Syllabus:

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

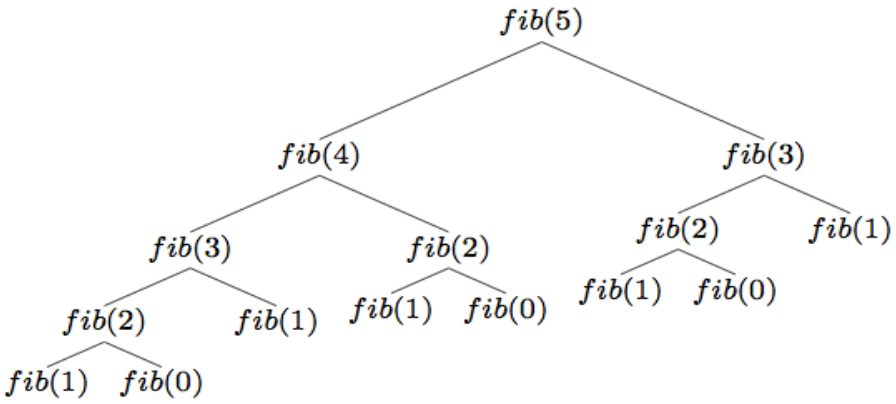
The **above code** is the **recursive implementation of Fibonacci numbers**. The fib() function is called twice: once to calculate (n - 1) and once to calculate (n - 2). Therefore, the time complexity of this function is 2 raised to the power 'n'.

For example:

Consider the case when n = 5.

1. We call fib(5), which in turn calls fib(4) and fib(3)
2. fib(4) in turn calls fib(3) and fib(2)
3. fib(3) in turn 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.

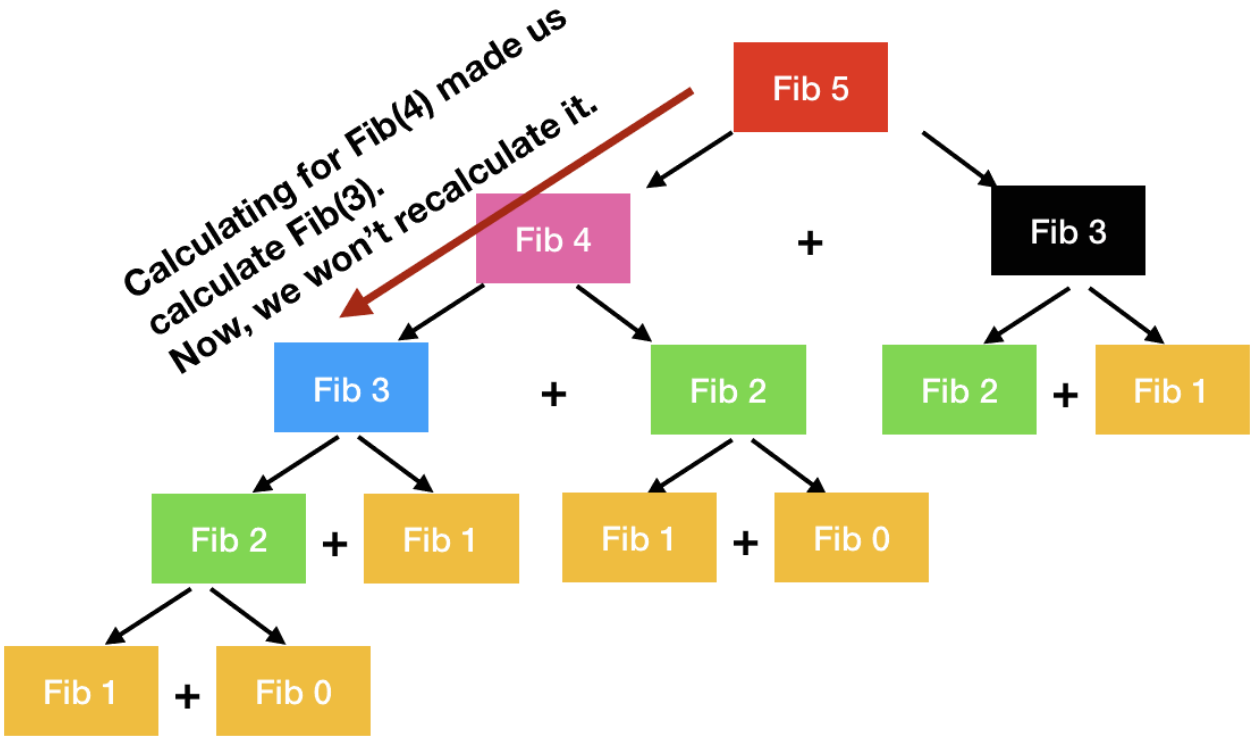
- **In the tabulation method,** we store the results of each function call in a data structure. Whenever an already stored value is needed, we fetch the value from the data structure instead of computing it repeatedly.

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

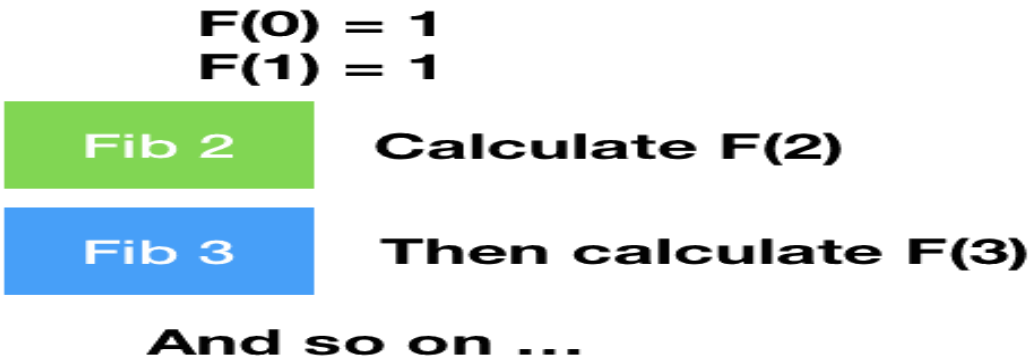
The way we solved the Fibonacci series was the top-down approach. We just start by solving the problem in a natural manner and stored the solutions of the subproblems along the way. We also use the term **memoization**, a word derived from memo for this.



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 2nd term and then 3rd 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.

- 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.

