IMAGE ANALYSIS SOFTWARE

MODULE: CSI_4_DSA_2122

DATE OF SUBMISSION: 13/05/2022

STUDENT ID: 4008609

Table of Contents

1. INTRODUCTION

This report outlines the implementation of an image analysis software using three different data structures allowing a comparison to be made using Big O performance analysis. Thus, finding the most efficient data structure for this task. The data structures used were tree sets, hash tables and maps. The implementation of sets (using java sets) ensures that the collection does not contain duplicates as we only want to identify unique colours. This report will also outlines the theoretical workings of the data structures implemented, thus facilitating the analysis and causes of specific time complexities for each data structure used.

2. Research and overview

2.1. Java sets

A mathematical set is a collection of unique elements that are unique (not duplicated). Java sets are represented through lists. In java sets, the data structure is searched and the list is only appended if the new item does not already exist within the collection.

2.2. Lists (Array based lists)

Java utilises array-based lists which uses a traditional array but the array size is variable. If an item needs to be added to the list, a new larger array is created, this increases the memory allocation, the contents of the previous array is then copied to the new array together with the new element. The time complexity for insertion, access, removal and changes are respectively, O(N), O(1), O(N) and O(1) [see section 2.6, below].

2.3. Tree(sets)

A tree is a hierarchical data structure consisting of nodes, branches and leaves. Each node can have a maximum of only two children. The insertion and deletion of elements in the tree is achieved in constant time. This implementation focuses on tree sets which doesn’t allow duplicated to exist and also takes advantage of binary searching.

2.3.1 Binary search trees

A more efficient method of searching lists is by using a binary search. This can only be implemented if the collection is already sorted. This is carried out by first identifying the mid point of the list and determining whether the element being searched is on the left/right half of the list. The algorithm will then appropriately half the list, usually multiple times, until the searched element is located. This results in the worst case time complexity of O(Log N)

2.4. Hashing

Hashing involves assigning a hash code to identify array element(s) (or index). This allows for multiple objects to be grouped using the same hash code, leading to a decrease in the number of possible hash codes. This results in a faster search time. If a duplicate element is inserted into a hash set, the previous element (duplicate) will be deleted and the new element is added

2.5. Maps

A map is a data structure also referred to as a dictionary in python where every element in a collection is assigned a key. Unordered hash maps typically use a hash table resulting in a reduction in time taken to look up a key.

2.6. Big O performance (asymptotic) analysis - theoretical

Asymptotic analysis is the performance of the algorithm in relation to the number of inputs increasing indefinitely to infinity. This is usually represented with the algorithms worst, average and best-case scenario. An example is searching which causes a worse case is when you are dealing with an entirely

unsorted list. The best-case scenario could be that the list is already sorted perfectly.

2.6.1 Constant time O(1)

A constant time complexity algorithm is one where the time taken to execute is not affected by the number of inputs. An example of a constant time function could be accessing an array directly using an index number rather than searching for the element otherwise.

2.6.2 Linear time O(N)

In linear time complexity the execution time is directly proportional to the number of inputs. Examples of methods/algorithms with a linear time complexity are; a non-nested loops, print statements and sequentially searching a list/collection.

2.6.3 Quadratic time O(N^2)

An algorithm can be described as having a quadratic time complexity of O(N^2) if the time taken to execute a command is exponential in relation to the number of inputs. Examples of algorithms with a quadratic time complexity are; nested for loops and bubble sort where two elements are swapped till an order is achieved.

2.6.4 Logarithmic time O(Log N)

If an algorithm has a logarithmic time complexity, it can be inferred that as the number of inputs increases, the time taken for the algorithms increases slowly. Hence, the performance is more favourable with an increasing list size.

