## 一、动态规划基本思想

一般来说，只要问题可以划分成规模更小的子问题，并且原问题的最优解中包含了子问题的最优解，则可以考虑用动态规划解决。动态规划的实质是分治思想和解决冗余，因此，动态规划是一种将问题实例分解为更小的、相似的子问题，并存储子问题的解而避免计算重复的子问题，以解决最优化问题的算法策略。

由此可知，动态规划法与分治法和贪心法类似，它们都是将问题实例归纳为更小的、相似的子问题，并通过求解子问题产生一个全局最优解。其中贪心法的当前选择可能要依赖已经作出的所有选择，但不依赖于有待于做出的选择和子问题。因此贪心法自顶向下，一步一步地作出贪心选择；而分治法中的各个子问题是独立的 (即不包含公共的子子问题)，因此一旦递归地求出各子问题的解后，便可自下而上地将子问题的解合并成问题的解。

但不足的是，如果当前选择可能要依赖子问题的解时，则难以通过局部的贪心策略达到全局最优解；如果各子问题是不独立的，则分治法要做许多不必要的工作，重复地解公共的子问题。解决上述问题的办法是利用动态规划。该方法主要应用于最优化问题，这类问题会有多种可能的解，每个解都有一个值，而动态规划找出其中最优(最大或最小)值的解。若存在若干个取最优值的解的话，它只取其中的一个。

在求解过程中，该方法也是通过求解局部子问题的解达到全局最优解，但与分治法和贪心法不同的是，动态规划允许这些子问题不独立，也允许其通过自身子问题的解作出选择，该方法对每一个子问题只解一次，并将结果保存起来，避免每次碰到时都要重复计算。

因此，动态规划法所针对的问题有一个显著的特征，即它所对应的子问题树中的子问题呈现大量的重复。动态规划法的关键就在于，对于重复出现的子问题，只在第一次遇到时加以求解，并把答案保存起来，让以后再遇到时直接引用，不必重新求解。

动态规划法的关键就在于，对于重复出现的子问题，只在第一次遇到时加以求解，并把答案保存起来，让以后再遇到时直接引用，不必重新求解。

## 二、动态规划实例

### 1、0-1背包问题

#### 问题描述

    假设我们有n件物品，分别编号为1,2,…n。其中编号为i的物品价值Vi,它的重量为Wi。为了简化问题，假设价值和重量都是整数。现在假设我们有一个背包，它能够承载的重量是W。现在，我们希望王背包里装这些物品，使得包里装的物品价值最大化，那么我们该如何选择装的物品？问题结构如下图所示：

#### 初步分析

    对于这个问题，一开始确实有点不太好入手。一堆的物品，每一个都有一定的质量和价值，我们能够装入的总重量有限制，该怎么来装使得价值最大呢？对于这n个物品，每个物品我们可能会选，也可能不选，那么我们总共就可能有2^n种组合选择方式。如果我们采用这种办法来硬算的话，则整体的时间复杂度就达到指数级别的，肯定不可行。  
现在我们换一种思路。既然每一种物品都有价格和重量，我们优先挑选那些单位价格最高的是否可行呢？比如在下图中，我们有3种物品，他们的重量和价格分别是10, 20, 30 kg和60, 100, 120

    那么按照单位价格来算的话，我们最先应该挑选的是价格为60的元素，选择它之后，背包还剩下50 - 10 = 40kg。再继续前面的选择，我们应该挑选价格为100的元素，这样背包里的总价值为60 + 100 = 160。所占用的重量为30, 剩下20kg。因为后面需要挑选的物品为30kg已经超出背包的容量了。我们按照这种思路能选择到的最多就是前面两个物品。如下图：

    按照我们前面的期望，这样选择得到的价值应该是最大的。可是由于有一个背包重量的限制，这里只用了30kg，还有剩下20kg浪费了。这会是最优的选择吗？我们看看所有的选择情况：

    很遗憾，在这几种选择情况中，我们前面的选择反而是带来价值最低的。而选择重量分别为20kg和30kg的物品带来了最大的价值。看来，我们刚才这种选择最佳单位价格的方式也行不通。

#### 动态规划解决问题

  既然前面两种办法都不可行，我们再来看看有没有别的方法。我们再来看这个问题。我们需要选择n个元素中的若干个来形成最优解，假定为k个。那么对于这k个元素a1, a2, ...ak来说，它们组成的物品组合必然满足总重量<=背包重量限制，而且它们的价值必然是最大的。因为它们是我们假定的最优选择嘛，肯定价值应该是最大的。假定ak是我们按照前面顺序放入的最后一个物品。它的重量为wk，它的价值为vk。既然我们前面选择的这k个元素构成了最优选择，如果我们把这个ak物品拿走，对应于k-1个物品来说，它们所涵盖的重量范围为0-(W-wk)。假定W为背包允许承重的量。假定最终的价值是V，剩下的物品所构成的价值为V-vk。这剩下的k-1个元素是不是构成了一个这种W-wk的最优解呢？

    我们可以用反证法来推导。假定拿走ak这个物品后，剩下的这些物品没有构成W-wk重量范围的最佳价值选择。那么我们肯定有另外k-1个元素，他们在W-wk重量范围内构成的价值更大。如果这样的话，我们用这k-1个物品再加上第k个，他们构成的最终W重量范围内的价值就是最优的。这岂不是和我们前面假设的k个元素构成最佳矛盾了吗？所以我们可以肯定，在这k个元素里拿掉最后那个元素，前面剩下的元素依然构成一个最佳解。

