Gegevensstructuren en Algoritmen: Practicum 1:

Hoe efficiënt zijn Selection Sort, Insertion Sort, en Quick Sort op random data?

Mathijs Hubrechtsen, r0664977

27 maart 2017

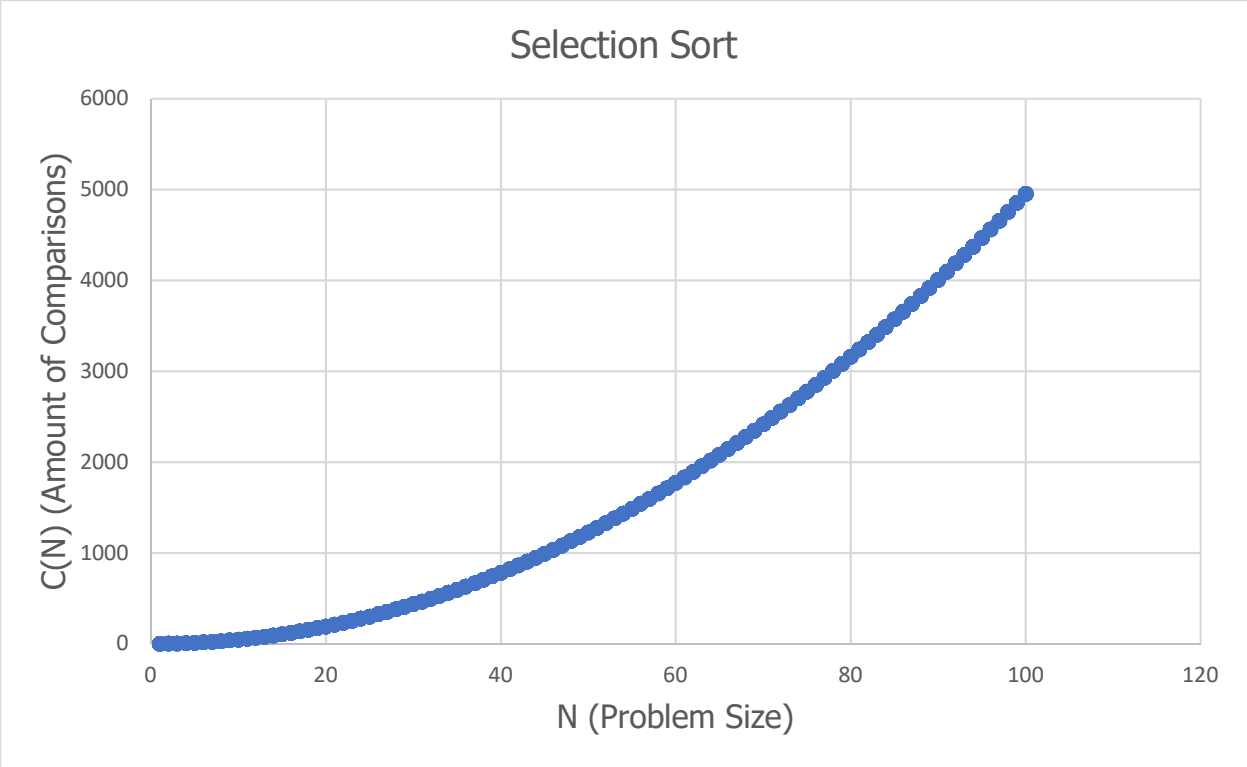
Inhoudsopgave:

1. Inleiding:

Dit verslag zal proberen te antwoorden op de vraag: *Hoe efficiënt zijn Selection Sort, Insertion Sort, en Quick Sort op random data?* Theoretisch zou Quick Sort het meest efficiënt zijn, gevolgd door Insertion Sort en tenslotte zou Selection Sort het minst efficiënt zijn. Mijn implementaties van deze algoritmes zijn redelijk standaard, met enkele optimalisaties. Ik heb de algoritmes toegepast op arrays met groottes 1, 2, …, 100 gevuld met random integers tussen 0 en 99. Voor elke individuele problem size N (of n), heb ik 15 testen genomen. Hiernaast heb ik ook voor elk algoritme doubling tests uitgevoerd voor problem sizes 250, 500, …, 8000 met opnieuw 15 testen voor elke individuele problem size. In het verslag zal ik ook vaak spreken over log en lg functies, hiermee bedoel ik $\log_2$. Ook zal ik af en toe verwijzingen maken naar mijn implementatie van een algoritme, voor de code van die implantatie verwijs ik door naar de java files waar we de effectieve algoritmes moesten schrijven.

