**动态规划**

http://www.hawstein.com/posts/dp-novice-to-advanced.html

# 概念：

动态规划算法通常基于一个递推公式及一个或多个初始状态。 当前子问题的解将由上一次子问题的解推出。

使用动态规划来解题只需要多项式时间复杂度， 因此它比回溯法、暴力法等要快许多。

通过把原问题分解为相对简单的子问题的方式求解复杂问题的方法。动态规划常常适用于有重叠子问题和最优子结构性质的问题。

**基本思想**

若要解一个给定问题，我们需要解其不同部分（即子问题），再合并子问题的解以得出原问题的解。 通常许多子问题非常相似，为此动态规划法试图仅仅解决每个子问题一次，从而减少计算量： **一旦某个给定子问题的解已经算出，则将其记忆化存储**，以便下次需要同一个子问题解之时直接查表。 这种做法在重复子问题的数目关于输入的规模呈指数增长时特别有用。

**分治与动态规划**

**共同点：**

二者都要求原问题具有最优子结构性质,都是将原问题分而治之,分解成若干个规模较小(小到很容易解决的程序)的子问题.然后将子问题的解合并,形成原问题的解.

**不同点：**

分治法将分解后的子问题看成相互独立的，通过用递归来做。

动态规划将分解后的子问题理解为相互间有联系，有重叠部分，需要记忆，通常用迭代来做。

使用动态规划**特征**：

1. 求一个问题的最优解

2. 大问题可以分解为子问题，子问题还有重叠的更小的子问题

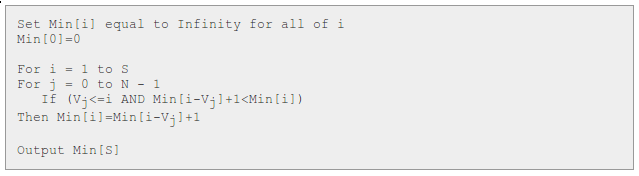
3. 整体问题最优解取决于子问题的最优解（状态转移方程）

4. 从上往下分析问题，从下往上解决问题

5. 讨论底层的边界问题

## 1、硬币问题：

状态转移方程：d(i)=min{ d(i-vj)+1 }，其中i-vj >=0，vj表示第j个硬币的面值;j=1,…,n表示硬币的种类。i表示需要凑齐i元的硬币。



## 2、背包问题

问题描述：

有N件物品和一个容量为V的背包。第i件物品的重量是c[i]，价值是w[i]。求解将哪些物品装入背包可使这些物品的重量总和不超过背包容量，且价值总和最大。

基本思路

这是最基础的背包问题，特点是：每种物品仅有一件，可以选择放或不放。

**简单背包：**

子问题定义状态：即f[i][v]表示前i件物品**恰**放入一个容量为v的背包可以获得的最大价值。则其状态转移方程便是：f[i][v]=max{f[i-1][v],f[i-1][v-c[i]]+w[i]}。

这个方程非常重要，基本上所有跟背包相关的问题的方程都是由它衍生出来的。所以有必要将它详细解释一下：“将前i件物品放入容量为v的背包中”这个子问题，若只考虑第i件物品的策略（放或不放），那么就可以转化为一个只牵扯前i-1件物品的问题。如果不放第i件物品，那么问题就转化为“前i-1件物品放入容量为v的背包中”；如果放第i件物品，那么问题就转化为“前i-1件物品放入剩下的容量为v-c[i]的背包中”，此时能获得的最大价值就是f [i-1][v-c[i]]再加上通过放入第i件物品获得的价值w[i]。

注意f[i][v]有意义当且仅当存在一个前i件物品的子集，其费用总和为v。所以按照这个方程递推完毕后，最终的答案并不一定是f[N] [V]，而是f[N][0..V]的最大值。**如果将状态的定义中的“恰”字去掉，在转移方程中就要再加入一项f[i][v-1]，这样就可以保证f[N] [V]就是最后的答案**。至于为什么这样就可以，由你自己来体会了。

#include<iostream>

#include<iomanip>

using namespace std;

int bag(int \*value, int \*weight, int num,int capac)

{

int \*\*V=new int\*[num+1];

for(int i=0;i<num+1;i++)

{

V[i] = new int[capac+1]();

}

int maxvalue=-1;

for(int n=1;n<num+1;n++)

{

for(int w=1;w<capac+1;w++)

{

int temp = 0;

if(w-weight[n-1]<0) temp = 0;

else temp=V[n-1][w-weight[n-1]]+value[n-1];

V[n][w]=max(temp,V[n-1][w]);

cout<<setw(4)<<V[n][w]<<" ";

if(capac == w) cout<<endl;

if(maxvalue<V[n][w]) maxvalue= V[n][w];

}

}

return maxvalue;

}

void test1()

{

int v[]={6,3,5,4,6};

int w[]={2,2,6,5,4};

cout<<bag(v,w,sizeof(w)/sizeof(int),10)<<endl;

