第11次实验

## 1. 动态规划

**(1)划分阶段**

动态规划用于求解**多阶段决策**问题。它是把整个问题划分为若干阶段后，依次为每一个阶段作出最优决策。使用动态规划有一个条件，就是后续阶段的最优决策恰好建立在以前阶段最优决策（不一定是前一个）的基础之上，这样当处理完最后一个阶段，问题的解也就随之产生。

一个阶段处理的问题与前一个阶段处理的问题一般具有极大的相似性。而且后一个阶段的问题一般是前一个阶段问题的扩展，这样随着逐个处理各个阶段，最后才能得到整个问题的全局最优解。

可以寻找原问题所包含的相似的一系列子问题（即子结构），这些子问题逐步解决就可找到原问题的解。子问题往往对应不同的**阶段。**

**(2)效益函数、状态变量**

划分了问题的阶段之后，一般都要考虑当前阶段所要解决的问题。这个问题往往是一个最优化问题，第*k*个阶段的对应一般可表示为*fk*(*sk*)的最优化问题。这里函数*fk*一般称为效益函数。效益函数的取值依赖于当前阶段的某些数值*sk*，*sk*就是所谓状态变量。

**(3)决策变量**

在每个阶段向下一个阶段演化过程中，一般需要我们做出某种决策，我们将其记作u*k*。第*k*个阶段的效益*fk*(*sk*)在不同的决策u*k*的作用之下产生不同的*fk*+1(*sk*+1)。从第*k*个阶段向第*k*+1个阶段演化过程中，一般可以总结出一个方程，即状态转移方程。

有了以上信息，就可以编写方程，一步步求出最优解。

## 2. 挖金矿问题

假设有多个金矿，其编号、产量见下表：

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 编号 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| 产金量  人数 | 800  50 | 1500  70 | 6700  800 | 5800  760 | 5678  800 | 1200  120 | 500  30 | 900  49 | 7000  1000 | 8888  1500 |

要求：

* 每座金矿必须挖或者不挖，不能只挖一部分；
* 每个工人只能分配到一个金矿劳动。

已知总人数：10000，如果希望挖到的金子越多越好，请问那些金矿需要挖？

**分析：**

1、阶段如何划分？

显然可以按挖了几个矿来划分，挖了1个、挖了2个、挖了3个、……、等等。初学者一个典型的问题是按下面方法分阶段：挖第1个、挖第2个、挖第3个、……，这表明对动态规划理解不够深入的表现。动态规划的后一个阶段应该包含前一个阶段的处理对象，后一个阶段的问题一般是前一个阶段问题的扩展。我们可以这样划分：挖前1个、挖前2个、挖前3个、……。

注意，挖前*k*个矿（第*k*阶段），是指将前*k*个矿通盘考虑，某一个矿可能挖，也可能不挖。若挖掘前*k*个矿的最佳挖掘策略是（d0，d1，……，dk-1），其中d*i*为true或false，表示第*i*个矿是否挖掘，那么若前*k+*1个矿的最佳挖掘策略**不一定是**（d0，d1，……，dk-1，dk）。 2、效益函数、状态转移方程

可以用*f*(*k-*1, 剩余人数)表示第*k*阶段效益函数，即挖了前*k*个金矿可以挖到多少金子。函数*f*的第一个参数为金矿标号，从0开始。这里在不同阶段，剩余人数是不同的，所以效益函数一定和人数有关。而每个矿的产金量、所需人数全是事先设定的，不应作为*f*函数的自变量。

考虑可能挖掘前*k*个金矿，*f* ( *k*-1*, peopleNum* )确定了以后，这是要考虑第*k+*1座金矿的问题。显然，对于第*k+*1座金矿（标号*k*）而言，挖掘前*k+*1个金矿的最优解有两种情况：

1. *peopleNum* 数量挖第*k+*1座金矿不够

这时，当前这座矿不可能挖，于是

*f* ( *k, peopleNum* ) = *f* ( *k*-1*, peopleNum* )

1. *peopleNum* 数量挖第*k+*1座金矿够了

这时，又有两种选择：挖掘第 *k*+1 座、不挖掘第 *k*+1座。最优解必然是这两种情形的下最佳解答中最优的一个。

情形一：**不挖**第 k+1 座金矿的最优解是

*f* ( *k, peopleNum* ) = *f* ( *k*-1 *, peopleNum* ) // 只考察前k座矿即可

情形二：**挖**第 k+1 座金矿的最优解中是

*f ( k, peopleNum ) =*

V[k] + 人数为[总人数-第k座需要人数]时，前k-1座矿最大效益

这里，V[k]是第k座矿的金币数量。

而人数为[总人数-第k座需要人数]时，前k-1座矿最大效益=

*f* ( *k-*1*, peopleNum – peopleNeed*[k])

