导航
博客园
首页
新随笔
联系
订阅XML
管理

< 2018年6月 >
日 一 二 三 四 五 六
27 28 29 30 31 1 2
3 4 5 6 7 8 9
10 11 12 13 14 15 16
17 18 19 20 21 22 23
24 25 26 27 28 29 30
1 2 3 4 5 6 7

常见的动态规划问题分析与求解

动态规划（Dynamic Programming，简称DP），虽然抽象后进行求解的思路并不复杂，但具体的形式千差万别，找出问题的子结构以及通过子结构重新构造最优解的过程很难统一，并不像回溯法具有解决绝大多数问题的银弹（全面解析回溯法：算法框架与问题求解）。为了解决动态规划问题，只能靠多练习、多思考了。本文主要是对一些常见的动态规划题目的收集，希望能有所帮助。难度评级受个人主观影响较大，仅供参考。

目录（点击跳转）

28 1

现代操作系统(4)
Linux(3)
分治法(2)
细节(2)
更多

随笔分类(266)
C(31)
C++(3)
DB(1)
Git(1)
Java(7)
Linux/Unix(31)
Linux内核(7)
nginx(4)
Python(3)
Simulink(6)
笔试面试题(11)
编程之美(5)
操作系统(10)
机器学习/神经网络(3)
面向对象(1)
嵌入式(2)
软件开发(10)
算法(23)
网络编程(19)
学习笔记(56)
珠玑之棧(7)
资料收集(25)

随笔档案(116)
2017年6月 (1)
2015年11月 (1)
2015年5月 (1)
2015年4月 (1)
2015年3月 (2)
2015年2月 (2)
2015年1月 (1)
2014年7月 (1)
2014年4月 (1)
2014年2月 (3)
2013年12月 (2)
2013年11月 (8)
2013年10月 (3)
2013年9月 (2)
2013年8月 (12)
2013年7月 (9)
2013年6月 (9)
2013年5月 (4)
2013年4月 (3)
2013年3月 (5)
2013年2月 (1)
2013年1月 (1)
2012年12月 (3)
2012年11月 (1)
2012年10月 (1)
2012年9月 (2)
2012年8月 (2)
2012年7月 (3)
2012年6月 (3)
2012年5月 (3)
2012年4月 (1)

附录2：《算法设计手册》第八章 动态规划 面试题解答

动态规划求解的一般思路：

判断问题的子结构（也可看作状态），当具有最优子结构时，动态规划可能适用。

求解重叠子问题。一个递归算法不断地调用同一问题，递归可以转化为查表从而利用子问题的解。分治法则不同，每次递归都产生新的问题。

重新构造一个最优解。

备忘录法：

动态规划的一种变形，使用自顶向下的策略，更像递归算法。

初始化时表中填入一个特殊值表示待填入，当递归算法第一次遇到一个子问题时，计算并填表；以后每次遇到时只需返回以前填入的值。

实例可以参照矩阵链乘法部分。

1.硬币找零

难度评级：★

假设有几种硬币，如1、3、5，并且数量无限。请找出能够组成某个数目的找零所使用最少的硬币数。

解法：

用待找零的数值k描述**子结构/状态**，记作sum[k]，其值为所需的最小硬币数。对于不同的硬币面值coin[0...n]，有sum[k] = min(sum[k-coin[0]] , sum[k-coin[1]], ...)+1。对应于给定数目的找零total，需要求解sum[total]的值。

```
typedef struct {
    int nCoin; //使用硬币数量
    //以下两个成员是为了便于构造出求解过程的展示
    int lastSum;//上一个状态
    int addCoin;//从上一个状态达到当前状态所用的硬币种类
} state;
```

```
state *sum = malloc(sizeof(state)*(total+1));

//init
for(i=0;i<=total;i++)
    sum[i].nCoin = INF;
sum[0].nCoin = 0;
sum[0].lastSum = 0;

for(i=1;i<=total;i++)
    for(j=0;j<n;j++)
        if(i-coin[j]>=0 && sum[i-coin[j]].nCoin+1<sum[i].nCoin)
        {
            sum[i].nCoin = sum[i-coin[j]].nCoin+1;
            sum[i].lastSum = j;
            sum[i].addCoin = coin[j];
        }

    if(sum[total].nCoin == INF)
    {
```

2012年1月 (3)
2011年12月 (1)
2011年11月 (3)
2011年9月 (3)
2011年8月 (3)
2011年7月 (11)

相册(31)
《大话设计模式》配图(24)
malloc配图(2)
VC维(3)
vi(1)
博客贴图(1)

常用资料
Linux Cross Reference
Linux在线手册
Markdown语法
Markdown在线编辑器
MATLAB中文论坛
在线C定义解释
可以解释各类型的定义，简单的如整型、浮点型，复杂的如数组、函数指针等。
在线LaTex转换
在线Shell (1)
在线Shell (2)
在线进制转换

友链
dariusdong
Shen Fan(范深)