2. Bespreking data:

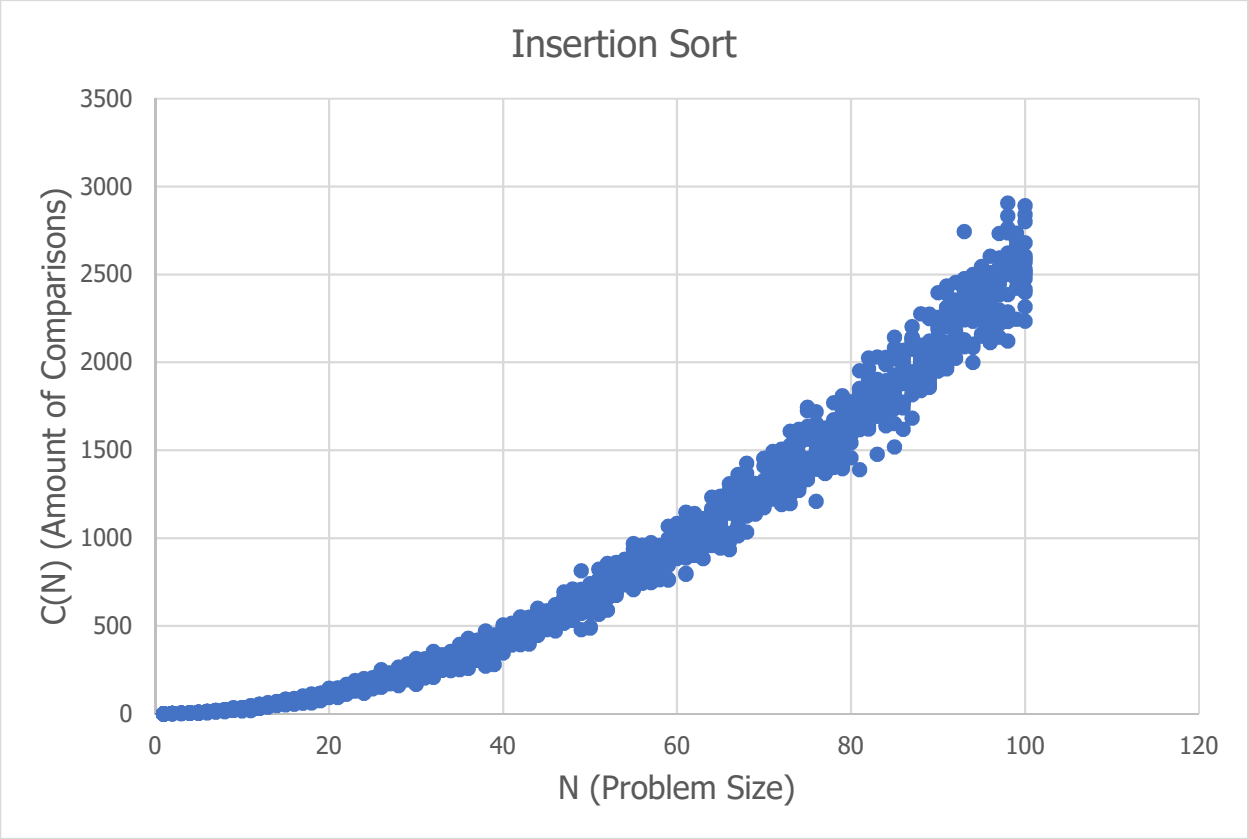
a. Selection Sort:



Het eerste dat opvalt bij het resultaat van Selection Sort is dat de testen van eenzelfde problem size allemaal exact evenveel vergelijkingen gebruiken. Dit is omdat Selection Sort altijd evenveel vergelijkingen doet op eenzelfde problem size. Selection Sort overloopt namelijk elke array helemaal van start tot einde, dan van start + 1 tot einde, enz. Dit is steeds ongeacht de data dat in de array staat. Hierdoor kunnen we dus logisch besluiten dat het aantal vergelijkingen voor eenzelfde problem size met het Selection Sort algoritme wel degelijk altijd gelijk zullen zijn. De spreiding is van belang voor de efficiëntie van het algoritme want als de spreiding groot is, zal de efficiëntie niet helemaal voorspelbaar zijn. Het algoritme zal dus minder betrouwbaar zijn. Maar bij Selection Sort is er geen spreiding, het is dus een zeer betrouwbaar algoritme.

We zien ook dat de stijging van de grafiek erg lijkt op de stijging die terug te vinden is in de kwadratische functie (vanaf 0). Deze hypothese wordt ook nog ondersteund door het feit dat Selection Sort een $\sim n^2/2$ functie is, het verschilt met slechts een constante in tilde notatie van de kwadratische functie. Hun verloop is dus zeker vergelijkbaar. De stijging van de grafiek is van belang voor de efficiëntie van het algoritme omdat hoe trager een algoritme stijgt, hoe efficiënter het is. Dit is omdat een traag verloop een lager aantal vergelijkingen nodig zal hebben op grotere problem sizes.

b. Insertion Sort:



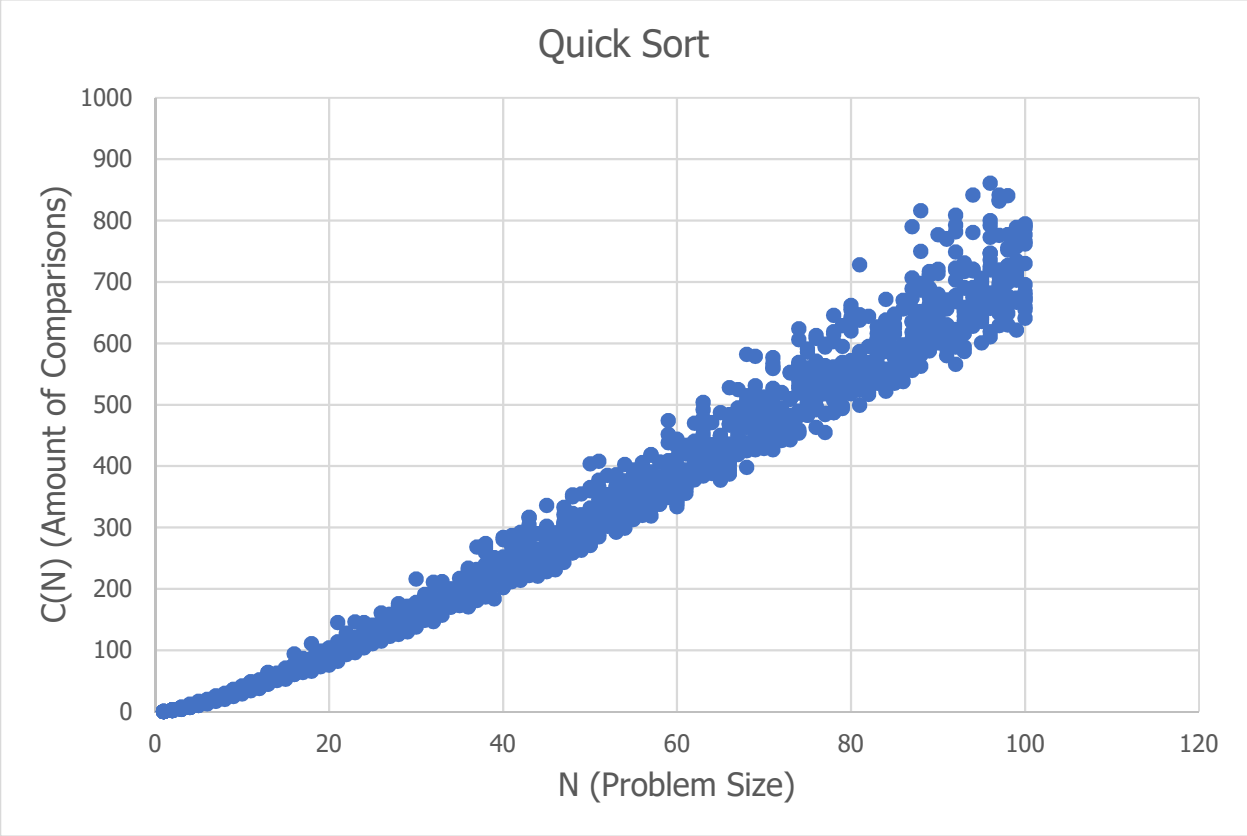
In tegenstelling tot Selection Sort, is het duidelijk dat het aantal vergelijkingen van Insertion Sort wel afhankelijk is van de data in de array, er is namelijk spreiding op het aantal vergelijkingen met dezelfde problem size. Een voorbeeld van zo'n verschil kunnen we vaststellen bij de arrays **A1 = [1, 2, 3, 4**] en **A2 = [4, 3, 2, 1]**. Als we Insertion Sort over **A1** laten lopen, dan zal Insertion Sort 6 vergelijkingen uitvoeren (Voor **A1[1]** => 1, **A1[2]** => 2, **A1[3]** = > 3; 1+2+3 = 6). Insertion Sort toegepast op **A2** levert slechts 3 vergelijkingen (Voor **A2[1]** => 1, **A2[2]** => 1, **A2[3]** = > 1; 1+1+1 = 3). Insertion Sort op **A2** levert dus de helft van het aantal vergelijkingen op **A1**, een duidelijk verschil.

