

## 0/1 KnapSack in C++

```

#include <iostream>
#include <vector>

using namespace std;

class ZeroOneKnapsack {
public:
    int knapsack(int n, vector<int>& vals,
vector<int>& wts, int cap) {
        vector<vector<int>> dp(n + 1, vector<int>(cap + 1, 0));

        for (int i = 1; i <= n; i++) {
            for (int j = 1; j <= cap; j++) {
                if (j >= wts[i - 1]) {
                    int remainingCap = j - wts[i - 1];

                    if (dp[i - 1][remainingCap] + vals[i - 1] >
dp[i - 1][j]) {
                        dp[i][j] = dp[i - 1][remainingCap] +
vals[i - 1];
                    } else {
                        dp[i][j] = dp[i - 1][j];
                    }
                } else {
                    dp[i][j] = dp[i - 1][j];
                }
            }
        }

        return dp[n][cap];
    }
};

int main() {
    ZeroOneKnapsack solution;

    // Input parameters
    int n = 5;
    vector<int> vals = {15, 14, 10, 45, 30};
    vector<int> wts = {2, 5, 1, 3, 4};
    int cap = 7;

    // Compute maximum value using knapsack
    function
    int maxVal = solution.knapsack(n, vals, wts, cap);

    // Output the maximum value
    cout << "Maximum value that can be obtained: " <<
maxVal << endl;

    return 0;
}

```

## Dry Run of the ZeroOneKnapsack Problem:

## Input:

```
n = 5;  
vals = {15, 14, 10, 45, 30};  
wts = {2, 5, 1, 3, 4};  
cap = 7;
```

### **Step 1: Initialize the DP Table**

We initialize a 2D DP table of size  $(n + 1) \times (\text{cap} + 1)$  to store the maximum values obtainable for each subproblem. Each cell  $\text{dp}[i][j]$  will represent the maximum value achievable with the first  $i$  items and a knapsack capacity of  $j$ .

Initially, the DP table is filled with zeros:

### Step 2: Fill the DP Table

We iterate through each item ( $i = 1$  to  $n$ ) and each knapsack capacity ( $j = 1$  to  $\text{cap}$ ). The idea is to decide whether to include the current item or not.

**Item 1 (Value = 15, Weight = 2)**

- **Capacity 1:**  $dp[1][1] = 0$  (Cannot include this item as the weight is greater than the capacity)
  - **Capacity 2:**  $dp[1][2] = \max(dp[0][2], dp[0][0] + 15) = \max(0, 15) = 15$
  - **Capacity 3:**  $dp[1][3] = \max(dp[0][3], dp[0][1] + 15) = \max(0, 15) = 15$
  - **Capacity 4:**  $dp[1][4] = \max(dp[0][4], dp[0][2] + 15) = \max(0, 15) = 15$
  - **Capacity 5:**  $dp[1][5] = \max(dp[0][5], dp[0][3] + 15) = \max(0, 15) = 15$
  - **Capacity 6:**  $dp[1][6] = \max(dp[0][6], dp[0][4] + 15) = \max(0, 15) = 15$
  - **Capacity 7:**  $dp[1][7] = \max(dp[0][7], dp[0][5] + 15) = \max(0, 15) = 15$

i\j	0	1	2	3	4	5	6	7
1	0	0	15	15	15	15	15	15
2	0	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0	0
5	0	0	0	0	0	0	0	0

#### Item 2 (Value = 14, Weight = 5)

- **Capacity 1 to 4:** The weight is greater than the capacity, so we can't include this item.
- **Capacity 5:**  $dp[2][5] = \max(dp[1][5], dp[1][0] + 14) = \max(15, 14) = 15$
- **Capacity 6:**  $dp[2][6] = \max(dp[1][6], dp[1][1] + 14) = \max(15, 14) = 15$
- **Capacity 7:**  $dp[2][7] = \max(dp[1][7], dp[1][2] + 14) = \max(15, 29) = 29$

i\j	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	0	15	15	15	15	15	15
2	0	0	15	15	15	15	15	29
3	0	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0	0
5	0	0	0	0	0	0	0	0

#### Item 3 (Value = 10, Weight = 1)

- **Capacity 1:**  $dp[3][1] = \max(dp[2][1], dp[2][0] + 10) = \max(0, 10) = 10$
- **Capacity 2:**  $dp[3][2] = \max(dp[2][2], dp[2][1] + 10) = \max(15, 10) = 15$
- **Capacity 3:**  $dp[3][3] = \max(dp[2][3], dp[2][2] + 10) = \max(15, 25) = 25$
- **Capacity 4:**  $dp[3][4] = \max(dp[2][4], dp[2][3] + 10) = \max(15, 25) = 25$
- **Capacity 5:**  $dp[3][5] = \max(dp[2][5], dp[2][4] + 10) = \max(15, 25) = 25$
- **Capacity 6:**  $dp[3][6] = \max(dp[2][6], dp[2][5] + 10) = \max(15, 25) = 25$
- **Capacity 7:**  $dp[3][7] = \max(dp[2][7], dp[2][6] + 10) = \max(29, 25) = 29$

i\j	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	0	15	15	15	15	15	15
2	0	0	15	15	15	15	15	29
3	0	10	15	25	25	25	25	29
4	0	0	0	0	0	0	0	0
5	0	0	0	0	0	0	0	0

#### Item 4 (Value = 45, Weight = 3)

- **Capacity 1 to 2:** Cannot include this item.
- **Capacity 3:**  $dp[4][3] = \max(dp[3][3], dp[3][0] + 45) = \max(25, 45) = 45$
- **Capacity 4:**  $dp[4][4] = \max(dp[3][4], dp[3][1] + 45) = \max(25, 55) = 55$
- **Capacity 5:**  $dp[4][5] = \max(dp[3][5], dp[3][2] + 45) = \max(25, 55) = 55$
- **Capacity 6:**  $dp[4][6] = \max(dp[3][6], dp[3][3] + 45) = \max(25, 70) = 70$
- **Capacity 7:**  $dp[4][7] = \max(dp[3][7], dp[3][4] + 45) = \max(29, 70) = 70$

#### Item 5 (Value = 30, Weight = 4)

- **Capacity 1 to 3:** Cannot include this item.
- **Capacity 4:**  $dp[5][4] = \max(dp[4][4], dp[4][0] + 30) = \max(55, 30) = 55$
- **Capacity 5:**  $dp[5][5] = \max(dp[4][5], dp[4][1] + 30) = \max(55, 30) = 55$
- **Capacity 6:**  $dp[5][6] = \max(dp[4][6], dp[4][2] + 30) = \max(70, 30) = 70$
- **Capacity 7:**  $dp[5][7] = \max(dp[4][7], dp[4][3] + 30) = \max(70, 75) = 75$

#### Step 3: Final DP Table

i\j	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	0	15	15	15	15	15	15
2	0	0	15	15	15	15	15	29
3	0	10	15	25	25	25	25	29
4	0	10	15	45	55	55	70	70
5	0	10	15	45	55	55	70	75

#### Result:

The maximum value that can be obtained with a knapsack capacity of 7 is 75.

Output:

Maximum value that can be obtained: 75

The maximum value that can be obtained is stored in  $dp[5][7] = 75$ .

## Best time to buy and sell stocks in C++

```
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

class BestTimeToBuyAndSellStock {
public:
    int maxProfit(vector<int>& prices) {
        if (prices.empty()) return 0;

        int maxP = 0;
        int minBP = prices[0];

        for (int prc : prices) {
            int tp = prc - minBP;
            if (tp > maxP) {
                maxP = tp;
            }
            minBP = min(minBP, prc);
        }

        return maxP;
    }

    int main() {
        BestTimeToBuyAndSellStock solution;

        // Test case 1
        vector<int> prices1 = {7, 1, 5, 3, 6, 4};
        int maxProfit1 = solution.maxProfit(prices1);
        cout << "Max profit for prices1: " << maxProfit1 << endl; // Output: 5

        return 0;
    }
}
```

**Output:-**

maxP = 5 (Maximum profit)

Let's walk through a **dry run in tabular form** of your code for:

`vector<int> prices1 = {7, 1, 5, 3, 6, 4};`

💡 **Variables:**

- `minBP` = Minimum Buying Price seen so far.
- `tp` = Temporary Profit (current price - `minBP`).
- `maxP` = Maximum Profit observed.

💡 **Dry Run Table:**

Day (Index)	Price	minBP (min so far)	tp = price - minBP	maxP (max profit so far)
0	7	7	0	0
1	1	1	0	0
2	5	1	4	4
3	3	1	2	4
4	6	1	5	5 ✓
5	4	1	3	5

✓ **Final Answer:**

Max profit for prices1: 5

## Best time to buy and Sell Stocks infinite in C++

```
#include <iostream>
#include <vector>

using namespace std;

class BestTimeToBuyAndSellStocksInfiniteTransactions {
public:
    int maxProfit(vector<int>& prices) {
        if (prices.empty()) return 0;

        int bd = 0; // Buy day
        int sd = 0; // Sell day
        int profit = 0;

        for (int i = 1; i < prices.size(); ++i) {
            if (prices[i] >= prices[i - 1]) {
                sd++;
            } else {
                profit += prices[sd] - prices[bd];
                bd = sd = i;
            }
        }

        profit += prices[sd] - prices[bd];
        return profit;
    }

    int main() {
        BestTimeToBuyAndSellStocksInfiniteTransactions
        solution;

        // Test case
        vector<int> prices = {11, 6, 7, 19, 4, 1, 6, 18, 4};
        int maxProfit = solution.maxProfit(prices);
        cout << "Max profit: " << maxProfit << endl; //
Output: 30

        return 0;
    }
}
```

Let's perform a **tabular dry run** of your code for the input:

prices = {11, 6, 7, 19, 4, 1, 6, 18, 4}

### ✓ Logic Summary:

- Buy at bd (buy day), sell at sd (sell day).
- Keep increasing sd as long as prices go up or stay the same.
- When price drops, add profit of the last segment (prices[sd] - prices[bd]) and reset bd = sd = i.

### Dry Run Table:

i	prices[i]	Action Taken	bd	sd	Segment Profit	Total Profit
0	11	Initial buy	0	0		0
1	6	Drop → sell at 11, profit = 0	1	1	11 - 11 = 0	0
2	7	Rise → extend sell day	1	2		0
3	19	Rise → extend sell day	1	3		0
4	4	Drop → sell at 19, profit = 19 - 6 = 13	4	4	19 - 6 = 13	13
5	1	Drop → sell at 4, profit = 0	5	5	4 - 4 = 0	13
6	6	Rise → extend sell day	5	6		13
7	18	Rise → extend sell day	5	7		13
8	4	Drop → sell at 18, profit = 18 - 1 = 17	8	8	18 - 1 = 17	30
—	—	Final segment (bd == sd == 8) → 0 profit			4 - 4 = 0	30

### ✓ Final Output:

	<p>Max profit: 30</p> <p> <b>Insight:</b></p> <p>You earned profit from:</p> <ul style="list-style-type: none"><li>• Buying at 6 → selling at 19 (Profit: 13)</li><li>• Buying at 1 → selling at 18 (Profit: 17)</li></ul>
--	---

Output:-  
Max profit: 30

## Climbing Stairs in C++

```
#include <iostream>
#include <vector>
#include <climits> // For INT_MAX

using namespace std;

void printMinSteps(vector<int>& arr) {
    int n = arr.size();
    vector<int> dp(n + 1, INT_MAX); // Use INT_MAX
for initialization

    dp[n] = 0; // Base case: 0 steps needed from the end

    for (int i = n - 1; i >= 0; i--) {
        if (arr[i] > 0) {
            int minSteps = INT_MAX;
            for (int j = 1; j <= arr[i] && (i + j) < dp.size(); j++) {
                if (dp[i + j] != INT_MAX) {
                    minSteps = min(minSteps, dp[i + j]);
                }
            }
            if (minSteps != INT_MAX) {
                dp[i] = minSteps + 1;
            }
        }
    }

    // Printing the dp array
    for (int i = 0; i < dp.size(); i++) {
        cout << " " << dp[i];
    }
    cout << endl;
}

int main() {
    vector<int> arr = {1, 5, 2, 3, 1};
    printMinSteps(arr);

    return 0;
}
```

Given:

`vector<int> arr = {1, 5, 2, 3, 1};`

The length of arr is **5**, so dp is initialized as:

`dp = [INT_MAX, INT_MAX, INT_MAX, INT_MAX, INT_MAX, 0] // (size = 6, last element is 0)`

### Dry Run with Iteration Table

The loop iterates from **i = n - 1 to 0**, checking possible jumps and updating dp[i].

Iteration (i)	arr[i]	Possible Jumps	Min Steps from Reachable Positions	Updated dp[i]
4 (last)	1	(4→5)	dp[5] = 0 → min(∞, 0)	dp[4] = 1
3	3	(3→4, 3→5)	dp[4] = 1, dp[5] = 0 → min(∞, 1, 0)	dp[3] = 1
2	2	(2→3, 2→4)	dp[3] = 1, dp[4] = 1 → min(∞, 1, 1)	dp[2] = 2
1	5	(1→2, 1→3, 1→4, 1→5)	dp[2] = 2, dp[3] = 1, dp[4] = 1, dp[5] = 0 → min(∞, 2, 1, 1, 0)	dp[1] = 1
0 (first)	1	(0→1)	dp[1] = 1 → min(∞, 1)	dp[0] = 2

### Final dp Array

After all iterations, the **dp array** will be:

`dp = [2, 1, 2, 1, 1, 0]`

### Output:

2 1 2 1 1 0

### Output:-

① Printed dp: 2 1 2 1 1 0

② The minimum steps to reach the end starting from index 0 is dp[0] = 2.

## Coin Change Combination in C++

```
#include <iostream>
#include <vector>

using namespace std;

int main() {
    vector<int> arr = {2, 3, 5};
    int amt = 7;
    vector<int> dp(amt + 1, 0);
    dp[0] = 1; // Base case: 1 way to make amount 0
    (using no coins)

    for (int i = 0; i < arr.size(); i++) {
        for (int j = arr[i]; j <= amt; j++) {
            dp[j] += dp[j - arr[i]];
        }
    }

    cout << dp[amt] << endl; // Output the number of
    combinations for amount `amt`

    return 0;
}
```

### Initial dp Array

Before processing:

$\text{arr} = [2, 3, 5]$

$\text{dp} = [1, 0, 0, 0, 0, 0, 0, 0]$

(Index represents amount: 0 to 7)

### Dry Run with Iteration Table

#### Processing coin 2

j (amt)	$\text{dp}[j] = \text{dp}[j] + \text{dp}[j - 2]$	Updated dp
2	$\text{dp}[2] += \text{dp}[0] =$ 1	[1, 0, 1, 0, 0, 0, 0, 0]
3	$\text{dp}[3] += \text{dp}[1] =$ 0	[1, 0, 1, 0, 0, 0, 0, 0]
4	$\text{dp}[4] += \text{dp}[2] =$ 1	[1, 0, 1, 0, 1, 0, 0, 0]
5	$\text{dp}[5] += \text{dp}[3] =$ 0	[1, 0, 1, 0, 1, 0, 0, 0]
6	$\text{dp}[6] += \text{dp}[4] =$ 1	[1, 0, 1, 0, 1, 0, 1, 0]
7	$\text{dp}[7] += \text{dp}[5] =$ 0	[1, 0, 1, 0, 1, 0, 1, 0]

#### Processing coin 3

j (amt)	$\text{dp}[j] = \text{dp}[j] + \text{dp}[j - 3]$	Updated dp
3	$\text{dp}[3] += \text{dp}[0] =$ 1	[1, 0, 1, 1, 1, 0, 1, 0]
4	$\text{dp}[4] += \text{dp}[1] =$ 0	[1, 0, 1, 1, 1, 0, 1, 0]
5	$\text{dp}[5] += \text{dp}[2] =$ 1	[1, 0, 1, 1, 1, 1, 1, 0]
6	$\text{dp}[6] += \text{dp}[3] =$ 1	[1, 0, 1, 1, 1, 1, 2, 0]
7	$\text{dp}[7] += \text{dp}[4] =$ 1	[1, 0, 1, 1, 1, 1, 2, 1]

### Processing coin 5

j (amt)	dp[j] = dp[j] + dp[j - 5]	Updated dp
5	dp[5] += dp[0] = 1	[1, 0, 1, 1, 1, 2, 2, 1]
6	dp[6] += dp[1] = 0	[1, 0, 1, 1, 1, 2, 2, 1]
7	dp[7] += dp[2] = 1	[1, 0, 1, 1, 1, 2, 2, 2]

### Final dp Array

After processing all coins:

dp = [1, 0, 1, 1, 1, 2, 2, 2]

### Final Output

2

This means **there are 2 ways to form amount 7 using {2, 3, 5}**:

1. 2 + 2 + 3
2. 2 + 5

**Output:-**  
2

### Coin Change Permutation in C++

```
#include <iostream>
#include <vector>

using namespace std;

int main() {
    vector<int> coins = {2, 3, 5};
    int tar = 7;
    vector<int> dp(tar + 1, 0);
    dp[0] = 1; // Base case: 1 way to make amount 0
    (using no coins)

    for (int amt = 1; amt <= tar; amt++) {
        for (int coin : coins) {
            if (coin <= amt) {
                int ramt = amt - coin;
                dp[amt] += dp[ramt];
            }
        }
    }

    cout << dp[tar] << endl; // Output the number of
    permutations to make the target amount

    return 0;
}
```

#### Initial dp Array

Before processing:

$dp = [1, 0, 0, 0, 0, 0, 0, 0, 0]$  // (Indexes represent amounts from 0 to 7)

