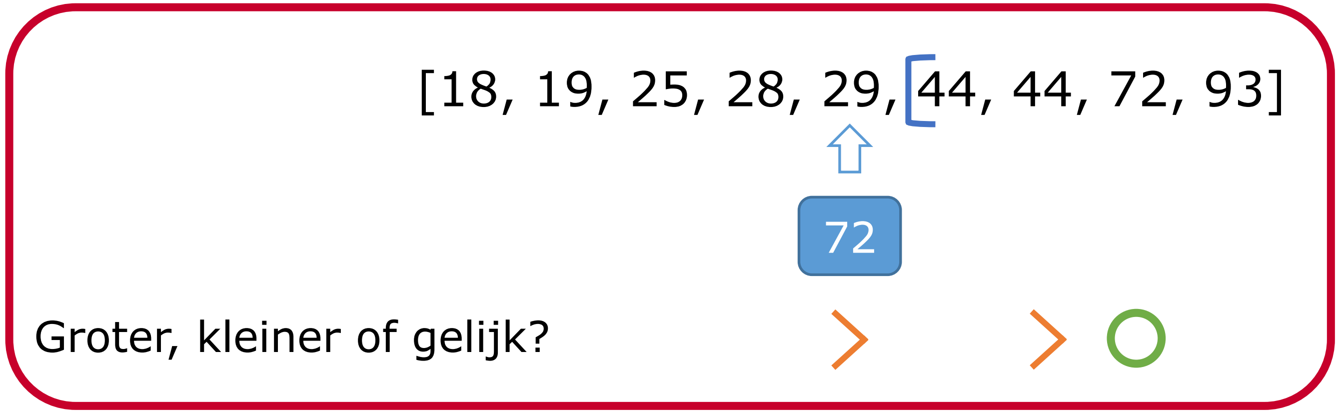
```java
/**
 * Finds a number in an unordered list.
 * Returns index or -1 if not found.
 */
public int findNumber(int list[], int number) {
    for (int i = 0; i < list.length; i++) {
        if (list[i] == number) {
            return i;
        }
    }
    return -1;
}
```

3. Voorbeeld 1: Zoeken

Binair zoeken

2. Vind 72 in [18, 19, 25, 28, 29, 44, 44, 72, 93]

Iemand een idee voor een snellere algoritme dan linear zoeken?



Hoe weten we of het getal er niet in zit?

In de *worst-case* scenario, hoe snel is dit algoritme?

3. Voorbeeld 1: Zoeken

Binair zoeken

2. Vind 72 in [18, 19, 25, 28, 29, 44, 44, 72, 93]

Iemand een idee voor een snellere algoritme dan linear zoeken?

[18, 19, 25, 28, 29, 44, 44, [72, 93]

72

Groter, kleiner of gelijk?

We halveren elke keer onze *range*, zodra die 1 is weten we het antwoord.

$$T(n) = \log_2 n$$

Hoe weten we of het getal er niet in zit?

In de *worst-case* scenario, hoe snel is dit algoritme?

Note: dit is geen formeel bewijs.
Formele bewijzen vallen buiten de scope van dit vak.
Bekijk het boek voor een formelere bewijs.

3. Voorbeeld 1: Zoeken

Binair zoeken

2. Vind 72 in [18, 19, 25, 28, 29, 44, 44, 72, 93]

Logaritme intermezzo!

$2^x = 1024$
$x = \log_2 1024 = 10$

$$a^x = b \Leftrightarrow b = \log_a x$$

Omdat logaritme met *base 2* enorm vaak voorkomen in Computer Science, laten we de 2 in $\log_2$ weg.

De running time wordt dan $T(n) = \log n$

We halveren elke keer onze *range*, zodra die 1 is weten we het antwoord.

$$T(n) = \log n$$

Note: dit is geen formeel bewijs.
Formele bewijzen vallen buiten de scope van dit vak.
Bekijk het boek voor een formelere bewijs.

