Lab Three

Joseph McDonough

Joseph.McDonough1@marist.edu

October 30, 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 a sort, a random selection of 42 items (the same 42 for each search) were selected and searched for using linear search, binary search, and the use of a hashing table.

Table 1.1: The 3 Searches

Search Type	**Comparison Count**	**Asymptotic Run-Time**
Linear	333	n
Binary	8	$\log_2 n$
Hashing	2	1

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

2 Asymptotic Analysis

2.1 Linear Search

```
 85  int i = 0;
 86  boolean found = false;
 87  int linearSortCount = 0;
 88
 89  while(i < myList.size() && !found)
 90  {
 91      linearSortCount++;
 92      if(myList.get(i).equalsIgnoreCase(item))
 93          found = true;
 94
 95      i++;
 96  }
 97
 98  Assignment3 sortedPair = new Assignment3(item, linearSortCount);
 99  linearResultsGroup.add(sortedPair);
100  return linearResultsGroup;
```

Linear search starts at the first element of the list and goes through the entire list (line 89) until either the item is found (line 92) or the entire list has been exhausted. The search is performed on a sorted list so if the desired item is first alphabetically, it will be found in one pass, but if the element is the last alphabetically, the sort will have to go through the entire list (in this case 666 times) before it finds it. And if the item is not in the list at all, it will still have to go through the whole list until the end is reached to truly return item not found. In theory, there could also be a check to see if the item should have appeared already in a sorted list (i.e. it would be known item "alpha" is not in the list if the current element is "bravo") but that is not a truly linear search.

While running the search multiple times for differing sets of 42 random items, the average search count was always in or around the range of 320 - 350 searches before the item was found. This makes sense as the average run-time for a list of size 666 would be about 333, that is half of the elements in the list.

Therefore, the expected run-time is $n/2$ but in calculating Big-Oh notation, constants are dropped, resulting in just $O(n)$, which is what truly happened in this scenario.

2.2 Binary Sort

```
119  int middleIndex = (int)((startIndex + stopIndex) / 2);
120  binarySearchCount++;
121
122  if(startIndex > stopIndex)
123      System.out.print("ITEM NOT FOUND");
124  else if(magicItems.get(middleIndex).equalsIgnoreCase(item))
125  {
126      binaryResultsGroup.add(new Assignment3(item, binarySearchCount));
127      binarySearchCount = 0;
128  }
129  else if(magicItems.get(middleIndex).compareToIgnoreCase(item) > 0)
130      binarySearch(magicItems, item, startIndex, middleIndex - 1, binaryResultsGroup);
131  else
132      binarySearch(magicItems, item, middleIndex + 1, stopIndex, binaryResultsGroup);
133
134  return binaryResultsGroup;
```

Binary search starts at the middle of the entire list. It is important to note that binary search is only effective on a sorted list. If the desired element so happens to be at the middle of the list, the search is complete (line 124). If not, it is then determined if the item is closer to the top or bottom of the alphabet. If it is closer to the top (i.e. "A"), then the function is ran again on the part of the list that starts at the beginning and ends one less than the middle element (line 129). The same occurs if the item is closer to the bottom (i.e. "Z"). The binary search function is ran on the remainder of the list that is one greater than the midpoint all the way to the end. This recursive function works until the element is ultimately the midpoint and is found, or there are no more midpoints to be had and the element is not present.

After running binary search on numerous sets of 42 randomly selected items, the average count was 8 nearly every time. This is expected as the average run-time for a binary search is $\log_2 n$, and for a list of size 666, that comes out to be 9.3. The result of 8 is within an order of magnitude of 9.3 and is an acceptable outcome.

The equation of $\log_2 n$ makes sense because binary search follows the same procedure as a divide and conquer algorithm (e.g. merge sort). It recursively is splitting the list in half, but there is no need to build back up. $\log_2 n$ is an accurate equation to determine the number of divides that could happen on a given list of size n.

2.3 Hashing Search

```
203  int count = 0;
204  boolean found = false;
205  int location = makeHashCode(item);
206  Queue hashQueue = myHashTable.get(location);
207
208  if(hashQueue.isEmpty())
209      count = -1;
210  else
211  {
212      Node nodeEval = hashQueue.queueFirst;
213      while(!found)
214      {
215          if(nodeEval.data.equalsIgnoreCase(item))
216              found = true;
217          else
218              nodeEval = nodeEval.reference;
219          count++;
220      }
221  }
222  if(!found)
223      count = -1;
224
225  return count;
```

A search performed using a hashing table performed the fasted out of the three sorts, and that was mainly due to the hashing algorithm not allowing for many collisions. The hashing search itself works by taking the hash of the item that is being searched for (line 205). The hash table uses chaining to handle collisions and the chaining is done with the use of a queue. Using the hash of the searched item, the function goes to that index of the hash table and takes the queue that resides there (line 206). If the queue is empty, then there is no way the item is present and there is no need to look any further (line 208). Otherwise, the function goes through the queue evaluating each node until the item is found. If the item is found, the while loop is exited (line 215). If the item is never found, it's the same as having an empty queue, a -1 is returned. Since a -1 search could never be achieved, this is a way of letting the user know the item is not found.

```
166  str = str.toUpperCase();
167  int length = str.length();
168  int letterTotal = 0;
169
170
171  for (int i = 0; i < length; i++)
172  {
173      char thisLetter = str.charAt(i);
174      int thisValue = (int)thisLetter;
175      letterTotal = letterTotal + thisValue;
176  }
177
178
179  int hashCode = (letterTotal * 1) modulo HASH\_TABLE\_SIZE;
180
181  return count;
```

The hashing algorithm sets all the letters to uppercase (line 166) and then adds the ASCII number for all those characters (line 175). The hash table size is predetermined to be of size 250, so the modulo operator is performed on the sum of the ASCII values and the 250 table size (line 179). The result is floored and is the hash value. This algorithm with this list of 666 items does a good enough job to keep collisions to a low and spreads the elements across most of the possible 250 hash value outcomes.

In most scenarios, a hash search works in constant time. It takes O(1) to perform the hash function on the desired item, then it takes another O(1) to get to the proper index in the hash table. Then the function has to go through the chain, which could technically be O(n) for a situation where the hashing algorithm resulted in all the values having the same value, but does not often happen. As in the case with this scenario, it would take a maximum of around 5 searches before the item was found. For a carefully selected algorithm to match the expected input, the expected search time on the chain is O(1). As a result, hashing is expected to take O(1) + O(1) + O(1). This results in an asymptotic run-time of O(1), which was the result and makes it the fastest of the three.