Schakelprogramma: Master in de toegepaste informatica

Gegevensstructuren en algoritmen

Practicum 1

Dries Janse
r0627054
29 maart 2019

Inhoud

1. Inleiding

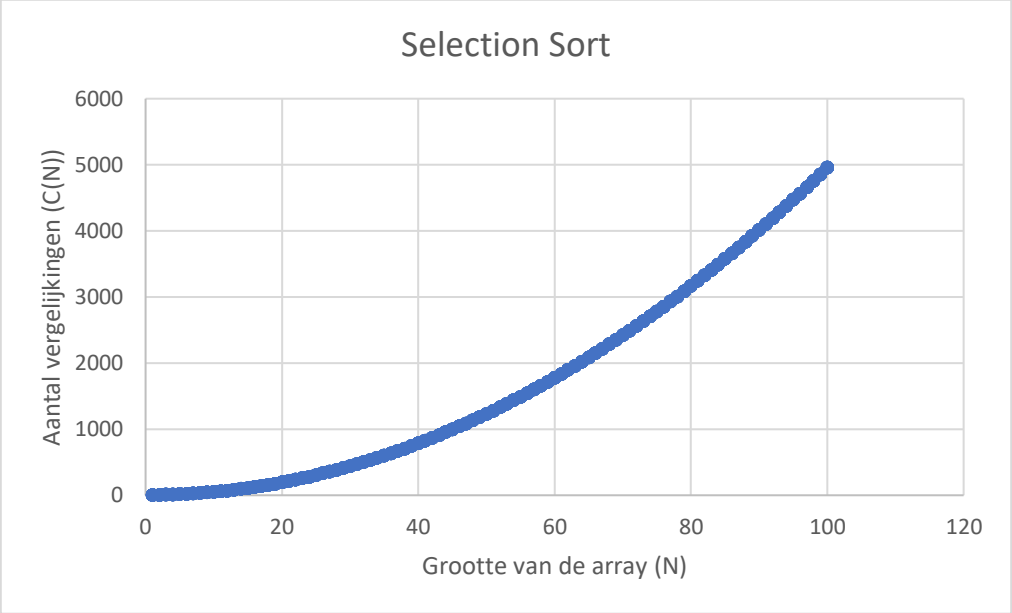
Dit verslag is geschreven naar aanleiding van het eerste practicum van het opleidingsonderdeel: Gegevensstructuren en algoritmen. Dit als onderdeel van het schakelprogramma: Master in de toegepaste informatica.

In dit practicum wordt besproken hoe efficiënt selection sort, insertion sort en quicksort zijn op random data. Dit gebeurt aan de hand van experimenten op elk van deze drie sorteeralgoritmen. Ten eerste wordt er besproken hoeveel onderlinge vergelijkingen er nodig zijn om invoer van verschillende grootte te sorteren. Ten tweede, wordt er een uitvoering van het doubling ratio experiment besproken voor quicksort en insertion sort. Hierin wordt bekeken of het werkelijke resultaat overeenstemt met het verwachte theoretische resultaat.

2. Vergelijkingen

In dit onderdeel worden de resultaten van de experimenten visueel voorgesteld per algoritme. Hierin heb ik telkens gebruik gemaakt van invoergroottes (N) lopende van 1 tot en met 100, zoals ook aangegeven werd in de opgave. Per invoergrootte werd het experiment 16 maal herhaald. Elke grafiek geeft het aantal vergelijkingen weer die nodig zijn om de invoer van de gegeven grootte te sorteren.

2.1 Selection sort



Figuur 1: Aantal vergelijkingen in verhouding tot de grootte van selection sort.

