UNIT III ALGORITHM DESIGN TECHNIQUES

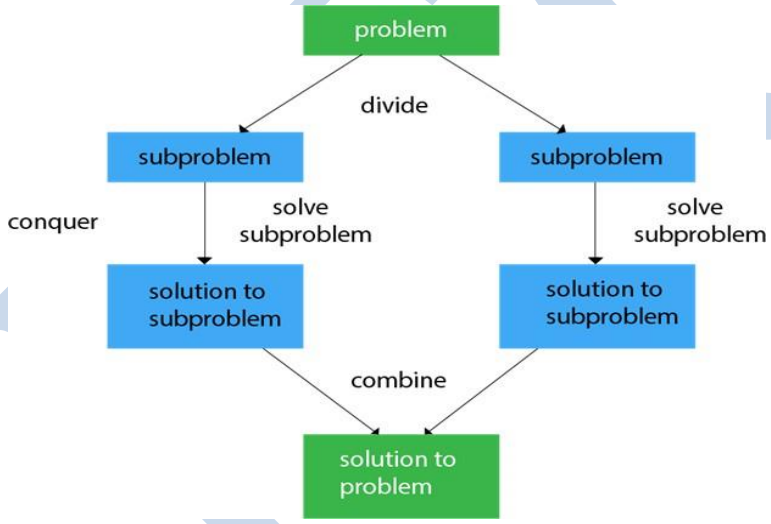
Divide and Conquer methodology: Finding maximum and minimum - Merge sort - Quick sort Dynamic programming: Elements of dynamic programming — Matrix-chain multiplication - Multi stage graph — Optimal Binary Search Trees. Greedy Technique: Elements of the greedy strategy - Activity-selection problem –- Optimal Merge pattern — Huffman Trees.

Divide and Conquer Introduction

Divide and conquer is a problem solving strategy that involves breaking a problem into smaller subproblems, solving each sub problem independently, and then combining the solutions to solve the original problem. This approach is often used in algorithm design and computer science

Divide and Conquer algorithm uses the following three steps.

- **Divide** the original problem into a set of sub problems.
- **Conquer:** Solve every sub problem individually, recursively.
- **Combine:** Combine the sub-problems to get the final solution of the whole problem.



Applications

The divide-and-conquer technique is the basis of efficient algorithms for many problems, such as:

- Finding maximum/minimum
- Sorting (e.g., quicksort, merge sort)
- Multiplying large numbers (e.g., the Karatsuba algorithm)
- Finding the closest pair of points
- Syntactic analysis (e.g., top-down parsers)
- Computing the discrete Fourier transform (FFT).

Finding maximum and minimum (Max-Min Problem)

Divide and conquer is a common algorithmic technique used to solve a variety of problems. One application of this technique is to find the maximum and minimum values in a given list of numbers.

Problem Statement

The Max-Min Problem in algorithm analysis is finding the maximum and minimum value.

Input & Output

- Input : ([8, 16, 24, 1, 25, 3, 10, 65, 55])
- Output : Max = 65& Min = 1

Methods

- Naïve method
- Using inbuilt function - sort()
- Divide and Conquer method (or) Recursive method

*Method 1:*Naïve Method (or) Brute force approach (or) Iteration

Naïve method is a basic method to solve any problem. In this method, the maximum and minimum number can be found separately.

Algorithm

```
Algorithm: Max-Min-Element (numbers[])
max := numbers[1]
min := numbers[1]

for i = 2 to n do
   if numbers[i] > max then
      max := numbers[i]
   if numbers[i] < min then
      min := numbers[i]
return (max, min)
```

```
arr = [10, 89, 9, 56, 4, 80, 8]
min = arr[0]
max = arr[0]

for i in range(len(arr)):
  if arr[i] < min:
    min = arr[i]
  if arr[i] > max:
    max = arr[i]

print ("Maximum value in the array : " , min)
print ("Maximum value in the array : " , max)
```

Execution

```
Maximum value in the array :  4
Maximum value in the array :  89
```

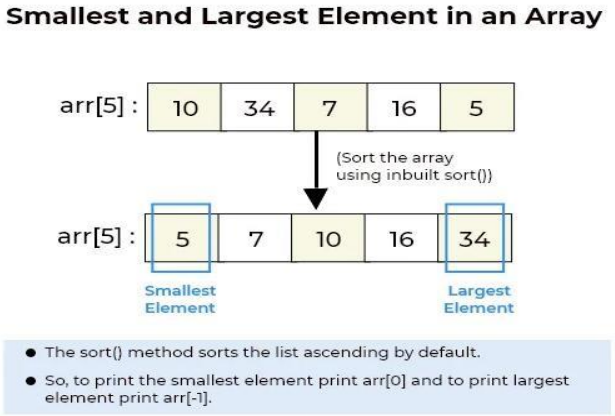
Algorithm analysis

- The worst case occur when elements is sorted in descending order. We will be making two comparisons at each iteration.
- Total number of comparison = 2*(n — 1), so Time Complexity =**2n — 2**.
- Space complexity = O(1)

*Method 2:*Using inbuilt function - sort()

- Sort the array using inbuilt sort()function

- Minimum element is at index 0 and maximum is at index -1
- print(arr[0]) and print(arr[-1])




```
arr = [10, 89, 9, 56, 4, 80, 8]
arr.sort()
print ("Maximum value in the array : " , arr[0])
print ("Maximum value in the array : " , arr[-1])
```

Execution

```
Maximum value in the array :  4
Maximum value in the array :  89
```

*Method 3:*Divide and Conquer method (or) Recursive method

The algorithm to find the maximum and minimum values in a given list using divide and conquer can be summarized in the following steps:

- If the list has only one element, return it as both the minimum and maximum value.
- If the list has two elements, compare them and return the minimum and maximum.
- If the list has more than two elements, divide it into two halves and recursively find the minimum and maximum in each half.
- Compare the minimum and maximum values found in each half and return the overall minimum and maximum.

```
def find_max_min(arr):
    if len(arr) == 1:
        return (arr[0], arr[0])
    elif len(arr) == 2:
        return (max(arr[0], arr[1]), min(arr[0], arr[1]))
    else:
        mid = len(arr) // 2
        left_max, left_min = find_max_min(arr[:mid])
        right_max, right_min = find_max_min(arr[mid:])
        return (max(left_max, right_max), min(left_min, right_min))

# Example usage
arr = [3, 5, 1, 9, 7, 4, 2, 8, 6]
max_val, min_val = find_max_min(arr)
print(f"Max value: {max_val}")
print(f"Min value: {min_val}")
```

Execution

```
Max value: 9
Min value: 1
```

Algorithm analysis

Let T (n) = time required to apply the algorithm on an array of size n. Here we divide the terms as T(n/2). Let us assume that n is in the form of power of 2. Hence, n = 2^k where k is height of the recursion tree.

$$T(n) = \begin{cases} T\left(\lfloor \frac{n}{2} \rfloor\right) + T\left(\lceil \frac{n}{2} \rceil\right) + 2 & for\ n > 2 \\ 1 & for\ n = 2 \\ 0 & for\ n = 1 \end{cases}$$

$$T(n) = 2\,T\left(\frac{n}{2}\right) + 2 \quad \rightarrow \text{Eq (i)}$$

T (2) = 1, time required to compare two elements/items. (Time is measured in units of the number of comparisons).

$$T\left(\frac{n}{2}\right) = 2T\left(\frac{n}{2^2}\right) + 2 \quad \rightarrow \text{Eq (ii)}$$

Put eq (ii) in eq (i)

$$T(n) = 2\left[2T\left(\frac{n}{2^2}\right) + 2\right] + 2$$

$$= 2^2 T\left(\frac{n}{2^2}\right) + 2^2 + 2$$

Similarly, apply the same procedure recursively on each subproblem

$$T(n) = 2^i T\left(\frac{n}{2^i}\right) + 2^i + 2^{i-1} + \ldots\ldots\ldots\ldots\ldots\ldots + 2 \quad \ldots\ldots \text{(Eq. 3)}$$

Recursion will stop, when

$$\frac{n}{2^i} = 2 \Rightarrow n = 2^{i+1} \quad \rightarrow \text{(Eq. 4)}$$

Put the equ.4 into equation3.

$$T(n) = 2^i T(2) + 2^i + 2^{i-1} + \ldots\ldots\ldots\ldots\ldots\ldots + 2$$

$$= 2^i .1 + 2^i + 2^{i-1} + \ldots\ldots\ldots\ldots\ldots\ldots + 2$$

$$= 2^i + \frac{2\left(2^i - 1\right)}{2-1}$$

$$= 2^{i+1} + 2^i - 2$$

$$= n + \frac{n}{2} - 2$$

$$= \frac{3n}{2} - 2$$

Number of comparisons by divide and conquer approach on n elements = 3n/2 – 2

Number of comparisons by naïve approach on n elements = (n-1) + (n-1) = 2n-2

Method	Time Complexity	Space Complexity
Brute Force Approach	O(n)	O(1)
Divide and Conquer Approach	O(nlogn)	O(logn) Due to the recursion stack.

Sorting

Sorting involves arranging a collection of elements in a specific order, such as numerical or alphabetical order. There are several algorithms for sorting, including bubble sort, selection sort, insertion sort, quicksort, mergesort, and heapsort.

The combination of sorting and divide and conquer is often used in solving problems that *involve sorting a large data set*. Merge sort & Quick sort uses divide and conquer method to sort a collection of elements.

Merge sort

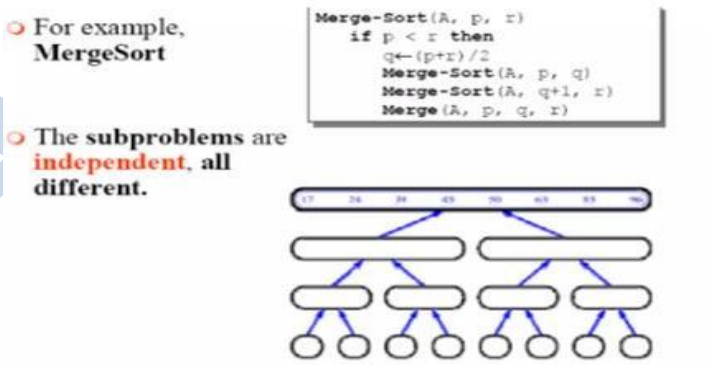
Merge sort is a sorting algorithm that works by dividing an array into two halves, recursively sorting each half, and then merging the sorted halves back together.

Basic idea

- Repeatedly divide the array in half until each sub-array contains only one element, which is already sorted by definition.
- The two sorted sub-arrays are then merged back together to form a larger sorted array, and the process continues until the entire array is sorted.

Here are the high-level steps of the merge sort algorithm:

- Divide the array into two halves.
- Recursively sort each half using merge sort.
- Merge the sorted halves back together.



The key step in merge sort is the merge operation, which combines two sorted sub-arrays into a larger sorted array. Here are the steps for the merge operation:

- Create a new array to hold the merged result.
- Initialize two pointers, one for each sub-array, pointing to the first element of each sub-array.
- Compare the values at the two pointers, and add the smaller value to the merged result array.
- Move the pointer of the sub-array whose value was added to the merged result array to the next element.
- Repeat steps 3 and 4 until one sub-array is fully processed.

- Add the remaining elements of the other sub-array to the merged result array.

