Katholieke
Universiteit
Leuven

**Department of
Computer Science**

GEGEVENSSTRUCTUREN EN ALGORITMEN:

Practicum 1

Mats Fockaert r0695565

Academic year 2020–2021

Inhoudsopgave

Lijst van figuren

Lijst van tabellen

1 Introductie

In dit practicum is er een hands-on ervaring met de geziene theorie i.v.m de eerste lessen. Deze lessen brachten de algoritmes, Selection sort, Insertion sort en Quicksort, naar voren.

Om een beter gevoel te krijgen voor deze algoritmes zijn testen en analyses vereist. In dit practicum volgen we de efficientie en de tijd op die nodig is om een algoritme correct te doen lopen. De efficientie hangt af van de hoeveelheid vergelijkingen die het sorteeralgoritme maakt omwille van het feit dat deze het meeste gewicht over tijd draagt.

Dit rapport probeert een antwoord te vinden op de vraag, "hoe efficient is elk algoritme voor een willekeurige array". De vereiste experimenten zijn:

- Laat alle drie sorteeralgoritmes lopen met arrays van verschillende lengtes en meet de hoeveelheid vergelijkingen. Plot deze resultaten.
- Voer een doubling-ratio experiment uit voor Quicksort en Insertion sort. Welk resultaat wordt theoretisch gezien verwacht? Is dit ook wat het experiment aantoont? Voorspel de 'running-time' wanneer er een grotere array wordt gegeven voor Quicksort en Insertion sort.
- Welk doubling-ratio is te verwachten voor een $\sim n^5$ algoritme?

2 Aanpak

De algoritmes werden geimplementeerd zoals gezien in het book [1] met kleine aanpassingen voor test en plot doeleinden. Elk algoritme werd geanalyseerd op het aantal vergelijkingen tussen de elementen in de array dat gesorteerd moest worden. Preliminaire testen werden uitgevoerd om snel na te kijken of de algoritmes correct uitgevoerd moeten worden. Deze test werd met een kleine set data van 10 elementen lang uitgevoerd. Voor grotere testen startte de lengte van de array met 0 elementen en nam het toe tot 1000 elementen in de array. De elementen in de array werden willekeurig gegenereerd en varieerden van 0 tot 1 000 000.

De tweede test was enkel voor Insertion sort en Quicksort, deze was het doubling-ratio experiment. Dit betekent dat beide startte met een array dat zal verdubbelen in lengte bij elke iteratie. De tijd die nodig is om een iteratie uit te voeren, wordt bijgehouden. De rijen startte met een lengte van 250 en groeide voor Insertion sort tot 512 000 en voor Quicksort tot 65 536 000.

Dit practicum is gemaakt met:

- Java, voor de code van de algoritmes waar de Comparable type gebruikt werd,
- JUnit, voor de testen, en
- Python, voor de dataverwerking en -vertoning (de werking van dit kan ook gevonden worden in een Jupyter Notebook verslag onder: "src/gna/csvs/data_plotter - Jupyter Notebook.pdf").

[1] Sedgewick, R., & Wayne, K. (2011). Algorithms (4th Edition).

3 Resultaat van de algoritmes

Lengte van de array	Selection sort	Insertion sort	Quicksort
10	45	20	30
20	190	80	82
50	1225	589	302
100	4950	2 622	802
200	19900	9 578	1 703
500	124750	62 296	5 315
1000	499500	252 570	12 580

Tabel 1: Partiele vertoning van de resultaten van de testen op de hoeveelheid vergelijkingen op de verschillende algoritmes.

Leuke anekdote Crashes in java, java.lang.OutOfMemoryError: Java heap space. Tijdens het runnen van het Doubling-ratio experiment, met Quicksort, bij 131 072 000 aantal elementen in de array, had mijn computer geen geheugen meer.