// Fibonacci Series

```
import java.util.*;

class dpFib
{
        static int count;
```

```
static int fibR(int n)
{
        count++;
        if(n <= 1)
                return n;
        return fibR(n - 1) + fibR(n - 2);
}

public static int fibMem(int n, Map<Integer,Integer> map)
{
        count++;
        if(n <= 1)
                return n;

        if(map.containsKey(n))
                return map.get(n);

        Integer fibN = fibMem(n-1, map) + fibMem(n-2, map);
        map.put(n, fibN);

        return fibN;
}

static int fibDP(int n)
{
        count++;
        int dp[] = new int[n + 1];
        dp[0] = 0;
        dp[1] = 1;

        for (int i= 2; i <= n; i++)
                dp[i] = dp[i-1] + dp[i-2];

        System.out.println("fib DP count: " + Arrays.toString(dp));

        return dp[n];
}

static int fib(int n)
{
        // Memoized version to find fibonacci series
        // To speed up we store the values of calculated states
        HashMap<Integer, Integer> memoizedMap = new HashMap<>();
```

```
            memoizedMap.put(0, 0);
            memoizedMap.put(1, 1);

            return fibMem(n, memoizedMap);
        }

        public static void main(String args[])
        {
            Scanner sc=new Scanner(System.in);
            int n = sc.nextInt();
            System.out.println(dpFib.fibR(n));
            System.out.println("fib Recur count: " + count);
            count = 0;
            System.out.println(dpFib.fib(n));
            System.out.println("fib Mem count: " + count);
            count = 0;
            System.out.println(dpFib.fibDP(n));
            System.out.println("fib DP count: " + count);
        }
}
```

Applications:

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

1. **Climbing Stairs:**

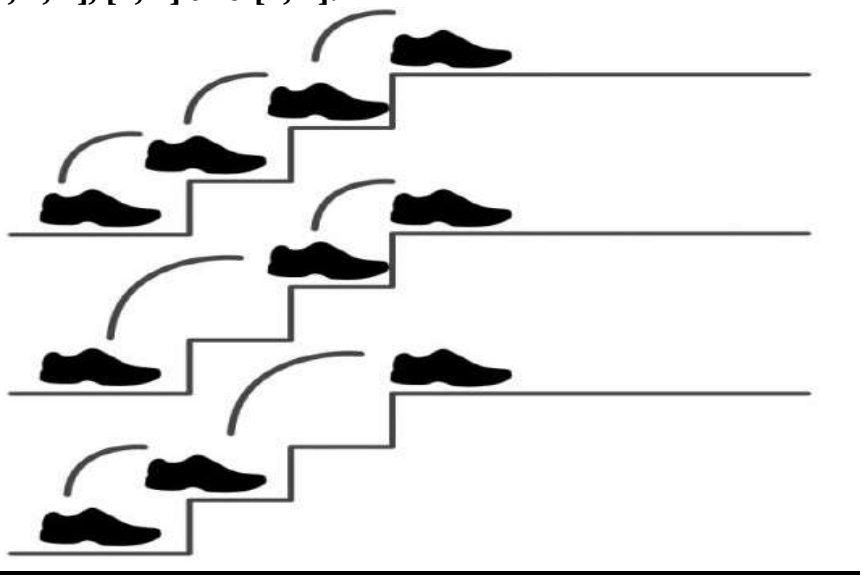
   Given a staircase of **N** steps and you can either climb 1 or 2 steps at a given time. The task is to return the count of distinct ways to climb to the top. **Note:** The order of the steps taken matters.

   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
{
   public int climbStairs(int n)
  {
      if (n == 1) {
         return 1;
      }
      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];
        }
        return dp[n];
    }
      public static void main(String[] args)
    {
        Scanner sc=new Scanner(System.in);
        int n=sc.nextInt();
        System.out.println(new ClimbingStairs().climbStairs(n));
    }
}
```

Sample i/p & o/p:

Case=1
input =9
output =55

Case =2
input =2
output =2

Case =3
input =20
output =10946

Java Program for Climbing Stairs Using Dynamic Programming (Top-Down Approach):

ClimbingStairs.java

```
import java.util.*;
class ClimbingStairs
{
   public int climbStairs(int n)
    {
       int[] memo=new int[n+1];

       if (n < 0)
        {
```

```
            return 0;
        }
        else if (memo[n] != 0)
        {
            // If we stored something in our cache reuse it and avoid recalculating everything
              return memo[n];
        }
        else if (n == 0)
        {
            return 1;
        }
        else
        {
// store our calculation inside our cache so we don't have to recalculate it again for
//memo[n]
            memo[n] = climbStairs(n-1) + climbStairs(n-2);
            return memo[n];
        }
    }

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

2. <u>Min Cost Climbing Stairs:</u>

- 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 minCostClimbingStairs(int[] cost)
    {
      int n = cost.length;
```

```
        int[] dp = new int[n];
        for (int i=0; i<n; i++) {
            if (i<2) dp[i] = cost[i];
            else dp[i] = cost[i] + Math.min(dp[i-1], dp[i-2]);
        }
        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().minCostClimbingStairs(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. 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*.**

Using Kadane's Algorithm

- *local_maximum at index i is the maximum of A[i] and the sum of A[i] and local_maximum at index i-1.*

```
local_maximum[i] = max(A[i], A[i] + local_maximum[i-1])
```

- This way, at every index ***i***, the problem boils down to finding the maximum of just two numbers, ***A[i]*** and ***(A[i] + local_maximum[i-1])***.
- Thus the maximum subarray problem can be solved by solving these sub-problems of finding ***local_maximums*** recursively. Also, note that ***local_maximum[0]*** would be ***A[0]*** itself.

- Using the above method, we need to iterate through the array just once, which is a lot better than our previous brute force approach. Or to be more precise, **the time complexity of Kadane's Algorithm is *O(n)*.**

Java Program For Maximum Sub Array using Dynamic Programming:

MaxSubArray.java

```
import java.util.*;

class MaxSubArray
{
   public int maxSubArray(int[] A)
  {
       int n = A.length;
       int[] dp = new int[n];//dp[i] means the maximum subarray ending with A[i];
       dp[0] = A[0];
       int max = dp[0];
             for(int i = 1; i < n; i++){
          dp[i] = A[i] + (dp[i - 1] > 0 ? dp[i - 1] : 0);
          max = Math.max(max, dp[i]);
       }
       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));
    }
}
```

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

4. 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:

The given problem can be solved by using the concepts of all distinct possible pairs from N points which are given by $^{N}C_{2}$. The idea is to store the frequency of pair of cells **(i, j)** having the values as **1s** in the map of pairs, say **M**. After generating the frequency map find the total count of corners rectangle formed using the above formula. Follow the steps below to solve the given problem:

- Initialize a variable, say **count** that stores the resultant count of corner rectangle.
- Initialize a map, say **m[]** that stores the frequency of the cell **(i, j)** having values as **1**.
- Iterate over the range **[0, M)** using the variable **i** and perform the following tasks:
    - Iterate over the range **[0, N)** using the variable **j** and if **mat[i][j]** equals **1** then Iterate over the range **[j_+1, N)** using the variable **k** and if the value of **mat[i][k]** equals **1** then increase the count of **m[{j, k}]** by **1.**
- Traverse over the map **m[]** using the variable **it** and add the value of **it.second*(it.second – 1)/2** to the variable **count.**
- After performing the above steps, print the value of **count** as the answer.

