HW03 动态规划

01 背景知识

- 本次作业主要考察动态规划的相关知识。动态规划的基本思想是将待求解问题分解成若干个问题的子问题，先求解子问题，然后从这些子问题的解得到原问题的解。与分治法不同的是，动态规划分解得到的子问题不是相互独立的，若用分治法则会产生很多的重复计算，因此在动态规划中采用一个表来记录所有已求解的子问题的答案。
- 动态规划的设计步骤：①找出最优解的性质②递归地定义最优值，写出动态规划方程③自底向上地计算最优值④根据最优值计算最优解
- 通常含有下面情况的一般都可以使用动态规划来解决：求最优解问题(最大值和最小值);求可行性(True 或 False);求方案总数;数据结构不可排序;算法不可使用交换.

02 LeetCode

题目描述：
给你一个字符串 `s` 和一个字符串列表 `wordDict` 作为字典。如果可以利用字典中出现的一个或多个单词拼接出 `s` 则返回 `true`。
注意：不要求字典中出现的单词全部都使用，并且字典中的单词可以重复使用。
题目分析与算法设计：

- 这道题要求判断一个字符串是否可以用字典中的字符串片段拼接而成，最简单的思路是使用暴力破解，通过索引遍历字符串，测试本次遍历到的是不是wordDict中的字符串，如果是则进行裁剪，判断剩余部分，直至结束。
- 自然而然可以想到第一种方法，使用递归进行字符串的裁剪，但是递归过程中会出现很多重复的子问题，如果每次都再重新判断一遍就会很消耗时间，因此想到使用动态规划算法，动态规划需要判断是否有最优子结构以及子问题是否相互独立。
- 同时，为了便于判断子串是否在字典中,使用了STL库中的unordered_set集合容器，它基于哈希表实现，可以在平均情况下以常量时间O(1)进行查找操作，比for循环效率要高。

```
string s;
unordered_set<string> wordDict;
```

