Algorithms Assignment 2

Thomas Breaton

October 2023

1 Main Function

```java
import java.io.File;
import java.io.FileNotFoundException;
import java.util.Scanner;
import java.util.Arrays;
import java.util.Random;
import java.util.ArrayList;
import java.text.DecimalFormat;

public class Main2 {

    private static final int HASH_TABLE_SIZE = 250;
    private static final int LINES_IN_FILE = 666;
    private static final DecimalFormat df = new DecimalFormat(pattern:"0.00");

    Run | Debug
    public static void main(String[] args) throws FileNotFoundException {
        File magicItemsFile = new File(pathname:"magicitems.txt");
        String[] magicItemsArray = fileToArray(magicItemsFile);
        String[] randomItems = randomItems(magicItemsArray);
        quickSort(magicItemsArray, lower:0, magicItemsArray.length - 1);
        int linearSeachComparisons = linearSearch(magicItemsArray, randomItems);
        System.out.println("Linear search comparisons: " + linearSeachComparisons);
        int binarySearchComparisons = binarySearch(magicItemsArray, randomItems);
        System.out.println("Binary search comparisons: " + binarySearchComparisons);

        int[] hashValues = new int[LINES_IN_FILE];
        int hashCode = 0;
        for (int i = 0; i < LINES_IN_FILE; i++) {
            // System.out.print(i);
            // System.out.print(". " + magicItemsArray[i] + " - ");
            hashCode = makeHashCode(magicItemsArray[i]);
            // System.out.format("%03d%n", hashCode);
            hashValues[i] = hashCode;
        }
        int[] bucketCount = analyzeHashValues(hashValues, magicItemsArray);
        ArrayList<String[]> hashes = new ArrayList<String[]>();
        hashes = fillHashes(magicItemsArray, bucketCount, hashes);
        double hashingComparisons = retrieveItems(hashes, magicItemsArray, randomItems);
        System.out.println("Hashing retrieval comparisons: " + df.format(hashingComparisons));
    }

```

Figure 1: Image of main function

This section of code consists of what I'm importing from the java library, some constants, and the code that I want to run.The main function in this code calls for the file of magic items to be turned into an array, 42 random items to be selected from the array of magic items, linear search to be performed for the 42 items, binary search to be performed for the 42 items, a hashing table to be created, and the same 42 items to be retrieved from the hash table.

2 Reading File Into Array

```
41      public static String[] fileToArray(File file) throws FileNotFoundException {
42
43          Scanner scan = new Scanner(file);
44
45          // Creates the array to store the magic itmes in
46          String[] itemsArray = new String[666];
47          int i = 0;
48
49          // Goes through the whole text file and adds each line to the array
50          while (scan.hasNextLine()) {
51              String data = scan.nextLine();
52              itemsArray[i] = data;
53              i++;
54          }
55          scan.close();
56          return itemsArray;
57      }
58
```

Figure 2: Image of fileToArray function

This section of code is the same from Assignment 1. Here we are taking the magicItems file and going line by line putting each item into an array. Once there are no more lines remaining in the file, the function returns the full array.

3 Generating 42 Random Items

```
59      public static final Random rand = new Random();
60
61      // Creates an array of 42 random items
62      public static String[] randomItems(String[] arr) {
63          String[] items = new String[42];
64          int i = 0;
65
66          // Generates a random number that corresponds to a item
67          // That item is added to the list if it hasn't already been selected
68          while (i < 42) {
69              int n = rand.nextInt(arr.length - 1);
70              int[] randomNumberTracker = new int[42];
71              items[i] = arr[n];
72
73              // Loops through an array of all the generated numbers to ensure that
74              // no number is used twice
75              randomNumberTracker[i] = n;
76              for (int k = 0; k < randomNumberTracker.length; k++) {
77                  if (randomNumberTracker[k] != n) {
78                      i++;
79                      break;
80                  }
81              }
82          }
83          return items;
84      }
85
```

Figure 3: Image of randomItems function

This function generates an array of 42 random items which will be what we are searching for later in the code. I created a new array to keep track of the 42 string values that will be selected. Then I created a while loop that iterates until the array is full. To ensure that no string value is used twice I created another new array that stores the random numbers that have already been created and aren't duplicates. If the new generated number isn't a duplicate then it is added to the array. Following that, the generated number is used to access the item and add it to the randomItems array that is returned.

