Greedy Algorithms

Introduction

- An algorithm design method
- Used to solve combinatorial optimization problems that goes through a sequence of steps (*decision points*)
- At each step, the algorithm makes the choice that looks best at the moment (*locally optimal choices*)
- *An example*:
  Make change of 36 Rs, with minimum number of coins.
  Available denominations are [50, 20, 10, 5, 2, 1]

  Greedy Method: To minimise the number of coins, keep on pulling out maximum denomination possible at each step. In this case, First 20, then 10, then 5 and finally 1. Thus at each step we use the optimal choice at that point.

Optimality of greedy algorithm

- Greedy algorithms do not always yield optimal solution.
  For example, assume that the denominations are [10, 5, 4, 1]. To make change of 8Rs, a greedy approach gives 4 coins (5, 1, 1, 1) whereas the optimal is 2 coins (4, 4)

- The problem must have two properties for a greedy algorithm to work:

  1. Optimal substructure
     The optimal solution for a problem contains optimal solutions to the sub-problems

  2. Greedy choice
     By making locally best choices it is possible to construct the globally optimum solution

Optimal substructure

The optimal solution for a problem contains optimal solutions to the sub-problems

For example, Consider the coin changing problem with denominations [50, 20, 10, 5, 2, 1].

For the given value 36, the optimal solution consists of the 4 coins (20, 10, 5, 1).

If we remove a coin from the optimum solution say 20, then the remaining coins (10, 5, 1) forms the optimum solution for the problem $36 - 20 = 16$

Coin changing problem exhibits optimal substructure property.

Greedy choice

By making locally best choices it is possible to construct the globally optimum solution.

Consider the coin changing problem again with denominations [50, 20, 10. 5, 2, 1].

For the optimum solution (20, 10, 5, 1), even if we select the coins in a different order, we will still yield the optimum solution.

Assume the contrary that the coins were selected in a different order and one of the coin say 10, is not in the solution. But in that case, the new solution contains more than one lower denomination coins whose sum is equal to 10. This is not optimal because we can replace the lower denomination coins with a single coin (10) and thus yield a better solution.

Fractional Knapsack

Fractional Knapsack

- Classical problem in algorithmic optimization
- In Knapsack problems, a knapsack with capacity W and n items with weights $w_1, w_2, , w_n$ and values $v_1, v_2, \ldots, v_n$ are given. The problem is to add items to the knapsack such that the total weight of the added items is $\leq W$ and the total value is maximized.
- Two variants

1. Fractional Knapsack
   Fractions of items can be taken. Greedy algorithm possible.
2. 0-1 Knapsack
   Take item fully or none. Greedy algorithm not possible.

Fractional Knapsack Greedy solution

We are given an instance of fractional Knapsack with capacity W = 30 and the weights and values of $n = 4$ items, as follows $(a, 140, 20)$, $(b, 50, 5)$, $(c, 60, 10)$, $(d, 50, 14)$

Greedy Algorithm:

Step 1. Compute *value to weight* ratio $\frac{v_i}{w_i}$ of each items.

Item	Value	Weight	Ratio
a	140	20	7
b	50	5	10
c	60	10	6
d	50	14	3.5

Fractional Knapsack example ...

Step 2. Sort all items in the decreasing order of *value to weight* ratio.

Item	Value	Weight	Ratio
b	50	5	10
a	140	20	7
c	60	10	6
d	50	14	3.5

Step 3. Fill the Knapsack with the (*fraction of the*) item in the decreasing order.