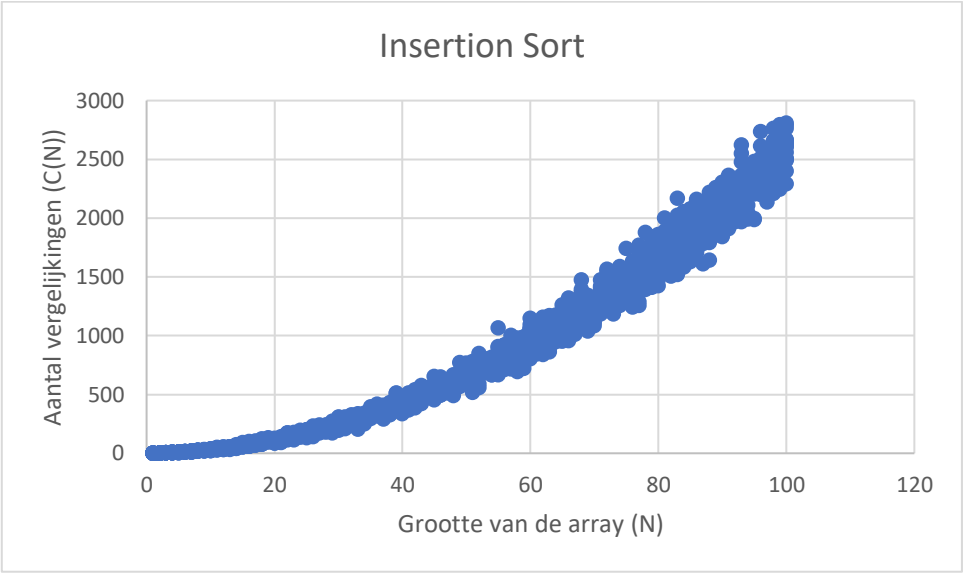
Het allereerste wat opvalt bij selection sort, en dat we niet kunnen terugvinden in een ander algoritme dat besproken wordt in dit verslag, is dat we een perfecte lijn kunnen trekken door de geplotte punten. Dit komt doordat er steeds eenzelfde aantal vergelijkingen uitgevoerd worden voor een gegeven grootte N. De resterende array wordt steeds doorlopen, op zoek naar het volgende kleinste getal. We kunnen dit verklaren, door de som van de volgende rekenkundige rij:

$$(n-1)+(n-2)+(n-3)+\cdots+2+1=\frac{n(n-1)}{2}$$

Bovenstaande formule zegt dat er n-1 vergelijkingen nodig zijn om het kleinste element te vinden in een rij van grootte N. Voor het volgende kleinste element te vinden zijn er nog n-2 vergelijkingen nodig. Dit gaat zo verder tot er één element overblijft dewelke 0 vergelijkingen nodig heeft. Selection sort is dus van grootte : $\sim\frac{n^2}{2}$, wat van kwadratische vorm is. Bovenstaande grafiek weerspiegelt perfect het theoretische model.

Hieruit kan besloten worden dat er bij eenzelfde probleemgrootte evenveel vergelijkingen moeten gemaakt worden ongeacht de volgorde van de waarde bij invoer.

2.2 Insertion sort



Figuur 2: Aantal vergelijkingen in verhouding tot de grootte van insertion sort.

Bij insertion sort kunnen we onmiddellijk zien dat er voor eenzelfde grootte N een verschillend aantal vergelijkingen mogelijk zijn. Dit kan verklaard worden doordat het aantal vergelijkingen afhankelijk is van de initiële volgorde van de elementen bij invoer.

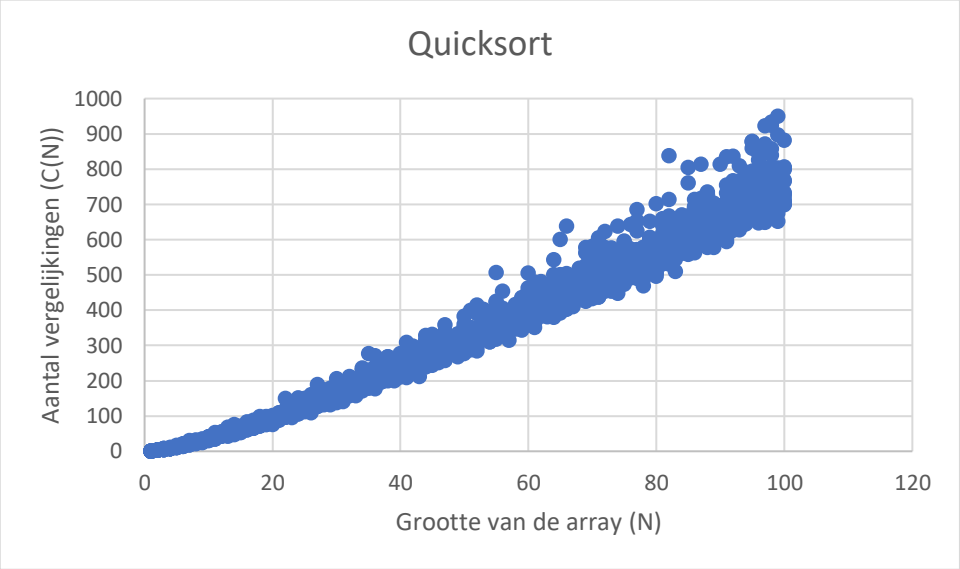
In het *beste* geval, dit is wanneer de array al gesorteerd is, heeft insertion sort n-1 vergelijkingen nodig wat neerkomt op: $\sim n$. We kunnen hier ook bemerken dat er 0 exchanges nodig waren.

