Hur man sorterar

viktor Björkén

September 2022

1 Introduktion

I denna rapport ska vi hantera hur olika sorterings algoritmer fungerar och hur deras tidskomplexitet skiller åt. Vi kommer gå igenom flera olika sorters algoritmer, för att få svar på vilken som är mest efektiv

2 Selection Sort

I denna första del ska vi använda oss av en så kallad "selection sort". Det denna sorting gör är att vi väljer det första elementet i arrayen och leter efter ett annat element som mindre, hittar vi vårat "minsta" värde byter vi plats.

Vi fyller i det som behövs i koden och får föjande kod för algoritmen

```
public static void selectionSort(int[] nums) {
                int minindex;
                for (int i = 0; i < nums.length - 1; i++) {
                        minindex = i;
                        for (int j = i; j < nums.length; j++) {
                                if (nums[j] < nums[minindex]) {
                                        minindex = j;
                                }
                        }
                        if (minindex != i) {
                                int tmp = nums[i];
                                nums[i] = nums[minindex];
                                nums[minindex] = tmp;
                        }
                }
        }
```

När vi kör våran algoritm får vi följande graf när vi har gjort mätningar på 1000 element (n).

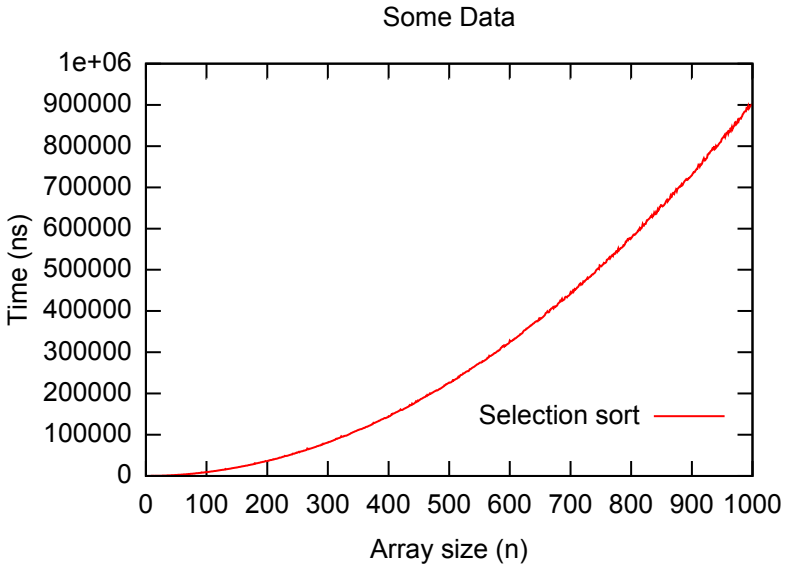

Figure 1: Selection Sort

Här ser vi att vi har tidskomplexiteten

$$O(n^2)$$

på våran selection sort. Detta då vi ser hur tiden ökar med antalet element i arrayen. Funktionen kan alltså ses som kvadratisk

3 Insertion Sort

Denna sortering fungerar på liknande sätt till den föregående algoritmen, skillanden är insertion sort delar upp arrayen i 2 delar en sorterad och en osorterad. Till en början är bara ett element i den sorterade delen, vi kollar sedan i den osorterade till vi hittar ett mindre värde, detta värde läggs sedan på rätt plats i den sorterade arrayen. Detta pågår tills alla värden är i den sorterade. Denna algoritm ser ut på följande sätt.

```
public static void InsertionSort( int[] nums) {
    int n = nums.length;
    for (int i = 1; i < n; ++i) {
        int key = nums[i];
        int j = i - 1;
```

```
        while (j >= 0 && nums[j] > key) {
            nums[j + 1] = nums[j];
            j = j - 1;
        }
        nums[j + 1] = key;
    }
}
```

När vi sedan kör våran benchmark, på samma sätt som i tidgare deluppgift får vi följande graf.

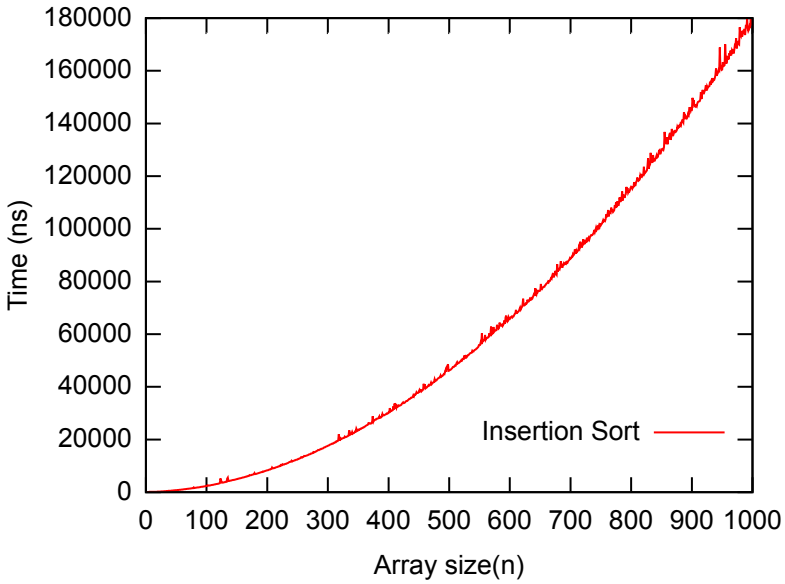

Figure 2: Insertion Sort

Vi ser tydligt att vi har samma sort tidskomplexitet som den tidigare selection sort, skillnaden är att denna är mycket mer tidseffektiv per antal element i arrayen, alltså bätte tid med samma antal element. Detta visar att denna också har en tidskomplexitet på

