Assignment-04-B403-cmtidmar

cmtidmar

April 2021

1 Introduction

Two days late 25 percent deduction.

2 Problem 1

There are N gas stations along a circular route, where the amount of gas at station i is gas[i]. You have a car that has an unlimited gas tank, and it costs cost[i] of gas to travel from station i to the next station (i + 1). You begin the journey with an empty tank at one of the gas stations.

a) Describe an efficient greedy algorithm to determine if you can travel around the circuit once in the clockwise direction. The algorithm must return the starting gas station's index if that is possible, or 1 if not.

An algorithm that you could potentially use to solve the gas station travel clockwise problem is first by initializing the variables you will need: a starting integer at 0 (int start = 0) and the current position you are at with the given gas[i] and cost[i] (int[] gas = new int[] and int[] cos = new int[]). Since you are starting at the first gas station, or the root, the current position you would be at is gas[0]-cost[0]. To put in it simpler terms, the gas stations can be treated as nodes, likewise as the starting position can be treated like the root. The algorithm also needs to keep track of a which nodes, or gas stations, were visited. So an empty integer array that is populated by the visited gas stations is necessary and a boolean value to determine if a node was visited.

```
int start = 0;
int[] gas = new int[];
int[] cost = new int[];
int currentPosition = gas[start] - cost[start];
int[] visitedStations = new int[];
boolean visited = false;
```

While you're at the starting position, mark true, and traverse through the gas stations. This will ensure all of the nodes will be eventually visited. If gas

station is not visited, mark as visited and put the gas station into the list of visitedStations, and then move to the next gas station (i++). If the current position was visited, skip to the next position. End the for-loop after you have visited all of the nodes or the current position is at the starting position, again. If the current position is less than the starting position, return 1 because then a full circle is not possible. However, if the current position is equal to the starting position, then return the index of the starting position

```
While (start)...do
    Start at first gas station/current position
    Mark visited as true
    Add node to visitedStations array
    For each iteration from the starting position...do
        For each iteration from start+1...do
        Traverse to the next unknown position
        If currentPosition[i] is not visited
            Mark as visited
            Add to the list of visited nodes
        If currentPosition[i] is already visited
            go to the next position
        If all gas stations are visited
            break
        If the next currentPosition[j] is the starting position
            set the currentposition[i] to the value of the starting
                position
            break
    end for
    set to a variable currentPosition[i]
end while
    If the current position is less than the starting position
        return 1
    If the next position is the starting position
        return the index of the starting position
```

b) Justify why your described algorithm is efficient.

The algorithm above is efficient as it has a time complexity of $O(n^3)$. The while loop will have the time complexity of $O(n)$ and the for loop will have the time complexity of $O(n^2)$. Altogether it will have the total time complexity of $O(n^3)$. Independently, each for loop will have ideal efficiently ($O(n)$ each) to visit all of the nodes as well as to see the next node, despite the total complexity being on the faster but less efficient side. The while loop will remain true until all nodes are visited or the starting position is seen again. The for loop will iterate through all of the unvisited nodes until a node is revisited. If the next node has already been seen or the next node is the starting node, break the for loop. The program will either return the value of the starting position, or 1 (if the starting position is not seen again). By visiting each node and keeping a list of the nodes visited, it will ensure that you will visit all gas stations.

3 Problem 2

A string s is said to be one distance apart from a string t if you can:Insert exactly one character into s to get t.Delete exactly one character from s to get t.Replace exactly one character of s with a different character to get t.

a) Describe and analyze an algorithm to implement the above algorithm using Dynamic Programming to return true if they are both one edit distance apart, otherwise return false.

Begin defining the method with inputs of string "s" and string "t" as you need to know the two strings.

```
boolean distance(String s, String t){...}
```

Simply enough, three rules are given that define whether the two strings are at one distance of another.

