GEGEVENSSTRUCTUREN EN ALGORITMEN: SORTEERALGORITMES

Liam Volckerick

MARCH 28, 2018

KULEUVEN

Inhoudstafel

Inleiding

In dit practicum bespreek ik de voornaamste kenmerken van enkele sorteeralgoritmen. Quick Sort, Selection Sort en Insertion Sort zullen in dit verslag in voldoende mate uitgelegd worden a.d.h.v. enkele experimenten zoals onder hande het doubleratio experiment. De keuze van het sorteeralgoritme hangt volledig af van welke data deze krijgt te verwerken. Zo zullen er zeer snelle algoritmes bestaan die bij lage invoer veel efficiënter zijn dan enkele andere algoritmes die het dan juist zeer goed doen voor een grote invoer. Enkele van deze verschillen zullen besproken worden van bovenstaand vermelde sorteeralgoritmes.

Rekenwerk van sorteeralgoritmes

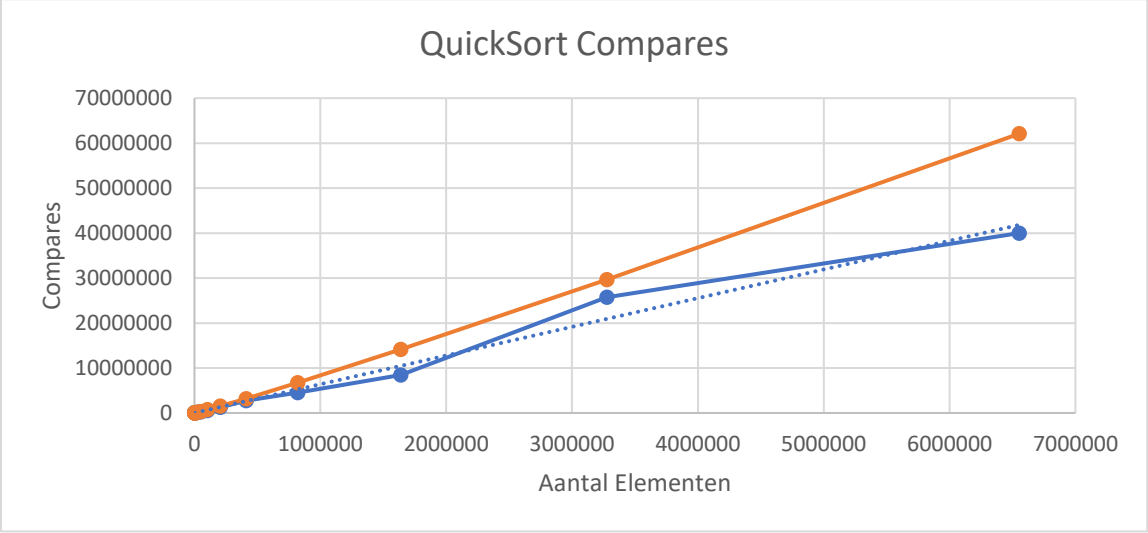
Quick Sort

Quick sort is de meest bekende voor het oor van een informaticus, onder alle sorteeralgoritmen. Zijn efficiëntie van haast lineair te zijn, $n \log_2 n$ als ook in zijn slechtste geval $n^2/2$. Zijn efficiëntie haalt hij uit zijn binnenste loop waardat we een array element vergelijken met een vaste waarde. De index van dit array element verhogen we steeds waardoor dit de efficiëntie van quicksort enorm verhoogt. Een andere grote factor die deze efficiëntie ten goede doet is dat deze weinige vergelijkingen maakt. Doordat we in quick sort de array verdeelt in twee evenwaardig grote arrays, in zijn optimale toestand, in elke andere toestand is deze casus hoogstwaarschijnlijk door middel van de random factor die onze pivot bepaalt. De pivot is het element waar rond zich de twee nieuwe subarrays zich rond gaan begeven.

