Dynamic Programming

optimal: 最佳的

Memoization(备忘录)

Top-down
备忘录技术（Memoization）是动态规划（Dynamic Programming）中的一种优化方法，用于提高递归算法的效率。其核心思想是将已经计算过的子问题的结果存储起来，以便在以后需要时可以直接使用，而不是重新计算。这种方法特别适用于具有重叠子问题性质的问题，例如Fibonacci数列计算、背包问题等。

备忘录技术的基本步骤

1. 识别子问题：确定问题可以分解成哪些子问题，并且这些子问题是否重复计算。
2. 存储结果：使用一个数据结构（如数组、字典、哈希表等）来存储已经计算过的子问题的结果。
3. 检查结果：在计算一个子问题之前，首先检查存储结构中是否已经有该子问题的结果。如果有，直接返回结果；如果没有，则计算该子问题并存储结果。

Top-down Fibonacci

原本的斐波那契：时间复杂度为T(n^2)

```
#include <iostream>

long long fibonacci(int n) {
```

```
    if (n <= 1) {
        return n;
    }
    return fibonacci(n - 1) + fibonacci(n - 2);
}

int main() {
    int n;
    std::cout << "输入你想计算的Fibonacci数列的项数: ";
    std::cin >> n;
    std::cout << "Fibonacci(" << n << ") = " << fibonacci(n) <<
std::endl;
    return 0;
}
```

使用备忘录后：时间复杂度为T(n)

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

std::vector<long long> memo;

long long fibonacci(int n) {
    if (memo[n] != -1) {
        return memo[n];
    }
    if (n <= 1) {
        return n;
    }
    memo[n] = fibonacci(n - 1) + fibonacci(n - 2);
    return memo[n];
}

int main() {
    int n;
    std::cout << "输入你想计算的Fibonacci数列的项数: ";
    std::cin >> n;

    // 初始化memo数组，并将所有元素设置为-1，表示未计算
    memo = std::vector<long long>(n + 1, -1);
```

```
    std::cout << "Fibonacci(" << n << ") = " << fibonacci(n) <<
std::endl;

    return 0;
}
```

备忘录技术的优点

1. 提高效率：通过避免重复计算显著减少了时间复杂度。
2. 简化代码：使用备忘录技术可以使递归代码更加简洁和直观。
3. 空间换时间：通过使用额外的存储空间（如数组或哈希表），换取时间效率的提升。

Tabulation(表格法)

Down-top
表格法通过构建一个表格来保存子问题的解，并从最简单的子问题开始逐步解决更复杂的问题。这种方法避免了递归调用和函数调用的开销，通常效率更高。

表格法的基本步骤

1. 定义表格：创建一个数组或矩阵来存储子问题的解。
2. 初始化表格：根据问题的基本情况初始化表格的初始值。
3. 填充表格：按照一定的顺序填充表格，从基本情况开始，逐步解决更复杂的子问题。
4. 读取结果：最终结果通常保存在表格的最后一个单元格中。

示例：使用表格法计算Fibonacci数列

以下是使用C++实现的表格法计算Fibonacci数列的代码：

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

long long fibonacci(int n) {
    if (n <= 1) {
        return n;
    }

    // 创建一个数组来存储Fibonacci数
    // 初始化为0
    // 函数内创建表格
    std::vector<long long> table(n + 1, 0);

    // 初始化基本情况
    table[0] = 0;
    table[1] = 1;

    // 填充表格
    for (int i = 2; i <= n; ++i) {
        table[i] = table[i - 1] + table[i - 2];
    }

    // 最终结果
    return table[n];
}

int main() {
    int n;
    std::cout << "输入你想计算的Fibonacci数列的项数: ";
    std::cin >> n;

    std::cout << "Fibonacci(" << n << ") = " << fibonacci(n) <<
std::endl;

    return 0;
}
```

表格法的优点

1. 避免递归开销：通过迭代的方式解决问题，避免了递归调用的开销。
2. 空间效率：虽然表格法需要额外的存储空间，但可以通过只保留必要的中间结果来优化空间使用。
3. 时间效率：通过一次填表，避免了重复计算，时间复杂度通常较低。

