1.动态规划的基本思想

　　动态规划算法通常用于求解具有某种最优性质的问题。其基本思想也是将待求解问题分解成若干个子问题，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，适合于用动态规划求解的问题，经分解得到子问题往往不是互相独立的。若用分治法来解这类问题，则分解得到的子问题数目太多，有些子问题被重复计算了很多次。如果我们能够保存已解决的子问题的答案，而在需要时再找出已求得的答案，这样就可以避免大量的重复计算，节省时间。我们可以用一个表来记录所有已解的子问题的答案。不管该子问题以后是否被用到，只要它被计算过，就将其结果填入表中。这就是动态规划法的基本思路。具体的动态规划算法多种多样，但它们具有相同的填表格式。 与分治法最大的差别是：适合于用动态规划法求解的问题，经分解后得到的子问题往往不是互相独立的（即下一个子阶段的求解是建立在上一个子阶段的解的基础上，进行进一步的求解）

2.动态规划的应用场景

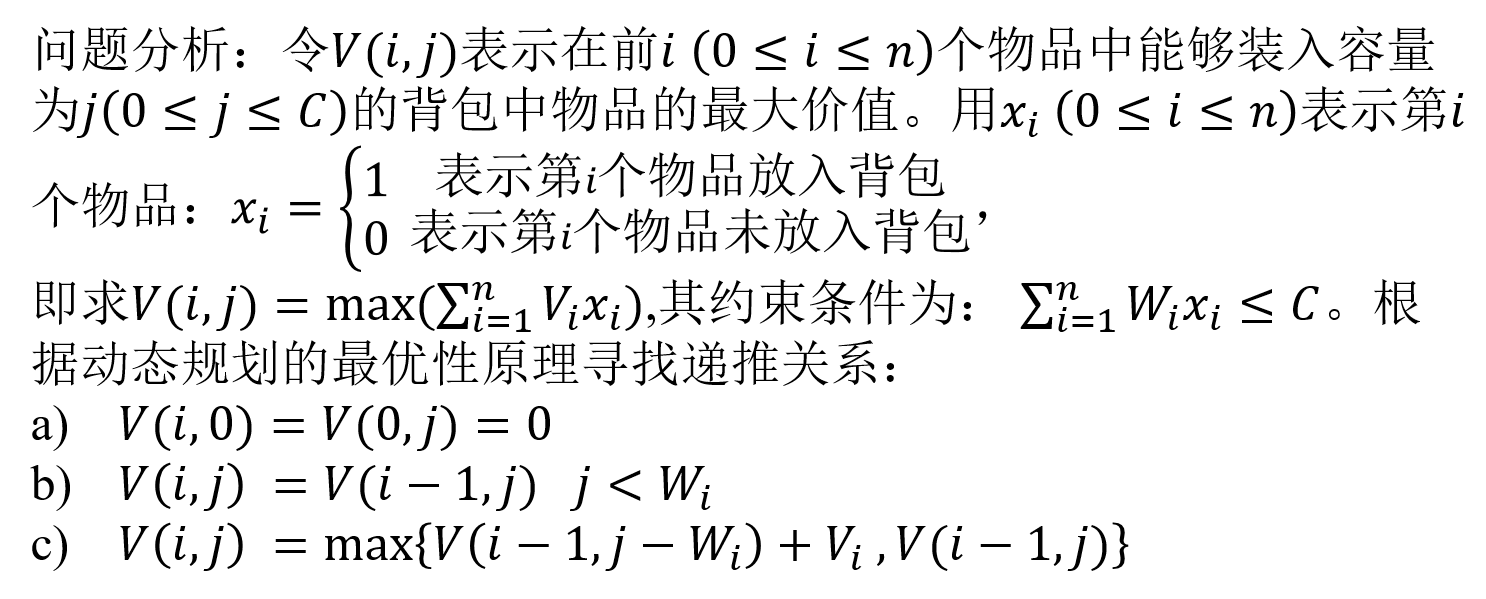
　　适用动态规划的问题必须满足最优化原理、无后效性和重叠性。

　　a.最优化原理（最优子结构性质） 最优化原理可这样阐述：一个最优化策略具有这样的性质，不论过去状态和决策如何，对前面的决策所形成的状态而言，余下的诸决策必须构成最优策略。简而言之，一个最优化策略的子策略总是最优的。一个问题满足最优化原理又称其具有最优子结构性质。

