**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

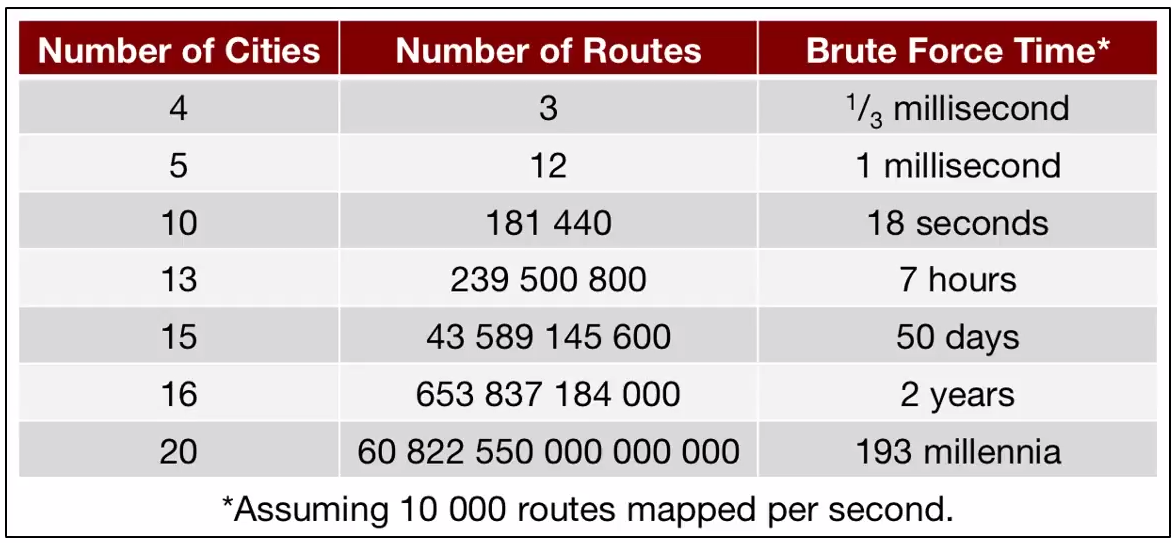
1-4-3-2

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**

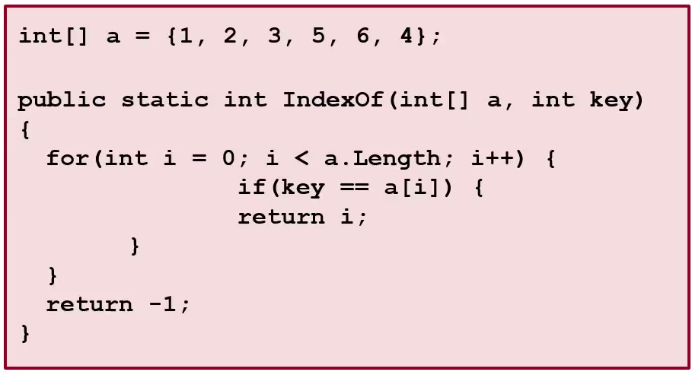
Bekende problemen waar algoritme en datastructuren een oplossing op probeert te vinden zijn:

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

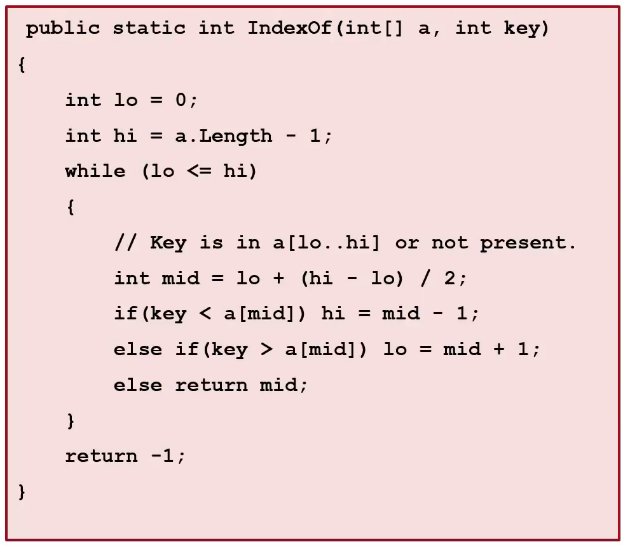
Er zijn veel oplossingen, maar het is belangrijk een **passende oplossing** te vinden voor jouw probleem, omdat elke datastructuur en algoritme zijn eigen specialiteiten en nadelen heeft.

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.

