

Fun 3

Algoritmen

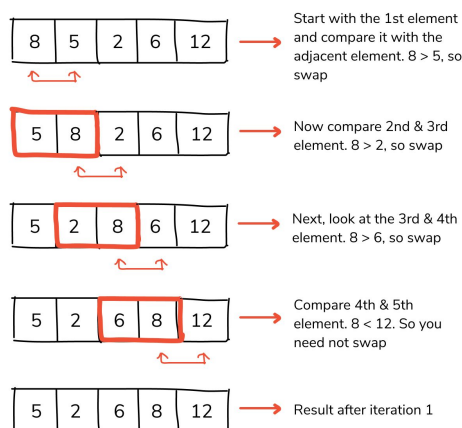
Het algoritme dat ik gekozen heb is **Bubble Sort**.

Het is een redelijk eenvoudig uit te leggen algoritmen.

De reden waarom ik toch voor dit algoritme gekozen heb is omdat het vaak gekozen is als voorbeeld (illustratie) van een algoritme. Echter is het erg inefficiënt. Vaak wordt gekozen voor een goed en efficiënt algoritme. Mij lijkt het juist leuk om onderzoek te doen naar een efficiënt algoritme.

Bubblesort werkt als volgt:

1. Loop door de rij van N elementen en vergelijk elk element met het volgende. Verwissel beide als ze in de verkeerde volgorde staan en schuif van een stapje op.
2. Loop opnieuw de rij door maar ga nu door tot het een na laatste element. Dit doe je omdat het laatste element het grootste is.
3. De vorige stap herhaal je nog een keer maar nu stop je na de 2 laatste elementen.
4. Herhaal dit maar stop bij N-1 getallen.



Een mooi voorbeeld hoe het algoritme werkt na 1 iteratie.

Big O notation

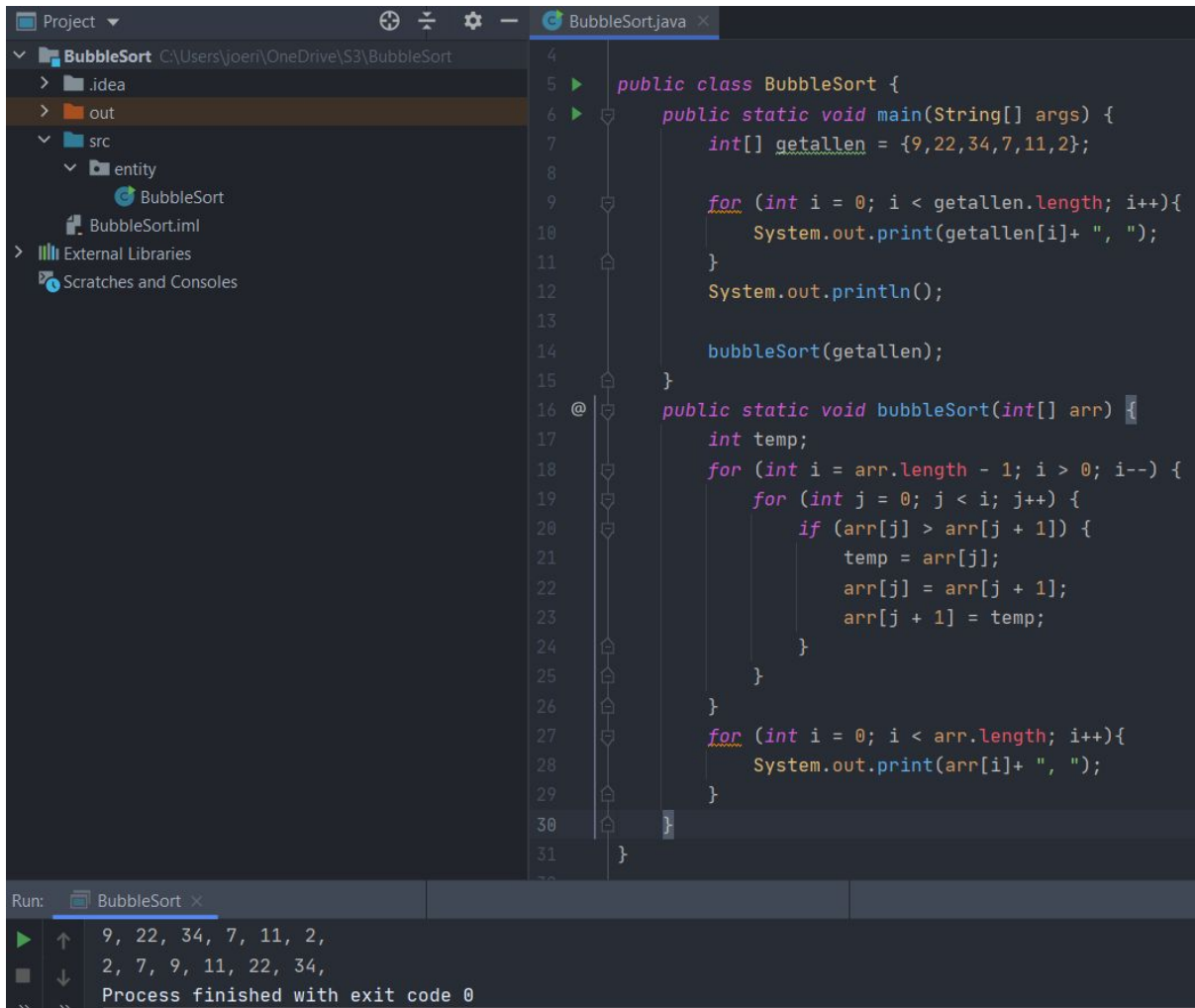
BubbleSort heeft een van de slechtste van Big O notaties.

