CMP-5014Y Coursework Assignment 2

100175876 (`eau16vfu`)

Sun, 19 May 2019 16:09

PDF prepared using PASS version 1.15 running on `Windows 10 10.0` (`amd64`).

☑ I agree that by submitting a PDF generated by PASS I am confirming that I have checked the PDF and that it correctly represents my submission.

Contents

Algorithm-Analysis2.pdf

 (Included PDF starts on next page.)

Algorithm Design and Implementation Q1:

O(n^2) Algorithm–

Informal Description:
Loop through the array and add each element a temporary integer
loop through the array again and check if the temp is equal to the current position.
If so add to the count. Come out of the nested loop and check if the tempCount is greater than the count of the current integer with the most occurrences. Repeat this loop until the count for each number in the array has been made then determine whether the count is high enough to be an SVD and print it to the console, else return error message.
Formal:

```
// O(n^2) notation
public static int getSVD(int[] arr) {
    int count = 1, tempCount;
    int popular = arr[0];
    int temp = 0;
    for (int i = 0; i < (arr.length - 1); i++) {
        temp = arr[i];
        tempCount = 0;
        for (int j = 1; j < arr.length; j++) {
            if (temp == arr[j])
                tempCount++;
        }
        if (tempCount > count) {
            popular = temp;
            count = tempCount;
        }
        break;
    }
    if (popular >= size / 2) {
        System.out.println("SVD of the array: " + popular);
        return popular;
    } else
        System.out.println("No SVD Available :(");
    return -1;
}
```

Fundamental Operation:
If temp == arr[j]
If tempCount > count

Runtime Complexity Function:
Best Case: the operation is performed n^2 times
Worst Case: the operation is performed n^2 times.

Characterised Runtime Complexity:
Exponential: t(n) = 2^n

O(nlog(n)) Algorithm-

Informal Description:
Sort the array, then loop though and if the current node is equal to the next add to the count. Else if the current count is greater than the max count set the current count

to the max count and set the SVD to the previous position on the array.
if the SVD is greater than half the length of the array then return the SVD.
Else return error message and -1;
Formal:

```java
// // Algorithm O(nlog(n))
static int getSVD(int arr[], int n) {

    // Sort the array, Arrays.sort has a guaranteed performance of n*log(n)
    Arrays.sort(arr);

    // Find the maximum frequency of a number
    // using linear traversal
    int max_count = 1, svd = arr[0];
    int curr_count = 1;

    for (int i = 1; i < n; i++) {
        if (arr[i] == arr[i - 1])
            curr_count++;
        else {
            if (curr_count > max_count) {
                max_count = curr_count;
                svd = arr[i - 1];
            }
            curr_count = 1;
        }
    }

    // If last element is most frequent
    if (curr_count > max_count) {
        max_count = curr_count;
        svd = arr[n - 1];
    }

    // If the current max_count is high enough
    // to be a dominating value
    if (arr.length / 2 < max_count) {
        System.out.println("SVD of the array: " + svd);
        return 1;
    } else
        System.out.println("No SVD available :(");
    return -1;
}
```

Fundamental Operation:
If arr[i] == arr[i - 1]
If curr_count > max_count

Runtime Complexity Function:
Best Case: the operation is performed n times
Worst Case: the operation is performed n * n times.

Characterised Runtime Complexity:
Log Linear: t(n) = nlog(n)

O(n) Algorithm:

Informal Description-
Create a hash table, add each element and store their frequency counts as key value pairs. Then traverse the hash table, find the element with the highest frequency count then print this to the console if the value is high enough to be dominating of the current array.

Formal:

```java
// Algorithm o(n)
static int mostFrequent(int arr[], int n) {

    // Insert all elements in hash
    Map<Integer, Integer> map = new HashMap<Integer, Integer>();

    for (int i = 0; i < n; i++) {
        int key = arr[i];
        // Store the frequency of each value as key value pairs
        if (map.containsKey(key)) {
            int freq = map.get(key);
            freq++;
            map.put(key, freq);
        } else {
            map.put(key, 1);
        }
    }

    // find max frequency.
    int max_count = 0, freq = -1;
    // traverse the hash table and get the key with the maximum
    // value in the list
    for (Entry<Integer, Integer> val : map.entrySet()) {
        if (max_count < val.getValue()) {
            freq = val.getKey();
            max_count = val.getValue();
        }
    }

    // Check if freq is a dominating
    if (freq > arr.length / 2) {
        System.out.println("SVD value is: " + freq);
        return freq;
    } else {
        System.out.println("No SVD value available");
        return -1;
    }
}
```