4 Quick Sort

```java
public static void quickSort(String[] arr, int lower, int higher) {
    int i = lower;
    int j = higher;
    String pivot = arr[i + (j - i) / 2];
    String temp = "";

    while (i <= j) {

        // Compares the strings disregarding capitalization
        while ((arr[i].toLowerCase()).compareTo(pivot.toLowerCase()) < 0) {
            i++;
        }

        while ((arr[j].toLowerCase()).compareTo(pivot.toLowerCase()) > 0) {
            j--;
        }

        // If the lower index is still less than the higher index, then the values of
        // said
        // indexes are swapped
        if (i <= j) {
            temp = arr[i];
            arr[i] = arr[j];
            arr[j] = temp;
            i++;
            j--;
        }

        // Calls the quickSort function recursivley
        if (lower < j) {
            quickSort(arr, lower, j);
        }
        if (i < higher) {
            quickSort(arr, i, higher);
        }
    }
}

```

Figure 4: Image of quickSort function

This code is also taken from Assignment 1. We use the quickSort function to sort the array so that binary search will work properly. It is also easier to maneuver through the array if its sorted.

5 Linear Search

```java
public static int linearSearch(String[] arr, String[] items) {
    // Keeps track of the number of comparisons for each
    int[] numComparisons = new int[42];

    // Loops through the magicItemsArray for each of the
    // randomly selected items
    for (int i = 0; i < items.length; i++) {
        int comparisons = 0;
        for (int k = 0; k < arr.length; k++) {
            comparisons++;
            if (arr[k].compareTo(items[i]) == 0) {
                // Adds number of comparisons to the Array for each iteration
                numComparisons[i] = comparisons;
                break;
            }
        }
    }

    // Loops through array and adds up the total number of iterations
    int sum = 0;
    for (int i = 0; i < numComparisons.length; i++) {
        sum = sum + numComparisons[i];
    }

    // Takes the average of the number of comparisons for all 42 items
    int avg = sum / 42;

    return avg;
}
```

Figure 5: Image of linearSearch function

This section of code is our linear search function. This function loops through the array for each random item and compares the item in the current position of the magicItems array with the current random item until a match is found. The asymptotic running time for a linear search is O(n) but in this case, since we have to do multiple searches, we have a nested loop so the function's asymptotic running time is O(n^2). Also this function counts the number of comparisons done and puts them into an array. Using this array the average amount of comparisons is found and returned to the user.

6 Binary Search

```java
public static int binarySearch(String[] arr, String[] items) {
    int[] numComparisons = new int[42];

    // Loops through every randomItem and counts how many comparisons
    // are needed to find each item during binary search
    for (int i = 0; i < items.length; i++) {
        int comparisons = 0;
        int high = arr.length - 1;
        int low = 0;

        // Loops through the magicItems array and changes the upper and/or lower
        // bound of the search until the item is found
        while (high >= low) {
            int arrItem = (low + high) / 2;
            if (arr[arrItem].compareTo(items[i]) > 0) {
                high = arrItem - 1;
                comparisons++;
            } else if (arr[arrItem].compareTo(items[i]) < 0) {
                low = arrItem + 1;
                comparisons++;
            } else {
                // Adds the total number of comparisons to an array
                numComparisons[i] = comparisons;
                break;
            }
        }
    }

    // Takes all of the total comparisons and finds and returns the average of all
    // 42 searches
    int sum = 0;
    for (int i = 0; i < numComparisons.length; i++) {
        sum = sum + numComparisons[i];
    }

    int avg = sum / 42;

    return avg;
}

```

Figure 6: Image of binarySearch function

This section of the code is our binary search function. Binary search works by looking at the middle of our array and comparing it the string we are trying to find. From there we break up the array into two pieces, if the string value is smaller we use the left and if the string values is larger we use the right. This continues until we find the string we are looking for, typically when there is only one item left to compare to. The asymptotic running time for binary search is O($log_2(n)$), but again since we are running through it for each random item the running time for this function actually becomes O($nlog_2(n)$). This function also counts the number of comparisons, stores them in an array, and finds and returns the average number of comparisons.

7 Hashing

```
194     private static int makeHashCode(String str) {
195         str = str.toUpperCase();
196         int length = str.length();
197         int letterTotal = 0;
198         // Iterate over all letters in the string, totalling their ASCII values.
199         for (int i = 0; i < length; i++) {
200             char thisLetter = str.charAt(i);
201             int thisValue = (int) thisLetter;
202             letterTotal = letterTotal + thisValue;
203 
204         }
205 
206         // Scale letterTotal to fit in HASH_TABLE_SIZE.
207         int hashCode = (letterTotal * 1) % HASH_TABLE_SIZE; // % is the "mod" operator
208 
209         return hashCode;
210     }
211 
```