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

CountCornerRectangles.java

```
import java.util.*;

class CountCornerRectangles
{
        /*
        According to the question description, 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)

        Time Complexity: O(Col * rows^2)
        */

        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
                                                counter++;
                                                System.out.println("i " + i + " j " + j + " k " + k
+ " ans " + ans + " counter " + counter);
                                        }
                                }
                        }
                }
                return ans;
        }

        /*
        According to the question description, 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 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. Thus, we use a 2D array dp[n][n] to track
the
        number of pairs of corner points before the current row with position in i, j,
        then the number of rectangles with a right line made up of the current two
        left points will be dp[i][j]. After that, update dp[i][j] by incrementing one.

        Time Complexity: O(M * N^2)
        */

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

      int nrows = grid.length, ncols = grid[0].length;
      int[][] dp = new int[ncols][ncols];
      int res = 0;
```

```
    for(int i = 0;i < nrows;i++)
                {
        for(int j = i + 1; j < nrows;j++)
                        {
            for(int k = 0; k < ncols; k++)
                                {
                                        if (grid[i][k] + grid[j][k] == 2)
                                        {
                                                res += dp[i][j];
                                                System.out.println("i " + i + " j " + j + " k " + k
+ " res " + res);
                                                dp[i][j]++;
                                        }
            }
                                System.out.println(Arrays.deepToString(dp));
        }
    }
                return res;
  }


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

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

5. Matrix Chain Multiplication:

- Suppose, We are given a sequence (chain) **(A1, A2……An)** of n matrices to be multiplied, and we wish to compute the product **(A1A2…..An)**.
- We can evaluate the above expression using the standard algorithm for multiplying pairs of matrices as a subroutine once we have parenthesized it to resolve all ambiguities in how the matrices are multiplied together.
- Matrix multiplication is associative, and so all parenthesizations yield the same product.
- For example, if the chain of matrices is **(A1, A2, A3, A4)** then we can fully parenthesize the product (**A1A2A3A4)** in five distinct ways:

  1:-(A1(A2(A3A4))) ,
  2:-(A1((A2A3)A4)),
  3:- ((A1A2)(A3A4)),
  4:-((A1(A2A3))A4),
  5:-(((A1A2)A3)A4) .

- We can multiply two matrices A and B only if they are compatible. the number of columns of A must equal the number of rows of B.
- If A is a p x q matrix and B is a q x r matrix, the resulting matrix C is a p x r matrix.
- The time to compute C is dominated by the number of scalar multiplications is pqr.
- We shall express costs in terms of the number of scalar multiplications.
- **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**.
- So problem is we can perform a many time of cost multiplication and repeatedly the calculation is performing. so this general method is very time consuming and tedious.
- So we can apply **dynamic programming** for solve this kind of problem.

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

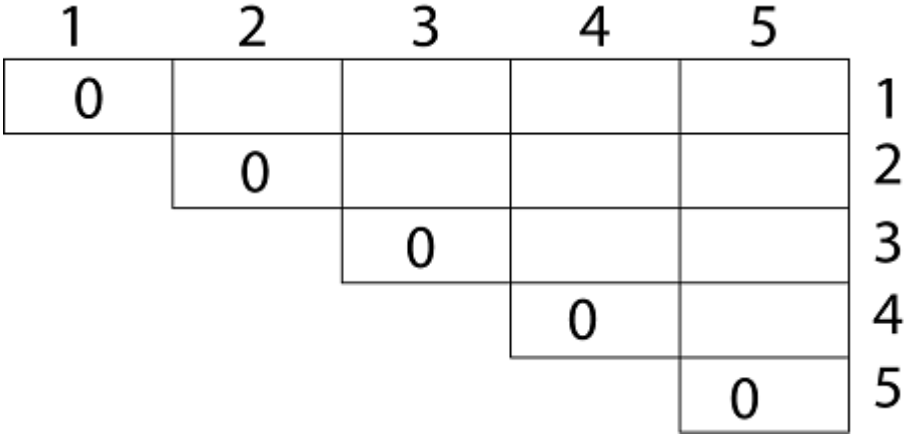
1. Characterize the structure of an optimal solution.
2. Recursively define the value of an optimal solution.
3. Compute the value of an optimal solution.
4. Construct an optimal solution from computed information.

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

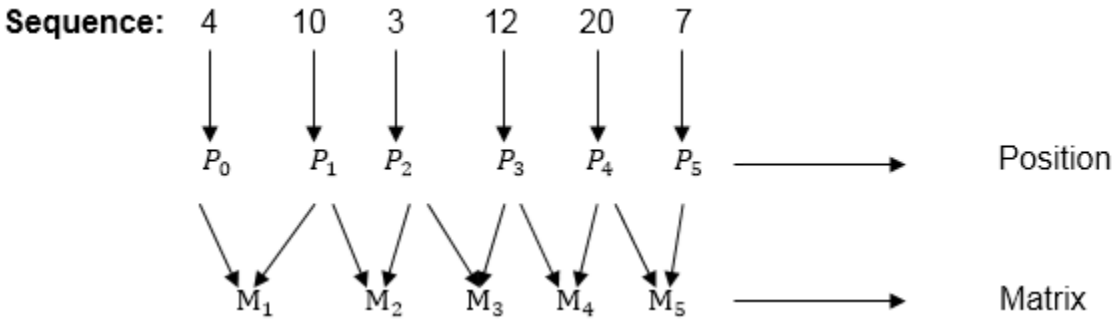
m[i , j] = min { m[i , k] + m[i+k , j] + pi-1*pk*pj } if i < j…. where p is dimension of matrix , i ≤ k < j …..

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], $0 \leq i, j \leq 5$. We know M [i, i] = 0 for all i.



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

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

M [1, 4] =1080

As comparing the output of different cases then '**1080**' is minimum output, so we insert 1080 in the table and (M1 xM2) x (M3 x M4) combination is taken out in output making,

M [2, 5] = M2 M3 M4 M5

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

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

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

M [2, 5] = 1350

As comparing the output of different cases then '**1350**' is minimum output, so we insert 1350 in the table and M2 x(M3 x 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

$$M[1,5] = \min \begin{cases} M[1,4] + M[5,5] + p_0p_4p_5 = 1080 + 0 + 4.20.7 = 1544 \\ M[1,3] + M[4,5] + p_0p_3p_5 = 264 + 1680 + 4.12.7 = 2016 \\ M[1,2] + M[3,5] + p_0p_2p_5 = 120 + 1140 + 4.3.7 = 1344 \\ M[1,1] + M[2,5] + p_0p_1p_5 = 0 + 1350 + 4.10.7 = 1630 \end{cases}$$

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[])
        {
            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));
        }
}
```

Output:

```
The minimum cost is 4500
```

The time complexity of the above top-down solution is $O(n^3)$ and requires $O(n^2)$ extra space, where `n` is the total number of matrices.

