**Aantekeningen Algoritmen Ingesproken lessen**

**Complexiteit**

*Voorbeeld:* Een verkoper wilt zo efficiënt mogelijk 4 huizen afgaan, en daarna terugkeren naar zijn eigen huis. Bereken de snelste route.

*Uitwerking:* Alle mogelijke routes opschrijven die de verkoper kan lopen:

1-2-3-4

1-3-4-2

1-4-2-3

1-3-2-4

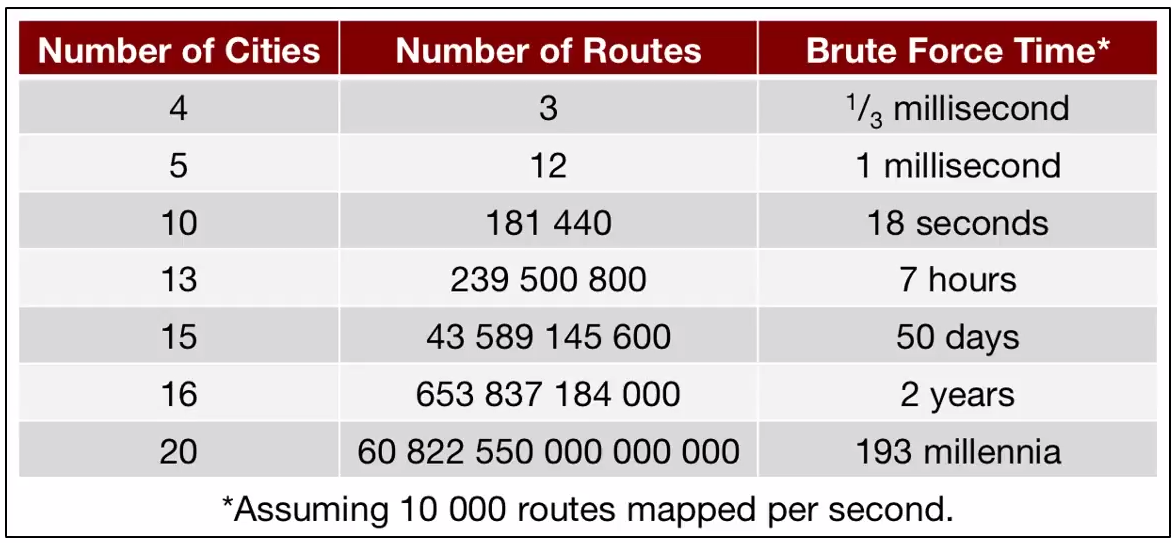
1-4-3-2

Enz.….

Dit komt neer op: **Het aantal huizen x het aantal huizen -1 x aantal huizen -2 …. X 1**

Ofwel: .

Dit is enorm inefficiënt, hoe meer steden, hoe veel meer langer het duurt om alle routes te bedenken



*Number of cities = .*

De heet ook wel de **Grootte van het probleem.**

De *Number of Routes*is het **kostenmodel**. Hoe meer routes, hoe groter het geheugen/kostenmodel.

Hiermee kan je ook de volgende regel trekken:

**De complexiteit van een probleem is de hoeveelheid tijd die een berekening nodig heeft, op basis van (de Grootte van het probleem).**

De complexiteit druk je uit met de volgende “formule”: , waarbij staat voor de complexiteit, en de grootte van het probleem.

**Algoritmen & Datastructuren**

Algoritmen is in de basis een soort recept, of een volgorde van handelingen. **Efficiëntie** staat centraal bij algoritmen.

Een **Datastructuur** is een manier om de data op te slaan of te behandelen in het geheugen. Een datastructuur **Ondersteund** een bepaald algoritme.

