Lab Two

Joseph McDonough

Joseph.McDonough@marist.edu

October 2, 2020

1 Results

A file, titled magicitems, is a text file that has the name of 666 different items. The items appear in the text file in a random order, such that it is not presorted in any way.

Upon running the string of items through four different sort types (selection, insertion, merge, and quick sort) the amount of comparisons were recorded. A comparison is when one item in the list is directly put up against another and evaluated such that the sort can do its task to produce an ascending lexicographical order.

Table 1.1: The 4 Sorts

Sort Type	**Comparison Count**	**Asymptotic Run-Time**
Selection	221,445	n^2
Insertion	114,309	n^2
Merge	6,302	$n(\log_2 n)$
Quick	˜ 7,000	$n(\log_2 n)$

Following an analysis of Big O, the comparison count for all the sorts are within the order of magnitude and can be deemed appropriate results for a list of size 666.

2 Asymptotic Analysis

2.1 Selection Sort

```
71 int comparisonCount = 0;
72 for(int i = 0; i < itemList.size(); i++)
73 {
74     int smallestPos = i;
75     for(int j = i + 1; j < itemList.size(); j++)
76     {
77         comparisonCount++;
78         if(itemList.get(smallestPos).compareToIgnoreCase(itemList.get(j)) > 0)
79             smallestPos = j;
80     }
81     swap(itemList, i, smallestPos);
82 }
83 return comparisonCount;
```

Amongst the four types of sorts ran, selection sort underwent the most comparisons, and this is expected. Selection sort does not alter or learn from the data. It uses nested for-loops (lines 72 and 75) and ultimately compares each item against every other. The first for-loop (line 72) starts with the very first item in the list and compares it to the remainder of the list (in this case 665 other items - line 75). That first element is checked to see if there is a lower element present in the list (line 78). If a lower element is found, it is swapped with the first (line 81). Then selection sort moves to the second items and repeats.

Selection sort fails to keep track of any element except which is the lowest at any given point in time. This lack of element tracking results in the same items getting passed over multiple times. If the sort is being used to put strings in ascending lexicographic order (as in this assignment), and the list was sorted except for the first element, it would not matter. A randomly sorted list (as in this assignment) has the exact same comparison count as an almost sorted list as long as the size is the same. This is also why the asymptotic run-time of n^2 is also the expected run-time of a selection sort.

Ultimately this is the cause of selection sort taking the longest of all evaluated sorts. Due to the use of nested for-loops, the asymptotic run-time of n^2 makes sense. A nested loop that peruses through an entire list would take, when factored out, $\frac{n^2+n}{2}$ time and resolves to be just n^2 after dropping the lesser orders.

2.2 Insertion Sort

```
 88 int comparisonCount = 0;
 89 for(int i = 1; i < itemList.size(); i++)
 90 {
 91     String currItem = itemList.get(i);
 92     int j = i-1;
 93     while(j >= 0 && (itemList.get(j).compareToIgnoreCase(currItem))>0)
 94     {
 95         comparisonCount++;
 96         swap(itemList, j+1, j);
 97         j--;
 98     }
 99     itemList.set(j+1, currItem);
100 }
```

Insertion sort is similar to selection sort in the sense that it utilizes two loops (lines 89 and 93), but is different in way it sort of adapts to the elements presented to it. This sort swaps elements as they appear. Once the insertion sort comes across a value that is less than the current value, it performs the swap (line 96). Then after swapping, it keeps working its way towards to the beginning of the list.

The worst-case scenario, that is when the asymptotic run-time of n^2 is reached, is when the list in sorted in reverse. This is because each element then would be swapped as it goes through the list due to every element it faces being lower. In that case, it works like a selection sort, pitting each element against each other. But because of the adaptive ability, insertion sort has the ability to be quicker, as seen in this assignment and a randomly sorted list. Insertion sort's comparison counts differs on the way the elements are previously arranged. On a list that is nearly sorted, insertion sort would work faster on the same list that was randomly sorted, meaning its understandable that its Big O(n^2) is truly a worse case as it could in fact perform better than it.

Like selection sort, the use of two loops means the possibility that each item could be compared against every other, which is why the asymptotic run-time is n^2.

