KU LEUVEN

FACULTEIT WETENSCHAPPEN

Niet vertrouwelijk

Algoritmes vergelijken

Een empirische analyse van drie bekende sorteermethoden

Nisse DEGELIN – r0627289

Gegevensstructuren en algoritmen

Philip DUTRE

Academiejaar: 2020–2021

1. INLEIDING

Het vermogen om gegevens op een efficiënte manier te sorteren is van belang in vele toepassingen (Abdel-Hafeez & Gordon-Ross, 2017). In besturingssystemen en gegevensbanken is het zelfs één van de belangrijkste bouwstenen. Het verbaast dan ook niet dat er de afgelopen decennia veel onderzoek naar het verfijnen van sorteermethoden werd verricht (Inoue, Moriyama, Komatsu & Nakatani, 2012). Ondanks de immer stijgende computerkracht blijft de nood voor efficiente algoritmen hoog. In sommige situaties is het belangrijk om zuinig om te gaan met de gebruikte geheugenruimte, in andere is vooral de snelheid doorslaggevend. Daarom is er geen duidelijk leidend algoritme dat in alle situaties beter is dan de andere (Abdel-Hafeez & Gordon-Ross, 2017). Zo blijkt uit Inoue et *al*. (2012) dat een populair algoritme als quicksort niet geschikt is voor bepaalde berekeningsmodellen binnen gedistribueerd programmeren.

In dit verslag worden drie methoden om gegevens te sorteren met elkaar vergeleken aan de hand van de invoergrootte n. Eerst volgt er een algemene analyse per algoritme. Daarna wordt deze analyse getest aan de hand van enkele experimenten. Vervolgens verkennen we een alternatief voor het inschatten van de tijdscomplexiteit, waarna een besluit volgt.

2. THEORETISCH KADER

Deze sectie is grotendeels gebasseerd op *Algorithms* (Sedgewick & Wayne, 2019). Het idee en de implementatie van elk algoritme wordt kort uigelegd, alsook de tijdscomplexiteit.

2.1 Selection sort

Het is bekend dat selection sort intuitief eenvoudig is en minder efficient. Het selecteert het kleinste element en zet dat op de eerste plaats, dan het tweede kleinste op de tweede plaats. Selection sort blijft deze methode volgen totdat de rij gesorteerd is. Kenmerkend voor selection sort is dat de benodigde sorteertijd onafhankelijk is van het al-gesorteerd-zijn van de rij. Selection sort is daarom ongeschikt voor rijen die in bepaalde mate al gesorteerd zijn.

```java
public long sort(Comparable[] array) throws IllegalArgumentException {
        comparisons = 0;
        if (array == null) {
                throw new IllegalArgumentException("argument 'array' must not be null.");}
        int n = array.length;
        for (int i = 0; i < n; i++) {
                int min = i;
                for (int j = i + 1; j < n; j++) {
                        if (less(array[j], array[min])) {
                                min = j;}
                }
                exchange(array, i, min);
        }
        return comparisons;
}
```

Figuur 1: Implementatie van selectionsort (Java) (Bron: eigen code gebasseerd op Sedgewick & Wayne (2019))

Zoals te zien is in figuur 1 vertaalt zich dit in een genestelde lus die het kleinste element zoekt vanaf het geselecteerde element. Indien het kleiner is wordt het verwisseld. Het aantal verwisselingen is buiten de binnenste lus en binnen de buitenste lus, dus er zijn n verwisselingen (van het 0de element tot en met het (n-1)^{de} element). Het aantal vergelijkingen is eerst (n – 1) (van het 1ste element tot en met het (n-1)^{de} element. Dan (n-2) (van het 2de element tot en met het (n-1)^{de} element. Dit is met andere woorden een triangulaire som wat zich vertaald in een benadering van het aantal vergelijkingen van de orde ~ $\frac{n^2}{2}$. Omdat dit de operatie is die het vaakst gebeurd is dit de bepalend voor de tijd die het algoritme nodig heeft.

2.2 Insertion sort

Een tweede intuitief sorteeralgoritme is insertion sort. Hierbij wordt de rij verdeeld in een gesorteerde zone, en een ongesorteerde zone. In tegenstelling tot insertion sort betekent dit niet dat het element in de gesorteerde zone op zijn defenitieve plaats staat. Het algoritme is flexibeler dan selection sort want de benodigde tijd is afhankelijk van hoe gesorteerd de rij al was. Daarom is het geschikt voor rijen die al in zekere mate gesorteerd zijn. Een tweede verschil met selection sort is dat het aantal vergelijkingen en verwisselingen van gelijke orde is.

```java
public long sort(Comparable[] array) throws IllegalArgumentException {
        comparisons = 0;
        if (array == null) {
                throw new IllegalArgumentException("argument 'array' must not be null.");
        }
        int n = array.length;
        for(int i = 1; i < n; i++) {
                for (int j = i; j>0 && less(array[j], array[j - 1]); j--){
                        exchange(array, j, j-1);
                }
        }
        return comparisons;
}
```