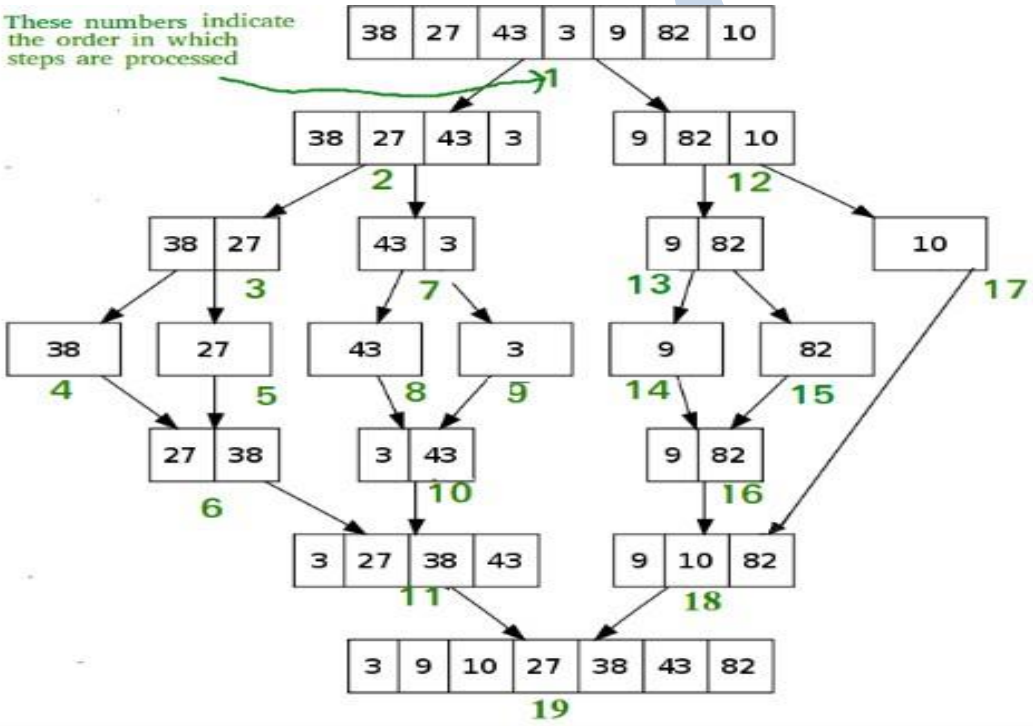
Example:

Here is an example of how merge sort works on the array [5, 2, 9, 1, 5, 6]:

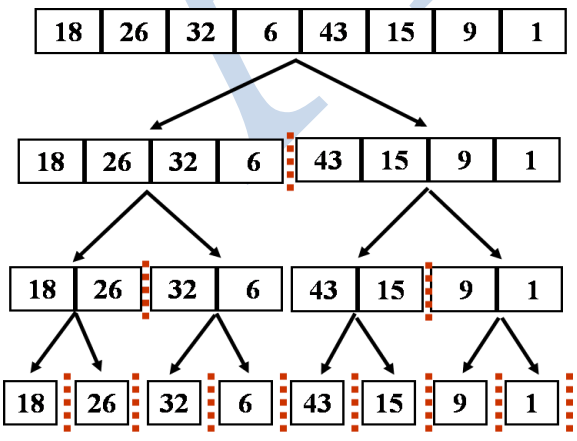
- Divide the array into two halves: [5, 2, 9] and [1, 5, 6].
- Recursively sort each half using merge sort:
  - [5, 2, 9] -> [2, 5, 9] and [1, 5, 6] -> [1, 5, 6]
- Merge the two sorted halves back together
  - [2, 5, 9] and [1, 5, 6] ->[1, 2, 5, 5, 6, 9].
- The final result is the sorted array [1, 2, 5, 5, 6, 9].

Working of merge sort

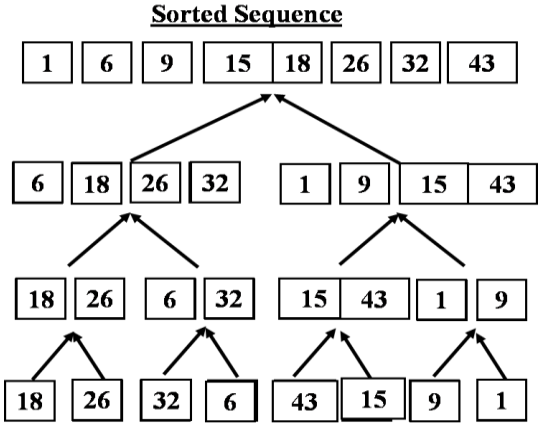
Example 1



Example 2



Splitting of List



Merging of Sorted Lists

```
def merge_sort(arr):
    # Base case: if the array is empty or contains only 1 element, it is already sorted
    if len(arr) <= 1:
        return arr

    # Divide the array into two halves
    mid = len(arr) // 2
    left_half = arr[:mid]
    right_half = arr[mid:]

    # Recursively sort each half
    left_half = merge_sort(left_half)
    right_half = merge_sort(right_half)

    # Merge the two sorted halves back together
    merged_arr = []
    i = 0
    j = 0
    while i < len(left_half) and j < len(right_half):
        if left_half[i] < right_half[j]:
            merged_arr.append(left_half[i])
            i += 1
        else:
            merged_arr.append(right_half[j])
            j += 1
    merged_arr += left_half[i:]
    merged_arr += right_half[j:]

    return merged_arr
arr = [5, 12, 49, 1, 65, 6]
sorted_arr = merge_sort(arr)
print(sorted_arr)
```

Execution:

```
[1, 5, 6, 12, 49, 65]
```

Quick sort (or) Partition-Exchange sort

Quick sort is a popular sorting algorithm that uses a divide-and-conquer strategy to sort an array. It works by partitioning the array into two sub-arrays:

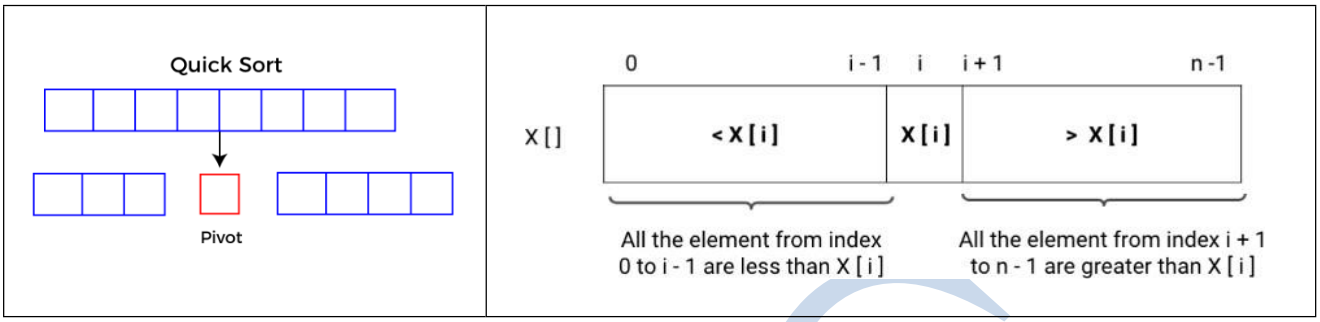
- Array 1: Containing elements smaller than a chosen pivot value
- Array 2: Containing elements larger than the pivot value.

The pivot value is then placed in its final sorted position in the array, and the algorithm is recursively applied to the two sub-arrays on either side of the pivot until the entire array is sorted.

Here are the high-level steps of the quick sort algorithm:

- Choose a pivot value from the array.
- Partition the array into two sub-arrays, one containing elements smaller than the pivot and the other containing elements larger than the pivot.

- Recursively apply quick sort to each sub-array.
- Combine the two sorted sub-arrays and the pivot value to form the final sorted array.



Divide and conquer idea of quicksort

The above approach is a divide and conquer approach of problem-solving: We divide sorting problem into two smaller subproblems using partition algorithm and solve each sub-problems recursively to get the final sorted array. We repeat same process of partition for smaller sub-arrays till we reach the base case.

Divide part: We divide problem into two smaller sub-problems by partitioning the array **X[l...r]** into two smaller subarrays around **pivot**. The partition process returns the **pivotIndex** i.e. index of pivot in the sorted array.

- Subproblem 1: Sorting array X[] from **l** to **pivotIndex - 1.**
- Subproblem 2: Sorting array X[] from **pivotIndex + 1** to **r.**

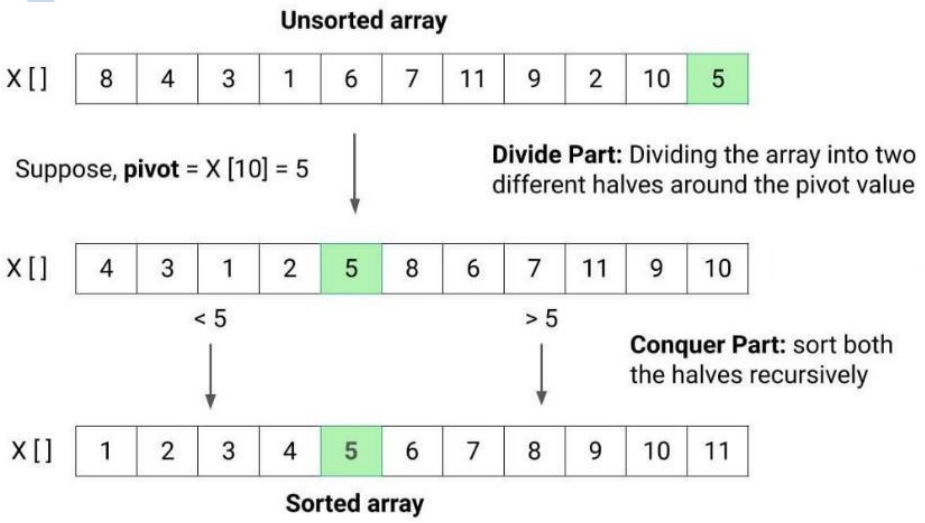
Conquer part: Now, we solve both sub-problems or sort both smaller arrays recursively. We should not worry about the solution of sub-problems, because recursion will do this work for us.

Combine part: This is a trivial case because after sorting both smaller arrays, whole array will get sorted. In other words, there is no need to write code for combine part.

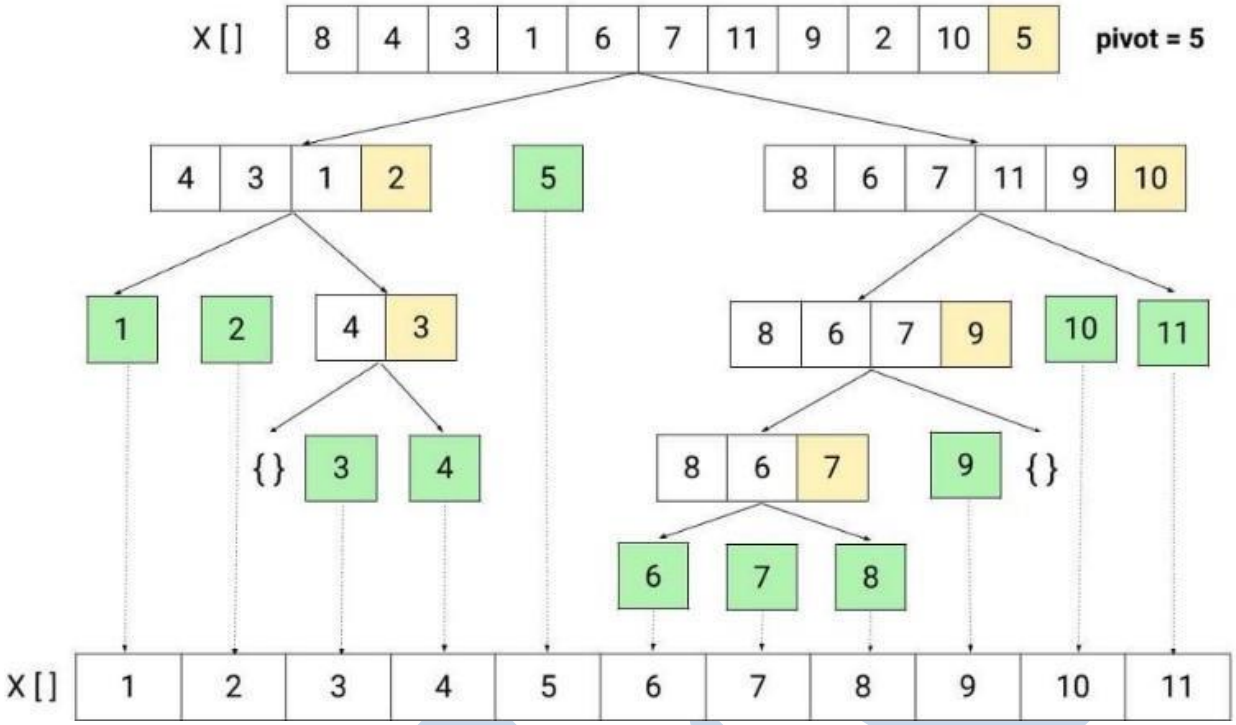
Choosing the pivot

Picking a good pivot is necessary for the fast implementation of quicksort. Some of the ways of choosing a pivot are as follows -

- Pivot can be random, i.e. select the random pivot from the given array.
- Pivot can either be the rightmost element of the leftmost element of the given array.
- Select median as the pivot element.



Visualizing Quick sort algorithm




```
def quick_sort(arr):
    if len(arr) <= 1:
        return arr

    pivot = arr[len(arr) // 2]
    left = [x for x in arr if x < pivot]
    middle = [x for x in arr if x == pivot]
    right = [x for x in arr if x > pivot]

    return quick_sort(left) + middle + quick_sort(right)

arr = [35, 2, 47, 13, 39, 3]
print(quick_sort(arr))
```

Execution

```
[2, 3, 13, 35, 39, 47]
```

Algorithm analysis of Merge sort & Quick sort

Method	**Time Complexity**	**Space Complexity**
Merge Sort	O(nlogn) Divide the input array into halves recursively until the array size is 1	O(n)
Quick Sort	O(nlogn) Depends on the choice of pivot element. - In best and average cases, when the pivot, divides the input array into two roughly equal-sized sub-arrays, the time complexity is O(nlogn). - However, in the worst case, when the pivot is chosen as the minimum or maximum element of the input array, the time complexity is O(n^2). This can be avoided by choosing a good pivot element, such as the median.	O(logn)

Dynamic programming

Dynamic programming is a technique that breaks the problems into sub-problems, and saves the result for future purposes so that we do not need to compute the result again.

