Introduction of Dynamic Programming

Dynamic Programming is the most powerful design technique for solving optimization problems.

Divide & Conquer algorithm partition the problem into disjoint subproblems solve the subproblems recursively and then combine their solution to solve the original problems.

Dynamic Programming is used when the subproblems are not independent, e.g. when they share the same subproblems. In this case, divide and conquer may do more work than necessary, because it solves the same sub problem multiple times.

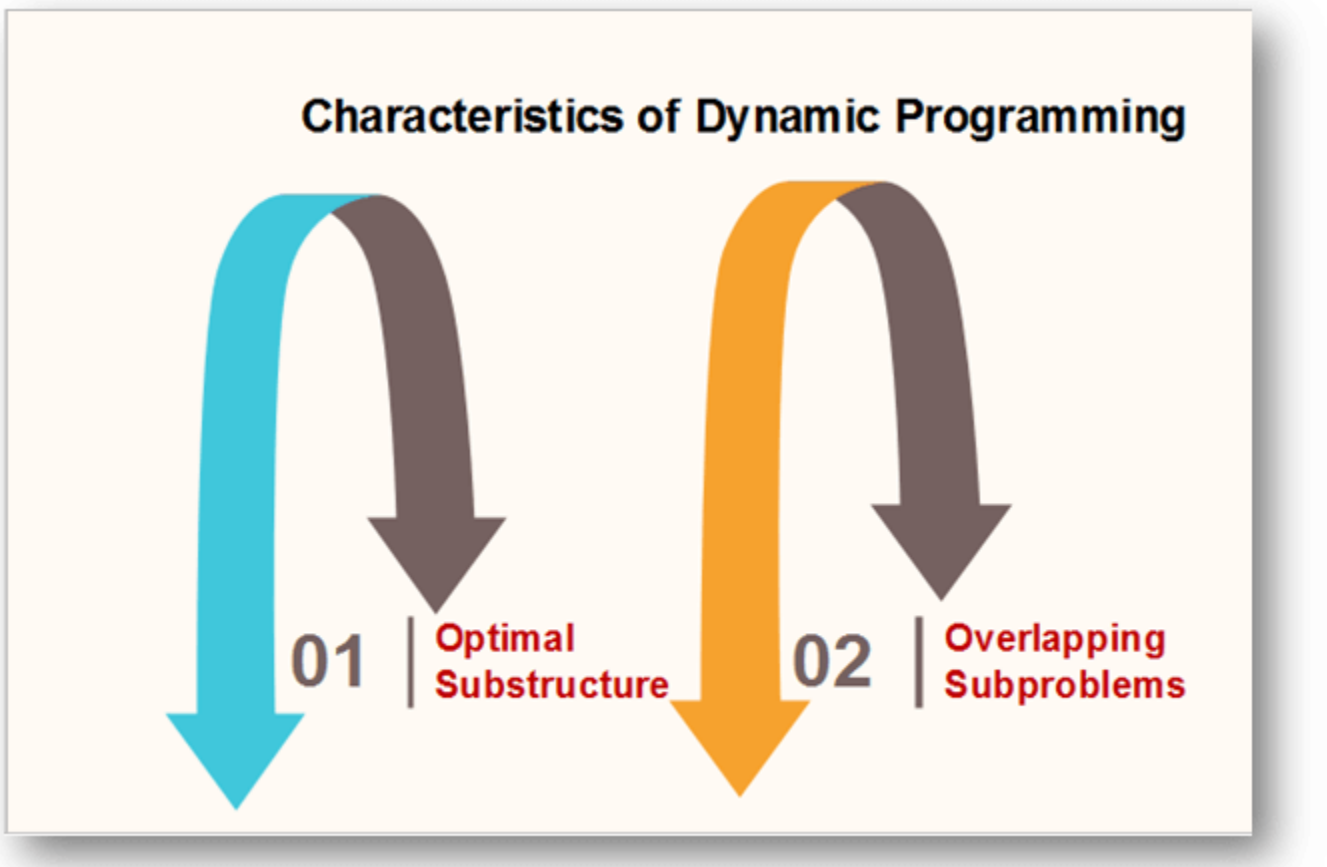
Dynamic Programming solves each subproblems just once and stores the result in a table so that it can be repeatedly retrieved if needed again.

Dynamic Programming is a **Bottom-up approach-** we solve all possible small problems and then combine to obtain solutions for bigger problems.

Dynamic Programming is a paradigm of algorithm design in which an optimization problem is solved by a combination of achieving sub-problem solutions and appearing to the "**principle of optimality**".

Characteristics of Dynamic Programming:

Dynamic Programming works when a problem has the following features:-



- **Optimal Substructure:** If an optimal solution contains optimal sub solutions then a problem exhibits optimal substructure.
- **Overlapping subproblems:** When a recursive algorithm would visit the same subproblems repeatedly, then a problem has overlapping subproblems.

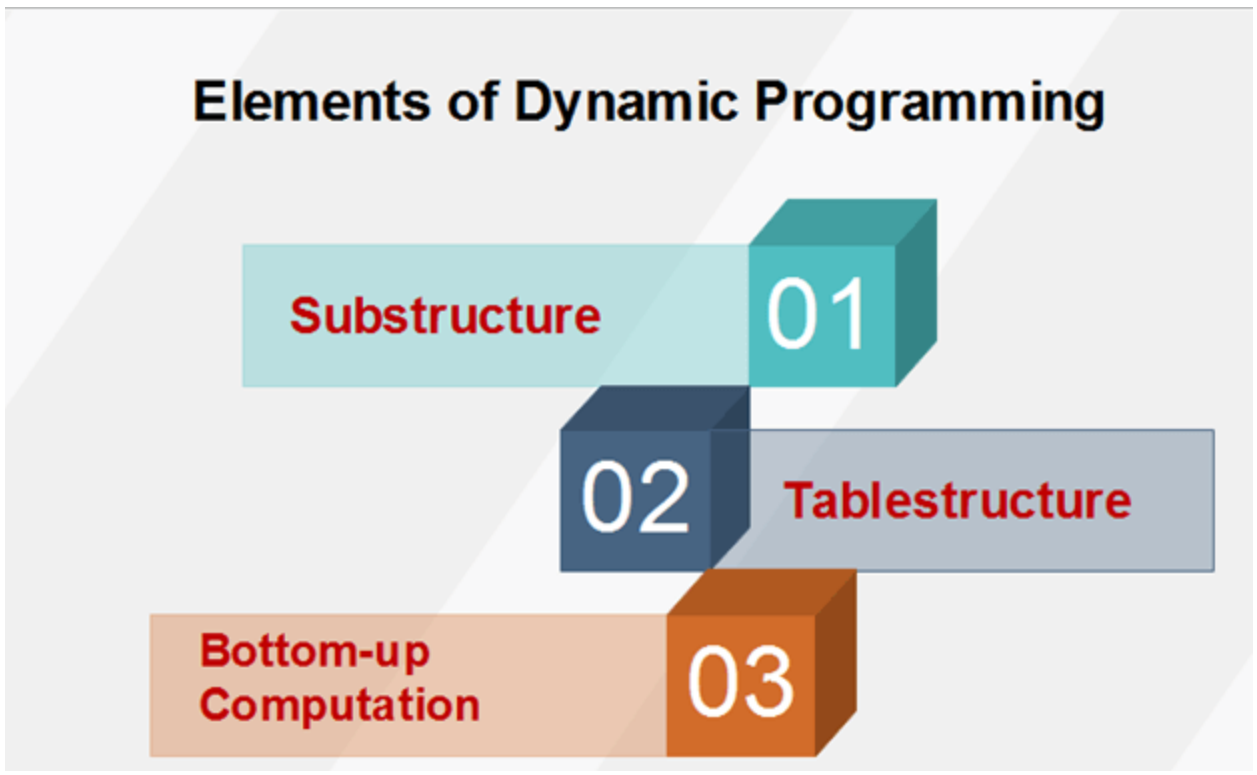
If a problem has optimal substructure, then we can recursively define an optimal solution. If a problem has overlapping subproblems, then we can improve on a recursive implementation by computing each subproblem only once.

If a problem doesn't have optimal substructure, there is no basis for defining a recursive algorithm to find the optimal solutions. If a problem doesn't have overlapping sub problems, we don't have anything to gain by using dynamic programming.

If the space of subproblems is enough (i.e. polynomial in the size of the input), dynamic programming can be much more efficient than recursion.

Elements of Dynamic Programming

There are basically three elements that characterize a dynamic programming algorithm:-

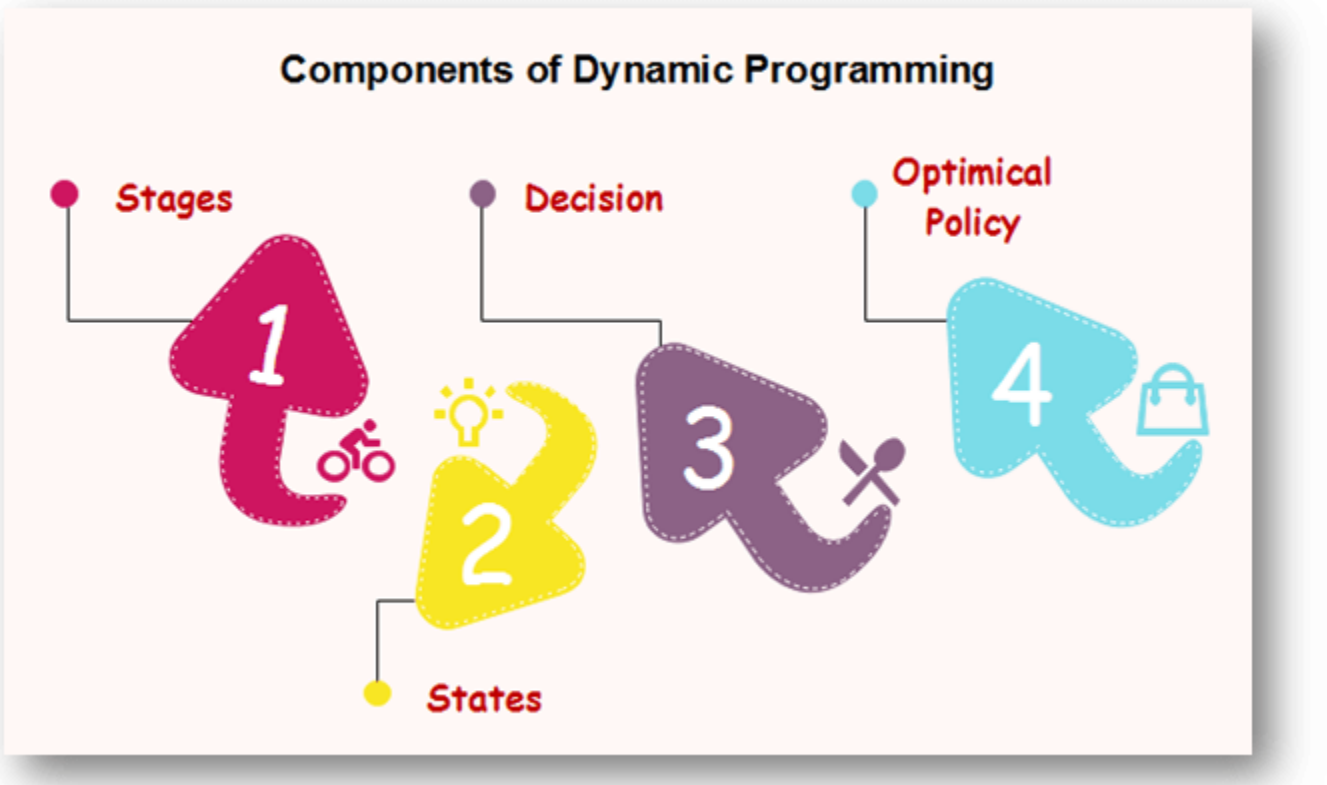


1. **Substructure:** Decompose the given problem into smaller subproblems. Express the solution of the original problem in terms of the solution for smaller problems.
2. **Table Structure:** After solving the sub-problems, store the results to the sub problems in a table. This is done because subproblem solutions are reused many times, and we do not want to repeatedly solve the same problem over and over again.
3. **Bottom-up Computation:** Using table, combine the solution of smaller subproblems to solve larger subproblems and eventually arrives at a solution to complete problem.