Figuur 2: Implementatie van insertion sort (Java) (Bron: eigen code gebasseerd op Sedgewick & Wayne (2019))

In de implementatie is er terug een genestelde lus. De eerste lus vertrekt van het 1ste element tot het (n-1)de element. De tweede lus wordt echter niet altijd evenveel keer uitgevoerd. In het beste geval kan het eerste element van de ongesorteerde zone gewoon toegevoegd worden aan de gesorteerde zone zonder dat er hiervoor verwisselingen nodig zijn. In het slechtste geval gedraagt het algoritme zich zoals selections sort. Gemiddeld gezien (over een groot aantal willekeurig geordende rijen) is de grootste term van de vorm $\sim \frac{n^2}{4}$.

2.3 Quick sort

Een minder intuitieve, maar wel een veel snellere en daarmee populairdere sorteermethode is quicksort. De essentie van quicksort is het opdelen van een rij in deelrijen die appart gesorteerd worden. In deze implementatie wordt initieel het 1ste element als pivot gekozen. Naargelang een element groter of kleiner is wordt het in één van de twee deelrijen gestopt. Dit verklaart waarom quick sort minder efficient is bij rijen die al gesorteerd zijn. In dat geval zou het kleinste element als pivot worden gekozen, waardoor de tweede deelrij nauwelijks korter is dan de oorspronkelijke rij.

In de implementatie valt het op dat er meerdere functies zijn. Bovendien is de middelste functie een recursieve functie.

De partition methode vervult een sleutelrol: dit is de functie die de rij in deelrijen opdeelt door elementen met de pivot te vergelijken. Dit gebeurd door een index links en een index rechts in de rij te plaatsen, en de

```java
public long sort(Comparable[] array) throws IllegalArgumentException {
        comparisons = 0;
        if (array == null) {
                throw new IllegalArgumentException("argument 'array' must not be null.");
        }
        sort(array, 0, array.length - 1);
        return comparisons;
}
private static void sort(Comparable[] array, int links, int rechts) {
        if (rechts <= links) {return;}
        int j = partition(array, links, rechts);
        // Sorteer linkerdeel, van links tot en met j-1
        sort(array, links, j-1);
        // Sorteer rechterdeel, van j+1 tot en met rechts
        sort(array, j+1, rechts);
}
// Zet alle elementen kleiner dan de pivot links, groter dan rechts. returnt pivot index
private static int partition(Comparable[] array, int links, int rechts) {
        // initialiseer de indexes die gaan opschuiven
        int i = links;
        int j = rechts + 1;
        Comparable pivot = array[links];
        while(true)
        { // schuif de twee indexen i en j op tot ze elkaar voorbij steken, wissel elk keer
                while (less(array[++i], pivot)) if ( i == rechts) break;
                while (less(pivot, array[--j])) if ( j == links) break;
                if (i >= j) break;
                exchange(array, i, j);
        }
        // Zet pivot in array[j]
```

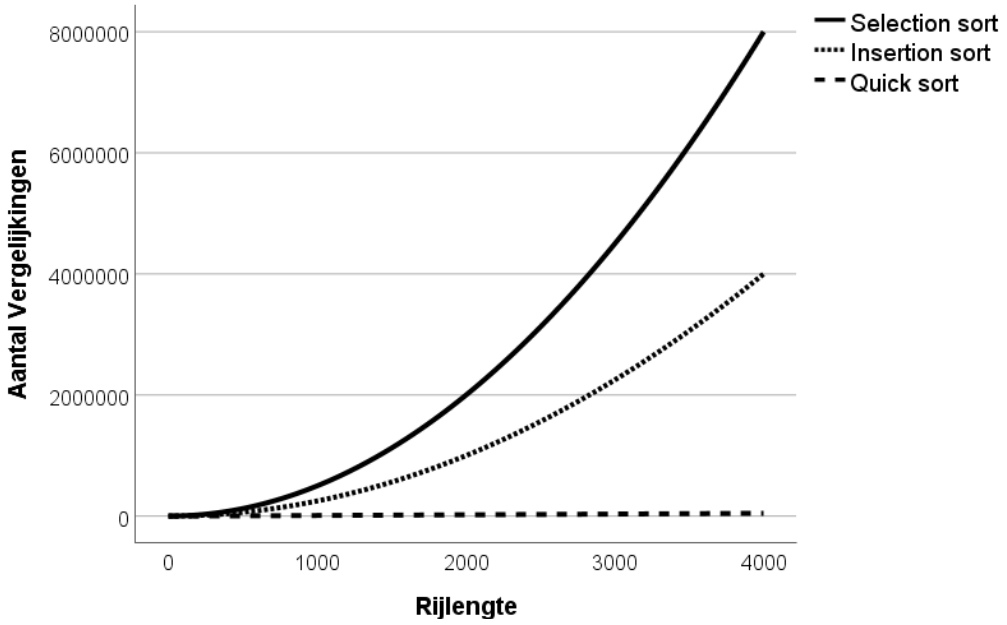
Figuur 3: Implementatie van quick sort (Java) (Bron: eigen code gebasseerd op Sedgewick & Wayne (2019))

