CMPT 435 - Fall 2021 - Dr. Labouseur

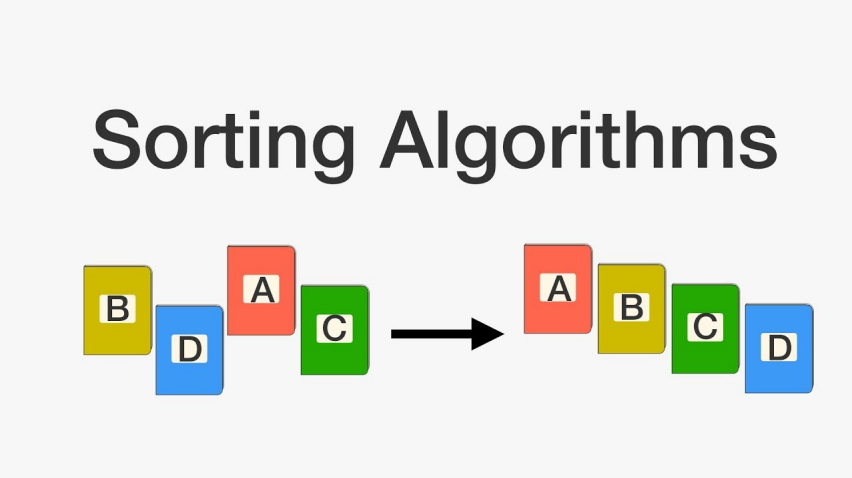
Assignment Two – Sorting Algorithms

Augusto Gonzalez-Bonorino
augusto.gonzalezbonorino1@marist.edu

October 8, 2021

1 Description of the program

The program aims to implement, and compare, the following sorting algorithms: Insertion, Selection, Merge and Quick sort. In regards to my implementation in Java I have decided to define one MainGonzalezBonorino class which contains the functions necessary to read the input text document containing the Strings to sort and the functions needed to implement each algorithm.The characteristics and details of each algorithm will be explored in detail throughout this documentation



2 Selection Sort

Selection sort runs in $O(n^2)$ due to the nested for loops used to traverse the array and compare each element, it works as follows: It repeatedly finds the minimum element of an unsorted section of the array and places that element at the beginning, thereby maintaining two sub arrays (one sorted and one being sorted) until the whole array is traversed. In each iteration the minimum element is moved to the sorted sub array.

```
1  public static int selectionSort(String[] magicList) {
2
3      knuthShuffle(magicList);
4      int len = magicList.length;
5      int numComparisons = 0;
6
7      for (int i = 0; i < len - 1; i++) {
8
9        int smallPos = i;
10
11       for (int j = i + 1; j <= len - 1; j++) {
12
13         if (magicList[j].compareToIgnoreCase(magicList[smallPos]) <
       0 ) //compare strings
14         {
15           smallPos = j;
16         }
17         numComparisons++;
18
19       } // inner for loop
20
21       // swap
22       String temp = magicList[smallPos];
23       magicList[smallPos] = magicList[i];
24       magicList[i] = temp;
25
26     } // outer for loop
27
28     return numComparisons;
29
30   } // selectionSort
```

The java implementation is fairly straightforward. First, to ensure the input array is shuffled, I implement a Knuth shuffle (a.k.a. the Fisher-Yates shuffle) which is commonly used to shuffle arrays. You will soon note that this function is used before implementing each sorting algorithm to make sure that we are not feeding the algorithm an already sorted array. Next, in the outer loop, we instantiate an integer called *smallPos* to keep track of the index of the smallest element. Then, we use that index to loop over each String in the magicitems.txt file and compare them. If, after the comparison was made, one of them was smaller than the current smallest element then *smallPos* is updated and the number of comparisons is incremented. Once we compared the elements, we swap them to create the sorted array. The algorithm repeats the aforementioned steps until the whole array is sorted.