Key idea

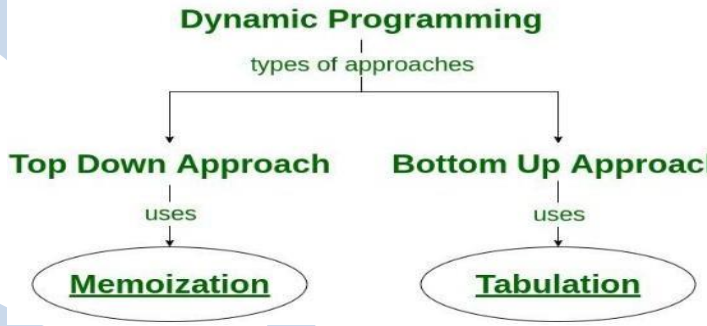
- Use memoization to avoid repeating computations that have already been performed. This is typically done by storing the results of sub-problems in a table or array, so that they can be looked up and reused later if needed.

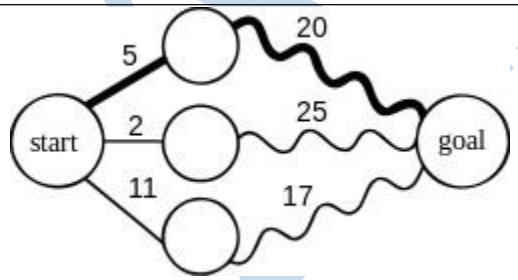
Note:

Memoization or memoisation→ Optimization technique to speed up computer programs by storing the results of expensive function calls. When the same inputs occur again, it sends the result.

Dynamic programming is an optimization method which was developed by Richard Bellman in 1950. It involves three steps:

- It breaks down the complex problem into simpler subproblems.
- It finds the optimal solution to these sub-problems.
- It stores the results of subproblems (memoization). The process of storing the results of subproblems is known as memorization.
- It reuses them so that same sub-problem is calculated more than once.
- Finally, calculate the result of the complex problem.



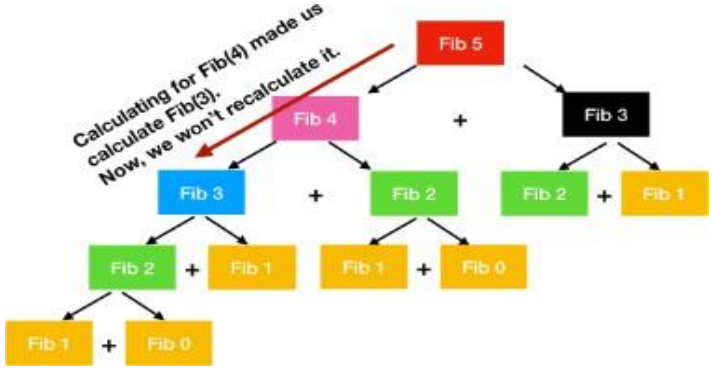


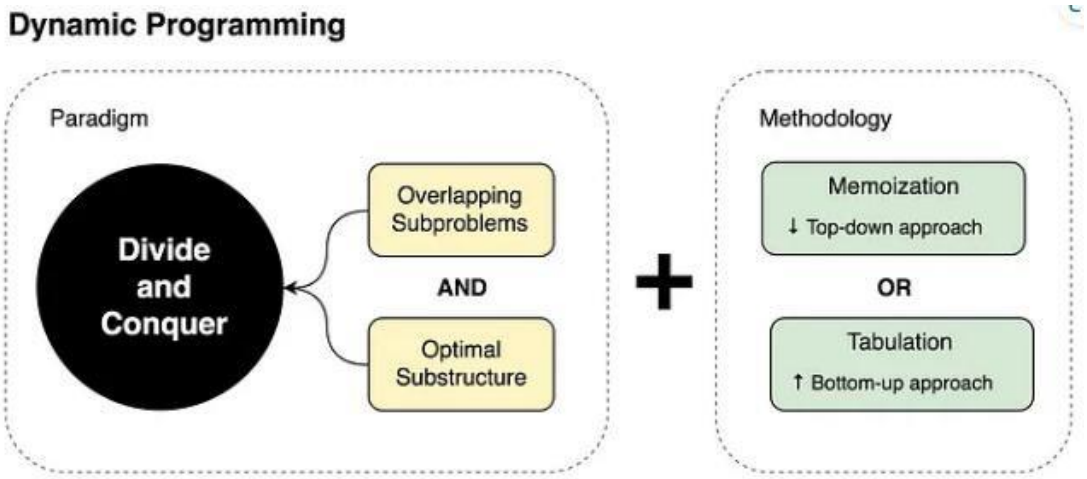
One real-time example of dynamic programming is finding the shortest path in a graph. Consider a GPS system trying to find the shortest path from your current location to your destination. The system uses dynamic programming to calculate the optimal route.

- Straight line indicates a single edge
- Wavy line indicates a shortest path between the two vertices
- Bold line is the overall shortest path from start to goal.

Fibonacci series:

F(n) = F(n-1) + F(n-2)





Top-down approach

The top-down approach follows the memorization technique. Here memorization is equal to the sum of recursion and caching. Recursion means calling the function itself, while caching means storing the intermediate results. When the recursion is too deep, the stack overflow condition will occur.

Bottom-Up approach

The bottom-up approach uses the tabulation technique. It solves the same kind of processing but it removes the recursion. If we remove the recursion, there is no stack overflow issue and no overhead of the recursive functions. In this tabulation technique, we solve the problems and store the results in a matrix.

Components of Dynamic programming

- **Stages**
  The given problem can be divided into a number of subproblems which are called stages. A stage is a small portion of given problem.
- **States**
  This indicates the subproblem for which the decision has to be taken. The variables which are used for taking a decision at every stage that is called as a state variable.
- **Decision**
  At every stage, there can be multiple decisions out of which one of the best decisions should be taken. The decision taken at each stage should be optimal; this is called as a stage decision.
- **Optimal policy**
  It is a rule which determines the decision at each and every stage; a policy is called an optimal policy if it is globally optimal. This is called as Bellman principle of optimality.

Applications of dynamic programming

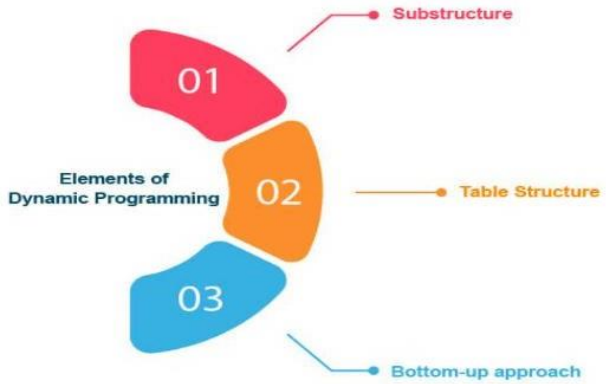
- 0/1 knapsack problem
- Mathematical optimization problem
- All pair Shortest path problem
- Reliability design problem
- Longest common subsequence (LCS)
- Flight control and robotics control
- Time sharing: It schedules the job to maximize CPU usage

CompareDivide & Conquer Method and Dynamic programming

Divide & Conquer Method	**Dynamic programming**
Solve the problems by dividing the problem into sub-problems	Solve the problems by dividing the problem into sub-problems
Three steps: - Divide the problem into a number of sub-problems. - Conquer the sub-problems by solving them recursively. - Combine the solution.	Five steps: - Breaks the complex problem into simpler sub-problems. - Find the optimal solution to these sub-problems. - Stores the results of subproblems (memoization). - Reuses the same sub-problem - Finally, calculate the result of the complex problem.
It is Recursive.	It is non-Recursive.
It is a top-down approach.	It is a Bottom-up approach.
Sub-problems are NOT dependent on each other	Sub-problems are dependent (or) interrelated to each other
It does more work on sub-problems and hence has more time consumption.	It solves sub-problems only once and then stores in the table.
Example: Merge Sort & Binary Search etc.	Example: Matrix Multiplication, Fibonocci, finding Shortest path, Knapsack problem, Optimal binary search tree

Elements of dynamic programming

Dynamic programming posses three important elements which are as given below:



Substructure: Sub-Structuring is the process of dividing the given problem statement into smaller sub-problems. By solving these sub-problems and combining their solutions, we can solve the original problem. This is the basis for the recursive nature of dynamic programming.

Table Structure: It is necessary to store the result of the sub-problem in a table.By reusing the solutions of sub-problems many times, we don't have to solve the same problem, again and again,

Bottom-up approach: The process of combining the solutions of sub-problems to achieve the final result using the table is done through bottom-up approach.

Note:

Comparison between feasible and optimal solution

Feasible solution

While solving a problem by using a greedy approach, the solution is obtained in a number of stages. The solution which satisfies the problem constraints they are called a feasible solution.

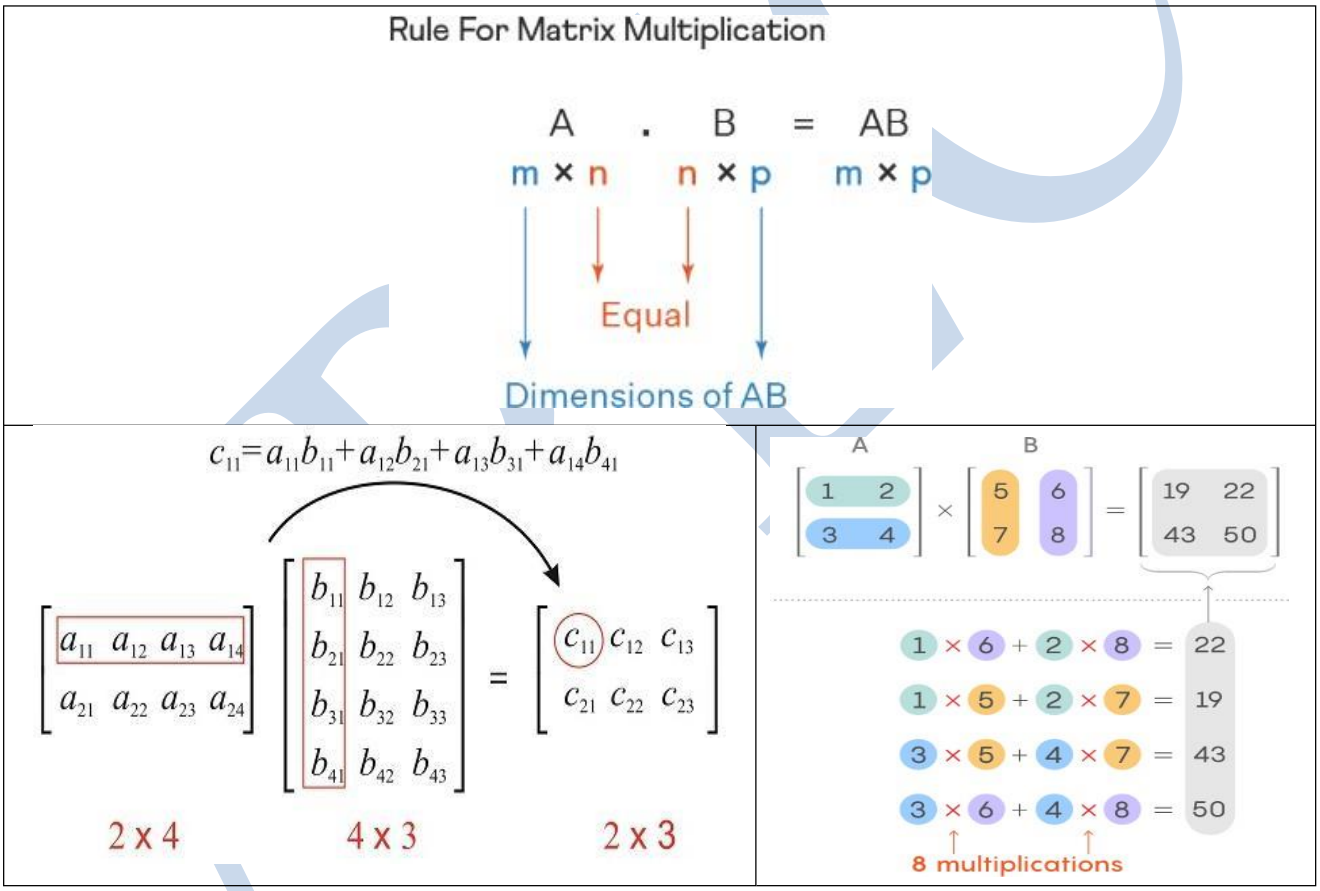
Optimal solution

Among all the feasible solution if the best solution either it can have a minimum or maximum value is chosen it is an optimal solution.

Matrix Chain Multiplication using Dynamic Programming

Matrix chain multiplication (or Matrix Chain Ordering Problem, MCOP) is an optimization problem.

Rule to matrix multiplication



Aim

- Find the most efficient way to multiply a given sequence of matrices. The problem is not actually to perform the multiplications but ***todecide the sequence of the matrix multiplications*** involved.

The matrix multiplication is **associative** as no matter how the product is parenthesized, the result obtained will remain the same.

Example of Matrix Chain Multiplication

- Consider 4 matrices A, B, C, and D.
  ((AB)C)D = ((A(BC))D) = (AB)(CD) = A((BC)D) = A(B(CD))
- if A is a 10 × 30 matrix, B is a 30 × 5 matrix, and C is a 5 × 60 matrix

(AB)C needs (10×30×5) + (10×5×60) = 1500 + 3000 = 4500 operations
A(BC) needs (30×5×60) + (10×30×60) = 9000 + 18000 = 27000 operations.
Clearly, the first method is more efficient.

