華中科技大學

算法设计与分析实验报告

学 生 姓 名: 胡思勖

学 号: U201514898

专 业: 计算机科学与技术

班 级: 计卓 1501

指 导 教 师: 王多强

2017 年 12 月 25 日

目录

实验一　最近点对问题

1.1　题目描述

给出平面上的一些点的坐标，求出距离最近的两个点的坐标。如果有多对距离相同的点，依次输出这些点的坐标。

测试文件格式:

- 输入文件：in.dat
  - 格式: 第一行为一个整数, 表示测试用例的组数, 其后跟相应组数的测试用例
  - 每组测试用例包括:
    * 首行: 一个整数, 表示本组测试用例包含的点数, 其后跟相应点数的行
    * 其后: 每行两个整数, 表示该点的 x、y 坐标
- 输出文件：out.dat
  - 格式: 每行输出一个测试用例的答案, 即本组测试用例中相距最近的两个点, 用点的坐标表示: 四个整数。前两个整数表示第一个点, 后两个整数表示第二个点。若有多对相距最近的点, 依次罗列。

1.2　算法设计

最符合直觉的方法为朴素算法，即枚举空间中所有可能的点对，并分别计算这些点对之间的距离，最小的那个距离即为答案。即 $\min\{distance(p_i, p_j) \mid i, j \in [1, n]\}$，其中 p_i、p_j 是输入中的点，n 是输入的规模，需要对于 (i, j) 进行枚举，时间复杂度为 $O(N^2)$。

然而朴素算法的时间复杂度过高，使用分治法将问题分解成更小的问题，并当问题的规模足够小时直接求解可以降低整体运算的时间复杂度。首先将输入的点按照其横坐标排序，然后取这些横坐标的中位数讲平面划分为左右两部分，这样使近似一半的点进入左半部分，另一半的点进入右半部分，这样将原来的问题划分为两个子问题：以左半部分点集 P 为输入的子问题和以右半部分点集 Q 为输入的子问题。对于原问题而言，其解只有三种情况：

- 最小距离点对的两个点均在 P 中。
- 最小距离点对的两个点均在 Q 中。

- 最小距离点对的两个点一个在 P 中，另一个在 Q 中。

对于第一种和第二种情况需要常数时间进行合并，但是若出现第三情况，对于左半部分中的任意一点右半部分中最多有 n/2 个点需要进行距离的计算，此时合并步骤的耗时为 $O(N^2)$，整个算法的计算时间也为 $O(N^2)$，与朴素算法相同。可以看到，在整个算法中是由于合并阶段的第三种情况导致的计算时间复杂度过高，因此要对这一部分进行限制。

假设 P 中的最近点对和 Q 中的最近点对已经求得，分别为 p_1, p_2 和 q_1, q_2，并令 θ 为这两个点对距离中较小的那一个，那么若平面上最小距离点对属于第三种情况（一个点在 P 中，另一个在 Q 中），则在这一点对 p_3, q_3 中，p_3 与用于划分的中点 m 的横坐标距离不超过 θ，否则 p_3 与 q_3 之间的距离将会超过 θ，从而与这是最小距离点对矛盾。同理，可以将 q_3 的横坐标限制在 $[x_m, x_m + \theta]$ 范围内，这样就将第三种情况中需要计算的点对限制在了 $x \in [x_m - \theta, x_m + \theta]$ 这样一个带状区域内。又由于 p_3 与 q_3 之间的距离小于 θ，因此从纵坐标来看，对于带状区域中的每一个点 k 只需要计算 $y \in [y_k - \theta, y_k + \theta]$ 这一区域的点。从而对于带状区域中的每一个点，只需要检查 $\theta \times 2\theta$ 的一个矩形中的点即可，如图1.1所示，对于带状区域左半部分中的点 k，在右半部分找到对应的高为 2θ，宽为 θ 的矩形并依次计算 k 与这一矩形中的点的距离即可。此外，可以证明这一矩形中的点不超过 6 个：将点 k 对应的矩形按照

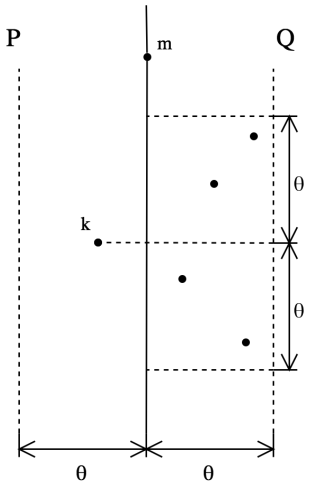

图 1.1: 包含点 q_3 的矩形

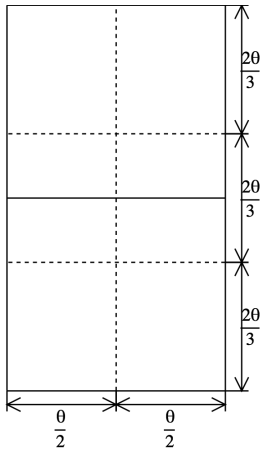

图 1.2: 矩形中点的个数

如1.2所示的方式划分为 6 个 $\frac{\theta}{2} \times \frac{2\theta}{3}$ 的矩形，由于在右半部分没有两个点之间的距离小于 θ 因此对于这 6 个矩形而言每个矩形中至多只有一个点，因为一个矩形中最大的两个点的距离为对角线长，即为 $\sqrt{(\frac{\theta}{2})^2 + (\frac{2\theta}{3})^2} \approx 0.83\theta < \theta$，因此在这种合并步骤中，对于第三种情况至多只需要检查 $6 \times n/2 = 3n$ 对点，而若这所有点是按照纵坐标进行排序过的，那么只需要 $O(n)$ 的时间就可以完成合并操作，此时最终的时间复杂度为 $O(n \log n)$。

从上述过程即可构建程序的伪代码：

Algorithm 1: 寻找最近点对算法

```
1: procedure FindNearestDotPairs(S)
2:     将 S 按照横坐标为中位数的点 m 近似划分为两等分 P, Q
3:     (A_1, d_1) ← FindNearestDotPairs(P)
4:     (A_2, d_2) ← FindNearestDotPairs(Q)
```

$\theta \leftarrow \min\{d_1, d_2\}, d \leftarrow \min\{d_1, d_2\}, A \leftarrow \min\{A_1, A_2\}$
for k in left half of $x \in [x_m - \theta, x_m + \theta]$ **do**
  for j in $x \in [x_m, x_m + \theta], y \in [y_k - \theta, y_k + \theta]$ **do**
    if $distance(j, k) < d$ **then**
      $d \leftarrow distance(j, k)$
      $A \leftarrow \{j, k\}$
    else if $distance(j, k) = d$ **then**
      $A \leftarrow A \bigcup \{j, k\}$
return (A, d)

将此程序按照递归部分划分，分为主函数和 fingNearestDotPairts 函数这两个过程，程序的流程图如1.3所示，其中主函数负责对于输入点集的稳定排序以及调用 findNearestDotPairs 过程，findNearest-DotPairs 则递归调用自身寻找最小点对。

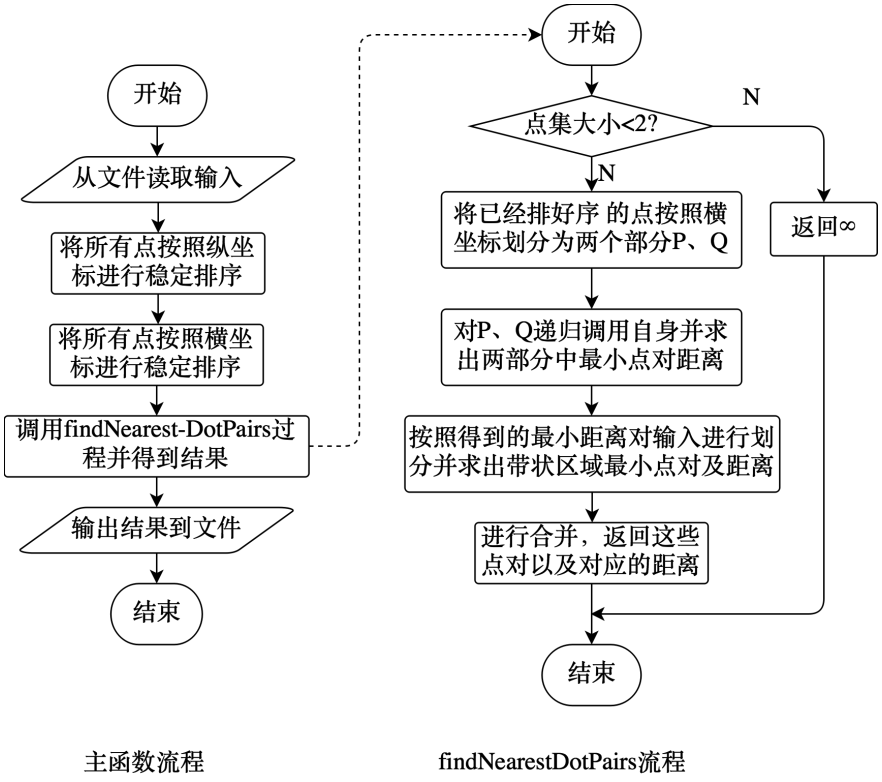

图 1.3: 程序流程图

1.3 测试分析

将源程序编译后进行测试，输入样例为给定的 1_self/in.dat，共有两组输入，第一组输入的点为 $(1, 9), (3, 8), (4, 6), (8, 1)$，第二组输入的点为 $(10, 8), (23, 98), (18, 32), (78, 43), (76, 25)$。对于第一组结果而言，期望得到的结果为 $(1, 9), (3, 8), (3, 8), (4, 6)$，其最小距离为 $\sqrt{5}$；对于第二组数据，期望得到的结

果为 $(76,25),(78,43)$，最小距离为 $2\sqrt{82}$，运行程序的过程以及得到的结果如图1.4所示，这与预期的结果一致，可以看出这一算法能够成功的计算出所有距离最小的点对。

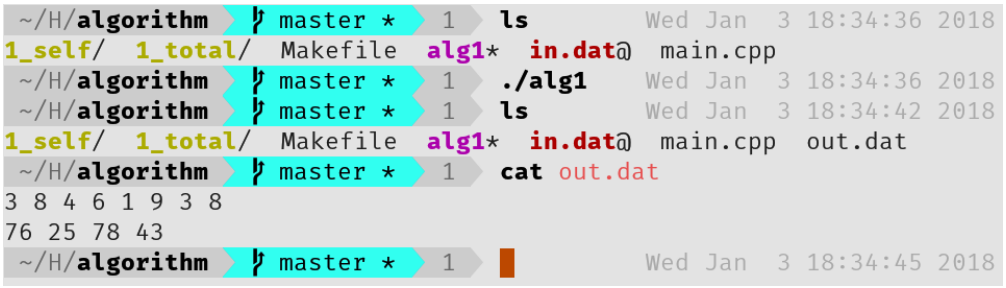

图 1.4: 测试结果

1.4 技术总结

这一问题是一个典型的使用分治法解决的问题。为了降低这一算法的时间复杂度，我们首先对于算法中最占用时间的部分进行分析，发现是合并操作导致了时间占用最长后，通过已知条件限制合并操作处理的数据量的大小来减少合并时间，其中，证明每一个 $\theta \times 2\theta$ 的矩形中至多只有 6 个点（即点的稀疏性）是关键的一步，这直接导致了在处理合并操作时时间复杂度从 $O(N^2)$ 降为 $O(N)$ 而使得整个程序的时间复杂度下降。