betreffende elementen te verwisselen indien ze aan de foute kant staan (omdat ze respectievelijk groter en kleiner dan de pivot zijn). Aan de hand van de partition functie is het mogelijk de tijdscomplexiteit in te schatten. Afhankelijk van de gekozen pivot (idealiter de mediaanwaarde) wordt de rij in min of meer gelijke deelrijen gesorteerd. Hoe meer gelijk de deelrijen zijn, hoe efficienter de sort functie werkt. Het beste geval is dus als de pivot de mediaan is, waardoor er twee gelijke deelrijen worden gemaakt. Dan gebruikt de partition functie ongeveer n vergelijkingen om tot 2 deelrijen te komen, die dan elk nog eens recursief behandeld worden. Dit leidt tot een gemiddelde groei-orde van de vorm ~$1.39\, n \ln(n)$. Echter, als de pivot ver verwijderd van de mediaan is, dan is quicksort van de orde ~ $\frac{n^2}{2}$, met andere woorden zoals selection sort of een slecht presterende insertion sort.

3. KWANTITATIEVE AANPAK

3.1 Verwachting

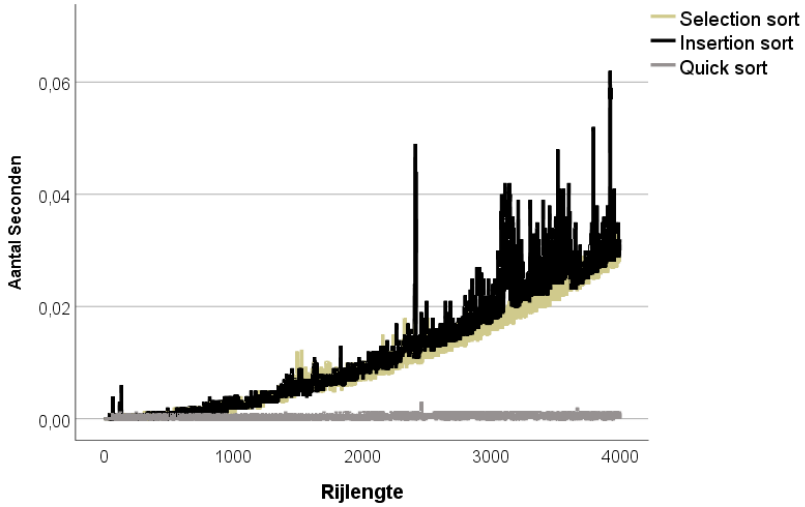
Figuur 4 bevat een grafische weergave van de geschatte complexiteit. Een grotere term vertaalt zich in een sneller stijgende lijn. Met andere woorden: voor een willekeurige rij met 2000 elementen heeft selection sort, insertion sort en quicksort respectievelijk 2000000, 1000000 en 21130,51 vergelijkingen nodig. Dit is echter een benadering: enkel de doorslaggevende term van de functie van het aantal vergelijkingen werd opgenomen. Bovendien is dit het gemiddelde aantal voor insertion -en quick sort.



Figuur 4: Theoretische functie van de drie sorteeralgoritmen (Bron: eigen grafiek en data)