积分与排名
积分 - 206953
排名 - 1230

最新评论
1. Re:gdb调试命令
就服你总结的
--LewisChan
2. Re:通过JDBC进行简单的增删改查（以MySQL为例）
看完之后瞬间清晰，楼主真的厉害
--一曲新词酒一杯
3. Re:通过JDBC进行简单的增删改查（以MySQL为例）
不错不错，收藏了。推荐下，分库分表中间件 Sharding-JDBC 源码解析 17 篇：&601 补...
--长安怎乱

阅读排行榜

```
        printf("can't make change.\n");
        return 0;
    }
    else
        //output
       ;
```

通过sum[total].lastSum和sum[total].addCoin，很容易通过循环逆序地或者编写递归调用的函数正序地输出从结束状态到开始状态使用的硬币种类。以下各题输出状态转换的方法同样，不再赘述。下面为了方便起见，有的题没有在构造子结构的解时记录状态转换，如果需要请类似地完成。

扩展：

(1)一个矩形区域被划分为N*M个小矩形格子，在格子(i,j)中有A[i][j]个苹果。现在从左上角的格子(1,1)出发，要求每次只能向右走一步或向下走一步，最后到达(N,M)，每经过一个格子就把其中的苹果全部拿走。请找出能拿到最多苹果数的路线。

难度评级：★

分析：

这道题中，当前位置(i,j)是**状态**，用M[i][j]来表示到达状态(i,j)所能得到的最多苹果数，那么M[i][j] = max(M[i-1][j],M[i][j-1]) + A[i][j] 。特殊情况是M[1][1]=A[1][1]，当i=1且j!=1时,M[i][j] = M[i][j-1] + A[i][j]；当i!=1且j=1时M[i][j] = M[i-1][j] + A[i][j]。

求解程序略。

(2)装配线调度（《算法导论》15.1）

难度评级：★

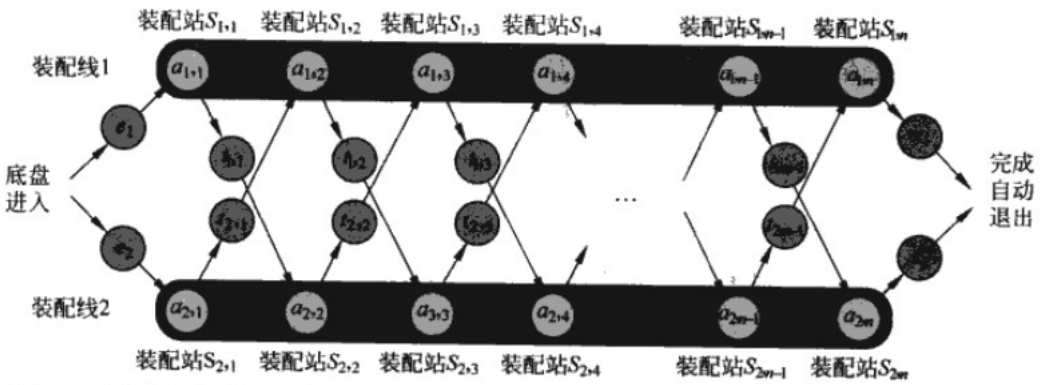


图 15-1 一个找出通过工厂装配线的最快方式的制造问题。共有两条装配线，每条有 n 个装配站；装配线 i 的第 j 个装配站表示为 $S_{i,j}$，在该站的装配时间是 $a_{i,j}$。一个汽车底盘进入工厂，然后进入装配线 i(i 为 1 或 2)，花费时间 e_i。在通过一条线的第 j 个装配站后，这个底盘来到任一条线的第(j+1)个装配站。如果它留在相同的装配线，则没有移动的开销；但是，如果在装配站 $S_{i,j}$ 后，它移动到了另一条线上，则花费时间 $t_{i,j}$。在离开一条线的第 n 个装配站后，完成的汽车花费时间 x_i 离开工厂。待求解的问题是确定应该在装配线 1 内选择哪些站、在装配线 2 内选择哪些站，才能使汽车通过工厂的总时间最小

2.字符串相似度/编辑距离（edit distance）

难度评级：★

1. 通过JDBC进行简单的增删改查（以MySQL为例）(326205)
2. 常见的动态规划问题分析与求解(68265)
3. 轻松记住大端小端的含义（附对大端和小端的解释）(55991)
4. 手把手教你编写一个具有基本功能的shell（已开源）(54554)
5. 《大话设计模式》Python版代码实现(35630)

评论排行榜

1. 手把手教你编写一个具有基本功能的shell（已开源）(23)
2. 如何写出正确的二分查找？——利用循环不变式理解二分查找及其变体的正确性以及构造方式(21)
3. 通过JDBC进行简单的增删改查（以MySQL为例）(17)
4. malloc()参数为0的情况(15)
5. 《C陷阱与缺陷》学习笔记（上）：词法陷阱、语法陷阱、语义陷阱(14)