Algorithm

- First, it will divide the matrix sequence into two sub-sequences.
- You will find the minimum cost of multiplying out each subsequence.
- You will add these costs together and in the price of multiplying the two result matrices.
- These procedures will be repeated for every possible matrix split and calculate the minimum.

Detailed Algorithm

- *Define the subproblems:*
  Let m[i,j] be the minimum number of scalar multiplications needed to compute the product of matrices Ai x Ai+1 x ... x Aj.
- *Identify the base cases:*
  When there is only one matrix, m[i,i] = 0.
- *Define the recurrence relation:*
  To find m[i,j], we need to consider all possible ways to split the product Ai x Ai+1 x ... x Aj into two sub-products, and then choose the one that requires the minimum number of scalar multiplications. Let k be the index at which we split the product, such that i<= k < j. Then we have:
  m[i,j] = min(m[i,k] + m[k+1,j] + ri * ck * cj)
  whereri, ck, and cj are the dimensions of matrices Ai, Ak+1, and Aj, respectively.
- *Compute the solution:*
  The solution to the original problem is m[1,n].

Python Code

```
def matrix_chain_multiplication(dimensions):
    n = len(dimensions)
    m = [[float('inf')] * n for _ in range(n)]
    for i in range(n):
        m[i][i] = 0
    for length in range(2, n+1):
        for i in range(n - length + 1):
            j = i + length - 1
            for k in range(i, j):
                cost = m[i][k] + m[k+1][j] + dimensions[i][0] *
dimensions[k+1][0] * dimensions[j][1]
                if cost < m[i][j]:
                    m[i][j] = cost
    return m[0][n-1]

# Example usage
dimensions = [(10, 20), (20, 30), (30, 40)]
min_cost = matrix_chain_multiplication(dimensions)
print(f"Minimum cost for multiplying  matrices {dimensions} is {min_cost}.")
```

Execution

```
Minimum cost for multiplying    matrices [(10, 20),  (20,  30),  (30,  40)]  is
18000.
```

Solve the following problem by Matrix Chain Multiplication:

Given the sequence: {4, 10, 3, 12, 20, and 7}.

Size of the matrices: 4 x 10, 10 x 3, 3 x 12, 12 x 20, 20 x 7.

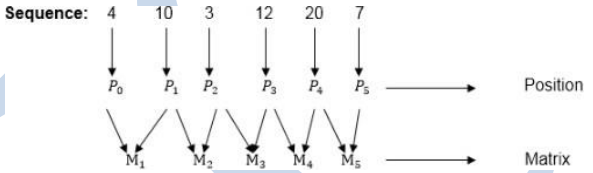
Compute M [i,j], 0 ≤ i, j≤ 5.

Solution:

We know M [i, i] = 0 for all i.

1	2	3	4	5	
0					1
	0				2
		0			3
			0		4
				0	5

Let us proceed with working away from the diagonal. We compute the optimal solution for the product of 2 matrices.



Here P0 to P5 are Position and M1 to M5 are matrix of size (pi to pi-1)

On the basis of sequence, we make a formula

For M_i ⟶ p [i] as column

p [i-1] as row

In Dynamic Programming, initialization of every method done by '0'.So we initialize it by '0'.It will sort out diagonally. We have to sort out all the combination but the minimum output combination is taken into consideration.

Calculation of Product of 2 matrices

1. m (1,2) = m1 x m2
   = 4 x 10 x 10 x 3
   = 4 x 10 x 3 = 120

2. m (2, 3) = m2 x m3
   = 10 x 3 x 3 x 12
   = 10 x 3 x 12 = 360

3. m (3, 4) = m3 x m4
   = 3 x 12 x 12 x 20
   = 3 x 12 x 20 = 720

4. m (4,5) = m4 x m5
   = 12 x 20 x 20 x 7
   = 12 x 20 x 7 = 1680

1	2	3	4	5	
0	120				1
	0	360			2
		0	720		3
			0	1680	4
				0	5

We initialize the diagonal element with equal i,j value with '0'. After that second diagonal is sorted out and we get all the values corresponded to it. Now the third diagonal will be solved out in the same way.

Calculation of Product of 3 matrices

M [1, 3] = M1 M2 M3

There are two cases by which we can solve this multiplication: (M1 x M2) + M3, M1+ (M2x M3)
After solving both cases we choose the case in which minimum output is there.

$$M[1,3] = \min \begin{cases} M[1,2] + M[3,3] + p_0 p_2 p_3 = 120 + 0 + 4.3.12 = 264 \\ M[1,1] + M[2,3] + p_0 p_1 p_3 = 0 + 360 + 4.10.12 = 840 \end{cases}$$

M [1, 3] =264

Comparing, 264 is minimum in both cases so we insert 264 in table and (M1 x M2) + M3.

M [2, 4] = M2 M3 M4

There are 2 cases by which we can solve this multiplication: (M2x M3)+M4, M2+(M3 x M4)
After solving both cases we choose the case in which minimum output is there.
DAA Example of Matrix Chain Multiplication

$$M[2,4] = \min \begin{cases} M[2,3] + M[4,4] + p_1p_3p_4 = 360 + 0 + 10.12.20 = 2760 \\ M[2,2] + M[3,4] + p_1p_2p_4 = 0 + 720 + 10.3.20 = 1320 \end{cases}$$

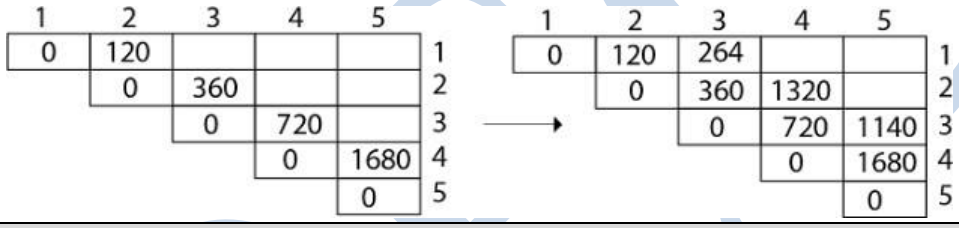
M [2, 4] = 1320
Comparing both output, 1320 is minimum in both cases so we insert 1320 in table and M2+(M3 x M4) this combination is chosen for the output making.

M [3, 5] = M3 M4 M5

There are two cases by which we can solve this multiplication: (M3 x M4) + M5, M3+ (M4xM5)
After solving both cases we choose the case in which minimum output is there.

$$M[3,5] = \min \begin{cases} M[3,4] + M[5,5] + p_2p_4p_5 = 720 + 0 + 3.20.7 = 1140 \\ M[3,3] + M[4,5] + p_2p_3p_5 = 0 + 1680 + 3.12.7 = 1932 \end{cases}$$

Comparing both output 1140 is minimum in both cases so we insert 1140 in table and (M3 x M4) + M5this combination is chosen for the output making.



Calculation of Product of 4 matrices

M [1, 4] = M1 M2 M3 M4

There are three cases by which we can solve this multiplication:

- (M1 x M2 x M3) M4
- M1 x(M2 x M3 x M4)
- (M1 xM2) x (M3 x M4)

After solving these cases we choose the case in which minimum output is there

$$M[1,4] = \min \begin{cases} M[1,3] + M[4,4] + p_0p_3p_4 = 264 + 0 + 4.12.20 = 1224 \\ M[1,2] + M[3,4] + p_0p_2p_4 = 120 + 720 + 4.3.20 = 1080 \\ M[1,1] + M[2,4] + p_0p_1p_4 = 0 + 1320 + 4.10.20 = 2120 \end{cases}$$

M [1, 4] =1080
As comparing the output of different cases then '1080' is minimum output, so we insert 1080 in the table and (M1 xM2) x (M3 x M4) combination is taken out in output making.

M [2, 5] = M2 M3 M4 M5

There are three cases by which we can solve this multiplication:

- (M2 x M3 x M4)x M5
- M2 x(M3 x M4 x M5)
- (M2 x M3)x (M4 x M5)

$$M[2,5] = \min \begin{cases} M[2,4] + M[5,5] + p_1p_4p_5 = 1320 + 0 + 10.20.7 = 2720 \\ M[2,3] + M[4,5] + p_1p_3p_5 = 360 + 1680 + 10.12.7 = 2880 \\ M[2,2] + M[3,5] + p_1p_2p_5 = 0 + 1140 + 10.3.7 = 1350 \end{cases}$$

M [2, 5] = 1350
As comparing the output of different cases then '1350' is minimum output, so we insert 1350 in the table and M2 x(M3 x M4 xM5)combination is taken out in output making.

1	2	3	4	5	
0	120	264			1
	0	360	1320		2
		0	720	1140	3
			0	1680	4
				0	5

→

1	2	3	4	5	
0	120	264	1080		1
	0	360	1320	1350	2
		0	720	1140	3
			0	1680	4
				0	5

Calculation of Product of 5 matrices

M [1, 5] = M1 M2 M3 M4 M5

There are five cases by which we can solve this multiplication:

- (M1 x M2 xM3 x M4)x M5
- M1 x(M2 xM3 x M4 xM5)
- (M1 x M2 xM3)x M4 xM5
- M1 x M2x(M3 x M4 xM5)

After solving these cases we choose the case in which minimum output is there

$$M[1,5] = \min \begin{cases} M[1,4] + M[5,5] + p_0 p_4 p_5 = 1080 + 0 + 4.20.7 = 1544 \\ M[1,3] + M[4,5] + p_0 p_3 p_5 = 264 + 1680 + 4.12.7 = 2016 \\ M[1,2] + M[3,5] + p_0 p_2 p_5 = 120 + 1140 + 4.3.7 = 1344 \\ M[1,1] + M[2,5] + p_0 p_1 p_5 = 0 + 1350 + 4.10.7 = 1630 \end{cases}$$

M [1, 5] = 1344

As comparing the output of different cases then '1344' is minimum output, so we insert 1344 in the table and M1 x M2 x(M3 x M4 x M5)combination is taken out in output making.

Final Output is:

1	2	3	4	5	
0	120	264	1080		1
	0	360	1320	1350	2
		0	720	1140	3
			0	1680	4
				0	5

→

1	2	3	4	5	
0	120	264	1080	1344	1
	0	360	1320	1350	2
		0	720	1140	3
			0	1680	4
				0	5

Step 3: Computing Optimal Costs: let us assume that matrix Ai has dimension pi-1x pi for i=1, 2, 3....n. The input is a sequence (p0,p1,..... pn) where length [p] = n+1. The procedure uses an auxiliary table m [1....n, 1.....n] for storing m [i, j] costs an auxiliary table s [1.....n, 1.....n] that record which index of k achieved the optimal costs in computing m [i, j].

The algorithm first computes m [i, j] ← 0 for i=1, 2, 3….n, the minimum costs for the chain of length 1.

Example

Matrices : A1 dimensions: (3 * 5) , A2 dimensions: (5 * 4) and A3 dimensions: (4 * 6)

Option 1 : ((A1 . A2) . A3) = ((3 * 5) . (5 * 4)) . (4 * 6)

Option 2 : (A1 . (A2 . A3)) = (3 * 5) . ((5 * 4) . (4 * 6))

1	Multiplication operations (3 * 5) . (5 * 4) = 3 .5 . 4 = 60	Multiplication operations (5 * 4) . (4 * 6) = 5 .4 . 6 = 120
2	Resultant matrix (3 * 5) . (5 * 4) = (3 . 4)	Resultant matrix (5 * 4) . (4 * 6) = (5 . 6)
3	Multiplication operations (3 * 4) . (4 * 6) = 3 .4 . 6 = 72	Multiplication operations (3 * 5) . (5 * 6) = 3 .5 . 6 = 90

Steps	Option 1	Option 2
4	Resultant matrix (3 * ~~4~~) . (~~4~~ * 6) = (3 . 6)	Resultant matrix (3 * ~~5~~) . (~~5~~ * 6) = (3 . 6)
5	Total Operations = 60 + 72 = 132	Total Operations = 120 + 90 = 210

Option 1 is clearly efficient than Option 2.

