Robert Ferguson and Ian Cresse

May 7th 2015

This algorithm analysis assumes that n is the number of characters in the file and m is the number of unique characters in n.

```
      private void frequencyCount(String message)
      {
1         char parts[] = message.toCharArray();
2         for (char c : parts)
          {
3             if (charFrequency.containsKey(c))
4                 charFrequency.put(c, charFrequency.get(c) + 1);
              else
5                  charFrequency.put(c, 1);
          }
      }
```

1: message.toCharArray(); takes n time

2: Loops over 3—5 n times

5 only happens when a new unique key is found. Therefore it will execute exactly m times.

Therefore, line 4 happens n-m times. (n = n-m+m)

3 is executed for each iteration of the loop. Takes $c_{contains}$ time.

4 takes $c_{put} + c_{get} + 1$ time (c_{inc}).

5 takes only c_{put} time.

The loop in total runs n times, but m of those times, it will only take c_{put} time instead of c_{inc} time. ccontains occurs regardless, so we can factor it out for a second.

$$\sum_{i=1}^{n-m} c_{inc} + \sum_{i=1}^{m} c_{put} = c_{inc}(n-m) + c_{put}(m) \quad \text{which is multiplied by } c_{contains}, \quad c_{contains}(c_{inc}(n-m)+c_{put}(m)) \text{ time.}$$

This method takes O(n-m) time.

```
        private void buildTree()
        {
1           root = innerBuildTree();
        }

        private HuffmanNode innerBuildTree()
        {
2           MyPriorityQueue<HuffmanNode> pq = new MyPriorityQueue<HuffmanNode>();
3           for (Character c : charFrequency.keySet())
4               pq.add(new HuffmanNode(c, charFrequency.get(c)));
5           while (pq.size() > 1)
6               pq.add(new HuffmanNode(pq.remove(), pq.remove()));
7           return pq.remove();
        }
```

1: A simple assignment which calls building the rest of the tree. Adds c_{root} time to the rest of the method.

2: Assignment and instantiation. c_{pq} time.

3: A loop that runs line 4 m times.

4: Adds a new HuffmanNode based on the current character and it's frequency. Takes $c_{huff} + c_{get}$ (c_{ghuff}) time. Adding to the priority queue takes $m\log_2 m$ time.

$$\sum_{i=1}^{m} c_{ghuff} m\log_2 m = c_{ghuff} m^2 \log_2 m \text{ time}$$

5: Gets the size from the PriorityQueue and preforms a Boolean check. c_{while} time.

6: Creates a new Huffman node based on retrieval of 2 Huffman nodes and then adds the new Huffman node back into the priority Queue. In effect, this reduces the size of the priority queue by 1. This means that the entire while loop will run m-1 times. Retrieving from the priority queue takes $m\log_2 m$ time.

$$\sum_{i=1}^{m-1} 3c_{ghuff} m\log_2 m = c_{while} m + 3c_{ghuff} m^2 \log_2 m - 3c_{ghuff} m\log_2 m \text{ time}$$

The method runs in $4c_{ghuff} m^2 \log_2 m + c_{while} m - 3c_{ghuff} m\log_2 m + c_{pq} + c_{root}$ time, or $O(m^2\log_2 m)$

```
        private void encode(HuffmanNode root)
        {
1           encode(root, "");
        }
        private void encode(HuffmanNode root, String path)
        {
2           if (isLeaf(root))
3               codes.put(root.value, path);
            else
            {
4               encode(root.left, path + '0');
5               encode(root.right, path + '1');
            }
        }
```

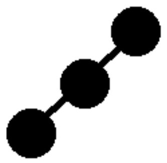
fig 1

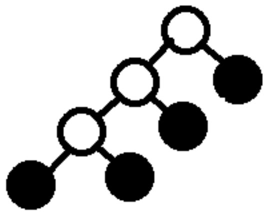
fig 2

0: On recursion

The worst case for a naive BST is where the real is really a linked list in disguise. (See fig 1)

By extension, the worst case for the size of a Huffman Tree is one where there are m-1 links in the tree that are for traversing and m links have values in them. (See fig 2) The worst case Huffman tree therefore has 2m-1 nodes.

