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;
}
```

Dry Run:

Given arr = {2, 5, 9, 12, 15, 18, 22, 26, 30, 34, 36, 38, 40, 41}:

1. **For i = 2:**
   arr[2] - arr[1] = 9 - 5 = 4, arr[1] - arr[0] = 5 - 2 = 3
   Not equal, no update for dp[2].
2. **For i = 3:**
   arr[3] - arr[2] = 12 - 9 = 3, arr[2] - arr[1] = 9 - 5 = 4
   Not equal, no update for dp[3].
3. **For i = 4:**
   arr[4] - arr[3] = 15 - 12 = 3, arr[3] - arr[2] = 12 - 9 = 3
   Equal, so dp[4] = dp[3] + 1 = 0 + 1 = 1.
   Add dp[4] to ans: ans = 1.
4. **For i = 5:**
   arr[5] - arr[4] = 18 - 15 = 3, arr[4] - arr[3] = 15 - 12 = 3
   Equal, so dp[5] = dp[4] + 1 = 1 + 1 = 2.
   Add dp[5] to ans: ans = 1 + 2 = 3.
5. **For i = 6:**
   arr[6] - arr[5] = 22 - 18 = 4, arr[5] - arr[4] = 18 - 15 = 3
   Not equal, no update for dp[6].
6. **For i = 7:**
   arr[7] - arr[6] = 26 - 22 = 4, arr[6] - arr[5] = 22 - 18 = 4
   Equal, so dp[7] = dp[6] + 1 = 0 + 1 = 1.
   Add dp[7] to ans: ans = 3 + 1 = 4.
7. **For i = 8:**
   arr[8] - arr[7] = 30 - 26 = 4, arr[7] - arr[6] = 26 - 22 = 4
   Equal, so dp[8] = dp[7] + 1 = 1 + 1 = 2.
   Add dp[8] to ans: ans = 4 + 2 = 6.
8. **For i = 9:**
   arr[9] - arr[8] = 34 - 30 = 4, arr[8] - arr[7] = 30 - 26 = 4
   Equal, so dp[9] = dp[8] + 1 = 2 + 1 = 3.
   Add dp[9] to ans: ans = 6 + 3 = 9.
9. **For i = 10:**
   arr[10] - arr[9] = 36 - 34 = 2, arr[9] - arr[8] = 34 - 30 = 4
   Not equal, no update for dp[10].
10. **For i = 11:**
    arr[11] - arr[10] = 38 - 36 = 2, arr[10] - arr[9] = 36 - 34 = 2
    Equal, so dp[11] = dp[10] + 1 = 0 + 1 = 1.
    Add dp[11] to ans: ans = 9 + 1 = 10.
11. **For i = 12:**
    arr[12] - arr[11] = 40 - 38 = 2, arr[11] - arr[10] = 38 - 36 = 2
    Equal, so dp[12] = dp[11] + 1 = 1 + 1 = 2.
    Add dp[12] to ans: ans = 10 + 2 = 12.
12. **For i = 13:**
    arr[13] - arr[12] = 41 - 40 = 1, arr[12] - arr[11] = 40 - 38 = 2
    Not equal, no update for dp[13].

<table>
<tr><td></td><td></td></tr>
<tr><td colspan="2">Output:-
12</td></tr>
</table>

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;
}
```

Initial Setup:

- n = 5
- dp is a vector of size n + 1 = 6, initially set to {1, 1, 0, 0, 0, 0}.

Loop Breakdown:

Iteration 1: i = 2

1. inside = 2 - 1 = 1
2. outside = 0

For inside = 1 and outside = 0:

  - dp[2] += dp[1] * dp[0] → dp[2] += 1 * 1 → dp[2] = 1.

Now, decrease inside to 0 and increase outside to 1.

For inside = 0 and outside = 1:

  - dp[2] += dp[0] * dp[1] → dp[2] += 1 * 1 → dp[2] = 2.

So, after this iteration, dp[2] = 2.

Iteration 2: i = 3

1. inside = 3 - 1 = 2
2. outside = 0

For inside = 2 and outside = 0:

  - dp[3] += dp[2] * dp[0] → dp[3] += 2 * 1 → dp[3] = 2.

Now, decrease inside to 1 and increase outside to 1.

For inside = 1 and outside = 1:

  - dp[3] += dp[1] * dp[1] → dp[3] += 1 * 1 → dp[3] = 3.

Now, decrease inside to 0 and increase outside to 2.

For inside = 0 and outside = 2:

  - dp[3] += dp[0] * dp[2] → dp[3] += 1 * 2 → dp[3] = 5.

So, after this iteration, dp[3] = 5.

Iteration 3: i = 4

1. inside = 4 - 1 = 3
2. outside = 0

   For inside = 3 and outside = 0:

   - dp[4] += dp[3] * dp[0] → dp[4] += 5 * 1 → dp[4] = 5.

   Now, decrease inside to 2 and increase outside to 1.

   For inside = 2 and outside = 1:

   - dp[4] += dp[2] * dp[1] → dp[4] += 2 * 1 → dp[4] = 7.

   Now, decrease inside to 1 and increase outside to 2.

   For inside = 1 and outside = 2:

   - dp[4] += dp[1] * dp[2] → dp[4] += 1 * 2 → dp[4] = 9.

   Now, decrease inside to 0 and increase outside to 3.

   For inside = 0 and outside = 3:

   - dp[4] += dp[0] * dp[3] → dp[4] += 1 * 5 → dp[4] = 14.

   So, after this iteration, dp[4] = 14.

Iteration 4: i = 5

1. inside = 5 - 1 = 4
2. outside = 0

   For inside = 4 and outside = 0:

   - dp[5] += dp[4] * dp[0] → dp[5] += 14 * 1 → dp[5] = 14.

   Now, decrease inside to 3 and increase outside to 1.

   For inside = 3 and outside = 1:

   - dp[5] += dp[3] * dp[1] → dp[5] += 5 * 1 → dp[5] = 19.

   Now, decrease inside to 2 and increase outside to 2.

For inside = 2 and outside = 2:

- dp[5] += dp[2] * dp[2] → dp[5] += 2 * 2 → dp[5] = 23.

Now, decrease inside to 1 and increase outside to 3.

For inside = 1 and outside = 3:

- dp[5] += dp[1] * dp[3] → dp[5] += 1 * 5 → dp[5] = 28.

Now, decrease inside to 0 and increase outside to 4.

For inside = 0 and outside = 4:

- dp[5] += dp[0] * dp[4] → dp[5] += 1 * 14 → dp[5] = 42.

So, after this iteration, dp[5] = 42.

Final Output:

The dp array is:

Copy code

```
1 1 2 5 14 42
```

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;
}
```

Step-by-Step Dry Run

Given input:

int arr[] = {2, 3, 5};

Here, the balloons' values are 2, 3, and 5.

- We initialize the dp array with zeros.

First Iteration (gap = 0, considering only single balloons):

- **i = 0, j = 0**:
    - maxCoins = INT_MIN
    - Only one balloon at index 0, so the value for bursting it is 1 * 2 * 1 = 2. The result is stored in dp[0][0].
- **i = 1, j = 1**:
    - maxCoins = INT_MIN
    - Only one balloon at index 1, so the value for bursting it is 1 * 3 * 1 = 3. The result is stored in dp[1][1].
- **i = 2, j = 2**:
    - maxCoins = INT_MIN
    - Only one balloon at index 2, so the value for bursting it is 1 * 5 * 1 = 5. The result is stored in dp[2][2].

Second Iteration (gap = 1, considering two consecutive balloons):

- **i = 0, j = 1**:
    - We check two possible balloons to burst, k = 0 and k = 1.
    - If we burst k = 0 first, the coins obtained are:
        - Left: 0, Right: dp[1][1] = 3, Value from bursting: 1 * 2 * 3 = 6, so total = 6 + 3 = 9.
    - If we burst k = 1 first, the coins obtained are:
        - Left: dp[0][0] = 2, Right: 0, Value from bursting: 1 * 3 * 5 = 15, so total = 2 + 15 = 17.
    - We store the maximum value 17 in dp[0][1].
- **i = 1, j = 2**:
    - We check two possible balloons to burst, k = 1 and k = 2.

	○ If we burst k = 1 first, the coins obtained are: ▪ Left: 0, Right: dp[2][2] = 5, Value from bursting: 2 * 3 * 5 = 30, so total = 30 + 5 = 35. ○ If we burst k = 2 first, the coins obtained are: ▪ Left: dp[1][1] = 3, Right: 0, Value from bursting: 2 * 3 * 5 = 30, so total = 3 + 30 = 33. ○ We store the maximum value 35 in dp[1][2]. **Third Iteration (gap = 2, considering the whole array):** • **i = 0, j = 2**: ○ We check three possible balloons to burst, k = 0, k = 1, and k = 2. ○ If we burst k = 0 first: ▪ Left: 0, Right: dp[1][2] = 35, Value from bursting: 1 * 2 * 5 = 10, so total = 10 + 35 = 45. ○ If we burst k = 1 first: ▪ Left: dp[0][0] = 2, Right: dp[2][2] = 5, Value from bursting: 1 * 3 * 5 = 15, so total = 2 + 15 + 5 = 22. ○ If we burst k = 2 first: ▪ Left: dp[0][1] = 17, Right: 0, Value from bursting: 1 * 3 * 5 = 15, so total = 17 + 15 = 32. ○ The maximum value 45 is stored in dp[0][2]. **Final Result:** • The value in dp[0][2] (maximum coins from bursting all balloons) is 45.
Output:- 45	

Catalan in C++

```cpp
#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;
}
```

This is essentially using the Catalan number recurrence relation.

Iteration 1: i = 2

- dp[2] = 0
- For j = 0: dp[2] += dp[0] * dp[1] = 0 + 1 * 1 = 1
- For j = 1: dp[2] += dp[1] * dp[0] = 1 + 1 * 1 = 2
- So, dp[2] = 2.

Iteration 2: i = 3

- dp[3] = 0
- For j = 0: dp[3] += dp[0] * dp[2] = 0 + 1 * 2 = 2
- For j = 1: dp[3] += dp[1] * dp[1] = 2 + 1 * 1 = 3
- For j = 2: dp[3] += dp[2] * dp[0] = 3 + 2 * 1 = 5
- So, dp[3] = 5.

Iteration 3: i = 4

- dp[4] = 0
- For j = 0: dp[4] += dp[0] * dp[3] = 0 + 1 * 5 = 5
- For j = 1: dp[4] += dp[1] * dp[2] = 5 + 1 * 2 = 7
- For j = 2: dp[4] += dp[2] * dp[1] = 7 + 2 * 1 = 9
- For j = 3: dp[4] += dp[3] * dp[0] = 9 + 5 * 1 = 14
- So, dp[4] = 14.

Iteration 4: i = 5

- dp[5] = 0
- For j = 0: dp[5] += dp[0] * dp[4] = 0 + 1 * 14 = 14
- For j = 1: dp[5] += dp[1] * dp[3] = 14 + 1 * 5 = 19
- For j = 2: dp[5] += dp[2] * dp[2] = 19 + 2 * 2 = 23
- For j = 3: dp[5] += dp[3] * dp[1] = 23 + 5 * 1 = 28
- For j = 4: dp[5] += dp[4] * dp[0] = 28 + 14 * 1 = 42
- So, dp[5] = 42.

Final Output:

The dp array is:

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;
}
```

Step-by-Step Dry Run:

Input:

string str = "abc";

- Length of the string n = 3.

Initialization:

dp[0] = 1; // Empty subsequence
unordered_map<char, int> lastOccurrence;

- Initially, dp = [1, 0, 0, 0] (the first element is 1 for the empty subsequence).
- lastOccurrence is empty.