在程序实现时，使用两次稳定排序先后处理纵坐标和横坐标是使得后续处理所用时间较少的关键，这样可以在合并操作时不需要重新对于部分输入的点进行排序，对于两个点分别在左右两边的情况，能够将每次查找的点限制在一个较为连续的内存中，使得在首次遇到超出矩形范围的点时即可跳出循环，进行下一轮比较，从而降低程序的运行时间。

实验二　大数乘法

2.1　题目描述

利用分治法设计一个计算两个 n 位的大整数相乘的算法, 要求计算时间低于 $O(n^2)$。

大整数 (biginteger): 位数很多的整数, 普通的计算机不能直接处理, 如:

9,834,975,972,130,802,345,791,023,498,570,345

对大整数的算术运算, 显然常规程序语言是无法直接表示的。编程实现大整数的加、减、乘运算, 需考虑操作数为 0、负数、任意位等各种情况。

测试文件格式:

- 输入文件：in.dat
  - 格式: 第一行为一个整数, 表示测试用例的组数, 其后跟相应组数的测试用例。每个测试用例一行, 包含 3 个整数 (长整数数字串)，前两个是待测试的操作数, 第 3 个整数表示操作类型 (1: 加法,2: 减法,3: 乘法)。
- 输出文件：out.dat
  - 格式: 每组测试用例输出一行, 最后不要加空行

2.2　算法设计

对于大数乘法而言，目前已知的方法有模拟法（逐位相乘并进位）、Karatsuba 算法、Toom-Cook 算法、快速傅里叶变换等。使用模拟方法需要对于其中大数中的每一位都要对另一个大数的所有位进一次乘法和一次加法进行处理（不包括进位），这一算法的时间复杂度为 $O(N^2)$，对于这一问题而言显得过于缓慢。Karatsuba 算法为 Toom-Cook 算法的一个特例，其时间复杂度为 $O^{\log 3/\log 2}$（将 4 次乘法减小为 3 次），而 Toom-Cook 将 9 次乘法减小为 5 次，其时间复杂度为 $O^{\log 5/\log 3}$，FFT 可以以更低的时间复杂度 $O(N\cdot\log(N)\cdot\log(\log(N)))$ 进行计算。基于实现容易程度以及乘法精度的考虑，最终采用 Karatsuba 算法实现两个大数的乘法。

Karatsuba 基于以下原理：令 x, y 为两个 B 进制的 n 位大整数，对于任意小于 n 的正整数 m，x, y 可以写为：

$$\begin{cases} x = x_1 B^m + x_0 \\ y = y_1 B^m + y_0 \end{cases}$$

此时，$x \cdot y$ 可以写为：

$$\begin{aligned} x \cdot y &= (x_1 B^m + x_0)(y_1 B^m + y_0) \\ &= z_2 B^{2m} + z_1 B^m + z_0 \end{aligned}$$

其中，

$$\begin{cases} z_0 = x_0 y_0 \\ z_1 = x_1 y_0 + x_0 y_1 \\ z_2 = x_1 y_1 \end{cases}$$

但是这依然需要计算 $4 \times N^2/4 = N^2$ 次乘法。但如果 z_2 和 z_0 已经实现，那么 z_1 可以表示为：

$$\begin{aligned} z_1 &= (x_1 + x_0)(y_1 + y_0) - x_1 y_1 - x_0 y_0 \\ &= (x_1 + x_0)(y_1 + y_0) - z_2 - z_0 \end{aligned}$$

于是原本需要 4 次的乘法在加上依赖关系后变成了 3 次，通过这一过程对于三次乘法的操作数进行递归调用，再加上其余加减法需要的时间为 $O(N)$，可以得到 $T(N) = 3T(\lceil n/2 \rceil) + cn + d$，根据主定理可得时间复杂度为 $O(n^{\log_2 3})$。需要注意的是，在计算 $(x_1 + x_0)(y_1 + y_0)$ 时可能会发生溢出（$0 \le \text{结果} \le 2B^m$），对于这种情况需要在计算时加以考虑。

对于大数相乘原数中出现负数的情况，首先去掉负号，作为两个无符号整数相乘，最后在结果中根据输入负号的个数决定是否加上负号。此外，在加法和减法的实现中要考虑结果中出现前缀 0 的情况，需要将这些 0 去除。其中，核心的无符号大整数算法如下：

Algorithm 2: 无符号大整数相乘算法

procedure BigIntMultiply(x, y)
  if $x.len < 2$ or $y.len < 2$ **then** ▷ $a.len$ 指 a 的长度
    使用模拟法计算 $product \leftarrow x \times y$
    return *product*
  $m \leftarrow \max\{x.size, y.size\}/2$
  按照 m 将 x, y 分为 x_1, x_0, y_1, y_0 ▷ 其中 $x_0.len = \min\{x.len, m\}, y_0.len = \min\{y.len, m\}$
  $z_0 \leftarrow$ BigIntMultiply(x_0, y_0)
  $z_1 \leftarrow$ BigIntMultiply($(x_1 + x_0), (y_1 + y_0)$)
  $z_2 \leftarrow$ BigIntMultiply(x_1, y_1)
  $product \leftarrow z_2 \times 10^{2m} + (z_1 - z_2 - z_0) \times 10^m + z_0$ ▷ $z_i \times 10^j$ 使用向字符串尾部添加 0 实现
  去除 *product* 前缀 0
  return *product*

在实际实现过程中，主函数获得输入后首先调用 BigNumMult 函数，BigNumMult 函数处理两个大数的符号，然后调用 BigNumMultHelper 函数对于无符号大整数进行计算。BigNumMultHelper 使用的即为上述 BitIntMultiply 算法。整个算法的流程图如2.1所示。此外，程序还实现了大整数加法和大整数减法，均使用模拟方法实现，此处不再给出流程图。

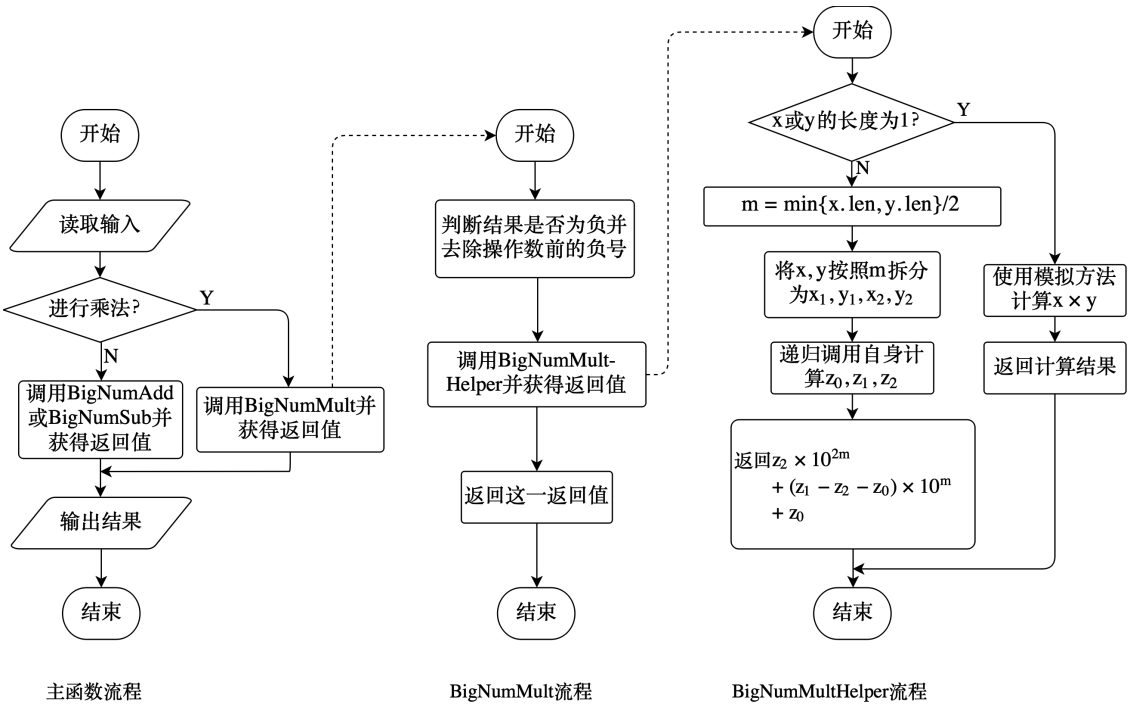

图 2.1: 程序大整数乘法部分流程图

2.3 测试分析

对于程序功能的完整性进行测试，包括对于前缀 0 的处理，对于负数的处理以及对于大数的处理。输入以及输出如2.2和2.3所示。可以看出，对于每一个输入，无论是否有前缀 0，输入是否存在负数，程

```
~/H/algorithm  master *+  2  cat in.dat
24
-12132134374136478324623 12132134374136478324623 1
0 312321312314324223 1
13467153478115678 14329816594165475175615628 4 2
-3241412347812341 -125748543715431952 3
12131231293129321 83 0 3
0 0 1
0 1 1
000 000 1
01 0 1
0 0 2
00 000 2
0000 1 2
1 0 2
001 001 2
1 2 3
2 2 3
02 02 3
04 04 3
-1 -1 1
-2 -1 2
-1 -3 3
-001 001 1
-003 004 2
-004 017 3
~/H/algorithm  master *+  2
```

图 2.2: 程序输入

```
~/H/algorithm  master *+  2
cat out.dat
0
312321312314324223
-14329816592818759827804060 6
4076028823186210814014490513196 32
0
0
1
0
1
0
0
-1
1
0
2
4
4
16
-2
-1
3
0
-7
-68
~/H/algorithm  master *+  2
```

图 2.3: 程序输出

序都能够正确的进行处理，说明程序功能正常。

2.4 技术总结

在大整数乘法算法中，对于通过数学变换将乘法的次数减少是降低时间复杂度的关键。这一思路在其他算法中也有体现，如 Toom-Cook 算法以及用于快速矩阵乘法的 Strassen 算法均是使用这一思想。这种变换值得我们学习并应用在今后的研究过程中。此外，对于负数的处理是值得关注的，在大整数加法和减法的实现中，对于负数的处理应该分情况讨论并调用处理无符号大数加减法的子程序。

实验三　最优二分查找树

3.1　题目描述

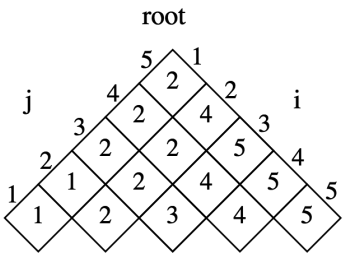

图 3.1: 输入表 *root*

设计伪代码 CONSTRUCT-OPTIMAL-BST(root)，输入为表 *root*，输出是最优二叉搜索树的结构。例如，对图3.1中的 *root* 表，应输出

k_2 为根
k_1 为 k_2 的左孩子
d_0 为 k_1 的左孩子
d_1 为 k_1 的右孩子
k_5 为 k_2 的右孩子
k_4 为 k_5 的左孩子
k_3 为 k_4 的左孩子
d_2 为 k_3 的左孩子
d_3 为 k_3 的右孩子
d_4 为 k_4 的右孩子
d_5 为 k_5 的右孩子

