Dynamic Programming

Dynamic programming approach is similar to divide and conquer in breaking down the problem into smaller and yet smaller possible sub-problems. But unlike, divide and conquer, these sub-problems are not solved independently. Rather, results of these smaller sub-problems are remembered and used for similar or overlapping sub-problems.

Dynamic programming is used where we have problems, which can be divided into similar sub-problems, so that their results can be re-used. Mostly, these algorithms are used for optimization. Before solving the in-hand sub-problem, dynamic algorithm will try to examine the results of the previously solved sub-problems. The solutions of sub-problems are combined in order to achieve the best solution.

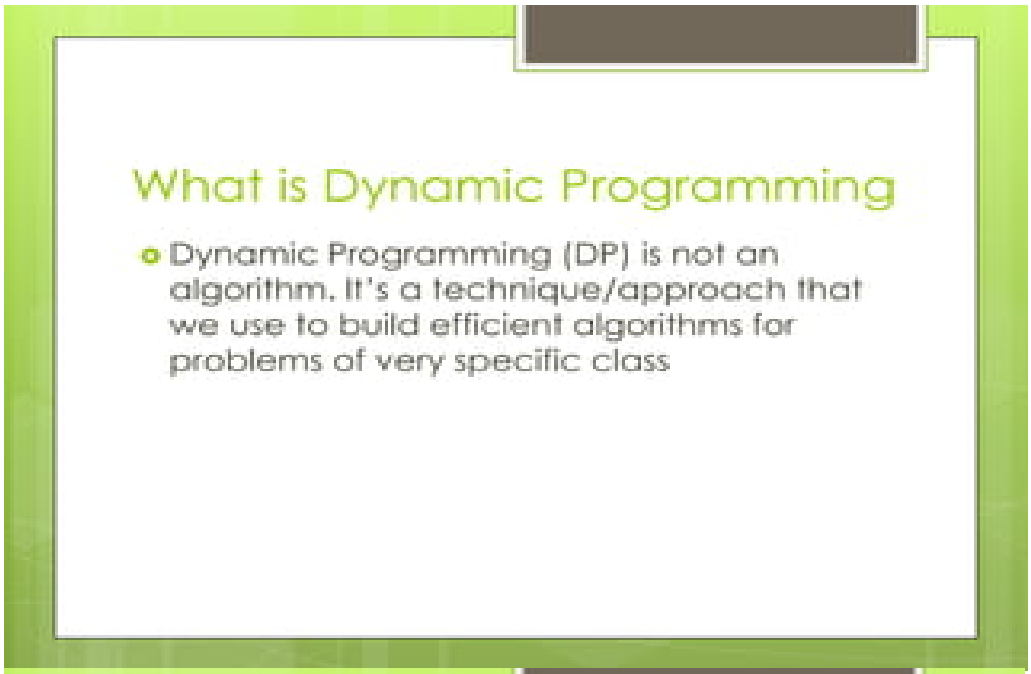
Matrix Chain Multiplication

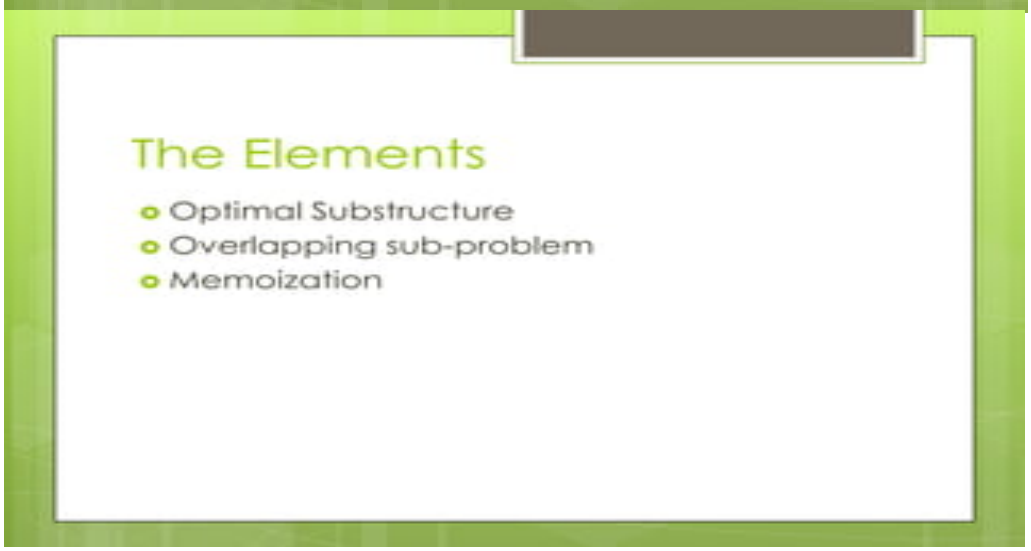
Given the dimension of a sequence of matrices in an array **arr[]**, where the dimension of the **i^{th}** matrix is **(arr[i-1] * arr[i])**, the task is to find the most efficient way to multiply these matrices together such that the total number of element multiplications is minimum.

Examples:

Input: *arr[] = {40, 20, 30, 10, 30}*
Output: *26000*
Explanation:*There are 4 matrices of dimensions 40×20, 20×30, 30×10, 10×30.*
Let the input 4 matrices be A, B, C and D.
The minimum number of multiplications are obtained by putting parenthesis in following way (A(BC))D.
*The minimum is 20*30*10 + 40*20*10 + 40*10*30*
Input: *arr[] = {1, 2, 3, 4, 3}*
Output: *30*
Explanation: *There are 4 matrices of dimensions 1×2, 2×3, 3×4, 4×3.*