对比

- 备忘录技术（Memoization）：自顶向下，通过递归和缓存子问题结果来避免重复计算。
- 表格法（Tabulation）：自底向上，通过迭代构建表格来存储子问题结果。

两种方法的选择取决于具体问题和实现的便利性，但在大多数情况下，表格法通常更为高效。

Assembly Line Scheduling Problem

装配线调度问题（Assembly Line Scheduling Problem）是一种经典的动态规划问题，通常用来优化产品在不同生产线上的制造时间。这个问题的目标是找到从两条或多条装配线中选择的最小生产时间路径。

以下是装配线调度问题的详细描述以及求解方法：

问题描述

考虑两条装配线，每条线有 `n` 个工作站。每个工作站都有一定的加工时间。工件可以在某些工作站之间切换，切换时间取决于所切换的站点。

输入：

- `a[i][j]`：第 `i` 条装配线第 `j` 个工作站的处理时间。
- `t[i][j]`：从第 `i` 条装配线第 `j` 个工作站切换到另一条装配线第 `j+1` 个工作站的时间。
- `e[i]`：第 `i` 条装配线的进入时间。
- `x[i]`：第 `i` 条装配线的退出时间。

输出：

- 完成所有工序所需的最短时间。

动态规划解法

我们可以使用动态规划来解决这个问题。定义 `T1[j]` 和 `T2[j]` 分别表示在 `j` 工作站结束后，在第一条和第二条装配线上的最小时间。

递推关系：

```
T1[j] = min(T1[j-1] + a[1][j], T2[j-1] + t[2][j-1] + a[1][j])
T2[j] = min(T2[j-1] + a[2][j], T1[j-1] + t[1][j-1] + a[2][j])
```

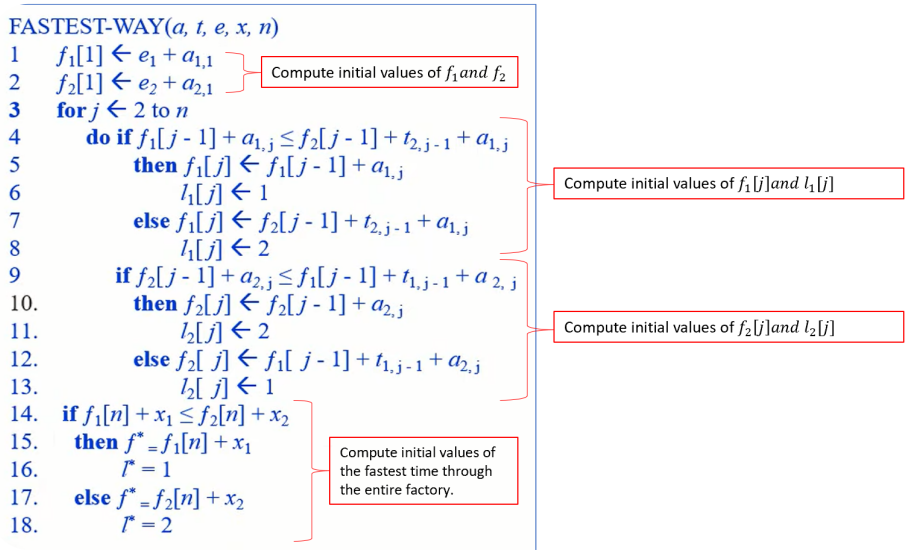
边界条件：

```
T1[0] = e[1] + a[1][0]
T2[0] = e[2] + a[2][0]
```

最终答案：

```
F = min(T1[n-1] + x[1], T2[n-1] + x[2])
```

伪代码：



代码实现

以下是基于上述公式的C++实现：

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

// Function to find the minimum time to complete the job on
assembly line
int assemblyLineScheduling(vector<vector<int>>& a,
vector<vector<int>>& t, vector<int>& e, vector<int>& x, int n) {
    // Time taken to leave station j on both lines
    vector<int> T1(n), T2(n);

    // Initializing the time taken for the first station
    T1[0] = e[0] + a[0][0];
    T2[0] = e[1] + a[1][0];

    // Filling up the DP tables T1[] and T2[]
    for (int j = 1; j < n; ++j) {
        T1[j] = min(T1[j - 1] + a[0][j], T2[j - 1] + t[1][j - 1]
+ a[0][j]);
        T2[j] = min(T2[j - 1] + a[1][j], T1[j - 1] + t[0][j - 1]
+ a[1][j]);
```

