Zoeken en sorteren

# Zoeken in een Array

Een zoekalgoritme wordt gebruikt om een bepaald element te zoeken in een gegeven array. Maw om de positie van dat element te bepalen. Of de array gesorteerd is of niet kan een invloed hebben op de te gebruiken methode.

* Ongesorteerd : minder efficiënt dan gesorteerd

## Sequentieel zoeken

Vb: een array met de namen van 5000 studenten.

Meest voor de hand liggende manier om een naam te vinden is een voor een door de namen gaan tot je de naam gevonden hebt. Deze manier heet lineair zoeken of sequentieel zoeken.

Het algoritme gaat als volgt:

De elementen worden een voor een vergeleken met de op te sporen waarde zoekItem, het algoritme stopt wanneer het item gevonden is of het einde van de array bereikt is. Als return waarde wordt de index van zoekItem teruggegeven. Indien de index van zoekItem niet voorkomt in de array wordt er -1 teruggeven.

Pseudocode:

Afbeelding met tekst

Automatisch gegenereerde beschrijving

## Binair zoeken

Lineair of sequentieel zoeken is eenvoudig te implementeren en werkt uiteraard ook als de array waarin wordt gezocht reeds gesorteerd is. Intuïtief is het echter duidelijk dat er een “betere” manier moet bestaan om te zoeken in een gesorteerde array.

Binair zoeken gaat op dezelfde manier tewerk als het zoeken in een woordenboek, maar dan heel systematisch. In een gesorteerde array wordt het middelste item bekeken. Als dit kleiner is dan het gezochte item, wordt er verder gezocht in de rechterhelft van de array, als het groter is in de linkerhelft.

Op het einde van dit proces blijft er een element over waarna de iteratie stopt. Door het vergelijken van dit element met het gezochte element weten we direct of het element voorkomt in de array.

Het idee dat het zoekproces in de oorspronkelijke array wordt herleid naar een zoekproces half zo groot, laat toe dit probleem op te lossen met recursie, maar uiteraard kan het algoritme ook iteratief worden geïmplementeerd.

Pseudocode:

Afbeelding met tekst

Automatisch gegenereerde beschrijving**Invoer**: een item zoekItem dat moet gevonden worden, een gesorteerde array genaamd rij van items van lengte n.

**Uitvoer**: de index van het eerste element in rij dat gelijk is aan zoekItem wordt teruggegeven of -1 indien zoekItem niet voorkomt in rij.

### 1.1.2.1 Recursieve implementatie van binair zoeken

Afbeelding met tekst

Automatisch gegenereerde beschrijving

## Sequentieel vs binair zoeken

Het algoritme van binair zoeken is ingewikkelder dan dat van sequentieel zoeken. Maar wil dat ook zeggen dat het beter is?

Wanneer we een algoritme gaan analyseren, zijn we geïnteresseerd in de volgende zaken:

1. De tijd die het algoritme nodig heeft om uit te voeren
2. De hoeveelheid RAM die het algoritme nodig heeft bij uitvoering

We concentreren ons voornamelijk op de tijd die nodig is om het algoritme uit te voeren.

Bij de analyse van de zoekalgoritmen zien we dat de asymptotische uitvoeringstijd bepaald wordt door het aantal vergelijkingen dat wordt uitgevoerd.

**Aantal vergelijkingen bij sequentieel zoeken**

Bij sequentieel zoeken kunnen we ons afvragen hoeveel keer de vergelijking rij[i] != zoekItem

* Beste geval: het te zoeken element staat helemaal vooraan de array
* Slechte geval: het te zoeken element staat helemaal achteraan de array

Gemiddeld gezien staat het te zoeken element ergens in het midden, zodat we verwachten ongeveer n/2 vergelijkingen uit te voeren (n = lengte array).

We zien dat de uitvoeringstijd verdubbelt wanneer de lengte van de invoerrij wordt verdubbeld. De tijdscomplexiteit is lineair of van de orde n. we noteren:



**Aantal vergelijkingen bij binair zoeken**

Bij binair zoeken tellen we hoeveel keer de vergelijking wordt uitgevoerd.



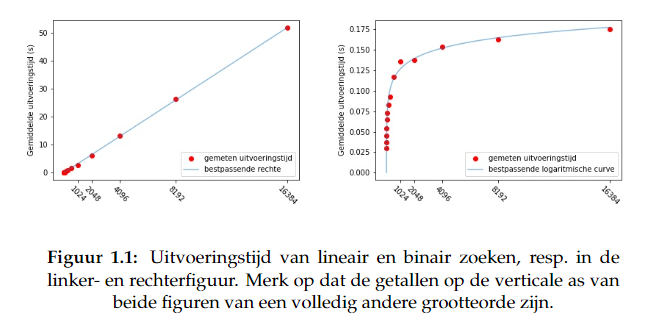
Voor de eenvoud van de analyse nemen we aan dat de lengte van de array een macht van twee is of

n =2^k waarbij k een natuurlijk getal is.