3. Voorbeeld 1: Zoeken

Binair zoeken

2. Vind 72 in [18, 19, 25, 28, 29, 44, 44, 72, 93]

Hoeveel scheelt het qua snelheid?

n	Linear zoeken	Binair zoeken
10	10 microseconde	3.32 microseconde
1 000	1 milliseconde	9.97 microseconde
1 000 000	1 seconde	19.93 microseconde
1 000 000 000	16.67 minuten	29.90 microseconde
1 000 000 000 000	11.57 **dagen**	39.86 microseconde
1e18	31.71 miljard jaar	59.79 microseconde

Handige praktijktip: een computer doet ongeveer 1 000 000 computations per seconde

3. Voorbeeld 1: Zoeken

Opdrachten

Opdracht 1: Implementeer Lineair zoeken voor Strings.

Input: ["Piet", "Jan", "Katja", "Annabel", "Frans", "Lieke"]

Output: Index van “Piet”

Opdracht 2: Implementeer Binair zoeken.

Input: [18, 19, 25, 28, 29, 44, 44, 72, 93]

Output: Index van 72

Opdracht 3: Implementeer Binair zoeken met Strings.

Input: ["Annabel", "Franks", "Jan", "Katja", "Lieke", "Piet“];

Output: Index van “Piet”

Tip: Gebruik `string1.compareTo(string2)`

Les van vandaag

1. Vakintroductie
2. Algoritmes en datastructuren
3. Voorbeeld 1: Linear en binair zoeken
4. **Tijds- en geheugencomplexiteit**
    1. Tijdscomplexiteit
    2. Big-O notatie
    3. Big-O notatie in de praktijk
    4. $\Omega(n^2)$ en $\Theta(\mathrm{n}^2)$
    5. Geheugencomplexiteit
5. Voorbeeld 2: Sorteren

4. Tijds- en geheugencomplexiteit

Tijdscomplexiteit

Hoe snel is een algoritme?

- Lineaire search $\rightarrow T(n) = n$
- Binaire search $\rightarrow T(n) = \log n$

Dit was **met** de aanname dat één operatie '1' tijd kostte.

Hoe kunnen we dit wat wiskundig netter oplossen?

4. Tijds- en geheugencomplexiteit

Big O-notatie

Met een asymptotische analyze!

Stel we hebben een running time functie van:

$$T(n) = 10n^3 + n^2 + 40n + 80$$

Voor $n = 1000$ is $T(n) = 10\,001\,040\,080$

- Waarvan $10\,000\,000\,000$ komt door de $10n^3$

We zijn vaak alleen geïntereseerd in de hoogste macht van n, omdat voor een hogere n deze vaak de volledige running time bepaalt.

$$\boxed{T(n) = O(n^3)}$$

4. Tijds- en geheugencomplexiteit

Big O-notatie

$$\boxed{T(n) = O(n^3)}$$

De bovengrens van de orde van toename is n^3.

- Als er 10 extra samples komen, dan groeit de running time met $10^3 = 1000$ operaties.

De wiskundige definitie is dat er een constante c bestaat zodat:

$$0 \leq T(n) \leq cn^3$$

- Voor n die groot genoeg is: $n > n_0$

Een aantal voorbeelden:

1. $O(n)$ is sneller dan $O(n \log n)$ (voor grote n, worst-case)
2. $O(n^2)$ is sneller dan $O(n^3)$ (voor grote n, worst-case)
3. $O(n^{10})$ is sneller dan $O(2^n)$ (voor grote n, worst-case)

4. Tijds- en geheugencomplexiteit

Big O-notatie

Waar of niet waar?

1. Een $O(n^2)$ algoritme is **altijd** langzamer dan $O(n \log n)$.
2. Orde $O(n\sqrt{n})$ is langzamer dan $O(n \log n)$ voor grote n in de worst-case.
3. Met $n = 2\,000\,000$ samples kan je makkelijk een $O(n^2)$ algoritme gebruiken.

Eigenlijk $\Theta(\mathrm{n})$ in plaats van $O(n)$; daar komen we straks op.

4. Tijds- en geheugencomplexiteit

Big O-notatie

Waar of niet waar?