推荐排行榜

1. 通过JDBC进行简单的增删改查（以MySQL为例）(70)
2. 常见的动态规划问题分析与求解(28)
3. 《大话设计模式》Python版代码实现(27)
4. 从《编程之美》买票找零问题说起，娓娓道来卡特兰数——兼爬坑指南(22)
5. 轻松记住大端小端的含义（附对大端和小端的解释）(17)

对于序列S和T，它们之间距离定义为：对二者其一进行几次以下的操作(1)删去一个字符；(2)插入一个字符；(3)改变一个字符。每进行一次操作，计数增加1。将S和T变为同一个字符串的最小计数即为它们的距离。给出相应算法。

解法：

将S和T的长度分别记为len(S)和len(T)，并把S和T的距离记为m[len(S)][len(T)]，有以下几种情况：

如果末尾字符相同，那么m[len(S)][len(T)]=m[len(S)-1][len(T)-1]；

如果末尾字符不同，有以下处理方式

修改S或T末尾字符使其与另一个一致来完成，m[len(S)][len(T)]=m[len(S)-1][len(T)-1]+1；

在S末尾插入T末尾的字符，比较S[1...len(S)]和S[1...len(T)-1]；

在T末尾插入S末尾的字符，比较S[1...len(S)-1]和S[1...len(T)]；

删除S末尾的字符，比较S[1...len(S)-1]和S[1...len(T)]；

删除T末尾的字符，比较S[1...len(S)]和S[1...len(T)-1]；

总结为，对于i>0,j>0的状态(i,j),m[i][j] = min(m[i-1][j-1]+(s[i]==s[j])?0:1 , m[i-1][j]+1, m[i][j-1] +1)。

这里的**重叠子结构**是S[1...i]，T[1...j]。

以下是相应代码。考虑到C语言数组下标从0开始，做了一个转化将字符串后移一位。

```
字符串相似度/edit distance
```

应用：

(1)子串匹配

难度评级：★★

修改两处即可进行子串匹配：

```
修改部分
```

如果j= strlen(S) - strlen(T)，那么说明T是S的一个子串。

（这部分是根据《算法设计手册》8.2.4和具体实例Skiena与Skienaa、Skiena与somta的分析获得的，解释不够全面，可能有误，请注意）

(2)最长公共子序列

难度评级：★★

将match时不匹配的代价转化为最大长度即可：

```
match()
```

此时，最小值是两者不同部分的距离。

（这部分同样也不好理解，对于最长公共子序列，建议直接使用下一部分中的解法）

扩展：

如果在编辑距离中各个操作的代价不同，如何寻找最小代价？

28 1

3.最长公共子序列(Longest Common Subsequence,lcs)

难度评级：★

对于序列S和T，求它们的最长公共子序列。例如X={A,B,C,B,D,A,B}，Y={B,D,C,A,B,A}则它们的lcs是{B,C,B,A}和{B,D,A,B}。求出一个即可。

解法：

和2类似，对于X[1...m]和Y[1...n]，它们的任意一个lcs是Z[1...k]。

(1)如果X[m]=Y[n]，那么Z[k]=X[m]=Y[n],且Z[1...k-1]是X[1...m-1]和Y[1...n-1]的一个lcs；

(2)如果X[m]!=Y[n]，那么Z[k]!=X[m]时Z是X[1...m-1]和Y的一个lcs；

(3)如果X[m]!=Y[n]，那么Z[k]!=Y[n]时Z是X和Y[1...n-1]的一个lcs；

下面是《算法导论》上用伪码描述的lcs算法。其中c[i][j]记录当前lcs长度，b[i][j]记录到达该状态的上一个状态。

```
LCS-LENGTH(X, Y)
 1  m = X.length
 2  n = Y.length
 3  let b[1..m, 1..n] and c[0..m, 0..n] be new tables
 4  for i = 1 to m
 5      c[i, 0] = 0
 6  for j = 0 to n
 7      c[0, j] = 0
 8  for i = 1 to m
 9      for j = 1 to n
10          if x_i == y_j
11              c[i, j] = c[i − 1, j − 1] + 1
12              b[i, j] = "↖"
13          elseif c[i − 1, j] ≥ c[i, j − 1]
14              c[i, j] = c[i − 1, j]
15              b[i, j] = "↑"
16          else c[i, j] = c[i, j − 1]
17              b[i, j] = "←"
18  return c and b
```

扩展1：

如何输出所有的LCS?

