Syllabus:

Dynamic Programming: Introduction, DP Techniques, Applications – Matrix Chain Multiplication, Optimal Binary Search Tree, All Pairs Shortest Paths, Travelling Salesperson Problem, Climbing Stairs, Min Cost Climbing Stairs, Maximum Sub Array, Number of Corner Rectangles, 0/1 Knapsack Problem.

Strings: Introduction, Count Substrings with Only One Distinct Letter, Valid Word Abbreviation, Longest Repeating Substring, Longest Common Subsequence, Longest Increasing Subsequence.

PART-1: Dynamic Programming.

Introduction to Dynamic Programming:

In our previous article on recursion, we explored how we can break a problem into smaller sub-problems and solve them individually. However, recursion is not the most optimal technique and has its share of obstacles. Fortunately, there is a powerful algorithmic technique called dynamic programming that helps us overcome the hurdles posed by recursion and solve problems optimally.

- Dynamic Programming (DP) is an algorithmic technique used when solving an optimization problem by breaking it down into simpler subproblems and utilizing the fact that the optimal solution to the overall problem depends upon the optimal solution to its subproblems.

  Before we delve into DP, let us look at the drawbacks of recursion:

- **It is not efficient in terms of memory:** Since recursion involves function calls, each recursive call creates an entry for all the variables and constants in the function stack. These values are kept there until the function returns. Therefore, recursion is always limited by the stack space in the system.
- If a recursive function requires more memory than what is available in the stack, a common and prevalent error called stack overflow occurs.
- **It is not fast:** Iteration (using loops) is faster than recursion because every time a function is called, there is an overhead of allocating space for the function and all its data in the function stack. This causes a slight delay in recursive functions when compared to iteration.
- In recursion, the same function can be called multiple times with the same arguments. In other words, the same result is calculated multiple times instead of just once. In dynamic programming, a recursive function is optimized by storing the intermediate results in a data structure.
- We store these results so that they are only calculated once. In other words, any recursive function in which the same results are calculated multiple times can be converted into DP. **Let us look at the Fibonacci number to get a better idea.**

Fibonacci Algorithm:

```
def fib(n):
    if(n == 1 or n == 2):
        return 1
    return fib(n - 1) + fib(n - 2)

n = int(input('Enter the nth term: '))
print(fib(n))
```

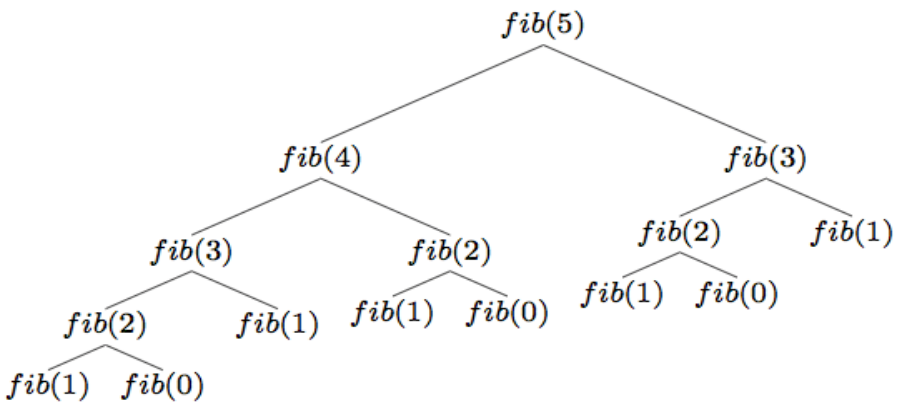
The **above code** is the **recursive implementation of Fibonacci numbers**. The fib() function is called twice: once to calculate (n - 1) and once to calculate (n - 2). Therefore, the time complexity of this function is 2 raised to the power 'n'.

For example:

Consider the case when n = 5.

1. We call fib(5), which inturn calls fib(4) and fib(3)
2. fib(4) inturn calls fib(3) and fib(2)
3. fib(3) inturn calls fib(2) and fib(1)
4. fib(2) and fib(1) return 1, and the program terminates.

As you can observe, fib(4), fib(3), fib(2) are called multiple times. Take a look at the image below to get a better idea:



- A better way to solve this would be to store all the intermediate results to calculate them once.
- There are **two ways** to do this, but in this article, we will explore **one method** called **tabulation**. The **other method** is called **memorization,** in which we store the intermediate results and return them within the function itself.
- To accomplish this, we can use any data structure such as a dictionary or an array.

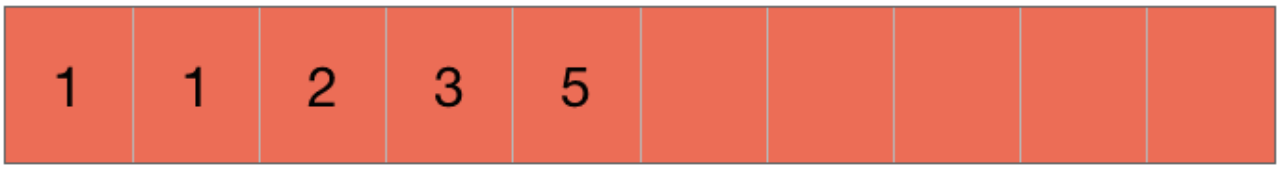
- **In the tabulation method,** we store the results of each function call in a data structure. Whenever an already stored value is needed, we fetch the value from the data structure instead of computing it repeatedly.

Using Dynamic Programming:

```
class Fib
{
```

```
	static int[] F = new int[50];
	public static void initF()
	{
		for(int i=0; i<50; i++)
		{
			F[i] = -1;
		}
	}

	public static int dynamicFibonacci(int n)
	{
		if (F[n] < 0)
		{
			if (n==0)
			{
				F[n] = 0;
			}
			else if (n == 1) {
				F[n] = 1;
			}
			else {
				F[n] = dynamicFibonacci(n-1) + dynamicFibonacci(n-2);
			}
		}
		return F[n];
	}

	public static void main(String[] args)
	{
		initF();
		System.out.println(dynamicFibonacci(46));
	}
}
```

Here, we are first checking if the result is already present in the array or not if F[n] == -1. If it is not, then we are calculating the result and then storing it in the array F and then returning it return F[n].



F

If in F, return.
Otherwise, calculate

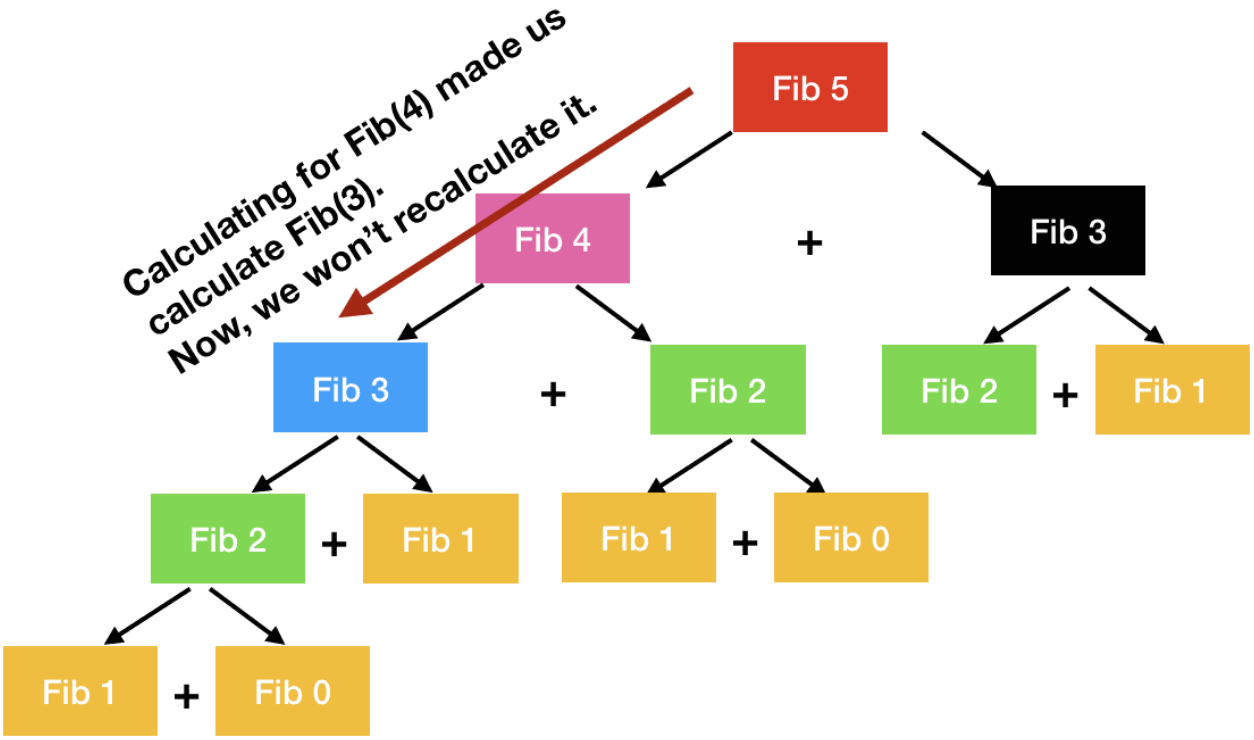
Running this code for the 100^{th} term gave the result almost instantaneously and this is the power of dynamic programming.
But are we sacrificing anything for the speed? Yes, memory. Dynamic programming basically trades time with memory. Thus, we should take care that not an excessive amount of memory is used while storing the solutions.

Two Approaches of Dynamic Programming

There are two approaches of the dynamic programming. The first one is the top-down approach and the second is the bottom-up approach.

Top-Down Approach

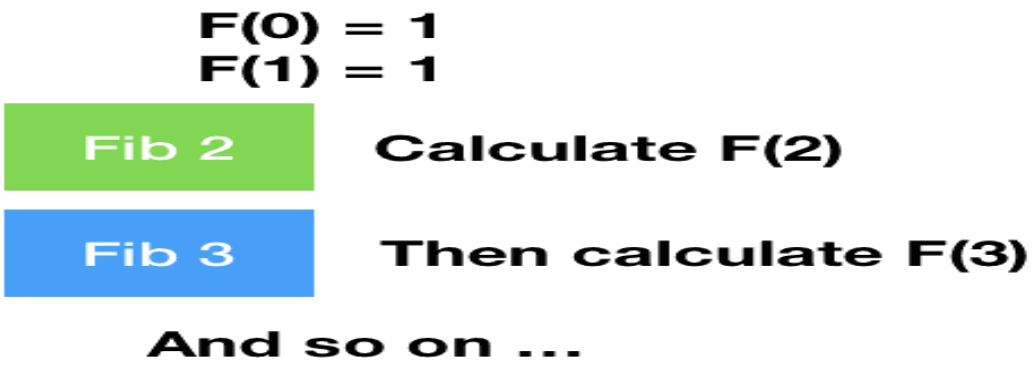
The way we solved the Fibonacci series was the top-down approach. We just start by solving the problem in a natural manner and stored the solutions of the subproblems along the way. We also use the term **memoization**, a word derived from memo for this.



In other terms, it can also be said that we just hit the problem in a natural manner and hope that the solutions for the subproblem are already calculated and if they are not calculated, then we calculate them on the way.

Bottom-Up Approach

The other way we could have solved the Fibonacci problem was by starting from the bottom i.e., start by calculating the 2nd2nd term and then 3rd3rd and so on and finally calculating the higher terms on the top of these i.e., by using these values.



We use a term **tabulation** for this process because it is like filling up a table from the start.

Let's again write the code for the Fibonacci series using bottom-up approach.

```
class Fib
{
```

```
    static int[] F = new int[50];
    public static int fibonacciBottomUp(int n)
    {
        F[n] = 0;
        F[1] = 1;

        for(int i=2; i<=n; i++)
        {
            F[i] = F[i-1] + F[i-2];
        }
        return F[n];
    }

    public static void main(String[] args)
    {
        System.out.println(fibonacciBottomUp(46));
    }
}
```

- As said, we started calculating the Fibonacci terms from the starting and ended up using them to get the higher terms.
- Also, the order for solving the problem can be flexible with the need of the problem and is not fixed. So, we can solve the problem in any needed order.
- Generally, we need to solve the problem with the smallest size first. So, we start by sorting the elements with size and then solve them in that order.
- Let's compare memoization and tabulation and see the pros and cons of both.

Memoization V/S Tabulation

- Memoization is indeed the natural way of solving a problem, so coding is easier in memoization when we deal with a complex problem. Coming up with a specific order while dealing with lot of conditions might be difficult in the tabulation.
- Also think about a case when we don't need to find the solutions of all the subproblems. In that case, we would prefer to use the memoization instead.
- However, when a lot of recursive calls are required, memoization may cause memory problems because it might have stacked the recursive calls to find the solution of the deeper recursive call but we won't deal with this problem in tabulation.
- Generally, memoization is also slower than tabulation because of the large recursive calls.

Applications:

1. **Matrix Chain Multiplication.**

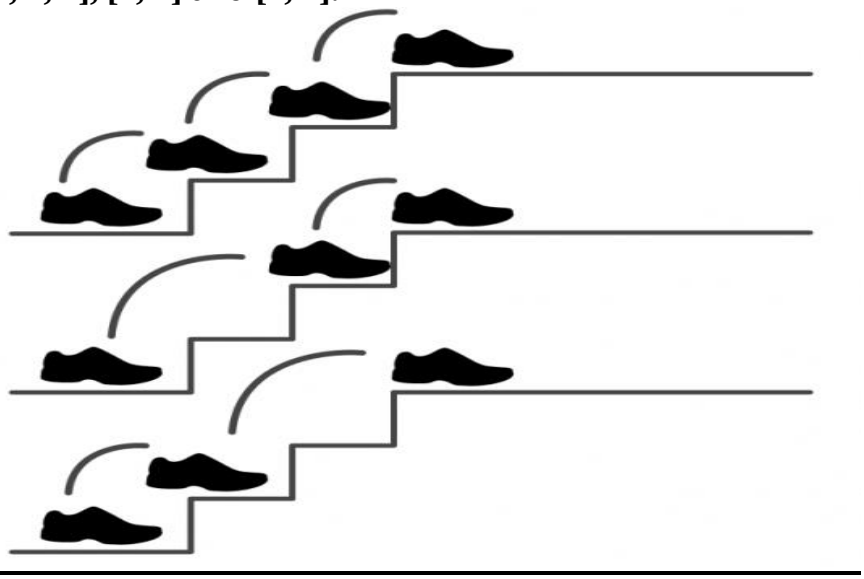
2. **Optimal Binary Search Tree.**
3. **All Pairs Shortest Paths.**
4. **Travelling Salesperson Problem.**
5. **0/1 Knapsack Problem.**
6. **Climbing Stairs.**
7. **Min Cost Climbing Stairs.**
8. **Maximum Sub Array.**
9. **Number of Corner Rectangles.**

1. <u>Climbing Stairs:</u>

Given a staircase of **N** steps and you can either climb 1 or 2 steps at a given time. The task is to return the count of distinct ways to climb to the top.
Note: The order of the steps taken matters.
Examples:
Input: N = 3

Output: 3

Explanation:
There are three distinct ways of climbing a staircase of 3 steps :
[1, 1, 1], [2, 1] and [1, 2].



Input: N = 2
Output: 2
Explanation:
There are two distinct ways of climbing a staircase of 3 steps :
[1, 1] and [2].