3. Implementation and results

3.1. Using Tree sets

3.1.1 Implementation

```
43  public static Collection<Color> distinctColoursUsingTreeSet(LoadedImage image) {
44      TreeSet<Color> collection = new TreeSet<>();       //Initalise treeset from util package
45      for (int x = 0; x < image.getWidth(); x++) {       //Loop to search image
46          for (int y = 0; y < image.getHeight(); y++) {   //loop to search image
47              Color c = image.getColor(x, y);           //uses Ccolor.java
48              {
49                  collection.add(c);              //Adds colour to treeset
50              }
51          }
52      }
53      return collection;
```

Figure 1: Code snippet using tree set to identify unique colours

The code snippet above (see fig 1) shows the implementation utilising tree sets to determine the number of unique colours in each image. A treeset object is created using the util package. The code loops through all pixels in the image and returns the number of unique colours. Since the tree set function already checks for duplicates before appending the tree, additional code is not required for manually checking for duplicate colours prior to appending the tree.

3.1.2 Execution time / results

Image 1	37 ms
Image 2	85 ms
Image 3	53 ms
Image 4	44 ms
Image 5	60 ms
Image 6	90 ms

Image 7	201 ms
Image 8	274 ms

Figure 2: Time taken to analyse each image using hash sets

The table above (see fig 2) shows that the time taken to analyse each image using tree sets. The execution time for this tree set method is significantly faster than the list method in all cases (all 8 images). The results from the list method are given in appendix 8.2 (for comparison only).

3.1.3 Actual performance analysis (Big O)

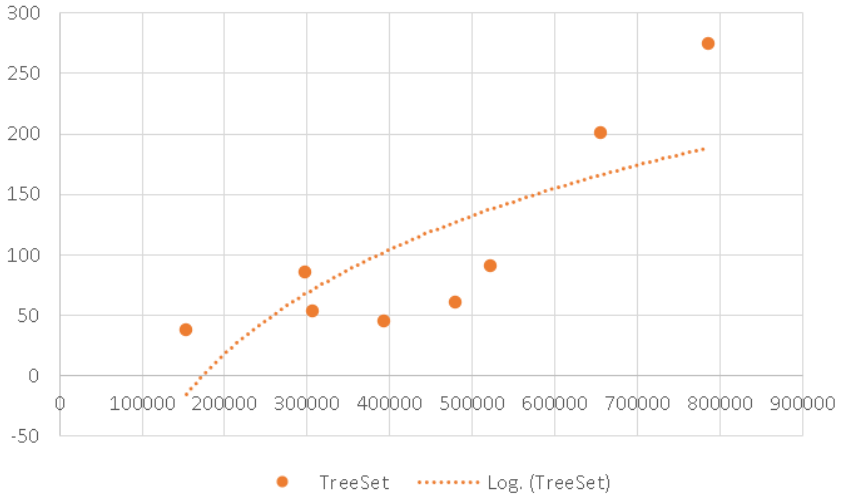

Figure 3: Big O plotted for tree set (Log (N))

The performance analysis chart above (see fig 3) shows a logarithmic time complexity when utilising tree sets to store and manage unique colours. This is due to the elements of the tree set being stored in order. The ordered lists allows searching to be done quickly using binary searching.

3.2. Using hash sets

3.2.1 Implementation

```
59  public static Collection<Color> distinctColoursUsingHashSet(LoadedImage image) {
60      HashSet<Color> collection = new HashSet<>();       //Initalise hashset from util package
61      for (int x = 0; x < image.getWidth(); x++) {          //nested loop to search image
62          for (int y = 0; y < image.getHeight(); y++) {
63              Color c = image.getColor(x, y);
64              {
65                  collection.add(c);                  //adds colour to hashset
66              }
67          }
68      }
69      return collection;
70  }
```

Figure 4: Code snippet of hash set to identify unique colours

The code snippet shown above (see fig 4), shows the implementation of a hash set to identify unique colours, this method is similar to tree sets as it utilises the java.util package which has a built in hash set function. The method creates a hashset object and loops through the pixels in the image, adding each new colour (assuming its not duplicated) to the hash set. Duplicate colours are automatically ignored when utilising java sets. This method also uses the Ccolour package.

