Lab 4 How To Sort

Weichen Chai

Fall 2022

Introduction

This lab report seeks to answer the questions presented in "How to sort" for ID1021. In this assignment, the goal is to analyse and understand the different sorting algorithms, and how efficient they are through bench marking using the in built java clock, which measures in nanoseconds.

Background

This task enhances our understanding in sorting algorithms. We will work with select sort, insertion sort, and merge sort. Each are different and therefore have different time complexities, so it is important to distinguish which are faster in which types of situations.

Tasks

Select sort

The first task in the assignment requires for us to benchmark the select sort algorithm in an array. This sort method first assigns a "candidate" in the first position, which will be compared to the all the values in the array. If a value is smaller than the current candidate, it will automatically replace it. The code that the task provides can be changed to:

```
for (int i = 0; i < array.length -1; i++) {
// let's set the first candidate to the index itself
int cand = i;
for (int j = i; j < array.length ; j++) {
    if(array[cand]>array[j]){
    cand = j
}
}
```

Since this sort method takes one loop to find an element in the array, and another loop to compare that element to other elements in the array, its time complexity is O(n^2). By running the benchmark, we can visualize:

Elements in array	**Time in Nanoseconds (Selected Sort)**
5	202
10	338
50	593
100	1982
500	43693
1000	149841
5000	3746025

The data shown above were run 10000, and averaged to find the most accurate representation of the time it would take to use the select sort method in arrays of various sizes. The data does indeed take the form of a O(n^2), therefore the hypothesis was correct.

Insertion sort

This task requires for us to work with another sort technique, namely , Insertion sort. In order to execute this method, we must make sure that the code compares two variables in the array. If the one after, when getting compared is smaller than the element before, their positions will be switched, this loop will continue until the array before the element currently being scanned is fully sorted. The code given by the task is:

```
for (int i = 1; i <array.length ; i++) {
            for (int j = i; j > 0 && array[i] <array[j]; j--) {

                    //Simple Swapping logic
                    int new = array[i];
                    array[i] = array[j];
                    array[j] = new;
```

After running the piece of code in a benchmark, we receive the following results.

The results shows that insertion sort technique follows the same pattern as selection sort, having the same time complexity of O(n^2). The time taken is slow at the start, however as the array size increases, it becomes faster than selection sort. This indicates that insertion sort is a faster way of sorting if the array size is huge.

Elements in array	Time in Nanoseconds (Insertion Sort)
5	112
10	165
50	618
100	2683
500	23403
1000	65841
5000	229212

Merge sort

The final sorting method in this assignment is called the merge sort. Using the recursive merge sort method, it separates the array into two halves, sorting each half individually before rejoining them in a complete sorted array. The code we were asked to complete in the task are as follows:

```
private static void sort(int[] org, int[] aux, int lo, int hi) {
.
.
// sort the items from lo to mid
            sort(org,aux,lo,mid);
// sort the items from mid+1 to hi
            sort(org,aux,mid+1,hi);
.
.
        }
    }
```

To implement the merge method we first copy all the elements to the temporary array, so that we can use it without draw backs.

```
// copy all items from lo to hi from org to aux
for (int i = 0 ; i < org.length; i++){
org[i] = aux[i]
}
```

Then we begin the merging process:

```
for ( int k = lo; k <= hi; k++) {
// if i is greater than mid, move the j item to the org array, update j
if(i> mid ){
    org[k] = aux [j]
    j++
```

```
// else if j is greater than hi, move the i item to the org array, update i
    else if (j > hi) {
        org[k] = aux[i];
        i++;
    }
// else if the i item is smaller than the j item,
// move it to the org array, update i
    else if (aux[i] < aux[j]) {
        org[k] = aux[i];
        i++;
    }
.
.
.
```

The time complexity for merge sort can be written in the following equation: T(n) = 2T(n/2) + O(n). Which is O(nLogn). The data received after bench marking the code is as follows.

Elements in array	**Time in Nanoseconds (Merge Sort)**
5	320
10	750
50	4287
100	9080
500	30245
1000	52269
5000	203350

As shown above, the trend of the data does indeed seem to be of O(nLogn). When compared to insertion and selection sort, it is much slower initially, when the array size is small, but slowly over takes both selection and insertion sort when the array gets bigger.