Input: n = 4
Output: 5
(1, 1, 1, 1), (1, 1, 2), (2, 1, 1), (1, 2, 1), (2, 2)

Java Program for Climbing Stairs Using Dynamic Programming(Bottom up Approach):

ClimbingStairs.java

```
import java.util.*;

class ClimbingStairs
{
	// Recursive implementation
	public int climbStairsRecur(int n)
	{
		//System.out.println("n " + n);
		if(n <= 2)
			return n;

	return climbStairsRecur(n-1) + climbStairsRecur(n-2);
	}

	// top down approach - Memoization
	public int climbStairsMem(int n)
	{
	if(n <= 2)
			return n;
		int[] dp = new int[n + 1];
		return solve(n, dp);
	}

	int solve(int n, int[] dp)
	{
		if(n <= 2)
			return n;
		//System.out.println(Arrays.toString(dp) + " n " + n);
		if(dp[n] != 0)
			return dp[n];
		return dp[n] = solve(n-1, dp) + solve(n-2, dp);
	}

	// bottom up approach - tabulation
	public int climbStairsTab(int n)
	{
	if (n <= 2) {
			return n;
```

```
        }
        int[] dp = new int[n + 1];
        dp[1] = 1;
        dp[2] = 2;
        for (int i = 3; i <= n; i++) {
                        dp[i] = dp[i - 1] + dp[i - 2];
        }
                //System.out.println("Tabulation" + Arrays.toString(dp));
        return dp[n];
        }

        public static void main(String[] args)
        {
                Scanner sc=new Scanner(System.in);
                int n=sc.nextInt();
                System.out.println(new ClimbingStairs().climbStairsRecur(n));
                System.out.println(new ClimbingStairs().climbStairsMem(n));
                System.out.println(new ClimbingStairs().climbStairsTab(n));
        }
}
```

Sample i/p & o/p:

Case=1
input =9
output =55

Case =2
input =2
output =2

Case =3
input =20
output =10946

2. **Min Cost Climbing Stairs:**

- You are given an integer array cost where cost[i] is the cost of i step on a staircase.
- Once you pay the cost, you can either climb one or two steps.
- You can either start from the step with index 0 , or the step with index 1 .
- Return the minimum cost to reach the top of the floor.

Example 1:
Input: cost = [10,15,20]

Output: 15

Explanation: You will start at index 1.
- Pay 15 and climb two steps to reach the top.
The total cost is 15.

Example 2:
Input: cost = [1,100,1,1,1,100,1,1,100,1]
Output: 6
Explanation: You will start at index 0.
- Pay 1 and climb two steps to reach index 2.
- Pay 1 and climb two steps to reach index 4.
- Pay 1 and climb two steps to reach index 6.
- Pay 1 and climb one step to reach index 7.
- Pay 1 and climb two steps to reach index 9.
- Pay 1 and climb one step to reach the top.
The total cost is 6.

Approach:

- At each step, we can consider the answer to be the combined cost of the current step, plus the lesser result of the total cost of each of the solutions starting at the next two steps.
- This means that, thinking backwards, we can solve for the smallest problem first, and then build down from there.
- For the last two steps, the answer is clearly their individual cost. For the third to last step, it's that step's cost, plus the lower of the last two steps.
- Now that we know that, we can store that data for later use at lower steps. Normally, this would call for a DP array, but in this case, we could simply store the values in-place.
- (Note: If we choose to not modify the input, we could create a DP array to store this information at the expense of O(N) extra space.)
- So we should iterate downward from the end, starting at the third step from the end, and update the values in cost[i] with the best total cost from cost[i] to the end.
- Then, once we reach the bottom of the steps, we can choose the best result of cost[0] and cost[1] and return our answer.

Time Complexity: O(N) where N is the length of cost
Space Complexity: O(1) or O(N) if we use a separate DP array

Java Program For Minimum Cost Climbing Stairs Using Dynamic Programming

MinCostClimbingStairs.java

```
import java.util.*;

class MinCostClimbingStairs
{
```

```
    public int minCostRecur(int n,int[] cost)
    {
       return Math.min(minCost(cost, n-1), minCost(cost, n-2));
    }

    private int minCost(int[] cost, int n)
    {
       if (n < 0) return 0;
       if (n==0 || n==1) return cost[n];
       return cost[n] + Math.min(minCost(cost, n-1), minCost(cost, n-2));
    }

    public int minCostMemoized(int n, int[] cost, int []dp)
    {
            if(n < 0)
                    return 0;

            if (n==0 || n==1)
                    return dp[n];

            if (dp[n] != -1)
     return dp[n];

            return dp[n] = cost[n] + Math.min(minCostMemoized(n-1, cost, dp),
minCostMemoized(n-2, cost, dp));
    }

    int minimumCost(int n, int cost[])
    {
            int dp[] = new int[n];
            Arrays.fill(dp, -1);
            dp[0] = cost[0];
            dp[1] = cost[1];
            int top = Math.min(minCostMemoized(n - 1, cost, dp), minCostMemoized(n -
2, cost, dp));
            return top;
    }

    public int minCostDP(int[] cost)
    {
            int n = cost.length;
            int[] dp = new int[n];
            dp[0] = cost[0];
            dp[1] = cost[1];
            for (int i=2; i<n; i++)
```

```
            {
                dp[i] = cost[i] + Math.min(dp[i-1], dp[i-2]);
            }
            //System.out.println(Arrays.toString(dp));
            return Math.min(dp[n-1], dp[n-2]);
        }

        public static void main(String[] args)
        {
            Scanner sc=new Scanner(System.in);
            int n=sc.nextInt();
            int ar[]=new int[n];
            for(int i=0;i<n;i++)
                ar[i]=sc.nextInt();

            System.out.println(new MinCostClimbingStairs().minCostRecur(n, ar));
            System.out.println(new MinCostClimbingStairs().minimumCost(n, ar));
            System.out.println(new MinCostClimbingStairs().minCostDP(ar));
        }
}
```

Sample i/p & o/p:

case =1
input =3
20 30 40
output =30

case =2
input =7
2 3 50 2 2 50 2
output =9

3. **0/1 Knapsack Problem:**

Given **N** items where each item has some weight and profit associated with it and also given a bag with capacity **W**, [i.e., the bag can hold at most **W** weight in it]. The task is to put the items into the bag such that the sum of profits associated with them is the maximum possible.

Note: The constraint here is we can either put an item completely into the bag or cannot put it at all [It is not possible to put a part of an item into the bag].

Examples:
Input: *N = 3, W = 4, profit[] = {1, 2, 3}, weight[] = {4, 5, 1}*
Output: *3*

Explanation: *There are two items which have weight less than or equal to 4. If we select the item with weight 4, the possible profit is 1. And if we select the item with weight 1, the possible profit is 3. So the maximum possible profit is 3. Note that we cannot put both the items with weight 4 and 1 together as the capacity of the bag is 4.*

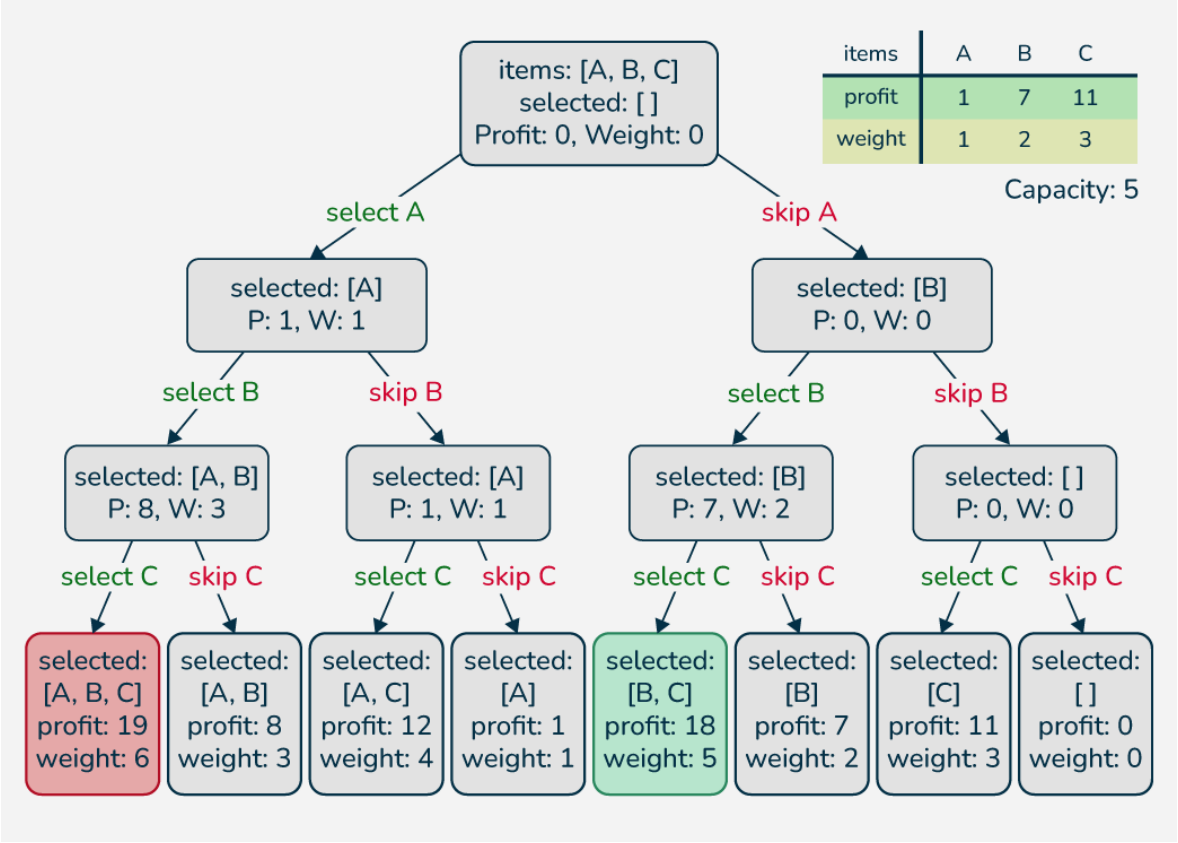
Input: *N = 3, W = 3, profit[] = {1, 2, 3}, weight[] = {4, 5, 6}*
Output: *0*

Recursion Approach for 0/1 Knapsack Problem:
To solve the problem follow the below idea:

A simple solution is to consider all subsets of items and calculate the total weight and profit of all subsets. Consider the only subsets whose total weight is smaller than W. From all such subsets, pick the subset with maximum profit.

Optimal Substructure: *To consider all subsets of items, there can be two cases for every item.*

- ***Case 1:*** *The item is included in the optimal subset.*
- ***Case 2:*** *The item is not included in the optimal set.*



Recursion Tree for 0-1 Knapsack

Follow the below steps to solve the problem:

The maximum value obtained from ‘N’ items is the max of the following two values.

- Case 1 (include the Nth item): Value of the Nth item plus maximum value obtained by remaining N-1 items and remaining weight i.e. (W-weight of the Nth item).
- Case 2 (exclude the Nth item): Maximum value obtained by N-1 items and W weight.
- If the weight of the ‘Nth‘ item is greater than ‘W’, then the Nth item cannot be included and **Case 2** is the only possibility.

```
/* A recursive implementation of 0-1 Knapsack problem */

class ZeroOneRecursion{

   static int knapSack(int W, int wt[], int val[], int n) {

      // Base Case

      if (n == 0 || W == 0)

         return 0;

      // If weight of the nth item is more than Knapsack capacity W,

      // then this item cannot be included in the optimal solution

      if (wt[n - 1] > W)

         return knapSack(W, wt, val, n - 1);


      // Return the maximum of two cases:

      // (1) nth item included

      // (2) not included

      else

         return Math.max(knapSack(W, wt, val, n - 1),

          val[n - 1] + knapSack(W - wt[n-1], wt, val, n-1));

   }

   // Driver code

   public static void main(String args[])

   {

      int profit[] = new int[] { 60, 100, 120 };

      int weight[] = new int[] { 10, 20, 30 };

      int W = 50;
```

```
        int n = profit.length;
        System.out.println(knapSack(W, weight, profit, n));
    }
}
```

Output

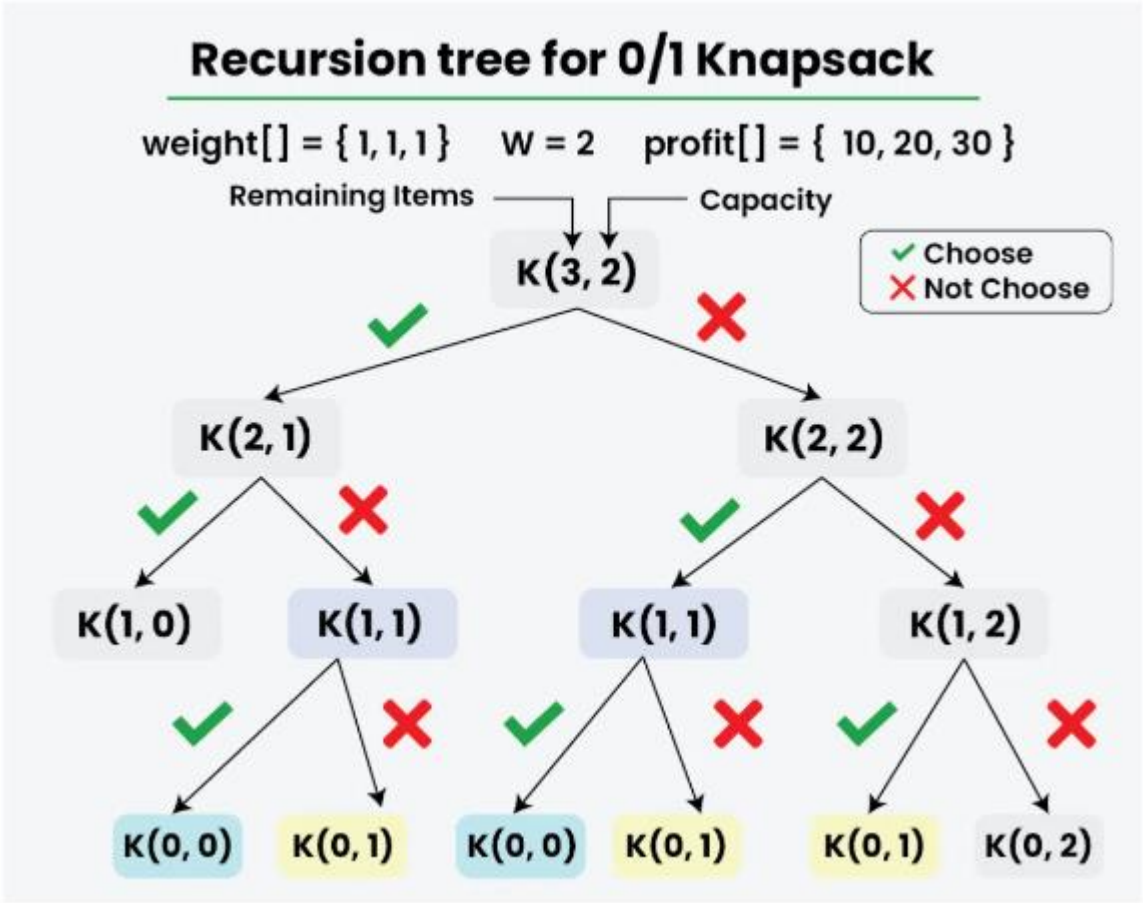
220

Time Complexity: O(2N)
Auxiliary Space: O(N), Stack space required for recursion

Dynamic Programming Approach for 0/1 Knapsack Problem

Memoization Approach for 0/1 Knapsack Problem:

Note: In Recursive approach, function using recursion computes the same subproblems again and again. See the following recursion tree, K(1, 1) is being evaluated twice.



As there are repetitions of the same subproblem again and again we can implement the following idea to solve the problem.