难度评级：★★

分析：

根据上面c[i,j]和b[i,j]的构造过程可以发现如果c[i-1,j]==c[i,j-1]，那么分别向上和向左返回的上一个状态都是可行的。如果将其标记为"左/上"并通过递归调用来生成从c[m,n]到c[1,1]的所有路径，就能找出所有的LCS。时间复杂度上界为O(mn)。

扩展2：

通过LCS获得最长递增自子序列。

分析：

对于1个序列，如243517698，最大值9，最小值1，那么通过将它与123456789求LCS得到的就是最长连续递增子序列23568。

这种做法不适用于最长连续非递减子序列，除非能获得重复最多的元素数目，如2433517698，那么可以用112233445566778899与之比较。

使用专门的最长递增子序列算法可以进行优化，详见下一部分。

4.最长递增子序列（Longest Increasing Subsequence,lis）

难度评级：★

对于一个序列如1，-1，2，-3，4，-5，6，-7，其最长第增子序列为1,2,4,6。

解法：

除了利用3中lcs来求解，这里使用求解lis问题的专门方法。

先看看如何确定**子结构**的表示。对于长度为k的序列s[1...k]，如果用lis[k]记录这个序列中最长子序列似乎没什么用，因为在构造lis[k+1]时，需要比较s[k]与前面长度为lis[k]的lis的最后一个元素、s[1...k]中长度为lis[k]-1的序列的最后一个元素等等，没有提供什么便利，这个方案被否决。

为了将每个lis[k]转化为构造lis[k+1]时有用的数据，把**子结构**记为以s[k]为结尾的lis的长度，那么对于s[k+1]，需要检查所有在它前面且小于它的元素s[i]，并令lis[k+1] = max(lis[i]+1)，(i=1 to k,s[k+1]>s[i])。这样，一个$O(n^2)$的算法便写成了。为了在处理完成后不必再一次遍历lis[1...n]，可以使用一个MaxLength变量保存当前记录中最长的lis。

```
typedef struct {
    int length;
    int prev;
} state;

//算法核心
state *a = malloc(sizeof(state) * n);
for(i=0;i<n;i++) {
    a[i].length = 1;
    a[i].prev = -1;
}

for(i=1;i<n;i++)
    for(j=1;j<i;j++)
        if(array[i]>array[j] && a[i].length < a[j].length + 1)
        {
            a[i].length = a[j].length + 1;
            a[i].prev = j;
            if(a[i].length > max_length) {
                max_length = a[i].length;
                max_end = i;
            }
        }
```

扩展：

求解lis的加速

难度评级：★★

分析：

在构造lis[k+1]的时候可以发现，对于s[k+1]，真正有用的元素s[i]<s[k+1]且lis[i]最大。如果记录了不同长度的lis的末尾元素，那么对于新加入的元素s[k+1]，找出前面比它小的且对应lis最长的，就是以s[k+1]为结尾的lis[k+1]的长度。

可以发现使用数组MaxV[1...MAXLENGTH]其中MaxV[i]表示长度为i的lis的最小末尾元素，完全可以lis[k+1]的更新。进一步地发现，其实lis[]数组已经没有用了，对于MaxV[1...MAXLENGTH]中值合法对是当前最长的lis，也即利用MaxV[]更新自身。

28 1

同时，根据MaxV[]的更新过程，可以得出当i<j时，MaxV[i]<MaxV[j]（假设出现了i>j且Max[i]=>Max[j]的情况，那么在之前的处理中，在发现j长度的lis时，应用它的第i个元素来更新Max[i]，仍会导致MaxV[i]<MaxV[j]，这与这个现状发生了矛盾，也即这个情况是不可能到达的）。这样，在寻找小于s[k+1]的值时，可以使用二分查找，从而把时间复杂度降低至O(nlogn)。

```
int lis_ologn(int *array, int length) {
    int i, left,right,mid,max_len = 1;
    int *MaxV;
    if(!array)
        return 0;
    MaxV = (int*)malloc(sizeof(int)*(length+1));
    MaxV[0] = -1;
    MaxV[1] = array[0];

    for(i=1;i<length;i++){
        //寻找范围是MaxV[1, ... , max_len]
        left = 1;
        right = max_len;
        //二分查找MaxV中第一个大于array[i]的元素
        while(left<right) {
            mid = (left+right)/2;
            if(MaxV[mid]<=array[i])
                left = mid + 1;
            else if(MaxV[mid]>array[i])
                right = mid;
        }
        if((MaxV[right]>array[i])&&(MaxV[right-1]<array[i]))
            MaxV[right] = array[i];
        else if (MaxV[right]<array[i]) {
            MaxV[right+1] = array[i];
            max_len++;
        }
    }
    return max_len;
}
```

在这个解法下，不妨考虑如何重构这个lis。

5.最大连续子序列和/积

难度评级：★

输入是具有n个数的向量x，输出时输入向量的任何连续子向量的最大和。

解法：

求和比较简单，以前写过比较全面的分析：http://www.cnblogs.com/wuyuegb2312/p/3139925.html#title4

这里只把O(n)的动态规划解法列在下面，其中只用一个变量保存过去的**状态**：

```
int max_array_v4(int *array,int length) {
    int i;
    int maxsofar = NI;
    int maxendinghere = 0;
    for(i=0;i<length;i++) {
        maxendinghere = maxnum(maxendinghere + array[i],array[i]);
        //分析：maxendinghere必须包含array[i]
        //当maxendinghere>0且array[i]>0，maxendinghere更新为两者和
        //当maxendinghere>0且array[i]<0，maxendinghere更新为两者和
```

