Huffman Encoding & Decoding Report

By: Nabeeha Syeda & Rishon Dass

Research

We were reading about the Huffman Code algorithm in textbooks and online. We were trying to find out how the algorithm behaves. Professor in class has explained a basic algorithm of Huffman coding but we used that as inspiration to make our implementation. Based on our research, the Huffman encoding algorithm takes the frequency of each character in a string and then sorts them in a dictionary from least to greatest by frequency. It then takes the dictionary and uses a tree structure where the lowest frequencies are at the base of the tree which then creates a parent node which is the sum of the two frequency values.

Design

During our brainstorming session, we concluded that we want to make a simple design based on the greedy algorithm, we chose this method over creating a more efficient one was because we wanted to implement knowledge we learned from the class into the project and finding a more efficient running algorithm would mean using someone else's algorithm, which is both unoriginal and plagiarism . We created node objects similar to a binary tree that consists of “data” which represented the frequency value, the “c” which represented the character of the string, the “code” which represents the encoded binary value of the character, and left and right arrays to store child nodes.

First, we needed a way to get the frequencies of the given string/input so in order to do this, we created a simple algorithm that would iterate through each individual character. Then it would store the value in a dictionary which has a key of the character with the value as its occurrence.

Next, we took our dictionary with the character frequencies and sorted them into least to greatest order by frequency. We did this by iterating through the dictionary and creating a basic min and max function values in order to sort the dictionary appropriately.

Then, we took our sorted dictionary and inserted it into our huffman algorithm which ran linearly and would iterate through the sorted dictionary and assign values of the dictionary to the node object we designed earlier, while leaving the child nodes empty temporarily. After that we designed a priority queue operation that was also linear which would assign the left and right children in addition to calculating the sum values for their parent nodes.

Encoding the tree was the next step in our process, we designed a function that is recursive since the binary tree was already created in the previous step, this helped our time complexity and turned into O(logn). The algorithm would start a root node with an initial code of “0” which would be a node that has a sum of its left and right subtrees. As the tree went left we would append another “0” to the parent nodes code and assign it to the left subtrees. As the tree went right it would append a “1” to the parent nodes code and assign it to the right subtrees accordingly.

Lastly we’d want to print the tree in the correct order, we designed a function that would iterate through the tree by using recursion and would print out the left subtrees first and then would print the right because of the recursive property.

Implementation

To implement the design we decided to use HTML 5, CSS 3, and Javascript ES6. We decided to use these as our team members were experienced with these languages and would be easier to access and compile for all members. In addition, javascript is fast, simple, reduces server load as it's mostly client-side, and has the ability to create a rich interface.

We split the code into 5 sections:

updater.js:

Listens to the page and activates events based on the input given. For example, if you were to type the text in the input fields it would take those and parse it and store the data locally for use. This file takes takes input/data given from the html file and converts it into variable data that we can use to derive, construct, and draw the tree

huffmanNode.js:

This file is the core feature of the entire program. We took an object oriented approach for this file. We created a prototype for the huffman function which would act as our "class". Then we have functions inside the prototype that consist of the main functions that we use to make the Huffman tree.

draw.js:

This file is solely dedicated to drawing the tree on the screen. It takes the data that is in a tree format and represents the nodes as circles along with their values, and their links to the children as lines. To get the animation, we used an external library called D3.js which helped us with the keeping position of the circle, so that way it looks visually appealing.

Index.js:

This file consisted of all the auxiliary functions that are used in the other files. We created this file so we can create our own local library of functions so we didn't have to repeat code.

Index.html & style.css:

The html file is to primarily get input from the user and the css is to create better visuals.

Analysis

The analysis of the program will help us better understand how the program works and will help us find the runtime. This information can help us estimate how efficient our code will run in the long run. It's important as using small inputs, such as 1,000 - 3,000 characters, would be relatively fast in any runtime compared to inputs like we use in the real world which can easily go into the 1 million character count. We're going to primarily focus on the encoding and decoding aspect of the program as it is the key function of the program that gives us the end result. Although, we will be looking at the auxiliary functions that are significant in the design.

Encoding:

Step 1: Input

We get the input from the html, this can be either a string or a file input. The data is gathered and stored by a listener which updates automatically when change is detected in the input.

```
encoder.addEventListener('input',
    function inputListener(e){
        document.getElementById("decoderInput").value = "";
        document.getElementById("DeCode").style.display =
"none";
        document.getElementById("DeCode").value = "";
        huffmanCoding.clearDecoder();
        fileInput.value = null;
        decoderInput.value = null;
        update();
    }

);
```

Note: not every single event listener is listed above, but are very similar to this one

Time complexity: O(1)

Reasoning: Each line of code is just accessing data that is directly passed through and isn't iterating or doing any type of searching

Step 2: Frequency

We take the input and use our frequency auxiliary function in order to gather the information from the input.

```
let frequency = (string) => {
    var freqDict = {}
    for (var i=0; i<string.length;i++) {
        var character = string.charAt(i);
        if (freqDict[character]) {
            freqDict[character]++;
        } else {
            freqDict[character] = 1;
        }
    }

    return freqDict;
}
```

Time complexity: O(n)
Reasoning: The time complexity is O(n) because it iterates through the string one character at a time, this makes the loop go n times where n is the number of characters in the given input.

Step 3: Create Tree

We take the frequency dict and sort it, then we create the huffman tree taking the frequency values and assigning it as the sum of for the parent value

```
createTree: function(){

        var charArray = [];
        var charFreq = [];

        this.sortedFrequency.forEach(x => charArray.push(x[0]));
        this.sortedFrequency.forEach(x => charFreq.push(x[1]));

        let n = charArray.length;
        let queue = [];



        for(var i = 0; i < n; i++){
            let huffNode = new HuffmanNode();
            huffNode.c = charArray[i];
            huffNode.data = charFreq[i];
            huffNode.left = null;
            huffNode.right = null;
            queue.push(huffNode);
```

```
        }

        var root = null;

        queue.sort(function(a,b){return a.data-b.data});
        if(queue.length > 0){
            root = queue[0];
        }

        while(queue.length>1){
            var x = queue[0];
            queue.shift();
            var y = queue[0];
            queue.shift();
            var f = new HuffmanNode();
            f.data = x.data + y.data;
            f.c = '-';
            f.left = x;
            f.right = y;
            root = f;

            queue.splice(0,0,f);
            queue.sort(function(a,b){return a.data-b.data});
        }
        return root;
    }
```

Time complexity: O(n)
Reasoning: First the data from the data has to be read each key at a time for a number of times to create the nodes. Then it parses through the data once again going through the nodes and using a queue to sort the data into a tree

Step 4: Encode

This step takes the tree we made previously and assigns 0 and 1's based on its location in the tree

```
printCode: function(root, s){
        if(root == null){
            return ;
        }
        if (root.left == null && root.right == null ){
            root.code += s;
            this.codeDict[root.c] = s;
            document.getElementById("TableCode").value +=root.c
+ ":" + s+"\n";
            return;
        }
```

```
            this.printCode(root.left,s+"1");
            this.printCode(root.right,s+"0");



        }
```

Time Complexity: O($log(n)$)
Reasoning: The algorithm is recursive and since it's based on a binary tree insertion, each value it adds is found by going left or right of the previous tree based on its value.

Step 5: Display Data

The last step is to display the data that we have calculated in the previous functions.

Decoding:

Decoding is the same process as encoding until the part it encodes the data. The same functions are utilized, however, they're in reverse order. First we get the input for the decoder which is a dictionary that we can find either in an existing instance or a input from the encoded file. So the order would be: input, decode, display data.

```
deCode: function(code,list){
        var output = "";
        if(code != ""){
            for(var i = 0; i < code.length;i++){
                for(var j = i; j <=code.length;j++){
                    for(key in list){

                        if(code.substring(i,j)==list[key]){
                            output += key;
                            i = j;
                            break;
                        }
                    }
                }


            }

        }
        if(output != ""){
```

```
            return output;
        }


    }
```

Time Complexity: O(n^3)
Reasoning: The 3 for loops are the reason why it would be O(n^3), as the code takes an input of 0 and 1s and a dictionary that has which code represents what character. The code needs to parse through each 0 and 1 at a time until it finds a match in the dictionary.

Conclusion

This project was a fun experience that let us explore different avenues and challenged our minds in order to come up with creative solutions when faced with a problem. One thing we would do differently if we were to do this again would be to create a more efficient running code. Although our code runs very fast given small inputs, the runtime would increase significantly when given “real world” inputs. Learning this concept was very engaging and created an interest that could be further developed.