3.2　算法设计

由于 $root[i,j]$ 表示的是包含关键字 $k_i,...,k_j$ 子树的根，并且二叉搜索树的子树中的节点都是连续的，因此 $root[i,j]$ 的左子树和又子树的根分别为 $root[i,root[i,j]-1]$ 和 $root[root[i,j]+1,j]$，从而可以进行递归找到所有子树的根以及他们与父节点的关系，进而构建整个子树。算法 CONSTRUCT-OPTIMAL-BST 如下：

Algorithm 3: 构建最优子树算法

```
1: Global k ← 0                                                    ▷ k 存储伪结点的序号
2: Global result ← {}                                              ▷ result 采用栈结构
3: procedure CONSTRUCT-OPTIMAL-BST(root, i, j)
4:     if i > j then
5:         k ← k + 1
6:         return nil
7:     child ← CONSTRUCT-OPTIMAL-BST(root, root[i, j] + 1, j)      ▷ 找出右孩子
8:     if child ≠ nil then
9:         result.push ("child 是 root[i, j] 的右孩子")
10:    else
11:        result.push ("d_{root.size−k} 是 root[i, j] 的右孩子")
12:    child ← CONSTRUCT-OPTIMAL-BST(root, i, root[i, j] − 1)      ▷ 找出左孩子
13:    if child ≠ nil then
14:        result.push ("child 是 root[i, j] 的左孩子")
15:    else
16:        result.push ("d_{root.size−k} 是 root[i, j] 的左孩子")
17:    return k_{root[i,j]}
```

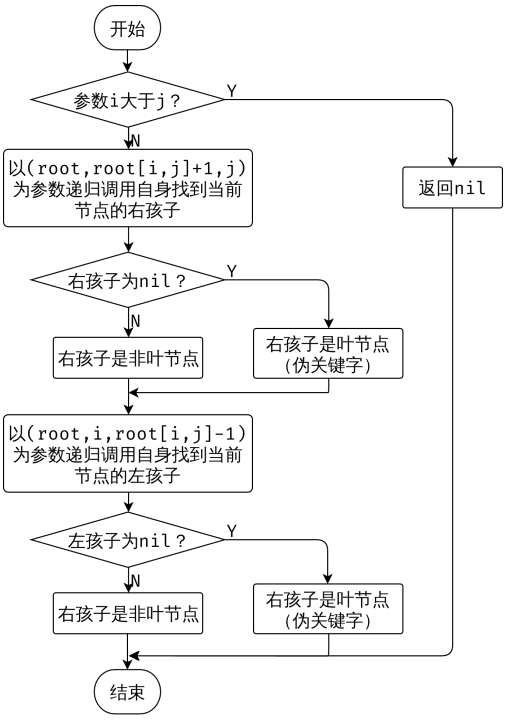

图 3.2: 程序流程图

运行 CONSTRUCT-OPTIMAL-BST 算法后，从栈 *result* 中逐个弹出元素并输出即为题目要求的结果。由于在每一层递归中需要在最后返回本节点，因此要按先序遍历的顺序输出，必须按照右孩

子-左孩子的顺序遍历并压栈。程序的流程图如图3.2所示。流程图中给出了构建源二叉树的程序执行框架，没有对于输入输出的细节问题进行详细描绘。

在这一递归算法中，每一层递归所需要的参数为 $root, i, j$，分别表示整个 root 表，以及当前节点在 root 表中的位置 (i, j)。在每一次递归调用中，若出现 $i > j$ 的情况说明已经到达了叶节点，因为所有的非叶节点在 $root$ 中的的坐标全部满足 $i \leq j$，此时返回 nil 以告知调用者已经到达了叶节点。否则递归调用自身，首先确定这一节点的右子树，然后确定这一节点的左子树，最后返回节点自身作为上一层调用的答案。

3.3 测试分析

对于事先给定的 in.dat 进行测试，in.dat 的内容，测试过程以及测试结果如图3.3所示。从图中可以看出，对于输入的 $root$ 矩阵程序能够正确的生成 out.dat，并且其中的内容为按照先序遍历对于原搜索树进行先序遍历的输出。由此可以说明程序的功能正常。

```
~/H/algorithm  master *  3  ls                    Tue Jan 16 18:11:32 2018
3_self/  3_total/  Makefile  alg3*  in.dat@  main.cpp
~/H/algorithm  master *  3  cat in.dat
7
1 2 2 2 3 3 5
0 2 3 3 3 5 5
0 0 3 3 4 5 5
0 0 0 4 5 5 6
0 0 0 0 5 6 6
0 0 0 0 0 6 7
0 0 0 0 0 0 7
~/H/algorithm  master *  3  ./alg3              Tue Jan 16 18:11:35 2018
~/H/algorithm  master *  3  ls                    Tue Jan 16 18:11:37 2018
3_self/  3_total/  Makefile  alg3*  in.dat@  main.cpp  out.dat
~/H/algorithm  master *  3  cat out.dat
k5是根
k2是k5的左孩子
k1是k2的左孩子
d0是k1的左孩子
d1是k1的右孩子
k3是k2的右孩子
d2是k3的左孩子
k4是k3的右孩子
d3是k4的左孩子
d4是k4的右孩子
k7是k5的右孩子
k6是k7的左孩子
d5是k6的左孩子
d6是k6的右孩子
d7是k7的右孩子
~/H/algorithm  master *  3                       Tue Jan 16 18:11:40 2018
```

图 3.3: 测试过程以及结果

3.4 技术总结

在通过 $root$ 矩阵对于原二叉搜索树的构建过程中，只有了解了 $root$ 矩阵是如何生成的才能够逆推原有的搜索树结构。此外，在此算法中对于左孩子和右孩子递归调用的顺序以及递归的停止调教是十分重要的。若不能够正确的处理这些问题则会输出错误的结果。

实验四　Floyd Warshell 最短路径算法

4.1　题目描述

补充 ALL-PATHS 算法, 增加前驱矩阵 (Chp.25.2), 使得在求出结点间的最短路径长度矩阵 A 后, 能够推导出每对结点间的最短路径。

- 输入文件：in.dat
  - 格式: 第一行为一个整数, 表示测试用例的组数, 其后跟相应组数的测试用例
  - 每组测试用例包括:
    * 首行: 一个整数, 表示本组测试用例包含的结点数 n, 其后跟 n 行
    * 其后: 每行 n 个整数, 表示结点间邻接关系及边的长度 (邻接成本矩阵)。边的长度 <100, 100 即表示结点间没有边。
- 输出文件：out.dat
  - 格式: 第一行为一个整数, 表示测试用例的组数, 其后跟相应组数的测试用例
  - 每组测试用例输出包括:
    * 首行: 一个整数, 表示本组测试用例包含的结点数 n, 其后跟 $n + n^2$ 行
    * 其后: 开始的 n 数, 表示结点间最短路径的长度 (A 矩阵)。路径的程度 <32767, 32767 即表示结点间没有可达的路径。其后 n2 行, 顺次输出结点对 (1,1)、(1,2)、...、(1,n), (2,1)、(2,2)、...、(2,n)、...(n,1)、...、(n,n) 之间的最短路径结点序列, 结点间用空格隔开。(i,i) 输出 i, 若 (i,j) 之间没有路径, 输出 NULL。

4.2　算法设计

ALL-PATHS 算法实现的是 Floyd-Warshall 算法，其原理是动态规划。令 $D_{i,j,k}$ 为从 i 到 j 的只以 $(1, 2, \ldots, k)$ 中的节点为中间节点的最短路径的长度，那么

$$D_{i,j,k} = \begin{cases} D_{i,j,k} = D_{i,j,k-1} + D_{k,j,k-1} & ,\text{最短路径经过点}k \\ D_{i,j,k} = D_{i,j,k-1} & ,\text{若最短路径不经过点}k \end{cases} \tag{4.1}$$

因此 $D_{i,j,k} = \min\{D_{i,j,k-1}, D_{i,k,k-1} + D_{k,j,k-1}\}$。

但是 Floyd-Warshall 算法只能给出任意两点间最短路径的长度而不能给出具体的路径。要使其能够得到具体的路径，需要前驱矩阵 Π 来存储最短路径树。令 $\pi_{ij}^{(k)}$ 为从节点 i 到节点 j 的一条在所有中间节点都取自集合 $1, 2, \ldots, k$ 的最短路径上 j 的前驱节点，那么对于任意两个节点 m, n 就可以通过从 π_{mn} 开始查询 Π 直到查询完整个路径来构建从 m 到 n 的最短路径。而对于 $k \geq 1$，

$$\pi_{i,j}^{(k)} = \begin{cases} \pi_{ij}^{(k-1)} & , d_{ij}^{(k-1)} \leq d_{ik}^{(k-1)} + d_{kj}^{(k-1)} \\ \pi_{kj}^{(k-1)} & , d_{ij}^{(k-1)} > d_{ik}^{(k-1)} + d_{kj}^{(k-1)} \end{cases} \tag{4.2}$$

于是通过公式4.1和公式4.2可以得到扩展过后的 ALL-PATHS 算法如下：

Algorithm 4: 最短路径构建算法

```
1:  procedure AllPaths(W)
2:      dist ← |V| × |V| array of ∞                    ▷ dist 为最短路径表 D
3:      next ← |V| × |V| array of nil                  ▷ next 为前驱子矩阵 Π
4:      for each edge (u, v) do                        ▷ 初始化 D 和 Π
5:          dist[u, v] ← W(u, v)                       ▷ 边 (u, v) 的权重
6:          next[u, v] ← v
7:      for k = 1 to |V| do                            ▷ 扩展的 Floyd-Warshall 算法
8:          for i = 1 to |V| do
9:              for j = 1 to |V| do
10:                 if dist[i, j] > dist[i, k] + dist[k, j] then
11:                     dist[i, j] ← dist[i, k] + dist[k, j]
12:                     next[i, j] ← next[i, k]
13:     for i = 1 to |V| do                            ▷ 构造最短路径
14:         for j = 1 to |V| do
15:             if next[u, v] = nil then
16:                 continue
17:             path[i, j] = [i]
18:             while i ≠ j do
19:                 i ← next[i][j]
20:                 path[i, j].append(i)
21:     return path, dist
```

在算法中，首先运行 Floyd-Warshall 算法，在运行时同时构造 D 和 Π 两个矩阵，在此算法运行完成后，通过 Π 矩阵求出每一个点对之间的路径，然后将结果返回给调用者进行输出。

程序的流程图如4.1所示，首先，通过使用 Floyd-Warshall 算法，找出每两个点对之间的最短路径的同时构建 Π 矩阵，在这一步骤完成后对于所有点对 (u, v) 在上一步得到的 Π 矩阵中找到最短路径，最后把得到的结果返回给上层进行输出（流程图中未给出）。需要注意的是，在算法开始前需要正确的初始化 D 和 Π，否则不能够得到正确答案。