#### Dry Run with Iteration Table

##### Iterating over amt from 1 to 7

amt	Coin Used	$dp[amt] = dp[amt] + dp[amt - coin]$	Updated dp
1	2 (skipped)	-	[1, 0, 0, 0, 0, 0, 0, 0]
	3 (skipped)	-	
	5 (skipped)	-	
2	2	$dp[2] += dp[0] = 1$	[1, 0, 1, 0, 0, 0, 0]
	3, 5 (skipped)	-	
	2	$dp[3] += dp[1] = 0$	[1, 0, 1, 0, 0, 0]
3	3	$dp[3] += dp[0] = 1$	[1, 0, 1, 1, 0, 0, 0]
	5 (skipped)	-	
	2	$dp[4] += dp[2] = 1$	[1, 0, 1, 1, 1, 0, 0]
4	3	$dp[4] += dp[1] = 0$	[1, 0, 1, 1, 1, 0, 0]
	5 (skipped)	-	
	2	$dp[5] += dp[3] = 1$	[1, 0, 1, 1, 1, 1, 0]
5	3	$dp[5] += dp[2] = 1$	[1, 0, 1, 1, 1, 1, 2]
	5	$dp[5] += dp[0] = 1$	[1, 0, 1, 1, 1, 1, 3]
	2	$dp[6] += dp[4] = 1$	[1, 0, 1, 1, 1, 1, 3]
6	3	$dp[6] += dp[3] = 1$	[1, 0, 1, 1, 1, 1, 3, 2]
	5	$dp[6] += dp[1] = 0$	[1, 0, 1, 1, 1, 1, 3, 2, 0]
	2	$dp[7] += dp[5] = 3$	[1, 0, 1, 1, 1, 1, 3, 2, 3]
7	3	$dp[7] += dp[4] = 1$	[1, 0, 1, 1, 1, 1, 3, 2, 4]
	5	$dp[7] += dp[2] = 1$	[1, 0, 1, 1, 1, 1, 3, 2, 5]

#### Final dp Array

After processing all amounts:

$dp = [1, 0, 1, 1, 1, 3, 2, 5]$

**Final Output**

5

This means **there are 5 different permutations to form amount 7 using {2, 3, 5}**:

1. **2 + 2 + 3**
2. **2 + 3 + 2**
3. **3 + 2 + 2**
4. **2 + 5**
5. **5 + 2**

Output:-

5

### Friend's Pairing in C++

```
#include <iostream>
#include <vector>

using namespace std;

int main() {
    int n = 3;

    vector<int> dp(n + 1);
    dp[1] = 1;
    dp[2] = 2;

    for (int i = 3; i <= n; i++) {
        dp[i] = dp[i - 1] + dp[i - 2] * (i - 1);
    }

    cout << dp[n] << endl;

    return 0;
}
```

Output:-  
4

### Dry Run with Iteration Table

#### Initial State

dp = [?, 1, 2] // (dp[0] is unused)

#### Iterating from i = 3 to n = 3

i	Calculation	Updated dp[i]
3	dp[3] = dp[2] + dp[1] * (3 - 1)	dp[3] = 2 + 1 * 2 = 4

#### Final dp Array

dp = [?, 1, 2, 4]

#### Final Output

4

## GoldMine in C++

```
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

int main() {
    int grid[4][4] = {
        {8, 2, 1, 6},
        {6, 5, 5, 2},
        {2, 1, 0, 3},
        {7, 2, 2, 4}
    };

    int n = 4; // Number of rows
    int m = 4; // Number of columns

    // Initialize dp array
    vector<vector<int>> dp(n, vector<int>(m, 0));

    // Fill dp array from rightmost column to left
    for (int j = m - 1; j >= 0; j--) {
        for (int i = n - 1; i >= 0; i--) {
            if (j == m - 1) {
                dp[i][j] = grid[i][j];
            } else if (i == n - 1) {
                dp[i][j] = grid[i][j] + max(dp[i][j + 1], dp[i - 1][j + 1]);
            } else if (i == 0) {
                dp[i][j] = grid[i][j] + max(dp[i][j + 1], dp[i + 1][j + 1]);
            } else {
                dp[i][j] = grid[i][j] + max(dp[i][j + 1], max(dp[i - 1][j + 1], dp[i + 1][j + 1]));
            }
        }
    }

    // Find the maximum value in the first column of dp
    // array
    int maxGold = dp[0][0];
    for (int i = 1; i < n; i++) {
        if (dp[i][0] > maxGold) {
            maxGold = dp[i][0];
        }
    }

    cout << maxGold << endl;

    return 0;
}
```

Let's do a **tabular dry run** of your gold mine problem (classic DP), where the goal is to find the **maximum gold** that can be collected from **any cell in the first column to the last column**, moving only in:

- right ( $\rightarrow$ )
- right-up ( $\nearrow$ )
- right-down ( $\searrow$ )

### Given grid[4][4]:

[8, 2, 1, 6]
[6, 5, 5, 2]
[2, 1, 0, 3]
[7, 2, 2, 4]

### DP Formula:

For  $dp[i][j]$ :

- If  $j == \text{last column}$ :  $dp[i][j] = \text{grid}[i][j]$
- If  $i == 0$ : no up  $\rightarrow$  use right and right-down
- If  $i == n-1$ : no down  $\rightarrow$  use right and right-up
- Else: consider all 3  $\rightarrow$  right, right-up, right-down

### Filling dp from right to left:

We'll fill the DP matrix from column  $j = 3$  to 0.

#### Step-by-step (column by column):

i\j	0	1	2	3
0	?	?	?	6
1	?	?	?	2
2	?	?	?	3
3	?	?	?	4

#### Fill Column 2 ( $j = 2$ ):

$dp[i][2] = \text{grid}[i][2] + \max(dp[i][3], dp[i-1][3], dp[i+1][3])$

i	grid[i][2]	dp options	max	dp[i][2]
3	2	dp[3][3]=4, dp[2] [3]=3	4	6
2	0	3, 2, 4	4	4

i	grid[i][2]	dp options	max	dp[i][2]
1	5	2, 6, 3	6	11
0	1	6, 2	6	7

Fill Column 1 (j = 1):

i	grid[i][1]	dp options	max	dp[i][1]
3	2	6, 4	6	8
2	1	4, 11, 6	11	12
1	5	11, 7, 4	11	16
0	2	7, 11	11	13

Fill Column 0 (j = 0):

i	grid[i][0]	dp options	max	dp[i][0]
3	7	8, 12	12	19
2	2	12, 16, 8	16	18
1	6	16, 13, 12	16	22
0	8	13, 16	16	24

✓ Final dp Table:

i\j	0	1	2	3
0	24	13	7	6
1	22	16	11	2
2	18	12	4	3
3	19	8	6	4

□ Max Gold = max(dp[0][0], dp[1][0], dp[2][0], dp[3][0]) = 24

✓ Output:

24

Output:  
24

## Min Cost Path in C++

```
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

int main() {
    int n = 4; // Number of rows
    int m = 4; // Number of columns
    int grid[4][4] = {
        {8, 2, 1, 6},
        {6, 5, 5, 2},
        {2, 1, 0, 3},
        {7, 2, 2, 4}
    };

    // Initialize dp array
    vector<vector<int>> dp(n, vector<int>(m, 0));

    // Fill dp array from bottom-right to top-left
    for (int i = n - 1; i >= 0; i--) {
        for (int j = m - 1; j >= 0; j--) {
            if (i == n - 1 && j == m - 1) {
                dp[i][j] = grid[i][j];
            } else if (i == n - 1) {
                dp[i][j] = dp[i][j + 1] + grid[i][j];
            } else if (j == m - 1) {
                dp[i][j] = dp[i + 1][j] + grid[i][j];
            } else {
                dp[i][j] = grid[i][j] + min(dp[i][j + 1], dp[i + 1][j]);
            }
        }
    }

    // Print the minimum cost path sum
    cout << dp[0][0] << endl;

    return 0;
}
```

### Input Grid:

[8, 2, 1, 6]
[6, 5, 5, 2]
[2, 1, 0, 3]
[7, 2, 2, 4]

We're filling the  $dp[i][j]$  table from **bottom-right to top-left**.

### ✓ DP Formula Recap:

```
if (i == n - 1 && j == m - 1)
    dp[i][j] = grid[i][j];
else if (i == n - 1)
    dp[i][j] = dp[i][j + 1] + grid[i][j];
else if (j == m - 1)
    dp[i][j] = dp[i + 1][j] + grid[i][j];
else
    dp[i][j] = grid[i][j] + min(dp[i][j + 1], dp[i + 1][j]);
```

### █ DP Table (Filled from bottom-right):

Let's build  $dp[i][j]$  step by step:

**Starting from  $dp[3][3] = grid[3][3] = 4$**

Then filling right-to-left and bottom-to-top:

i \ j	0	1	2	3
0	?	?	?	?
1	?	?	?	?
2	?	?	?	?
3	15	8	6	4

Now build upward:

### Row 2:

- $dp[2][3] = grid[2][3] + dp[3][3] = 3 + 4 = 7$
- $dp[2][2] = 0 + \min(7, 6) = 6$
- $dp[2][1] = 1 + \min(6, 8) = 7$
- $dp[2][0] = 2 + \min(7, 15) = 9$

### Row 1:

- $dp[1][3] = 2 + 7 = 9$
- $dp[1][2] = 5 + \min(9, 6) = 11$
- $dp[1][1] = 5 + \min(11, 7) = 12$
- $dp[1][0] = 6 + \min(12, 9) = 15$

### Row 0:

- $dp[0][3] = 6 + 9 = 15$
- $dp[0][2] = 1 + \min(15, 11) = 12$

- $dp[0][1] = 2 + \min(12, 12) = 14$
- $dp[0][0] = 8 + \min(14, 15) = 22$

✓ Final DP Table:

i\j	0	1	2	3
0	22	14	12	15
1	15	12	11	9
2	9	7	6	7
3	15	8	6	4

☛ Output:

22

Output:

22

## Paint Houses in C++

```
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

int main() {
    // Input array representing costs to paint each
    // house with three colors
    vector<vector<int>> arr = {{1, 5, 7}, {5, 8, 4}, {3, 2,
    9}, {1, 2, 4}};
    int n = arr.size(); // Number of houses

    // Initialize dp array
    vector<vector<long long>> dp(n, vector<long
    long>(3, 0));

    // Base case: First row initialization
    dp[0][0] = arr[0][0];
    dp[0][1] = arr[0][1];
    dp[0][2] = arr[0][2];

    // Fill dp array from second row onwards
    for (int i = 1; i < n; i++) {
        dp[i][0] = arr[i][0] + min(dp[i - 1][1], dp[i - 1][2]);
        dp[i][1] = arr[i][1] + min(dp[i - 1][0], dp[i - 1][2]);
        dp[i][2] = arr[i][2] + min(dp[i - 1][0], dp[i - 1][1]);
    }

    // Find the minimum cost to paint all houses
    long long ans = min(dp[n - 1][0], min(dp[n - 1][1],
    dp[n - 1][2]));

    // Output the minimum cost
    cout << ans << endl;

    return 0;
}
```

Output:-  
8

### Input Matrix (Cost of painting houses):

House 0: [1, 5, 7]  
 House 1: [5, 8, 4]  
 House 2: [3, 2, 9]  
 House 3: [1, 2, 4]

We denote the colors as:

- 0 → Red
- 1 → Blue
- 2 → Green

### DP Table Filling Explanation:

House	dp[i][0] (Red)	dp[i][1] (Blue)	dp[i][2] (Green)
0	1	5	7
1	$5 + \min(5, 7) = 10$	$8 + \min(1, 7) = 9$	$4 + \min(1, 5) = 5$
2	$3 + \min(9, 5) = 8$	$2 + \min(10, 5) = 7$	$9 + \min(10, 9) = 18$
3	$1 + \min(7, 18) = 8$	$2 + \min(8, 18) = 10$	$4 + \min(8, 7) = 11$

### Final DP Table:

House	Red	Blue	Green
0	1	5	7
1	10	9	5
2	8	7	18
3	8	10	11

### Output:

The minimum total cost is:

$$\min(8, 10, 11) = 8$$

## Target sum Subset in C++

```
#include <iostream>
#include <vector>
using namespace std;

bool targetSumSubsets(vector<int>& arr, int target) {
    int n = arr.size();
    vector<vector<bool>> dp(n + 1, vector<bool>(target + 1, false));

    for (int i = 0; i <= n; i++) {
        for (int j = 0; j <= target; j++) {
            if (i == 0 && j == 0) {
                dp[i][j] = true;
            } else if (i == 0) {
                dp[i][j] = false;
            } else if (j == 0) {
                dp[i][j] = true;
            } else {
                if (dp[i - 1][j]) {
                    dp[i][j] = true;
                } else {
                    int val = arr[i - 1];
                    if (j >= val && dp[i - 1][j - val]) {
                        dp[i][j] = true;
                    }
                }
            }
        }
    }
    return dp[n][target];
}
```

```
int main() {
    vector<int> arr = {4, 2, 7, 1, 3};
    int target = 10;

    if (targetSumSubsets(arr, target)) {
        cout << "True" << endl;
    } else {
        cout << "False" << endl;
    }

    return 0;
}
```

We have **array**:

$\text{arr} = \{4, 2, 7, 1, 3\}$ ,  $\text{target} = 10$

We create a **dp table of size  $(n+1) \times (\text{target}+1)$** :  
 $\text{dp}[i][j] \rightarrow i$  is the first  $i$  elements,  $j$  is the sum.

### Initial Table (Before Processing)

i \ j	0	1	2	3	4	5	6	7	8	9	10
0	T	F	F	F	F	F	F	F	F	F	F
1											
2											
3											
4											
5											

- **dp[0][0] = true** → A sum of 0 can be achieved with an empty subset.
- **dp[0][j] = false** for  $j > 0$  → No subset can sum up to a positive number with zero elements.

### Step 2: Fill the Table

We iterate through  $i = 1$  to  $n$ , updating  $\text{dp}[i][j]$ .

#### Processing $\text{arr}[0] = 4$

We consider only element 4.

- $\text{dp}[1][4] = \text{true}$  (We can form sum 4 using  $\{4\}$ )

i \ j	0	1	2	3	4	5	6	7	8	9	10
0	T	F	F	F	F	F	F	F	F	F	F
1	T	F	F	F	T	F	F	F	F	F	F

#### Processing $\text{arr}[1] = 2$

Now considering  $\{4, 2\}$ :

- $\text{dp}[2][2] = \text{true}$  (Subset  $\{2\}$ )
- $\text{dp}[2][4] = \text{true}$  (Subset  $\{4\}$ )
- $\text{dp}[2][6] = \text{true}$  (Subset  $\{4, 2\}$ )

i\j	0	1	2	3	4	5	6	7	8	9	10
0	T	F	F	F	F	F	F	F	F	F	F
1	T	F	F	F	T	F	F	F	F	F	F
2	T	F	T	F	T	F	T	F	F	F	F

### Processing arr[2] = 7

Now considering {4,2,7}:

- dp[3][7] = true (Subset {7})
- dp[3][9] = true (Subset {2,7})
- dp[3][10] = true (Subset {4,2,7})

i\j	0	1	2	3	4	5	6	7	8	9	10
0	T	F	F	F	F	F	F	F	F	F	F
1	T	F	F	F	T	F	F	F	F	F	F
2	T	F	T	F	T	F	T	F	F	F	F
3	T	F	T	F	T	F	T	T	F	T	T

### Processing arr[3] = 1

Now considering {4,2,7,1}:

- dp[4][1] = true (Subset {1})
- dp[4][3] = true (Subset {2,1})
- dp[4][5] = true (Subset {4,1})
- dp[4][8] = true (Subset {7,1})

i\j	0	1	2	3	4	5	6	7	8	9	10
0	T	F	F	F	F	F	F	F	F	F	F
1	T	F	F	F	T	F	F	F	F	F	F
2	T	F	T	F	T	F	T	F	F	F	F
3	T	F	T	F	T	F	T	T	F	T	T
4	T	T	T	T	T	T	T	T	T	T	T

### Processing arr[4] = 3

Including 3 confirms all sums, but **dp[5][10]** remains **true**.

Final Answer

	Since $dp[5][10] = \text{true}$ , we <b>return true</b> , meaning a subset exists with the sum <b>10</b> .  Output: True
--	--

Output:-  
True  
 $dp[n][\text{target}]$  is  $dp[5][10] = \text{true}$

## Tiling with Dominoes in C++

```
#include <iostream>
#include <vector>

using namespace std;

int main() {
    int n = 2;

    vector<int> dp(n + 1);
    dp[1] = 1;
    dp[2] = 2;

    for (int i = 3; i <= n; i++) {
        dp[i] = dp[i - 1] + dp[i - 2];
    }

    cout << dp[n] << endl;

    return 0;
}
```

Given:

- $n = 2$

We create a  $dp$  vector of size  $n+1 = 3$  and initialize the base cases:

- $dp[1] = 1$
- $dp[2] = 2$

### Initial dp Table:

i\dp	0	1	2
i=0	0		
i=1		1	
i=2			2

At this point:

- $dp[0] = 0$  (this entry is not used)
- $dp[1] = 1$
- $dp[2] = 2$

### For Loop Execution:

The `for` loop iterates from  $i = 3$  to  $n$ . But, since  $n = 2$ , the loop condition  $i = 3 \leq 2$  is **false**, so the loop doesn't run.

### Final dp Table:

The table remains unchanged from initialization:

i\dp	0	1	2
i=0	0		
i=1		1	
i=2			2

### Final Output:

The program prints  $dp[n]$ , where  $n = 2$ , so  $dp[2] = 2$  is printed.

Output:-  
2

## All paths minimum jumps in C++

```
#include <iostream>
#include <climits>
#include <queue>
using namespace std;

class Pair {
public:
    int i, s, j;
    string psf;

    Pair(int i, int s, int j, string psf) {
        this->i = i;
        this->s = s;
        this->j = j;
        this->psf = psf;
    }
};

void solution(const int arr[], int n) {
    int dp[n];
    fill_n(dp, n, INT_MAX);
    dp[n - 1] = 0;

    for (int i = n - 2; i >= 0; i--) {
        int steps = arr[i];
        int min_steps = INT_MAX;

        for (int j = 1; j <= steps && i + j < n; j++) {
            if (dp[i + j] != INT_MAX && dp[i + j] <
min_steps) {
                min_steps = dp[i + j];
            }
        }

        if (min_steps != INT_MAX) {
            dp[i] = min_steps + 1;
        }
    }

    cout << dp[0] << endl;

    queue<Pair> q;
    q.emplace(0, arr[0], dp[0], "0");

    while (!q.empty()) {
        Pair rem = q.front();
        q.pop();

        if (rem.j == 0) {
            cout << rem.psf << "." << endl;
        }

        for (int j = 1; j <= rem.s && rem.i + j < n;
j++) {
            int ci = rem.i + j;
            if (dp[ci] != INT_MAX && dp[ci] ==
rem.j - 1) {
                q.emplace(ci, arr[ci], dp[ci], rem.psf
+ "->" + to_string(ci));
            }
        }
    }
}
```

### Dry Run:

#### Step 1: Calculate the dp array (minimum jumps to reach the end from each index)

The dp array keeps track of the minimum number of jumps required to reach the last index from any given index. Let's calculate the dp array starting from the last index (since we know that  $dp[n-1] = 0$  as no jumps are needed from the last index):

- $dp[9] = 0$  (since we're already at the last index).
- $dp[8] = INT\_MAX$  (can't reach the last index from index 8, because there are no valid jumps).
- $dp[7] = 1$  (one jump to index 9, because  $arr[7] = 2$  allows jumping to index 9).
- $dp[6] = 1$  (one jump to index 9, because  $arr[6] = 4$  allows jumping to index 9).
- $dp[5] = 2$  (minimum of  $dp[6] + 1$  and  $dp[7] + 1$ , so  $\min(1+1, 1+1) = 2$ ).
- $dp[4] = 2$  (minimum of  $dp[5] + 1$  and  $dp[6] + 1$ , so  $\min(2+1, 1+1) = 2$ ).
- $dp[3] = 2$  (minimum of  $dp[4] + 1$  and  $dp[5] + 1$ , so  $\min(2+1, 2+1) = 2$ ).
- $dp[2] = 3$  (can't jump to a valid position from here).
- $dp[1] = 3$  (same as above, can't jump to a valid position).
- $dp[0] = 4$  (minimum of  $dp[1] + 1$ ,  $dp[2] + 1$ , and  $dp[3] + 1$ , so  $\min(3+1, 3+1, 2+1) = 4$ ).