Wanneer k =0 dan is de lengte van de rij gelijk aan 1 en wordt de vergelijking geen enkele keer uitgevoerd.

Wanneer k=3 (m.a.w. de lengte van de rij is 8) dan zal na de eerste uitvoering van vergelijking (1.1) de lengte van de deelrij gereduceerd worden tot4. In dit geval zijn er dus 1+2=3 uitvoeringen van de vergelijking (1.1).Op deze manier zien we dat wanneer n=2^k er precies k uitvoeringen zullen zijn van de vergelijking (1.1).

Het verband wordt gegeven door:



Merk op dat voor kleine waarden van n, de uitvoeringstijdens van de twee methoden niet significant van elkaar verschillen. Het verschil in gedrag manifesteert zich wanneer de rijen waarin moet gezocht worden iets langer zijn.

# Sorteren van een array

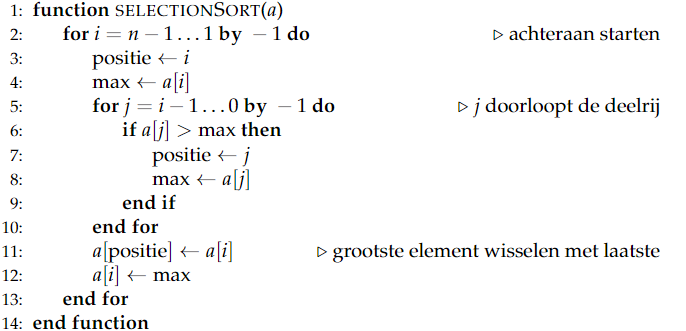
In vorig hoofdstuk hebben we gezien dat het doorzoeken van een array veel efficiënter gaat wanneer deze gesorteerd is. In deze sectie bekijken we een drietal algoritmen om een array te sorteren

## Sorteren door selectie

Het basisidee van sorteren door selectie kan eenvoudig samengevat worden:

1. Zoek het grootste element en plaats het achteraan
2. Sorteer de rest van de array

Meer in detail: in de te sorteren array gaan we op zoek naar het grootste element. Indien dit element niet achteraan staat in de rij, moet dit element verwisseld worden met het element op de laatste plaats. Zo staat het grootste element achteraan de rij. De n-1 overige elementen moeten nu gesorteerd worden.

Pseudocode:

**Complexiteitsanalyse**

De uitvoeringstijd wordt bepaald door het aantal keer dat de vergelijking a[j] > max, dit aantal komt overeen met het aantal keer dat de teller j van waarde wijzigt.

Afbeelding met tekst, tafel

Automatisch gegenereerde beschrijving

De tabel geeft duidelijk weer dat de binnenste lus (n(n-1))/2 keer wordt herhaald. Hieruit besluiten we:



## Sorteren door tussenvoegen

Het basisidee achter sorteren door tussenvoegen kan als volgt samengevat worden:

1. Veronderstel dat er reeds een deel vooraan de array gesorteerd is
2. Neem het eerste element van het niet gesorteerde deel en voeg dit element toe op de juiste plaats in het gesorteerde deel. Op deze manier wordt het gesorteerde deel uitgebreid.

Sorteren door tussenvoegen of “card sort” kan maw het best vergeleken worden met het op volgorde steken van kaarten. In het algoritme is dit: indien de eerste i elementen reeds gesorteerd zijn, dan gaan we kijken naar het (i+1)-ste element. Dit element wordt op de juiste plaats tussengevoegd.

**Complexiteitsanalyse**

Anders dan bij het sorteren door selectie, hangt de uitvoeringstijd van het algoritme in dit geval niet enkel af van de inputgrootte. Ook de manier waarop de elementen bij aanvang geordend zijn speelt een rol.

Pseudocode: Afbeelding met tekst

Automatisch gegenereerde beschrijving

## Sorteren door mengen

Sorteren door mengen is een ingewikkelder algoritme dan de voorgaande twee eenvoudige sorteeralgoritmes maar is ook een heel stuk efficiënter. Het basisidee is het volgende:

1. Sorteer de eerste helft van de array.
2. Sorteer de tweede helft van de array
3. Meng de twee gesorteerde deelrijen samen tot een gesorteerde array

De eerste twee stappen gebeuren op een recursieve manier.

Meer in detail: voor het samenvoegen of mergen van de deelrijen wordt een hulprij aangemaakt. In deze hulprij worden de deelrijen samengevoegd.

Voor het samenvoegen wordt in de beide deelrijen links gestart. Dit is voor beiden het kleinste element. Deze twee worden vergeleken met elkaar, het kleinste wordt opgeslagen in de hulprij. Er wordt voor beide rijen een teller bijgehouden die de huidige positie in die rij bijhoudt.

Pseudocode:

Afbeelding met tekst

Automatisch gegenereerde beschrijving

Afbeelding met tekst

Automatisch gegenereerde beschrijvingAfbeelding met tekst

Automatisch gegenereerde beschrijving