Let the input 4 matrices be A, B, C and D.
The minimum number of multiplications are obtained by putting parenthesis in following way ((AB)C)D.
*The minimum number is 1*2*3 + 1*3*4 + 1*4*3 = 30*
***Input:** arr[] = {10, 20, 30}*
***Output:** 6000*
***Explanation:** There are only two matrices of dimensions 10×20 and 20×30.*
*So there is only one way to multiply the matrices, cost of which is 10*20*30*





What is Substructure?

- A substructure is a structure itself that helps a bigger structure of the same kind to exist.
- The substructure does not play an auxiliary role
- It is an essential part of the super structure
- It is only smaller in size compared to the super structure
- It has the same properties of the superstructure

Optimal Substructure

- So optimal substructure is simply the optimal selection among all the possible substructures that can help a super structure of the same kind to exist
- Example: To know how to multiply n matrices optimally we must multiply the last matrix with the optimal multiplication result of the n-1 other matrices. The base case for the recursion here is that the optimal parenthesization of

Optimal Substructure

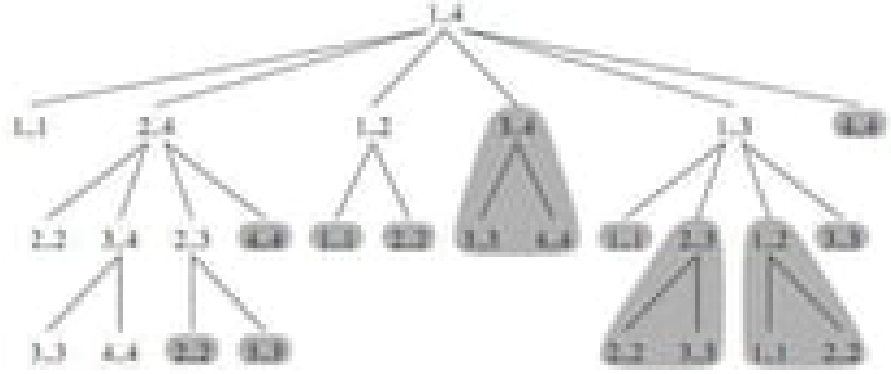
- To build a building with the best possible strength, we need to build each level as optimally as possible
- To solve a mathematical problem, we all the smaller problems inside that problem to have the correct result. Each problem might depend on the previous problem solved. So we need to take care of all the problems optimally, with correct calculations.