This means that in the worse case, a method that recurs on any non-leaf node will recut m-1 times, and the base case will occur m times. However, checking for the base case, line 2, will occur 2m-1 times.

Assuming a left-bias, line 4 will occur m times and line 5 will occur m-1 times.

With that in mind:

1: Calls the recursive part of the function, c_{call} time.

2: c_{check} time. Will be called 2m-1 times.

3: c_{put} time. Will be called m times.

4 & 5 both of these functions will take c_{append} time, plus the time it takes to recur on the roots. They will be called 2m-1 times in total.

We end up with $2c_{check}m - c_{check} + c_{put}m + 2c_{append}m - c_{append}$ time, which is O(m)

```
1    long stop = inputString.length();
2    for (int i = 0; i < stop; i++)
     {
3        String bitString = encodedFile.getBinaryString(inputString.charAt(i));
4        for (int j = 0; j < bitString.length(); j++)
         {
5            if (bitString.charAt(j) == '1')
6            compressionStream.writeBit(1);
7            else if (bitString.charAt(j) == '0')
8            compressionStream.writeBit(0);
         }
     }
```

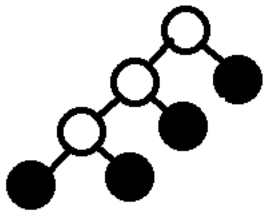
fig 2

1: Assigns a variable, cassign time.

2-8: Loop. Will run n times in total.

2: Increments i. c_{inci}

3: Gets the binary string associated with the character's position in the Huffman tree. Takes $c_{charAt} + c_{getBinary} + c_{assign}$ time. These constants, along with incrementing the loop take $c_{outLoop}$ total time.

4-8: Loop. Will run 2m-1 times in total, the maximum length of any binary string (see fig 2)

5: c_{charAt} and cbool time, (c_{if})

6: c_{write} time.

7: Same as 5

8: same as 6

Because it's structured as an if/else if to check to malformed input, the worst case scenario for this series of loops is when a binary string is x 0's in a row, and ending with a 1, such as 00001.

Line 5 will be run 2m-1 times since we always check it over the entire length of the string, while line 7 will be run 2m-2 times, one less for each time line 5 was successful, which we are assuming to be once.

$$\left(\sum_{j=1}^{2m-1} 2c_{if} + c_{write} \right) - c_{if} = 4(c_{if}+c_{write})m - 2c_{if}+c_{write}$$

Now that we have the value of the j loop, we can calculate the value of the i loop.

To simplify, c_{if} and c_{write} will be consolidated into c_{inner}

$$\sum_{i=1}^{n} c_{outloop}(4(c_{inner})m - 2c_{inner}) = c_{outloop} \sum_{i=1}^{n} (4(c_{inner})m - 2c_{inner}) =$$

$$c_{outloop} \sum_{i=1}^{n} 4(c_{inner})m - \quad c_{outloop} \sum_{i=1}^{n} 2c_{inner}$$

Which is equivalent to $4c_{outloop}c_{inner}nm - 2c_{outloop}c_{inner}n$, which is O(nm)

Conclusion:

The methods that comprise Huffman's algorithm are bound by O(n-m), $O(m^2\log_2 m)$, O(m) and O(nm)

Before we can come to a conclusion, we must talk about the relationship of the variables n and m.

As stated on the first page, n is the length of the message that is being coded and m is the number of unique characters in that message. It is always the case that n >=m. It would not make sense for there to be more unique characters in a message than there are characters.

It is however completely possible that n > m. In fact, that is usually the case. The first method is instead bound by O(n) since the n should overpower the work that m saves.

Additionally, as long as $n > m\log_2 m$, that is, each character appears on average $\log_2 m$ times, $nm > m^2\log_2 m$.

Theoretically, the worst case run time of this implementation of Huffman's algorithm runs in $O(m^2\log_2 m)$ time, but that would mean that each character in the entire string appears less than $\log_2 m$ times, which for ASCII would mean a message where each character shows up no more than 3 times, and all the ASCII characters are present. In reality, it runs in O(nm) time.