#	Fraction of item added	Remaining capacity	Value
0	...	30	0
1	(1) b	25	50
2	(1) a	5	190
3	(1/2) c	0	220

Fractional Knapsack Algorithm

```
Fractional knapsack ( a , p , w , c )

//a is array of items,p is array  for profits, w is array  for weights and
c is array for capacity of knapsack. //

{
   n := length [a] ;                    // finding total number of items given.
   for i := 1 to n
   do
        ratio [ i ]  := p [ i ] / w [ i ] ;
        x [ i ] := 0 ;
  done
   sort ( a, ratio ) ;                  // to arrange item with highest value first
   weight := 0
   i := 1
```

Fractional Knapsack Algorithm

```
while ( i <= n  and  weight < c )
   {
      if ( weight +w [ i ] <= c )
       then
            x [ i ] := i;
            weight := weight + w [ i ] ;
       else
            r := ( c- weight ) / w [ i ] ;
            x [ i ] := r ;
            weight := c ;
     i := i + 1 ;
  }

   output ( x )
}
```

Optimality of the greedy algorithm

Greedy Choice property. We need to show that there exists an optimal solution that selects the fraction f of an item i, as the greedy choice did. Let T^* be another optimal solution that does not select the fraction f of the item i, as the greedy choice did.

This means T^* has selected only a lesser fraction of i, because the fraction f is a greedy choice.

From T^* we can remove some items with total weight $f * weight(i)$ and can replace with $f*i$.

Since i has better *value to weight* ratio, this gives another optimal solution.

Optimality of the greedy algorithm

Optimal substructure property. Let $T = \{i_1, i_2, \ldots, i_k\}$ be an optimal solution of the fractional Knapsack problem S with weight W. We need to show that the solution set T^* we obtain by removing an item i_j from T is an optimal solution for the the problem $S* = S - \{i_j\}$, with weight $W - weight(i_j)$.

Assume the contrary that the solution T^* of the problem is not optimal.

Then it means that there exists an optimal solution T^{**} such that $value(T^{**}) \geq value(T^*)$.

If so, then $T^{**} \cup \{i_j\}$ is an optimal solution to the original problem S,

which is a contradiction.

Task Scheduling

Task Scheduling

- Also called interval scheduling problem
- Motivated by applications in resources scheduling
- We are given a set $S = \{a_1, a_2, \ldots, a_n\}$ of n activities that are to be scheduled on some resource, which can serve only one activity at a time. Each activity a_i has a time interval specified by *start time* s_i and a *finish time* f_i. Two activities are *compatible* if their intervals doesn't overlap. The problem is to schedule as many compatible activities as possible on the resource.

An example

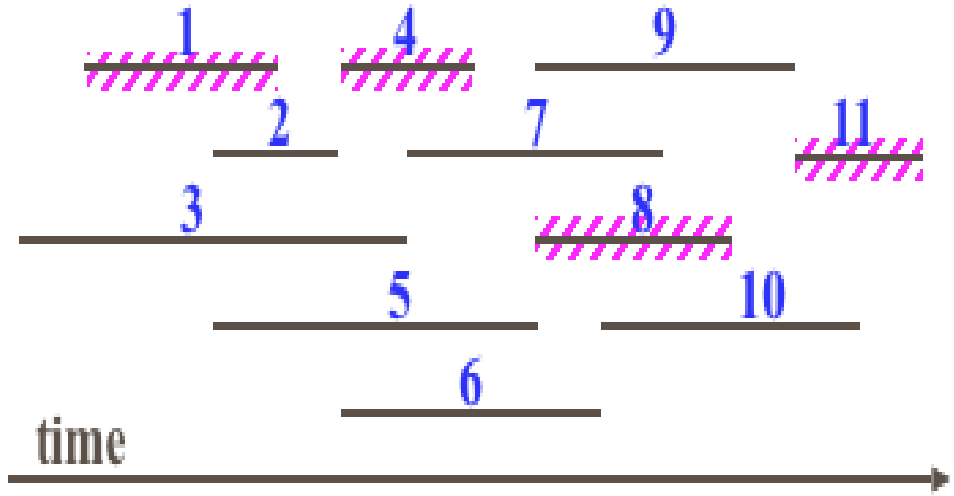
Consider the following set S of activities with their start and finish times.

a_i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	9	9	10	11	12	14	16

The subset $\{a_3, a_9, a_{11}\}$ consists of mutually compatible activities, but it is not a maximum subset.