3.2 De praktijk

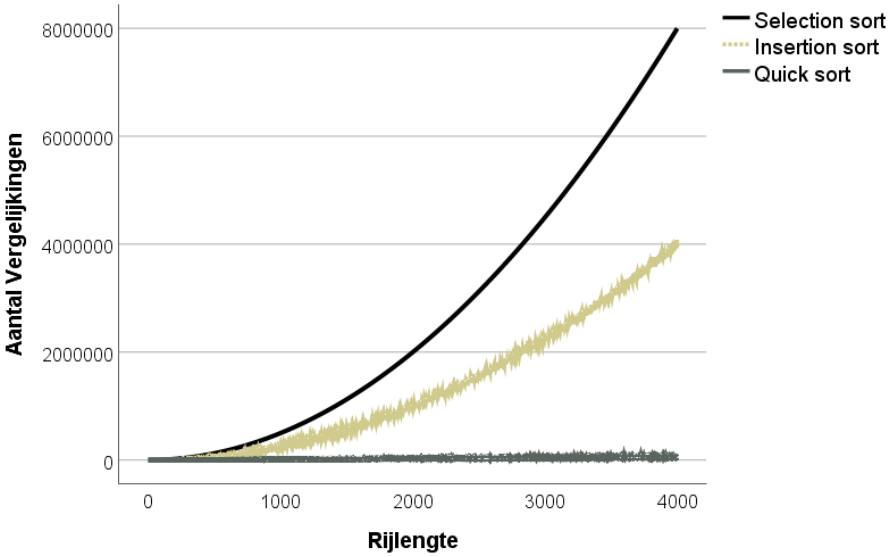
Wat kan er verwacht worden als deze grafiek geplot wordt met echte data? Allereerst is de verwachting natuurlijk dat figuur 4 in grote lijnen gevolgd wordt. Aangezien de tijdscomplexiteit van insertion –en quicksort het gemiddelde is van het beste en slechtst mogelijke geval, is het aannemelijk dat er hier meer uitschieters zijn dan bij selection sort.



Figuur 5: Aantal seconden voor het sorteren van oplopende rijen voor de drie sorteeralgoritmen (Bron: eigen grafiek en data)

Een eerste experiment meet het aantal seconden dat elke sorteermethode nodig heeft om een rij met een willekeurige orde te sorteren. Het resultaat is te zien in figuur 5. In grote lijnen volgt het aantal seconden inderdaad de geschatte complexiteit: quick sort heeft een verwaarloosbare tijd, selection en insertion sort volgt een stijgende curve. Een tweede vaststelling zijn de grote uitschieters bij insertion sort. Er zijn ook geen uitschieters in negatieve zin.

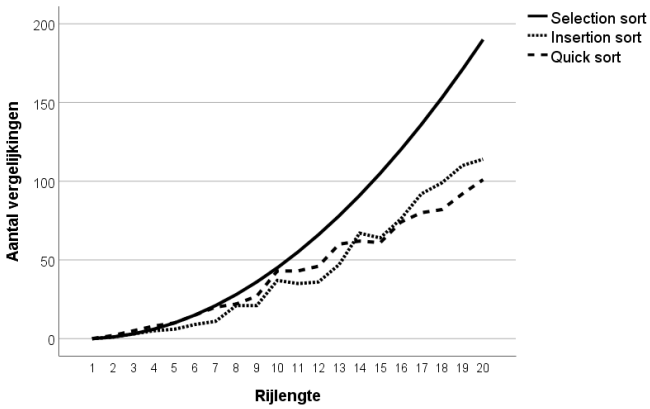
In zekere mate kunnen deze verklaard worden door het verschil tussen complexiteit in het slechtst mogelijke geval ($\sim\frac{n^2}{2}$) en het gemiddelde geval ($\sim\frac{n^2}{4}$) bij insertion sort. Wat echter opvalt is dat insertion sort eerder meer seconden nodig heeft dan selection sort, hoewel dit theoretisch juist omgekeerd zou moeten zijn. Mogelijk komt dit doordat de tijd niet alleen afhankelijk is van de efficientie van het algoritme, maar ook op de computer waarop ze uitgevoerd worden. Deze verklaring is echter onvoldoende aangezien alle algoritmes op dezelfde computer werden uitgevoerd. Dit wijst erop dat het meten van het aantal seconden een te groffe maat is voor het meten van complexiteit.



Figuur 6: Aantal vergelijkingen voor het sorteren van oplopende rijen voor de drie sorteeralgoritmen (Bron: eigen grafiek en data)

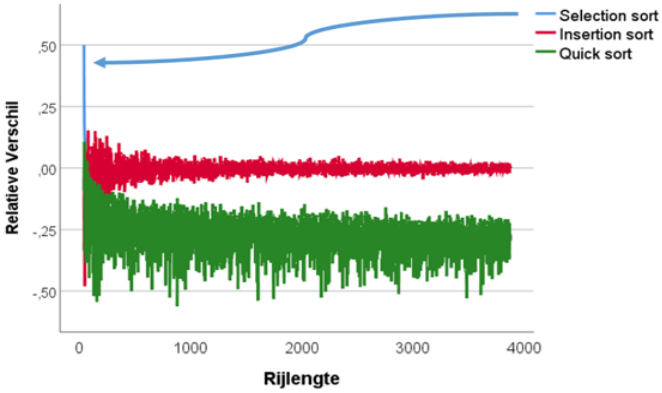
Figuur 6 plot het aantal vergelijkingen dat de implementaties nodig hadden om de 4001 rijen te sorteren. De curves volgen de theoretische complexiteit bijna volkomen. Ook de verwachting dat insertion –en quick sort meer uitschieters hebben wordt ingelost (hoewel deze klein zijn). Bij een rij van 2000 elementen hebben selection, insertion en quick sort respectievelijk 1999000, 1001740 en 26300 vergelijkingen nodig, wat nauw aansluit bij het geschatte aantal vergelijkingen.