Iteration 1 (i = 1, character = 'a'):

- dp[1] = 2 * dp[0] = 2 * 1 = 2 (considering subsequences from previous).
- 'a' has not been seen before, so no need to subtract.
- lastOccurrence['a'] = 1.
- After this iteration, dp = [1, 2, 0, 0].

Iteration 2 (i = 2, character = 'b'):

- dp[2] = 2 * dp[1] = 2 * 2 = 4.
- 'b' has not been seen before, so no need to subtract.
- lastOccurrence['b'] = 2.
- After this iteration, dp = [1, 2, 4, 0].

Iteration 3 (i = 3, character = 'c'):

- dp[3] = 2 * dp[2] = 2 * 4 = 8.
- 'c' has not been seen before, so no need to subtract.
- lastOccurrence['c'] = 3.
- After this iteration, dp = [1, 2, 4, 8].

Final Result:

- dp[n] = dp[3] = 8.
- Subtract 1 to exclude the empty subsequence: 8 - 1 = 7.

Output:

7

Explanation of Output:

The distinct subsequences of "abc" are:

- "" (empty subsequence)
- "a"

	- "b" - "c" - "ab" - "ac" - "bc" - "abc" Thus, there are **7 distinct subsequences**, excluding the empty subsequence.
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;
}
```

Initial Setup:

- str = "abccbc"
- n = 6 (length of the string)
- dp is a 2D array where dp[i][j] represents the number of palindromic subsequences in the substring str[i...j].

Step-by-Step Execution:

Initialize dp:

- For g = 0 (single character substrings), we initialize dp[i][i] = 1, since any single character is a palindrome by itself.
- For g = 1 (two-character substrings), we initialize dp[i][i+1] = 2 if the characters are the same, otherwise dp[i][i+1] = 1.

Loop through all values of g (gap between i and j):

For each g, we calculate dp[i][j] where j = i + g and i is the starting index.

Iteration 1: g = 0 (substrings of length 1)

We go through each character and set dp[i][i] = 1 (each character is a palindrome of length 1).

- dp[0][0] = 1 (for 'a')
- dp[1][1] = 1 (for 'b')
- dp[2][2] = 1 (for 'c')
- dp[3][3] = 1 (for 'c')
- dp[4][4] = 1 (for 'b')
- dp[5][5] = 1 (for 'c')

Iteration 2: g = 1 (substrings of length 2)

We check pairs of consecutive characters. If they are the same, dp[i][i+1] = 2, otherwise dp[i][i+1] = 1.

- dp[0][1] = 1 (for "ab" as 'a' != 'b')
- dp[1][2] = 1 (for "bc" as 'b' != 'c')
- dp[2][3] = 2 (for "cc" as 'c' == 'c')
- dp[3][4] = 1 (for "cb" as 'c' != 'b')
- dp[4][5] = 1 (for "bc" as 'b' != 'c')

Iteration 3: g = 2 (substrings of length 3)

Now we look at substrings of length 3 and calculate dp[i][j] for each.

- dp[0][2] = 2 (for "abc" as 'a' != 'c', using the formula dp[0][1] + dp[1][2] - dp[1][1])
- dp[1][3] = 3 (for "bcc" as 'b' != 'c', using the formula dp[1][2] + dp[2][3] - dp[2][2])

	- dp[2][4] = 3 (for "ccc" as 'c' == 'c', using the formula dp[2][3] + dp[3][4] + 1) - dp[3][5] = 3 (for "cbc" as 'c' == 'c', using the formula dp[3][4] + dp[4][5] + 1) **Iteration 4: g = 3 (substrings of length 4)** - dp[0][3] = 3 (for "abcc" as 'a' != 'c', using the formula dp[0][2] + dp[1][3] - dp[1][2]) - dp[1][4] = 4 (for "bccb" as 'b' == 'b', using the formula dp[1][3] + dp[2][4] + 1) - dp[2][5] = 6 (for "ccbc" as 'c' != 'c', using the formula dp[2][4] + dp[3][5] - dp[3][4]) **Iteration 5: g = 4 (substrings of length 5)** - dp[0][4] = 5 (for "abccb" as 'a' != 'b', using the formula dp[0][3] + dp[1][4] - dp[1][3]) - dp[1][5] = 7 (for "bccbc" as 'b' != 'c', using the formula dp[1][4] + dp[2][5] - dp[2][4]) **Iteration 6: g = 5 (substrings of length 6)** - dp[0][5] = 9 (for "abccbc" as 'a' != 'c', using the formula dp[0][4] + dp[1][5] - dp[1][4]) **Final Output:** The result stored in dp[0][5] (which is the value representing the number of palindromic subsequences in the entire string "abccbc") is 9
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;
}
```

Dry Run Example for n = 5

Let's break down the example when n = 5.

1. **Initialization**:
    - dp[0] = 1 (One way to form an empty sequence).
    - dp[1] = 1 (One way to form a sequence of length 1, either "V" or "M").
2. **Filling dp[2] to dp[5]**:
    - **For i = 2**:
        - dp[2] = dp[0] * dp[1] + dp[1] * dp[0]
        - dp[2] = 1 * 1 + 1 * 1 = 2
    - **For i = 3**:
        - dp[3] = dp[0] * dp[2] + dp[1] * dp[1] + dp[2] * dp[0]
        - dp[3] = 1 * 2 + 1 * 1 + 2 * 1 = 5
    - **For i = 4**:
        - dp[4] = dp[0] * dp[3] + dp[1] * dp[2] + dp[2] * dp[1] + dp[3] * dp[0]
        - dp[4] = 1 * 5 + 1 * 2 + 2 * 1 + 5 * 1 = 14
    - **For i = 5**:
        - dp[5] = dp[0] * dp[4] + dp[1] * dp[3] + dp[2] * dp[2] + dp[3] * dp[1] + dp[4] * dp[0]
        - dp[5] = 1 * 14 + 1 * 5 + 2 * 2 + 5 * 1 + 14 * 1 = 42
3. **Output**:
    - The final value of dp[5] is 42, which is the number of valid valley-mountain sequences of length 5.

Output:-

```
42
```

Edit Distance C++

```cpp
#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();

    // Initialize the 2D array with dimensions (m+1) x (n+1)
    int dp[m + 1][n + 1] = {0};

    // Fill the dp array
    for (int i = 0; i <= m; i++) {
        for (int j = 0; j <= n; j++) {
            if (i == 0) {
                dp[i][j] = j; // If s1 is empty, insert all characters of s2
            } else if (j == 0) {
                dp[i][j] = i; // If s2 is empty, remove all characters of s1
            } else {
                int f1 = 1 + dp[i - 1][j - 1]; // Replace
                int f2 = 1 + dp[i - 1][j];     // Delete
                int f3 = 1 + dp[i][j - 1];     // Insert
                dp[i][j] = min({f1, f2, f3});
            }
        }
    }

    cout << dp[m][n] << endl; // Output the result

    return 0;
}
```

Dry Run of the Program

Let's go through the dry run of the code with the input strings s1 = "cat" and s2 = "cut".

Input:

- s1 = "cat"
- s2 = "cut"
- m = 3 (Length of s1)
- n = 3 (Length of s2)

Step 1: Initialize the dp array

We create a 2D DP table with dimensions (m+1) x (n+1), which is a 4x4 table since m = 3 and n = 3.

int dp[4][4] = {0};

Step 2: Fill the dp table

Now, let's fill the table using the given conditions.

1. **When i = 0** (Empty string s1):
   - dp[0][0] = 0 (Both strings are empty)
   - dp[0][1] = 1 (Insert 1 character 'c' from s2)
   - dp[0][2] = 2 (Insert 2 characters 'cu' from s2)
   - dp[0][3] = 3 (Insert 3 characters 'cut' from s2)

   The first row looks like this:

   dp[0] = {0, 1, 2, 3}

2. **When j = 0** (Empty string s2):
   - dp[1][0] = 1 (Remove 1 character 'c' from s1)
   - dp[2][0] = 2 (Remove 2 characters 'ca' from s1)
   - dp[3][0] = 3 (Remove 3 characters 'cat' from s1)

   The first column looks like this:

   dp[1] = {1, 0, 0, 0}
   dp[2] = {2, 0, 0, 0}
   dp[3] = {3, 0, 0, 0}

3. **When i = 1 and j = 1** (comparing 'c' and 'c'):
   - Since s1[0] == s2[0], no operation is required.
   - dp[1][1] = dp[0][0] = 0

After this, the table looks like this:

dp[1] = {1, 0, 0, 0}

4. **When i = 1 and j = 2** (comparing 'c' and 'u'):
    - We need to perform an **insert** operation.
    - dp[1][2] = 1 + min(dp[0][2], dp[1][1], dp[0][1]) = 1 + min(2, 0, 1) = 1 + 1 = 2

   After this step, the table looks like:

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

5. **When i = 1 and j = 3** (comparing 'c' and 't'):
    - We need to perform an **insert** operation.
    - dp[1][3] = 1 + min(dp[0][3], dp[1][2], dp[0][2]) = 1 + min(3, 2, 2) = 1 + 2 = 3

   After this step, the table looks like:

   dp[1] = {1, 0, 2, 3}

Continue filling the rest of the table similarly:

6. **When i = 2 and j = 1** (comparing 'a' and 'c'):
    - We need to perform a **delete** operation.
    - dp[2][1] = 1 + min(dp[1][1], dp[2][0], dp[1][0]) = 1 + min(0, 2, 1) = 1 + 0 = 1
7. **When i = 2 and j = 2** (comparing 'a' and 'u'):
    - We need to perform a **replace** operation.
    - dp[2][2] = 1 + min(dp[1][2], dp[2][1], dp[1][1]) = 1 + min(2, 1, 0) = 1 + 0 = 1
8. **When i = 2 and j = 3** (comparing 'a' and 't'):
    - We need to perform an **insert** operation.
    - dp[2][3] = 1 + min(dp[1][3], dp[2][2], dp[1][2]) = 1 + min(3, 1, 2) = 1 + 1 = 2
9. **When i = 3 and j = 1** (comparing 't' and 'c'):
    - We need to perform a **delete** operation.
    - dp[3][1] = 1 + min(dp[2][1], dp[3][0], dp[2][0]) = 1 + min(1, 3, 2) = 1 + 1 = 2

10. **When i = 3 and j = 2** (comparing 't' and 'u'):
    - We need to perform a **replace** operation.
    - dp[3][2] = 1 + min(dp[2][2], dp[3][1], dp[2][1]) = 1 + min(1, 2, 1) = 1 + 1 = 2
11. **When i = 3 and j = 3** (comparing 't' and 't'):
    - Since s1[2] == s2[2], no operation is required.
    - dp[3][3] = dp[2][2] = 1

Final DP Table:

dp[0] = {0, 1, 2, 3}
dp[1] = {1, 0, 2, 3}
dp[2] = {2, 1, 1, 2}
dp[3] = {3, 2, 2, 1}

Final Output:

The value at dp[m][n] is dp[3][3] = 3.

So, the minimum number of operations (insertions, deletions, or replacements) required to convert "cat" to "cut" is **3**.

Output:-
3

Egg drop C++

```cpp
#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] = {0}; // Array with (n + 1) rows and (k + 1) columns

    // 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;
}
```

Dry Run of the Program

Let's go through the dry run of the eggDrop function with n = 4 (number of eggs) and k = 2 (number of floors).