1. Insert exactly one character into s to get t.
2. Delete exactly one character from s to get t.
3. Replace exactly one character of s with a different character to get t

Looking at each rule, individually, you can use nested for loops to iterate through each letter of strings "s" and "t". If and only if all three rules fit/one distance apart, then return true. Else, return false.

```
// using a java-like language
    Int a = 0, b = 0, c = 0;
distance(String s, String t){
       for (i = 0; i < s.length; i++){
           for (j = 0; j < t.length; j++){
               // Inserts exactly one character to get t
               if (s.length != t.length && (s.charAt(i) != t.charAt(j)){
                      String addChar = t.charAt(j);
                      String addCharToS = s + addChar;
                      a = 1; }

               // deletes exactly one character from s to get t
               if (s.length != t.length){
                   Stringbuilder stringBuilderS = new StringBuilder(s);
                   Stringbuilder stringBuilderT = new StringBuilder(t);
                   if (stringBuilderS.charAt(i) !=
                        stringBuilderT.charAt(j)){
                       StringBuilder.deleteCharAt(i);
                       String DeleteCharFromS = StringBuilder.toString();
                        b = 1;}
                       }
               // Replaces exactly one character in s to get t
```

```
            if (s.length == t.length){
                if (s.charAt(i) != t.charAt(j){
                    String replaceCharInS = s.replace(i, j);
                    c = 1;}
            }
        }
    }
if (a && b && && c == 1 || s.equals(j)){
    return true; }
else { return false;}
 }
```

The program iterates through all of the letters in both string "s" and string "t". It compares each value at each position of the strings. While comparing, the program determines whether string "s" can be altered to match string "t". If a statement is satisfied, an integer (either a, b, or c) will be set to 1. At the end, if all three statements are satisfied, and a, b, and c are all equal to 1, then return true. However, the program can only return true if and only if a, b, and c are equal to 1. Otherwise, if the program cannot satisfy all three statements, then it returns false.

b) What is the time and space complexity and justify the same.
The time complexity of this algorithm is $O(n^2)$ because each loop iterates through each element in either of the strings. While it's iterating through, each value at each position of strings "s" and "t" are being compared to one another. The program only makes a letter swap, letter insertion, and a letter deletion. If all statements can be done to string "s" to match string "t", then the program will return true. The program will also return true with the base case of if "s" already equals "t". Otherwise, if all of the statements are not met, or if "s" does not already equal "t", then the program will return false. It's efficient as you have to iterate through all of the letters in both strings and compare them to each other in order to find whether you can alter string "s" to match string "t".

4 Problem 3

You are a professional robber planning to rob houses along a street. Each house has a certain amount of money stashed, the only constraint stopping you from robbing each of them is that adjacent houses have security systems connected and it will automatically contact the police if two adjacent houses were broken into on the same night.

Describe and analyze a dynamic programming algorithm to return the maximum amount of money you can rob tonight without alerting the police.
For example :
Input: nums = [1,2,3,1]

Output: 4 (Rob house 1 (money = 1) and then rob house 3 (money = 3).)
Total amount you can rob = 1 + 3 = 4.
Input: nums = [2,7,9,3,1]
Output: 12 (Rob house 1 (money = 2), rob house 3 (money = 9) and rob house 5 (money = 1).)
Total amount you can rob = 2 + 9 + 1 = 12.

Begin with creating the function and its necessary input, the list of houses you can potentially rob with the list only containing an integer value of how much money each house has.

```
int robbingHouses(List nums){...}
```

Next, initialize the variables you will need when determining houses.

```
int robbingHouses(List<integer> nums){
    int totalAmountRobbed = 0;
```

Then, you need iterate through each of the houses in the "nums" arrayList. To determine whether you can rob a house without getting caught by security cameras, you need to make sure that two houses you are robbing are not adjacent from each other. Since the problem specifies that the alarm will go off if two houses adjacent are robbed on the same night, you can use modulo 2 to determine whether you can rob the house. If the house value modulo 2 has a remainder, then the house cannot be robbed. If the house value modulo 2 does not have a remainder/has a remainder of 0, then the house can be robbed. The print statements will show on terminal which houses to rob without getting caught, and how much money you will get if you rob these two houses.

```
int robbingHouses(List nums){
int sum = 0;
       for (int i = 0; i < nums.size(); i++)
              {
                  if (i % 2 == 0) {
                      sum = sum + nums.get(i);
                      System.out.println("Rob house: " + nums.get(i));
                  }
              }
       System.out.println("Total Amount robbed: " + sum);
       return sum;
   }
   public static void main(String[] args){
      List<Integer> num = new ArrayList<>();
      num.add(1);
      num.add(2);
      num.add(3);
      num.add(1);
      rob(num);
```

```
    }
    }
}
```

