Design and Analysis of Algorithms

- ***Course Code***: BCSE304L
- ***Course Type***: Theory (ETH)
- ***Slot***: A1+TA1 & & A2+TA2
- ***Class ID***: VL2023240500901
  VL2023240500902

A1+TA1

Day	Start	End
Monday	08:00	08:50
Wednesday	09:00	09:50
Friday	10:00	10:50

A2+TA2

Day	Start	End
Monday	14:00	14:50
Wednesday	15:00	15:50
Friday	16:00	16:50

Syllabus- Module 1

Module:1	Design Paradigms: Greedy, Divide and Conquer Techniques	6 hours

Overview and Importance of Algorithms - Stages of algorithm development: Describing the problem, Identifying a suitable technique, Design of an algorithm, Derive Time Complexity, Proof of Correctness of the algorithm, Illustration of Design Stages -
Greedy techniques: Fractional Knapsack Problem, and Huffman coding
Divide and Conquer: Maximum Subarray, Karatsuba faster integer multiplication algorithm.

<u>Greedy method</u>

- Greedy algorithms build up a solution piece by piece, making a series of choices at each stage. The solution is constructed in stages, and at each stage, the algorithm makes a locally optimal choice.
  - At each step or stage of the algorithm, the greedy approach selects the best available option without considering the consequences of this choice on future steps. This means that the algorithm aims to find a feasible solution at each stage.
    - **Local Optimality**: The algorithm chooses the locally optimal solution in the hope that this will lead to a globally optimal solution. However, there's no guarantee that the solution obtained will always be globally optimal. The approach is inherently myopic, focusing on immediate gains.
  - **No Backtracking**: Greedy algorithms do not revisit or revise their choices. Once a decision is made, it is final and not reconsidered in subsequent steps. This characteristic simplifies the algorithm but can also limit its ability to find the globally optimal solution.
  - **Efficiency**: Greedy algorithms are often efficient and have low time complexity. They are suitable for solving certain types of problems where a locally optimal choice leads to a globally optimal solution.

General Algorithm for Greedy method

- Problem should be solved in stages
- Each stage we will consider one input from a given problem
- If that input is feasible then we will include it in the solution
- so by including all those feasible in inputs we will get an optimal solution.

```
Algorithm Greedy(a,n)
//a[1:n] contains the n inputs.
{
        Solution :=0;
        For i=1 to n do
        {
            X:=select(a);
            If Feasible(solution, x) then
                Solution :=Union(solution, x);
        }
        Return solution;
}
```

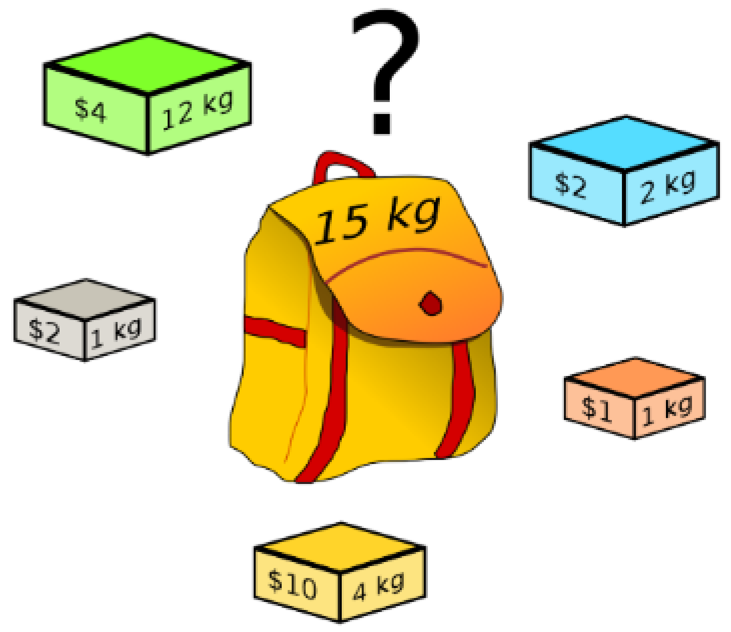
Selection → Function, that selects an input from a[] and removes it. The selected input's value is assigned to x.
Feasible → Boolean-valued function that determines whether x can be included into the solution vector.
Union → function that combines x with solution and updates the objective function.

Application of Greedy Method

- Huffman Coding
- fibonacci heap
- Job sequencing with deadline
- Fractional Knapsack Problem
- Minimum cost spanning trees
- Single source shortest path problem.

0/1 Knapsack Problem Using Dynamic Programming

Recollecting of Knapsack Problem (introduction)



Knapsack Problem

Items should be placed into the knapsack such that-

- The value or profit obtained by putting the items into the knapsack is maximum.
- And the weight limit of the knapsack does not exceed.

knapsack problem

- In the 0-1 Knapsack problem, we are not allowed to break items. We either take the whole item or don’t take it.
- Fractional Knapsack, we can break items for maximizing the total value of knapsack.

Example:
Items as (value, weight) pairs
arr[] = {{60, 10}, {100, 20}, {120, 30}}
Knapsack Capacity, W = 50;

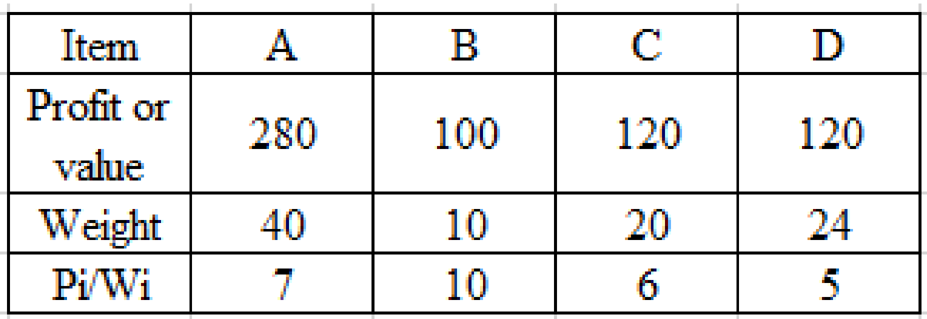
0-1 Knapsack problem Output:
Maximum possible value = 220
by taking items of weight 20 and 30 kg
Hence total price will be 120+100= 220

Fractional Knapsack Output :
Maximum possible value = 240
By taking full items of 10 kg, 20 kg and 2/3rd of last item of 30 kg
60+100+80=240

knapsack problem using Greedy Method

Weights=[40, 10, 20, 24]
Values=[280, 100, 120, 120]
Knapsack capacity (W)=60

- n items
- Weight of ith item $w_i > 0$
- Profit or value for ith item Pi>0
- Capacity of the knapsack is W.



Item	A	B	C	D
Profit or value	280	100	120	120
Weight	40	10	20	24
Pi/Wi	7	10	6	5

X_i of ith item $O \leq X_i \leq 1$

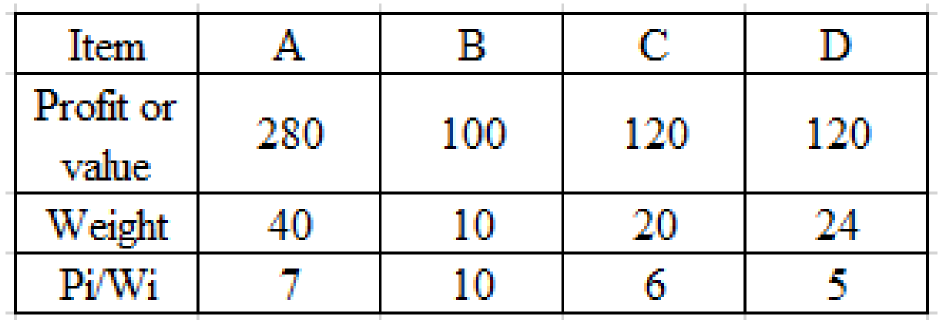
- O→not considered the item
- 1→ Considered the item
- Fractional part of the item → in between 0 or 1.