Input:

- n = 4 (number of eggs)
- k = 2 (number of floors)

Step 1: Initialize the DP Table

The DP table is a 2D array of size (n+1) x (k+1):

int dp[n+1][k+1] = {0}; // Array with 5 rows (0 to 4 eggs) and 3 columns (0 to 2 floors)

So, the DP table initially looks like this:

```
dp[0] = {0, 0, 0}  // 0 eggs: impossible to drop
dp[1] = {0, 0, 0}  // 1 egg
dp[2] = {0, 0, 0}  // 2 eggs
dp[3] = {0, 0, 0}  // 3 eggs
dp[4] = {0, 0, 0}  // 4 eggs
```

Step 2: Fill the DP Table

- **Case 1: One egg (i = 1)**

If we have only one egg (i = 1), the number of trials needed to check all j floors is equal to j because we must start from the 1st floor and test each floor one by one. This is why dp[1][j] = j.

So, for i = 1:

- dp[1][1] = 1 (1 trial for 1 floor)
- dp[1][2] = 2 (2 trials for 2 floors)

At this point, the table looks like this:

```
dp[0] = {0, 0, 0}
dp[1] = {0, 1, 2}
dp[2] = {0, 0, 0}
dp[3] = {0, 0, 0}
dp[4] = {0, 0, 0}
```

- **Case 2: One floor (j = 1)**

If we have only one floor (j = 1), then only 1 trial is needed, no matter how many eggs we have. So, for all i (eggs), dp[i][1] = 1.

At this point, the table becomes:

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

dp[2] = {0, 1, 0}
dp[3] = {0, 1, 0}
dp[4] = {0, 1, 0}

- **Case 3: More than 1 egg and more than 1 floor (i > 1, j > 1)**

Now, we compute the minimum number of trials for each i (eggs) and j (floors) by testing each floor from 1 to j and determining the worst-case number of drops.

For each floor floor:

- If the egg breaks, we look at the number of drops for i - 1 eggs and floor - 1 floors (dp[i - 1][floor - 1]).
- If the egg survives, we look at the number of drops for i eggs and j - floor floors (dp[i][j - floor]).
- The worst-case number of drops is 1 + max(breaks, survives).
- We want to minimize this worst-case number of drops.

Let's calculate the values for each combination of i and j.

For i = 2, j = 2:

- Try floor 1:
  - If the egg breaks, check dp[1][0] (0 floors) → 0 drops.
  - If the egg survives, check dp[2][1] (1 floor) → 1 drop.
  - So, max(0, 1) + 1 = 2 drops.

Therefore, dp[2][2] = 2.

The table becomes:

dp[0] = {0, 0, 0}
dp[1] = {0, 1, 2}
dp[2] = {0, 1, 2}
dp[3] = {0, 1, 0}
dp[4] = {0, 1, 0}

For i = 3, j = 2:

- Try floor 1:
  - If the egg breaks, check dp[2][0] → 0 drops.
  - If the egg survives, check dp[3][1] → 1 drop.
  - max(0, 1) + 1 = 2 drops.

Therefore, dp[3][2] = 2.

The table becomes:

dp[0] = {0, 0, 0}
dp[1] = {0, 1, 2}
dp[2] = {0, 1, 2}
dp[3] = {0, 1, 2}
dp[4] = {0, 1, 0}

For i = 4, j = 2:

- Try floor 1:
    - If the egg breaks, check dp[3][0] → 0 drops.
    - If the egg survives, check dp[4][1] → 1 drop.
    - max(0, 1) + 1 = 2 drops.

Therefore, dp[4][2] = 2.

The table becomes:

dp[0] = {0, 0, 0}
dp[1] = {0, 1, 2}
dp[2] = {0, 1, 2}
dp[3] = {0, 1, 2}
dp[4] = {0, 1, 2}

Step 3: Output the Result

The final value in dp[n][k] (i.e., dp[4][2]) is 2.

So, the minimum number of trials needed to find 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 and overall 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;
}
```

Dry Run of the Program

Let's break down how the program works with the input array {5, 6, 7, 4, 3, 6, 4}.

Input:

- arr[] = {5, 6, 7, 4, 3, 6, 4}
- n = 7 (the size of the array)

Initialization:

- currentSum = arr[0] = 5 (initialize current sum with the first element)
- overallSum = arr[0] = 5 (initialize overall sum with the first element)

Now, we iterate over the array starting from index 1.

Iteration:

1. **i = 1** (element = 6):
    - currentSum = 5, which is positive.
    - Add 6 to currentSum: currentSum = 5 + 6 = 11.
    - Since currentSum = 11 is greater than overallSum = 5, update overallSum = 11.
2. **i = 2** (element = 7):
    - currentSum = 11, which is positive.
    - Add 7 to currentSum: currentSum = 11 + 7 = 18.
    - Since currentSum = 18 is greater than overallSum = 11, update overallSum = 18.
3. **i = 3** (element = 4):
    - currentSum = 18, which is positive.
    - Add 4 to currentSum: currentSum = 18 + 4 = 22.
    - Since currentSum = 22 is greater than overallSum = 18, update overallSum = 22.
4. **i = 4** (element = 3):
    - currentSum = 22, which is positive.
    - Add 3 to currentSum: currentSum = 22 + 3 = 25.
    - Since currentSum = 25 is greater than overallSum = 22, update overallSum = 25.
5. **i = 5** (element = 6):
    - currentSum = 25, which is positive.
    - Add 6 to currentSum: currentSum = 25 + 6 = 31.
    - Since currentSum = 31 is greater than overallSum = 25, update overallSum = 31.
6. **i = 6** (element = 4):
    - currentSum = 31, which is positive.

	○ Add 4 to currentSum: currentSum = 31 + 4 = 35. ○ Since currentSum = 35 is greater than overallSum = 31, update overallSum = 35. **Final Result:** • The maximum sum of the subarray is 35.
Output:- 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;
}
```

Dry Run of the Program

Let's break down how the program works with the input grid:

Input:

- The input grid arr[MAX_ROWS][MAX_COLS] is:

Copy code

```
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
```

- The number of rows (rows) = 5
- The number of columns (cols) = 6

Initializations:

- dp[MAX_ROWS][MAX_COLS] is initialized to 0.
- largestSide = 0, which will keep track of the largest side of the square submatrix found.

The DP table (dp[i][j]) will store the size of the largest square submatrix whose bottom-right corner is at position (i, j).

Process:

We start iterating from the bottom-right corner of the matrix (i = rows - 1, j = cols - 1) and move upwards and to the left.

Iteration details:

1. **For i = 4, j = 5**:
    - arr[4][5] = 1
    - Since it's the last row or column (i == rows - 1 or j == cols - 1), dp[4][5] = arr[4][5] = 1.
    - largestSide = max(largestSide, dp[4][5]) = max(0, 1) = 1.
2. **For i = 4, j = 4**:
    - arr[4][4] = 1
    - Since it is the last row or column, dp[4][4] = arr[4][4] = 1.
    - largestSide = max(largestSide, dp[4][4]) = max(1, 1) = 1.
3. **For i = 4, j = 3**:
    - arr[4][3] = 1
    - Since it's the last row or column,

dp[4][3] = arr[4][3] = 1.
   - largestSide = max(largestSide, dp[4][3]) = max(1, 1) = 1.

4. **For i = 4, j = 2**:
   - arr[4][2] = 1
   - Since it's the last row or column, dp[4][2] = arr[4][2] = 1.
   - largestSide = max(largestSide, dp[4][2]) = max(1, 1) = 1.

5. **For i = 4, j = 1**:
   - arr[4][1] = 1
   - Since it's the last row or column, dp[4][1] = arr[4][1] = 1.
   - largestSide = max(largestSide, dp[4][1]) = max(1, 1) = 1.

6. **For i = 4, j = 0**:
   - arr[4][0] = 1
   - Since it's the last row or column, dp[4][0] = arr[4][0] = 1.
   - largestSide = max(largestSide, dp[4][0]) = max(1, 1) = 1.

7. **For i = 3, j = 5**:
   - arr[3][5] = 0
   - Since arr[3][5] == 0, dp[3][5] = 0.

8. **For i = 3, j = 4**:
   - arr[3][4] = 1
   - dp[3][4] = min(dp[3][5], min(dp[4][4], dp[4][5])) + 1 = min(0, min(1, 1)) + 1 = 1.
   - largestSide = max(largestSide, dp[3][4]) = max(1, 1) = 1.

9. **For i = 3, j = 3**:
   - arr[3][3] = 1
   - dp[3][3] = min(dp[3][4], min(dp[4][3], dp[4][4])) + 1 = min(1, min(1, 1)) + 1 = 2.
   - largestSide = max(largestSide, dp[3][3]) = max(1, 2) = 2.

10. **For i = 3, j = 2**:
    - arr[3][2] = 1
    - dp[3][2] = min(dp[3][3], min(dp[4][2], dp[4][3])) + 1 = min(2, min(1, 1)) + 1 = 2.
    - largestSide = max(largestSide, dp[3][2]) = max(2, 2) = 2.

11. **For i = 3, j = 1**:
    - arr[3][1] = 0
    - Since arr[3][1] == 0, dp[3][1] = 0.

12. **For i = 3, j = 0**:
    - arr[3][0] = 0

	○ Since arr[3][0] == 0, dp[3][0] = 0. 13. **For i = 2, j = 5**: ○ arr[2][5] = 0 ○ Since arr[2][5] == 0, dp[2][5] = 0. 14. **For i = 2, j = 4**: ○ arr[2][4] = 1 ○ dp[2][4] = min(dp[2][5], min(dp[3][4], dp[3][5])) + 1 = min(0, min(1, 0)) + 1 = 1. ○ largestSide = max(largestSide, dp[2][4]) = max(2, 1) = 2. 15. **For i = 2, j = 3**: ○ arr[2][3] = 1 ○ dp[2][3] = min(dp[2][4], min(dp[3][3], dp[3][4])) + 1 = min(1, min(2, 1)) + 1 = 2. ○ largestSide = max(largestSide, dp[2][3]) = max(2, 2) = 2. 16. **For i = 2, j = 2**: ○ arr[2][2] = 1 ○ dp[2][2] = min(dp[2][3], min(dp[3][2], dp[3][3])) + 1 = min(2, min(2, 2)) + 1 = 3. ○ largestSide = max(largestSide, dp[2][2]) = max(2, 3) = 3. 17. **For i = 2, j = 1**: ○ arr[2][1] = 1 ○ dp[2][1] = min(dp[2][2], min(dp[3][1], dp[3][2])) + 1 = min(3, min(0, 2)) + 1 = 1. ○ largestSide = max(largestSide, dp[2][1]) = max(3, 1) = 3. 18. **For i = 2, j = 0**: ○ arr[2][0] = 0 ○ Since arr[2][0] == 0, dp[2][0] = 0. **Result:** The largest square submatrix has side length 3
Output:- 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 = "abcd";
    string s2 = "abbd";

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

    return 0;
}
```

Dry Run:

Let's break down the execution of the code with the strings s1 = "abcd" and s2 = "abbd":

- **Step 1: Initializing the DP table**

  We initialize a DP table of size (5x5) (since s1.length() = 4 and s2.length() = 4, we add 1 for the zero-indexed table).

  int dp[5][5] = {0};

  Initial table:

  ```
  0 0 0 0 0
  0 0 0 0 0
  0 0 0 0 0
  0 0 0 0 0
  0 0 0 0 0
  ```

- **Step 2: Filling the DP table**

  We start filling the table from the bottom-right corner to the top-left (i.e., in reverse order of the strings).

  - Compare s1[3] = 'd' with s2[3] = 'd': They are equal, so dp[3][3] = 1 + dp[4][4] = 1 + 0 = 1.
  - Compare s1[3] = 'd' with s2[2] = 'b': They are different, so dp[3][2] = max(dp[4][2], dp[3][3]) = max(0, 1) = 1.
  - Compare s1[3] = 'd' with s2[1] = 'b': They are different, so dp[3][1] = max(dp[4][1], dp[3][2]) = max(0, 1) = 1.
  - Compare s1[3] = 'd' with s2[0] = 'a': They are different, so dp[3][0] = max(dp[4][0], dp[3][1]) = max(0, 1) = 1.

  Now, the table looks like:

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

  - Compare s1[2] = 'c' with s2[3] = 'd': They are different, so dp[2][3] = max(dp[3][3], dp[2][4]) = max(1, 0) = 1.
  - Compare s1[2] = 'c' with s2[2] = 'b': They are different, so dp[2][2] = max(dp[3][2], dp[2][3]) = max(1, 1) = 1.

- Compare s1[2] = 'c' with s2[1] = 'b': They are different, so dp[2][1] = max(dp[3][1], dp[2][2]) = max(1, 1) = 1.
- Compare s1[2] = 'c' with s2[0] = 'a': They are different, so dp[2][0] = max(dp[3][0], dp[2][1]) = max(1, 1) = 1.

Now, the table looks like:

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

- Compare s1[1] = 'b' with s2[3] = 'd': They are different, so dp[1][3] = max(dp[2][3], dp[1][4]) = max(1, 0) = 1.
- Compare s1[1] = 'b' with s2[2] = 'b': They are equal, so dp[1][2] = 1 + dp[2][3] = 1 + 1 = 2.
- Compare s1[1] = 'b' with s2[1] = 'b': They are equal, so dp[1][1] = 1 + dp[2][2] = 1 + 1 = 2.
- Compare s1[1] = 'b' with s2[0] = 'a': They are different, so dp[1][0] = max(dp[2][0], dp[1][1]) = max(1, 2) = 2.

Now, the table looks like:

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

- Compare s1[0] = 'a' with s2[3] = 'd': They are different, so dp[0][3] = max(dp[1][3], dp[0][4]) = max(1, 0) = 1.
- Compare s1[0] = 'a' with s2[2] = 'b': They are different, so dp[0][2] = max(dp[1][2], dp[0][3]) = max(2, 1) = 2.
- Compare s1[0] = 'a' with s2[1] = 'b': They are different, so dp[0][1] = max(dp[1][1], dp[0][2]) = max(2, 2) = 2.
- Compare s1[0] = 'a' with s2[0] = 'a': They are equal, so dp[0][0] = 1 + dp[1][1] = 1 + 2 = 3.

Final DP table:

3 2 2 1 0
2 2 2 1 0

	1 1 1 1 0 1 1 1 1 0 0 0 0 0 0 **Final Answer:** • The length of the Longest Common Subsequence (LCS) is dp[0][0] = 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;
}
```

Dry Run of the Program

Input:

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

Initializations:

- n = arr.size() = 10
- dp is initialized to {1, 1, 1, 1, 1, 1, 1, 1, 1, 1} because each element starts with a subsequence length of 1.
- omax = 1, which stores the overall maximum length of the LIS.

Steps:

We iterate through the array and compute the LIS length for each index:

Iteration 1 (i = 1):

- arr[1] = 22
- For j = 0: arr[1] > arr[0] (22 > 10), so we check dp[0] = 1.
  - max_len = max(0, dp[0]) = 1
- dp[1] = max_len + 1 = 1 + 1 = 2
- omax = max(omax, dp[1]) = max(1, 2) = 2

Iteration 2 (i = 2):

- arr[2] = 9
- For j = 0: arr[2] > arr[0] (9 > 10) is false.
- For j = 1: arr[2] > arr[1] (9 > 22) is false.
- No update in dp[2], it remains 1.
- omax = max(omax, dp[2]) = max(2, 1) = 2

Iteration 3 (i = 3):

- arr[3] = 33
- For j = 0: arr[3] > arr[0] (33 > 10), so we check dp[0] = 1.
  - max_len = max(0, dp[0]) = 1
- For j = 1: arr[3] > arr[1] (33 > 22), so we check dp[1] = 2.
  - max_len = max(1, dp[1]) = 2
- For j = 2: arr[3] > arr[2] (33 > 9), so we check dp[2] = 1.
  - max_len = max(2, dp[2]) = 2
- dp[3] = max_len + 1 = 2 + 1 = 3
- omax = max(omax, dp[3]) = max(2, 3) = 3

Iteration 4 (i = 4):

- arr[4] = 21
- For j = 0: arr[4] > arr[0] (21 > 10), so we check dp[0] = 1.

    - max_len = max(0, dp[0]) = 1
- For j = 1: arr[4] > arr[1] (21 > 22) is false.
- For j = 2: arr[4] > arr[2] (21 > 9), so we check dp[2] = 1.
    - max_len = max(1, dp[2]) = 1
- For j = 3: arr[4] > arr[3] (21 > 33) is false.
- dp[4] = max_len + 1 = 1 + 1 = 2
- omax = max(omax, dp[4]) = max(3, 2) = 3

Iteration 5 (i = 5):

- arr[5] = 50
- For j = 0: arr[5] > arr[0] (50 > 10), so we check dp[0] = 1.
    - max_len = max(0, dp[0]) = 1
- For j = 1: arr[5] > arr[1] (50 > 22), so we check dp[1] = 2.
    - max_len = max(1, dp[1]) = 2
- For j = 2: arr[5] > arr[2] (50 > 9), so we check dp[2] = 1.
    - max_len = max(2, dp[2]) = 2
- For j = 3: arr[5] > arr[3] (50 > 33), so we check dp[3] = 3.
    - max_len = max(2, dp[3]) = 3
- For j = 4: arr[5] > arr[4] (50 > 21), so we check dp[4] = 2.
    - max_len = max(3, dp[4]) = 3
- dp[5] = max_len + 1 = 3 + 1 = 4
- omax = max(omax, dp[5]) = max(3, 4) = 4

Iteration 6 (i = 6):

- arr[6] = 41
- For j = 0: arr[6] > arr[0] (41 > 10), so we check dp[0] = 1.
    - max_len = max(0, dp[0]) = 1
- For j = 1: arr[6] > arr[1] (41 > 22), so we check dp[1] = 2.
    - max_len = max(1, dp[1]) = 2
- For j = 2: arr[6] > arr[2] (41 > 9), so we check dp[2] = 1.
    - max_len = max(2, dp[2]) = 2
- For j = 3: arr[6] > arr[3] (41 > 33), so we check dp[3] = 3.
    - max_len = max(2, dp[3]) = 3
- For j = 4: arr[6] > arr[4] (41 > 21), so we check dp[4] = 2.
    - max_len = max(3, dp[4]) = 3
- For j = 5: arr[6] > arr[5] (41 > 50) is false.
- dp[6] = max_len + 1 = 3 + 1 = 4
- omax = max(omax, dp[6]) = max(4, 4) = 4

Iteration 7 (i = 7):

- arr[7] = 60
- For j = 0: arr[7] > arr[0] (60 > 10), so we check dp[0] = 1.
    - max_len = max(0, dp[0]) = 1
- For j = 1: arr[7] > arr[1] (60 > 22), so we check dp[1] = 2.

    - max_len = max(1, dp[1]) = 2
- For j = 2: arr[7] > arr[2] (60 > 9), so we check dp[2] = 1.
    - max_len = max(2, dp[2]) = 2
- For j = 3: arr[7] > arr[3] (60 > 33), so we check dp[3] = 3.
    - max_len = max(2, dp[3]) = 3
- For j = 4: arr[7] > arr[4] (60 > 21), so we check dp[4] = 2.
    - max_len = max(3, dp[4]) = 3
- For j = 5: arr[7] > arr[5] (60 > 50), so we check dp[5] = 4.
    - max_len = max(3, dp[5]) = 4
- For j = 6: arr[7] > arr[6] (60 > 41), so we check dp[6] = 4.
    - max_len = max(4, dp[6]) = 4
- dp[7] = max_len + 1 = 4 + 1 = 5
- omax = max(omax, dp[7]) = max(4, 5) = 5

Further iterations will follow the same process, updating dp[i] and omax as needed.

Finally, after processing all elements, the length of the longest increasing subsequence will be omax = 6.

Output:-

```
6
1 2 1 2 4 4 5 6 1
```

Longest Bitonic Subseq In C++

```cpp
#include <iostream>
#include <vector>
using namespace std;
int LongestBitonicSubseq(int arr[], int n) {
    vector<int> lis(n, 1); // lis[i] will store the length of LIS ending at index i
    vector<int> lds(n, 1); // lds[i] will store the length of LDS starting at index i

    // Computing LIS
    for (int i = 1; i < n; i++) {
        for (int j = 0; j < i; j++) {
            if (arr[j] <= arr[i]) {
                lis[i] = max(lis[i], lis[j] + 1);
            }
        }
    }

    // Computing LDS
    for (int i = n - 2; i >= 0; i--) {
        for (int j = n - 1; j > i; j--) {
            if (arr[j] <= 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 < 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 << LongestBitonicSubseq(arr, n) << endl;

    return 0;
}
```

Step 1: Compute lis (Longest Increasing Subsequence)

The lis logic in your code checks every previous index j for every current index i (j < i) and ensures:

```cpp
if (arr[j] <= arr[i]) {
    lis[i] = max(lis[i], lis[j] + 1);
}
```

This means:

- It allows increasing subsequences.
- It also includes elements that are **equal** (since arr[j] <= arr[i]).

Step 2: Compute lds (Longest Decreasing Subsequence)

The lds logic in your code checks every later index j for every current index i (j > i) and ensures:

```cpp
if (arr[j] <= arr[i]) {
    lds[i] = max(lds[i], lds[j] + 1);
}
```

This means:

- It allows decreasing subsequences.
- It also includes elements that are **equal** (since arr[j] <= arr[i]).

Step 3: Compute omax (Longest Bitonic Subsequence)

The total length of the Longest Bitonic Subsequence is computed as:

```cpp
omax = max(omax, lis[i] + lds[i] - 1);
```

This combines lis[i] and lds[i] for every index i, but subtracts 1 to avoid double-counting the pivot element.

Test Input

The array is:

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

Let’s compute lis, lds, and omax step-by-step **exactly as per your code**.

