概念

分治法将问题划分成一些独立的子问题，递归地求解各子问题，然后合并子问题的解而得到原问题的解。

动态规划适用于子问题不是独立的情况，也就是个子问题包含公共的子子问题。动态规划对每个子子问题只求解一次，将其结果保存在一张表中，从而避免每次遇到各个子问题时重新计算答案。

动态规划的算法可分为以下4个步骤：

1. 描述最优解的结构。
2. 递归定义最优解的值。
3. 按自底向上的方式计算最优解的值。
4. 由计算的结果构造一个最优解。

0-1背包问题

假设有n个物品，它们的重量分别为$w_1, w_2, \ldots, w_n$，价值分别为$v_1, v_2, \ldots, v_n$，给一承重为W的背包，求装入的物品具有最大的价值总和。

首先给出利用动态规划计算0-1背包问题的递归式：

$$OPT(i,W)=\begin{cases}0 & if\ i=0\\ OPT(i-1,W) & if\ w_i>W\\ \max\{OPT(i-1,W), v_i+OPT(i-1,W-w_i)\} & otherwise\end{cases}$$

在这个递归式中，$OPT(i,W)$表示前i件物品放入容量为W的背包中的最大价值。

初始化时，$OPT(0,j)=0$，意为当没有物品放入时，不管背包容量多少，其最大价值为0；$OPT(i,0)$意为当背包容量为0时，不管从前i件物品中怎么取，最大价值都是0。

接着进行动态规划，在已知$f(i-1,j)$时，即已知在前$i-1$件物品放入容量为j的背包时的最大价值情况下，求$f(i,j)$。

在求$f(i,j)$时，首先判断物品i的重量是否超过目前背包的重量j，如果超过，则这个物品i放不进背包，则$f(i,j)=f(i-1,j)$。如果背包可以放下物品i，则尝试把背包中的重量减去物品i的重量w_i，这样$f(i-1,j-w_i)$表示前i-1件物品在背包容量为$i-w_i$下的最大价值，此时如果放入物品i，那么价值就变为$f(i-1,j-w_i)+v_i$。判断$f(i-1,j-w_i)+v_i$与$f(i-1,j)$的大小，选择较大的一个作为在背包容量为i下的最大价值。以上就是对于这个递归式的算法描述。

相对应的伪代码如下图所示：

$$OPT(i,W)=\begin{cases}0 & if\ i=0\\ OPT(i-1,W) & if\ w_i>W\\ \max\{OPT(i-1,W),v_i+OPT(i-1,W-w_i)\} & otherwise\end{cases}$$

可以将这个二维数组可视化，有如下图所示的五个物品和其对应的价值，且背包容量为11，那么如何让背包中装入的物品有最大的价值?

Item	Value	Weight
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

遍历过程如下所示，从上到下代表i的遍历，从左到右代表j的遍历，最后可得到最大价值为40。

Item	Value	Weight
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

0-1背包问题已经被证明是NP完全问题，而它却有着一个动态规划解法，该解法有着O(nW)的时间复杂度，其中n是物品的个数，W是背包限制的最大负重。但是这种动态规划的算法称为伪多项式时间算法，这种算法不能真正意义上实现多项式时间内解决问题。

最长公共子序列（LCS问题）

以最长公共子序列为例，将动态规划的四个步骤按部就班地走一遍。

Step1 描述最优解的结构（分析问题）

假设$X = A, B, C, B$，$Y = B, D, C, A, B$，我们可以容易地想到暴力解法，即将枚举出X中的所有子序列，然后检查每个子序列是否是Y的子序列。假设X和Y的长度分别是m和n，那么暴力解法的时间复杂度是$O(2^m n)$。

我们可以观察到，LCS问题具有最优子结构，设
$X =< x_1, x_2, ..., x_m >$和$Y =< y_1, y_2, ..., y_n >$为两个序列，
$Z =< z_1, z_2, ..., Z_k >$为X和Y的任意一个LCS，则有LCS的最优子结构定理：

1. 如果$x_m = y_n$，那么$z_k = x_m = y_n$，而且Z_{k-1}是 X_{m-1}和Y_{n-1}的一个LCS。
2. 如果$x_m \neq y_n$，那么$z_k \neq x_m$意味着Z是 X_{m-1}和Y的一个LCS。
3. 如果$x_m \neq y_n$，那么$z_k \neq y_n$意味着Z是 X和Y_{n-1}的一个LCS。

Step2 递归定义最优解的值（递归解决）

用$C[i,j]$表示X_i和Y_j的最长公共子序列LCS的长度，则有公式：

$$C[i,j] = \begin{cases} 0, & \text{当}i=0\text{或}j=0 \\ C[i-1,j-1]+1, & \text{当}i,j>0\text{且}x_i = y_j \\ max(C[i,j-1],C[i-1,j]) & \text{当}i,j>0\text{且}x_i \neq y_j \end{cases}$$

Step3 按自底向上的方式计算最优解的值（计算LCS的长度）

LCS_LENGTH以两个序列为输入，将LCS的长度保存到二维数组c中，将构造过程保存到另一个二维数组b中，伪代码如下所示：

```
def LCS_LENGTH(X,Y):
    m = length(X)
    n = length(Y)

    # 初始化
    for i = 1 to m:
        c[i][0] = 0
    for j = 1 to n:
        c[0][j] = 0

    # 计算LCS的长度
    for i = 1 to m:
        for j = 1 to n:
```

```
            if x[i] == y[j]:
                c[i, j] = c[i-1, j-1] + 1
                b[i, j] = '\'
            elif c[i-1, j] >= c[i, j-1]:
                c[i, j] = c[i-1, j]
                b[i, j] = '|'
            else:
                c[i, j] = c[i, j-1]
                b[i, j] = '-'
return c, b
```