W=60

Objective function:

Maximize $$\sum_{i=1}^{n} x_i p_i$$

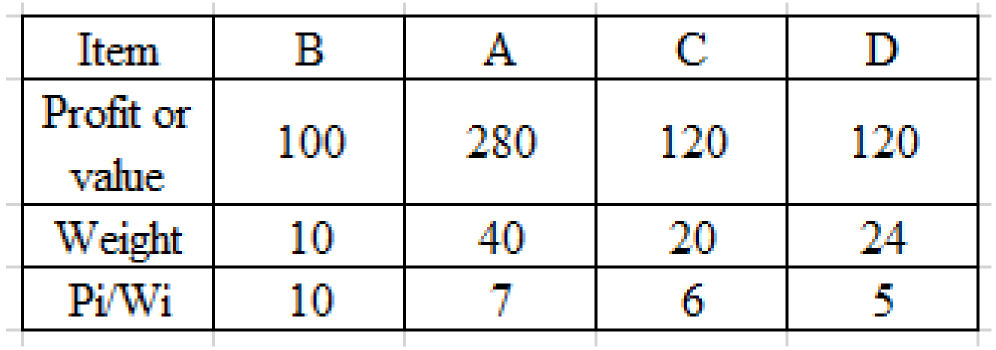
Subject to Constraint $$\sum_{i=1}^{n} x_i w_i \leq W$$



Item	A	B	C	D
Profit or value	280	100	120	120
Weight	40	10	20	24
Pi/Wi	7	10	6	5

W=60

Based on Pi/Wi, given items were sorted as shown below.

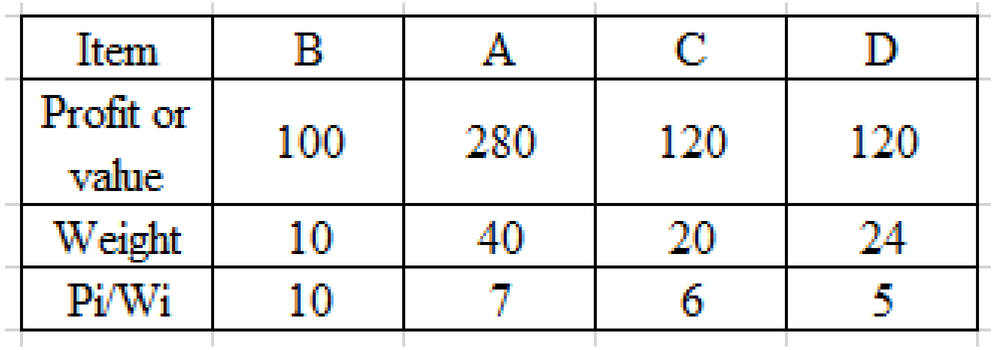


Item	B	A	C	D
Profit or value	100	280	120	120
Weight	10	40	20	24
Pi/Wi	10	7	6	5

10 + 40+ (1/2) 20 =60

100 + 280+ 60 =440

```
Algorithm GreedyKnapsack(m,n)
// p[i:n] and [1:n] contain the profits and weights respectively
// if the n-objects ordered such that p[i]/w[i]>=p[i+1]/w[i+1],
// m is size of knapsack and x[1:n] is the solution vector
{
    For i:=1 to n do x[i]:=0.0
    U:=m;
    For i:=1 to n do
    {
        if(w[i]>U) then break;
        x[i]:=1.0;
        U:=U-w[i];
    }
    If(i<=n) then x[i]:=U/w[i];
}
```



Item	B	A	C	D
Profit or value	100	280	120	120
Weight	10	40	20	24
Pi/Wi	10	7	6	5

```
Algorithm GreedyKnapsack(m,n)
// p[i:n] and [1:n] contain the profits and weights respectively
// if the n-objects ordered such that p[i]/w[i]>=p[i+1]/w[i+1],
// m is size of knapsack and x[1:n] is the solution vector
{
For i:=1 to n do x[i]:=0.0
U:=m;
For i:=1 to n do
{
if(w[i]>U) then break;
x[i]:=1.0;
U:=U-w[i];
}
If(i<=n) then x[i]:=U/w[i];
}
```