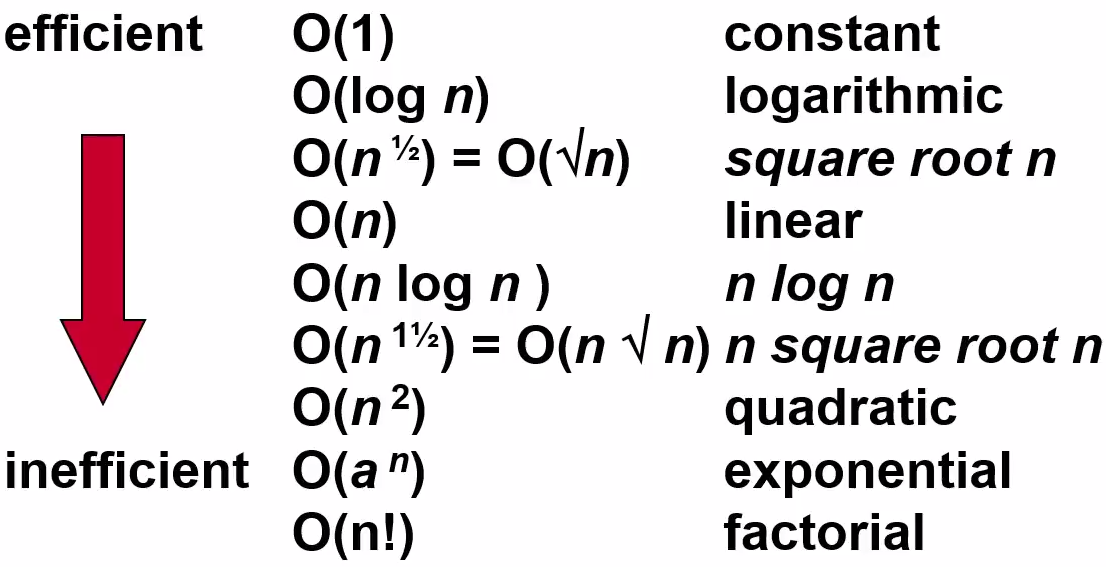
*Voorbeeld 1:*

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:

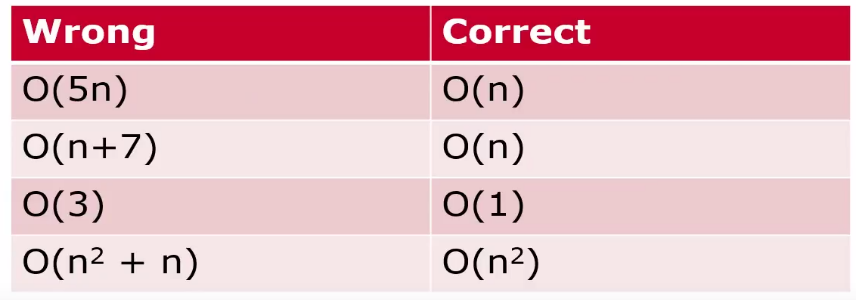
* Geheugencomplexiteit (de hoeveelheid RAM die het opneemt)
* Rekencomplexiteit (de hoeveelheid CPU die het opneemt)
  + Dit is uit te rekenen met **Tijdmetingen**



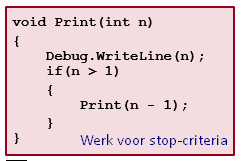
Bij complexiteit moet je altijd ook de volgende vraag stellen om tot de juiste complexiteit te komen:

***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.

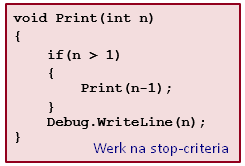
Een ander voorbeeld is complexiteit . de is een constante. Dit heeft **geen invloed** op de complexiteit van het algoritme. Als je n 2 keer zo groot maakt, blijft de complexiteit lineair.

**Recursie**



Recursie is het principe wanneer een methode zichzelf aanroept vanuit zijn methode-body. Het toepassen van recursie kan een (wiskundig) probleem vaak vermakkelijken. Er zijn 2 eigenschappen van recursie:

* **Er is altijd een stop-criterium**
* **Het probleem wordt opnieuw uitgedrukt in een versimpelde versie van hetzelfde probleem**



Dat laatste doe je vaak door te verkleinen met -1.

Recursie **lijkt veel** en **heeft ongeveer dezelfde werking** als een **iteratieve methode**.

**Stack**

Elke methode-aanroep plaatst zijn **return adres en variabelen** (dus ook parameters) op de **Stack**. De stack is het geheugen van jouw computer. Het risico van Recursie is **Stack-Overflow**. Dit komt voor als er **teveel informatie op de stack staat**. De stack is een **Abstracte datatype** (Dus ze kunnen worden bewerkt).

Als een recursieve methode wordt aangeroepen **voor** zijn werk wordt verricht, schrijft hij de return-adres en de variabelen **boven op** de Stack, en roept daarna de volgende methode aan, net zo lang tot de stopcriterium is bereikt. Daarna gaat het programma **De Stack van boven naar beneden** aflezen, om zo terug te komen naar de eerste methode.

**Trees**

Een tree is een mooi voorbeeld van recursie, waarbij **de grootte van het probleem** de diepte van de boom is. Het **maken** van een boom met recursie is altijd **exponentieel**: Vergroot je met , dan worden er 2 keer zoveel takken getekend (het werk): .

