C++ Vector and Dynamic Programming Complete Guide

Contents

Introduction

This guide aims to help readers systematically learn C++ Vector containers and dynamic programming algorithms, from basic concepts to advanced applications, providing a complete learning path and abundant code examples.

1 Part 1: C++ Vector Master Tutorial

1.1 Vector Basic Concepts

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

// What is Vector?
// - Dynamic array, automatic memory management
// - Continuous storage, supports random access
// - Size can be dynamically adjusted
```

Listing 1: Vector Basic Concepts

1.2 Declaration and Initialization

```
// Basic declarations
vector<int> vec1;                    // Empty vector
vector<int> vec2(5);                 // 5 elements, default value 0
vector<int> vec3(5, 10);             // 5 elements, each is 10
vector<int> vec4 = {1, 2, 3, 4, 5};  // Initialization list

// Initialize from array
int arr[] = {1, 2, 3, 4, 5};
vector<int> vec5(arr, arr + 5);

// Initialize from another vector
vector<int> vec6(vec4);
vector<int> vec7(vec4.begin(), vec4.end());
```

Listing 2: Vector Declaration and Initialization

1.3 Basic Operations with Examples

1.3.1 Adding Elements

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

int main() {
    vector<int> vec = {1, 2, 3};

    // push_back() - add element at end
    vec.push_back(4);
    cout << "After push_back(4): ";
```

```
    for (int num : vec) cout << num << " ";  // Output: 1 2 3 4

    // insert() - insert at specific position
    vec.insert(vec.begin(), 0);  // Insert 0 at beginning
    cout << "After inserting 0 at beginning: ";
    for (int num : vec) cout << num << " ";  // Output: 0 1 2 3 4

    return 0;
}
```

Listing 3: Adding Elements

1.3.2 Accessing Elements

```
vector<string> fruits = {"apple", "banana", "cherry"};

// Using [] operator
cout << fruits[0] << endl;         // Output: apple

// Using at() (safe, checks bounds)
cout << fruits.at(1) << endl;      // Output: banana

// Access first and last elements
cout << fruits.front() << endl;    // Output: apple
cout << fruits.back() << endl;     // Output: cherry
```

Listing 4: Accessing Elements

1.3.3 Removing Elements

```
vector<int> vec = {1, 2, 3, 4, 5, 6};

// pop_back() - remove last element
vec.pop_back();                     // Remove 6

// erase() - remove element at specific position
vec.erase(vec.begin());            // Remove first element 1

// clear() - remove all elements
vec.clear();                       // Clear vector
```

Listing 5: Removing Elements

1.3.4 Size and Capacity Operations

```
vector<int> vec;

// Size information
cout << vec.size();       // Number of elements
cout << vec.capacity();   // Capacity
cout << vec.empty();      // Whether empty

// Resizing
vec.resize(10);           // Resize to 10 elements
vec.reserve(100);         // Pre-allocate capacity
```

Listing 6: Size and Capacity Operations

1.4 Range-based for Loop

```
vector<string> data = {"line1", "line2", "line3"};

// Read-only traversal
for (const string& row : data) {
    cout << row << endl;  // Can read, cannot modify
}

// Modifiable traversal
for (string& row : data) {
    row += " modified";   // Can modify original elements
}
```

Listing 7: Range-based for Loop

1.5 Two-dimensional and Multi-dimensional Vector

```
// Two-dimensional vector
vector<vector<int>> matrix;

// Initialization methods
vector<vector<int>> matrix1(3, vector<int>(4, 0)); // 3x4 all-zero
    matrix
vector<vector<int>> matrix2 = {
    {1, 2, 3},
    {4, 5, 6},
    {7, 8, 9}
};

// Access
matrix[0][1] = 10;
```

Listing 8: Multi-dimensional Vector

1.6 Performance Optimization Techniques

```
// 1. Pre-allocate capacity
vector<int> vec;
vec.reserve(1000);  // Pre-allocate if approximate size is known

// 2. Use emplace_back to avoid copying
vector<pair<int, string>> vec;
vec.emplace_back(1, "hello");  // Construct in place, avoid temporary
    objects

// 3. Move semantics
vector<string> vec1 = {"a", "b", "c"};
vector<string> vec2 = move(vec1);  // Move instead of copy
```

Listing 9: Performance Optimization

2 Part 2: Dynamic Programming Complete Tutorial

2.1 Dynamic Programming Basic Concepts

Dynamic Programming (DP) is a method for solving complex problems by breaking them down into overlapping subproblems and building up solutions from these subproblems.