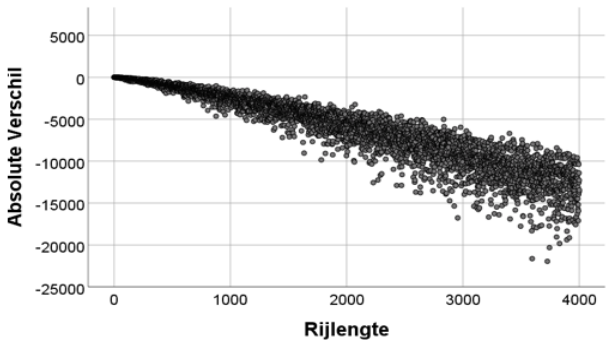
Omdat de schaal van figuur 6 niet toelaat de algoritmes voor kleine lijsten te vergelijken wordt in figuur 7 het resultaat geplot voor de eerste 21 rijen. Het valt op dat quick sort voor kleine rijen (hier tot 14 elementen) meer vergelijkingen nodig heeft dan insertion sort. Dit kan verklaard worden doordat quicksort een recursieve functie is, waardoor er extra oproepen nodig zijn. In dit geval is een eenvoudig algoritme als insertion sort dus efficienter.

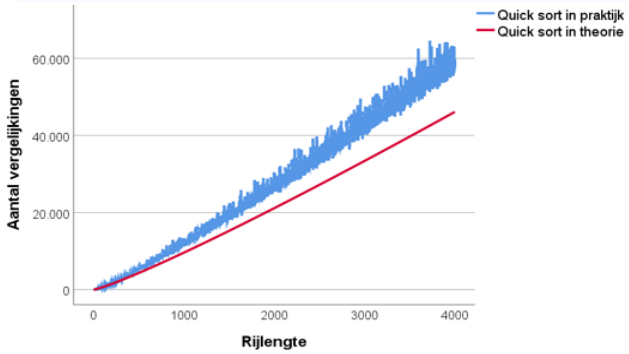


Figuur 7: Aantal vergelijkingen voor het sorteren van kleine oplopende rijen voor de drie sorteeralgoritmen (Bron: eigen grafiek en data)

Uit figuur 6 bleek wel dat het geplotte aantal vergelijkingen de theoretisch functies volgen, maar in welke mate volgen ze deze? En is dit voor elk algoritme gelijk? Alleen al uit de uitschieters spreekt het vermoeden van niet. Figuur 8a toont het relatieve veschil tussen het geschatte aantal vergelijkingen en het aantal vergelijkingen in de praktijk voor de drie sorteermethoden. Uiteraard valt het te verwachten dat schattingen op basis van enkel de grootste term niet precies gelijk zijn aan het exacte aantal vergelijkingen. Bij selection sort is de afwijking het kleinste (gemiddeld 0,0022). Ook dit was te verwachten, aangezien hier niet het gemiddelde is genomen van een beste en slechtste geval. Bij insertion sort ligt het verschil hoger, maar gemiddeld gezien is de fout ongeveer nul. Vooral bij quick sort zijn er grotere verschillen. Opmerkelijk is dat het verschil niet mooi rond nul schommelt zoals bij insertion sort, maar wel rond -0,27. Dit vertaalt zich in een neerwaartse vorm in figuur 8b. Het aantal vergelijkingen van quicksort wordt dus systematisch onderschat (figuur 8c).