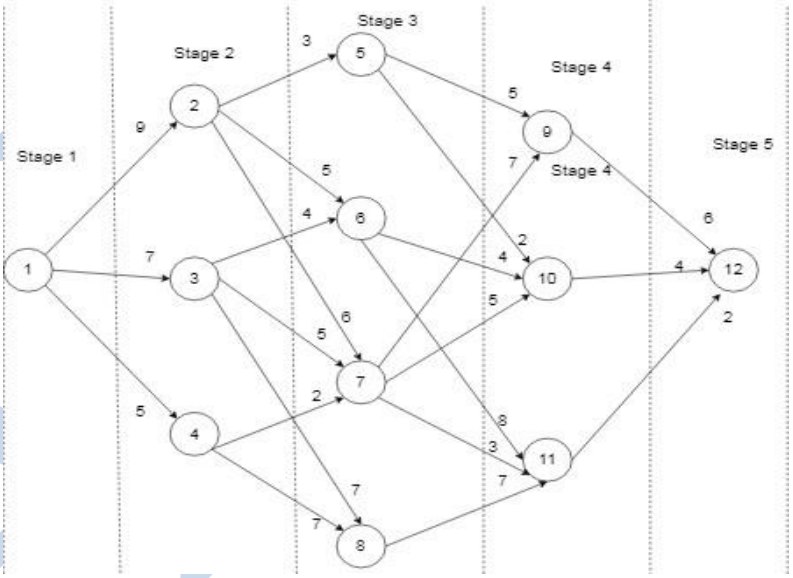
Algorithm analysis

Approach	Time Complexity	Space Complexity
Recursive Solution	$O(2^n)$	O(n)
Dynamic Programming	$O(n^3)$	$O(n^2)$

Multistage graph Dynamic Programming

Definition

A multistage graph G=(V, E) is a directed and weighted graph in which vertices are divided into stages. In between the starting and ending vertex, there will vertices in different stages that connect the starting and ending vertex. The main aim of this graph is to find the minimum cost path between from starting vertex s to ending vertex t.



Steps

- Divide the graph into multiple stages.
- Define the sub-problems for each stage and compute the optimal solution for each sub-problem.
- Use the optimal solution for the last stage to recursively compute the optimal solution for the previous stages.
- Compute the optimal solution for the first stage, which is the solution to the original problem.
- The algorithm operates in the backward direction, i.e. it starts from the last vertex of the graph and proceeds in a backward direction to find minimum cost path.

Applications

- Shortest path problems in a directed acyclic graph (DAG) with multiple stages
- Minimum spanning tree problems
- Network flow problems. One common example of a problem that can be solved using multistage graph dynamic programming is the shortest path problem

Advantages

- It is more efficient and can be used to solve complex problems.
- Multistage graphs are also easier to implement and can be more easily scaled.

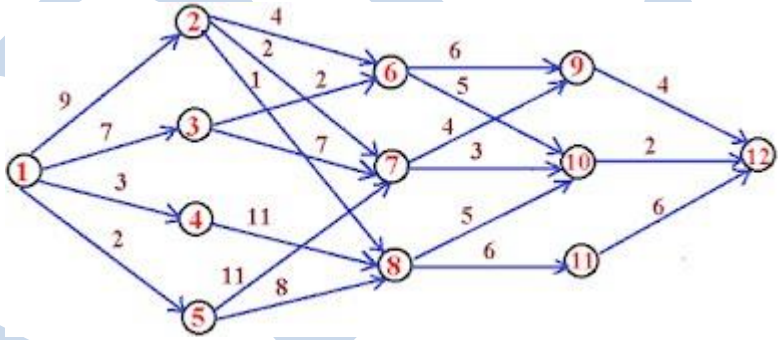
Disadvantages

- Uses more space.
- Difficult to interpret since the information is spread out over multiple stages.

Algorithm analysis

Time complexity: O(nm^2), where n is the number of stages in the graph and m is the maximum number of nodes in any stage. This is because the algorithm needs to compute the optimal solution for each node in each stage, and the computation for each node takes O(m) time. Since there are N stages, the overall time complexity is O(nm^2).

Solve the following multistage graph and find the minimum cost



Solution

This problem is solved by using tabular method, So we draw a table contain all vertices(v),cost(c) and destination (d).

V	1	2	3	4	5	6	7	8	9	10	11	12
Cost												
d												

Table - 1

Here our main objective is to select those paths which have minimum cost.So we can say thatit is a minimization problem. It can be solved by the principle of optimality which says thesequence of decisions.That means in every stage we have to take decision.

In this problem we will start from 12 so its distance is 12 and cost is 0.

Now calculate for V(12) and stage 5.

Here Cost(5 , 12) = 0 i.e Cost(stage number , vertex)

Now update distance and cost in table for stage 5

V	1	2	3	4	5	6	7	8	9	10	11	12
Cost												0
d												12

Table - 2

Now calculate for V(9 ,10 ,11) and stage 4.
cost(4 , 9) =4
cost(4 , 10) =2
cost(4 , 11) =5
Here the distance is 2
So update the new cost and distance in table for stage 4

V	1	2	3	4	5	6	7	8	9	10	11	12
Cost									4	2	5	0
d									12	12	12	12

Table - 3

Now calculate for V(6 , 7 , 8) and stage 3.
For calculating v(6) we must find out there connecting link in previous stage i.e 4. which is 9and 10.
cost(v,d) + cost(stage+1,vs) , where vs; vertices of that stage.

For v(6)	**For v(7)**
cost(6,9) + cost(4,9) = 6 + 4 = 10 cost(6,10) + cost(4,10) = 5 + 2 = 7 Find minimum from [10 , 7] which is 7 and the vertex give the minimum cost is 10.	cost(7,9) + cost(4,9) = 4 + 4 = 8 cost(7,10) + cost(4,10) = 3 + 2 = 5 Find minimum from [8 , 5] which is 5and the vertex give the minimum cost is 10.
For v(8) cost(8,10) + cost(4,10) = 5 + 2 = 7 cost(8,11) + cost(4,11) = 6 + 5 = 11 Find minimum from [7,11] is 7 and the vertex given the minimum cost is 10. So Now update the new cost and distance in table for stage 3.	

V	1	2	3	4	5	6	7	8	9	10	11	12
Cost						7	5	7	4	2	5	0
d						10	10	10	12	12	12	12

Table - 4

Now calculate for V(2 , 3 , 4 , 5) and stage 2.
For calculating v(2) we must find out there connecting link in previous stage i.e 3. which is 6, 7 and 8.
cost(v,d) + cost(stage+1,vs) , where vs; vertices of that stage.

For v(2) **Find cost(stage,vertex) i.e cost(2,2)** cost(2,6) + cost(3,6) = 4 + 7 = 11 cost(2,7) + cost(3,7) = 2 + 5 = 7 cost(2,8) + cost(3,8) = 1 + 7 = 8 Find minimum from [11,7,8] is 7 and the vertex given the minimum cost is 7.	**For v(3)** **Find cost(stage,vertex) i.e cost(2,3)** cost(3,6) + cost(3,6) = 2 + 7 = 9 cost(3,7) + cost(3,7) = 7 + 5 = 12 Find minimum from [9,12] is 9 and the vertex given the minimum cost is 6
For v(4) **Find cost(stage,vertex) i.e cost(2,4)** cost(4,8) + cost(3,8) = 11+ 7 = 18 So here the cost is 18 and the vertex given the minimum cost is 8.	**For v(5)** **Find cost(stage,vertex) i.e cost(2,5)** cost(5,7) + cost(3,7) = 11 + 5 = 16 cost(5,8) + cost(3,8) = 8 + 7 = 15 Find minimum from [16,15] is 15 and the vertex given the minimum cost is 8

Now update the new cost and distance in table for stage 2.

V	1	2	3	4	5	6	7	8	9	10	11	12
Cost		7	9	18	15	7	5	7	4	2	5	0
d		7	6	8	8	10	10	10	12	12	12	12

Table - 5

Now calculate for V(1) and stage 1.
For calculating v(1) we must find out there connecting link in previous stage i.e 2. which is 2, 3 , 4 and 5.
cost(v,d) + cost(stage+1,vs) , where vs; vertices of that stage.
For v(1) find cost(stage,vertex) i.e cost(1,1)
cost(1,2) + cost(2,2) = 9 + 7 =16
cost(1,3) + cost(2,3) = 7 + 9 =16
cost(1,4) + cost(2,4) = 3 + 18 = 21
cost(1,5) + cost(2,5) = 2 + 15 = 17

Find minimum from [16,16,21,17] is 16 and the vertex given the minimum cost is2,3 because here we get 16 twice so we consider both vertices.
Now update the new cost and distance in table for stage 1.

V	1	2	3	4	5	6	7	8	9	10	11	12
Cost	16	7	9	18	15	7	5	7	4	2	5	0
d	2/3	7	6	8	8	10	10	10	12	12	12	12

Table - 6

Note: Formula used for calculating cost for any stage.

$$cost(x,y) = \min[\ cost(y,v) + cost(x+1,v)]$$

Where x=stage number, y=current vertex number, v=vertex of stage x+1.
Now we apply dynamic programming where dynamic programming is a sequence of decision on the basis of available data.Here decision is taken in forward direction.
Using the above table, 2 paths are available.

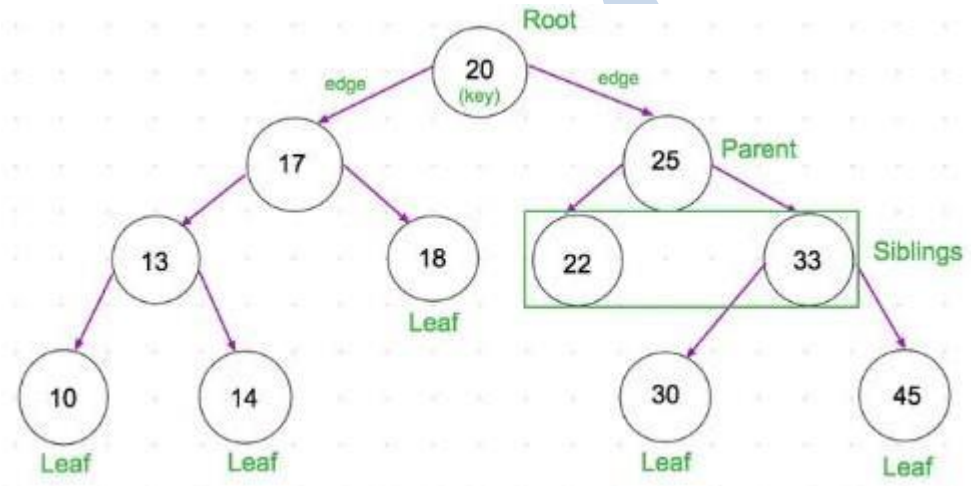
Path 1	*Path 2*

Now take decision by taking 2 for vertex 1. Find d(stage,vertex) for taking decision as:$d(1,1) = 2$ $d(2,2) = 7$ $d(3,7) = 10$ $d(4,10) = 12$ The shortest path will be :: $2 \rightarrow 7 \rightarrow 10 \rightarrow 12$	Now take decision by taking 3 for vertex 1. Find d(stage,vertex) for taking decision as: $d(1,1) = 3$ $d(2,3) = 6$ $d(3,6) = 10$ $d(4,10) = 12$ The shortest path will be :: $3 \rightarrow 6 \rightarrow 10 \rightarrow 12$

Optimal Binary Search Tree (OBST)

Binary Search Tree (BST)

- In BST the nodes in the left sub-tree have lesser value than the root node and the nodes in the right sub-tree have greater value than the root node.



- Like all tree data structure, binary search tree has a root, the top node (just one node), parent node has at most two children nodes, which are called siblings. The edge is the connection between one node and another. The node without children is called leaf.

Optimal Binary Search Tree

- An optimal binary search tree is a binary search tree that is constructed in such a way that the total cost of searching is minimized. The cost of searching in a binary search tree is the sum of the depth of each key multiplied by its frequency.
- To construct an optimal binary search tree, we need to use dynamic programming. The idea is to solve sub-problems and store the results in a table, and then use the results of the sub-problems to solve larger problems. We start by considering all possible sub-trees of the tree, and we calculate the cost of each sub-tree.

Basic Idea

- Given a sorted array $key\ [0..n-1]$ of search keys and an array $freq[0..n-1]$ of frequency counts, where $freq[i]$ is the number of searches for $keys[i]$. Construct a binary search tree of all keys such that the total cost of all the searches is as small as possible.

Let us first define the cost of a BST. The cost of a BST node is the level of that node multiplied by its frequency. The level of the root is 1.

For example: 10, 20, 30 are the keys, and the following are the binary search trees that can be made out from these keys.
Formula for calculating the number of trees:

- The formula for calculating the number of trees with n nodes is given by the Catalan number:

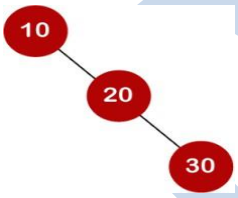
$$\frac{{}^{2n}C_n}{n+1}$$

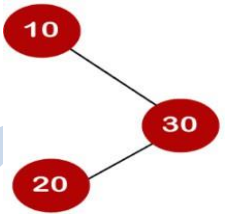
- When n = 3 , Number of trees = $\frac{{}^{2*3}C_3}{3+1} = \frac{{}^{6}C_3}{4} = \frac{6*5*4}{3*2*1}/4 = 5$

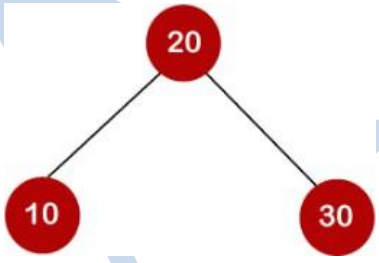
- When 3 keys are these, possible trees = 5

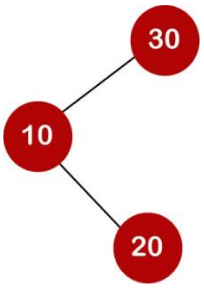
The cost required for searching an element depends on the comparisons to be made to search an element. Now, we will calculate the average cost of time of the above binary search trees.