28 1

```
        //当maxendinghere<0且array[i]<0, maxendinghere更新为array[i]
        //当maxendinghere<0且array[i]>0, maxendinghere更新为array[i]
        maxsofar = maxnum(maxsofar,maxendinghere);
    }
    return maxsofar;
}
```

扩展1：

难度评级：★

给定一个正浮点数数组，求它的一个最大连续子序列乘积的值。

解法：

对数组中每个元素取对数，构成新的数列，在新的数列上使用求最大连续子序列的算法。

如果求对数开销较大，建议使用扩展2的方法。

扩展2：

难度评级：★

给定一个浮点数数组，其值可正可负可零，求它的一个最大连续子序列乘积的值。（假定计算过程中，任意一个序列的积都不超过浮点数最大表示）

解法：

在最大连续子序列和算法的基础上进行修改。由于负负得正，对于当前状态array[k]，需要同时计算出它的最大值和最小值。即：

new_maxendinghere = max3(maxendinghere*array[k],minendinghere*array[k],array[k])

new_minendinghere = min3(maxendinghere*array[k],minendinghere*array[k],array[k])

此后对已遍历部分的最大积进行更新：

maxsofar = max(maxsofar,new_maxendinghere)

如果不习惯用常数个变量来表示，可以看看http://blog.csdn.net/wzy_1988/article/details/9319897，再想想用数组保存是不是浪费了空间。（计算max[k]、min[k]只用到了max[k-1]、min[k-1]，没有必要保存全部状态）

6.矩阵链乘法

难度评级：★

一个给定的矩阵序列$A_1A_2...A_n$计算连乘乘积，有不同的结合方法，并且在结合时，矩阵的相对位置不能改变，只能相邻结合。根据矩阵乘法的公式，10*100和100*5的矩阵相乘需要做10*100*5次标量乘法。那么对于维数分别为10*100、100*5、5*50的矩阵A、B、C，用(A*B)*C来计算需要10*100*5 + 10*5*500 =7500次标量乘法；而A*(B*C)则需要100*5*50+10*100*50=75000次标量乘法。

那么对于由n个矩阵构成的链$<A_1,A_2,...,A_n>$，对i-1,2,...n，矩阵A_i的维数为$p_{i-1}*p_i$，对乘积$A_1A_2...A_n$求出最小化标量乘法的加括号方案。

解法：

尽管可以通过递归计算取1<=k<n使得P(n)=ΣP(k)P(n-k)，遍历所有P(n)种方案，但这并不是一个高 28 1

经过以上几道题的锻炼，很容易看出，子结构是求$A_i...A_j$的加括号方法m[i][j]可递归地定义为

$$m[i][j]=\begin{cases}0 & if\ i=j\\ \min\limits_{i\le k<j}\{m[i][k]+\ m[k+1][j]+\ p_{i-1}p_kp_j\} & if\ i<j\end{cases}$$

这样，只需利用子结构求解m[1][n]即可，并在在构造m[1][n]的同时，记录状态转换。下面的代码展示了这个过程，不再仔细分析。

矩阵链乘法

扩展：

矩阵链乘法的备忘录解法（伪码），来自《算法导论》第15章。

```
MEMOIZED-MATRIX-CHAIN(p)
1  n ← length[p]−1
2  for i ← 1 to n
3      do for j ← i to n
4             do m[i, j] ← ∞
5  return LOOKUP-CHAIN(p, 1, n)

LOOKUP-CHAIN(p, i, j)
1  if m[i, j]<∞
2    then return m[i, j]
3  if i =j
4      then m[i, j] ← 0
5      else for k ← i to j−1
6             do q ← LOOKUP-CHAIN (p, i, k)+LOOKUP-CHAIN(p, k+1, j)+p_{i−1} p_k p_j
7               if q<m[i, j]
8                 then m[i, j] ← q
9  return m[i, j]
```

7.0-1背包

难度评级：★★

一个贼在偷窃一家商店时发现了n件物品，其中第i件值v_i元，重w_i磅。他希望偷走的东西总和越值钱越好，但是他的背包只能放下W磅。请求解如何放能偷走最大价值的物品，这里v_i、w_i、W都是整数。

解法：

如果每个物品都允许切割并只带走其一部分，则演变为部分背包问题，可以用贪心法求解。0-1背包问题经常作为贪心法不可解决的实例（可通过举反例来理解），但可以通过动态规划求解。