3.2.2 Execution time / results

Image 1	25 ms
Image 2	55 ms
Image 3	25 ms
Image 4	22 ms
Image 5	27 ms
Image 6	42 ms

Image 7	85 ms
Image 8	100 ms

Figure 5: Time taken to analyse each image using hash sets

The table above (see fig 5), shows the time taken to analyse each image using a hash set, from this we can observe that the time taken for each image is lower when compared to tree sets and also significantly lower than the list method This is due to the advantages (see section 2) of hashing causing a reduction in search time.

3.2.3 Actual performance analysis (Big O)

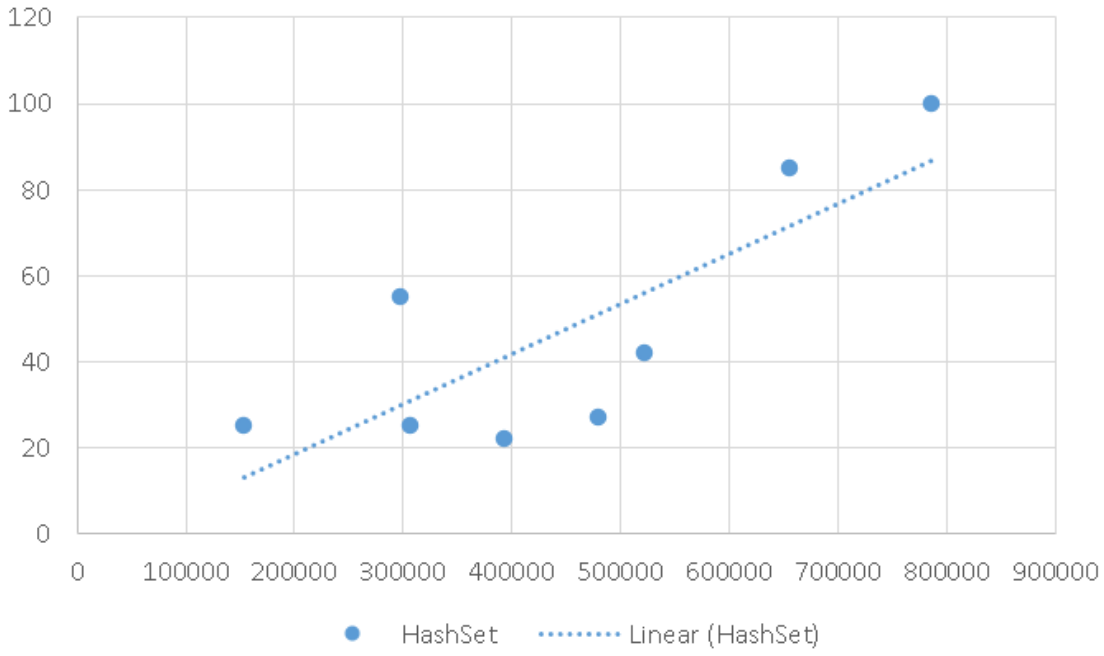

Figure 6: Big O plotted for hash set (O (1))

The big O graph plotted above (see fig 6), shows that that time complexity of hash sets is O(1) (constant). Meaning that the execution time is not affected by the number of inputs. In this implementation, it can be observed that the execution time does increase slightly with number of pixels, however this is expected as one hash code can contain multiple colours and the method must also add the colour to the hash which takes time.

3.3. Using maps

3.3.1 Implementation

```
77  public static Map<Color, Integer> frequencyCountUsingMap(LoadedImage image) {
78      Map<Color, Integer> collection = new HashMap<>();              //initalises hashmap
79      Set<Color> img = (Set<Color>) distinctColoursUsingHashSet(image);//initalises
                                                                                hashset
80      for(Color color:img){                          //loops image
81          collection.put(color,image.getHeight()*image.getWidth());    //adds image to
                                                                              freqcount
82      }
83      return collection;
```

Figure 7: Code snippet for frequency counter using maps to identify unique colours

The code snippet above (see figure 7) shows the utilisation of maps and sets to produce a frequency map of all colours in each image. A method “frequencycountusingmap” is created. A new hash map is initialised . A for loop is created /started . This loops through all the pixels in the image and adds colour to the map. In this implementation only the first image is shown, however this method can be used to output frequency maps for multiple images.