Note: Bottom-up means:-

i. Start with smallest subproblems.
ii. Combining their solutions obtain the solution to sub-problems of increasing size.
iii. Until solving at the solution of the original problem.

Components of Dynamic programming



1. **Stages:** The problem can be divided into several subproblems, which are called stages. A stage is a small portion of a given problem. For example, in the shortest path problem, they were defined by the structure of the graph.
2. **States:** Each stage has several states associated with it. The states for the shortest path problem was the node reached.
3. **Decision:** At each stage, there can be multiple choices out of which one of the best decisions should be taken. The decision taken at every stage should be optimal; this is called a stage decision.
4. **Optimal policy:** It is a rule which determines the decision at each stage; a policy is called an optimal policy if it is globally optimal. This is known as Bellman principle of optimality.
5. Given the current state, the optimal choices for each of the remaining states does not depend on the previous states or decisions. In the shortest path problem, it was not necessary to know how we got a node only that we did.

6. There exist a recursive relationship that identify the optimal decisions for stage j, given that stage j+1, has already been solved.
7. The final stage must be solved by itself.

Development of Dynamic Programming Algorithm

It can be broken into four steps:

1. Characterize the structure of an optimal solution.
2. Recursively defined the value of the optimal solution. Like Divide and Conquer, divide the problem into two or more optimal parts recursively. This helps to determine what the solution will look like.
3. Compute the value of the optimal solution from the bottom up (starting with the smallest subproblems)
4. Construct the optimal solution for the entire problem form the computed values of smaller subproblems.

Applications of dynamic programming

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

Differentiate between Divide & Conquer Method vs Dynamic Programming.

Divide & Conquer Method	Dynamic Programming
1.It deals (involves) three steps at each level of recursion: **Divide** the problem into a number of subproblems. **Conquer** the subproblems by solving them recursively. **Combine** the solution to the subproblems into the solution for original subproblems.	**1.**It involves the sequence of four steps: ○ Characterize the structure of optimal solutions. ○ Recursively defines the values of optimal solutions. ○ Compute the value of optimal solutions in a Bottom-up minimum. ○ Construct an Optimal Solution from computed information.
2. It is Recursive.	**2.** It is non Recursive.
3. It does more work on subproblems and hence has more time consumption.	**3.** It solves subproblems only once and then stores in the table.
4. It is a top-down approach.	**4.** It is a Bottom-up approach.
5. In this subproblems are independent of each other.	**5.** In this subproblems are interdependent.
6. For example: Merge Sort & Binary Search etc.	**6. For example:** Matrix Multiplication.

Fibonacci sequence

Fibonacci sequence is the sequence of numbers in which every next item is the total of the previous two items. And each number of the Fibonacci sequence is called Fibonacci number.

Example: 0 ,1,1,2,3,5,8,13,21,..... is a Fibonacci sequence.

The Fibonacci numbers F_nare defined as follows:

$F_0 = 0$

$F_n=1$

$F_n=F_{(n-1)}+ F_{(n-2)}$

FIB (n)

```
1. If (n < 2)
2. then return n
3. else return FIB (n - 1) + FIB (n - 2)
```

Figure: shows four levels of recursion for the call fib (8):

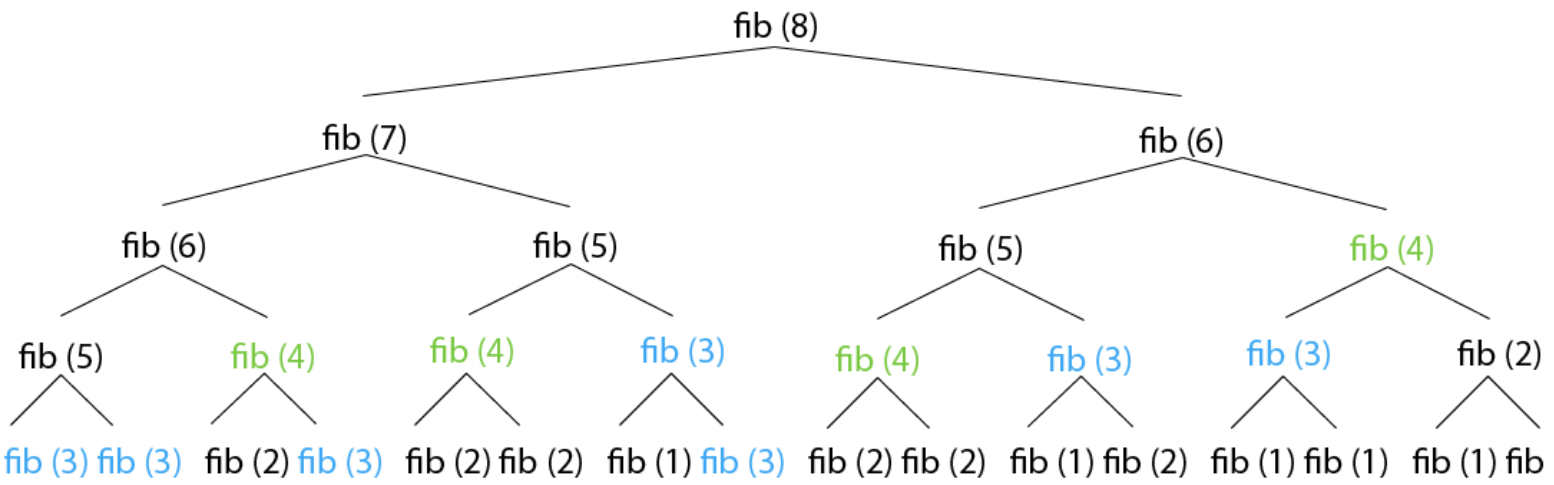


Figure: Recursive calls during computation of Fibonacci number

A single recursive call to fib (n) results in one recursive call to fib (n - 1), two recursive calls to fib (n - 2), three recursive calls to fib (n - 3), five recursive calls to fib (n - 4) and, in general, Fk-1 recursive calls to fib (n - k) We can avoid this unneeded repetition by writing down the conclusion of recursive calls and looking them up again if we need them later. This process is called memorization.

Here is the algorithm with memorization

MEMOFIB (n)

```
1 if (n < 2)
2 then return n
3 if (F[n] is undefined)
4 then F[n] ← MEMOFIB (n - 1) + MEMOFIB (n - 2)
5 return F[n]
```

If we trace through the recursive calls to MEMOFIB, we find that array F [] gets filled from bottom up. I.e., first F [2], then F [3], and so on, up to F[n]. We can replace recursion with a simple for-loop that just fills up the array F [] in that order

ITERFIB (n)

```
1 F [0] ← 0
2 F [1] ← 1
3 for i ← 2 to n
4 do
5 F[i] ← F [i - 1] + F [i - 2]
6 return F[n]
```

This algorithm clearly takes only O (n) time to compute Fn. By contrast, the original recursive algorithm takes O ($\emptyset^{n}$),$\emptyset$ = $\frac{1+\sqrt{5}}{2}$ = 1.618. ITERFIB conclude an exponential speedup over the original recursive algorithm.

Matrix Chain Multiplication

It is a Method under Dynamic Programming in which previous output is taken as input for next.

Here, Chain means one matrix's column is equal to the second matrix's row [always].

In general:

If $A = [a_{ij}]$ is a p x q matrix

$B = [b_{ij}]$ is a q x r matrix

$C = [c_{ij}]$ is a p x r matrix

Then

$$AB = C \text{ if } c_{ij} = \sum_{k=1}^{q} a_{ik}\ b_{kj}$$

Given following matrices {A_1,A_2,A_3,...A_n} and we have to perform the matrix multiplication, which can be accomplished by a series of matrix multiplications

A_1 xA_2 x,A_3 x.....x A_n

Matrix Multiplication operation is **associative** in nature rather commutative. By this, we mean that we have to follow the above matrix order for multiplication but we are free to **parenthesize** the above multiplication depending upon our need.

In general, for 1≤ i≤ p and 1≤ j ≤ r

$$C[i, j] = \sum_{k=1}^{q} A[i, k]B[k, j]$$

It can be observed that the total entries in matrix 'C' is 'pr' as the matrix is of dimension p x r Also each entry takes O (q) times to compute, thus the total time to compute all possible entries for the matrix 'C' which is a multiplication of 'A' and 'B' is proportional to the product of the dimension p q r.

It is also noticed that we can save the number of operations by reordering the parenthesis.

Example1: Let us have 3 matrices, A_1,A_2,A_3 of order (10 x 100), (100 x 5) and (5 x 50) respectively.

Three Matrices can be multiplied in two ways:

1. **A_1,(A_2,A_3):** First multiplying(A_2 and A_3) then multiplying and resultant withA_1.
2. **(A_1,A_2),A_3:** First multiplying(A_1 and A_2) then multiplying and resultant withA_3.

No of Scalar multiplication in Case 1 will be:

1. (100 x 5 x 50) + (10 x 100 x 50) = 25000 + 50000 = 75000

No of Scalar multiplication in Case 2 will be:

1. (100 x 10 x 5) + (10 x 5 x 50) = 5000 + 2500 = 7500

To find the best possible way to calculate the product, we could simply parenthesis the expression in every possible fashion and count each time how many scalar multiplication are required.

Matrix Chain Multiplication Problem can be stated as "find the optimal parenthesization of a chain of matrices to be multiplied such that the number of scalar multiplication is minimized".