}

int main(){

test1();

}

**完全背包：**

每个物品的数量不限，其他条件相同。

只需要将这句temp=V[n-1][w-weight[n-1]]+value[n-1]代码改成：

temp=V[n][w-weight[n-1]]+value[n-1];

**多重背包：**

多重背包中每个物品的个数都是给定的，可能不是一个，绝对不是无限个。

int bag(int \*value, int \*weight, int \*geshu,int num,int capac)

{

int \*\*V=new int\*[num+1];

for(int i=0;i<num+1;i++)

{

V[i] = new int[capac+1]();

}

int maxvalue=-1;

for(int n=1;n<num+1;n++)

{

for(int w=1;w<capac+1;w++)

{

int temp = 0;

**if(w-weight[n-1]<0) V[n][w]=V[n-1][w];**

**else**

**{**

**int count=min(geshu[n-1],w/weight[n-1]);**

**for(int k=0;k<=count;k++)**

**{**

**int temp1=V[n-1][w-k\*weight[n-1]]+k\*value[n-1];**

**if(temp1>temp) temp=temp1;**

**}**

**V[n][w]=temp;**

**}**

V[n][w]=max(temp,V[n-1][w]);

cout<<setw(4)<<V[n][w]<<" ";

if(capac == w) cout<<endl;

if(maxvalue<V[n][w]) maxvalue= V[n][w];

}

}

**for(int i = 0;i < capac+1; i++)**

**{**

**delete V[i];**

**V[i] = 0;**

**}**

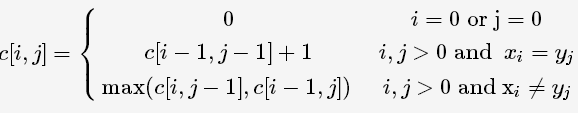
**delete [] V;**

return maxvalue;

}

## 3、最长公共子序列（不连续）（长度）

记c[i][j]为子数组A[0]-A[i]和子数组B[0]-B[j]的最长公共子序列，状态转移方程为：



**编程方法：一副格子图，则c[i][j]格子内的数目只能通过其相邻左上方、上方和左方格子内的数目来确定。**

**技巧：将动态存储空间的大小设为m+1 \* n+1,格子的最上行和最左列均初始化为零。**

public static int lcs(String str1, String str2) {

int len1 = str1.length();

int len2 = str2.length();

int \*\*c = new int\*[len1+1][len2+1];

for(int temp = 0;temp<=len+1;temp++)

{

c[temp] = new int [len2+1];

}

for (int i = 0; i <= len1; i++) {

for( int j = 0; j <= len2; j++) {

if(i == 0 || j == 0) {

c[i][j] = 0;

} else if (str1.charAt(i-1) == str2.charAt(j-1)) {

c[i][j] = c[i-1][j-1] + 1;

} else {

c[i][j] = max(c[i - 1][j], c[i][j - 1]);

}

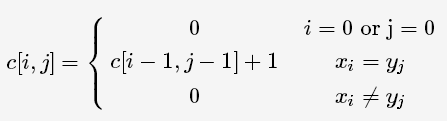
}

}

return c[len1][len2];

}

## 4、最长公共子串（连续）（长度）



public static int lcs(String str1, String str2) {

int len1 = str1.length();

int len2 = str2.length();

int result = 0; //记录最长公共子串长度¨¨

int c[][] = new int[len1+1][len2+1]; //这样动态创建二维数组是不对的

for (int i = 0; i <= len1; i++) {

for( int j = 0; j <= len2; j++) {

if(i == 0 || j == 0) {

c[i][j] = 0;

} else if (str1.charAt(i-1) == str2.charAt(j-1)) {

c[i][j] = c[i-1][j-1] + 1;

result = max(c[i][j], result);

} else {

c[i][j] = 0;

}

}

}

return result;

}

## 5、最小数目的平方数

（类似找硬币题目，找到最小数目的平方数的和等于目标值）给一个正整数 n, 找到若干个完全平方数(比如1, 4, 9, ... )使得他们的和等于 n。你需要让平方数的个数最少。

int leastnumof(int n)

{

int \*p = new int[n+1];

p[0] =0;

for(int i=1;i<=n;i++)

{

int limit = int(sqrt(double(i)));

int min = i;

for(int j=1;j<=limit;j++)

{

int index = i-j\*j;

if(p[index]+1<min) min = p[index]+1;

}

p[i] = min;

}

return p[n];

}

## 6、最长回文字串（连续）且输出

**相当于求字符串p和其倒序的最长公共子串。**

int longesthuiwen(char p[],int len){

int result = 0,end = 0; //记录最长公共子串长度,最长回文串的位置

vector<vector<int>> v(len+1,vector<int>(len+1,0));

for (int i = 1; i <= len; i++) {

for( int j = 1; j <= len; j++){

if (p[i-1] == p[len-j]) {

v[i][j] = v[i-1][j-1] + 1;

if(v[i][j]>result){

result =v[i][j];

end = i-1;

