Sortera en array

Mattias Sandberg

13 Sep 2022

Introduktion

I denna uppgift ska vi utforska några sorteringsalgoritmer. Detta ska vi göra eftersom alla olika sökningsalgoritmer har sina för och nackdelar och det är därmed viktigt att veta vilka man ska använda. I uppgifterna nedan kommer vi enbart att jobba med arrayer med en bestämd storlek, dev ej utökande dataset. I rapporten kommer jag att uttrycka exekveringstiden som en funktion av storleken på arrayen.

Selection Sort

Sorteringsalgoritmen Selection Sort fungerar genom att algoritmen börjar med att sätta upp ett index till den första positionen i arrayen. Sen hittar algoritmen det minsta värdet i arrayen och byter plats på det värdet och ursprungs index. Sedan flyttas index fram en position och samma process utförs till arrayen är sorterad. När det inte hittas något mindre värde inför "corner cases" för att slutgöra algoritmen. Nedan kommer en graf över exikveringstiden i förhållande till antal element i arrayen. Denna graf är gjord i gnuplot och mätvärderna uttagna från Selection Sorts main funktion System.nanoTime();

```
public static void selectionSort(int[] number) {
                int minimumindex;
                for (int i = 0; i < number.length - 1; i++) {
                        minimumindex = i;
                        for (int k = i; k < number.length; k++) {
                                if (number[k] < number[minimumindex]) {
                                        minimumindex = k;
                                }
                        }
                        if (minimumindex != i) {
                                int temporary = number[i];
                                number[i] = number[minimumindex];
                                number[minimumindex] = temporary;
```

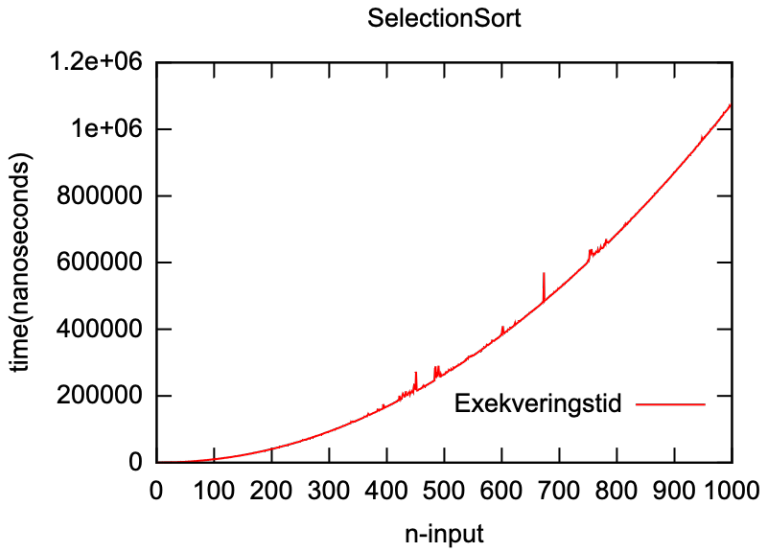

Figure 1: Graf över SelectionSort (tidskomplexitet)

Om det plottas en graf med Array som x-axel och Nanoseconds som y-axel så representeras en kvadratisk funktion och därmed är tidskomplexitet Ordo $O(n^2)$. Best case för denna algoritm är även det $O(n^2)$.

Insertion Sort

Selection Sort algoritmen går att utveckla genom att istället för att ta det minsta värdet i arrayen som ska sorteras så stoppar vi in ett föremål i den redan sorterade delen av arrayen. Algoritmen för detta är först och främst att börja med index på första poistionen i arrayen. Om föremålet på index positionen är mindre än föremålet före index positionen flyttas det föremålet mot början på arrayen och detta är själva ”insert” delan av algoritmen. När värdet blir instoppad på rätt plats flyttas index frammåt. När ett värde flyttas till början av arrayen måste plats utgöras och därmed flyttas alla föremål i arrayen ett steg. Nedan kommer en graf över exikverinstiden i förhållande till n-inputs. Grafen är konstruerad med hjälp av gnuplot där polynomen för Selection sort och insertion sort jämförs i samma graf. Grafen nedan förklarar att insert sort är likväl selection sort $O(n^2)$ dvs att tidskomplexiteten är kvadratisk. Den nya förbättrade algoritmen är cirka 30 procent snabbare än selection sort. Best case Insertion Sort O(n).

