《光棱坦克》解题报告

IOI 2010 国家集训队论文

李新野

2010

李新野

汕头市金山中学

2009 年 11 月 22 日

《光棱坦克》解题报告

目录

题意简述

给定一个点集包含 N 个平面上的点的点集 $\{(x_i,y_i)\}$，求满足以下两个条件的非空点列 $\{(x_{p_i},y_{p_i})\}$的数目：

1) 对任意 $i<j$，有 $y_{p_i}>y_{p_j}$。

2) 对任意 $i\geq 3$，有 $x_{p_{i-1}}<x_{p_i}<x_{p_{i-2}}$或 $x_{p_{i-2}}<x_{p_i}<x_{p_{i-1}}$。

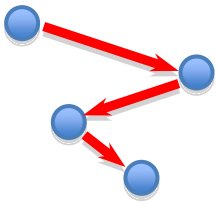
答案对一个给定的数字 Q 取模就好了。

初步分析

因为最终点列需要满足的条件只与坐标之间的大小关系有关，而跟坐标的大小没有绝对关系。因为题目保证 $i\neq j$ 时，$x_i\neq x_j$ 且 $y_i\neq y_j$。于是我们不妨让 x 坐标最小的点的 x 坐标为1，x 坐标第二小的点的 x 坐标为2，依此类推，x 坐标最大的点的 x 坐标为 N。同样的，可以让 y 坐标的取值也是1到 N。

为了后面讲述的方便，不妨按照 x 坐标的大小对所有的点排序，使得对任意的 i 有 $x_i=i$，也就是 $x_1<x_2<x_3<\mathrm{K}<x_N$。

如果我们按照顺序把相邻的每一对点之间连上线段，形成一条折线，就像是光棱坦克射出的激光一样：



于是，每一条折线相当于一种点列。这可以在后面的分析中有助于更加形象的理解题意。

搜索算法

分析

我们可以立刻想到一个正确的算法——搜索。我们可以用一个深度优先搜索，用一个堆栈纪录 p_i 的值。

如果我们暴力生成所有的点列，最后检验每个排列是否符合要求的话，那么一共需要检验 $\sum_{i=1}^{N} P_i^N$ 种点列（这里的符号 P 指的是排列数）。很明显，即使 N 非常小，这种算法也会使用过多的运行时间。

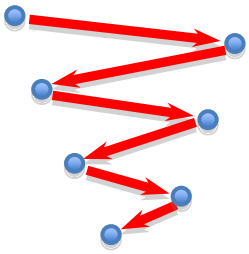
但是，如果我们在搜索的过程中剪枝，判断目前搜索到的序列是否符合题目要求的两个条件，便可以大幅度减少运行时间。

运行时间分析

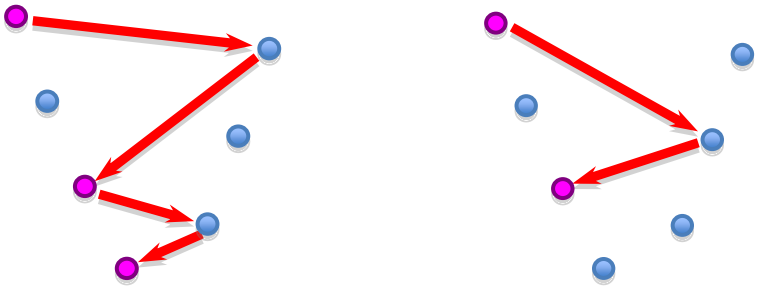
因为点列的长度任意，无论每次迭代的时候搜索到的序列长度为多少，都应该记入答案。所以加上剪枝后，搜索算法的运行时间可以近似地看成与答案（未取模）的大小成正比。

因为对任意 $i<j$，有 $y_{p_i} > y_{p_j}$，所以当属于点列的点确定下来之后，这个点列也自然确定了。于是可以证明，最终的方案数一定是小于等于 2^N-1 的，比起没有加上剪枝的搜索，速度有了明显的提升。