If we get a subproblem the first time, we can solve this problem by creating a 2-D array that can store a particular state (n, w). Now if we come across the same state (n, w) again instead of calculating it in exponential complexity we can directly return its result stored in the table in constant time.

```
// Here is the top-down approach of dynamic programming

import java.io.*;

class ZeroOneMemo {

    // Returns the value of maximum profit

    static int knapSackRec(int W, int wt[], int val[], int n, int[][] dp)

    {

        // Base condition

        if (n == 0 || W == 0)

            return 0;


        if (dp[n][W] != -1)

            return dp[n][W];


        if (wt[n - 1] > W)

            // Store the value of function call stack in table before return

            return dp[n][W] = knapSackRec(W, wt, val, n - 1, dp);

        else

            // Return value of table after storing

            return dp[n][W]  = Math.max((val[n - 1]

                    + knapSackRec(W - wt[n - 1], wt, val, n - 1, dp)),

                   knapSackRec(W, wt, val, n - 1, dp));

    }


    static int knapSack(int W, int wt[], int val[], int N)

    {

        // Declare the table dynamically

        int dp[][] = new int[N + 1][W + 1];
```

```
        // Loop to initially filled the
        // table with -1
        for (int i = 0; i < N + 1; i++)
            for (int j = 0; j < W + 1; j++)
                dp[i][j] = -1;

        return knapSackRec(W, wt, val, N, dp);
    }

    // Driver Code
    public static void main(String[] args)
    {
        int profit[] = { 60, 100, 120 };
        int weight[] = { 10, 20, 30 };

        int W = 50;
        int N = profit.length;

        System.out.println(knapSack(W, weight, profit, N));
    }
}
```

Output

220

Time Complexity: O(N * W). As redundant calculations of states are avoided.
Auxiliary Space: O(N * W) + O(N). The use of a 2D array data structure for storing intermediate states and O(N) auxiliary stack space(ASS) has been used for recursion stack

Bottom-up Approach for 0/1 Knapsack Problem:

To solve the problem follow the below idea:

Since subproblems are evaluated again, this problem has Overlapping Sub-problems property. So the 0/1 Knapsack problem has both properties of a dynamic programming problem. Like other typical Dynamic Programming(DP) problems, re-computation of the same subproblems can be avoided by constructing a temporary array K[][] in a bottom-up manner.

Illustration:

Below is the illustration of the above approach:

Let, ***weight[] = {1, 2, 3}, profit[] = {10, 15, 40}, Capacity = 6***

- *If no element is filled, then the possible profit is 0.*

weight--> item i/	0	1	2	3	4	5	6
0	0	0	0	0	0	0	0
1							
2							
3							

- ***For filling the first item in the bag:*** *If we follow the above mentioned procedure, the table will look like the following.*

weight--> item i/	0	1	2	3	4	5	6
0	0	0	0	0	0	0	0
1	0	10	10	10	10	10	10
2							
3							

- ***For filling the second item:***
  When jthWeight = 2, then maximum possible profit is max (10, DP[1][2-2] + 15) = max(10, 15) = 15.
  When jthWeight = 3, then maximum possible profit is max(2 not put, 2 is put into bag) = max(DP[1][3], 15+DP[1][3-2]) = max(10, 25) = 25.

weight→ item↓	0	1	2	3	4	5	6
0	0	0	0	0	0	0	0
1	0	10	10	10	10	10	10
2	0	10	15	25	25	25	25
3							

- ***For filling the third item:***
  When jthWeight = 3, the maximum possible profit is max(DP[2][3], 40+DP[2][3-3]) = max(25, 40) = 40.
  When jthWeight = 4, the maximum possible profit is max(DP[2][4], 40+DP[2][4-3]) = max(25, 50) = 50.
  When jthWeight = 5, the maximum possible profit is max(DP[2][5], 40+DP[2][5-3]) = max(25, 55) = 55.
  When jthWeight = 6, the maximum possible profit is max(DP[2][6], 40+DP[2][6-3]) = max(25, 65) = 65.

weight→ item↓/	0	1	2	3	4	5	6
0	0	0	0	0	0	0	0
1	0	10	10	10	10	10	10
2	0	10	15	25	25	25	25
3	0	10	15	40	50	55	65

```
// A Dynamic Programming based solution for 0-1 Knapsack problem


import java.io.*;
class ZeroOneTab {
    // Returns the maximum value that can be put in a knapsack of capacity W
static int KnapSackDP(int[] val, int[] weight, int n, int wt)
{
        int i, w;
        int dp[][] = new int[n + 1][wt + 1];
         // Build table dp[][] in bottom up manner
        for (i = 1; i <= n; i++){
                for (w = 1; w <= wt; w++){
                        if (w >= weight[i-1])      // included
                        {
                                dp[i][w] = Math.max(dp[i-1][w], dp[i-1][w - weight[i-1]] + val[i-1]);
                        }
                        else       // not included
                        {
```

```
                                dp[i][w] = dp[i-1][w];
                            }
                        }
                    }
                    System.out.println(Arrays.deepToString(dp));
                    return dp[n][wt];
                    }
}

   // Driver code
   public static void main(String args[])
   {
      int profit[] = new int[] { 60, 100, 120 };
      int weight[] = new int[] { 10, 20, 30 };
      int W = 50;
      int n = profit.length;
      System.out.println(knapSack(W, weight, profit, n));
   }
}
```

Output

220

Time Complexity: O(N * W). where 'N' is the number of elements and 'W' is capacity.
Auxiliary Space: O(N * W). The use of a 2-D array of size 'N*W'.

4. Maximum Sub Array:

Given an integer array nums , find the contiguous subarray (containing at least one number) which has the largest sum and return *its sum*.
A **subarray** is a **contiguous** part of an array.

Example 1:
Input: nums = [-2,1,-3,4,-1,2,1,-5,4]
Output: 6
Explanation: [4,-1,2,1] has the largest sum = 6.

Example 2:
Input: nums = [1]
Output: 1

Example 3:
Input: nums = [5,4,-1,7,8]
Output: 23

Constraints:
• 1 <= nums.length <= 10
• -10 <= nums[i] <= 10

Approach:

"Maximum Subarray Problem" and came across Kadane's Algorithm

The **maximum subarray problem** is the task of finding the largest possible sum of a contiguous subarray, within a given one-dimensional array A[1…n] of numbers.

For example, for the array given above, the contiguous subarray with the largest sum is [4, -1, 2, 1], with sum 6. We would use this array as our example for the rest of this article. Also, we would assume this array to be zero-indexed, *i.e.* -2 would be called as the '0th' element of the array and so on. Also, ***A[i]*** would represent the value at index ***i*.**

Java Program For Maximum Sub Array using Dynamic Programming:

MaxSubArray.java

```
import java.util.*;

class MaxSubArray
{
	public int maxSubArray(int[] arr)
	{
		int n = arr.length;
		int max = Integer.MIN_VALUE, sum = 0;

		for (int i = 0; i < n; i++)
		{
			sum += arr[i];
```

```
                if (sum > max)
                        max = sum;
                //System.out.println("sum " + sum + " max " + max);

                if (sum < 0)   // Greedy: negative sum is worst than restarting with a
new range
                sum = 0;
            }
            return max;
        }

        public int maxSubArrayDP(int[] arr)
        {
            int n = arr.length;
            int[] dp = new int[n]; //dp[i] means the maximum subarray ending with A[i];
            dp[0] = arr[0];
            int max = dp[0];

            for(int i = 1; i < n; i++)
            {
                // Either take the current element and continue with the previous sum
OR restart a new range
                dp[i] = arr[i] + (dp[i - 1] > 0 ? dp[i - 1] : 0);
                max = Math.max(max, dp[i]);
                //System.out.println("max " + max + " dp " + Arrays.toString(dp));
            }
            //System.out.println(Arrays.toString(dp));
            return max;
        }

        public static void main(String[] args)
        {
            Scanner sc=new Scanner(System.in);
            int n=sc.nextInt();
            int ar[]=new int[n];
            for(int i=0;i<n;i++)
                ar[i]=sc.nextInt();
            System.out.println(new MaxSubArray().maxSubArray(ar));
            System.out.println(new MaxSubArray().maxSubArrayDP(ar));
        }
}
```

Sample i/p & output:

case =1

input =9
-2 1 -3 4 -1 2 1 -5 4
output =6

case =2
input =2
1 -2
output =1

case =3
input =20
6 -2 -7 7 -4 3 5 -5 7 -7 9 4 7 -5 -7 5 1 2 -3 3
output =26

5. Matrix Chain Multiplication:

If a chain of matrices is given, we have to find the minimum number of the correct sequence of matrices to multiply.
The problem is not actually to perform the multiplications, but merely to decide in which order to perform the multiplications.

Note: no matter how we parenthesize the product, the result will be the same.

For example, if we had four matrices A, B, C, and D, we would have:
(ABC)D = (AB)(CD) = A(BCD) = ...

However, the order in which we parenthesize the product affects the number of simple arithmetic operations needed to compute the product, or the efficiency.

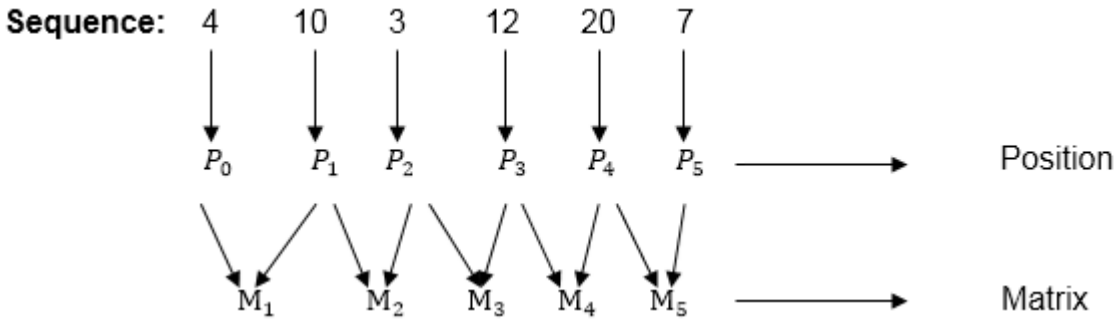
For example, suppose A is a 10 × 30 matrix, B is a 30 × 5 matrix, and C is a 5 × 60 matrix.
Then,
(AB)C = (10×30×5) + (10×5×60) = 1500 + 3000 = 4500 operations
A(BC) = (30×5×60) + (10×30×60) = 9000 + 18000 = 27000 operations.

Clearly the first parenthesization requires less number of operations.

Note that in the matrix-chain multiplication problem, we are not actually multiplying matrices. Our goal is only to determine an order for multiplying matrices that has the lowest cost.

Recursion Approach:
Here is the recursive algorithm :

1) Take the sequence of matrices and separate into two sub sequences
2) Find the minimum cost of multiplying each sub sequence
3) Add these cost together, and add in the cost of mutiplying the two result matrices
4) Do this for each position at which the sequence of matrices can be split,
and take the minimum over all of them



Java Program For Chain Matrix Multiplication using Recursion:

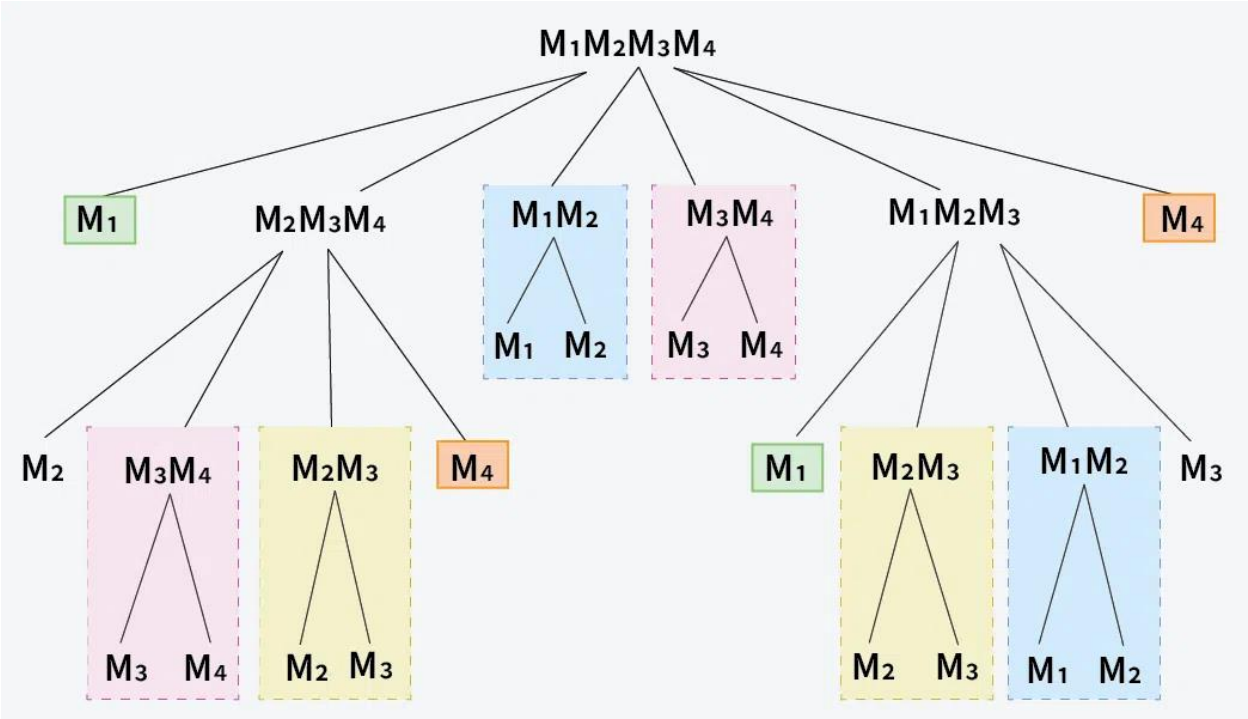
MatrixChainMultiplicationRecursion.java

```
import java.util.*;
class MatrixChainMultiplicationRecursion
{
	// Matrix Pi has dimension P[i-1] x P[i] for i = 1..n
	static int MatrixChainOrder(int p[], int i, int j)
	{
		System.out.println("i " + i + " j " + j);
		if (i == j)
			return 0;

		int min = Integer.MAX_VALUE;

		// Place parenthesis at different places between first and last matrix,
		// recursively calculate count of multiplications for each parenthesis placement
		// and return the minimum count
		for (int k = i; k < j; k++)
		{
			// recur for `M[i+1]…M[k]` to get an `i × k` matrix
			int count = MatrixChainOrder(p, i, k);

			// recur for `M[k+1]…M[j]` to get an `k × j` matrix
			count += MatrixChainOrder(p, k + 1, j);

			// cost to multiply two `i × k` and `k × j` matrix
			count += p[i - 1] * p[k] * p[j];

			System.out.println("i " + i + " k " + k + " j " + j + " min " + min + " count " +
count);
			if (count < min)
				min = count;
		}
		// Return minimum count
		return min;
	}

	public static void main(String args[])
	{
		/*
		5
```

```
                4 5 3 6 2
                */
                Scanner sc=new Scanner(System.in);
                int n=sc.nextInt();
                int arr[] = new int[n];
                for(int i=0;i<n;i++)
                        arr[i]=sc.nextInt();

                System.out.println(MatrixChainOrder(arr, 1, n - 1));
        }
}
```

The time complexity of the above naive recursive approach is exponential. Observe that the above function computes the same subproblems again and again.

Let us saywe are given an array of 5 elements that means we are given N-1 i.e 4 matrixes.
See the following recursion tree for a matrix chain of size 4.



There are so many overlapping subproblems . Let's optimize it using DP !!

when we used the Dynamic programming technique we shall follow some steps.

1. Characterize the structure of an optimal solution.
2. Recursively define the value of an optimal solution.
3. Compute the value of an optimal solution.

4. Construct an optimal solution from computed information.

We have matrices of any of order. our goal is find optimal cost multiplication of matrices.when we solve the this kind of problem using DP step 2 we can get

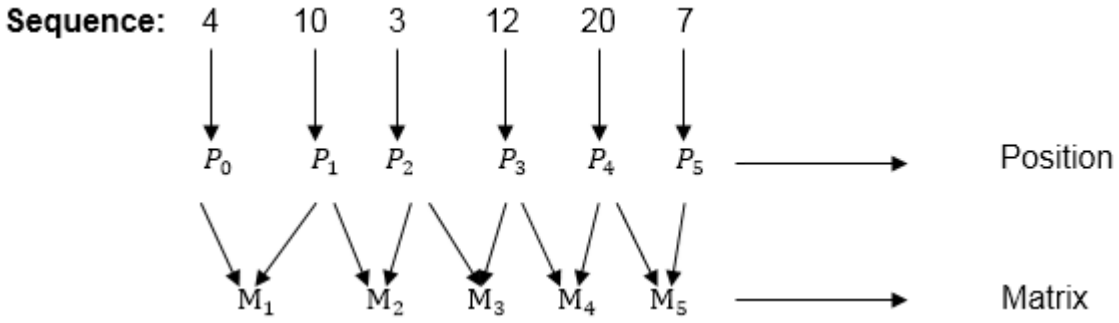
m[i , j] = min { m[i , k] + m[i+k , j] + pi-1*pk*pj } if i < j…. where p is dimension of matrix , i ≤ k < j …..