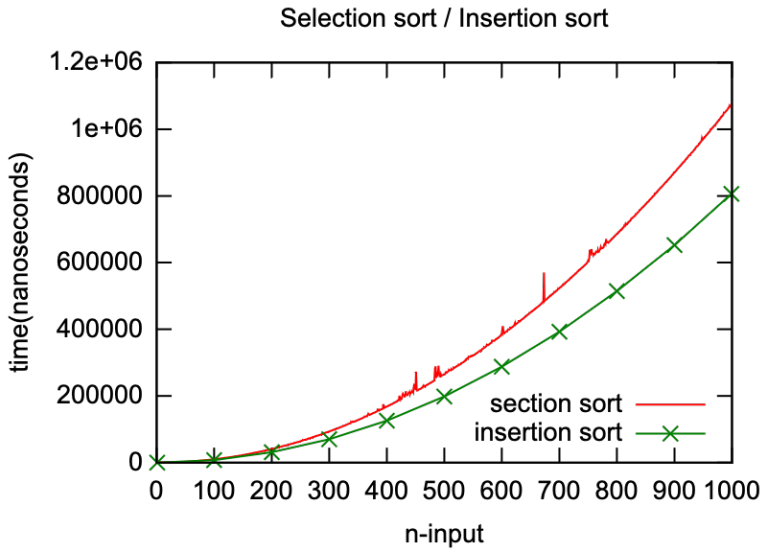

Figure 2: Graf över Selection sort och Insertion sort (tidskomplexitet)

Merge Sort

Merge sort algoritmen skiljer sig mest från det två tidigare sorteringsalgoritmer genom att den använder en ytterligare array där den temporät sparar värderna innan den slutliga arrayen konstrueras. Algoritmen kommer också införas rekursivt vilket var nytt för mig och var en utmaning till en början. Merge algoritmen börjar med att arrayen delas upp i två, och sedan placeras den första sorterade delan i en array och den andra sorterade i en annan array. Sedan slås ”merge” det två arrayerna ihop till en sorterad array. Nedan kommer en kort bit kod vilket handlar om att splita arrayen i två och använda det rekursiva anroppen. Nedan visas även en graf från gnuplot över exekveringstid i förhållande till storleken på arrayen n.

```
// Delar upp arrayen i två arrays, en höger och vänsterdel
        int middle = number.length / 2;
        int[] right = Arrays.copyOfRange(number, middle, number.length);
        int[] left = Arrays.copyOfRange(number, 0, middle);

        // rekursiva anrop
        mergeSort(right);
        mergeSort(left);
```