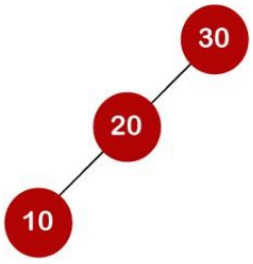
Example 1:

<table>
<tr>
<td>Case 1:


$average\ number\ of\ comparisons = \frac{1+2+3}{3} = 2$</td>
<td>Case 2:


$average\ number\ of\ comparisons = \frac{1+2+3}{3} = 2$</td>
</tr>
<tr>
<td>Case 3:


$average\ number\ of\ comparisons = \frac{1+2+2}{3} = 5/3$</td>
<td>Case 4:


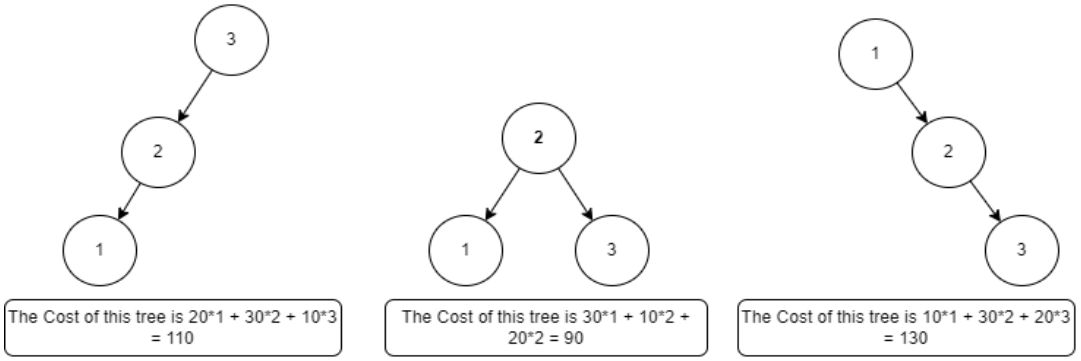
$average\ number\ of\ comparisons = \frac{1+2+3}{3} = 2$</td>
</tr>
<tr>
<td>Case 5:


$average\ number\ of\ comparisons = \frac{1+2+3}{3} = 2$</td>
<td>In Case 3, the number of comparisons is less because the height of the tree is less, so it's a BALANCED binary search tree.</td>
</tr>
</table>

Examples on finding the minimum cost given keys with frequencies

Example 2:

```
You are given 'arr' = [1, 2, 3] and 'freq' = [10, 30, 20], we can construct binary search
trees such that
```




```
We can clearly see that the minimum cost of the tree is 90.
```

Example 3:

```
Input:  keys[] = {10, 12}, freq[] = {34, 50}
There can be following two possible BSTs
        10                       12
          \                     /
           12                 10
          I                    II
Frequency of searches of 10 and 12 are 34 and 50 respectively.
The cost of tree I is 34*1 + 50*2 = 134
The cost of tree II is 50*1 + 34*2 = 118
```

Example 4:

```
Input:  keys[] = {10, 12, 20}, freq[] = {34, 8, 50}
There can be following possible BSTs
    10               12                 20         10               20
      \             /   \               /            \             /
      12          10     20           12               20         10
        \                             /               /           \
         20                         10               12            12
     I              II              III             IV            V
Among all possible BSTs, cost of the fifth BST is minimum.
Cost of the fifth BST is 1*50 + 2*34 + 3*8 = 142
```

Optimal Binary Search Tree using Dynamic Programming approach

Let's assume that frequencies associated with the keys 10, 20, 30 are 3, 2, 5.

Solution

Consider the below table, which contains the keys and frequencies.

Given keys and frequencies:

	1	2	3	4
Keys →	10	20	30	40
Frequency →	4	2	6	3

Initial Cost Table:

i \ j	0	1	2	3	4
0					
1					
2					
3					
4					

Step 1:

Calculate the values where j - i is equal to zero.

When i =0, j=0, then j-i = 0 (0,0)

When i = 1, j=1, then j-i = 0 (1,1)

When i = 2, j=2, then j-i = 0 (2,2)

When i = 3, j=3, then j-i = 0 (3,3)

When i = 4, j=4, then j-i = 0 (4,4)

c[0,0] = 0, c[1 ,1] = 0, c[2,2] = 0, c[3,3] = 0, c[4,4] = 0

Step 2:

Calculate the values where j - i equal to 1.

When j=1, i =0 then j-i = 1 (0,1)

When j =2, i = 1 then j-i = 1 (1,2)

When j =3, i = 2 then j-i = 1 (2,3)

When j =4, i = 3 then j-i = 1 (3,4)

Cost of c[0,1] = 4 [Key is 10 & Cost of key 10 = 4]

Cost of c[1,2] = 2 [Key is 20 & Cost of key 20 = 2]

Cost of c[2,3] = 6 [Key is 30 & Cost of key 30 = 6]

Cost of c[3,4] = 3 [Key is 40 & Cost of key 40 = 3]

Cost Table after step 1 and step 2

	0	1	2	3	4
0	0	4			
1		0	2		
2			0	6	
3				0	3
4					0

Step 3:

Calculate the values where j - i = 2

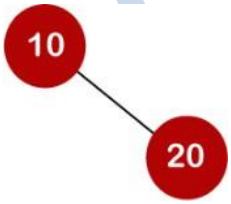
When j=2, i=0 then j-i = 2 (0,2)

When j=3, i=1 then j-i = 2 (1,3)

When j=4, i=2 then j-i = 2 (2,4)

<u>In this case, we will consider two keys.</u>

When i=0 and j=2, then keys 10 and 20. There are two possible trees that can be made out from these two keys shown below:



In first binary tree, cost = 4*1 + 2*2 = 8

In second binary tree, cost = 4*2 + 2*1 = 10

The minimum cost is 8; therefore, c[0,2] = 8

Cost Table:

Step 3 - Continued:

When i=1 and j=3, then keys 20 and 30. There are two possible trees that can be made out from these two keys shown below:

In the first binary tree, cost = 1*2 + 2*6 = 14

In the second binary tree, cost = 1*6 + 2*2 = 10

The minimum cost is 10; therefore, c[1,3] = 10

When i=2 and j=4, we will consider the keys at 3 and 4, i.e., 30 and 40. There are two possible trees that can be made out from these two keys shown as below:

In the first binary tree, cost = 1*6 + 2*3 = 12

In the second binary tree, cost = 1*3 + 2*6 = 15

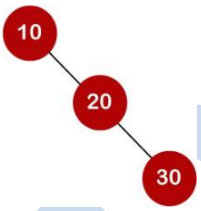
The minimum cost is 12, therefore, c[2,4] = 12

Cost Table:

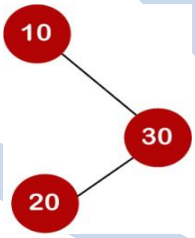
		0	1	2	3	4
0		0	4	8		
1			0	2		
2				0	6	
3					0	3
4						0

i \ 1	0	1	2	3	4
0	0	4	8^1		
1		0	2	10^3	
2			0	6	12^3
3				0	3
4					0

Step 4:
Calculate the values when j-i = 3
When j=3, i=0 then j-i = 3 (0,3)
When j=4, i=1 then j-i = 3 (1,3)
When i=0, j=3 then we will consider three keys, i.e., 10, 20, and 30.
The following are the trees that can be made if 10 is considered as a root node.

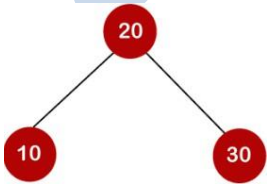


In the above tree, 10 is the root node, 20 is the right child of node 10, and 30 is the right child of node 20.
Cost = 1*4 + 2*2 + 3*6 = 26

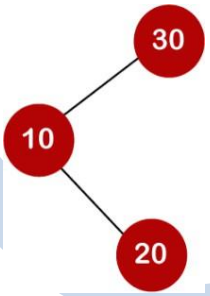


In the above tree, 10 is the root node, 30 is the right child of node 10, and 20 is the left child of node 20.
Cost =1*4 + 2*6 + 3*2 = 22

The following tree can be created if 20 is considered as the root node.



In the above tree, 20 is the root node, 30 is the right child of node 20, and 10 is the left child of node 20.
Cost = 1*2 + 4*2 + 6*2 = 22
The following are the trees that can be created if 30 is considered as the root node.



In the above tree, 30 is the root node, 10 is the left child of node 30 and 20 is the right child of node 10.
Cost = 1*6 + 2*4 + 3*2 = 20
Therefore, the minimum cost is 20 which is the 3rd root. So, c[0,3] is equal to 20.

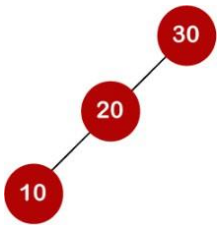
When i=1 and j=4 then we will consider the keys 20, 30, 40

c[1,4] = min{ c[1,1] + c[2,4], c[1,2] + c[3,4], c[1,3] + c[4,4] } + 11

= min{0+12, 2+3, 10+0}+ 11

= min{12, 5, 10} + 11

The minimum value is 5;
Therefore, c[1,4] = 5+11 = 16



In the above tree, 30 is the root node, 20 is the left child of node 30, and 10 is the left child of node 20.
Cost = 1*6 + 2*2 + 3*4 = 22

i \ j	0	1	2	3	4
0	0	4	8^1	20^3	
1		0	2	10^3	16^3
2			0	6	12^3
3				0	3
4					0

Step 5:

Calculate the values when j-i = 4
When j=4 and i=0 then j-i = 4
In this case, we will consider four keys, i.e., 10, 20, 30 and 40. The frequencies of 10, 20, 30 and 40 are 4, 2, 6 and 3 respectively.

w[0, 4] = 4 + 2 + 6 + 3 = 15

If we consider 10 as the root node then
C[0, 4] = min {c[0,0] + c[1,4]}+ w[0,4]
= min {0 + 16} + 15= 31

If we consider 20 as the root node then
C[0,4] = min{c[0,1] + c[2,4]} + w[0,4]
= min{4 + 12} + 15
= 16 + 15 = 31

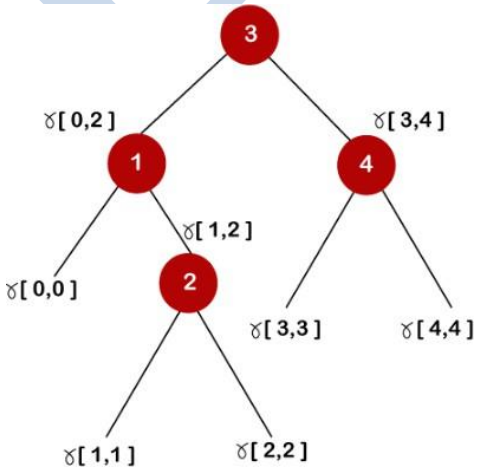
If we consider 30 as the root node then,
C[0,4] = min{c[0,2] + c[3,4]} +w[0,4]
= min {8 + 3} + 15
= 26

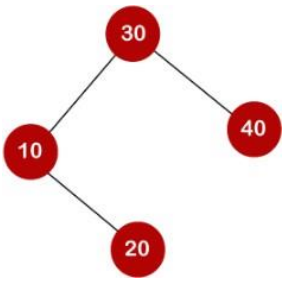
If we consider 40 as the root node then,
C[0,4] = min{c[0,3] + c[4,4]} + w[0,4]
= min{20 + 0} + 15
= 35
In the above cases, we have observed that 26 is the minimum cost; therefore, c[0,4] is equal to 26.

Cost Table:

i \ j	0	1	2	3	4
0	0	4	8^1	20^3	26^3
1		0	2	10^3	16^3
2			0	6	12^3
3				0	3
4					0

The optimal binary tree can be created as:





General formula for calculating the minimum cost is:

$$C[i,j] = min\{c[i,k-1] + c[k,j]\} + w(i,j)$$

Algorithm Analysis

Time complexity	Space complexity
$O(n^3)$ where n is the number of keys. This is because the algorithm uses a table of size $n \; x \; n$, and each cell in the table requires $O(n)$ operations to compute. ie. $(n \; x \; n \; x \; n) = \; n^3$	$O(n^2)$ which is the size of the table used to store intermediate results.

Greedy Technique

Definition

Greedy Algorithms work step-by-step, and always choose the steps which provide immediate profit/benefit. It chooses the "locally optimal solution", without thinking about future consequences.

- Greedy algorithms may not always lead to the optimal global solution, because it does not consider the entire data. The choice made by the greedy approach does not consider future data and choices.
- The greedy technique is used for optimization problems (find the maximum or minimum).

Components of Greedy Algorithm

Greedy algorithms consist of five components:-

- *Candidate set* – A solution is created from the set.
- *Selection function* – Used to choose the best candidate to be added to the solution.
- *Feasibility function* – Used to determine whether a candidate can be used to contribute to the solution.
- *Objective function* – Used to assign a value to a solution or a partial solution.
- *Solution function* – Used to indicate whether a complete solution has been reached.