但是即使加上优化，该算法运行时间仍然是指数级的。考虑类似下图的极端数据：



也就是对于这种数据，把所有的点按照 y 坐标连接起来的时候，就是一个符合条件的点列。对这种数据，很容易证明答案是指数级的：顶点可以分成左边和右边两堆顶点，因为有对称性，不妨假设左边的点数目为 $\left\lfloor \frac{N}{2} \right\rfloor$，右边的点的数目为 $\left\lfloor \frac{N}{2} \right\rfloor$；因为对任何一个左边的点的非空子集，我们都可以找到一个符合条件的点列，使得属于该点列的左边的点恰好为该非空子集：



于是便证明了答案的大小至少是 $2^{\left\lceil \frac{N}{2} \right\rceil}-1$，很明显是指数级别的。

而对于随机生成的数据，虽然答案要比极端数据小很多。但是当 $N \geq 100$ 的时候，运行时间已经无法让人忍受了。

实验数据

以下是对随机数据和极端数据的答案大小和搜索算法运行时间的数据：

数据类型	N	方案数	搜索算法运行时间
随机数据	20	1062	0.001 s
随机数据	40	29324	0.008 s
随机数据	60	682689	0.203 s

随机数据	80	27990853	11.713 s
随机数据	100	23389727	7.757 s
随机数据	120	138525495	109.492 s
极端数据	10	364	0.001 s
极端数据	20	46345	0.006 s
极端数据	30	5702854	0.548 s
极端数据	40	701408690	67.408 s

进行测试的搜索程序为../Programs/tanko.cpp。测试机器 CPU: 2.66 GHz Intel Core 2 Duo; 内存: 4 GB 1067 MHz DDR3。编译程序版本: g++ 4.2.1。

参考程序

../Programs/tanko.cpp

动态规划算法

状态设计

我们观察以下点列需要满足的两个条件：

1) 第一个条件是，对任意$i<j$，有$y_{p_i}>y_{p_j}$。其实这个条件也可以写作：对任意$i\geq 2$，有$y_{p_{i-1}}>y_{p_i}$。

2) 第二个条件是，对任意$i\geq 3$，有$x_{p_{i-1}}<x_{p_i}<x_{p_{i-2}}$或$x_{p_{i-2}}<x_{p_i}<x_{p_{i-1}}$。

可以发现，当我们已经确定了点列的前K个元素（K本身不确定）的时候，第$K+1$个元素只跟第K个和第$K-1$个元素有关，第$K+2$个元素只跟第$K+1$个元素和第K个元素有关……所以，当目前确定了K个元素之后，无论K有多大，无论前$K-2$个元素是哪些点，只要最后两个元素确定，后面的状态也就确定了。

所以可以设状态 $f(i,j)$ 表示前 K 个元素确定（而 K 本身不确定），其中倒数第二个元素是点 i，最后一个元素是点 j（ $p_{K-1}=i$， $p_K=j$ ）的情况下的点列的数目。

为了方便，当 $y_i \le y_j$ 的时候，也就是点 i 不可能出现点 j 之前的情况的时候，我们设 $f(i,j)=0$。

图形解释

我们再次分析点列，或者说折线需要满足的两个条件：

1) 对任意 $i<j$，有 $y_{p_i} > y_{p_j}$。

2) 对任意 $i \ge 3$，有 $x_{p_{i-1}} < x_{p_i} < x_{p_{i-2}}$ 或 $x_{p_{i-2}} < x_{p_i} < x_{p_{i-1}}$。

首先我们来分析一下第一个条件。

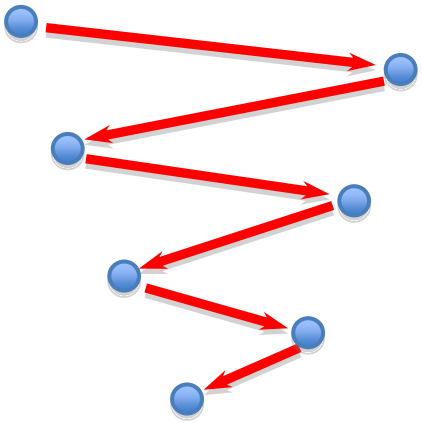
第一个条件是说，对任意 $i<j$，有 $y_{p_i} > y_{p_j}$。也就是说，对这一条折线来说，每个点的 y 坐标递增。后面的点的 y 坐标一定会大于前面的点。