The output will be, for example 1,

```
Rob house: 1
Rob house: 3
Total Amount robbed: 4
```

Now, evaluating for the second example, using the same function, but different input values,

```
    public static void main(String[] args){
       List<Integer> num = new ArrayList<>();
       num.add(2);
       num.add(7);
       num.add(9);
       num.add(3);
       num.add(1);
       rob(num);
    }
```

The output for example 2 of would be,

```
Rob house: 2
Rob house: 9
Rob house: 1
Total Amount robbed: 12
```

For the program the time complexity is O(n). It is efficient because, like the previous problems, you will need to iterate through all houses in the list to determine would houses you can rob.

5 Problem 4

Regular expressions are used by string search algorithms. In regular expressions:
• '?' Matches any single character.
• '' Matches any sequence of characters (including the empty sequence).
You can assume:
(a) s which is a string that contains only lowercase English letters.
(b) p which is a string that contains only lowercase English letters, '?' or '*'.
Note: The matching of the pattern within the string should cover the entire input string (not a part of it).
Describe and analyze an algorithm (along with time complexity and space complexity)which matches a string s with pattern p using:

(a) top-down dynamic programming (i.e Recursion with Memoization)
For top-down programming, you start with the initial, most complex problem then slowly break it down into small, more manageable problems to solve. You can essentially take the two input strings to make a tree out of them by starting from the root/full string and breaking them down little by little on each level. Another way to think about it is it is kind of comparable to the divide and conquer method in merge sort. After starting at the root of both strings, continuously "break" them in half on each level of the tree until you can compare individual elements from string "s" to individual elements from from string "p". If string "s" only doesn't have a character to compare to the corresponding element in string "p" then (when the broken down strings are mismatched), string "s" does not match the pattern of "p". Top-down programming has a fairly inefficient time complexity when you are working with large inputs because it takes signficiant time to essentially divide the problems down, solve the problems, then put the sub-solutions together to create the final solution. The time complexity would be around O(nlogn) similar to merge sort's divide and conquer time complexity. While the spacial complexity would be O(n) as you do not know how much space you will need since you need to break the original problem down into sub-problems until you get to the smallest problem.

b **(b)** bottom-up dynamic programming (i.e. tabulation)
For bottom-up dynamic programming, you start at the smallest problems and then work your way upwards solving the sub-problems as you go. Like the top-down problem, you can think of it as a tree. Only the tree is already made and broken into the smallest problems. To start, begin at the leftmost bottom input of both trees, "s" and "p". Traverse your way upwards whilst comparing the elements of "s" and "p" at each levels. Since bottom-up programming starts at the lowest point, we already know how much space we need when solving the sub-problems upwards, making the spacial complexity to be more efficient than top-down programming. The time complexity of bottom-up programming is also faster than top-down programming as you start at the smallest, easiest sub-problems and slowly build your way up to the original problem whilst adding the sub-solutions together in order to get to the actual solution. Taking all of this into account, the time complexity for bottom-up should be O(n) as you traverse upwards through each element in string "s" and string "p" and the space complexity should be O(1) as you already know how much space is needed since you start from the smallest point/sub-problem whilst working your way upwards.

6 Problem 5

a. Consider the string S = ddcabcdd composed of characters a, b, c, and d. Calculate the frequencies of those four characters, draw the Huffman tree, and deduce Huffman codes for them. Depending on how you define the tree and assign 0/1, it could have multiple Huffman trees.

The frequencies of a, b, c, and d are 1, 1, 2, and 4, where a = 1, b = 1 , c = 2, and d = 4. For part 5.a. I did the entire problem on paper, however, in the announcement I was not sure if only the Huffman tree could be on paper or the Huffman tree and frequency chart could as well, since the table is used to define the chart. But also, I showed each physical step. Henceforth, I will rewrite the logic of the problem on the OverLeaf document as well.