```
    }

    // Returning the minimum time required to complete the job
    return min(T1[n - 1] + x[0], T2[n - 1] + x[1]);
}

int main() {
    // Number of stations
    int n = 4;

    // Processing time on both lines
    vector<vector<int>> a = {{4, 5, 3, 2}, {2, 10, 1, 4}};

    // Transition time between the lines
    vector<vector<int>> t = {{0, 7, 4, 5}, {0, 9, 2, 8}};

    // Entry time for both lines
    vector<int> e = {10, 12};

    // Exit time for both lines
    vector<int> x = {18, 7};

    cout << "The minimum time to complete the job is " <<
assemblyLineScheduling(a, t, e, x, n) << endl;

    return 0;
}
```

解释

- `a` 表示两条装配线在各个站点的处理时间。
- `t` 表示从一条线的当前站点到另一条线的下一个站点的切换时间。
- `e` 和 `x` 分别表示进入和退出装配线的时间。
- `T1` 和 `T2` 用来存储每个站点在每条装配线上的最小时间。

该算法通过动态规划求解，保证了最小时间复杂度为 `O(n)`。

暴力解法思路

暴力解法会尝试每一种可能的装配顺序，然后计算总的时间，选择其中最小的一个。尽管这种方法可以找到最优解，但它的时间复杂度非常高，是指数级的 `O(2^n)`，其中 `n` 是工作站的数量。这种方法对于小规模问题是可行的，但对于大规模问题则不可行。

实现暴力解法

以下是用 C++ 实现的暴力解法代码：

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

int minAssemblyTime(vector<vector<int>>& a, vector<vector<int>>&
t, vector<int>& e, vector<int>& x, int n, int line, int station)
{
    if (station == n - 1) {
        return (line == 0 ? x[0] : x[1]) + a[line][station];
    }

    int stay = a[line][station] + minAssemblyTime(a, t, e, x, n,
line, station + 1);
    int switchLine = a[line][station] + t[line][station] +
minAssemblyTime(a, t, e, x, n, 1 - line, station + 1);

    return min(stay, switchLine);
}

int main() {
    int n = 4;
    vector<vector<int>> a = {{4, 5, 3, 2}, {2, 10, 1, 4}};
    vector<vector<int>> t = {{0, 7, 4, 5}, {0, 9, 2, 8}};
    vector<int> e = {10, 12};
    vector<int> x = {18, 7};

    int minTimeLine1 = e[0] + minAssemblyTime(a, t, e, x, n, 0,
0);
```

```
    int minTimeLine2 = e[1] + minAssemblyTime(a, t, e, x, n, 1,
0);

    cout << "The minimum time to complete the job is " <<
min(minTimeLine1, minTimeLine2) << endl;

    return 0;
}
```

解释

1. 递归函数：`minAssemblyTime` 用于计算从当前装配线和站点开始到结束的最小时间。
    - 如果当前站点是最后一个站点，直接返回当前站点的处理时间加上退出时间。
    - 否则，计算继续在当前装配线和切换到另一条装配线的总时间，返回其中较小的一个。
2. 主函数：在主函数中，分别计算从两条装配线开始的总时间，并选择较小的一个作为最终结果。

这种方法尝试了所有可能的路径，因此可以找到最优解，但由于其指数级的时间复杂度，只适合于小规模的问题。在实际应用中，通常会使用前述的动态规划方法来高效地解决该问题。

时间复杂度对比

The assembly line scheduling uses only one for loop, as shown in the Pseudocode. Thus, the time complexity is T(n)=O(n).

Using the brute force provided us with T(n)=O(2^n), but using Dynamic programming to solve the Assembly line scheduling reduce it to T(n)=O(n).