Applications of Greedy Algorithms

- Finding an optimal solution (Activity selection, Fractional Knapsack, Job Sequencing, Huffman Coding).
- Finding close to the optimal solution for NP-Hard problems like Travelling Sales Person.

Properties required for the Greedy Algorithm

- *Greedy choice property:* We can reach a globally optimized solution by creating a locally optimized solution for each sub-module of the problem.
- *Optimal substructure*: Solutions to sub-problems of optimal solutions are also optimal.

Steps to achieve Greedy Algorithm

- *Feasible:* Algorithm should follow all constraints to return at least one solution to the problem.
- *Local optimal choice:* Make optimum choices from currently available options.
- *Unalterable:* We cannot change a decision at any subsequent point while execution

Activity selection problem – Uses greedy algorithm.

The activity selection problem is an optimization problem used to find the maximum number of activities a person can perform if they can only work on one activity at a time. This problem is also known as the interval scheduling maximization problem (ISMP).

Basic idea

- Choose the activity that ends first, and then choose subsequent activities that do not overlap with the previously chosen activity and have the earliest end time.
- Greedy approach is used - since we want to maximize the activities that can be executed, thus yielding an optimal solution.

Working process

- Two activities, say i and j, are said to be non-conflicting if, $S_j >= F_i$ where S_j denote the starting time of activitiy j and F_i refers to finishing time of the activitiy i.

Algorithm

- Sort the activities by their end time in ascending order.
- Select the first activity in the sorted list and mark it as selected.
- For each subsequent activity, if its start time is greater than or equal to the end time of the last selected activity, select it and mark it as selected.
- Repeat step 3 until no more activities are left.

Example problem

Problem 1

List of activities to be performed.

Activity	1	2	3	4	5	6
Start	5	1	0	3	5	8
Finish	9	2	6	4	7	9

Step1

Sort the activities in ascending order of Finish times.

Activity	1	2	3	4	5	6
Start	1	3	0	5	5	8
Finish	2	4	6	7	9	9

Step2

Select the first activity in the sorted list.

Activity	1	2	3	4	5	6
Start	1	3	0	5	5	8
Finish	2	4	6	7	9	9

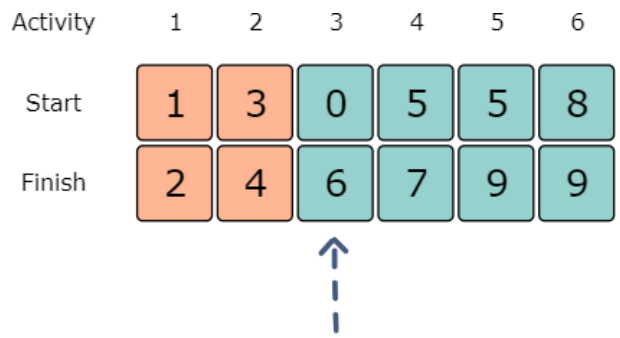
Step 3.1

Select the next activity in the sorted list its `start` time is greater than or equal to the `finish` time of the previously selected activity.

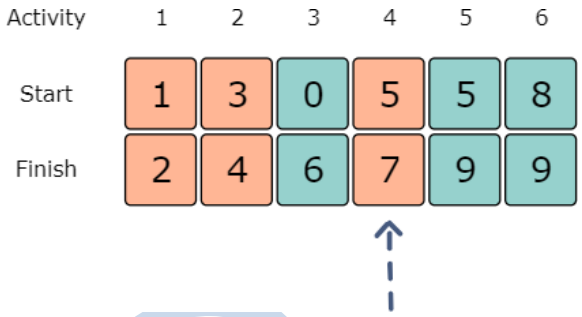
Activity	1	2	3	4	5	6
Start	1	3	0	5	5	8
Finish	2	4	6	7	9	9



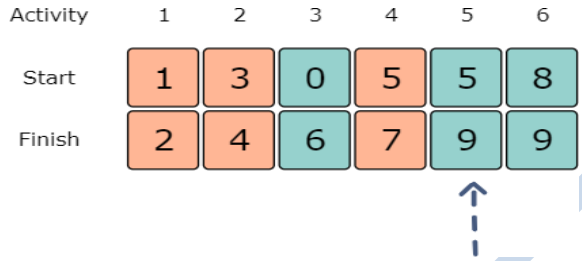
Step 3.2



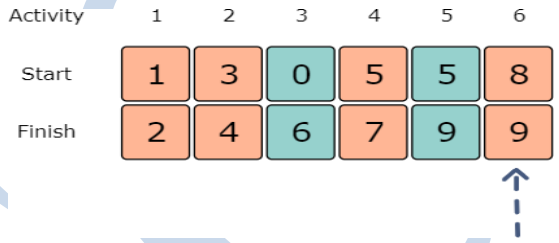
Step 3.3



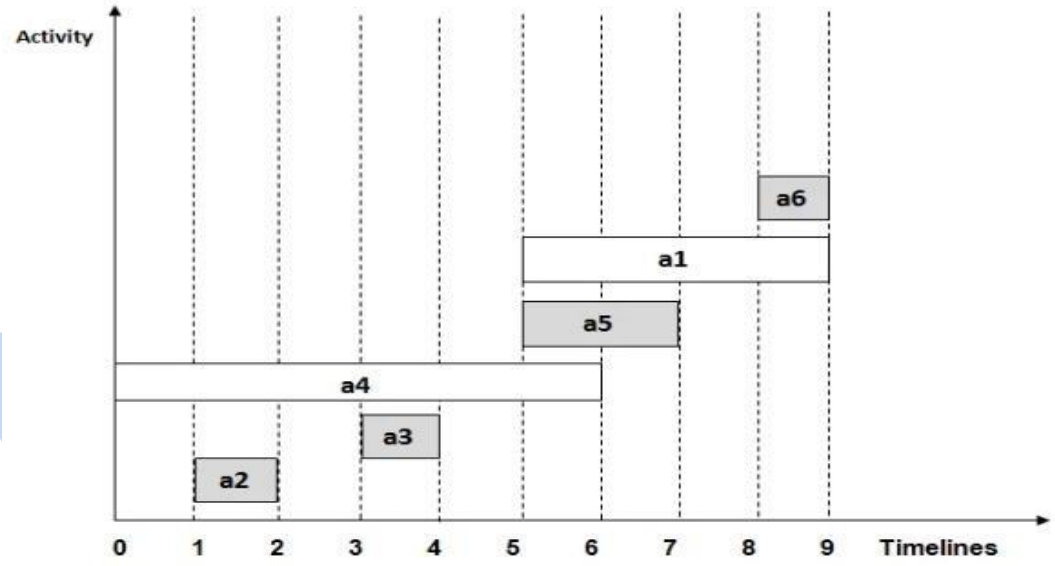
Step 3.4



Step 3.5



Hence, the execution schedule of maximum number of non-conflicting activities will be:
(1,2), (3,4), (5,7), (8,9)



Problem 2

Given 10 activities along with their start and end time as

A = (A1 A2 A3 A4 A5 A6 A7 A8 A9 A10)

S_i = (1,2,3,4,7,8,9,9,11,12)

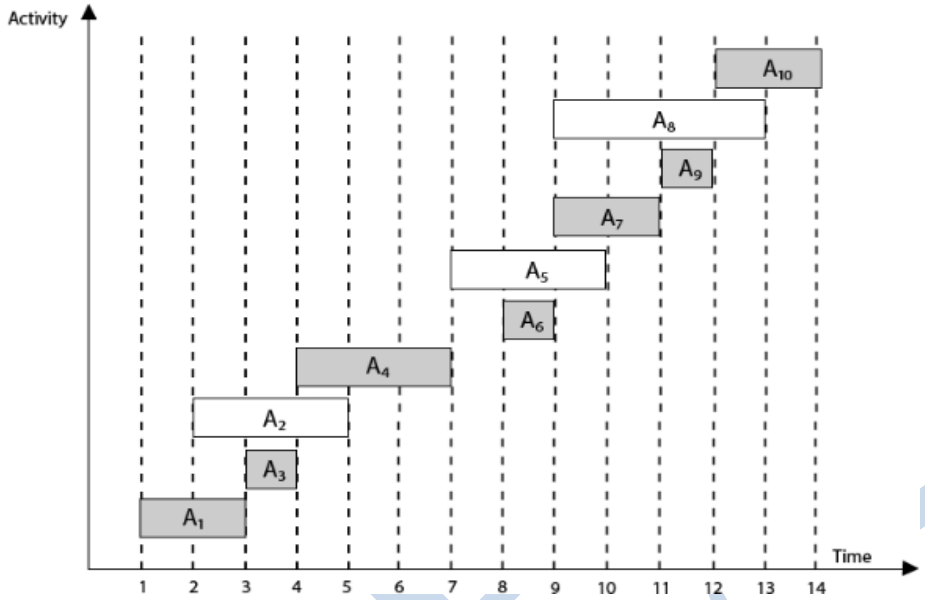
F_i = (3,5,4,7,10,9,11,13,12,14)

Activities	**A_1**	**A_2**	**A_3**	**A_4**	**A_5**	**A_6**	**A_7**	**A_8**		**A_{10}**
Start time	1	2	3	4	7	8	9	9		12
Finish time	3	5	4	7	10	9	11	13	12	14

Solution:

Sort the activities in ascending order of Finish times.

Activities	A_1	A_3	A_2	A_4	A_6	A_5	A_7	A_9	A_8	A_{10}
Start time	1	3	2	4	8	7	9	11	9	12
Finish time	3	4	5	7	9	10	11	12	13	14



Thus the final Activity schedule is:

$(A_1\ A_3\ A_4\ A_6\ A_7\ A_9\ A_{10})$

Algorithm Analysis

Time Complexity	Space Complexity
When activities are sorted - $O(n)$ When activities are **not** sorted - $O(n\ logn)$	$O(n)$ Need to store the start and end times of each activity in memory

Python Program

```
def activity_selection(start, finish):
    n = len(finish)
    activities = []
    i = 0
    activities.append(i)
    for j in range(n):
        if start[j] >= finish[i]:
            activities.append(j)
            i = j
    return activities

start = [1, 3, 0, 5, 8, 5]
finish = [2, 4, 6, 7, 9, 9]
selected_activities = activity_selection(start, finish)
print("Selected activities:", selected_activities)
```

Execution

```
Selected activities: [0, 1, 3, 4]
```

Optimal Merge Patterns

Merge a set of sorted patterns of different length into a single sorted pattern. We need to find an optimal solution, where the resultant pattern will be generated in minimum time. If the number of sorted patterns is given, there are many ways to merge them into a single sorted file. This merge can be performed pair wise. Hence, this type of merging is called as 2-way merge patterns.

Algorithm

i. Given a set of k sorted pattern of sizes n1, n2, ..., nk.
ii. Create an empty result pattern of size n1+n2+...+nk.
iii. Initialize a priority queue (min heap) of size k to hold the k input patterns.
iv. While the priority queue has more than one pattern:
    a. Remove the two smallest patterns from the priority queue.
    b. Merge the two arrays into a single sorted pattern using a modified merge function, which keeps track of the number of comparisons.
    c. Add the merged pattern to the priority queue.
v. The final pattern in the priority queue is the desired result.

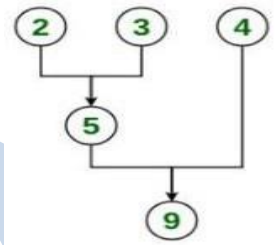
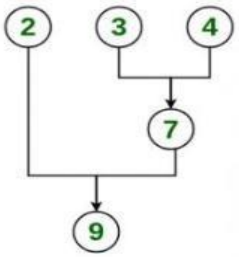
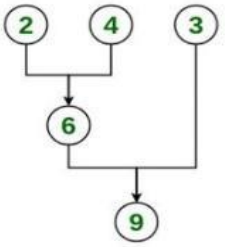
Example 1

Given 3 files with sizes 2, 3, 4 units. Find an optimal way to combine these files

Solution

Input: n = 3, size = {2, 3, 4}

Different ways to combine patterns are:

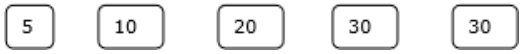
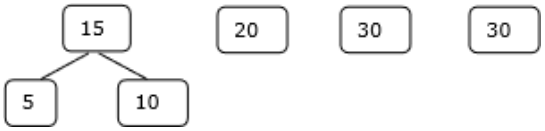
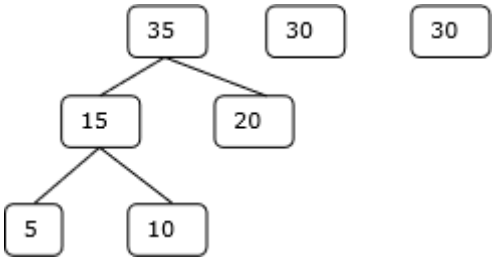
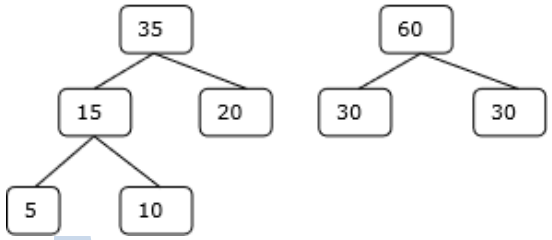
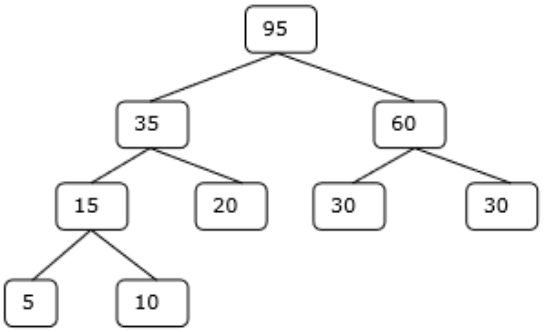
Method 1	*Method 2*	*Method 3*
 Cost = 5 + 9 = 14	 Cost = 7 + 9 = 16	 Cost = 6 + 9 = 15
Optimal Output using Method 1 = 14		

Example 2

Let us consider the given files, f1, f2, f3, f4 and f5 with 20, 30, 10, 5 and 30 number of elements respectively.