2.1.1 Characteristics of DP Problems:

- **Optimal substructure**: Optimal solution contains optimal solutions to subproblems
- **Overlapping subproblems**: Subproblems recur multiple times
- **No aftereffect**: Current state depends only on previous states, not future states

2.2 DP Five-Step Method

```
// 1. Define the meaning of dp array
// 2. Determine the state transition equation
// 3. Initialize base cases
// 4. Determine traversal order
// 5. Verify with examples
```

Listing 10: DP Five-Step Method

2.3 Classic Problems Explained

2.3.1 Fibonacci Sequence

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

// Method 1: Recursive (brute force) - Time complexity O(2^n)
int fib_recursive(int n) {
    if (n <= 1) return n;
    return fib_recursive(n-1) + fib_recursive(n-2);
}

// Method 2: Dynamic Programming - Time complexity O(n)
int fib_dp(int n) {
    if (n <= 1) return n;
    vector<int> dp(n+1);
    dp[0] = 0;
    dp[1] = 1;
    for (int i = 2; i <= n; i++) {
        dp[i] = dp[i-1] + dp[i-2];
    }
    return dp[n];
}

// Method 3: Optimized DP - Space complexity O(1)
int fib_optimized(int n) {
```

```
    if (n <= 1) return n;
    int prev1 = 0, prev2 = 1;
    for (int i = 2; i <= n; i++) {
        int current = prev1 + prev2;
        prev1 = prev2;
        prev2 = current;
    }
    return prev2;
}
```

Listing 11: Fibonacci Sequence

2.3.2 Climbing Stairs Problem

```
int climbStairs(int n) {
    if (n <= 2) return n;

    // dp[i] represents number of ways to reach i-th stair
    vector<int> dp(n+1);
    dp[1] = 1;  // 1 way to reach 1st stair
    dp[2] = 2;  // 2 ways to reach 2nd stair (1+1, 2)

    for (int i = 3; i <= n; i++) {
        // State transition: can climb 1 from i-1 or 2 from i-2
        dp[i] = dp[i-1] + dp[i-2];
    }

    return dp[n];
}
```

Listing 12: Climbing Stairs Problem

2.3.3 0-1 Knapsack Problem

```
// Problem: n items, knapsack capacity W, each item has weight and value
int knapsack_01(int W, vector<int>& weight, vector<int>& value) {
    int n = weight.size();
    // dp[i][w] represents max value with first i items and capacity w
    vector<vector<int>> dp(n+1, vector<int>(W+1, 0));

    for (int i = 1; i <= n; i++) {
        for (int w = 1; w <= W; w++) {
            if (weight[i-1] <= w) {
                // Choice: include or exclude current item
                dp[i][w] = max(
                    dp[i-1][w],  // Exclude
                    dp[i-1][w - weight[i-1]] + value[i-1]  // Include
                );
            } else {
                // Current item too heavy, cannot include
                dp[i][w] = dp[i-1][w];
            }
        }
    }
    return dp[n][W];
}
```

```cpp

// Space optimized version
int knapsack_01_optimized(int W, vector<int>& weight, vector<int>& value
    ) {
    int n = weight.size();
    vector<int> dp(W+1, 0);

    for (int i = 0; i < n; i++) {
        // Reverse traversal to avoid reusing items
        for (int w = W; w >= weight[i]; w--) {
            dp[w] = max(dp[w], dp[w - weight[i]] + value[i]);
        }
    }
    return dp[W];
}
```

Listing 13: 0-1 Knapsack Problem

2.4 Intermediate DP Problems

2.4.1 Longest Increasing Subsequence (LIS)

```cpp
int lengthOfLIS(vector<int>& nums) {
    int n = nums.size();
    if (n == 0) return 0;

    // dp[i] represents LIS ending at nums[i]
    vector<int> dp(n, 1);
    int max_len = 1;

    for (int i = 1; i < n; i++) {
        for (int j = 0; j < i; j++) {
            if (nums[i] > nums[j]) {
                dp[i] = max(dp[i], dp[j] + 1);
            }
        }
        max_len = max(max_len, dp[i]);
    }

    return max_len;
}
```

Listing 14: Longest Increasing Subsequence

2.4.2 Longest Common Subsequence (LCS)

```cpp
int longestCommonSubsequence(string text1, string text2) {
    int m = text1.length(), n = text2.length();
    // dp[i][j] represents LCS of text1[0..i-1] and text2[0..j-1]
    vector<vector<int>> dp(m+1, vector<int>(n+1, 0));

    for (int i = 1; i <= m; i++) {
        for (int j = 1; j <= n; j++) {
            if (text1[i-1] == text2[j-1]) {
                dp[i][j] = dp[i-1][j-1] + 1;
```

```
            } else {
                dp[i][j] = max(dp[i-1][j], dp[i][j-1]);
            }
        }
    }

    return dp[m][n];
}
```

Listing 15: Longest Common Subsequence

2.4.3 Edit Distance

```
int minDistance(string word1, string word2) {
    int m = word1.length(), n = word2.length();
    // dp[i][j] represents min operations to convert word1[0..i-1] to word2[0..j-1]
    vector<vector<int>> dp(m+1, vector<int>(n+1, 0));

