CMPT 435 - Fall 2021 - Dr. Labouseur

Assignment Two – LaTeX Sorts

Tyler Vultaggio

October 2021

1 Sort

For this assignment there were a number of sorts used to show the different amounts of time it takes for specific sorts to run. These sorts all had different run times and numbers of comparisons. A chart of my specific comparisons is listed below.

2 Comparisons

Sort Name Comparisons Asymptotic Running time:
Selection Sort 221445 (n^2)
Insertion Sort 114369 (n^2)
Merge Sort 5446 (n log n)
Quick Sort 7266 (n log n)

3 Selection Sort

For selection sort, the absolute best and worst case scenarios (Ω() or O()) are both actually n 2 . This is because the internal loop will always run through the whole list for as many items as are in the list, which ends up being the number of items times the number of items, or n 2 .

3.1 Selection Sort

```
1 public class SelectionSort
2 {
3         public static String[] sSort(String[] SelectionArray)
```

```
4           {
5                   int n = SelectionArray.length;
6                   int compare;
7                   int min;
8                   int countCompare = 0;
9           // One by one move boundary of unsorted subarray
10          for (int i = 0; i < n-1; i++)
11          {
12              // Find the minimum element in unsorted array
13              min = i;
14              for (int j = i+1; j < n; j++)
15              {
16                  compare = SelectionArray[j].compareTo(SelectionArray[min]);
17                  countCompare++;
18                  if (compare < 0)
19                  {
20                                          min = j;
21                  }
22              }
23
24              // Swap the min element into the i index
25              String temp = SelectionArray[min];
26              SelectionArray[min] = SelectionArray[i];
27              SelectionArray[i] = temp;
28          }
29          System.out.println(countCompare);
30                  return SelectionArray;
31          }
32
33          /*
34           * This is a function so that I can see the sorted array printed.
35           * Thus I know that it is working properly.
36           */
37
38          public static void printArray(String[] printableArray)
39          {
40                  for(int i = 0; i <      printableArray.length; i++)
41                  {
42                          System.out.println(     printableArray[i]);
43                  }
44          }
45
46          public static void main(String[] args) throws FileNotFoundException
47          {
48                  String[] SelectionArray = new String[666];
49                  SelectionArray = myFileReader.fileArray();
50                  SelectionArray = sSort(SelectionArray);
51                  //printArray(SelectionArray);
52
53
54          }
55
56  }
```

4 Insertion Sort

For insertion sort, there is a bit more optimization. The sort has the ability to check if a given item in the list is in sorted order, which means that its best case is Ω(n) rather than a constant n 2 . Even though its best case is n, the average and worst cases stay at n 2 , which means it is not all that much faster, but it can check for items that are already sorted meaning it can cut down on more time.

4.1 Insertion Sort

```
1  public class InsertionSort
2  {
3          public static String[] iSort(String[] InsertionArray)
4          {
5                  int n = InsertionArray.length;
6                  String key;
7                  int countCompare = 0;
8          for (int i = 1; i < n; ++i)
9          {
10             key = InsertionArray[i];
11             int j = i - 1;
12
13             /* Move elements of arr[0..i-1], that are
14                greater than key, to one position ahead
15                of their current position */
16             while (j >= 0 && InsertionArray[j].compareTo(key) >= 0)
17             {
18                 InsertionArray[j + 1] = InsertionArray[j];
19                 j = j - 1;
20                 countCompare++;
21             }
22             InsertionArray[j + 1] = key;
23         }
24                 System.out.println(countCompare);
25                 return InsertionArray;
26         }
27
28         /*
29          * This is a function so that I can see the sorted array printed.
30          * Thus I know that it is working properly.
31          */
32
33         public static void printArray(String[] printableArray)
34         {
35                 for(int i = 0; i <      printableArray.length; i++)
36                 {
37                         System.out.println(     printableArray[i]);
38                 }
39         }
40
41         public static void main(String[] args) throws FileNotFoundException
42         {
43                 String[] InsertionArray = new String[666];
44                 InsertionArray = myFileReader.fileArray();
```

```
                InsertionArray = iSort(InsertionArray);
                //printArray(InsertionArray);

        }

}
```

5 Merge Sort

Merge sort is a distinct step up in terms of speed and comparisons, running at a blazing hot (relatively) pace of n log n no matter what case the code is running in. Due to the nature of divide and conquer, any problem no matter what case can be done with the same complexity.