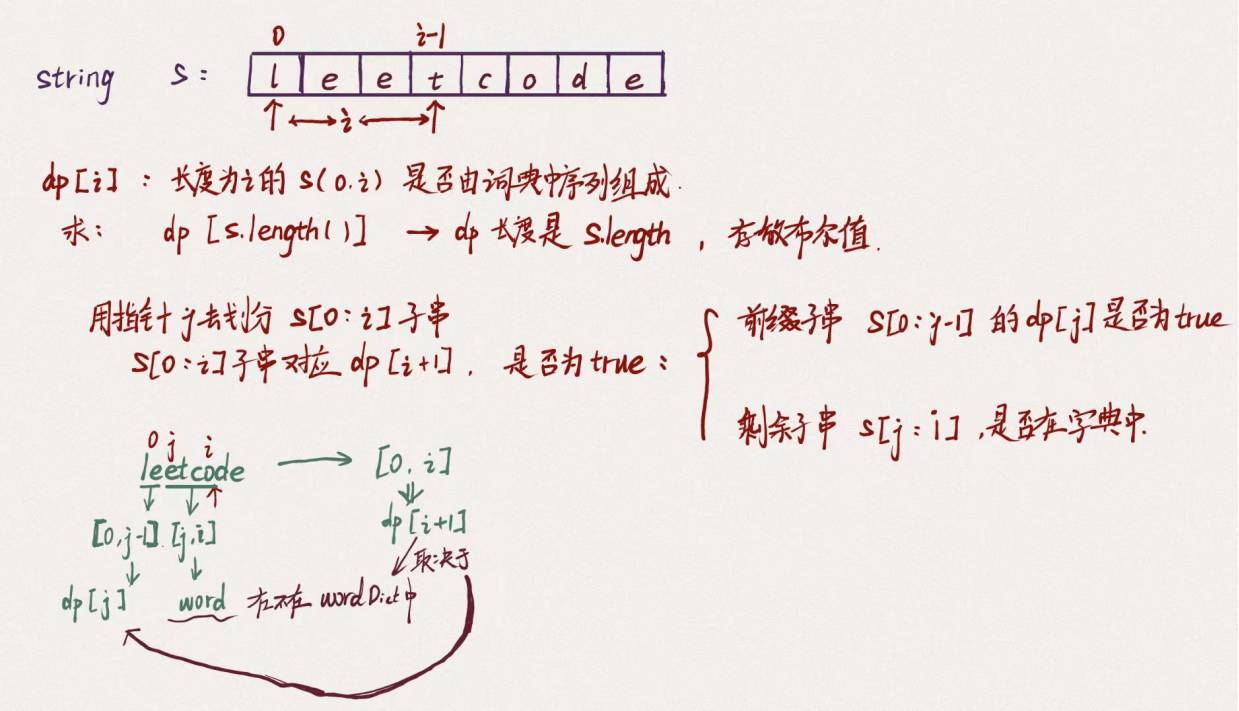
递归思路： 递归算法需要设计递归方程和边界条件。

- 初始情况是如果输入字符串为空辨明已经成功拆分好，返回`true`；
- 终止条件有二：①当字符串的所有前缀`prefix`都在字典中，且后缀(递归调用表示)也满足可以被拆分为字典中的单词序列时，递归调用结束，返回`true` ②如果出现无法继续拆分的情况，即字符串的某个前缀不在字典中，或者剩余的后缀无法继续拆分为字典中的单词序列，那么递归调用结束，返回`false`
- 递归过程：对于字符串`s`，我们可以调用递归函数，将其不断分割为若干子串，比方前缀`prefix`和后缀，并检查前缀是否存在于字典中，并对后缀进行递归调用，如果所有的子串都存在于字典中，则说明字符串`s`可以被成功拆分为字典中的单词序列。
- 递归方程：T(n) = O(n) + T(n-1) + O(n-1) + T(n-2) + O(n-2) + ... + T(1) + O(1) = $O(n^2)$
- 时间复杂度：在每一次递归调用中，对字符串进行了多次子串分割和字典查找操作。假设字符串长度为 n，最坏情况下需要考虑每个前缀子串，并进行字典查找操作。因此，时间复杂度可以表示为 $O(n^2)$，其中 n 是字符串的长度。
- 空间复杂度：递归调用的过程中会涉及递归栈的空间，最坏情况下会达到字符串长度，因此为$O(n)$
  核心代码：

```
/**
 * \brief  递归方法判断字符串是否可以被拆分为字典中的单词
 * \param  s: 待判断的字符串
 * \param  wordDict: 字典
 * \return 是否可以被拆分为字典中的单词
 */
bool wordSpilt_re(string s, unordered_set<string>& wordDict)
{
        if(s.empty()) //判断是否为空，为空则说明可以被分割，返回true
                return true;
        for (int i = 1; i ≤ s.length(); ++i) {
                string prefix = s.substr(0, i);
                if (wordDict.find(prefix) ≠ wordDict.end() &&
wordSpilt_re(s.substr(i), wordDict)) {
                        return true; //如果前缀在字典中，且后缀也在字典
中，则返回true
                }
        }
        return false;
}
```

动态规划思路： 本道题的问题可以切分成多个互不相干的子问题，并且子问题之间还有重叠的更小的子问题。这道题实际是判断可行性，同时符合动态规划算法的要求。

- 我们可以将dp数组定义为bool型，用于存储若干子问题是否可行，例如`dp[i]`表示字符串`s`的前`i`个字符是否可以被拆分为字典中的序列
- 动态规划过程：从字符串s的开头开始，逐个字符考虑，不断扩展字符串的长度，更新dp数组，对于字符串s的每个位置i，可以尝试将其分割为两个部分`s[0,j]`和`s[j,i]`，其中j表示分割点。如果前半部分s[0,j]可以被拆分为字典中的单词序列，并且后半部分s[j,i]也在字典中，则说明整个字符串s[0,i]可以被拆分为字典中的单词序列。
- dp[s.length()]即dp[i+1]存储每个分割点是否可以被拆分开，这样就避免了重叠子问题重复计算。
- 递推关系：$dp[i] = dp[j] \wedge (s[j,i] \in \text{wordDict}), 其中\ 0 \leq j < i$
- 时间复杂度：该算法使用了双重循环来填充dp动态规划数组，外层循环遍历字符串每个字符，内层循环遍历前半部分所有可能位置，所以为$O(n^2)$
- 空间复杂度：算法使用了一个大小为s.length()+1的动态规划数组dp及一个unordered_set字典，因此为$O(n)$