为了找出子结构的形式，粗略地分析发现，对前k件物品形成最优解时，需要决策第k+1件是否要装入背包。但是此时剩余容量未知，不能做出决策。因此把剩余容量也考虑进来，形成的**状态**由已决策的物品数目和剩余容量两者构成。这样，所有状态可以放入一个n*(W+1)的矩阵c中，其值为当前包中物品总价值，这时有

$$c[i][j]=\begin{cases}c[i-1][j] & if\ w_i>j\\ \max\{c[i-1][j-w_i]+v_i\ ,\ c[i-1][j]\} & if\ w_i\leqslant j\end{cases}$$

28 1

根据这个递推公式，很容易写出求解代码。

0-1背包问题示例代码

8.有代价的最短路径

难度评级：★★★

无向图G中有N个顶点，并通过一些边相连接，边的权值均为正数。初始时你身上有M元，当走过i点时，需要支付S(i)元，如果支付不起表示不能通过。请找出顶点1到顶点N的最短路径。如果不存在则返回一个特殊值，如果存在多条则返回最廉价的一条。限制条件：1<N<=100; 0<=M<=100；对任意i, 0<=S[i]<=100。

解法：

如果不考虑经过顶点时的花费，这就简化成了一个一般的两点间最短路径问题，可以用Dijkstra算法求解。加入了花费限制之后，就不能直接求解了。

考察从顶点0到达顶点i的不同状态，会发现它们之间的区别是：总花费相同但路径长度不同、总花费不同但路径长度不同。为了寻找最短路径，必然要保存到达i点的最短路径；同时为了找到最小开销，应该把到达i点的开销也进行保存。根据题目的数值限制，可以将总开销作为到达顶点i的一个状态区分。这样，就可以把Min[i][j]表示为到达顶点i（并经过付钱）时还剩余j元钱的最短路径的长度。在此基础上修改Dijkstra算法，使其能够保存到达同一点不同花费时的最短长度，最终的Min[N-1][0...M]中最小的即为所求。以下是求解过程的伪代码。

```
//初始化
对所有的(i,j),Min[i][j] = ∞,state[i][j] = unvisited;
Min[0][M] = 0;

while(1) {
    for 所有unvisited的(i,j)找出M[i][j]最小的，记为(k,l)
    if Min[k][l] = ∞
        break;

    state[k][l] = visited;
    for 所有顶点k的邻接点p
        if (l-S[p]>=0 && Min[p][l-S[p]]>Min[k][l]+Dist[k][p])
            Min[p][l-S[p]] = Min[k][l]+Dist[k][p];

    //通过Dijstra算法寻找不同花费下的最小路径
}

for 所有j，找出Min[N-1][j]最小的
    如果存在多个j，那么选出其中j最大的
```

9.瓷砖覆盖（状态压缩DP）

难度评级：★★★

用 1 * 2 的瓷砖覆盖 n * m 的地板，问共有多少种覆盖方式?

解法：

28 1

（启发来自于：poj 2411 & 编程之美 4.2 瓷砖覆盖地板，下文叙述做了点修改）

分析**子结构**，按行铺瓷砖。一块1*2瓷砖，横着放对下一行的状态没有影响；竖着放时，下一行的对应一格就会被占用。因此，考虑第i行的铺法时只需考虑由第i-1行造成的条件限制。枚举枚举第i-1行状态即可获得i行可能的状态，这里为了与链接一文一致，第i-1行的某格只有两个状态：空或者放置。空表示第i行对应位置需要放置一个竖着的瓷砖，这时在铺第i行时，除去限制以外，只需考虑放还是不放横着的瓷砖这2种情况即可（不必分为放还是不放、横到下一层还是竖着一共4种）。同时对于第i-1行的放法，用二进制中0和1表示有无瓷砖，那么按位取反恰好就是第i行的限制条件。

```
//原作者：limchiang
//出处：http://blog.csdn.net/limchiang/article/details/8619611
#include <stdio.h>
#include <string.h>

/** n * m 的地板 */
int n,m;

/** dp[i][j] = x 表示使第i 行状态为j 的方法总数为x */
__int64 dp[12][2049];

/* 该方法用于搜索某一行的横向放置瓷砖的状态数,并把这些状态累加上row-1 行的出发状态的方法数
 * @name row 行数
 * @name state 由上一行决定的这一行必须放置竖向瓷砖的地方，s的二进制表示中的1 就是这些地方
 * @name pos 列数
 * @name pre_num row-1 行的出发状态为~s 的方法数
 */
void dfs( int row, int state, int pos, __int64 pre_num )
{
    /** 到最后一列  */
    if( pos == m ){
        dp[row][state] += pre_num;
        return;
    }

    /** 该列不放 */
    dfs( row, state, pos + 1, pre_num );

    /** 该列和下一列放置一块横向的瓷砖 */
    if( ( pos <= m-2 ) && !( state & ( 1 << pos ) ) && !( state & ( 1 << ( pos + 1 ) ) ) )
        dfs( row, state | ( 1 << pos ) | ( 1 << ( pos + 1 ) ), pos + 2, pre_num );
}
int main()
{
    while( scanf("%d%d",&n,&m) && ( n || m ) ){
        /** 对较小的数进行状压，已提高效率 */
        if( n < m ){
            n=n^m;
            m=n^m;
            n=n^m;
        }

        memset( dp, 0, sizeof( dp ) );

        /** 初始化第一行 */
        dfs( 1, 0, 0, 1 );

        for( int i = 2; i <= n; i ++ )
            for( int j = 0; j < ( 1 << m ); j ++ ){
                if( dp[i-1][j] ){
                    __int64 tmp = dp[i-1][j];

                    /* 如果i-1行的出发状态某处未放，必然要在i行放一个竖的方块，
                     * 所以我对上一行状态按位取反之后的状态就是放置了竖方块的状态
                     */
                    dfs( i, ( ~j ) & ( ( 1 << m ) - 1 ), 0, tmp ) ;
                }
                else continue;
```

