Assignment Two

Shannon Brady

shannon.brady2@Marist.edu

October 8, 2022

1 Selection Sort

1. Line 10: Traverse the array of unsorted elements (of size n) from 0 to n-2.
    a) The final element does not need to be traversed, given that it will already be in the correct position following the final comparison and swap.
2. Line 12-17: Look for the smallest element in the array by comparing all elements to array[i]. If there is a value less than array[i], take note of the index and increment the number of comparison.
3. Line 20-22: Swap the element at minIndex with array[i], regardless if a new minIndex is found. The next iteration of the outer for loop begins and the cycle is repeated accordingly.
4. Line 26: print –> method that prints sorted array to terminal
5. Line 32: printNumComparisons –> method that prints total number of comparisons to terminal
6. Running time analysis: The running time of Selection Sort is $O(n^2)$. This is due to the fact that for every iteration of the outer for loop, which must run roughly n times, there are n iterations within the inner loop. Since constant factors and O(1) statements are negligible, this algorithm maintains a $O(n^2)$ time complexity.

```
1  class SelectionSort {
2
3      int numComparisons = 0;
4
5      public void sort(String[] array) {
6          int n = array.length;
7          int minIndex = 0;
8
9          // no need to traverse entire array
10         // final element will already be in correct position
11         for (int i=0; i<n-2; i++) {
12             minIndex = i;
13             for (int j=i+1; j<n; j++) {
```

```
14                  // find the index of the 'smallest' element
15                  if (array[j].compareTo(array[minIndex]) < 0) {
16                      minIndex = j;
17                  }
18                  numComparisons++;
19              }
20              // swap smallest element with the current element
21              String temp = array[minIndex];
22              array[minIndex] = array[i];
23              array[i] = temp;
24          }
25      }
26
27      public void print(String[] array) {
28          for (int i=0; i<array.length; i++) {
29              System.out.println(array[i]);
30          }
31      }
32
33      public void printNumComparisons() {
34          System.out.println("\nSelection Sort\nNumber of Comparisons: " + numComparisons);
35      }
36  }
```

2 Insertion Sort

1. Line 8: Traverse the unsorted array of size n from array[1] to array[n]. Track the current element and the index of the previous element.
2. Line 13-17: Compare the current element to its predecessor. Increment the predecessor index until the current element is no longer less than the elements before. Increment the number of comparisons accordingly.
3. Line 20: Move the greater element one position forward to allow room for the swapped element.
4. Line 24: print –> method that prints sorted array to terminal
5. Line 30: printNumComparisons –> method that prints total number of comparisons to terminal
6. Running time analysis: Insertion Sort has a running time complexity of $O(n^2)$, given that for every element within the array, there can potentially be n number of swaps. Worst-case scenario would be if the array was in reverse order, as this would require every possible loop iteration. Conversely, if the array was already sorted, traversal of the inner loop wouldn't be necessary, resulting in order of n time.

```
1   class InsertionSort {
2
3       int numComparisons = 0;
4
5       public void sort(String[] array) {
6           int n = array.length;
7
8           for (int i=1; i<n; i++) {
9               String currElement= array[i];
10              int j = i-1;
```

```
11
12              // compare the current element to the previous element
13              while (j>= 0 && array[j].compareTo(currElement) > 0) {
14                  // keep comparing until current element is no longer less than elements before
15                  array[j+1] = array[j];
16                  j = j-1;
17                  numComparisons++;
18              }
19              // move greater elements one index forward
20              array[j+1] = currElement;
21          }
22      }
23
24      public void print(String[] array) {
25          for (int i=0; i<array.length; i++) {
26              System.out.println(array[i]);
27          }
28      }
29
30      public void printNumComparisons() {
31          System.out.println("\nInsertion Sort\nNumber of Comparisons: " + numComparisons);
32      }
33
34  }
```