然后我们来分析一下第二个条件。

因为有对任意 $i \ge 3$，有 $x_{p_{i-1}} < x_{p_i} < x_{p_{i-2}}$ 或 $x_{p_{i-2}} < x_{p_i} < x_{p_{i-1}}$。这也就是说，每个点的 x 坐标都在前面两个点的 x 坐标之间。而且我们可以试着把 $i=3,4,5\mathrm{K}$ 时的不等式联起来就有 $x_{p_1} < x_{p_3} < x_{p_5} < \mathrm{K} < x_{p_n} < \mathrm{K}\ x_{p_6} < x_{p_4} < x_{p_2}$ 或者是 $x_{p_2} < x_{p_4} < x_{p_6} < \mathrm{K} < x_{p_n} < \mathrm{K}\ x_{p_5} < x_{p_3} < x_{p_1}$。也就是说，对于每两个相邻的点，后面的所有的点的 x 坐标都在这两个点的坐标之间。

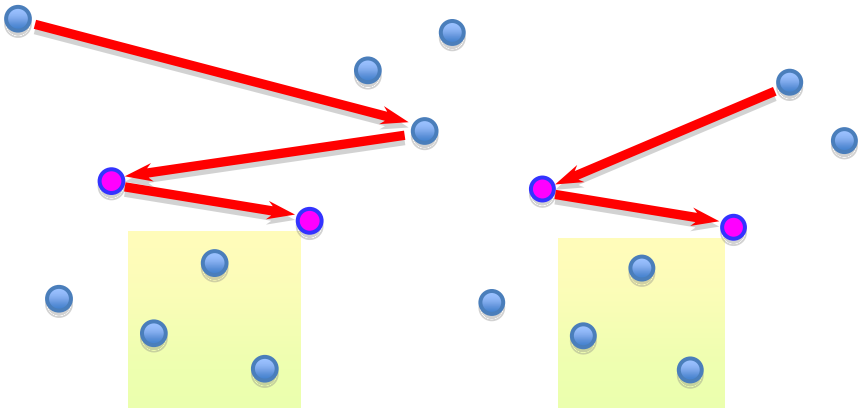
若 $x_{p_{i-1}} < x_{p_i} < x_{p_{i-2}}$，那么 $x_{p_{i-1}} < x_{p_{i+1}} < x_{p_i}$；若 $x_{p_{i-2}} < x_{p_i} < x_{p_{i-1}}$，那么 $x_{p_i} < x_{p_{i+1}} < x_{p_2}$。我们再简化一下，就是说如果有 $x_{p_{i-2}} > x_{p_{i-1}}$，那么有 $x_{p_{i-1}} < x_{p_i}$ 和 $x_{p_{i+1}} > x_{p_i}$；如果有 $x_{p_{i-2}} < x_{p_{i-1}}$，那么就有 $x_{p_{i-1}} > x_{p_i}$ 和 $x_{p_{i+1}} < x_{p_i}$。也就是说，相邻的每两个点的 x 坐标的大小关系一直在改变。

我们得到的，是一条y坐标递增；对于每两个相邻的点，后面的所有的点的x坐标的范围都在这辆个点的x坐标之间；而且相邻的每个点的x坐标的大小关系一直在改变的折线：



而且，这样子的折线也一一对应于符合条件的点列。

在已经确定了一个点列的前K个点（K不确定），而且点列的第K和$K-1$个点一定的时候。比如下面这两种情况。



红色的箭头是已经确定的折线，紫色的点是折在线最后两个点。于是，这条折线如果不只K个点，后面的点肯定是在阴影部份之中。我们可以在阴影部份内的点找符合两个条件，而且可以与已经确定的部份衔接起来的点列（包括空列）的数目。如果第$K-1$个点是点i，第K个点是点j（$p_{K-1}=i$，$p_K=j$），这个数目其实就是前面所说$f(i,j)$。

状态转移

现在我们要计算 $f(i,j)$ 的值。

分两种情况讨论并求和：

1) 如果点列只有 K 个点，也就是点 i 和点 j 是最后两个点的情况。因为前 K 个点已经确定，所以只有一种情况。

2) 如果不只 K 个点，那么枚举第 $K+1$ 个点，也就是 p_{K+1} 的值。当 $p_{K+1}=k$ 的时候，我们实际上就确定了前 $K+1$ 个点，这时候的方案数就是 $f(j,k)$。