核心代码：

```
/**
 * \brief 使用动态规划方法判断字符串是否可以被拆分为字典中的单词
 * \param s 输入的字符串
 * \param wordDict 字典
 * \return 如果字符串可以被拆分为字典中的单词，则返回true，否则返回false
 */
bool wordSpilt_DP(string s, unordered_set<string>& wordDict) {
    vector<bool> dp(s.length() + 1, false);
    // dp[i]表示s的前i个字符是否可以被拆分为字典中的单词
    dp[0] = true; // 空字符串可以被拆分
    for (int i = 1; i ≤ s.length(); ++i) {// 遍历字符串s
```

```
        for (int j = 0; j < i; ++j) {// 遍历字符串s的前i个字符
            if (dp[j] && wordDict.find(s.substr(j, i - j)) ≠
wordDict.end()) {
                dp[i] = true;
                // 如果前j个字符可以被拆分，且s[j,i)在字典中，则s[0,i)可以
被拆分
                break;
            }
        }
    }
    return dp[s.length()];// 返回此部分子串是否可以被拆分
}
```

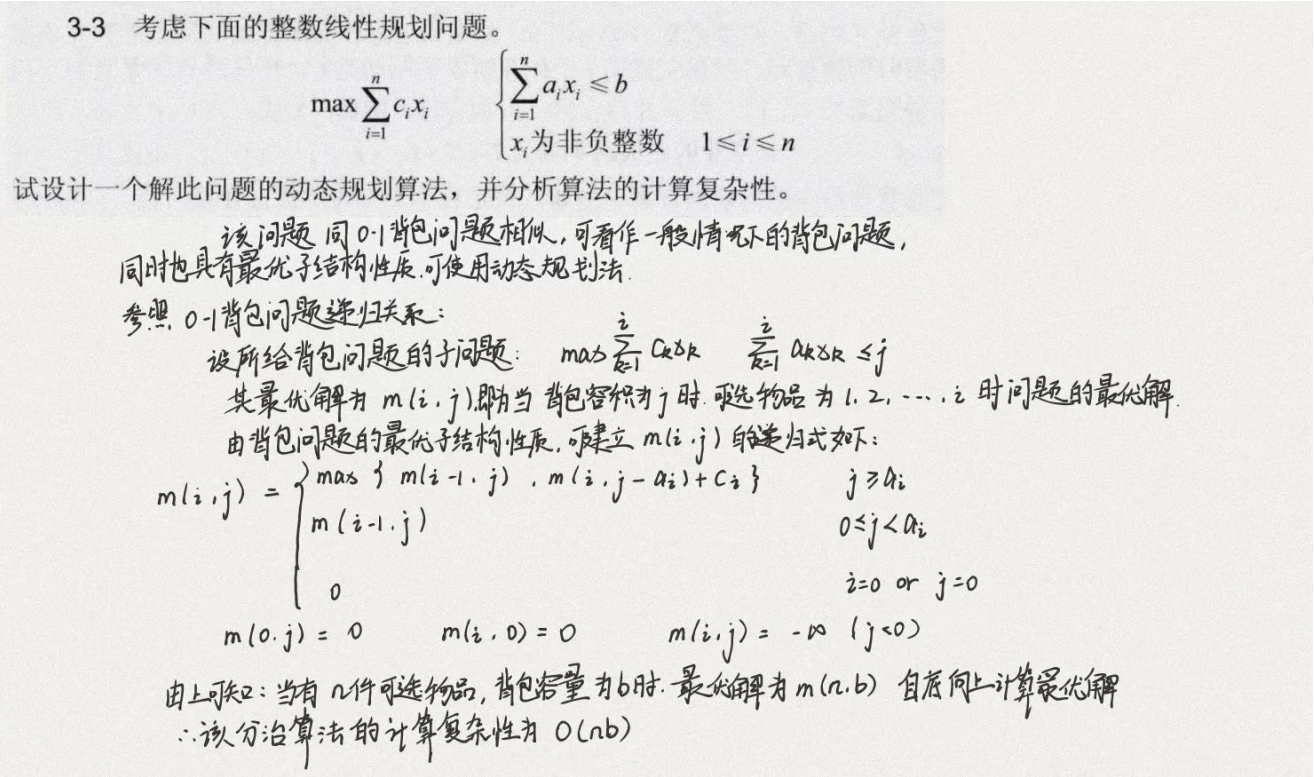
遇到的问题：

- 动态规划的过程考虑不清，导致部分条件判断出误，应当先在草稿纸上捋清楚过程再编写代码。
  源代码：
  核心代码同上，头文件和main函数如下：

```
#include<iostream>
#include<string>
#include<vector>
#include<unordered_set>
using namespace std;
///核心函数部分
int main() {
    string s;
    cout << "Enter the string: ";
    cin >> s;
    int n;
    cout << "Enter the number of words in dictionary: ";
    cin >> n;
    cout << "Enter the words of dictionary: ";