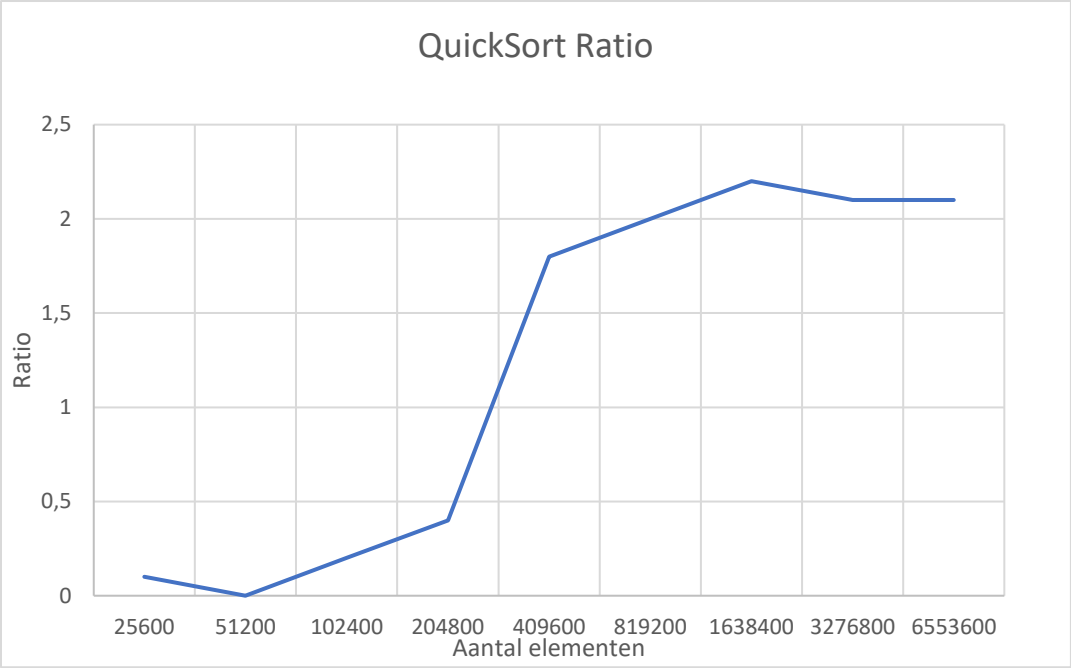
In zijn beste toestand zal de pivot zodanig gekozen worden zodat deze steeds de meegegeven array exact splitst in twee even grote subarrays. Waardoor we het aantal vergelijkingen van dit algoritme ongeveer kunnen gelijk stellen aan $2nln(n)$.

In zijn slechtste toestand zal quick sort $n^2/2$ vergelijkingen hanteren maar door in de binnenste loop van quick sort(het verdelen van de huidige array naar twee subarrays de gekozen pivot aan een random gekozen getal te duiden, zal dat voorkomen dat we op deze toestand terecht komen. Deze slechtste toestand zal in een onefficiënt sorteeralgoritme resulteren. De hoeveelheid geheugen die nodig zal zijn om deze gegeven array te sorteren zal zich lineair gedragen. Dit is zeer slecht indien we te maken hebben met enorm grote arrays. Maar door hoe dat quick sort in elkaar zit is de werkelijke kans enorm klein dat we ooit te maken zullen hebben met de slechtste toestand en dus blijft quick sort zijn efficiënte hangen rond zijn gemiddelde aantal vergelijkingen. Deze bedraagt $1{,}39n \log_2 n$. De wiskundige afleiding van het bedrag van het gemiddeld aantal vergelijkingen houd ik buiten dit verslag. Een andere optimalisatie om deze slechtste toestand tegen te gaan zou kunnen zijn door het mediaan te nemen van drie random getallen uit de array en deze dan gelijk te stellen aan de pivot. Hierdoor zal de slechtste toestand nooit aangehaald worden.

In onderstaande grafiek wordt het aantal elementen in fucntie van het aantal vergelijkingen gesteld. In mijn testen zien we dat mijn meetresultaten zich beter voor deden dan de verwachte curve van $1{,}39n \log_2 n$. Waarbij n het aantal getallen bedraagt die zich in de array bevinden. Dit kan zijn doordat de pivot steeds gunstelijk werd gekozen.



Figuur 1: QuickSort vergelijkingen, blauw: eigenlijke, oranje: verwacht volgens avg case



Figuur 2: DoubleRatioExperiment Quick Sort

Quick sort gedraagt zich bij verdubbeling van het aantal elementen vrij stabiel rond factor 2. Indien de invoer van het algoritme te klein is, zullen deze tijden zodanig klein zijn dat we deze niet kunnen uitdrukken in milliseconden maar wel in nanoseconden. Indien elke meting zou zijn gedaan in nanoseconden, zouden we een grafiek vinden die zich rond de constante 2 zal begeven en dus een y-functie zal afbeelden zoals in het verdere verloop zou worden aangetoond in bovenstaande grafiek.