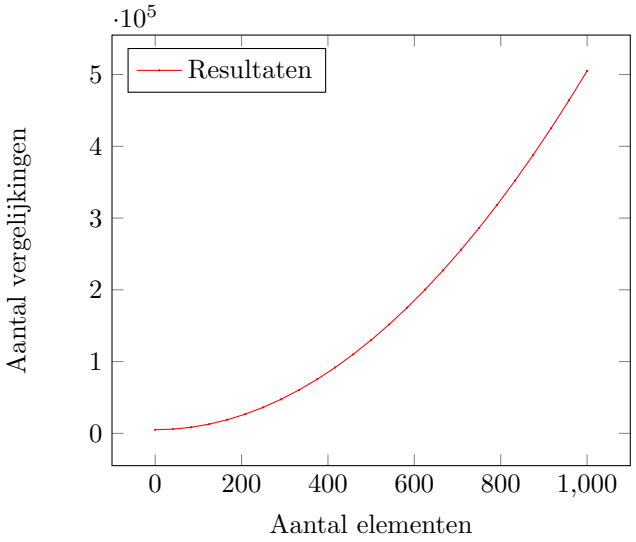
Aantal elementen in de rij	Insertion sort Uitvoeringstijd	dbr	Quicksort Uitvoeringstijd	dbr
250	$< 0,001$	/	$< 0,001$	/
500	$0,001$	/	$< 0,001$	/
1000	$0,002$	$2,0000$	$< 0,001$	/
2000	$0,006$	$3,0000$	$< 0,001$	/
4000	$0,014$	$2,3300$	$< 0,001$	/
8000	$0,056$	$4,0000$	$0,001$	/
16000	$0,231$	$4,1250$	$0,002$	$2,0000$
32000	$1,047$	$4,5325$	$0,004$	$2,0000$
64000	$4,714$	$4,5024$	$0,007$	$1,7500$
128000	$19,624$	$4,1629$	$0,019$	$2,71429$
256000	$104,683$	$5,3344$	$0,046$	$2,42105$
512000	$779,293$	$7,4443$	$0,137$	$2,97826$
1024000	/	/	$0,358$	$2,61314$
2048000	/	/	$0,617$	$1,72346$
4096000	/	/	$1,378$	$2,23339$
8192000	/	/	$3,101$	$2,25036$
16384000	/	/	$7,244$	$2,33602$
32768000	/	/	$16,051$	$2,21576$
65536000	/	/	$36,391$	$2,4634$

Tabel 2: De uitvoeringstijd in seconden en de doubling ratio (dbr) voor insertion sort en quicksort.[2]

[2]De plaatsen met / zijn te klein om te berekenen of werden niet uitgevoerd tijdens de test.

3.1 Selection sort

Tabel 1 en grafiek 1 voor selection sort tonen dat het aantal vergelijkingen kwadratisch stijgt met de grootte van de array die gesorteerd moet worden, de curve die gemeten wordt is bijna equivalent met die van $x^2/2$.

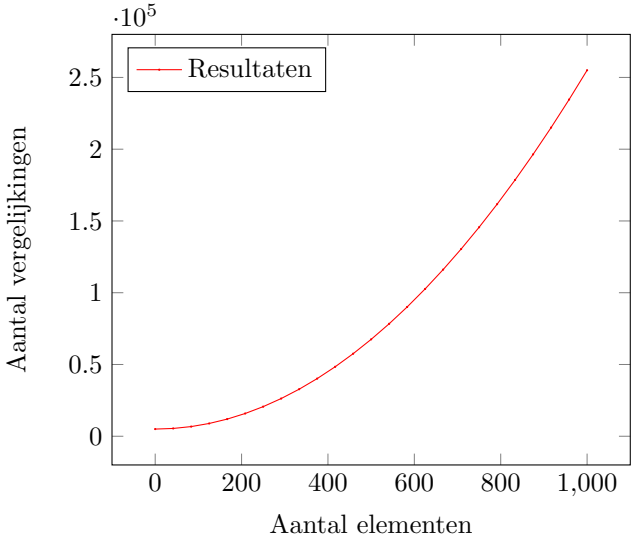


Figuur 1: Selction sort: aantal vergelijkingen in een array met een bepaald aantal elementen.

