# Sorting report

This report will show what I did to modify the sorting and present the results of different sorting. Then, by obverting and comparing these data, the results will be discussed.

## 1. Initial sorting results are shown below.

Selection Sort

| size | Random | Ordered | Reverse |
|---|---|---|---|
| 100 | 0.000s | 0.000s | 0.000s |
| 1000 | 0.002s | 0.001s | 0.002s |
| 10,000 | 0.242s | 0.260s | 0.452s |

* Data are the averages of 11 times of testing

Insertion Sort

| size | Random | Ordered | Reverse |
|---|---|---|---|
| 100 | 0.000s | 0.000s | 0.000s |
| 1000 | 0.001s | 0.000s | 0.001s |
| 10,000 | 0.122s | 0.000s | 0.271s |

* Data are the averages of 11 times of testing

Merge Sort

| size | Random | Ordered | Reverse |
|---|---|---|---|
| 100 | 0.000s | 0.000s | 0.000s |
| 1000 | 0.000s | 0.000s | 0.000s |
| 10,000 | 0.006s | 0.000s | 0.000s |
| 100,000 | 0.037s | 0.023s | 0.019s |
| 1,000,000 | 0.632s | 0.374s | 0.320s |

* Data are the averages of 11 times of testing

Quick Sort

| size | Random | Ordered | Reverse |
|---|---|---|---|
| 100 | 0.000s | 0.000s | 0.000s |
| 1000 | 0.001s | 0.000s | 0.000s |
| 10,000 | 0.003s | 0.001s | 0.001s |
| 100,000 | 0.041s | 0.031s | 0.027s |
| 1,000,000 | 0.740s | 0.479s | 0.480s |

* Data are the averages of 11 times of testing

Quick Sort 2

| size | Random | Ordered | Reverse |
|---|---|---|---|
| 100 | 0.000s | 0.000s | 0.000s |
| 1000 | 0.000s | 0.000s | 0.000s |
| 10,000 | 0.002s | 0.001s | 0.001s |
| 100,000 | 0.038s | 0.029s | 0.030s |
| 1,000,000 | 0.775s | 0.512s | 0.527s |

* Data are the averages of 11 times of testing

Arrays.sort

| size | Random | Ordered | Reverse |
|---|---|---|---|
| 100 | 0.000s | 0.000s | 0.000s |
| 1000 | 0.000s | 0.000s | 0.000s |
| 10,000 | 0.005s | 0.000s | 0.000s |
| 100,000 | 0.041s | 0.005s | 0.008s |
| 1,000,000 | 0.667s | 0.067s | 0.178s |

* Data are the averages of 11 times of testing

## 2. Modifications

1. **Selection Sort**:

Original version: The standard algorithm runs through the remaining part of the array, looking for the minimum value and then swaps it to the front.

For the modified version by changing the algorithm, so that it also looks for the maximum value in the remaining part, and swaps it to the back, so that it builds up a sorted list from the front and the back at the same time.

From above description, the pseudocode I think might like this:

```
While (current first index == end index) {
    Search the the smallest items through the remaining part of the array
    Swap the item of current index with smallest item
    Search the the smallest items through the remaining part of the array
    Swap the item of current index with biggest item
}
```

By running and testing the code with the above pseudocode thinking. There was a bug found which is that if the first item is the biggest item, it will be swapped with the smallest item and the smallest item will be swapped to the end. The result is not what we want. Therefore, an if-statement is added to avoid this situation. The right version of pseudocode is that,

```
While (current first index == end index) {
    Search the the smallest items through the remaining part of the array
    Swap the item of current index with smallest item
    Search the the smallest items through the remaining part of the array
        If (currentFirstItem == biggest item) {
        Swap the current item with the current first index item
        }
        else{
        Swap the item of current index with biggest item
        }
}
```

Result for sorting by modified version of selection (Selection Sort 2)

Selection Sort 2

| size | Random | Ordered | Reverse |
|---|---|---|---|
| 100 | 0.000s | 0.000s | 0.000s |
| 1000 | 0.006s | 0.001s | 0.001s |
| 10,000 | 0.223s | 0.280s | 0.368s |

* Data are the averages of 11 times of testing

Selection Sort 1

| size | Random | Ordered | Reverse |
|---|---|---|---|
| 100 | 0.000s | 0.000s | 0.000s |
| 1000 | 0.002s | 0.001s | 0.002s |
| 10,000 | 0.242s | 0.260s | 0.452s |

* Data are the averages of 11 times of testing
*red means selection sort2 faster than sort1, blue means that the sort2 slower than sort1

It was noticed that the modified selection sorting faster than original one when the data is reverse, especially the big enough data size. This maybe because the times of swapping will be more and more, if the times of iterating was less and less.