Item	B	A	C	D
Profit or value	100	280	120	120
Weight	10	40	20	24
Pi/Wi	10	7	6	5

m = W = 60

1, 2, 3 – 10/20

100 280

100+280+60 = 440

x_i = 1, 2

B A

x_i = 1, 2,

10 → 60

U = 60 − 10 = 50

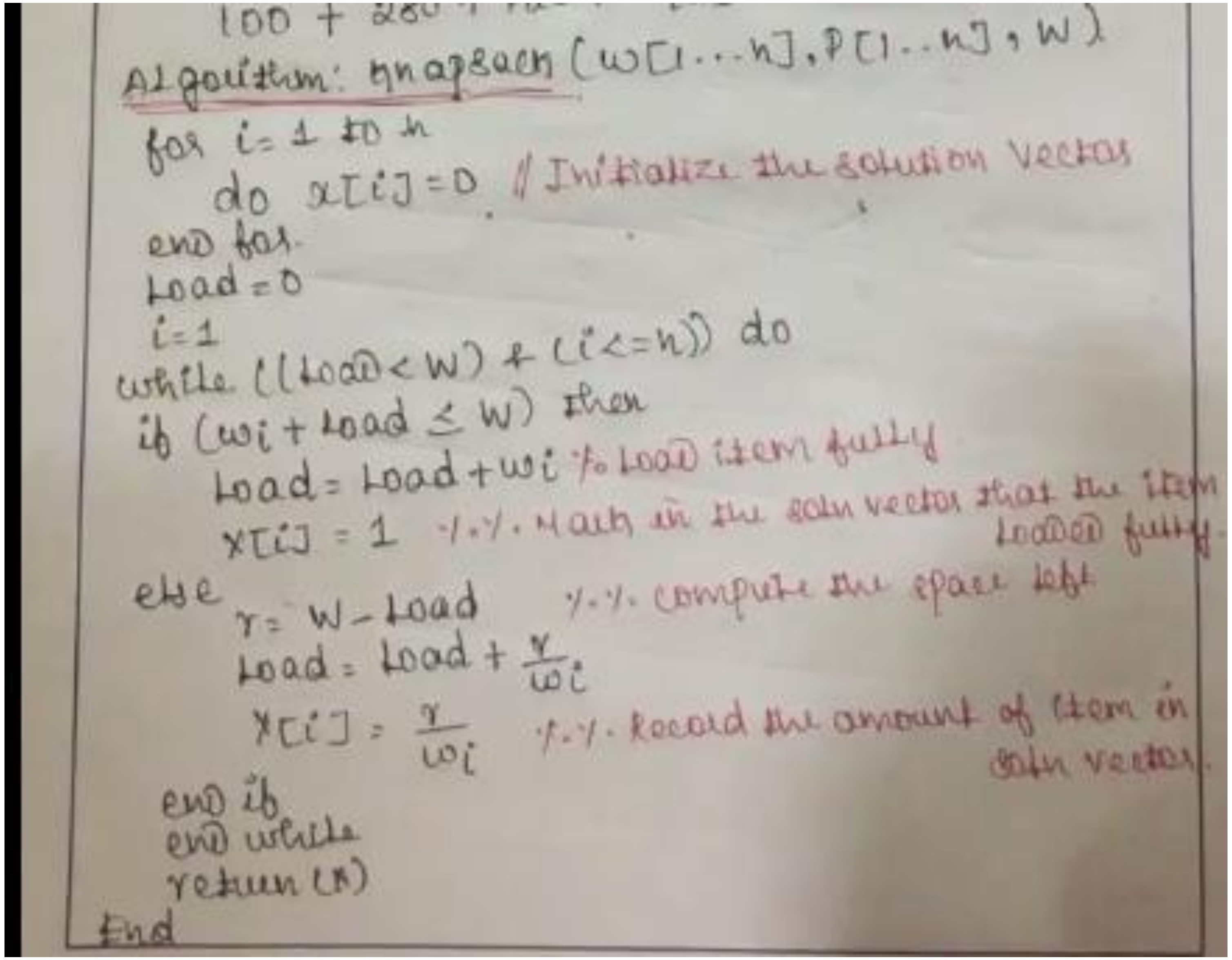
U = 50 − 40 = 10

20 > 10

60 − 40

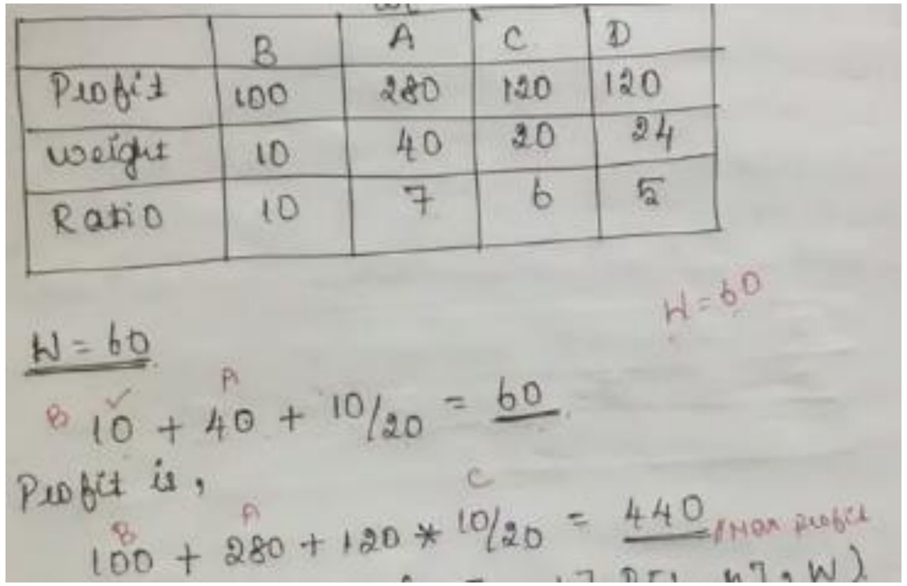
10/20

1, 2, 10/20



$100 + 280$

```
Algorithm: Knapsack (w[1...n], P[1..n], W)
 for i = 1 to n
     do x[i] = 0   // Initialize the solution vector
 end for
 Load = 0
 i = 1
while ((Load < W) & (i <= n)) do
 if (w_i + Load ≤ W) then
     Load = Load + w_i   %. Load item fully
     x[i] = 1   %.%. Mark in the soln vector that the item loaded fully.
 else
     r = W - Load   %.%. compute the space left
     Load = Load + r/w_i
     x[i] = r/w_i   %.%. Record the amount of item in soln vector.
 end if
 end while
 return (x)
End
```



	B	A	C	D
Profit	100	280	120	120
weight	10	40	20	24
Ratio	10	7	6	5

$W = 60$ $W=60$

$\overset{B\ ✓}{10} + \overset{A}{40} + 10/20 = \underline{60}$

Profit is,

$\overset{B}{100} + \overset{A}{280} + \overset{C}{120} * 10/20 = \underline{440}$ // Max profit

Important notes about **Fractional Knapsack in Greedy Method**

Definition: Fractional Knapsack is a classic optimization problem where the goal is to select items with maximum total value in a knapsack (a container) without exceeding its capacity. In the fractional version, it is allowed to take fractions of items, optimizing the total value-to-weight ratio.

Procedure:

- Sort Items: Sort items based on their value-to-weight ratio in descending order.
- Greedy Selection: Iterate through the sorted items and greedily select fractions of items until the knapsack is full.

Sample Input and Output:

- Input: Items: [(v1, w1), (v2, w2), ..., (vn, wn)] (value, weight)
- Knapsack capacity: W
- Output: Maximum total value that can be obtained.

Important notes about **Fractional Knapsack in Greedy Method**

Time Complexity in Brute Force Approach: In a brute force approach, you would consider all possible combinations of items (including fractions) and calculate the total value for each combination. This results in a time complexity of **O(2^n)**, where n is the number of items.

Time Complexity in Greedy Techniques: The time complexity of the greedy approach is **O(n log n)**, where n is the number of items. The dominant factor comes from the sorting step, as the greedy selection process is linear.

Recurrence Relation: The recurrence relation for Fractional Knapsack in the greedy approach can be expressed as: T(n)=Tsort(n)+Tselection(n)

where:

- Tsort(n) is the time complexity of sorting the items (typically O(nlogn)).
- Tselection(n) is the time complexity of the greedy selection process.

Important notes about **Fractional Knapsack in Greedy Method**

Proof of Time Complexity:

- The proof involves analyzing the time complexity of the sorting step and the greedy selection step:
- Sorting Step (Tsort(n)): Sorting the items based on value-to-weight ratio is typically O(nlogn) using efficient sorting algorithms like quicksort or mergesort.
- Greedy Selection Step (Tselection(n)): The greedy selection process involves iterating through the sorted items and making decisions based on the value-to-weight ratio. Since each item is considered only once during this process, the selection step is linear, O(n).
- Therefore, the overall time complexity is the sum of the sorting and selection complexities:
  T(n)=O(nlogn)+O(n)
- Since O(nlogn) dominates O(n) for large values of n, the overall time complexity is O(nlogn).
- In conclusion, the O(nlogn) time complexity for Fractional Knapsack in the greedy approach is mainly due to the sorting step, which sorts the items based on their value-to-weight ratio.

Huffman Coding

- Huffman Coding is a technique of compressing data to reduce its size without losing any of the details. That why it also called as lossless data compression mechanism.
    - It was first developed by David Huffman.
    - It is widely used in image (JPEG or JPG) compression.
- **Technique used**: Huffman Coding is generally useful to compress the data in which there are frequently occurring characters.
- In computer each character is a sequence of 0's and 1's and stores using 8-bits. The mechanism is called fixed-length encoding because each character uses the same number of fixed-bit storage.
- By using variable-length encoding it possible to reduce the amount of space required to store a character.

Huffman Coding

B	C	A	A	D	D	D	C	C	A	C	A	C	A	C

Each character occupies 8 bits. There are a total of 15 characters in the above string. Thus, a total of 8 * 15 = 120 bits are required to send this string.

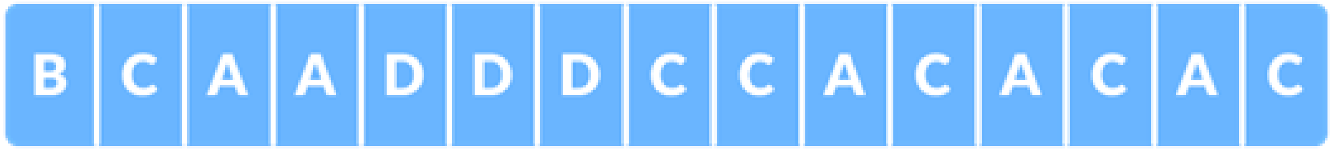
- Huffman Coding technique, we can compress the string to a smaller size.
- Huffman coding first creates a tree using the frequencies of the character and then generates code for each character.
- Once the data is encoded, it has to be decoded. Decoding is done using the same tree.

Huffman Coding prevents any ambiguity in the decoding process using the concept of prefix code ie. a code associated with a character should not be present in the prefix of any other code.

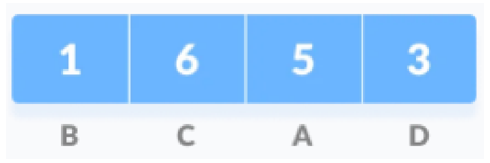
Huffman Coding Algorithm

```
create a priority queue Q consisting of each unique character.
sort then in ascending order of their frequencies.
for all the unique characters:
    create a newNode
    extract minimum value from Q and assign it to leftChild of newNode
    extract minimum value from Q and assign it to rightChild of newNode
    calculate the sum of these two minimum values and assign it to the value of newNode
    insert this newNode into the tree
return rootNode
```

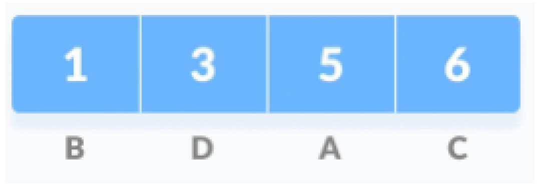
Huffman Coding



S1: Calculate the frequency of each character in the string



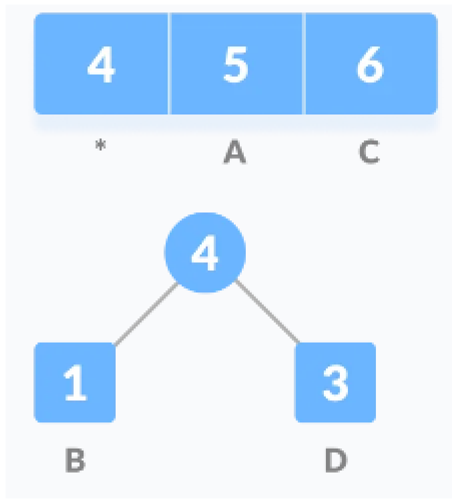
S2: Sort the characters in increasing order of the frequency. These are stored in a priority queue Q.



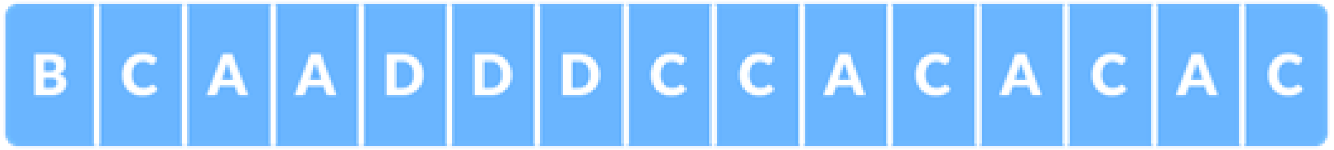
S3: Make each unique (min frequent) character as a leaf node.