1. Take the lowest two frequencies, and add their numerical frequency value to get the parent node of 2 to create the beginning of the Huffman Tree.

2. Next, take the character with the 3rd lowest frequency and add it to the tree. Then add the frequency of the 3rd character to the previous parent node to get to the parent node of 4.

3. Since there are only 4 character inputs, add the last character to the tree. Then take the frequency of the last character and add it to the parent node of "c", so the parent node of character "d" and "4" should be 8.

4. Now, you have your tree. Every time the tree traverses left, insert the value "0". Likewise, when the tree traverses right, insert a "1".

5. Lastly, you need to decode for each character input. Start with "a". Work your way from the root, 8, and traverse down until you get to node "a". Character "a" will have the code "110", a frequency of "1", and a bit of "3" (since you do the frequency of "a" times how many numbers are in the Huffman code you just found). Character "b" will have the code "111", a frequency of "1", and a bit of "3". Character "c" will have the code "10", a frequency of "2", and a bit of "4". Character "d" will have the code "0", a frequency of "4", and a bit of "4".

b. Consider a string S composed of characters a, e, i, o, u, s and t where a is repeated7 times, e is repeated 9 times, i is repeated 15 times, o is repeated 86 times, u is repeated 13 times, s is repeated 5 times and t is repeated 12 times. Calculate the frequencies of those characters, draw the Huffman tree, and deduce Huffman codes for them.

The frequencies of a, e, i, o, u, s, and t are 7, 9, 15, 86, 13, 5, and 12, where a = 7, e = 9, i = 15 , o = 86, u = 13, s = 5, and t = 12.

1. Take the lowest two frequencies, and add their numerical frequency value to get the parent node of 12 to create the beginning of the Huffman Tree.

2. Next, take the character with the 3rd lowest frequency and add it to the tree. Then add the frequency of the 3rd character to the previous parent node to get to the parent node of 21.

3. The parent node 21, is significantly larger than the next lowest character frequencies (12 and 13). Repeat step 1 to make a separate Huffman tree with t = 12 and u = 13. That should give you the parent node of 25.

4. We can see that i = 15 is closer to the parent node of 21. Insert "i" into the tree with the parent node of 21 then add the two nodes together to get the parent node of 36.

5. We notice that the parent nodes 25 and 26 are relatively close together. So combine them into one tree and find the parent node, which will be 61.

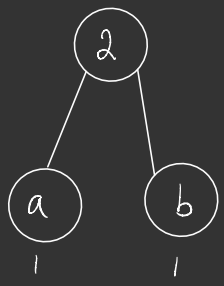
6. We now have one character left, 0 = 86. Since 61 is less than 86, add 86 to the right side of the tree then add 61 and 86 together to get their parent node, 147.

7. Now, you have your tree. Every time the tree traverses left, insert the value "0". Likewise, when the tree traverses right, insert a "1".

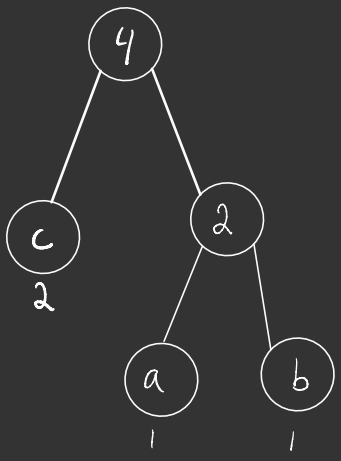
8. Lastly, you need to decode for each character input. Start with "a". Work your way from the root, 147, and traverse down until you get to node "a". Character "a" will have the code "01111", a frequency of "7", and a bit of "35" (since you do the frequency of "a" times how many numbers are in the Huffman code you just found). Character "e" will have the code "0110", a frequency of "9", and a bit of "36". Character "i" will have the code "010", a frequency of "15", and a bit of "45". Character "o" will have the code "1", a frequency of "86", and a bit of "86". Character "u" will have the code "001", a frequency of "13", and a bit of "39". Character "s" will have the code "01110", a frequency of "5", and a bit of "25". Character "t" will have the code "000", a frequency of "12", and a bit of "36".