Thus, the dp array will look like this:

$$dp = \{4, 3, 3, 2, 2, 2, 1, 1, INT\_MAX, 0\}$$

#### Step 2: Generate paths using BFS

Next, we use BFS to generate all valid paths from the start (index 0) to the end (index 9) using the minimum number of jumps ( $dp[0] = 4$ ).

We initialize the queue with the first index 0 and process each index in the queue, exploring all possible jumps from that index:

1. Start from index 0, jump to index 3 (because  $dp[3] = 2$  and  $dp[0] = dp[3] + 1$ ).
2. From index 3, jump to index 5 (because  $dp[5] = 2$  and  $dp[3] = dp[5] + 1$ ).
3. From index 5, jump to index 6 (because  $dp[6] = 1$  and  $dp[5] = dp[6] + 1$ ).
4. From index 6, jump to index 9 (because  $dp[9] = 0$  and  $dp[6] = dp[9] + 1$ ).

This gives the path: 0 -> 3 -> 5 -> 6 -> 9.

```

    }
}

int main() {
    const int arr[] = {3, 3, 0, 2, 1, 2, 4, 2, 0, 0};
    int n = sizeof(arr) / sizeof(arr[0]);
    solution(arr, n);
    return 0;
}

```

Similarly, another valid path is:

1. Start from index 0, jump to index 3.
2. From index 3, jump to index 5.
3. From index 5, jump to index 7 (because  $dp[7] = 1$  and  $dp[5] = dp[7] + 1$ ).
4. From index 7, jump to index 9 (because  $dp[9] = 0$ ).

This gives the path: 0 -> 3 -> 5 -> 7 -> 9.

### **Step 3: Final Output**

The correct output should be:

4  
0->3->5->6->9.  
0->3->5->7->9.

**Output:-**

4  
0->3->5->6->9.  
0->3->5->7->9.

## Arithmetic Slices in C++

```
#include <iostream>
#include <vector>
using namespace std;

int solution(const vector<int>& arr) {
    vector<int> dp(arr.size(), 0);
    //vector<int> dp;
    int ans = 0;
    for (size_t i = 2; i < arr.size(); i++) {
        if (arr[i] - arr[i - 1] == arr[i - 1] - arr[i - 2]) {
            dp[i] = dp[i - 1] + 1;
            ans += dp[i];
        }
    }
    return ans;
}

int main() {
    vector<int> arr = {2, 5, 9, 12, 15, 18, 22, 26, 30, 34,
    36, 38, 40, 41};
    cout << solution(arr) << endl;
    return 0;
}
```

### Given Input

vector<int> arr = {2, 5, 9, 12, 15, 18, 22, 26, 30, 34, 36, 38, 40, 41};

- **Size of array:** n = 14

### Step-by-Step Dry Run

We'll track how  $dp[i]$  and  $ans$  evolve.

### Initialization

Index (i)	arr[i]	dp[i]	ans (Sum of dp[i])
0	2	-	-
1	5	-	-

### Loop Execution ( $i = 2$ to $i = 13$ )

i	arr[i]	Check Condition $arr[i] - arr[i-1] == arr[i-1] - arr[i-2]$	dp[i] Calculation	ans Update
2	9	$(9 - 5) == (5 - 2)$ → 4 == 3 ✗	$dp[2] = 0$	ans = 0
3	12	$(12 - 9) == (9 - 5)$ → 3 == 4 ✗	$dp[3] = 0$	ans = 0
4	15	$(15 - 12) == (12 - 9)$ → 3 == 3 ✓	$dp[4] = dp[3] + 1 = 1$	ans = 1
5	18	$(18 - 15) == (15 - 12)$ → 3 == 3 ✓	$dp[5] = dp[4] + 1 = 2$	ans = 3
6	22	$(22 - 18) == (18 - 15)$ → 4 == 3 ✗	$dp[6] = 0$	ans = 3
7	26	$(26 - 22) == (22 - 18)$ → 4 == 4 ✓	$dp[7] = dp[6] + 1 = 1$	ans = 4
8	30	$(30 - 26) == (26 - 22)$ → 4 == 4 ✓	$dp[8] = dp[7] + 1 = 2$	ans = 6
9	34	$(34 - 30) == (30 - 26)$ → 4 == 4 ✓	$dp[9] = dp[8] + 1 = 3$	ans = 9
10	36	$(36 - 34) == (34 - 30)$ → 2 == 4 ✗	$dp[10] = 0$	ans = 9
11	38	$(38 - 36) == (36 - 34)$ → 2 == 2 ✓	$dp[11] = dp[10] + 1 = 1$	ans = 10
12	40	$(40 - 38) == (38 - 36)$ → 2 == 2 ✓	$dp[12] = dp[11] + 1 = 2$	ans = 12
13	41	$(41 - 40) == (40 - 38)$ → 1 == 2 ✗	$dp[13] = 0$	ans = 12

**Final dp Table**

<b>Index (i)</b>	<b>arr[i]</b>	<b>dp[i]</b>	<b>ans (Sum of dp[i])</b>
0	2	-	-
1	5	-	-
2	9	0	0
3	12	0	0
4	15	1	1
5	18	2	3
6	22	0	3
7	26	1	4
8	30	2	6
9	34	3	9
10	36	0	9
11	38	1	10
12	40	2	12
13	41	0	12

**Final Output**

12

Output:-  
12

## Balanced Parenthesis in C++

```
#include <iostream>
#include <vector>
using namespace std;

int main() {
    int n = 5;
    vector<int> dp(n + 1, 0);
    dp[0] = 1;
    dp[1] = 1;

    for (int i = 2; i <= n; i++) {
        int inside = i - 1;
        int outside = 0;
        while (inside >= 0) {
            dp[i] += dp[inside] * dp[outside];
            inside--;
            outside++;
        }
    }

    for (int i = 0; i < dp.size(); i++) {
        cout << dp[i] << " ";
    }

    // char c = 'b';
    // cout << (c - '0') << endl;
}

return 0;
}
```

### Dry Run with Table

Let's analyze **step-by-step calculations for n = 5.**

#### Initialization

i	inside	outside	Computation	dp[i]
0	-	-	dp[0] = 1	1
1	-	-	dp[1] = 1	1

#### Filling dp Array

i	inside	outside	Computation (dp[i] += dp[inside] * dp[outside])	dp[i]
2	1	0	dp[2] += dp[1] * dp[0] = 1 * 1	1
	0	1	dp[2] += dp[0] * dp[1] = 1 * 1	2
3	2	0	dp[3] += dp[2] * dp[0] = 2 * 1	2
	1	1	dp[3] += dp[1] * dp[1] = 1 * 1	3
	0	2	dp[3] += dp[0] * dp[2] = 1 * 2	5
4	3	0	dp[4] += dp[3] * dp[0] = 5 * 1	5
	2	1	dp[4] += dp[2] * dp[1] = 2 * 1	7
	1	2	dp[4] += dp[1] * dp[2] = 1 * 2	9
	0	3	dp[4] += dp[0] * dp[3] = 1 * 5	14
5	4	0	dp[5] += dp[4] * dp[0] = 14 * 1	14
	3	1	dp[5] += dp[3] * dp[1] = 5 * 1	19
	2	2	dp[5] += dp[2] * dp[2] = 2 * 2	23
	1	3	dp[5] += dp[1] * dp[3] = 1 * 5	28

i	inside	outside	Computation ( $dp[i] += dp[inside] * dp[outside]$ )	$dp[i]$
0	4		$dp[5] += dp[0] * dp[4] = 1 * 14$	42

**Final dp Array Output**  
1 1 2 5 14 42

**Final Output ( $dp[5]$ )**  
42

This means **42 unique BSTs** can be formed using 5 nodes.

Output:-  
1 1 2 5 14 42

### Burst Balloons In C++

```
#include <iostream>
#include <climits>
using namespace std;

int sol(int arr[], int n) {
    int dp[n][n];

    // Initialize the dp array with zeros
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            dp[i][j] = 0;
        }
    }

    for (int g = 0; g < n; g++) {
        for (int i = 0, j = g; j < n; i++, j++) {
            int maxCoins = INT_MIN;
            for (int k = i; k <= j; k++) {
                int left = (k == i) ? 0 : dp[i][k - 1];
                int right = (k == j) ? 0 : dp[k + 1]
[j];
                int val = (i == 0 ? 1 : arr[i - 1]) *
arr[k] * (j == n - 1 ? 1 : arr[j + 1]);
                int total = left + right + val;
                maxCoins = max(maxCoins,
total);
            }
            dp[i][j] = maxCoins;
        }
    }
    return dp[0][n - 1];
}

int main() {
    int arr[] = {2, 3, 5};
    int n = sizeof(arr) / sizeof(arr[0]);
    cout << sol(arr, n) << endl;
    return 0;
}
```

### Dry Run of sol(arr, 3)

#### Given Input:

arr[] = {2, 3, 5}  
n = 3

#### Step 1: Initialize DP Table (dp[n][n])

dp = { {0, 0, 0},  
 {0, 0, 0},  
 {0, 0, 0} }

#### Step 2: Iterate Over Gaps (g)

##### Gap g = 0 (Single Balloons)

For g = 0, each cell dp[i][i] represents bursting a single balloon.

i	j	k (only choic e)	Left	Right	Value	dp[i] [j]
0	0	0	0	0	$1 \times 2 \times 3 = 6$	6
1	1	1	0	0	$2 \times 3 \times 5 = 30$	30
2	2	2	0	0	$3 \times 5 \times 1 = 15$	15

#### Updated DP Table:

dp = { {6, 0, 0},  
 {0, 30, 0},  
 {0, 0, 15} }

##### Gap g = 1 (Two Balloons)

Now we consider **two consecutive balloons**.

**Case (i=0, j=1):**

k	Left	Right	Value	Total
0	0	30	$1 \times 2 \times 5 = 10$	40
1	6	0	$1 \times 3 \times 5 = 15$	21

$$dp[0][1] = \max(40, 21) = 40$$

**Case (i=1, j=2):**

k	Left	Rig ht	Value	Total
1	0	15	$2 \times 3 \times 1 = 6$	21
2	30	0	$2 \times 5 \times 1 = 10$	40

$$dp[1][2] = \max(21, 40) = 40$$

**Updated DP Table:**

$$dp = \{ \{6, 40, 0\}, \\ \{0, 30, 40\}, \\ \{0, 0, 15\} \}$$

**Gap g = 2 (Full Array)**

Now we consider the **entire array** (i=0, j=2).

k	Left (dp[0] [k-1])	Right (dp[k+1] [2])	Value	Total
0	0	40	$1 \times 2 \times 1 = 2$	42

<b>k</b>	<b>Left (dp[0] [k-1])</b>	<b>Right (dp[k+1] [2])</b>	<b>Value</b>	<b>Total</b>
1	6	15	$1 \times 3 \times 1 = 3$	24
2	40	0	$1 \times 5 \times 1 = 5$	45

$$dp[0][2] = \max(42, 24, 45) = 45$$

#### **Final DP Table:**

$$dp = \{ \{6, 40, 45\}, \\ \{0, 30, 40\}, \\ \{0, 0, 15\} \}$$

#### **Final Answer:**

The function returns  $dp[0][n-1] = dp[0][2] = 45$ .

#### **Final Output:**

45

Output:-  
45

## Catalan in C++

```
#include <iostream>
using namespace std;

int main() {
    int n = 6;
    int dp[n];
    dp[0] = 1;
    dp[1] = 1;

    for (int i = 2; i < n; i++) {
        dp[i] = 0;
        for (int j = 0; j < i; j++) {
            dp[i] += dp[j] * dp[i - j - 1];
        }
    }

    for (int i = 0; i < n; i++) {
        cout << dp[i] << " ";
    }

    return 0;
}
```

### Step-by-Step Execution

#### Initialization:

$dp[0] = 1;$   
 $dp[1] = 1;$

#### Iteration Table for $dp[2]$ to $dp[5]$

i	j	Computation	dp[i]
2	0	$dp[0] \times dp[1] = 1 \times 1 = 1$	1
2	1	$dp[1] \times dp[0] = 1 \times 1 = 1$	2

**Final:**  $dp[2] = 2$

i	j	Computation	dp[i]
3	0	$dp[0] \times dp[2] = 1 \times 2 = 2$	2
3	1	$dp[1] \times dp[1] = 1 \times 1 = 1$	3
3	2	$dp[2] \times dp[0] = 2 \times 1 = 2$	5

**Final:**  $dp[3] = 5$

i	j	Computation	dp[i]
4	0	$dp[0] \times dp[3] = 1 \times 5 = 5$	5
4	1	$dp[1] \times dp[2] = 1 \times 2 = 2$	7
4	2	$dp[2] \times dp[1] = 2 \times 1 = 2$	9
4	3	$dp[3] \times dp[0] = 5 \times 1 = 5$	14

**Final:**  $dp[4] = 14$

i	j	Computation	dp[i]
5	0	$dp[0] \times dp[4] = 1 \times 14 = 14$	14
5	1	$dp[1] \times dp[3] = 1 \times 5 = 5$	19
5	2	$dp[2] \times dp[2] = 2 \times 2 = 4$	23
5	3	$dp[3] \times dp[1] = 5 \times 1 = 5$	28
5	4	$dp[4] \times dp[0] = 14 \times 1 = 14$	42

**Final:**  $dp[5] = 42$

#### Final DP Array:

$dp[] = \{1, 1, 2, 5, 14, 42\}$

#### Final Output:

1 1 2 5 14 42

Output:-

1	1	2	5	14	42
---	---	---	---	----	----

### Count Distinct Subsequence C++

```
#include <iostream>
#include <unordered_map>
using namespace std;

int countDistinctSubsequences(const string& str) {
    int n = str.length();
    int dp[n + 1];
    dp[0] = 1; // Empty subsequence

    unordered_map<char, int> lastOccurrence;

    for (int i = 1; i <= n; i++) {
        dp[i] = 2 * dp[i - 1];
        char ch = str[i - 1];
        if (lastOccurrence.find(ch) != lastOccurrence.end()) {
            int j = lastOccurrence[ch];
            dp[i] -= dp[j - 1];
        }
        lastOccurrence[ch] = i;
    }
    return dp[n] - 1;
}

int main() {
    string str = "abc";
    cout << countDistinctSubsequences(str) << endl;
    return 0;
}
```

### Dry Run with Input "abc"

#### Initialization:

str = "abc";  
n = 3;  
dp[0] = 1; // Empty subsequence  
lastOccurrence = {} // Initially empty

#### Iteration Table

i	str[i-1]	dp[i] Calculation	dp[i] Value	lastOccurrence Update
1	'a'	dp[1]=2×dp[0]=2×1	2	{'a': 1}
2	'b'	dp[2]=2×dp[1]=2×2	4	{'a': 1, 'b': 2}
3	'c'	dp[3]=2×dp[2]=2×4	8	{'a': 1, 'b': 2, 'c': 3}

#### Final Calculation

Result=dp[n]-1=8-1=7

(The -1 removes the empty subsequence.)