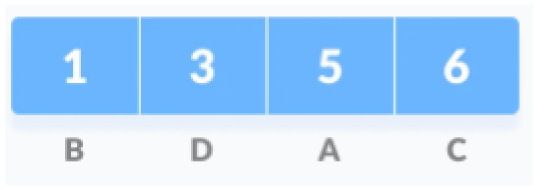
S4: Create an empty node z. Assign the minimum frequency to the left child of z and assign the second minimum frequency to the right child of z. Set the value of the z as the sum of the above two minimum frequencies.

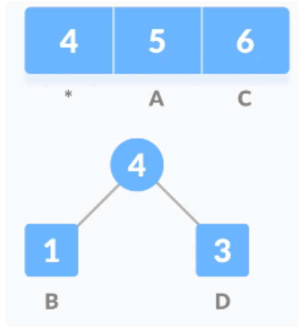
S5: Remove these two minimum frequencies from Q and add the sum into the list of frequencies (* denote the internal nodes in the figure).



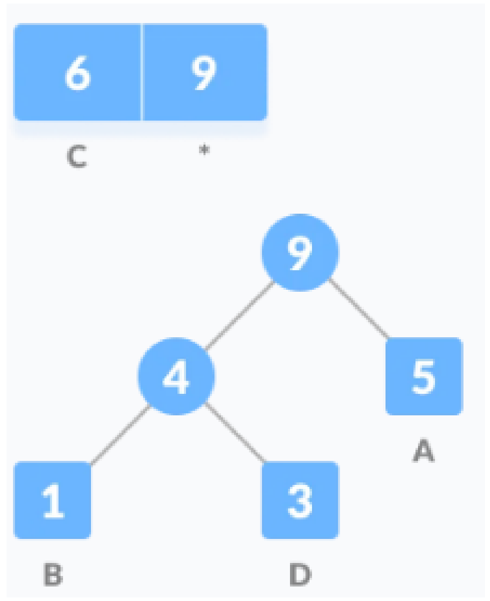
Huffman Coding

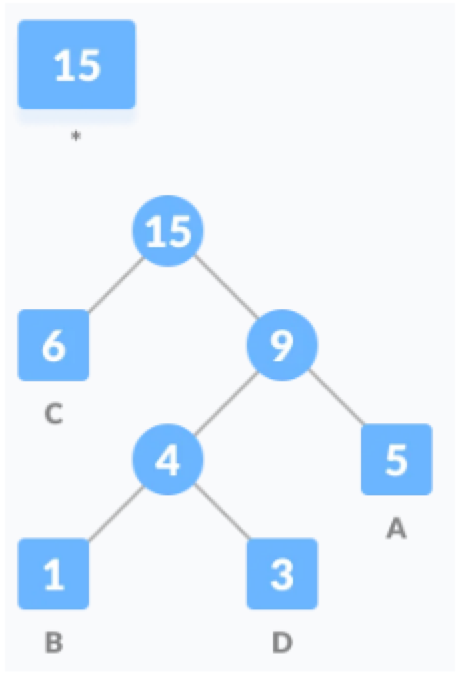




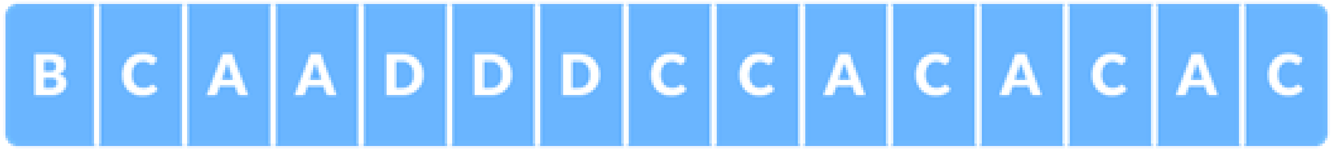


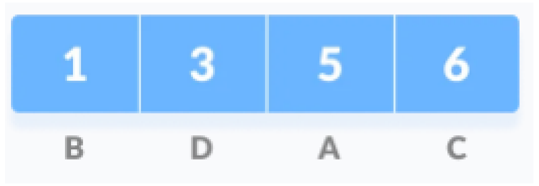
S6: Insert node z into the tree.
S7: Repeat steps 3 to 5 for all the characters.

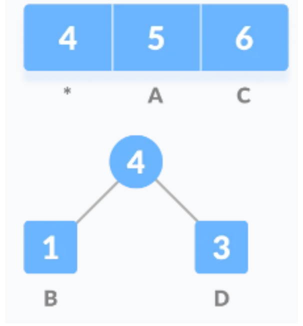


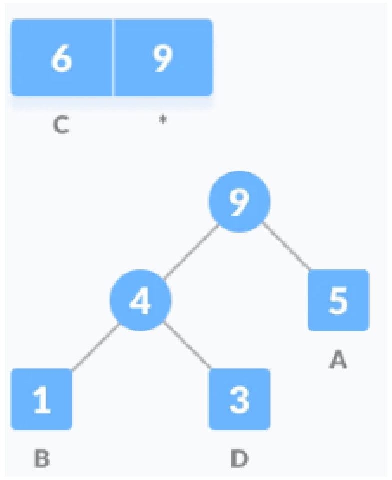


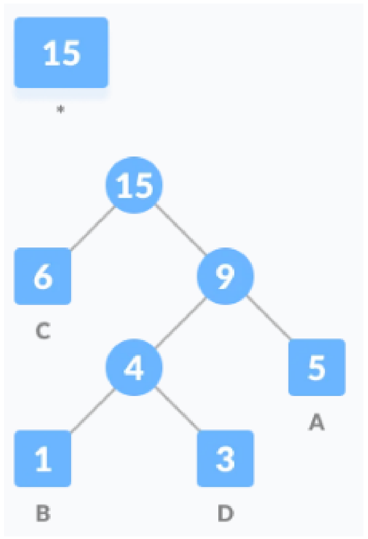
Huffman Coding



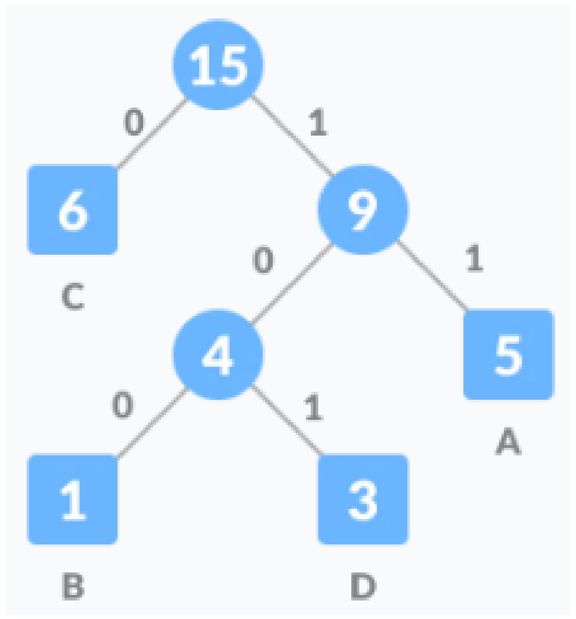






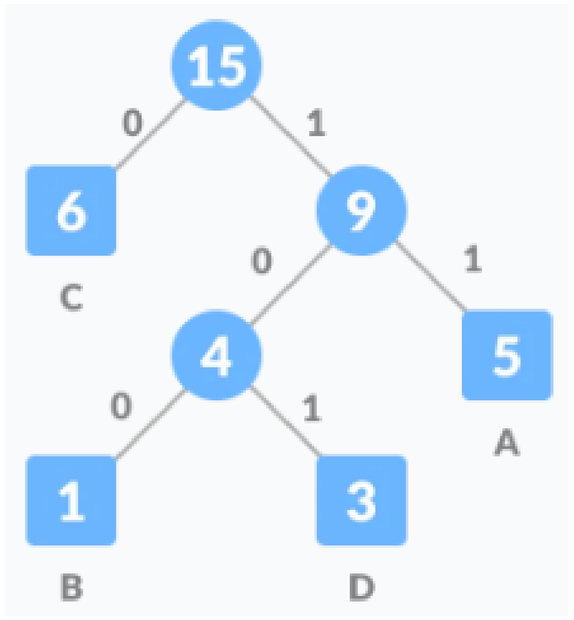


S8: Assign 0 to the left edge and 1 to the right edge.