We kunnen ook vaststellen dat Insertion Sort veel minder vergelijkingen gebruikt dan Selection Sort. Bijvoorbeeld bij Selection Sort worden er op een array met grootte 80 ongeveer 3000 vergelijkingen gebruikt, bij Insertion Sort zijn dit er slechts ongeveer 1750, bijna de helft. Dit is wat we eigenlijk ook verwachten want Selection Sort is een $\sim n^2/2$ algoritme terwijl Insertion Sort een $\sim n^2/4$ Algoritme is. Theoretisch zou Insertion Sort dus de helft van het aantal vergelijkingen van Selection Sort gebruiken. ((1/2) / 2 = 1/4) In de praktijk is dit niet exact zo omdat we natuurlijk veel lagere orde termen laten vallen in tilde notatie.

De tilde notaties die net zijn aangehaald leiden natuurlijk ook tot een andere interessante opvatting, namelijk dat het verloop van Selection Sort en Insertion Sort ook sterk op elkaar lijken. Ze verschillen allebei met slechts een constante van de kwadratische functie. Dit verband is ook

vast te stellen op de grafieken, beide lijken namelijk sterk op die van de kwadratische functie (vanaf 0). Hun verloop is dus vergelijkbaar met een kwadratische functie (vanaf 0).

c. Quick Sort:



We zien onmiddellijk dat zoals bij Insertion Sort, bij Quick Sort het aantal vergelijkingen afhankelijk is van de data in de array. De reden hiervoor ligt in de Partition methode, meer exact, het "pivot" element van de methode. In mijn implementatie is het pivot element steeds het eerste element van de array. Het pivot element is het element dat gebruikt wordt om de array op te delen in twee delen: 1 deel bevat alle elementen kleiner of gelijk dan de pivot, het ander alle element groter of gelijk aan de pivot.
/ In mijn Quick Sort implementatie kan een element dat gelijk is aan de pivot in beide delen terechtkomen, in andere Quick Sort implementaties is dat mogelijk niet zo.*/*
Het is duidelijk dat de pivot een cruciale rol speelt in het aantal vergelijkingen van een array. Bij voorbeeld neem **A1 = [1, 2, 3, 4, 5]** en **A2 = [3, 1, 2, 4, 5]**. **A1** levert hier 10 vergelijkingen op, **A2** maar 6. Het aantal vergelijkingen hangt dus wel degelijk af van de data.
/ **A1** neemt als pivot1 = 1, doet dan 4 vergelijkingen en vormt slechts 1 deel (alles groter dan 1). Hierna neemt **A1** als pivot2 = 2, doet dan 3 vergelijkingen en vormt dan opnieuw slechts 1 deel. Het herhaalt dit proces nog twee keer, en verkrijgt dan 2 + 1 = 3 extra vergelijkingen. C(**A1**) is dus 4 + 3 + 3 = 10.*
***A2** neemt als pivot1 = 3, doet 4 vergelijkingen en vormt dan twee gelijke delen. Deze 2 delen doen elk 1 vergelijking. C(**A2**) is dus 4 + 1 + 1 = 6. */*

Ook opvallend is dat de spreiding bij Quick Sort groter is (of toch zeker zo lijkt) dan bij Insertion Sort. Dit is mogelijk omdat de keuze van pivot misschien veel meer leidt tot grote spreiding in resultaten dan Insertion Sort. Hier zien we duidelijk het effect van spreiding op de efficiënte van een algoritme. Quick Sort's spreiding is vrij hoog, waardoor, zeker bij kleinere problem sizes, het verschil tussen het best-case en worst-case scenario vrij groot zal zijn. De efficiëntie is dus niet zo betrouwbaar als die van Selection Sort en Insertion Sort. Maar zelfs als Quick Sort zijn worst-case scenario behaalt, is het niet minder efficiënt als Selection Sort. Dit kunnen we niet afleiden uit de grafiek, maar we weten wel uit de theorie dat Quick Sort's worst-case gelijk is aan $\sim n^2/2$, dit is eigenlijk gewoon Selection Sort.