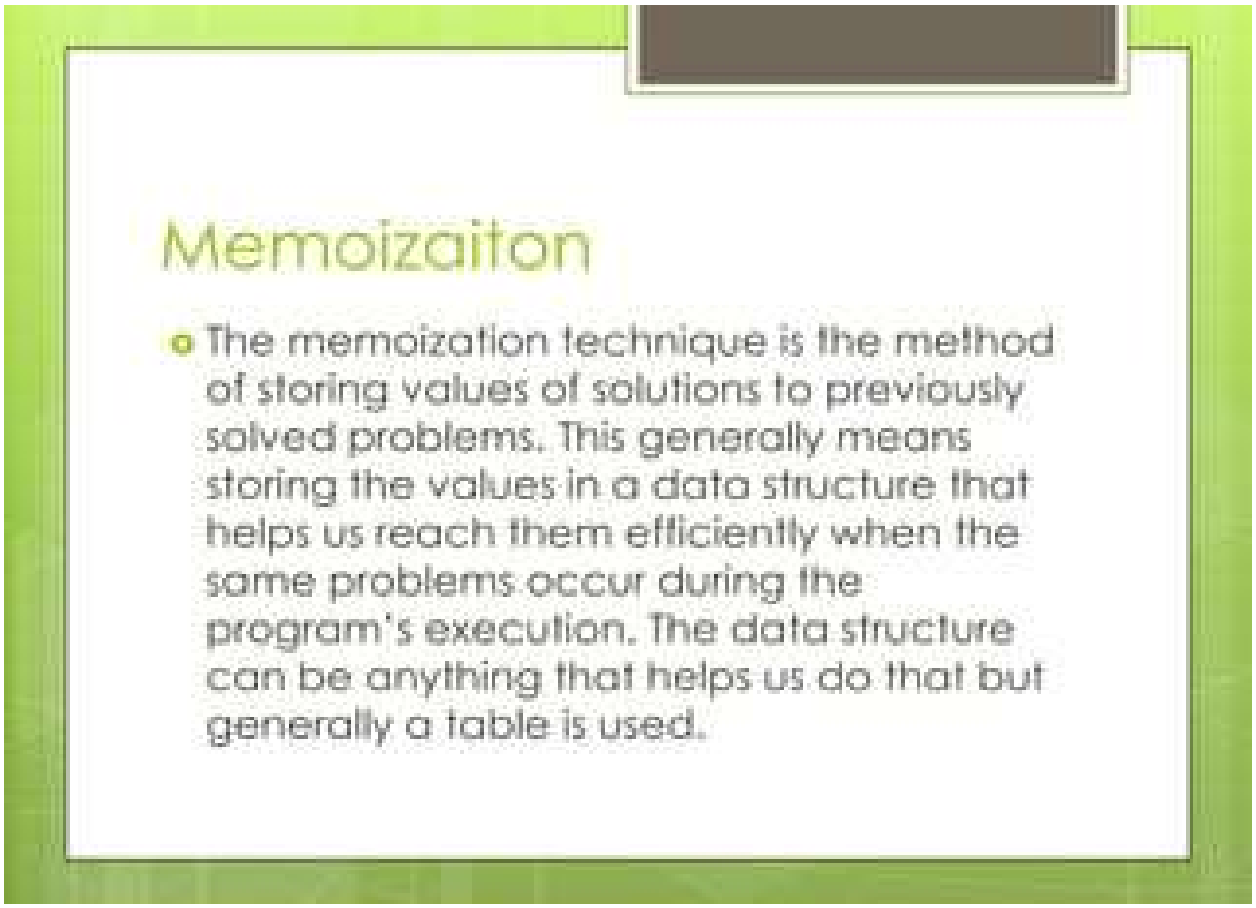
Overlapping Sub-problem

- Overlapping sub-problem is found in those problems where bigger problems share the same smaller problems. This means, while solving larger problems through their sub-problems we find the same sub-problems in two or more different large problems. In these cases a sub-problem is usually found to be solved previously.