Longest Common Subsequence(LCS)

最长公共子序列问题（Longest Common Subsequence，简称 LCS）是计算机科学中的一个经典问题。给定两个序列，LCS 问题的目标是找到它们的最长公共子序列。子序列是从一个序列中删除一些（可以是零个）元素后得到的序列，而不改变其余元素的相对顺序。

举个例子，考虑两个序列：

- 序列1：`ABCBDAB`
- 序列2：`BDCAB`

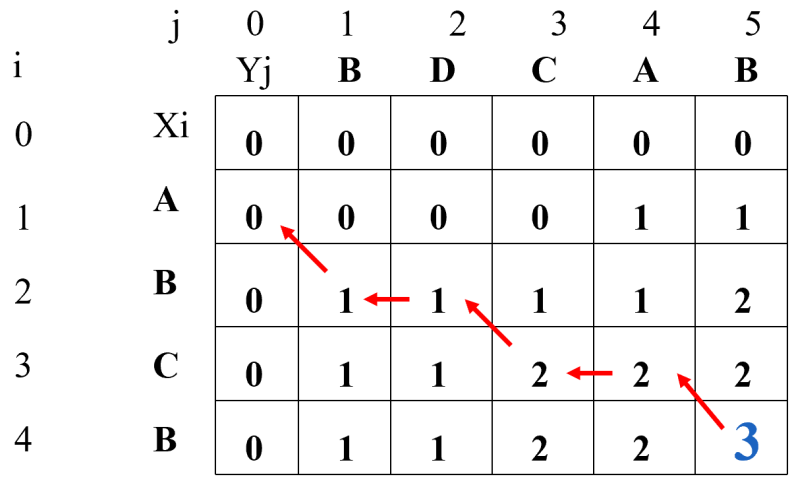
它们的最长公共子序列是 `BCAB`，长度为 4。

LCS 问题的动态规划解决方案

LCS 问题可以通过动态规划方法来高效地解决。我们定义一个二维数组 `dp`，其中 `dp[i][j]` 表示序列1的前 `i` 个字符和序列2的前 `j` 个字符的最长公共子序列的长度。

动态规划的步骤

1. 初始化：当其中一个序列为空时，LCS 长度为 0。因此，对于所有的 `i`，`dp[i][0] = 0`，对于所有的 `j`，`dp[0][j] = 0`。



2. 状态转移：
   - 如果序列1的第 `i` 个字符和序列2的第 `j` 个字符相同，那么 `dp[i][j] = dp[i-1][j-1] + 1`。
   - 如果不相同，那么 `dp[i][j] = max(dp[i-1][j], dp[i][j-1])`。
3. 计算结果：最终，`dp[m][n]` 就是序列1的长度为 `m` 和序列2的长度为 `n` 的最长公共子序列的长度。

伪代码示例

这里是非递归的方法，即动态规划

```
def lcs(X, Y):
    m = len(X)
    n = len(Y)
    dp = [[0] * (n + 1) for _ in range(m + 1)]

    for i in range(1, m + 1):
        for j in range(1, n + 1):
            if X[i - 1] == Y[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]

# 示例使用
```

```
X = "ABCBDAB"
Y = "BDCAB"
print(f"LCS长度是: {lcs(X, Y)}")  # 输出 LCS长度是: 4
```

这个算法的时间复杂度是 O(m * n)，空间复杂度是min{n,m}，其中 `m` 和 `n` 分别是两个序列的长度。

C++实现

为了减少递归带来的重复计算，我们可以使用动态规划表来存储中间结果，避免在计算过程中多次重复同样的子问题。以下是一个完整的 C++ 实现，其中动态规划表用于减少重复的递归调用：

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

// 返回两个字符串 X 和 Y 的最长公共子序列的长度
int lcs(const string &X, const string &Y) {
    int m = X.length();
    int n = Y.length();

    // 创建一个二维数组 dp，用于存储子问题的解
    vector<vector<int>> dp(m + 1, vector<int>(n + 1, 0));

    // 填充 dp 表格
    for (int i = 1; i <= m; ++i) {
        for (int j = 1; j <= n; ++j) {
            if (X[i - 1] == Y[j - 1]) {
                dp[i][j] = dp[i - 1][j - 1] + 1;
            } else {
                dp[i][j] = max(dp[i - 1][j], dp[i][j - 1]);
            }
        }
    }

    // dp[m][n] 是 X 和 Y 的最长公共子序列的长度
    return dp[m][n];
}
```

```cpp
// 递归构造 LCS
string constructLCS(const string &X, const string &Y, const
vector<vector<int>> &dp) {
    int i = X.length();
    int j = Y.length();
    string lcs;

    while (i > 0 && j > 0) {
        if (X[i - 1] == Y[j - 1]) {
            lcs.push_back(X[i - 1]);
            --i;
            --j;
        } else if (dp[i - 1][j] > dp[i][j - 1]) {
            --i;
        } else {
            --j;
        }
    }