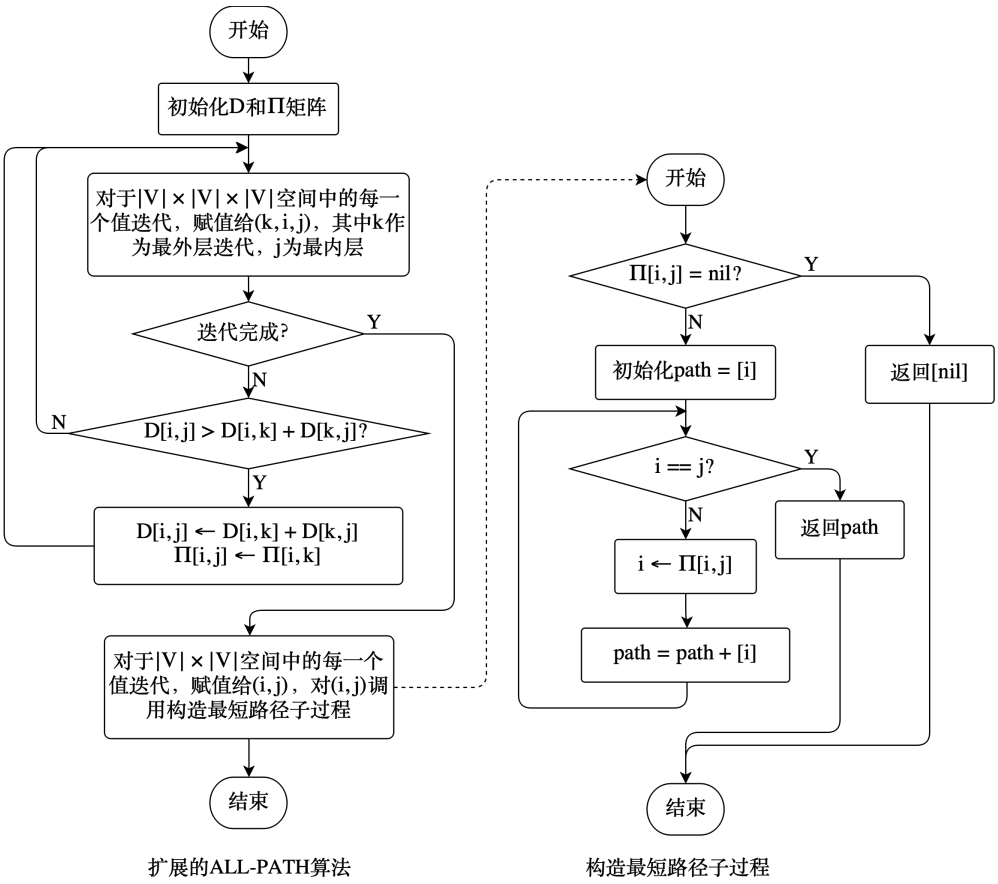

图 4.1: 程序流程图

4.3 测试分析

对于给定的 in.dat 运行编译好的程序，in.dat 对应的拓扑图如4.2所示，in.dat 的内容、运行的过程以及输出的 out.dat 中的结果如4.3所示。从图中可以看出，程序能够正确的生成 out.dat，并且输出结果与预期结果一致。

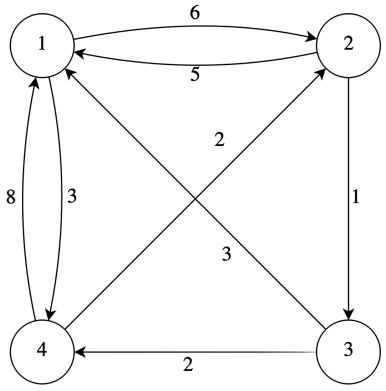

图 4.2: 数据对应的拓扑图

```
 ~/H/algorithm  master *  4  ls      Tue Jan 16 21:13:45 2018
4_self/  4_total/  Makefile  alg4*  in.dat@  main.cpp
 ~/H/algorithm  master *  4  cat in.dat
1
4
0 6 100 3
5 0 1 100
3 100 0 2
8 2 100 0
 ~/H/algorithm  master *  4  ./alg4
 ~/H/algorithm  master *  4  cat out.dat
1
4
0 5 6 3
4 0 1 3
3 4 0 2
6 2 3 0
1
2 4 1
3 2 4 1
4 1
1 3 2
2
3 2
4 3 2
1 3
2 4 3
3
4 3
1 3 2 4
2 4
3 2 4
4
 ~/H/algorithm  master *  4                Tue Jan 16 21:14:07 2018
```

图 4.3: 测试过程以及输出结果

4.4 技术总结

构建最短路径的关键技术是前驱矩阵 Π 的生成，只要能够生成前去矩阵 Π，就可以通过依次迭代的方法找到任意两点之间的最短路径，此外，本程序还采用了直接在原来的矩阵上进行迭代的技术，从而使空间复杂度从 $O(N^3)$ 下降到了 $O(N^2)$。在直接在原有空间上进行迭代时需要注意不能够把将来还需要使用的值覆盖掉，即需要特别注意迭代的顺序，以保证结果的正确性。

实验五　Wooden Sticks (POJ 1065)

5.1　题目描述

5.1.1　Description

There is a pile of n wooden sticks. The length and weight of each stick are known in advance. The sticks are to be processed by a woodworking machine in one by one fashion. It needs some time, called setup time, for the machine to prepare processing a stick. The setup times are associated with cleaning operations and changing tools and shapes in the machine. The setup times of the woodworking machine are given as follows:

- The setup time for the first wooden stick is 1 minute.
- Right after processing a stick of length l and weight w , the machine will need no setup time for a stick of length l' and weight w' if $l \leq l'$ and $w \leq w'$. Otherwise, it will need 1 minute for setup.

You are to find the minimum setup time to process a given pile of n wooden sticks. For example, if you have five sticks whose pairs of length and weight are (9,4), (2,5), (1,2), (5,3), and (4,1), then the minimum setup time should be 2 minutes since there is a sequence of pairs (4,1), (5,3), (9,4), (1,2), (2,5) .

5.1.2　Input

The input consists of T test cases. The number of test cases (T) is given in the first line of the input file. Each test case consists of two lines: The first line has an integer n , $1 \leq n \leq 5000$, that represents the number of wooden sticks in the test case, and the second line contains $2n$ positive integers $l_1, w_1, l_2, w_2, ..., l_n, w_n$, each of magnitude at most 10000 , where l_i and w_i are the length and weight of the i th wooden stick, respectively. The $2n$ integers are delimited by one or more spaces.

5.1.3　Output

The output should contain the minimum setup time in minutes, one per line.

5.1.4 Sample Input

```
3
5
4 9 5 2 2 1 3 5 1 4
3
2 2 1 1 2 2
3
1 3 2 2 3 1
```

5.1.5 Sample Output

```
2
1
3
```

5.2 算法设计

首先将所有的木棍按照重量进行排序，然后按照长度进行排序（均为稳定排序），排序完成后所有的木棍就是按照长度排序好的非降序序列，且当长度相等时是按照重量排好的非降序序列。然后求出这一序列对于重量的最长下降子序列的长度，就是所得到的结果。

证明：假设上述最长下降子序列的长度为 m，根据木棍的排列规则可知这下降子序列中的木棍的长度不可能相等（由于长度相等时重量是按照非降序进行排序的），因此这一最长下降子序列中的任意两个木棍 i, j 都满足 $l_i < l_j$ 且 $w_i > w_j$，此时它们两两“不兼容”（不可能位于同一个对于长度和重量而言都是非降序的子序列中）。因此所花费的费用至少为 m。

使用贪心算法，维护一个当前所有非降子序列中最后一个值的列表 M，然后对于排好序的木棍进行依次进行迭代，每一次迭代首先观察其长度和重量是否都大于 M 中的某一个木棍的对应值，若是，更新该木棍的对应值，将这一木棍的属性插入列表末尾，最后得到的列表的长度就是最长下降子序列的长度 m（可用剪切-粘贴法证明如果不按照这一规则构建 M 则 M 的长度将会大于或等于 m），因此能够在花费为 m 的情况下处理所有的木棍。又已知话费至少为 m，因此 m 即为问题的解。求解的算法如下：

Algorithm 5: Wooden Sticks 问题求解算法

```
1: procedure WoodenSticks(S)                          ▷ S 为所有木棍 {长度，重量} 的列表
2:     StableSort(S, weight)                          ▷ 按照重量稳定排序
3:     StableSort(S, length)                          ▷ 按照长度稳定排序
4:     M ← {}
5:     for stick in S do
6:         if stick.weight ≥ min{s.weight|s ∈ M} then
7:             s ← max{s|s ∈ M ⋀ s.weight < stick.weight}
```

```
8:            s ← stick
9:        else
10:           S.append(stick)
11:   return S.length
```

这一算法的流程图如5.1所示，首先，也是最重要的一步：对于重量和长度进行稳定排序，以构建一个首先按照长度然后按照重量的费降序序列。然后对于这一序列中的每一个木棍进行迭代，如果木棍的重量大于 M 中的重量最小的木棍的重量，那么更新列表中比这一木棍重量小的木棍中重量最大的那个为当前的木棍。

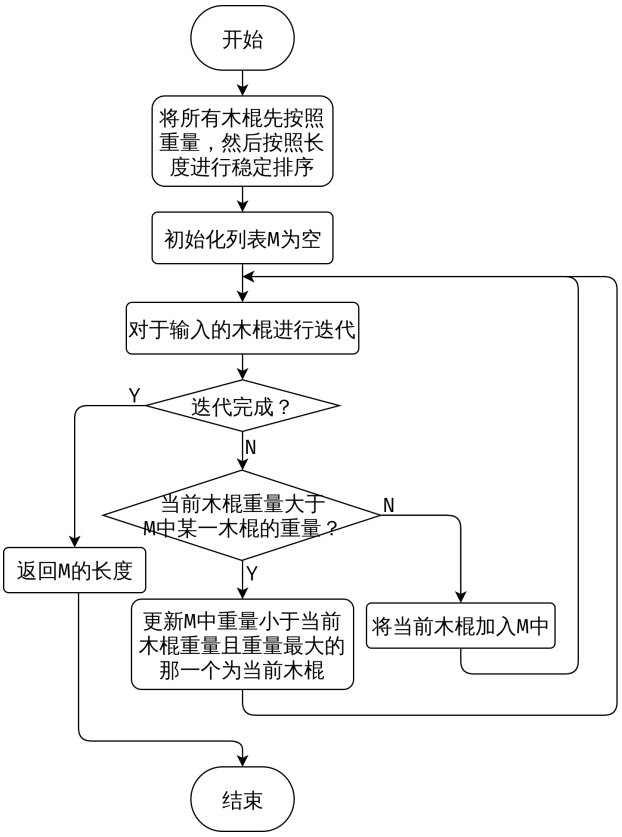

图 5.1: 程序流程图

5.3 测试分析

如5.2所示，对于 POJ 提供的测试用例，程序在 47MS 内全部通过，由此可以说明程序运行正确。

Run ID	User	Problem	Result	Memory	Time	Language	Code Length	Submit Time
18011077	husixu1	1065	Accepted	388K	47MS	C++	1305B	2018-01-02 10:20:34

图 5.2: POJ 测试结果

5.4 技术总结

对于这一问题而言，使用稳定排序以及贪心算法是至关重要的。如果不适用稳定排序可能则不能保证生成先按照长度非降序再按照重量非降序的序列，对于降序最长子序列的统计而言，使用贪心算法则可以在保证答案正确的情况下减少所需要的时间。

实验六　Gone Fishing (POJ 1042)

6.1　题目描述

6.1.1 Description

John is going on a fishing trip. He has h hours available ($1 \leq h \leq 16$), and there are n lakes in the area ($2 \leq n \leq 25$) all reachable along a single, one-way road. John starts at lake 1, but he can finish at any lake he wants. He can only travel from one lake to the next one, but he does not have to stop at any lake unless he wishes to. For each $i = 1, \ldots, n-1$, the number of 5-minute intervals it takes to travel from lake i to lake $i+1$ is denoted $t_i (0 < t_i \leq 192)$. For example, $t_3 = 4$ means that it takes 20 minutes to travel from lake 3 to lake 4. To help plan his fishing trip, John has gathered some information about the lakes. For each lake i, the number of fish expected to be caught in the initial 5 minutes, denoted $f_i (f_i \geq 0)$, is known. Each 5 minutes of fishing decreases the number of fish expected to be caught in the next 5-minute interval by a constant rate of $d_i (di \geq 0)$. If the number of fish expected to be caught in an interval is less than or equal to d_i , there will be no more fish left in the lake in the next interval. To simplify the planning, John assumes that no one else will be fishing at the lakes to affect the number of fish he expects to catch.