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.

```
import java.util.*;
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 = 1; len < n - 1; len++)
		{
			for (i = 1; i < n - len; i++)
			{
				j = i + len;
				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];
```

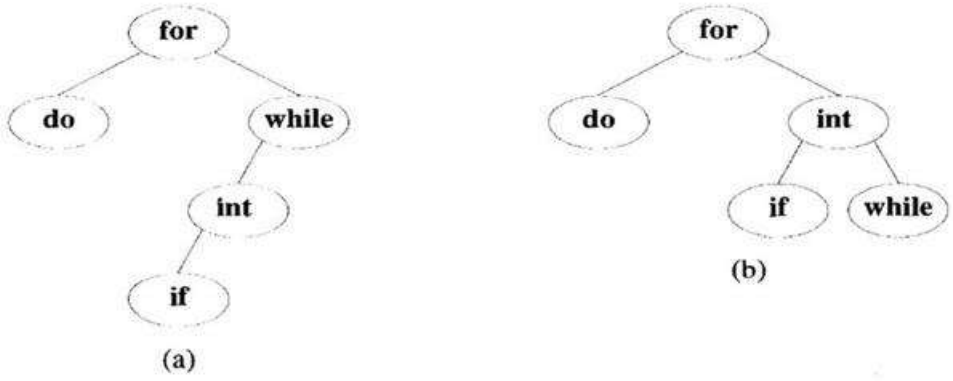
```
    }

    public static void main(String args[])
    {
        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));
    }
}
```

Output:

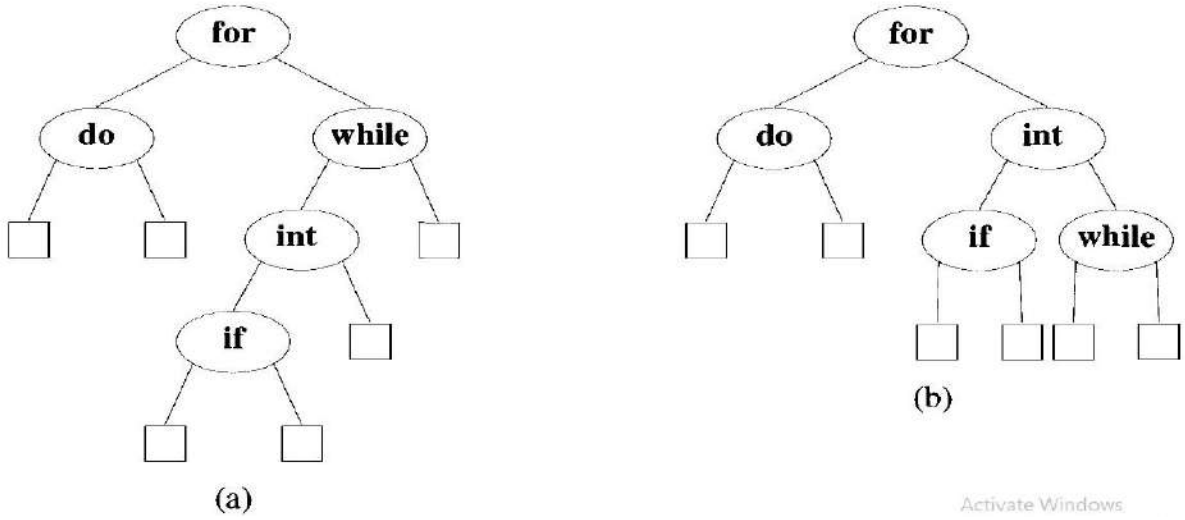
The minimum cost is 4500

6. 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 compute the 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 time is *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.



- Given a fixed set of identifiers, we wish to create a binary search tree organization.
- We may expect different binary search trees for the same identifier set to have different performance characteristics.
- The tree of Figure(a) in, the worst case, requires four comparisons to find an identifier, whereas the tree of Figure(b) requires only three.
  On the average the two trees need 12/5 and 11/5 comparisons respectively.
  For example in the case of tree(a), it takes 1,2,2,3 and 4 comparisons, respectively,
  to find the identifiers for, do, while, int & if.
  Thus the average number of comparisons is **(1+2+2+3+4)/5=12/5.**
- This calculation assumes that each identifier is searched for with equal probability and that no unsuccessful searches (i.e.searches for identifiers not in the tree) are made.
- In a general situation, we can expect different identifiers to be searched for with different frequencies (or probabilities).
- In addition, we can expect unsuccessful searches also to be made.
- Let us assume that the given set of identifiers is {a1,a2,……,an} with a1 < a2 < ……< an.
- Let p(i) be the probability with which we search for a_i.
- Let q(i) be the probability that the identifier 'x' being searched for is such that $a_i < x < a_{i+1}$, where $0 \le i \le n$.
  Then, $\Sigma_{0 \le i \le n}$ q(i) the probability of an unsuccessful search.
  Clearly,
  $$\Sigma_{0 \le i \le n} p(i) + \Sigma_{0 \le i \le n} q(i) = 1.$$
  - Given this data, we wish to construct an optimal binary search tree for {a1,a2,….. a_n,}.
  - In obtaining a cost function for binary search trees, it is useful to add a ''fictitious'' node in place of every empty sub tree in the search tree. Such nodes, called "External nodes". All other nodes are internal nodes.
  - In Fig. Square boxes indicates External Nodes.



- **If a binary search tree represents n identifiers, then there will be exactly 'n' internal nodes and n+1 (fictitious) external nodes.**
- Every internal node represents a point where a successful search may terminate.
- Every external node represents a point where an unsuccessful search may terminate.
- The expected cost contribution from the internal node for a_i is
  →p(i) * level(a $_i$).
- The expected cost contribution from the External node for E_i is
  →q(i) * level((E $_i$) -1).
- The preceding discussion leads to the following formula for the expected Cost of a binary search tree

$$\Sigma_{1 \le i \le n} p(i) * level(a_i) + \Sigma_{0 \le i \le n} q(i) * level((E_i) - 1) = 1.$$

```
// OBSTDP.java

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 root
considered
                        // 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];
                    System.out.println("i " + i + " j " + j + " r " + r + " c " + c);
                    if(c < dp[i][j])
                        dp[i][j]=c;
                }
                System.out.println(Arrays.deepToString(dp));
            }
        }
        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];
        }
                System.out.println("fsum " + " i "+ i + " j " + j + " sum " + s);
        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));
        }
```

}

7. All Pairs Shortest Paths:

- Let G = (V, E) be a Directed graph with n vertices. Let cost be a cost Adjacency matrix for G such that cost(i , i)=0, $1 \le i \le n$.
- Then **cost(i, j) is the length (or cost) of edge (i, j) if (i, j) Є E(G).**
  And cost(i, j)= ∞ and if i≠ j and (i,j) ∉ E(G).
- The all-pairs shortest-path problem is to determine a matrix 'A' such that A(i, j) is the length of a shortest path from i to j.
- The matrix 'A' can be obtained by solving n single-source problems using the algorithm Shortest Paths. Since each application of this procedure requires $0(n^2)$ time.
- The matrix 'A' can be obtained in $0(n^3)$ time.
- We obtain an alternate $0(n^3)$ solution to this problem using the principle of optimality.
- Let us examine a shortest i to j path in G, i≠ j This path originates at vertex i and goes through some intermediate vertices (possibly none) and terminates at vertex j.
- We can assume that this path contains no cycles for if there is a cycle, then this can be deleted without increasing the path length (no cycle has negative length)
- If 'k' is an intermediate vertex on this shortest path, then the sub paths from i to k and from k to j must be shortest paths from i to k and k to j, respectively.
- Otherwise, the i to j path is not of minimum length. So, the principle of optimality holds.
- This alerts us to the prospect of using dynamic programming.
- If 'k' is the intermediate vertex with highest index, then the i to k path is a shortest i to k path in G going through no vertex with index greater than k -1.
- Similarly the k to j path is a shortest k to j path in G going through no vertex of index greater than k-1.
- We can regard the construction of a shortest i to j path as first requiring a decision as to which is the highest indexed intermediate vertex k.
- Once this decision has been made, we need to find two shortest paths,
- One from i to k and the other from k to j.
- Neither of these may go through a Vertex with index greater than k -1.
- Using A^k (i, j) to represent the length of a shortest path from i to j going through no vertex of index greater than k, we obtain

