# Opdrachten kwartaal 4 week 6 – Algoritmen, performance en big-O-notatie

Als een algoritme op een grote hoeveelheid data wordt uitgevoerd, dan is het niet alleen belangrijk dat het algoritme correct werkt, maar ook dat het snel (genoeg) is. De executietijd van een algoritmen is afhankelijk van vele factoren zoals, de gebruikte hardware en compiler maar ook van de zogenoemde computationele complexiteit[1] van het algoritme. De complexiteit van een algoritme kan van een bepaalde orde zijn. Deze orde wordt gespecificeerd met behulp van de zogenoemde *big-O-notatie*[2].

Je leert deze les hoe:

- je de orde (big-O-notatie) van een algoritme kunt bepalen;
- verschillende implementaties van algoritmen die functioneel gelijk zijn, toch in orde kunnen verschillen.

**Lees nu hoofdstuk 7 van het dictaat Objectgeoriënteerd Programmeren in C++.**

Alle bestanden die nodig zijn voor de onderstaande opdrachten zijn te vinden in: `timed.zip`[3]

## Sorteren

In `stopwatch.h` en `stopwatch.cpp` is de class `Stopwatch` gedeclareerd en gedefinieerd. Deze class kan gebruikt worden om tijd te meten. In het voorbeeldprogramma `timed_sort.cpp` wordt de tijd gemeten die het `bubble_sort` algoritme nodig heeft om een vector met $n$ elementen te sorteren.

**4.6.1** Start het voorbeeldprogramma en zoek een waarde van $n$ waarbij het sorteren ongeveer 1 seconde duurt. Voer deze waarde 10 maal in en vergelijk de resultaten. Maak een grafiekje met behulp van Excel. De tijd die nodig is om $n$ getallen te sorteren is *niet* constant! Kun je dat verklaren?

**4.6.2** Bepaal de complexiteit van het `bubble_sort` algoritme in big-O-notatie door de source code te analyseren. Stel vervolgens de orde van dit algoritme empirisch vast door de benodigde tijd te meten voor minstens 10 verschillende waarden van $n$. Dit kun je

[1] Zie eventueel: https://en.wikipedia.org/wiki/Computational_complexity_theory.

[2] Zie eventueel: https://en.wikipedia.org/wiki/Big_O_notation.

[3] https://bitbucket.org/HR_ELEKTRO/ems31/raw/master/Opdrachten/progs/timed.zip

natuurlijk automatiseren door het programma aan te passen. Maak een grafiek met behulp van Excel. In Excel kun je een zogenaamde trendlijn bepalen die de gemeten waarden zo goed mogelijk (met een bepaalde functie) benadert. Geef de gevonden functie en de waarde van $R^2$ (die aangeeft hoe goed de trendlijn de meetwaarden benadert) ook weer in de grafiek. Wat is de complexiteit van dit algoritme uitgedrukt in big-O-notatie? Komt dit overeen met de complexiteit die is gevonden door de source code te analyseren?

**4.6.3** De C++ standaardfunctie `sort` is veel efficiënter dan `bubble_sort`. Vervang de aanroep naar de (door mij) zelfgemaakte `bubble_sort` door een aanroep naar de standaardfunctie `sort`. Bepaal eerst een waarde van $n$ waarbij de tijd die nodig is om te sorteren ongeveer 1 seconde is. Probeer de orde van de C++ standaard `sort` te bepalen door de benodigde tijd te meten voor minstens 10 verschillende waarden van $n$. Maak een grafiek met behulp van Excel, bepaal ook welke trendlijn het beste past (de hoogste waarde van $R^2$ heeft). Wat is de complexiteit van dit algoritme uitgedrukt in big-O-notatie? Zoek de complexiteit van `std::sort` op in de documentatie[4]. Komt dit overeen met je bevindingen? Verklaar eventuele verschillen.

### Meerderheidselement

Een meerderheidselement in een array van $n$ elementen is een element dat *meer dan* $n/2$ keer in de array voorkomt. De array: 3, 3, 4, 2, 4, 4, 5, 4, 4 heeft als meerderheidselement de waarde 4. De array: 3, 3, 4, 2, 4, 4, 5, 4, 3 heeft *geen* meerderheidselement.

Dit algoritme kan gebruikt worden bij verkiezingen waar een zogenoemd meerderheidsstelsel met absolute meerderheid wordt gebruikt[5]. Als er veel stemgerechtigden zijn, is het van belang dat dit algoritme snel (genoeg) is.