## 2. Insertion sorting

Original version: Insertion sort inserts each item into the sorted region that it is building up from the left. It does this by searching down the sorted region, one item at a time.

Modified version: It ought to be faster to use binary search on that region to find the right place to insert the item (in the same way that you used binary search to add an item in your `SortedArraySet`). Modify InsertionSort to do this.

The modified version code shown in the sorting.java file. The results of two version of soring shown and compared below

Insertion Sort 2 (modified version)

| size | Random | Ordered | Reverse |
|---|---|---|---|
| 100 | 0.000s | 0.000s | 0.000s |
| 1000 | 0.002s | 0.000s | 0.000s |
| 10,000 | 0.023s | 0.000s | 0.041s |

* Data are the averages of 11 times of testing

Insertion Sort

| size | Random | Ordered | Reverse |
|---|---|---|---|
| 100 | 0.000s | 0.000s | 0.000s |
| 1000 | 0.001s | 0.000s | 0.001s |
| 10,000 | 0.122s | 0.000s | 0.271s |

* Data are the averages of 11 times of testing
*red means insertion sort2 faster than sort1, blue means that the sort2 slower than sort1

By comparing the results of different versions of insertion sorting, it was noticed that the modified sorting will faster a lot, when the size of data is big enough. Another thing is shown from the two table, the time is always approximately 0, if the data is stored in a sorted array. I think that because the insertion sorting just sort the sorted data too fast so that the time is too small which is can be represented in 0.000 seconds.

## 3. Merge Sorting

The "standard" merge sort is recursive, but the pattern of subranges that it merges is completely predefined, and doesn't really need recursion. It could start by merging all the pairs of size 1 subranges (0-1 and 1-2, 2-3 and 3-4, 4-5 and 5-6, etc), then all the size 2 subranges: (0-2 and 2-4, 4-6 and 6-8, etc), then all the size 4 subranges:(0-4 and 4-8, 8-12 and 12-16, etc), etc.

This could be done with an outer loop that steps up the ranges:
`for(int range = 1; range<size; range = range*2) {`
and an inner loop that steps along all the subranges of the current range size:
`for(int start = 0; start<size; start = start+2*range) {`
calling merge on the subrange from `start` to `start+range` and `start+range` to `start+2*range`.

The pseudocode I think like this:

*While (Size of the subrange ==original data size) {*
*Step along all the subranges*
*mergeSort 2();*
*}*

The results of two version of soring shown and compared below:

Merge Sort 2

| size | Random | Ordered | Reverse |
|---|---|---|---|
| 100 | 0.000s | 0.000s | 0.000s |
| 1000 | 0.000s | 0.000s | 0.000s |
| 10,000 | 0.004s | 0.000s | 0.000s |
| 100,000 | 0.036s | 0.023s | 0.017s |
| 1,000,000 | 0.618s | 0.356s | 0.281s |

* Data are the averages of 11 times of testing

Merge Sort

| size | Random | Ordered | Reverse |
|---|---|---|---|
| 100 | 0.000s | 0.000s | 0.000s |
| 1000 | 0.000s | 0.000s | 0.000s |
| 10,000 | 0.006s | 0.000s | 0.000s |
| 100,000 | 0.037s | 0.023s | 0.019s |
| 1,000,000 | 0.632s | 0.374s | 0.320s |

* Data are the averages of 11 times of testing
*red means merge sort2 faster than sort1, blue means that the sort2 slower than sort1

It is obvious that modified merge sorting is faster than original merge sorting whatever the type or size of array.

## 5. Quick Sorting

The main problem with Quicksort is choosing a good pivot. The first item in the subrange is the worst possible choice for an almost sorted array. Other choices are:

- The mid value of the subrange (given code does this).
- The median of three values (eg, the first, the mid, and the last) of the subrange , which is what I did in the this assigment.

Another problem with Quicksort is that it is slow for sorting very small subranges. It is actually faster to use Insertion sort once a subrange gets small (eg up to 5 items). Change so that we use the medium of three as the pivot and change the recursive call so that it calls a version of insertion sort on any small subrange.

Quick Sort 3 (modified version)

| size | Random | Ordered | Reverse |
|---|---|---|---|
| 100 | 0.000s | 0.000s | 0.000s |
| 1000 | 0.001s | 0.000s | 0.000s |
| 10,000 | 0.003s | 0.001s | 0.001s |
| 100,000 | 0.039s | 0.027s | 0.035s |
| 1,000,000 | 0.617s | 0.468s | 0.480s |

* Data are the averages of 11 times of testing

Quick Sort

| size | Random | Ordered | Reverse |
|---|---|---|---|
| 100 | 0.000s | 0.000s | 0.000s |
| 1000 | 0.001s | 0.000s | 0.000s |
| 10,000 | 0.003s | 0.001s | 0.001s |
| 100,000 | 0.041s | 0.031s | 0.029s |
| 1,000,000 | 0.811s | 0.540s | 0.534s |

* Data are the averages of 11 times of testing

*red means modified quick sort faster than sort1, blue means that the sort2 slower than sort1

As two tables shown, it is obliviously that the modified quacking sorting is faster a lot than the given quick sorting, especially for the big size of array. Therefore, I believe that my code works well for modified quick sorting.