Number of ways for parenthesizing the matrices:

There are very large numbers of ways of parenthesizing these matrices. If there are n items, there are (n-1) ways in which the outer most pair of parenthesis can place.

(A_1) (A_2,A_3,A_4,.....A_n)
Or (A_1,A_2) (A_3,A_4 .....A_n)
Or (A_1,A_2,A_3) (A_4 .....A_n)
.....

Or(A_1,A_2,A_3.....A_{n-1}) (A_n)

It can be observed that after splitting the kth matrices, we are left with two parenthesized sequence of matrices: one consist 'k' matrices and another consist 'n-k' matrices.

Now there are 'L' ways of parenthesizing the left sublist and 'R' ways of parenthesizing the right sublist then the Total will be L.R:

$$p(n) = \begin{cases} 1 & \text{if } n = 1 \\ \sum_{k=1}^{n-1} p(k)p(n-k) & \text{if } n \geq 2 \end{cases}$$

Also p (n) = c (n-1) where c (n) is the nth **Catalon number**

$$c(n) = \frac{1}{n+1}\binom{2n}{n}$$

On applying Stirling's formula we have

$c(n) = \Omega\left(\frac{4^n}{n^{1.5}}\right)$

Which shows that 4^n grows faster, as it is an exponential function, then $n^{1.5}$.

Development of Dynamic Programming Algorithm

1. Characterize the structure of an optimal solution.
2. Define the value of an optimal solution recursively.
3. Compute the value of an optimal solution in a bottom-up fashion.
4. Construct the optimal solution from the computed information.

Dynamic Programming Approach

Let $A_{i,j}$ be the result of multiplying matrices i through j. It can be seen that the dimension of $A_{i,j}$ is p_{i-1} x p_j matrix.

Dynamic Programming solution involves breaking up the problems into subproblems whose solution can be combined to solve the global problem.

At the greatest level of parenthesization, we multiply two matrices

$A_{1.....n}=A_{1....k}$ x $A_{k+1....n})$

Thus we are left with two questions:

- How to split the sequence of matrices?
- How to parenthesize the subsequence $A_{1.....k}$ and$A_{k+1.....n}$?

One possible answer to the first question for finding the best value of 'k' is to check all possible choices of 'k' and consider the best among them. But that it can be observed that checking all possibilities will lead to an exponential number of total possibilities. It can also be noticed that there exists only O (n^2) different sequence of matrices, in this way do not reach the exponential growth.

Step1: Structure of an optimal parenthesization: Our first step in the dynamic paradigm is to find the optimal substructure and then use it to construct an optimal solution to the problem from an optimal solution to subproblems.

Let $A_{i....j}$ where i≤ j denotes the matrix that results from evaluating the product

A_i A_{i+1}....A_j.

If i < j then any parenthesization of the product A_i A_{i+1} .....A_j must split that the product between A_k and A_{k+1} for some integer k in the range i ≤ k ≤ j. That is for some value of k, we first compute the matrices $A_{i.....k}$ & $A_{k+1....j}$ and then multiply them together to produce the final product $A_{i....j}$. The cost of computing $A_{i....k}$ plus the cost of computing $A_{k+1....j}$ plus the cost of multiplying them together is the cost of parenthesization.

Step 2: A Recursive Solution: Let m [i, j] be the minimum number of scalar multiplication needed to compute the matrix$A_{i....j}$.

If i=j the chain consist of just one matrix $A_{i....i}=A_i$ so no scalar multiplication are necessary to compute the product. Thus m [i, j] = 0 for i= 1, 2, 3....n.

If i<j we assume that to optimally parenthesize the product we split it between A_k and A_{k+1} where i≤ k ≤j. Then m [i,j] equals the minimum cost for computing the subproducts $A_{i....k}$ and $A_{k+1....j}$+ cost of multiplying them together. We know A_i has dimension p_{i-1} x p_i, so computing the product $A_{i....k}$ and $A_{k+1....j}$takes p_{i-1} p_k p_j scalar multiplication, we obtain

$m\ [i,j] = m\ [i,\ k] + m\ [k + 1,\ j] + p_{i-1}\ \ p_k\ p_j$

There are only (j-1) possible values for 'k' namely k = i, i+1.....j-1. Since the optimal parenthesization must use one of these values for 'k' we need only check them all to find the best.

So the minimum cost of parenthesizing the product A_i A_{i+1}.....A_j becomes

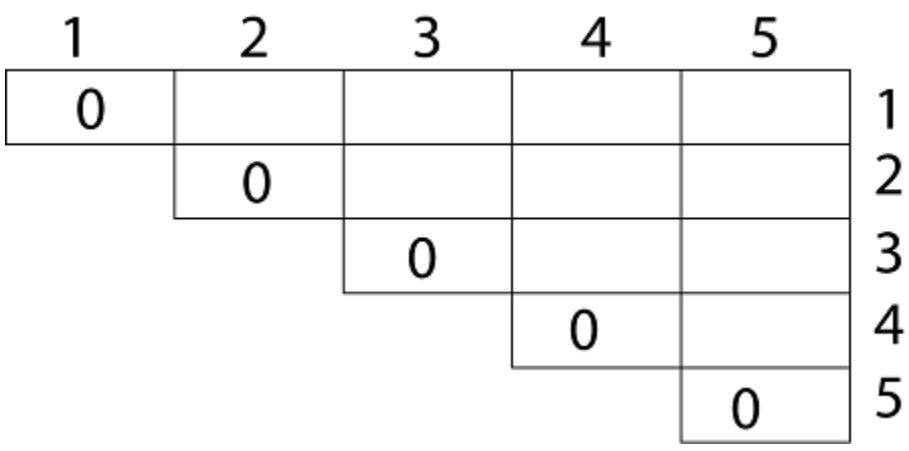
$$m\ [i,j] = \begin{cases} 0 & \text{if } i = j \\ \min\limits_{i \le k < j}\{m\ [i,k] + m\ [k+1,j] + p_{i-1}\, p_k p_j\} & \text{if } i < j \end{cases}$$

To construct an optimal solution, let us define s [i,j] to be the value of 'k' at which we can split the product $A_i A_{i+1} \ldots A_j$ To obtain an optimal parenthesization i.e. s [i, j] = k such that

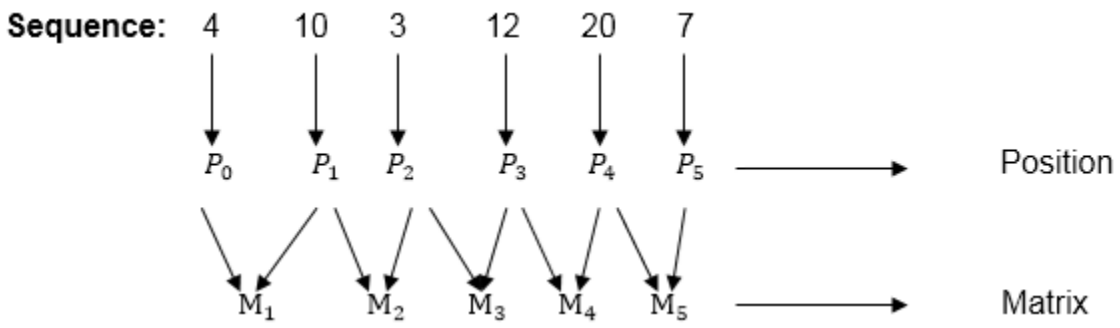
$$m[i,j] = m[i,k] + m[k+1,j] + p_{i-1} p_k p_j$$

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.



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



Here P_0 to P_5 are Position and M_1 to M_5 are matrix of size (p_i to p_{i-1})

On the basis of sequence, we make a formula

For M_i ⟶ p [i] as column

p [i-1] as row

In Dynamic Programming, initialization of every method done by '0'.So we initialize it by '0'.It will sort out diagonally.

We have to sort out all the combination but the minimum output combination is taken into consideration.

Calculation of Product of 2 matrices:

1. $m(1,2) = m_1 \times m_2$
   $= 4 \times 10 \times 10 \times 3$
   $= 4 \times 10 \times 3 = 120$

2. $m(2,3) = m_2 \times m_3$
   $= 10 \times 3 \times 3 \times 12$
   $= 10 \times 3 \times 12 = 360$

3. $m(3,4) = m_3 \times m_4$
   $= 3 \times 12 \times 12 \times 20$
   $= 3 \times 12 \times 20 = 720$

4. $m(4,5) = m_4 \times m_5$
   $= 12 \times 20 \times 20 \times 7$
   $= 12 \times 20 \times 7 = 1680$

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: (M_1 x M_2) + M_3, M_1+ (M_2x M_3)
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 (M_1 x M_2) + M_3 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: (M_2x M_3)+M_4, M_2+(M_3 x M_4)
2. After solving both cases we choose the case in which minimum output is there.

$$M[2,4] = \min \begin{cases} M[2,3] + M[4,4] + p_1 p_3 p_4 = 360 + 0 + 10.12.20 = 2760 \\ M[2,2] + M[3,4] + p_1 p_2 p_4 = 0 + 720 + 10.3.20 = 1320 \end{cases}$$

M [2, 4] = 1320

As Comparing both output **1320** is minimum in both cases so we insert **1320** in table and M_2+(M_3 x M_4) 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: (M_3 x M_4) + M_5, M_3+ (M_4xM_5)
2. After solving both cases we choose the case in which minimum output is there.

$$M[3,5] = \min \begin{cases} M[3,4] + M[5,5] + p_2 p_4 p_5 = 720 + 0 + 3.20.7 = 1140 \\ M[3,3] + M[4,5] + p_2 p_3 p_5 = 0 + 1680 + 3.12.7 = 1932 \end{cases}$$

`M [3, 5] = 1140`

As Comparing both output **1140** is minimum in both cases so we insert **1140** in table and (M_3 x M_4) + M_5this 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. (M_1 x M_2 x M_3) M_4
2. M_1 x(M_2 x M_3 x M_4)
3. (M_1 xM_2) x (M_3 x M_4)

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 (M_1 xM_2) x (M_3 x M_4) 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. (M_2 x M_3 x M_4)x M_5
2. M_2 x(M_3 x M_4 x M_5)
3. (M_2 x M_3)x (M_4 x M_5)

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 M_2 x(M_3 x M_4 xM_5)combination is taken out in output making.

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

→

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

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. (M_1 x M_2 xM_3 x M_4)x M_5
2. M_1 x(M_2 xM_3 x M_4 xM_5)
3. (M_1 x M_2 xM_3)x M_4 xM_5
4. M_1 x M_2x(M_3 x M_4 xM_5)

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 M_1 x M_2 x(M_3 x M_4 x M_5)combination is taken out in output making.

Final Output is:

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

→

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

Step 3: Computing Optimal Costs: let us assume that matrix A_i has dimension p_{i-1}x p_i for i=1, 2, 3....n. The input is a sequence (p_0,p_1,.....p_n) where length [p] = n+1. The procedure uses an auxiliary table m [1....n, 1.....n] for storing m [i, j] costs an auxiliary table s [1.....n, 1.....n] that record which index of k achieved the optimal costs in computing m [i, j].