Step 1: Compute lis

Index (i)	Value (arr[i])	LIS (lis[i]) Calculation
0	10	lis[0] = 1 (initial)
1	22	lis[1] = 2 (10 → 22)
2	9	lis[2] = 1 (no increase)
3	33	lis[3] = 3 (10 → 22 → 33)
4	21	lis[4] = 2 (10 → 21)
5	50	lis[5] = 4 (10 → 22 → 33 → 50)
6	41	lis[6] = 4 (10 → 22 → 33 → 41)
7	60	lis[7] = 5 (10 → 22 → 33 → 50 → 60)
8	80	lis[8] = 6 (10 → 22 → 33 → 50 → 60 → 80)
9	3	lis[9] = 1 (no increase)

LIS Array: {1, 2, 1, 3, 2, 4, 4, 5, 6, 1}

Step 2: Compute lds

Index (i)	Value (arr[i])	LDS (lds[i]) Calculation
9	3	lds[9] = 1 (initial)
8	80	lds[8] = 2 (80 → 3)
7	60	lds[7] = 3 (60 → 3)
6	41	lds[6] = 4 (41 → 3)
5	50	lds[5] = 5 (50 → 41 → 3)
4	21	lds[4] = 2 (21 → 3)

Index (i)	Value (arr[i])	LDS (lds[i]) Calculation
3	33	lds[3] = 4 (33 → 21 → 3)
2	9	lds[2] = 2 (9 → 3)
1	22	lds[1] = 3 (22 → 9 → 3)
0	10	lds[0] = 3 (10 → 9 → 3)

LDS Array: {3, 3, 2, 4, 2, 5, 4, 3, 2, 1}

Step 3: Compute omax

LBS[i]=LIS[i]+LDS[i]−1LBS[i] = LIS[i] + LDS[i] - 1LBS[i]=LIS[i]+LDS[i]−1

Index (i)	LIS (lis[i])	LDS (lds[i])	LBS (lis[i] + lds[i] - 1)
0	1	3	3
1	2	3	4
2	1	2	2
3	3	4	6
4	2	2	3
5	4	5	8
6	4	4	7
7	5	3	7
8	6	2	7
9	1	1	1

Maximum LBS: 7

Correct Output: 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 = "xyzabcp";
   string s2 = "pqabcxy";

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

   return 0;
}
```

Input:

- s1 = "xyzabcp"
- s2 = "pqabcxy"

Initial Setup:

- m = s1.length() = 7
- n = s2.length() = 7
- dp is a (m+1) x (n+1) matrix initialized to 0. (i.e., dp[8][8])
- maxLen = 0

Table Format for dp:

The rows represent s1 (0 to m) and the columns represent s2 (0 to n).

Step 1: Initialize the dp Matrix

The dp matrix is initialized to all zeros:

```
dp = [
  [0, 0, 0, 0, 0, 0, 0, 0],
  [0, 0, 0, 0, 0, 0, 0, 0],
  [0, 0, 0, 0, 0, 0, 0, 0],
  [0, 0, 0, 0, 0, 0, 0, 0],
  [0, 0, 0, 0, 0, 0, 0, 0],
  [0, 0, 0, 0, 0, 0, 0, 0],
  [0, 0, 0, 0, 0, 0, 0, 0],
  [0, 0, 0, 0, 0, 0, 0, 0]
]
```

Step 2: Iterative Calculation

We iterate over i (1 to m) and j (1 to n), and compute dp[i][j] based on the characters s1[i-1] and s2[j-1].

Key Rule:

- If s1[i-1] == s2[j-1]: dp[i][j] = dp[i-1][j-1] + 1
- Otherwise: dp[i][j] = 0
- Update maxLen to track the largest value of dp[i][j].

Fill the Table:

i = 1, s1[0] = 'x':

- Compare 'x' with each character of s2:

  dp[1][1] = 0 ('x' != 'p')
  dp[1][2] = 0 ('x' != 'q')
  dp[1][3] = 0 ('x' != 'a')
  dp[1][4] = 0 ('x' != 'b')
  dp[1][5] = 0 ('x' != 'c')
  dp[1][6] = 1 ('x' == 'x')
  dp[1][7] = 0 ('x' != 'y')

  Updated dp:

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

i = 2, s1[1] = 'y':

- Compare 'y' with each character of s2:

  dp[2][1] = 0 ('y' != 'p')
  dp[2][2] = 0 ('y' != 'q')
  dp[2][3] = 0 ('y' != 'a')
  dp[2][4] = 0 ('y' != 'b')
  dp[2][5] = 0 ('y' != 'c')
  dp[2][6] = 0 ('y' != 'x')
  dp[2][7] = 2 ('y' == 'y', dp[1][6] + 1)

  Updated dp:

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

i = 3, s1[2] = 'z':

- Compare 'z' with each character of s2:

  dp[3][1] = 0 ('z' != 'p')
  dp[3][2] = 0 ('z' != 'q')
  dp[3][3] = 0 ('z' != 'a')
  dp[3][4] = 0 ('z' != 'b')
  dp[3][5] = 0 ('z' != 'c')
  dp[3][6] = 0 ('z' != 'x')
  dp[3][7] = 0 ('z' != 'y')

  Updated dp:

dp[3] = [0, 0, 0, 0, 0, 0, 0, 0]

i = 4, s1[3] = 'a':

- Compare 'a' with each character of s2:

dp[4][1] = 0   ('a' != 'p')
dp[4][2] = 0   ('a' != 'q')
dp[4][3] = 1   ('a' == 'a', dp[3][2] + 1)
dp[4][4] = 0   ('a' != 'b')
dp[4][5] = 0   ('a' != 'c')
dp[4][6] = 0   ('a' != 'x')
dp[4][7] = 0   ('a' != 'y')

Updated dp:

dp[4] = [0, 0, 1, 0, 0, 0, 0, 0]

i = 5, s1[4] = 'b':

- Compare 'b' with each character of s2:

less
Copy code
dp[5][1] = 0   ('b' != 'p')
dp[5][2] = 0   ('b' != 'q')
dp[5][3] = 0   ('b' != 'a')
dp[5][4] = 2   ('b' == 'b', dp[4][3] + 1)
dp[5][5] = 0   ('b' != 'c')
dp[5][6] = 0   ('b' != 'x')
dp[5][7] = 0   ('b' != 'y')

Updated dp:

dp[5] = [0, 0, 0, 2, 0, 0, 0, 0]

i = 6, s1[5] = 'c':

- Compare 'c' with each character of s2:

dp[6][1] = 0   ('c' != 'p')
dp[6][2] = 0   ('c' != 'q')
dp[6][3] = 0   ('c' != 'a')
dp[6][4] = 0   ('c' != 'b')
dp[6][5] = 3   ('c' == 'c', dp[5][4] + 1)
dp[6][6] = 0   ('c' != 'x')
dp[6][7] = 0   ('c' != 'y')

Updated dp:

dp[6] = [0, 0, 0, 0, 3, 0, 0, 0]

i = 7, s1[6] = 'p':

- Compare 'p' with each character of s2:

  dp[7][1] = 1 ('p' == 'p', dp[6][0] + 1)
  dp[7][2] = 0 ('p' != 'q')
  dp[7][3] = 0 ('p' != 'a')
  dp[7][4] = 0 ('p' != 'b')
  dp[7][5] = 0 ('p' != 'c')
  dp[7][6] = 0 ('p' != 'x')
  dp[7][7] = 0 ('p' != 'y')

  Updated dp:

  dp[7] = [1, 0, 0, 0, 0, 0, 0, 0]

Final Result:

- maxLen = 3, which corresponds to the substring "abc".

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;
}
```

Input:

str = "abccba", n = 6

Initialization:

1. Create a **DP table** (dp[i][j]) of size 6×66 \times 66×6 initialized to 0.
2. **Base case**: Fill diagonal elements (g = 0) because every single character is a palindrome of length 1.

Initial DP Table (after g = 0):

i\j	a	b	c	c	b	a
a	1	0	0	0	0	0
b	0	1	0	0	0	0
c	0	0	1	0	0	0
c	0	0	0	1	0	0
b	0	0	0	0	1	0
a	0	0	0	0	0	1

Iterate Over Gaps:

Gap = 1 (g = 1):

- Compare adjacent characters:
    - If str[i] == str[j], then dp[i][j] = 2.
    - Else, dp[i][j] = 1.

i\j	a	b	c	c	b	a
a	1	1	0	0	0	0
b	0	1	1	0	0	0
c	0	0	1	2	0	0
c	0	0	0	1	1	0
b	0	0	0	0	1	1
a	0	0	0	0	0	1

Gap = 2 (g = 2):

- For substrings of length 3 (str[i...j]):
    - If str[i] == str[j]: dp[i][j]=2+dp[i+1][j−1]dp[i][j] = 2 + dp[i+1][j-1]dp[i][j]=2+dp[i+1][j−1]

  - Else: dp[i][j]=max(dp[i][j−1],dp[i+1][j])dp[i][j] = \max(dp[i][j-1], dp[i+1][j])dp[i][j]=max(dp[i][j−1],dp[i+1][j])

i\j	**a**	**b**	**c**	**c**	**b**	**a**
a	1	1	1	0	0	0
b	0	1	1	2	0	0
c	0	0	1	2	2	0
c	0	0	0	1	1	2
b	0	0	0	0	1	1
a	0	0	0	0	0	1

Gap = 3 (g = 3):

- For substrings of length 4:
  - Use the same recurrence relation.

i\j	**a**	**b**	**c**	**c**	**b**	**a**
a	1	1	1	2	0	0
b	0	1	1	2	4	0
c	0	0	1	2	2	0
c	0	0	0	1	1	2
b	0	0	0	0	1	1
a	0	0	0	0	0	1

Gap = 4 (g = 4):

i\j	**a**	**b**	**c**	**c**	**b**	**a**
a	1	1	1	2	4	0
b	0	1	1	2	4	4
c	0	0	1	2	2	0
c	0	0	0	1	1	2
b	0	0	0	0	1	1
a	0	0	0	0	0	1

Gap = 5 (g = 5):

i\j	**a**	**b**	**c**	**c**	**b**	**a**
a	1	1	1	2	4	6
b	0	1	1	2	4	4
c	0	0	1	2	2	0
c	0	0	0	1	1	2
b	0	0	0	0	1	1
a	0	0	0	0	0	1

<table>
<tr><td></td><td>Final Result:
• dp[0][n-1] = dp[0][5] = 6.</td></tr>
<tr><td colspan="2">Output:-
6</td></tr>
</table>

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;
}
```

Input:

str = "abccbc", n=6n = 6n=6

Initial Setup:

1. **dp table**:
   A boolean n×nn \times nn×n table is used to check if str[i...j] is a palindrome.
2. **Initialization**:
   - Single-character substrings (dp[i][i]) are palindromes, so initialize dp[i][i] = true for all i.
   - len = 0 (to store the length of the longest palindromic substring).

Step 1: Check for substrings of length 2.

- For each pair of adjacent characters (i,i+1)(i, i+1)(i,i+1):
  - If str[i] == str[i+1], set dp[i][i+1] = true and update len = 2.
  - Otherwise, set dp[i][i+1] = false.

DP Table After Length 2 Check:

i\j	**a**	**b**	**c**	**c**	**b**	**c**
a	T	F	-	-	-	-
b	-	T	F	-	-	-
c	-	-	T	T	-	-
c	-	-	-	T	F	-
b	-	-	-	-	T	F
c	-	-	-	-	-	T

len = 2 (because cc is a palindrome of length 2).

Step 2: Check for substrings of length > 2.

We now iterate for substrings of increasing length (g = 2, 3, ..., n-1).

Gap = 2 (g = 2):

For substrings of length 3, check if:

str[i] == str[j] and dp[i+1][j-1]\text{str[i] == str[j] \text{ and } dp[i+1][j-1]}
str[i] == str[j] and dp[i+1][j-1]

Substrings	**Result**	**DP Update**	**Reason**
"abc"	False	dp[0][2] = false	a != c
"bcc"	False	dp[1][3] = false	b != c
"ccb"	False	dp[2][4] = false	c != b
"cbc"	True	dp[3][5] = true	c == c and dp[4][4] == true

DP Table After Gap 2:

i\j	**a**	**b**	**c**	**c**	**b**	**c**
a	T	F	F	-	-	-
b	-	T	F	F	-	-
c	-	-	T	T	F	-
c	-	-	-	T	F	T
b	-	-	-	-	T	F
c	-	-	-	-	-	T

len = 3 (because cbc is a palindrome of length 3).

Gap = 3 (g = 3):

For substrings of length 4, check if:

str[i] == str[j] and dp[i+1][j-1]\text{str[i] == str[j] \text{ and } dp[i+1][j-1]}
str[i] == str[j] and dp[i+1][j-1]

Substrings	**Result**	**DP Update**	**Reason**
"abcc"	False	dp[0][3] = false	a != c
"bccb"	True	dp[1][4] = true	b == b and dp[2][3] == true
"ccbc"	False	dp[2][5] = false	c != c

DP Table After Gap 3:

i\j	**a**	**b**	**c**	**c**	**b**	**c**
a	T	F	F	F	-	-
b	-	T	F	F	T	-
c	-	-	T	T	F	-
c	-	-	-	T	F	T
b	-	-	-	-	T	F
c	-	-	-	-	-	T

len = 4 (because bccb is a palindrome of length 4).

Gap = 4 (g = 4):

For substrings of length 5, check if:

str[i] == str[j] and dp[i+1][j-1]\text{str[i] == str[j] \text{ and } dp[i+1][j-1]}
str[i] == str[j] and dp[i+1][j-1]

Substrings	**Result**	**DP Update**	**Reason**
"abccb"	False	dp[0][4] = false	a != b
"bccbc"	False	dp[1][5] = false	b != c

DP Table After Gap 4:

No new updates, and **len = 4** remains unchanged.

Gap = 5 (g = 5):

For substrings of length 6, check:

str[i] == str[j] and dp[i+1][j-1]\text{str[i] == str[j] \text{ and } dp[i+1][j-1]}
str[i] == str[j] and dp[i+1][j-1]

Substrings	**Result**	**DP Update**	**Reason**
"abccbc"	False	dp[0][5] = false	a != c

Final Result:

- Longest palindromic substring length = **4** (bccb).

Output:-

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;
}
```

Step-by-Step Dry Run

Initialization of dp[]:

Initially, dp[] is:

dp[] = {10, 22, 9, 33, 21, 50, 41, 60, 80, 3}

Each element is initialized to the value of the corresponding element in arr[].