PROBLEM 5.a.

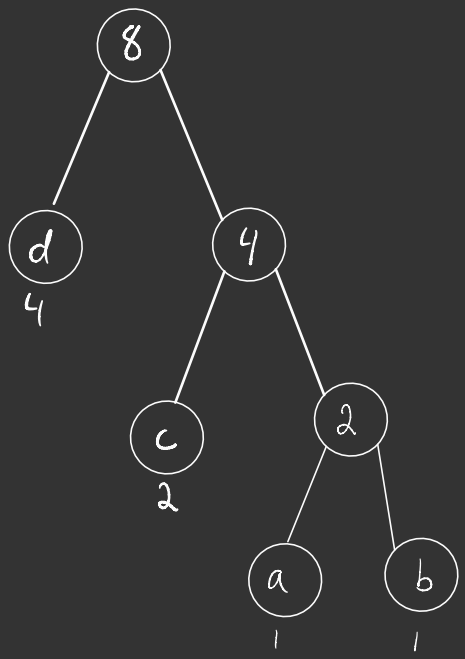
Step 1. Take the lowest two frequencies to create a beginning Huffman tree.



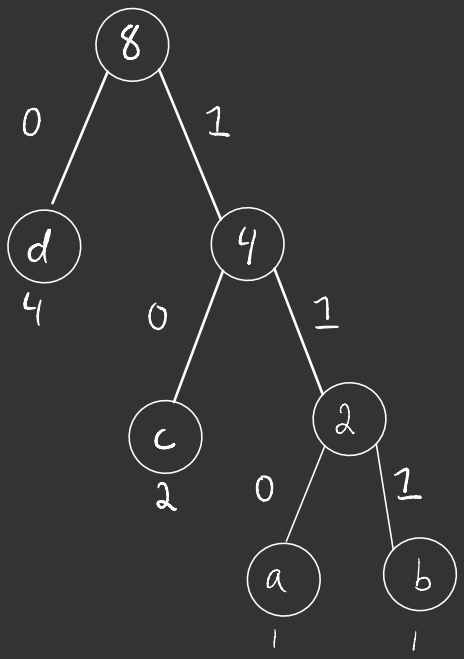
Step 2. Next, take the char with the 3rd lowest frequency. Add to tree.



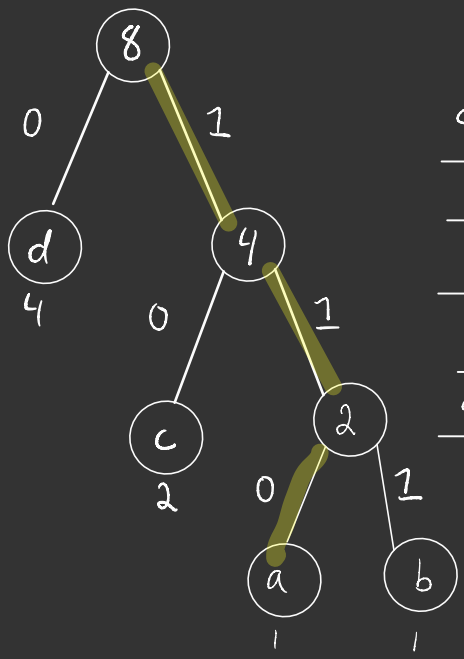
Step 3. Since the input only has 4 character values, add the last character to the tree.