3.3.2 Frequency map (output)

Figure 8: Cropped image of frequency count using maps

The image above (see figure 8) shows a cropped output of the frequency map produced for the first image. It shows each colour used in the image using a bar chart.

3.3.3 Execution time

Image 1	25 ms
Image 2	49 ms
Image 3	32 ms
Image 4	29 ms
Image 5	29 ms
Image 6	48 ms
Image 7	98 ms
Image 8	135 ms

Figure 9: Time taken to produce frequency map

The run time above (see figure 9) shows the time taken to produce a frequency map for each image**.** Although producing a frequency map output is a slightly different task to that of identifying unique colours using a list, it can be seen that the execution times given in figure 9 (see above) are significantly faster.

3.4. Comparison of actual Big O performance

image	pixels	unique	list	treeset	hashset	freqmap
01-pagodas.jpg -	153120	52148	2516	37	25	25
02-foxboro.png -	298125	126848	29496	85	55	49
03-Ataleoftwokitties_restored.jpg -	307200	63347	13872	53	25	32
04-bug.jpg -	393216	67939	5408	44	22	29
05-pyramids.jpg -	480000	24309	8449	60	27	29
06-tianzishan_wulingyuan_zhangjiajie_2012.jpg -	523264	66350	19439	90	42	48
07-Duke_Humfrey's_Library_Interior_3,_Bodleian_Library,_Oxford,_UK_-_Diliff.jpg	655872	205443	132644	201	85	98
08-Randell_Cottage_03.jpg -	786432	283516	307328	274	100	135

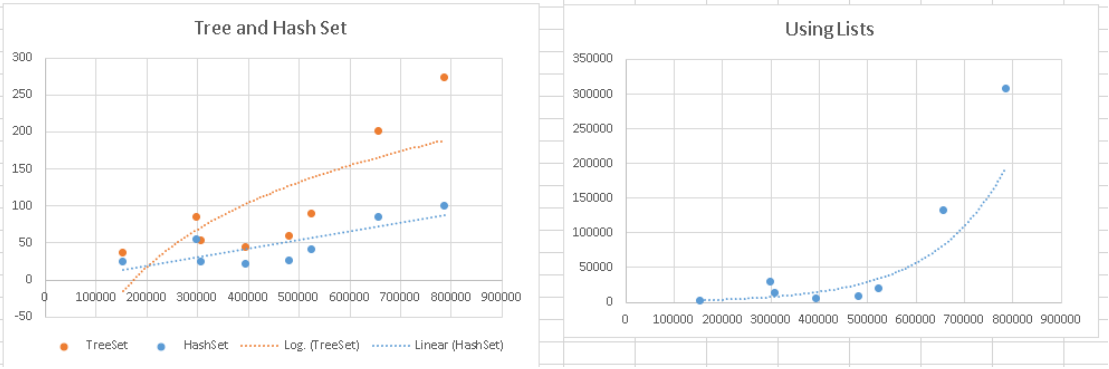

Figure 10: Excel snippet for Big O performance of lists, trees and hash sets.

3.4.1 List performance

The image above (see fig 10) shows the asymptotic efficiency of the two data structures implemented as well the list method provided. As predicted, the list has a time complexity of $O(N^2)$. Meaning that as the number of inputs (pixels) increases, the time taken to analyse the image will quadratic (see section 2.6.3). This is due to the fact the method must determine if the colour has previously been added to the array list, this is different to sets, which don't allow duplicates, or hash tables which can store multiple elements under one hash code.

