Insertion - Selection en QuickSort

GNA - practicum

Inleiding:

In dit practicum gaan we een aantal sorteer algoritmes onder de loep nemen om zo te bepalen welke het snelste of gemakkelijkste zouden zijn. Dit doen we aan de hand van een aantal tests die we uitvoeren. In deze tests kijken we naar het aantal keer er elementen uit de rij vergeleken worden en het aantal keer er elementen verwisseld worden in de rij. Als laatste gaan we nog kijken naar een "Doubling Ratio" experiment voor Insertion Sort en QuickSort, hierin verdubbelen we steeds de lengte van de gegeven rij en gaan we kijken hoe lang het duurt tot de rij volledig gesorteerd is. Ook zullen we bij elk onderdeel de code even kort toelichten. Maar voor we de resultaten van de experimenten gaan bespreken zullen we kort bekijken hoe deze algoritmen juist werken.

Hoe werken de algoritmes?:

We gaan kijken naar 3 algoritmes namelijk InsertionSort, SelectionSort en QuickSort. Hoe gaan deze nu te werk?

InsertionSort start in het begin van de rij en vergelijkt de eerste 2 elementen. Als het eerste element groter is dan het tweede, worden de twee elementen omgewisseld. Daarna worden de volgende twee elementen vergeleken. Stel nu dat de eerste 2 elementen groter zijn dan het derde, dan wordt eerst het tweede en het derde element van plaats gewisseld. Daarna wordt dan het eerste en het tweede (wat oorspronkelijk het derde element in de rij was) omgewisseld. Kort samengevat kijkt InsertionSort dus naar een element en schuift het zo ver naar voor tot het element ervoor kleiner is dan dat element. Als uiteindelijk het laatste element juist geplaatst wordt zijn we er zeker van dat de rij volledig en juist gesorteerd is.

SelectionSort loopt de hele rij zo veel keer af als er elementen in de rij zitten. Tijdens de rij afgelopen wordt, wordt het kleinste element onthouden. Eens het einde van de rij bereikt is wordt het onthouden kleinste element verwisselt met het element dat op de eerste plaats staat. Nadat er helemaal over de rij gegaan is en het kleinste element vooraan geplaatst werd blijft het algoritme van die plaats in de rij af. Het zal dus vanaf het volgende element in de rij kijken welk element het kleinste is. Dit wil dus zeggen; wanneer er voor de eerste keer over de rij gegaan wordt het kleinste element dus op de eerste plaats in de rij komt te staan.

QuickSort hangt eigenlijk hard af van geluk. Er word een random element in de rij gekozen waar alle elementen mee vergeleken worden, dit element noemen we de pivot. Elk element dat kleiner is wordt links van dit elementen geplaatst, de grotere rechts. Dit noemen we partitioning. Dan wordt met behulp van recursie het linkse en het rechtse deel apart verder gesorteerd. Beide zullen dus terug partitioning gebruiken om de kleine delen te sorteren. Wanneer er enkel nog rijen gesorteerd moeten worden van lengte 1 zijn we zeker dat alles gesorteerd is en kunnen we stoppen met sorteren.

Meest gunstige geval:

[1] Het meest gunstige geval kunnen we vergelijken met een rij die reeds gesorteerd is. Als we dan kunnen kijken naar het aantal keer er elementen uit de rij vergeleken moeten worden, zal het snelste algoritme datgene zijn dat het minste moet vergelijken. We bekijken hier een gesorteerde rij van 1000 elementen die door alle 3 de algoritmen zal gaan. Aangezien er geen elementen van plaats verwisseld moeten worden zullen we alleen naar het aantal keer er elementen vergeleken moeten worden kijken. Als we de test uitvoeren zien we deze resultaten:

```
InsertionSort (gunstige rij) vergelijkingen = 999

SelectionSort (gunstige rij) vergelijkingen = 499500

QuickSort (gunstige rij) vergelijkingen = 499500
```

We zien hier dat InsertionSort duidelijk het minste aantal vergelijkingen zal moeten uitvoeren. Wat ook opvalt is dat zowel SelectionSort als QuickSort evenveel vergelijkingen moet uitvoeren. Laten we dieper in gaan op InsertionSort. Hier zien we, dat wanneer we een rij van 1000 elementen sorteren die al gesorteerd is, er maar 999 keer elementen vergeleken moeten worden. Als we dit gaan berekenen voor een rij van n getallen zullen we zien dat er n - 1 keer vergeleken moet worden. We zullen dus eerst element 1 en 2 vergelijken, dan 2 en 3 en zo verder. Zo komen we er dus op uit dat wanneer we n elementen hebben er slechts n - 1 keer vergeleken moet worden. Dit komt neer op ~n.