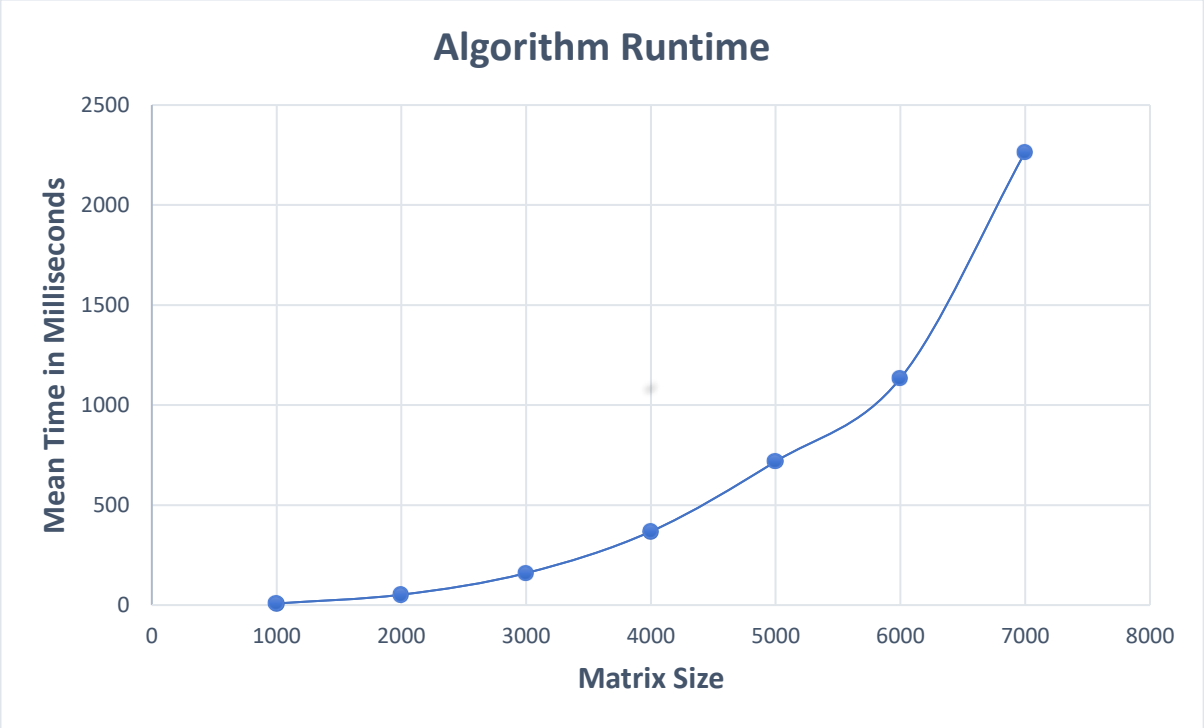
Fundamental Operation:
if (max_count < val.getValue())

Runtime Complexity Function:
Best Case: the operation is performed 1 times
Worst Case: the operation is performed n times

Characterised Runtime Complexity:
Linear: t(n) = n

Data Structure Design and Implementation Q2

The graph below shows the runtimes of the test algorithm across different matrix sizes, it shows that the algorithm is polynomial. When n reaches a size of 7000 the runtime almost doubles which is consistent with the runtime complexity of t(n) = 2^n.



ArrayHashTable.java

```java
import java.util.ArrayList;

/**
 *
 * @author: White, Robin
 */
public class ArrayHashTable extends HashTable {
    Object table[][]; // Two dimensional array
    int counts[]; // number of elements at each location
    int chainSize = 5; // Initial size of each array, defualt to 5

    /**
     * Default constructor to create an instance of an ArrayHashTable object and
     * initialise it.
     */
    public ArrayHashTable() {
        // Initialize table to an array of arrays of size capacity
        table = new Object[capacity][];
        counts = new int[capacity];

        // Initalise all values in table to null
        for (int i = 0; i < capacity; i++) {
            table[i] = null;
        }

        // Initalise all counts to 0
        for (int i = 0; i < capacity; i++) {
            counts[i] = 0;
        }
    }

    @Override
    boolean add(Object obj) {
        // Generate hash code within the capacity.
        int hash = obj.hashCode() % this.capacity;

        // Check if entry is empty and create an array of size chainSize
        if (table[hash] == null) {
            Object[] chainAry = new Object[chainSize];
            table[hash] = chainAry;
        }

        // Checks for duplicate objects in the hash table
        if (!contains(obj)) {
            // If the array is full, double the size of the array
            // and copy over the values
            if (table[hash][table[hash].length - 1] != null) {

                // Copy the chain array into a temporary array
                Object tempArr[] = new Object[this.table[hash].length * 2];
                System.arraycopy(this.table[hash], 0, tempArr, 0, this.table[hash
                    ].length);

                // Copy the chains back but with the chain size doubled
                this.table[hash] = tempArr;
            }

            // Add the given object to the hash table in the next free space
            boolean added = false;
            int i = 0;
            while (!added) {
```

```
61                  // Check if current position is null then add the passed obj
                    //and increase the count of the current hash array and size by 1
63                  if (table[hash][i] == null) {
                        table[hash][i] = obj;
65                      counts[hash]++;
                        size++;
67                      added = true;
//                        System.out.println("Added " + table[hash][i] + " with hash
    " + hash + " to hash table.");
69                  }
                    i++;
71              }
                // Item successfully added
73              return true;
            }
75          return false;
        }
77
        @Override
79      boolean contains(Object obj) {
            int hash = obj.hashCode() % this.capacity;
81
            // Loop though chain and check for given object, if found return true
83          for (int i = 0; i < counts[hash]; i++) {
                if (table[hash][i] == obj) {
85                  System.out.println(table[hash][i] + " found at positions: [" +
                       hash + "," + i + "].");
                    return true;
87              }
            }
89          return false;
        }
91
        @Override
93      boolean remove(Object obj) {
            int hash = obj.hashCode() % this.capacity;
95
            for (int i = 0; i < counts[hash]; i++) {
97              // If array contains the object passed, set the given object to null
                if (table[hash][i] == obj) {
99                  System.out.println("Removed " + table[hash][i] + " " + "from hash
                        table.");
                    table[hash][i] = null;
101
                    // Move all elements in the array back one to compensate for
103                 // the object that was removed
                    for (int j = i + 1; j < counts[hash]; j++) {
105                     table[hash][j - 1] = table[hash][j];
                    }
107
                    counts[hash]--;
109                 size--;

111                 // Successfully removed the object
                    return true;
113             }
            }
115         return false;
        }
117 }
```

