

0/1 KnapSack in C++

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

using namespace std;

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

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

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

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

int main() {
    ZeroOneKnapsack solution;

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

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

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

    return 0;
}
```

Dry Run of the ZeroOneKnapsack Problem:

Input:

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

Step 1: Initialize the DP Table

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

Initially, the DP table is filled with zeros:

| i\j | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|-----|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 2 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 3 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 4 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 5 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

Step 2: Fill the DP Table

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

Item 1 (Value = 15, Weight = 2)

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

| i\j | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|-----|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

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

Item 2 (Value = 14, Weight = 5)

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

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

Item 3 (Value = 10, Weight = 1)

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

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

Item 4 (Value = 45, Weight = 3)

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

Item 5 (Value = 30, Weight = 4)

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

Step 3: Final DP Table

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

Result:

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

Output:

Maximum value that can be obtained: 75

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

Best time to buy and sell stocks in C++

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

```
using namespace std;
```

```
class BestTimeToBuyAndSellStock {
public:
```

```
    int maxProfit(vector<int>& prices) {
        if (prices.empty()) return 0;
```

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

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

```
        return maxP;
    }
};
```

```
int main() {
    BestTimeToBuyAndSellStock solution;

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

    return 0;
}
```

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

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

🧠 Variables:

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

🔍 Dry Run Table:

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

✓ Final Answer:

Max profit for prices1: 5

Output:-

maxP = 5 (Maximum profit)

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

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

using namespace std;

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

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

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

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

int main() {
    BestTimeToBuyAndSellStocksInfiniteTransactions
    solution;

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

    return 0;
}
```

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

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


✓ Logic Summary:

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

📄 Dry Run Table:

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

✓ Final Output:

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

Climbing Stairs in C++

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

using namespace std;

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

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

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

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

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

    return 0;
}
```

Given:

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

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

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

Dry Run with Iteration Table

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

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

Final dp Array

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

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

Output:

2 1 2 1 1 0

Output:-

🕒 Printed dp: 2 1 2 1 1 0

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

Coin Change Combination in C++

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

using namespace std;

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

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

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

    return 0;
}
```

Initial dp Array

Before processing:

arr=[2, 3, 5]

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

(Index represents amount: 0 to 7)

Dry Run with Iteration Table

Processing coin 2

| j (amt) | dp[j] = dp[j] + dp[j - 2] | Updated dp |
|------------|------------------------------|--------------------------|
| 2 | dp[2] += dp[0] = 1 | [1, 0, 1, 0, 0, 0, 0, 0] |
| 3 | dp[3] += dp[1] = 0 | [1, 0, 1, 0, 0, 0, 0, 0] |
| 4 | dp[4] += dp[2] = 1 | [1, 0, 1, 0, 1, 0, 0, 0] |
| 5 | dp[5] += dp[3] = 0 | [1, 0, 1, 0, 1, 0, 0, 0] |
| 6 | dp[6] += dp[4] = 1 | [1, 0, 1, 0, 1, 0, 1, 0] |
| 7 | dp[7] += dp[5] = 0 | [1, 0, 1, 0, 1, 0, 1, 0] |

Processing coin 3

| j (amt) | dp[j] = dp[j] + dp[j - 3] | Updated dp |
|------------|------------------------------|--------------------------|
| 3 | dp[3] += dp[0] = 1 | [1, 0, 1, 1, 1, 0, 0, 0] |
| 4 | dp[4] += dp[1] = 0 | [1, 0, 1, 1, 1, 0, 0, 0] |
| 5 | dp[5] += dp[2] = 1 | [1, 0, 1, 1, 1, 1, 0, 0] |
| 6 | dp[6] += dp[3] = 1 | [1, 0, 1, 1, 1, 1, 2, 0] |
| 7 | dp[7] += dp[4] = 1 | [1, 0, 1, 1, 1, 1, 2, 1] |

Processing coin 5

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

Final dp Array

After processing all coins:

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

Final Output

2

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

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

Output:-

2

Coin Change Permutation in C++

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

using namespace std;

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

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

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

    return 0;
}
```

Initial dp Array

Before processing:

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

Dry Run with Iteration Table

Iterating over amt from 1 to 7

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

Final dp Array

| | |
|--------------------------|--|
| | <p>After processing all amounts:</p> <p><code>dp = [1, 0, 1, 1, 1, 3, 2, 5]</code></p> <p>Final Output</p> <p>5</p> <p>This means there are 5 different permutations to form amount 7 using {2, 3, 5}:</p> <ol style="list-style-type: none">1. 2 + 2 + 32. 2 + 3 + 23. 3 + 2 + 24. 2 + 55. 5 + 2 |
| <p>Output:-</p> <p>5</p> | |

Friend's Pairing in C++

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

using namespace std;

int main() {
    int n = 3;

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

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

    cout << dp[n] << endl;

    return 0;
}
```

Dry Run with Iteration Table

Initial State

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

Iterating from i = 3 to n = 3

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

Final dp Array

dp = [?, 1, 2, 4]

Final Output

4

Output:-
4

GoldMine in C++

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

using namespace std;

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

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

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

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

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

    cout << maxGold << endl;

    return 0;
}
```

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

- right (→)
- right-up (↗)
- right-down (↘)

📌 **Given grid[4][4]:**

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

🧠 **DP Formula:**

For $dp[i][j]$:

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

🔄 **Filling dp from right to left:**

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

Step-by-step (column by column):

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

Fill Column 2 ($j = 2$):

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

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

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

Fill Column 1 (j = 1):

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

Fill Column 0 (j = 0):

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

✔ Final dp Table:

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

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

✔ Output:

24

Output:
24

Min Cost Path in C++

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

using namespace std;

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

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

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

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

    return 0;
}
```

Input Grid:

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

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

✓ DP Formula Recap:

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

📊 DP Table (Filled from bottom-right):

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

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

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

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

Now build upward:

Row 2:

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

Row 1:

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

Row 0:

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

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

✔ **Final DP Table:**

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

📄 **Output:**

22

Output:
22

Paint Houses in C++

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

using namespace std;

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

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

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

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

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

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

    return 0;
}
```

Input Matrix (Cost of painting houses):

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

We denote the colors as:

- 0 → Red
- 1 → Blue
- 2 → Green

DP Table Filling Explanation:

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

✔ Final DP Table:

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

Output:

The minimum total cost is:

min(8, 10, 11) = 8

Output:-
8

Target sum Subset in C++

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

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

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

    return dp[n][target];
}

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

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

    return 0;
}
```

We have **array**:

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

We create a **dp table of size (n+1) x (target+1)**:
dp[i][j] → i is the first i elements, j is the sum.

Initial Table (Before Processing)

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

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

Step 2: Fill the Table

We iterate through i = 1 to n, updating dp[i][j].

Processing arr[0] = 4

We consider only element 4.

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

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

Processing arr[1] = 2

Now considering {4,2}:

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

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

Processing arr[2] = 7

Now considering {4,2,7}:

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

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

Processing arr[3] = 1

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

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

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

Processing arr[4] = 3

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

Final Answer

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

Tiling with Dominoes in C++

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

using namespace std;

int main() {
    int n = 2;

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

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

    cout << dp[n] << endl;

    return 0;
}
```

Given:

- $n = 2$

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

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

Initial dp Table:

| i\dp | 0 | 1 | 2 |
|------|---|---|---|
| i=0 | 0 | | |
| i=1 | | 1 | |
| i=2 | | | 2 |

At this point:

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

For Loop Execution:

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

Final dp Table:

The table remains unchanged from initialization:

| i\dp | 0 | 1 | 2 |
|------|---|---|---|
| i=0 | 0 | | |
| i=1 | | 1 | |
| i=2 | | | 2 |

Final Output:

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

Output:-

2