  $$\mathbf{A(i, j)= \min\{ \min_{1\le k\le n}\{A^{K-1}(i, k) + A^{K-1}(k, j)\}, cost(i, j)\}}$$

  $$\mathbf{Clearly,\ A^0(i, j) = cost(i, j),\ 1\le i\le n\ ,\ 1\le j\le n.}$$

- We can obtain a recurrence for $A^k(i, j)$ using an argument similar to that used before.

- A shortest path from i to j going through no vertex higher than 'k' either goes through vertex k or it does not.
- If it does, $A^k(i, j) = A^{k-1}(i, k) + A^{k-1}(k, j)$

- If it does not, then no intermediate vertex has index greater than k-1.
  - Hence $A^k(i, j) = A^{k-1}(i, j)$

  Combining

$A^k(i, j)=\min\{A^{K-1}(i, j) , A^{K-1}(i, k) +A^{K-1}(k, j)\}$ $k\geq 1$

Pseudo code:

```
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  }
```


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 :

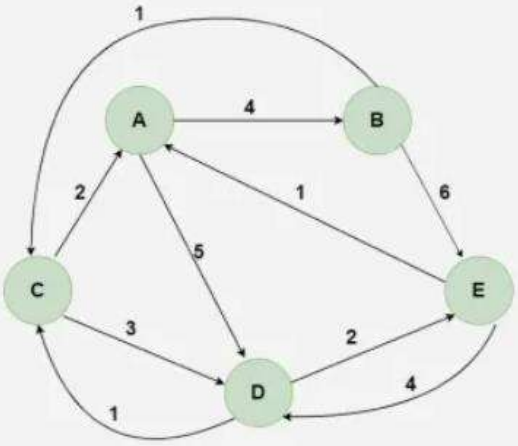
Solution:

Optimal substructure formula for Floyd's algorithm,

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

01 Step | **Understanding the Problem**: Given a matrix dist[n][n] where dist[i][j] represents the weight from node i to j (e.g., A → B → C). If no direct edge exists, dist[i][j] = INF(e.g. 10^8)and for all nodes, dist[i][i] = 0.

	A	B	C	D	E
A	0	4	10^8	5	10^8
B	10^8	0	1	10^8	6
C	2	10^8	0	3	10^8
D	10^8	10^8	1	0	2
E	1	10^8	10^8	4	0



02 Step | Treat first node (e.g. A) as an intermediate node and calculate dist[][] for every (i, j) node using the formula
dist[i][j] = min(dist[i][j], dist[i][A] + dis[A][j]).

	A	B	C	D	E
A	0	4	10^8	5	10^8
B	10^8	0	1	10^8	6
C	2	10^8	0	3	10^8
D	10^8	10^8	1	0	2
E	1	10^8	10^8	4	0

	A	B	C	D	E
A	0	4	10^8	5	10^8
B	10^8	0	1	10^8	6
C	2	6	0	3	10^8
D	10^8	10^8	1	0	2
E	1	5	10^8	4	0

03 Step Treat second node (e.g. B) as an intermediate node and calculate dist[][] for every (i, j) node using the formula
dist[i][j] = min(dist[i][j], dist[i][B] + dis[B][j]).

	A	B	C	D	E
A	0	4	10^8	5	10^8
B	10^8	0	1	10^8	6
C	2	6	0	3	10^8
D	10^8	10^8	1	0	2
E	1	5	10^8	4	0

	A	B	C	D	E
A	0	4	5	5	10
B	10^8	0	1	10^8	6
C	2	6	0	3	12
D	10^8	10^8	1	0	2
E	1	5	6	4	0

04 Step Treat third node (e.g. C) as an intermediate node and calculate dist[][] for every (i, j) node using the formula
dist[i][j] = min(dist[i][j], dist[i][C] + dis[C][j]).

	A	B	C	D	E
A	0	4	5	5	10
B	10^8	0	1	10^8	6
C	2	6	0	3	12
D	10^8	10^8	1	0	2
E	1	5	6	4	0

	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

05 Step

Treat fourth node (e.g. D) as an intermediate node and calculate dist[][] for every (i, j) node using the formula
dist[i][j] = min(dist[i][j], dist[i][D] + dis[D][j]).

	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

	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

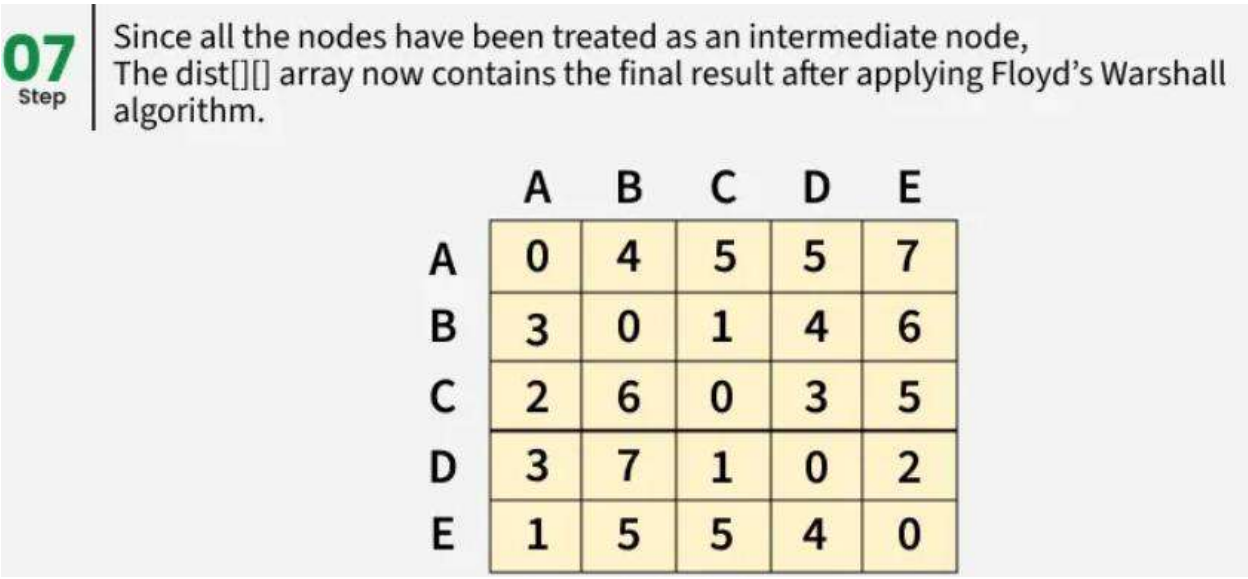
---

06 Step

Treat fifth node (e.g. E) as an intermediate node and calculate dist[][] for every (i, j) node using the formula
dist[i][j] = min(dist[i][j], dist[i][E] + dis[E][j]).