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

Algorithm of Matrix Chain Multiplication

MATRIX-CHAIN-ORDER (p)

```
1. n   length[p]-1
2. for i ← 1 to n
3. do m [i, i] ← 0
4. for l ← 2 to n    // l is the chain length
5. do for i ← 1 to n-l + 1
6. do j ← i+ l -1
7. m[i,j] ←  ∞
8. for k  ← i to j-1
9. do q  ← m [i, k] + m [k + 1, j] + p_{i-1} p_k p_j
10. If q < m [i,j]
11. then m [i,j] ← q
12. s [i,j] ← k
13. return m and s.
```

We will use table s to construct an optimal solution.

Step 1: Constructing an Optimal Solution:

PRINT-OPTIMAL-PARENS (s, i, j)

```
1. if i=j
2. then print "A"
3. else print "("
4. PRINT-OPTIMAL-PARENS (s, i, s [i, j])
5. PRINT-OPTIMAL-PARENS (s, s [i, j] + 1, j)
6. print ")"
```

Analysis: There are three nested loops. Each loop executes a maximum n times.

1. l, length, O (n) iterations.
2. i, start, O (n) iterations.
3. k, split point, O (n) iterations

Body of loop constant complexity

Total Complexity is: O (n^3)

Algorithm with Explained Example

Question: P [7, 1, 5, 4, 2}

Solution: Here, P is the array of a dimension of matrices.

So here we will have 4 matrices:

A_{17x1} A_{21x5} A_{35x4} A_{44x2}

i.e.

First Matrix A_1 have dimension 7 x 1

Second Matrix A_2 have dimension 1 x 5

Third Matrix A_3 have dimension 5 x 4
Fourth Matrix A_4 have dimension 4 x 2

Let say,
From P = {7, 1, 5, 4, 2} - (Given)
And P is the Position
p_0 = 7, p_1 =1, p_2 = 5, p_3 = 4, p_4=2.
Length of array P = number of elements in P
∴length (p)= 5
From step 3
Follow the steps in Algorithm in Sequence
According to Step 1 of Algorithm Matrix-Chain-Order

Step 1:

n ← length [p]-1
Where n is the total number of elements
And length [p] = 5
∴ n = 5 - 1 = 4

n = 4

Now we construct two tables m and s.
Table m has dimension [1.....n, 1.......n]
Table s has dimension [1.....n-1, 2.......n]

	1	2	3	4
1	0	35 ∞	48 ∞	42
2		0	20 ∞	28 ∞
3			0	
4				0

m-Table
[1....n, 1.....n]

	2	3	4
1	1	1	1
2		2	3
3			3

n-Table
[1....n-1, 2.....n]

Now, according to step 2 of Algorithm

1. **for** i ← 1 to n
2. **this** means: **for** i ← 1 to 4 (because n =4)
3. **for** i=1
4. m [i, i]=0
5. m [1, 1]=0
6. Similarly **for** i = 2, 3, 4
7. m [2, 2] = m [3,3] = m [4,4] = 0
8. i.e. fill all the diagonal entries "0" in the table m
9. Now,
10. l ← 2 to n
11. l ← 2 to 4 (because n =4)

Case 1:

1. When l - 2

```
for (i ← 1 to n - l + 1)
```

i ← 1 to 4 - 2 + 1

i ← 1 to 3

When i = 1

do j ← i + l - 1

j ← 1 + 2 - 1

j ← 2

i.e. j = 2

Now, m [i, j] ← ∞

i.e. m [1,2] ← ∞

Put ∞ in m [1, 2] table

for k ← i to j-1

k ← 1 to 2 - 1

k ← 1 to 1

k = 1

Now q ← m [i, k] + m [k + 1, j] + p_{i-1} p_k p_j

for l = 2

i = 1

j =2

k = 1

q ← m [1,1] + m [2,2] + p_0x p_1x p_2

and m [1,1] = 0

for i ← 1 to 4

∴ q ← 0 + 0 + 7 x 1 x 5

q ← 35

We have m [i, j] = m [1, 2] = ∞

Comparing q with m [1, 2]

q < m [i, j]

i.e. **35 < m [1, 2]**

35 < ∞

True

then, **m [1, 2] ← 35 (∴ m [i,j] ← q)**

s [1, 2] ← k

and the value of **k = 1**

s [1,2] ← 1

Insert "1" at dimension s [1, 2] in table s. And 35 at m [1, 2]

2. l remains 2

L = 2

i ← 1 to n - l + 1

i ← 1 to 4 - 2 + 1

i ← 1 to 3

for i = 1 done before

Now value of i becomes 2

i = 2

j ← i + l - 1

j ← 2 + 2 - 1

j ← 3

j = 3

m [i , j] ← ∞

i.e. **m [2,3] ← ∞**

Initially insert ∞ at m [2, 3]

Now, for k ← i to j - 1

k ← 2 to 3 - 1

k ← 2 to 2

i.e. k =2

Now, q ← m [i, k] + m [k + 1, j] + p_{i-1} p_k p_j

For l =2

i = 2

j = 3

k = 2

q ← m [2, 2] + m [3, 3] + p_1x p_2 x p_3

q ← 0 + 0 + 1 x 5 x 4

q ← 20

Compare q with m [i ,j]

If q < m [i,j]

i.e. 20 < m [2, 3]

20 < ∞

True

Then m [i,j] ← q

m [2, 3] ← 20

and s [2, 3] ← k

and k = 2

s [2,3] ← 2

3. Now i become 3

i = 3

l = 2

j ← i + l - 1

j ← 3 + 2 - 1

j ← 4

j = 4

Now, m [i, j] ← ∞

m [3,4] ← ∞

Insert ∞ at m [3, 4]

for k ← i to j - 1

k ← 3 to 4 - 1

k ← 3 to 3

i.e. k = 3

Now, q ← m [i, k] + m [k + 1, j] + p_{i-1} p_k p_j

i = 3

l = 2

j = 4

k = 3

q ← m [3, 3] + m [4,4] + p_2 x p_3 x p_4

q ← 0 + 0 + 5 x 2 x 4

q 40

Compare q with m [i, j]

If q < m [i, j]

40 < m [3, 4]

40 < ∞

True

Then, m [i,j] ← q

m [3,4] ← 40

and s [3,4] ← k

s [3,4] ← 3

Case 2: l becomes 3

L = 3

for i = 1 to n - l + 1

i = 1 to 4 - 3 + 1

i = 1 to 2

When i = 1

j ← i + l - 1

j ← 1 + 3 - 1

j ← 3

j = 3

Now, m [i,j] ← ∞

m [1, 3] ← ∞

for k ← i to j - 1

k ← 1 to 3 - 1

k ← 1 to 2

Now we compare the value for both k=1 and k = 2. The minimum of two will be placed in m [i,j] or s [i,j] respectively.

(A) When k = 1	**(B)** When k =2
L =3,i=1,j=3	L =3,i=1,j=3
q ⟵ m [i, k] + m [k + 1, j] + p_{i-1} p_k p_j	q ⟵ m [i, k] + m [k + 1, j] + p_{i-1} p_k p_j
q ⟵ m [1,1] + m [2,3] + p_0x p_1x p_3	q ⟵ m [1,2] + m [3,3] +p_0x p_2x p_3
q ⟵ 0 + 20 + 7 x 1x 4	q ⟵ 35 + 0 + 7 x 5 x 3
q ⟵ 48	**q ⟵ 140**

Now from above

Value of q become minimum for **k=1**

∴ m [i,j] ← q

m [1,3] ← 48

Also m [i,j] > q

i.e. 48 < ∞

∴ m [i , j] ← q

m [i, j] ← 48

and s [i,j] ← k

i.e. m [1,3] ← 48

s [1, 3] ← 1

Now i become 2

i = 2

l = 3

then j ← i + l -1

j ← 2 + 3 - 1

j ← 4

j = 4

so m [i,j] ← ∞

m [2,4] ← ∞

Insert initially ∞ at m [2, 4]

for k ← i to j-1

k ← 2 to 4 - 1

k ← 2 to 3

Here, also find the minimum value of m [i,j] for two values of k = 2 and k =3

(A) When k =2	**(B)** When k = 3
i=2,l=3, j=4	i=2,l=3, j=4
q ⟵ m [i, k] + m [k + 1, j] + $p_{i-1}\ p_k\ p_j$	q ⟵ m [i, k] + m [k + 1, j] + $p_{i-1}\ p_k\ p_j$
q ⟵ m [2,2] + m [3,4] + $p_1 x\ p_2 x p_4$	q ⟵ m [2,3] + m [4,4] +$p_1 x p_3 x\ p_4$
q ⟵ 0 + 40 +7 x 5 x 2	q ⟵ 20 + 0 + 1 x 4 x 2
q ⟵ 110	**q ⟵ 28**

1. But 28 < ∞
2. So m [i,j] ← q
3. And q ← 28
4. m [2, 4] ← 28
5. and s [2, 4] ← 3
6. e. It means in s table at s [2,4] insert 3 and at m [2,4] insert 28.

Case 3: l becomes 4

L = 4

For i ← 1 to n-l + 1

i ← 1 to 4 - 4 + 1

i ← 1

i = 1

do j ← i + l - 1

j ← 1 + 4 - 1

j ← 4

j = 4

Now m [i,j] ← ∞

m [1,4] ← ∞

for k ← i to j -1

k ← 1 to 4 - 1

k ← 1 to 3

When k = 1

q ← m [i, k] + m [k + 1, j] + $p_{i-1}\ p_k\ p_j$

q ← m [1,1] + m [2,4] + $p_0 x p_4 x\ p_1$

q ← 0 + 28 + 7 x 2 x 1

q ← 42

Compare q and m [i, j]

m [i,j] was ∞
i.e. m [1,4]
if q < m [1,4]
   42< ∞
    True
Then m [i,j] ← q
        m [1,4] ← 42
and s [1,4] 1   ? k =1
When k = 2
       L = 4, i=1, j = 4
q ← m [i, k] + m [k + 1, j] + p_{i-1} p_k p_j
q ← m [1, 2] + m [3,4] + p_0 xp_2 xp_4
q ← 35 + 40 + 7 x 5 x 2
q ← 145
Compare q and m [i,j]
Now m [i, j]
      i.e. m [1,4] contains 42.
So if q < m [1, 4]
But 145 less than or not equal to m [1, 4]
    So 145 less than or not equal to 42.
So no change occurs.
When k = 3
       l = 4
       i = 1
       j = 4
q ← m [i, k] + m [k + 1, j] + p_{i-1} p_k p_j
q ← m [1, 3] + m [4,4] + p_0 xp_3 x p_4
q ← 48 + 0 + 7 x 4 x 2
q ← 114
Again q less than or not equal to m [i, j]
       i.e. 114 less than or not equal to m [1, 4]
            114 less than or not equal to 42

So no change occurs. So the value of m [1, 4] remains 42. And value of s [1, 4] = 1

Now we will make use of only s table to get an optimal solution.