Figure 7: Image of makeHashCode function

```
212     // Took the code from the website provided and removed the code that I didn't
213     // need
214     // I need the bucketCount array so I could populate the ArrayList with the
215     // proper items at the proper hash values
216     private static int[] analyzeHashValues(int[] hashValues, String[] arr) {
217 
218         // Sort the hash values.
219         Arrays.sort(hashValues);
220 
221         int asteriskCount = 0;
222         int[] bucketCount = new int[HASH_TABLE_SIZE];
223         int totalCount = 0;
224         int arrayIndex = 0;
225 
226         for (int i = 0; i < HASH_TABLE_SIZE; i++) {
227             asteriskCount = 0;
228             while ((arrayIndex < LINES_IN_FILE) && (hashValues[arrayIndex] == i)) {
229                 asteriskCount = asteriskCount + 1;
230                 arrayIndex = arrayIndex + 1;
231             }
232             bucketCount[i] = asteriskCount;
233             totalCount = totalCount + asteriskCount;
234         }
235 
236         return bucketCount;
237     }
238 
```

Figure 8: Image of analyzeHashValues function

This section of code is how we get the hash values for each item and keep track of how many items are stored at each hash value. The makeHashCode function works by taking a string, making it all uppercase, and getting the total ASCII value of the string. Then that total is divided by 250 and the remainder is taken and returned as the hash value. I modified the analyzeHashValues function so it returns the array bucketCount which contains the amount of items stored at each hash value. This is important because of the way I fill the hash table.

8 Fill Hash Table

```
239     public static ArrayList<String[]> fillHashes(String[] magicArr, int[] buckCount, ArrayList<String[]> hashes) {
240         // Loops through the buckCount array (which has how many values are chained at
241         // each hash value)
242         for (int i = 0; i < HASH_TABLE_SIZE; i++) {
243             String[] temp = new String[buckCount[i]];
244             int tempCounter = 0;
245             // Loops through the magicArr (magicItemsArray) and checks if each item's hash
246             // value is equal to the current has value. If it is, then the item is added to
247             // a temporary String Array
248             for (int k = 0; k < LINES_IN_FILE; k++) {
249                 int tempHashCode = makeHashCode(magicArr[k]);
250                 if (tempHashCode == i) {
251                     temp[tempCounter] = magicArr[k];
252                     tempCounter++;
253                 }
254             }
255             // Once all the items are added to the current hash value array that array is
256             // stored in the hashes ArrayList
257             hashes.add(temp);
258         }
259         return hashes;
260     }
261
```

Figure 9: Image of fillHashes function

This function is how the hash table is filled with the proper values. To start, I created an ArrayList of Arrays so that the table is easy to navigate through. This ArrayList along with the magicItemsArray and bucketCount Array, from the analyzeHashValues function, are taken as inputs. Then we loop through the ArrayList and create a temporary Array of the length of bucketCount[i] and implement a counter so the items are placed through the whole Array. Next we loop through the magicItemsArray and compare the hash values to the current value of the ArrayList, if they are the same the item is added to the temporary Array. Once the array has been looped through, the temporary Array is added to the ArrayList.

9 Retrieve Items From Hash Table

```java
    public static double retrieveItems(ArrayList<String[]> hashes, String[] magicArr, String[] randItems) {
        int[] numComparisons = new int[42];
        int comparisons = 0;

        // Loops through the randItems (randomItems) Array and for each item checks its
        // gets the hash value. We also rerieve the proper array for the corresponding
        // hash value from the ArrayList
        for (int i = 0; i < randItems.length; i++) {
            int tempHashCode = makeHashCode(randItems[i]);
            String[] tempFromHashes = hashes.get(tempHashCode);
            comparisons = 1;

            // Loops through the Array for the hash value of the current random item and
            // compares each string in the array to see which one is the same as the random
            // item
            for (int k = 0; k < tempFromHashes.length; k++) {
                comparisons++;
                if (randItems[i].compareTo(tempFromHashes[k]) == 0) {
                    numComparisons[i] = comparisons;
                    break;
                }
            }
        }

        // Takes the average number of comparisons that it takes to find/retrieve the
        // item
        int sum = 0;
        for (int i = 0; i < numComparisons.length; i++) {
            sum = sum + numComparisons[i];
        }

        double avg = sum / 42.00;

        return avg;
    }
}
```

Figure 10: Image of retrieveItems function

This function is how we search through the hash table to find the 42 items we generated before. This is done by looping through the randomItems Array and generating the hash value of each item. Then using that hash value, we take the String being stored at that point in the ArrayList. From there, we loop through the Array we selected and we compare the strings to the random item. Once it is found the loop breaks and the next item is searched for. The function keeps track of the number of comparisons and stores them in an Array, then the average number of comparisons is found and returned in the same way as linear and binary search. The asymptotic running time for this function is $\Omega(1)$ and O(n). In this case we are closer to O(n) because of chaining and having to loop through the smaller Arrays in the ArrayList.