5.1 Merge Sort

```
public class MergeSort
{
        //I made this a global variable since I needed to keep the changes to it everytime mer
        public static int countCompare = 0;

        public static void merge(String[] mergeArray, int l, int m, int r)
    {
        // Find sizes of two subarrays to be merged
        int n1 = m - l + 1;
        int n2 = r - m;


        /* Create temp arrays */
        String L[] = new String[n1];
        String R[] = new String[n2];

        /*Copy data to temp arrays*/
        for (int i = 0; i < n1; ++i)
            L[i] = mergeArray[l + i];
        for (int j = 0; j < n2; ++j)
            R[j] = mergeArray[m + 1 + j];

        /* Merge the temp arrays */

        // Initial indexes of first and second subarrays
        int i = 0, j = 0;

        // Initial index of merged subarray array
        int k = l;
        while (i < n1 && j < n2)
        {
            if (L[i].compareTo(R[j]) <= 0)
            {
                mergeArray[k] = L[i];
                i++;
                countCompare++;
            }
```

```
38              else
39              {
40                  mergeArray[k] = R[j];
41                  j++;
42                  countCompare++;
43              }
44              k++;
45          }
46
47          /* Copy remaining elements of L[] if any */
48          while (i < n1)
49          {
50              mergeArray[k] = L[i];
51              i++;
52              k++;
53          }
54
55          /* Copy remaining elements of R[] if any */
56          while (j < n2)
57          {
58              mergeArray[k] = R[j];
59              j++;
60              k++;
61          }
62          //System.out.println(countCompare);
63      }
64
65          public static String[] mSort(String MergeArray[], int l, int r)
66      {
67          if (l < r) {
68              // Find the middle point
69              int m =l+ (r-l)/2;
70
71              // Sort first and second halves
72              mSort(MergeArray, l, m);
73              mSort(MergeArray, m + 1, r);
74
75              // Merge the sorted halves
76              merge(MergeArray, l, m, r);
77          }
78
79          return MergeArray;
80      }
81
82          /*
83           * This is a function so that I can see the sorted array printed.
84           * Thus I know that it is working properly.
85           */
86
87          public static void printArray(String[] printableArray)
88          {
89                  for(int i = 0; i <      printableArray.length; i++)
90                  {
91                          System.out.println(     printableArray[i]);
92                  }
93          }
94
```

```
95          public static void main(String[] args) throws FileNotFoundException
96          {
97                  String[] MergeArray = new String[666];
98                  MergeArray = myFileReader.fileArray();
99                  MergeArray = mSort(MergeArray, 0, MergeArray.length-1);
100                 System.out.println(countCompare);
101                 //printArray(MergeArray);
102         }
103
104 }
```

6 Quick Sort

Quick sort is exactly like its name suggests: quick. It utilizes a pivot to make its divide and conquer in place allowing for less data usage than merge sort and usually in comparable time. But this comes at a costs: a chance of loosing the speed lottery. The worst case of Quick Sort is a stunningly slow n 2 , while its average and best case scenarios are at the speedy pace of n log n.