3 Insertion Sort

Insertion sort, like Selection, has a run time of $O(n^2)$. Nevertheless, this algorithm provides an improvement over Selection sort because it avoids looping over the array if a subset of it is already sorted, thereby saving us some precious resources. Therefore, while the worst-case scenario is $O(n^2)$ it is possible to reduce it to $O(n)$ if the array is already sorted. The algorithm divides the array into two sub arrays (sorted and unsorted). Then, values from the unsorted part are picked and placed at the correct position in the sorted part.

```
1  public static int insertionSort(String[] magicList) {
2
3      knuthShuffle(magicList);
4      int len = magicList.length;
5      int numComparisons = 0;
6
7      for (int j = 1; j <= len - 2; j++) {
8
9        String key = magicList[j];
10       int i = j - 1;
11
12       while ( (i >= 0) && ( magicList[i].compareToIgnoreCase(key) >
          0) ) {
13
14         magicList[i + 1] = magicList[i];
15         i = i - 1;
16
17         numComparisons++;
18
19       } // while loop
20
21       magicList[i + 1] = key;
22
23     } // for loop
24
25     return numComparisons;
26
27   } // insertionSort
```

After shuffling the input array of Strings we assign the first element to a variable I have called *key*. Then, we compare the current element *key* with the one that precedes it, which is located at index *i*. If the key element is smaller than its predecessor, the program compares it to the previous elements and moves the greater elements one position up to make space for the swapped element. Finally, the current smallest element is inserted into the sorted array. The algorithm repeats these steps until the array is fully sorted.

4 Merge Sort

Merge sort is a recursive algorithm that runs in $O(n \cdot logn)$ for every possible case because it always divide the original array, and sub arrays, in half and then traverses that array of length n to compare the elements. Merge sort leverages the concept of **Divide and Conquer** by dividing each array in half until we are left with n arrays of length 1, where n represents the length of the original array. I have implemented the algorithm recursively by making use of two functions: *mergeSort* and *merge*.

```
public static int mergeSort(String[] magicList, int comparisons) {


    int len = magicList.length;

    if (len < 2)
      return comparisons;

    // Array's midpoint
    int midPoint = len / 2;

    // Left and right sub-arrays
    String[] left = new String[midPoint];
    String[] right = new String[len - midPoint];

    // Populate sub-arrays
    for (int i = 0; i < midPoint; i++)
      left[i] = magicList[i];

    for (int j = midPoint; j < len; j++)
      right[j - midPoint] = magicList[j];

    // Recursive call to keep dividing sub-arrays
    comparisons = mergeSort(left, comparisons);
    comparisons = mergeSort(right, comparisons);
    comparisons = merge(left, right, magicList, comparisons);

    return comparisons;

  }
```

mergeSort divides the input array in half, and the subsequent sub arrays. Note that the for loops are used to populate the arrays *left* and *right* which represents the two sub arrays that were created by dividing the original array in half. Once the array has been completely "divided", the function calls *merge* which takes care of the sorting. Let's take a look at how this is achieved:

```
public static int merge(String[] left, String[] right, String[]
    magicList, int comparisons) {

    // merge sub-arrays back together
    int i = 0;
    int j = 0;
    int k = 0;

    knuthShuffle(magicList);
```

```
    while (i < left.length && j < right.length) {

      comparisons++;

      if (left[i].compareToIgnoreCase(right[j]) < 0 ) {

        magicList[k] = left[i];
        i++;

      } // if statement

      else {

        magicList[k] = right[j];
        j++;

      } // else statement

      k++;

    } // while loop


    while (i < left.length) {

      magicList[k] = left[i];
      i++;
      k++;

    } // while loop

    while (j < right.length) {

      magicList[k] = right[j];
      j++;
      k++;

    } // while loop

    return comparisons;

  } // mergeSort
```

The sorting is achieve while merging the sub arrays. First, we compare two of the sub arrays of length one and merge them into a sorted array of length two. Same process is repeated to compare the arrays of length two, and so on until we have fully merged the sorted sub arrays into a completely sorted array of length n (length of the original array).

5 Quick Sort

Like Merge sort, Quick sort is a **Divide and Conquer**. The biggest difference is that instead of dividing the array in half, we divided it along a *pivot* element. There are multiple options on how to pick such a pivot:

1. Always pick first element as pivot.
2. Always pick last element as pivot.
3. Pick a random element as pivot.
4. Pick median as pivot.