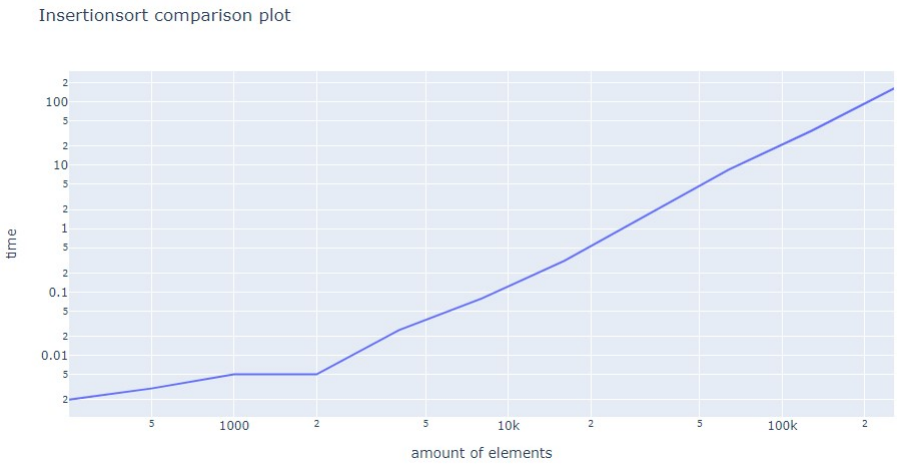
3.2 Insertion sort

Tabel 1 en grafiek 2 tonen bijna dezelfde resultaten als Selection sort, enkel is hier de stijging half zo stijl ($x^2/4$).

De resultaten voor het Doubling-ratio experiment zijn getoond in tabel 2 en op figuur 3, de resultaten leken 4 te volgen maar dan stegen ze tot 7.



Figuur 2: Insertion sort: aantal vergelijkingen in een array met een bepaald aantal elementen.

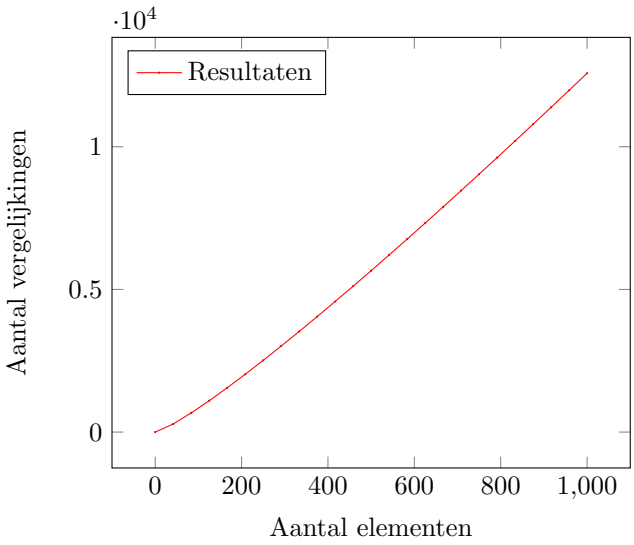


Figuur 3: Insertion sort: doubling ratio experiment.

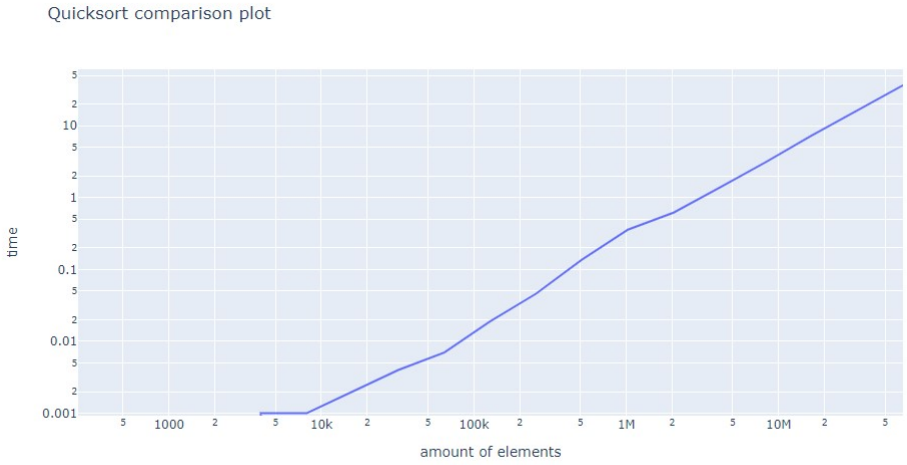
3.3 Quicksort

Tabel 1 en grafiek 4 tonen een linearitmische stijging die lijkt op $2nlogn$.

Het Doubling-ratio experiment toonde een gemiddelde verhouding van 2 (tabel 2 en figuur 5). Als we kijken naar de curve van figuur 5, is het duidelijk dat dit lijkt op deze in figuur 4, waarbij deze een linearitmische stijging heeft.



Figuur 4: Quicksort: aantal vergelijkingen in een array met een bepaald aantal elementen.