于是可以得到 $f(i,j)$ 的状态转移方程式（注意已经假定 $y_i>y_j$，否则有 $f(i,j)=0$）：

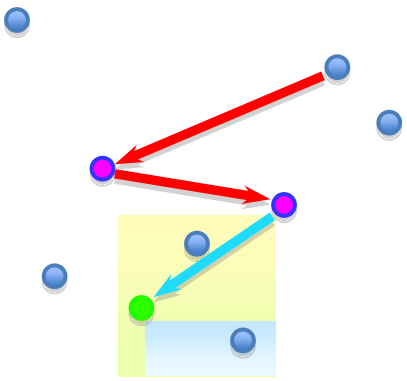
- 若 $i<j$，则有 $f(i,j)=1+\sum_{k=i+1}^{j-1} f(j,k)$。

- 若 $j<i$，则有 $f(i,j)=1+\sum_{k=j+1}^{i-1} f(j,k)$。

注意，因为我们之前已经假定 $x_i=i$。所以上式隐含条件 $i<k<j$ 和 $j<k<i$ 实际上就相当于 $x_i<x_k<x_j$ 和 $x_j<x_k<x_i$。而且我们也假定了 $y_j\le y_k$ 时 $f(j,k)=0$，也就是说，实际上我们只加上了 $y_k<y_j$ 时的 $f(j,k)$ 而已。

我们观察到该状态转移方程式只包含加法，根据同余的可加减性，所以可以直接在运算过程中就对 Q 取模，对最后答案无影响。

图形解释



如图，我们枚举原来黄绿色阴影下每个点作为点列的第$K+1$个元素。假设绿色的点是点列中新加的点k，青色箭头是折线中新加入的线段。于是，这时候，点列中在点k后的所有的点都必须在图标的青色阴影之中。这时候的方案数也就是$f(j,k)$。

动态规划的顺序

因为对这道题目来说，状态关系比较复杂，因此，可以考虑使用记忆化搜索来实现。

当然，实际上也可以直接使用动态规划。可以按照一下三种顺序来进行动态规划：

1) 我们观察到，因为有$i<k<j$或者$j<k<i$，所以有$|i-j|<|j-k|$，于是我们可以按照$|i-j|$从小到大的顺序来计算$f(j,k)$，进行 DP。而且当$i+1=j$或者$j+1=i$的情况，没有符合条件的k的存在，所以可以作为边界条件。

2) 当然我们还可以看到另外一个隐含条件$y_k<y_j$，否则有$f(j,k)=0$，于是可以忽略。也就是说，我们也可以按照y_j从小到大的顺序来计算$f(j,k)$，来进行 DP。同样地我们注意到当$y_j=1$时，所有的$f(j,k)=0$，所以可以作为边界条件。

3) 我们还有条件$y_i>y_j$。所以我们可以按照y_i从小到大的顺序来计算$f(i,j)$。同样地当$y_i=1$的时候所有的$f(i,j)=0$，可以作为边界条件。

最终答案的计算

如果我们得到了所有的$f(i,j)$，最终的答案也很容易得出。我们设最终点列长度为M。当$M=1$时，容易知道点列的数目为N。当$M\geq 2$时，我们枚举点列的前两个点，设他们分别为点i和点j，此时的符合条件点列数目就是$f(i,j)$。

于是最终答案就是：

$$N+\sum_{i=1}^{N}\sum_{j=1}^{N}f(i,j)$$

这个式子同样也是只包含加法运算，所以也可以在计算过程中对Q取模而不会影响最终答案。

时间复杂度分析

因为一共有$O(N^2)$个状态，计算每种状态的时候我们都要枚举k，于是计算$f(i,j)$需要$O(N^3)$的时间。

至于对最终答案的计算，需要把$O(N^2)$种状态加起来。这一部份的时间复杂度为$O(N^2)$。

因为前面还要对点的x坐标和y进行预处理，对点进行重新排序，这一部份的时间复杂度为$O(N\log N)$。

所以，最终的时间复杂度就是$O(N^3)$。无法通过所有数据。

参考程序

该程序按照y_i从小到大的顺序进行 DP。

../Programs/tank1.cpp

数据结构优化

分析

我们观察一下状态转移方程式：

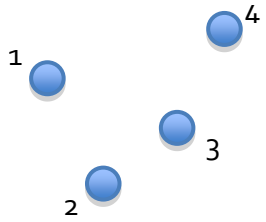
- 若$i<j$，则有$f(i,j)=1+\sum_{k=i+1}^{j-1}f(j,k)$。