Nog iets dat opvalt is dat de stijging van de grafiek van het Quick Sort algoritme heel verschillend is van Selection Sort en Insertion Sort die op zich niet zo hard verschillen met elkaar. Een verklaring hiervoor ligt in het feit dat Selection Sort en Insertion Sort respectievelijk $\sim n^2/2$ en $\sim n^2/4$ algoritmes zijn. Ze verschillen dus slechts met een constante in hun tilde notatie. Quick Sort daarentegen is een ~1.39nlg(n) algoritme, wat veel meer verschilt met de vorige twee algoritmes dan enkel een constante term.

Als we het verloop verder bekijken valt het ook op dat het niet bepaald gemakkelijk is om op het zicht een vergelijkbare functie te geven voor Quick Sort. Quick Sort lijkt wel lineair, maar is het duidelijk niet. Als we bijvoorbeeld de richtingscoëfficiënt berekenen tussen (20, 100) en (40, 200):

$$M_1 = \frac{(200 - 100)}{(40 - 20)} = 5.$$

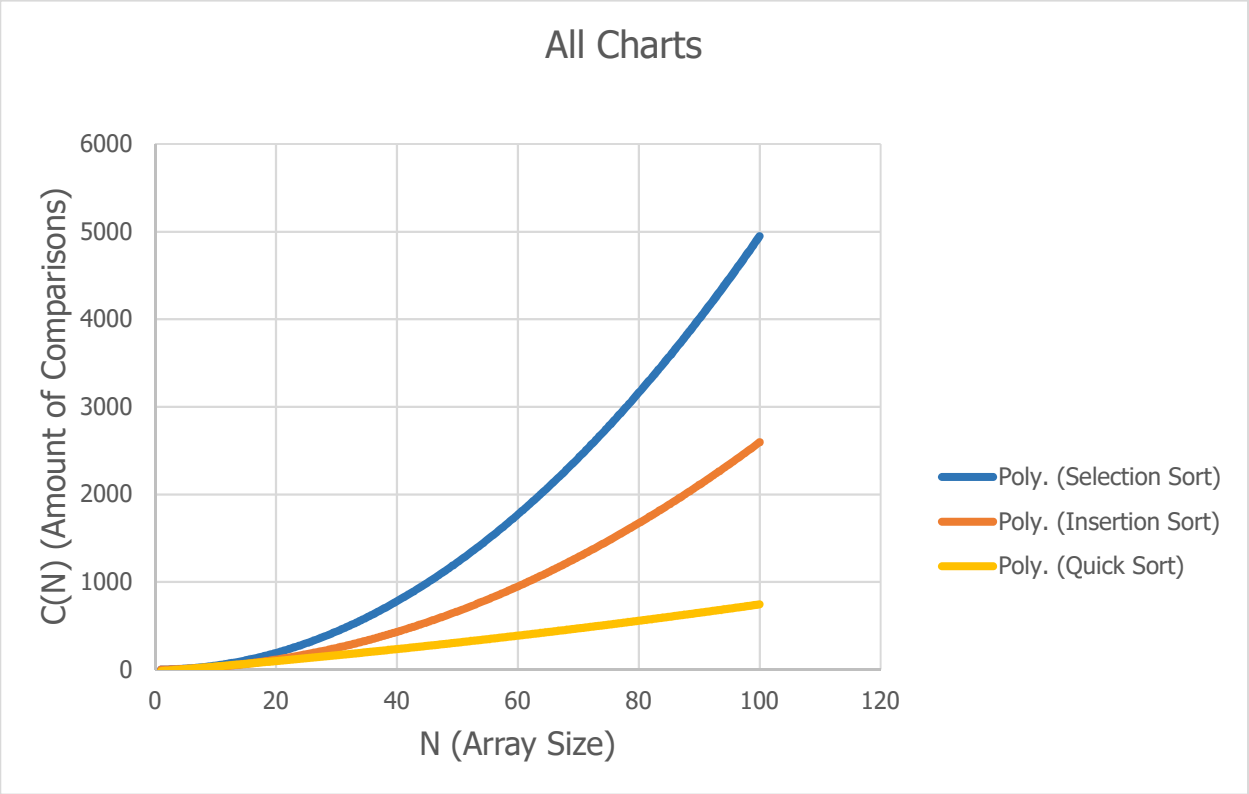
Is het duidelijk dat deze richtingscoëfficiënt niet diezelfde is als diegene tussen (80, 550) en (100, 700):

$$M_2 = \frac{(700 - 550)}{(100 - 80)} = 7{,}5.$$

/ De gekozen punten zijn een ruwe schatting van het gemiddelde van het aantal vergelijkingen dat Quick Sort doet op een array met een bepaalde grootte. */*

Maar Quick Sort is ook duidelijk niet kwadratisch, het verloop stijgt daar veel te traag voor.
De tilde notatie van Quick Sort is opnieuw de verklaring hier, ~1,39nlog(n) wil zeggen dat Quick Sort vergelijkbaar is met een nlog(n) functie, een functie die ligt tussen n en n^2.

d. Besluit:



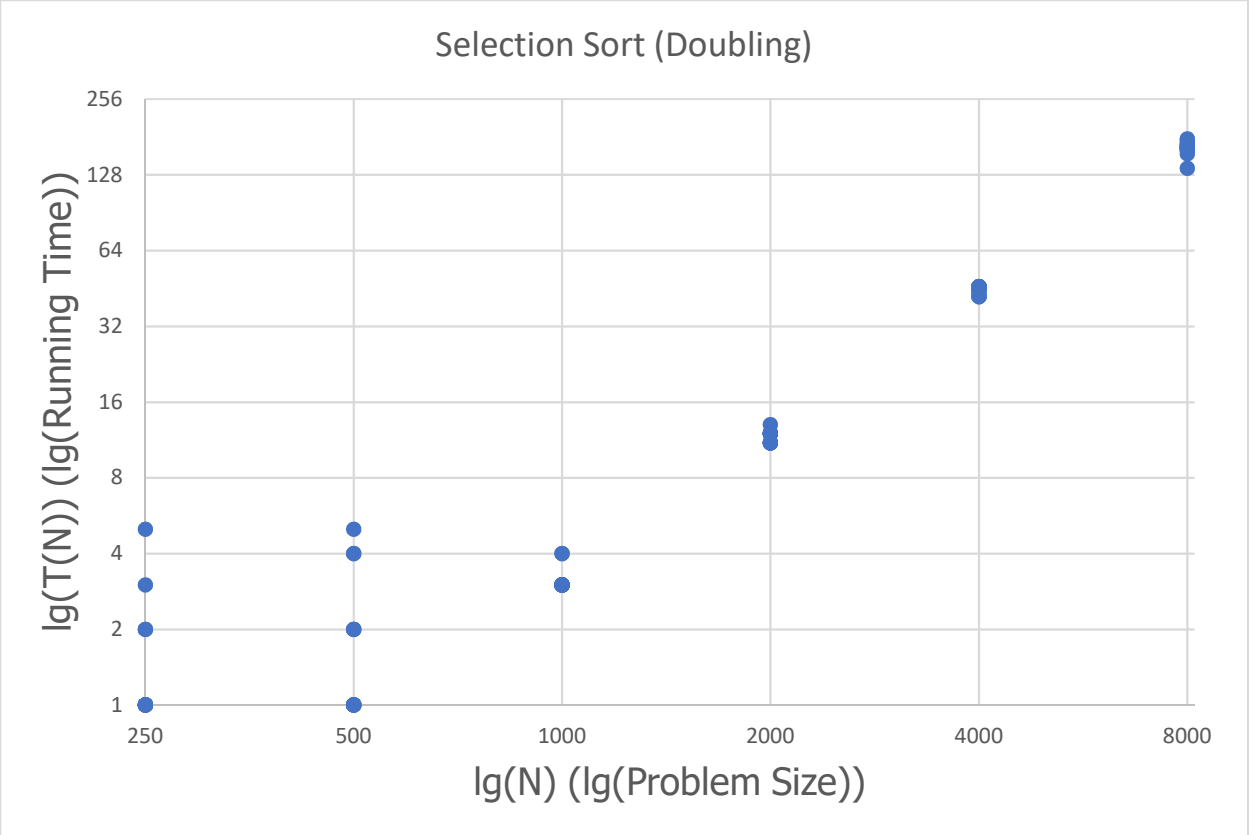
Als we al de data samenvatten in 1 grafiek, en dan voor elk algoritme enkel de trendlijn plotten kunnen we de 3 algoritmes nog duidelijker vergelijken. Zoals besproken, zien we dat Selection Sort en Insertion Sort een soortgelijk verloop hebben, met als verschil dat Insertion Sort trager stijgt. Het is hier ook duidelijk te zien dat Insertion Sort nooit trager zal zijn dan Selection Sort, de grafieken kruisen namelijk niet wanneer de data groter blijft worden. Dit volgt ook uit de theorie want Insertion Sort's slechtste geval is $\sim n^2/2$, dit is gelijk aan de tilde notatie van Selection Sort, het is dus nooit trager. (Als we de lagere orde termen buiten beschouwing laten)

Op deze plot zien we ook duidelijk dat Quick Sort wel degelijk het meeste efficiënte algoritme is, het verloop is het traagste en het is nooit trager dan de vorige twee algoritmes. Het eerste deel van deze bewering hebben we al bewezen, het tweede volgt net zoals bij Insertion Sort uit de theorie dat ook Quick Sort's slechtste geval gelijk is aan $\sim n^2/2$, wat opnieuw gelijk is aan de tilde notatie van Selection Sort. Dit wil natuurlijk niet zeggen dat Quick Sort ook de meest betrouwbare resultaten levert, maar het is wel degelijk het efficiëntste van de drie.

Als we nu teruggaan naar onze centrale vraag: *Hoe efficiënt zijn Selection Sort, Insertion Sort, en Quick Sort op random data?* Kunnen we besluiten dat Insertion Sort efficiënter is dan Selection Sort, en dat Quick Sort efficiënter is dan beide Selection en Insertion Sort.

3. Doubling ratio:

a. Selection Sort:



/ T(N) is in milliseconden. */*

Als we een doubling test uitvoeren op Selection Sort en dan plotten in een log-log plot bekomen we de grafiek hierboven. Het idee van een doubling test is dat we voor elke test een problem size nemen die dubbel zo groot als de vorige is. Hier starten we bij 250. We verwachten dat wanneer we dit doen, we een lineair verband zien. De reden hiervoor is dat Selection Sort een $\sim n^2/2$ algoritme is, dit is van de vorm $T(N) = a*N^b$. Als een verband van die vorm is, dan is zijn grafiek in een log-log plot een rechte. Omdat we zien dat de grafiek inderdaad een rechte is (des te groter de problem size wordt) kunnen we stellen dat de bewering dat Selection Sort's data op de vergelijking $T(N) = a*N^b$, ligt correct is. Deze rechte wordt duidelijker des te groter we de problem size nemen.

