0/1 KnapSack in C++

```cpp
#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

Input:

- Number of items: n = 5
- Values: vals = {15, 14, 10, 45, 30}
- Weights: wts = {2, 5, 1, 3, 4}
- Capacity: cap = 7

Steps:

1. **Initialize the DP Table**:
   - dp is a 2D table of size (n+1) × (cap+1) (i.e., 6 × 8).
   - Initially, all entries are 0.

DP Table Construction

Base Case:

dp[0][j] = 0 for all j
dp[i][0] = 0 for all i

DP Transitions:

- **Row 1 (i = 1, item with value = 15, weight = 2)**:
  - For j = 1: dp[1][1] = 0 (weight exceeds capacity).
  - For j = 2: dp[1][2] = 15 (item included).
  - For j = 3 to 7: dp[1][j] = 15 (item included).
- **Row 2 (i = 2, item with value = 14, weight = 5)**:
  - For j = 1 to 4: dp[2][j] = dp[1][j].
  - For j = 5: dp[2][5] = max(dp[1][5], vals[1] + dp[1][5 - wts[1]]) = max(15, 14) = 15.
  - For j = 6: dp[2][6] = max(15, 14) = 15.
  - For j = 7: dp[2][7] = max(15, 15 + 14) = 29.
- **Row 3 (i = 3, item with value = 10, weight = 1)**:
  - Updates based on the new item's inclusion.
- **Row 4 (i = 4, item with value = 45, weight = 3)**:
  - Updates based on the new item's inclusion.
- **Row 5 (i = 5, item with value = 30, weight = 4)**:
  - Updates based on the new item's inclusion.

	Final DP Table:
	
	dp = {
	{ 0, 0, 0, 0, 0, 0, 0, 0 },
	{ 0, 0,15,15,15,15,15,15 },
	{ 0, 0,15,15,15,15,29,29 },
	{ 0,10,15,25,25,25,29,40 },
	{ 0,10,15,45,55,55,70,70 },
	{ 0,10,15,45,55,55,70,75 },
	}

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

Best time to buy and sell stocks in C++

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

using namespace std;

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

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

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

        return maxP;
    }
};

int main() {
    BestTimeToBuyAndSellStock solution;

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

    return 0;
}
```

Input:

- prices = {7, 1, 5, 3, 6, 4}

Initialization:

- maxP = 0 (maximum profit so far)
- minBP = prices[0] = 7 (minimum buying price)

Iteration:

1. **Day 1** (prc = 7):
   - tp = prc - minBP = 7 - 7 = 0
   - maxP = max(maxP, tp) = max(0, 0) = 0
   - minBP = min(minBP, prc) = min(7, 7) = 7
2. **Day 2** (prc = 1):
   - tp = prc - minBP = 1 - 7 = -6
   - maxP = max(maxP, tp) = max(0, -6) = 0
   - minBP = min(minBP, prc) = min(7, 1) = 1
3. **Day 3** (prc = 5):
   - tp = prc - minBP = 5 - 1 = 4
   - maxP = max(maxP, tp) = max(0, 4) = 4
   - minBP = min(minBP, prc) = min(1, 5) = 1
4. **Day 4** (prc = 3):
   - tp = prc - minBP = 3 - 1 = 2
   - maxP = max(maxP, tp) = max(4, 2) = 4
   - minBP = min(minBP, prc) = min(1, 3) = 1
5. **Day 5** (prc = 6):
   - tp = prc - minBP = 6 - 1 = 5
   - maxP = max(maxP, tp) = max(4, 5) = 5
   - minBP = min(minBP, prc) = min(1, 6) = 1
6. **Day 6** (prc = 4):
   - tp = prc - minBP = 4 - 1 = 3
   - maxP = max(maxP, tp) = max(5, 3) = 5
   - minBP = min(minBP, prc) = min(1, 4) = 1

Output:-
maxP = 5 (Maximum profit)

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

```cpp
#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;
}
```

Dry Run

Input:

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

Step-by-Step Execution:

- **Initialization**:
  - bd = 0, sd = 0, profit = 0.
- **Iterate Over Prices**:
  - **Day 1 (Price 6)**:
    - prices[1] < prices[0] → Calculate profit:
      - profit += prices[0] - prices[0] = 0 → No profit.
      - Update bd = 1, sd = 1.
  - **Day 2 (Price 7)**:
    - prices[2] >= prices[1] → sd = 2.
  - **Day 3 (Price 19)**:
    - prices[3] >= prices[2] → sd = 3.
  - **Day 4 (Price 4)**:
    - prices[4] < prices[3] → Calculate profit:
      - profit += prices[3] - prices[1] = 19 - 6 = 13.
      - Update bd = 4, sd = 4.
  - **Day 5 (Price 1)**:
    - prices[5] < prices[4] → No profit.
      - Update bd = 5, sd = 5.
  - **Day 6 (Price 6)**:
    - prices[6] >= prices[5] → sd = 6.
  - **Day 7 (Price 18)**:
    - prices[7] >= prices[6] → sd = 7.
  - **Day 8 (Price 4)**:
    - prices[8] < prices[7] → Calculate profit:
      - profit += prices[7] - prices[5] = 18 - 1 = 17.
      - Update bd = 8, sd = 8.
- **After Loop**:
  - Add remaining profit:
    - profit += prices[8] - prices[8] = 0.

Final Profit:

	• profit = 13 + 17 = 30.

Output:-
Max profit: 30

Climbing Stairs in C++	

<pre>#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;
}</pre>
	Input:
	
	• arr = {1, 5, 2, 3, 1}
	
	Initialization:
	
	• n = 5 (size of arr)
	• dp = {INT_MAX, INT_MAX, INT_MAX, INT_MAX, INT_MAX, 0} (base case: dp[n] = 0)
	
	Iterations:
	
	Step 1: Start from i = 4:
	
	• arr[4] = 1 → Maximum jump = 1
	• Valid jump: j = 1
	    ○ dp[4] = min(dp[5]) + 1 = 0 + 1 = 1
	• Updated dp: {INT_MAX, INT_MAX, INT_MAX, INT_MAX, 1, 0}
	
	Step 2: i = 3:
	
	• arr[3] = 3 → Maximum jump = 3
	• Valid jumps: j = 1, 2
	    ○ dp[3] = min(dp[4], dp[5]) + 1 = min(1, 0) + 1 = 1
	• Updated dp: {INT_MAX, INT_MAX, INT_MAX, 1, 1, 0}
	
	Step 3: i = 2:
	
	• arr[2] = 2 → Maximum jump = 2
	• Valid jumps: j = 1, 2
	    ○ dp[2] = min(dp[3], dp[4]) + 1 = min(1, 1) + 1 = 2
	• Updated dp: {INT_MAX, INT_MAX, 2, 1, 1, 0}
	
	Step 4: i = 1:
	
	• arr[1] = 5 → Maximum jump = 5
	• Valid jumps: j = 1, 2, 3, 4
	    ○ dp[1] = min(dp[2], dp[3], dp[4], dp[5]) + 1 = min(2, 1, 1, 0) + 1 = 1
	• Updated dp: {INT_MAX, 1, 2, 1, 1, 0}

	Step 5: i = 0: • arr[0] = 1 → Maximum jump = 1 • Valid jump: j = 1 ○ dp[0] = min(dp[1]) + 1 = 1 + 1 = 2 • Updated dp: {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++	

<pre>#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;
}</pre>
	Input:
	
	• arr = {2, 3, 5}
	• amt = 7
	
	Initialization:
	
	• dp = {1, 0, 0, 0, 0, 0, 0, 0} (size = amt + 1, initialized to 0 except dp[0] = 1).
	