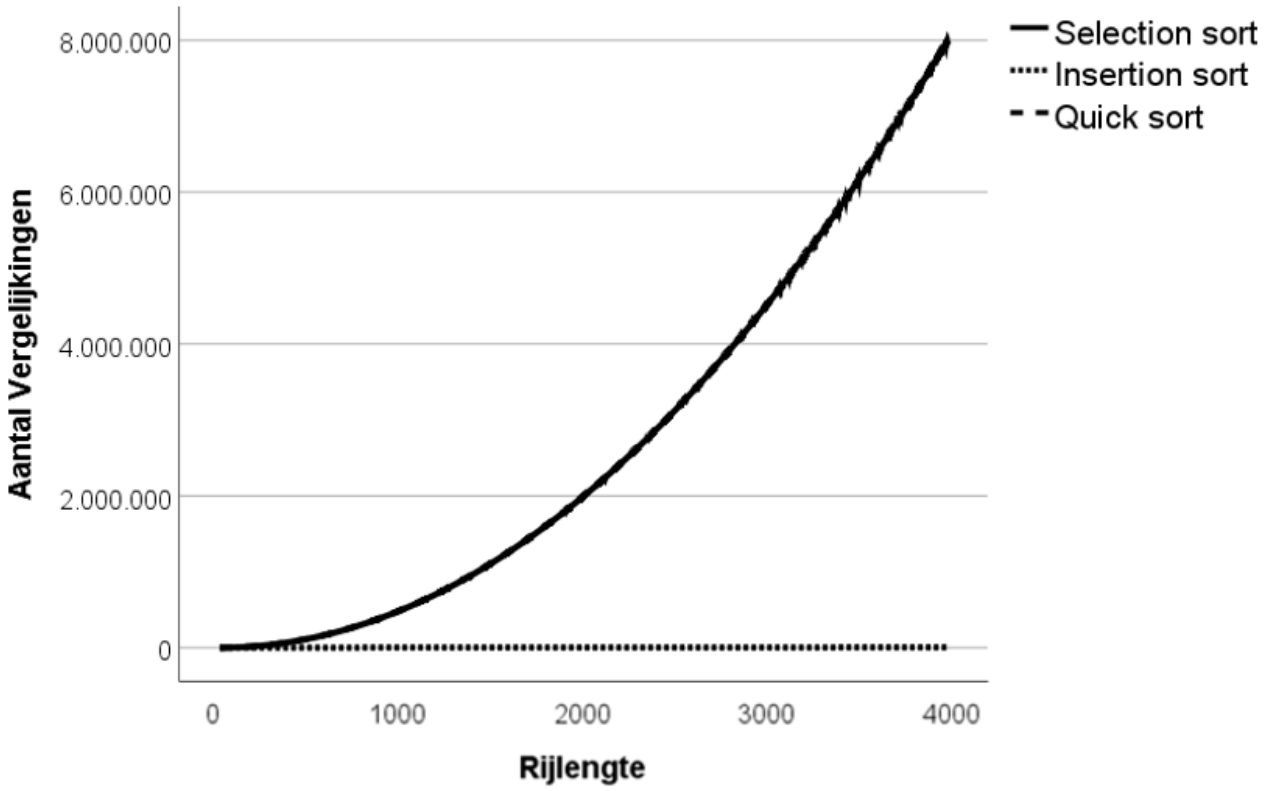






Figuur 8: (in wijzers in) (a) Relatieve verschil tussen het theoretische aantal vergelijkingen en het aantal vergelijkingen in de praktijk voor de drie sorteeralgoritmen. (b) Absolute verschil tussen het theoretische aantal vergelijkingen en het aantal vergelijkingen in de praktijk voor quick sort. (c) Aantal vergelijkingen volgens theorie en in de praktijk voor quicksort (Bron: eigen grafiek en data)

Een andere vraag is hoe de algoritmen zich gedragen bij rijen die al gesorteerd zijn (figuur 9). Dan zou in theorie quick sort dezelfde complexiteit hebben (~ $\frac{n^2}{2}$) als selection sort. De oorzaak is de slechte verdeling in deelrijen door e partition functie. Dit is afhankelijk van de implementatie van quicksort. Een variant die eerst shuffelt heeft dit probleem niet. Insertion sort zou (n – 1) vergelijkingen nodig hebben aangezien de buitenste lus vanaf 1 begint. Daardoor wordt de conditie van de binnenste lus (n - 1) keer gecontroleerd (en evenveel keer verworpen). Deze verwachtingen worden bevestigd op onderstaande figuur. Selection sort en quick sort zijn niet van elkaar te onderscheiden terwijl insertion sort minder vergelijkingen nodig heeft dan quicksort er nodig had bij een ongesorteerde rij. Insertion sort heeft inderdaad 3999 vergelijkingen nodig voor een rij van 4000 elementen.