2.3 Merge Sort

```
if (itemsSize > 1)
{
    int midpoint = itemsSize / 2;
    List<String> leftItems = Arrays.asList(new String[midpoint]);
    List<String> rightItems = Arrays.asList(new String[itemsSize - midpoint]);

    for (int i = 0; i < midpoint; i++)
        leftItems.set(i, items.get(i));

    for (int j = midpoint; j < itemsSize; j++)
        rightItems.set(j - midpoint, items.get(j));

    mergeSort(leftItems, midpoint);
    mergeSort(rightItems, itemsSize - midpoint);

    merge(items, leftItems, rightItems, midpoint, itemsSize - midpoint);
}
```

Merge sort's helper function - merge

```
131 int leftInc = 0;
132 int rightInc = 0;
133 int tempInc = 0;
134
135 while (leftInc < left && rightInc < right)
136 {
137     mergeComparisonCount++;
138
139     if (leftItems.get(leftInc).compareToIgnoreCase(rightItems.get(rightInc)) < 0)
140         temp.set(tempInc++, leftItems.get(leftInc++));
141     else
142         temp.set(tempInc++, rightItems.get(rightInc++));
143 }
144
145 while (leftInc < left)
146 {
147     mergeComparisonCount++;
148     temp.set(tempInc++, leftItems.get(leftInc++));
149 }
150
151 while (rightInc < right)
152 {
153     mergeComparisonCount++;
154     temp.set(tempInc++, rightItems.get(rightInc++));
155 }
```

Merge sort is the first sort in the group that has a different asymptotic run-time, which is $n(\log_2 n)$. Merge sort does not follow the same outline as the previous two with its nested loops. Instead merge sort abides by the divide and conquer strategy. That is, it first divides the list into sub-lists of size 1 (lines 110-118). It takes the starting list, and creates two sub-lists, a left and right half (lines 110-112). The left and right halves get filled and the process repeats on both halves until every single element is in its own array (that is of size 1 - itself). Once there are only sub-lists of size 1, the divide stage is complete and each list is considered sorted. The divide stage works in $\log_2 n$ time. The list gets divided in half each time, and $\log_2 n$ is the appropriate equation to demonstrate that where n is the amount items in the list

The helper function, merge, is what does the conquering. As the list is built back up to its original size, it is sorting the lists that are being fused together. With this technique, the sorting begins with the first and ends with the last conquer. The sorting is not saved until the very end, which means merge sort can sort in less comparisons than selection and insertion. Due to the sorting being done on the way up, and in a recursive manner, it is the origin of the n run-time as n is the list size and all elements need to be fused back together.

As merge sort is a divide and conquer process, the run-time of the two process ought to be multiplied together, resulting in an overall asymptotic run-time of $n(\log_2 n)$. Merge sort follows a recursive algorithm, much like quick sort, and is predictable in its run-time (like selection sort) because no matter how well or poorly the list was sorted prior to the algorithm, it's run-time will always be of $n(\log_2 n)$.

2.4 Quick Sort

```
161  if (items.size() > 1)
162  {
163      Random rand = new Random();
164      int pivotNumber = rand.nextInt(items.size());
165      String pivotItem = items.get(pivotNumber);
166
167      swap(items, 0, pivotNumber);
168      int k = 0;
169
170      for (int i = 1; i < items.size(); i++)
171      {
172          quickComparisonCount++;
173          if (items.get(k).compareToIgnoreCase(items.get(i)) > 0)
174          {
175              swap(items, k, i);
176              k++;
177              swap(items, k, i);
178
179          }
180      }
181
182      quickSort(items.subList(0, k));
183      quickSort(items.subList(k + 1, items.size()));
184  }
```

Divide and conquer with the use of recursion is the process the quick sort algorithm follows. Unlike merge sort, the divide and conquer is done at the same time, and is not done by splitting the list in half until there is nothing left to split. Quick sort chooses a pivot point in the list. This code was written in Java and makes use of the Random class to generate a random number to be selected as the pivot number (line 163-164). The pivot element is then swapped with the first element in the list such that the pivot element is at the front. After the pivot element is selected, the items that are of a lower lexicographical (that is closer to "A") are put to the left of the pivot (lines 173-177). Since the pivot element starts at element 0, it is essentially being pushed back one place each time a lower element appears and is placed before it. After all the items are compared against the pivot, the pivot element is considered to be in its final resting place and the recursive call repeats the process on both sides on the pivot.

The same process as stated above happens to the left and right half of the list and this goes on until every sub-list is of size 1. Since the sorting (conquering) is going on as the list is being broken down (dividing), when each list is of size 1, the process is done. No need to build back up to a complete list. Because of divide and conquer and its similarities to merge sort, its Big O is also $n(\log_2 n)$. The "dividing" of the lists until there is just one element per list of run-time $\log_2 n$ and the "conquering" process and a recursion use results in a run-time n and an overall asymptotic run-time of $n(\log_2 n)$.