Bij SelectionSort zien we dat er 499500 vergelijkingen gebeurt zijn voor een rij van 1000 elementen te sorteren. We kunnen dit gaan berekenen en dan zien we dat we dezelfde resultaten bekomen. Hoe komen we aan die formule? Wanneer SelectionSort een rij sorteert zal deze n - 1 keer over de hele rij gaan. Per iteratie zullen er eerst n - 1 vergelijkingen plaatsvinden. Bij de volgende iteratie maar n - 2 en zo verder tot we aan iteratie n - 1 komen die slechts 1 vergelijking zal uitvoeren. Dit komt dus neer op (n - 1) + (n - 2) + … + 2 + 1 = n(n - 1) / 2 = ~(n^2). Als we dus 1000 elementen invullen in deze rij, bekomen we inderdaad 499500 vergelijkingen.

We zullen nu gaan kijken hoe we tot de resultaten van QuickSort komen. Wanneer QuickSort in het beste geval uitgevoerd wordt, zal de rij steeds perfect in de helft verdeeld worden. Als dit gebeurt zal er steeds maar de helft van het aantal vergelijkingen uitgevoerd moeten worden. Dit is dus: 2T(n/2) + partitioning(n). We weten dat de partitioning steeds n vergelijkingen uitvoert dus wordt onze formule: 2T(n/2) +n. Als dit verder uitgerekend wordt zullen er dus ~nlog2(n) vergelijkingen uitgevoerd worden.

Meest ongunstige geval:

Voor dit experiment gaan we kijken naar het meest ongunstige geval. We nemen een rij waar de meeste elementen verplaatst moeten worden en er het meeste vergeleken moet worden. Hieruit kunnen we ook afleiden welk algoritme het beste is wanneer er een onrustige rij wordt meegegeven. (Voor elk volgende stukje tekst)

We zullen weer eerst beginnen bij InsertionSort. Wanneer we een ongunstige rij aan dit algoritme willen geven, zullen we ervoor moeten zorgen dat, wanneer we naar een element in de rij kijken, elk element ervoor groter is. Hierdoor zal er dus telkens gewisseld moeten worden en zal er ook telkens vergeleken moeten worden. Omdat we vooraan de rij beginnen (bij element twee) zullen er bij de eerste stap reeds 1 vergelijking en 1 verwisseling plaatsvinden. Het tweede element zal dan 2 keer verschoven en vergeleken moeten worden, het derde drie keer en zo verder tot we aan het einde van de rij komen. Het laatste element zal dan n - 1 keer vergeleken en verwisseld moeten worden. Als we dit allemaal optellen bekomen we: (n - 1) + (n - 2) + … + 2 + 1 = n(n - 1) / 2 = ~(n^2) / 2

[1] SedgeWick, R. (2011). Algorithms (Fourth ed.).

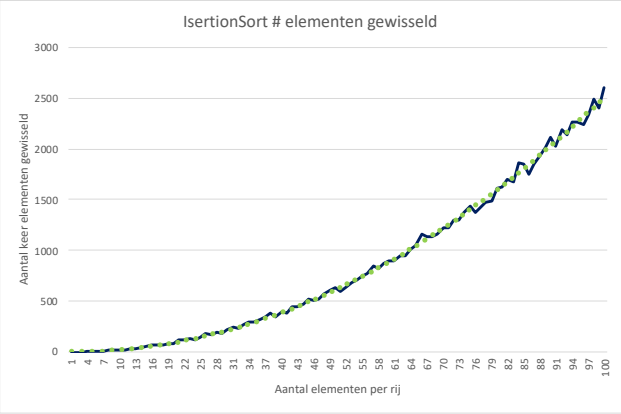
Laten we dan eens gaan kijken naar SelectionSort. Aangezien er n - 1 keer over de rij gegaan wordt zal er dus maximaal n - 1 elementen verwisseld worden. Want in het slechtste geval zal er elke keer dat er over de rij gegaan wordt, 1 keer een element verwisseld moeten worden. Omdat we altijd een element minder moeten gaan vergelijken, zal er in het slechtste geval dus ook elk element in de rij vergeleken moeten worden. Aangezien het dus elke iteratie een element minder is, zullen er eerst n - 1 vergelijkingen moeten uitvoeren, dan n - 2 vergelijkingen en zo verder. Dit zal dus neer komen op (n - 1) + (n - 2) + … + 2 + 1 = n(n - 1) / 2 = ~(n^2) / 2