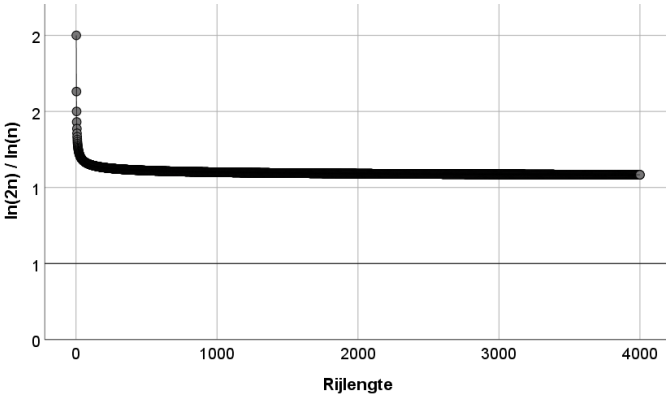


Figuur 9: Aantal vergelijkingen voor oplopende rijlengte van voorgesorteerde rijen voor de drie algoritmen (Bron: eigen grafiek en data)

3.3 Een alternatief: dubbling ratio experimenten

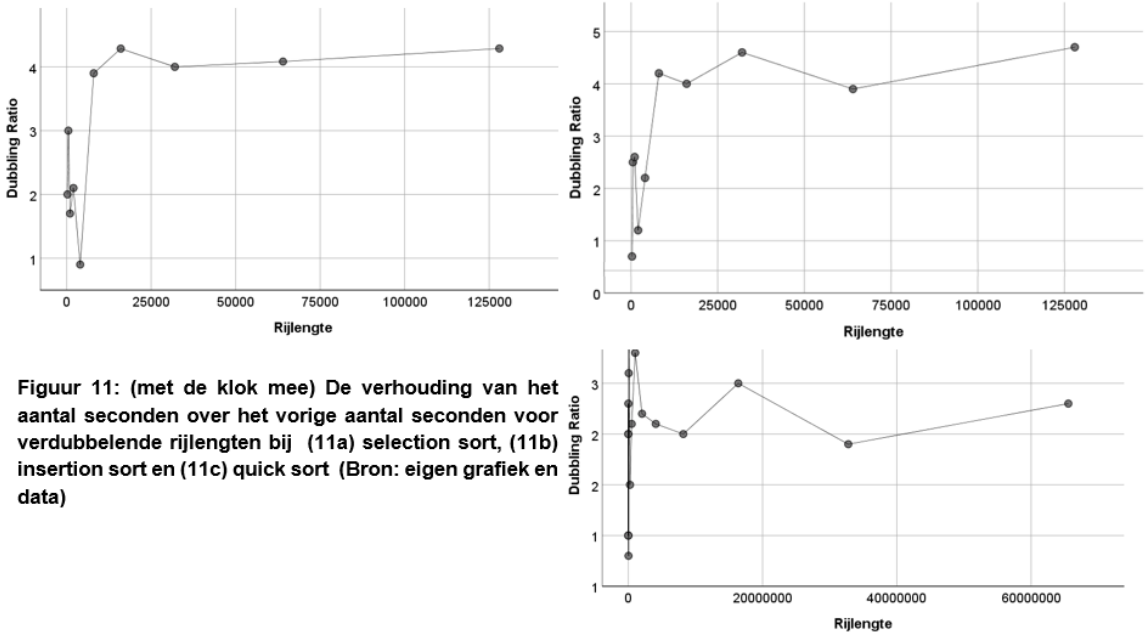
Een dubbling ratio experiment heeft hetzelfde doel als het bepalen van de tijdscomplexiteit, namelijk de prestaties van een algoritme kunnen voorspellen bij een grotere invoer. Alleen levert dit geen algemene functie op, maar een factor van de tweede macht. Concreet wordt telkens de invoergrootte n verdubbelt en de tijd per invoer gemeten. Telkens wordt de verhouding van de tijd op tijdstip$_t$ en de tijd op tijdstip$_{t-1}$ genoteerd. Voor de meeste programma's wordt verwacht dat deze verhouding een limiet bereikt van de vorm 2^b. Eens de dubbling ratio verworven is, is het dus mogelijk om groffe schattingen te maken van de benodigde tijd voor een verdubbelde invoer. Bovendien is er ook een verband met de tilde notatie van de tijdscomplexiteit, dat als volgt genoteerd kan worden: $\frac{T(2n)}{T(n)} \sim 2^b$. In woorden: de verhouding van de tijdscomplexiteit van een dubbele invoergrootte herleid zich tot een factor van de tweede macht. Op basis hiervan kan er als hypothese gesteld worden dat de tijdscomplexiteit van selection sort, insertion sort en quick sort respectievelijk gelijk is aan:

1. $\frac{(2n)^2 \,/\, 2}{n^2 \,/\, 2} = 4$
2. $\frac{(2n)^2 \,/\, 4}{n^2 \,/\, 4} = 4$
3. $\frac{1.39\,(2n)\ln(2n)}{1.39\,n\ln(n)} = 2\,\frac{\ln(2n)}{\ln(n)} \sim 2$ aangezien deze verhouding convergeert naar 1 voor grote waarden zoals te zien is in figuur 10. Bovendien ligt vanaf n = 4,5 de verhouding al onder 1,5.