28 1

```
            }

        /** 注意并不是循环i 输出 dp[n][i]中的最大值 */
        printf( "%I64d\n",dp[n][(1<<m)-1] );

    }
    return 0;
}
```

10.工作量划分

难度评级：★★

假设书架上一共有9本书，每本书各有一定的页数，分配3个人来进行阅读。为了便于管理，分配时，各书要求保持连续，比如第1、2、3本书分配给第1人，4、5分配给第二人，6，7，8，9分配给第3人，但不能1，4，2分配给第1人，3，5，6分配给第2人。即用两个隔板插入8个空隙中将9本书分成3部分，书不能换位。同时，分配时必须整本分配，同一本书不能拆成两部分分给两个人。为了公平起见，需要将工作量最大的那一部分最小化，请设计分配方案。用$s_1,...,s_n$表示各本书的页数。

解法：

继续从**子结构**的角度出发，发现如果前面的k-1份已经分好了，那么第k份自然也就分好了。用M[n][k]表示将n本书分成k份时最小化的k份中的最大工作量，从第k份也就是最后一份的角度来看，总数-它的不同情况下数量 = 前k-1份的数量和。

$$M[n][k] = \min_{i=1}^{n} \max(M[i][k-1], \sum_{j=i+1}^{n} s_j)$$

除此以外，初始化为

$$M[1][k] = s_1, for\ all\ k > 0$$
$$M[n][1] = \sum_{i=1}^{n} s_1$$

自底向上地可以求得使M[n][k]最小化的解。

工作量划分

其他：

这个问题被称为线性分割(linear partition)问题，有不少的应用情形。如，一系列任务分配给几个并行进程，那么分配工作量最大任务的那个进程将成为影响最终完成时间的瓶颈。将最大的工作量尽量减少，能够使所有工作更快地完成。

11.三次捡苹果

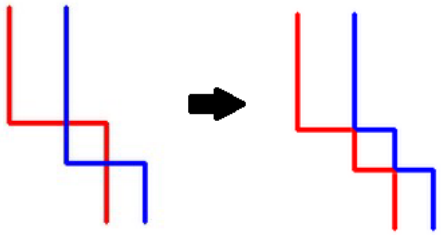
难度评级：★★★

（问题1的相关问题(1)的进一步扩展）一个矩形区域被划分为N*M个小矩形格子，在格子(i,j)中有A[i][j]个苹果。现在从左上角的格子(1,1)出发，要求每次只能向右走一步或向下走一步，每经过一个格子就把其中的苹果全部拿走，最后到达(N,M)。此时，只允许向上或向左走一步，反方向走回(1,1)。这一趟可以不走第一趟的路线，但当经过第- 趟所经过的格子时，里面已经没有苹果了。到达(1,1)后，再次反方向地只允许向右或向下走，走到(N,M)，同样可以不走前 28 1 求这三趟的走法，使得最终能拿取最多数量的苹果。

解法：

这个问题有两个难点，首先要理清三条路线的关系。可以发现，虽然第二趟方向相反，但其实和从(1,1)走到(N,M)是一样的，即三趟路线全部可以转化为从(1,1)向下或向右走到(N,M)的过程。

观察三条路线可以发现，实际中走的时候如果路线有交叉，可以把这种情况转化为相遇而不交错的情况如下图：



这样做的好处是，对于红线和蓝线上同一行j的点的坐标(i,j)(i',j)，总有i<=i',这样就能够把三条路线划分成左、中、右三条有序的路线。

经过两次转化，可以构造**子结构**了。用Max[y-1][i][j][k]表示在y-1行时，三条路线分别在i、j、k时所能取得的最大苹果数，用Max[y-1][i][j][k]可以求解任意的Max[y][i'][j'][k']，其中i' = i to j' , j' = j to k', k' = k to M. 如果线路重叠，苹果已经被取走，不用重复考虑。因此处理每一行时为了简单起见最好维护一个该位置苹果是否被取走的标志位，方便在路线重叠时计算。根据上面的范围关系，先求k'的所有情况，然后是j'，最后才是i'。这样Max[N][M][M][M]就是三趟后所能取得的最多苹果数。