In het *slechtste* geval, dit is wanneer de array in omgekeerde volgorde gesorteerd is, gebruikt insertion sort: $\sim \frac{n^2}{2}$ vergelijkingen. Dit is bekomen door de som te nemen van dezelfde rekenkundige rij als in de vorige sectie. Het eerste element gebruikt 0 vergelijkingen, het tweede maakt er 1, het derde maakt er 2, enzoverder. Met andere woorden is het aantal vergelijkingen van insertion sort in het slechtste geval gelijk aan deze van selection sort.

Al de geplotte punten in bovenstaande grafiek liggen dus tussen: $\sim n$ en $\sim \frac{n^2}{2}$ vergelijkingen. Uit het beste en slechtste geval kunnen we stellen dat er gemiddeld gezien: $\sim \frac{n^2}{4}$ vergelijkingen nodig zijn om een rij te sorteren. Wat gemiddeld gezien dubbel zo snel is dan selection sort voor een voldoende grootte N.

Een laatste punt dat opgemerkt kan worden uit de grafiek is dat de punten meer verspreid liggen naarmate de invoergrootte N groter wordt. Dit is te verklaren doordat het interval tussen $\sim n$ en $\sim \frac{n^2}{2}$ steeds groter wordt naarmate de N toeneemt.

2.3 Quicksort



Figuur 3: Aantal vergelijkingen in verhouding tot de grootte van quicksort.

Net zoals bij insertion sort kunnen we afleiden dat er voor eenzelfde grootte N een verschillend aantal vergelijkingen nodig zijn. Dit kan verklaard worden door de keuze van de pivot. De pivot wordt op de juiste positie geplaatst in de array. Alle elementen voor de pivot zijn kleiner dan de pivot, alle elementen achter de pivot zijn groter dan de pivot. Beide secties in deze array worden recursief verder gesorteerd.

In het *slechtste* geval, krijgen we steeds 0 elementen voor (ofwel achter) de pivot en de resterende n-1 elementen achter (ofwel voor) de pivot. We kunnen hier ook wel zeggen dat de splitsing niet gebalanceerd is. De tijdscomplexiteit is $\sim\frac{n^2}{2}$.

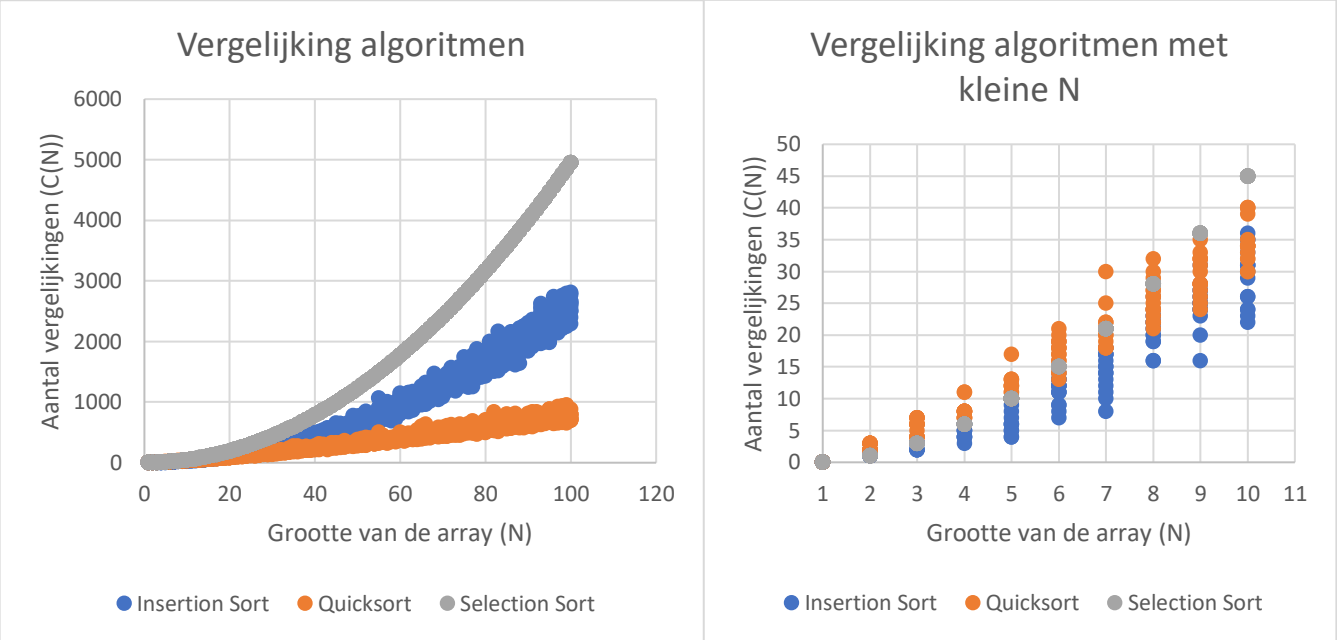
In het *beste* geval, hebben we steeds evenveel elementen voor de pivot als achter de pivot. Wat neerkomt op $\sim n \log_2 n$. En *gemiddeld* geeft dit $\sim 1.39n \log_2 n$. Voor de uitwerking van deze beide gevallen zou ik willen verwijzen naar de slides.