I opted for picking the last element of the array as my pivot element. The asymptotic analysis of Quick sort is interesting to explore as it varies with how sorted the input array is from the beginning and how the partition is implemented. The worst case scenario for Quick Sort is $O(n^2)$ which is significantly worse than Merge sort, it occurs when the partition method always picks the greatest or smallest element, thus having to traverse the arrays a maximum amount of times. On the contrary, the best case is $O(n \cdot logn)$ which occurs when the partition method always picks the element in the middle of the array. Moreover, the average case is also $O(n \cdot logn)$ which is calculated by considering all the possible permutations of the array and the time that would take. Such mathematical analysis goes beyond the scope of this documentation. Now to the fun part. I talked a lot about the possible performances of Quick Sort, but, how does it actually work? I implemented it using two functions, namely *quickSort* and *partition*. Let's take a look at *quickSort*:

```
1  public static int quickSort(String[] magicList, int lowIndex, int
       highIndex) {
2
3      knuthShuffle(magicList);
4      if (lowIndex < highIndex) {
5
6        int partIndex = partition(magicList, lowIndex, highIndex);
7
8        quickSort(magicList, lowIndex, partIndex - 1);
9        quickSort(magicList, partIndex + 1, highIndex);
10
11     } // if statement
12
13     return quickSortComparisons;
14   } // Quick Sort
```

Similarly as in mergeSort, *quickSort* divides the array(s). In addition, it generates the index used for partitioning the arrays which I have called *partIndex* by making use of the main function of the algorithm *partition*. *Partition* takes care of the sorting as we divide the arrays, I implemented it as follows:

```
1  public static int partition(String[] magicList, int lowIndex, int
       highIndex) {
2
```

```
3
4     // Take the last element as the pivot value
5     String pivot = magicList[highIndex];
6     int idx = lowIndex - 1;
7
8     for (int j = lowIndex; j < highIndex; j++) {
9
10      if (magicList[j].compareToIgnoreCase(pivot) < 0) {
11
12        idx++;
13
14        String temp = magicList[idx];
15        magicList[idx] = magicList[j];
16        magicList[j] = temp;
17
18      } // if statement
19
20      quickSortComparisons++;
21
22    } // for loop
23
24    String temp2 = magicList[idx + 1];
25    magicList[idx + 1] = magicList[highIndex];
26    magicList[highIndex] = temp2;
27
28    return idx + 1;
29
30  } // partition
```

Partition takes in the String list, the index of the first element and the index of the last one. As previously mentioned, the first thing is to choose a pivot element. For this, I chose to use the last element of the String array as my pivot element. Now, the magic happens inside the for loop, which runs while $j =$ *lowIndex* is less than the index of the last element. If the element at j is smaller than our pivot element, the index is incremented and the elements smaller are placed to the left while the elements greater than it are placed to the right. Once the loop runs its course we perform a last swap, thus effectively sorting the entire array. Finally, we return the results *idx* variable plus 1 which is used my *quickSort* to instantiate the partition index mentioned earlier. *Partition* is a very efficient algorithm that runs in $O(n)$ time.

6 Table with results

The following table shows that number of comparisons made by each algorithm. It is worth noting that Insertion and Quick Sort vary each time the program is run because of the characteristics detailed under their respective sections.

Algorithm	**Comparisons**
Selection Sort	221445
Insertion Sort	106603
Merge Sort	2998
Quick Sort	6809

We can clearly see that Selection sort is the least efficient algorithm of all four, which is exactly what we would expect. Merge and Quick sort are the most efficient thanks to the recursive nature they possess. Due to the Divide and Conquer both Merge and Quick sort can effectively reduce run time.

7 Further thoughts

I encounter several challenges while working on this project, specially while figuring out how to count the comparisons of the recursive algorithms. Nevertheless, it was a great experience to implement these algorithms from scratch. After this much work I like to relax, and I imagine that after reading this extensive documentation you might want to do so too. So, here is a little joke that I hope you enjoy,