Huffman Coding

B	C	A	A	D	D	D	C	C	A	C	A	C	A	C



Character	Frequency	Code	Size
A	5	11	5*2 = 10
B	1	100	1*3 = 3
C	6	0	6*1 = 6
D	3	101	3*3 = 9
4 * 8 = 32 bits			28 bits

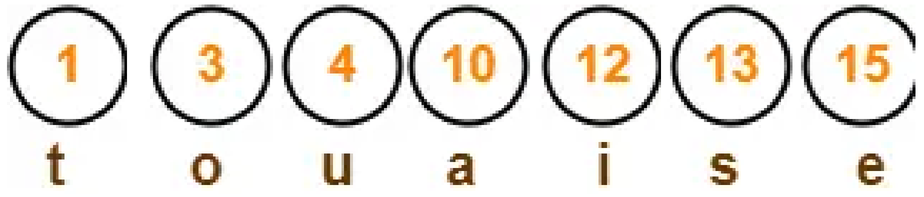
Without encoding, the total size of the string was 120 bits.
After encoding the size is reduced to 32 + 15 + 28 = 75.

Src: Programiz

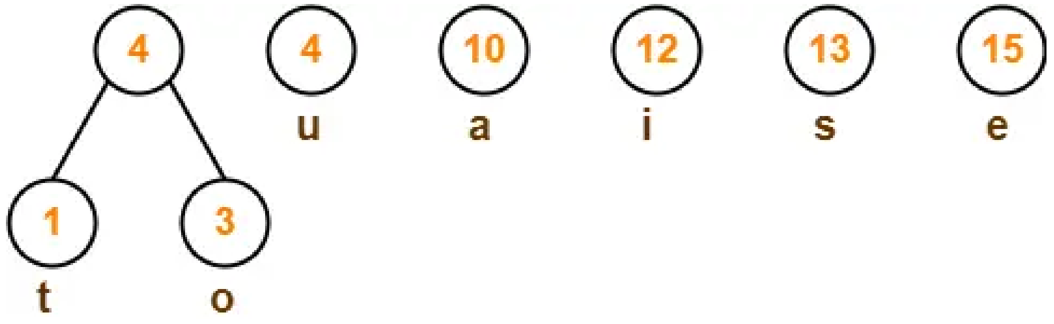
Huffman Coding Example-2

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

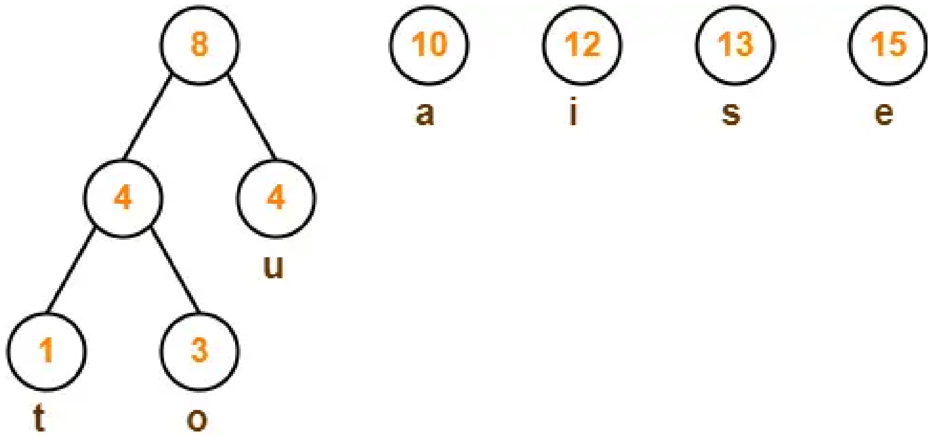
Step-01



Step-02



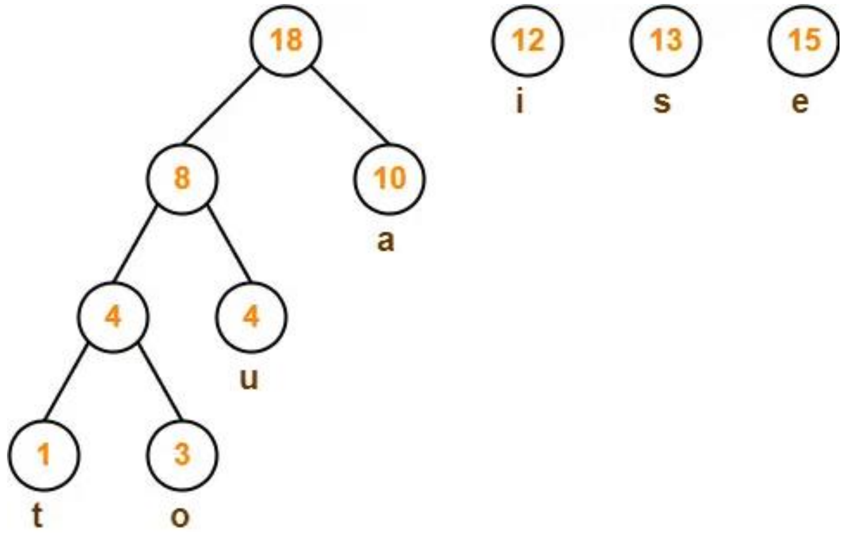
Step-03



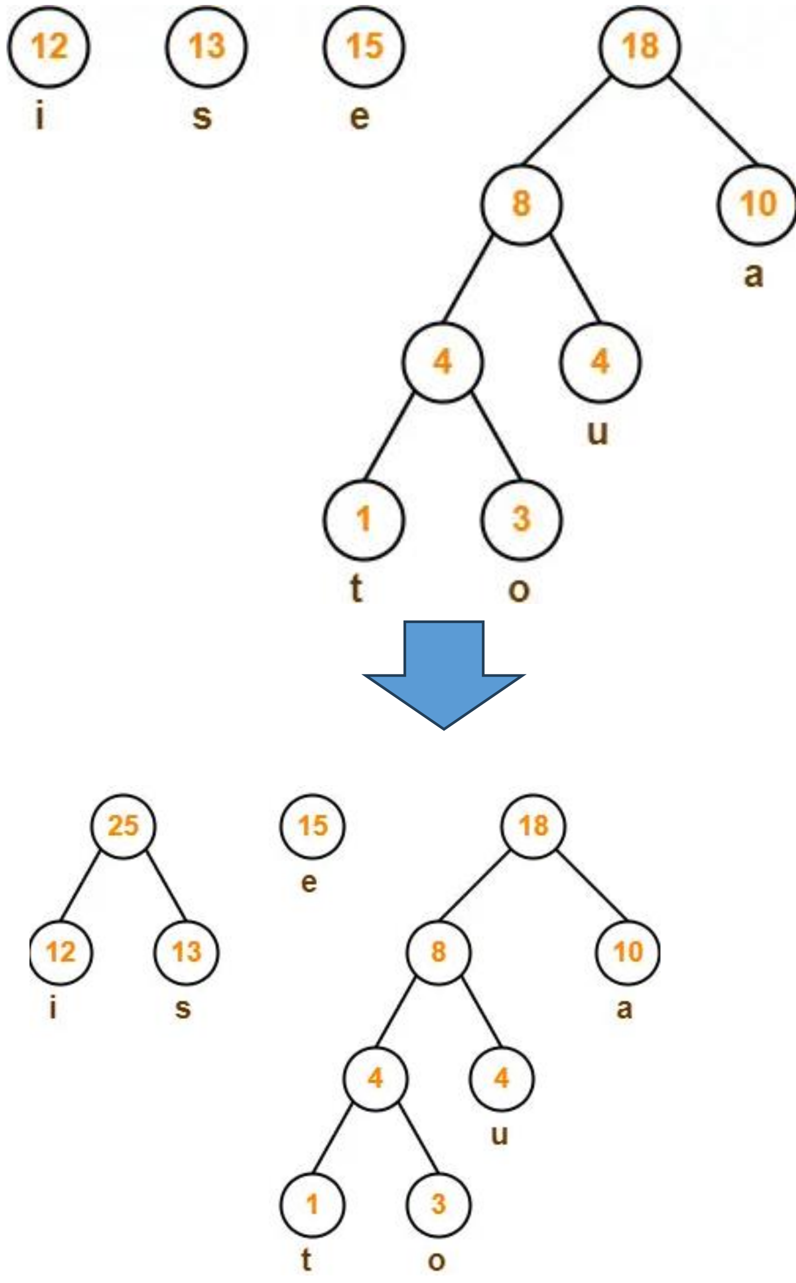
Huffman Coding Example-2

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

Step-04



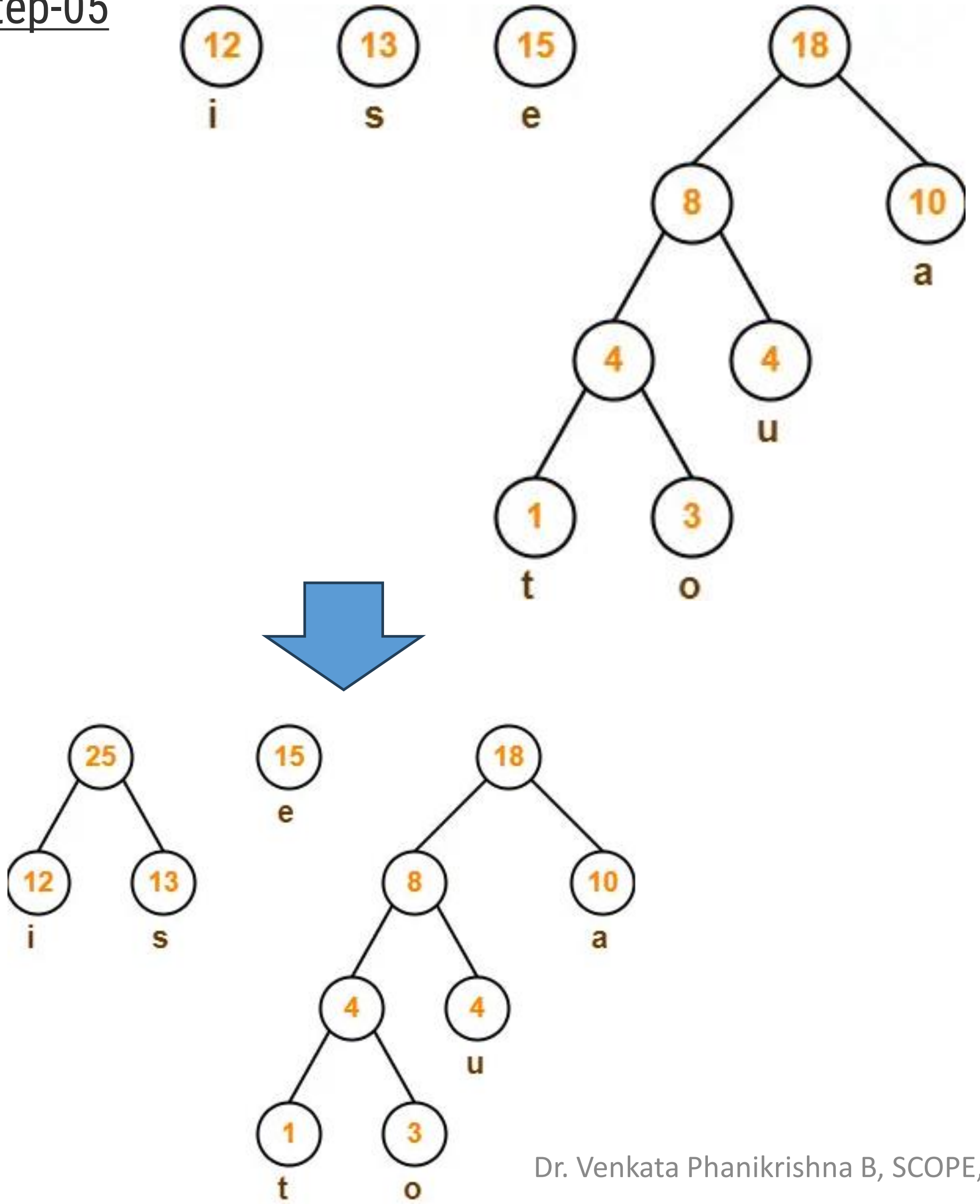
Step-05



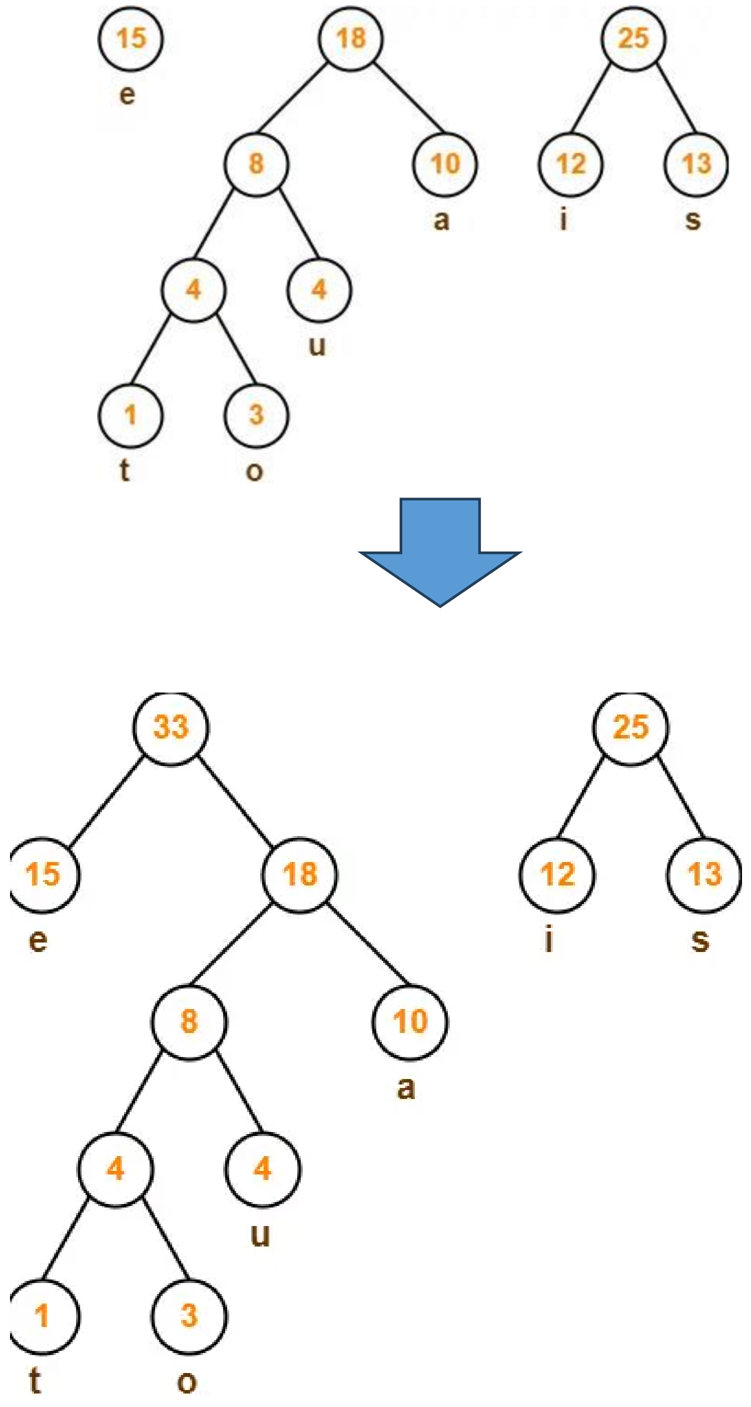
Huffman Coding Example-2