In mijn implementatie wordt steeds het eerste element gekozen als pivot. Ook wordt er geen gebruik gemaakt van een initiële shuffle van de array zoals vermeld in de opgave. Dit is een 'bad practice', want zo kunnen we niet steeds de willekeurigheid garanderen. Stel dat de array al gesorteerd was (of in omgekeerde volgorde) dan komt men steeds in het slechtste geval terecht. Om dit te voorkomen kan er gebruik gemaakt worden van een initiële shuffle of de pivot moet willekeurig gekozen worden.

Een andere optimalisatie die toegepast kan worden is de mediaan nemen van 3 gekozen elementen uit de rij. Deze mediaan wordt dan gebruikt als pivot.

Ook bij quicksort wil ik bemerken dat de punten meer verspreid liggen naarmate de invoergrootte N groter word. Dit is te verklaren doordat het interval tussen $\sim\frac{n^2}{2}$en $\sim n \log_2 n$ steeds groter wordt naarmate de N toeneemt.

2.4 Conclusie



Figuur 4: Vergelijking tussen de vorige algoritmen.

Alle voorgaande algoritmen kunnen we makkelijk visueel vergelijken met elkaar, zoals weergegeven in figuur 4. Naarmate de N groter wordt zien we een duidelijk verschil tussen het aantal vergelijkingen dat elk algoritme nodig heeft. We zien hier dat quicksort het traagste stijgt en hierdoor het minst aantal vergelijkingen nodig heeft.

Toch wil ik graag bemerken dat quicksort niet het meest efficiënte is, in functie van het aantal vergelijkingen, op een kleine array grootte (N). Uit de rechtste grafiek van figuur 4, kunnen we afleiden dat insertion sort minder vergelijkingen nodig heeft dan quicksort om de array te sorteren.

Een mogelijke optimalisatie die toegepast kan worden bij quicksort is het gebruik maken van insertion sort bij kleine subarrays.

Dit zou gewijzigd kunnen worden door het volgende te vervangen:

If(hi <= lo) return;

Door,

If(hi <= lo + CUTOFF) { new InsertionSort.sort(a, lo, hi) ; return ;}

Hierin kunnen we CUTOFF kiezen tussen 5 en 15.

Een laatste bemerking die ik heb bij figuur 4, is dat quicksort een kleinere spreiding heeft dan insertion sort. Dit is te wijten doordat het interval tussen best case en worst case kleiner is bij quicksort.

3. Doubling ratio experiment

In deze sectie bespreek ik het doubling ratio experiment voor quicksort en insertion sort. Dit experiment werd uitgevoerd zoals beschreven in het handboek op bladzijde 192. Ik begin telkens met een array met 2500 willekeurige elementen en blijf deze verdubbelen tot ik een array met 160000 elementen heb. De fysieke tijd in beide experimenten is steeds uitgedrukt in milliseconden.