    unordered_set<string> wordDict;
    for (int i = 0; i < n; ++i) {
        string word;
        cin >> word;
        wordDict.insert(word);
    }
    if (wordSpilt_DP(s, wordDict))
        cout << "true" << endl;
    else
        cout << "false" << endl;
```

```
    return 0;
}
```

03 算法分析题



3-3 考虑下面的整数线性规划问题。

$$\max \sum_{i=1}^{n} c_i x_i \qquad \begin{cases} \sum_{i=1}^{n} a_i x_i \leqslant b \\ x_i \text{为非负整数} \quad 1 \leqslant i \leqslant n \end{cases}$$

试设计一个解此问题的动态规划算法，并分析算法的计算复杂性。

该问题同0-1背包问题相似，可看作一般情况下的背包问题，同时也具有最优子结构性质，可使用动态规划法。

参照0-1背包问题递归关系：

设所给背包问题的子问题：$\max \sum_{k=1}^{i} c_k x_k \quad \sum_{k=1}^{i} a_k x_k \leq j$

其最优解为 $m(i,j)$，即当背包容积为 j 时，可选物品为 $1, 2, \cdots, i$ 时问题的最优解。

由背包问题的最优子结构性质，可建立 $m(i,j)$ 的递归式如下：

$$m(i,j) = \begin{cases} \max\{m(i-1,j),\ m(i,j-a_i)+c_i\} & j \geq a_i \\ m(i-1,j) & 0 \leq j < a_i \\ 0 & i=0 \text{ or } j=0 \end{cases}$$

$m(0,j) = 0 \qquad m(i,0) = 0 \qquad m(i,j) = -\infty \ (j<0)$

由上可知：当有 n 件可选物品，背包容量为 b 时，最优解为 $m(n,b)$，自底向上计算最优解。

∴该分治算法的计算复杂性为 $O(nb)$

0-1 背包问题具有最优子结构性质，所以可以用动态规划方法求解，可以据此定义递归关系，建立递归方程，并以自底向上的方式计算最优值，根据计算最优值时的得到的信息，构造最优解。