#### Final Output

7

The distinct non-empty subsequences of "abc":

a, b, c, ab, ac, bc, abc

Output:-

7

### Count Palindromic Subsequence C++

```
#include <iostream>
#include <string>
using namespace std;

int
countPalindromicSubseq(const
string& str) {
    int n = str.length();
    int dp[n][n] = {0}; //
Initialize the 2D array

    for (int g = 0; g < n; g++) {
        for (int i = 0, j = g; j < n;
i++, j++) {
            if (g == 0) {
                dp[i][j] = 1;
            } else if (g == 1) {
                dp[i][j] = (str[i] ==
str[j]) ? 2 : 1;
            } else {
                if (str[i] == str[j]) {
                    dp[i][j] = dp[i][j - 1] + dp[i + 1][j] + 1;
                } else {
                    dp[i][j] = dp[i][j - 1] + dp[i + 1][j] - dp[i + 1][j - 1];
                }
            }
        }
    }

    return dp[0][n - 1];
}

int main() {
    string str = "abccbc";
    cout <<
    countPalindromicSubseq(str)
    << endl;
    return 0;
}
```

#### Step 1: Single Character ( $g = 0$ )

Each **single character** is a palindrome:

$$dp[i][i] = 1$$

#### Updated DP Table:

1	0	0	0	0	0
0	1	0	0	0	0
0	0	1	0	0	0
0	0	0	1	0	0
0	0	0	0	1	0
0	0	0	0	0	1

#### Step 2: Two-Character Substrings ( $g = 1$ )

i	j	Substring	str[i] == str[j]?	dp[i][j]
0	1	"ab"	✗	1
1	2	"bc"	✗	1
2	3	"cc"	✓	2
3	4	"cb"	✗	1
4	5	"bc"	✗	1

#### Updated DP Table:

1	1	0	0	0	0
0	1	1	0	0	0
0	0	1	2	0	0
0	0	0	1	1	0
0	0	0	0	1	1
0	0	0	0	0	1

#### Step 3: Three-Character Substrings ( $g = 2$ )

i	j	Substring	str[i] == str[j]?	Formula Used	dp[i][j]
0	2	"abc"	✗	$dp[0][2] = dp[0][1] + dp[1][2] - dp[1][1]$	2
1	3	"bcc"	✗	$dp[1][3] = dp[1][2] + dp[2][3] - dp[2][2]$	3
2	4	"ccb"	✗	$dp[2][4] = dp[2][3] + dp[3][4] - dp[3][3]$	3
3	5	"cbc"	✓	$dp[3][5] = dp[3][4] + dp[4][5] + 1$	3

#### Updated DP Table:

1	1	2	0	0	0
0	1	1	3	0	0
0	0	1	2	3	0
0	0	0	1	1	3
0	0	0	0	1	1

0 0 0 0 0 1

#### Step 4: Four-Character Substrings (g = 3)

i	j	Substring	str[i] == str[j]?	Formula Used	dp[i][j]
0	3	"abcc"	✗	$dp[0][3] = dp[0][2] + dp[1][3] - dp[1][2]$	4
1	4	"beccb"	✓	$dp[1][4] = dp[1][3] + dp[2][4] + 1$	7
2	5	"ccbc"	✓	$dp[2][5] = dp[2][4] + dp[3][5] + 1$	7

#### Updated DP Table:

1 1 2 4 0 0  
0 1 1 3 7 0  
0 0 1 2 3 7  
0 0 0 1 1 3  
0 0 0 0 1 1  
0 0 0 0 0 1

#### Step 4: Four-Character Substrings (g = 4)

i	j	Substring	str[i] == str[j]?	Formula Used	dp[i][j]
0	4	"abccb"	✗	$dp[0][4] = dp[0][3] + dp[1][4] - dp[1][3]$	5
1	5	"bccbc"	✓	$dp[1][5] = dp[1][4] + dp[2][5] + 1$	9

#### Updated DP Table:

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

#### Step 5: Final Computation (g = 5)

$$\begin{aligned}
 dp[0][5] &= dp[0][4] + dp[1][5] - dp[1][4] \\
 dp[0][5] &= dp[0][4] + dp[1][5] - dp[1][4] \\
 dp[0][5] &= 7 + 7 - 5 = 9 \\
 dp[0][5] &= 9
 \end{aligned}$$

Output:-  
9

### Count Distinct Subsequence C++

```
#include <iostream>
using namespace std;

int countValleysAndMountains(int n) {
    int dp[n + 1] = {0}; // Initialize the array with zeros
    dp[0] = 1; // Base case: empty sequence
    dp[1] = 1; // Sequence of length 1: either V or M

    for (int i = 2; i <= n; i++) {
        int valleys = 0;
        int mountains = i - 1;

        while (mountains >= 0) {
            dp[i] += dp[valleys] * dp[mountains];
            valleys++;
            mountains--;
        }
    }

    return dp[n];
}

int main() {
    int n = 5;
    cout << countValleysAndMountains(n) << endl;
    return 0;
}
```

Output:-  
42

#### Step-by-Step Calculation

i	dp[i] Computation	dp[i] Value
0	dp[0] = 1	1
1	dp[1] = dp[0] * dp[0]	1
2	dp[2] = dp[0] * dp[1] + dp[1] * dp[0]	2
3	dp[3] = dp[0] * dp[2] + dp[1] * dp[1] + dp[2] * dp[0]	5
4	dp[4] = dp[0] * dp[3] + dp[1] * dp[2] + dp[2] * dp[1] + dp[3] * dp[0]	14
5	dp[5] = dp[0] * dp[4] + dp[1] * dp[3] + dp[2] * dp[2] + dp[3] * dp[1] + dp[4] * dp[0]	42

Final Output  
42

## Edit Distance C++

```
#include <iostream>
#include <string>
#include <algorithm>

using namespace std;

int main() {
    string s1 = "cat";
    string s2 = "cut";
    int m = s1.length();
    int n = s2.length();

    int dp[m + 1][n + 1];

    // Base cases
    for (int i = 0; i <= m; i++) dp[i][0] = i; // Deleting all characters
    for (int j = 0; j <= n; j++) dp[0][j] = j; // Inserting all characters

    // Fill the DP table
    for (int i = 1; i <= m; i++) {
        for (int j = 1; j <= n; j++) {
            if (s1[i - 1] == s2[j - 1]) {
                dp[i][j] = dp[i - 1][j - 1]; // No operation needed
            } else {
                dp[i][j] = 1 + min({dp[i - 1][j - 1], // Replace
                                    dp[i - 1][j], // Delete
                                    dp[i][j - 1]}); // Insert
            }
        }
    }

    cout << dp[m][n] << endl; // Output the minimum edit distance
}

return 0;
}
```

Dry Run ( $s_1 = \text{"cat"}$ ,  $s_2 = \text{"cut"}$ )

### Step 1: Initialize the DP Table

The **first row** (when  $s_1$  is empty) represents **insertions**, and the **first column** (when  $s_2$  is empty) represents **deletions**.

i\j	0	1	2	3
0	0	1	2	3
1	1	-	-	-
2	2	-	-	-
3	3	-	-	-

### Step 2: Fill the DP Table

#### Iteration 1 ( $i=1$ , $s_1=\text{"c"}$ ):

- $j=1$ ,  $s_2=\text{"c"}$  → **Same character, copy diagonal** →  $dp[1][1] = dp[0][0] = 0$
- $j=2$ ,  $s_2=\text{"cu"}$  → **Insert 'u'** →  $dp[1][2] = \min(\text{Replace:1}, \text{Delete:2}, \text{Insert:0}) + 1 = 1$
- $j=3$ ,  $s_2=\text{"cut"}$  → **Insert 't'** →  $dp[1][3] = \min(\text{Replace:2}, \text{Delete:3}, \text{Insert:1}) + 1 = 2$

i\j	0	1	2	3
0	0	1	2	3
1	1	0	1	2
2	2	-	-	-
3	3	-	-	-

#### Iteration 2 ( $i=2$ , $s_1=\text{"ca"}$ ):

- $j=1$ ,  $s_2=\text{"c"}$  → **Delete 'a'** →  $dp[2][1] = \min(\text{Replace:1}, \text{Delete:0}, \text{Insert:2}) + 1 = 1$
- $j=2$ ,  $s_2=\text{"cu"}$  → **Replace 'a' with 'u'** →  $dp[2][2] = \min(\text{Replace:0}, \text{Delete:1}, \text{Insert:1}) + 1 = 1$
- $j=3$ ,  $s_2=\text{"cut"}$  → **Insert 't'** →  $dp[2][3] = \min(\text{Replace:1}, \text{Delete:2}, \text{Insert:1}) + 1 = 2$

i\j	0	1	2	3
0	0	1	2	3
1	1	0	1	2
2	2	-	-	-
3	3	-	-	-

i\j	0	1	2	3
1	1	0	1	2
2	2	1	1	2
3	3	-	-	-

### Iteration 3 ( $i=3$ , $s1="cat"$ ):

- $j=1$ ,  $s2="c" \rightarrow \text{Delete 'at'}$   $\rightarrow dp[3][1] = \min(\text{Replace:2}, \text{Delete:1}, \text{Insert:3}) + 1 = 2$
- $j=2$ ,  $s2="cu" \rightarrow \text{Delete 't'}$   $\rightarrow dp[3][2] = \min(\text{Replace:1}, \text{Delete:1}, \text{Insert:2}) + 1 = 2$
- $j=3$ ,  $s2="cut" \rightarrow \text{Replace 'a' with 'u'}$   $\rightarrow dp[3][3] = dp[2][2] = 1$  (since 'c' and 't' match)

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

### Step 3: Output the Result

↙ The **minimum edit distance** is  $dp[3][3] = 1$ , meaning we need **one operation (replace 'a' with 'u')** to convert "cat" to "cut".

Output:-  
1

## Egg drop C++

```
#include <iostream>
#include <climits>
using namespace std;

int eggDrop(int n, int k) {
    // Initialize a 2D array for DP table
    int dp[n + 1][k + 1]; // Array with (n + 1) rows and
    (k + 1) columns
    for (int i = 0; i <= n; i++) {
        for (int j = 0; j <= k; j++) {
            dp[i][j] = 0;
        }
    }

    // Fill the DP table
    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= k; j++) {
            if (i == 1) {
                dp[i][j] = j; // If only one egg is available, we
                need j trials
            } else if (j == 1) {
                dp[i][j] = 1; // If only one floor is there, one
                trial needed
            } else {
                int minDrops = INT_MAX;
                // Check all floors from 1 to j to find the
                minimum drops needed
                for (int floor = 1; floor <= j; floor++) {
                    int breaks = dp[i - 1][floor - 1]; // Egg
                    breaks, check below floors
                    int survives = dp[i][j] - floor; // Egg
                    survives, check above floors
                    int maxDrops = 1 + max(breaks,
                    survives); // Maximum drops needed in worst case
                    minDrops = min(minDrops, maxDrops); //
                    Minimum drops to find the critical floor
                }
                dp[i][j] = minDrops;
            }
        }
    }

    return dp[n][k]; // Return the minimum drops
    needed
}

int main() {
    int n = 4; // Number of eggs
    int k = 2; // Number of floors

    cout << eggDrop(n, k) << endl; // Output the
    minimum drops required
    return 0;
}
```

Step 1: Understanding the DP State

- $dp[i][j]$  = **Minimum number of trials** needed to find the critical floor with  $i$  eggs and  $j$  floors.
- If we have **1 egg**, we must check **each floor one by one**  $\rightarrow dp[1][j] = j$
- If we have **1 floor**, only **1 trial** is needed  $\rightarrow dp[i][1] = 1$

Step 2: Dry Run for  $n = 4$  (eggs),  $k = 2$  (floors)

We build the **DP table** from  $dp[1][1]$  up to  $dp[4][2]$ .

### Step 2.1: Initialize Base Cases

<b>dp[i][j]</b>	<b>0 Floors</b>	<b>1 Floor</b>	<b>2 Floors</b>
<b>0 Eggs</b>	0	0	0
<b>1 Egg</b>	0	1	2
<b>2 Eggs</b>	0	1	?
<b>3 Eggs</b>	0	1	?
<b>4 Eggs</b>	0	1	?

### Step 2.2: Fill DP Table Using Recurrence

For  $dp[i][j]$ , we check all floors  $f$  from 1 to  $j$ , and take the worst-case minimum:

$$dp[i][j] = 1 + \min_{f=1}^j \max(dp[i-1][f-1], dp[i][j-f])$$

#### Filling for $dp[2][2]$

- Try dropping from **floor 1**:
  - If **breaks**, check below:  $dp[1][0] = 0$
  - If **survives**, check above:  $dp[2][1] = 1$
  - **Max**  $\rightarrow \max(0,1) + 1 = 2$
- Try dropping from **floor 2**:
  - If **breaks**, check below:  $dp[1][1] = 1$
  - If **survives**, check above:  $dp[2][0] = 0$
  - **Max**  $\rightarrow \max(1,0) + 1 = 2$
- **Final Result:**  $dp[2][2] = \min(2,2) = 2$

#### Filling for $dp[3][2]$

- Try dropping from **floor 1**:
  - If **breaks**, check below:  $dp[2][0] = 0$
  - If **survives**, check above:  $dp[3][1] = 1$

- **Max** →  $\max(0,1) + 1 = 2$
- Try dropping from **floor 2**:
  - If **breaks**, check below:  $dp[2][1] = 1$
  - If **survives**, check above:  $dp[3][0] = 0$
  - **Max** →  $\max(1,0) + 1 = 2$
- **Final Result:**  $dp[3][2] = \min(2,2) = 2$

#### Filling for $dp[4][2]$

- Try dropping from **floor 1**:
  - If **breaks**, check below:  $dp[3][0] = 0$
  - If **survives**, check above:  $dp[4][1] = 1$
  - **Max** →  $\max(0,1) + 1 = 2$
- Try dropping from **floor 2**:
  - If **breaks**, check below:  $dp[3][1] = 1$
  - If **survives**, check above:  $dp[4][0] = 0$
  - **Max** →  $\max(1,0) + 1 = 2$
- **Final Result:**  $dp[4][2] = \min(2,2) = 2$

Final DP Table

<b>dp[i][j]</b>	<b>0 Floors</b>	<b>1 Floor</b>	<b>2 Floors</b>
<b>0 Eggs</b>	0	0	0
<b>1 Egg</b>	0	1	2
<b>2 Eggs</b>	0	1	2
<b>3 Eggs</b>	0	1	2
<b>4 Eggs</b>	0	1	2

Step 3: Final Answer  
 $dp[4][2] = 2$

Thus, the **minimum trials needed** to determine the critical floor with **4 eggs** and **2 floors** is **2**.

Output:-  
 2

### Kadane Max Sum Subarray C++

```
#include <iostream>
using namespace std;

int maxSubArraySum(const int arr[], int n) {
    int currentSum = arr[0]; // Initialize current sum
    int overallSum = arr[0];

    for (int i = 1; i < n; i++) {
        if (currentSum >= 0) {
            currentSum += arr[i]; // Add current element
            to current sum if positive
        } else {
            currentSum = arr[i]; // Start new subarray if
            current sum is negative
        }

        if (currentSum > overallSum) {
            overallSum = currentSum; // Update overall
            sum if current sum is greater
        }
    }

    return overallSum; // Return maximum sum found
}

int main() {
    const int arr[] = {5, 6, 7, 4, 3, 6, 4}; // Input array
    int n = sizeof(arr) / sizeof(arr[0]); // Determine the
    number of elements in the array

    cout << maxSubArraySum(arr, n) << endl; //
    Output maximum sum of subarray
    return 0;
}
```

Output:-  
35

Dry Run with Given Input

Given array:

{5,6,7,4,3,6,4}

#### **Step 2.1: Initialize Variables**

currentSum = arr[0] = 5  
overallSum = arr[0] = 5

#### **Step 2.2: Iterate Through Array**

Index (i)	Element (arr[i])	currentSum	overallSum
0	5	5	5
1	6	(5 + 6) = 11	<b>11</b>
2	7	(11 + 7) = 18	<b>18</b>
3	4	(18 + 4) = 22	<b>22</b>
4	3	(22 + 3) = 25	<b>25</b>
5	6	(25 + 6) = 31	<b>31</b>
6	4	(31 + 4) = 35	<b>35</b>

Step 3: Final Answer

Maximum Subarray Sum = 35

### Largest submatrix C++

```
#include <iostream>
#include <algorithm>
using namespace std;

// Define the maximum size for the grid (you can
adjust this as needed)
const int MAX_ROWS = 100;
const int MAX_COLS = 100;

// Function to find the largest square submatrix
int largestSquareSubmatrix(const int
arr[MAX_ROWS][MAX_COLS], int rows, int cols) {
    int dp[MAX_ROWS][MAX_COLS] = {0}; // DP table
    int largestSide = 0;

    // Fill the dp array
    for (int i = rows - 1; i >= 0; i--) {
        for (int j = cols - 1; j >= 0; j--) {
            if (i == rows - 1 || j == cols - 1) {
                dp[i][j] = arr[i][j];
            } else {
                if (arr[i][j] == 0) {
                    dp[i][j] = 0;
                } else {
                    int minSide = min(dp[i][j + 1], min(dp[i +
1][j], dp[i + 1][j + 1]));
                    dp[i][j] = minSide + 1;
                }
            }
            if (dp[i][j] > largestSide) {
                largestSide = dp[i][j];
            }
        }
    }

    return largestSide; // Return the side length of the
largest square submatrix
}

int main() {
    // Define the array and its dimensions
    const int arr[MAX_ROWS][MAX_COLS] = {
        {0, 1, 0, 1, 0, 1},
        {1, 0, 1, 0, 1, 0},
        {0, 1, 1, 1, 1, 0},
        {0, 0, 1, 1, 1, 0},
        {1, 1, 1, 1, 1, 1}
    };
    int rows = 5;
    int cols = 6;

    cout << largestSquareSubmatrix(arr, rows, cols) <<
endl;

    return 0;
}
```

Output:-  
3

#### Step 2.1: Given Matrix (arr)

```
0 1 0 1 0 1
1 0 1 0 1 0
0 1 1 1 1 0
0 0 1 1 1 0
1 1 1 1 1 1
```

#### Step 2.2: DP Table Construction

##### Step 2.2.1: Initialize dp[][] (Same as arr[][] for last row & last column)

```
0 1 0 1 0 1
1 0 1 0 1 0
0 1 1 1 1 0
0 0 1 1 1 0
1 1 1 1 1 1 <- (Same as `arr` because it's the
last row)
```

##### Step 2.2.2: Fill the dp[][] Table Bottom-Up

i, j	arr[i][j]	Formula Applied	dp[i][j]
(3,4)	1	1 + min(1, 1, 1)	2
(3,3)	1	1 + min(1, 1, 2)	2
(3,2)	1	1 + min(1, 2, 1)	2
(2,4)	1	1 + min(2, 1, 1)	2
(2,3)	1	1 + min(2, 2, 1)	2
(2,2)	1	1 + min(2, 2, 2)	3 (Largest Square Found)

#### Final dp[][] Matrix

```
0 1 0 1 0 1
1 0 1 0 1 0
0 1 2 2 2 0
0 0 2 2 2 0
1 1 1 1 1 1
```

Step 3: Final Answer  
Largest Square Side = 3

## LCS in C++

```
#include <iostream>
#include <string>
#include <algorithm> // For std::max
using namespace std;

// Define maximum possible sizes for
// the strings
const int MAX_M = 100;
const int MAX_N = 100;

int LCS(const string& s1, const
        string& s2) {
    int m = s1.length();
    int n = s2.length();

    // Initialize DP table with zeros
    int dp[MAX_M + 1][MAX_N + 1] =
{0};

    for (int i = m - 1; i >= 0; i--) {
        for (int j = n - 1; j >= 0; j--) {
            if (s1[i] == s2[j]) {
                dp[i][j] = 1 + dp[i + 1][j + 1];
            } else {
                dp[i][j] = max(dp[i + 1][j],
dp[i][j + 1]);
            }
        }
    }

    return dp[0][0];
}

int main() {
    string s1 = "abed";
    string s2 = "abbd";

    cout << LCS(s1, s2) << endl;

    return 0;
}
```

### Step-by-Step Execution:

We initialize a **DP table**  $dp [MAX\_M+1] [MAX\_N+1]$  with all zeros.

