Assignment Two

Veronica Longley

October 8, 2021

Our Goal: In this assignment we will attempt to sort the list of 666 magic items using 4 sorting methods: selection sort, insertion sort, merge sort, and quick sort. We will then compare the number of comparisons each sort completed to sort the list and determine why these numbers vary.

1 Selection Sort

```
1  static int selectionSort(String items[], final int TOTNUM)
2  {
3          int comparisons = 0;
4          //for loop to go through list and hold space for lowest
5          //value to be swapped to
6          for(int i = 0; i < TOTNUM - 1; i++)
7          {
8                  int minIndex = i;
9                  String minItem = items[i];
10                 String temp = "";
11
12                 //for loop to compare the rest of the element in the
13                 //array with the value from the for i loop
14                 for (int j = i + 1; j < TOTNUM; j++)
15                 {
16                         comparisons ++;
17                         //if items[j] is less than the current minimum
18                         //item in the list assign the minimum index to
19                         // be j
20                         if (items[j].compareToIgnoreCase(minItem) < 0)
21                         {
22
23                                 minItem = items[j];
24                                 minIndex= j;
25                         }//if
26                 }//for j
27
28                 //swap if not already in correct index
29                 if (minIndex != i)
30                 {       temp = items[minIndex];
31                 items[minIndex] = items[i];
32                 items[i] = temp;
33                 }//if
34         }//for i
35         return comparisons;
36 }//selection sort
```

First we will look at Selection Sort. This method first locates the smallest value in the array then swaps it with the first position in the array. This is then repeated for the second smallest value in the array which is the put in the second position of the array and so on. The first for loop, line 6, is used to go through the list one by one and hold the spot in the array for the lowest value to be swapped to. The actual swapping happens inside the second if in line 29 while the comparisons happen once the second for loop begins, line 14. Selection sort has a running time of n squared. After running this method on a list of 666 items we have 221445 comparisons which is in the same realm as n squared. We return this value to be used later.

2 Insertion Sort

```
1  static int insertion (String items[],final int TOTNUM )
2  {
3          int comparisons = 0;
4          //for loop to go through array from beginning to end
5          for(int i = 1; i < TOTNUM ; i++)
6          {
7                  String key = items[i];
8                  int j = i -1;
9  
10                 //while the item being added is greater than the end of the
11                 //already sorted items and j has not reached the start of
12                 //the array we slide values until j has found its position
13                 while((j > -1) && ( items[j] .compareToIgnoreCase(key) >0 ))
14                 {
15                         comparisons ++;
16                         //slide affected values to new location
17                         items [j+1] = items [j];
18             j--;
19                 }//for j
20 
21                 //increment the key
22                 items[j+1] = key;
23         }//for i
24         return comparisons;
25 
26 }//insertion
```

Insertion Sort works by creating a sub-list of size one, the first element in the list, and adding to the sub-list one by one until all items are back together. When an item is added the method scans through the sub-list until it finds the location for the new item, so the sub-list is always sorted. The key, defined in line 7, represents the last item in the sorted sub-list. The while loop, line 13, goes through the sorted sub-list and slides, line 17, values around until the appropriate open spot for the new item is found. For insertion sort we do not get a definite number of comparisons because it depends on the order of the shuffled list. If the while loop does not have to go though the sub-list as much because the list is partially ordered, the number of comparisons will be less. Insertion sort also has a running time of n squared and running the program a few times outputs values: 110734, 108522, 118723, 110127, and so on. This value is also returned to use later.