Use of step 4 Algorithm

Initial call of step 4 is (s, 1, n)

Where i=1

j = n and n= 4

i≠ j

else part

print "("

Print-optimal-Parens (s, i, s [i, j])

here i==j yes

print "A_1"

Print-optimal-Parens (s, 1, 4)

Print-optimal-Parens (s, s [i,j] + 1,j]

Print-optimal-Parens (s, 2, 4)

print "("

Print-optimal (s, 2, 3)

Print ")"

Print-optimal (s, 4, 4)

Print ")"

Starting from the beginning we are parenthesizing Matrix Chain Multiplication:

$(A_1)\ ((A_2\ A_3)A_4))$

Longest Common Sequence (LCS)

A subsequence of a given sequence is just the given sequence with some elements left out.

Given two sequences X and Y, we say that the sequence Z is a common sequence of X and Y if Z is a subsequence of both X and Y.

In the longest common subsequence problem, we are given two sequences $X = (x_1\ x_2....x_m)$ and $Y = (y_1\ y_2\ y_n)$ and wish to find a maximum length common subsequence of X and Y. LCS Problem can be solved using dynamic programming.

Characteristics of Longest Common Sequence

A brute-force approach we find all the subsequences of X and check each subsequence to see if it is also a subsequence of Y, this approach requires exponential time making it impractical for the long sequence.

Given a sequence $X = (x_1\ x_2.....x_m)$ we define the ith prefix of X for i=0, 1, and 2...m as $X_i = (x_1\ x_2.....x_i)$. For example: if X = (A, B, C, B, C, A, B, C) then X_4= (A, B, C, B)

Optimal Substructure of an LCS: Let $X = (x_1\ x_2....x_m)$ and $Y = (y_1\ y_2.....)\ y_n)$ be the sequences and let $Z = (z_1\ z_2.....z_k)$ be any LCS of X and Y.

- If $x_m = y_n$, then $z_k=x\_m=y_n$ and Z_{k-1} is an LCS of $X_{m-1}$and Y_{n-1}
- If $x_m \neq y_n$, then $z_k \neq x_m$ implies that Z is an LCS of $X_{m-1}$and Y.
- If $x_m \neq y_n$, then $z_k \neq y_n$ implies that Z is an LCS of X and Y_{n-1}

Step 2: Recursive Solution: LCS has overlapping subproblems property because to find LCS of X and Y, we may need to find the LCS of X_{m-1} and Y_{n-1}. If $x_m \neq y_n$, then we must solve two subproblems finding an LCS of X and Y_{n-1}.Whenever of these LCS's longer is an LCS of x and y. But each of these subproblems has the subproblems of finding the LCS of X_{m-1} and Y_{n-1}.

Let c [i,j] be the length of LCS of the sequence $X_i$and Y_j.If either i=0 and j =0, one of the sequences has length 0, so the LCS has length 0. The optimal substructure of the LCS problem given the recurrence formula

$$c\ [i,\ j] = \begin{cases} 0 & \text{if } i = 0 \quad \text{or} \quad j = 0 \\ c\ [i-1, j-1]1 & \text{if } i, j > 0 \text{ and } x_i = y_j \\ \max(c[i, j-1], c[i-1, j]) & \text{if } i, j > 0 \text{ and } x_i \neq y_j \end{cases}$$

Step3: Computing the length of an LCS: let two sequences $X = (x_1\ x_2.....x_m)$ and $Y = (y_1\ y_2..... y_n)$ as inputs. It stores the c [i,j] values in the table c [0.....m,0.....n].Table b [1.....m, 1.....n] is maintained which help us to construct an optimal solution. c [m, n] contains the length of an LCS of X,Y.

Algorithm of Longest Common Sequence

LCS-LENGTH (X, Y)

```
1. m ← length [X]
2. n ← length [Y]
3. for i ← 1 to m
4. do c [i,0] ← 0
5. for j ← 0 to m
6. do c [0,j] ← 0
7. for i ← 1 to m
8. do for j ← 1 to n
9. do if xi= yj
10. then c [i,j] ← c [i-1,j-1] + 1
11. b [i,j] ← "↖"
12. else if c[i-1,j] ≥ c[i,j-1]
13. then c [i,j] ← c [i-1,j]
14. b [i,j] ← "↑"
15. else c [i,j] ← c [i,j-1]
16. b [i,j] ← "← "
17. return c and b.
```

Example of Longest Common Sequence

Example: Given two sequences X [1...m] and Y [1.....n]. Find the longest common subsequences to both.

x: A	B	C	B	D	A	B
y: B	D	C	A	B	A

here X = (A,B,C,B,D,A,B) and Y = (B,D,C,A,B,A)

m = length [X] and n = length [Y]

m = 7 and n = 6

Here x_1= x [1] = A y_1= y [1] = B

x_2= B y_2= D

x_3= C y_3= C

x_4= B y_4= A

x_5= D y_5= B

x_6= A y_6= A

x_7= B

Now fill the values of c [i, j] in m x n table

Initially, for i=1 to 7 c [i, 0] = 0

For j = 0 to 6 c [0, j] = 0

That is:

i	j	0	1	2	3	4	5	6
		y_j	B	D	C	A	B	A
0	X_i	0	0	0	0	0	0	0
1	A	0						
2	B	0						
3	C	0						
4	B	0						
5	D	0						
6	A	0						
7	B	0						

Now for i=1 and j = 1

x_1 and y_1 we get $x_1 \neq y_1$ i.e. A ≠ B

And c [i-1,j] = c [0, 1] = 0

c [i, j-1] = c [1,0] = 0

That is, c [i-1,j]= c [i, j-1] so c [1, 1] = 0 and b [1, 1] = ' ↑ '

Now for i=1 and j = 2

x_1 and y_2 we get $x_1 \neq y_2$ i.e. A ≠ D

c [i-1,j] = c [0, 2] = 0

c [i, j-1] = c [1,1] = 0

That is, c [i-1,j]= c [i, j-1] and c [1, 2] = 0 b [1, 2] = ' ↑ '

Now for i=1 and j = 3

x_1 and y_3 we get $x_1 \neq y_3$ i.e. A ≠ C

c [i-1,j] = c [0, 3] = 0

c [i, j-1] = c [1,2] = 0

so c [1,3] = 0 b [1,3] = ' ↑ '

Now for i=1 and j = 4

x_1 and y_4 we get. $x_1=y_4$ i.e A = A

c [1,4] = c [1-1,4-1] + 1

= c [0, 3] + 1

= 0 + 1 = 1

c [1,4] = 1

b [1,4] = ' ↖ '

Now for i=1 and j = 5

x_1 and y_5 we get $x_1 \neq y_5$

c [i-1,j] = c [0, 5] = 0

c [i, j-1] = c [1,4] = 1

Thus c [i, j-1] > c [i-1,j] i.e. c [1, 5] = c [i, j-1] = 1. So b [1, 5] = '←'

Now for i=1 and j = 6

x_1 and y_6 we get $x_1=y_6$

c [1, 6] = c [1-1,6-1] + 1

= c [0, 5] + 1 = 0 + 1 = 1

c [1,6] = 1

b [1,6] = ' ↖ '

i		j	0	1	2	3	4	5	6
			y_i	B	D	C	A	B	A
0	X_i		0	0	0	0	0	0	0
1	A		0	↑ 0	↑ 0	↑ 0	↖ 1	← 1	↖ 1
2	B		0						
3	C		0						
4	B		0						
5	D		0						
6	A		0						
7	B		0						

Now for i=2 and j = 1

We get x_2 and y_1 B = B i.e. $x_2= y_1$

c [2,1] = c [2-1,1-1] + 1

= c [1, 0] + 1

= 0 + 1 = 1

c [2, 1] = 1 and b [2, 1] = ' ↖ '

Similarly, we fill the all values of c [i, j] and we get

i		j	0	1	2	3	4	5	6
			y_i	B	D	C	A	B	A
0	X_i		0	0	0	0	0	0	0
1	A		0	↑ 0	↑ 0	↑ 0	↖ 1	← 1	↖ 1
2	B		0	↖ 1	← 1	← 1	↑ 1	↖ 2	← 2
3	C		0	↑ 1	↑ 1	↖ 2	← 2	↑ 2	↑ 2
4	B		0	↖ 1	↑ 1	↑ 2	↑ 2	↖ 3	← 3
5	D		0	↑ 1	↖ 2	↑ 2	↑ 2	↑ 3	↑ 3
6	A		0	↑ 1	↑ 2	↑ 2	↖ 3	↑ 3	↖ 4
7	B		0	↖ 1	↑ 2	↑ 2	↑ 3	↖ 4	↑ 4

Step 4: Constructing an LCS: The initial call is PRINT-LCS (b, X, X.length, Y.length)

PRINT-LCS (b, x, i, j)

```
1. if i=0 or j=0
2. then return
3. if b [i,j] = ' ↖ '
4. then PRINT-LCS (b,x,i-1,j-1)
5. print x_i
6. else if b [i,j] = ' ↑ '
7. then PRINT-LCS (b,X,i-1,j)
8. else PRINT-LCS (b,X,i,j-1)
```

Example: Determine the LCS of (1,0,0,1,0,1,0,1) and (0,1,0,1,1,0,1,1,0).

Solution: let X = (1,0,0,1,0,1,0,1) and Y = (0,1,0,1,1,0,1,1,0).

$$c[i,j] = \begin{cases} 0 & \text{if } i = 0 \quad \text{or} \quad j = 0 \\ c[i-1, j-1] + 1 & \text{if } i, j > 0 \quad \text{and} \quad x_i = y_i \\ \max(c[i, j-1], c[i-1, j]) & \text{if } i, j > 0 \text{ and } x_i \neq y_i \end{cases}$$

We are looking for c [8, 9]. The following table is built.

			x =(1,0,0,1,0,1,0,1)					y =(0,1,0,1,1,0,1,1,0)			
i	j	0	1	2	3	4	5	6	7	8	9
		y_j	0	1	0	1	1	0	1	1	0
0	X_i	0	0	0	0	0	0	0	0	0	0
1	1	0	↑ 0	↖ (1)	↑ 1	↖ 1	← 1	↖ 1	↖ 0	↖ 1	← 1
2	0	0	↖ 1	↑ 1	↖ (2)	← 2	← 2	↖ 2	← 2	← 2	↖ 2
3	0	0	↖ 1	↑ 1	↖ 2	↑ 2	↑ 2	↖ (3)	← 3	← 3	↖ 3
4	1	0	↑ 1	↖ 2	↑ 2	↖ 3	↖ 3	↑ 3	↖ (4)	↖ 4	← 4
5	0	0	↖ 1	↑ 2	↖ 3	↑ 3	↑ 3	↖ 4	↑ 4	↑ 4	↖ 5
6	1	0	↑ 1	↖ 2	↑ 3	↖ 4	↖ 4	↑ 4	↖ 5	↖ (5)	↑ 5
7	0	0	↖ 1	↑ 2	↖ 3	↑ 4	↑ 4	↖ 5	↑ 5	↑ 5	↖ (6)
8	1	0	↑ 1	↖ 2	↑ 3	↖ 4	↖ 5	↑ 5	↖ 6	↖ 6	↑ 6

From the table we can deduct that LCS = 6. There are several such sequences, for instance (1,0,0,1,1,0) (0,1,0,1,0,1) and (0,0,1,1,0,1)

0/1 Knapsack Problem: Dynamic Programming Approach:

Knapsack Problem:

Knapsack is basically means bag. A bag of given capacity.