6.1 Quick Sort

```
1  public class QuickSort
2  {
3
4          public static int countCompare = 0;
5          // A utility function to swap two elements
6          static void swap(String[] QuickArray, int i, int j)
7          {
8              String temp = QuickArray[i];
9              QuickArray[i] = QuickArray[j];
10             QuickArray[j] = temp;
11         }
12
13         public static int randomPivot(String[] QuickArray)
14         {
15                 Random ran = new Random();
16                 int ran1 = ran.nextInt(QuickArray.length);
17                 int ran2 = ran.nextInt(QuickArray.length);
18                 int ran3 = ran.nextInt(QuickArray.length);
19                 //System.out.println(ran1 + " " + ran2 + " " + ran3);
20                 if(ran1 <= ran2 && ran1 <= ran3)
21                 {
22                         return ran1;
23                 }
24                 else if(ran2 <= ran1 && ran2 <= ran3)
25                 {
26                         return ran2;
27                 }
28                 else
29                 {
30                         return ran3;
31                 }
```

```
32
33          }
34
35
36          /*
37           * This function takes last element as pivot, places
38           * the pivot element at its correct position in sorted
39           * array, and places all smaller (smaller than pivot)
40           * to left of pivot and all greater elements to right of pivot
41           */
42
43          static int partition(String[] QuickArray, int low, int high)
44          {
45                  //System.out.println(randomPivot(QuickArray));
46
47              // pivot
48              String pivot = QuickArray[high];
49
50              // Index of smaller element and
51              // indicates the right position
52              // of pivot found so far
53              int i = (low - 1);
54
55              for(int j = low; j <= high - 1; j++)
56              {
57
58                  // If current element is smaller
59                  // than the pivot
60                  countCompare++;
61                  if (QuickArray[j].compareTo(pivot) < 0)
62                  {
63
64                      // Increment index of
65                      // smaller element
66                      i++;
67                      swap(QuickArray, i, j);
68
69                  }
70              }
71              swap(QuickArray, i + 1, high);
72              return (i + 1);
73          }
74
75          /* The main function that implements QuickSort
76                    QuickArray[] --> Array to be sorted,
77                    start --> Starting index,
78                    end --> Ending index
79           */
80          static String[] qSort(String[] QuickArray, int start, int end)
81          {
82              if (start < end)
83              {
84
85                  // pi is partitioning index, QuickArray[p]
86                  // is now at right place
87                  int part = partition(QuickArray, start, end);
88
```

```
                // Separately sort elements before
                // partition and after partition
                qSort(QuickArray, start, part - 1);
                qSort(QuickArray, part + 1, end);

            }

            return QuickArray;
        }

        /*
         * This is a function so that I can see the sorted array printed.
         * Thus I know that it is working properly.
         */

        public static void printArray(String[] printableArray)
        {
                for(int i = 0; i <      printableArray.length; i++)
                {
                        System.out.println(     printableArray[i]);
                }
        }

        public static void main(String[] args) throws FileNotFoundException
        {
                String[] QuickArray = new String[666];
                QuickArray = myFileReader.fileArray();
                QuickArray = qSort(QuickArray, 0, QuickArray.length-1);
                System.out.println(countCompare);
                //printArray(QuickArray);
        }

}
```

7 fileReader

The fileReader part takes a text file in this case magicitems.txt and it takes a scanner and a reader and turns each line into a string and puts it into an array once the string is in the array we clean up the string by removing all of the spaces, number, and special characters in the sting.

7.1 fileReader Sort

```
public class myFileReader
{
        public static String[] fileArray() throws FileNotFoundException
        {
                Scanner scanner = new Scanner(new File("magicitems.txt"));
                String[] myArray = new String[666];
                int index = 0;

                while(scanner.hasNextLine())
                {
```

```
                        myArray[index] = scanner.nextLine();
                        myArray[index] = myArray[index].replaceAll("[0-9+,_()/.]", "");
                        myArray[index] = myArray[index].replaceAll("\\s", "");
                        myArray[index] = myArray[index].replaceAll("'", "");
                        myArray[index] = myArray[index].replaceAll("-", "");
                        myArray[index] = myArray[index].toLowerCase();
                        myArray[index] = myArray[index].substring(0,1).toUpperCase() + myArray
                        index = index + 1;
                }
                scanner.close();

                return myArray;
        }

        public static void main(String[] args) throws IOException
        {

        }

}
```