The subset $\{a_1, a_4, a_8, a_{11}\}$ is a largest subset of mutually compatible activities.

Another largest subset is $\{a_2, a_4, a_9, a_{11}\}$.



The greedy algorithm

Which of the following parameter can be used for greedy choice?

1. Start time
2. Finish time
3. Shortest activity ($f_i - s_i$)

Greedy Approach : Recursive

```
RECURSIVE-ACTIVITY-SELECTOR(s, f, i, j)
1 m ← i +1
2 while m < j and sm < fi    //Find the first activity in Sij
3 do m ← m + 1
4 if m < j
5 then return {am}∪ RAS(s, f, m, j)
6 else return 0
```

Greedy Approach : Iterative

GREEDY-ACTIVITY-SELECTOR(s, f)

1 $n \leftarrow length[s]$

2 $a \leftarrow \{a_1\}$

3 $i \leftarrow 1$

4 for $m \leftarrow 2$ **to** n

5 do if $s_m \geq f_i$

6 then $A \leftarrow A \cup \{a_m\}$

7 $i \leftarrow m$

8 return A

The greedy algorithm

1. Sort the activities by finish time and set $R = \phi$
2. Pick the first activity a_i in the sorted list and add to R
3. Remove all activities that are not compatible with a_1
4. Repeat steps 3 and 4 until the list is finished

Consider the following activities again.

a_i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	9	9	10	11	12	14	16

The list is already sorted in the increasing order of finish time. We add a_1 to R and delete all incompatible activities (a_2, a_3, a_5, a_{10}). Then add a_4 to R and delete the incompatible activities (a_6, a_7), then add a_8 and delete a_9, and finally add a_{11} to R.

Optimal substructure and greedy choice

1. **Greedy choice** : There exists an optimal solution containing the greedy choice (a_1).

   Let R be an optimal solution and if R contains a_1, we are done. otherwise let a_k be the first activity in R, then we can remove a_k and safely add a_1 because $f_1 \leq f_k$. The new R is also optimal .

2. **Optimal substructure**: The optimal solution can be made from the greedy choice plus an optimal solution to the remaining sub problem.

   Proof is left as an exercise.

Huffman Encoding

- Huffman coding is a form of statistical coding
- Not all characters occur with the same frequency!
- Yet all characters are allocated the same amount of space

1 char = 1 byte, be it e or x

Steps:

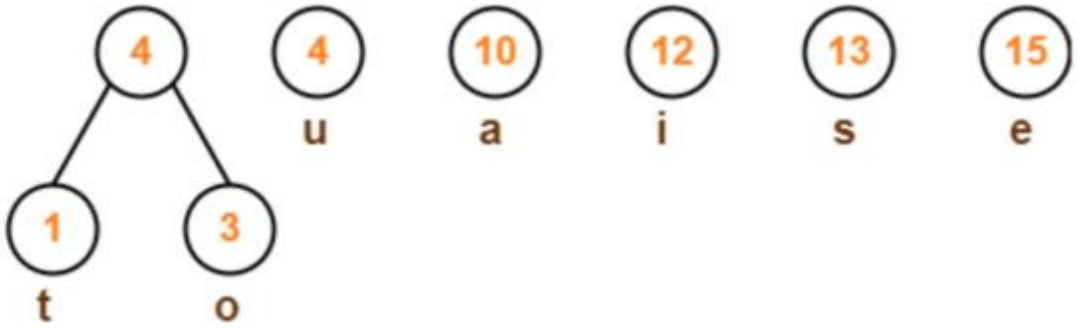
1. Scan text to be compressed and tally occurrence of all characters.
2. Sort or prioritize characters based on number of occurrences in text.
3. Build Huffman code tree based on prioritized list.
4. Perform a traversal of tree to determine all code words.
5. Scan text again and create new file using the Huffman codes.