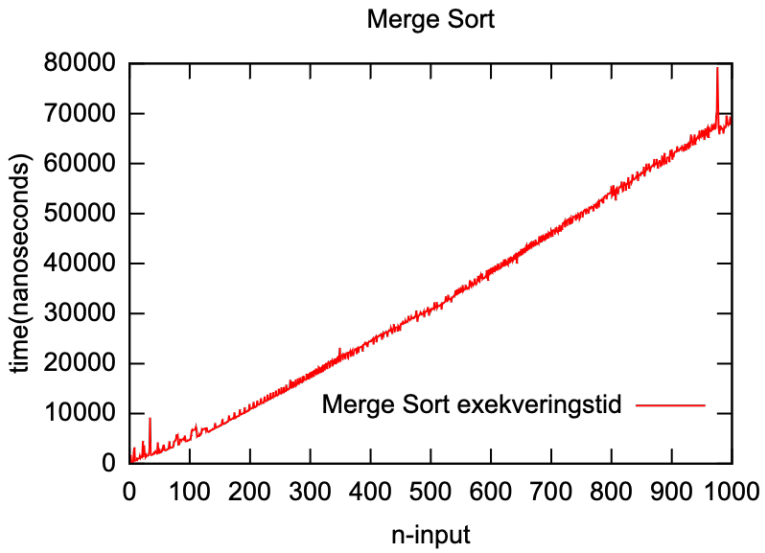

Figure 3: Graf över Merge Sort (tidskomplexitet)

Tidskomplexitetn för Merge sort är för både Worst-Best-Average case O(nlog(n)). Min graf med mina mätvärden visar mera ett linjärt polynom men polynomet är igentligen logaritmiskt: Min analys är att det hade behövts fler n-värden (större arrayer) dock tar det väldigt lång tid på min personliga laptop. Anledningen till att Merge Sorts tidskomplexitet är logaritmisk är eftersom jag förra labben lärde mig från Binary Search att om man delar ett värde för varje steg så kan det representeras som en logaritmisk funktion O(log(n)). Algoritmen utför också enstegs operationer när mitten på delarrayerna ska hittas, tidskomplexitetn för det är O(1). För att återföra arraysen (merge) är tidskomplexiteten O(n). Den totala Tidskomplexiteten för Merge Stort sorteringsalgoritmen kommer därmed bli O(n(log(n)) vilket är fullt rimligt. När jag jämför det tre sorteringsalgoritmerna hitils i grafen nedan är merge sort betydligt snabbre än det andra upp till iallafall array storlekar som min dator är bekväma att testa med. Detta visualiseras i grafen nedan:

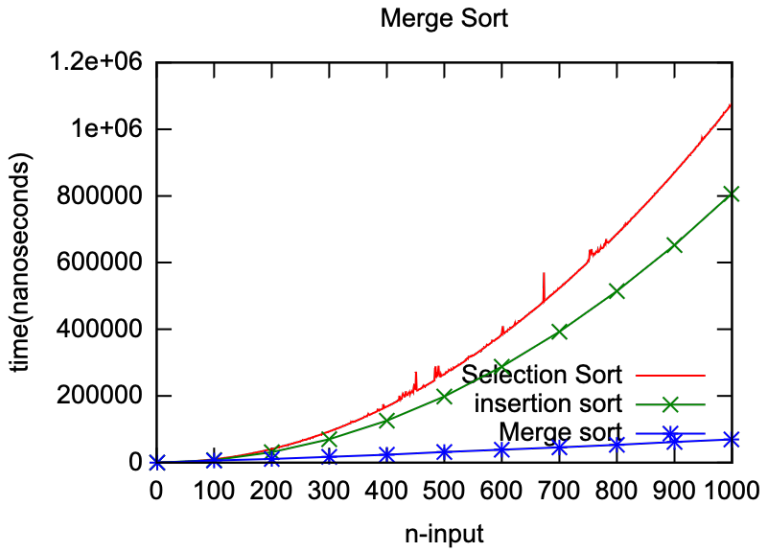

Figure 4: Graf Merge Sort, Selection Sort, Insertion Sort (tidskomplexitet)

Quick sort

En av dom mest använda sorteringsalgoritmerna heter Quick Sort och är likväl Merge Sort en rekursiv algoritm. En av det stora skillnaderna är att Quick Sort inte behöver en temporär array. Quick Sort algoritmen går ut på att lägga det lägre värderna till vänster i arrayen och det större till höger i arrayen. Vi sorterar sedan det två olika delarna rekursivt och sedan är det klart. Fördelen med detta är att vi slipper konstruera en ny array, nackdelen är dock att det finns speciella fall Quick Sort algoritmen till och med är sämmre än Selection Sort. Därmed är Merge Sort en sorteringsalgoritm som är mer tillförlitlig. Delar av koden som jag bedömmer mest vesäntlig samt en graf över exikveringstid representeras nedan:

```
public static void QuickSort(int[] number) {
                QuickSort(number, 0, number.length - 1);
        }

        private static void QuickSort(int[] number, int f, int l) {
                if (l > f) {
                        int p = Partition(number, f, l);
                        QuickSort(number, f, p - 1);
```

```
                QuickSort(number, p + 1, l);
            }
        }
```