	Iterations:
	
	Step 1: Using Coin 2 (arr[0]):
	
	• For each j from 2 to 7:
	    ○ dp[j] += dp[j - 2]
	• Updates:
	
	dp[2] = dp[2] + dp[0] = 0 + 1 = 1
	dp[3] = dp[3] + dp[1] = 0 + 0 = 0
	dp[4] = dp[4] + dp[2] = 0 + 1 = 1
	dp[5] = dp[5] + dp[3] = 0 + 0 = 0
	dp[6] = dp[6] + dp[4] = 0 + 1 = 1
	dp[7] = dp[7] + dp[5] = 0 + 0 = 0
	
	• dp = {1, 0, 1, 0, 1, 0, 1, 0}
	
	Step 2: Using Coin 3 (arr[1]):
	
	• For each j from 3 to 7:
	    ○ dp[j] += dp[j - 3]
	• Updates:
	
	dp[3] = dp[3] + dp[0] = 0 + 1 = 1
	dp[4] = dp[4] + dp[1] = 1 + 0 = 1
	dp[5] = dp[5] + dp[2] = 0 + 1 = 1
	dp[6] = dp[6] + dp[3] = 1 + 1 = 2
	dp[7] = dp[7] + dp[4] = 0 + 1 = 1
	
	• dp = {1, 0, 1, 1, 1, 1, 2, 1}
	
	Step 3: Using Coin 5 (arr[2]):
	
	• For each j from 5 to 7:
	    ○ dp[j] += dp[j - 5]
	• Updates:
	
	dp[5] = dp[5] + dp[0] = 1 + 1 = 2
	dp[6] = dp[6] + dp[1] = 2 + 0 = 2
	dp[7] = dp[7] + dp[2] = 1 + 1 = 2

	- dp = {1, 0, 1, 1, 1, 2, 2, 2} **Final DP Array:** dp = {1, 0, 1, 1, 1, 2, 2, 2} **Output:** - dp[amt] = dp[7] = 2 - There are **2 ways** to form amount 7 using coins {2, 3, 5}.

Output:-
2

Coin Change Permutation in C++

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

using namespace std;

int main() {
    vector<int> coins = {2, 3, 5, 6};
    int tar = 10;
    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;
}
```

Dry Run:

Input:

coins = {2, 3, 5, 6}, target = 10

Initialization:

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

Loop Execution:

For amount amt = 1:

- coin = 2: No, as coin > amt.
- coin = 3: No, as coin > amt.
- coin = 5: No, as coin > amt.
- coin = 6: No, as coin > amt.

dp[1] = 0

For amount amt = 2:

- coin = 2: Yes, we can use one 2 to make 2. dp[2] += dp[0] (dp[0] is 1).
- coin = 3: No.
- coin = 5: No.
- coin = 6: No.

dp[2] = 1

For amount amt = 3:

- coin = 2: Yes, use one 2 and then add 1 way to make 1 (dp[1]).
- coin = 3: Yes, one 3 will form 3 (dp[0]).
- coin = 5: No.
- coin = 6: No.

dp[3] = 2

For amount amt = 4:

- coin = 2: Yes, use 2 and then form dp[2] ways.
- coin = 3: Yes, use 3 and then form dp[1] ways.
- coin = 5: No.
- coin = 6: No.

dp[4] = 3

For amount amt = 5:

- coin = 2: Yes, use 2 and form dp[3] ways.
- coin = 3: Yes, use 3 and form dp[2] ways.
- coin = 5: Yes, use 5 to make dp[0].

- coin = 6: No.

dp[5] = 4

For amount amt = 6:

- coin = 2: Yes, use 2 and form dp[4] ways.
- coin = 3: Yes, use 3 and form dp[3] ways.
- coin = 5: Yes, use 5 and form dp[1] ways.
- coin = 6: Yes, use 6 to make dp[0].

dp[6] = 5

For amount amt = 7:

- coin = 2: Yes, use 2 and form dp[5] ways.
- coin = 3: Yes, use 3 and form dp[4] ways.
- coin = 5: Yes, use 5 and form dp[2] ways.
- coin = 6: Yes, use 6 and form dp[1] ways.

dp[7] = 8

For amount amt = 8:

- coin = 2: Yes, use 2 and form dp[6] ways.
- coin = 3: Yes, use 3 and form dp[5] ways.
- coin = 5: Yes, use 5 and form dp[3] ways.
- coin = 6: Yes, use 6 and form dp[2] ways.

dp[8] = 12

For amount amt = 9:

- coin = 2: Yes, use 2 and form dp[7] ways.
- coin = 3: Yes, use 3 and form dp[6] ways.
- coin = 5: Yes, use 5 and form dp[4] ways.
- coin = 6: Yes, use 6 and form dp[3] ways.

dp[9] = 20

For amount amt = 10:

- coin = 2: Yes, use 2 and form dp[8] ways.
- coin = 3: Yes, use 3 and form dp[7] ways.
- coin = 5: Yes, use 5 and form dp[5] ways.
- coin = 6: Yes, use 6 and form dp[4] ways.

dp[10] = 33

Final Output:

dp[10] = 33

Output:-
33

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:

Input:

n = 3

Initialization:

dp[1] = 1
dp[2] = 2

Loop Execution:

Step 1: i = 3