Overlapping Sub-problem

- Overlapping sub-problems can be found in Matrix-Chain-Multiplication (MCM) problem.





Using Dynamic Programming to find the LCS

Let us take two sequences:

X | A | C | A | D | B

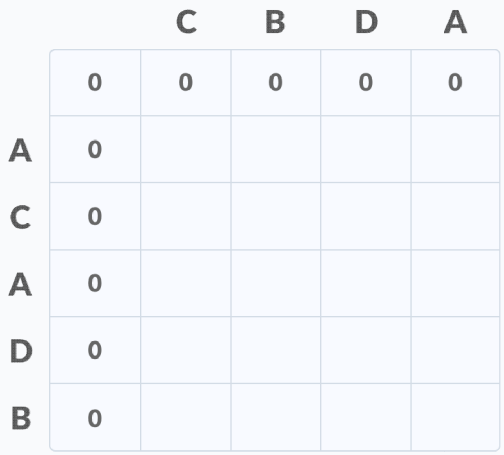
The first sequence

Y | C | B | D | A

Second Sequence

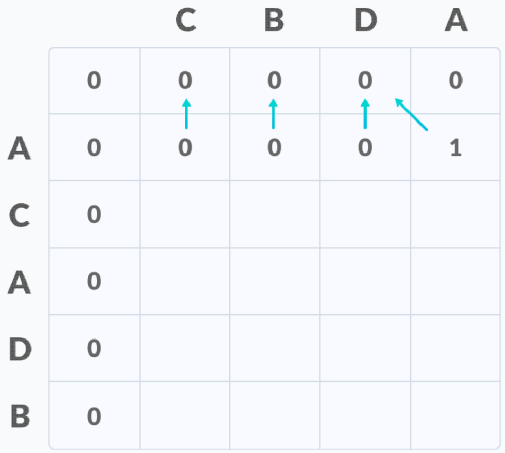
The following steps are followed for finding the longest common subsequence.

1. Create a table of dimension `n+1*m+1` where n and m are the lengths of `X` and `Y` respectively. The first row and the first column are filled with zeros.

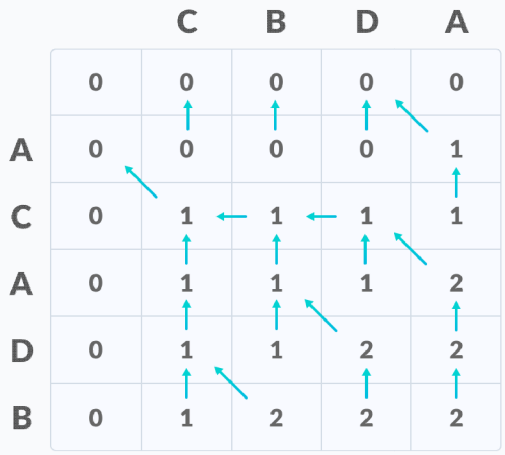
   Initialise a table
2. Fill each cell of the table using the following logic.
3. If the character correspoding to the current row and current column are matching, then fill the current cell by adding one to the diagonal element. Point an arrow to the diagonal cell.
4. Else take the maximum value from the previous column and previous row element for filling the current cell. Point an arrow

to the cell with maximum value. If they are equal, point to any



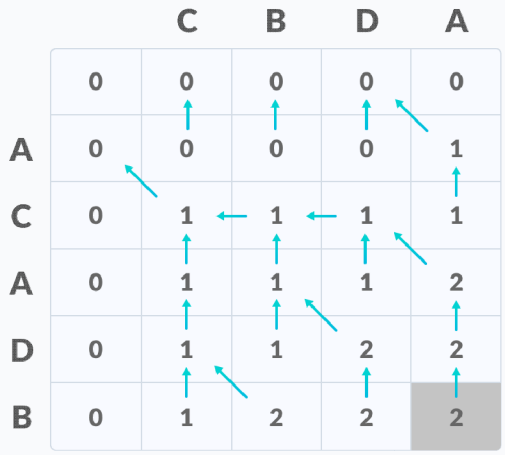
of them. Fill the values

5. **Step 2** is repeated until the table is filled.