    // 反转字符串
    reverse(lcs.begin(), lcs.end());
    return lcs;
}

int main() {
    string X = "ABCBDAB";
    string Y = "BDCAB";

    int m = X.length();
    int n = Y.length();

    // 创建一个二维数组 dp，用于存储子问题的解
    vector<vector<int>> dp(m + 1, vector<int>(n + 1, 0));

    // 填充 dp 表格
    for (int i = 1; i <= m; ++i) {
        for (int j = 1; j <= n; ++j) {
            if (X[i - 1] == Y[j - 1]) {
                dp[i][j] = dp[i - 1][j - 1] + 1;
            } else {
                dp[i][j] = max(dp[i - 1][j], dp[i][j - 1]);
```

```
            }
        }
    }

    cout << "LCS 长度是: " << dp[m][n] << endl;
    cout << "LCS 是: " << constructLCS(X, Y, dp) << endl;

    return 0;
}
```

代码说明：

1. `lcs` 函数：使用动态规划填充 `dp` 表，并返回两个字符串的最长公共子序列的长度。
2. `constructLCS` 函数：根据 `dp` 表递归地构造最长公共子序列，避免了重复的计算。
3. `main` 函数：初始化两个字符串 `X` 和 `Y`，填充 `dp` 表，输出最长公共子序列的长度和具体的子序列。

这样，通过利用动态规划表存储中间结果，我们减少了重复的递归计算，显著提高了效率。

注意事项

在构造最长公共子序列（LCS）的过程中，选择较大的值是为了确保选择的是最长子序列的路径。动态规划表 `dp` 存储的是子问题的解，`dp[i][j]` 表示字符串 `X[0..i-1]` 和 `Y[0..j-1]` 的最长公共子序列的长度。

动态规划表的填充过程

1. 如果 `X[i-1] == Y[j-1]`，则 `dp[i][j] = dp[i-1][j-1] + 1`。
2. 如果 `X[i-1] != Y[j-1]`，则 `dp[i][j] = max(dp[i-1][j], dp[i][j-1])`。

为什么选择较大的值

当 `X[i-1] != Y[j-1]` 时，LCS 要么包含 `X[i-1]`，要么包含 `Y[j-1]`，但不能同时包含这两个字符。因此，需要选择使 LCS 最长的那个子问题的解，即 `dp[i-1][j]` 和 `dp[i][j-1]` 中较大的那个值。

代码修正

以下是修正后的代码，保证在 `constructLCS` 函数中选择较大的值：

```
// 递归构造 LCS
string constructLCS(const string &X, const string &Y, const
vector<vector<int>> &dp) {
    string lcs;
    int i = X.size();
    int j = Y.size();

    while (i > 0 && j > 0) {
        if (X[i - 1] == Y[j - 1]) {
            lcs.push_back(X[i - 1]);
            --i;
            --j;
        } else if (dp[i - 1][j] > dp[i][j - 1]) { //要选择较大的
值
            --i;
        } else {
            --j;
        }
    }