Doubling Ratio Experiment Selection Sort Results:

N (Problem Size)	T(N) (Running Time)	Doubling Ratio
250	1.53333333	-
500	1.73333333	1.13043478
1000	3.13333333	4.00400802
2000	11.6666667	3.72340426
4000	44.6666667	3.82857143
8000	163.8	3.66716418

/ T(N) is de gemiddelde uitvoeringstijd in milliseconden op eenzelfde probleemgrootte. */*

Als we nu de resultaten bekijken in tabel vorm (zie boven), kunnen we ook het doubling ratio berekenen. Het doubling ratio is gelijk aan $\frac{T(2*N)}{T(N)}$, de uitvoeringstijd van een bepaalde problem size gedeeld door de uitvoeringstijd van de vorige. Normaal gezien zou dit ratio een limiet naderen des te groter de problem size wordt. Theoretisch gezien zou de limiet 4 moeten naderen, dit zal ik aantonen bij Insertion Sort. Bij de bovenstaande tabel lijkt de limiet eigenlijk weg te gaan van 4, naar 3,5. Maar normaal zou deze limiet bij grotere probleem groottes wel deze limiet naderen. De reden waarom het steeds meer en meer een limiet nadert is omdat bij grotere problem sizes, de lagere orde termen van de tijdscomplexiteit buiten beschouwing kunnen worden gelaten. Ook zal het feit dat er andere processen op de computer uitgevoerd worden tijdens de uitvoering van het algoritme van minder belang worden.

Doubling Ratio Experiment Selection Sort Voorspellingen:

N (Problem Size)	T(N) (Predict.)	T(N) (Actual)	DR (Predict.)	DR (Actual)
16000	655.2	617.933333	4	3.77248677
32000	2620.8	2437.4	4	3.94443845
64000	10483.2	9592.4	4	3.93550505

/ T(N) is in milliseconden. */*

Nu we een vermoeden hebben van het doubling ratio kunnen we ook de uitvoeringstijd voorspellen voor grotere problem sizes. In de bovenstaande tabel hebben we dit gedaan voor problem sizes die iets groter zijn dan onze originele testen. Als we daarna op deze problem size ook effectief gaan testen, zien we dat onze vermoedens inderdaad dicht bij hun effectieve resultaat liggen. Het doubling ratio wordt ook steeds nauwkeuriger, het blijft 4 naderen.

Nu we vrij zeker zijn dat de doubling ratio correct is, kunnen we ook een voorspelling doen over een zeer grote problem size. Als voorbeeld zullen we een grootte nemen die 8 keer groter is dan onze grootste meting: $T(N_{max}) * DR^8 = 163.8 * (4^8) = 10734796.8$ milliseconden wat ongeveer gelijk is aan 3 minuten.

b. Insertion Sort:

/ T(N) is in milliseconden. */*

Net als bij Selection Sort, verwachten we dat bij het plot van Insertion Sort ook een lineair verband zien. Insertion is namelijk $\sim n^2/4$, wat ook van de vorm $T(N) = a*N^b$ is. En net zoals bij Selection Sort, is deze hypothese correct. Het plot volgt inderdaad een lineair verband des te groter de problem size wordt.

Nu is ook een goed moment om andere factoren van een running test te bespreken. Ten eerste meet mijn testmethode de volledige uitvoeringstijd, vanaf de start van de oproep tot het einde. Dit wil zeggen dat de overhead van de functie oproep en andere factoren ook in T(N) zitten, dit is belangrijk omdat het wil zeggen dat zeker bij lagere problem sizes, T(N) nog niet veel zegt over de meest tijdrovende operatie. Hierdoor zijn zeker bij lagere problem sizes T(N) en de doubling ratio weinig indicatief van de effectieve efficiëntie van het algoritme. Een andere factor is het feit dat de processor tijdens de uitvoer van het algoritme ook bezig is met andere programma's. Dit is de reden waarom er veel spreiding te verwachten is, zeker bij lagere problem sizes. Dit in tegenstelling tot de traditionele testen waar geen spreiding was bij bijvoorbeeld Selection Sort.

Doubling Ratio Experiment Insertion Sort:

N (Problem Size)	T(N) (Running Time)	Doubling Ratio
250	1.66666667	-
500	2.26666667	1.36
1000	3.26666667	1.44117647
2000	12.5333333	3.83673469
4000	45.7333333	3.64893617
8000	140.533333	3.0728863

/ T(N) is de gemiddelde uitvoeringstijd in milliseconden op eenzelfde probleemgrootte. */*

Ook zoals bij Selection Sort zullen we nu de data nader bekijken in tabel vorm en het doubling ratio berekenen. Dit doubling ratio nadert theoretisch, net zoals bij Selection Sort, ook 4. Dit is omdat Selection Sort en Insertion Sort, allebei stijgen volgens de grafiek van n^2 (vanaf 0). Voor een problem size die dubbel zo groot is als de vorige zal n^2 een resultaat leveren dat 4 keer zo groot is: $T(2) = 2^2 = 4$. Ook hier zien we dat de doubling ratio nog niet helemaal gelijk is aan 4 en meer lijkt 3 te naderen, we zullen opnieuw voorspellingen maken over iets groter problem sizes om na te kijken of de hypothese dat de limiet 4 nadert toch correct is.