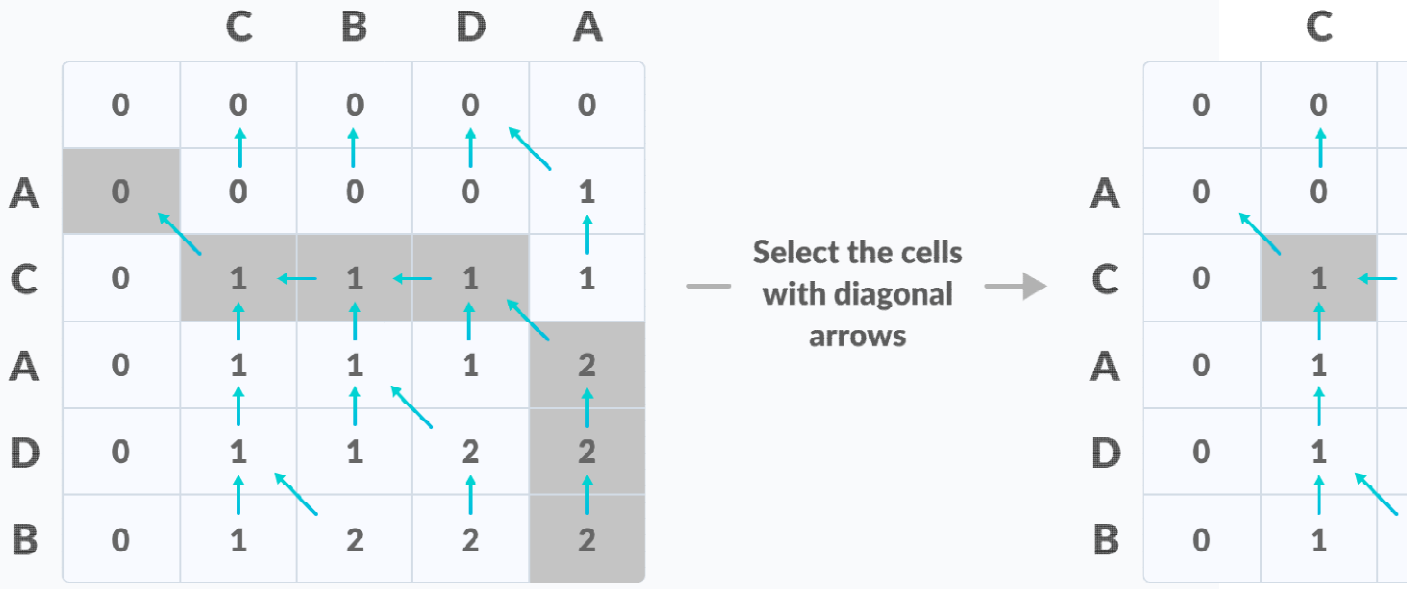
Fill all the values

6. The value in the last row and the last column is the length of the longest common subsequence.



The bottom right corner is the length of the LCS

7. In order to find the longest common subsequence, start from the last element and follow the direction of the arrow. The elements corresponding to () symbol form the longest common subsequence.



Create a path according to the arrows

Thus, the longest common subsequence is `CA`.

LCS

Greedy Algorithms In Python

Greedy algorithms aim to make the optimal choice at that given moment. Each step it chooses the optimal choice, without knowing the future. It attempts to find the globally optimal way to solve the entire problem using this method.

Greedy Algorithms Called Greedy

We call algorithms *greedy* when they utilise the greedy property. The greedy property is:

At that exact moment in time, the optimal choice to make

Greedy algorithms are greedy. They do not look into the future to decide the global optimal solution. They are only concerned with the optimal solution locally. This means that **the overall optimal solution may differ from the solution the algorithm chooses.**

They never look backwards at what they've done to see if they could optimize

globally. This is the main difference between Greedy and Dynamic Programming.