We want to pack n items in your luggage.

- The ith item is worth v_i dollars and weight w_i pounds.
- Take as valuable a load as possible, but cannot exceed W pounds.
- v_i w_i W are integers.

1. W ≤ capacity
2. Value ← Max

Input:

- Knapsack of capacity
- List (Array) of weight and their corresponding value.

Output: To maximize profit and minimize weight in capacity.

The knapsack problem where we have to pack the knapsack with maximum value in such a manner that the total weight of the items should not be greater than the capacity of the knapsack.

Knapsack problem can be further divided into two parts:

1. Fractional Knapsack: Fractional knapsack problem can be solved by **Greedy Strategy** where as 0 /1 problem is not.

It cannot be solved by **Dynamic Programming Approach.**

0/1 Knapsack Problem:

In this item cannot be broken which means thief should take the item as a whole or should leave it. That's why it is called **0/1 knapsack Problem**.

- Each item is taken or not taken.
- Cannot take a fractional amount of an item taken or take an item more than once.
- It cannot be solved by the Greedy Approach because it is enable to fill the knapsack to capacity.
- **Greedy Approach** doesn't ensure an Optimal Solution.

Example of 0/1 Knapsack Problem:

Example: The maximum weight the knapsack can hold is W is 11. There are five items to choose from. Their weights and values are presented in the following table:

Weight Limit (i):	0	1	2	3	4	5	6	7	8	9	10	11
$w_1 = 1\ v_1 = 1$												
$w_2 = 2\ v_2 = 6$												
$w_3 = 5\ v_3 = 18$												
$w_4 = 6\ v_4 = 22$												
$w_5 = 7\ v_5 = 28$												

The [i, j] entry here will be V [i, j], the best value obtainable using the first "i" rows of items if the maximum capacity were j. We begin by initialization and first row.

Weight Limit (i):	0	1	2	3	4	5	6	7	8	9	10	11
$w_1 = 1\ v_1 = 1$	0	1	1	1	1	1	1	1	1	1	1	1
$w_2 = 2\ v_2 = 6$	0											
$w_3 = 5\ v_3 = 18$	0											
$w_4 = 6\ v_4 = 22$	0											
$w_5 = 7\ v_5 = 28$	0											

$$V\ [i, j] = \max\ \{V\ [i - 1, j],\ v_i + V\ [i - 1, j - w_i]$$

Weight Limit (i):	0	1	2	3	4	5	6	7	8	9	10	11
$w_1 = 1\ v_1 = 1$	0	1	1	1	1	1	1	1	1	1	1	1
$w_2 = 2\ v_2 = 6$	0	1	6	7	7	7	7	7	7	7	7	7
$w_3 = 5\ v_3 = 18$	0											
$w_4 = 6\ v_4 = 22$	0											
$w_5 = 7\ v_5 = 28$	0											

Weight Limit (i):	0	1	2	3	4	5	6	7	8	9	10	11
$w_1 = 1\ v_1 = 1$	0	1	1	1	1	1	1	1	1	1	1	1
$w_2 = 2\ v_2 = 6$	0	1	6	7	7	7	7	7	7	7	7	7
$w_3 = 5\ v_3 = 18$	0	1	6	7	7	18	19	24	25	25	25	25
$w_4 = 6\ v_4 = 22$	0											
$w_5 = 7\ v_5 = 28$	0											

The value of V [3, 7] was computed as follows:

$$\begin{aligned} V\ [3, 7] &= \max\ \{V\ [3 - 1, 7],\ v_3 + V\ [3 - 1, 7 - w_3] \\ &= \max\ \{V\ [2, 7],\ 18 + V\ [2, 7 - 5]\} \\ &= \max\ \{7,\ 18 + 6\} \\ &= 24 \end{aligned}$$

Weight Limit (i):	0	1	2	3	4	5	6	7	8	9	10	11
$w_1 = 1\ v_1 = 1$	0	1	1	1	1	1	1	1	1	1	1	1
$w_2 = 2\ v_2 = 6$	0	1	6	7	7	7	7	7	7	7	7	7
$w_3 = 5\ v_3 = 18$	0	1	6	7	7	18	19	24	25	25	25	25
$w_4 = 6\ v_4 = 22$	0	1	6	7	7	18	22	24	28	29	29	40
$w_5 = 7\ v_5 = 28$	0											

Finally, the output is:

Weight Limit (i):	0	1	2	3	4	5	6	7	8	9	10	11
$w_1 = 1\ v_1 = 1$	0	1	1	1	1	1	1	1	1	1	1	1
$w_2 = 2\ v_2 = 6$	0	1	6	7	7	7	7	7	7	7	7	7
$w_3 = 5\ v_3 = 18$	0	1	6	7	7	18	19	24	25	25	25	25
$w_4 = 6\ v_4 = 22$	0	1	6	7	7	18	22	24	28	29	29	40
$w_5 = 7\ v_5 = 28$	0	1	6	7	7	18	22	28	29	34	35	40

The maximum value of items in the knapsack is 40, the bottom-right entry). The dynamic programming approach can now be coded as the following algorithm:

Algorithm of Knapsack Problem

KNAPSACK (n, W)

```
1. for w = 0, W
2. do V [0,w] ← 0
3. for i=0, n
4. do V [i, 0] ← 0
5. for w = 0, W
6. do if (wi≤ w & vi + V [i-1, w -  wi]> V [i -1,W])
7. then V [i, W] ← vi + V [i - 1, w - wi]
8. else V [i, W] ← V [i - 1, w]
```

DUTCH NATIONAL FLAG

Dutch National Flag (DNF) - It is a programming problem proposed by Edsger Dijkstra. The flag of the Netherlands consists of three colors: white, red, and blue. The task is to randomly arrange balls of white, red, and blue in such a way that balls of the same color are placed together. For DNF (Dutch National Flag), we sort an array of 0, 1, and 2's in linear time that does not consume any extra space. We have to keep in mind that this algorithm can be implemented only on an array that has three unique elements.

ALGORITHM -

- Take three-pointers, namely - low, mid, high.
- We use low and mid pointers at the start, and the high pointer will point at the end of the given array.

CASES :

- If array [mid] =0, then swap array [mid] with array [low] and increment both pointers once.
- If array [mid] = 1, then no swapping is required. Increment mid pointer once.
- If array [mid] = 2, then we swap array [mid] with array [high] and decrement the high pointer once.

CODE -

(IN C LANGUAGE)

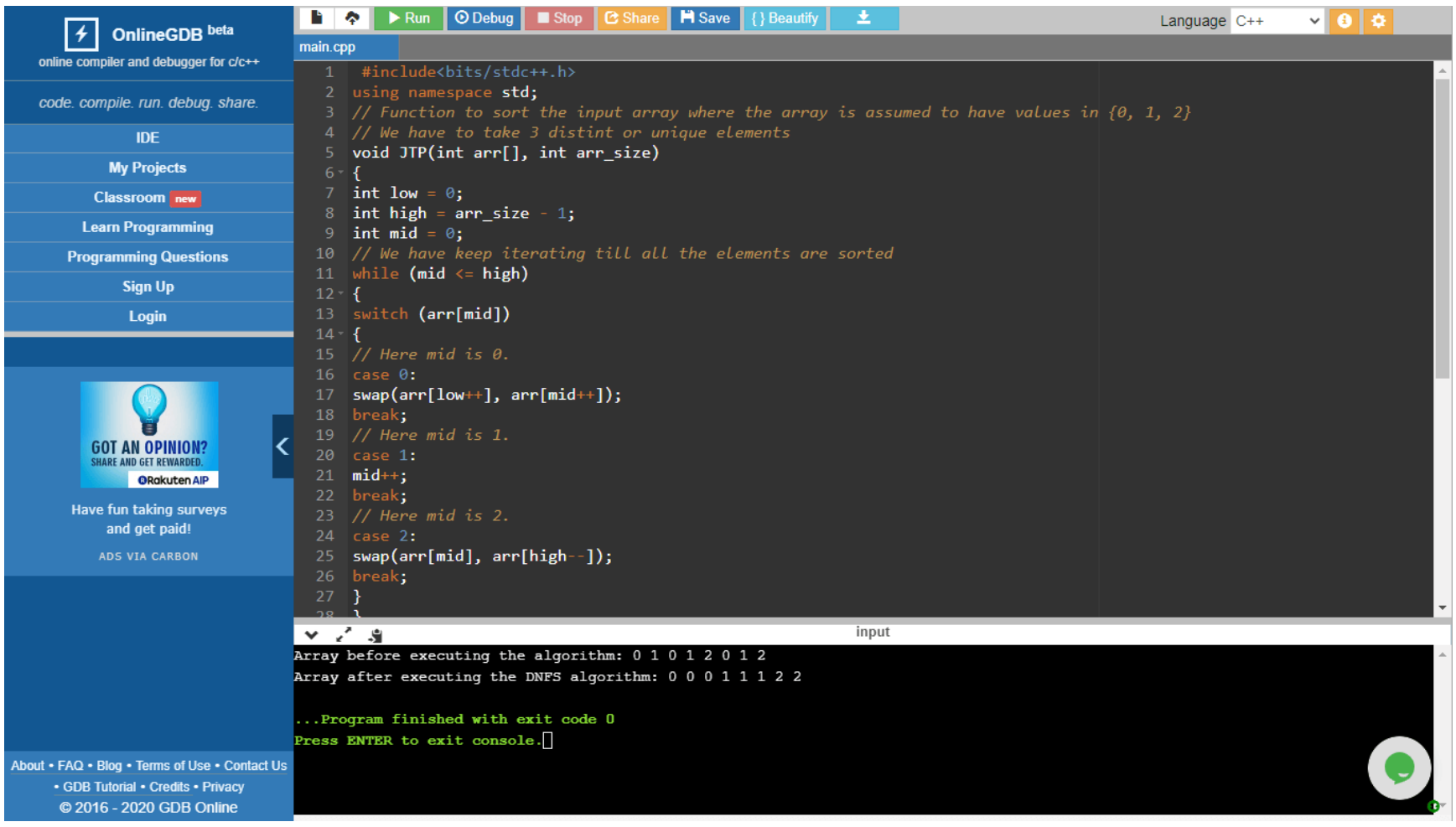
```
1. #include<bits/stdc++.h>
2. using namespace std;
3. // Function to sort the input array where the array is assumed to have values in {0, 1, 2}
4. // We have to take 3 distint or unique elements
5. void JTP(int arr[], int arr_size)
6. {
7. int low = 0;
8. int high = arr_size - 1;
9. int mid = 0;
```

```
// We have keep iterating till all the elements are sorted
while (mid <= high)
{
switch (arr[mid])
{
// Here mid is 0.
case 0:
swap(arr[low++], arr[mid++]);
break;
// Here mid is 1.
case 1:
mid++;
break;
// Here mid is 2.
case 2:
swap(arr[mid], arr[high--]);
break;
}
}
// Now, we write the function to print array arr[]
void printArray(int arr[], int arr_size)
{
// To iterate and print every element, we follow these steps
for (int i = 0; i < arr_size; i++)
cout << arr[i] << " ";
}
//Main Code
int main()
{
int arr[] = {0,1,0,1,2,0,1,2};
int n = sizeof(arr)/sizeof(arr[0]);
cout << "Array before executing the algorithm: ";
printArray(arr, n);
JTP(arr, n);
cout << "\nArray after executing the DNFS algorithm: ";
printArray(arr, n);
return 0;
}
```