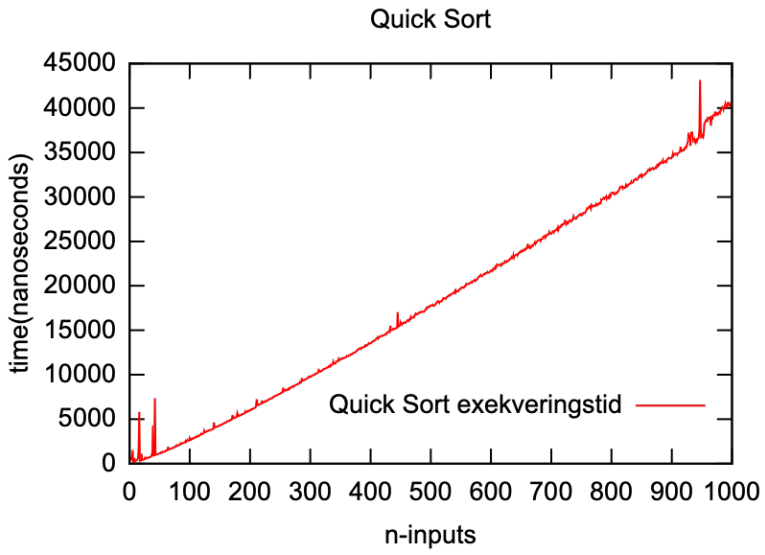

Figure 5: Graf över Quick Sort (tidskomplexitet)

Tidskomplexiteten för Quick Sort är som Merge Sort O(nlog(n)) fast ungefär dubbelt så snabb. Best case och Average case är O(nlog(n)) men worst case är $O(n^2)$. Skillnaden i exekveringstid mellan Quick sort och Merge sort representeras i rapportens sista graf nedan:

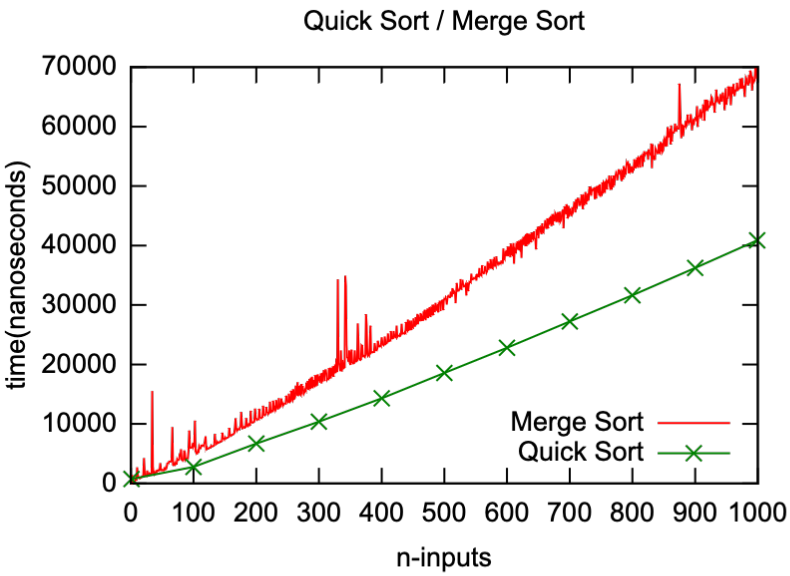

Figure 6: Graf över Merge Sort, Quick Sort (tidskomplexitet)