- 若 $j<i$，则有 $f(i,j)=1+\sum_{k=j+1}^{i-1} f(j,k)$。

注意到，我们每次转移需要知道 $\sum_{k=i+1}^{j-1} f(j,k)$ 或者 $\sum_{k=j+1}^{i-1} f(j,k)$ 的值，也就是要支持对形如 $\sum_{j'<j<j''} f(i,j)$ 的求和式，也就是 j 在某一段区间取值内 $f(i,j)$ 的和的询问操作。

一个例子

比如下面对下面这个数据，四个点的坐标分别为 $(1,3),\ (2,1),\ (3,2),\ (4,4)$：



下面是当 i 和 j 取不同的取值时 $f(i,j)$ 的值：

	j = 1	j = 2	j = 3	j = 4
i = 1	0	1	2	0
i = 2	0	0	0	0
i = 3	0	1	0	0
i = 4	4	1	1	0

我们要支持询问形如 $\sum_{j'<j<j''} f(i,j)$ 的求和式，也就是要对表格中某一横行的几个连续的格子求和。比如上面的例子，要求 $f(1,3)$ 需要对青色的格子求和，求 $f(4,2)$ 需要对绿色的格子求和，求 $f(4,1)$ 需要对黄色的格子求和。

使用数据结构

所以，我们可以对 $f(i,j)$ 中的每个 i，也就是上面的表的每一个横行建立一棵线段树。这样子就可以在 $O(\log N)$ 的时间结界内支持对形如 $\sum_{j'<j<j''} f(i,j)$ 的求和式的询问。

我们可以按照之前介绍的三种动态规划顺序的任何一种来计算 $f(i,j)$，计算的时候利用线段树进行区间求和。计算后就插入到线段树之中。

因为一共有 $O(N^2)$ 个 $f(i,j)$ 需要进行求和与插入操作，每次求和与插入操作都需要 $O(\log N)$ 的时间，所以最后的时间复杂度就是 $O(N^2 \log N)$。比原来的算法优了不少。

当然我们也可以用树状数组来代替线段树，这样子每次查询和修改都要 $O(\log N)$，总时间复杂度也是 $O(N^2 \log N)$。但是实际运行效果要比线段树好很多。

如果你时间比较充裕，也可以用各种平衡二叉查找树，也是支持 $O(\log N)$ 时间的查询和插入操作。还可以块状链表，支持 $O(\sqrt{N})$ 的查询操作和 $O(\sqrt{N})$ 或者 $O(1)$ 的修改操作。在时间复杂度上面都是要比直接计算要好一点。