Elements of the Greedy Strategy

Optimal Substructure:

An optimal solution to the problem contains within it optimal solutions to sub-problems. A' = A - {1} (greedy choice) A' can be solved again with the greedy algorithm. S' = { i ∈ S, s_i ≥ f_i }

When do you use DP versus a greedy approach? Which should be faster?

The 0 - 1 knapsack problem:

A thief has a knapsack that holds at most W pounds. Item i : (v_i, w_i) (v = value, w = weight) thief must choose items to maximize the value stolen and still fit into the knapsack. Each item must be taken or left (0 - 1).

Fractional knapsack problem:

takes parts, as well as wholes

Both the 0 - 1 and fractional problems have the optimal substructure property: Fractional: v_i / w_i is the value per pound. Clearly you take as much of the item with the greatest value per pound. This continues until you fill the knapsack. Optimal (Greedy) algorithm takes **O** (n lg n), as we must sort on $v_i / w_i = d_i$.

Consider the same strategy for the 0 - 1 problem:

W = 50 lbs. (maximum knapsack capacity)

$w_1 = 10$	$v_1 = 60$	$d_1 = 6$
$w_2 = 20$	$v_2 = 100$	$d_2 = 5$
$w_3 = 30$	$v_3 = 120$	$d_3 = 4$

were d is the value density

Greedy approach: Take all of 1, and all of 2: $v_1 + v_2 = 160$, optimal solution is to take all of 2 and 3: $v_2 + v_3 = 220$, other solution is to take all of 1 and 3 $v_1 + v_3 = 180$. All below 50 lbs.

When solving the 0 - 1 knapsack problem, empty space lowers the effective d of the load. Thus each time an item is chosen for inclusion we must consider both

- i included
- i excluded

Activity Selection Problem

Given N **activities with their** start **time and** end **time. The task is to find the solution set having** a maximum number of non-conflicting **activities**

that can be executed within the given time, assuming only a single activity can be performed at a given time.

Examples:

Input: start[] = [10, 12, 20}]
end[] = [20, 25, 30]
Output: [0, 2]
Explanation: A maximum of two activities can be performed, i.e. Activity 0 and Activity 2[0-based indexing].

Input: start[] = [1, 3, 0, 5, 8, 5]
finish[] = [2, 4, 6, 7, 9, 9]
Output: [0, 1, 3, 4]
Explanation: A maximum of four activities can be performed, i.e. Activity 0, Activity 1, Activity 3, and Activity 4[0-based indexing].

Huffman Coding

Huffman coding is a lossless data compression algorithm. The idea is to assign variable-length codes to input characters, lengths of the assigned codes are based on the frequencies of corresponding characters.

Steps to build Huffman Tree

Input is an array of unique characters along with their frequency of occurrences and output is Huffman Tree.

1. Create a leaf node for each unique character and build a min heap of all leaf nodes (Min Heap is used as a priority queue. The value of frequency field is used to compare two nodes in min heap. Initially, the least frequent character is at root)

2. Extract two nodes with the minimum frequency from the min heap.

3. Create a new internal node with a frequency equal to the sum of the two nodes frequencies. Make the first extracted node as its left child and the other extracted node as its right child. Add this node to the min heap.
4. Repeat steps#2 and #3 until the heap contains only one node. The remaining node is the root node and the tree is complete.
   Let us understand the algorithm with an example:

character	Frequency
a	5
b	9
c	12
d	13
e	16
f	45

Step 1. Build a min heap that contains 6 nodes where each node represents root of a tree with single node.
Step 2 Extract two minimum frequency nodes from min heap. Add a new internal node with frequency 5 + 9 = 14.