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

Step-05



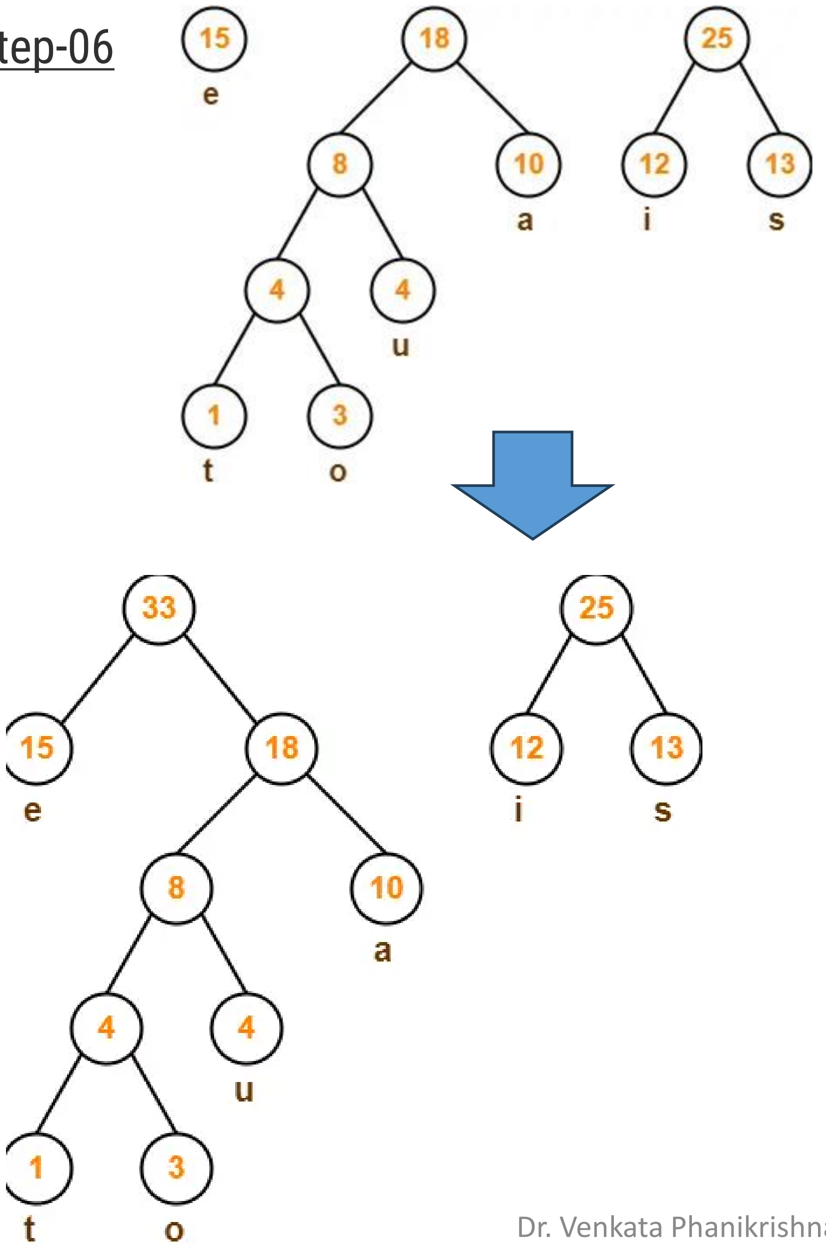
Step-06



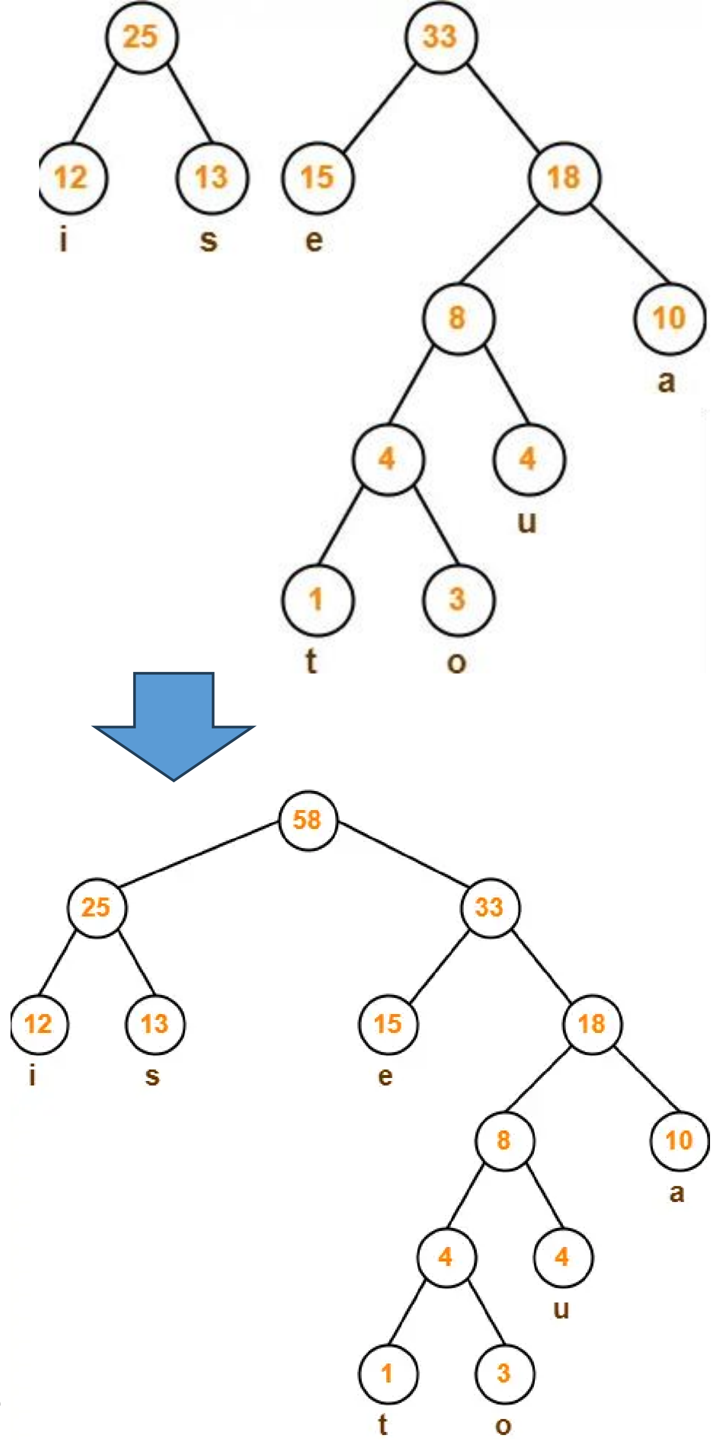
Huffman Coding Example-2

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

Step-06

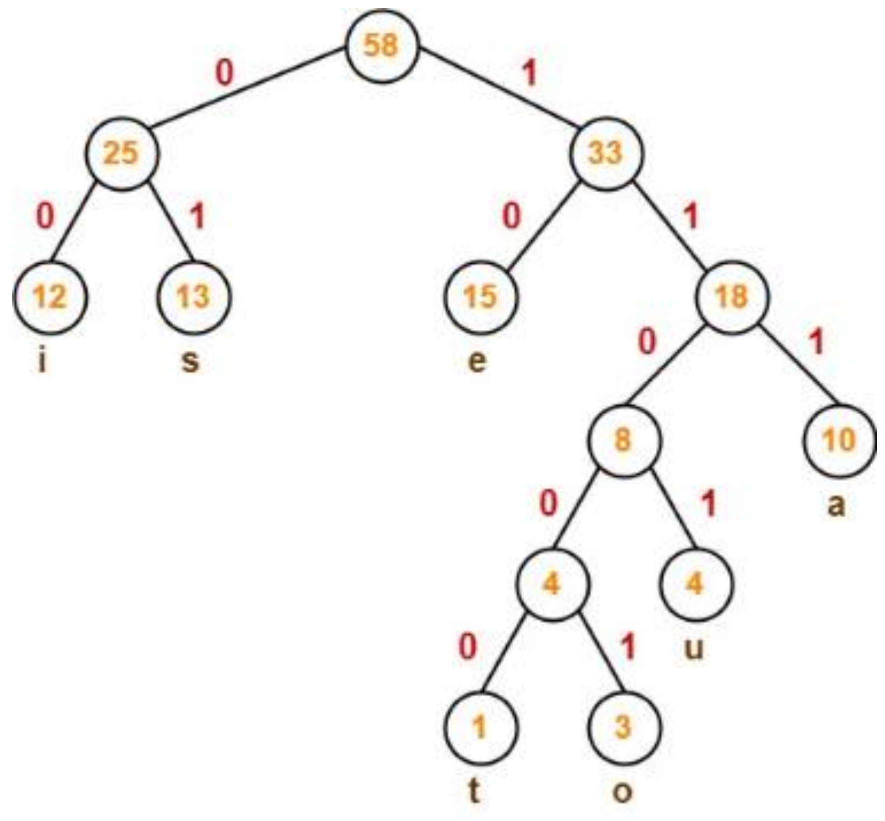


Step-07



Huffman Coding Example-2

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



Huffman Tree

.Ch	Fre	Code	Size
T	1	11000	1*5=5
O	3	11001	3*5=15
U	4	1101	4*4=16
A	10	111	10*3=30
I	12	00	12*2=24
S	13	01	13*2=26
E	15	10	15*2=30
7*8=56	58		146

Huffman Coding Example-2

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

.Ch	Fre	Code	Size
T	1	11000	1*5=5
O	3	11001	3*5=15
U	4	1101	4*4=16
A	10	111	10*3=30
I	12	00	12*2=24
S	13	01	13*2=26
E	15	10	15*2=30
7*8=56	58		146

$$\text{Average code length per character} = \frac{\Sigma(\text{frequency}_i \times \text{code length}_i)}{\Sigma\,\text{frequency}_i}$$

$$= \Sigma(\text{probability}_i \times \text{code length}_i)$$

Average code length

$= \sum(\text{frequency}_i \text{ x code length}_i) / \sum(\text{frequency}_i)$

= { (10 x 3) + (15 x 2) + (12 x 2) + (3 x 5) + (4 x 4) + (13 x 2) + (1 x 5) } / (10 + 15 + 12 + 3 + 4 + 13 + 1)

= 146/58=2.517=2.52

Total number of bits in Huffman encoded message

= Total number of characters in the message x Average code length per character

=56*2.52=141.2

Huffman Coding Example-2

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

.Ch	Fre	Code	Size
T	1	11000	1*5=5
O	3	11001	3*5=15
U	4	1101	4*4=16
A	10	111	10*3=30
I	12	00	12*2=24
S	13	01	13*2=26
E	15	10	15*2=30
7*8=56	58		146

Without encoding, the total size of the string was 58*8=464 bits.