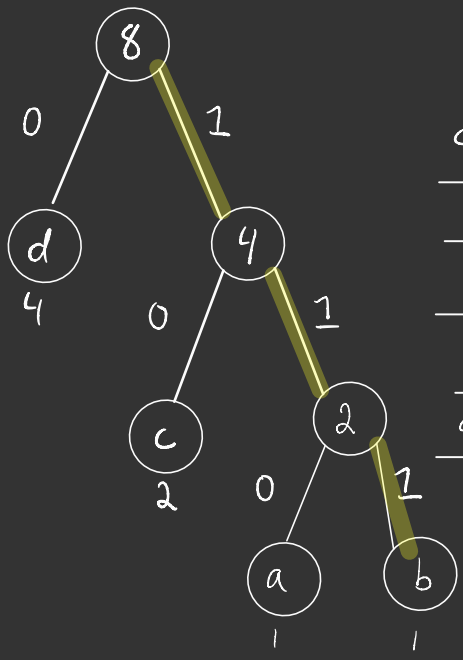
Step 4. Every time the tree traverses left, insert the value of 0, and every time the tree traverses right, insert the value of 1.



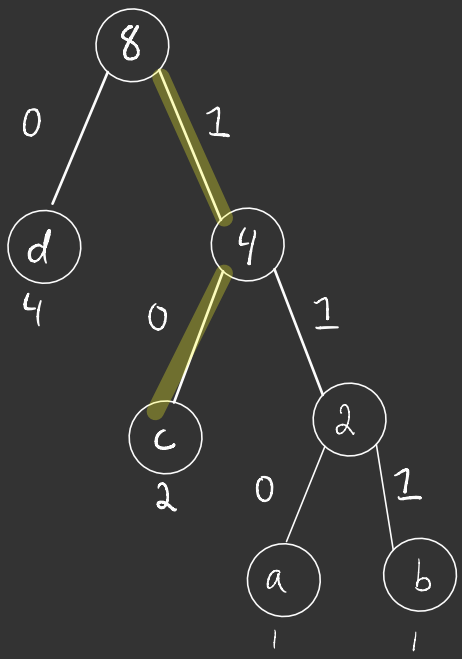
Step 4. Deduce the code of each character using the Huffman tree.



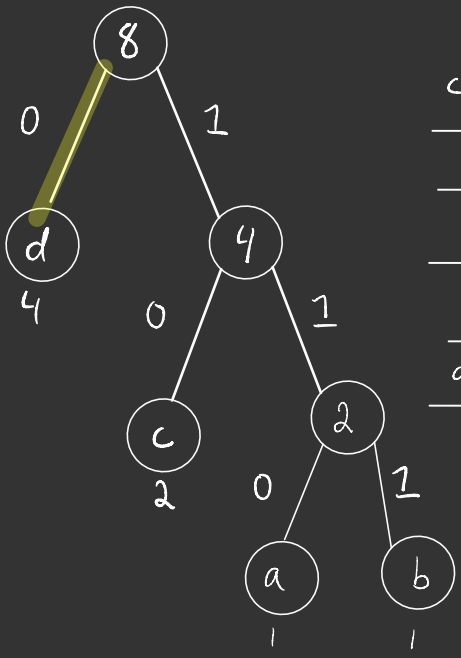
Character	code	frequency	bit
a	110	1	3
b			
c			
d			



Character	code	frequency	bit
a	110	1	3
b	111	1	3
c			
d			



Character	code	frequency	bit
a	110	1	3
b	111	1	3
c	10	2	4
d			

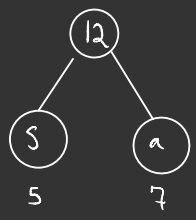


Character	code	frequency	bit
a	110	1	3
b	111	1	3
c	10	2	4
d	0	4	4

PROBLEM 5.6.

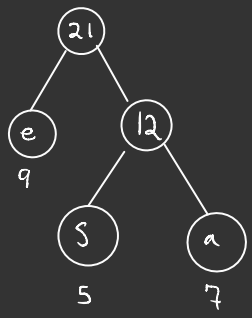
character	frequency
a	7
e	9
i	15
o	86
u	13
s	5
t	12

Step 1.