**Bekende problemen**

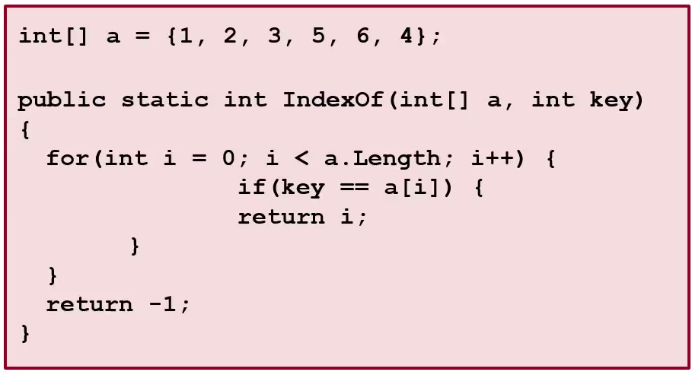
Bekende problemen waar algoritme en datastructuren een oplossing op probeert te vinden zijn:

* Hoe zoek ik efficiënt data?
* Hoe sorteer ik efficiënt data? (Data is immers makkelijker te zoeken als het gesorteerd is)

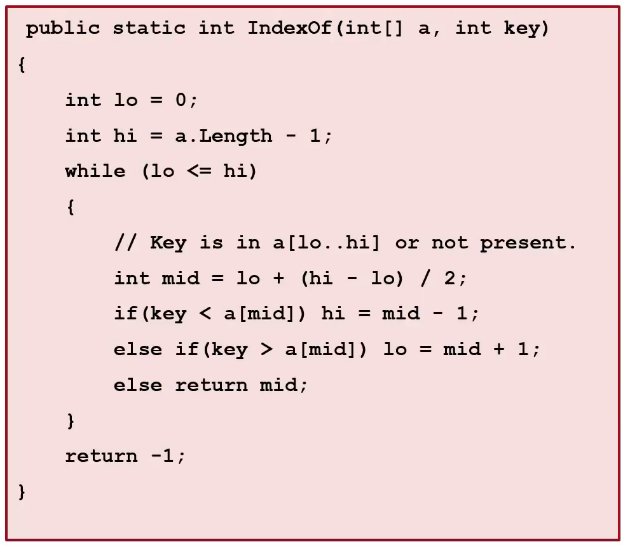
Er zijn veel oplossingen, maar het is belangrijk een **passende oplossing** te vinden voor jouw probleem, omdat elke datastructuur en algoritme zijn eigen specialiteiten en nadelen heeft.

Om de complexiteit van een probleem of stuk code te achterhalen, moet je altijd de volgende vraag stellen:

**Als ik het probleem twee keer zo groot maak, hoeveel keer extra moet ik het werk uitvoeren?**

**Linear Search:**

Lineair zoeken zie je vaak in algoritmen die geen aanname doen van de volgorde van de array. Hierbij wordt elke waarde in de array wordt vergeleken met de zoekopdracht. Is het een match, dan wordt de index teruggegeven, zo nee, verder zoeken. ***Zie SearchUnsortedList.cs***. Typisch hierbij is dat de complexiteit . Het werk moet twee keer zovaak moeten worden uitgevoerd als het grootte van het probleem twee keer zo groot wordt (**lineair verband**)



**Binary Search:**

Een binary Search doet de aanname dat de array gesorteerd is. Door deze aanname kan er efficiënter gezocht worden (***Zie SearchSortedList.cs)***. Hij zoekt in het midden, kijkt of de waarde van de zoekopdracht hoger of lager dan het midden ligt, en kapt dan de helft van de array waar de waarde sowieso niet in zit weg. Voor elke verdubbeling van het probleem, hoef je hierdoor dus maar 1 keer meer werk te leveren (Dit is een **Logaritmisch verband).** Dit is Complexiteit

De tegenhanger van logaritmisch verband is het **exponentieel verband**, ofwel , waarbij als je het probleem 2x vergroot, je keer zoveel extra werk moet leveren.

**Probleem oplossen**

Het maken van een algoritme is lastig, het programmeren hiervan nog wel meer. Een voorbeeld is het berekenen van een priemgetal. Je kan verschillende algoritmen hierop loslaten die verschillende voor en nadelen hebben.