Figuur 5: Quicksort: doubling ratio experiment.

4 Conclusies van het project

4.1 Selection sort

De resultaten tonen duidelijk een plot equivalent met $n^2/2$, waarbij n de hoeveelheid elementen in de array is.

Dit is simpelweg te verklaren als we kijken wat Selection sort effectief doet. Het itereert n keer over de array achter het element dat vergeleken wordt om het kleinste element te vinden. Sinds er n elementen zijn, moet het algoritme er n keer overgaan. Het maakt dan volgende vergelijkingen:

$$(n-1) + ... + 2 + 1 = (n-1)\frac{n}{2} \tag{4.1a}$$

$$= \frac{n^2 - n}{2} \tag{4.1b}$$

$$= \sim \frac{n^2}{2} \tag{4.1c}$$

Dit is niet super efficient. Het zou dus ook niet gebruikt moeten worden bij grotere arrays.

Het loopt wel in een constante tijd n. Dit betekent dat het altijd zal eindigen in een vast tijdsbestek.

4.2 Insertion sort

De hoeveelheid vergelijkingen zijn vergelijkbaar met deze van Selection sort. Zoals eerder vermeld is de trend van de curve volgens $n^2/4$. Dit is omdat:

- Insertion sort itereert over de array en tegelijkertijd vergelijkt het de twee naburige elementen.
- Wanneer de algortimes het naburig paar vindt en deze zijn niet correct gesorteerd, wisselt hij deze.
- Hierna, vergelijkt hij het element voor het vergeleken paar en kijkt of dit element nog steeds in de volgorde staat met het eerste element van het paar. Indien nodig, wisselt hij de elementen, en herhaalt dit.

We weten nu dat volgende cases zich kunnen presenteren:

- In het beste geval: de array is reeds gesorteerd. Er kan snel over de array gegaan worden. Dit produceert $n-1$ vergelijkingen. De tijd complexiteit is in dit geval lineair: $\sim n$.
- In het slechtste geval: de array is achterwaarts gesorteerd. Dit betekent dat elk element helemaal naar voren gebracht moet worden. We moeten dan $n^2/2$ vergelijkingingen maken (analoog met de berekening in (4.1)).
- Het gemiddelde van de twee cases:

$$\frac{\frac{n^2}{2} + n - 1}{2} = \sim \frac{n^2}{4} \tag{4.2}$$

Dit algoritme is dus sneller dan Selection sort, maar de complexiteit is nog steeds kwadratisch (wat sub-optimaal is) en kan enkele optimalisaties gebruiken.

4.2.1 Doubling-ratio experiments

De resultaten van de verhouding lijken eerst te blijven hangen rond 4, maar verhogen dan naar 5 en vervolgens 7. Dit is waarschijnlijk te wijten aan het feit dat wanneer iets langere tijd wordt uitgevoerd, verschillen in snelheid al snel beginnen op te vallen. Die verschillen in snelheid zullen waarschijnlijk veroorzaakt zijn doordat de computer waarop de tests werden uitgevoerd niet volledig vrijgemaakt was voor die test. Met andere woorden, er waren andere programma's bezig met hun uitvoering, waardoor de resultaten niet honderd procent betrouwbaar zijn.

Desondanks is 4 als doubling ratio wel wiskundig af te leiden. Als we weten dat de gemiddelde tijdscomplexiteit van Insertion sort gelijk is aan $\sim n^2/4$, dan is de doubling ratio als volgt te berekenen:

$$\sim \frac{(2*n)^2}{4} = \sim 4 * \frac{n^2}{4} = \sim n^2 \tag{4.3a}$$

$$\Updownarrow \tag{4.3b}$$

$$\text{doubling ratio} = 4 \tag{4.3c}$$

Deze doubling ratio kunnen we dan gebruiken om de uitvoeringstijd van grote arrays te berekenen:

Aantal elementen in de array	Uitvoeringstijd
256 000	78, 496
512 000	313, 984
1 024 000	1 255, 936
2 048 000	5 023, 744

Tabel 3: Voorspelde uitvoeringstijd in seconden voor een bepaald aantal elementen voor insertion sort