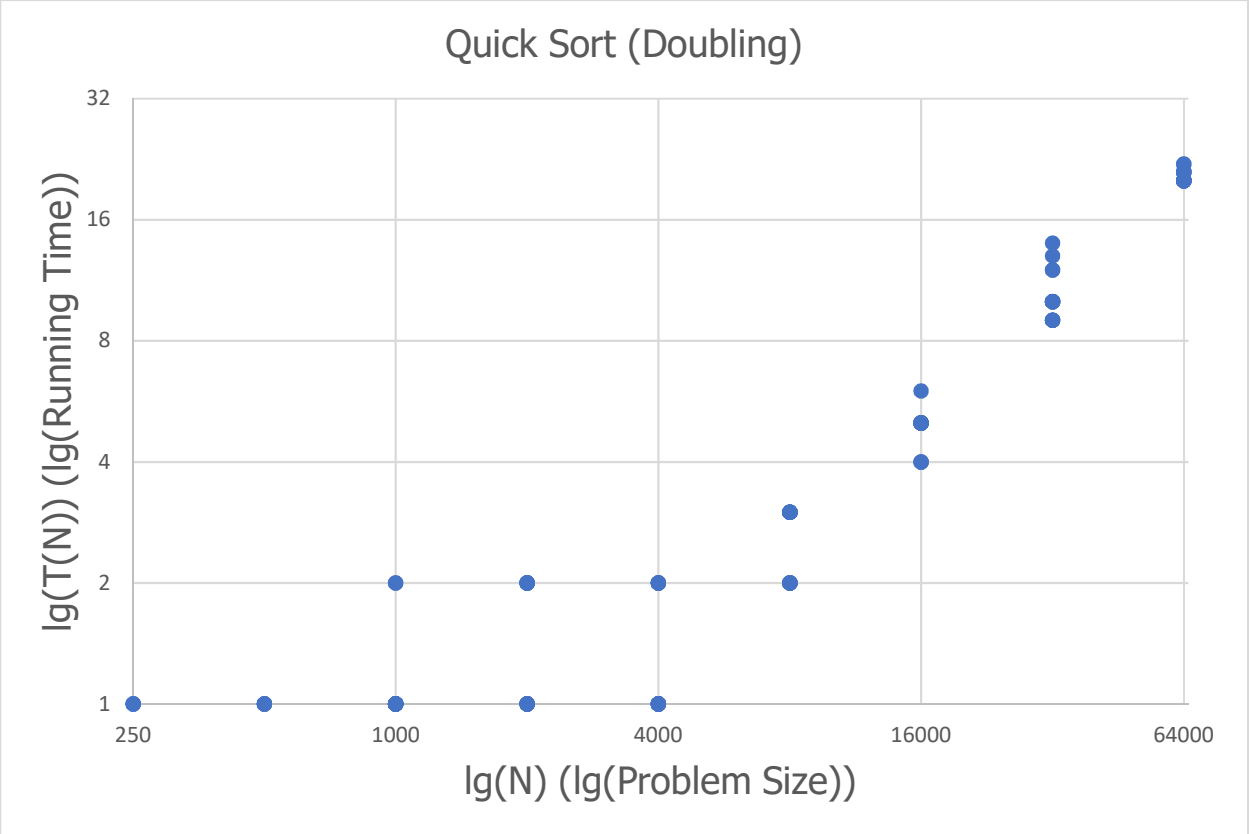
Doubling Ratio Experiment Selection Sort Voorspellingen:

N (Problem Size)	T(N) (Predict.)	T(N) (Actual)	DR (Predict.)	DR (Actual)
16000	562.133333	643.866667	4	4.58159393
32000	2248.53333	2109.73333	4	3.27666183
64000	8994.13333	8654.06667	4	4.10197181

/ T(N) is in milliseconden. */*

Zoals verwacht tonen onze voorspellingen voor Insertion Sort ook aan dat de limiet 4 nadert. We zullen dus ook 4 nemen voor de voorspelling met een grotere problem size. Als we voor Insertion Sort een problem size nemen die 8 keer zo groot is dan onze grootste meting bekomen we: $T(N_{max}) * DR^8 = 140.533333 * (4^8) = 9209992.51149$ milliseconden wat ongeveer gelijk is aan 2 minuten en 36 seconden.

c. Quick Sort:



/ T(N) is in milliseconden. */*

Quick Sort's doubling test levert de bovenstaande grafiek op. Er is ook een lineair verband te zien in het log-log plot, wat leidt tot de bewering dat Quick Sort stijgt volgens een functie van de vorm $T(N) = a*N^b$. Wat wel opvallend is, is dat het lijkt alsof Quick Sort sneller in een lineair verband terecht komt. Dit is vermoedelijk toevallig.

Doubling Ratio Experiment Quick Sort:

N (Problem Size)	T(N) (Running Time)	Doubling Ratio
250	0.26666667	-
500	1.13333333	4.25
1000	1.46666667	1.29411765
2000	1.33333333	0.90909091
4000	1.33333333	1
8000	2.6	1.95

/ T(N) is de gemiddelde uitvoeringstijd in milliseconden op eenzelfde probleemgrootte. */*

We zien dat in tegenstelling tot Selection Sort en Insertion Sort, het doubling ratio helemaal niet 4 nadert. Dit is vrij logisch want Quick Sort is een ~1,39nlog(n) algoritme wat meer verschilt met een $\sim n^2$ dan slechts een constante. We verwachten dus dat het doubling ratio 2 nadert want $T(2) = 2*\log_2(2) = 2$. Als N 2 keer zo groot wordt, zou normaal T(N) ook ongeveer 2 keer zo groot worden. Als we kijken naar de tabel lijkt het dat T(N) deze theoretische limiet inderdaad zal naderen bij grotere problem sizes. We zullen nog steeds bekijken of deze voorspelling correct is.