Method 1	*Method 2*
If merge operations are performed, then	Sorting the files - f4, f3, f1, f2, f5
M1 = merge f1 and f2 => 20 + 30 = 50	M1 = merge f4 and f3 => 5 + 10 = 15
M2 = merge M1 and f3 => 50 + 10 = 60	M2 = merge M1 and f1 => 15 + 20 = 35
M3 = merge M2 and f4 => 60 + 5 = 65	M3 = merge M2 and f2 => 35 + 30 = 65
M4 = merge M3 and f5 => 65 + 30 = 95	M4 = merge M3 and f5 => 65 + 30 = 95
Total number of operations is	Total number of operations is
50 + 60 + 65 + 95 = 270	**15 + 35 + 65 + 95 = 210**

Method 3 using Optimal merge pattern

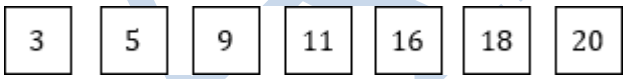
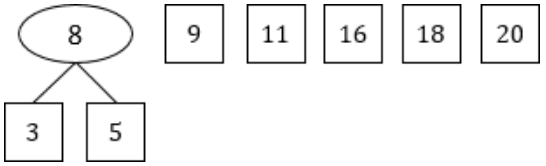
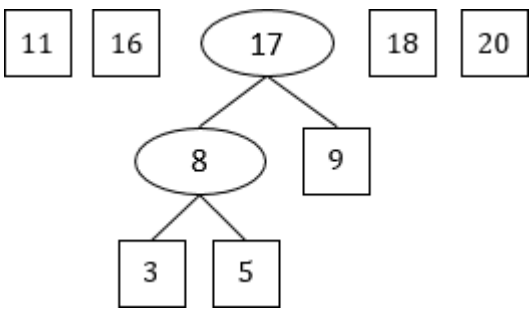
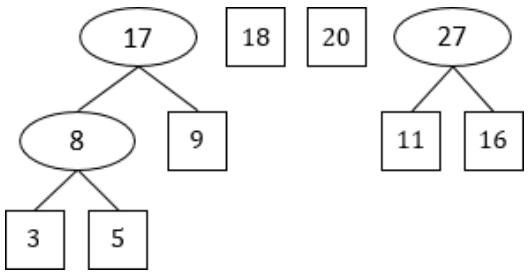
<table>
<tr>
<td>Initial Set

</td>
<td>Step 1

</td>
</tr>
<tr>
<td>Step 2

</td>
<td>Step 3

</td>
</tr>
<tr>
<td>Step 4

</td>
<td>Step 5
Total number of operations is
15 + 35 + 60 + 95 = 205</td>
</tr>
</table>

Example3

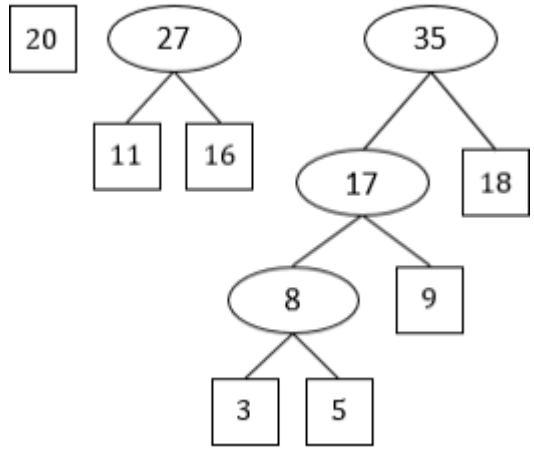
Consider the sequence {3, 5, 9, 11, 16, 18, 20}. Find optimal merge pattern for this data

Solution:

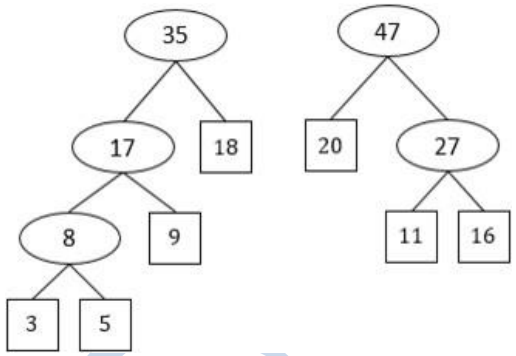
At each step, merge the two smallest sequences

<table>
<tr>
<td>Step 1: Given sequence is

</td>
<td>Step 2: Merge the two smallest sequences and sort in ascending order

</td>
</tr>
<tr>
<td>Step 3: Merge the two smallest sequences and sort in ascending order

</td>
<td>Step 4: Merge the two smallest sequences and sort in ascending order

</td>
</tr>
</table>

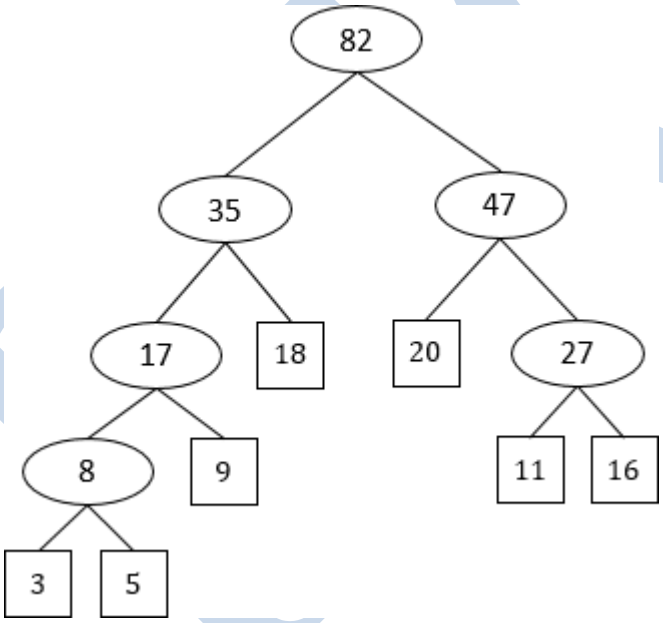
Step 5: Merge the two smallest sequences and sort in ascending order



Step 6: Merge the two smallest sequences and sort in ascending order



Step 7: Merge the two smallest sequences and sort in ascending order



Total time = 8 + 17 + 27 + 35 + 47 + 82 = 216

Algorithm Analysis

Time complexity	Space complexity
$O(nlogn)$ - where n is the total number of elements in all the input arrays. The algorithm uses a min-heap to find the minimum element among all the arrays repeatedly until all elements have been merged into a single array.	$O(n)$ The algorithm needs to store all the elements in memory in order to perform the merging operation.

Huffman coding

Definition

- Huffman coding is a ***lossless data compression algorithm*** that assigns variable-length codes to symbols in a message based on their frequency of occurrence.

Working of Huffman coding

Step 1 - Frequency analysis

- Analyze the message to determine the frequency of occurrence of each symbol.
    - This is done by counting the number of occurrences of each symbol in the message.
    - Arrange the nodes in ascending order based on their weights.

Step 2 - Building a tree

- Binary tree is constructed where each leaf node represents a symbol and its frequency of occurrence and each internal node represents the sum of the frequencies of its children.
    - The tree is built using a bottom-up approach, starting with the symbols with the lowest frequency and building up to the root of the tree.

Step 3 - Assigning codes

- Once the tree is built, each symbol is assigned a binary code based on its position in the tree.
    - A symbol that appears more frequently in the message will have a shorter code, while a symbol that appears less frequently will have a longer code.

Step 4 - Encoding the message

- Each symbol in the message is replaced with its corresponding binary code.
    - The encoded message is the concatenation of all the binary codes.

Step 5 - Decoding the message

- Encoded message is traversed from left to right, starting at the root of the tree. When a leaf node is reached, the corresponding symbol is output and the traversal continues from the root of the tree.
    - This process continues until the entire encoded message has been decoded.

Example

Suppose the string below is to be sent over a network.

Initial string

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

Each ASCII 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.

Using the Huffman Coding technique, we can compress the string to a smaller size

Step 1 - Calculate the frequency of each character in the string.

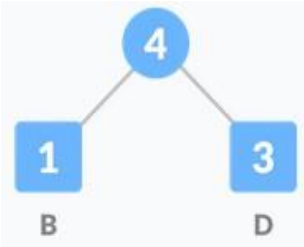
1	6	5	3
B	C	A	D

Step 2 - Sort the characters in increasing order of the frequency. These are stored in a priority queue Q.

1	3	5	6
B	D	A	C

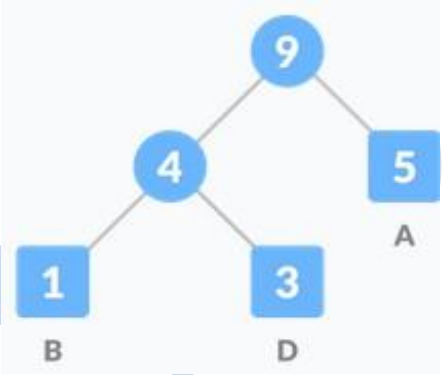
Step 3 - Make each unique character as a leaf node. Create an empty node *. Assign the minimum frequency to the left child of z and assign the second minimum frequency to the right child of *. Set the value of the * as the sum of the above two minimum frequencies.

4	5	6
*	A	C



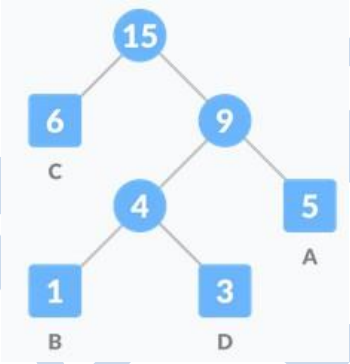
Step 4 - Remove these two minimum frequencies from Q and add the sum into the list of frequencies (* denote the internal nodes in the figure above).

6	9	
C	*	

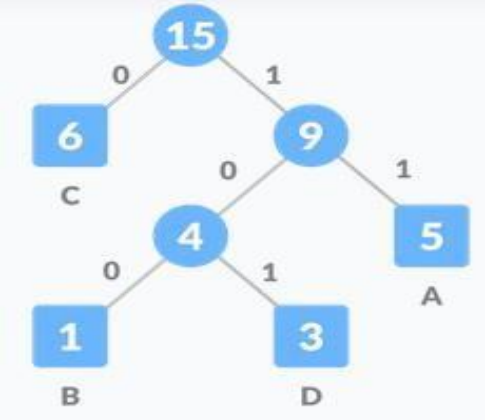


Repeat steps 3 to 5 for all the characters.

15
*



Step 5 - For each non-leaf node, assign 0 to the left edge and 1 to the right edge.



Step 6 – Encoding the Message

For sending the above string over a network, we have to send the tree as well as the above compressed-code. The total size is given by the table below.

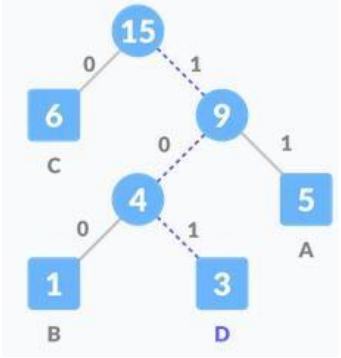
Character	Frequency	Code	Size
A	5	11	5*2 = 10
B	1	100	1*3 = 3
C	6	0	6*1 = 6
D	3	101	3*3 = 9
4 * 8 = 32 bits	**15 bits**		**28 bits**

Without encoding, the total size of the string = 120 bits.

After encoding the size is reduced to = 32 + 15 + 28 = 75 bits.

Step 7 – Decoding the Message

For decoding the code, we can take the code and traverse through the tree to find the character. Let 101 is to be decoded, we can traverse from the root as in the figure below.



Algorithm Analysis

Time complexity	Space complexity
$O(nlogn)$ - where n is the number of characters in the message It involves building a binary tree from the characters in the message, and sorting the tree nodes by frequency	$O(n)$ Storage for the binary tree and the codes assigned to each character.

APEC