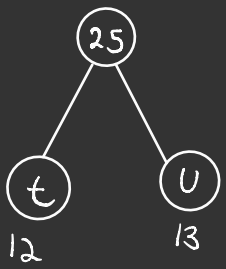
character	frequency
~~a~~	~~7~~
e	9
i	15
o	86
u	13
~~s~~	~~5~~
t	12

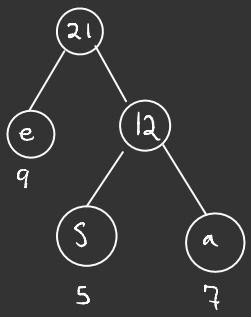
Step 2.



character	frequency
~~a~~	~~7~~
~~e~~	~~9~~
i	15
o	86
u	13
~~s~~	~~5~~
t	12

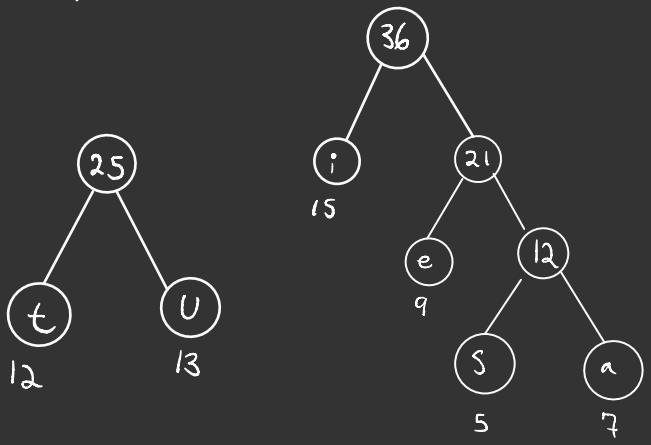
Step 3.





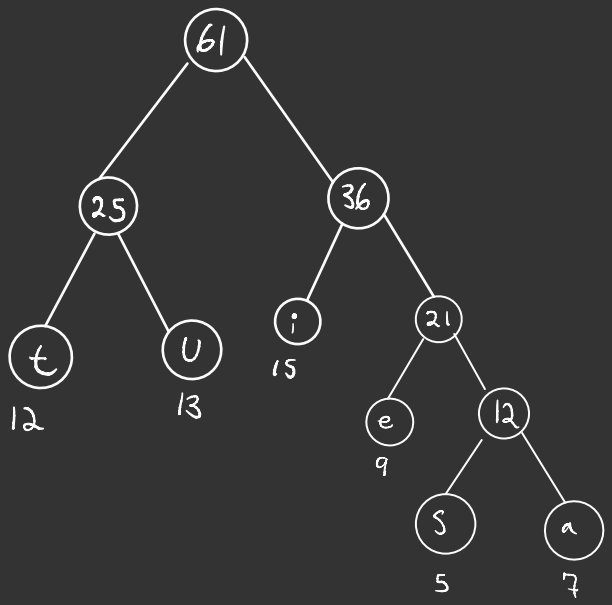
character	frequency
~~a~~	~~7~~
~~e~~	~~9~~
i	15
o	86
~~u~~	~~13~~
~~s~~	~~5~~
~~t~~	~~12~~

Step 4.



character	frequency
a	7
e	9
i	15
0	86
U	13
S	5
t	12

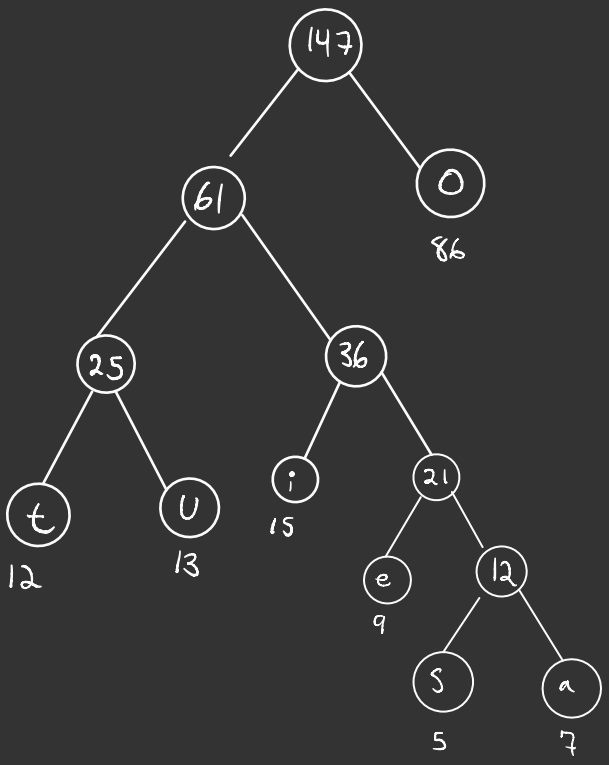
Step 5.