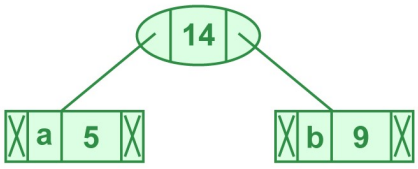


Illustration of step 2

Now min heap contains 5 nodes where 4 nodes are roots of trees with single element each, and one heap node is root of tree with 3 elements

character	Frequency
c	12
d	13
Internal Node	14
e	16
f	45

Step 3: Extract two minimum frequency nodes from heap. Add a new internal node with frequency 12 + 13 = 25

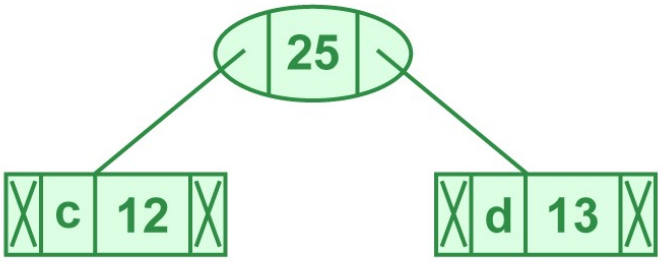


Illustration of step 3

Now min heap contains 4 nodes where 2 nodes are roots of trees with single element each, and two heap nodes are root of tree with more than one nodes

character	Frequency
Internal Node	14
e	16
Internal Node	25
f	45

Step 4: Extract two minimum frequency nodes. Add a new internal node with frequency 14 + 16 = 30

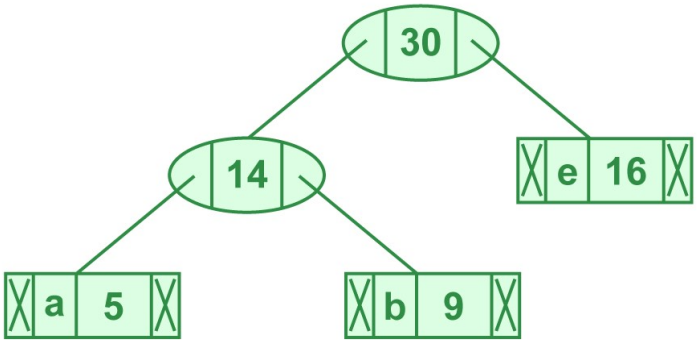


Illustration of step 4

Now min heap contains 3 nodes.

character	Frequency
Internal Node	25
Internal Node	30
f	45

Step 5: Extract two minimum frequency nodes. Add a new internal node with frequency 25 + 30 = 55

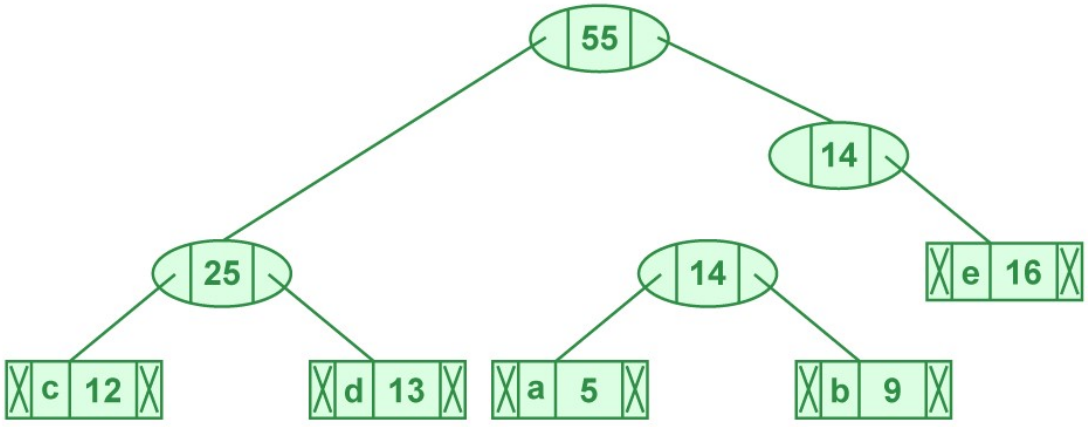


Illustration of step 5

Now min heap contains 2 nodes.