于是，状态转移方程：

*f* (*k, peopleNum* ) *=* ***Max***( *f* ( *k, peopleNum* ) = *f* ( *k-*1 *, peopleNum* )，

V[k] +*f* ( *k-*1 *, peopleNum – peopleNeed*[k]) )

* **解法1：**

在计算*f* (*k, peopleNum* )时，可能要用到*f* ( *k-*1 *, peopleNum – peopleNeed*[k])，可是*peopleNum – peopleNeed*[k]是多少？？但是*peopleNum – peopleNeed*[k]一定是个小于等于总人数*peopleNum*的值。如果我们这么做：

先计算出*f* (*k-*1 *,* 0)、*f* (*k-*1 *,* 1)、*f* (*k-*1 *,* 2)、……、*f* (*k-*1 *,* 10000)，

那么，*f* ( *k-1 , peopleNum* )一定在上面一行中可以找到，

*f* ( *k-*1 *, peopleNum – peopleNeed*[k])也一样可以找到

于是，我们可计算出*f* (*k ,* 0)、*f* (*k ,* 1)、*f* (*k ,* 2)、……、*f* (*k ,* 10000)

依次类推，最后一行最后一个就是我们要的结果。

**伪代码：**

//maxGold[i][j]保存了*j*个人挖前*i+*1个金矿能够得到的最大金子数

//maxGold[i][j] 就是存储上面的*f* (*i , j*)

int maxGold[100][10001]; **//数组太大，定义为全局变量，放在函数外！**

int n = 10; //金矿数

int peopleTotal=10000; //可以用于挖金子的总人数

//每座金矿需要的人数

int peopleNeed[100]={50, 70, 800, 760, 800, 120, 30, 49, 1000,1500};

//每座金矿能够挖出来的金子数

int gold[100] = {800, 1500,6700,5800,5678,1200,500,900,7000,8888};

为maxGold[0][0]~ maxGold[0][10000]赋值 //人数为0~10000时，挖1个矿，能得多少

（注：maxGold[0][0]~ maxGold[0][49]为0，maxGold[0][50]~ maxGold[0][10000]为800）

// 逐行向下计算maxGold[ ][ ]

for(int i=1;i<10;i++) // 挖前i个矿

{

for(int j=0;j<=peopleTotal;j++) // 人数j从0~ peopleTotal

{

int peoLeft=j - peopleNeed[i]; // 计算j与peopleNeed[i]之差

if(peoLeft>=0) //人数够挖标号为i-1的矿

{

利用状态转移函数，借助maxGold[i-1][j]、maxGold[i-1][peoLeft]计算 当前矿的含金量计算maxGold[i][j]

}else{ //人数不够挖标号为i-1的矿

maxGold[i][j] 等同于矿数-1，人数不变的情况；

}

}

}

输出结果；

----------------------------------------------------------------------------------------------------------------------

* **解法2：**

这里其实是对解法1做一定修改得到的。算法1计算了maxGold[][]每一项，这里先将maxGold[][]每一项设为-1（第一行除外，还是按上面方法初始化）。然后，将状态转换函数

*f* (*k, peopleNum* ) *=* ***Max***( *f* ( *k, peopleNum* ) = *f* ( *k-*1 *, peopleNum* )，

V[k] +*f* ( *k-*1 *, peopleNum – peopleNeed*[k]) )

用递归函数实现。

* 计算*f* (*k, peopleNum* )时先查矩阵maxGold[*k*][ *peopleNum*]，如果是-1，就递归计算*f* (*k, peopleNum* )，计算完成后将*f* (*k, peopleNum* )存入maxGold[*k*][ *peopleNum*]，以便后续使用这个数值时可以直接查询maxGold[*k*][ *peopleNum*]即可。
* 如果发现某一项maxGold[*k*][ *peopleNum*]不是-1，则直接取出使用即可。

**伪代码：**

定义n——金矿数、peopleTotal——可以用于挖金子的人数、peopleNeed[]——每座金矿需要的人数、gold[]——每座金矿的金子数，和前面的做法一样。

