**3rd Milestone of Project**

**Algorithm Huffman(X)**

**Input:** String x of length n with d distinct characters

**Output:** coding tree for X

1. Compute the frequency f (c) of each character c of X.
2. (c1, f[c1] ) , (c2, f[c2] ) ,…, (cn, f[cn] )
3. Initialize a priority queue Q
4. For i=1 to n-1 Do
5. Create a single-node binary tree storing c
6. T = allocate new node.
7. Insert T into Q with key f (c).
8. While Size ( )> 1 do
9. *F1* Q.min()
10. *T1* Q.removeMin()
11. *F2* Q.min()
12. *T2* Q.removeMin()
13. Create a new binary tree T with left sub tree T1 and right sub tree T2
14. T.left=T1 T.right=T2 Insert T into Q with key f1 +f2
15. T.f = T1.f1 + T2.f2
16. return Tree Q.removeMin()

**Correctness of algorithm:**

The question is that greedy algorithm is correct or incorrect , that is does it compute the tree that minimizes the expected encoding length and optimal substructure properties the cost of any encoding tree T is **B(T) = ∑ x p(x) dT (x). where p(x) is the probability and d(x) is length.** Our approach will be to show that any tree that differs from the one constructed by Huffman’s algorithm can be converted into one that is equal to Huffman’s tree without increasing its cost. This is done by identifying an appropriate place where the two solutions differ, modify the non-greedy solution so that it is a bit closer to the greedy solution, and showing that this modification can be done so that the cost does not increase. By repeating this, we will eventually modify any solution into the greedy solution in a manner that does not increase this cost. This implies that the greedy solution is has the minimum cost. Our approach based on the following observations.

* Huffman tree is full binary tree
* in any optimal code tree
* Huffman’s algorithm produces an optimal prefix code tree

First, observe that the Huffman tree is a full binary tree, meaning that every internal node has exactly two children. It would never pay to have an internal node with only one child, since we could replace this node with its child without increasing the tree’s cost. So we may safely limit consideration to full binary trees. . Our next observation is that in any optimal code tree, the two characters with the lowest probabilities will be siblings at the maximum depth in the tree. Once we have this fact, we will merge these two characters into a single character whose probability is the sum of their individual probabilities. As a result, we will now have one less character in our alphabet. This will allow us to apply induction to the remaining n − 1 characters. Let’s first prove the above assertion that the two characters of lowest probability may be assumed to be siblings at the lowest level of the tree.

**Observation 1:**

Consider the two characters, x and y with the smallest probabilities. Then there is an optimal code tree in which these two characters are siblings at the maximum depth. Let T be any optimal prefix code tree, and let b and c be two siblings at the maximum depth of the tree. (There may be many such siblings, and if so pick any such pair.) If {x, y} = {b, c} we are done. Otherwise, from the fact that x and y have the lowest probabilities, we may label the nodes such that p(b) ≤ p(c) and p(x) ≤ p(y). Now, since x and y have the two smallest probabilities it follows that p(x) ≤ p(b) and p(y) ≤ p(c). (In both cases they may be equal.) Because b and c are at the deepest level of the tree we know that dT (b) ≥ dT (x) and dT (c) ≥ dT (y). (Again, they may be equal.) Thus, we have p(b)−p(x) ≥ 0 and dT (b)−dT (x) ≥ 0, and hence their product is nonnegative. Now, suppose that we switch the positions of x and b in the tree, resulting in a new tree T. Next let us see how the cost changes as we go from T to T 0 . Almost all the nodes contribute the same to the expected cost in both trees. The only exceptions are nodes x and b. By subtracting the old contributions of these nodes and adding in the new contributions we have

B(T ´ ) = B(T) − (old cost for b and x) + (new cost for b and x)

= B(T) − (p(x)dT (x) + p(b)dT (b)) + (p(x)dT (b) + p(b)dT (x)).