```
dp[3] = dp[2] + dp[1] * (3 - 1)
      = 2 + 1 * 2
      = 2 + 2
      = 4
```

Final Result:

dp[3] = 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;
}
```

Dry Run

Input Grid:

```
grid = {
  {8, 2, 1, 6},
  {6, 5, 5, 2},
  {2, 1, 0, 3},
  {7, 2, 2, 4}
}
```

Steps:

1. **Initialization**:
   - n = 4 (rows), m = 4 (columns).
   - Create a dp table with the same dimensions as grid.
2. **Filling DP Table**:
   - Start from the last column (j = 3) and work backward to the first column (j = 0).

Filling DP Table:

- **Column 3 (last column)**:

  dp[i][3] = grid[i][3] for all i
  ```
  dp = {
    {0, 0, 0, 6},
    {0, 0, 0, 2},
    {0, 0, 0, 3},
    {0, 0, 0, 4}
  }
  ```

- **Column 2**:

  dp[0][2] = grid[0][2] + max(dp[0][3], dp[1][3]) = 1 + max(6, 2) = 7
  dp[1][2] = grid[1][2] + max(dp[0][3], dp[1][3], dp[2][3]) = 5 + max(6, 2, 3) = 11
  dp[2][2] = grid[2][2] + max(dp[1][3], dp[2][3], dp[3][3]) = 0 + max(2, 3, 4) = 4
  dp[3][2] = grid[3][2] + max(dp[2][3], dp[3][3]) = 2 + max(3, 4) = 6
  ```
  dp = {
    {0, 0, 7, 6},
    {0, 0, 11, 2},
    {0, 0, 4, 3},
    {0, 0, 6, 4}
  }
  ```

- **Column 1**:

  dp[0][1] = grid[0][1] + max(dp[0][2], dp[1][2]) = 2 + max(7, 11) = 13
  dp[1][1] = grid[1][1] + max(dp[0][2], dp[1][2], dp[2][2]) = 5 + max(7, 11, 4) =

	16
	dp[2][1] = grid[2][1] + max(dp[1][2], dp[2][2], dp[3][2]) = 1 + max(11, 4, 6) = 12
	dp[3][1] = grid[3][1] + max(dp[2][2], dp[3][2]) = 2 + max(4, 6) = 8
	dp = {
	{0, 13, 7, 6},
	{0, 16, 11, 2},
	{0, 12, 4, 3},
	{0, 8, 6, 4}
	}
	
	• **Column 0**:
	
	dp[0][0] = grid[0][0] + max(dp[0][1], dp[1][1]) = 8 + max(13, 16) = 24
	dp[1][0] = grid[1][0] + max(dp[0][1], dp[1][1], dp[2][1]) = 6 + max(13, 16, 12) = 22
	dp[2][0] = grid[2][0] + max(dp[1][1], dp[2][1], dp[3][1]) = 2 + max(16, 12, 8) = 18
	dp[3][0] = grid[3][0] + max(dp[2][1], dp[3][1]) = 7 + max(12, 8) = 19
	dp = {
	{24, 13, 7, 6},
	{22, 16, 11, 2},
	{18, 12, 4, 3},
	{19, 8, 6, 4}
	}
	