    // Initialize
    for (int i = 0; i <= m; i++) dp[i][0] = i;  // Delete all characters
    for (int j = 0; j <= n; j++) dp[0][j] = j;  // Insert all characters

    for (int i = 1; i <= m; i++) {
        for (int j = 1; j <= n; j++) {
            if (word1[i-1] == word2[j-1]) {
                dp[i][j] = dp[i-1][j-1];  // Characters match, no operation
            } else {
                dp[i][j] = min({
                    dp[i-1][j] + 1,    // Delete word1[i-1]
                    dp[i][j-1] + 1,    // Insert word2[j-1]
                    dp[i-1][j-1] + 1   // Replace word1[i-1] with word2[j-1]
                });
            }
        }
    }

    return dp[m][n];
}
```

Listing 16: Edit Distance

2.5 Advanced DP Techniques

2.5.1 State Compression

```
// Traveling Salesman Problem (TSP) with state compression
int tsp(vector<vector<int>>& distance) {
    int n = distance.size();
    vector<vector<int>> dp(1 << n, vector<int>(n, INT_MAX));

    dp[1][0] = 0;  // Start from city 0

```

```
    for (int mask = 1; mask < (1 << n); mask++) {
        for (int i = 0; i < n; i++) {
            if (dp[mask][i] == INT_MAX) continue;

            for (int j = 0; j < n; j++) {
                if (!(mask & (1 << j))) {  // If city j not visited
                    int new_mask = mask | (1 << j);
                    dp[new_mask][j] = min(dp[new_mask][j],
                                         dp[mask][i] + distance[i][j]);
                }
            }
        }
    }

    // Return to starting city 0
    int result = INT_MAX;
    int full_mask = (1 << n) - 1;
    for (int i = 1; i < n; i++) {
        if (dp[full_mask][i] != INT_MAX && distance[i][0] != INT_MAX) {
            result = min(result, dp[full_mask][i] + distance[i][0]);
        }
    }

    return result;
}
```

Listing 17: State Compression Example

2.6 Practice Training Plan

Table 1: Dynamic Programming Practice Plan

Stage	Learning Content
Beginner Stage (1-2 weeks)	• Fibonacci Sequence • Climbing Stairs • Minimum Path Sum • House Robber • Coin Change
Intermediate Stage (2-3 weeks)	• Longest Increasing Subsequence • Longest Common Subsequence • Edit Distance • 0-1 Knapsack Problem • Unbounded Knapsack Problem
Advanced Stage (3-4 weeks)	• Stock Buying and Selling Series • Regular Expression Matching • Wildcard Matching • Partition Equal Subset Sum • Target Sum
Master Stage (Continuous practice)	• State Compression DP • Interval DP • Tree DP • Digit DP • Probability DP

3 Part 3: Vector Applications in Dynamic Programming

3.1 Using Vector as DP Array

```
// 1D DP array
vector<int> dp(n, 0);

// 2D DP array
vector<vector<int>> dp(m, vector<int>(n, 0));

// 3D DP array
vector<vector<vector<int>>> dp(x, vector<vector<int>>(y, vector<int>(z,
    0)));
```

Listing 18: Using Vector as DP Array

3.2 Vector Performance Optimization in DP

```
// Pre-allocate DP array size
vector<int> dp;
dp.reserve(n + 1);  // Pre-allocate space

// Use references to avoid copying
void solveDP(const vector<int>& input, vector<int>& dp) {
    // Use const reference to avoid copying input data
    // Directly modify dp array
}

// Move semantics optimization
vector<int> createDP(int n) {
    vector<int> dp(n);
    // Initialize dp array
    return dp;  // Use move semantics to return
}
```

Listing 19: Vector Performance Optimization

3.3 Comprehensive Example: Solving DP with Vector

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

int uniquePaths(int m, int n) {
    // Use 2D vector as DP array
    vector<vector<int>> dp(m, vector<int>(n, 0));

    // Initialize first row and column
    for (int i = 0; i < m; i++) dp[i][0] = 1;
    for (int j = 0; j < n; j++) dp[0][j] = 1;

    // Dynamic programming
    for (int i = 1; i < m; i++) {
        for (int j = 1; j < n; j++) {
            dp[i][j] = dp[i-1][j] + dp[i][j-1];
        }
    }

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

int main() {
    int paths = uniquePaths(3, 7);
    cout << "Number of unique paths: " << paths << endl;
    return 0;
}
```

Listing 20: Comprehensive Example

Learning Recommendations

Vector Learning Recommendations:

- **Understand underlying principles**: Learn about vector's memory management and resizing mechanism
- **Master common operations**: Become proficient with adding, removing, accessing operations
- **Learn performance optimization**: Use reserve, emplace_back and other optimization techniques appropriately
- **Multi-dimensional applications**: Master the use of 2D and 3D vectors

Dynamic Programming Learning Recommendations:

- **Understanding over memorization**: Understand the meaning of state transition equations
- **Start with simple problems**: Master basic problems before tackling difficult ones
- **Draw diagrams for analysis**: Use charts to help understand state transition processes
- **Persistent practice**: Solve at least 1-2 DP problems daily
- **Summarize and generalize**: Organize solution patterns for similar problems

Appendix: Useful Resources

Online Practice Platforms:

- LeetCode
- HackerRank
- Codeforces

Recommended Books:

- Introduction to Algorithms
- Competitive Programming
- Cracking the Coding Interview