3 Merge Sort

1. Line 8: Find the midpoint index of the unsorted array.
2. Line 11-12: Given that Merge Sort implements the divide and conquer problem solving methodology, the sort function must be called recursively to sort the entire array.
    a) This represents the divide portion of the algorithm. The unsorted array will be split into smaller unsorted sub-arrays until each sub-array contains only one item, which is effectively "sorted".
3. Line 15-29: Create and initialize two separate arrays, one for all elements to the left of the midpoint and one for all elements to the right of the midpoint.
4. Line 33: Call the merge function to combine the left and right sub-arrays.
5. Line 39-43: Keep track of index values for the left array, right array, and the entire array.
6. Line 45-51: If the left[i] is less than right[j], add left[i] to the merged array and increment the merged array index and the left index. Otherwise, add right[j] to the merged array and increment the indices accordingly. Increment the number of comparisons.
7. Line 54592: Add all remaining elements to the merged array, if there are any.
8. Line 62: print –> method that prints sorted array to terminal
9. Line 68: printNumComparisons –> method that prints total number of comparisons to terminal
10. Running time analysis: Merge sort has a time complexity of $O(nlogn)$, which can be expressed by the following recurrence relation: $T(n) = 2T(n/2)+O(n)$. The logn ($2T(n/2)$ portion represents the divide portion of the algorithm, where each sub-array is divided in half recursively. The n portion represents

the conquer portion. Since merging each sub-array ultimately results in traversing through the entire original array of n items, this takes n order of time.

```java
class MergeSort {

    int numComparisons = 0;

    public void sort(String[] array, int start, int end) {
        if (start < end) {
            // find midpoint
            int mid = start+(end-start)/2;

            // sort first and second halves
            sort(array, start, mid);
            sort(array, mid + 1, end);

            // create arrays for elements on either side of midpoint
            String[] left = new String[mid-start+1];
            String[] right = new String[end-mid];

            // left index
            int i=0;
            // right index
            int j=0;
            // initialize arrays to be merged
            while (i<left.length) {
                left[i] = array[start + i];
                i++;
            }
            while (j<right.length) {
                right[j] = array[mid + 1 + j];
                j++;
            }

            // merge the sorted halves
            merge(array, start, left, right);
        }
    }

    public void merge(String[] array, int start, String[] left, String[] right) {
        // index for first half of array
        int i = 0;
        // index for second half of array
        int j = 0;
        // initial index of merged subarray
        int k = start;

        while (i<left.length && j<right.length) {
            if (left[i].compareTo(right[j]) < 0) {
                array[k++] = left[i++];
            } else {
                array[k++] = right[j++];
            }
            numComparisons++;
```

```
52          }
53          // include remaining elements
54          while (i < left.length) {
55              array[k++] = left[i++];
56          }
57          while (j < right.length) {
58              array[k++] = right[j++];
59          }
60      }
61
62      public void print(String[] array) {
63          for (int i = 0; i < array.length; ++i) {
64              System.out.println(array[i]);
65          }
66      }
67
68      public void printNumComparisons() {
69          System.out.println("\nMerge Sort\nNumber of Comparisons: " + numComparisons);
70      }
71  }
```

4 QuickSort

1. Line 7-13: The sort method is the driver method of this Quicksort implementation. After finding a pivot index, the array is then recursively sorted on either side of the value.
2. Line 21, Line 40-51: Set a random pivot value by getting three random index values between the start and end indexes and find the median. Swap the element at the pivot index with the element at the end index (Line 54-59).
3. Line 22: Initialize the pivot element, which is now at the end of the array.
4. Line 25: Keep track of the smaller element
5. Line 27-34: Looping from start to end, swap the smaller element with the current element anytime array[j] is less than the pivot. Increment the number of comparisons.
6. Swap the end element (the pivot) with the next element and return that index.
7. Line 61: print –> method that prints sorted array to terminal
8. Line 67: printNumComparisons –> method that prints total number of comparisons to terminal
9. Running time analysis: The expected or average running time of Quicksort is $O(nlogn)$, which can be expressed by the following recurrence relation: $T(n) = 2T(n/2) + O(n)$. Though Merge Sort and Quicksort both implement divide and conquer, Quicksort combines the divide and conquer portions into one consilidated process, whereas Merge Sort separates them. In worst-case scenarios, the pivot element would either be the first or last element in the array, which can lead to $O(n^2)$ time complexity. This implementation of Quicksort essentially finds a random pivot element that is likely to be closer to the center of the array. This serves to limit the number of comparisons and maintain average running time.

```
1   import java.util.Random;
2
3   class QuickSort {
```

```
4
5       int numComparisons = 0;
6
7       public void sort(String[] array, int start, int end) {
8           if (start < end) {
9               int partitionIndex = partition(array, start, end);
10              sort(array, start, partitionIndex-1);
11              sort(array, partitionIndex+1, end);
12          }
13      }
14
15      int partition(String[] array, int start, int end) {
16          // places pivot in correct postion,
17          // all smaller elements to the left,
18          // and all greater elements to the right
19
20          // this function places the pivot (the median of three random values) at the end of the array
21          setRandomPivot(array, start, end);
22          String pivot = array[end];
23
24          // index of string that's 'smaller' (closer to top of alphabet)
25          int i = start-1;
26
27          for (int j=start; j<end; j++) {
28              // current element is smaller than pivot
29              if (array[j].compareTo(pivot) < 0) {
30                  i++;
31                  swap(array, i, j);
32                  numComparisons++;
33              }
34          }
35          // end is the pivot index
36          swap(array, i+1, end);
37          return i+1;
38      }
39
40      public void setRandomPivot(String[] array, int start, int end) {
41          // find three random index values between start and end
42          Random rand= new Random();
43          int rand1 = rand.nextInt(end-start)+start;
44          int rand2 = rand.nextInt(end-start)+start;
45          int rand3 = rand.nextInt(end-start)+start;
46
47          // calculate the median of the three values
48          // https://stackoverflow.com/questions/1582356/fastest-way-of-finding-the-middle-value-of-a-triple
49          int pivot = Math.max(Math.min(rand1, rand2), Math.min(Math.max(rand1, rand2), rand3));
50
51          swap(array, pivot, end);
52      }
53
54      public void swap(String[] array, int x, int y) {
55          // swaps two elements in an array
56          String temp = array[x];
57          array[x] = array[y];
```

```
58          array[y] = temp;
59      }
60
61      public void print(String[] array) {
62          for (int i = 0; i < array.length; ++i) {
63              System.out.println(array[i]);
64          }
65      }
66
67      public void printNumComparisons() {
68          System.out.println("\nQuicksort\nNumber of Comparisons: " + numComparisons);
69      }
70  }
```