背包问题的相关代码：

```
void Knapsack(int *v, int *w, int c, int n, int (*m)[maxn]) {
    //先判断第n个物品能不能装入背包
    int jMax = min(w[n] - 1, c);
    //当0≤j≤wn时，m(n，j)=0
    for (int j = 0; j ≤ jMax; j++)
        m[n][j] = 0;
    //当j≥wn时，m(n，j)=vn
    for (int j = w[n]; j ≤ c; j++)
        m[n][j] = v[n];
    //再从n-1往前开始判断第n个物品到第i个物品能不能装下
    for (int i = n - 1; i > 1; i--) {
        jMax = min(w[i] - 1, c);
        for (int j = 0; j < jMax; j++)
            m[i][j] = m[i + 1][j];
        for (int j = w[i]; j ≤ c; j++)
            m[i][j] = max(m[i + 1][j], m[i + 1][j - w[i]] + v[i]);
    }
```

```
    //判断第n个到第1个物品能不能装下
    m[1][c] = m[2][c];
    if (c ≥ w[1])
        m[1][c] = max(m[1][c], m[2][c - w[1]] + v[1]);
}
//回溯查找最优序列，能装下的赋值为1，不能装下的赋值为0
void Traceback(int (*m)[maxn], int *w, int c, int n, int *x) {
    for (int i = 1; i < n; i++) {
        if (m[i][c] == m[i + 1][c])
            x[i] = 0;
        else {
            x[i] = 1;
            c -= w[i];
        }
    }
    x[n] = (m[n][c]) ? 1 : 0;
}
```

04 算法设计题

3-4

题目描述：

3-4　数字三角形问题。

问题描述：给定一个由 n 行数字组成的数字三角形，如图 3-7 所示。

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

图 3-7　数字三角形

试设计一个算法，计算出从三角形的顶至底的一条路径，使该路径经过的数字总和最大。

算法设计：对于给定的由 n 行数字组成的数字三角形，计算从三角形的顶至底的路径经过的数字和的最大值。

数据输入：由文件 input.txt 提供输入数据。文件的第 1 行是数字三角形的行数 n，$1 \leqslant n \leqslant 100$。接下来 n 行是数字三角形各行中的数字。所有数字在 0～99 之间。

结果输出：将计算结果输出到文件 output.txt。文件第 1 行中的数是计算出的最大值。

输入文件示例	输出文件示例
input.txt	output.txt
5	30
7	
3 8	
8 1 0	
2 7 4 4	
4 5 2 6 5	

题目分析与代码设计：

采用自底向上的分析方法对问题进行分析，观察发现：

- ①数字三角形的最后一行上的数字到底的数字和最大的路径为该行上的数字本身；
- ②数字三角形倒数第二行上的数字到底的数字和最大的路径为该行上的数字加上该行左下或者右下的数字之和。

- ③以此类推...我们可以自底而上推出数字三角形的顶部到底部的路径经过的数字和的最大值
- 通过上述分析可以看到，该问题可以分解成多个存在重叠的子问题，存在最优子结构，所以可以使用动态规划的方法自底向上进行求解。
- 设该问题的子问题——从数字三角形的第i行，第j列的数字出发到低端所经过的路径上的数字和的最大值为maxNum

  $$maxNum(i,j)=\begin{cases}maxNum(i,j),i=n\\maxNum(i,j)+max\{maxNum(i+1,j),maxNum(i+1,j+1)\},1\end{cases}$$

  初始时：

				7				
			3		8			
		8		1		0		
	2		7		4		4	
4		5		2		6		5

向上走：

				7+23=30				
			3+20=23		8+13=21			
		8+12=20		1+12=13		0+10=10		
	2+5=7		**7+5=12**		4+6=10		4+6=10	
4		**5**		2		6		5

由此自底向上进行计算，dp[1,1]即为最大路径和。
优化思路： 顶点处只有一个值，可以把最后的最大值放在顶点处。规则类似于比赛机制，两两比较取出胜者，最后顶端即为最大值，比赛方向为自底向上进行迭代。对于(i,j)位置的节点，其进入路径的最大值，就是在下一层的左下路径和、右下路径和的最大值中取得更大的一个，加入进去，状态转移其实和上面的差不多，但是这种思路可以直接利用累加的方式，把最大值积累起来。
状态转移方程：

$$dp[i][j]=\begin{cases}dp[i-1][j]+num[i][0], & j=0(\text{每层最左侧数字})\\dp[i-1][j-1]+num[i][j], & i=j(\text{每层最右侧数字})\\max\{dp[i-1][j],dp[i-1][j-1]\}+num[i][j], & otherwise\end{cases}$$