character	Frequency
f	45

```
Internal Node    55
```

Step 6: Extract two minimum frequency nodes. Add a new internal node with frequency 45 + 55 = 100

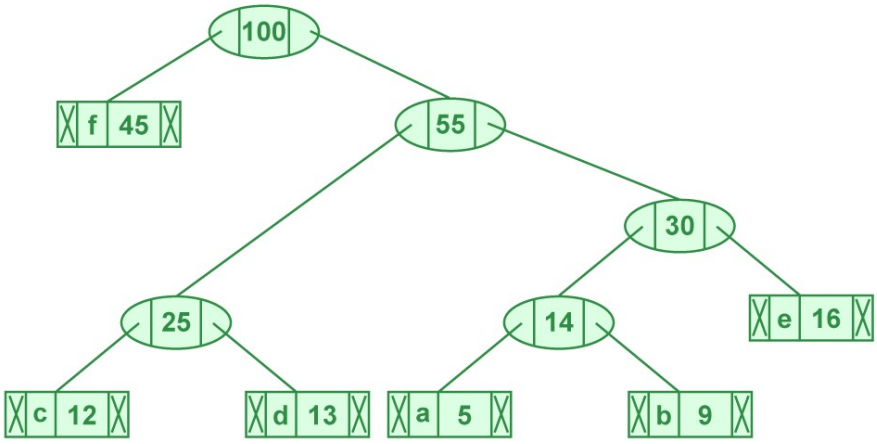


Illustration of step 6

Now min heap contains only one node.

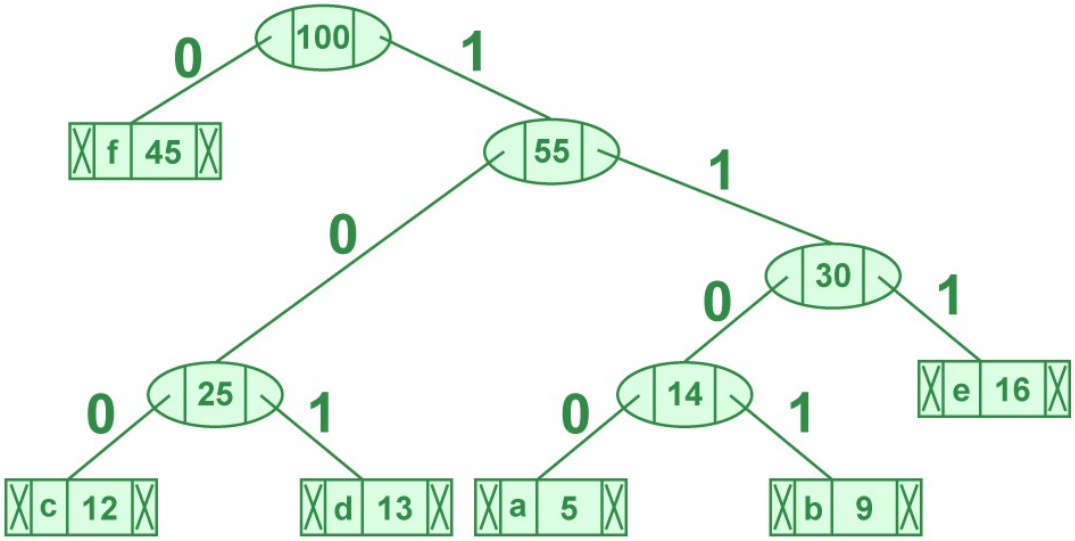
```
character      Frequency

Internal Node    100
```

Since the heap contains only one node, the algorithm stops here.

Steps to print codes from Huffman Tree:

Traverse the tree formed starting from the root. Maintain an auxiliary array. While moving to the left child, write 0 to the array. While moving to the right child, write 1 to the array. Print the array when a leaf node is encountered.



Steps to print code from HuffmanTree

The codes are as follows:

```
character   code-word
    f           0
    c           100
    d           101
    a           1100
    b           1101
    e           111
```