3.4.2 Tree set performance

As backed by the research (see sec 2), the tree set method is shown to have a time complexity of O (Log N). This is achieved using a self balancing binary search tree which allows for the searching to be carried out faster, as elements are already stored in order. This further allows for binary searching to be carried out where the algorithm determines a mid-point and halves the collection until the element is located resulting in a logarithmic time complexity.

3.4.3 Hash set performance

The performance of the hash set implementation (see fig 10) shows that the method was unable to achieve a perfect constant time O(1) in regards to asymptotic performance (there was a slight variation). This is because each hash code could contain multiple elements (unique colours), hence, direct access would require additional searching to determine if a colour already exists within the collection. The method must also add the colour to the hash table which takes additional time.

4. Conclusion

Through the asymptotic analysis carried out on different data structures, it can be seen that lists are most inefficient, especially when dealing with a large number of inputs. This is due to the limitations on searching as unsorted elements can be difficult to locate, furthermore the lack of hashing results in the inability to decrease in the number of search iterations.

Also, it can be seen, the benefits of self balancing binary searching are significant on tree sets due to their logarithmic time complexity. This can however cause a slight increase in time complexity as elements must initially be stored in ascending order. The analysis also found that hash sets allow for much larger datasets to be managed easier due to hash sets having a near constant time complexity (depending on implementation and hash efficiency). This is mostly because multiple elements can be stored in the same bucket within the hash table.

5. Reflecting on learning

This assignment has significantly improved my understanding of why certain data structures have different efficiencies and their dependency on specific implementation. This has further expanded my knowledge on the significance of software performance, especially using big O, and its impact on the user of such

software. I now understand which data structures may be best to implement for a given scenario

6. Table of figures

7. References

[1] Mike, CM. (2022) Resource files. [Data structures and algorithms, CSI_4_DSA_2122]. London south bank university, week 7.

[2] Mike, CM. (2022) 05- Queues, Stacks and Performance Analysis.ppt[Data structures and algorithms, CSI_4_DSA_2122]. London south bank university, week 5.

[3] Mike, CM. (2022) 08 - Duplicates and Ordered sets.ppt [Data structures and algorithms, CSI_4_DSA_2122]. London south bank university, week 8.

8. Appendix

8.1 Code listing (self implemented only)

```
public static Collection<Color> distinctColoursUsingTreeSet(LoadedImage image) {
    TreeSet<Color> collection = new TreeSet<>();        //Initalise treeset from util package
    for (int x = 0; x < image.getWidth(); x++) {        //Loop to search image
        for (int y = 0; y < image.getHeight(); y++) {   //loop to search image
            Color c = image.getColor(x, y);             //uses Ccolor.java
            {
                collection.add(c);                  //Adds colour to treeset
            }
        }
    }
    return collection;
}

/**
 * Student to provide implementation.
 */
public static Collection<Color> distinctColoursUsingHashSet(LoadedImage image) {
    HashSet<Color> collection = new HashSet<>();        //Initalise hashset from util package
    for (int x = 0; x < image.getWidth(); x++) {            //nested loop to search image
        for (int y = 0; y < image.getHeight(); y++) {
            Color c = image.getColor(x, y);
            {
                collection.add(c);                      //adds colour to hashset
            }
        }
    }
    return collection;
}
```

```
public static Map<Color, Integer> frequencyCountUsingMap(LoadedImage image) {
    Map<Color, Integer> collection = new HashMap<>();
    Set<Color> img = (Set<Color>) distinctColoursUsingHashSet(image);
    for(Color color:img){
        collection.put(color,image.getHeight()*image.getWidth());
    }
    return collection;
}
```

8.2 Runtime/results using lists (for reference only)

The results below are given purely for comparison. These results were generated using existing code (and not generated using my code)

Image 1	2164 ms
Image 2	14596 ms
Image 3	11948 ms
Image 4	4124 ms
Image 5	7695 ms
Image 6	16120 ms
Image 7	69856 ms
Image 8	300237 ms