Bij mijn grootste meting bedraagde de uitvoeringstijd van 6553600 elementen 1018ms wat ongeveer een seconde bedraagt. Stel dat we een invoer nemen van 52428800 dan zal de uitvoeringstijd 8 keer zo lang duren. Wat dus zal uitkomen op 8144 ms ≅ 8 seconden. We bekomen deze uitkomst doordat we de invoer vergroten met 8. Acht is een macht van twee, deze bedraagt drie. Wat ons dus zal resulteren, met behulp van de ratio van ongeveer 2 bij verdubbeling, op een tijdsduur van 2^3 -keer de voorgaande tijd.

Insertion Sort

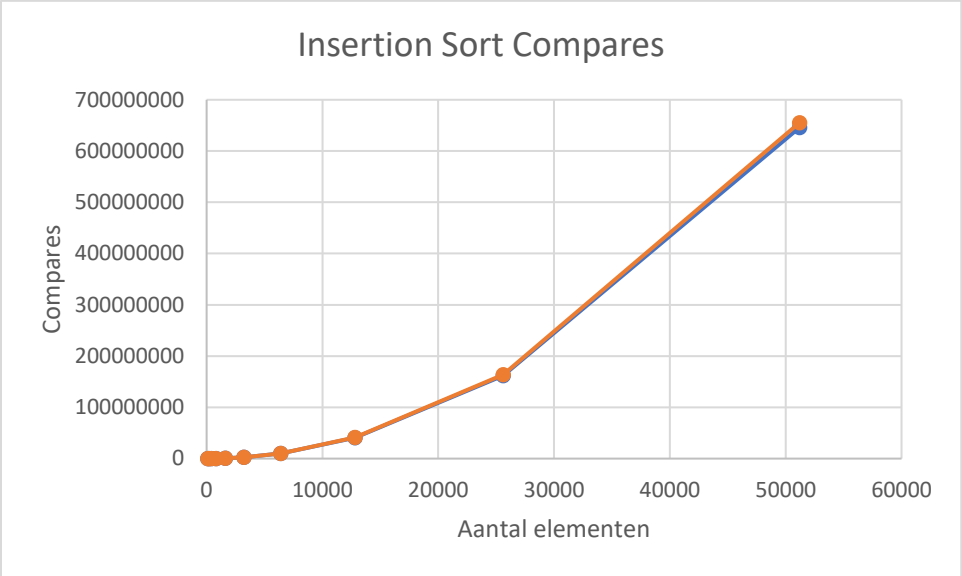
Insertion sort is een sorteeralgoritme dat te vergelijken valt met het ordenen van de kaarten in onze hand tijdens een kaartspel. Dusdanig zal insertion sort de gegeven input steeds overlopen en vergelijken met één element. Hierdoor zal hij elk moment ordenen door steeds met één ander vast element te vergelijken. Dit vaste element staat reeds al gesorteerd en wordt opgehoogd met één wanneer de hele invoer is vergeleken met deze waarde. Hierdoor zal insertion sort veel efficiënter zijn dan enig welke andere sorteeralgoritme bij een grote invoer dat reeds al wat gesorteerd is. Hierdoor zullen de swaps en de tijdscomplexiteit dusdanig afnemen.

In het beste geval zal insertion sort haast een lineair aantal vergelijkingen maken. In een reeds voorgesorteerde array zal het aantal vergelijkingen n-1 bedragen. Dit aangezien insertion sort de hele array afgaat zonder dat er zich een verplaatsing hoort plaats te vinden volgens het sorteer principe van insertion sort.

In het slechtste geval zal insertion sort een aantal vergelijkingen van $\frac{n^2}{2}$ hanteren. En dus ook $\frac{n^2}{2}$ aantal swaps. Dit zal alleen gebeuren wanneer er bij elke verhoging van de maximale index een nieuwe waarde wordt toegevoegd die blijkt aan het begin te staan van die array. Hierdoor zullen we

voor het laatste element van een n-lange array n-1 compares moeten doen. Dit leidt tot gevolg dat bij een willekeurige array met lengte n het aantal compares zal bestaan als volgt: $1 + 2 + 3 + \cdots + (n-2) + (n-1) = (n^2 - n)/2 \cong n^2/2$.