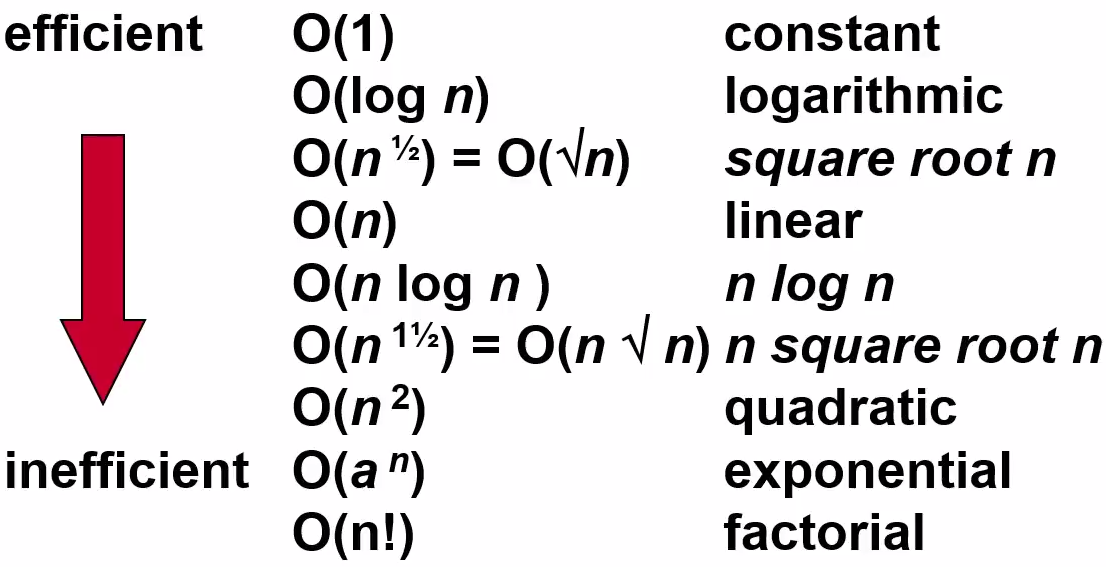
*Voorbeeld 1:*

Elke keer de deler +1 doen. Is getal x deelbaar door 2? Nee => Deelbaar door 3? Nee => Deelbaar door x? Ja => Priemgetal!

**Invloeden op Complexiteit**

Verschillende algoritmes die hetzelfde resultaat behalen, gedragen zich allemaal verschillend. Daarom is het belangrijk om een algoritme te laten passen bij een probleem. **Complexiteit gaat over de efficiëntheid van het algoritme**. Complexiteit wordt uitgedrukt met . Er zijn twee verschillende soorten complexiteit:

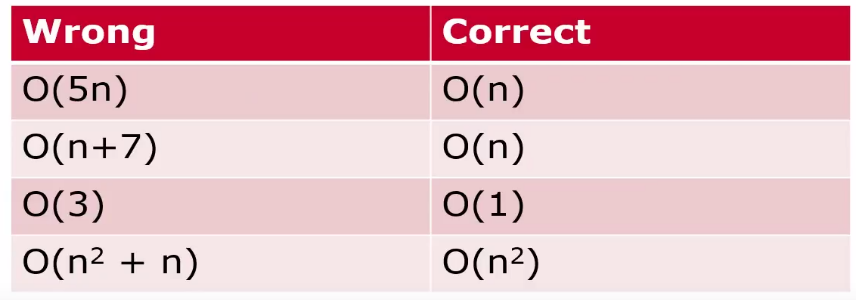
* Geheugencomplexiteit (de hoeveelheid RAM die het opneemt)
* Rekencomplexiteit (de hoeveelheid CPU die het opneemt)
  + Dit is uit te rekenen met **Tijdmetingen**



Bij complexiteit moet je altijd ook de volgende vraag stellen om tot de juiste complexiteit te komen:

***Welk onderdeel van het algoritme heeft de meeste invloed op de complexiteit?***

Je moet hierbij denken aan bijvoorbeeld complexiteit , als extreem groot is, valt de in het niet. Deze heeft dus een **kleine invloed** op de complexiteit en mag dus worden weggelaten.

Een ander voorbeeld is complexiteit . de is een constante. Dit heeft **geen invloed** op de complexiteit van het algoritme. Als je n 2 keer zo groot maakt, blijft de complexiteit lineair.

**Recursie**