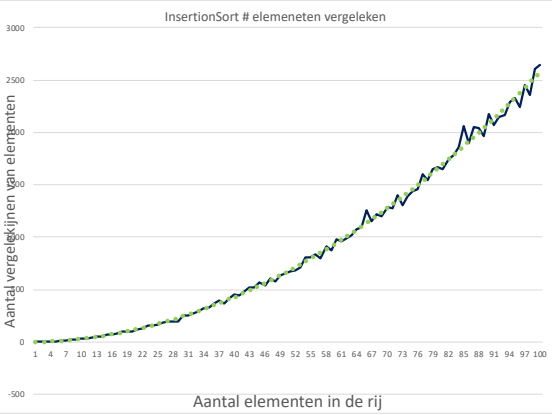
Voor QuickSort zal een rij zeer ongunstig zijn wanneer er de pivot het grootste getal is. Hierdoor zal de rij niet in twee verdeeld worden. Er zullen dus een eerste keer n - 1 vergelijkingen gebeuren, bij de tweede iteratie zullen er n - 2 vergelijkingen gebeuren en zo verder tot de rij gesorteerd is. In totaal zullen er dus (n - 1) + (n - 2) + … + 2 + 1 vergelijkingen gebeuren, wat neerkomt op n(n - 1) / 2 vergelijkingen of ~(n^2) / 2. We kunnen dit voorkomen door voordat we de rij gaan sorteren de rij eerst volledig random van plaats te laten veranderen. Hierdoor zal de kans klein zijn dat de rij zich in worst case zal bevinden.

Mijn resultaten:

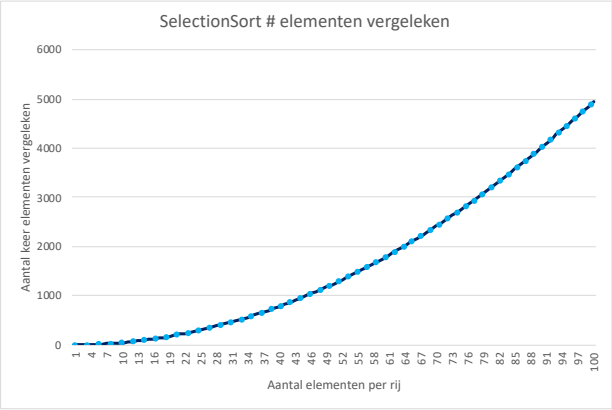
Omdat we ook willen weten wat de resultaten zijn als we een random array meegeven met het algoritme, hebben we ook dit getest. In deze test werden 100 arrays gesorteerd deze hadden een lengte van 1, 2, 3 tot 100. Met deze resultaten kunnen we een grafiek plotten en zo bepalen hoeveel keer gemiddeld er elementen gewisseld en vergeleken moeten worden. Om betere resultaten te krijgen hebben we de test 5 keer uitgevoerd en van die 5 testen het gemiddelde genomen.