Het gemiddelde aantal compares bedraagt $n^2/4$. Deze is zeer mooi weergeven in onderstaande figuur 3. De testen die ik heb uitgevoerd op insertion sort kwamen praktisch bijna altijd neer op de curve van het verwachte gemiddelde van insertion sort.



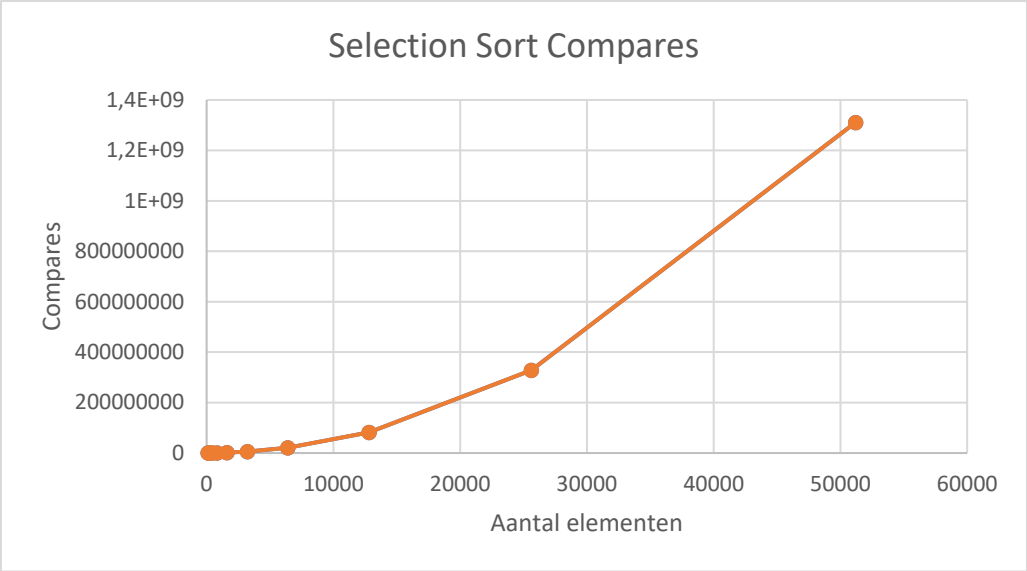
Figuur 3: Insertion Compares, Blauw: eigenlijke meting, Oranje: verwachte resultaat volgens avg. case.

Het double ratio experiment bij insertion sort toont aan dat insertion sort niet geschikt is voor zeer gorte arrays. Zo zien we in figuur 4 dat dit sorteeralgoritme leidt tot een ratio van ongeveer 4, bij verdubbeling van de array zal dus de noodzakelijke tijd om de array te sorteren met een factor van 4 vermenigvuldigd wordt. Bij mijn grootste meting bedraagde de uitvoeringstijd van 12800 elementen 247ms. Stel dat we een invoer nemen van 102400 dan zal de uitvoeringstijd 64 keer zo lang duren. Wat dus zal uitkomen op 15808 ms ≅ 16 seconden. Hier vermenigvuldigen we de voorgaande tijdsduur met de ratio tot de derde macht. Aangezien we de input vergroten met acht en $2^3 = 8$ zullen we dus uitkomen dat de verwachte tijdsduur voor acht keer grotere input $4^3 * TijdErvoor = NieuweTijdsDuur$.