De meeste andere sorteeralgoritmen doen het stukken beter. Bijvoorbeeld insertion sort is hier heel veel complexer en heeft een beter performance.

Dit komt doordat mede doordat je gebruik maakt van een geneste for loop. Je doet meerdere acties per element. Ook ga je erg vaak over dezelfde getallen heen. Dit leidt tot " $O(N^2)$ ", waarbij N het aantal te sorteren items is.

POC

De code die de reeks getallen sorteert.



```
Project ▾
  ▾ BubbleSort C:\Users\joeri\OneDrive\S3\BubbleSort
    ▾ .idea
    ▾ out
    ▾ src
      ▾ entity
        BubbleSort
        BubbleSort.iml
    ▾ External Libraries
    ▾ Scratches and Consoles

BubbleSort.java
4
5 public class BubbleSort {
6   public static void main(String[] args) {
7     int[] getallen = {9,22,34,7,11,2};
8
9     for (int i = 0; i < getallen.length; i++){
10      System.out.print(getallen[i]+ " ");
11    }
12    System.out.println();
13
14    bubbleSort(getallen);
15  }
16  public static void bubbleSort(int[] arr) {
17    int temp;
18    for (int i = arr.length - 1; i > 0; i--) {
19      for (int j = 0; j < i; j++) {
20        if (arr[j] > arr[j + 1]) {
21          temp = arr[j];
22          arr[j] = arr[j + 1];
23          arr[j + 1] = temp;
24        }
25      }
26    }
27    for (int i = 0; i < arr.length; i++){
28      System.out.print(arr[i]+ " ");
29    }
30  }
31 }

Run: BubbleSort
9, 22, 34, 7, 11, 2,
2, 7, 9, 11, 22, 34,
Process finished with exit code 0
```

Efficiëntie

Met een kleine reeks getallen is het nog wel redelijk te doen. Echter zodra de reek enkele duizende getallen bevat is de computer erg lang bezig om deze in de juiste volgorde te zetten. Bij grote reeksen zou ik echter naar andere algoritmes kijken die het een stuk efficiënter oplossen. Voor commercieel gebruik is dit dan ook niet aan te raden.

Reflectie

Ik vond deze opdracht best complex en moeilijk. Ik heb onderzoek gedaan naar verschillende algoritme en daar zaten best een aantal complexe tussen (Dijkstra). Voor mij was dit dan ook best een moeilijke opdracht. Ik vond het leuk om te leren hoe algoritmen werken en welke toepassingen het allemaal kent. Maar zoals al aangegeven zijn veel algoritmen al een standaard oplossing die enkel nog maar hoeft worden aangepast voor jouw probleem. Ik ben erachter gekomen dat ik het wel interessant vind maar dat het me ook erg veel energie kost om te begrijpen. Ik wil mezelf dan ook meegeven dat ik er zeker niet voor moet terugdeinzen, maar dat ik complexe algoritmes beter aan andere over kan later die hier meer gedreven in zijn. Toch vond ik het leuk om eens te kijken naar inefficiënte

algoritmes. Vaak wordt gekozen wat de beste performance heeft maar ik vond het leuk om te onderzoeken waarom sommige algoritmes niet altijd de juiste oplossing zijn.

Backtracking

Als aanvulling op het bubble sort heb ik nog een backtracking algoritme geïmplementeerd. Dit werd ook als mogelijk voorbeeld gegeven in de opdrachtbeschrijving.

Het is een stuk efficiënter dan bubblesort of brute kracht omdat niet alle oplossingen bekeken hoeven te worden.

Bij een zoekprobleem moet er een oplossing geselecteerd worden uit een heel plausibele mogelijkheden. Tijdens het oplossen van het probleem moet men een keuze maken. Als achteraf blijkt dat dit niet leidt tot een oplossing dan moet men terugkeren naar het vorige keuzemoment, om vanaf hier een andere keuze te maken. Het terugkeren wordt backtracking genoemd.

Big O notatie vond ik erg filosofisch en complex. Maar ik denk dat ik het nu het kwartje is gevallen. Omdat bij backtracking in de sudoku hij elk vakje moet checken. En vervolgens moet hij de standaard sudoku checks doen. Dit is kijken of het getal al in de rij, kolom en subbox voorkomt. Dat betekent dus hij kijkt 4 keer over het geheel heen. Wat resulteert in $O(N^4)$.

Meting

Met een simpel eenvoudige sudoku doet mijn programma er ongeveer 150ms over om het op te lossen. Dit komt omdat er veel lege plaatsen zijn en er veel mogelijke oplossingen zijn.

Met een 10 sterren sudoku doet hij er slechts 40ms over. Dit is te verklaren omdat deze sudoku al deels is ingevuld er minder mogelijkheden zijn om uit te proberen.

Opmerkelijk is dat met een lege sudoku hij er 10ms over doet. Dit komt omdat bij het invullen van de sudoku hij direct een getal kan plaatsen wat later niet meer veranderd hoeft te worden.

Reflectie

Na de bubblesort was ik wel weer gemotiveerd om nog een ander algoritme toe te passen. Ook om voor mezelf om te trainen en beter te worden in algoritmes. Ik snapte hierna ook beter hoe algoritmes werken en in welke toepassingen het gebruikt kan worden. Wat vooral duidelijker is geworden in hoe vaak een algoritme eigenlijk bezig met het controleren van de mogelijkheden.