定义maxGold[][]。maxGold[i][j]保存了j个人挖前i+1个金矿能够得到的最大金子数，等于-1时表示未知

// 递归函数，有people个人和前mineNum个金矿时能够得到的最大金子数，

// 注意mineNum是从0开始编号的

int **GetMaxGold**( int mineNum, int people)

{

int retMaxGold;

//如果这个问题曾经计算过 [对应动态规划中的“做备忘录”]

if (maxGold[mineNum][people] != -1)

{

返回 maxGold[mineNum][people] 的值

}

else if( people >= peopleNeed[mineNum] ) //如果人数够开采这座金矿

{

挖当前矿的最大值 maxGoldIfDig = gold[mineNum]+

**GetMaxGold**(mineNum-1, people - peopleNeed[mineNum]); //递归

不挖当前矿的最大值 maxGoldIfNotDig =

矿数为mineNum-1人数为people情形取值； //递归

取retMaxGold 为 maxGoldIfNotDig 和 maxGoldIfDig 的最优

}

else

{ //人数不够开采这座金矿

retMaxGold 取值等同于矿数为mineNum-1人数为people情形； //递归

}

maxGold[mineNum][people] = retMaxGold; //做备忘录，写入矩阵

return retMaxGold;

}

int main(int argc, char\*\* argv)

{

初始化数据，将maxGold[][]都设为-1；

为maxGold[0][0]~ maxGold[0][10000]赋值 //人数为0~10000挖1个矿，能得多少

（人数<50得0金币，超过50得800）

输出GetMaxGold( n-1, peopleTotal); // n座矿，人数为peopleTotal

return 0;

}

### 实验1

用上面解法1或解法2编写程序，解决挖金矿问题，用本节开始所给出的数据验证。

## 3. 01背包问题

小明要远行，他有一个容量为15kg的背包，另外有4个物品，物品的质量和价值如下表所示：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **物品** | A1 | A2 | A3 | A4 |
| **质量*w*** | 3 | 4 | 5 | 6 |
| **价值*v*** | 4 | 5 | 6 | 7 |

小明希望能用他的背包带走的物品总价值最大，你能告诉他应该怎么做吗？

**0-1背包 等价于 挖金子**

|  |  |  |  |
| --- | --- | --- | --- |
|  | 替换1 | 替换2 | 替换3 |
| 挖金子 | 金矿金子 | 人员需求 | 已知总人数 |
| 0-1背包 | 物品价值 | 物品重量 | 背包载重量 |

教材中将阶段划分从最后一件物品开始，第1阶段可选择物品n，第2阶段可选择物品n-1、n，第3阶段可选择物品n-2、n-1、n，…….，第n阶段可选择物品1、2、……、n。

* 设m(i, j)是背包容量为j，可选择物品为i，i+1，……，n时的最优值。
* 可以建立计算m(i, j)的递归式如下：



假定第*i+*1行的m(i, j)的值已经全部知道，则第*i* 行的m(i, j)有两种情形:

(1) 容量j小于第i件物品重量

这时，第i件物品不可能装进去

m(i, j) = m(i+1, j)

(2) 容量j大于等于第i件物品重量

这时，第i件物品可以装进去。

最优解有可能第i件物品装进去了，于是 m(i, j) = m(i+1, j-*w*i)+*v*i

或者第i件物品不必装进去，于是 m(i, j) = m(i+1, j)

我们用数组c[][]存储m(i, j)的值。

对应(1)有：

jMax=min(w[i]-1,m);

for(j=0;j<=jMax;j++) c[i][j]=c[i+1][j];

对应(2)有：

for(j=w[i];j<=m;j++) c[i][j]=max(c[i+1][j],c[i+1][j-w[i]]+v[i]);

---------------------------------------------------------------------------------------------------------------

/\*构造最优解，即到底拿了那些物品。\*/

// x[i]——1 拿了第i件物品

// x[i]——0 没有拿第ii件物品

void traceBack(int \*x)

{

int i,j;

j=m; // 从最大容量开始搜索

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

{

//前后两个阶段，容量相同时价值也一样，说明最后处理的物品i 没有放入

if(c[i][j]==c[i+1][j]) x[i]=0;

else { //前后两阶段，容量相同时价值不一样，说明最后处理的物品i放入

x[i]=1; j-=w[i];

}

}

x[n]=(c[n][j])?1:0;

}

### 实验2

用上面动态规划法编写程序，解决01背包问题，用本节开始所给出的数据验证。