3 Merge Sort

```
1  static void mergeSort (String items[])
2  {
3  
4          //if the sub-lists are not yet legth one
5          if(items.length > 1)
6          {
7                  //divide in half again
8                  String[] left = new String [items.length/2];
9                  String[] right = new String [items.length- (items.length /2)];
10 
11                 //copy items into new arrays
12                  for(int i = 0; i< left.length; i++)
13                  {
14                          left[i]= items[i];
15                  }//for i
16                  for(int i = 0; i< right.length; i++)
17                  {
18                          right[i]= items[i+items.length/2];
19                  }//for i
20
21                  //recursively calls merge sort for each side
22                  mergeSort(left);
23                  mergeSort(right);
24                  merge(items, left, right);
25
26          }//if
27
28  }//mergeSort
29
30  static void merge(String[]items, String[]left, String[]right)
31  {
32          //merges sub-arrays
33          int x = 0;
34          int y = 0;
35
36          for(int j = 0; j < items.length; j++)
37          {
38                  mergeComparisons++;
39                  if(y >= right.length || (x < left.length &&
40                  left[x].compareToIgnoreCase(right[y]) < 0))
41                  {
42                          items[j]= left[x];
43                          x++;
44                  }//if
45                  else
46                  {
47                          items[j] = right[y];
48                          y++;
49                  }//else
50          }//for
51
52  }//merge
```

Next we have Merge Sort. This requires two methods, merge and Merge Sort. Merge, line 30, is responsible for merging the two sub-arrays, left and right, together. It compares elements of the sub-arrays placing the smaller ones first, line 39. Merge Sort, line 1, first checks if the sub-arrays are of length one yet, line 5. If they are not, it creates new arrays left and right, line 8 and 9, then copies the items into the appropriate half, lines 12-19, then calls merge sort on both the left and the right again until the sub-arrays are of length one, line 22-23. Finally it calls merge to put the sub-arrays back together, line 24. Merge Sort has a running time of n log base 2 (n). It will always have the same number of comparisons, so every time merge sort is run on a list of 666 items there will be 6302 comparisons. It is important to note that here the counter for comparisons must be a global variable as these functions are recursive, so it would otherwise reset each time the function was called.

4 Quick Sort

```
static int partition(String items[], int left, int right)
{
        random(items, left, right);
        String pivot = items[right];

        //indicates elements smaller than the pivot
        int x = (left - 1);
        for(int j = left; j < right; j++)
        {
                quickComparisons++;
                //if current position is smaller than pivot
                if(items[j].compareToIgnoreCase(pivot) < 0)
                {
                        //increase the number of elements smaller than pivot
                        x++;

                        //swap
                        String temp = items[x];
            items[x] = items[j];
            items[j] = temp;

                }//if

        }//for

        //swap remaining upper elements
        String temp2 = items[x+1];
    items[x+1] = items[right];
    items[right] = temp2;

    return x+1;
}//partition

static void quicksort (String items[], int left, int right)
{

        if(left < right)
        {
                //index of pivot
                int piv = partition(items, left, right);
                //sort elements above and below the pivot
                //recursion
                quicksort(items, left, piv-1);
                quicksort(items, piv+1, right);

        }//if
}//quicksort

static void random(String items [], int left, int right)
{
        Random random = new Random();
        int pivot = random.nextInt(right - left)+ left;
        String temp = items[pivot];
        items[pivot]= items[right];
        items[right] = temp;
}//random
```

Quick sort also requires two methods, partition and quick sort. Partition randomly selects the pivot using the random method shown at line 49. It then uses a for loop, line 8, to check if the current position in the list is smaller than the pivot. If it is, we increase x, line 15, which counts the number of elements smaller than the pivot. We then move those elements in front of the pivot, lines 18-20, and move the remaining elements to spaces after the pivot, line 27-29. The quick sort method, line 34, checks if the method needs to continue sorting then calls partition to create the pivot and recursively calls quick sort on both sides of the pivot. This is done until all sub-arrays broken apart by pivots are of length one. Here we must again use a global variable to count the number of comparisons as the recursive nature of quick sort would otherwise continue to reset this value. Quick sort does not have a specific number of comparisons for an array with 666 items because it depends on the pivot value chosen. If the pivot value is always the max or min of the list, the running time of quick sort will be n squared. On average though, this does not happen, so quick sort tends to be n log base 2 (n) with a few comparison counters of 6415, 7059, 6775, 7536 and so on.

5 Shuffle

```java
static void shuffle (String items [], final int TOTNUM)
{
        Random random = new Random();
        String temp = "";
        random.nextInt();
        for (int i = 0; i < TOTNUM; i++)
        {
                int change = i + random.nextInt(TOTNUM - i);
                temp = items[i];
        items[i] = items[change];
        items[change] = temp;
        }//for
}//shuffle
```

This is an example of a Knuth, or Fisher-Yates, Shuffle. This method puts all indexes of the array in a hat essentially and randomly permutes elements until none are remaining. This method also uses the random method shown in the quick sort section to 'draw from the hat'. It has a running time of n as it must only go through the list once to shuffle the elements.