}

}

}

}

for(int m=0;m<result;m++){

cout<<p[end-result+1+m];

}

cout<<endl;

return result;

}

## 7、最长回文序列（不连续且不输出回文子序列）

int longesthuiwen(char p[],int len){

vector<vector<int>> v(len+1,vector<int>(len+1,0));

vector<int> index;

int maxnum =0;

for (int i = 1; i <= len; i++) {

for( int j = 1; j <= len; j++){

if (p[i-1] == p[len-j]){

v[i][j] = v[i-1][j-1] + 1;

}

else v[i][j] = max(v[i-1][j],v[i][j-1]);

}

}

return v[len][len];

}

## 8、最长非降子序列

一个序列有N个数：A[1],A[2],…,A[N]，求出**最长非降子序列**的长度。

**解：**

**方法一：**首先，我们要找到某个状态的最优解，然后在它的帮助下，找到下一个状态的最优解。

用d[i]来存储当元素A[i]为数组的最后一个元素时，非降序列的最长长度；当元素A[i+1]是数组最后一个元素时，要用A[i+1]与前面的i个元素逐个比较，若有A[i+1]>=A[i]&&d[i+1]<d[i]+1时，则d[i+1]=d[i]+1。

#include <iostream>

using namespace std;

int lis(int A[], int n){

int \*d = new int[n];

int len = 1;

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

d[i] = 1;

for(int j=0; j<i; ++j)

if(A[j]<=A[i] && d[j]+1>d[i])

d[i] = d[j] + 1;

if(d[i]>len) len = d[i];

}

delete[] d;

return len;

}

int main(){

int A[] = {

5, 3, 4, 8, 6, 7

};

cout<<lis(A, 6)<<endl;

return 0;

}

**方法二**：用动态数组B来存储对应长度的LIS的最小元素末尾，每次遇到新元素A[i]时，二分查找B中第一个大于等于A[i]的元素，如果都小于A[i]则返回B的end+1，将值放入到B[返回值]。

**int** upper\_bound(**int** arr[], **int** s, **int** e, **int** key)  
{  
    **int** mid;  
    **if** (arr[e] <= key)  
        **return** e + 1;  
    **while** (s < e)  
    {  
        mid = s + (e - s) / 2;  
        **if** (arr[mid] <= key)  
            s = mid + 1;  
        **else**  
            e = mid;  
    }  
    **return** s;  
}  
  
**int** LIS(**int** d[], **int** n)  
{  
    **int** i = 0, len = 1, \*end = (**int** \*)alloca(**sizeof**(**int**) \* (n + 1));  
    end[1] = d[0]; //初始化：长度为1的LIS末尾为d[0]  
    **for** (i = 1; i < n; i++)  
    {  
        **int** pos = upper\_bound(end, 1, len, d[i]); //找到插入位置  
        end[pos] = d[i];  
        **if** (len < pos) //按需要更新LIS长度  
            len = pos;  
    }  
    **return** len;  
}

**方法三**：排序，再与原序列求最长子序列。

## 9、楼层抛珠问题

某幢大楼有100层。你手里有两颗一模一样的玻璃珠。当你拿着玻璃珠在某一层往下扔的时候，一定会有两个结果，玻璃珠碎了或者没碎。这幢大楼有个临界楼层。低于它的楼层，往下扔玻璃珠，玻璃珠不会碎，等于或高于它的楼层，扔下玻璃珠，玻璃珠一定会碎。玻璃珠碎了就不能再扔。现在让你设计一种方式，使得在该方式下，最坏的情况扔的次数比其他任何方式最坏的次数都少。也就是设计一种最有效的方式。

根据题意很容易写出状态转移方程：N层楼如果从n层投下玻璃珠，最坏的尝试次数是：clip_image002[6]

那么所有层投下的最坏尝试次数的最小值即为问题的解：clip_image002[8]。其中F(1)=1.

5 #include<iostream>

6 using namespace std;

7

8 int max(int a, int b)

9 {

10 return (a > b)? a : b;

11 }

12

13 int dp[101];

14 //N<=100;

15 int floorThr(int N)

16 {

17 for (int i = 2; i <= N; i++)

18 {

19 dp[i] = i;

20 for (int j = 1; j<i; j++)

21 {

22 int tmp = max(j, 1 + dp[i - j]); //j的遍历相当于把每层都试一遍

23 if (tmp<dp[i])

24 dp[i] = tmp;

25 }

26 }

27 return dp[N];

28 }

29

30 int main()

31 {

32 dp[0] = 0;

33 dp[1] = 1;

34 int dis = floorThr(100);

35 cout << dis << endl;

36 system("Pause");

37 }