4.3 Quicksort

4.3.1 Aantal vergelijkingen

Bij quicksort zijn de resultaten duidelijk: de trend die te zien is, is grotendeels gelijk aan $2n\,log(n)$. Het algoritme wordt uitgevoerd door de array te verdelen in 2 rijen. Het eerste element wordt vergeleken met de rest van de Aarray, als het element kleiner is dan het eerste element, dan wordt het links van het eerste element geplaatst. Als het element groter is dan het eerste element, dan wordt het rechts van het eerste element geplaatst. Daarna herhaalt het algoritme dit op zowel het eerste als het tweede deel en blijft recursief doorgaan tot de rij gesorteerd is. We kunnen de complexiteit op de volgende manier berekenen:

- In het beste geval:

het eerste element verdeeld de array in precies de helft

$\Updownarrow$

$k = n/2ennn - k = n/2$

$$
\begin{aligned}
T(n) &= 2T(n/2) + n \qquad \text{met } T(n) \text{ het totale werk} && \text{(4.4a)}\\
&\Updownarrow\ T(n/2) = 2T(n/4) + n/2 && \text{(4.4b)}\\
&= 2(2T(n/4) + n/2) + n && \text{(4.4c)}\\
&\Updownarrow && \text{(4.4d)}\\
&= 2^2T(n/4) + 2n && \text{(4.4e)}\\
&\Updownarrow\ T(n/4) = 2T(n/8) + n/4 && \text{(4.4f)}\\
&= 2^2(2T(n/8) + n/4) + 2n && \text{(4.4g)}\\
&\Updownarrow && \text{(4.4h)}\\
&= 2^3T(n/8) + 3n && \text{(4.4i)}\\
&\Updownarrow \text{ Verdergaan tot en met k} && \text{(4.4j)}\\
&= 2^kT(n/2^k) + kn && \text{(4.4k)}\\
& && \text{(4.4l)}
\end{aligned}
$$

Dit gaat maar door tot $n = 2^k$, dus tot $k = log(n)$, wat volgt tot:

$$
\begin{aligned}
T(n) &= n\,T(1) + n\,log(n) && \text{(4.5)}\\
&\Updownarrow && \text{(4.6)}\\
&= 2n\,log(n) && \text{(4.7)}\\
& && \text{(4.8)}
\end{aligned}
$$

Dit laat zien dat de tijdscomplexiteit van quicksort gelijk is aan $\sim n\,log(n)$. Dit is dus zeer efficiënt.

- Worst case scenario: hier blijft het eerste element vanvoor staan waardoor de tijdscomplexiteit weer $\sim n^2/2$ is. De reden dat dit niet is voorgekomen in de tests is omdat de rijen willekeurig waren gegenereerd. In de realiteit zou men de rij eerst shufflen, waardoor de kans groter is dat we een best-case scenario krijgen.

Door de worst-case scenario uit te sluiten, krijgen we een zeer efficiënt algoritme met complexiteit $\sim n\,log(n)$. Dit is dan ook het doel waar men naartoe streeft als men een algoritme kiest of maakt. Zo is de snelste uitvoeringstijd altijd gegarandeerd.

4.3.2 Doubling-ratio test

Hier zijn de resultaten veel preciezer dan bij insertion sort, doordat de uitvoeringstijd vele malen kleiner is. Ook hier is de doubling ratio wiskundig af te leiden als we weten dat de gemiddelde tijdscomplexiteit $\sim 2n\,log(n)$ is:

$$\sim 2(2*n)\,log(2*n) = \sim 2*2n\,log(n) \tag{4.9a}$$
$$\Updownarrow \tag{4.9b}$$
$$\text{doubling-ratio} = 2 \tag{4.9c}$$

Deze doubling ratio kunnen we dan gebruiken om de uitvoeringstijd van grote arrays te berekenen:

Aantal elementen in de array	Uitvoeringstijd
131 072000	$88,850$
262 144000	$177,700$
524 288000	$355,400$
1 048 576000	$710,800$

Tabel 4: Voorspelde uitvoeringstijd in seconden voor een bepaald aantal elementen voor quicksort

4.4 Voorspelling van doubling ratio van algoritme met complexiteit $\sim n^5$

Wiskundig is dit aan te tonen met:

$$\sim (2*n)^5 = \sim 10*n^5 \tag{4.10a}$$
$$\Updownarrow \tag{4.10b}$$
$$\text{doubling ratio} = 10 \tag{4.10c}$$

Het gevolg hiervan is dat de tijd nodig om een object dat dubbel zo groot is te verwerken, tien keer zoveel tijd nodig heeft. Dit is zeer inefficiënt is.