For i = 0 (First Element: 10)

- **maxSum = 10**
- There are no previous elements, so no update is made in dp[].
- **dp[0] = 10**
- **omax = max(INT_MIN, 10) = 10**

For i = 1 (Element: 22)

- **maxSum = 22**
- Check all previous elements (arr[0] = 10):
    - **arr[0] <= arr[1] (10 <= 22)**: Yes
        - **maxSum = max(22, dp[0] + arr[1]) = max(22, 10 + 22) = 32**
- **dp[1] = 32**
- **omax = max(10, 32) = 32**

For i = 2 (Element: 9)

- **maxSum = 9**
- Check all previous elements (arr[0] = 10, arr[1] = 22):
    - **arr[0] <= arr[2] (10 <= 9)**: No
    - **arr[1] <= arr[2] (22 <= 9)**: No
- **dp[2] = 9**
- **omax = max(32, 9) = 32**

For i = 3 (Element: 33)

- **maxSum = 33**
- Check all previous elements (arr[0] = 10, arr[1] = 22, arr[2] = 9):
    - **arr[0] <= arr[3] (10 <= 33)**: Yes
        - **maxSum = max(33, dp[0] + arr[3]) = max(33, 10 + 33) = 43**
    - **arr[1] <= arr[3] (22 <= 33)**: Yes
        - **maxSum = max(43, dp[1] + arr[3]) = max(43, 32 + 33) = 65**
    - **arr[2] <= arr[3] (9 <= 33)**: Yes
        - **maxSum = max(65, dp[2] + arr[3]) = max(65, 9 + 33) = 65**
- **dp[3] = 65**
- **omax = max(32, 65) = 65**

For i = 4 (Element: 21)

- **maxSum = 21**
- Check all previous elements (arr[0] = 10, arr[1] = 22, arr[2] = 9, arr[3] = 33):
    - **arr[0] <= arr[4] (10 <= 21)**: Yes
        - **maxSum = max(21, dp[0] + arr[4]) = max(21, 10 + 21) = 31**
    - **arr[1] <= arr[4] (22 <= 21)**: No
    - **arr[2] <= arr[4] (9 <= 21)**: Yes
        - **maxSum = max(31, dp[2] + arr[4]) = max(31, 9 + 21) = 31**
    - **arr[3] <= arr[4] (33 <= 21)**: No
- **dp[4] = 31**
- **omax = max(65, 31) = 65**

For i = 5 (Element: 50)

- **maxSum = 50**
- Check all previous elements:
    - **arr[0] <= arr[5] (10 <= 50)**: Yes
        - **maxSum = max(50, dp[0] + arr[5]) = max(50, 10 + 50) = 60**

  - **arr[1] <= arr[5] (22 <= 50)**: Yes
    - **maxSum = max(60, dp[1] + arr[5]) = max(60, 32 + 50) = 82**
  - **arr[2] <= arr[5] (9 <= 50)**: Yes
    - **maxSum = max(82, dp[2] + arr[5]) = max(82, 9 + 50) = 82**
  - **arr[3] <= arr[5] (33 <= 50)**: Yes
    - **maxSum = max(82, dp[3] + arr[5]) = max(82, 65 + 50) = 115**
  - **arr[4] <= arr[5] (21 <= 50)**: Yes
    - **maxSum = max(115, dp[4] + arr[5]) = max(115, 31 + 50) = 115**
- **dp[5] = 115**
- **omax = max(65, 115) = 115**

For i = 6 (Element: 41)

- **maxSum = 41**
- Check all previous elements (arr[0] = 10, arr[1] = 22, arr[2] = 9, arr[3] = 33, arr[4] = 21, arr[5] = 50):
  - **arr[0] <= arr[6] (10 <= 41)**: Yes
    - **maxSum = max(41, dp[0] + arr[6]) = max(41, 10 + 41) = 51**
  - **arr[1] <= arr[6] (22 <= 41)**: Yes
    - **maxSum = max(51, dp[1] + arr[6]) = max(51, 32 + 41) = 73**
  - **arr[2] <= arr[6] (9 <= 41)**: Yes
    - **maxSum = max(73, dp[2] + arr[6]) = max(73, 9 + 41) = 73**
  - **arr[3] <= arr[6] (33 <= 41)**: Yes
    - **maxSum = max(73, dp[3] + arr[6]) = max(73, 65 + 41) = 106**
  - **arr[4] <= arr[6] (21 <= 41)**: Yes
    - **maxSum = max(106, dp[4] + arr[6]) = max(106, 31 + 41) = 106**
  - **arr[5] <= arr[6] (50 <= 41)**: No
- **dp[6] = 106**
- **omax = max(115, 106) = 115**

	Final Output: After performing similar checks for all the remaining elements, the maximum sum found will be **255**. Thus, the **Maximum Sum Increasing Subsequence** is **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;
}
```

Initial Setup:

We have:

- s1 = "cat"
- s2 = "cut"
- c1 = 1 (cost to remove a character from s1)
- c2 = 1 (cost to remove a character from s2)

The goal is to find the **Longest Common Subsequence (LCS)** and then calculate the minimum cost of making the two strings identical by removing characters from them.

Step 1: Initialize DP Table

We initialize the DP table with dimensions (m+1) x (n+1), where m is the length of s1 and n is the length of s2.

- m = 3 (length of s1)
- n = 3 (length of s2)

So, the DP table will be a 4x4 matrix (since we include the 0th index for base cases).

DP Table (Initial):
[0, 0, 0, 0]
[0, 0, 0, 0]
[0, 0, 0, 0]
[0, 0, 0, 0]

Step 2: Fill DP Table to Calculate LCS Length

We fill the DP table using the following logic:

- If s1[i] == s2[j], then dp[i][j] = dp[i + 1][j + 1] + 1 (this means the current characters match, and we can extend the LCS).
- If s1[i] != s2[j], then dp[i][j] = max(dp[i + 1][j], dp[i][j + 1]) (this means we have to choose the maximum LCS length from skipping one character from either string).

Now, let's fill the DP table step by step.

1. **For i = 2 and j = 2**:
    - s1[2] = "t" and s2[2] = "t", they match, so dp[2][2] = dp[3][3] + 1 = 0 + 1 = 1.

    DP Table after filling dp[2][2]:
    [0, 0, 0, 0]
    [0, 0, 0, 0]
    [0, 0, 1, 0]
    [0, 0, 0, 0]

2. **For i = 2 and j = 1**:

- s1[2] = "t" and s2[1] = "u", they do not match, so dp[2][1] = max(dp[3][1], dp[2][2]) = max(0, 1) = 1.

DP Table after filling dp[2][1]:
[0, 0, 0, 0]
[0, 0, 0, 0]
[0, 1, 1, 0]
[0, 0, 0, 0]

3. **For i = 2 and j = 0**:
    - s1[2] = "t" and s2[0] = "c", they do not match, so dp[2][0] = max(dp[3][0], dp[2][1]) = max(0, 1) = 1.

   DP Table after filling dp[2][0]:
   [0, 0, 0, 0]
   [0, 0, 0, 0]
   [1, 1, 1, 0]
   [0, 0, 0, 0]

4. **For i = 1 and j = 2**:
    - s1[1] = "a" and s2[2] = "t", they do not match, so dp[1][2] = max(dp[2][2], dp[1][3]) = max(1, 0) = 1.

   DP Table after filling dp[1][2]:
   [0, 0, 0, 0]
   [0, 0, 1, 0]
   [1, 1, 1, 0]
   [0, 0, 0, 0]

5. **For i = 1 and j = 1**:
    - s1[1] = "a" and s2[1] = "u", they do not match, so dp[1][1] = max(dp[2][1], dp[1][2]) = max(1, 1) = 1.

   DP Table after filling dp[1][1]:
   [0, 0, 0, 0]
   [0, 1, 1, 0]
   [1, 1, 1, 0]
   [0, 0, 0, 0]

6. **For i = 1 and j = 0**:
    - s1[1] = "a" and s2[0] = "c", they do not match, so dp[1][0] = max(dp[2][0], dp[1][1]) = max(1, 1) = 1.

   DP Table after filling dp[1][0]:
   [0, 0, 0, 0]
   [1, 1, 1, 0]
   [1, 1, 1, 0]
   [0, 0, 0, 0]

7. **For i = 0 and j = 2**:
    - s1[0] = "c" and s2[2] = "t", they do not match, so dp[0][2] = max(dp[1][2], dp[0][3]) = max(1, 0) = 1.

   DP Table after filling dp[0][2]:

[0, 0, 1, 0]
[1, 1, 1, 0]
[1, 1, 1, 0]
[0, 0, 0, 0]

8. **For i = 0 and j = 1**:
    - s1[0] = "c" and s2[1] = "u", they do not match, so dp[0][1] = max(dp[1][1], dp[0][2]) = max(1, 1) = 1.

DP Table after filling dp[0][1]:
[0, 1, 1, 0]
[1, 1, 1, 0]
[1, 1, 1, 0]
[0, 0, 0, 0]

9. **For i = 0 and j = 0**:
    - s1[0] = "c" and s2[0] = "c", they match, so dp[0][0] = dp[1][1] + 1 = 1 + 1 = 2.

DP Table after filling dp[0][0]:
[2, 1, 1, 0]
[1, 1, 1, 0]
[1, 1, 1, 0]
[0, 0, 0, 0]

Step 3: Calculate LCS Length

After filling the DP table, the length of the **Longest Common Subsequence (LCS)** is found at dp[0][0], which is 2. This means the LCS of "cat" and "cut" is of length 2.

Step 4: Calculate the Cost

Now, we calculate the remaining characters in s1 and s2:

- Remaining characters in s1: 3 - 2 = 1 (the character "a" needs to be removed).
- Remaining characters in s2: 3 - 2 = 1 (the character "u" needs to be removed).

The total cost is:

$$cost=1\times c1+1\times c2=1\times 1+1\times 1=2$$

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;
}
```

Dry Run with the Input arr[] = {20, 30, 2, 10}

We need to compute dp[0][n-1], which gives the result for the entire array.