After encoding the size is reduced to 56 + 58 + 146 =260

Important notes about **Huffman coding in greedy techniques**

Procedure:

- **Frequency Calculation**: Calculate the frequency of each character in the input data.
- **Priority Queue**: Create a priority queue (min-heap) based on the frequencies of the characters.
- **Build Huffman Tree**: While there is more than one node in the queue, repeatedly extract two nodes with the lowest frequencies, create a new internal node with these two nodes as children, and insert the new node back into the queue.
- **Encoding**: Assign binary codes to each character based on the Huffman tree. The code for each character is obtained by traversing the tree from the root to the leaf corresponding to that character, with left branches representing 0 and right branches representing 1.
- **Compression**: Replace each character in the input data with its corresponding Huffman code to generate the compressed output.

Important notes about **Huffman coding in greedy techniques**

- **Time Complexity in Brute Force Approach**: In a brute force approach, you would consider all possible binary trees and calculate the encoding cost for each. This would result in a time complexity of **O(2^n)**, where n is the number of characters or symbols in the input data.

- **Time Complexity in Greedy Techniques**: The greedy approach constructs the Huffman tree in **O(n log n)** time, where n is the number of unique symbols in the input. The use of a priority queue speeds up the process of finding the minimum frequency nodes. The overall time complexity is dominated by the heap operations during the tree construction.

Important notes about **Huffman coding in greedy techniques**

Recurrence Relation: Let's denote T(n) as the time complexity of Huffman coding for n symbols. The recurrence relation for Huffman coding can be expressed as follows: T(n)=T1(n)+T2(n)

where:

- T1(n) is the time complexity of building the initial priority queue.
- T2(n) is the time complexity of constructing the Huffman tree.

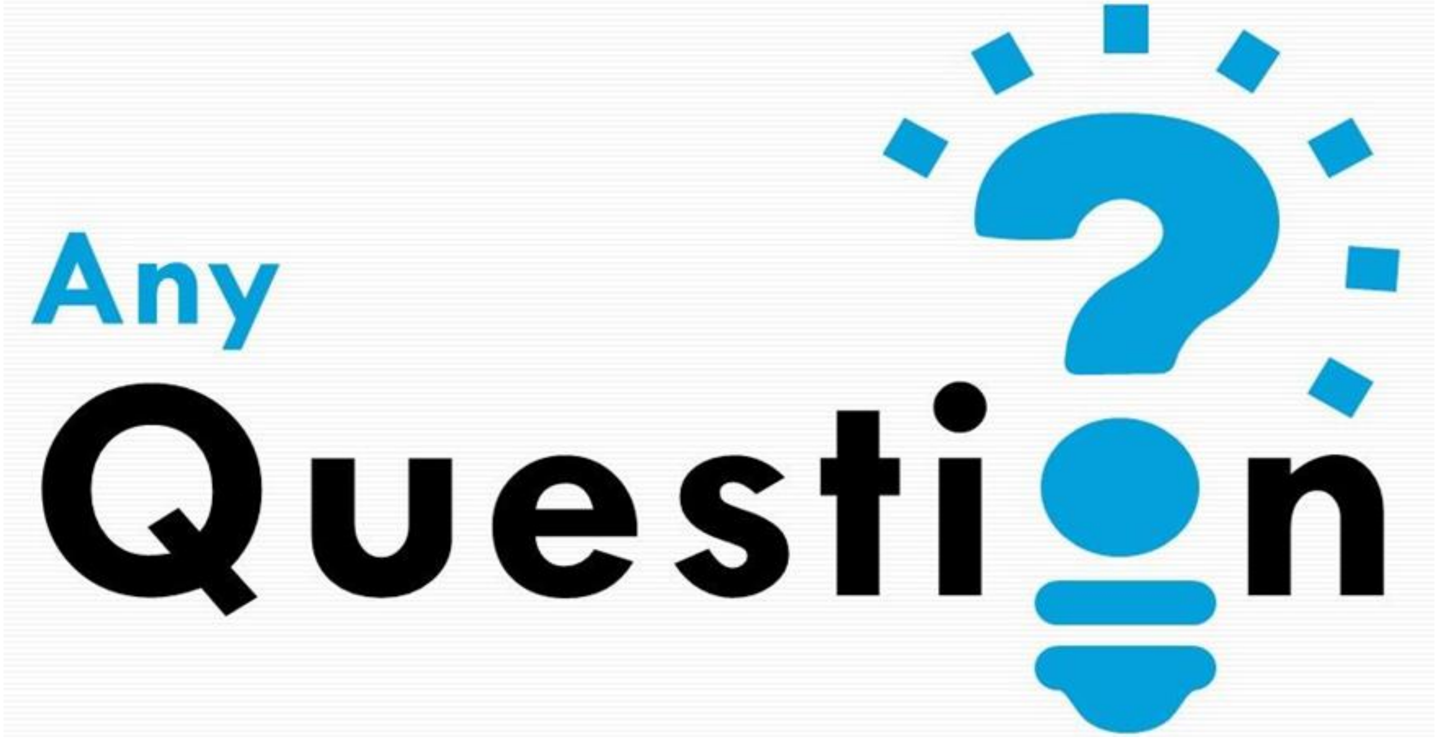
Building the Initial Priority Queue: The time complexity of building the initial priority queue with n symbols is typically O(n). This involves counting the frequencies of each symbol and creating a priority queue based on these frequencies. T1(n)=O(n)

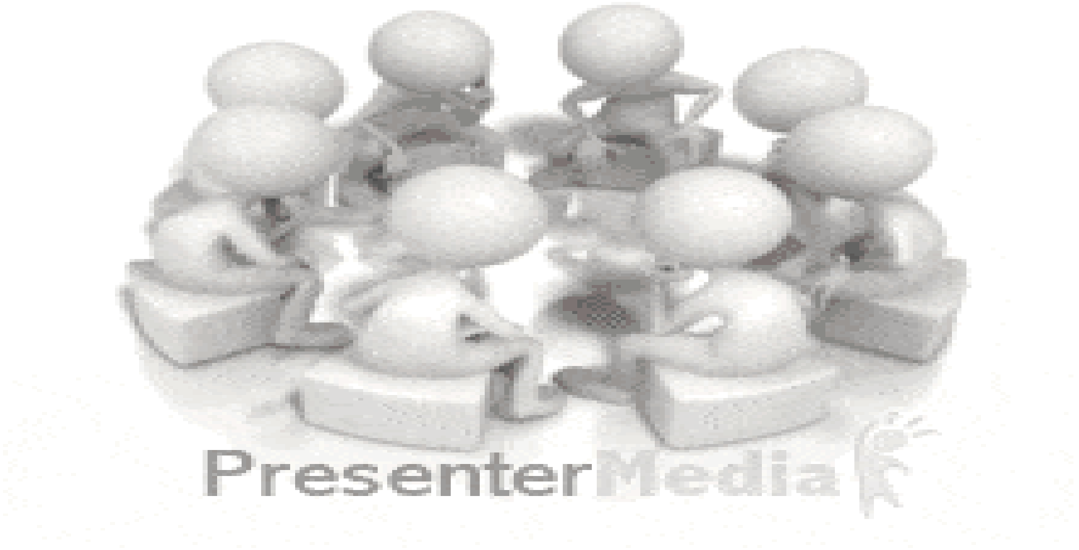
Constructing the Huffman Tree: The time complexity of constructing the Huffman tree is dominated by the operations of extracting the minimum frequency nodes and inserting new nodes back into the priority queue. In a proper min-heap implementation, these operations take O(logn) time. T2(n)=(n−1)⋅O(logn)

Overall Time Complexity: Combining the two components, the overall time complexity T(n) for Huffman coding is: T(n)=O(n)+(n−1)⋅O(logn)

Simplifying and expressing it in big-O notation, we get: T(n)=O(nlogn)

This indicates that the time complexity of Huffman coding is O(nlogn), where n is the number of symbols or characters in the input. The logarithmic factor comes from the operations performed during the construction of the Huffman tree using a priority queue.


Any
Question


PresenterMedia

Thank You!

FOR YOUR
ATTENTION