Figuur 10: Verhouding van het natuurlijk logaritme (Bron: eigen grafiek en data)

In figuur 11 is de grafische weergave van de resultaten van het experiment te zijn.



Figuur 11: (met de klok mee) De verhouding van het aantal seconden over het vorige aantal seconden voor verdubbelende rijlengten bij (11a) selection sort, (11b) insertion sort en (11c) quick sort (Bron: eigen grafiek en data)

De rijlengte werd verdubbelt tot het punt waarop het 'lang begon te duren', in caso er meer dan een minuut nodig was om de rij te sorteren. Bij quick sort wordt de limiet bereikt doordat de rij zijn maximumlengte bereikt heeft. De verwachtingen worden in grote lijnen ingelost, op enkele kanttekeningen na. Zo is er eerder sprake van een gemiddelde waar de dubbling ratio rond schommelt dan een echte convergentie. Bij kleine rijen en ook bij erg grote rijen is er een relatief grote afwijking van de voorspelde dubbling ratio. Desondanks is het duidelijk rond welke macht van twee de dubbling ratio zich bevindt, namelijk de voorspelde waarde.

Via een dergelijk experiment kan dus bepaald worden hoe snel een programma werkt bij een bepaalde invoergrootte. Op deze manier kan voor grote lijsten berekend worden hoeveel tijd de sorteermethoden nodig hebben. Ter illustratie wordt de tijd berekend die elke methode nodig zou hebben bij een invoer die acht keer zo groot is als de grootste invoer van het experiment. Voor selection en insertion sort is dit een rij van 1.024.000 elementen. Dan moet het aantal verdubbelingen om op die rij te komen vanaf 128.000 elementen geteld worden. Dit zijn drie verdubelingen. Dus moet de tijd met 3 keer met de dubbling ratio vermenigvuldigt worden, wat leidt tot ongeveer 23 minuten. Dit leidt ons tot volgende conclusie:

ALGEMEEN

De doubling ratio = $\frac{T(2n)}{T(n)} \sim 2^b$.

De tijd nodig om een rij van kn elementen te sorteren is gelijk aan (dubbling ratio)$^{lg(k)}$ * de tijd om n elementen te sorteren.

Een sorteermethode van de orde $\sim n^5$ heeft dus een doubling ratio van 32. Als quicksort 49,60 seconden nodig heeft om een rij van 65.536.000 elementen te sorteren, dan heeft het 2lg(8) * 49,60s nodig om een rij van 524.288.000 elementen te sorteren. Quick sort heeft dus maar 6,37 minuten nodig voor deze rij, terwijl de twee andere algoritmen al meer dan 20 minuten nodig hebben voor een rij die 64 keer kleiner is.

4. BESLUIT

Deze paper bestudeerde de efficiëntie van drie sorteermethoden. Eerst werd theoretisch de tijdscomplexiteit geschat via de tilde notatie. Het aantal seconden dat elke algoritme nodig had bleek deze schatting grofweg te volgen, maar het bleek niet verfijnd genoeg om een onderscheid tussen selection en insertion sort te maken. Het aantal vergelijkingen in de praktijk bleek de voorspelde complexiteit goed te volgen, terwijl insertion sort het meest geschikt bleek voor kleine rijen. Voor rijen die al gesorteerd zijn heeft quicksort dezelfde tijdscomplexiteit als selection sort (als de pivot het eerste element is en er geen shuffle operatie is). Selection sort lijkt dus in geen enkele situatie aangewezen te zijn, terwijl insertion sort soms wel beter is dan quick sort. De tilde notatie van selection sort sluit het dichtste aan bij de realiteit, terwijl de notatie voor quicksort soms tot 20.000 vergelijkingen minder voorspelde. Een alternatief voor het bepalen van de tijdscomplexiteit werd gevonden in een dubbling ratio experiment. Bovendien kan uit de tijdscomplexiteit ook de dubbling ratio worden afgeleid. Omgekeerd kan de tijdcomplexiteit grofweg geschat worden aan de hand van de dubbling ratio.

5. REFERENTIELIJST

Inoue, H., Moriyama, T., Komatsu, H., & Nakatani, T.. (2012). A high-performance sorting algorithm for multicore single-instruction multiple-data processors. *Software, Practice & Experience*, 42(6), 753–777. https://doi.org/10.1002/spe.1102

Abdel-Hafeez, S., & Gordon-Ross, A. (2017). An Efficient O(N) Comparison-Free Sorting Algorithm. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 25(6), 1930–1942. https://doi.org/10.1109/TVLSI.2017.2661746

Sedgewick, R. & Wayne, K. (2019). *Algorithms* (4de editie). Addison-Wesley.