- **Strings Given:**

$s_1 = "abcd" \quad (m = 4)$   
 $s_2 = "abbd" \quad (n = 4)$

- **Table Size:**  $dp [5] [5]$  (since we use indices 0 to 4 inclusive)

### Dry Run Table (Index-Based Execution of DP Table)

Step	i	j	s1[i]	s2[j]	Match?	Formula Used	dp[i][j] Value
1	3	3	'd'	'd'	Yes	$dp[i][j] = 1 + dp[i+1][j+1]$	$dp[3][3] = 1 + 0 = 1$
2	3	2	'd'	'b'	No	$dp[i][j] = \max(dp[i+1][j], dp[i][j+1])$	$dp[3][2] = \max(0, 1) = 1$
3	3	1	'd'	'b'	No	$dp[3][1] = \max(0, 1) = 1$	
4	3	0	'd'	'a'	No	$dp[3][0] = \max(0, 1) = 1$	
5	2	3	'c'	'd'	No	$dp[2][3] = \max(1, 0) = 1$	
6	2	2	'c'	'b'	No	$dp[2][2] = \max(1, 1) = 1$	
7	2	1	'c'	'b'	No	$dp[2][1] = \max(1, 1) = 1$	
8	2	0	'c'	'a'	No	$dp[2][0] = \max(1, 1) = 1$	
9	1	3	'b'	'd'	No	$dp[1][3] = \max(1, 0) = 1$	
10	1	2	'b'	'b'	Yes	$dp[1][2] = 1 + dp[2][3] = 1 + 1 = 2$	
11	1	1	'b'	'b'	Yes	$dp[1][1] = 1 + dp[2][2] = 1 + 1 = 2$	
12	1	0	'b'	'a'	No	$dp[1][0] = \max(1, 2) = 2$	
13	0	3	'a'	'd'	No	$dp[0][3] = \max(1, 0) = 1$	
14	0	2	'a'	'b'	No	$dp[0][2] = \max(2, 1) = 2$	
15	0	1	'a'	'b'	No	$dp[0][1] = \max(2, 2) = 2$	
16	0	0	'a'	'a'	Yes	$dp[0][0] = 1 + dp[1][1] = 1 + 2 = 3$	

### **Final DP Table After Execution**

	a	b	b	d
a	3	2	2	1
b	2	2	2	1
c	1	1	1	1
d	1	1	1	1

### **Final Output**

LCS ("abcd", "abbd") = 3

The longest common subsequence is "abd" (of length 3).

Output:-  
3

## LIS in C++

```
#include <iostream>
#include <vector>
#include <algorithm> // For std::max
using namespace std;

void LIS(const vector<int>& arr) {
    int n = arr.size();
    vector<int> dp(n, 1); // dp[i] will
    store the length of LIS ending at
    index i
    int omax = 1; // To store the overall
    maximum length of LIS

    // Compute the length of the
    Longest Increasing Subsequence
    for (int i = 1; i < n; i++) {
        int max_len = 0;
        for (int j = 0; j < i; j++) {
            if (arr[i] > arr[j]) {
                if (dp[j] > max_len) {
                    max_len = dp[j];
                }
            }
        }
        dp[i] = max_len + 1;
        if (dp[i] > omax) {
            omax = dp[i];
        }
    }

    cout << omax << " "; // Print the
    length of the LIS

    // Printing the LIS length values
    (optional)
    for (int i = 0; i < n; i++) {
        cout << dp[i] << " ";
    }
    cout << endl;
}

int main() {
    vector<int> arr = {10, 22, 9, 33, 21,
    50, 41, 60, 80, 3};

    LIS(arr);

    return 0;
}
```

Let's perform a **dry run** of the given C++ program with the input:

```
arr = {10, 22, 9, 33, 21, 50, 41, 60, 80, 3}
```

## Understanding the Code

The program finds the **length of the Longest Increasing Subsequence (LIS)** using **dynamic programming**.

- $dp[i]$  stores the length of the **LIS ending at index  $i$** .
- The **final answer** is the maximum value in  $dp[]$ .

## Step-by-Step Dry Run

Step	i	j	arr[i]	arr[j]	arr[i] > arr[j]	dp[j]	max_len	dp[i]	omax
1	1	0	22	10	Yes	1	1	2	2
2	2	0	9	10	No	-	0	1	2
3	2	1	9	22	No	-	0	1	2
4	3	0	33	10	Yes	1	1	-	-
5	3	1	33	22	Yes	2	2	-	-
6	3	2	33	9	Yes	1	2	3	3
7	4	0	21	10	Yes	1	1	-	-
8	4	1	21	22	No	-	1	-	-
9	4	2	21	9	Yes	1	1	-	-
10	4	3	21	33	No	-	1	2	3
11	5	0	50	10	Yes	1	1	-	-
12	5	1	50	22	Yes	2	2	-	-
13	5	2	50	9	Yes	1	2	-	-
14	5	3	50	33	Yes	3	3	-	-
15	5	4	50	21	Yes	2	3	4	4
16	6	0	41	10	Yes	1	1	-	-
17	6	1	41	22	Yes	2	2	-	-
18	6	2	41	9	Yes	1	2	-	-
19	6	3	41	33	Yes	3	3	-	-
20	6	4	41	21	Yes	2	3	-	-
21	6	5	41	50	No	-	3	4	4
22	7	0	60	10	Yes	1	1	-	-
23	7	1	60	22	Yes	2	2	-	-
24	7	2	60	9	Yes	1	2	-	-
25	7	3	60	33	Yes	3	3	-	-
26	7	4	60	21	Yes	2	3	-	-
27	7	5	60	50	Yes	4	4	-	-
28	7	6	60	41	Yes	4	4	5	5

	29	8	0	80		10	Yes	1	1	-	-
	30	8	1	80		22	Yes	2	2	-	-
	31	8	2	80		9	Yes	1	2	-	-
	32	8	3	80		33	Yes	3	3	-	-
	33	8	4	80		21	Yes	2	3	-	-
	34	8	5	80		50	Yes	4	4	-	-
	35	8	6	80		41	Yes	4	4	-	-
	36	8	7	80		60	Yes	5	5	6	6
	37	9	0	3		10	No	-	0	-	-
	38	9	1	3		22	No	-	0	-	-
	39	9	2	3		9	No	-	0	-	-
	40	9	3	3		33	No	-	0	-	-
	41	9	4	3		21	No	-	0	-	-
	42	9	5	3		50	No	-	0	-	-
	43	9	6	3		41	No	-	0	-	-
	44	9	7	3		60	No	-	0	-	-
	45	9	8	3		80	No	-	0	1	6

### Final Output

6 1 2 1 3 2 4 4 5 6 1

- **LIS Length:** 6
- **LIS DP Table:** [1, 2, 1, 3, 2, 4, 4, 5, 6, 1]

Output:-

6  
1 2 1 2 4 4 5 6 1

Longest Bitonic Subseq In C++																																																																																				
Step-by-Step Dry Run																																																																																				
<pre>#include &lt;iostream&gt; #include &lt;vector&gt; using namespace std; int LongestBitonicSubseq(int arr[], int n) {     vector&lt;int&gt; lis(n, 1); // lis[i] will store the                            // length of LIS ending at index i     vector&lt;int&gt; lds(n, 1); // lds[i] will store the                            // length of LDS starting at index i      // Computing LIS     for (int i = 1; i &lt; n; i++) {         for (int j = 0; j &lt; i; j++) {             if (arr[j] &lt;= arr[i]) {                 lis[i] = max(lis[i], lis[j] + 1);             }         }     }      // Computing LDS     for (int i = n - 2; i &gt;= 0; i--) {         for (int j = n - 1; j &gt; i; j--) {             if (arr[j] &lt;= arr[i])                 lds[i] = max(lds[i], lds[j] + 1);         }     }      int omax = 0; // To store the overall maximum                   // length of LBS      // Finding the length of the Longest Bitonic     // Subsequence     for (int i = 0; i &lt; n; i++) {         omax = max(omax, lis[i] + lds[i] - 1);     }     return omax; }  int main() {     int arr[] = {10, 22, 9, 33, 21, 50, 41, 60, 80, 3};     int n = sizeof(arr) / sizeof(arr[0]);      cout &lt;&lt; LongestBitonicSubseq(arr, n) &lt;&lt; endl;      return 0; }</pre>	<p><b>Step 1: Compute lis[] (Longest Increasing Subsequence)</b></p> <p>We iterate from <b>left to right</b>, storing the longest increasing sequence <b>ending at each index</b>.</p> <table border="1"> <thead> <tr> <th>i</th><th>arr[i]</th><th>LIS Calculation (lis[i] = max(lis[i], lis[j] + 1))</th><th>lis[i]</th></tr> </thead> <tbody> <tr> <td>0</td><td>10</td><td>lis[0] = 1 (base case)</td><td>1</td></tr> <tr> <td>1</td><td>22</td><td><math>10 &lt; 22 \rightarrow \text{lis}[1] = \text{lis}[0] + 1 = 2</math></td><td>2</td></tr> <tr> <td>2</td><td>9</td><td>No valid j</td><td>1</td></tr> <tr> <td>3</td><td>33</td><td><math>10 &lt; 33 \rightarrow \text{lis}[3] = \text{lis}[0] + 1 = 2</math></td><td>2</td></tr> <tr> <td></td><td></td><td><math>22 &lt; 33 \rightarrow \text{lis}[3] = \text{lis}[1] + 1 = 3</math></td><td>3</td></tr> <tr> <td>4</td><td>21</td><td><math>10 &lt; 21 \rightarrow \text{lis}[4] = \text{lis}[0] + 1 = 2</math></td><td>2</td></tr> <tr> <td>5</td><td>50</td><td><math>10 &lt; 50 \rightarrow \text{lis}[5] = \text{lis}[0] + 1 = 2</math></td><td>2</td></tr> <tr> <td></td><td></td><td><math>22 &lt; 50 \rightarrow \text{lis}[5] = \text{lis}[1] + 1 = 3</math></td><td>3</td></tr> <tr> <td></td><td></td><td><math>33 &lt; 50 \rightarrow \text{lis}[5] = \text{lis}[3] + 1 = 4</math></td><td>4</td></tr> <tr> <td>6</td><td>41</td><td><math>10 &lt; 41 \rightarrow \text{lis}[6] = \text{lis}[0] + 1 = 2</math></td><td>2</td></tr> <tr> <td></td><td></td><td><math>22 &lt; 41 \rightarrow \text{lis}[6] = \text{lis}[1] + 1 = 3</math></td><td>3</td></tr> <tr> <td></td><td></td><td><math>33 &lt; 41 \rightarrow \text{lis}[6] = \text{lis}[3] + 1 = 4</math></td><td>4</td></tr> <tr> <td>7</td><td>60</td><td><math>10 &lt; 60 \rightarrow \text{lis}[7] = \text{lis}[0] + 1 = 2</math></td><td>2</td></tr> <tr> <td></td><td></td><td><math>22 &lt; 60 \rightarrow \text{lis}[7] = \text{lis}[1] + 1 = 3</math></td><td>3</td></tr> <tr> <td></td><td></td><td><math>33 &lt; 60 \rightarrow \text{lis}[7] = \text{lis}[3] + 1 = 4</math></td><td>4</td></tr> <tr> <td></td><td></td><td><math>50 &lt; 60 \rightarrow \text{lis}[7] = \text{lis}[5] + 1 = 5</math></td><td>5</td></tr> <tr> <td>8</td><td>80</td><td><math>10 &lt; 80 \rightarrow \text{lis}[8] = \text{lis}[0] + 1 = 2</math></td><td>2</td></tr> <tr> <td></td><td></td><td><math>22 &lt; 80 \rightarrow \text{lis}[8] = \text{lis}[1] + 1 = 3</math></td><td>3</td></tr> <tr> <td></td><td></td><td><math>33 &lt; 80 \rightarrow \text{lis}[8] = \text{lis}[3] + 1 = 4</math></td><td>4</td></tr> </tbody> </table>	i	arr[i]	LIS Calculation (lis[i] = max(lis[i], lis[j] + 1))	lis[i]	0	10	lis[0] = 1 (base case)	1	1	22	$10 < 22 \rightarrow \text{lis}[1] = \text{lis}[0] + 1 = 2$	2	2	9	No valid j	1	3	33	$10 < 33 \rightarrow \text{lis}[3] = \text{lis}[0] + 1 = 2$	2			$22 < 33 \rightarrow \text{lis}[3] = \text{lis}[1] + 1 = 3$	3	4	21	$10 < 21 \rightarrow \text{lis}[4] = \text{lis}[0] + 1 = 2$	2	5	50	$10 < 50 \rightarrow \text{lis}[5] = \text{lis}[0] + 1 = 2$	2			$22 < 50 \rightarrow \text{lis}[5] = \text{lis}[1] + 1 = 3$	3			$33 < 50 \rightarrow \text{lis}[5] = \text{lis}[3] + 1 = 4$	4	6	41	$10 < 41 \rightarrow \text{lis}[6] = \text{lis}[0] + 1 = 2$	2			$22 < 41 \rightarrow \text{lis}[6] = \text{lis}[1] + 1 = 3$	3			$33 < 41 \rightarrow \text{lis}[6] = \text{lis}[3] + 1 = 4$	4	7	60	$10 < 60 \rightarrow \text{lis}[7] = \text{lis}[0] + 1 = 2$	2			$22 < 60 \rightarrow \text{lis}[7] = \text{lis}[1] + 1 = 3$	3			$33 < 60 \rightarrow \text{lis}[7] = \text{lis}[3] + 1 = 4$	4			$50 < 60 \rightarrow \text{lis}[7] = \text{lis}[5] + 1 = 5$	5	8	80	$10 < 80 \rightarrow \text{lis}[8] = \text{lis}[0] + 1 = 2$	2			$22 < 80 \rightarrow \text{lis}[8] = \text{lis}[1] + 1 = 3$	3			$33 < 80 \rightarrow \text{lis}[8] = \text{lis}[3] + 1 = 4$	4			
i	arr[i]	LIS Calculation (lis[i] = max(lis[i], lis[j] + 1))	lis[i]																																																																																	
0	10	lis[0] = 1 (base case)	1																																																																																	
1	22	$10 < 22 \rightarrow \text{lis}[1] = \text{lis}[0] + 1 = 2$	2																																																																																	
2	9	No valid j	1																																																																																	
3	33	$10 < 33 \rightarrow \text{lis}[3] = \text{lis}[0] + 1 = 2$	2																																																																																	
		$22 < 33 \rightarrow \text{lis}[3] = \text{lis}[1] + 1 = 3$	3																																																																																	
4	21	$10 < 21 \rightarrow \text{lis}[4] = \text{lis}[0] + 1 = 2$	2																																																																																	
5	50	$10 < 50 \rightarrow \text{lis}[5] = \text{lis}[0] + 1 = 2$	2																																																																																	
		$22 < 50 \rightarrow \text{lis}[5] = \text{lis}[1] + 1 = 3$	3																																																																																	
		$33 < 50 \rightarrow \text{lis}[5] = \text{lis}[3] + 1 = 4$	4																																																																																	
6	41	$10 < 41 \rightarrow \text{lis}[6] = \text{lis}[0] + 1 = 2$	2																																																																																	
		$22 < 41 \rightarrow \text{lis}[6] = \text{lis}[1] + 1 = 3$	3																																																																																	
		$33 < 41 \rightarrow \text{lis}[6] = \text{lis}[3] + 1 = 4$	4																																																																																	
7	60	$10 < 60 \rightarrow \text{lis}[7] = \text{lis}[0] + 1 = 2$	2																																																																																	
		$22 < 60 \rightarrow \text{lis}[7] = \text{lis}[1] + 1 = 3$	3																																																																																	
		$33 < 60 \rightarrow \text{lis}[7] = \text{lis}[3] + 1 = 4$	4																																																																																	
		$50 < 60 \rightarrow \text{lis}[7] = \text{lis}[5] + 1 = 5$	5																																																																																	
8	80	$10 < 80 \rightarrow \text{lis}[8] = \text{lis}[0] + 1 = 2$	2																																																																																	
		$22 < 80 \rightarrow \text{lis}[8] = \text{lis}[1] + 1 = 3$	3																																																																																	
		$33 < 80 \rightarrow \text{lis}[8] = \text{lis}[3] + 1 = 4$	4																																																																																	

		$50 < 80 \rightarrow lis[8] = lis[5] + 1 = 5$	5
		$60 < 80 \rightarrow lis[8] = lis[7] + 1 = 6$	6
9	3	No valid j	1

### Final lis[] Array

lis = [1, 2, 1, 3, 2, 4, 4, 5, 6, 1]

### Step 2: Compute lds[] (Longest Decreasing Subsequence)

We iterate from **right to left**, storing the longest decreasing sequence **starting from each index**.

i	arr[i]	LDS Calculation ( $lds[i] = \max(lds[i], lds[j] + 1)$ )	lds[i]
9	3	$lds[9] = 1$ (base case)	1
8	80	$lds[8] = 1$	1
7	60	$lds[7] = \max(lds[7], lds[8] + 1) = 2$	2
6	41	$lds[6] = \max(lds[6], lds[7] + 1) = 3$	3
5	50	$lds[5] = \max(lds[5], lds[6] + 1) = 4$	4
4	21	$lds[4] = 2$	2
3	33	$lds[3] = \max(lds[3], lds[4] + 1) = 3$	3
2	9	$lds[2] = \max(lds[2], lds[4] + 1) = 2$	2
1	22	$lds[1] = \max(lds[1], lds[2] + 1) = 3$	3
0	10	$lds[0] = \max(lds[0], lds[2] + 1) = 2$	2

### Final lds[] Array

lds = [2, 3, 2, 3, 2, 4, 3, 2, 1, 1]

### Step 3: Compute omax (Overall Maximum LBS)

Using:

$$\text{omax} = \max(\text{lis}[i] + \text{lds}[i] - 1)$$

i	lis[i]	lds[i]	lis[i] + lds[i] - 1
0	1	2	<b>2</b>
1	2	3	4
2	1	2	<b>2</b>
3	3	3	<b>5</b>
4	2	2	<b>3</b>
5	4	4	<b>7</b>
6	4	3	<b>6</b>
7	5	2	<b>6</b>
8	6	1	<b>6</b>
9	1	1	<b>1</b>

The **maximum** value in this list is 7.