Example of Matrix Chain Multiplication

Example: We are given the sequence {4, 10, 3, 12, 20, and 7}. The matrices have size 4 x 10, 10 x 3, 3 x 12, 12 x 20, 20 x 7. We need to compute M [i,j], 0 ≤ i, j≤ 5. 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.



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.

Now product of 3 matrices:

M [1, 3] = M1 M2 M3

1. There are two cases by which we can solve this multiplication:

   (M1 x M2) + M3, M1+ (M2x M3)

2. 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

As Comparing both output **264** is minimum in both cases so we insert **264** in table and (M1 x M2) + M3 this combination is chosen for the output making.

M [2, 4] = M2 M3 M4

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

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

M [2, 4] = 1320

As 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

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

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

M [3, 5] = 1140

As 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.

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

→

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

Now Product of 4 matrices:

M [1, 4] = M1 M2 M3 M4

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

1. (M1 x M2 x M3) M4
2. M1 x(M2 x M3 x M4)
3. (M1 xM2) x (M3 x M4)

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

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

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:

1. (M2 x M3 x M4)x M5
2. M2 x(M3 x M4 x M5)
3. (M2 x M3)x (M4 x M5)

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

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

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 M4xM5)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

Now Product of 5 matrices:

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

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

1. (M1 x M2 xM3 x M4)x M5
2. M1 x(M2 xM3 x M4 xM5)
3. (M1 x M2 xM3)x M4 xM5
4. 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

So we can get the optimal solution of matrices multiplication….

Java Program For Chain Matrix Multiplication using memoization:

MatrixChainMultiplicationMemoised.java

The idea is to use **memoization**. Now each time we compute the minimum cost needed to multiply out a specific subsequence, save it. If we are ever asked to compute it again, simply give the saved answer and do not recompute it.

```
import java.util.*;

class MatrixChainMultiplicationMemoised
{
        static int[][] dp;
        // Matrix Pi has dimension p[i-1] x p[i] for i = 1..n
        static int matrixChainMemoised(int p[], int i, int j)
```

```
	{
		System.out.println("i " + i + " j " + j);
		if (i == j)
			return 0;

		if (dp[i][j] != 0)
			return dp[i][j];

		dp[i][j] = Integer.MAX_VALUE;

		for (int k = i; k < j; k++)
		{
			dp[i][j] = Math.min(dp[i][j], matrixChainMemoised(p, i, k) +
matrixChainMemoised(p, k + 1, j) + p[i - 1] * p[k] * p[j]);
		}
		System.out.println(Arrays.deepToString(dp));
		return dp[i][j];
	}

	static int MatrixChainOrder(int[] p, int n)
	{
		dp = new int[n][n];
		return matrixChainMemoised(p, 1, n-1);
	}

	public static void main(String args[])
	{
		/*
		5
		4 5 3 6 2

		6
		4 10 3 12 20 7
		*/
		Scanner sc=new Scanner(System.in);
		int n=sc.nextInt();
		int arr[] = new int[n];
		for(int i=0;i<n;i++)
			arr[i]=sc.nextInt();

		System.out.println(MatrixChainOrder(arr, n));
	}
}
```

Java Program For Chain Matrix Multiplication using bottom-up approach :

MatrixChainMultiplicationDP.java

The following **bottom-up approach** computes, for each `2 <= k <= n`, the minimum costs of all subsequences of length `k`, using the prices of smaller subsequences already computed. It has the same asymptotic runtime and requires no recursion.

```
class MatrixChainMultiplicationDP
{
	// Matrix Pi has dimension p[i-1] x p[i] for i = 1..n
	static int MatrixChainOrder(int p[], int n)
	{
		int dp[][] = new int[n][n];

		int i, j, k, len, cost;

		/* m[i, j] = Minimum number of scalar multiplications needed
		to compute the matrix P[i]P[i+1]...P[j] =P[i..j] where
		dimension of P[i] is p[i-1] x p[i] */

		// len is chain length.
		for (len = 2; len < n; len++)
		{
			for (i = 1; i <= n - len; i++)
			{
				j = i + len - 1;
				if (j == n)
					continue;
				dp[i][j] = Integer.MAX_VALUE;
				for (k = i; k < j; k++)
				{
					cost = dp[i][k] + dp[k + 1][j] + p[i - 1] * p[k] * p[j];
					System.out.println("i " + i + " k " + k + " j " + j + " len "
+ len + " cost " + cost);
					if (cost < dp[i][j])
						dp[i][j] = cost;
				}
			}
		}
		System.out.println(Arrays.deepToString(dp));
		return dp[1][n - 1];
	}
```

```
	public static void main(String args[])
	{
		/*
		6
		4 10 3 12 20 7
		*/
		Scanner sc=new Scanner(System.in);
		int n=sc.nextInt();
		int arr[] = new int[n];
		for(int i=0;i<n;i++)
			arr[i]=sc.nextInt();

		System.out.println(MatrixChainOrder(arr, n));
	}
}
```

6. Number of Corner Rectangles:

Given a grid where each entry is only 0 or 1, find the number of corner rectangles.

A corner rectangle is 4 distinct 1s on the grid that form an axis-aligned rectangle.

Note that only the corners need to have the value 1. Also, all four 1s used must be distinct.

Example 1:

Input:

grid = [[1, 0, 0, 1, 0],
[0, 0, 1, 0, 1],
[0, 0, 0, 1, 0],
[1, 0, 1, 0, 1]]

Output: 1

Explanation: There is only one corner rectangle, with corners grid[1][2], grid[1][4], grid[3][2], grid[3][4].

Example 2:

Input:

grid = [[1, 1, 1],
[1, 1, 1],
[1, 1, 1]]

Output: 9

Explanation: There are four 2x2 rectangles, four 2x3 and 3x2 rectangles, and one 3x3 rectangle.

Example 3: Input: grid = [[1, 1, 1, 1]]

Output: 0

Explanation: Rectangles must have four distinct corners.

Approach:

- As long as we have two corner points of the rows of a rectangle and corresponding two corner points of the columns, it forms a rectangle.
- So we can fix two rows of the input matrix, every time we find two corner points in this row we treat them as the left line of a rectangle, we try to find how many right lines it can have in the previous scanned lines (by checking the columns)

Java Program for Count Number Of Corner Rectangles Using dynamic Programming:

CountCornerRectangles.java

```
import java.util.*;

class CountCornerRectangles {

	public int countCornerRectangles(int[][] grid){
		if (grid == null || grid.length <= 1 || grid[0].length <= 1)
		{
	 return 0;
      }

		int ans = 0;
		int nrows = grid.length, ncols = grid[0].length;
		for (int i = 0; i < nrows; i++){
			for (int j = i + 1; j < nrows; j++){
				for (int k = 0, counter = 0; k < ncols; k++){
					if (grid[i][k] + grid[j][k] == 2){
						ans += counter++;
						System.out.println("i " + i + " j " + j + " k " + k + "
ans " + ans + " counter " + counter);
					}
				}
			}
		}
		return ans;
	}


	public static void main(String args[])
	{
		Scanner sc=new Scanner(System.in);
		int m =sc.nextInt();
		int n=sc.nextInt();
		int grid[][]=new int[m][n];
		for(int i=0;i<m;i++)
			for(int j=0;j<n;j++)
				grid[i][j]=sc.nextInt();
```

```
                System.out.println(new
CountCornerRectangles().countCornerRectangles(grid));
        }
}
```

Sample i/p & o/p:

Example-1
input =3 4
1 0 1 0
1 1 1 1
0 1 1 1
output =4

Example-2
input =4 6
1 0 1 1 1 1
1 0 1 1 0 1
0 1 0 1 1 1
1 1 1 1 0 1
output =25

Example-3
input =5 5
1 1 1 1 1
1 1 1 1 1
1 1 1 1 1
1 1 1 1 1
1 1 1 1 1
output =100

Time Complexity: **$O(N*M^2)$**
Auxiliary Space: **$O(M^2)$**

7. All Pairs Shortest Paths (Floyd Warshall Algorithm):

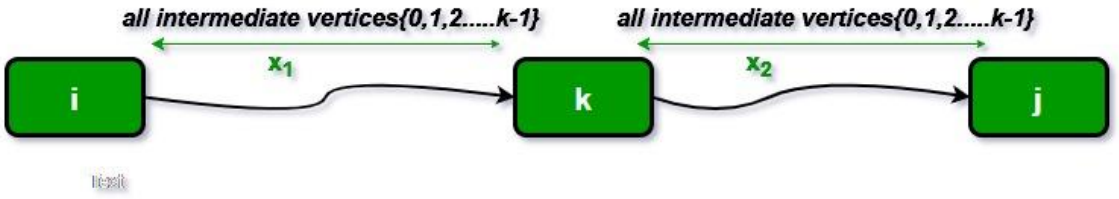
The Floyd Warshall Algorithm is an all pair shortest path algorithm, works for both the directed and undirected weighted graphs. But, it does not work for the graphs with negative cycles (where the sum of the edges in a cycle is negative).
It follows Dynamic Programming approach to check every possible path going via every possible node in order to calculate shortest distance between every pair of nodes.

Idea Behind Floyd Warshall Algorithm:

Let a graph G[][] with V vertices from 1 to N. Now we have to evaluate a shortestPathMatrix[][] where shortestPathMatrix[i][j] represents the shortest path between vertices i and j.

Obviously the shortest path between i to j will have some k number of intermediate nodes. The idea behind floyd warshall algorithm is to treat each and every vertex from 1 to N as an intermediate node one by one.

The following figure shows the above optimal substructure property in floyd warshall algorithm:



Floyd Warshall Algorithm:

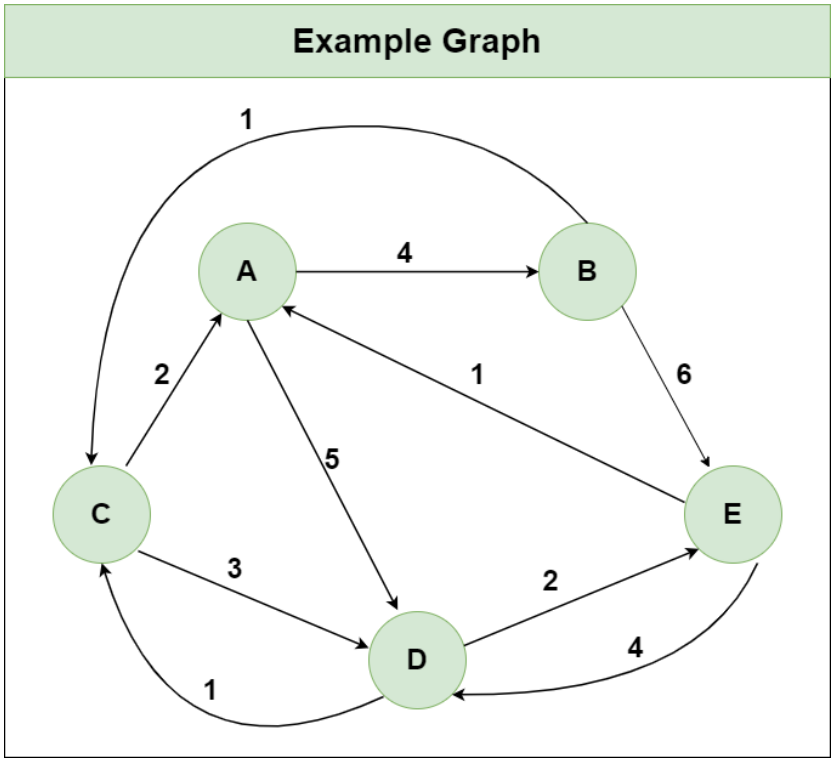
1. Initialize the solution matrix same as the input graph matrix as a first step.
2. Then update the solution matrix by considering all vertices as an intermediate vertex.
3. The idea is to pick all vertices one by one and updates all shortest paths which include the picked vertex as an intermediate vertex in the shortest path.
4. When we pick vertex number k as an intermediate vertex, we already have considered vertices {0, 1, 2, .. k-1} as intermediate vertices.
5. For every pair (i, j) of the source and destination vertices respectively, there are two possible cases.
    a. k is not an intermediate vertex in shortest path from i to j. We keep the value of dist[i][j] as it is.
    b. k is an intermediate vertex in shortest path from i to j. We update the value of dist[i][j] as dist[i][k] + dist[k][j], if dist[i][j] > dist[i][k] + dist[k][j]

Pseudo-Code of Floyd Warshall Algorithm :

```
0   Algorithm AllPaths(cost, A, n)
1   // cost[1 : n, 1 : n] is the cost adjacency matrix of a graph with
2   // n vertices; A[i, j] is the cost of a shortest path from vertex
3   // i to vertex j. cost[i, i] = 0.0, for 1 ≤ i ≤ n.
4   {
5       for i := 1 to n do
6           for j := 1 to n do
7               A[i, j] := cost[i, j]; // Copy cost into A.
8       for k := 1 to n do
9           for i := 1 to n do
10              for j := 1 to n do
11                  A[i, j] := min(A[i, j], A[i, k] + A[k, j]);
12  }
```

Example :

Suppose we have a graph as shown in the image:



Step 1: *Initialize the Distance[][] matrix using the input graph such that Distance[i][j]= weight of edge from* ***i*** *to* ***j****, also Distance[i][j] = Infinity if there is no edge from* ***i*** *to* ***j.***

Step1: Initializing Distance[ ][ ] using the Input Graph

	A	B	C	D	E
A	0	4	∞	5	∞
B	∞	0	1	∞	6
C	2	∞	0	3	∞
D	∞	∞	1	0	2
E	1	∞	∞	4	0

Step 2: *Treat node **A** as an intermediate node and calculate the Distance[][] for every {i,j} node pair using the formula:*
*= Distance[i][j] = minimum (Distance[i][j], (Distance from i to **A**) + (Distance from **A** to j))*
*= Distance[i][j] = minimum (Distance[i][j], Distance[i][**A**] + Distance[**A**][j])*

Step 2: Using Node A as the Intermediate node

Distance[i][j] = min (Distance[i][j], Distance[i][A] + Distance[A][j])

	A	B	C	D	E
A	0	4	∞	5	∞
B	∞	?	?	?	?
C	2	?	?	?	?
D	∞	?	?	?	?
E	1	?	?	?	?

→

	A	B	C	D	E
A	0	4	∞	5	∞
B	∞	0	1	∞	6
C	2	6	0	3	12
D	∞	∞	1	0	2
E	1	5	∞	4	0

Step 3: Treat node **B** as an intermediate node and calculate the Distance[][] for every {i,j} node pair using the formula:
= Distance[i][j] = minimum (Distance[i][j], (Distance from i to **B**) + (Distance from **B** to j))
= Distance[i][j] = minimum (Distance[i][j], Distance[i][**B**] + Distance[**B**][j])

Step 3: Using Node B as the Intermediate node

Distance[i][j] = min (Distance[i][j], Distance[i][B] + Distance[B][j])

	A	B	C	D	E
A	?	4	?	?	?
B	∞	0	1	∞	6
C	?	6	?	?	?
D	?	∞	?	?	?
E	?	5	?	?	?