Example

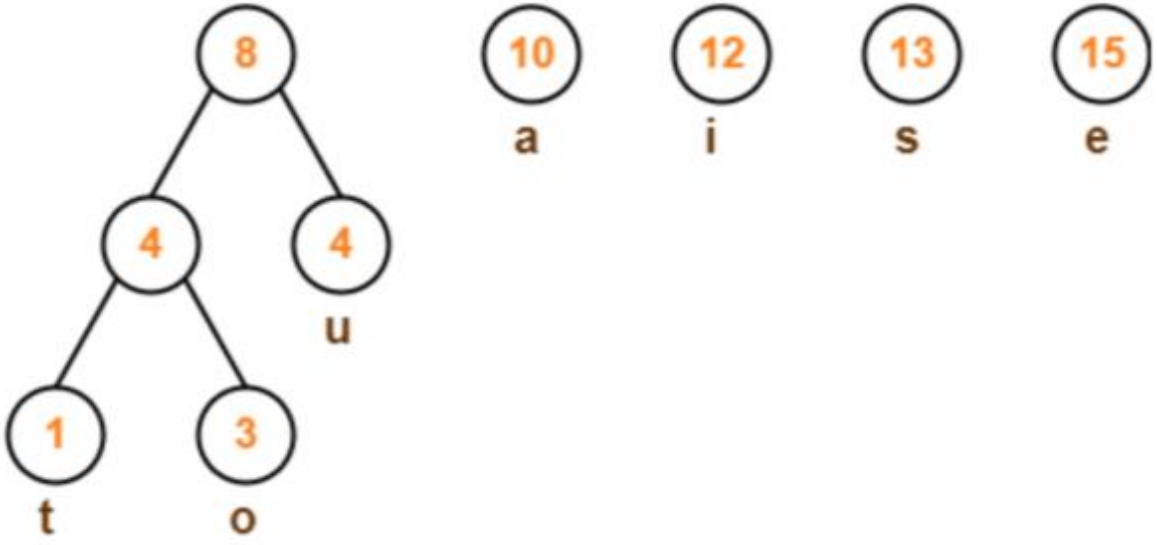
Characters	Frequencies
a	10
e	15
i	12
o	3
u	4
s	13
t	1

Step-01:

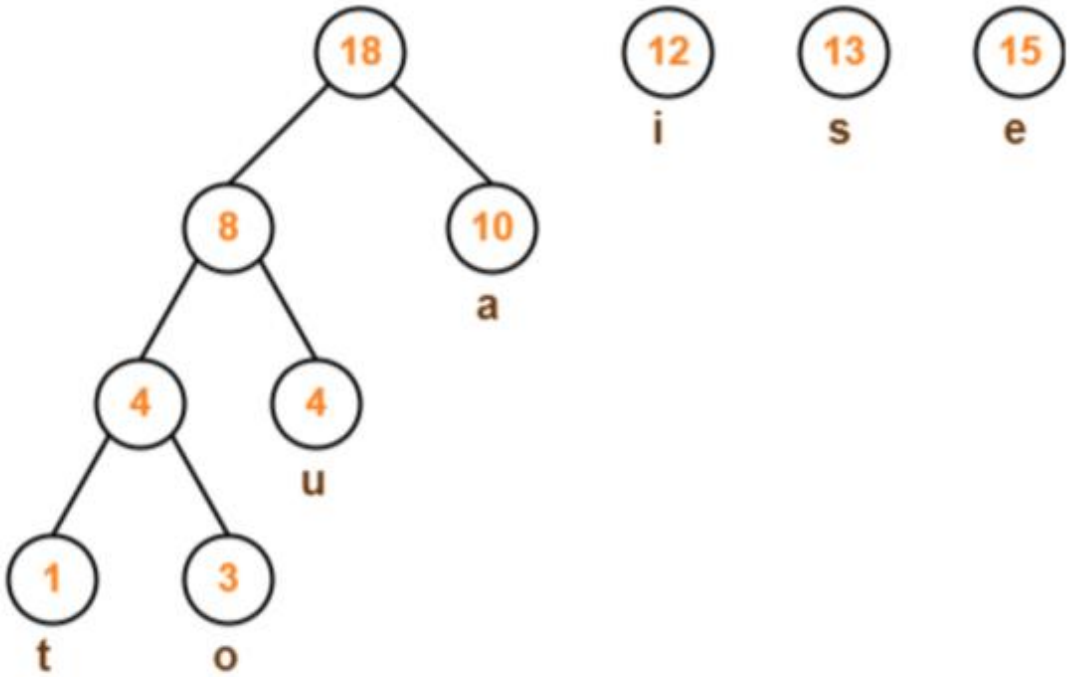
Step-02:

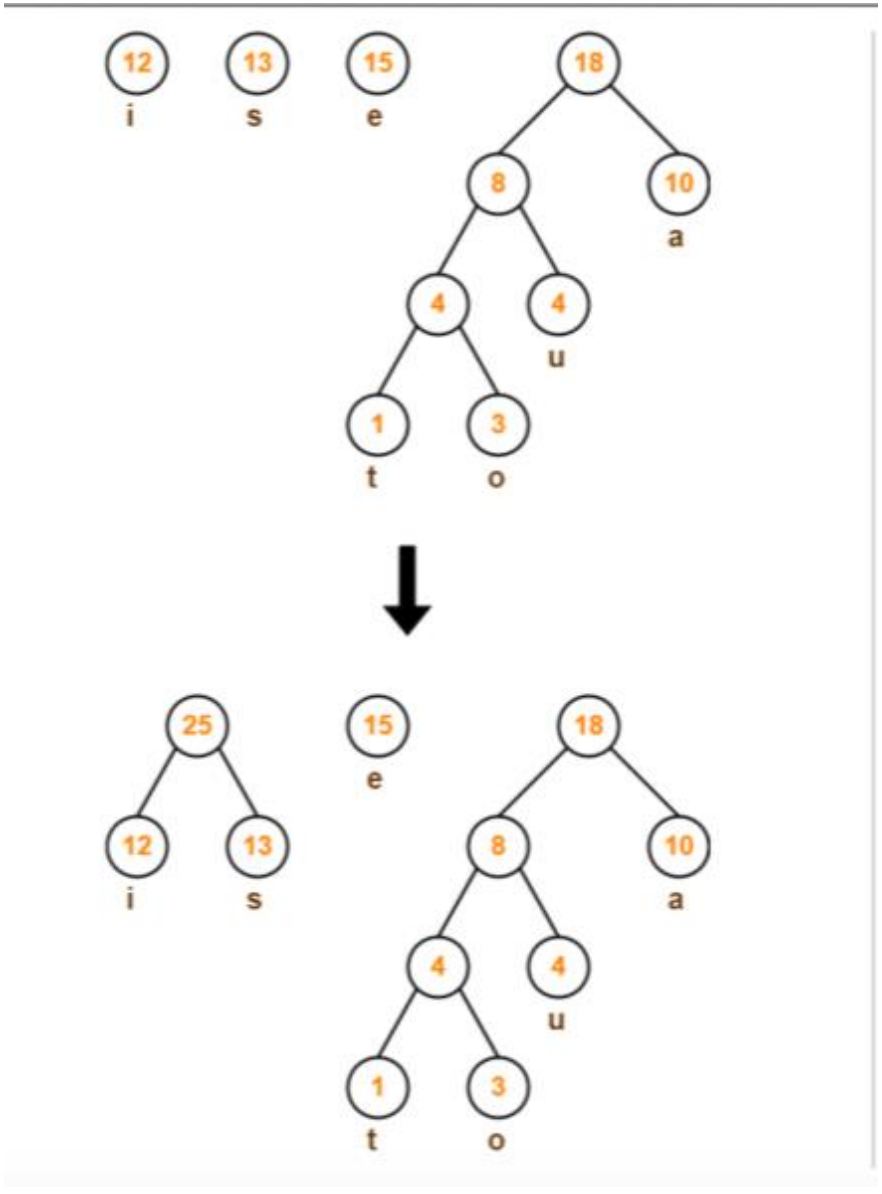