	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

	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



	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;

	/*
	Time Complexity: O(V^3)
	Auxiliary Space: O(V^2)

	Ideally you should take INF as INT_MAX. In that case modify the below code to

	if (dist[i][k] != INF && dist[k][j] != INF && dist[i][k] + dist[k][j] < dist[i][j])
	{
		dist[i][j] = dist[i][k] + dist[k][j];
	}

	*/
	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];

            printSolution(dist);


        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(dist[i][k] != 99 && dist[k][j]!= 99)
                    {
                        // If vertex k is on the shortest path from i to j,
then update the value of dist[i][j]
                        dist[i][j]    =    Math.min(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);
        }
}
```

8. 0/1 Knapsack Problem:

- Given a set of items, each having different weight and value or profit associated with it.
- Find the set of items such that the total weight is less than or equal to a capacity of the knapsack and the total value earned is as large as possible.
- The knapsack problem is useful in solving resource allocation problem.
  Let X = < x_1, x_2, x_3, ... , x_n> be the set of n items. Sets W = <w_1, w_2, w_3, . . . , w_n> and V = < v_1, v_2, v_3, . . . , v_n> are weight and value associated with each item in X.
  Knapsack capacity is M unit.
- The knapsack problem is to find the set of items which maximizes the profit such that collective weight of selected items does not cross the knapsack capacity.
- Select items from X and fill the knapsack such that it would maximize the profit. Knapsack problem has two variations. 0/1 knapsack, that does not allow breaking of items. Either add an entire item or reject it. It is also known as a binary knapsack. Fractional knapsack allows breaking of items. Profit will be earned proportionally.

Mathematical formulation

We can select any item only ones. We can put items x_i in knapsack if knapsack can accommodate it. If the item is added to a knapsack, the associated profit is accumulated.

Knapsack problem can be formulated as follow :

Maximize sum=vi*xi subjected to sum=wi*xi <= M

Xi in(0,1) for binary knapsack

// ZeroOneKanpsack.java

```
/*
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.

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.
*/

class ZeroOneKnapsack
{
        static int KnapSackRecur(int[] val, int[] weight, int n, int w) {
                System.out.println("Recursion n " + n + " w "+ w);
         if (n == 0 || w == 0)// Base Case
                return 0;
         // If weight of the nth item is more than Knapsack capacity w,
         // then this item cannot be included
        if (weight[n - 1] > w)
                return KnapSackRecur(val, weight, n - 1, w);
        else {
```

```
                return Math.max(val[n - 1] +
                        KnapSackRecur(val, weight, n - 1, w - weight[n - 1]),
                        KnapSackRecur(val, weight, n - 1, w));
        }
}

/*
Memoization is basically an extension to the recursive approach so
that we can overcome the problem of calculating redundant cases
and thus increased complexity
*/
static int KnapSackMemoization(int[] val, int[] weight, int n, int w, int [][]dp)
{
        System.out.println("Memoization n " + n + " w "+ w);
                // Base Case
         if (n == 0 || w == 0)
                return 0;

        if (dp[n][w] != 0)
                return dp[n][w];

      // If weight of the nth item is more than Knapsack capacity W,
      // then this item cannot be included
      if (weight[n - 1] > w)
         return dp[n][w] = KnapSackMemoization(val, weight, n - 1, w, dp);

      // Return the maximum of two cases:

      // (1) nth item included. If we use it, let us add the current value, val[n-1], to our solution using any of the previous n-1 items, but with capacity w - weight[n - 1] (current_weight)

      // (2) not included
      else
         return dp[n][w] = Math.max(val[n - 1] +
                KnapSackMemoization(val, weight, n - 1, w - weight[n - 1], dp),
                KnapSackMemoization(val, weight, n - 1, w, dp));
        }

// To call memoization function
static int KnapSack(int[] val, int[] weight, int n, int w) {
        int dp[][] = new int[n + 1][w + 1];
        return KnapSackMemoization(val, weight, n, w, dp);
}

/*
In a DP[][] table let's consider all the possible weights from 1 to W
```

as the columns and cost and weight that can be kept as the rows.

The state DP[i][j] will denote maximum value of j-weight considering all values from 1 to ith.
So if we consider wi (weight in ith row) we can fill it in all columns which have weight values > wi. Now two possibilities can take place:

Fill wi in the given column.
Do not fill wi in the given column.

Now we have to take a maximum of these two possibilities:
1) If we do not fill ith weight in jth column then DP[i][j] state will be same as DP[i-1][j]

2) If we fill the weight, DP[i][j] will be equal to the value of wi + value of the column weighing j-wi in the previous row.

So we take the maximum of these two possibilities to fill the current state.
*/

```
    static int KnapSackDP(int[] val, int[] weight, int n, int capacity){
            int i, w;
    int dp[][] = new int[n + 1][capacity + 1];

    // Build table dp[][] in bottom up manner
    for (i = 1; i <= n; i++){
        for (w = 1; w <= capacity; 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][capacity];
}

public static void main(String[] args){
        Scanner sc=new Scanner(System.in);
        int n = sc.nextInt();
        int w = sc.nextInt();
        int val[] = new int[n];
        int wt[] = new int[n];
```