With a little algebraic manipulation we obtain

B(T ´ ) = B(T) + p(x)(dT (b) − dT (x)) − p(b)(dT (b) − dT (x))

= B(T) − (p(b) − p(x))(dT (b) − dT (x))

≤ B(T),

where the last step follows because (p(b) − p(x))(dT (b) − dT (x)) ≥ 0. Thus the cost does not increase. (Given our assumption that T was already optimal, it certainly cannot decrease either, since otherwise we would have a contradiction.) Since T was an optimal tree, T 0 is also an optimal tree. By a similar argument, we can switch y with c to obtain a new tree T 00. Again, the same sort of argument implies that T 00 is also optimal. The final tree T 00 satisfies the statement of the claim. The above claim applies to just one pair of nodes, those with the lowest probabilities. To show that the entire Huffman tree is optimal, we need to extend this argument do this by induction. In order to reduce from n characters to n − 1, we will do the same reduction that Huffman’s algorithm does; namely we will merge characters x and y into a new meta-character z, whose probability is the sum of the probabilities of x and y.

**Observation 2:**

Let Tn be any prefix-code tree that satisfies the property of Claim 1 (lowest probability symbols x and y are siblings at the deepest level). Let Tn−1 be the tree that results by replacing these two nodes and their parent with a single leaf node z of probability

p(z) = p(x) + p(y). Then B(Tn) = B(Tn−1) + p(z).

Let d denote the depths of x and y in Tn. Clearly, z is at depth d − 1 in Tn−1

Because z replaces x and y the costs of the two trees satisfies

B(Tn) = B(Tn−1) − (z’s cost in B(Tn−1)) + (x and y’s costs in B(Tn))

= B(Tn−1) − p(z)(d − 1) + (p(x)d + p(y)d)

= B(Tn−1) − p(z)(d − 1) + p(z)d

= B(Tn−1) + p(z).

Note that the cost of trees Tn and Tn−1 differ only by the fixed term p(z), which does not depend on the tree’s structure. Therefore (subject to this replacement), minimizing the cost of Tn is equivalent to minimizing the cost of Tn−1. This allows us to prove our main result.

**Observation 3:**

Huffman’s algorithm produces an optimal prefix code tree.

The proof is by induction on n, the number of characters. The basis case (n = 1) is trivial, since there is only one tree possible. If n ≥ 2, then by Claim 1, we know that the two characters x and y of lowest probability are siblings at the deepest level of an optimal tree. Huffman’s algorithm replaces these nodes by a character z whose probability is the sum of their probabilities. By induction, Huffman’s algorithm computes the optimum tree over the resulting alphabet of n − 1 symbols. Call it Tn−1. Replacing z with nodes x and y results in a tree Tn whose cost is higher by the fixed amount p(z) = p(x) + p(y). Since Tn−1 is optimal, and the cost of replacement does not depend on the tree’s structure, Tn is also optimal.

**Time Complexity**

Huffman’s algorithm proceeds as shown in above. Since the alphabet contains 6 letters, the initial queue size is n , and the final tree represents the optimal prefix code. The codeword for a letter is the sequence of edge labels on the simple path from the root to the letter.

* Line 1-2 compute the frequencies of all characters from c1 to cn in input string X.
* Line 3-7 initializes the min-priority queue Q with the characters in X. **For** loop is used to store all character’s frequency in priority queue Q.
* Lines 8–15 repeatedly extracts the two nodes T1 and T2 of lowest frequency from the queue using while loop, replacing them in the queue with a new node T, representing their merger. The frequency of T ’ is computed as the sum of the frequencies of T1 and T2 in line 15.
* The node T’ has T1 as its left child and T2 as its right child. (This order is arbitrary; switching the left and right child of any node yields a different code of the same cost.)
* After (n −1) mergers, line 16 returns the one node left in the queue, which is the root of the code tree.

.

To analyze the running time of Huffman’s algorithm, we assume that Q is implemented as a binary min-heap. For a set X of n characters, we can initialize Q using **For loop** in line 3-7 in O(n) time using the **BUILD-MIN-HEAP** procedure. The **While loop** in lines 8-15 executes exactly when only one last node left in queue (**size=1)** which takes n-1 iterations, and since each heap operation requires time O(lgn), the loop contributes O(nlgn) to the running time. Thus, the total running time of HUFFMAN on a set of n characters is **O(nlgn).**