	Final Step:
	
	• The maximum value in the first column (dp[i][0] for all i) is:
	
	maxGold = max(24, 22, 18, 19) = 24

Output: 24	

Min Cost Path in C++

```cpp
#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;
}
```

Dry Run

Input Grid:

```
grid = {
    {8, 2, 1, 6},
    {6, 5, 5, 2},
    {2, 1, 0, 3},
    {7, 2, 2, 4}
}
```

Steps:

1. **Initialization**:
   - Create a dp table with dimensions' n × m (initialized to 0).
2. **Filling the DP Table**:
   - Start from the bottom-right corner (n-1, m-1) and work backwards.

Filling DP Table:

- **Bottom-right corner (i = 3, j = 3)**:

  dp[3][3] = grid[3][3] = 4

- **Last row (i = 3)**:

  dp[3][2] = grid[3][2] + dp[3][3] = 2 + 4 = 6
  dp[3][1] = grid[3][1] + dp[3][2] = 2 + 6 = 8
  dp[3][0] = grid[3][0] + dp[3][1] = 7 + 8 = 15

- **Last column (j = 3)**:

  dp[2][3] = grid[2][3] + dp[3][3] = 3 + 4 = 7
  dp[1][3] = grid[1][3] + dp[2][3] = 2 + 7 = 9
  dp[0][3] = grid[0][3] + dp[1][3] = 6 + 9 = 15

- **Remaining cells**:
  - **Row 2**:

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

  - **Row 1**:

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

<table>
<tr><td></td><td>

- **Row 0**:

  dp[0][2] = grid[0][2] + min(dp[0][3], dp[1][2]) = 1 + min(15, 11) = 12
  dp[0][1] = grid[0][1] + min(dp[0][2], dp[1][1]) = 2 + min(12, 12) = 14
  dp[0][0] = grid[0][0] + min(dp[0][1], dp[1][0]) = 8 + min(14, 15) = 22

Final DP Table:

dp = {
    {22, 14, 12, 15},
    {15, 12, 11, 9},
    {9, 7, 6, 7},
    {15, 8, 6, 4}
}
</td></tr>
<tr><td colspan="2">Output:
22</td></tr>
</table>

Paint Houses in C++	

<pre>#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;
}</pre>
	Input:
	arr = {{1, 5, 7}, {5, 8, 4}, {3, 2, 9}, {1, 2, 4}}
	n = 4 (number of houses)
	
	Steps:
	
	1. Initialization of dp Array:
	  ○ dp[i][j] will store the minimum cost to paint up to the i-th house, ending with color j.
	  ○ Base case: For the first house (i = 0), we directly take the cost from the input arr.
	
	  dp[0][0] = arr[0][0] = 1
	  dp[0][1] = arr[0][1] = 5
	  dp[0][2] = arr[0][2] = 7
	
	2. Filling the dp Array (Dynamic Programming):
	For each house i from 1 to n-1, calculate the cost for each color j by considering the minimum cost of the other two colors for the previous house.
	Formula:
	
	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])
	
	3. Extract the Minimum Cost:
	After filling the dp array, the result is the minimum value from the last row (dp[n-1]).
	
	Dry Run Details:
	
	Step 1: Initialization (i = 0)
	