Write a program to help John plan his fishing trip to maximize the number of fish expected to be caught. The number of minutes spent at each lake must be a multiple of 5.

6.1.2 Input

You will be given a number of cases in the input. Each case starts with a line containing n. This is followed by a line containing h. Next, there is a line of n integers specifying $f_i (1 \leq i \leq n)$, then a line of n integers $d_i (1 \leq i \leq n)$, and finally, a line of n - 1 integers $t_i (1 \leq i \leq n-1)$. Input is terminated by a case in which $n = 0$.

6.1.3 Output

For each test case, print the number of minutes spent at each lake, separated by commas, for the plan achieving the maximum number of fish expected to be caught (you should print the entire plan on one line even if it exceeds 80 characters). This is followed by a line containing the number of fish expected.

If multiple plans exist, choose the one that spends as long as possible at lake 1, even if no fish are expected to be caught in some intervals. If there is still a tie, choose the one that spends as long as possible at lake 2, and so on. Insert a blank line between cases.

6.1.4 Sample Input

```
2

1
10 1
2 5
2
4
4
10 15 20 17
0 3 4 3
1 2 3
4
4
10 15 50 30
0 3 4 3
1 2 3
0
```

6.1.5 Sample Output

```
45, 5
Number of fish expected: 31

240, 0, 0, 0
Number of fish expected: 480

115, 10, 50, 35
Number of fish expected: 724
```

6.2 算法设计

对于题目进行分析，发现若能够知道 John 最终停止湖 k，那么他最终旅行的时间是固定的，为 $\sum_{i=1}^{k-1} t_i$，从总时间中减去这段时间后，可以看做 John 可以在湖 $[1,k]$ 之间随意的移动而不需要花费时间，最终只需统计在每个湖上花费的总时间 $total_i$ 然后在湖 i 上花费 $total_i$ 这么长的时间即可。

因此将旅行所用的时间减去后，剩下的时间可以使用贪心算法，在每一个 5 分钟选择湖 $[1,k]$ 中收益最大的那个，并花费 5 分钟在其上钓鱼即可。如果在某个 5 分钟时间段内两个或多个湖收益相等且最大，那么选择其中编号最小的进行处理。

对于每一个给定的湖的树龄 n，对于每一个可能的 k 运行上述算法，然后取能够取得的最好的结果返回即可。具体算法如下：

Algorithm 6: GoneFishing 问题求解算法

procedure GoneFishing(F, D, T, h) ▷ $F = \{f_i\}, D = \{d_i\}, T = \{t_i\}$
  $bestPath \leftarrow []$
  $bestResult \leftarrow -1$
  for k=1 to n **do**
    $m = h * 60 - \sum_1^{k-1} t_i$ ▷ 能够用于钓鱼的分钟数
    $rResult \leftarrow 0$ ▷ 当前 k 值下的最佳结果
    $rPath \leftarrow [0] \times k$ ▷ 当前 k 值下的最佳路径
    while $m \neq 0$ **do**
      $i \leftarrow \min\{i | i \in [1,k] \bigwedge f_i \geq f_j, \forall j \in [1,k]\}$
      $rResult \leftarrow rResult + f_i$
      $rPath[i] \leftarrow rPath[i] + 5$
      $f_i \leftarrow \max\{f_i - d_i, 0\}$
      $m \leftarrow m - 5$
    if $rResult > bestResult$ **then**
      $bestPath \leftarrow rPath$
      $bestResult \leftarrow rResult$
  return $bestPath, bestResult$

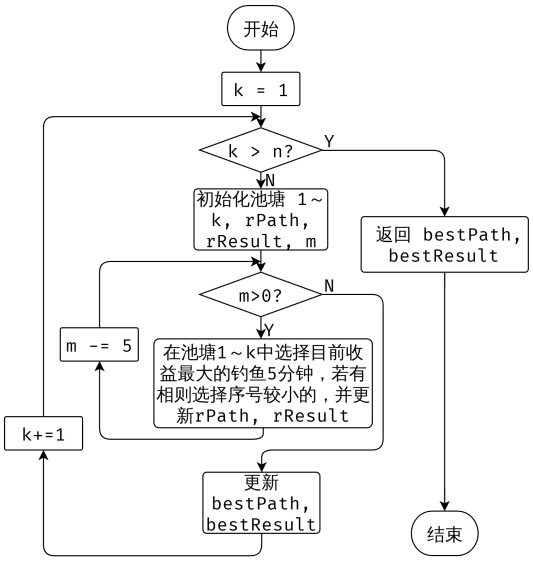

图 6.1: 程序流程图

与设计中预想的一样，在算法中，首先对于每一个可能的 k 值进行枚举，从总的时间中减去旅行所用的时间后每次选取收益最大的池塘在上面钓鱼即可，流程图如图6.1所示。

6.3 测试分析

如6.2所示，对于 POJ 提供的测试用例，程序在 188MS 内全部通过，由此可以说明程序运行正确。

Run ID	User	Problem	Result	Memory	Time	Language	Code Length	Submit Time
18011425	husixu1	1042	Accepted	704K	188MS	G++	1755B	2018-01-02 14:29:18

图 6.2: POJ 测试结果

6.4 技术总结

对于 “Each 5 minutes of fishing decreases the number of fish expected to be caught in the next 5-minute interval” 这句话的理解应该是：如果不在该 lake 上面钓鱼，那么该 lake 里面的鱼是不会减少的，只有当选择了该 lake，那么鱼会稳定减少直至为 0。此外，使用贪心算法也是能够快速的结束这道题的关键。

实验七　Corn Field (POJ 3254)

7.1　题目描述

7.1.1　Description

Farmer John has purchased a lush new rectangular pasture composed of M by $N(1 \leq M \leq 12; 1 \leq N \leq 12)$ square parcels. He wants to grow some yummy corn for the cows on a number of squares. Regrettably, some of the squares are infertile and can't be planted. Canny FJ knows that the cows dislike eating close to each other, so when choosing which squares to plant, he avoids choosing squares that are adjacent; no two chosen squares share an edge. He has not yet made the final choice as to which squares to plant.

Being a very open-minded man, Farmer John wants to consider all possible options for how to choose the squares for planting. He is so open-minded that he considers choosing no squares as a valid option! Please help Farmer John determine the number of ways he can choose the squares to plant.

7.1.2　Input

Line 1: Two space-separated integers: M and N

Lines 2...M+1: Line i+1 describes row i of the pasture with N space-separated integers indicating whether a square is fertile (1 for fertile, 0 for infertile)

7.1.3　Output

Line 1: One integer: the number of ways that FJ can choose the squares modulo 100,000,000.

7.1.4　Sample Input

```
2 3
1 1 1
0 1 0
```

7.1.5 Sample Output

9

7.1.6 Hint

Number the squares as follows:

1 2 3
4

There are four ways to plant only on one squares (1, 2, 3, or 4), three ways to plant on two squares (13, 14, or 34), 1 way to plant on three squares (134), and one way to plant on no squares. 4+3+1+1=9.

7.2 算法设计

由于 $1 \leq N \leq 12$，并且每一片田只可能有能种草和不能种草两种属性，因此可以使用状态压缩。使用一张长为 M 表 I 来表示这是否为不肥沃的，这样直接用每一行的田地属性与种草状态进行与操作，若不为 0 则表示这是一种非法状态。

考虑玩状态压缩后考虑题目的求解方法。使用动态规划的方法，使用一张 $M \times 2^N$ 的表 D，其中 $D[i,j]$ 表示当第 i 行状态为 j 时由多少种可行的方法。显然，当 $j\&I[i]$ 时是非法状态，此时 $D[i,j]$ 应该为 0，否则 $D[i,j]$ 应该为上一行所有与之不发生冲突的状态的可能性之和。因此可以得到：

$$D[i,j] = \begin{cases} 0 & ,j\&I[i] \neq 0 \\ \sum\limits_{k\&j=0} D[i-1,k] & ,j\&I[i] = 0 \end{cases}$$

然后根据此状态转移方程依次对于 D 中的每一行进行迭代即可。最终的结果就是 D 中最后一行的和 $\sum_{k=1}^{2^N} D[M,k]$。第一行的初始状态应该为：

$$D[1,j] = \begin{cases} 0 & ,j\&I[1] \neq 0 \\ 1 & ,j\&I[1] = 0 \end{cases}$$

依照初始状态以及状态转移方程的算法如下：

Algorithm 7: CornField 问题求解算法

procedure CornField(I)
  $D \leftarrow$ a $|M| \times |2^N|$ table of 0 ▷ 状态转移表
  $D[1,k] \leftarrow (k\&I[1] = 0?1:0)$
  for $i = 2$ to M **do**
    for $j = 1$ to 2^N **do**
      if $j\&I[i] = 0$ **then**
        $D[i,j] \leftarrow \sum_{k\&j=0} D[i-1,k]$
  return $\sum\limits_{k=1}^{2^N} D[M,k]$

其对应的流程图如7.1所示。在程序开始前首先初始化状态转移表 D，然后初始化状态转移表的第一行。初始化完成后对于 D 接下来的每一行进行迭代，通过 D 第 i 行与第 $i-1$ 行的关系来计算整张状态转移表。最后，返回 D 最后银行所有元素的和作为答案。

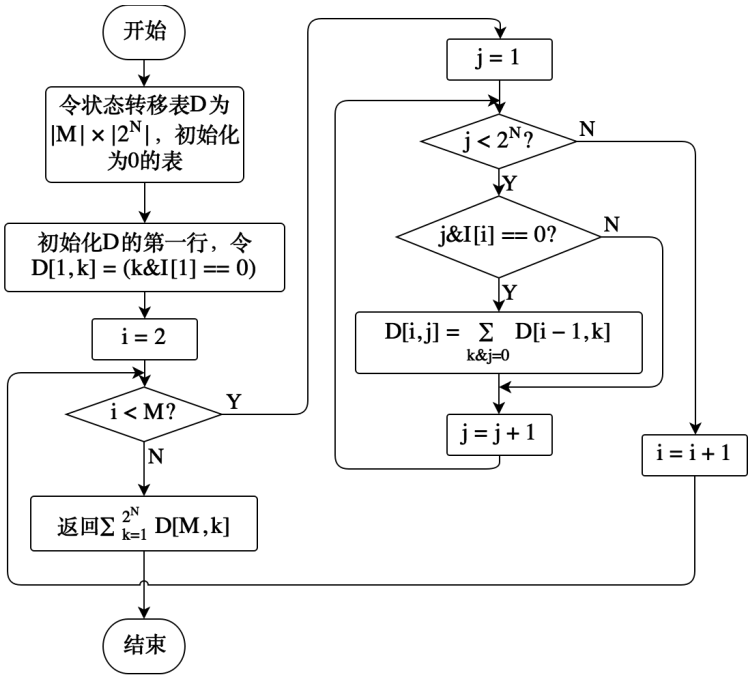

图 7.1: 程序流程图

7.3 测试分析

如7.2所示，对于 POJ 提供的测试用例，程序在 1MS 内全部通过，由此可以说明程序运行正确。

Run ID	User	Problem	Result	Memory	Time	Language	Code Length	Submit Time
18011547	husixu1	3254	Accepted	1120K	0MS	G++	1481B	2018-01-02 15:38:30

图 7.2: POJ 测试结果

7.4 技术总结

解决本题的关键在于使用状态压缩，利用一个数的不同二进制位来表示多个实体的二元状态，这样既能够节省空间也能够批量计算节省所需要花费的时间。此外，能够成功的找到状态转移表的两个维度也是解决本题的基础，这一步需要配合状态压缩来完成。

实验八　Paid Roads (POJ 3411)

8.1　题目描述

8.1.1　Description

A network of m roads connects N cities (numbered from 1 to N). There may be more than one road connecting one city with another. Some of the roads are paid. There are two ways to pay for travel on a paid road i from city a_i to city b_i :

- in advance, in a city c_i (which may or may not be the same as a_i);
- after the travel, in the city b_i.