Output:-7

### Longest Common substring In C++

```
#include <iostream>
#include <string>
#include <vector>
using namespace std;

int LongestCommonSubstring(string s1,
string s2) {
    int m = s1.length();
    int n = s2.length();
    vector<vector<int>> dp(m + 1,
vector<int>(n + 1, 0));
    //int dp[m+1][n+1]={0};
    int maxLen = 0;

    for (int i = 1; i <= m; i++) {
        for (int j = 1; j <= n; j++) {
            if (s1[i - 1] == s2[j - 1]) {
                dp[i][j] = dp[i - 1][j - 1] + 1;
                maxLen = max(maxLen, dp[i][j]);
            } else {
                dp[i][j] = 0;
            }
        }
    }

    return maxLen;
}

int main() {
    string s1 = "abcp";
    string s2 = "abcy";

    cout << LongestCommonSubstring(s1, s2)
    << endl;

    return 0;
}
```

### Step-by-Step DP Table Construction

i	j	s1[i-1]	s2[j-1]	Match?	dp[i][j] Calculation	Updated maxLen
1	1	a	a	✓	dp[0][0] + 1 = 1	1
1	2	a	b	✗	0	1
1	3	a	c	✗	0	1
1	4	a	y	✗	0	1
2	1	b	a	✗	0	1
2	2	b	b	✓	dp[1][1] + 1 = 2	2
2	3	b	c	✗	0	2
2	4	b	y	✗	0	2
3	1	c	a	✗	0	2
3	2	c	b	✗	0	2
3	3	c	c	✓	dp[2][2] + 1 = 3	3
3	4	c	y	✗	0	3
4	1	p	a	✗	0	3
4	2	p	b	✗	0	3
4	3	p	c	✗	0	3
4	4	p	y	✗	0	3

### Final DP Table

	_	a	b	c	y
_	0	0	0	0	0
a	0	1	0	0	0
b	0	0	2	0	0
c	0	0	0	3	0
p	0	0	0	0	0

### Final Answer

- **Longest Common Substring length = 3**  
("abc")
- **Output:**

	3
Output:-	3

## Longest Palindromic subseq In C++

```
#include <iostream>
#include <string>
#include <vector>
using namespace std;

int LongestPalindromicSubsequence(string str) {
    int n = str.length();
    //vector<vector<int>> dp(n, vector<int>(n, 0));
    int dp[n][n]={0};

    for (int g = 0; g < n; g++) {
        for (int i = 0, j = g; j < n; i++, j++) {
            if (g == 0) {
                dp[i][j] = 1;
            } else if (g == 1) {
                dp[i][j] = (str[i] == str[j]) ? 2 : 1;
            } else {
                if (str[i] == str[j]) {
                    dp[i][j] = 2 + dp[i + 1][j - 1];
                } else {
                    dp[i][j] = max(dp[i][j - 1], dp[i + 1][j]);
                }
            }
        }
    }

    return dp[0][n - 1];
}

int main() {
    string str = "abccba";

    int longestPalSubseqLen =
    LongestPalindromicSubsequence(str);
    cout << longestPalSubseqLen << endl;

    return 0;
}
```

### Step-by-Step Dry Run

Let's walk through each step of filling the DP table for the input string "abccba".

#### Initial Setup

- Length of string  $n = 6$
- Initialize a 2D DP table  $dp[6][6]$  with all zeros.

#### Step 1: Base Case for Substrings of Length 1

When  $g == 0$ , each character is a subsequence of length 1.

	a	b	c	c	b	a
a	1					
b		1				
c			1			
c				1		
b					1	
a						1

#### Step 2: Substrings of Length 2

When  $g == 1$ , we check if adjacent characters match.

	a	b	c	c	b	a
a	1	1				
b		1	2			
c			1	2		
c				1	2	
b					1	2
a						1

#### Step 3: Substrings of Length 3 and Beyond

For substrings of length greater than 2, we

follow the general case rules.

<b>g (Gap)</b>	<b>i</b>	<b>j</b>	<b>Formula Used</b>	<b>dp[i][j]</b>
2	0	2	$dp[1][1] + 2$ (Match a == a)	3
2	1	3	$\max(dp[1][2], dp[2][3])$ (Max of 1 and 2)	2
2	2	4	$dp[3][3] + 2$ (Match b == b)	3
2	3	5	$\max(dp[3][4], dp[4][5])$ (Max of 1 and 2)	2
3	0	3	$dp[1][2] + 2$ (Match a == a)	3
3	1	4	$\max(dp[1][3], dp[2][4])$ (Max of 2 and 3)	3
3	2	5	$\max(dp[2][4], dp[3][5])$ (Max of 3 and 2)	3
4	0	4	$dp[1][3] + 2$ (Match a == a)	4
4	1	5	$\max(dp[1][4], dp[2][5])$ (Max of 3 and 3)	4
5	0	5	$dp[1][4] + 2$ (Match a == a)	6

### Final DP Table

	<b>a</b>	<b>b</b>	<b>c</b>	<b>c</b>	<b>b</b>	<b>a</b>
<b>a</b>	1	1	3	3	4	6
<b>b</b>		1	2	2	3	4
<b>c</b>			1	2	3	3
<b>c</b>				1	2	3
<b>b</b>					1	2
<b>a</b>						1

### Final Answer

The length of the **Longest Palindromic Subsequence** is stored in  $dp[0][n-1] = dp[0][5] = 6$ .

### Output:

6

Output:-  
6

### Longest Palindromic substring In C++

```
#include <iostream>
#include <string>
using namespace std;

int LongestPalindromicSubstring(string str) {
    int n = str.length();
    bool dp[n][n];
    int len = 0;

    // Initialize dp array
    for (int i = 0; i < n; i++) {
        dp[i][i] = true;
    }

    // Check for substrings of length 2
    for (int i = 0; i < n - 1; i++) {
        if (str[i] == str[i + 1]) {
            dp[i][i + 1] = true;
            len = 2; // Update length of longest
palindromic substring
        } else {
            dp[i][i + 1] = false;
        }
    }

    // Check for substrings of length > 2
    for (int g = 2; g < n; g++) {
        for (int i = 0, j = g; j < n; i++, j++) {
            if (str[i] == str[j] && dp[i + 1][j - 1]) {
                dp[i][j] = true;
                len = g + 1; // Update length of longest
palindromic substring
            } else {
                dp[i][j] = false;
            }
        }
    }

    return len;
}

int main() {
    string str = "abccbc";
    int longestPalSubstrLen =
LongestPalindromicSubstring(str);
    cout << longestPalSubstrLen << endl;

    return 0;
}
```

### Step-by-Step Dry Run

#### Step 1: Initialize DP Table ( $g = 0$ )

Each **single character** is a palindrome ( $dp[i][i] = \text{true}$ ).

	a	b	c	c	b	c
a	✓					
b		✓				
c			✓			
c				✓		
b					✓	
c						✓

**Longest palindrome so far:**  $\text{len} = 1$  (since all single characters are palindromes).

#### Step 2: Substrings of Length 2 ( $g = 1$ )

We check adjacent characters  $\text{str}[i] == \text{str}[i+1]$ .

	a	b	c	c	b	c
a	✓	✗				
b		✓	✗			
c			✓	✓		
c				✓	✗	
b					✓	✗
c						✓

**Updated longest palindrome:**  $\text{len} = 2$  ("cc" at  $dp[2][3]$ ).

### Step 3: Substrings of Length 3+ ( $g \geq 2$ )

For substrings of length  $g + 1$ , we check:

$$dp[i][j] = (\text{str}[i] == \text{str}[j]) \text{ AND } dp[i+1][j-1]$$

**For  $g = 2$  (substrings of length 3):**

	a	b	c	c	b	c
a	✓	✗	✗			
b		✓	✗	✗	✓	
c			✓	✓	✗	✗
c				✓	✗	✗
b					✓	✗
c						✓

**Updated longest palindrome: len = 3 ("bccb" at  $dp[1][4]$ ).**

**For  $g = 3$  (substrings of length 4):**

	a	b	c	c	b	c
a	✓	✗	✗	✗		
b		✓	✗	✗	✓	✗
c			✓	✓	✗	✗
c				✓	✗	✗
b					✓	✗
c						✓

**Updated longest palindrome: len = 4 ("bccb")**

at  $dp[1][4]$ ).

**For  $g = 4$  (substrings of length 5):**

	a	b	c	c	b	c
a	✓	✗	✗	✗	✗	
b		✓	✗	✗	✓	✗
c			✓	✓	✗	✗
c				✓	✗	✗
b					✓	✗
c						✓

No update to len (remains 4).

**For  $g = 5$  (full string, length 6):**

	a	b	c	c	b	c
a	✓	✗	✗	✗	✗	✗
b		✓	✗	✗	✓	✗
c			✓	✓	✗	✗
c				✓	✗	✗
b					✓	✗
c						✓

Final longest palindrome: len = 4 ("bccb").

**Final Answer**

The longest palindromic substring in "abccbc"

	has length 4 ("bccb").
	Output: 4
Output:- 4	

### Max Sum Increasing subseq In C++

```
#include <iostream>
#include <climits>
using namespace std;

int MaxSumIncreasingSubseq(int arr[], int size) {
    int omax = INT_MIN;
    int* dp = new int[size];
    //int dp[size];

    for (int i = 0; i < size; i++) {
        int maxSum = arr[i];
        for (int j = 0; j < i; j++) {
            if (arr[j] <= arr[i]) {
                maxSum = max(maxSum, dp[j] +
arr[i]);
            }
        }
        dp[i] = maxSum;
        omax = max(omax, dp[i]);
    }

    delete[] dp; // Don't forget to free the allocated
memory
    return omax;
}

int main() {
    int arr[] = {10, 22, 9, 33, 21, 50, 41, 60, 80, 3};
    int size = sizeof(arr) / sizeof(arr[0]);

    int maxSum = MaxSumIncreasingSubseq(arr,
size);
    cout << maxSum << endl;

    return 0;
}
```

arr = {10, 22, 9, 33, 21, 50, 41, 60, 80, 3}

### Step-by-Step Dry Run (Table Format)

Index (i)	arr[i]	Initial dp[i]	Comparisons (j < i, arr[j] ≤ arr[i])	Updated dp[i]
0	10	10	-	10
1	22	22	j=0 (10 ≤ 22) → dp[1] = max(22, 10+22)	32
2	9	9	-	9
3	33	33	j=0 (10 ≤ 33) → dp[3] = max(33, 10+33) j=1 (22 ≤ 33) → dp[3] = max(43, 32+33)	65
4	21	21	j=0 (10 ≤ 21) → dp[4] = max(21, 10+21)	31
5	50	50	j=0 (10 ≤ 50) → dp[5] = max(50, 10+50) j=1 (22 ≤ 50) → dp[5] = max(60, 32+50) j=3 (33 ≤ 50) → dp[5] = max(100, 65+50)	100
6	41	41	j=0 (10 ≤ 41) → dp[6] = max(41, 10+41) j=1 (22 ≤ 41) → dp[6] = max(51, 32+41) j=3 (33 ≤ 41) → dp[6] = max(91, 65+41)	91
7	60	60	j=0 (10 ≤ 60) → dp[7] = max(60, 10+60) j=1 (22 ≤ 60) → dp[7] = max(70, 32+60) j=3 (33 ≤ 60) → dp[7] = max(110, 65+60) j=5 (50 ≤ 60) → dp[7] = max(150, 100+60)	150

			j=0,1,3,5,6,7 (comparing all increasing values) → dp[8] = max(10+80, 32+80, 65+80, 100+80, 91+80, 150+80)	
8	80	80		<b>255</b>

  

9	3	3	-	3
---	---	---	---	---

### Final DP Table

Index (i)	arr[i]	dp[i] (Max Sum IS Ending at i)
0	10	10
1	22	32
2	9	9
3	33	65
4	21	31
5	50	100
6	41	91
7	60	150
8	80	255
9	3	3

### Final Answer

Output: 255

#### Summary:

- The **largest increasing subsequence** contributing to **255** is:

10 → 22 → 33 → 50 → 60 → 80

- Sum = 10 + 22 + 33 + 50 + 60 + 80 = 255

Output:-

255

{10, 22, 33, 50, 60, 80} → sum = 10 + 22 + 33 + 50 + 60 + 80 = 255

## Min Cost to make strings identical C++

```
#include <iostream>
#include <string>
#include <vector>
using namespace std;

int minCostToMakeIdentical(string s1, string s2, int c1, int c2) {
    int m = s1.length();
    int n = s2.length();

    // Initialize dp array with size (m+1)x(n+1)
    vector<vector<int>> dp(m + 1, vector<int>(n + 1, 0));

    // Fill dp array
    for (int i = m - 1; i >= 0; i--) {
        for (int j = n - 1; j >= 0; j--) {
            if (s1[i] == s2[j]) {
                dp[i][j] = 1 + dp[i + 1][j + 1];
            } else {
                dp[i][j] = max(dp[i + 1][j], dp[i][j + 1]);
            }
        }
    }

    // Calculate length of LCS
    int lcsLength = dp[0][0];
    cout << "Length of Longest Common Subsequence: "
    << lcsLength << endl;

    // Calculate remaining characters in s1 and s2 after
    // LCS
    int s1Remaining = m - lcsLength;
    int s2Remaining = n - lcsLength;

    // Calculate minimum cost to make strings identical
    int cost = s1Remaining * c1 + s2Remaining * c2;
    return cost;
}

int main() {
    string s1 = "cat";
    string s2 = "cut";
    int c1 = 1;
    int c2 = 1;

    int minCost = minCostToMakeIdentical(s1, s2, c1, c2);
    cout << "Minimum cost to make strings identical: "
    << minCost << endl;

    return 0;
}
```

### Step-by-Step DP Table Construction

#### Strings:

s1 = "cat"  
s2 = "cut"

We create a **(m+1) × (n+1) DP table**, where:

- dp[i][j] stores the **length of LCS of s1[i:] and s2[j:]**.

#### DP Table Initialization (Bottom-Up)

i\j	c	u	t	(empty)
c	?	?	?	0
a	?	?	?	0
t	?	?	?	0
(empty)	0	0	0	0

#### Filling the Table

We start from the **bottom-right** and move **backwards**.

##### 1. Comparing 't' in s1 with 't' in s2:

s1[2] == s2[2] ('t' == 't')

- So, dp[2][2] = 1 + dp[3][3] = 1

##### 2. Comparing 't' in s1 with 'u' in s2:

s1[2] != s2[1] ('t' ≠ 'u')

- So, dp[2][1] = max(dp[3][1], dp[2][2]) = max(0, 1) = 1

##### 3. Comparing 't' in s1 with 'c' in s2:

s1[2] != s2[0] ('t' ≠ 'c')

- So, dp[2][0] = max(dp[3][0], dp[2][1]) = max(0, 1) = 1

##### 4. Comparing 'a' in s1 with 't' in s2:

s1[1] != s2[2] ('a' ≠ 't')

- So, dp[1][2] = max(dp[2][2], dp[1][3]) = max(1, 0) = 1

##### 5. Comparing 'a' in s1 with 'u' in s2:

s1[1] != s2[1] ('a' ≠ 'u')

- So,  $dp[1][1] = \max(dp[2][1], dp[1][2]) = \max(1, 1) = 1$

#### 6. Comparing 'a' in s1 with 'c' in s2:

$s1[1] \neq s2[0]$  ('a' ≠ 'c')

- So,  $dp[1][0] = \max(dp[2][0], dp[1][1]) = \max(1, 1) = 1$

#### 7. Comparing 'c' in s1 with 't' in s2:

$s1[0] \neq s2[2]$  ('c' ≠ 't')

- So,  $dp[0][2] = \max(dp[1][2], dp[0][3]) = \max(1, 0) = 1$

#### 8. Comparing 'c' in s1 with 'u' in s2:

$s1[0] \neq s2[1]$  ('c' ≠ 'u')

- So,  $dp[0][1] = \max(dp[1][1], dp[0][2]) = \max(1, 1) = 1$

#### 9. Comparing 'c' in s1 with 'c' in s2:

$s1[0] == s2[0]$  ('c' == 'c')

- So,  $dp[0][0] = 1 + dp[1][1] = 2$

### Final DP Table

i\j	c	u	t	(empty)
c	2	1	1	0
a	1	1	1	0
t	1	1	1	0
(empty)	0	0	0	0

### Final Calculation

- **LCS Length** =  $dp[0][0] = 2$
- **Remaining characters to delete:**

s1: "cat" → Remove 1 character ('a')  
 s2: "cut" → Remove 1 character ('u')

- **Total Cost:**

$$\text{Cost} = (1 \times 1) + (1 \times 1) = 1 + 1 = 2$$

Output:-

Length of Longest Common Subsequence: 2

Minimum cost to make strings identical: 2

### Optimal strategy for a game In C++

```
#include <iostream>
#include <algorithm>

using namespace std;

int main() {
    int arr[] = {20, 30, 2, 10};
    int n = sizeof(arr) / sizeof(arr[0]);

    int dp[n][n]; // Create a 2D array of size n x n

    for (int g = 0; g < n; g++) {
        for (int i = 0, j = g; j < n; i++, j++) {
            if (g == 0) {
                dp[i][j] = arr[i];
            } else if (g == 1) {
                dp[i][j] = max(arr[i], arr[j]);
            } else {
                int val1 = arr[i] + min((i + 2 <= j ? dp[i + 2][j] : 0), (i + 1 <= j - 1 ? dp[i + 1][j - 1] : 0));
                int val2 = arr[j] + min((i + 1 <= j - 1 ? dp[i + 1][j - 1] : 0), (i <= j - 2 ? dp[i][j - 2] : 0));
                dp[i][j] = max(val1, val2);
            }
        }
    }

    cout << dp[0][n - 1] << endl; // Print the maximum value that can be collected
}

return 0;
}
```

### Step-by-Step Dry Run with Table

#### Initialization

Given input:

```
int arr[] = {20, 30, 2, 10};
```

Size of arr:

$n = 4$ ;

A **2D DP table ( $dp[i][j]$ )** is used, where  $dp[i][j]$  represents the **maximum score the first player can collect from  $arr[i]$  to  $arr[j]$** .