	dp[0][0] = 1
	dp[0][1] = 5
	dp[0][2] = 7
	
	Step 2: Fill dp for i = 1
	
	dp[1][0] = arr[1][0] + min(dp[0][1], dp[0][2])
	    = 5 + min(5, 7) = 5 + 5 = 10
	
	dp[1][1] = arr[1][1] + min(dp[0][0], dp[0][2])
	    = 8 + min(1, 7) = 8 + 1 = 9
	
	dp[1][2] = arr[1][2] + min(dp[0][0], dp[0][1])
	    = 4 + min(1, 5) = 4 + 1 = 5

	State of dp: dp[1] = {10, 9, 5} Step 3: Fill dp for i = 2 dp[2][0] = arr[2][0] + min(dp[1][1], dp[1][2]) = 3 + min(9, 5) = 3 + 5 = 8 dp[2][1] = arr[2][1] + min(dp[1][0], dp[1][2]) = 2 + min(10, 5) = 2 + 5 = 7 dp[2][2] = arr[2][2] + min(dp[1][0], dp[1][1]) = 9 + min(10, 9) = 9 + 9 = 18 State of dp: dp[2] = {8, 7, 18} Step 4: Fill dp for i = 3 dp[3][0] = arr[3][0] + min(dp[2][1], dp[2][2]) = 1 + min(7, 18) = 1 + 7 = 8 dp[3][1] = arr[3][1] + min(dp[2][0], dp[2][2]) = 2 + min(8, 18) = 2 + 8 = 10 dp[3][2] = arr[3][2] + min(dp[2][0], dp[2][1]) = 4 + min(8, 7) = 4 + 7 = 11 State of dp: dp[3] = {8, 10, 11} Step 5: Extract the Result The minimum cost to paint all houses is the minimum value in the last row of dp: ans = min(dp[3][0], dp[3][1], dp[3][2]) = 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;
}
```

Dry Run

Input:

- Array: arr = {4, 2, 7, 1, 3}
- Target: target = 10

Steps:

1. **Initialize DP Table**:
   - dp has dimensions (n+1) × (target+1), i.e., 6 × 11 (since n = 5 and target = 10).
2. **Fill the DP Table**:
   - Start filling the table row by row, column by column.

DP Table Construction

Initial DP Table:

dp[i][j] = false for all i, j

Base Cases:

- dp[i][0] = true for all i.
- dp[0][j] = false for j > 0.

DP Transitions:

- **Row 1 (i = 1, element = 4)**:
  - For j = 1, 2, 3: dp[1][j] = false (4 cannot form these sums).
  - For j = 4: dp[1][4] = true (4 forms sum 4).
  - For j = 5 to 10: dp[1][j] = false.
- **Row 2 (i = 2, element = 2)**:
  - For j = 1: dp[2][1] = false.
  - For j = 2: dp[2][2] = true (2 forms sum 2).
  - For j = 4: dp[2][4] = true (Subset {4}).
  - For j = 6: dp[2][6] = true (Subset {4, 2}).
  - For j = 7 to 10: dp[2][j] = false.
- **Row 3 (i = 3, element = 7)**:
  - For j = 7: dp[3][7] = true (7 forms sum 7).
  - For j = 9: dp[3][9] = true (Subset {2, 7}).
  - For j = 10: dp[3][10] = true (Subset {4, 7}).
- **Row 4 (i = 4, element = 1)**:
  - For j = 1: dp[4][1] = true (1 forms sum 1).
  - For j = 10: dp[4][10] = true (Subset

<table>
<tr>
<td></td>
<td>
{4, 7}).
• Row 5 (i = 5, element = 3):
  ○ For j = 10: dp[5][10] = true (Subset {4, 3, 3}).

Final DP Table:

dp = {
{T, F, F, F, F, F, F, F, F, F, F},
{T, F, F, F, T, F, F, F, F, F, F},
{T, F, T, F, T, F, T, F, F, F, F},
{T, F, T, F, T, F, T, T, F, T, T},
{T, T, T, F, T, T, T, T, T, T, T},
}
</td>
</tr>
<tr>
<td colspan="2">Output:-
True
dp[n][target] is dp[5][10] = true</td>
</tr>
</table>

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;
}
```

Initial Setup:

- Input: n = 2.
- A dp array of size n+1 is created, i.e., dp[3].

Step 1: Initialize Base Cases

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

At this point, the dp array looks like:

dp = [0, 1, 2]

Step 2: Iterative Calculation

The for loop starts from i = 3 and runs up to n. However, since n = 2, the loop condition i <= n is **not satisfied**. Hence, the loop does **not execute**.

Step 3: Output the Result

The program outputs the value of dp[n], which is dp[2] = 2.

Output:-
2