OUTPUT -

Array before executing the algorithm: 0 1 0 1 2 0 1 2

Array after executing the DNFS algorithm: 0 0 0 1 1 1 2 2



CODE -

(IN JAVA)

```java
import java.io.*;
class DNF {
static void JTP(int arr[], int arr_size)
{
int low = 0;
int high = arr_size - 1;
int mid = 0, temp=0; // We use temporary variable for swapping
while (mid <= high)
{
switch (arr[mid])
{
case 0: // Here mid pointer points is at 0.
{
temp = arr[low];
arr[low] = arr[mid];
arr[mid] = temp;
low++;
mid++;
break;
}
case 1: // Here mid pointer points is at 1.
mid++;
break;
case 2: // Here mid pointer points is at 2.
{
temp = arr[mid];
arr[mid] = arr[high];
arr[high] = temp;
high--;
break;
}
}
}
```

```
35. // Now we have to call function to print array arr[]
36. static void printArray(int arr[], int arr_size)
37. {
38. int i;
39. for (i = 0; i < arr_size; i++)
40. System.out.print(arr[i]+" ");
41. System.out.println("");
42. }
43. //Now we use driver function to check for above functions
44. public static void main (String[] arguments)
45. {
46. int arr[] = {0, 1, 0, 1, 2, 0, 1, 2};
47. int arr_size = arr.length;
48. System.out.println("Array before executing the DNFS algorithm : ");
49. printArray(arr, arr_size);
50. JTP(arr, arr_size);
51. System.out.println("\nArray after executing the DNFS algorithm : ");
52. printArray(arr, arr_size);
53. }
54. }
```

OUTPUT -

Array before executing the DNFS algorithm : 0 1 0 1 2 0 1 2

Array after executing the DNFS algorithm : 0 0 0 1 1 1 2 2

C++ Java Python

```
1  import java.io.*;
2  class DNF {
3  static void JTP(int arr[], int arr_size)
4  {
5  int low = 0;
6  int high = arr_size - 1;
7  int mid = 0, temp=0; // We use temporary variable for swapping
8  while (mid <= high)
9  {
10 switch (arr[mid])
11 {
12 case 0: // Here mid pointer points is at 0.
13 {
14 temp = arr[low];
15 arr[low] = arr[mid];
16 arr[mid] = temp;
17 low++;
18 mid++;
19 break;
20 }
```

RUN

Close

```
Output                                                    2.741s

Array before executing the DNFS algorithm :
0 1 0 1 2 0 1 2

Array after executing the DNFS algorithm :
0 0 0 1 1 1 2 2
```

CODE -

(IN PYTHON)

```
1.  def JTP( arr, arr_size):
2.    low = 0
3.    high = arr_size - 1
4.    mid = 0
5.    while mid <= high:
6.      if arr[mid] == 0:
7.        arr[low],arr[mid] = arr[mid],arr[low]
8.        low = low + 1
```

```
9.        mid = mid + 1
10.     elif arr[mid] == 1:
11.       mid = mid + 1
12.     else:
13.       arr[mid],arr[high] = arr[high],arr[mid]
14.       high = high - 1
15.   return arr
16.
17. # Function to print array
18. def printArray(arr):
19.     for k in arr:
20.         print k,
21.     print
22.
23. # Driver Program
24. arr = [0, 1, 0, 1, 1, 2, 0, 1, 2]
25. arr_size = len(arr)
26. print " Array before executing the algorithm: ",
27. printArray(arr)
28. arr = JTP(arr, arr_size)
29. print "Array after executing the DNFS algorithm: ",
30. printArray(arr)
```

OUTPUT -

Array before executing the algorithm: 0 1 0 1 1 2 0 1 2

Array after executing the DNFS algorithm: 0 0 0 1 1 1 1 2 2

C++ Java Python

```
def JTP( arr, arr_size):
  low = 0
  high = arr_size - 1
  mid = 0
  while mid <= high:
    if arr[mid] == 0:
      arr[low],arr[mid] = arr[mid],arr[low]
      low = low + 1
      mid = mid + 1
    elif arr[mid] == 1:
      mid = mid + 1
    else:
      arr[mid],arr[high] = arr[high],arr[mid]
      high = high - 1
  return arr

# Function to print array
def printArray(arr):
    for k in arr:
        print k,
    print
```

RUN

Close

Output 0.649s

```
 Array before executing the algorithm:  0 1 0 1 1 2 0 1 2
Array after executing the DNFS algorithm:  0 0 0 1 1 1 1 2 2
```

Longest Palindrome Subsequence

It is a sequence of characters in a string that can be spelled and read the same both ways, forward and backward. Words like civic, redivider, deified, radar, level, madam, rotor, refer, kayak, racecar, and reviver. But in palindromic subsequence, a sequence can but not necessarily appear in the same relative order, but the chance of being necessarily contiguous and palindromic in nature is negligible.

Dynamic Programming Solution -

Example - We have been given a sequence as "BDBADBDCBDCADB." Then the longest palindrome will be eleven - "BDABDCDBADB", this is the longest palindromic subsequence here. 'BBABB,' 'DAD,' 'BBDBB' and many more are also

palindromic subsequences of the given sequence, but they are not the longest. In simpler words, sequences generate subsequences, then we compare their length and find which palindromic subsequence is the longest.

We follow these steps to achieve the most extended palindrome sequence using Dynamic Programming Solution -

First, we reverse the sequence.

Then we use the LCS algorithm (Longest Common Subsequence) to find the longest common subsequence among the original sequence and reversed sequence. Here original LCS and reverse LCS are a function that returns the longest common subsequence between the pair of strings, now the answer from LCS will be the longest palindromic subsequence.

Let LP (a, b) = Length of longest palindromic subsequence in array Z from index a to b

LP (a, b) = LP (a+1, b-1) + 2: if Z [a] = X [b]

= max [LP (a+1, b), LP (a, b-1)]: if Z [a]! = Z [b]

= 1 if a = b

= 1 if a = b - 1 and Z [a]! = Z [b]

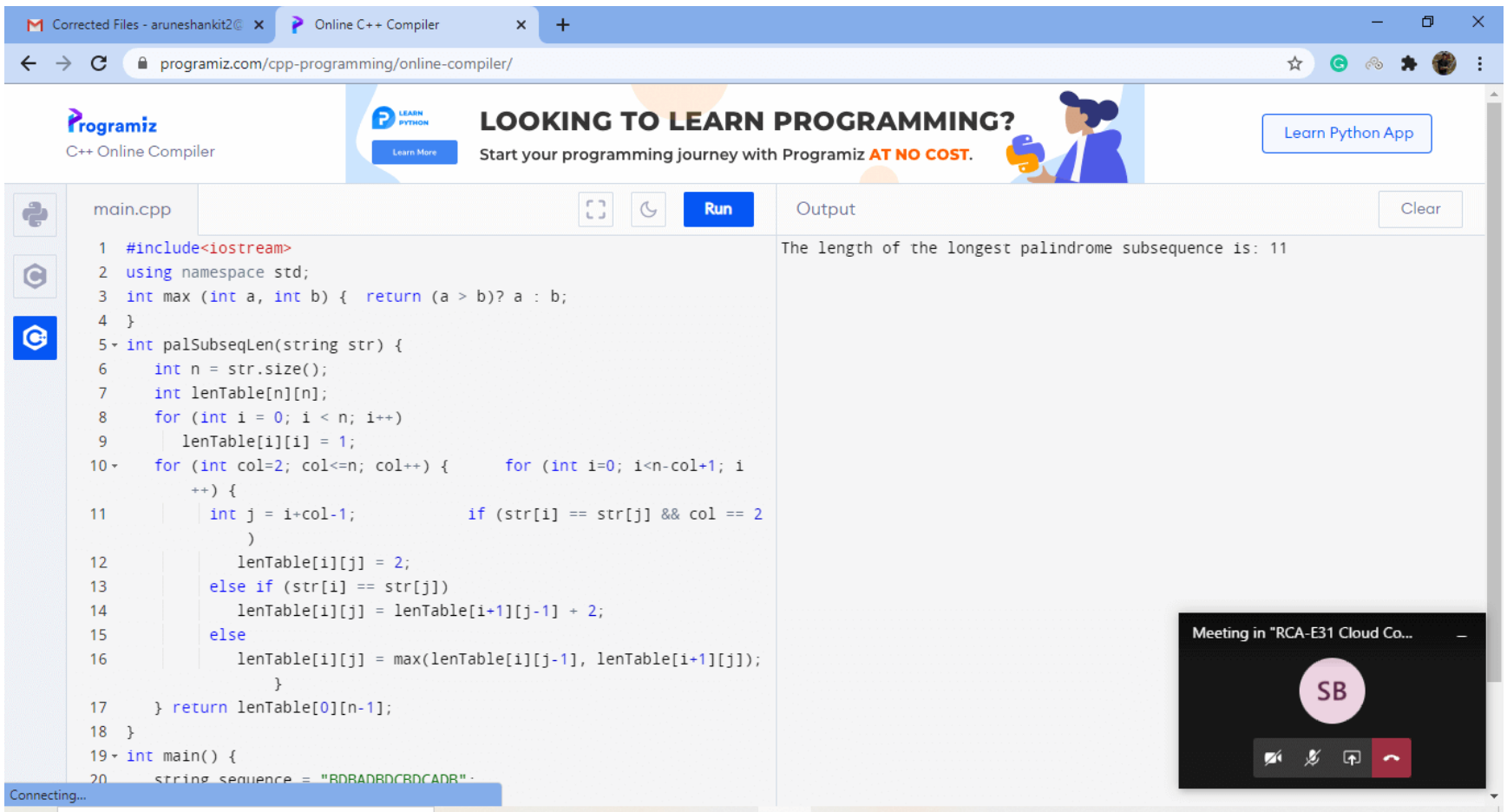
= 2 if a = b - 1 and Z [a] = Z [b]