#### Step 1: Fill Diagonal ( $g = 0$ )

When  $i == j$ , only one element is available, so:

i	j	dp[i][j]
0	0	20
1	1	30
2	2	2
3	3	10

#### Step 2: Fill $g = 1$ (Two Elements)

When  $g = 1$ , two elements are available, so the first player picks the maximum:

i	j	Computation	dp[i][j]
0	1	max(20, 30)	30
1	2	max(30, 2)	30

i	j	Computation	dp[i][j]
2	3	max(2, 10)	10

### Step 3: Fill g = 2 (Three Elements)

Now, we consider **three elements** and the optimal choices:

i	j	Computation	dp[i][j]
0	2	max(20 + min(2, 30), 2 + min(30, 20)) → max(20+2, 2+20) = 22	22
1	3	max(30 + min(10, 2), 10 + min(2, 30)) → max(30+2, 10+2) = 32	32

### Step 4: Fill g = 3 (Entire Array)

i\j	0	1	2	3
0	20	30	22	40
1		30	30	32
2			2	10
3				10

### Final Output:

40

Output:-  
40



```

vector<int>(cap + 1, 0));

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

        if (j >= wts[i - 1]) {
            dp[i][j] = max(dp[i][j], dp[i - 1][j - wts[i - 1]] + vals[i - 1]);
        }
    }
}

int ans = dp[n][cap];
cout << "Maximum value: " << ans <<
endl;

deque<Pair> que;
que.push_back(Pair(n, cap, ""));

printPaths(dp, vals, wts, n, cap, "", que);
}

int main() {
    vector<int> vals = {15, 14, 10, 45, 30};
    vector<int> wts = {2, 5, 1, 3, 4};
    int cap = 7;

    knapsackPaths(vals, wts, cap);

    return 0;
}

```

<b>3</b> (val=10, wt=1)	0	10	15	25	25	25	25	25	
<b>4</b> (val=45, wt=3)	0	10	15	45	55	60	70	70	
<b>5</b> (val=30, wt=4)	0	10	15	45	55	60	70	<b>75</b>	

The **maximum value** obtained is 75 at dp[5][7].

## Step 2: Print All Paths

Using **backtracking**, the function printPaths reconstructs paths that lead to dp[n][cap] = 75.

## Backtracking Paths

1. Start at dp[5][7] = 75
  - o dp[4][3] = 45 → Item 5 (index 4, value 30, weight 4) is included.
2. Now at dp[4][3] = 45
  - o dp[3][0] = 0 → Item 4 (index 3, value 45, weight 3) is included.

Thus, one of the optimal selections is {30, 45}.

## Final Output

Maximum value: 75

4 3

Output:-	
Maximum value: 75	
3 4	

### Perfect Square In C++

```
#include <iostream>
#include <vector>
#include <climits>
#include <cmath>
using namespace std;

int main() {
    vector<int> arr = {0, 1, 2, 3, 1, 2, 3, 4, 2,
1, 2, 3};
    int n = arr.size();
    vector<int> dp(n + 1, INT_MAX); // dp
array where dp[i] represents the minimum
number of perfect squares summing up to i
    //int dp[n+1]={INT_MAX};
    dp[0] = 0; // Base case: 0 requires 0
squares
    dp[1] = 1; // 1 requires 1 square (1)

    for (int i = 2; i <= n; i++) {
        for (int j = 1; j * j <= i; j++) {
            dp[i] = min(dp[i], dp[i - j * j] + 1);
        }
    }

    // Output the dp array
    for (int i = 0; i <= n; i++) {
        cout << dp[i] << " ";
    }
    cout << endl;

    return 0;
}
```

Dry Run with Table

We compute  $dp[i]$  for  $i = 0$  to  $12$  using the given transition formula.

<b>i</b>	<b>Perf ect Squ ares (≤ i)</b>	<b>dp[i] Computation</b>	<b>dp[i]</b>
<b>0</b>	-	$dp[0] = 0$	<b>0</b>
<b>1</b>	1	$dp[1] = \min(dp[1 - 1] + 1) = 1$	<b>1</b>
<b>2</b>	1	$dp[2] = \min(dp[2 - 1] + 1) = 2$	<b>2</b>
<b>3</b>	1	$dp[3] = \min(dp[3 - 1] + 1) = 3$	<b>3</b>
<b>4</b>	1, 4	$dp[4] = \min(dp[4 - 1] + 1, dp[4 - 4] + 1) = \min(4, 1) = 1$	<b>1</b>
<b>5</b>	1, 4	$dp[5] = \min(dp[5 - 1] + 1, dp[5 - 4] + 1) = \min(2, 2) = 2$	<b>2</b>
<b>6</b>	1, 4	$dp[6] = \min(dp[6 - 1] + 1, dp[6 - 4] + 1) = \min(3, 3) = 3$	<b>3</b>
<b>7</b>	1, 4	$dp[7] = \min(dp[7 - 1] + 1, dp[7 - 4] + 1) = \min(4, 4) = 4$	<b>4</b>
<b>8</b>	1, 4	$dp[8] = \min(dp[8 - 1] + 1, dp[8 - 4] + 1) = \min(5, 2) = 2$	<b>2</b>
<b>9</b>	1, 4, 9	$dp[9] = \min(dp[9 - 1] + 1, dp[9 - 4] + 1, dp[9 - 9] + 1) = \min(3, 3, 1) = 1$	<b>1</b>
<b>10</b>	1, 4, 9	$dp[10] = \min(dp[10 - 1] + 1, dp[10 - 4] + 1, dp[10 - 9] + 1) = \min(2, 4, 2) = 2$	<b>2</b>
<b>11</b>	1, 4, 9	$dp[11] = \min(dp[11 - 1] + 1, dp[11 - 4] + 1, dp[11 - 9] + 1) = \min(3, 5, 3) = 3$	<b>3</b>
<b>12</b>	1, 4, 9	$dp[12] = \min(dp[12 - 1] + 1, dp[12 - 4] + 1, dp[12 - 9] + 1) = \min(4, 3, 4)$	<b>3</b>

		= 3	
--	--	-----	--

Final Output (dp Array)

The DP array will be:

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

Output:-

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

### Print all LIS In C++

```

#include <iostream>
#include <vector>
#include <deque>
using namespace std;

struct Pair {
    int l; // length of the LIS
    int i; // index in the array
    int v; // value at index i in the array
    string psf; // path so far

    Pair(int l, int i, int v, string psf) {
        this->l = l;
        this->i = i;
        this->v = v;
        this->psf = psf;
    }
};

void printAllLIS(vector<int>& arr) {
    int n = arr.size();
    vector<int> dp(n, 1); // dp array to store
    the length of LIS ending at each index
    int omax = 0; // maximum length of LIS
    found
    int omi = 0; // index where the LIS with
    maximum length ends

    // Finding the length of LIS ending at
    each index
    for (int i = 0; i < n; i++) {
        int maxLen = 0;
        for (int j = 0; j < i; j++) {
            if (arr[i] > arr[j]) {
                if (dp[j] > maxLen) {
                    maxLen = dp[j];
                }
            }
        }
        dp[i] = maxLen + 1;

        if (dp[i] > omax) {
            omax = dp[i];
            omi = i;
        }
    }

    deque<Pair> q;
    q.push_back(Pair(omax, omi, arr[omi],
    to_string(arr[omi])));

    while (!q.empty()) {

```

Dry Run Example

**Input:**

vector<int> arr = {10, 22, 9, 33, 21, 50, 41, 60, 80, 3};

**Step 1: Compute dp Array**

Index i	arr[i]	LIS Length (dp[i])	Previous LIS Contributor (dp[j])
0	10	1	-
1	22	2	10 (dp[0] + 1)
2	9	1	-
3	33	3	22 (dp[1] + 1)
4	21	2	10 (dp[0] + 1)
5	50	4	33 (dp[3] + 1)
6	41	4	33 (dp[3] + 1)

```

Pair rem = q.front();
q.pop_front();

if (rem.l == 1) {
    cout << rem.psf << endl; // print the
path when the length of LIS is 1
} else {
    for (int j = rem.i - 1; j >= 0; j--) {
        if (dp[j] == rem.l - 1 && arr[j] <=
rem.v) {
            q.push_back(Pair(dp[j], j,
arr[j], to_string(arr[j]) + " -> " + rem.psf));
        }
    }
}
}

int main() {
    vector<int> arr = {10, 22, 9, 33, 21, 50,
41, 60, 80, 3};
    printAllLIS(arr);
    return 0;
}

```

7	60	<b>5</b> (Max LIS)	50 (dp[5] + 1)
8	80	<b>6</b> (Max LIS)	60 (dp[7] + 1)
9	3	<b>1</b>	-

### Step 2: Print All LIS Paths

The **longest increasing subsequence has length 6** and ends at 80.

Backtracking from 80, possible LIS paths:

10 -> 22 -> 33 -> 50 -> 60 -> 80  
10 -> 22 -> 33 -> 41 -> 60 -> 80

Output:-

10 -> 22 -> 33 -> 41 -> 60 -> 80  
10 -> 22 -> 33 -> 50 -> 60 -> 80

Print all path with max gold In C++																																																																																													
<pre>#include &lt;iostream&gt; #include &lt;vector&gt; #include &lt;queue&gt; using namespace std;  struct Pair {     int i, j;     string psf;      Pair(int i, int j, string psf) {         this-&gt;i = i;         this-&gt;j = j;         this-&gt;psf = psf;     } };  void printMaxGoldPath(vector&lt;vector&lt;int&gt;&gt;&amp; arr) {     int m = arr.size();     int n = arr[0].size();      // dp array to store maximum gold     // collected to reach each cell     vector&lt;vector&lt;int&gt;&gt; dp(m,     vector&lt;int&gt;(n, 0));      // Initialize dp array for the last column     for (int i = 0; i &lt; m; i++) {         dp[i][n - 1] = arr[i][n - 1];     }      // Fill dp array using dynamic     // programming approach     for (int j = n - 2; j &gt;= 0; j--) {         for (int i = 0; i &lt; m; i++) {             int maxGold = dp[i][j + 1]; //              Maximum gold by going right from current             cell              if (i &gt; 0) {                 maxGold = max(maxGold, dp[i -                 1][j + 1]); // Maximum gold by going                 diagonal-up-right             }             if (i &lt; m - 1) {                 maxGold = max(maxGold, dp[i +                 1][j + 1]); // Maximum gold by going                 diagonal-down-right             }             dp[i][j] = arr[i][j] + maxGold; //             Total gold collected to reach current cell         }     } }</pre>	<p><b>Given Input Matrix (arr):</b></p> <table style="margin-left: auto; margin-right: auto;"> <tr><td>3</td><td>2</td><td>3</td><td>1</td></tr> <tr><td>2</td><td>4</td><td>6</td><td>0</td></tr> <tr><td>5</td><td>0</td><td>1</td><td>3</td></tr> <tr><td>9</td><td>1</td><td>5</td><td>1</td></tr> </table> <p><b>Step 1: Initialize dp Table</b></p> <ul style="list-style-type: none"> <li>• Copy the last column (<math>j = 3</math>) from arr to dp:</li> </ul> <table style="margin-left: auto; margin-right: auto;"> <tr><td>0</td><td>0</td><td>0</td><td>1</td></tr> <tr><td>0</td><td>0</td><td>0</td><td>0</td></tr> <tr><td>0</td><td>0</td><td>0</td><td>3</td></tr> <tr><td>0</td><td>0</td><td>0</td><td>1</td></tr> </table> <p><b>Step 2: Fill dp Table from Right to Left</b></p> <p><b>Column 2 (<math>j = 2</math>)</b></p> <p>Each <math>dp[i][j] = arr[i][j] + \max(dp[i][j+1], dp[i-1][j+1], dp[i+1][j+1])</math></p> <table style="margin-left: auto; margin-right: auto;"> <tr><td>0</td><td>0</td><td>4</td><td>1</td><td>→ 3 + max(1) = 4</td></tr> <tr><td>0</td><td>0</td><td>9</td><td>0</td><td>→ 6 + max(3,0) = 9</td></tr> <tr><td>0</td><td>0</td><td>6</td><td>3</td><td>→ 1 + max(3,1) = 6</td></tr> <tr><td>0</td><td>0</td><td>8</td><td>1</td><td>→ 5 + max(3) = 8</td></tr> </table> <p><b>Column 1 (<math>j = 1</math>)</b></p> <table style="margin-left: auto; margin-right: auto;"> <tr><td>0</td><td>11</td><td>4</td><td>1</td><td>→ 2 + max(4,9) = 11</td></tr> <tr><td>0</td><td>13</td><td>9</td><td>0</td><td>→ 4 + max(9,6) = 13</td></tr> <tr><td>0</td><td>9</td><td>6</td><td>3</td><td>→ 0 + max(6,8) = 9</td></tr> <tr><td>0</td><td>14</td><td>8</td><td>1</td><td>→ 1 + max(8) = 14</td></tr> </table> <p><b>Column 0 (<math>j = 0</math>)</b></p> <table style="margin-left: auto; margin-right: auto;"> <tr><td>13</td><td>11</td><td>4</td><td>1</td><td>→ 3 + max(11,13) = 13</td></tr> <tr><td>15</td><td>13</td><td>9</td><td>0</td><td>→ 2 + max(13,9) = 15</td></tr> <tr><td>18</td><td>9</td><td>6</td><td>3</td><td>→ 5 + max(9,14) = 18 ↘ (Expected max value)</td></tr> <tr><td>23</td><td>14</td><td>8</td><td>1</td><td>→ 9 + max(14) = 23</td></tr> </table> <p><b>Step 3: Find Maximum Gold in Column 0</b></p>	3	2	3	1	2	4	6	0	5	0	1	3	9	1	5	1	0	0	0	1	0	0	0	0	0	0	0	3	0	0	0	1	0	0	4	1	→ 3 + max(1) = 4	0	0	9	0	→ 6 + max(3,0) = 9	0	0	6	3	→ 1 + max(3,1) = 6	0	0	8	1	→ 5 + max(3) = 8	0	11	4	1	→ 2 + max(4,9) = 11	0	13	9	0	→ 4 + max(9,6) = 13	0	9	6	3	→ 0 + max(6,8) = 9	0	14	8	1	→ 1 + max(8) = 14	13	11	4	1	→ 3 + max(11,13) = 13	15	13	9	0	→ 2 + max(13,9) = 15	18	9	6	3	→ 5 + max(9,14) = 18 ↘ (Expected max value)	23	14	8	1	→ 9 + max(14) = 23
3	2	3	1																																																																																										
2	4	6	0																																																																																										
5	0	1	3																																																																																										
9	1	5	1																																																																																										
0	0	0	1																																																																																										
0	0	0	0																																																																																										
0	0	0	3																																																																																										
0	0	0	1																																																																																										
0	0	4	1	→ 3 + max(1) = 4																																																																																									
0	0	9	0	→ 6 + max(3,0) = 9																																																																																									
0	0	6	3	→ 1 + max(3,1) = 6																																																																																									
0	0	8	1	→ 5 + max(3) = 8																																																																																									
0	11	4	1	→ 2 + max(4,9) = 11																																																																																									
0	13	9	0	→ 4 + max(9,6) = 13																																																																																									
0	9	6	3	→ 0 + max(6,8) = 9																																																																																									
0	14	8	1	→ 1 + max(8) = 14																																																																																									
13	11	4	1	→ 3 + max(11,13) = 13																																																																																									
15	13	9	0	→ 2 + max(13,9) = 15																																																																																									
18	9	6	3	→ 5 + max(9,14) = 18 ↘ (Expected max value)																																																																																									
23	14	8	1	→ 9 + max(14) = 23																																																																																									

```

}

// Find the maximum gold collected in
// the first column
int maxGold = dp[0][0];
int maxRow = 0;
for (int i = 1; i < m; i++) {
    if (dp[i][0] > maxGold) {
        maxGold = dp[i][0];
        maxRow = i;
    }
}

// Print the maximum gold collected
cout << maxGold << endl;

// Queue to perform BFS for path tracing
queue<Pair> q;
q.push(Pair(maxRow, 0,
to_string(maxRow))); // Start from the cell
with maximum gold in the first column

// BFS to print all paths with maximum
gold collected
while (!q.empty()) {
    Pair rem = q.front();
    q.pop();

    if (rem.j == n - 1) {
        cout << rem.psf << endl; // Print
path when reaching the last column
    } else {
        int currentGold = dp[rem.i][rem.j];
        int rightGold = dp[rem.i][rem.j + 1];
        int diagonalUpGold = (rem.i > 0) ?
dp[rem.i - 1][rem.j + 1] : -1;
        int diagonalDownGold = (rem.i < m
- 1) ? dp[rem.i + 1][rem.j + 1] : -1;

        // Add paths to queue based on the
direction with maximum gold
        if (rightGold == currentGold -
arr[rem.i][rem.j + 1]) {
            q.push(Pair(rem.i, rem.j + 1,
rem.psf + " H")); // Move horizontally to the
right
        }
        if (diagonalUpGold == currentGold
- arr[rem.i - 1][rem.j + 1]) {
            q.push(Pair(rem.i - 1, rem.j + 1,
rem.psf + " LU")); // Move diagonally up-
right
        }
    }
}

```

- The **maximum gold collected is 18 at row 2.**

#### Step 4: Find All Paths (Using BFS)

Starting from  $dp[2][0] = 18$ :

- $dp[2][1] = 9$
- $dp[3][1] = 14$
- $dp[3][2] = 8$
- $dp[3][3] = 1$

#### Valid Path:

$2 \rightarrow LD \rightarrow 3 \rightarrow LU \rightarrow 3 \rightarrow H \rightarrow 1$

#### Final Output

Maximum Gold: 18  
Path: 2 LD 3 LU 3 H 1

```
        if (diagonalDownGold ==  
currentGold - arr[rem.i + 1][rem.j + 1]) {  
            q.push(Pair(rem.i + 1, rem.j + 1,  
rem.psf + " LD")); // Move diagonally down-  
right  
        }  
    }  
}  
  
int main() {  
    vector<vector<int>> arr = {  
        {3, 2, 3, 1},  
        {2, 4, 6, 0},  
        {5, 0, 1, 3},  
        {9, 1, 5, 1}  
    };  
  
    printMaxGoldPath(arr);  
  
    return 0;  
}
```

**Output:-**

18

### Print all path with minimum Cost In C++

```
#include <iostream>
#include <vector>
#include <queue>
using namespace std;

struct Pair {
    string psf; // path so far
    int i; // current row index
    int j; // current column index

    Pair(string psf, int i, int j) {
        this->psf = psf;
        this->i = i;
        this->j = j;
    }
};

void printAllPaths(vector<vector<int>>& arr) {
    int m = arr.size();
    int n = arr[0].size();

    // dp array to store minimum cost to
    // reach each cell
    vector<vector<int>> dp(m,
    vector<int>(n, 0));

    // Initialize dp table
    dp[m-1][n-1] = arr[m-1][n-1];
    for (int i = m - 2; i >= 0; i--) {
        dp[i][n-1] = arr[i][n-1] + dp[i + 1][n -
1];
    }
    for (int j = n - 2; j >= 0; j--) {
        dp[m-1][j] = arr[m-1][j] + dp[m - 1][j +
1];
    }
    for (int i = m - 2; i >= 0; i--) {
        for (int j = n - 2; j >= 0; j--) {
            dp[i][j] = arr[i][j] + min(dp[i][j + 1],
dp[i + 1][j]);
        }
    }

    // Minimum cost to reach the top-left
    // corner
    cout << dp[0][0] << endl;

    // Queue to perform BFS
    queue<Pair> q;
    q.push(Pair("", 0, 0));
}
```

### Dry Run of Minimum Cost Path Problem

We will compute the **dynamic programming (DP) table** step-by-step to ensure that we get the minimum cost sum **46** for the given matrix.