3.1 Insertion sort

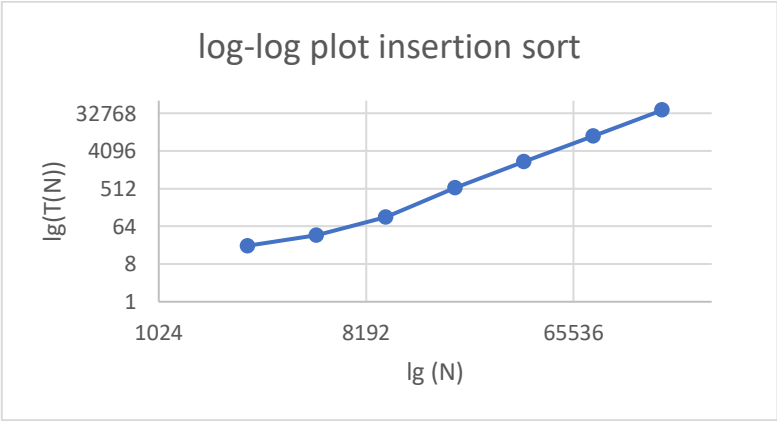
Grootte array (N)	Aantal milliseconden	Doubling ratio	Log ratio
2500	22		
5000	39	1.7727	0.8259
10000	107	2.7436	1.4560
20000	546	5.1028	2.3513
40000	2252	4.1245	2.0442
80000	9289	4.1248	2.0443
16000	38959	4.1941	2.0683

Tabel 1: waarden doubling ratio van insertion sort.

Bovenstaande tabel geeft de waarden weer van het doubling ratio experiment voor insertion sort. Deze doubling ratio is berekend met de volgende formule: $\frac{T(2n)}{T(n)}$. Ook weten we dat insertion sort van de volgende complexiteit is: $\sim\frac{n^2}{4}$. Met vorige 2 formules is het makkelijk om de theoretische doubling ration te berekenen:

$$\frac{T(2n)}{T(n)} = \frac{(2n)^2}{4} * \frac{4}{n^2} = \frac{16}{4} = 4$$

Als we naar de berekende waarden kijken in tabel 1, is deze doubling ratio niet gelijk aan 4. Dit zouden we kunnen wijten aan mijn computer; doordat ook andere programma's resources tegelijkertijd gebruiken.



Figuur 5: Log-log plot van uitvoer tijd voor een probleemgrootte N van insertion sort.

In figuur 5, zien we dat we voor een groter wordende N een rechte benaderen. Deze rechten kan berekend worden door:

$$\log_2 T(N)) = b * \log_2 N + C$$

$$T(N) = a * N^b, met\ a = 2^c$$

Tot slot neem ik een invoer die 8 maal groter is dan mijn grootste meting en voorspel ik de uitvoeringstijd. De tijd nog om array van 128000 elementen te sorteren is 2553217025 milliseconden of wel 42553 minuten.

$$T(128000) = 4^8 * 38959 = 2553217024$$

3.2 Quick sort

Grootte array (N)	Aantal milliseconden	Doubling ratio	Log ratio
2500	2		
5000	2	1.0	0.
10000	3	1.5	0.58
20000	6	2.0	1
40000	6	1.0	0
80000	11	1.8	0.87
16000	25	2.3	1.18

Tabel 1: waarden doubling ratio van quick sort.

Bovenstaande tabel geeft de waarden weer van het doubling ratio experiment voor quicksort. Deze doubling ratio is net zoals in vorige sectie berekend met de volgende formule: $\frac{T(2n)}{T(n)}$. Ook weten we dat quicksort van de volgende complexiteit is: $\sim 1.39n\log_2(n)$. Met vorige 2 formules is het makkelijk om de theoretische doubling ration te berekenen:

$$\frac{T(2n)}{T(n)} = \frac{2.78n\log_2(2n)}{1.39n\log_2(n)} = 2 * \frac{(\log_2(n) + \log_2(2))}{\log_2(n)}$$

Uit deze formule kunnen we afleiden dat voor enorm grootte N de doubling ratio naar 2 gaat. Voor andere N waarden, kunnen we de doubling ratio berekenen met bovenstaande formule.

Tot slot neem ik een invoer die 8 maal groter is dan mijn grootste meting en voorspel ik de uitvoeringstijd. De tijd nog om array van 128000 elementen te sorteren is 1638400 milliseconden of wel 27 minuten.

$$T(128000) = 4^8 * 25 = 1638400$$

We kunnen hieruit besluiten dat quicksort veel sneller sorteert dan insertion sort voor een probleem van deze grootte.

4. Doubling ratio bij $\sim n^5$

Als we een algoritme hebben van complexiteit: $\sim n^5$, dan kunnen we berekenen dat de doubling ratio gelijk is aan 32. Omdat 2^5 = 32, kunnen we stellen dat T(2N) gelijk is aan 32 keer T(N).