```
        for(int i = 0; i < n; i++){
                val[i] = sc.nextInt();
                wt[i] = sc.nextInt();
        }
        //System.out.println(KnapSackRecur(val, wt, n, w));
        //System.out.println(KnapSack(val, wt, n, w));
        System.out.println(KnapSackDP(val, wt, n, w));
}
}
```

i. Using Tabular Method:

If the weight of the item is larger than the remaining knapsack capacity, we skip the item, and the solution of the previous step remains as it is. Otherwise, we should add the item to the solution set and the problem size will be reduced by the weight of that item. Corresponding profit will be added for the selected item.

Complexity analysis

With n items, there exist 2^n subsets, the brute force approach examines all subsets to find the optimal solution. Hence, the running time of the brute force approach is $O(2^n)$. This is unacceptable for large n.

Dynamic programming finds an optimal solution by constructing a table of size n ´ M, where n is a number of items and M is the capacity of the knapsack. This table can be filled up in O(nM) time, same is the space complexity.

- Running time of Brute force approach is $O(2^n)$.
- Running time using dynamic programming with memorization is O(n * M).

9. Travelling Salesperson Problem:

Problem Statement:

A traveler needs to visit all the cities from a list, where distances between all the cities are known and each city should be visited just once. What is the shortest possible route that he visits each city exactly once and returns to the origin city?

- Travelling salesman problem is the most notorious computational problem.
- We can use brute-force approach to evaluate every possible tour and select the best one. For n number of vertices in a graph, there are $(n - 1)!$ number of possibilities.
- Instead of brute-force using dynamic programming approach, the solution can be obtained in lesser time, though there is no polynomial time algorithm.
- Let G = (V,E) be a directed graph with edge costs C_{ij}.

- The variable C_{ij} is defined such that

- Cij > 0 for all i and j and $C_{ij} = \infty$ if (i, j) $\notin$ E.
- Let |V| = n and assume n > 1.
- A tour of G is a directed simple cycle that includes every vertex in V.
- The cost of a tour is "the sum of the cost of the edges on the tour".
- The traveling sales person problem is to find a tour of minimum cost.
- The traveling sales person problem finds application in a variety of situations.

Example 1:

- Suppose we have to route a postal van to pick up mail from mail boxes located at 'n' different sites.
- An 'n + 1' vertex graph can be used to represent the situation.
- One vertex represents the post office from which the Postal van starts and to which it must return.
- Edge(i, j)is assigned a cost equal to the distance from site 'i' to site 'j'.
- The route taken by the postal van is a tour, and we are interested in finding a tour of minimum length.

Example2:

- Suppose we wish to use a robot arm to tighten the nuts on some piece of machinery on an assembly line.
- The arm will start from its initial position (which is over the first nut to be tightened), successively move to each of the remaining nuts, and return to the initial position.
- The path of the arm is clearly a tour on a graph in which vertices represent the nuts.
- A minimum-cost tour will minimize the time needed for the arm to complete its task (note that only the total arm movement time is variable; the nut tightening time is independent of the tour).

Example 3:

- From a production environment in which several commodities are manufactured on the same set of machines.
- The manufacture proceeds in cycles. In each production cycle, 'n' different commodities are produced. When the machines are changed from production of commodity 'i' to commodity 'j', a change over cost C_{ij} is incurred.
- It is desired to find a sequence in which to manufacture these commodities.
- This sequence should minimize the sum of changeover costs (the remaining production costs are sequence independent).
- Since the manufacture proceeds cyclically, it is necessary to include the cost of starting the next cycle.
- This is just the change over cost from the last to the first commodity.

- Hence, this problem can be regarded as a traveling sales person problem on an ‘n’ vertex graph with edge
- Cost C_{ij}'s being the change over cost from commodity ‘i’ to commodity ’j’.

In the following discussion we shall, without loss of generality, regard a tour to be a simple path that starts and ends at vertex ‘1’.

- Every tour consists of an edge (i, k) for some k ∈V-{1}. and a path from vertex ‘k’ to vertex ‘1’.
- The path from vertex ‘k’ to vertex ‘1’ goes through each vertex in V-{1,k} exactly once.
- It is easy to see that if the tour is optimal, then the path from ‘k’ to ‘1’ must be a shortest ‘k’ to ‘1’ path going through all vertices in V-{1,k} .
- Hence, the principle of optimality holds.
- Let g(i, S) be the length of a shortest path starting at vertex ‘i’ ,going through all vertices in ‘S’, and terminating at vertex ‘1’.
- The function g(1 ,V-{1}) is the length of an optimal sales person tour.

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

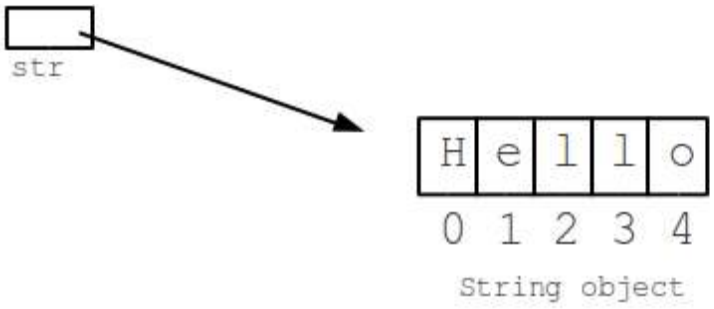
Introduction:

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

Tthe + (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***”

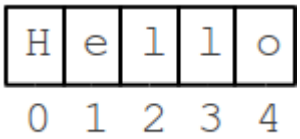
String Length

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

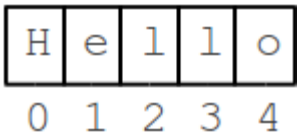
- Index numbers -- 0, 1, 2, ...
- Leftmost char is at index 0
- Last char is at index length-1



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

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.**

1. 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.

Hints:
What if we divide the string into substrings containing only one distinct character with maximal lengths?
Now that you have sub-strings with only one distinct character, Try to come up with a formula that counts the number of its sub-strings.

We can define the **Dynamic Programming** **function** F(N) as the different substrings that end the n-th character. If S[i] == S[i – 1], thus F[i] = F[i – 1] + 1, because we can have (1..i+i) or (i) solutions. Then the overall solution is to add up these numbers from F[0] to F[N-1].

Time and Space complexity.

- The time complexity is O(n) because we are traversing each element in the string at most 2 times.
- The space complexity is O(1) because the space used does not depend on the input length.

Java Program for Count Substrings with Only One Distinct Letter using Dynamic Programming CountSubstrings.java

```
import java.util.*;

class CountSubstrings
{
        /* DP solution
        Time complexity: O(n)
        Space complexity: O(n)
        */

        // aaabbb
        public int countLettersDP(String s)
        {
      int dp[] = new int [s.length()];
      int sum = 1;
                dp[0] = 1;

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

                        sum += dp[i];
      }
                System.out.println(Arrays.toString(dp));
      return sum;
   }

        // Keep track of the count of same letters, and the total sum
        /*
        Time complexity: O(n)
        Space complexity: O(1)
        */

        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().countLettersDP(s));
        }
}
```

2. 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.

Java Program for Valid Word Abbreviation:
ValidWordAbbreviation.java

```
class ValidWordAbbreviation
{
        // apple (i)  ape (j)

 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));
    }
}
```

Input:
internationalization
i12iz4n
Output:
true

3. 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

Dynamic Programming Approach:

- Idea is to look for every same character and save its index.
- Check whether difference between index is less than longest repeating and non-overlapping substring size.
- Here, dp[i][j] stores length of the matching and non-overlapping substrings ending with i'th and j'th characters.
- If the characters at $(i-1)^{th}$ and $(j-1)^{th}$ position matches dp[i-1][j-1] is less than the length of the considered substring (j-1) then maximum value of dp[i][j] and the maximum index i till this point is updated. The length of the longest repeating and non-overlapping substring can be found by the maximum value of dp[i][j] and the substring itself can be found using the length and the ending index which is the finishing index of the suffix.

Complexity:

- Worst case time complexity: **$O(n^2)$**
- Average case time complexity: **$O(n^2)$**
- Best case time complexity: **$O(n^2)$**
- Space complexity: **$O(n^2)$**

- Building a 2-D table requires two for loops hence the time-complexity of this algorithm will be **$O(n^2)$**.Also making a 2-D table would require n^2 space hence the space complexity of this algorithm will also be **$O(n^2)$**.

Java Program for Longest Repeating Substring: LRS.java