Elke keer dat een tak getekend moet worden, wordt er **zo diep mogelijk** en **zo links mogelijk** gewerkt. Dit heet het **Depth First Traversal.** Hoe vaker er een tak getekend moet worden, hoe meer geheugen er gebruikt moet worden om op de stack op te slaan. Dit is typisch

Het **maken** van bomen is **exponentieel**, en erg “duur”, het **lezen** van bomen is **logaritmisch** en erg efficient. Hierover later meer.

**Call by Value, Call by Reference**

**Reference**

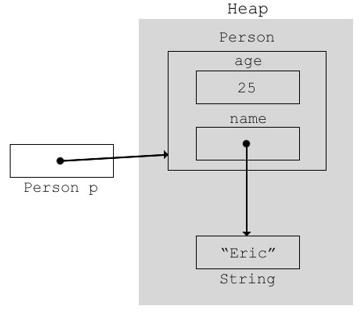
Een reference is een **geheugen-adres** dat wijst naar een **object** op de **Heap**. Er zijn twee verschillende referenties:

* **32-bit referenties**
* **64-bit referenties**

Dit wilt zeggen dat een 32-bits referentie een geheugenadres van maximaal **4 gigabite** kan onthouden. Een 64-bits referentie kan een geheugenadres van **16 exabite** onthouden.

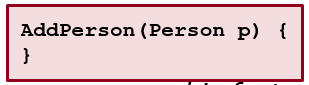
**Reference object**

Een referentie object is een object dat je niet direct benaderd, maar via een referentie. In dit geval is “p” een referentie naar het object Person() (te zien aan het keyword new).

Het **Person**-**object** wordt aangemaakt op de **Heap**. De heap is een **Deel van het geheugen** waar objecten worden opgeslagen. Er wordt alleen iets op de heap opgeslagen als dit object wordt aangemaakt met hey keyword new.

De reference-waarde van P is de **waarde van het geheugenadres** van het Person object. De reference-waarde van P kan op twee plekken bestaan. Namelijk:

* **Op de Call-Stack**
* **Op de Heap**

De reference-waarde van P staat op de Call-Stack als hij is **meegegeven als parameter**, of altijd hij als **lokale variabele** is aangemaakt. Het object name bevat een String object. Omdat String ook een object is, staat deze ook op de Heap.

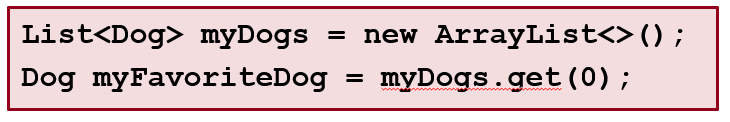
Toegang tot een reference-object is **langzamer** omdat er niet direct tegen het object gepraat wordt. Er kunnen **meerdere referenties naar hetzelfde object** zijn.

**Value object**

Een Value object is een object dat direct benaderd kan worden zonder een referentie nodig te hebben. Een Value object **bestaat op de Heap** (mits dit een onderdeel is van een **reference-object** zoals de age in de Person object). Als een value-object wordt meegegeven als parameter, wordt er **een kopie van het object zelf** mee gegeven, en **niet** een referentie naar dat value-object.

Toegang tot een value-object is **snel** omdat het direct wordt aangeroepen

**Generics**

Generics is de code **tussen punthaken: <Type>**. In de punthaken geef je aan **welk object** wordt meegegeven in bijvoorbeeld een array:

De voordelen van Generics:

* **Er kan maar 1 soort object worden ingevoerd**
* **Je kan de output verwachten/voorspellen**
* **Geen cast toepassen**
* **Beter onderhoudbaar**

**Stack**

Een Stack is een **Abstract Datatype**. Dit wilt zeggen dat het niet ergens **opgeslagen op de computer** is, maar dat het **nog geïmplementeerd** moet worden. Op de Stack worden items **op elkaar** gestapeld, en kan je alleen acties uitvoeren op het bovenste item. Dit resulteert in: **First in Last Out.** Het laatst ingevulde item komt er als eerste uit, en andersom. De volgende acties kunnen plaats vinden:

* **Push ->** Plaats een item bovenop de stack
* **Pop ->** Haal een item van de stack
* **Peek ->** Bekijk de waarde van het bovenste item op de stack
* **Size ->** Bekijk de grootte van de stack

Deze 4 acties heten de **Functionele specificaties** waar de Stack aan moet voldoen (“**Wat** moet de Stack kunnen doen?”). Een Stack wordt vaak gemaakt met een **interface,** omdat de interface verplicht alle implementerende classes te voldoen aan de functionele specificaties. De **non-functionele** eisen kunnen dan bijvoorbeeld zaken als uitbreidbaarheid (Extendab**ility**), verversingssnelheid -> **-ilities** (“Hoe goed moet de Stack iets kunnen?”)**.**

Om een stack te **implementeren** heb je een onderliggende datastructuur nodig. Dit zijn veelal de volgende:

