Programming Assignment 2: Report [30 points]

Write a short report that includes the following information:

(a) Provide the pseudocode and complexity of the rst method you implemented.

Pseudocode of Algorithm 1

*the number of characters: n
In **Program2.java** write function:

Algorithm 1 computeEncodingTree(charFrequencies)

```
//transform all char and frequencies to nodes and add each Node to an priority queue //O(n)
create priority queue pq using new MyComparator
for i 1 to n do
    new Node nodei ←its index and frequency //O(1)
    add to priority queue pq //O(1)
end for
create a root node, nodes in the middle all have index -1 and frequency -1. // O(1)
while pq size is larger than 1, construct the tree // O(n), proof later do
    poll smallest node from priority queue pq //poll+reorder the queue=O(1)+O(log(n))
    then poll the smallest from pq again // poll + reorder the queue = O(1) + O(log (n))
    Set a parent node of top two nodes (smallest two nodes) in the tree // O(1)
    Add the new parent back to the Priority Queue pq //O(1)
end while
return the last node in the queue // is the root node we are looking for to return //O(1)
```

A class MyComparator was set for the priority queue *pq*. The function compare is:

Algorithm 2 compare(node1, node2)

```
return frequency of node1 - frequency of node2
```

Time Complexity Analysis

The while loop with priority queue size larger than 1 takes the most operation times. In each loop, two nodes are polled from the priority queue *pq* and one new node will be add to the priority queue. The queue has n nodes inside at the beginning and decrease one node in each loop until only one node left in the queue. The loop repeats $n-1$ times. Then in each loop. We repeatedly poll nodes from the queue, and each time we poll and reorder the priority queue(a min-heap). Since we started with n nodes as leaves and the queue only decrease one each loop, it takes at most $\log(numberofnodesinthequeue) = O(\log n)$ to complete. Because set a parent node and add it to the queue takes constant time, Each loop takes $2 * O(\log n)$, and it runs $O(n)$ times, therefore, in big-O notation we can write the whole loop time complexity $O(n \log n)$.

Other than this loop, the transformation from ArrayList char and frequencies takes O(n) because we spent constant time on creating a node, putting frequencies and index in each node, and placing the node in priority queue *pq*.

Since $O(n) < O(n \log n)$, the time complexity of first method/function computeEncodingTree is $O(n \log n)$.

(b) Provide the pseudocode and complexity of the second method you implemented.

Pseudocode of Algorithm2

*the number of characters: n

Plan for this algorithm: Set a queue to do **BFS** and input each node's encoding. In each loop, we first poll out previous layer's nodes(start with root), then either end as a leaf and give the corresponding binary Encoding to ArrayList *huffman_encoding* or add an additional 0(left) or 1(right) in the node's field *biEnoding* and continuously add its children to the queue

Algorithm 3 computeEncoding(*root*, *huffman_encoding*)

```
create a queue to store nodes and poll nodes from the tree //O(1)
//the next layer of nodes to add 0(left) or 1(right) //O(1)
set biencoding of root node to be an empty string
queue ←root in the queue to start //O(1)
//O(number of nodes in the tree) = O(2n-1) = O(n)
while queue is non-empty do
   poll the first node in the queue and record its binary encoding // O(1)
   if the node is a leaf, i.e. has neither left nor right child then
      set current encoding to corresponding index of huffman_encoding //O(1)
   else
      if the node has right child then
         Give encoding to this node and add to the queue //O(1) + O(1)
      end if
      if the node has left child then
         Give encoding to this node and add to the queue //O(1) + O(1)
      end if
   end if
end while
```

In order to store the binary encoding in each node, the following field is created in **Node.java**.