4.1 Main Class and Results

1. Line 12-24: Initialize an array with all 666 magic items.
2. Line 75-86: shuffle –> A $O(n)$ shuffle routine based on the Knuth shuffle.
3. Line 47-51: Test Selection Sort
4. Line 54-58: Test Insertion Sort
5. Line 61-65: Test Merge Sort
6. Line 68-72: Test Quicksort

```
1   import java.io.File;
2   import java.io.FileNotFoundException;
3   import java.util.Random;
4   import java.util.Scanner;
5
6   class Assignment_2 {
7
8       public static Random randomGen = new Random();
9
10      public static void main(String[] args) {
11
12          // Get magic items from the input file
13          File file = new File("magicitems.txt");
14          Scanner myReader = null;
15          String[] magicItems = null;
16          int i = 0;
17          int numLines = 0;
18
19          try {
20              myReader = new Scanner(file);
21              while (myReader.hasNextLine()) {
22                  // count the number of lines in the file
23                  numLines++;
24                  myReader.nextLine();
25              }
26              myReader.close();
27
```

```
            // create magic items array
            magicItems = new String[numLines];

            myReader = new Scanner(file);
            while (myReader.hasNextLine()) {
                // initialize magic items array with magic items
                String data = myReader.nextLine();
                magicItems[i] = data.toLowerCase();
                i++;
            }
            myReader.close();
        } catch (FileNotFoundException e) {
            System.out.println("An error occurred.");
            e.printStackTrace();
        }

// ---------------------------------------------------------------------

        // 1. Selection Sort
        shuffle(magicItems);
        SelectionSort ssObj = new SelectionSort();
        ssObj.sort(magicItems);
        // ssObj.print(magicItems);
        ssObj.printNumComparisons();

        // 2. Insertion Sort
        shuffle(magicItems);
        InsertionSort isObj = new InsertionSort();
        isObj.sort(magicItems);
        // isObj.print(magicItems);
        isObj.printNumComparisons();

        // 3. Merge Sort
        shuffle(magicItems);
        MergeSort msObj = new MergeSort();
        msObj.sort(magicItems, 0, magicItems.length-1);
        // msObj.print(magicItems);
        msObj.printNumComparisons();

        // 4. Quicksort
        shuffle(magicItems);
        QuickSort qsObj = new QuickSort();
        qsObj.sort(magicItems, 0, magicItems.length-1);
        // qsObj.print(magicItems);
        qsObj.printNumComparisons();
    }

    public static void shuffle(String[] array) {
        int n = 0; // number of shuffled elements
        while (n < array.length-1) {
            n++;
            int randIndex = randomGen.nextInt(n); // select a random index value

            // swap the next array element with a random element
```

```
            String temp = array[n];
            array[n] = array[randIndex];
            array[randIndex] = temp;
        }
    }
}
```

4.2 Results

Algorithm	Number of Comparisons
Selection Sort	221444
Insertion Sort	110864
Merge Sort	5436
Quicksort	4077