Code -

```
#include<iostream>
using namespace std;
int max (int a, int b) {  return (a > b)? a : b;
}
int palSubseqLen(string str) {
   int n = str.size();
   int lenTable[n][n];
   for (int i = 0; i < n; i++)
      lenTable[i][i] = 1;
   for (int col=2; col<=n; col++) {      for (int i=0; i<n-col+1; i++) {
      int j = i+col-1;           if (str[i] == str[j] && col == 2)
         lenTable[i][j] = 2;
      else if (str[i] == str[j])
         lenTable[i][j] = lenTable[i+1][j-1] + 2;
      else
         lenTable[i][j] = max(lenTable[i][j-1], lenTable[i+1][j]); }
   } return lenTable[0][n-1];
}
int main() {
   string sequence = "BDBADBDCBDCADB";
   int n = sequence.size();
   cout << "The length of the longest palindrome subsequence is: " << palSubseqLen(sequence);
}
```

OUTPUT -

The length of the longest palindrome subsequence is: 11



Now, if we were to combine all the above cases into a mathematical equation:

We call the original sequence X = (x1x2 ... xm) and reverse as Y = (y1y2 ... ym). Here, the prefixes of X are X1, 2, 3 ... m and the prefixes of Y are Y1, 2, 3 ... m.

Let LCS (XI, Yj) represent the set of the longest common subsequence of prefixes Xi and Yj.

Then:

LCS (Xi , Yj) = ∅ ; if i = 0 or j = 0

LCS (Xi , Yj) = LCS(Xi-1, Yj-1) ^ xi ; if i > 0, j > 0 & xi = yj

LCS (Xi , Yj) = max{ LCS (Xi, Yj-1), LCS (Xi-1, Yj) } ; if i > 0, j > 0 & xi ≠ yj

If the last characters match, then the sequence LCS (Xi-1 , Yj-1) is extended by that matching character xi. Otherwise, the best result from LCS (Xi , Yj-1) and LCS (Xi-1 , Yj) is used.

In the recursive method, we compute some sub-problem, divide it, and repeatedly perform this kind of task. So it's a simple but very tedious method. The time complexity in recursive solution is more. The worst-case time complexity is exponential O (2^n), and auxiliary space used by the program is O (1).

In X, if the last and first characters are the same -

X (0, n - 1) = X (1, n - 2) + 2

If not, then

X (0, n - 1) = MAX (X (1, n - 1), X (0, n - 2)).

CODE -

(IN JAVA)

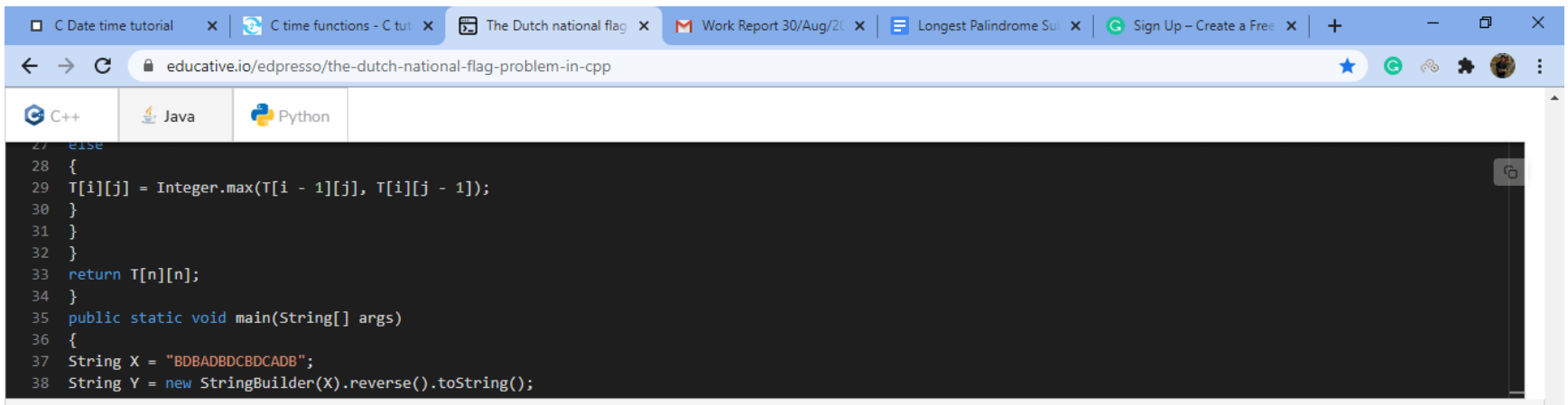
```
1. class Main
2. {
3. public static String longestPalindrome(String X, String Y, int m, int n, int[][] T)
4. {
5. if (m == 0 || n == 0) {
6. return "";
7. }
8. if (X.charAt(m - 1) == Y.charAt(n - 1))
9. {
10. return longestPalindrome(X, Y, m - 1, n - 1, T) + X.charAt(m - 1);
```

```
}
if (T[m - 1][n] > T[m][n - 1]) {
return longestPalindrome(X, Y, m - 1, n, T);
}
return longestPalindrome(X, Y, m, n - 1, T);
}
public static int LCSLength(String X, String Y, int n, int[][] T)
{
for (int i = 1; i <= n; i++)
{
for (int j = 1; j <= n; j++)
{
if (X.charAt(i - 1) == Y.charAt(j - 1))
{
T[i][j] = T[i - 1][j - 1] + 1;
}
else
{
T[i][j] = Integer.max(T[i - 1][j], T[i][j - 1]);
}
}
}
return T[n][n];
}
public static void main(String[] args)
{
String X = "BDBADBDCBDCADB";
String Y = new StringBuilder(X).reverse().toString();
int[][] T = new int[X.length() + 1][X.length() + 1];
System.out.println("The length of Longest Palindromic Subsequence is "
                    + LCSLength(X, Y, X.length(), T));
System.out.println("The Longest Palindromic Subsequence is "
                    + longestPalindrome(X, Y, X.length(), X.length(), T));
}
}
```

OUTPUT -

The length of the Longest Palindromic Subsequence is 11

The Longest Palindromic Subsequence is BDADBCBDADB



Now, if we were to combine all the above cases into a mathematical equation:

We call the original sequence X = (x1x2 ... xm) and reverse as Y = (y1y2 ... ym). Here, the prefixes of X are X1, 2, 3 ... m and the prefixes of Y are Y1, 2, 3 ... m.

Let LCS (XI, Yj) represent the set of the longest common subsequence of prefixes Xi and Yj.

Then:

LCS (Xi , Yj) = ∅ ; if i = 0 or j = 0

LCS (Xi , Yj) = LCS(Xi-1, Yj-1) ^ xi ; if i > 0, j > 0 & xi = yj

LCS (Xi , Yj) = max{ LCS (Xi, Yj-1), LCS (Xi-1, Yj) } ; if i > 0, j > 0 & xi ≠ yj

If the last characters match, then the sequence LCS (Xi-1 , Yj-1) is extended by that matching character xi. Otherwise, the best result from LCS (Xi , Yj-1) and LCS (Xi-1 , Yj) is used.

In the recursive method, we compute some sub-problem, divide it, and repeatedly perform this kind of task. So it's a simple but very tedious method. The time complexity in recursive solution is more. The worst-case time complexity is exponential O (2^n), and auxiliary space used by the program is O (1).

In X, if the last and first characters are the same -

X (0, n - 1) = X (1, n - 2) + 2

If not, then

X (0, n - 1) = MAX (X (1, n - 1), X (0, n - 2)).

CODE -

(IN JAVA)

```
class Main
{
public static String longestPalindrome(String X, String Y, int m, int n, int[][] T)
{
if (m == 0 || n == 0) {
return "";
}
if (X.charAt(m - 1) == Y.charAt(n - 1))
{
return longestPalindrome(X, Y, m - 1, n - 1, T) + X.charAt(m - 1);
```

```
}
if (T[m - 1][n] > T[m][n - 1]) {
return longestPalindrome(X, Y, m - 1, n, T);
}
return longestPalindrome(X, Y, m, n - 1, T);
}
public static int LCSLength(String X, String Y, int n, int[][] T)
{
for (int i = 1; i <= n; i++)
{
for (int j = 1; j <= n; j++)
{
if (X.charAt(i - 1) == Y.charAt(j - 1))
{
T[i][j] = T[i - 1][j - 1] + 1;
}
else
{
T[i][j] = Integer.max(T[i - 1][j], T[i][j - 1]);
}
}
}
return T[n][n];
}
public static void main(String[] args)
{
String X = "BDBADBDCBDCADB";
String Y = new StringBuilder(X).reverse().toString();
int[][] T = new int[X.length() + 1][X.length() + 1];
System.out.println("The length of Longest Palindromic Subsequence is "
                    + LCSLength(X, Y, X.length(), T));
System.out.println("The Longest Palindromic Subsequence is "
                    + longestPalindrome(X, Y, X.length(), X.length(), T));
}
}
```

OUTPUT -

The length of the Longest Palindromic Subsequence is 11

The Longest Palindromic Subsequence is BDADBCBDADB

C++ Java Python

```java
class Main
{
public static String longestPalindrome(String X, String Y, int m, int n, int[][] T)
{
if (m == 0 || n == 0) {
return "";
}
if (X.charAt(m - 1) == Y.charAt(n - 1))
{
return longestPalindrome(X, Y, m - 1, n - 1, T) + X.charAt(m - 1);
}
if (T[m - 1][n] > T[m][n - 1]) {
return longestPalindrome(X, Y, m - 1, n, T);
}
return longestPalindrome(X, Y, m, n - 1, T);
}
public static int LCSLength(String X, String Y, int n, int[][] T)
{
for (int i = 1; i <= n; i++)
{
for (int j = 1; j <= n; j++)
{
if (X.charAt(i - 1) == Y.charAt(j - 1))
{
T[i][j] = T[i - 1][j - 1] + 1;
}
else
```

RUN

Close

Output 1.439s

```
The length of Longest Palindromic Subsequence is 11
The Longest Palindromic Subsequence is BDADBCBDADB
```

Optimal Substructure -

It satisfies overlapping subproblem properties. In a two dimensional array, Longest Common Subsequence can be made as a memo, where LCS[X][Y] represents the length between original with length X and reverse, with length. The longest palindromic subsequence can be generated by backtracking technique, after filling and using the above algorithm