The payment is P_i in the first case and R_i in the second case.
Write a program to find a minimal-cost route from the city 1 to the city N.

8.1.2　Input

The first line of the input contains the values of N and m. Each of the following m lines describes one road by specifying the values of $a_i, b_i, c_i, P_i, R_i (1 \leq i \leq m)$. Adjacent values on the same line are separated by one or more spaces. All values are integers, $1 \leq m, N \leq 10, 0 \leq P_i, R_i \leq 100, P_i \leq R_i (1 \leq i \leq m)$.

8.1.3　Output

The first and only line of the file must contain the minimal possible cost of a trip from the city 1 to the city N. If the trip is not possible for any reason, the line must contain the word 'impossible'.

8.1.4　Sample Input

```
4  5
1  2  1  10  10
2  3  1  30  50
3  4  3  80  80
2  1  2  10  10
```

1 3 2 10 50

8.1.5 Sample Output

110

8.2 算法设计

本题是单源最短路径算法的一个扩充，可以使用每个节点带有状态的 dijkstra 算法实现。但观察题目发现存在 $1 \leq m, N \leq 10$ 这一条件，也就是说城市的数量和路的数量都不超过 10，是一个非常小的数据集，因此最终采用了较为简单的深度优先搜索（DFS）来解决本题。

使用递归调用自身的方法实现深度优先搜索，在每一层递归时判断当前递归的这一条边所对应的 c_i 是否在已经经过了的那些城市中，若是则使用较低的价格 P_i，否则使用 R_i，在搜索时需要注意递归的结束条件，由于总共只有 10 条边，因此在经过某个城市 5 次后即可断定已经进入死循环，可以结束当前递归，因此在每层递归中传入一个表 V 来表示每个城市经过的次数，传入 n 表示当前递归的城市，此外到达终点也需要结束当前递归。最终实现的算法如下：

Algorithm 8: PaidRoads 问题求解算法

```
1:  procedure PaidRoads(V, n)
2:      if n = N then
3:          return 0;
4:      if V[n] ≥ 5 then
5:          return ∞;
6:      minSum ← ∞
7:      for each node b_i adjacent to node n do          ▷ 与当前节点 n 相邻的每个节点
8:          if V[c_i] ≠ 0 then
9:              sum ← P_i
10:         else
11:             sum ← R_i
12:         V[b_i] ← V[b_i] + 1
13:         p ← PaidRoads(V, b_i)                        ▷ 从当前节点到 b_i 的花费
14:         V[b_i] ← V[b_i] − 1
15:         if p ≠ ∞ then
16:             sum ← sum + p
17:             if sum < minSum then
18:                 minSum ← sum;
19:     return minSum
```

算法对应的流程图如8.1所示。在每一层递归开始钱，首先判断是否已经到达终点，若是则返回 0，否则判断是否是第 5 次访问当前节点，若是则返回 ∞ 表示不能从这一节点找到到达终点的路径。之后

对于当前节点的所有邻接节点进行遍历并递归调用自身，来找到从当前节点到终点的最短路径，并返回。若不能找到最短路径则返回 ∞。此算法的调用者只需传入初始化为 0 的列表 V 以及节点 1 即可。

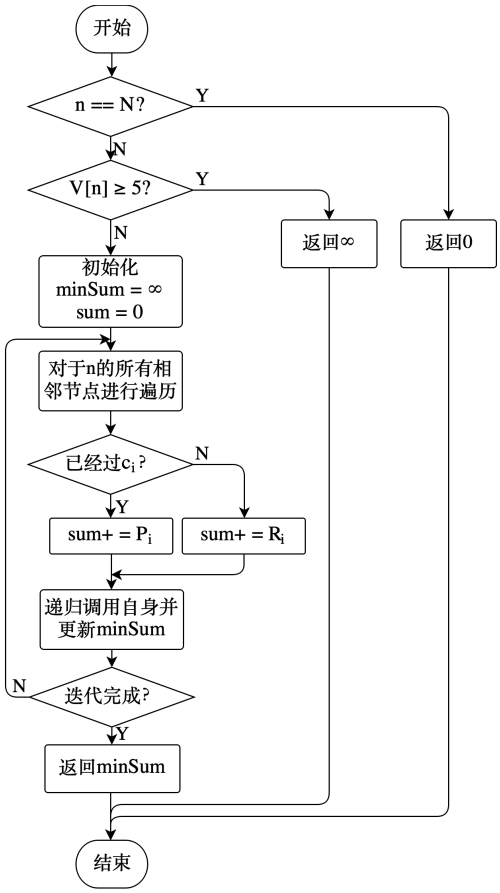

图 8.1: 程序流程图

8.3 测试分析

如8.2所示，对于 POJ 提供的测试用例，程序在 32MS 内全部通过，由此可以说明程序运行正确。

Run ID	User	Problem	Result	Memory	Time	Language	Code Length	Submit Time
18055073	husixu1	3411	Accepted	704K	32MS	G++	1679B	2018-01-17 23:49:17

图 8.2: POJ 测试结果

8.4 技术总结

实现本算法最重要的部分就是找到递归停止的条件，以及每一层递归根据已经经过的路径进行判断，只要能够正确的对于这两个条件进行判断，就能够在较短的时间内对于整个图正确地完成 DFS。

附录 A　源代码

A.1　最近点对问题源代码

```cpp
#include<iostream>
#include<fstream>
#include<vector>
#include<utility>
#include<algorithm>
#include<climits>

using namespace std;
typedef pair<int, int> Dot;
typedef vector<Dot> DotList;

inline uint64_t distanceSquare(const Dot &dotA, const Dot &dotB) {
    return (dotA.first - dotB.first) * (dotA.first - dotB.first)
            + (dotA.second - dotB.second) * (dotA.second - dotB.second);
}

inline uint64_t distanceSquare(const int posA, const int posB){
    return (posA - posB) * (posA - posB);
}

/// @brief find nearest dots
/// @param dots dots
/// @param result the result
/// @return nearest distance's square
uint64_t findNearestDotPairs(const DotList &dots, DotList &result);

int main() {
    // open input file and output file
    ifstream inFile;
    inFile.open("in.dat");
    ofstream outFile;
    outFile.open("out.dat");

```

```
34      // total num
35      unsigned groupNum;
36      inFile >> groupNum;
37
38      // groups of test
39      for (unsigned i = 0; i < groupNum; ++i) {
40          unsigned dotNum;
41          int x, y;
42          DotList dots;
43          inFile >> dotNum;
44
45          // read dot axies
46          for (unsigned i = 0; i < dotNum; ++i) {
47              inFile >> x >> y;
48              dots.push_back({x, y});
49          }
50
51          // sort according to y, then x
52          stable_sort(dots.begin(), dots.end(), [&](const Dot & a, const Dot & b)->
                bool {
53              return a.second < b.second;
54          });
55          stable_sort(dots.begin(), dots.end(), [&](const Dot & a, const Dot & b)->
                bool {
56              return a.first < b.first;
57          });
58
59
60          // start calculating
61          DotList result;
62          findNearestDotPairs(dots, result);
63
64          // output
65          for (auto dot : result)
66              outFile << dot.first << " " << dot.second << " ";
67          outFile << endl;
68      }
69      inFile.close();
70      outFile.close();
71      return 0;
72  }
73
74  uint64_t findNearestDotPairs(const DotList &dots, DotList &result) {
75      if (dots.size() < 2)
76          return UINT64_MAX;
77
```

```
78      // find left and right
79      vector<pair<int, int>> resultLeft, resultRight, resultMiddle;
80      uint64_t minDistanceL = findNearestDotPairs(DotList(dots.begin(), dots.begin
            () + dots.size() / 2), resultLeft);
81      uint64_t minDistanceR = findNearestDotPairs(DotList(dots.begin() + dots.size
            () / 2, dots.end()), resultRight);
82      uint64_t minDistanceM = minDistanceL < minDistanceR ? minDistanceL :
            minDistanceR;
83
84      Dot midDot = dots[dots.size() / 2];
85
86      // iterate through all dots
87      for (unsigned i = 0; i < dots.size(); ++i) {
88          // do not calculate dots left to the middle stripe
89          if (dots[i].first < midDot.first
90                  && distanceSquare(dots[i].first, midDot.first) > minDistanceM)
91              continue;
92
93          // break if all middle dots reside at left part are calaulated (dots are
                sorted so yeah)
94          if (dots[i].first >= midDot.first)
95              break;
96
97          // calculate through the righter half of the middle stripe
98          for (unsigned j = i; j < dots.size(); ++j) {
99              if (dots[j].first < midDot.first)
100                 continue;
101
102             // restrict to minDistance * 2 minDistance rectangle
103             if (distanceSquare(dots[j].second, dots[i].second) > minDistanceM)
104                 continue;
105
106             // calculation finished
107             if (dots[j].first > midDot.first &&
108                     distanceSquare(dots[j].first, midDot.first) > minDistanceM)
109                 break;
110
111             if (distanceSquare(dots[i], dots[j]) < minDistanceM) {
112                 // found a better one
113                 minDistanceM = distanceSquare(dots[i], dots[j]);
114                 resultMiddle.clear();
115                 resultMiddle.push_back(dots[i]);
116                 resultMiddle.push_back(dots[j]);
117             } else if (distanceSquare(dots[i], dots[j]) == minDistanceM) {
118                 // found a equal one
119                 resultMiddle.push_back(dots[i]);
```

```
120                 resultMiddle.push_back(dots[j]);
121             }
122         }
123     }
124     // combine
125     // minDistanceL can only >= minDistanceM
126     if (minDistanceL == minDistanceM)
127         resultMiddle.insert(resultMiddle.end(), resultLeft.begin(), resultLeft.
                end());
128     // minDistanceR can only >= minDistanceM
129     if (minDistanceR == minDistanceM)
130         resultMiddle.insert(resultMiddle.end(), resultRight.begin(), resultRight.
                end());
131
132     result = resultMiddle;
133     return minDistanceM;
134 }
```