### Given Input Matrix (arr):

```
{1, 2, 3, 4},
{5, 6, 7, 8},
{9, 10, 11, 12},
{13, 14, 15, 16}
```

### Step 1: Understanding the DP Approach

1. **Base Case:** The last cell ( $dp[3][3]$ ) is the same as  $arr[3][3] = 16$ .
2. **Filling Last Row (Right to Left):**
  - o  $dp[i][j] = arr[i][j] + dp[i][j+1]$
3. **Filling Last Column (Bottom to Top):**
  - o  $dp[i][j] = arr[i][j] + dp[i+1][j]$
4. **Filling the Rest (Bottom-Up, Right-to-Left):**
  - o  $dp[i][j] = arr[i][j] + \min(dp[i+1][j], dp[i][j+1])$

### Step 2: Construct DP Table Step-by-Step

#### 1. Initialize $dp[3][3]$ (Bottom-Right Cell)

$$dp[3][3] = arr[3][3] = 16$$

#### 2. Fill the Last Row (Right to Left)

$$\begin{aligned} dp[i][j] &= arr[i][j] + dp[i][j+1] \\ dp[3][j] &= arr[3][j] + dp[3][j+1] \end{aligned}$$

$i=3$	$j=3$	$j=2$ $(15+16)$	$j=1$ $(14+31)$	$j=0$ $(13+45)$

```

while (!q.empty()) {
    Pair rem = q.front();
    q.pop();

    if (rem.i == m - 1 && rem.j == n - 1) {
        cout << rem.psf << endl; // print
        path when reaching the bottom-right
        corner
    } else if (rem.i == m - 1) {
        q.push(Pair(rem.psf + "H", rem.i,
        rem.j + 1)); // go right
    } else if (rem.j == n - 1) {
        q.push(Pair(rem.psf + "V", rem.i +
        1, rem.j)); // go down
    } else {
        if (dp[rem.i][rem.j + 1] < dp[rem.i + 1][rem.j]) {
            q.push(Pair(rem.psf + "H", rem.i,
            rem.j + 1)); // go right
        } else if (dp[rem.i][rem.j + 1] >
        dp[rem.i + 1][rem.j]) {
            q.push(Pair(rem.psf + "V", rem.i +
            1, rem.j)); // go down
        } else {
            q.push(Pair(rem.psf + "V", rem.i +
            1, rem.j)); // go down
            q.push(Pair(rem.psf + "H", rem.i,
            rem.j + 1)); // go right
        }
    }
}

int main() {
    vector<vector<int>> arr = {
        {1, 2, 3, 4},
        {5, 6, 7, 8},
        {9, 10, 11, 12},
        {13, 14, 15, 16}
    };

    printAllPaths(arr);
    return 0;
}

```

arr	16	15	14	13
dp	16	<b>31</b>	<b>45</b>	<b>58</b>

### 3. Fill the Last Column (Bottom to Top)

$$dp[i][j] = arr[i][j] + dp[i+1][j]$$

$$dp[i][j] = arr[i][j] + dp[i+1][j]$$

i=2	j=3 <b>(12+16)</b>	j=2	j=1	j=0
arr	12	11	10	9
dp	<b>28</b>	-	-	-
i=1	j=3 <b>(8+28)</b>	j=2	j=1	j=0
arr	8	7	6	5
dp	<b>36</b>	-	-	-
i=0	j=3 <b>(4+36)</b>	j=2	j=1	j=0
arr	4	3	2	1
dp	<b>40</b>	-	-	-

### 4. Fill the Rest of the DP Table

$$dp[i][j] = arr[i][j] + \min(dp[i+1][j], dp[i][j+1])$$

$$dp[i][j] = arr[i][j] + \min(dp[i+1][j], dp[i][j+1])$$

i=2	j=2 <b>(11+min(31, 28))</b>	j=1 <b>(10+min(41, 38))</b>	j=0 <b>(9+min(45, 40))</b>
arr	11	10	9
dp	<b>39</b>	<b>38</b>	<b>40</b>
i=1	j=2 <b>(7+min(39, 36))</b>	j=1 <b>(6+min(38, 44))</b>	j=0 <b>(5+min(45, 43))</b>
arr	7	6	5
dp	<b>43</b>	<b>44</b>	<b>45</b>

$i=0$	$j=2$ $(3+\min(43, 40))$	$j=1$ $(2+\min(41, 44))$	$j=0$ $(1+\min(45, 43))$
arr	3	2	1
dp	43	45	46

### Final DP Table

46	45	43	40
45	44	43	36
40	38	39	28
58	45	31	16

✓ Minimum Cost Path Sum = 46 (Matches G++ Output)

### Step 3: Extracting All Paths

Now, we use BFS (`queue<Pair>`) to trace all paths from  $(0, 0)$  to  $(3, 3)$  following the minimum cost. The paths may vary but should sum up to 46.

1. **Move Right H** if  $dp[i][j+1]$  is smaller.
2. **Move Down V** if  $dp[i+1][j]$  is smaller.
3. **If both are equal, try both paths (H and V).**

### Possible Paths (psf values in BFS)

V V V H H H (Down-Down-Down-Right-Right-Right)

### Output:-

46

HHVVVV

Rod cutting In C++																																																										
<pre>#include &lt;iostream&gt; #include &lt;vector&gt; #include &lt;algorithm&gt;  using namespace std; int solution(vector&lt;int&gt;&amp; prices) {     vector&lt;int&gt; np(prices.size() + 1);     for (int i = 0; i &lt; prices.size(); i++) {         np[i + 1] = prices[i];     }      vector&lt;int&gt; dp(np.size());     dp[0] = 0;     dp[1] = np[1];      for (int i = 2; i &lt; dp.size(); i++) {         dp[i] = np[i];          int li = 1;         int ri = i - 1;         while (li &lt;= ri) {             if (dp[li] + dp[ri] &gt; dp[i]) {                 dp[i] = dp[li] + dp[ri];             }              li++;             ri--;         }     }      return dp[dp.size() - 1]; }  int main() {     vector&lt;int&gt; prices = {1, 5, 8, 9, 10, 17,     17, 20};      cout &lt;&lt; solution(prices) &lt;&lt; endl;     return 0; }</pre>	<b>Dry Run (Tabular)</b> <p><b>Given Prices</b></p> <table style="margin-left: auto; margin-right: auto;"> <tr> <td>Length:</td> <td>1</td> <td>2</td> <td>3</td> <td>4</td> <td>5</td> <td>6</td> <td>7</td> <td>8</td> </tr> <tr> <td>Prices:</td> <td>1</td> <td>5</td> <td>8</td> <td>9</td> <td>10</td> <td>17</td> <td>17</td> <td>20</td> </tr> </table> <p><b>DP Computation Table</b></p> <table border="1" style="width: 100%; border-collapse: collapse;"> <thead> <tr> <th style="text-align: center;">Rod Length (np[i])</th> <th style="text-align: center;">Price (np[i])</th> <th style="text-align: center;">Possible Cuts (li, ri)</th> <th style="text-align: center;">Best Revenue (dp[i])</th> </tr> </thead> <tbody> <tr> <td style="text-align: center;">1</td> <td style="text-align: center;">1</td> <td style="text-align: center;">(1)</td> <td style="text-align: center;">1</td> </tr> <tr> <td style="text-align: center;">2</td> <td style="text-align: center;">5</td> <td style="text-align: center;"><math>(1,1) \rightarrow 1+1=2</math></td> <td style="text-align: center;">5</td> </tr> <tr> <td style="text-align: center;">3</td> <td style="text-align: center;">8</td> <td style="text-align: center;"><math>(1,2) \rightarrow 1+5=6, (2,1) \rightarrow 5+1=6</math></td> <td style="text-align: center;">8</td> </tr> <tr> <td style="text-align: center;">4</td> <td style="text-align: center;">9</td> <td style="text-align: center;"><math>(1,3) \rightarrow 1+8=9, (2,2) \rightarrow 5+5=10</math></td> <td style="text-align: center;">10</td> </tr> <tr> <td style="text-align: center;">5</td> <td style="text-align: center;">10</td> <td style="text-align: center;"><math>(1,4) \rightarrow 1+10=11, (2,3) \rightarrow 5+8=13</math></td> <td style="text-align: center;">13</td> </tr> <tr> <td style="text-align: center;">6</td> <td style="text-align: center;">17</td> <td style="text-align: center;"><math>(1,5) \rightarrow 1+13=14, (2,4) \rightarrow 5+10=15, (3,3) \rightarrow 8+8=16</math></td> <td style="text-align: center;">17</td> </tr> <tr> <td style="text-align: center;">7</td> <td style="text-align: center;">17</td> <td style="text-align: center;"><math>(1,6) \rightarrow 1+17=18, (2,5) \rightarrow 5+13=18, (3,4) \rightarrow 8+10=18</math></td> <td style="text-align: center;">18</td> </tr> <tr> <td style="text-align: center;">8</td> <td style="text-align: center;">20</td> <td style="text-align: center;"><math>(1,7) \rightarrow 1+18=19, (2,6) \rightarrow 5+17=22, (3,5) \rightarrow 8+13=21, (4,4) \rightarrow 10+10=20</math></td> <td style="text-align: center;">22</td> </tr> </tbody> </table> <p><b>Final answer</b></p> <p>The maximum revenue we can get for <b>length = 8</b> is <b>22</b>.</p>				Length:	1	2	3	4	5	6	7	8	Prices:	1	5	8	9	10	17	17	20	Rod Length (np[i])	Price (np[i])	Possible Cuts (li, ri)	Best Revenue (dp[i])	1	1	(1)	1	2	5	$(1,1) \rightarrow 1+1=2$	5	3	8	$(1,2) \rightarrow 1+5=6, (2,1) \rightarrow 5+1=6$	8	4	9	$(1,3) \rightarrow 1+8=9, (2,2) \rightarrow 5+5=10$	10	5	10	$(1,4) \rightarrow 1+10=11, (2,3) \rightarrow 5+8=13$	13	6	17	$(1,5) \rightarrow 1+13=14, (2,4) \rightarrow 5+10=15, (3,3) \rightarrow 8+8=16$	17	7	17	$(1,6) \rightarrow 1+17=18, (2,5) \rightarrow 5+13=18, (3,4) \rightarrow 8+10=18$	18	8	20	$(1,7) \rightarrow 1+18=19, (2,6) \rightarrow 5+17=22, (3,5) \rightarrow 8+13=21, (4,4) \rightarrow 10+10=20$	22
Length:	1	2	3	4	5	6	7	8																																																		
Prices:	1	5	8	9	10	17	17	20																																																		
Rod Length (np[i])	Price (np[i])	Possible Cuts (li, ri)	Best Revenue (dp[i])																																																							
1	1	(1)	1																																																							
2	5	$(1,1) \rightarrow 1+1=2$	5																																																							
3	8	$(1,2) \rightarrow 1+5=6, (2,1) \rightarrow 5+1=6$	8																																																							
4	9	$(1,3) \rightarrow 1+8=9, (2,2) \rightarrow 5+5=10$	10																																																							
5	10	$(1,4) \rightarrow 1+10=11, (2,3) \rightarrow 5+8=13$	13																																																							
6	17	$(1,5) \rightarrow 1+13=14, (2,4) \rightarrow 5+10=15, (3,3) \rightarrow 8+8=16$	17																																																							
7	17	$(1,6) \rightarrow 1+17=18, (2,5) \rightarrow 5+13=18, (3,4) \rightarrow 8+10=18$	18																																																							
8	20	$(1,7) \rightarrow 1+18=19, (2,6) \rightarrow 5+17=22, (3,5) \rightarrow 8+13=21, (4,4) \rightarrow 10+10=20$	22																																																							
Output:- 22																																																										

Temple offering In C++					
<pre>#include &lt;iostream&gt; #include &lt;algorithm&gt; using namespace std;  int totalOfferings(int* height, int n) {     int* larr = new int[n]; // Left offerings array     int* rarr = new int[n]; // Right offerings array     // Calculate left offerings     larr[0] = 1;     for (int i = 1; i &lt; n; i++) {         if (height[i] &gt; height[i - 1]) {             larr[i] = larr[i - 1] + 1;         } else {             larr[i] = 1;         }     }      // Calculate right offerings     rarr[n - 1] = 1;     for (int i = n - 2; i &gt;= 0; i--) {         if (height[i] &gt; height[i + 1]) {             rarr[i] = rarr[i + 1] + 1;         } else {             rarr[i] = 1;         }     }      // Calculate total offerings     int ans = 0;     for (int i = 0; i &lt; n; i++) {         ans += max(larr[i], rarr[i]);     }      // Free allocated memory     delete[] larr;     delete[] rarr;      return ans; }  int main() {     int height[] = {2, 3, 5, 6, 4, 8, 9};     int n = sizeof(height) / sizeof(height[0]);     cout &lt;&lt; totalOfferings(height, n) &lt;&lt;     endl;     return 0; }</pre>					
<b>Input:</b>					
height[] = {2, 3, 5, 6, 4, 8, 9}					
Index i	Height height[i] ]	Left Offerings larr[i]	Right Offerings rarr[i]	Final Offerings max(lar r[i], rarr[i] )	
0	2	1	1	1	
1	3	2	1	2	
2	5	3	1	3	
3	6	4	2	4	
4	4	1	1	1	
5	8	2	2	2	
6	9	3	3	3	
<b>Total Offerings:</b>					
1 + 2 + 3 + 4 + 1 + 2 + 3 = 16					
<b>✓ Output:</b>					
16					
<b>Output:-</b>					
16					

## Word Break In C++

```
#include <iostream>
#include <unordered_set>
#include <vector>
using namespace std;

bool solution(string sentence,
unordered_set<string>& dict) {
    int n = sentence.length();
    vector<int> dp(n, 0);

    for (int i = 0; i < n; i++) {
        for (int j = 0; j <= i; j++) {
            string word = sentence.substr(j, i - j
+ 1);
            if (dict.find(word) != dict.end()) {
                if (j > 0) {
                    dp[i] += dp[j - 1];
                } else {
                    dp[i] += 1;
                }
            }
        }
    }

    cout << dp[n - 1] << endl;
    return dp[n - 1] > 0;
}

int main() {
    unordered_set<string> dict = {"i", "like",
"pep", "coding", "pepper", "eating",
"mango", "man", "go", "in", "pepcoding"};
    string sentence =
"ilikepeppereatingmangoinpepcoding";

    cout << boolalpha << solution(sentence,
dict) << endl;
    return 0;
}
```

### Iterative Tabular Dry Run for Word Break Problem

We will dry-run the **fixed DP approach** using the sentence: **"ilikepeppereatingmangoinpepcoding"**

#### Dictionary:

```
{"i", "like", "pep", "coding", "pepper",
"eating", "mango", "man", "go", "in",
"pepcoding"}
```

#### Step 1: Define DP Table

- Let  $dp[i]$  represent whether the substring  $s[0...i-1]$  can be segmented.
- We initialize  $dp[0] = \text{true}$  (empty string is always valid).
- We will iterate over all positions  $i$  and check all possible substrings  $s[j...i-1]$  to see if they exist in the dictionary and if  $dp[j]$  is true.

#### Step 2: Iterative Dry Run in Tabular Form

<b>i</b>	<b>Substring (sentence[0...i-1])</b>	<b>Valid Segment Found?</b>	<b>dp[i] Value</b>
0	""	Base case	true
1	"i"	✓ ("i" in dict)	true
2	"il"	✗	false
3	"ili"	✗	false
4	"ilik"	✗	false
5	"ilike"	✓ ("like" in dict, $dp[1]$ is true)	true
6	"ilikep"	✗	false
7	"ilikepe"	✗	false
8	"ilike pep"	✓ ("pep")	true

		in dict, dp[5] is true)	
9	"ilikepepp"	✗	false
10	"ilikepeppe"	✗	false
11	"ilikepepper"	✓ ("pepper" in dict, dp[5] is true)	true
12	"ilikepeppere"	✗	false
13	"ilikepepperea"	✗	false
14	"ilikepeppereat"	✗	false
15	"ilikepeppereati"	✗	false
16	"ilikepeppereatin"	✗	false
17	"ilikepeppereating"	✓ ("eating" in dict, dp[11] is true)	true
18	"ilikepeppereatingm"	✗	false
19	"ilikepeppereatingma "	✗	false
20	"ilikepeppereatingma n"	✓ ("man" in dict, dp[17] is true)	true
21	"ilikepeppereatingma ng"	✗	false
22	"ilikepeppereatingma ngo"	✓ ("mango" in dict, dp[17] is true)	true
23	"ilikepeppereatingma ngoi"	✗	false
24	"ilikepeppereatingma ngoin"	✓ ("in" in dict, dp[22] is true)	true

25	"ilikepeppereatingma ngoinp"	✗	false
26	"ilikepeppereatingma ngoinpe"	✗	false
27	"ilikepeppereatingma ngoinpep"	✓ ("pep" in dict, dp[24] is true)	true
28	"ilikepeppereatingma ngoinpepc"	✗	false
29	"ilikepeppereatingma ngoinpepcō"	✗	false
30	"ilikepeppereatingma ngoinpepcod"	✗	false
31	"ilikepeppereatingma ngoinpepcodi"	✗	false
32	"ilikepeppereatingma ngoinpepcodin"	✗	false
33	"ilikepeppereatingma ngoinpepcoding"	✓ ("pepcodi ng" in dict, dp[24] is true)	true

### Step 3: Final dp Array

```
[ T  T  F  F  F  T  F  F  T  F  F  F  T  F
F  F  F  F  T  F  F  T  F  T  F  T  F  T  F
T  F  F  F  F  F  T ]
```

Since  $dp[n] = dp[33] = \text{true}$ , we conclude that the sentence can be segmented into words from the dictionary.

Output:-

```
4
true
```