character	frequency
a	7
e	9
i	15
0	86
U	13
S	5
t	12

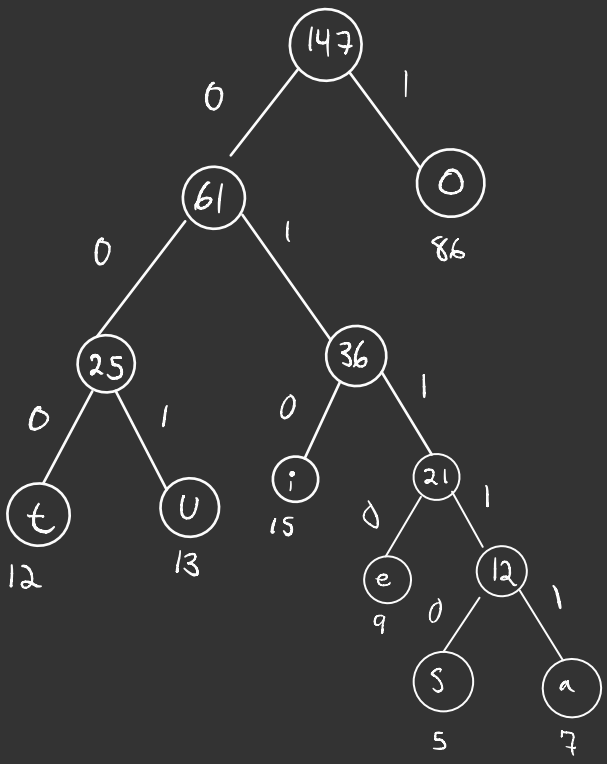
Step 6. Final Huffman tree

character	frequency
a	7
e	9
i	15
o	86
u	13
s	5
t	12

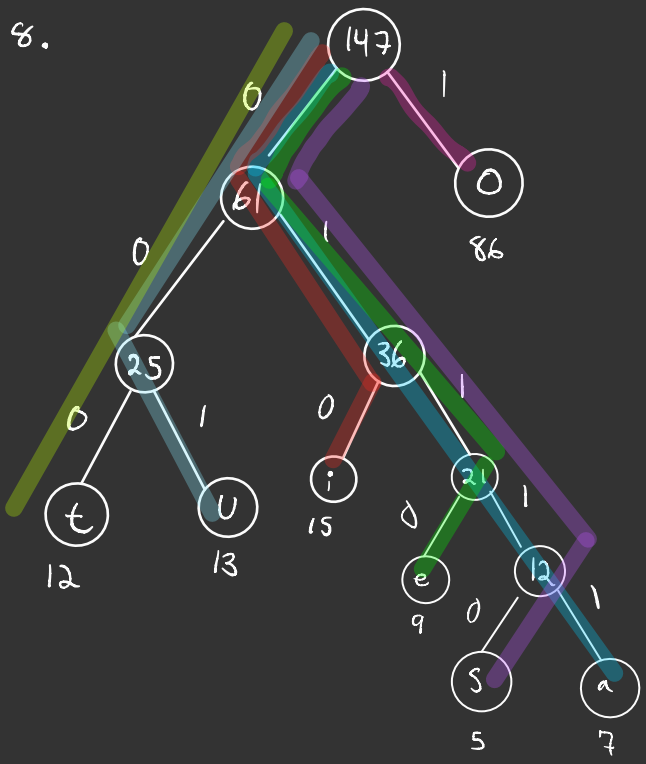


Step 7.

character	frequency
a	7
e	9
i	15
o	86
u	13
s	5
t	12



Step 8.



character	code	frequency	bit
a	01111	7	35
e	0110	9	36
i	010	15	45
o	1	86	86
u	001	13	39
s	01110	5	25
t	000	12	36