核心代码：

```
/**
 * \brief  动态规划求解三角形最大路径和
```

```
 * \return 返回顶部最大路径和
 */
int MaxSum() {
    // 初始化最底部的dp值为三角形底部的值
    for (int j = 1; j ≤ n; ++j)
        dp[n][j] = D[n][j];
    // 自底向上计算最大路径和
    for (int i = n - 1; i ≥ 1; --i) {
        for (int j = 1; j ≤ i; ++j) {
            dp[i][j] = maxs(dp[i + 1][j], dp[i + 1][j + 1]) + D[i][j];
        }
    }
    return dp[1][1]; // 返回顶部的最大路径和
}
```

时间复杂度： 初始化for循环为$O(n^2)$,自底向上计算最大路径和次数为$\sum_{i=1}^{n} i = \frac{n(n-1)}{2}$,所以时间复杂度为$O(n^2)$
空间复杂度： 使用一个二维数组dp用来存储每个位置的最大路径和，数组大小事n*n，所以空间复杂度为 $O(n^2)$

遇到的问题：

- 这道题遇到的问题主要在于找到递推关系，从底向上求解子问题。状态的转移依靠的是每一次的选择，因此考虑如何改变状态，就是在考虑我们每一次要做什么决策。这道题的决策：选择上一层的一个节点（在左上或右上），然后把自己纳入路径中，但是这样定义有两种初始情况：① 三角形的左侧边，这条边上的数字只有右上可取② 三角形的右侧边，只有左上可取
- 另外这是一个三角形，我使用了两个二维数组来存储三角形中数字和dp数组，实际开销很大，由于最后只要将顶端保存最大值即可，实际可以仅修改原数组，不需要领开辟一个二维数组存储中间数据；此外，可以采用压缩矩阵的方法对三角形进行压缩

3-10

题目：

3-10　最大长方体问题。

问题描述：一个长、宽、高分别为 m、n、p 的长方体被分割成 $m\times n\times p$ 个小立方体。每个小立方体内有一个整数。试设计一个算法，计算所给长方体的最大子长方体。子长方体的大小由它所含所有整数之和确定。

算法设计：对于给定的长、宽、高分别为 m、n、p 的长方体，计算最大子长方体的大小。

数据输入：文件 input.txt 提供输入数据，第 1 行是 3 个正整数 m、n、p（$1\leqslant m, n, p\leqslant 50$）。在接下来的 $m\times n$ 行中每行 p 个正整数，表示小立方体中的数。

结果输出：将计算结果输出到文件 output.txt。文件的第 1 行中的数是计算出的最大子长方体的大小。

输入文件示例	输出文件示例
input.txt	output.txt
3 3 3 0 −1 2 1 2 2 1 1 −2 −2 −1 −1 −3 3 −2 −2 −3 1 −2 3 3 0 1 3 2 1 −3	14

题目分析与算法设计：

三维问题我们可以采用分层求解的思路，先简化为二维的，二维的简化为一维的，通过求解不同有重叠的子问题，得出各个维度的最优解。先从最简单的开始分析：

- ①首先先从一维开始考虑：先编写一维的“最大字段和”的解法，遍历整个一维数组，设dp[i]为前i个数的最大和，如果某个数nums[i]<0，dp[i]=dp[i-1],否则为dp[i-1]+nums[i].
- ②然后考虑二维的情况：基于“最大字段和”，编写二维的“最大子矩阵和”的解法，将二维数组抽象为多个一维数组，遍历调用一维求最大和的方法
- ③最后考虑三维的情况：基于“最大子矩阵和”，编写三维的“最大子长方体和”的解法，将三维数组抽象为多个二维数组，遍历调用二维的求最大和的方法

  max_1D 求解最大字段和

  设计思路： 使用动态规划(Kadane算法)求解一维数组的最大子数组和，使用一个辅助变量b来记录当前子数组和，另一个变量sum来记录最大子数组和，当累加的子数组和b开始大于0时，继续累加；当b小于等于0时，将b更新为当前元素的值，重新开始累加.