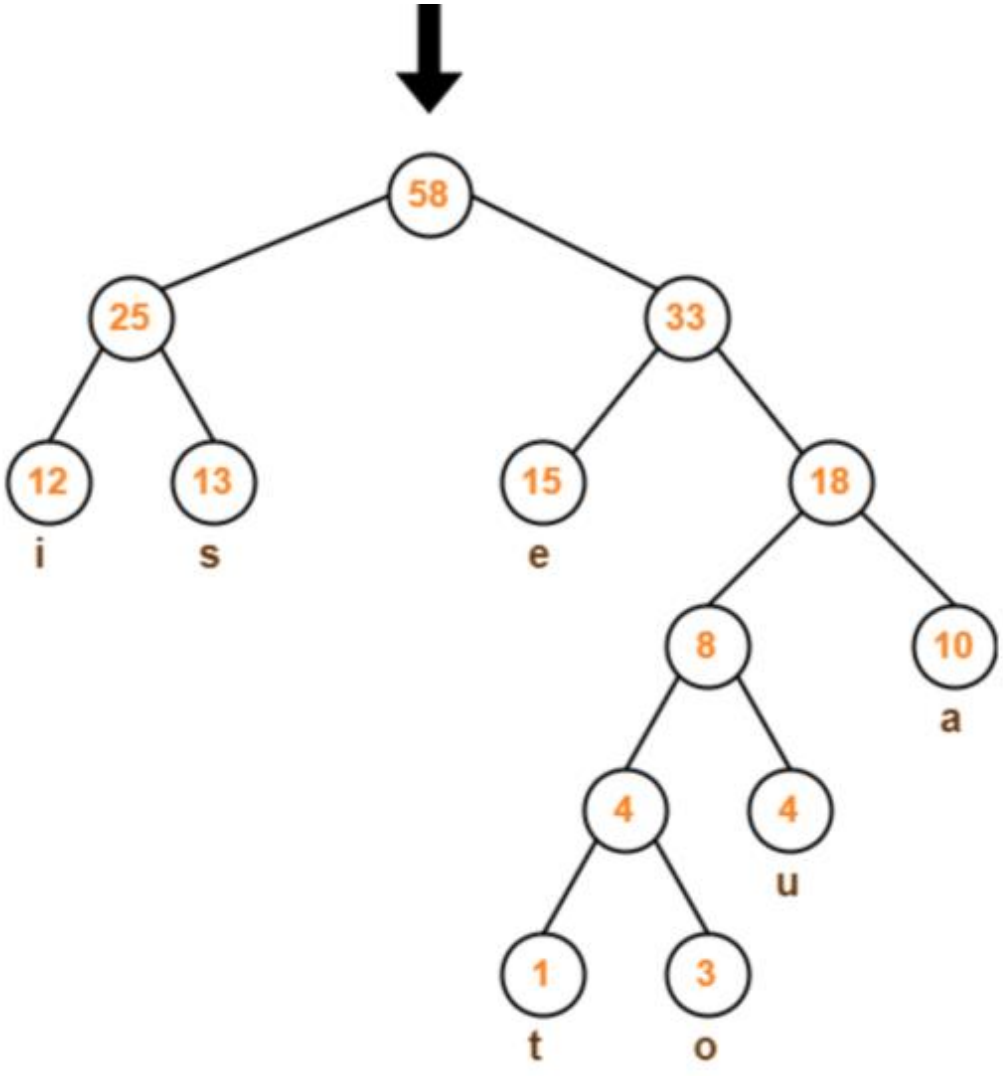
Step-03:



Step-04:

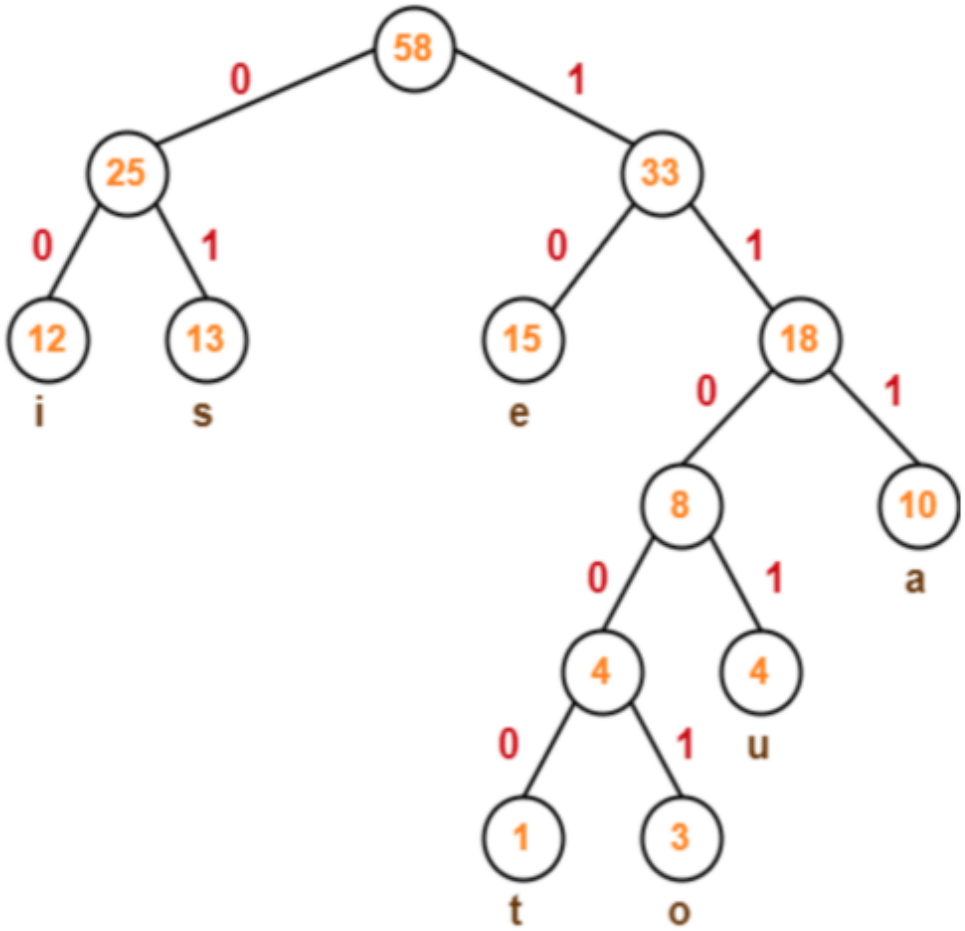



12
i
13
s
15
e
18
8
10
a
4
4
u
1
3
t
o
25
12
i
13
s
15
e
18
8
10
a
4
4
u
1
3
t
o



Huffman Tree

After assigning weight to all the edges, the modified Huffman Tree is-



Huffman Tree

Huffman Code

To write Huffman Code for any character, traverse the Huffman Tree from root node to the leaf node of that character.
Following this rule, the Huffman Code for each character is -

- a = 111
- e = 10
- i = 00
- o = 11001
- u = 1101
- s = 01
- t = 11000

From here, we can observe-

- Characters occurring less frequently in the text are assigned the larger code.
- Characters occurring more frequently in the text are assigned the smaller code.