HashTable.java

```
1  public abstract class HashTable {
2
3      int capacity = 10;
4      int size = 0;
5
6      /**
7       *
8       * @return size of the hash table.
9       */
10     public int size() {
11         return size;
12     }
13
14     /**
15      * Adds the given object to the hash table as long as it is not a duplicate
16      *
17      * @param obj
18      * @return true if the element is successfully added
19      */
20     abstract boolean add(Object obj);
21
22     /**
23      * Loops through the table and checks for a given object
24      *
25      * @param obj
26      * @return true if this hash table contains the given object
27      */
28     abstract boolean contains(Object obj);
29
30     /**
31      *
32      * @param obj
33      * @return Removes the given object from the array if present
34      */
35     abstract boolean remove(Object obj);
   }
```

Testing.java

```
 import java.io.FileWriter;
import java.io.IOException;
import java.io.PrintWriter;
import java.util.Random;

public class Testing {

    public static void main(String[] args) {
        int reps = 60;
//        ArrayHashTable hTable = new ArrayHashTable();
//
//        System.out.println("Adding Test Data");
//        // Add values 1 to reps to the hash table.
//        for (int i = 0; i < reps; i += 2) {
//            hTable.add(i);
//        }
//
//        System.out.println("-------------------------------------------------")
    ;
//
//        System.out.println("Check Test Data is Added");
//        // Check that all values were added correctly.
//        for (int j = 0; j < reps; j += 2) {
//            hTable.contains(j);
//        }
//
//        System.out.println("-------------------------------------------------")
    ;
//
//        System.out.println("Remove Test Data");
//        // Removes all values that were added.
//        for (int k = 0; k < reps; k += 2) {
//            hTable.remove(k);
//        }

         timingExperiment();
    }

    /**
     * An experiment that calculates the average time, variance and standard
     * deviation of adding and deleting in the hash table. The hash table is
     * used a specific number of times defined by reps with the size of the
     * matrix continually increasing.
     */
    public static void timingExperiment() {

        Random r = new Random();
        ArrayHashTable hTable = new ArrayHashTable();
        int reps = 100;
        int n = 1000; // Starting matrix size

        while (n <= 50000) {
            System.out.println("--------------------------------");
            System.out.println("Current Matrix Size: " + n);
            System.out.println("--------------------------------");
            int[] numbers = new int[n];


            for (int j = 0; j < n; j++) {
                numbers[j] = Math.abs(r.nextInt());
            }
```

```
60
                // Record mean and std deviation of performing an operation
62              // reps times
                double sum = 0;
64              double sumSquared = 0;

66              for (int i = 0; i < reps; i++) {
                    long t1 = System.nanoTime();
68
                    for (int j = 0; j < n; j++) {
70                      hTable.add(numbers[j]);
                    }
72
                    for (int j = 0; j < n; j++) {
74                      hTable.remove(numbers[j]);
                    }
76
                    //Get runtime in nanoseconds
78                  long t2 = System.nanoTime() - t1;

80                  // Convert to milliseconds to make the result more readable
                    sum += (double) t2 / 1000000.0;
82                  sumSquared += (t2 / 1000000.0) * (t2 / 1000000.0);
                }
84              //Calculate the mean time taken for each rep
                double mean = sum / reps;
86
                //Calculate the variance to see the range of the set from its mean
88              double variance = sumSquared / reps - (mean * mean);

90              //Calculate standard deviation to see how the mean runtime
                //can variate
92              double stdDev = Math.sqrt(variance);

94              // Print results to console
                System.out.println("Mean: " + mean);
96              System.out.println("Variance: " + variance);
                System.out.println("Standard Deviation: " + stdDev);
98              System.out.println();

100             if (n < 20000) {
                    n += 1000;
102             }

104             else if (n < 50001) {
                    n += 5000;
106             }
            }
108     }
}
```