10 Results

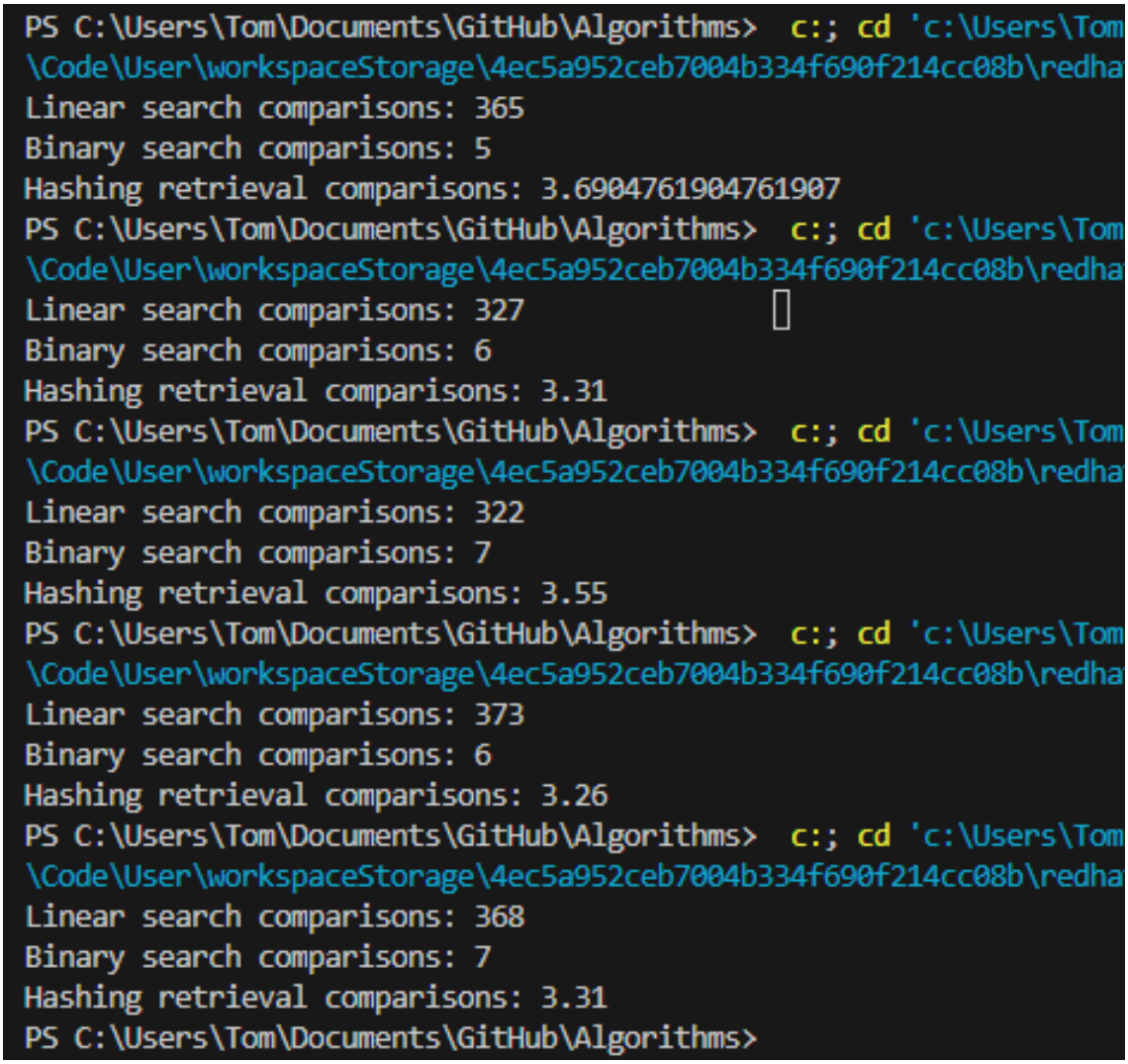

Figure 11: Image of the output of the code

These are the results of the code. The average number of comparisons per search does vary due to new items being selected every time the code is ran, but the results are always pretty consistent. We can see that linear search has the biggest jumps between runs but the results are still always around 333, which is half of the total lines of the Array. Binary search and retrieving from the hash table are very consistent being right around 6 and 3.30 respectively.