　　b.无后效性 将各阶段按照一定的次序排列好之后，对于某个给定的阶段状态，它以前各阶段的状态无法直接影响它未来的决策，而只能通过当前的这个状态。换句话说，每个状态都是过去历史的一个完整总结。这就是无后向性，又称为无后效性。

　　c.子问题的重叠性动态规划将原来具有指数级时间复杂度的搜索算法改进成了具有多项式时间复杂度的算法。其中的关键在于解决冗余，这是动态规划算法的根本目的。动态规划实质上是一种以空间换时间的算法，它在实现的过程中，不得不存储产生过程中的各种状态，所以它的空间复杂度要大于其它的算法。

3.01背包问题建模：如下图



　　图片中abc三个公式详细解析：a式表示前𝑖个物品中挑选放入承重为0的背包中和没有物品放入承重为𝑗的背包中是相等为0。 b式表明：如果第𝑖个物品的重量大于背包的容量，则装人前𝑖个物品得到的最大价值和装入前𝑖−1个物品得到的最大价是相同的，即物品𝑖不能装入背包。 c式表明：如果第𝑖个物品的重量小于背包的容量，则会有一下两种情况： (1)如果把第𝑖个物品装入背包，则背包物品的价值等于第𝑖−1个物品装入容量位𝑗−𝑊\_𝑖 的背包中的价值加上第𝑖个物品的价值𝑉\_𝑖; (2)如果第𝑖个物品没有装入背包，则背包中物品价值就等于把前𝑖−1个物品装入容量为𝑗的背包中所取得的价值。显然，取二者中价值最大的作为把前𝑖个物品装入容量为𝑗的背包中的最优解。

4.动态规划的时间效率为O(nc)其中n表示物品的个数，c表示背包的容量。空间的效率就是用于存储二维数组的占用空间大小，即为O(nc).

#include<stdio.h>

#include<cstdlib>

int V[200][200];//前i个物品装入容量为j的背包中获得的最大价值

int max(int a, int b)

{

if (a >= b)

return a;

else return b;

}

int KnapSack(int n, int w[], int v[], int x[], int C)

{

int i, j;

for (i = 0; i <= n; i++)

V[i][0] = 0;

for (j = 0; j <= C; j++)

V[0][j] = 0;

for (i = 0; i < n; i++){

for (j = 0; j < C+1; j++){

if (j<w[i])

V[i][j] = V[i - 1][j];

else

V[i][j] = max(V[i - 1][j], V[i - 1][j - w[i]] + v[i]);

}

}

j = C;

for (i = n - 1; i >= 0; i--)

{

if (V[i][j]>V[i - 1][j])

{

x[i] = 1;

j = j - w[i];

}

else

x[i] = 0;

}

printf("选中的物品是:\n");

for (i = 0; i<n; i++)

printf("%d ", x[i]);

printf("\n");

for (int i = 0; i < n; i++){

for (int j = 0; j < C+1; j++){

printf("%d\t ", V[i][j]);

if (j == C){

printf("\n");

}

}

}

return V[n - 1][C];

}

int main()

{

int s;//获得的最大价值

int w[5] = {2,2,6,5,4};//物品的重量

int v[5] = {6,3,5,4,6};//物品的价值

int x[5];//物品的选取状态

int n = 5;

int C=10;//背包最大容量

s = KnapSack(n, w, v, x, C);

printf("最大物品价值为:\n");

printf("%d\n", s);

system("pause");

return 0;

}