Algorithm 4 Node

```
set field index
set field frequency
set field left
set field right
set field parent
set field biEncoding

...functions already in Node.java to extract or set fields' data
```

Two functions in **Node.java** is added to set and extract *biEncoding*:

Algorithm 5 setBinaryEncoding(string)

```
node's biEncoding ←string
```

Algorithm 6 getBinaryEncoding()

```
  return node's biEncoding
```

By the end of the while loop, all nodes will be giving an encoding but only leaves contains corresponding $huffman\_encoding$ for each character we need, all other nodes are helping leaf nodes to get its encoding. Then, return the resulted corresponding encoding

Time Complexity Analysis

To begin with, creating queue, setting root encoding(an empty string), and adding *root* to *queue* all takes constant time.

Then we run through the while loop. In each loop, we will process one node that is at the beginning of the queue. Depending on the status of the node, we will either give a corresponding encoding to its child/children or input the current nodes encoding to ArrayList $huffman\_encoding$. According to how the Huffman tree was built, the node that represents a character must be a leaf. So we only input coding to ArrayList $huffman\_encoding$ when we find a leaf, that is a node having no child. Here, processing the node takes only constant time, such as checking if it has a child, setting encoding of current nodes children, and inputting into ArrayList $huffman\_encoding$. Therefore, the number of times that the while loop runs is the time complexity of the whole loop. In the first method, we construct the tree by creating one additional node each time extracting two nodes, so n-1 additional nodes are created other than n nodes we transformed directly from ArrayList $charFrequencies$. In total, the tree contains $n + (n-1) = 2n-1$ nodes. Thus the loop run $2n-1$ times. Ignoring the constant gives $O(n)$.

In total, the time complexity of Algorithm *computeEncoding* is $O(n)$.

(c) Prove that your encoding is prex-free.

Prove by contradiction.

Assume the encoding is not prefix-free, which means a characters encoding contains another characters encoding. Assuming one binary encoding is x and contains the other encoding y. By the definition of prefix-free and applies oppositely, x is longer than y and the first n codes of x is the same as y, where n is the length of y. In Huffmans method, the code is determined by how to get to the leaf node from the root of the tree. Left child will add 0 in the encoding and right child add 1. Applying the situation of x and y, since the first n codes of x is the same as y, node y will have children so that x becomes an extension of y. Thus, the node representing x is the descendent of the node representing y. However, all nodes that representing the characters that we want to encode are leaf nodes. This contradict that node of y has descendents.

(d) Prove that yours is the optimal character-by-character, lossless, prex-free encoding.

Prove the encoding is character-by-character.

Definition of character-by-character encoding is assumed to be: each character will have its own corresponding encoding. To prove that algorithm is character-by-character encoding, we use strong induction.

Base case: Start with two characters. According to the algorithm, first two characters with smallest frequency will be transformed to two nodes and combined by sharing a parent node with frequency equals to the sum of its childrens frequency. Then this parent node is the root of Huffman tree and two leaf nodes are representing two characters. Since we encode each leaf nodes according to the path from root to that leaf node, two characters has their corresponding encoding uniquely.

Inductive Hypothesis: Assume n characters and all cases with less than n characters will hold the truth that each one has its own encoding, then each character will still have its own encoding when there are n+1 characters.

Inductive Case: Since any leaf nodes has its unique path from the root to itself, we want to prove $n + 1$ nodes will be constructed by $n + 1$ characters. Knowing n characters construct tree T_n with n leaf nodes. Then we want to show that $(n + 1)$ characters construct tree T_{n+1} with $n + 1$ leaf nodes. Assume $(n + 1)th$ character has a frequency f.

- If f is smaller than all other characters frequencies, the node representing the $(n + 1)th$ character will be added to the tree first and its parent node created will be regarded as a node n_{parent} with the sum of frequencies of the $(n+1)th$ node and the other node with smallest frequency. Now the priority queue contains n nodes and we know n characters construct tree T_n with n leaf nodes. Since one leaf node is n_{parent}, which actually contains two leaf nodes. The final tree will have n+1 leaf nodes.

- If f is larger than the sum of frequencies of i nodes and smaller than the frequencies or the sum of frequencies of other $n - i$ nodes: we know i characters construct a Huffman tree T_i with i nodes. Because the $(n+1)th$ node is the next node with smallest frequency, in the algorithm we will combine the $(n + 1)th$ node with T_i by sharing a parent node n_{parent} and add n_{parent} back to the priority queue. The queue currently hold n-i+1 nodes and these nodes will construct a tree with n-i leaf nodes. Since n_{parent} has $i + 1$ leaf nodes, there are $n - i + i + 1 = n + 1$ leaf nodes.