Step4 由计算的结果构造一个最优解（构建LCS）

根据LCS_LENGTH返回的表b，可以构建一个LCS序列，输出所有值为'\'的元素，即可得到LCS，PRINT_LCS的伪代码如下所示：

```
def PRINT_LCS(b, X, i, j):
    if i==0 or j==0:
        return 0
    if b[i, j] == '\':
        PRINT_LCS(b, X, i-1, j-1)
        print X[i]
    elif b[i, j] == '|':
        PRINT_LCS(b, X, i-1, j)
    elif PRINT_LCS(b, X, i, j-1)
```

为了加深理解，使用C++实现了以上伪代码，使用PPT上的例子，最终得出结果如下图所示：

```
The X is ABCB
The Y is BDCAB

0 0 0 0 0 0
0 0 0 0 1 1
0 1 1 1 1 2
0 1 1 2 2 2
0 1 1 2 2 3

The length of LCS is: 3
The LCS is: B C B
```

全部C++代码如下所示:

```
#include "stdafx.h"
#include <iostream>
using namespace std;
#define m 4
#define n 5
#define skew 0
#define up 1
#define level 2

void lcs_length(char*X, char*Y, int c[m+1][n+1], int b[m+1][n+1]){
    int i, j;
    for (i=0; i<m; i++)
        c[i][0] = 0;
    for (j=0; j<n; j++)
        c[0][j] = 0;
    for (i=1; i<=m; i++)
        for (j=1; j<=n; j++)
        {
            if (X[i] == Y[j])
            {
                c[i][j] = c[i-1][j-1] + 1;
                b[i][j] = skew;
            }
```

```
            else if (c[i-1][j] >= c[i][j-1])
            {
                c[i][j] = c[i-1][j];
                b[i][j] = up;
            }
            else
            {
                c[i][j] = c[i][j-1];
                b[i][j] = level;
            }
        }
}

void print_lcs(int b[m+1][n+1], char* X, int i, int j)
{
    if (i==0 || j==0)
        return;
    if (b[i][j] == skew)
    {
        print_lcs(b, X, i-1, j-1);
        cout << X[i] << ' ';
    }
    else if(b[i][j] == level)
        print_lcs(b, X, i, j-1);
    else print_lcs(b, X, i-1, j);
}

int main(){
    char X[m+1] = {' ','A','B','C','B'};
    char Y[n+1] = {' ','B','D','C','A','B'};
    int c[5][6] = {0};
    int b[5][6] = {0};
    int i, j;
    cout << "The X is ABCB" << endl;
```

```
    cout << "The Y is BDCAB" << endl << endl;

    lcs_length(X, Y, c, b);
    for(i=0; i<=m; i++)
    {
        for (j=0; j<=n; j++)
            cout << c[i][j] << " ";
        cout << endl;
    }
    cout << endl << "The length of LCS is: " << c[m][n] << 
endl;
    cout << "The LCS is: ";
    print_lcs(b, X, m, n);

    return 0;
}
```

编辑距离（Edit Distance）

假设X和Y是两个字符串，我们要用最少的操作将X转换为Y，三种操作可供使用，分别是**删除**、**插入**和**替换**，最少操作的数目称为编辑距离。

与LCS类似，也可得到编辑距离的公式：

$$C[i,j]=\begin{cases}0, & \text{当}i=0\text{或}j=0\\ C[i-1,j-1]+1, & \text{当}i,j>0\text{且}x_i=y_j\\ min(C[i,j-1],C[i-1,j],C[i-1,j-1]) & \text{当}i,j>0\text{且}x_i\neq y_j\end{cases}$$

下面给出一个例子，具体的实现与LCS类似，这里不再赘述。

X = ACGGTTA Y = CGTAT

	0	C	G	T	A	T
0	0	1	2	3	4	5
A	1	1	2	3	3	4
C	2					
G	3					
G	4					
T	5					
T	6					
A	7					???

矩阵连乘问题（Chain MatrixMultiplication）

给定n个矩阵 $\{A1, A2, \ldots, An\}$ ，其中Ai与$Ai+1$是可乘的，$i = 1, 2, \ldots, n-1$。确定计算矩阵连乘积的计算次序，使得依此次序计算矩阵连乘积需要的数乘次数最少。

假如我们要得到计算从A_i到A_j的最优计算次序。首先假设这个计算次序在矩阵A_k和A_{k+1}之间断开，且$i \leq k < j$，则计算量为前一部分的计算量、后一部分的计算量以及两部分相乘的计算量之和。

可以递归地定义$C[i, j]$为：

$$C(i,j) = \begin{cases} 0 & i = j \\ min_{i \le k < j}(C(i,k) + C(k+1,j) + m_{i-1}m_k m_j) & i < j \end{cases}$$

可以看到，k的位置只有$j-i$种可能，此算法的时间复杂度为$O(n^3)$，给出一个这种动态规划算法的计算实例。

- Find an optimal parenthesization of a matrix-chain product whose sequence of dimensions is (5, 10, 3, 12, 5, 50, 6).

i\j	1	2	3	4	5	6
1	0	150	330	405	1655	2010
2		0	360	330	2430	1950
3			0	180	930	1770
4				0	3000	1860
5					0	1500
6						0