Assignment 4: Sorting an array

Mahad Ahmed

Spring Fall 2022

Introduction

In this assignment the main purpose was to try out different search algorithms and find out what their pros and cons are. In this assignment we only worked with a arrays of given size i.e doesn't take in any new elements.

Task 1, Selection sort

In this task we implemented a simple search algorithm which is not very efficient as you soon shall see. But it is simple to implement and works for smaller arrays.

```
public static void selectionSort(int[] array) {
       for (int i = 0; i < array.length -1; i++) {
           int cand = i;
           for (int j = i; j < array.length ; j++)
                  if (array[j] < array[cand])
                      cand = j;

           int temp = array[i];
           array[i] = array[cand];
           array[cand] = temp;
       }
   }
}
```

The algorithm itself is very easy to implement. Firstly you have to start by setting a index to the first position in your array. Then starting from the index you go through the whole array searching for the smallest number compared to the index. Then switch those two and hop on to the next element on your array until everything is sorted. This only works somewhat when the array is already small. But a bigger array or data collection will cause our computer to malfunction or we will get bored of how long it takes.

Benchmark, Selection sort

Here we have the benchmark for the algorithm in the first task.

Elements n	Run time(ms)
100	11
200	35
400	118
800	417
1600	1495
6400	22 318

Table 1: Benchmark of the algorithm *Selection sort*

As you can see the more elements we get the slower our algorithm is. The run time seems to double or even more every time we double the size of the array. We have two loops, one that selects an element in the array and another that compares that element with the other elements in the array. Not very time efficient. A nested for loop gives us $O(n*n) = O(n^2)$

Task 2, Insertion sort

The next algorithm we have implemented is called insertion sort. In this algorithm we we start by pointing to the first index in our array and look if the index position is smaller than the item before the move. If it is we move the item towards the beginning.

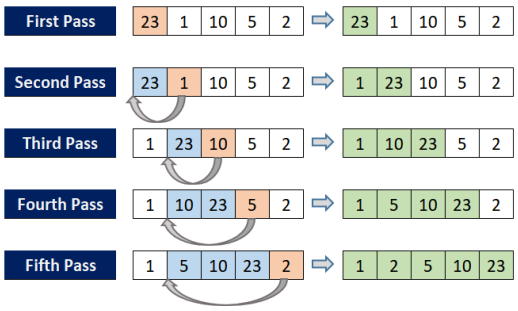

Figure 1: An illustration of how a insertion sort algorithm works

Implementation of the insertion sort algorithm

Here I will walk you through the code implementing the algorithm.

```
public class Insertion {
    public static void sort(int[] array) {
        for (int i = 0; i < array.length; i++) {
            // for each element from i towards 1, swap the item
                found with the
            // item before it if it is smaller
            for (int j = i; j > 0 && array[j] < array[j-1]; j--) {
                int temp = array[j];
                array[j] = array[j-1];
                array[j-1] = temp;
            }
        }
    }
}
```

In this code section above we implemented a simple algorithm. We just swap until we are happy with its position and move on to the next index thus always giving us the smallest element first and so on. This way of sorting an algorithm is not very efficient but insertion is not as bad as selection sort. Even though they both have the same time complexity insertion sort is a bit faster.

Benchmark, Insertion sort algorithm

Here is the benchmark for insertion sort.

Elements n	**Run time(ms)**
100	4
200	11
400	58
800	151
1600	570
6400	8992

Table 2: Benchmark of the binary algorithm

Here we can see that insertion sort is quite a bit faster than selection sort but it still has the same time complexity for larger elements. It is still not that efficient as it shows how it handles larger and larger elements. Time complexity for this is the same as selection sort i.e $O(n^2)$.

Task 3, Merge sort

Divide and conquer. This is what merge sort is based upon. But what does divide and conquer mean in this context? In this context divide and conquer means that the array that is being sorted is divided up into two equal sized arrays, then sorted and merged back as one array hence *Merge sort*. We also make a temporary "storage" when we are going to merge the elements. But how do we merge? The nice thing with having two sorted arrays is that the smallest is always first and so on. Only thing we have to do is to compare the cards and put the smallest in the storage and so on. Like if you have a deck of card and you have divided the the deck in two equal sizes and sorted each deck. Then the smallest cards will be facing up and you just pick the smallest from each pile, compare them and put the smallest first on the table. Take the other one and compare it to the next card in the pile were we found the smallest.

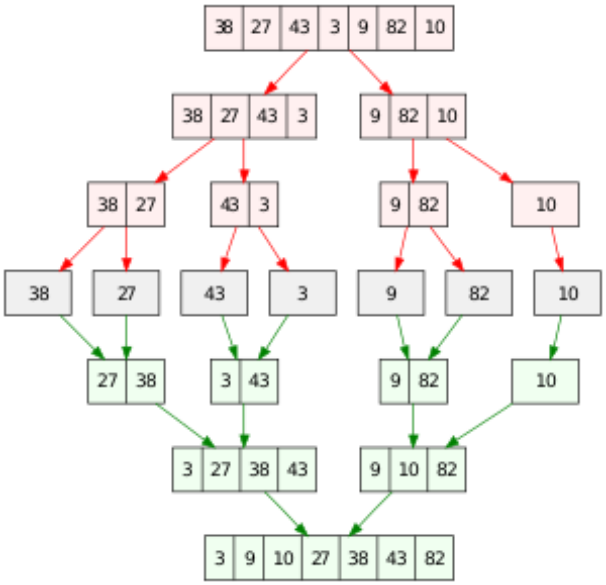

Figure 2: An illustration of how a merge sort algorithm works

Implementation of task 3: Merge sort

Here we implemented the merge sort algorithm.

```
public static void sort(int[] org) {
    if (org.length == 0)
        return;
    int[] aux = new int[org.length];
    sort(org, aux, 0, org.length - 1);
```

```
}

private static void sort(int[] org, int[] aux, int lo, int hi) {
    if (lo != hi) {
        int mid = lo + (hi-lo)/2;
        sort(org, aux, lo, mid);
        sort(org, aux, mid + 1, hi);
        merge(org, aux, lo, mid, hi);
    }
}

private static void merge(int[] org, int[] aux, int lo, int
    mid, int hi) {
    for (int i = 0; i < org.length; i++)
        aux[i] = org[i];
    int i = lo;
    int j = mid + 1;
    for (int k = lo; k <= hi; k++) {
        if (i > mid)
            org[k] = aux[j++];
        else if (j > hi)
            org[k] = aux[i++];
        else if (aux[i] < aux[j])
            org[k] = aux[i++];
        else
            org[k] = aux[j++];
    }
}
```

We implemented this code in the following way. Firstly declaring an array, it's mid, right and left variable. Then performing the merge function.

Benchmark, Merge sort

Elements n	Run time(ms)
100	9
200	20
400	76
800	124
1600	337
6400	6128

Table 3: Benchmark of merge sort algorithm

As you can see this is way faster than the others we have implemented. Merge sort is a recursive algorithm meaning it calls itself with smaller values

then returns the result and doing it over and over again. Like I explained withe the cards, we are splitting the deck into smaller and smaller group of cards until we stop. Therefore the time complexity for this algorithm is *O(nlog(n))*