Initialization:

- **dp table (initial values):**

```
dp[][] = {
    {0, 0, 0, 0},
    {0, 0, 0, 0},
    {0, 0, 0, 0},
    {0, 0, 0, 0}
}
```

Step-by-Step Iteration:

1. **g = 0** (Subarrays of size 1):
    - dp[0][0] = arr[0] = 20
    - dp[1][1] = arr[1] = 30
    - dp[2][2] = arr[2] = 2
    - dp[3][3] = arr[3] = 10
2. **g = 1** (Subarrays of size 2):
    - dp[0][1] = max(arr[0], arr[1]) = max(20, 30) = 30
    - dp[1][2] = max(arr[1], arr[2]) = max(30, 2) = 30
    - dp[2][3] = max(arr[2], arr[3]) = max(2, 10) = 10
3. **g = 2** (Subarrays of size 3):
    - For dp[0][2]: We compute two options:
        - val1 = arr[0] + min(dp[2][2], dp[1][1]) = 20 + min(2, 30) = 22
        - val2 = arr[2] + min(dp[1][1], dp[0][0]) = 2 + min(30, 20) = 22
        - dp[0][2] = max(22, 22) = 22
    - For dp[1][3]: We compute two options:
        - val1 = arr[1] + min(dp[3][3], dp[2][2]) = 30 + min(10, 2) = 32
        - val2 = arr[3] + min(dp[2][2], dp[1][1]) = 10 + min(2, 30) = 12
        - dp[1][3] = max(32, 12) = 32
4. **g = 3** (Subarrays of size 4):
    - For dp[0][3]: We compute two

options:

  - val1 = arr[0] + min(dp[2][3], dp[1][2]) = 20 + min(10, 30) = 30
  - val2 = arr[3] + min(dp[1][2], dp[0][1]) = 10 + min(30, 30) = 40
  - dp[0][3] = max(30, 40) = 40

Final DP Table:

After all iterations, the final dp table looks like this:

```
dp[][] = {
   {20, 30, 22, 40},
   {0, 30, 30, 32},
   {0, 0, 2, 10},
   {0, 0, 0, 10}
}
```

Result:

The final value dp[0][3] = 40 is the maximum sum that can be collected by the first player in this game.

Output:-

```
40
```

Paths of 0-1 knapsack In C++

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

struct Pair {
    int i;
    int j;
    string psf;

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

void printPaths(vector<vector<int>>& dp,
vector<int>& vals, vector<int>& wts, int i,
int j, string psf, deque<Pair>& que) {
    while (!que.empty()) {
        Pair rem = que.front();
        que.pop_front();

        if (rem.i == 0 | | rem.j == 0) {
            cout << rem.psf << endl;
        } else {
            int exc = dp[rem.i - 1][rem.j];

            if (rem.j >= wts[rem.i - 1]) {
                int inc = dp[rem.i - 1][rem.j -
wts[rem.i - 1]] + vals[rem.i - 1];

                if (dp[rem.i][rem.j] == inc) {
                    que.push_back(Pair(rem.i - 1,
rem.j - wts[rem.i - 1], to_string(rem.i - 1) +
" " + rem.psf));
                }
            }

            if (dp[rem.i][rem.j] == exc) {
                que.push_back(Pair(rem.i - 1,
rem.j, rem.psf));
            }
        }
    }
}

void knapsackPaths(vector<int>& vals,
vector<int>& wts, int cap) {
    int n = vals.size();
    vector<vector<int>> dp(n + 1,
```

Knapsack Problem Explanation:

- **Items**: There are 5 items with associated values and weights:
    - Item 1: Value = 15, Weight = 2
    - Item 2: Value = 14, Weight = 5
    - Item 3: Value = 10, Weight = 1
    - Item 4: Value = 45, Weight = 3
    - Item 5: Value = 30, Weight = 4
- **Knapsack Capacity**: The knapsack has a capacity of 7 units.

Dynamic Programming Table (dp):

The dynamic programming table is built to calculate the maximum value that can be achieved for each capacity using the first i items. The size of the table is (n+1) x (cap+1) where n is the number of items and cap is the knapsack capacity.

DP table construction:

The DP table dp[i][j] represents the maximum value achievable with the first i items and a knapsack capacity j. The recurrence relation is as follows:

1. If item i is not included: dp[i][j] = dp[i-1][j]
2. If item i is included (i.e., if the weight of the item is less than or equal to the remaining capacity): dp[i][j] = max(dp[i][j], dp[i-1][j-wts[i-1]] + vals[i-1])

Here's how the DP table looks after filling:

Items/ Capacity	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1 (15, 2)	0	0	15	15	15	15	15	15
2 (14, 5)	0	0	15	15	15	14	15	15
3 (10, 1)	0	10	15	15	25	25	25	25
4 (45, 3)	0	10	15	45	45	45	55	55
5 (30, 4)	0	10	15	45	45	45	55	75

- The maximum value achievable with a capacity of 7 is 75, which occurs by including the items 3 and 4 (corresponding to the values 10 and 45, respectively).

Path Finding:

```
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;
}
```

Once the DP table is filled, the program uses a breadth-first search (BFS) approach to backtrack and find all the possible paths that lead to the maximum value. The paths are stored in a deque, and for each item, the program checks whether the current value is achieved by including or excluding the item.

The function printPaths recursively finds all paths that lead to the maximum value and prints them. The path output is based on the indices of the items included in the optimal knapsack configuration.

Output:

- The **maximum value** for the knapsack is 75.
- The path "3 4" indicates that items 3 and 4 were included in the optimal solution, which leads to the maximum value.

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;
}
```

Explanation of the Code:

- **Input Array**: The array you provided is {0, 1, 2, 3, 1, 2, 3, 4, 2, 1, 2, 3}. However, the actual input to the problem is simply the number n, where we want to find the minimum number of perfect squares for all numbers from 0 to n.
- **DP Array (dp)**: The dp array stores the minimum number of perfect squares that sum up to each index value. The array is initialized to INT_MAX to signify that no solution has been found yet, and it is updated with the minimum value as we iterate.
- **Base Cases**:
  - dp[0] = 0: 0 requires no squares.
  - dp[1] = 1: 1 can be represented as a square of 1 (1^2).
- **Recursive Case**:
  - For each value i from 2 to n, the code checks all possible perfect squares j*j that can be subtracted from i. It calculates the minimum value of dp[i] by comparing it with dp[i - j*j] + 1, where +1 accounts for using the square j*j.
- **Output**: The program prints the values in the dp array from index 0 to n.

Example Walkthrough:

The goal is to find the minimum number of perfect squares that sum up to each number from 0 to the length of the array.

DP Table Calculation:

1. **dp[0]** = 0 (Base case: 0 requires 0 squares)
2. **dp[1]** = 1 (Base case: 1 can be written as 1^2)
3. **dp[2]** = 2 (2 can be written as 1^2 + 1^2)
4. **dp[3]** = 3 (3 can be written as 1^2 + 1^2 + 1^2)
5. **dp[4]** = 1 (4 can be written as 2^2)
6. **dp[5]** = 2 (5 can be written as 4 + 1^2)
7. **dp[6]** = 3 (6 can be written as 4 + 1^2 + 1^2)
8. **dp[7]** = 4 (7 can be written as 4 + 1^2 + 1^2 + 1^2)
9. **dp[8]** = 2 (8 can be written as 4 + 4)
10. **dp[9]** = 1 (9 can be written as 3^2)
11. **dp[10]** = 2 (10 can be written as 9 + 1^2)
12. **dp[11]** = 3 (11 can be written as 9 + 1^2 + 1^2)
13. **dp[12]** = 3 (12 can be written as 9 + 1^2 + 1^2

<table>
<tr><td></td><td>+ 1^2)

Output:

After the dp array is computed, the output will be:

0 1 2 3 1 2 3 4 2 1 2 3 3</td></tr>
<tr><td colspan="2">Output:-
0 1 2 3 1 2 3 4 2 1 2 3 3</td></tr>
</table>

Print all LIS In C++

```cpp
#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 with Input Array {10, 22, 9, 33, 21, 50, 41, 60, 80, 3}

1. **Step 1: Calculate the LIS Lengths** (dp array):
    - We start with dp[i] = 1 for all i (since a single element is trivially a subsequence of length 1).
    - Iterating through each i and for each i, checking all previous j to update dp[i]:

    For each index:

    - **Index 0 (value 10)**: dp[0] = 1 (no previous elements).
    - **Index 1 (value 22)**: dp[1] = max(dp[0] + 1) = 2.
    - **Index 2 (value 9)**: dp[2] = 1 (no elements before it are smaller).
    - **Index 3 (value 33)**: dp[3] = max(dp[0] + 1, dp[1] + 1) = 3.
    - **Index 4 (value 21)**: dp[4] = max(dp[0] + 1) = 2.
    - **Index 5 (value 50)**: dp[5] = max(dp[0] + 1, dp[1] + 1, dp[3] + 1) = 4.
    - **Index 6 (value 41)**: dp[6] = max(dp[0] + 1, dp[1] + 1, dp[3] + 1) = 4.
    - **Index 7 (value 60)**: dp[7] = max(dp[0] + 1, dp[1] + 1, dp[3] + 1, dp[5] + 1) = 5.
    - **Index 8 (value 80)**: dp[8] = max(dp[0] + 1, dp[1] + 1, dp[3] + 1, dp[5] + 1, dp[7] + 1) = 6.
    - **Index 9 (value 3)**: dp[9] = 1.

    The dp array will look like this after processing:

    dp = {1, 2, 1, 3, 2, 4, 4, 5, 6, 1}

2. **Step 2: Find the Maximum LIS Length**:
    - The maximum LIS length omax = 6 and the index where it ends omi = 8 (corresponding to value 80).
3. **Step 3: Backtrack to Find All LIS**:
    - A deque q is initialized with the Pair containing the maximum LIS.
    - The initial Pair object in the deque:

      q = {Pair(6, 8, 80, "80")}

```
        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;
}
```

- Now, backtrack and find all possible subsequences:
  - For Pair(6, 8, 80, "80"), we look for elements before index 8 that can form a LIS of length 5. We find:
    - dp[7] == 5 and arr[7] = 60 <= 80, so we push Pair(5, 7, 60, "60 -> 80").
    - Similarly, we continue for other indices, building the subsequences.

After backtracking, we find two possible LIS:

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

4. **Step 4: Output the Results**: The output is:

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

Output:-

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

Print all path with max gold In C++

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

struct Pair {
    int i, j;
    string psf;

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

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

    // dp array to store maximum gold collected to reach each cell
    vector<vector<int>> dp(m, vector<int>(n, 0));

    // Initialize dp array for the last column
    for (int i = 0; i < m; i++) {
        dp[i][n - 1] = arr[i][n - 1];
    }

    // Fill dp array using dynamic programming approach
    for (int j = n - 2; j >= 0; j--) {
        for (int i = 0; i < m; i++) {
            int maxGold = dp[i][j + 1]; // Maximum gold by going right from current cell
            if (i > 0) {
                maxGold = max(maxGold, dp[i - 1][j + 1]); // Maximum gold by going diagonal-up-right
            }
            if (i < 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
        }
```

Step-by-Step Execution:

Input:

```
arr = [
    [3, 2, 3, 1],
    [2, 4, 6, 0],
    [5, 0, 1, 3],
    [9, 1, 5, 1]
]
```

Step 1: Initialize dp Array

We create a dp matrix of the same dimensions as arr and initialize the last column with the values of arr's last column:

```
dp = [
    [0, 0, 0, 1],
    [0, 0, 0, 0],
    [0, 0, 0, 3],
    [0, 0, 0, 1]
]
```

Step 2: Fill dp Array (Dynamic Programming)

We calculate the maximum gold collectible for each cell in reverse column order (from right to left):

- **Column 2 (j = 2):**
    - Row 0: dp[0][2] = arr[0][2] + max(dp[0][3], dp[1][3]) = 3 + max(1, 0) = 4
    - Row 1: dp[1][2] = arr[1][2] + max(dp[1][3], dp[0][3], dp[2][3]) = 6 + max(0, 1, 3) = 9
    - Row 2: dp[2][2] = arr[2][2] + max(dp[2][3], dp[1][3], dp[3][3]) = 1 + max(3, 0, 1) = 4
    - Row 3: dp[3][2] = arr[3][2] + max(dp[3][3], dp[2][3]) = 5 + max(1, 3) = 8
    - Updated dp:

      ```
      dp = [
          [0, 0, 4, 1],
          [0, 0, 9, 0],
          [0, 0, 4, 3],
          [0, 0, 8, 1]
      ]
      ```
- **Column 1 (j = 1):**

```
    }

    // 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
            }
```

  - Row 0: dp[0][1] = arr[0][1] + max(dp[0][2], dp[1][2]) = 2 + max(4, 9) = 11
  - Row 1: dp[1][1] = arr[1][1] + max(dp[1][2], dp[0][2], dp[2][2]) = 4 + max(9, 4, 4) = 13
  - Row 2: dp[2][1] = arr[2][1] + max(dp[2][2], dp[1][2], dp[3][2]) = 0 + max(4, 9, 8) = 9
  - Row 3: dp[3][1] = arr[3][1] + max(dp[3][2], dp[2][2]) = 1 + max(8, 4) = 9
  - Updated dp:

```
dp = [
   [0, 11, 4, 1],
   [0, 13, 9, 0],
   [0,  9, 4, 3],
   [0,  9, 8, 1]
]
```

- **Column 0 (j = 0):**
  - Row 0: dp[0][0] = arr[0][0] + max(dp[0][1], dp[1][1]) = 3 + max(11, 13) = 16
  - Row 1: dp[1][0] = arr[1][0] + max(dp[1][1], dp[0][1], dp[2][1]) = 2 + max(13, 11, 9) = 15
  - Row 2: dp[2][0] = arr[2][0] + max(dp[2][1], dp[1][1], dp[3][1]) = 5 + max(9, 13, 9) = 18
  - Row 3: dp[3][0] = arr[3][0] + max(dp[3][1], dp[2][1]) = 9 + max(9, 9) = 18
  - Updated dp:

```
dp = [
   [16, 11, 4, 1],
   [15, 13, 9, 0],
   [18,  9, 4, 3],
   [18,  9, 8, 1]
]
```

Step 3: Find Maximum Gold

- The maximum gold collectible is max(dp[0][0], dp[1][0], dp[2][0], dp[3][0]) = 18.
- Starting rows for this maximum gold: Row 2 and Row 3.

Step 4: Trace All Paths Using BFS

Start BFS from the cells with maximum gold in

```
            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;
}
```

the first column (Row 2 and Row 3):

1. **Starting from Row 2, Column 0 (psf = "2"):**
    - Move diagonally up-right (LU): dp[1][1] = 13 → New path: "2 LU".
    - Move right (H): dp[2][1] = 9 → New path: "2 H".
    - Move diagonally down-right (LD): dp[3][1] = 9 → New path: "2 LD".
2. **Starting from Row 3, Column 0 (psf = "3"):**
    - Move diagonally up-right (LU): dp[2][1] = 9 → New path: "3 LU".
    - Move right (H): dp[3][1] = 9 → New path: "3 H".

Final Output:

```
18

Paths:
2 LU ... (continue tracing)
2 H ...
2 LD ...
3 LU ...
3 H ...
```

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 the Code

1. **Initial Setup**: The arr grid:

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

2. **Filling the DP Table**:
   - The bottom-right corner dp[3][3] is initialized as arr[3][3] = 16.
   - The last row and column are filled:
     - dp[3][2] = 16 + 12 = 28
     - dp[3][1] = 28 + 8 = 36
     - dp[3][0] = 36 + 4 = 40
     - dp[2][3] = 16 + 12 = 28
     - dp[2][2] = 28 + 8 = 36
     - dp[2][1] = 36 + 7 = 43
     - dp[2][0] = 43 + 5 = 48
     - And so on...
   - Final dp table:

     Copy code
     ```
     46  50  54  58
     51  55  59  62
     59  63  67  72
     60  64  68  72
     ```

3. **BFS to Find All Paths**:
   - The BFS starts from dp[0][0], trying to find paths with minimum cost.
   - The BFS explores possible moves using the dp values:
     - It starts by pushing the initial position (0, 0) with the path so far as "" into the queue.
     - After processing all possible paths, the minimum cost path HHHVVV is printed.

```
    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;
}
```

Output:-

```
46
HHHVVV
```

Print all path with minimum Cost In C++

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

using namespace std;
int solution(vector<int>& prices) {
    vector<int> np(prices.size() + 1);
    for (int i = 0; i < prices.size(); i++) {
        np[i + 1] = prices[i];
    }

    vector<int> dp(np.size());
    dp[0] = 0;
    dp[1] = np[1];

    for (int i = 2; i < dp.size(); i++) {
        dp[i] = np[i];

        int li = 1;
        int ri = i - 1;
        while (li <= ri) {
            if (dp[li] + dp[ri] > dp[i]) {
                dp[i] = dp[li] + dp[ri];
            }

            li++;
            ri--;
        }
    }

    return dp[dp.size() - 1];
}

int main() {
    vector<int> prices = {1, 5, 8, 9, 10, 17,
17, 20};

    cout << solution(prices) << endl;

    return 0;
}
```

Dry Run of the Code

Given prices = {1, 5, 8, 9, 10, 17, 17, 20} (rod lengths from 1 to 8):

- **Step 1**: Initialize np and dp:
    - np = {0, 1, 5, 8, 9, 10, 17, 17, 20}
    - dp = {0, 1, 0, 0, 0, 0, 0, 0, 0}
- **Step 2**: Start filling dp:
    - For i = 2 (rod length 2):
        - dp[2] = np[2] = 5
        - Check splits: 1 + 4 = 5 (no better than dp[2] = 5)
    - For i = 3 (rod length 3):
        - dp[3] = np[3] = 8
        - Check splits: 1 + 7 = 8, 5 + 3 = 8 (no better than dp[3] = 8)
    - For i = 4 (rod length 4):
        - dp[4] = np[4] = 9
        - Check splits: 1 + 8 = 9, 5 + 4 = 9 (no better than dp[4] = 9)
    - For i = 5 (rod length 5):
        - dp[5] = np[5] = 10
        - Check splits: 1 + 9 = 10, 5 + 5 = 10, 8 + 2 = 10 (no better than dp[5] = 10)
    - For i = 6 (rod length 6):
        - dp[6] = np[6] = 17
        - Check splits: 1 + 16 = 17, 5 + 12 = 17, 8 + 9 = 17, 9 + 8 = 17, 10 + 7 = 17 (no better than dp[6] = 17)
    - For i = 7 (rod length 7):
        - dp[7] = np[7] = 17
        - Check splits: 1 + 16 = 17, 5 + 12 = 17, 8 + 9 = 17, 9 + 8 = 17, 10 + 7 = 17, 17 + 0 = 17
    - For i = 8 (rod length 8):
        - dp[8] = np[8] = 20
        - Check splits: 1 + 19 = 20, 5 + 15 = 20, 8 + 12 = 20, 9 + 11 = 20, 10 + 10 = 20, 17 + 3 = 20, 17 + 3 = 20
- **Step 3**: After filling all values, the maximum revenue is found at dp[8] = 22.

Output:-

```
22
```

Temple offering In C++

```
#include <iostream>
#include <algorithm>
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 < n; i++) {
        if (height[i] > 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 >= 0; i--) {
        if (height[i] > height[i + 1]) {
            rarr[i] = rarr[i + 1] + 1;
        } else {
            rarr[i] = 1;
        }
    }

    // Calculate total offerings
    int ans = 0;
    for (int i = 0; i < 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 << totalOfferings(height, n) << endl;
    return 0;
}
```

Step-by-Step Execution for Input: {2, 3, 5, 6, 4, 8, 9}

1. **Initialization**:

   height = {2, 3, 5, 6, 4, 8, 9}
   n = 7
   larr = {1, 1, 1, 1, 1, 1, 1}
   rarr = {1, 1, 1, 1, 1, 1, 1} **Calculating Left Offerings**:

   - For i = 1: height[1] = 3 > height[0] = 2, so larr[1] = larr[0] + 1 = 2
   - For i = 2: height[2] = 5 > height[1] = 3, so larr[2] = larr[1] + 1 = 3
   - For i = 3: height[3] = 6 > height[2] = 5, so larr[3] = larr[2] + 1 = 4
   - For i = 4: height[4] = 4 <= height[3] = 6, so larr[4] = 1
   - For i = 5: height[5] = 8 > height[4] = 4, so larr[5] = larr[4] + 1 = 2
   - For i = 6: height[6] = 9 > height[5] = 8, so larr[6] = larr[5] + 1 = 3

   After this, larr = {1, 2, 3, 4, 1, 2, 3}.

2. **Calculating Right Offerings**:
   - For i = 5: height[5] = 8 > height[6] = 9, so rarr[5] = 1
   - For i = 4: height[4] = 4 <= height[5] = 8, so rarr[4] = 1
   - For i = 3: height[3] = 6 > height[4] = 4, so rarr[3] = rarr[4] + 1 = 2
   - For i = 2: height[2] = 5 <= height[3] = 6, so rarr[2] = 1
   - For i = 1: height[1] = 3 > height[2] = 5, so rarr[1] = rarr[2] + 1 = 2
   - For i = 0: height[0] = 2 <= height[1] = 3, so rarr[0] = 1

   After this, rarr = {1, 2, 1, 2, 1, 1, 1}.

3. **Final Offerings Calculation**: Now calculate the total offerings by summing the maximum of left and right offerings for each person:

   total = max(larr[0], rarr[0]) + max(larr[1], rarr[1]) + max(larr[2], rarr[2]) +

	max(larr[3], rarr[3]) + max(larr[4], rarr[4]) + max(larr[5], rarr[5]) + max(larr[6], rarr[6]) = 1 + 2 + 3 + 4 + 1 + 2 + 3 = 16
Output:- 16	

Word Break In C++

```cpp
#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;
}
```

Step-by-Step Dry Run

Input:

- sentence = "ilikepeppereatingmangoinpepcoding"
- dict = {"i", "like", "pep", "coding", "pepper", "eating", "mango", "man", "go", "in", "pepcoding"}

Initialization:

- dp = [0, 0, 0, ..., 0] (length of 39 for the sentence "ilikepeppereatingmangoinpepcoding").
- We iterate over each character of the sentence.

Loop Execution:

- **At i = 0** (i.e., the first character 'i'):
    - Substring: "i" (valid word found in the dictionary), so dp[0] = 1.
- **At i = 1** (i.e., the second character 'l'):
    - Substrings: "il" (not in dictionary), "l" (not in dictionary).
- **At i = 2** (i.e., the third character 'i'):
    - Substrings: "ili" (not in dictionary), "li" (not in dictionary), "i" (valid word, so dp[2] += dp[0] = 1).
- **At i = 3** (i.e., the fourth character 'k'):
    - Substrings: "like" (valid word found in dictionary), so dp[3] = dp[0] + 1 = 1.
- **At i = 4** (i.e., the fifth character 'e'):
    - Substrings: "like" (valid word found in dictionary), "i" (valid word found in dictionary), so dp[4] = 1 (from "like") and dp[4] = dp[3] + 1 = 2 (from "i").
- **And so on...** The algorithm continues checking for substrings, updating the dp array for each valid word found.

Output:

- After filling the dp array, dp[38] contains the number of ways to segment the entire sentence. It turns out that there are 4 ways to split the sentence into valid words:
    - "i like pepper eating mango in pep coding"
    - "i like pepper eating mango in pepcoding"

<table>
<tr><td></td><td>○ "i like pep per eating mango in pep coding"
○ "i like pep per eating mango in pepcoding"

So, the function will print 4 and return true because the sentence can indeed be segmented.</td></tr>
<tr><td colspan="2">Output:-
4
true</td></tr>
</table>