A.2 大数乘法源代码

```
 1 #include<iostream>
 2 #include<fstream>
 3 #include<vector>
 4 #include<string>
 5 #include<algorithm>
 6
 7 using namespace std;
 8
 9 /// @brief add two big number
10 /// @param a one big number
11 /// @param b another big number
12 /// @return a+b
13 string bigNumAdd(const string &a, const string &b);
14
15 /// @brief substract big number
16 /// @param a one big number
17 /// @param b another big number
18 /// @return a-b
19 string bigNumSub(const string &a, const string &b);
20
21 /// @brief multiply two big number
22 /// @param a one big number
23 /// @param b another big number
24 /// @return a*b
25 string bigNumMult(const string &a, const string &b);
```

```
26
27  /// @brief add two unsigned big number
28  /// @param a one unsigned big number
29  /// @param b another unsigned big number
30  /// @return a+b
31  string bigNumAddHelper(const string &a, const string &b);
32
33  /// @brief substract two unsigned big number, and a must be bigger than b
34  /// @param a one unsigned big number
35  /// @param b another unsigned big number, less than a
36  /// @return a-b
37  string bigNumSubHelper(const string &a, const string &b);
38
39  /// @brief multiply two unsigned big number using karatsuba
40  /// @param a one unsigned big number
41  /// @param b another unsigned big number
42  /// @return a*b
43  string bigNumMultHelper(const string &a, const string &b);
44
45  /// @brief compair two unsigned big number
46  /// @param a one unsigned big number
47  /// @param b another unsigned big number
48  /// @return (a < b)
49  bool bigNumCompareHelper(const string &a, const string &b);
50
51  /// @brief return the negative of a negative big number
52  /// @param a a negative number, must start with '-'
53  /// @return -a
54  inline string negative(const string &a) {
55      return string(a.begin() + 1, a.end());
56  }
57
58  int main() {
59      unsigned casenNum;
60      ifstream inFile;
61      ofstream outFile;
62      inFile.open("in.dat");
63      outFile.open("out.dat");
64
65      // read case numbers
66      inFile >> casenNum;
67
68      // calculate each case
69      for (unsigned i = 0; i < casenNum; ++i) {
70          string a, b;
71          int op;
```

```
72
73          // read data
74          inFile >> a >> b >> op;
75
76          // do operation
77          // TODO: deal with negative number
78          switch (op) {
79              case 1:
80                  outFile << bigNumAdd(a, b) << endl;
81                  break;
82              case 2:
83                  outFile << bigNumSub(a, b) << endl;
84                  break;
85              case 3:
86                  outFile << bigNumMult(a, b) << endl;
87                  break;
88              default:
89                  break;
90          }
91      }
92
93      inFile.close();
94      outFile.close();
95      return 0;
96  }
97
98  string bigNumAdd(const string &a, const string &b) {
99      bool isANegative = false, isBNeagtive = false;
100     string result;
101
102     if (a.front() == '-')
103         isANegative = true;
104     if (b.front() == '-')
105         isBNeagtive = true;
106
107     if (!isANegative && !isBNeagtive)
108         result =  bigNumAddHelper(a, b);
109
110     if (isANegative && isBNeagtive)
111         result =  "-" + bigNumAddHelper(negative(a), negative(b));
112
113     if (isANegative && !isBNeagtive) {
114         // |a| < b ?
115         if (bigNumCompareHelper(negative(a), b)) {
116             result =  bigNumSubHelper(b, negative(a));
117         } else {
```

```
118             result =  "-" + bigNumSubHelper(negative(a), b);
119         }
120     }
121
122     if (!isANegative && isBNeagtive) {
123         // |b| < a?
124         if (bigNumCompareHelper(negative(b), a)) {
125             result =  bigNumSubHelper(a, negative(b));
126         } else {
127             result =  "-" + bigNumSubHelper(negative(b), a);
128         }
129     }
130     if (result == "-0")
131         return "0";
132     return result;
133 }
134
135 string bigNumSub(const string &a, const string &b) {
136     if (b.front() == '-') {
137         return bigNumAdd(a, negative(b));
138     } else {
139         return bigNumAdd(a, "-" + b);
140     }
141 }
142
143 string bigNumAddHelper(const string &a, const string &b) {
144     int carry = 0;
145     string result;
146
147     int indexA = a.size() - 1, indexB = b.size() - 1, actual;
148     for (; indexA >= 0  && indexB >= 0 ; --indexA, --indexB) {
149         actual = carry + static_cast<int>(a[indexA]) - '0' + static_cast<int>(b[
                indexB]) - '0';
150         result.push_back(actual % 10 + '0');
151         carry = actual / 10;
152     }
153     for (; indexA >= 0; --indexA) {
154         actual = carry + static_cast<int>(a[indexA]) - '0';
155         result.push_back(actual % 10 + '0');
156         carry = actual / 10;
157     }
158     for (; indexB >= 0; --indexB) {
159         actual = carry + static_cast<int>(b[indexB]) - '0';
160         result.push_back(actual % 10 + '0');
161         carry = actual / 10;
162     }
```

```
163     if (carry)
164         result.push_back(carry + '0');
165 
166     while (result.size() && result.back() == '0')
167         result.pop_back();
168     if (result.empty())
169         result.push_back('0');
170     reverse(result.begin(), result.end());
171     return result;
172 }
173 
174 string bigNumMult(const string &a, const string &b) {
175     bool isNegative = false;
176     string absA, absB;
177     string result;
178 
179     if (a.front() == '-') {
180         isNegative = !isNegative;
181         absA = negative(a);
182     } else {
183         absA = a;
184     }
185 
186     if (b.front() == '-') {
187         isNegative = !isNegative;
188         absB = negative(b);
189     } else {
190         absB = b;
191     }
192 
193     return (isNegative ? "-" : "") + bigNumMultHelper(absA, absB);
194 }
195 
196 string bigNumSubHelper(const string &a, const string &b) {
197     int borrow = 0;
198     string result;
199 
200     int indexA = a.size() - 1, indexB = b.size() - 1, actual;
201     for (; indexA >= 0 && indexB >= 0; --indexA, --indexB) {
202         actual = a[indexA] - b[indexB] - borrow;
203         result.push_back((actual + 10) % 10 + '0');
204         borrow = actual < 0 ? 1 : 0;
205     }
206     for (; indexA >= 0; --indexA) {
207         actual = a[indexA] - '0' - borrow;
208         result.push_back((actual + 10) % 10 + '0');
```

```
209         borrow = actual < 0 ? 1 : 0;
210     }
211
212     while (result.size() &&  result.back() == '0')
213         result.pop_back();
214     if (result.empty())
215         result.push_back('0');
216     reverse(result.begin(), result.end());
217     return result;
218 }
219
220 string bigNumMultHelper(const string &a, const string &b) {
221     if (a.size() == 1) {
222         string result("0");
223         for (int i = 0; i < a[0] - '0'; ++i)
224             result = bigNumAdd(result, b);
225         return result;
226     } else if (b.size() == 1) {
227         string result("0");
228         for (int i = 0; i < b[0] - '0'; ++i)
229             result = bigNumAdd(result, a);
230         return result;
231     } else if (a.size() == 0 || b.size() == 0) {
232         return "";
233     }
234
235     unsigned halfSize = a.size() > b.size() ? a.size() / 2 : b.size() / 2;
236
237     string high1(a.begin(), (halfSize > a.size() ? a.begin() : a.end() - halfSize
            ));
238     string low1((halfSize > a.size() ? a.begin() : a.end() - halfSize), a.end());
239     string high2(b.begin(), (halfSize > b.size() ? b.begin() : b.end() - halfSize
            ));
240     string low2((halfSize > b.size() ? b.begin() : b.end() - halfSize), b.end());
241
242     string z0 = bigNumMultHelper(low1, low2);
243     string z1 = bigNumMultHelper(bigNumAdd(low1, high1), bigNumAdd(low2, high2));
244     string z2 = bigNumMultHelper(high1, high2);
245
246     string temp1 = z2, temp2 = bigNumSub(bigNumSub(z1, z2), z0);
247     for (unsigned i = 0; i < 2 * halfSize; ++i)
248         temp1.push_back('0');
249     for (unsigned i = 0; i < halfSize; ++i)
250         temp2.push_back('0');
251     return bigNumAdd(bigNumAdd(temp1, temp2), z0);
252 }
```

```
253
254 bool bigNumCompareHelper(const string &a, const string &b) {
255     unsigned startA = 0, startB = 0;
256     for (; startA < a.size() && a[startA] == '0'; ++startA);
257     for (; startB < b.size() && b[startB] == '0'; ++startB);
258
259     if (a.size() - startA < b.size() - startB)
260         return true;
261     if (a.size() - startA > b.size() - startB)
262         return false;
263
264     for (; startA < a.size(); ++startA, ++startB) {
265         if (a[startA] < b[startB])
266             return true;
267         if (a[startA] > b[startB])
268             return false;
269     }
270     return false;
271 }
```

A.3 最优二分查找树源代码

```
 1 #include<iostream>
 2 #include<fstream>
 3 #include<vector>
 4 #include<array>
 5
 6 using namespace std;
 7
 8 /// @brief construct optimal BST
 9 /// @param[in] root the root table
10 /// @param[in] i pass 0
11 /// @param[in] j pass root[0].size() - 1
12 /// @param[out] result a array of (i, j, d, m), in which d could only be 0, 1.
13 /// if d == 0 then Ki is the left child of Kj, else Ki is the right child of Kj,
14 /// m indicates if i is a fake node
15 /// @return something you should not care
16 int constructOptimalBST(const vector<vector<int>> &root, int i, int j, vector<
       array<int, 4>> &result);
17
18 static int fakeCounter;
19
20 int main() {
21     ifstream inFile;
22     ofstream outFile;
```

```
23
24      inFile.open("in.dat");
25      outFile.open("out.dat");
26
27      unsigned matrixSize;
28      inFile >> matrixSize;
29
30      vector<vector<int>> root(matrixSize, vector<int>(matrixSize, 0));
31      for (unsigned i = 0; i < matrixSize; ++i)
32          for (unsigned j = 0; j < matrixSize; ++j)
33              inFile >> root[i][j];
34
35      vector<array<int, 4>> result;
36
37      fakeCounter = -1;
38      int rootNode = constructOptimalBST(root, 0, matrixSize - 1, result);
39
40      outFile << string("k") << rootNode << string("是根") << endl;
41      for (auto i : result) {
42          outFile << string(i[3] ? "d" : "k") << (i[3] ? i[0] : i[0])
43                  << string("是k") << i[1]
44                  << string("的") << (i[2] ? string("右") : string("左"))
45                  << string("孩子") << endl;
46      }
47
48      inFile.close();
49      outFile.close();
50      return 0;
51  }
52
53  int constructOptimalBST(const vector<vector<int>> &root, int i, int j, vector<
        array<int, 4>> &result) {
54      if (i > j){
55          ++fakeCounter;
56          return -1;
57      }
58
59      int child;
60      // first insert right to front
61      if((child = constructOptimalBST(root, root[i][j], j, result)) != -1)
62          result.insert(result.begin(), {child, root[i][j], 1, 0});
63      else
64          result.insert(result.begin(), {static_cast<int>(root.size()) -
                fakeCounter, root[i][j], 1, 1});
65
66      // then insert left to front
```

```
67      if((child = constructOptimalBST(root, i, root[i][j] - 2,  result)) != -1)
68          result.insert(result.begin(), {child, root[i][j], 0, 0});
69      else
70          result.insert(result.begin(), {static_cast<int>(root.size()) -
                fakeCounter, root[i][j], 0, 1});
71
72      return root[i][j];
73  }
```