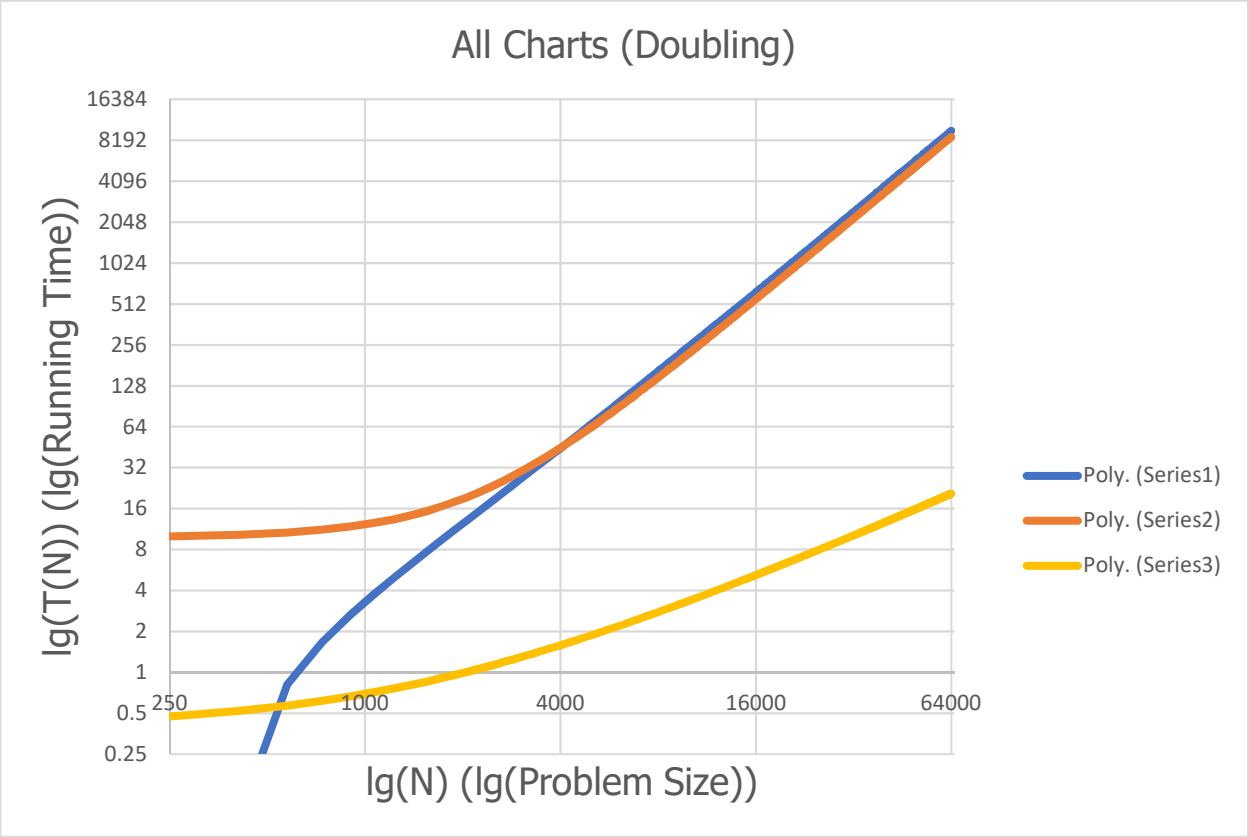
Doubling Ratio Experiment Quick Sort Voorspellingen:

N (Problem Size)	T(N) (Predict.)	T(N) (Actual)	DR (Predict.)	DR (Actual)
16000	5.2	4.86666667	2	1.87179487
32000	10.4	10.4666667	2	2.15068493
64000	20.8	20.6	2	1.96815287

/ T(N) is in milliseconden. */*

Als we voorspellingen proberen te maken met als doubling ratio 2 blijkt deze keuze vrij accuraat. De limiet nadert inderdaad duidelijk 2. Omdat onze voorspellingen vrij dicht bij de echte metingen liggen, zullen we dus ook 2 als doubling ratio nemen voor de voorspelling met een grotere problem size. Voor een problem size nemen die 8 keer zo groot is dan onze grootste meting bekomen we: $T(N_{max}) * DR^8 = 2{,}6 * (4^8) = 170393.6$ milliseconden wat ongeveer gelijk is aan ongeveer 3 *(!)* seconden. Dit is veel sneller dan beide Selection Sort en Insertion Sort en is een duidelijk bewijs waarom dat Quick Sort het efficiëntste algoritme van de drie is.

d. Besluit:



Als we al de doubling tests allemaal plotten in hetzelfde plot, zien we heel duidelijk dat de doubling ratio's van Selection Sort en Insertion Sort gelijk zijn. Hun trendlijnen snijden namelijk nooit (na 4000) en hun richtingscoëfficiënt is gelijk. In contrast zien we dat Quick Sort opnieuw hard verschilt van de vorige twee algoritmes. Ten eerste stijgt de grafiek veel trager, dit is omdat het doubling ratio van Quick Sort 2 is en die van de anderen 4. De reden waarom dat voor 4000 de algoritmes niet helemaal lineair zijn hebben we al behandeld. De overhead van de oproep van het algoritme en het feit dat de processor ook met andere toepassingen bezig is, spelen hier een belangrijke rol.

Met de kennis dat we hebben verworven in deze testen, kunnen we ook voorspellingen doen over de doubling ratio's van algoritmes met een tilde-notatie verschillend van die van Selection Sort, Insertion Sort en Quick Sort. Bijvoorbeeld, we weten nu dat we bij een $\sim n^5$ algoritme, een doubling ratio van 32 verwachten. Want $(2)^5 = 32$, oftewel voor N' = 2N verwachten we dat T(N') 32 keer zo groot is.

4. Besluit:

Nu we veel data verzameld hebben en de algoritmes uitvoerig hebben besproken kunnen we teruggaan naar onze originele vraag: *Hoe efficiënt zijn Selection Sort, Insertion Sort, en Quick Sort op random data?* Het blijkt duidelijk, uit de traditionele metingen en de doubling tests, dat Quick Sort het efficiëntste is van de drie. Het verloop van Quick Sort is namelijk het traagste van de drie, dit zagen we bijvoorbeeld bij Quick Sort's doubling ratio dat bijna de helft is die van de anderen. Tussen Selection en Insertion Sort, is Insertion Sort het efficiëntste, dit hebben we aangetoond bij de traditionele metingen.

Wel is het zo dat deze volgorde omkeert bij de spreiding van data. Quick Sort is het slechtste door zijn Partition methode, gevolgd door Insertion Sort. Selection Sort heeft optimale spreiding: geen spreiding. Een manier om de spreiding bij Quick Sort te verbeteren is door een random shuffle te doen in het begin van de Partition methode, maar dit was niet gevraagd. Maar zoals eerder gezegd, zelfs al doet Quick Sort's worst-case scenario zich voor, zal het dan nog steeds ten minste even efficiënt zijn als Selection Sort. Dus zelfs al is Quick Sort het minst betrouwbaar, het is nog steeds zeker het efficiëntste.

Dit Besluit: Quick Sort > Insertion Sort > Selection Sort (op vlak van efficiëntie), is exact wat we hadden voorspeld in het begin van dit verslag, de testen zijn dus consistent met de theorie.