```
import java.util.*;

class LRS
{
        /*
        Time complexity: O(n^2)
        Space complexity: O(n^2)
        */
public int longestRepeatingSubstringDP(String s) {
      public int longestRepeatingSubstringDP(String s) {
      int length = s.length();
      int[][] dp = new int[length + 1][length + 1];
      int maxLength = 0;
```

```
        // Use DP to find the longest common substring
        for (int i = 0; i < length; i++) {
            for (int j = i + 1; j < length; j++) {
                // If characters match, extend the length of
                // the common substring
                if (s.charAt(i) == s.charAt(j) ) {
                    dp[i+1][j+1] = dp[i][j] + 1;
                    maxLength = Math.max(maxLength, dp[i+1][j+1]);
                }
            }
        }

        System.out.println("dp " + Arrays.deepToString(dp));
        return maxLength;
    }

        /*
        Time complexity: O(n^3): 2 nested for loops and O(n) for each substring call
        Space complexity: O(n^2): To store substrings
        */

        public int longestRepeatingSubstring(String s)
        {
                int maxLength = 0;
                Set<String> set = new HashSet<>();

                for (int i = 0; i < s.length(); i++)
                {
                        for (int j = i + 1; j <= s.length(); j++)
                        {
                                String substring = s.substring(i, j);

                                if (set.contains(substring))
                                {
                                        maxLength = Math.max(maxLength,
substring.length());
                                }
                                else
                                {
                                        set.add(substring);
                                }
                        }
                }
```

```
            return maxLength;
        }

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

Input: abbaba

Output: 2

4. 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.

Approach:

The dynamic programming paradigm consists of two methods known as the top-down approach and the bottom-up approach. The top-down approach memorizes results in a recursive solution, whereas the bottom-up approach builds a solution from the base case. Hence, for this particular problem, we will formulate a bottom-up solution.

Problem Statement: Create a program to find out the length of longest common subsequence for strings A = "XY", B = "XPYQ".

Explanation:

- The dynamic programming approach cuts down the number of function calls.
- It saves the outcome of each function call; so that it can be reused without the need for duplicate calls in the future.
- The results of each comparison between elements of A and B are maintained in tabular format to remove redundant computations.
- Due to that, the time taken by a dynamic programming approach to solve the LCS problem is equivalent to the time taken to fill the table, that is, O(m*n).
- This complexity is relatively low in comparison to the recursive paradigm. Hence, dynamic programming is considered as an optimal strategy to solve this space optimization problem.

Java Program For Longest Common Sub Sequences: LCSdp.java

```
import java.util.*;

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

    public static int lcs(String s1, String s2)
    {
```

```
		int[] dp = new int[s2.length()+1];
		for(int i = 0; i < s1.length(); i++)
		{
			int prev = dp[0];
			for(int j = 1; j < dp.length; j++)
			{
				int temp = dp[j];
				System.out.println("i " + i + " j " + j + " prev " + prev + " s1 
char " + s1.charAt(i) + " s2 char " + s2.charAt(j-1));

				/*
					If characters match, add 1 to the previous value
					Otherwise, take the maximum of the left and current
				*/
				if(s1.charAt(i) == s2.charAt(j-1))
					dp[j] = prev + 1;
				else
					dp[j] = Math.max(dp[j-1], dp[j]);

				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));
	}
}
```

Input: text1 = "abcde", text2 = "ace"
Output: 3

5. <u>Longest Increasing Subsequence:</u>

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:

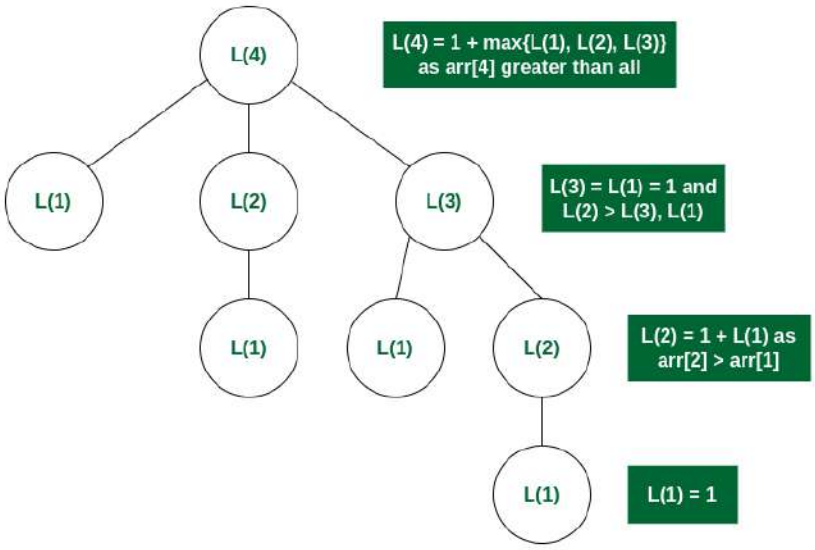
Input: arr[] = {5,4,1,2,3}
Output: Length of LIS = 3
Explanation: The longest increasing subsequence is 1,2,3

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

Consider arr[] = [3, 10, 2, 11]

L(i): Denotes LIS of subarray ending at position 'i'

If we will draw the recursion tree for the above approach then we can easily see in the below image, there are some overlapping subproblems and in which we can easily apply substructure properties. Hence we can say we can store some state and use it in dynamic programming.



Elements of Dynamic Programming Approach

Here the recursion approach is top-down. Hence we can use it in the implementation of our dynamic programming bottom-up. What are the different states or elements that are possible for dynamic programmatic

We have to define problem variables: There is only one parameter on which the state of the problem depends i.e. which is N here, the size of the array.

We have to define size and table structure: For to write the code in a bottom-up manner we have to first define the size and table structure.
Input: arr[] = {3, 10, 2, 11}
LIS[] = {1, 1, 1, 1} (initially)

Iteration-wise simulation (array index is from 1 to n):

1. arr[2] > arr[1] {LIS[2] = max(LIS [2], LIS[1]+1 = 2}
2. arr[3] < arr[1] {No change}
3. arr[3] < arr[2] {No change}
4. arr[4] > arr[1] {LIS[4] = max(LIS [4], LIS[1]+1 = 2}
5. arr[4] > arr[2] {LIS[4] = max(LIS [4], LIS[2]+1 = 3}
6. arr[4] > arr[3] {LIS[4] = max(LIS [4], LIS[3]+1 = 3}

Java Program for Longest Increasing Subsequence Using Dynamic Programming: LIS.java

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

```
class LISdp {
        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++)
                {
                        System.out.println("i " + i + " j " + 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)
   {
                /*
                6
                15 10 5 8 6 9
                */
                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));
        }
}
```

Case=1

Input:
5,4,1,2,3
Output:
3

Case=2
Input:
7,5
Output:
1

Time complexity: O(N^2), Where N is the size of the array.
Space complexity: O(N)