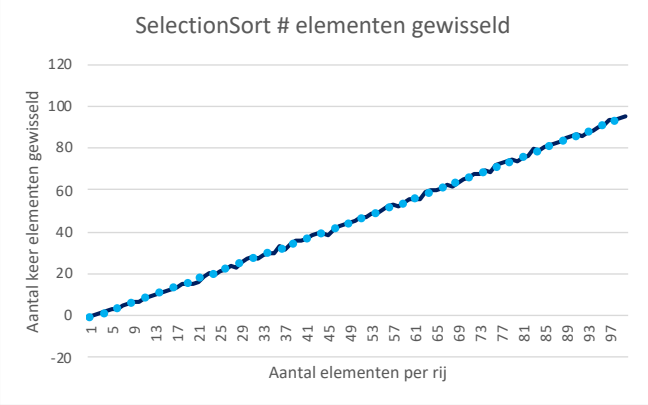
We zullen starten bij InsertionSort. Als we de gemiddelde resultaten op een grafiek plotten, krijgen we onderstaande grafieken. Nu we de resultaten hebben, kunnen we deze vergelijken met wat we in theorie zouden moeten bekomen. Om te berekenen hoeveel keer er vergeleken en gewisseld moet worden, zullen we het algoritme wat grondiger moeten bekijken. Zo hebben we in het meest gunstige geval al gezien dat er minstens n - 1 vergelijkingen moeten gebeuren. Aangezien de rijen die we hier testen random rijen zijn, zal de kans klein zijn dat er een rij reeds gesorteerd is. Om nu te kijken naar het gemiddeld aantal vergelijkingen zullen we terug naar het meest ongunstige geval moeten kijken. We weten dat er dan (~(n^2) / 2) vergelijkingen en verwisselingen gebeuren. Bij het gemiddelde bedenken we dat de helft van de getallen links van het element waar me mee vergelijken groter zijn waardoor er dus de helft minder vergelijkingen en verwisselingen gebeuren. We kunnen er dus vanuit gaan dat er dan ~(n^2) / 4 vergelijkingen en verwisselingen zouden gebeuren. Als we dat vergelijken met onze bekomen resultaten zien we dat als we bijvoorbeeld 100 in onze formule invullen we 100^2 / 4 = 2500 vergelijkingen en verwisselingen zouden moeten krijgen. Nu we dit vergelijken zien we dat de bekomen resultaten goed overeenkomen met de theorie.