    现在我们经过前面的推理已经得到了一个基本的递推关系，就是一个最优解的子解集也是最优的。可是，我们该怎么来求得这个最优解呢？我们这样来看。假定我们定义一个函数c[i, w]表示到第i个元素为止，在限制总重量为w的情况下我们所能选择到的最优解。那么这个最优解要么包含有i这个物品，要么不包含，肯定是这两种情况中的一种。如果我们选择了第i个物品，那么实际上这个最优解是c[i - 1, w-wi] + vi。而如果我们没有选择第i个物品，这个最优解是c[i-1, w]。这样，实际上对于到底要不要取第i个物品，我们只要比较这两种情况，哪个的结果值更大不就是最优的么？

    在前面讨论的关系里，还有一个情况我们需要考虑的就是，我们这个最优解是基于选择物品i时总重量还是在w范围内的，如果超出了呢？我们肯定不能选择它，这就和c[i-1, w]一样。

    另外，对于初始的情况呢？很明显c[0, w]里不管w是多少，肯定为0。因为它表示我们一个物品都不选择的情况。c[i, 0]也一样，当我们总重量限制为0时，肯定价值为0。

    这样，基于我们前面讨论的这3个部分，我们可以得到一个如下的递推公式：

    有了这个关系，我们可以更进一步的来考虑代码实现了。我们有这么一个递归的关系，其中，后面的函数结果其实是依赖于前面的结果的。我们只要按照前面求出来最基础的最优条件，然后往后面一步步递推，就可以找到结果了。

    我们再来考虑一下具体实现的细节。这一组物品分别有价值和重量，我们可以定义两个数组int[] v, int[] w。v[i]表示第i个物品的价值，w[i]表示第i个物品的重量。为了表示c[i, w]，我们可以使用一个int[i][w]的矩阵。其中i的最大值为物品的数量，而W表示最大的重量限制。按照前面的递推关系，c[i][0]和c[0][w]都是0。而我们所要求的最终结果是c[n][w]。所以我们实际中创建的矩阵是(n + 1) x (W + 1)的规格。

具体数据：物品个数n = 5,物品重量w[n] = {0,2,2,6,5,4}，物品价值v[n] = {0,6,3,5,4,6}，W = 10。

#### 代码实现

1. #include <iostream>
2. #define  max(a,b)((a)>(b)?a:b)
3. using namespace std;
5. int Weight[] = {0,2,2,6,5,4};  *//物品的重量*
6. int Value[] = {0,6,3,5,4,6};  *//物品的价值*
7. int table[6][11];     *//存放表示到第i个元素为止，在限制总重量为w的情况下我们所能选择到的最优解*
8. bool Frist\_Flag = true;  *//主要是初始化table[][] = Value[1]*
9. int main(void)
10. {
11. for (int i = 1;i <= 5;i++)
12. {
13. for (int j = 1;j <=10;j++)
14. {
15. if (Frist\_Flag == true)
16. {
17. if (Weight[1] <= j)
18. {
19. table[i][j] = Value[1];
20. Frist\_Flag = false;
21. }
23. }
24. else
25. {
26. if (Weight[i] > j)
27. {
28. table[i][j] = table[i-1][j];
29. }
30. else
31. {
32. table[i][j] = max(table[i-1][j-Weight[i]]+Value[i],table[i-1][j]);
33. }
34. }
35. }
36. }
37. cout << "MaxValue = "<< table[5][10] << endl;
38. return 0;
40. }

### **2、最大子数组和问题**

#### **问题**

一个有N个整数元素的一维数组（A[0],A[1],…A[N-1]），这个数组有很多子数组，求子数组和的最大值？注意：子数组必须是连续的、不需要返回子数组的具体位置、数组中包含：正、负、零整数、子数组不能空。

例如：

int A[5] = {-1,2,3,-4,2};

符合条件的子数组为2，3，即答案为5；

#### 穷举法：

1. int MaxSubStringSum(int \*A,int n)
2. {
3. int MaxSum = A[0];
4. int sum = 0;
5. for (int i = 0;i < n;i++)
6. {
7. sum = 0;
8. for (int j = i;j < n;j++)
9. {
10. sum += A[j];
11. MaxSum = max(MaxSum,sum);
13. }
14. }
15. return MaxSum;
16. }

穷取法最为直接，当然耗时也较多，时间复杂度为O(n^2);

#### 进一步分析

我们利用穷举法虽然简单易懂，但是其时间复杂度很大，我们试着优化。现在考虑数组的第一个元素A[0]，与和最大的子数组(A[i],……A[j])之间的关系，有以下三种关系：

 1) i = j = 0;A[0]本身构成和最大的子数组

 2) j > i = 0;和最大的子数组以A[0]开头

 3) i > 0;A[0]与和最大子数组没有关系

从上面3中情况可以看出，可以将一个大问题（N个元素的数组）转化为一个较小的问题（N - 1个元素的数组）。假设已经知道(A[1],……A[N-1])中和最大的子数组和为MaxSum[1]，并且知道，(A[1],……A[N-1])中包含A[1]的和最大的子数组为TempMaxSum[1]。我们就可以把(A[0],……A[N-1])求和最大子数组问题转换为，MaxSum[0] = max{A[0],A[0]+TempMaxSum[1],MaxSum[1]}。

#### 代码实现：

1. int MaxSubStringSum(int \*A,int n)
2. {
3. int MaxSum = A[0];
4. int TempMaxSum = A[0];
5. for(int i = 1;i < n;i++)
6. {
7. TempMaxSum = max(A[i],TempMaxSum + A[i]);
8. MaxSum = max(MaxSum,TempMaxSum);
9. }
10. return MaxSum;
11. }