- If f is larger than the sum of all characters frequency, the node representing the $(n+1)th$ character is the last node that will be added into the tree. Other n characters construct tree T' with n leaf nodes with its root n'. By the algorithm, combine n' and the $(n+1)th$ node by sharing parent n_{parent}. This new n_{parent} become the root of the final Huffman tree that has $n + 1$ leaf nodes.

In conclusion, the $(n+1)th$ character will construct $n+1$ leaf nodes that each has its own encoding. Therefore with a list of characters, each character will always have its own corresponding unique encoding.

Prove the encoding is lossless.

The algorithm produce a Huffman tree that each leaf node representing a character and the encoding is the unique path from the root of the tree to that leaf. With proof from part (c) and above, we know the encoding is character-by-character and prefix-free, then since each character has unique encoding, the encoding can be decoded uniquely as well. Therefore, the encoding is lossless.

Prove the encoding is prefix-free.

In part (c) we proved this algorithm is prefix-free by contradiction.

Prove the encoding is optimal.

Prove by contradiction.

Let the list of characters be set A and the corresponding frequencies be $f(c_i)$ where c_i represents each characters. The number of characters are n. The depth, which is the length of binary encoding, is $d(x, T)$ where x is the node and T is the tree.

Base case: $n = 1$, so in this case the only node is the root of the tree, so cost is zero, which is optimal.

Inductive Hypothesis: Assume the encoding is optimal for $n-1$ characters, and $n-1$ characters's corresponding nodes construct a tree T_0 that is optimal. The corresponding character list and their frequencies are A_0 and f_0. Prove the encoding is still optimal for n characters in Huffman tree T with list of characters A.

Inductive Case: Here the sum of binary encoding length of the given list A and f is represented as $S(T)$. We know:

$$S(T_0) = \sum_{x\in A_0} f(x)d(x, T)$$

Assume for the sake of contradiction that tree T is not giving optimal encoding. Then there is an optimal tree T_a with two nodes w and y such that these two nodes has the lowest frequencies. Since these two nodes has two lowest frequencies, they will be extracted from the priority queue pq first when we extract the first two nodes and combine by a new parent node. Thus w and y are sharing one parent node. Then if we remove these two modes w and y from tree T_a, the new tree is T'_a, which remain optimal because the parent of w and y was put back to pq and that queue with $n-1$ node must produce an optimal tree. Since T_0 is the tree with $n-1$ nodes, we can regard T'_a as T_0 without losing generality.

Now, try to use the sum of binary encoding length of tree T_0 to represent that of T:

$$\begin{aligned}
S(T) &= (\sum_{x\in A\ \{w,y\}} f(x)d(x,T)) + f(w)d(w,T) + f(y)d(y,T) \\
&= (\sum_{x\in A\ \{w,y\}} f(x)d(x,T)) + (f(w)+f(y)(d(z,T_0)+1 \\
&= (\sum_{x\in A\ \{w,y\}} f(x)d(x,T)) + f'(z)d(z,T_0) + f(w) + f(y) \\
&= (\sum_{x\in A_0} f(x)d(x,T)) + f(w) + f(y) \\
&= S(T_0) + f(w) + f(y)
\end{aligned}$$

Thus, similarly we know $S(T_a) = S(T'_a) + f(w) + f(y)$. Then, if $S(T) > S(T_a)$,

$$S(T_0) = S(T) - f(w) - f(y) > S(T_a) - f(w) - f(y) = S(T'_a)$$

But $S(T_0)$ is optimal, so $S(T_0) = S(T'_a)$. This is a contradiction.

Therefore, if the algorithm produce optimal encoding for $n-1$ character, it also produce optimal encoding for n characters.

Overall, the algorithm is optimal.