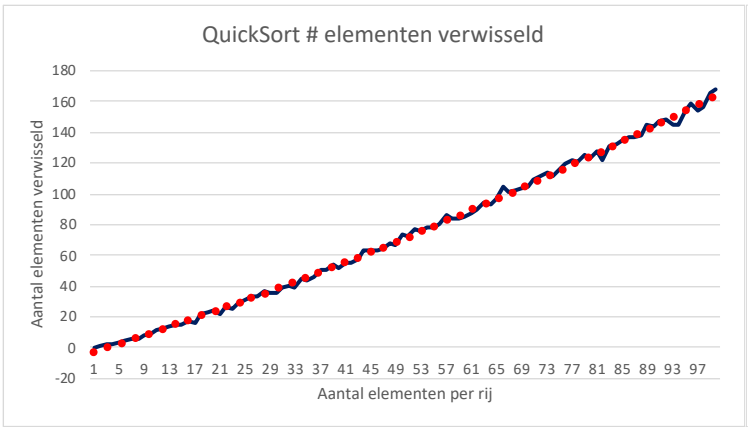


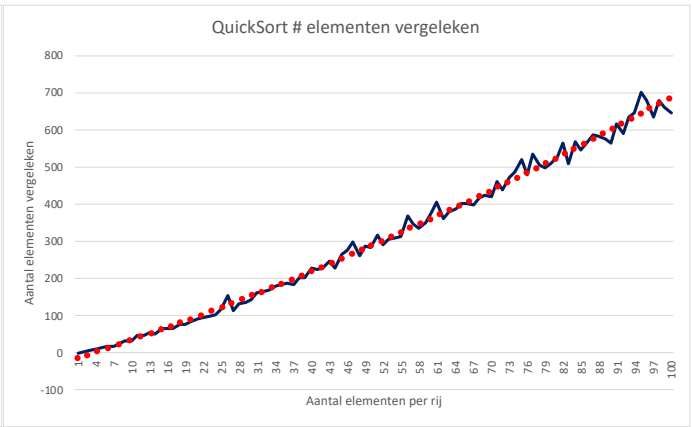
[2] we dezelfde testen uitvoeren bij SelectionSort bekomen we onderstaande resultaten. Hier heb ik weer de testen 5 keer uitgevoerd en het gemiddelde ervan genomen. Welke resultaten zouden we nu theoretisch moeten uitkomen? We weten uit vorige berekeningen van meest gunstige en minst gunstige rijen dat het aantal vergelijkingen hetzelfde blijft. Deze zullen dus voor eender welke rij, hoe random ze ook mag zijn, er evenveel vergelijkingen moeten gebeuren. Namelijk (n^2) / 2 - n / 2 of in grote orde ~(n^2) / 2. Als we nu de grootste rij invullen in deze formule krijgen we dus voor n = 100, 4950 vergelijkingen. Aangezien er altijd evenveel vergelijkingen gebeuren zien we dat de grafiek geen uitsteeksels heeft. Het aantal elementen dat verwisseld wordt is niet constant, dit komt doordat er soms al een element juist staat waardoor het niet meer verwisseld moet worden. Als we dan gaan berekenen wat we in theorie zouden moeten uitkomen moeten we weer naar de best en worst case gaan kijken. In het beste geval gebeuren er natuurlijk geen verwisselingen. In het slechtste geval gebeuren er n - 1 verwisselingen (~n). Ook wanneer er een random rij mee gegeven wordt aan het algoritme zal de kans zeer klein zijn dat er een element juist staat, daarom gaan we ervan uit dat er ook ongeveer n - 1 verwisselingen plaatsvinden, wat neerkomt op ~n. Als we dit vergelijken met de bekomen resultaten zien we dat dit ook visueel vast te stellen is.





In onderstaande grafieken zien we de resultaten van QuickSort. De resultaten zijn op dezelfde manier als bij de vorige algoritmes bekomen. Als we eerst berekenen wat we theoretisch zouden moeten bekomen, kunnen we dat vergelijken met de bekomen resultaten. Voor QuickSort zal de gemiddeld aantal vergelijking sterk overeenkomen met wat we bij het beste geval berekent hebben. Dit komt doordat wanneer de rij bijvoorbeeld gesplitst wordt in 10%-90% in plaats van 50%-50% we ongeveer dezelfde formule gebruiken om dit voor te stellen, namelijk T(9n/10) + T(n/10) + partitioning(n). Als dit dus verder uitgerekend wordt komen we dus ook uit op een aantal vergelijkingen van ~c*nlog(n). Voor het aantal elementen die verwisseld moeten worden zullen we praktisch dezelfde formule krijgen wel met een andere constante.



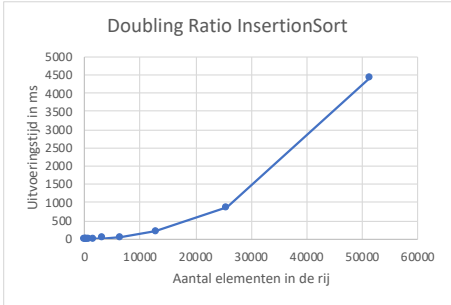


SedgeWick, R. (2011). Algorithms (Fourth ed.).

Doubling ratio experiment:

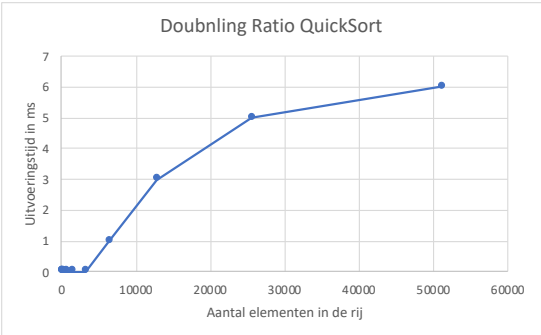
In het "Doubling Ratio" experiment gaan we twee sorteeralgoritmen, namelijk Insertion Sort en QuickSort, vergelijken. We geven steeds een rij mee die 2 keer zo lang is als de vorige rij. Ook gaan we daarna voorspellen hoe lang het duurt om een nog eens dubbel zo lange rij te sorteren. Bij dit experiment gaan we niet kijken naar het aantal vergelijkingen of aantal keer elementen gewisseld worden, omdat we dit al eerder hebben besproken en het dus dezelfde resultaten zal opleveren.