	A	B	C	D	E
A	0	4	5	5	10
B	∞	0	1	∞	6
C	2	6	0	3	12
D	∞	∞	1	0	2
E	1	5	6	4	0

Step 4: Treat node **C** as an intermediate node and calculate the Distance[][] for every {i,j} node pair using the formula:
= Distance[i][j] = minimum (Distance[i][j], (Distance from i to **C**) + (Distance from **C** to j))
= Distance[i][j] = minimum (Distance[i][j], Distance[i][**C**] + Distance[**C**][j])

Step 4: Using Node C as the Intermediate node

Distance[i][j] = min (Distance[i][j], Distance[i][C] + Distance[C][j])

	A	B	C	D	E
A	?	?	5	?	?
B	?	?	1	?	?
C	2	6	0	3	12
D	?	?	1	?	?
E	?	?	6	?	?

	A	B	C	D	E
A	0	4	5	5	10
B	3	0	1	4	6
C	2	6	0	3	12
D	3	7	1	0	2
E	1	5	6	4	0

Step 5: Treat node **D** as an intermediate node and calculate the Distance[][] for every {i,j} node pair using the formula:
= Distance[i][j] = minimum (Distance[i][j], (Distance from i to **D**) + (Distance from **D** to j))
= Distance[i][j] = minimum (Distance[i][j], Distance[i][**D**] + Distance[**D**][j])

Step 5: Using Node D as the Intermediate node

Distance[i][j] = min (Distance[i][j], Distance[i][D] + Distance[D][j])

	A	B	C	D	E
A	?	?	?	5	?
B	?	?	?	4	?
C	?	?	?	3	?
D	3	7	1	0	2
E	?	?	?	4	?

	A	B	C	D	E
A	0	4	5	5	7
B	3	0	1	4	6
C	2	6	0	3	5
D	3	7	1	0	2
E	1	5	5	4	0

Step 6: Treat node **E** as an intermediate node and calculate the Distance[][] for every {i,j} node pair using the formula:
= Distance[i][j] = minimum (Distance[i][j], (Distance from i to **E**) + (Distance from **E** to j))
= Distance[i][j] = minimum (Distance[i][j], Distance[i][**E**] + Distance[**E**][j])

Step 6: Using Node E as the Intermediate node

Distance[i][j] = min (Distance[i][j], Distance[i][E] + Distance[E][j])

	A	B	C	D	E
A	?	?	?	?	7
B	?	?	?	?	6
C	?	?	?	?	5
D	?	?	?	?	2
E	1	5	5	4	0

	A	B	C	D	E
A	0	4	5	5	7
B	3	0	1	4	6
C	2	6	0	3	5
D	3	7	1	0	2
E	1	5	5	4	0

Step 7: Since all the nodes have been treated as an intermediate node, we can now return the updated Distance[][] matrix as our answer matrix.

Step 7: Return Distance[ ][ ] matrix as the result

	A	B	C	D	E
A	0	4	5	5	7
B	3	0	1	4	6
C	2	6	0	3	5
D	3	7	1	0	2
E	1	5	5	4	0

Java Program for All Pairs Shortest Path: AllPiarsShortestpath.java

```
import java.util.*;
import java.lang.*;
import java.io.*;

class AllPairShortestPath
{
        static int V;
        void floydWarshall(int graph[][])
        {
                int dist[][] = new int[V][V];
                int i, j, k;

                for (i = 0; i < V; i++)
                        for (j = 0; j < V; j++)
                                dist[i][j] = graph[i][j];
```

```
		for (k = 0; k < V; k++)		// Intermediate vertices
		{
			// Pick all vertices as source one by one
			for (i = 0; i < V; i++)
			{
				// Pick all vertices as destination for the above picked source
				for (j = 0; j < V; j++)
				{
					// If vertex k is on the shortest path from i to j, then update
the value of dist[i][j]
					if (dist[i][j] > dist[i][k] + dist[k][j])
					{
						dist[i][j] = dist[i][k] + dist[k][j];
					}
				}
			}
		}
		printSolution(dist);
	}

	void printSolution(int dist[][])
	{
		for (int i=0; i<V; ++i)
		{
			for (int j=0; j<V; ++j)
			{
				if (dist[i][j]==99)
					System.out.print("INF ");
				else
					System.out.print(dist[i][j]+" ");
			}
			System.out.println();
		}
	}

	public static void main (String[] args)
	{
		Scanner sc = new Scanner(System.in);
		V = sc.nextInt();
```

```
                int graph[][] = new int[V][V];
                for(int i = 0; i < V; i++)
                        for(int j = 0;j < V; j++)
                                graph[i][j] = sc.nextInt();

                AllPairShortestPath a = new AllPairShortestPath();

                a.floydWarshall(graph);
        }
}
```

Input:

```
4
0 3 99 7
8 0 2 99
5 99 0 1
2 99 99 0
```

Output:

```
0 3 5 6
5 0 2 3
3 6 0 1
2 5 7 0
```

Complexity analysis of All Pairs Shortest Path Algorithm:

It is very simple to derive the complexity of all pairs' shortest path problem from the above algorithm. It uses three nested loops. The innermost loop has only one statement. The complexity of that statement is Q(1).

The running time of the algorithm is computed as :

$$T(n) = \sum_{k=1}^{n} \sum_{i=1}^{n} \sum_{j=1}^{n} \Theta(1) = \sum_{k=1}^{n} \sum_{i=1}^{n} n = \sum_{k=1}^{n} n^2 = O(n^3)$$

Example

Problem: Apply Floyd's method to find the shortest path for the below-mentioned all pairs.

	1	2	3	4
1	0	∞	3	∞
2	2	0	∞	∞
3	∞	7	0	1
4	6	∞	∞	0

Solution:

Optimal substructure formula for Floyd's algorithm,

$$D^k [i, j] = \min \{ D^{k-1} [i, j], D^{k-1} [i, k] + D^{k-1} [k, j] \}$$

$D^o =$

	1	2	3	4
1	0	∞	3	∞
2	2	0	∞	∞
3	∞	7	0	1
4	6	∞	∞	0

Iteration 1 : k = 1 :

$D^1[1, 2] = \min \{ D^o [1, 2], D^o [1, 1] + D^o [1, 2] \}$

$= \min \{\infty, 0 + \infty\} = \infty$

$D^1[1, 3] = \min \{ D^o [1, 3], D^o [1, 1] + D^o [1, 3] \}$

$= \min \{3, 0 + 3\} = 3$

$D^1[1, 4] = \min \{ D^o [1, 4], D^o [1, 1] + D^o [1, 4] \}$
$= \min \{\infty, 0 + \infty\} = \infty$

$D^1[2, 1] = \min \{ D^o [2, 1], D^o [2, 1] + D^o [1, 1] \}$
$= \min \{2, 2 + 0\} = 2$

$D^1[2, 3] = \min \{ D^o [2, 3], D^o [2, 1] + D^o [1, 3] \}$
$= \min \{\infty, 2 + 3\} = 5$

$D^1[2, 4] = \min \{ D^o [2, 4], D^o [2, 1] + D^o [1, 4] \}$
$= \min \{\infty, 2 + \infty\} = \infty$

$D^1[3, 1] = \min \{ D^o [3, 1], D^o [3, 1] + D^o [1, 1] \}$
$= \min \{\infty, 0 + \infty\} = \infty$

$D^1[3, 2] = \min \{ D^o [3, 2], D^o [3, 1] + D^o [1, 2] \}$
$= \min \{7, \infty + \infty\} = 7$

$D^1[3, 4] = \min \{ D^o [3, 4], D^o [3, 1] + D^o [1, 4] \}$
$= \min \{1, \infty + \infty\} = 1$

$D^1[4, 1] = \min \{ D^o [4, 1], D^o [4, 1] + D^o [1, 1] \}$
$= \min \{6, 6 + 0\} = 6$

$D^1[4, 2] = \min \{ D^o [4, 2], D^o [4, 1] + D^o [1, 2] \}$
$= \min \{\infty, 6 + \infty\} = \infty$

$D^1[4, 3] = \min \{ D^o [4, 3], D^o [4, 1] + D^o [1, 3] \}$
$= \min \{\infty, 6 + 3\} = 9$

$D^1 =$

	1	2	3	4
1	0	∞	3	∞
2	2	0	5	∞
3	∞	7	0	1
4	6	∞	9	0

Note : Path distance for highlighted cell is improvement over original matrix.

Iteration 2 (k = 2) :

$D^2[1, 2] = D^1 [1, 2] = \infty$

$D^2[1, 3] = \min \{ D^1 [1, 3], D^1 [1, 2] + D^1 [2, 3] \}$
$= \min \{3, \infty + 5\} = 3$

$D^2[1, 4] = \min \{ D^1 [1, 4], D^1 [1, 2] + D^1 [2, 4] \}$
$= \min \{\infty, \infty + \infty\} = \infty$

$D^2[2, 1] = D^1 [2, 1] = 2$
$D^2[2, 3] = D^1 [2, 3] = 5$
$D^2[2, 4] = D^1 [2, 4] = \infty$
$D^2[3, 1] = \min \{ D^1 [3, 1], D^1 [3, 2] + D^1 [2, 1] \}$
$= \min \{\infty, 7 + 2\} = 9$

$D^2[3, 2] = D^1 [3, 2] = 7$
$D^2[3, 4] = \min \{ D^1 [3, 4], D^1 [3, 2] + D^1 [2, 4] \}$
$= \min \{1, 7 + \infty\} = 1$

$D^2[4, 1] = \min \{ D^1 [4, 1], D^1 [4, 2] + D^1 [2, 1] \}$

= min {6, ∞ + 2} = 6

$D^2[4, 2] = D^1[4, 2] = \infty$
$D^2[4, 3] = \min \{ D^1[4, 3], D^1[4, 2] + D^1[2, 3] \}$
= min {9, ∞ + 5} = 9

$D^2 =$

	1	2	3	4
1	0	∞	3	∞
2	2	0	5	∞
3	9	7	0	1
4	6	∞	9	0

Iteration 3 (k = 3) :

$D^3[1, 2] = \min \{ D^2[1, 2], D^2[1, 3] + D^2[3, 2] \}$
= min {∞, 3 + 7} = 10

$D^3[1, 3] = D^2[1, 3] = 3$

$D^3[1, 4] = \min \{ D^2[1, 4], D^2[1, 3] + D^2[3, 4] \}$
= min {∞, 3 + 1} = 4

$D^3[2, 1] = \min \{ D^2[2, 1], D^2[2, 3] + D^2[3, 1] \}$
= min {2, 5 + 9} = 2

$D^3[2, 3] = D^2[2, 3] = 5$

$D^3[2, 4] = \min \{ D^2[2, 4], D^2[2, 3] + D^2[3, 4] \}$
= min {∞, 5 + 1} = 6

$D^3[3, 1] = D^2[3, 1] = 9$

$D^3[3, 2] = D^2[3, 2] = 7$

$D^3[3, 4] = D^2[3, 4] = 1$

$D^3[4, 1] = \min \{ D^2[4, 1], D^2[4, 3] + D^2[3, 1] \}$
$= \min \{6, 9 + 9\} = 6$

$D^3[4, 2] = \min \{ D^2[4, 1], D^2[4, 3] + D^2[3, 2] \}$
$= \min \{\infty, 9 + 7\} = 16$

$D^3[4, 3] = D^2[4, 3] = 9$

$D^3 =$

	1	2	3	4
1	0	10	3	4
2	2	0	5	6
3	9	7	0	1
4	6	16	9	0

Iteration 4 (k = 4) :

$D^4[1, 2] = \min \{ D^3[1, 2], D^3[1, 4] + D^3[4, 2] \}$
$= \min \{10, 4 + 16\} = 10$

$D^4[1, 3] = \min \{ D^3[1, 3], D^3[1, 4] + D^3[4, 1] \}$
$= \min \{3, 4 + 9\} = 3$

$D^4[1, 4] = D^3[1, 4] = 4$
$D^4[2, 1] = \min \{ D^3[2, 1], D^3[2, 4] + D^3[4, 1] \}$
$= \min \{2, 6 + 6\} = 2$

$D^4[2, 3] = \min \{ D^3[2, 3], D^3[2, 4] + D^3[4, 3] \}$
$= \min \{5, 6 + 9\} = 5$

$D^4[2, 4] = D^3[2, 4] = 6$
$D^4[3, 1] = \min \{ D^3[3, 1], D^3[3, 4] + D^3[4, 1] \}$

= min {9, 1 + 6} = 7

$D^4[3, 2] = \min \{ D^3[3, 2], D^3[3, 4] + D^3[4, 2] \}$
$= \min \{7, 1 + 16\} = 7$

$D^4[3, 4] = D^3[3, 4] = 1$

$D^4[4, 1] = D^3[4, 1] = 6$

$D^4[4, 2] = D^3[4, 2] = 16$

$D^4[4, 3] = D^3[4, 3] = 9$

Final distance matrix is,

$$D^4 = \begin{bmatrix} 0 & 10 & 3 & 4 \\ 2 & 0 & 5 & 6 \\ 7 & 7 & 0 & 1 \\ 6 & 16 & 9 & 0 \end{bmatrix}$$

8. Optimal Binary Search Tree

- Optimal Binary Search Tree extends the concept of Binary search tree.
- Binary Search Tree (BST) is a *nonlinear* data structure which is used in many scientific applications for reducing the search time.
- In BST, left child is smaller than root and right child is greater than root. This arrangement simplifies the search procedure.
- Optimal Binary Search Tree (OBST) is very useful in dictionary search.
- The probability of searching is different for different words. OBST has great application in translation.
- A Binary Search Tree (BST) is a tree where the key values are stored in the internal nodes.
- The external nodes are null nodes.
- The keys are ordered lexicographically, i.e. for each internal node all the keys in the left sub-tree are less than the keys in the node, and all the keys in the right sub-tree are greater.
- When we know the frequency of searching each one of the keys, it is quite easy to computethe expected cost of accessing each node in the tree.
- An optimal binary search tree is a BST, which has minimal expected cost of locating each node.
- Search time of an element in a BST is *O(n)*, whereas in a Balanced-BST search timeis *O(log n)*.
- Again, the search time can be improved in Optimal Cost Binary Search Tree, placing the most frequently used data in the root and closer to the root element, while placing the least frequently used data near leaves and in leaves.

Note: The keys[] is assumed to be sorted in increasing order. If keys[] is not sorted, then add code to sort keys, and rearrange freq[] accordingly.

OBST using Recursive approach:

```
import java.util.*;

/*The optimal cost for freq[i..j] can be recursively calculated using the following formula.
optcost(i, j) = sum{k=i to j} freq[k]+ min{r=i to j} optcost(i, r-1) + optcost(r+1, j)

We need to calculate optCost(0, n-1) to find the result.

The idea of above formula is simple, we one by one try all nodes as root (r varies from i to j in
second term).

When we make rth node as root, we recursively calculate optimal cost from i to r-1 and r+1 to j.
We add sum of frequencies from i to j (see first term in the above formula)


This can be divided into 2 parts one is the freq[r]+sum of frequencies of all elements from i to j
except r.

The term freq[r] is added because it is going to be root and that means level of 1.

So, freq[r]*1=freq[r]. Now the actual part comes, we are adding the frequencies of remaining
elements because as we take r as root then all the elements other than that are going 1 level down
than that is calculated in the subproblem.


For calculating optcost(i,j) we assume that the r is taken as root and calculate min of
opt(i,r-1)+opt(r+1,j) for all i<=r<=j.


Here for every subproblem we are  choosing one node as a root.

But in reality the level of subproblem root and all its descendant nodes will be 1 greater than the
level of the parent problem root.


Therefore the frequency of all the nodes except r should be added which accounts to the descend
in their level compared to level assumed in subproblem.


Time Complexity: Exponential


*/

// A naive recursive implementation of optimal binary search tree problem

public class OBSTRecursive
{
        // A recursive function to calculate cost of optimal binary search tree
        static int optCost(int freq[], int i, int j)
        {
```

```
        // Base cases
        if (j < i)          // no elements in this subarray
                return 0;
        if (j == i)         // one element in this subarray
                return freq[i];

        // Get sum of freq[i], freq[i+1], ... freq[j]
        int sum = fsum(freq, i, j);

        // Initialize minimum value
        int min = Integer.MAX_VALUE;

        for (int r = i; r <= j; ++r)
        {
                min = Math.min(optCost(freq, i, r-1) + optCost(freq, r+1, j), min);
        }

        // Return minimum value
        return min + sum;
}

static int optimalSearchTree(int keys[], int freq[], int n)
{
        return optCost(freq, 0, n-1);
}

// A utility function to get sum of array elements freq[i] to freq[j]
static int fsum(int freq[], int i, int j)
{
        int s = 0;
        for (int k = i; k <=j; k++)
                s += freq[k];

        return s;
}

public static void main(String[] args)
{
        Scanner sc = new Scanner(System.in);
        int n = sc.nextInt();
        int keys[] = new int[n];
        int freq[] = new int[n];
        for(int i = 0; i < n; i++)
```

```
                keys[i] = sc.nextInt();

            for(int i = 0; i < n; i++)
                freq[i] = sc.nextInt();

            System.out.println("Cost of Optimal BST is " + optimalSearchTree(keys, freq, n));
        }
    }
```

Input Format:

Line-1: An integer N, number of nodes.
Line-2: N space separate integers, node vals[].
Line-3: N space separate integers, node cost[].