参考资料

《算法导论》第15章动态规划、第16章贪心算法

《算法设计手册》第八章动态规划

《编程珠玑》相关问题

《编程之美》相关问题

poj 2411 & 编程之美 4.2 瓷砖覆盖地板

最大连续子序列乘积

Dynamic Programming: From novice to advanced

附录1：其他的一些动态规划问题与解答（链接）

双调欧几里德旅行商问题（算法导论思考题15-1）

评：网络上的很多中文版本，都不如直接看这篇文章里的英文原版解答理解的清楚。

整齐打印（算法导论思考题15-4）

评：难度不高，注意要求的是空格数的立方和最小。

动态规划应用：Viterbi算法

评：需要一些马尔科夫链的知识。理解起来不是很容易，理解以后是不是有种像是更多个生产线的装备线调度?

最高效益的调度（算法导论思考题15-7）

评：和0-1背包问题何其相似。

附录2：《算法设计手册》第八章 动态规划 面试题解答

8-24.

给定一个硬币种类的集合，找出凑出给定一个值所用的最小硬币数目。

解答：

正文问题1已做解答，略。

8-25.

长度为n的数组，其中元素可正可负可零。找出数组索引i,j使得从i到j的元素之和最大。

解答：

最大连续自序列和问题，请参考正文问题5的解答。

8-26.

假设你有一页纸，正面和背面都写满了字母，当剪掉正面上一个字母时，这一页的背面的字母也会被剪掉。设计一个算法来验证是否能通过这张纸生成一个给定的字符串？提供一个函数，当你输入一个字母的位置时能够返回它背面的字母。（叙述关键思路即可）

解答：

目前我所看到的最容易理解的解法是使用最大流来解的：
http://stackoverflow.com/questions/6135443/dynamic-programming-question

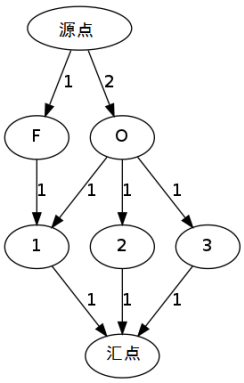
下面把思路大意翻译一下。

假设需要拼成的字符串是"FOO"，同时这张纸的正反面对应位置上的内容（可以通过提供的函数获得）分别是：

位置	1	2	3	4
正面	F	C	O	Z
反面	O	O	K	Z

由于位置4上的字母的正反面都用不到，忽略。

把这个表格转化成一个两层结点的流量网络



其中源点下面第一层表示拼成所需字符串的所有字母，源点到达该点的流量是需要的数目。第一层与第二层相连接表示某一位置上对应的是哪个所需字母，比如位置1正反面分别是F和O，表示它能提供1个F的入度和1个O的入度，但又由于一个片纸无论正反面只能用一次，那么它只能提供1的出度，直接连接汇点。

这个问题是否有解，等价于这个流量网络中是否存在一个流使得源点的流的出度等于汇点流的的入度，即一个最大流问题。

分类：笔试面试题,算法

标签：算法, 动态规划

« 上一篇：全面解析回溯法：算法框架与问题求解
» 下一篇：编程实践中C语言的一些常见细节

posted on 2013-09-11 10:13 五岳 阅读(68267) 评论(2) 编辑 收藏

评论

#1楼 2013-09-12 09:36 sudox

问题11，有点问题把。用Max[y-1][i][j][k]表示在y-1行时，三条路线分别在i、j、k时所能取得的最大苹果数。(y-1，i)可以走到（y-1,i+1），这种情况考虑了吗?

支持(0) 反对(0)

#2楼[楼主] 2013-09-12 12:02 五岳

@ sudox
原文有个笔误，把“ i' ”写成了“ x' ”，即先求最右边k'的所有情况，然后是j'，最后才是i'，重叠时也考虑到了，那个i' = i to j' 表示i'取值为[i,j']这个闭区间的整数。
两条线路仅重叠而不交错即为i' = j'的情况；
交错时按照图中转化为不交错不重叠或者仅重叠的情况

28 1

支持(0) 反对(0)

刷新评论 刷新页面 返回顶部

注册用户登录后才能发表评论，请 登录 或 注册，访问网站首页。

最新IT新闻:

· 在这件事上，刘强东被嘲讽吹牛功夫不如马云
· 载人龙飞船在NASA接受最后测试 或用于2024年登陆火星
· 传美团招股书披露：去年营收340亿，净亏损190亿
· 还敢吹"毫无PS痕迹"？小心被Adobe官方AI打脸
· 清华毕业华裔中年IT男跳楼身亡 或曾遭高通裁员两次
» 更多新闻...

最新知识库文章:

· 如何提升你的能力？给年轻程序员的几条建议
· 程序员的那些反模式
· 程序员的宇宙时间线
· 突破程序员思维
· 云、雾和霭计算如何一起工作
» 更多知识库文章...