$$O(n^2)$$

4 Merge Sort

Om man kan dra flera likheter mellan insertion sort och selection sort fungerar merge sort på ett litet annat sätt. I merge sorten delast arrayen upp i delar där varje del sedan sorteras tillbaka ”upp” igen. När dessa sorteras tillbaka är alltså då vi ”mergear” de, därav namnet.

Vi inplementerar denna algoritm på följande sätt, med hjälp av koden vi fått i uppgiften samt andra sidor där merge sorten förklaras.

```
public static void mergeSort(int[] nums) {

        if (nums.length <= 1) { // base case
                return;
        }

        // delas i 2
        int middle = nums.length / 2;
        int[] left = Arrays.copyOfRange(nums, 0, middle);
        int[] right = Arrays.copyOfRange(nums, middle, nums.length);

        // rekrusiva delen
        mergeSort(left);
        mergeSort(right);

        // Mergear i upp arrayerna
        int l = 0;
        int r = 0;
        while (l + r < nums.length) {
                if (l < left.length && r < right.length) {
                        if (left[l] < right[r]) {
                                nums[l + r] = left[l++];
                        } else {
                                nums[l + r] = right[r++];
                        }
                } else if (l < left.length) {
                        nums[l + r] = left[l++];
                } else {
                        nums[l + r] = right[r++];
                }
        }
```

Vi kan då med hjälp av denna kod köra och benchmarka fram våra värden för att få fram en graf. Vi kommer göra detta på samma sätt som de tidigare uppgifterna, för att få ett så likvärdigt resultat som möjligt för att kunna gemföra.

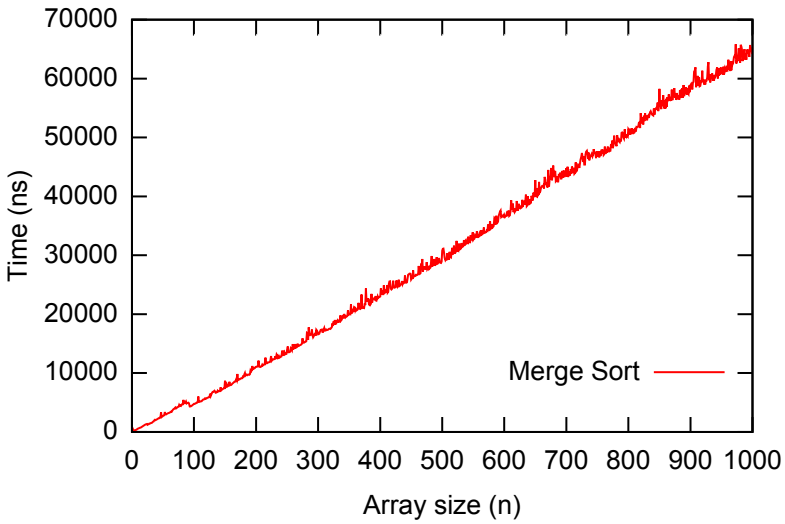

Figure 3: Merge Sort

Tidskomplexitet ser väldigt linjär ut från mitt resultat men faktum är att merge sort har en tidskomplexitet på O(n log(n)). Detta vet vi med tanke hur binary search funktionen såg ut i förra uppgiften. När vi delar upp våran array i delar varje steg kommer vi få en logaritmisk funktion. När vi sedan ”mergar” ihop våra del-arrayer får vi en tidskomplexitet på O(n).

Därför blir helheten för merge sort O(n log(n)).

Vi kan dessutom se klart och tydligt via våra grafer att merge sort är klart den snabbaste sorterings algoritmen som vi har testa. Förutom att den kommer vara mer effektiv på högre värden med tanke på tidskomplexiteten så ser vi på våran tidsaxel att vi har en betydligt lägre tid för samma storlek array.

5 Ännu bättre?

Vi har nu sätt hur merge sort är en bra och robust sorterings algoritm men faktum är att det finns algoritmer som tar i tu med de probelem som uppstår med merge sort. En av dessa är quick sort, detta är en av de mest använda sorterings algoritmerna och av bra anledning. Tidskomplexiteten för en quicksort algoritm är den samma som merge sort O(n log(n)), men får alltså ändå en mer effektiv exekveringstid. Ett sätt att skriva denna kod är på följande:

```
        static void quickSort(int[] arr, int low, int high)
{
    if (low < high)
    {
```

```
        int pi = partition(arr, low, high);

        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}
```

Denna kod kommer behövas kompleteras med funktioner för bland annat ”delningen”