1. Een $O(n^2)$ algoritme is **altijd** langzamer dan $O(n \log n)$.
   - Niet waar → Voor kleine n kan $O(n^2)$ sneller zijn dan $O(n \log n)$
   - Niet waar → $O(n^2)$ is in de *worst-case* langzamer, maar kan gemiddeld genomen zelfs sneller zijn dan $O(n \log n)$.
2. Een $O(n\sqrt{n})$ algoritme is langzamer dan $O(n \log n)$ voor grote n in de *worst-case*.
   - Waar → $\sqrt{n}$ groeit langzamer dan $\log n$
3. Met $n = 2\,000\,000$ samples kan je makkelijk een $O(n^2)$ algoritme gebruiken.
   - Niet waar → Als je aanneemt van $1\,000\,000$ computations per seconde, dan zou dit $2\,000\,000^2 /\ 1\,000\,000 = 4\,000\,000$ seconde duren, oftewel 46.3 dagen.

Eigenlijk $\Theta(\mathrm{n})$ in plaats van $O(n)$; daar komen we straks op.

4. Tijds- en geheugencomplexiteit

Big-O notatie in de praktijk

Hoe snel is een algoritme?

- Lineaire search $\rightarrow T(n) = n$
- Binaire search $\rightarrow T(n) = \log n$

Dit was **met** de aanname dat één operatie '1' tijd kostte.

$O(1)$ operatie

Hoe kunnen we dit wat wiskundig netter oplossen?

$$T(n) = O(n)$$

4. Tijds- en geheugencomplexiteit

Big-O notatie in de praktijk

Hoe bereken je de complexiteit van Lineair Zoeken?

```
/**
 * Finds a number in an unordered list.
 * Returns index or -1 if not found.
 */
public int findNumber(int list[], int number) {
    for (int i = 0; i < list.length; i++) {
        if (list[i] == number) {
            return i;
        }
    }
    return -1;
}
```

$O(n)$

$O(1)$

$O(1)$

$$T(n) = O(n) \cdot O(1) + O(1) = O(n)$$

4. Tijds- en geheugencomplexiteit

Big-O notatie in de praktijk

Hoe snel is het volgende algoritme?

```
public int findNumber(int list[], int number) {
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            step(i, j);
        }
    }
    return -1;
}
```

$$\boxed{T(n) = O(n^2)}$$

4. Tijds- en geheugencomplexiteit

Big-O notatie in de praktijk — oefeningen

Oefening 1

```
for (int i = 0; i < n; i++) {
    for (int j = 0; j < 100; j++) {
        step(i, j); // O(1)
    }
}
```

$O(n)$

Oefening 2

```
for (int i = 0; i < n; i++) {
    for (int j = 0; j < i; j++) {
        step(i, j); // O(1)
    }
}
```

$O(n^2)$

Nog vragen?

Oefening 3

```
for (int i = 0; i < n; i++) {
    for (int j = 0; j < K; j++) {
        step(i, j); // O(1)
    }
}
```

$O(Kn)$

Oefening 4

```
for (int i = 0; i < n; i++) {
    doSomething(i); // O(1)
}
for (int j = 0; j < K; j++) {
    doSomething(j); // O(1)
}
```

$O(n + K)$

4. Tijds- en geheugencomplexiteit

$\Omega(n^2)$ *en* $\Theta(\mathrm{n}^2)$

Bij asympotische tijdsanalyze zie je ook af en toe Ω en Θ naast O:

1. Orde $O(n^2) \rightarrow$ Bovengrens van de orde van toename is n^2
   - "Het is maximaal orde n^2"
2. Orde $\Omega(n^2) \rightarrow$ Ondergrens van de orde van toename is n^2
   - "Het is minimaal orde n^2"
3. Orde $\Theta(n^2) \rightarrow$ Boven- en ondergrens van de orde van toename is n^2
   - "Het algoritme heeft orde n^2"

4. Tijds- en geheugencomplexiteit

Geheugencomplexiteit

Net als dat tijd een belangerijke resource is, kan geheugen soms ook een probleem leveren.