Output Format:

Print an integer, minimum search cost of BST.

Sample Input-1:

```
3
10 20 30
23 32 14
```

Sample Output-1:

```
106
```

Explanation:

The BST is:

```
        20
        /\
      10  30
```

Sample Input-2:

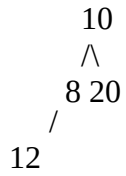
```
4
8 10 12 20
26 34 8 50
```

Sample Output-2:

```
210
```

Explanation:

The BST is:



OBST using Memoization approach:

```
import java.util.*;
/* The optimal cost for freq[i..j] can be recursively calculated using the following formula.
optcost(i, j) = sum{k=i to j} freq[k]+ min{r=i to j} optcost(i, r-1) + optcost(r+1, j)

We need to calculate optCost(0, n-1) to find the result.

The idea of above formula is simple, we one by one try all nodes as root (r varies from i to j in
second term).

When we make rth node as root, we recursively calculate optimal cost from i to r-1 and r+1 to j.
We add sum of frequencies from i to j (see first term in the above formula)

This can be divided into 2 parts one is the freq[r]+sum of frequencies of all elements
from i to j except r.

The term freq[r] is added because it is going to be root and that means level of 1,

So, freq[r]*1=freq[r]. Now the actual part comes, we are adding the frequencies of remaining
elements because as we take r as root then all the elements other than that are going 1 level down
than that is calculated in the subproblem.

For calculating optcost(i,j) we assume that the r is taken as root and calculate min of
opt(i,r-1)+opt(r+1,j) for all i<=r<=j.

Here for every subproblem we are  choosing one node as a root.

But in reality, the level of subproblem root and all its descendant nodes will be 1 greater than
thelevel of the parent problem root.

Therefore, the frequency of all the nodes except r should be added which accounts to the
descendin their level compared to level assumed in subproblem.

*/


public class OBSTMemoised
{
        static int[][] dp;
        // A recursive function to calculate cost of optimal binary search tree

        static int optCost(int freq[], int i, int j)
        {
                // Base cases

                if (j < i)          // no elements in this subarray
```

```
                return 0;
        if (j == i)         // one element in this subarray
                return freq[i];

        if(dp[i][j] != 0)
        {
                 return dp[i][j];
        }

        // Get sum of freq[i], freq[i+1], ... freq[j]

        int sum = fsum(freq, i, j);

        // Initialize minimum value

        int min = Integer.MAX_VALUE;

        for (int r = i; r <= j; ++r)
        {
                min = Math.min(optCost(freq, i, r-1) + optCost(freq, r+1, j), min);
        }

        // Return minimum value

        return dp[i][j] = min + sum;
}

static int optimalSearchTree(int keys[], int freq[], int n)
{
        dp = new int[n][n];
        return optCost(freq, 0, n-1);
}

// A utility function to get sum of array elements freq[i] to freq[j]

static int fsum(int freq[], int i, int j)
{
        int s = 0;
        for (int k = i; k <=j; k++)
                s += freq[k];
        return s;
}

public static void main(String[] args)
{
        Scanner sc = new Scanner(System.in);
        int n = sc.nextInt();
        int keys[] = new int[n];
        int freq[] = new int[n];
        for(int i = 0; i < n; i++)
```

```
                keys[i] = sc.nextInt();

            for(int i = 0; i < n; i++)
                freq[i] = sc.nextInt();

            System.out.println("Cost of Optimal BST is " + optimalSearchTree(keys, freq, n));
        }
}
```

Program for OBST using Tabulation approach:

```
import java.util.*;
import java.util.*;

/*

We use an auxiliary array cost[n][n] to store the solutions of subproblems.
cost[0][n-1] will hold the final result. All diagonal values must be filled first,
then the values which lie on the line just above the diagonal.

In other words, we must first fill all cost[i][i] values,
then all cost[i][i+1] values, then all cost[i][i+2] values.


Time complexity: n^3
Space complexity: n^2

*/


class OBSTDP
{
        /*A Dynamic Programming based function that calculates minimum cost of a Binary
        Search Tree. */

        int minSearchCostBST(int keys[], int freq[], int n)
        {
            /* Create an auxiliary 2D matrix to store results of subproblems */

            int dp[][] = new int[n][n];

            /* dp[i][j] = Optimal cost of binary search tree that can be formed from keys[i]
            to keys[j]. dp[0][n-1] will store the resultant cost */


            // For a single key, dp is equal to freq/frequency of the key

            for (int i = 0; i < n; i++)
                dp[i][i] = freq[i];
```

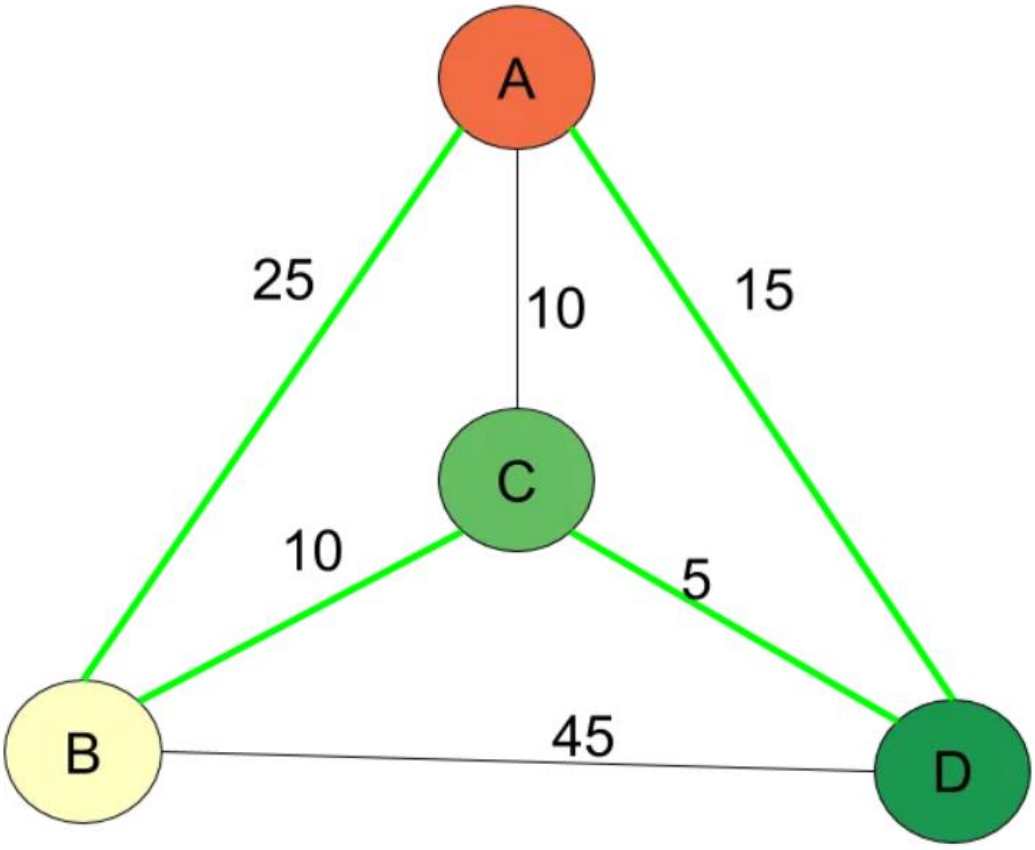
```
        /* Now we need to consider chains of length 2, 3, ...
         len is chain length.*/

        for (int len = 2; len <= n; len++)
        {
                // i is row number in cost[][]

                for (int i = 0; i <= n - len; i++)
                {
                        // Get column number j from row number i and chain length len

                        int j = i + len - 1;
                        dp[i][j] = Integer.MAX_VALUE;
                        int sum = fsum(freq, i, j);

                        // Try making all keys in interval keys[i..j] as root

                        for (int r = i; r <= j; r++)
                        {
                                /* Formula to calculate the minimum cost where r is
                                rootconsidered

                                c[i,j] = min{c[i, r-1] + c[r+1,j]} + w(i,j) where i << r < j*/


                                int c = sum;
                                if(r > i) c += dp[i][r-1];
                                        if(r < j) c += dp[r+1][j];
                                                if(c < dp[i][j])
                                                        dp[i][j]=c;
                        }
                }
        }
        return dp[0][n - 1];
}

// A utility function to get sum of array elements freq[i] to freq[j]

int fsum(int freq[], int i, int j)
{
        int s = 0;
        for (int k = i; k <= j; k++)
        {
                if (k >= freq.length)
                        continue;
                s += freq[k];
        }
return s;
}
```

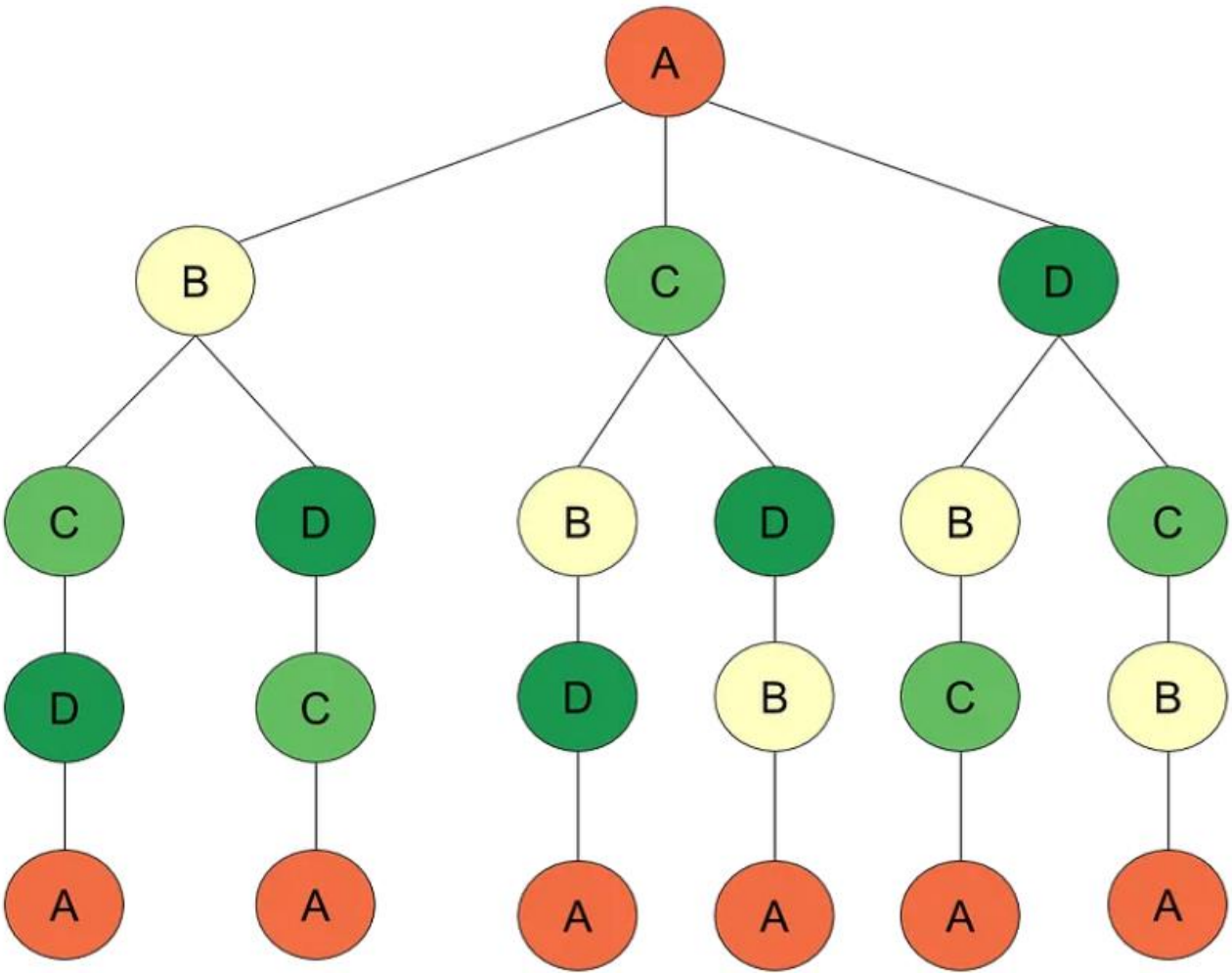
```
	public static void main(String[] args)
	{
		Scanner sc = new Scanner(System.in);
		int n = sc.nextInt();
		int keys[] = new int[n];
		int freq[] = new int[n];
		for(int i = 0; i < n; i++)
			keys[i] = sc.nextInt();
		for(int i = 0; i < n; i++)
			freq[i] = sc.nextInt();

		System.out.println(new OBSTDP().minSearchCostBST(keys, freq, n));
	}
}
```

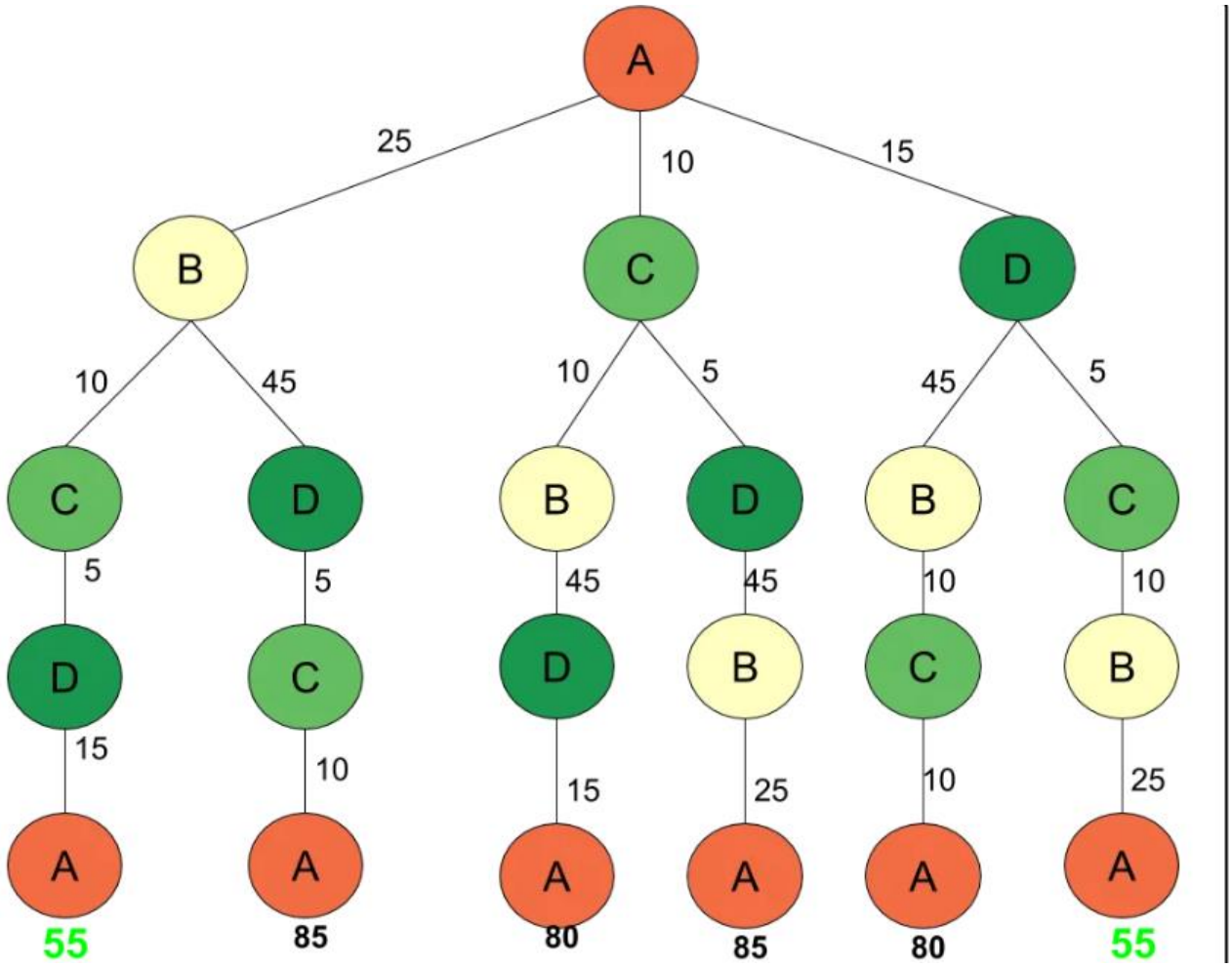
9.Travelling Salesperson Problem

Problem Statement:

We will be given a graph that will be represented in the form of an adjacency matrix, say distance, that denotes the distance between 2 cities, say, i and j. The distance[i][j] denotes the distance between 2 cities if the value of dist[i][j] == 0; this implies the given cities are not connected i.e. no direct edge exists between them. We can start from any node and we need to return to the original city through the shortest distance covering all the cities



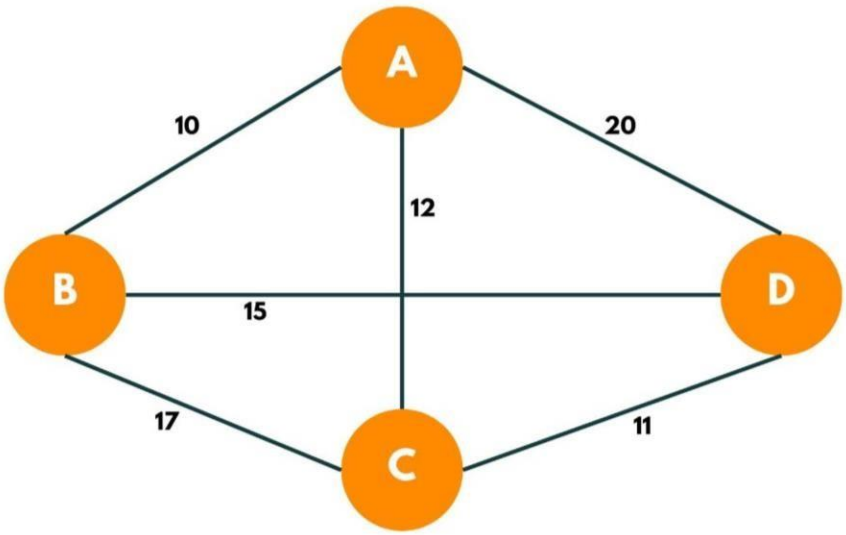

A
B
C
D
C
D
B
D
B
C
D
C
D
B
C
B
A
A
A
A
A
A



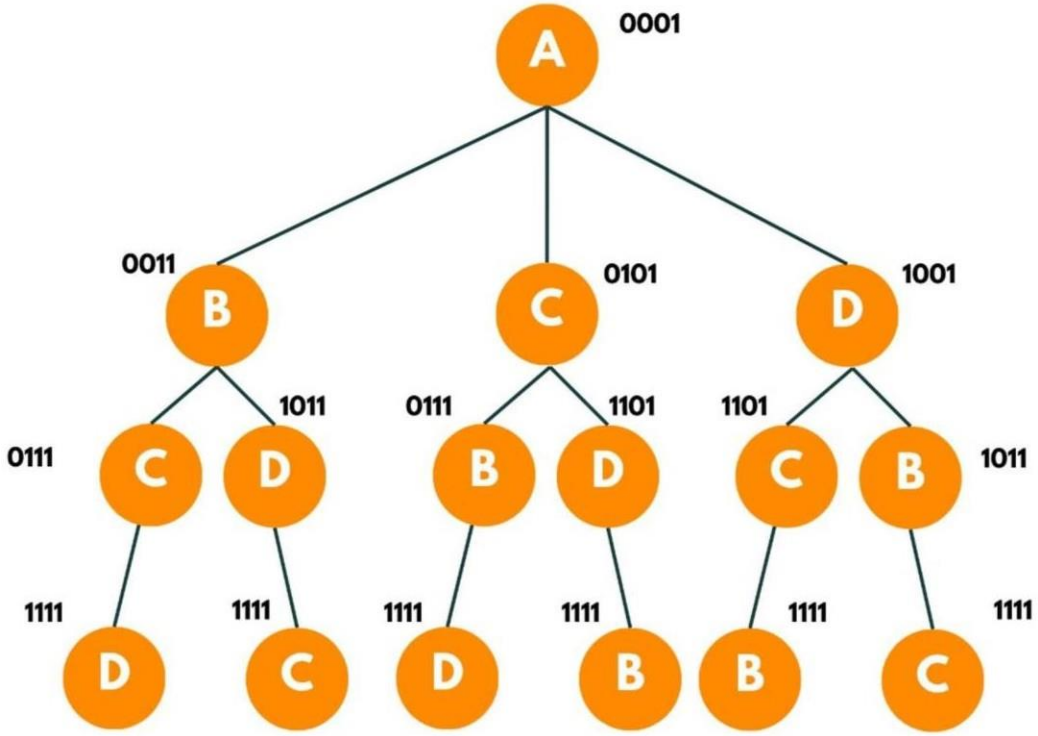
Here is the **algorithm** for Travelling Salesman Problem:

1. Define the mask as (1<<n)-1.
2. Create a function, say, tsp() having mask and city as arguments. As the mask denotes a set of cities visited so far, we iterate over the mask and get to know which city isn't visited.
3. The base case is when we visit all the cities i.e. mask has all of its bits set, we return the distance between the current city and the original city.
4. Then we try to visit the unvisited cities, we iterate over n cities and we check if the nth bit in the mask is 1 or 0.
    a. If the city is not visited, we make recursive calls by adding the distance of the original city and the city not visited and try to get the distance remaining from recursive calls
    b. In the recursive call, we pass a mask with the current city visited and the current city.

Given the following graph and the distance between cities he has traveled.



The recursion tree would look like this:



Program for TSP using Dynamic Programming approach:

```
import java.io.*;
import java.util.*;
public class TSPDP
{
        static int n;
        static int[][] distance;
        static int allCityVisited;
        static int [][]dp;

        /*To avoid overlapping subproblems i.e. avoiding a state which has already been  computed,
        we check dp[mask][currCity]. If this comes out to be -1, implies the city hasn't been visited,
        else the city has already been visited and we return dp[mask][currCity] = ans. */

        static int tsp(int mask, int currCity)
        {
                // If all cities are visited, return the distance from last city to first one

                if(mask == allCityVisited)
                {
                        return distance[currCity][0];
                }

                // If we arrive at a city which is already been computed

                if(dp[mask][currCity] != -1)
                {
                        return dp[mask][currCity];
                }

                // Here we will try to go to every other city to take the minimum answer

                int minDis = Integer.MAX_VALUE;

                // Visiting the unvisited cities

                for(int city=0; city < n; city++)
                {
                        // Checking if city has been visited or not, by checking ith bit in mask

                        if((mask & (1<<city)) ==0 )
                        {
                                // Changing the city as visited (visited nask)

                                int vmask = mask | (1 << city);
```

```
                    /* Storing the distance of current city to the city and then
                    fetchingremaining distance from recursive call. */
                    int dist = distance[currCity][city] + tsp(vmask, city);
                    // Storing the shortest distance
                    minDis = Math.min(minDis, dist);
                }
            }
            return dp[mask][currCity] = minDis;
        }

        // Initially we make a 2D array of [2^n][n] and initially put -1 at each position
        static int getMinPath_TSP()
        {
            dp = new int[1<<n][n];
            allCityVisited = (1<<n)-1;

            for(int i = 0; i < (1<<n); i++)
            {
                for(int j = 0; j < n; j++)
                {
                    dp[i][j] = -1;
                }
            }

            return tsp(1, 0);
        }

        public static void main(String[] args)
        {
            Scanner sc=new Scanner(System.in);
            n = sc.nextInt();
            distance = new int[n][n];
            for(int i = 0; i < n;i++)
                for(int j = 0;j < n;j++)
                    distance[i][j] = sc.nextInt();

            int minPath = getMinPath_TSP();
            System.out.println(Arrays.deepToString(dp));
            System.out.println(minPath);
        }
    }
```

Input Format:

Line-1: An integer N.
Next N lines: N space separated integers, cost of i to j.

Output Format:

Print an integer, the minimum cost to travel all the places and return to place-0.

Sample Input-1:

4
0 10 9 15
13 0 14 9
11 14 0 5
12 7 3 0

Sample Output-1:

33

Sample Input-2:

5
0 15 5 12 15
15 0 7 13 9
5 7 0 10 15
12 13 10 0 7
15 9 15 7 0

Sample Output-2:

40

PART-2 Strings: Introduction, Count Substrings with Only One Distinct Letter, Valid Word Abbreviation, Longest Repeating Substring, Longest Common Subsequence, Longest Increasing Subsequence.

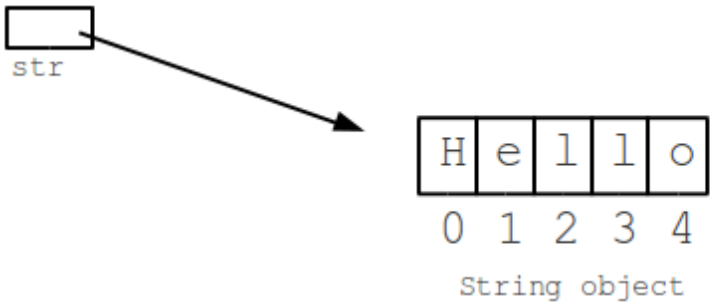
Introduction:

Strings are an incredibly common type of data in computers. This page introduces the basics of Java strings: chars, +, length(), and substring().

A Java string is a series of characters gathered together, like the word "Hello", or the phrase "practice makes perfect". Create a string in the code by writing its chars out between double quotes.

- String stores text -- a word, an email, a book
- All computer languages have strings, look similar
- "In double quotes"
- Sequence of characters ("char")
- String str = "Hello";

This picture shows the string object in memory, made up of the individual chars H e l l o. We'll see what the index numbers 0, 1, 2 .. mean later on.



String + Concatenation

The + (plus) operator between strings puts them together to make a new, bigger string. The bigger string is just the chars of the first string put together with the chars of the second string.

String a = "kit" + "ten"; // a is "kitten"

Strings are not just made of the letters a-z. Chars can be punctuation and other miscellaneous chars. For example in the string "hi ", the 3rd char is a space. This all works with strings stored in variables too, like this:

String fruit = "apple";
String stars = "***";

String a = fruit + stars; // a is "apple*

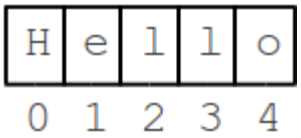
String Length

The "length" of a string is just the number of chars in it. So "hi" is length 2 and "Hello" is length 5. The length() method on a string returns its length, like this:

String a = "Hello";
int len = a.length(); // len is 5

String Index Numbers

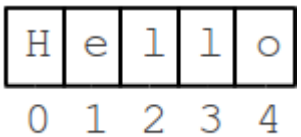
- Index numbers -- 0, 1, 2, ...
- Leftmost char is at index 0
- Last char is at index length-1



The chars in a string are identified by "index" numbers. In "Hello" the leftmost char (H) is at index 0, the next char (e) is at index 1, and so on. The index of the last char is always one less than the length. In this case the length is 5 and 'o' is at index 4. Put another way, the chars in a string are at indexes 0, 1, 2, .. up through length-1. We'll use the index numbers to slice and dice strings with substring() in the next section.

String Substring v1

- str.substring(start)
- Chars beginning at index start
- Through the end of the string
- Later: more complex 2-arg substring()



The substring() method picks out a part of string using index numbers to identify the desired part. The simplest form, **substring(int start)** takes a start index number and returns a new string made of the chars starting at that index and running through the end of the string:

String str = "Hello";
String a = str.substring(1); // a is "ello" (i.e. starting at index 1)
String b = str.substring(2); // b is "llo"

String c = str.substring(3); // c is "lo"

Above str.substring(1) returns "ello", picking out the part of "Hello" which begins at index 1 (the "H" is at index 0, the "e" is at index 1).

Java String valueOf()

- The **java string valueOf()** method converts different types of values into string. By the help of string valueOf() method, you can convert int to string, long to string, boolean to string, character to string, float to string, double to string, object to string and char array to string.

Internal implementation

```
public static String valueOf(Object obj)
 {
     return (obj == null) ? "null" : obj.toString();
  }
```

Signature

The signature or syntax of string valueOf() method is given below:

1. **public static** String valueOf(**boolean** b)
2. **public static** String valueOf(**char** c)
3. **public static** String valueOf(**char**[] c)
4. **public static** String valueOf(**int** i)
5. **public static** String valueOf(**long** l)
6. **public static** String valueOf(**float** f)
7. **public static** String valueOf(**double** d)
8. **public static** String valueOf(Object o)

Returns

String representation of given value

Java String valueOf() method example

```
public class StringValueOfExample
{
        public static void main(String args[])
        {
                int value=30;
                String s1=String.valueOf(value);
                System.out.println(s1+10);//concatenating string with 10
        }
}
```

Output: `3010`

3010

Java String valueOf(boolean bol) Method Example

This is a boolean version of overloaded valueOf() method. It takes boolean value and returns a string. Let's see an example.

```
public class StringValueOfExample2
{
   public static void main(String[] args)
 {
      // Boolean to String
      boolean bol = true;
      boolean bol2 = false;
      String s1 = String.valueOf(bol);
      String s2 = String.valueOf(bol2);
      System.out.println(s1);
      System.out.println(s2);
   }
}
```

Output:
true
false

Applications:

1. **Count Substrings with Only One Distinct Letter.**
2. **Valid Word Abbreviation.**
3. **Longest Repeating Substring.**
4. **Longest Common Subsequence.**
5. **Longest Increasing Subsequence.**

Count Substrings with Only One Distinct Letter

Given a string s, return the number of substrings
that have only **one distinct** letter.
Example 1:
Input: s = "aaaba"
Output: 8
Explanation: The substrings with one distinct letter are "aaa", "aa", "a", "b".
"aaa" occurs 1 time.
"aa" occurs 2 times.
"a" occurs 4 times.
"b" occurs 1 time.
So the answer is 1 + 2 + 4 + 1 = 8.

Example 2:
Input: s = "aaaaaaaaaa"
Output: 55
Constraints:
1 <= s.length <= 1000
s[i] consists of only lowercase English letters.

Java Program for Count Substrings with Only One Distinct Letter

```
import java.util.*;

class CountSubstrings
{
	public int countLetters(String s)
	{
		int sum = 1;
		int count = 1;

		for(int i = 1; i < s.length(); i++)
		{
			if(s.charAt(i) == s.charAt(i-1))
				count++;
			else
				count = 1;

			sum += count;
		}
		return sum;
```

```
        }

        public static void main(String args[])
        {
                Scanner sc = new Scanner(System.in);
                String s=sc.next();
                System.out.println(new CountSubstrings().countLetters(s));
        }
}
```

Sample Input-1:

---

abcde

Sample Output-1:

5

Sample Input-2:

---

aaabbb

Sample Output-2:

12

Valid Word Abbreviation

Given a **non-empty** string s and an abbreviation abbr , return whether the string matches with the given abbreviation.