Er zijn verschillende algoritmen om het meerderheidselement van een array te vinden:

- **Methode 1:**
  Tel hoe vaak het eerste element voorkomt. Als dit element meer dan $n/2$ keer voorkomt, dan is dit het meerderheidselement. Zo niet, tel dan hoe vaak het tweede element voorkomt. Als dit element meer dan $n/2$ keer voorkomt, dan is dit het meerderheids-

[4] Zie: https://en.cppreference.com/w/cpp/algorithm/sort.

[5] Zie eventueel: https://nl.wikipedia.org/wiki/Meerderheidsstelsel.

element. Enzovoort, totdat je voor alle elementen geteld hebt hoe vaak ze voorkomen. Als je dan nog steeds geen meerderheidselement hebt gevonden, dan is het er ook niet.

- **Methode 2:**
  Sorteer de array en tel steeds het aantal elementen met dezelfde waarde (die dan achter elkaar staan). Zodra je meer dan n/2 elementen met dezelfde waarde vindt, heb je het meerderheidselement gevonden.
- **Methode 3:**
  Bij deze methode gaan we er (in eerste instantie) vanuit dat er een meerderheidselement is. Als er een meerderheidselement is, dan kun je dat vinden door steeds twee *verschillende* waarden uit de array te verwijderen. Totdat de array nog maar één waarde bevat, deze waarde is dan het meerderheidselement. Dit 'verwijderen' kan op een slimme manier gedaan worden door element voor element te 'bekijken'. Het element wat eventueel het meerderheidselement zou kunnen zijn, noemen we de kandidaat. We houden ook in een teller bij hoe vaak we de kandidaat hebben gezien. We gaan nu één voor één de elementen van de array af. Bij het 'bekijken' van een element zijn er twee mogelijkheden.
    - De teller is 0:
      We hebben nog geen kandidaat dus we kiezen dit element als kandidaat en zetten de teller op 1.
    - De teller is $> 0$:
      Als het element gelijk is aan de kandidaat, verhogen we de teller met 1 en anders verlagen we de teller met 1.

  Als we alle elementen hebben 'bekeken', blijft er een kandidaat over (teller $> 0$) of niet (teller $= 0$). Omdat we er (in eerste instantie) vanuit gegaan zijn dat de array een meerderheidselement heeft moeten we nu, als er een kandidaat is, nog wel even controleren of dit het meerderheidselement is door te tellen hoe vaak de kandidaat in de array voorkomt.
  *Uitleg:*
  Als de teller 0 wordt, hebben we net zoveel elementen gehad die gelijk waren aan de kandidaat (telkens +1) als elementen die ongelijk waren aan de kandidaat (telkens -1). Deze elementen vormen dus paartjes van ongelijke elementen en die kunnen we dus allemaal uit de array verwijderen. Als er een meerderheidselement was in de oorspronkelijke array, dan heeft de resterende rij hetzelfde meerderheidselement! Dit zou wel of niet de vorige kandidaat kunnen zijn maar dat maakt niets uit! Deze methode is bedacht door Moore en Boyer. Zie http://www.cs.utexas.edu/users/moore/best-

ideas/mjrty/index.html voor een eenvoudige uitleg en een stap voor stap demo. Zie http://www.cs.utexas.edu/users/boyer/mjrty.pdf voor een uitgebreide uitleg.

**4.6.4** Schrijf drie functies die kunnen bepalen of een array een meerderheidselement heeft (met de drie hierboven beschreven methoden). Als de array een meerderheidselement heeft, dan moet de functie dit element ook teruggeven.

Het prototype van de functies is als volgt:

```
bool zoek_meerderheids_element_methode_n(int& resultaat, const ↩
 ↪ vector<int>& v);
```

De parameter `v` refereert naar een `vector` die de getallen bevat. De functies geven **`false`** terug als er geen meerderheidselement gevonden is. De functies geven **`true`** terug als er wel een meerderheidselement gevonden is. Dit meerderheidselement wordt dan opgeslagen in de variabele waar de parameter `resultaat` naar refereert.

Bepaal de complexiteit van de drie functies in big-O-notatie door jullie source code te analyseren. Stel vervolgens, met behulp van het programma timed_zme.cpp, de orde van deze functies empirisch vast door de benodigde tijd te meten voor minstens 10 verschillende waarden van $n$. Bedenk dat de big-O-notatie de *worst case* complexiteit aangeeft. Hoe kun je er voor zorgen dat de functies de *worst case* executietijd nodig hebben? Maak een grafiek met behulp van Excel. Bepaal ook welke trendlijn het beste past (de hoogste waarde van $R^2$ heeft). Wat is de gemeten complexiteit van deze drie functies uitgedrukt in big-O-notatie? Komt dit overeen met de complexiteit die is gevonden door jullie source code te analyseren?