    reverse(lcs.begin(), lcs.end());
    return lcs;
}
```

解释

1. 填充动态规划表 `dp`：
    - `dp[i][j]` 的值表示 `X[0..i-1]` 和 `Y[0..j-1]` 的 LCS 长度。

- `dp[i][j] = dp[i-1][j-1] + 1` 如果 `X[i-1] == Y[j-1]`，否则 `dp[i][j] = max(dp[i-1][j], dp[i][j-1])`。

2. 构造 LCS：
   - 从 `dp` 表的右下角开始，如果 `X[i-1] == Y[j-1]`，则将该字符添加到 LCS 中并移动到 `dp[i-1][j-1]`。
   - 否则，选择较大的值移动：如果 `dp[i-1][j] > dp[i][j-1]`，则移动到 `dp[i-1][j]`，否则移动到 `dp[i][j-1]`。

这样，构造出来的 LCS 是正确的最长公共子序列。

The 0-1 Knapsack Problem

问题描述

0-1 背包问题是经典的优化问题，广泛应用于计算机科学和运筹学领域。问题的描述如下：

给定一组物品，每个物品都有一个重量和一个价值，我们希望从中选择若干个物品放入一个容量固定的背包中，使得背包内物品的总重量不超过背包的容量，并且背包内物品的总价值最大。

具体地，假设有 n 个物品，每个物品 i 有一个重量 w_i 和一个价值 v_i，背包的容量为 W。0-1 背包问题要求找出一个选择方案，使得在满足总重量不超过 W 的前提下，总价值 V 最大。

数学公式可以表述为：

$$\max \sum_{i=1}^{n} v_i x_i$$

$$\text{subject to } \sum_{i=1}^{n} w_i x_i \leq W$$

$$x_i \in \{0, 1\}$$

其中，x_i 是决策变量，表示物品 i 是否被选择。如果选择物品 i，则 $x_i = 1$；否则 $x_i = 0$。

求解0-1背包问题常用的方法有动态规划、分支限界法、回溯法等。其中，动态规划法是最常用的一种，它通过构建一个二维数组来存储子问题的解，从而避免重复计算。

以下是动态规划求解0-1背包问题的基本步骤：

1. 创建一个二维数组 (dp)，其中 (dp[i][j]) 表示前 (i) 个物品在容量为 (j) 的背包中的最大价值。
2. 初始化边界条件：当没有物品可选择或者背包容量为 0 时，最大价值为 0，即 (dp[0][j] = 0) 和 (dp[i][0] = 0)。
3. 填充动态规划表格：
   - 如果当前物品 (i) 不放入背包，则 (dp[i][j] = dp[i-1][j])；
   - 如果当前物品 (i) 放入背包，则 (dp[i][j] = \max(dp[i-1][j], dp[i-1][j-w_i] + v_i))。
4. 最终结果保存在 (dp[n][W]) 中。

下面是一个动态规划算法的伪代码示例：

```
function knapsack(values, weights, W):
    n = len(values)
    dp = array of size (n+1) x (W+1) initialized to 0

    for i from 1 to n:
        for j from 1 to W:
            if weights[i-1] <= j:
                dp[i][j] = max(dp[i-1][j], dp[i-1][j-weights[i-
1]] + values[i-1])
            else:
                dp[i][j] = dp[i-1][j]

    return dp[n][W]
```

这个伪代码中，`values` 和 `weights` 分别是物品的价值和重量列表，`W` 是背包的容量。最终返回的结果是最大价值。

代码实现

下面是用C++实现0-1背包问题的代码，带有详细的注释：

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

// 函数声明
int knapsack(const std::vector<int>& values, const
std::vector<int>& weights, int W);

int main() {
    // 物品的价值
    std::vector<int> values = {60, 100, 120};
    // 物品的重量
    std::vector<int> weights = {10, 20, 30};
    // 背包的最大容量
    int W = 50;

    // 计算背包能装下的最大价值
    int max_value = knapsack(values, weights, W);

    // 输出结果
    std::cout << "最大价值: " << max_value << std::endl;

    return 0;
}

// knapsack函数实现
int knapsack(const std::vector<int>& values, const
std::vector<int>& weights, int W) {
    // 物品的数量
    int n = values.size();

    // 创建二维数组dp，dp[i][j]表示前i个物品在容量为j的背包中的最大
价值
```

```
    std::vector<std::vector<int>> dp(n + 1, std::vector<int>(W +
1, 0));

    // 填充dp数组
    for (int i = 1; i <= n; ++i) {
        for (int j = 1; j <= W; ++j) {
                // 若第i个物品的重量小于等于背包重量
            if (weights[i - 1] <= j) { // 重量index是从0开始的
                // 如果当前物品可以放入背包，则考虑放入或不放入的最
大价值
                dp[i][j] = std::max(dp[i - 1][j], dp[i - 1][j -
weights[i - 1]] + values[i - 1]);
            } else {
                // 当前物品不能放入背包，最大价值等于不放入当前物品
的情况
                dp[i][j] = dp[i - 1][j];
            }
        }
    }

    // 返回背包的最大价值
    return dp[n][W];
}
```