A string such as "word" contains only the following valid abbreviations:

["word", "1ord", "w1rd", "wo1d", "wor1", "2rd", "w2d", "wo2", "1o1d", "1or1", "w1r1", "1o2", "2r1", "3d", "w3", "4"]

Notice that only the above abbreviations are valid abbreviations of the string "word" . Any other string is not a valid abbreviation of "word".

Note: Assume s contains only lowercase letters and abbr contains only lowercase letters and digits.

Example 1:

Given **s** = "internationalization", abbr = "i12iz4n"

Return true.

Example 2:
Given s = "apple", abbr = "a2e"
Return false.

Algorithm

1) Initialize Pointers: Start with two pointers, pw for the word and pa for the abbr. Both should start from index 0.

2) Iterate through abbr: While both pointers are within bounds of their respective strings

(pw < len(word) and pa < len(abbr)), compare the characters:

Direct Match: If word[pw] equals abbr[pa], simply move both pointers to the next characters (pw++, pa++).

Handling Numbers: If abbr[pa] is a number, it indicates a length of characters to skip in word:

Check if the number starts with 0. If yes, it's invalid since numbers can't have leading zeros. Return False in such cases.

Calculate the total number by moving the pa pointer across consecutive numeric characters, updating a num variable that accumulates this numerical value.

Move the pw pointer forward by this num value, effectively skipping these characters in the word.

Mismatch: If neither conditions above apply, it means there's a mismatch (e.g. a non-matching character in abbr that's not a number). In this case, return False.

3) Final Validity Check: After exhausting both word and abbr, ensure both pw and pa have traversed their respective strings completely. If yes, the abbreviation is valid; otherwise, it is not.

Java Program for Valid Word Abbreviation: **ValidWordAbbreviation.java**

```
import java.util.*;


class ValidWordAbbreviation
{
   public boolean validWordAbbreviation(String word, String abbr)
  {
      int pw = 0;
```

```
    int pa = 0;
    while(pw < word.length() && pa < abbr.length()) {
                        // characters should match
       if(Character.isLetter(abbr.charAt(pa))) {
          if(word.charAt(pw) != abbr.charAt(pa))
                                    return false;
          pw++;
          pa++;
       }
       else {
       // If it is numeric, the word should get updated by that length
          int number = 0;
          while(pa < abbr.length() && Character.isDigit(abbr.charAt(pa))) {
             number *= 10;
             number += abbr.charAt(pa++) - '0';
             System.out.println("number " + number);
             if(number == 0)
                return false;
          }
          pw += number;
        System.out.println("pw " + pw + " pa " + pa);
       }
    }
       System.out.println("Final pw " + pw + " pa " + pa);
       return pw == word.length() && pa == abbr.length();
  }
       public static void main(String args[])
       {
```

```
        Scanner sc=new Scanner(System.in);
        String s=sc.next();
        String t=sc.next();
        System.out.println(new ValidWordAbbreviation().validWordAbbreviation(s,t));
    }
}
```

Longest Repeating Substring

Given a string S, find out the length of the longest repeating substring(s). Return 0 if no repeating substring exists.

Example 1:

Input: "abcd"

Output: 0

Explanation: There is no repeating substring.

Example 2:

Input: "abbaba"

Output: 2

Explanation: The longest repeating substrings are "ab" and "ba", each of which occurs twice.

Example 3:

Input: "aabcaabdaab"

Output: 3

Explanation: The longest repeating substring is "aab", which occurs 3 times.

Example 4:

Input: "aaaaa"

Output: 4

Explanation: The longest repeating substring is "aaaa", which occurs twice.

Note:

The string S consists of only lowercase English letters from 'a' - 'z'.

1 <= S.length <= 1500

LRS.java

```
import java.util.*;

class LRS
{
	public int longestRepeatingSubstring(String s)
	{
		int n = s.length();
		for(int len = (n-1); len >= 1; len--){
			HashSet<String> set = new HashSet<>();
			for(int i = 0; i <= (n-len); i++){
				String subs = s.substring(i,i+len);
				System.out.println(subs);
				if(set.contains(subs)){
					return len;
				}
				set.add(subs);
			}
		}
		return 0;
	}
}

	public static void main(String args[])
	{
		Scanner sc=new Scanner(System.in);
		String s=sc.next();
		System.out.println(new LRS().longestRepeatingSubstring(s));
	}
}
```

Longest Common Subsequence

Given two strings text1 and text2 , return the length of their longest common subsequence.
A *subsequence* of a string is a new string generated from the original string with some characters(can be none) deleted without changing the relative order of the remaining characters.

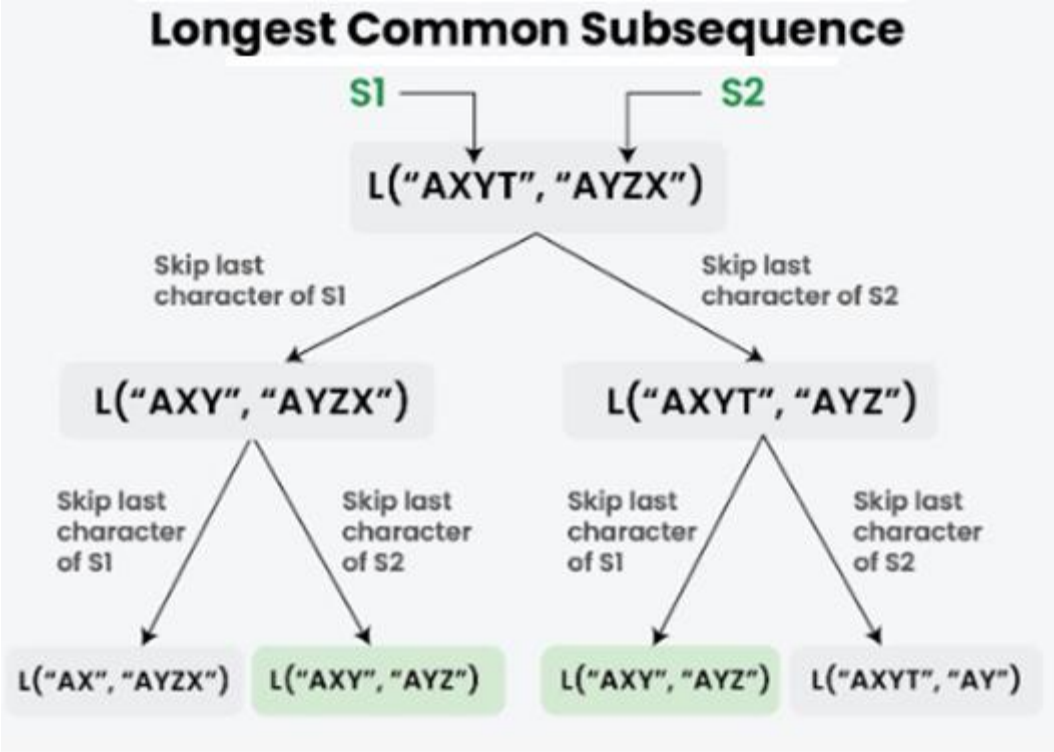
(eg, "ace" is a subsequence of "abcde" while "aec" is not). A *common subsequence* of two strings is a subsequence that is common to both strings.
If there is no common subsequence, return 0.

Example 1:
Input: text1 = "abcde", text2 = "ace"
Output: 3
Explanation: The longest common subsequence is "ace" and its length is 3.

Example 2:
Input: text1 = "abc", text2 = "abc"
Output: 3
Explanation: The longest common subsequence is "abc" and its length is 3.

Example 3:
Input: text1 = "abc", text2 = "def"
Output: 0
Explanation: There is no such common subsequence, so the result is 0.

Constraints:
1 <= text1.length <= 1000
1 <= text2.length <= 1000
The input strings consist of lowercase English characters only.



Overlapping Subproblems in Longest Common Subsequence

LCSRecursive.java

```
import java.util.*;

class LCSRecursive
{
	/*
	Time Complexity: O(2 ^ n)
	Space Complexity: O(1)
	*/

	/*
	Consider the input strings abcde and abode
	Last characters match for the strings. So length of LCS can be written as:
	lcs("abcde", "abode") = 1 + lcs("abcd", "abod")

	Consider the input strings abcde and aboda
	Last characters do not match for the strings. So length of LCS can be written as:
	lcs("abcde", "aboda") = Math.max(lcs("abcde", "abod"), lcs("abcd", "aboda"))
	*/

	public int lcs(char[] c1, char[] c2, int m, int n)
	{
		if (m == 0 || n == 0)
```

```
                return 0;
        if (c1[m -1 ] == c2[n-1])
                return 1 + lcs(c1, c2, m-1, n-1);
        else
                return Math.max(lcs(c1, c2, m, n-1), lcs(c1, c2, m-1, n));
    }
    public static void main(String args[])
    {
        Scanner sc=new Scanner(System.in);
        String s1=sc.next();
        String s2=sc.next();
        System.out.println(new LCSRecursive().lcs(s1.toCharArray(),s2.toCharArray(),
        s1.length(), s2.length()));
    }
}
```

LCSMemo.java

```
import java.util.*;
class LCSMemo
{
    /*
    Time Complexity: O(m * n)
    Space Complexity: O(m * n)
    */

    static int[][] dp;

    public int lcs(char[] c1, char[] c2, int m, int n)
    {
        if (m == 0 || n == 0)
            return 0;

        // if the same state has already been computed
        if (dp[m - 1][n - 1] != 0)
        {
            return dp[m - 1][n - 1];
        }

        if (c1[m -1 ] == c2[n-1])
            dp[m - 1][n - 1] = 1 + lcs(c1, c2, m-1, n-1);
        else
            dp[m - 1][n - 1] = Math.max(lcs(c1, c2, m, n-1), lcs(c1, c2, m-1, n));

        return dp[m - 1][n - 1];
    }
```

```
        public static void main(String args[])
        {
                Scanner sc = new Scanner(System.in);
                String s1=sc.next();
                String s2=sc.next();
                dp = new int[s1.length() + 1][s2.length() + 1];

                System.out.println(new LCSMemo().lcs(s1.toCharArray(),s2.toCharArray(),
                s1.length(), s2.length()));
        }
}
```

LCSdp.java

```
import java.util.*;

class LCSdp
{
        /*
        Time Complexity: O(m * n)
        Space Complexity: O(m * n)
        */

        public static int lcs(String text1, String text2)
        {
                int[] dp = new int[text2.length()+1];
                for(int i = 0; i < text1.length(); i++)
                {
                        int prev = dp[0];
                        for(int j = 1; j < dp.length; j++)
                        {
                                int temp = dp[j];
                                if(text1.charAt(i) != text2.charAt(j-1))
                                {
                                        dp[j] = Math.max(dp[j-1], dp[j]);
                                }
                                else
                                {
                                        dp[j] = prev +1;
                                }
                                prev = temp;
                        }
                }
                System.out.println(Arrays.toString(dp));
                return dp[dp.length-1];
        }
```

```
        public static void main(String args[])
        {
                // asian always
                // abcde adobe
                // abcde abode
                Scanner sc = new Scanner(System.in);
                String s1 = sc.next();
                String s2 = sc.next();
                System.out.println(new LCSdp().lcs(s1, s2));
        }
}
```

Longest Increasing Subsequence

The Longest Increasing Subsequence (LIS) problem is to find the length of the longest subsequence of a given sequence such that all elements of the subsequence are sorted in increasing order.

For example, the length of LIS for {10, 22, 9, 33, 21, 50, 41, 60, 80} is 6 and LIS is {10, 22, 33, 50, 60, 80}.

arr[]	10	22	9	33	21	50	41	60	80
LIS	1	2		3		4		5	6

Example:

Input: arr[] = {5,4,1,2,3}
Output: Length of LIS = 3
Explanation: The longest increasing subsequence is 1,2,3

Input: arr[] = {7,5}
Output: Length of LIS = 1
Explanation: The longest increasing subsequences are {5} or {7}.

LISRecursive.java

```
import java.util.*;

class LISRecursive
{
	/* Iterate from i = 1 to i = nums.length - 1.
	At each iteration, use a loop to iterate
	from j = 0 to j = i - 1 (all the elements before i).
	For each element before i, check if that element is smaller than nums[i].
	Time complexity: Exponential
	Space complexity: O(n) */

	static int longest; // stores the LIS

	public int lengthOfLISRecur(int[] nums)
	{
		if (nums == null || nums.length == 0)
			return 0;

		int max = 1;
		for(int i = 1; i < nums.length; i++)
		{
			max = Math.max(max, helper(nums, i));
		}
		return max;
	}
	protected int helper(int[] nums, int i)
	{
		System.out.println("i " + i);
		if(i == 0)
			return 1;
		int longest = 1;
		for (int j = 0; j < i; j++)
		{
			if(nums[i] > nums[j])
				longest = Math.max(longest, helper(nums, j) + 1);
		}
```

```
            return longest;
        }

        public static void main(String[] args)
        {
            Scanner sc = new Scanner(System.in);
            int n = sc.nextInt();
            int ar[] = new int[n];
            for(int i = 0; i < n; i++)
                ar[i] = sc.nextInt();
            System.out.println(new LISRecursive().lengthOfLISRecur(ar));
        }
}
```

LISMemo.java

```
import java.util.*;

class LISMemo
{
        // Time complexity: O(n^2)
        // Space complexity: O(n)

        public int lengthOfLISMemo(int[] nums)
        {
            if (nums == null || nums.length == 0)
                return 0;

            int[] dp = new int[nums.length];
            int max = 1;
            for(int i = 1; i < nums.length; i++)
            {
                max = Math.max(max, helper(nums, i, dp));
            }
            return max;
        }

        protected int helper(int[] nums, int i, int[] dp)
        {
            if(i == 0)
            return 1;

            if(dp[i] != 0)
```

```
                return dp[i];

            int longest = 1;
            for (int j = 0; j < i; j++)
            {
                if(nums[i] > nums[j])
                    longest = Math.max(longest, helper(nums, j, dp) + 1);
            }

            dp[i] = longest;
            return longest;
        }




        public static void main(String[] args)
        {
            Scanner sc = new Scanner(System.in);
            int n = sc.nextInt();
            int ar[] = new int[n];
            for(int i = 0; i < n; i++)
                ar[i] = sc.nextInt();
            System.out.println(new LISMemo().lengthOfLISMemo(ar));
        }
}
```

LISdp.java

```
import java.util.*;
class LISdp
{
        /*Time complexity: O(n^2)
        Space complexity: O(n)
        Initialize an array dp with length nums.length and all elements equal to 1.
        dp[i] represents the length of the longest increasing subsequence that ends with the element
        at index i.

        Iterate from i = 1 to i = nums.length - 1.
        At each iteration, use a second for loop to iterate from j = 0 to j = i - 1 (all the elements
        before i).
        For each element before i, check if that element is smaller than nums[i].
        If so, set dp[i] = max(dp[i], dp[j] + 1).

        Return the max value from dp. */

        public int lengthOfLISDP(int[] nums)
```

```
    {
        if (nums == null || nums.length == 0)
        return 0;

        int[] dp = new int[nums.length];
        int longest = 0;
        Arrays.fill(dp, 1);

        for (int i = 1; i < nums.length; i++)
        {
            for (int j = 0; j < i; j++)
            {
                if (nums[i] > nums[j])
                {
                    dp[i] = Math.max(dp[i], dp[j] + 1);
                }
            }
            longest = Math.max(longest, dp[i]);
        }
        System.out.println(Arrays.toString(dp));
        return longest;
    }



    public static void main(String[] args)
    {
        Scanner sc = new Scanner(System.in);
        int n = sc.nextInt();
        int ar[] = new int[n];
        for(int i = 0; i < n; i++)
            ar[i] = sc.nextInt();
        System.out.println(new LISdp().lengthOfLISDP(ar));
    }
}
```