We kunnen op eenzelfde manier de tijd noteren:

$$\boxed{G(n) = O(n)}$$

Als we bijvoorbeeld een array van n getallen willen bijhouden, dan is ons geheugenverbruik $O(n)$ (en $\theta(n)$, maar werken met big-O is makkelijker).

4. Tijds- en geheugencomplexiteit

Geheugencomplexiteit

```java
/**
 * Finds a number in an unordered list.
 * Returns index or -1 if not found.
 */
public int findNumber(int list[], int number) {
    for (int i = 0; i < list.length; i++) {
        if (list[i] == number) {
            return i;
        }
    }
    return -1;
}
```

Wat is het geheugengebruik van dit algoritme?

Wat is het extra geheugengebruik van dit algoritme?

$$G(n) = O(n)$$

$$G(n) = O(1)$$

4. Tijds- en geheugencomplexiteit

Samenvatting

Om de snelheid van een algoritme te berekenen, gebruiken we de big-O notatie, wat de bovengrens van de orde van toename is:

1. Relatief makkelijke manier om de algoritmesnelheid in te schatten
2. Kan schatten hoe groot n, de verzamelingsgrootte, mag zijn
3. Kan snelheid tussen algoritmes vergelijken

$$\boxed{T(n) = O(\log n)}$$

Voor een aantal voorbeelden hebben we deze snelheid bepaald.

Het geheugengebruik kan op eenzelfde manier berekend worden

$$\boxed{G(n) = O(n)}$$

Les van vandaag

1. Vakintroductie
2. Algoritmes en datastructuren
3. Voorbeeld 1: Linear en binair zoeken
4. Tijds- en geheugencomplexiteit

Les van vandaag

Samenvatting

Algoritme: recept voor het oplossen van een programmeerprobleem.
Data-structuur: verschillende structuren om je data op te slaan.

Zoekprobleem: vindt een element in een array

1. Linear zoeken $\rightarrow O(n)$ [Gesorteerde en ongesorteerde lijst]
2. Binair zoeken $\rightarrow O(\log n)$ [Alleen gesorteerde lijst]

Voor de uitvoeringstijd van een algoritme gebruiken we de big-O notatie.

- Orde $O(n^2)$ betekent dat in de worst-case en voor grote n de orde van toename n^2 is.
- Dit is equivalent aan *hoogste macht* in de uitvoeringstijd functie $T(n)$.
- $O(2^n) > O(n^3) > O(n^2) > O(n\sqrt{n}) > O(n \log n)$

We gebruiken dezelfde notatie voor het geheugengebruik en extra geheugengebruik.

Opdrachten

Informatie

Oefenfiles voor deze opdrachten staan op Brightspace.

De opdrachten zijn optioneel, maar zeer aangeraden.

- Opdrachten *hoeven niet* ingeleverd te worden, aangezien de antwoorden komen later deze week online. Je kan ze zelf nakijken.
- Bij twijfel kan je je opgave submitten via Brightspace, dan kijk ik ernaar.

Deze week is er één oefenopdracht:

1. Priemgetallen

Opdrachten

Opdracht 1: Priemgetallen

In encryptie worden vaak grote priemgetallen gebruikt. Er wordt gebruik gemaakt van het problem van ontbinden in factoren van een product van twee grote priemgetallen.

Een algoritme voor ontbinden in factoren:

```
// zoek een priemfactor van n
// return de kleinste factor, of n als n priem is
long zoekFactor(long n) {
    for (long i = 2; i < n; i++)
        if (n % i == 0)
            return i;
    return n;
}
```

Opgave delen:

a. Schrijf de *worst-case* uitvoeringstijd van het huidige algoritme op.
b. De $i < n$ kan beter. Verbeter het algoritme. Wat is nu de *worst-case* uitvoeringstijd?
c. Naast de formele uitvoeringstijd, voeg een teller toe die telt hoe vaak de for-loop uitgevoerd wordt voor en na de verbetering. Wat valt op?