详细注释

1. 函数声明和主函数：
    - `knapsack`函数声明，接受三个参数：物品的价值列表、物品的重量列表和背包的最大容量。
    - 在`main`函数中，定义物品的价值和重量列表，以及背包的最大容量`W`。
    - 调用`knapsack`函数并输出结果。
2. knapsack函数实现：
    - 获取物品的数量`n`。

- 创建一个二维数组`dp`，大小为`(n+1) x (W+1)`，并初始化为0。`dp[i][j]`表示前`i`个物品在容量为`j`的背包中的最大价值。
- 使用两个嵌套的`for`循环填充`dp`数组：
  - 外层循环遍历物品`i`。
  - 内层循环遍历背包容量`j`。
  - 如果当前物品的重量小于等于当前背包容量`j`，则考虑两种情况：放入当前物品和不放入当前物品，并取两者的最大值。
  - 如果当前物品的重量大于当前背包容量`j`，则不能放入当前物品，最大价值等于不放入当前物品的情况。
- 返回`dp[n][W]`，即背包的最大价值。

这个实现使用了动态规划的方法，时间复杂度为`O(nW)`，空间复杂度为`O(nW)`。

这段代码 `dp[i - 1][j - weights[i - 1]] + values[i - 1]` 出现在动态规划解法中，表示当前考虑将第 (i) 个物品放入背包时的状态转移。为了更好地理解它的含义，让我们逐步分析：

1. `dp[i - 1][j - weights[i - 1]]`：
   - `dp[i - 1][j - weights[i - 1]]` 表示在容量为 (j - weights[i - 1]) 的情况下，考虑前 (i - 1) 个物品能达到的最大价值。
   - 也就是说，这是不包括第 (i) 个物品时，在减少容量后所能得到的最大价值。
2. `+ values[i - 1]`：
   - `values[i - 1]` 是第 (i) 个物品的价值（注意数组索引从 0 开始，因此第 (i) 个物品在数组中的索引是 (i - 1)）。
   - 将当前第 (i) 个物品的价值加上，即如果选择这个物品，所能获得的总价值。

3. 综合含义:
    - 组合起来，这段代码的意思是：如果将第 (i) 个物品放入背包，那么在背包容量减少 (weights[i - 1]) 后（即剩余容量为 (j - weights[i - 1])），前 (i - 1) 个物品所能达到的最大价值加上当前第 (i) 个物品的价值。
    - 换句话说，这是选择放入第 (i) 个物品时，所能获得的总最大价值。

为了更直观地理解，我们来看一个具体的例子：

假设有如下物品：

- 物品1：重量=10，价值=60
- 物品2：重量=20，价值=100
- 物品3：重量=30，价值=120

背包的容量 (W = 50)。

在考虑是否将物品3放入背包时：

- 如果背包容量为 50，放入物品3后，剩余的容量为 (50 - 30 = 20)。
- 那么 `dp[2][20]` 表示在剩余容量为 20 的情况下，前两个物品能达到的最大价值。
- 加上当前物品3的价值，即 `dp[2][20] + 120`。

总结：
这段代码 `dp[i - 1][j - weights[i - 1]] + values[i - 1]` 计算了在当前背包容量 (j) 下，如果选择放入第 (i) 个物品，所能获得的最大总价值。这是动态规划状态转移的关键步骤之一，用于更新 `dp[i][j]`。

Folyd Warshall Algorithm

弗洛伊德算法，也称为弗洛伊德-沃肖尔算法，是一种用于在加权图中找到所有顶点对之间最短路径的图论算法。它特别适用于可能有负权重但没有负权重环的图。以下是该算法的简要概述：

弗洛伊德-沃肖尔算法

1. 初始化：
   - 创建一个矩阵 `dist`，其中 `dist[i][j]` 表示从顶点 `i` 到顶点 `j` 的最短距离。
   - 将 `dist[i][j]` 初始化为 `i` 和 `j` 之间边的权重。如果没有边，则设置为无穷大 (`inf`)。对于所有的 `i`，设置 `dist[i][i]` 为 0。
2. 算法：
   - 使用三个嵌套循环来更新距离矩阵：

```
for k in range(n):  # n 是顶点的数量
    for i in range(n):
        for j in range(n):
            if dist[i][j] > dist[i][k] + dist[k][j]:
                dist[i][j] = dist[i][k] + dist[k][j]
```