各种数据结构的原理，功能与具体实现方法请参考相关的书籍文章，这里不做赘述。

参考程序

下面的程序都是按照 y_i 的大小顺序计算 $f(i,j)$ 的。

线段树版本：

../Programs/tank5.cpp

树状数组版本：

../Programs/tank6.cpp

标准解法

分析

假设有i,j满足$i<j$且$y_i>y_j$，那么我们有：

$$f(i,j)=1+\sum_{k=i+1}^{j-1}f(j,k)$$

假设另有i'满足$i'<i<j$且$y_{i'}>y_j$，那么有：

$$f(i',j)=1+\sum_{k=i'+1}^{j-1}f(j,k)$$

下式减去上式得：

$$f(i',j)-f(i,j)=\sum_{k=i'+1}^{i}f(j,k)$$

也就是说。两式抵消掉了很大一部分，只剩下$\sum_{k=i'+1}^{i}f(j,k)$。

对于某个确定的j，找出所有的i，使之满足$i<j$并且$y_i>y_j$。再把它们从大到小排序，于是得到$i_1>i_2>i_3>\mathrm{K}$

于是可知：

$$f(i_1,j)=1+\sum_{k=i_1+1}^{j-1}f(j,k)$$

$$f(i_2,j)=f(i_1,j)+\sum_{k=i_2+1}^{i_1}f(j,k)$$

$$f(i_3,j)=f(i_2,j)+\sum_{k=i_3+1}^{i_2}f(j,k)$$

$$\vdots$$

同理，我们可以用类似的方法，对某个确定的j，算出所有满足$i>j$和$y_i>y_j$的$f(i,j)$。

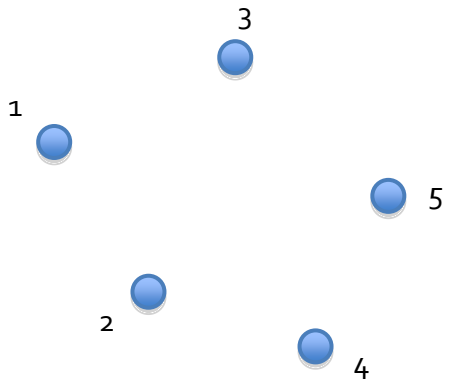
这样子我们对于某个确定的j，就可以算出所有满足$y_i>y_j$的$f(i,j)$了，因为每个$f(j,k)$最多只被计算一次，所以时间复杂度为$O(N)$。而当$y_i\le y_j$时已有$f(i,j)=0$。

也就是说，对每个确定的j，可以在$O(N)$的时间内算出所有的$f(i,j)$。而j的取值为1到N，也就是说，我们可以用$O(N^2)$的时间算出每个$f(i,j)$。

我们已经知道预处理和排序需要$O(N\log N)$，从每个$f(i,j)$计算出最终答案需要$O(N^2)$，所以总的时间复杂度是$O(N^2)$，可以通过全部数据。

一个例子

下面是一个例子，五个点的坐标分别是$(1,4),\ (2,2),\ (3,5),\ (4,1),\ (5,3)$：



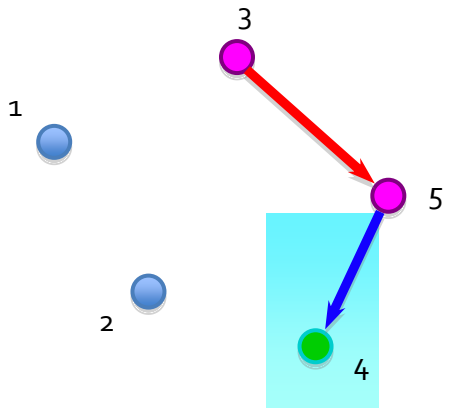
我们按照y_j从小到大的顺序来计算$f(i,j)$。假设我们已经得到所有$y_j\le 2$，也就是$j=4$和$j=2$时所有的$f(i,j)$的值：

	j = 1	j = 2	j = 3	j = 4	j = 5
i = 1		1		1	
i = 2		0		1	
i = 3		1		1	

i = 4		0		0	
i = 5		2		1	

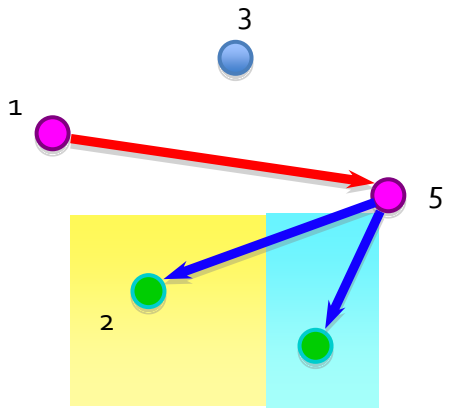
这时候我们要计算当$j=5$时，所有的$f(i,j)$的取值。

计算 f(3,5)



已知有$f(3,5)=1+\sum_{k=4}^{4}f(5,k)$，其中$\sum_{k=4}^{4}f(5,k)$就是上页表中青色格子的总和。在进行状态转移的时候，我们要枚举点列中，在点3和点5之后的点是哪一个点。下一个点只能在上图青色阴影部份中寻找。所以$\sum_{k=4}^{4}f(5,k)$也就是我们在青色区域内枚举点列的下一个点，再把答案加起来。

计算 f(3,5)