We zullen beginnen bij InsertionSort. Als we de resultaten die hieronder staan in een grafiek plotten krijgen we deze grafiek. We zien dat wanneer we de lengte van een rij verdubbelen, de tijd die nodig is om de rij te sorteren, exponentieel toeneemt. Dit heeft te maken met dat wanneer we steeds grotere rijen zullen sorteren er meer en meer elementen verwisseld moeten worden (ook exponentieel meer). Dit zal veel tijd in beslag nemen
Nu we weten hoe de grafiek verloopt kunnen we gaan bekijken hoe lang het zou duren moesten we de grootste rij maal 8 doen (409.600 elementen). Aangezien de grafiek exponentieel stijgt zal het waarschijnlijk heel lang duren voordat de rij gesorteerd is. Als we een voorspelling zouden kunnen maken zal het bijna een uur duren vooraleer de rij gesorteerd is.




```
Insertion Sort
50, totalCompares = 680, totalSwaps = 634, time: 0 ms
100, totalCompares = 2473, totalSwaps = 2378, time: 0 ms
200, totalCompares = 10240, totalSwaps = 10048, time: 0 ms
400, totalCompares = 37238, totalSwaps = 36844, time: 1 ms
800, totalCompares = 163950, totalSwaps = 163157, time: 3 ms
1600, totalCompares = 630855, totalSwaps = 629265, time: 7 ms
3200, totalCompares = 2588852, totalSwaps = 2585659, time: 14 ms
6400, totalCompares = 10216363, totalSwaps = 10209971, time: 57 ms
12800, totalCompares = 40673489, totalSwaps = 40660694, time: 181 ms
25600, totalCompares = 163423880, totalSwaps = 163398289, time: 1012 ms
51200, totalCompares = 655612777, totalSwaps = 655561590, time: 3971 ms
```

Als we gaan kijken naar QuickSort bekomen we de volgende resultaten. We zien hier dat de grafiek een logaritmische stijging volgt. Dit komt doordat wanneer we kijken naar het aantal elementen dat verwisseld moet worden wanneer we steeds grotere rijen nemen ook een logaritmische stijging volgt.
Zo kunnen we ook gaan voorspellen wat er gebeurt wanneer we het grootste testresultaat maal 8 doen (409.600 elementen). Aangezien de stijging slechts logaritmisch is zal de grafiek dus afzwakken tot een bepaalde tijd. Volgens mijn verwachtingen zal dit rond de 9ms liggen. We kunnen dus concluderen dat QuickSort zeer geschikt is voor zeer grote rijen.




```
Quick Sort
50, totalCompares = 215, totalSwaps = 62, time: 0 ms
100, totalCompares = 420, totalSwaps = 162, time: 0 ms
200, totalCompares = 935, totalSwaps = 376, time: 0 ms
400, totalCompares = 2623, totalSwaps = 814, time: 0 ms
800, totalCompares = 5827, totalSwaps = 1840, time: 0 ms
1600, totalCompares = 12800, totalSwaps = 4137, time: 0 ms
3200, totalCompares = 27646, totalSwaps = 8937, time: 0 ms
6400, totalCompares = 62533, totalSwaps = 19194, time: 0 ms
12800, totalCompares = 144354, totalSwaps = 41568, time: 1 ms
25600, totalCompares = 283524, totalSwaps = 88956, time: 2 ms
51200, totalCompares = 593517, totalSwaps = 191105, time: 5 ms
```

Conclusie:

Nu we alle resultaten hebben besproken kunnen we kijken welk algoritme nu juist het snelste is. Als het op tijd aankomt zagen we dat QuickSort de grote favoriet is doordat de tijd die nodig is om een rij te sorteren praktisch constant blijft, zelfs voor extreem grote rijen. Als we dit vergelijken met InsertionSort zien we dat QuickSort duidelijk de snelste is.
Als de rij die we meegeven nu het meest gunstige geval is voor dat algoritme zien we dat InsertionSort dan weer beter uitkomt doordat het het minste vergelijkingen zal moeten uitvoeren. Terwijl wanneer de rij het meest ongunstig is voor het algoritme QuickSort dan weer beter doet. SelectionSort zal meer vergelijkingen moeten uitvoeren maar veel minder elementen moet verwisselen dan QuickSort. [3]

[3] SedgeWick, R. (2011). Algorithms (Fourth ed.).