   - 这里，`k` 是一个中间顶点。算法检查从 `i` 到 `j` 通过 `k` 的路径是否比直接从 `i` 到 `j` 的路径更短。
3. 结果：
   - 运行算法后，`dist[i][j]` 将包含从顶点 `i` 到顶点 `j` 的最短距离。

示例

假设你有一个包含4个顶点的图，且其邻接矩阵如下：

```
  0   3   inf  7
  8   0   2    inf
```

```
   5   inf 0    1
   2   inf inf  0
```

运行弗洛伊德-沃肖尔算法后，你会得到：

```
   0   3   5    7
   8   0   2    3
   5   6   0    1
   2   5   7    0
```

这表示所有顶点对之间的最短路径距离。

优点和缺点

优点:

- 可以处理带有负权重的图。
- 简单且易于实现。

缺点:

- 时间复杂度高：O(n^3)，对于大型图不太高效。

弗洛伊德算法是计算机科学中的基础算法，被广泛应用于网络路由协议和各种需要找到所有节点对之间最短路径的应用中。

C++实现

下面是添加了详细注释的弗洛伊德-沃肖尔算法 C++ 实现代码：

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

using namespace std;
```

```cpp
// 定义一个表示无穷大的常量
const int INF = numeric_limits<int>::max();

int main()
{
    // 初始化带权有向图的距离矩阵
    vector<vector<int>> A = {
        {0, 5, -3, INF},    // 从节点 0 到其他节点的初始距离
        {9, 0, INF, 3},     // 从节点 1 到其他节点的初始距离
        {4, 5, 0, INF},     // 从节点 2 到其他节点的初始距离
        {INF, INF, 1, 0}    // 从节点 3 到其他节点的初始距离
    };

    // 获取图中节点的数量
    int n = A.size();

    // 弗洛伊德-沃肖尔算法的三重循环
    // 外层循环 k 表示中间节点
    for (int k = 0; k < n; k++) {
        // 中层循环 i 表示起点
        for (int i = 0; i < n; i++) {
            // 内层循环 j 表示终点
            for (int j = 0; j < n; j++) {
                // 检查是否存在通过中间节点 k 的路径
                // 防止溢出：仅在 A[i][k] 和 A[k][j] 都不是无穷大时更新距离
                if (A[i][k] < INF && A[k][j] < INF) {
                    // 更新 A[i][j] 为经过 k 的更短路径
                    A[i][j] = min(A[i][j], A[i][k] + A[k][j]);
                }
            }
        }
    }

    // 输出所有节点对之间的最短路径
    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < n; ++j) {
            // 如果 A[i][j] 仍然是无穷大，说明 i 和 j 之间没有路径
            if (A[i][j] == INF) {
                cout << "INF ";
            } else {
```

```
                // 否则输出最短路径长度
                cout << A[i][j] << " ";
            }
        }
        cout << endl; // 换行，输出下一行结果
    }

    return 0;
}
```

代码说明：

1. 初始化图的距离矩阵:
   - `vector<vector<int>> A` 定义了一个带权有向图的距离矩阵，其中 `INF` 表示节点之间没有直接路径。
   - 例如，`A[0][1] = 5` 表示从节点 0 到节点 1 的距离为 5；`A[0][3] = INF` 表示从节点 0 到节点 3 没有直接路径。
2. 获取节点数量:
   - `int n = A.size();` 获取图中节点的数量。
3. 弗洛伊德-沃肖尔算法的实现:
   - 外层循环 `k` 遍历所有可能的中间节点。
   - 中层循环 `i` 遍历所有可能的起点。
   - 内层循环 `j` 遍历所有可能的终点。
   - `if (A[i][k] < INF && A[k][j] < INF)` 检查是否存在通过中间节点 `k` 的路径，以防止溢出。
   - `A[i][j] = min(A[i][j], A[i][k] + A[k][j]);` 更新从节点 `i` 到节点 `j` 的最短路径长度。
4. 输出结果:

- 遍历距离矩阵 `A`，如果 `A[i][j] == INF`，则输出 `INF`，表示节点 `i` 和节点 `j` 之间没有路径。
- 否则输出从节点 `i` 到节点 `j` 的最短路径长度。