已知 $f(1,5)=1+\sum_{k=2}^{4}f(5,k)$。其中 $\sum_{k=2}^{4}f(5,k)$ 也就是 14 页的表中黄色和青色格子的总和。其中青色格子在计算 $f(3,5)$ 的时候已经算进去了，只要多计算黄色格子的总和就可以了，黄色格子也就是 $\sum_{k=2}^{3}f(5,k)$，所以 $f(1,5)=f(3,5)+\sum_{k=2}^{3}f(5,k)$。注意这里虽然 $f(5,3)$ 还没有被计算出来，这是因为我们是按照 y_j 从小到大的顺序来计算 $f(i,j)$ 的，所以一定有 $y_5<y_3$，所以 $f(5,3)=0$，可以忽略。

当然， $f(1,5)=f(3,5)+\sum_{k=2}^{3}f(5,k)$ 也可以看作是要求 $f(1,5)$ 的时候，要在上页的图中下一个节点，除了要在青色区域寻找外，还要在黄色区域寻找。所以要比 $f(3,5)$ 多加上一个 $\sum_{k=2}^{3}f(5,k)$。

得到答案

因为有 $y_i \le y_j$ 时 $f(i,j)=0$，所以 $f(2,5)=f(4,5)=f(5,5)=0$。于是就可以在 $O(N)$ 时间算出 $j=5$ 时所有的 $f(i,j)$。

按照这种方法，可以在 $O(N^2)$ 的时间复杂度内算出所有的 $f(i,j)$：

	j = 1	j = 2	j = 3	j = 4	j = 5
i = 1	0	1	0	1	4
i = 2	0	0	0	1	0
i = 3	2	1	0	1	2
i = 4	0	0	0	0	0

i = 5	0	2	0	1	0

所以，最终答案（未对Q取模），也就是符合题目要求的两个条件的非空点列的总数就是21。

参考程序

C++语言版本（因为使用了 vector 容器，因此速度比较慢）：

../Programs/tank2.cpp

C 语言版本：

../Programs/tank3.c

Pascal 语言版本：

../Programs/tank4.pas

各算法运行时间比较

以下是各算法在不同数据规模下的运行时间。其中"-"表示运行时间超过 60 s。

N	搜索	$O(N^3)$ 动态规划	$O(N^2 \log N)$ 线段树	$O(N^2 \log N)$ 树状数组	$O(N \cdot 2^2)$ 标准算法
15	0.000 s	0.000 s	0.000 s	0.000 s	0.000 s
30	0.003 s	0.001 s	0.001 s	0.001 s	0.000 s
60	0.112 s	0.002 s	0.001 s	0.001 s	0.001 s
120	58.678 s	0.012 s	0.005 s	0.002 s	0.001 s
250	-	0.103 s	0.023 s	0.010 s	0.002 s
500	-	0.817 s	0.111 s	0.041 s	0.008 s
1000	-	6.589 s	0.567 s	0.222 s	0.040 s
2000	-	54.166 s	2.951 s	1.320 s	0.176 s
3000	-	-	7.833 s	3.414 s	0.409 s