A.4 Floyd-Warshall 最短路径算法源代码

```
 1  #include<iostream>
 2  #include<fstream>
 3  #include<vector>
 4  #include<algorithm>
 5
 6  using namespace std;
 7  static const int Infinity = 100;
 8  static const int InfinityPathLen = 32767;
 9
10  /// @brief find all shortest path
11  /// @param matrix the input matrix
12  /// @param resultLen shortest len matrix
13  /// @param resultPath shortest path matrix
14  void allPaths(const vector<vector<int>> &matrix,
15          vector<vector<int>> &resultLen,
16          vector<vector<vector<int>>> &resultPath);
17
18  int main() {
19      ifstream inFile;
20      ofstream outFile;
21      inFile.open("in.dat");
22      outFile.open("out.dat");
23
24      unsigned caseNum, matrixSize;
25      inFile >> caseNum;
26      outFile << caseNum << endl;
27      // iterate all cases
28      while (caseNum--) {
29          inFile >> matrixSize;
30          outFile << matrixSize << endl;
31          vector<vector<int>> matrix(matrixSize, vector<int>(matrixSize, 0));
32          vector<vector<int>> shortestLength(matrixSize, vector<int>(matrixSize,
                InfinityPathLen));
```

```
33          vector<vector<vector<int>>> shortestPath(matrixSize, vector<vector<int>>(
                matrixSize, vector<int>()));
34          // read input
35          for (unsigned i = 0; i < matrixSize; ++i)
36              for (unsigned j = 0; j < matrixSize; ++j)
37                  inFile >> matrix[i][j];
38
39          // do calculation
40          allPaths(matrix, shortestLength, shortestPath);
41
42          // output len
43          for (unsigned i = 0; i < matrixSize; ++i) {
44              for (unsigned j = 0; j < matrixSize; ++j)
45                  outFile << shortestLength[i][j] << " ";
46              outFile << endl;
47          }
48          // output path
49          for (unsigned i = 0; i < matrixSize; ++i) {
50              for (unsigned j = 0; j < matrixSize; ++j) {
51                  if (i == j)
52                      outFile << i + 1;
53                  else if (shortestPath[i][j].size() == 0)
54                      outFile << "NULL";
55                  else
56                      for (auto node : shortestPath[i][j])
57                          outFile << node + 1 << " ";
58                  outFile << endl;
59              }
60          }
61      }
62
63      inFile.close();
64      outFile.close();
65      return 0;
66  }
67
68  void allPaths(const vector<vector<int>> &matrix,
69          vector<vector<int>> &resultLen,
70          vector<vector<vector<int>>> &resultPath) {
71      unsigned matrixSize = matrix.size();
72      vector<vector<int>> next(matrixSize, vector<int>(matrixSize, 0));
73
74      // initialization
75      resultLen = matrix;
76      for (unsigned i = 0; i < matrixSize; ++i)
77          for (unsigned j = 0; j < matrixSize; ++j) {
```

```
78              if (matrix[i][j] >= Infinity)
79                  next[i][j] = -1;
80              else
81                  next[i][j] = j;
82          }
83
84      // Floyd-Warshall
85      for (unsigned k = 0; k < matrixSize; ++k)
86          for (unsigned i = 0; i < matrixSize; ++i)
87              for (unsigned j = 0; j < matrixSize; ++j)
88                  if (resultLen[i][j] > resultLen[i][k] + resultLen[k][j]) {
89                      resultLen[i][j] = resultLen[i][k] + resultLen[k][j];
90                      next[i][j] = next[i][k];
91                  }
92
93      // find path
94      unsigned tempNode;
95      for (unsigned i = 0; i < matrixSize; ++i) {
96          for (unsigned j = 0; j < matrixSize; ++j) {
97              if (next[i][j] == -1)
98                  continue;
99              tempNode = i;
100             resultPath[i][j].push_back(tempNode);
101             while (tempNode != j) {
102                 tempNode = next[tempNode][j];
103                 resultPath[i][j].push_back(tempNode);
104             }
105             reverse(resultPath[i][j].begin(), resultPath[i][j].end());
106         }
107     }
108 }
```

A.5 Wooden Sticks(POJ 1065) 算法源代码

```
1  #include<iostream>
2  #include<fstream>
3  #include<vector>
4  #include<utility>
5  #include<algorithm>
6  #include<climits>
7
8  using namespace std;
9  typedef pair<int, int> Stick;
10
11 bool compareF(const Stick &a, const Stick &b){return a.first < b.first;}
```

```
12 bool compareS(const Stick &a, const Stick &b){return a.second < b.second;}
13
14 int main(){
15     unsigned caseNum, stickNum;
16     cin >> caseNum;
17
18     while(caseNum--){
19         cin >> stickNum;
20         vector<Stick> sticks;
21         int result = 0;
22         vector<int> curMaxList;
23
24         int length, weight;
25
26         // read inputs
27         while(stickNum--){
28             cin >> length >> weight;
29             sticks.push_back(pair<int, int>(length, weight));
30         }
31
32         stable_sort(sticks.begin(), sticks.end(), &compareS);
33         stable_sort(sticks.begin(), sticks.end(), &compareF);
34
35         for(unsigned i=0; i < sticks.size(); ++i){
36             unsigned j=0;
37             for(j=0; j < curMaxList.size(); ++j)
38                 if(sticks[i].second >= curMaxList[j]){
39                     curMaxList[j] = sticks[i].second;
40                     break;
41                 }
42             if(j == curMaxList.size()){
43                 curMaxList.push_back(sticks[i].second);
44                 ++result;
45             }
46         }
47         cout << result << endl;
48     }
49     return 0;
50 }
```

A.6 Gone Fishing(POJ 1042) 算法源代码

```
1 #include<climits>
2 #include<iostream>
3 #include<vector>
```

```
 4 #include<queue>
 5 #include<utility>
 6
 7 using namespace std;
 8
 9 struct Pool {
10     int index;
11     int f;
12     int d;
13     friend bool operator< (Pool a, Pool b);
14 };
15 bool operator< (Pool a, Pool b) {
16     if (a.f == b.f)
17         return a.index > b.index;
18     return a.f < b.f;
19 }
20
21 int main() {
22     bool first = true;
23     while (true) {
24         int n, h, temp;
25         vector<int> t;
26         cin >> n;
27         if (n == 0)
28             break;
29         vector<Pool> pools(n);
30         cin >> h;
31         h *= 12;
32         for (int i = 0; i < n; ++i) {
33             cin >> temp;
34             pools[i].index = i;
35             pools[i].f = temp;
36         }
37         for (int i = 0; i < n; ++i) {
38             cin >> temp;
39             pools[i].d = temp;
40         }
41         for (int i = 0; i < n - 1; ++i) {
42             cin >> temp;
43             t.push_back(temp);
44         }
45
46         vector<int> bestPath(n, 0);
47         int bestResult = 0;
48         // enumerate all possible terminus
49         for (int terminus = 0; terminus < n; ++terminus) {
```

```
50              // reduce interval time
51              int hour = h;
52              for (int i = 0; i < terminus; ++i)
53                  hour -= t[i];
54
55              // init
56              priority_queue<Pool> priQueue;
57              for (int i = 0; i <= terminus; ++i)
58                  priQueue.push(pools[i]);
59
60              vector<int> roundPath(n, 0);
61              int roundResult = 0;
62
63              // run simulation
64              while (hour--) {
65                  // do move
66                  ++roundPath[priQueue.top().index];
67                  roundResult += priQueue.top().f;
68
69                  Pool tempPool = priQueue.top();
70                  tempPool.f -= priQueue.top().d;
71                  if (tempPool.f < 0)
72                      tempPool.f = 0;
73                  priQueue.pop();
74                  priQueue.push(tempPool);
75              }
76              if (roundResult > bestResult) {
77                  bestResult = roundResult;
78                  bestPath = roundPath;
79              }
80          }
81          if (first)
82              first = false;
83          else
84              cout << endl;
85
86          for (unsigned i = 0; i < bestPath.size(); ++i) {
87              if (i)
88                  cout << ", ";
89              cout << bestPath[i] * 5;
90          }
91          cout << endl <<  "Number of fish expected: " << bestResult << endl;
92      }
93      return 0;
94  }
```

A.7 Corn Field(POJ 3254) 算法源代码

```
#include<iostream>
#include<vector>
#include<cstring>

using namespace std;

const long long mod = 100000000;

// table[i][j] is the possibility when line i is at status j
// (only take former i lines into consideration)
unsigned long long table[12][1 << 12];
int infertiled[12];

inline bool isValid(int i, int s) {
    return !((s & (s << 1)) || (s & infertiled[i]));
}

inline bool isAdjacent(int a, int b) {
    return a & b;
}

int main() {
    int m, n;
    cin >> n >> m;

    memset(table, 0, sizeof(table));
    memset(infertiled, 0, sizeof(infertiled));

    int bit;
    for (int i = 0; i < n; ++i)
        for (int j = 0; j < m; ++j) {
            cin >> bit;
            infertiled[i] |= ((bit ^ 1) << j);
        }

    // initialize
    for (int status = 0; status < (1 << m); ++status)
            table[0][status] = isValid(0, status);

    // iterate
    for (int i = 1; i < n; ++i) {
        for (int status = 0; status < (1 << m); ++status) {
            table[i][status] = 0;

```

```
45              if (!isValid(i, status))
46                  continue;
47
48              for (int previous = 0; previous < (1 << m); ++previous)
49                  if(isValid(i-1, previous) && !isAdjacent(previous, status))
50                      (table[i][status] += table[i - 1][previous]) %= mod;
51          }
52      }
53
54      unsigned long long result = 0;
55      for (int i = 0; i < (1 << m); ++i) {
56          result += table[n - 1][i];
57          result %= mod;
58      }
59
60      cout << result << endl;
61      return 0;
62  }
```

A.8 Paid Roads(POJ 3411) 算法源代码

```
1   #include<iostream>
2   #include<vector>
3   #include<climits>
4
5   using namespace std;
6   const int infinity = INT_MAX;
7
8   struct Edge {
9       int adjNode;
10      int cNode;
11      int cPrice;
12      int adjPrice;
13  };
14
15  int DFS(const vector<vector<Edge> > &cities,
16          vector<int> &visit,
17          int node, int n);
18
19  int main() {
20      int roadNum, cityNum;
21      cin >> cityNum >> roadNum;
22
23      // read edges
24      vector<vector<Edge> > cities(cityNum + 1, vector<Edge>());
```

```
25      for (int i = 0; i < roadNum; ++i) {
26          int a, b, c, p, r;
27          cin >> a >> b >> c >> p >> r;
28          Edge temp;
29          temp.adjNode = b;
30          temp.cNode = c;
31          temp.cPrice = p;
32          temp.adjPrice = r;
33          cities[a].push_back(temp);
34      }
35
36      // DFS
37      vector<int> visit(cityNum + 1, 0);
38      int result = DFS(cities, visit, 1, cityNum);
39      if(result == -1)
40          cout << "impossible" << endl;
41      else
42          cout <<  result << endl;
43
44      return 0;
45  }
46
47  int DFS(const vector<vector<Edge> > &cities,
48          vector<int> &visit,
49          int node, int n ) {
50
51      if (node == n)
52          return 0;
53      if (visit[node] >= 5)
54          return -1;
55
56      ++visit[node];
57
58      int minSum = INT_MAX, sum, add;
59      bool valid = false;
60
61      for (unsigned i = 0; i < cities[node].size(); ++i) {
62          if(visit[cities[node][i].cNode])
63              sum = cities[node][i].cPrice;
64          else
65              sum = cities[node][i].adjPrice;
66
67          if((add = DFS(cities, visit, cities[node][i].adjNode, n)) != -1){
68              sum += add;
69              valid = true;
70              if(sum < minSum)
```

```
71                  minSum = sum;
72          }
73      }
74
75      --visit[node];
76      if(valid)
77          return minSum;
78      else
79          return -1;
80  }
```