## 3. algorism of testing and printing all the results

The exact code was written in the sorting.java file. In this selection, the below comments are used to shown my thinking of testing and printing results

```
String[] data                                                    //create
String[] arrayType = {"Random", "Sorted", "Reverse"};            //type of array
int[] size = {100, 1000, 10000, 100000, 1000000};                //size of array
long[] timeCostArray = new long[testingTimes];                   //time cost array
long[][] table2DArray = new long[5][3];                  // 2D array to make a nice table

 //traverse read every type of array ( outer loop)
for (int arrayTypeNum = 0; arrayTypeNum < arrayType.length; arrayTypeNum++)

        // traverse read every size of array  ( inside loop)
         for (int arraySizeIndex = 0; arraySizeIndex < size.length; arraySizeIndex++)

        if (too big size of array && slow sorting) continue;
        // sorting the data by for-loop for given times
      for (int numOfTimes = 0; numOfTimes < timeCostArray.length; numOfTimes++)
        create a random array, then sorting it into different types of array.(sorted, reversed)
        calculating the cost of sorting;
        Add the cost to the timeCostArray

  End the nested loop, find the average cost of each sorting.

Finally, print everything out.
```

For testing, just uncomment these “UI.printf("Sorted correctly: %b\n", testSorted(data));)” shown below.

```
switch (sortType){
    case "Selection Sort":
    selectionSort(data);
    break;
    case "Tim Sort":
    timSort(data);
    UI.printf("Sorted correctly: %b\n", testSorted(data));
    break;

    case "Selection Sort 2":
    selectionSort2(data);
    //UI.printf("Sorted correctly: %b\n", testSorted(data));
    break;
    case "Insertion Sort":
    insertionSort(data);

    break;
    case "Insertion Sort 2":
    insertionSort2(data);
    //UI.printf("Sorted correctly: %b\n", testSorted(data));
    break;
    case "Merge Sort":
    mergeSort(data);

    break;
    case "Merge Sort 2":
    mergeSort(data);
    //UI.printf("Sorted correctly: %b\n", testSorted(data));
    break;
    case "Quick Sort":
    quickSort(data);

    break;
    case "Quick Sort 2":
    quickSort2(data);

    break;
    case "Quick Sort 3":
    quickSort3(data);
    //UI.printf("Sorted correctly: %b\n", testSorted(data));
    break;
    case "Arrays.sort":
    Arrays.sort(data);
    //UI.printf("Sorted correctly: %b\n", testSorted(data));
    break;
}
```

## 4.Reflection

I really enjoyed this assignment, since it is the last assignment in this trimester. By writing the report, this assignment gave me a lot of help to think how to develop different sorting and compare the advantages and disadvantage of different sorting for different type or size of data. So, I think that I know how to choose the best sorting for different type of data in the future.
On the other hand, at the beginning, I am a bit suck about how the given code work and how I can use these given methods to sorting different data. It took me a bit long time to figure out how this assignment works. But now, I felt happy that I have designed a nice interface of this assignment. Hence, I can be testing every type of data for every times by just clicking a button. Furthermore, the data in these data I think is still not accurate enough because sometimes my laptop works slower than the cost of sorting also become bigger, which is a kind of annoying me. Hence, I have to test the data as much as possible to ensure they are accurate enough.

I have also attempted the challenge part and test Tim Sort, the result shown below

Tim Sort

| size | Random | Ordered | Reverse |
|---|---|---|---|
| 100 | 0.001s | 0.005s | 0.002s |
| 1000 | 0.002s | 0.005s | 0.005s |
| 10,000 | 0.017s | 0.007s | 0.010s |

* Data are the averages of 12 times of testing

As the table shown, Tim sort is a very fast sorting. It is based on Merge sort, but also uses insertion sort. It is faster than Merge sort on partially sorted data. Therefore, I create the Tim sort by combining merge sorting and insertion sorts. Because insertion sort is faster in sorting small number of data. I just using the insertion sorting “from array” or “subrange” during the merge sorting. Even though I am not 100% sure what I did is right, I hope that I could get some points from my attempt. Because I testing the data correcting is true during Tim sorting.