  时间复杂度： $O(n)$

  空间复杂度： $O(1)$

  相关代码：

```cpp
/**
 * 计算一维数组的最大子数组和
 * @param array 一维数组
 * @return 最大子数组和
 */
int max_1D(const vector<int>& array) {
    int maxSum = 0;
    int currentSum = 0;
    for (int value : array) {
        // 如果当前和小于0，则从当前元素开始重新计算
        currentSum = max(value, currentSum + value);
        // 更新最大和
        maxSum = max(maxSum, currentSum);
    }
    return maxSum;
}
```

max_2D 求解最大子矩阵和

设计思路： 使用动态规划求解一维数组的最大子矩阵和，使用两层循环来遍历所有有可能的子矩阵，也就是层切片，其中内层循环用于计算每一列的元素和，外层遍历所有行，并调用max_1D函数来计算当前子矩阵的最大和。这里使用动态规划思想，通过记录前面已经计算过的子长方体和来避免重复计算，从而提高了计算效率.

时间复杂度： $O(m \times n)$ m，n分别代表矩阵的行和列,对于每个i（行），我们都需要计算n个一维最大子数组和（max_1D函数的调用），所以这部分的时间复杂度是O(n),外层循环遍历m行，所以这部分的时间复杂度是$O(mn)$。

空间复杂度： $O(n)$ 需要一个额外的一维数组b来存储每一列的累加和

相关代码：

```cpp
/**
 * 计算二维矩阵的最大子矩阵和
 * @param matrix 二维矩阵
 * @return 最大子矩阵和
 */
int max_2D(const vector<vector<int>>& matrix) {
    int maxSum = 0;
    int rows = matrix.size();
    int cols = matrix[0].size();
    // 枚举左右边界
    for (int left = 0; left < cols; ++left) {
        vector<int> sums(rows, 0);
        // 枚举右边界
        for (int right = left; right < cols; ++right) {
            // 更新每一行的和
            for (int i = 0; i < rows; ++i)
```

```
                sums[i] += matrix[i][right];
            // 计算并更新当前和
            int currentMax = max_1D(sums);
            maxSum = max(maxSum, currentMax);
        }
    }
    return maxSum;
}
```

max_3D 求解最大子长方体和

设计思路： 类似于二维情况，通过固定长方体的一个顶点，然后逐层向下扩展来计算所有可能得子长方体的和。对于每一个固定顶点，使用max_2D函数计算以该顶点为起点的所有子长方体的和,遍历所有可能的顶点，找到最大的子长方体和.

时间复杂度： $O(m \times m \times n \times n \times p)$外层循环遍历m层，中间层循环遍历n列，这两个循环的时间复杂度是$O(mn)$。对于每个(i, j)位置，我们调用max_2D函数，其时间复杂度是O(m n p)，因为我们需要遍历所有可能的子矩阵。将这些组合起来，得到总体的时间复杂度为$O(m * n * (m * n * p))$。

空间复杂度： $O(np)$ 需要一个二维数组b来存储每个可能的子矩阵的累加和。

相关代码：

```
/**
 * 计算三维长方体的最大子长方体和
 * @param cuboid 三维长方体
 * @return 最大子长方体和
 */
int max_3D(const vector<vector<vector<int>>>& cuboid) {
    int maxSum = 0;
    int depth = cuboid.size();
    int height = cuboid[0].size();
    int width = cuboid[0][0].size();
    // 枚举z轴方向
    for (int z = 0; z < depth; ++z) {
        // 枚举y轴方向
        vector<vector<int>> slices(height, vector<int>(width, 0));
        // 枚举z轴方向
        for (int x = z; x < depth; ++x) {
            // 更新每一层的和
            for (int y = 0; y < height; ++y) {
                for (int xx = 0; xx < width; ++xx)
                    slices[y][xx] += cuboid[x][y][xx];
            }
            int currentMax = max_2D(slices);
            maxSum = max(maxSum, currentMax);
        }
```

```
    }
    return maxSum;
}
```