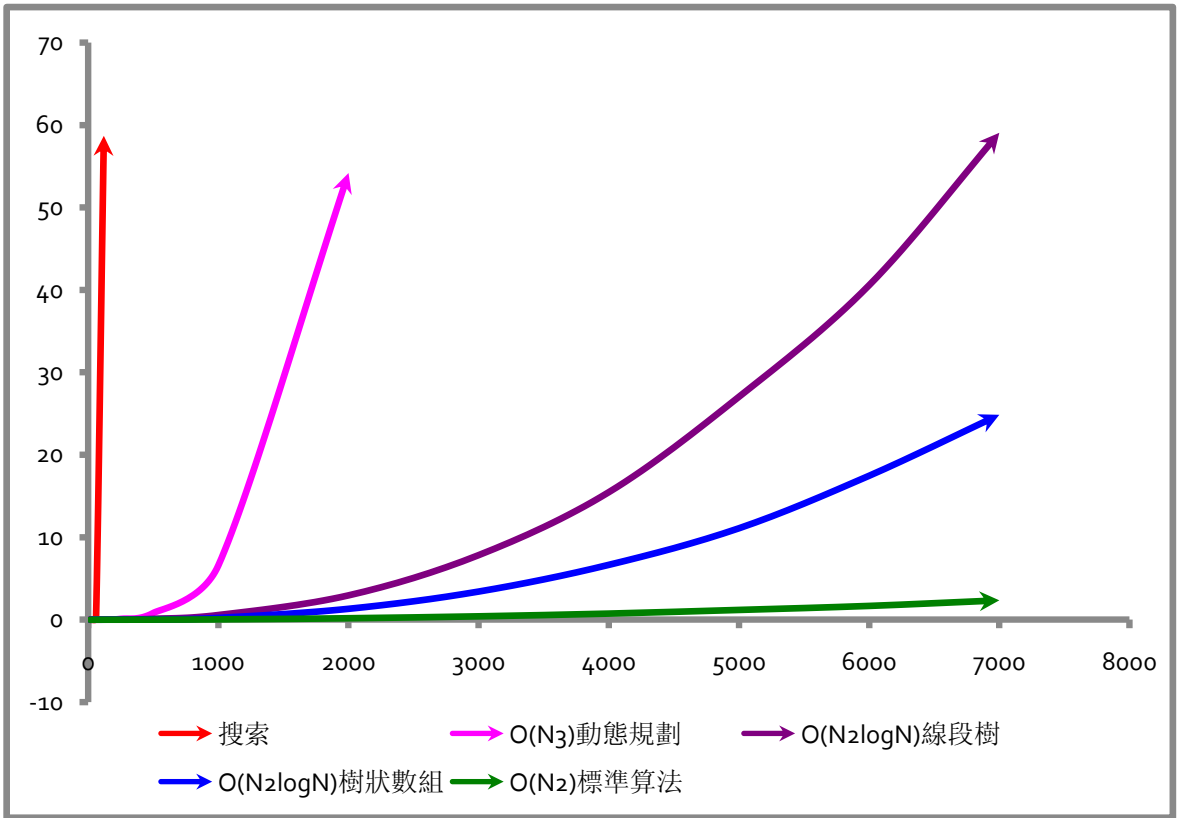
4000	-	-	15.469 s	6.654 s	0.742 s
5000	-	-	27.038 s	11.090 s	1.184 s
6000	-	-	40.567 s	17.458 s	1.672 s
7000	-	-	59.046 s	24.852 s	2.349 s

以上数据全部为随机数据。

各算法的测试程序均为前文所提供的 C++语言的参考程序。测试机器 CPU: 2.66 GHz Intel Core 2 Duo; 内存: 4 GB 1067 MHz DDR3。编译程序版本: g++ 4.2.1。

更加详细的搜索算法的运行时间的测试请参见本文第 5 页。

下图是各种算法的运行时间与数据规模之间的关系。



更优的算法？

本人到目前为止无法想到更优的算法。

如果按照 $f(i,j)$ 这种设计状态的方法，那么一共有 $O(N^2)$ 个状态。如果把所有的状态都计算出来，并在最后累加。那么就至少要花费 $O(N^2)$ 的时间。因此，如果有更优的，也就是时间复杂度低于 $O(N^2)$ 的算法，则必须抛弃这种状态设计的方法，或者想办法，使得即使不用一一计算出所有的状态也能得到最终答案。

总结

这是一道比较有趣的动态规划题目。首先题目背景比较有趣，而且数学模型比较简单。一题多解，这道题可以使用搜索，数据结构与动态规划，能得到不同的分数。而且动态规划的状态关系不是很清晰，有多种动态规划的顺序。必须选择正确的动态规划的顺序，找到不同状态间的关系，才能得到标准解法。

这道题总体难度比较简单。但是作为 NOI 难度也是可以的。毕竟还有更简单的 NOI 题目（比如“水喝完了怎么办？”）。

感谢

感谢珠海一中的潭瑞阳同学，他提供了一道题目，题目描述与类似而解法却迥然不同。感谢潮阳实验学校的郭晓旭同学，她参与题目讨论，并提供了一个错误的 $O(N\log N)$ 算法。他们都括宽了我的思路，增进了我对题目的了解。

感谢国家集训队胡伟栋教练和唐文斌教练，感谢他们的指导与鼓励。

感谢汕头金山中学信息学竞赛班的指导老师姚肖华老师，感谢她对我们行动上与精神上的支持。