Figuur 4: DoubleRatio experiment Insertion Sort

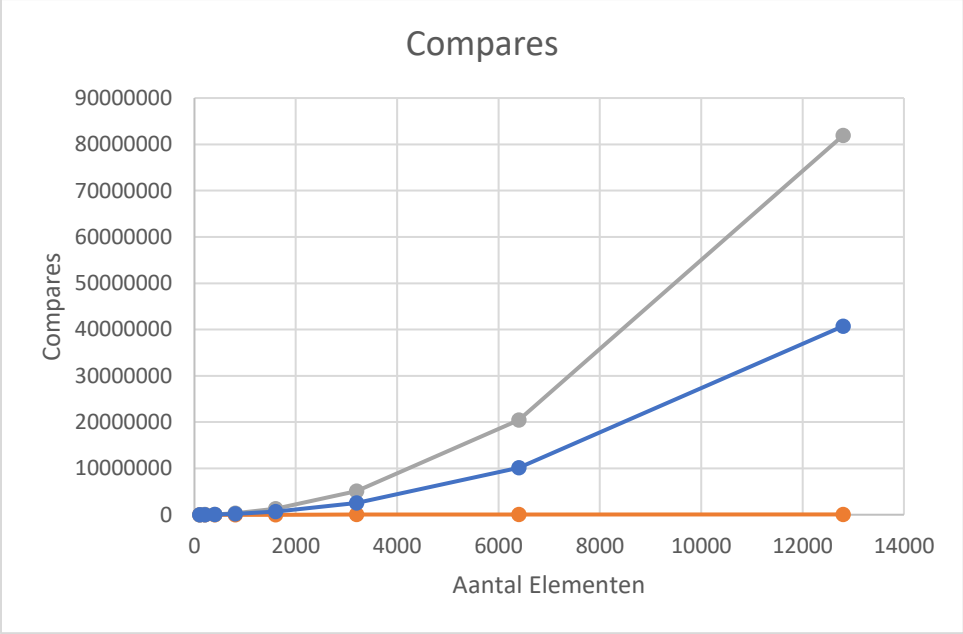
Selection Sort

Selection sort is een sorteeralgoritme dat berust op het kleinste element te vinden in de huidige gegeven array en deze te plaatsen op de top van de array. Hierna volgt hetzelfde principe voor het tweede element van de array enz. Dit wordt gehanteerd voor elk ander overblijvend element van de array. Selection sort is een sorteeralgoritme dat niet afhangt van de structuur van de gegeven data. Hiermee bedoel ik of de gegeven input volledig random is, reeds gesorteerd of half op volgorde gezet. Het algoritme zal altijd de hele array afgaan om het kleinste element te vinden. Het enige dat de snelheid van selection sort aantast is de grootte van de input. Hoe groter deze wordt, hoe langer selection sort erover zal doen om deze array te sorteren. Dit algoritme neemt altijd de vaste hoeveelheid vergelijkingen aan van tilde $\frac{n^2}{2}$ of ookwel n(n-1)/2 om zeer precies te zijn. Deze komt doordat selection sort telkens dat het het kleinste element heeft gevonden, het aantal 'te onderzoeken elementen' met steeds één vermindert. Waardoor we tot de volgende afleiding komen van het 'aantal vergelijkingen dat gemaakt worden': $(n-1)+(n-2)+\cdots+3+2+1+0=(n^2-n)/2 \cong n^2/2$. Aangezien deze aantal vergelijkingen nooit kan afwijken, verbaast het niet dat de uitgevoerde testen perfect overeenkomen met de verwachte aantal vergelijkingen bij verschillende aantal elementen. Dit valt te zien in figuur 5.



Figuur 5: aantal vergelijkingen selection sort, blauw: meetresultaten, oranje: verwachte resultaten

Vaststelling



Figuur 6: Oranje: QuickSort, Blauw: InsertionSort, Grijs: SelectionSort

Op zeer grote data zien we dat quick sort ruim het meest efficiënte sorteeralgoritme is op random data. Waarom dit zo is, werd reeds boven al uitgewerkt. Als volgende komt insertion sort waarna selection sort opvolgt. In het slechtste geval van insertion sort zou insertion sort overeenkomen met de grafiek van selection sort. Bij kleinere inputs hangt het dan weer af hoedanig de input reeds is gesorteerd. Indien deze reeds gesorteerd is, zal insertion sort de meest efficiënte zijn. Al is de kans zeer klein indien we at random te werk gaan om de invoer door te geven.

Verdubbelingsexperiment gegeven algoritme

Aangezien we het verdubbelingsexperiment noteren als volgt: $\frac{T(2n)}{T(n)} = \frac{a(2n)^2}{an^2}$. Hieruit kunnen we afleiden bij een $\sim n^5$ algoritme dat $\frac{a(2n)^5}{an^5} = 2^5$ en dus bij een $\sim n^5$ algoritme de verdubbelingsratio 32 bedraagt. Bij elke verdubbeling van het aantal elementen zal dus de tijdscomplexiteit vergroten met een factor 32.