时间复杂度 $O(m*m*n*n*p)$
空间复杂度 $O(m*n*p)$
遇到的问题：
理解问题：一开始可能难以理解题目要求，特别是当涉及到多维数组时，后来看解析发现可以将三维拍成二维，二维压成一维来看，层层调用。
边界条件：在处理数组和矩阵时，很容易忘记边界条件，例如，在计算子数组和时，需要考虑从数组的任何位置开始和结束的所有可能组合。
状态定义与转化：动态规划问题的第一步是定义状态，以及状态如何进行转化，从最大字段和入手，发现可以采用自顶向上的方法进行解决。对于最大子结构和问题，状态转移方程要考虑包含当前位置的子数组和与不包含当前位置的子数组和之间的最大值。
源代码：

```
#include <iostream>
#include <vector>
using namespace std;
/**
 * 计算一维数组的最大子数组和
 * @param array 一维数组
 * @return 最大子数组和
 */
int max_1D(const vector<int>& array) {
    int maxSum = 0;
    int currentSum = 0;
    for (int value : array) {
        // 如果当前和小于0，则从当前元素开始重新计算
        currentSum = max(value, currentSum + value);
        // 更新最大和
        maxSum = max(maxSum, currentSum);
    }
    return maxSum;
}

/**
 * 计算二维矩阵的最大子矩阵和
 * @param matrix 二维矩阵
 * @return 最大子矩阵和
 */
int max_2D(const vector<vector<int>>& matrix) {
    int maxSum = 0;
    int rows = matrix.size();
```

```cpp
    int cols = matrix[0].size();
    // 枚举左右边界
    for (int left = 0; left < cols; ++left) {
        vector<int> sums(rows, 0);
        // 枚举右边界
        for (int right = left; right < cols; ++right) {
            // 更新每一行的和
            for (int i = 0; i < rows; ++i)
                sums[i] += matrix[i][right];
            // 计算并更新当前和
            int currentMax = max_1D(sums);
            maxSum = max(maxSum, currentMax);
        }
    }
    return maxSum;
}

/**
 * 计算三维长方体的最大子长方体和
 * @param cuboid 三维长方体
 * @return 最大子长方体和
 */
int max_3D(const vector<vector<vector<int>>>& cuboid) {
    int maxSum = 0;
    int depth = cuboid.size();
    int height = cuboid[0].size();
    int width = cuboid[0][0].size();
    // 枚举z轴方向
    for (int z = 0; z < depth; ++z) {
        // 枚举y轴方向
        vector<vector<int>> slices(height, vector<int>(width, 0));
        // 枚举z轴方向
        for (int x = z; x < depth; ++x) {
            // 更新每一层的和
            for (int y = 0; y < height; ++y) {
                for (int xx = 0; xx < width; ++xx)
                    slices[y][xx] += cuboid[x][y][xx];
            }
            int currentMax = max_2D(slices);
            maxSum = max(maxSum, currentMax);
        }
    }
    return maxSum;
}
int main() {
    int m, n, p;
    cin >> m >> n >> p;
```

```
    vector<vector<vector<int>>> cuboid(m, vector<vector<int>>(n,
vector<int>(p, 0)));
    for (int i = 0; i < m; ++i) {
        for (int j = 0; j < n; ++j) {
            for (int k = 0; k < p; ++k) {
                cin >> cuboid[i][j][k];
            }
        }
    }
    cout << max_3D(cuboid) << endl;
    return 0;
}
```

05 总结

通过本次作业，我对动态规划的认识更深了一步，明白了自底向上的求解方法，以及动态规划算法的适用问题，使用条件及算法设计流程。在上课时，对于动态规划及递归分治的区别了解不是很透彻，导致做题出现了种种问题，还好借助课本解析，网络博客了解了很多相关知识，改了很多错误。