6 ...In Main Method

```java
public class main
{
        public static int mergeComparisons;
        public static int quickComparisons;
static Scanner keyboard = new Scanner(System.in);
        public static void main(String[] args)
        {
                // TODO Auto-generated method stub
                // TODO Auto-generated method stub
```

```
            //initialize variables used to read in items from list
            String fileName = "";
            final int NUMOFITEMS = 666;
            String [] magicItems = new String [NUMOFITEMS];
            String theitem = null;
            fileName = "magicitems";
            //input item names from file and store in array
            try
                    {
                    Scanner readFile = new Scanner (new File (fileName));
                    int i = 0;
                    while(readFile.hasNextLine())
                    {
                            theitem = readFile.nextLine();
                            magicItems[i] = theitem;
                            i++;
                    }//while
                    readFile.close();
                    }//try
            catch(FileNotFoundException ex)
                    {
                    System.out.println("Failed to find file: "+ fileName);
                    }//catch
            catch(InputMismatchException ex)
                    {
                    System.out.println("Type mismatch");
                    System.out.println(ex.getMessage());
                    }//catch
            catch(NumberFormatException ex)
                    {
                    System.out.println("Fail to convert String into an integer");
                    System.out.println(ex.getMessage());
                    }//catch
            catch(NullPointerException ex)
                    {
                    System.out.println("Null pointer exception. ");
                    System.out.println(ex.getMessage());
                    }//catch
            catch(Exception ex)
                    {
                    System.out.println("Oops, something went wrong. ");
                    ex.printStackTrace();
                    }//catch
            int ssnum = 0;
            int isnum = 0;
            ssnum = selectionSort(magicItems, NUMOFITEMS);
            shuffle(magicItems, NUMOFITEMS);
            isnum = insertion(magicItems, NUMOFITEMS);
            shuffle(magicItems, NUMOFITEMS);
            mergeSort(magicItems);
            shuffle(magicItems, NUMOFITEMS);
            quicksort(magicItems, 0, NUMOFITEMS-1);

        System.out.println("Selection sort does a total of " + ssnum+
        " comparisons. ");
        System.out.println("Insertion sort does a total of " + isnum+
        "  comparisons. ");
        System.out.println("Merge sort does a total of " + mergeComparisons+
68              " comparisons. ");
69              System.out.println("Quick sort does a total of " +quickComparisons+
70              " comparisons. ");
```

The main method here is used to implement each of the above methods. We first define a global variable for the number of comparisons for merge sort and quick sort as these are the recursive methods. We then read in the file with 666 magic items. Next we initialize the counter for selection sort and insertion sort, line 53 and 54 respectively. Then we call each of the sorting methods being sure to shuffle the list in between each sorting. Lastly we print out the number of comparisons for each sort as stored in the earlier mentioned variables.

7 Conclusion

The below table shows the appropriate name, asymptomatic running time and one, or a few if appropriate, outputs from running our sorts.

Sort	Running Time	Results
Selection Sort	n^2	221445
Insertion Sort	n^2	109337, 114395, 107572...
Merge Sort	n log base 2 (n)	6302
Quick Sort	n log base 2 (n) up to n^2	6633, 6426, 7018...

In conclusion, I would like to note a few important trends amount the sorts. We see that selection sort and insertion sort have the same running time though insertion sort does much less comparisons than insertion sort. This is because it is not necessarily forced to run though the entire list when sorting every single element, it must only find its place in the sub-array. For this reason, insertion sort is likely the better option over selection sort as it is more efficient in that sense. In terms of recursive sorts, merge sort tends to be the safe call with a consistent running time of n log base 2 of n while quick sort on average is the same; however depending on the chosen pivots, may degrade to n squared, though this is quite rare and unlucky. Quick sort also should never be better then merge sort, making merge sort the safest choice amount the four.