DAY1

Task 1 : FROG

SOLUTION
A naïve $O(N^3)$ time algorithm for the problem iterates through all $O(N^2)$ line segments induced by the point set *S*, and determines how far each segment spacing can be extended to either direction within the point set (*O(# landings) = O(N)*).

An efficient $O(N^2)$ time algorithm for the problem is based on an algorithm for finding an *equally-spaced collinear subset of a set.* The algorithm works by "overlapping" all equally spaced triples in order to determine all maximal equally-spaced collinear subsets. The "overlapping" is performed by constructing an undirected graph where for each equally-spaced triple (p_A, p_B, p_C) we create nodes <*A,B*> and <*B,C*> and the edge (<*A,B*>, <*B,C*>); connected components in this graph correspond to maximal equally-spaced collinear subsets in the original set. Observe that a frog path is simply a linear chain of connected nodes (with at least one edge and two nodes, meaning at least 3 flattened plants) in this graph. Each node in this graph has degree at most two, so the edge set and vertex set both have size $O(N^2)$. Hence we can find all maximal equally-space collinear subsets in $O(N^2)$ time from the graph.

The only detail here is how to efficiently find the equally-spaced triples from which the graph is created. The obvious method of iterating over all triples of flattened plants would worsen the complexity to $O(N^3)$. If instead the field is stored as a two-dimensional array (every plant has an entry) giving the identity of the landing on that plant (e.g., if the 100th flattened plant were at (10,12), then the array value at (10,12) is 100), you can loop over pairs of flattened plants p_A and p_B, and then look up p_C from the array in constant time since you know what the location of p_C must be if it exists. This strategy takes $O(N^2)$ time but uses O(*field size*) memory – in particular, it needs 5000*5000 entries of a short integer each, or 50MB. Because the above graph also needs $O(N^2)$ space to store it, this strategy unfortunately would exceed the memory limit of 64MB. However, as this array is very sparse, it may be stored in memory as a hash table, which in the expected case does not affect the time complexity (but which in the worst case does). The third and best option is to construct the graph in linear time and constant memory by sorting the locations (e.g., row major, column minor) and keeping 3 pointers into the list (*A*, *B*, and *C* for pointing to p_A, p_B, and p_C, $A<B<C$) as follows; loop *A* over all values, and for each *A* march *B* and *C* down the list, moving either *B* or *C* forward at each step so as to try to maintain as close to equal spacing as possible; when exactly equal spacing is found, enter the nodes and edge into the graph.

There is also an $O(N^2)$ dynamic programming algorithm to solve this problem, which is plagued by the same memory problems as illustrated above. In addition to storing the identity matrix described above, store another $O(N^2)$ matrix containing whose rows are indexed by p_A; along the row are *N* entries, one per plant p_B giving the number of landings in a candidate frog path which goes through p_A and p_A but which only uses points which sort before p_A in the ordered list (i.e., pretend the field ends at p_A, and look for frog paths of any length in that smaller field – the idea is to find partial frog paths which violate none of the frog path conditions in the region of the field already examined). Assuming the table is filled up to row *A*, row *A+1* is filled by considering all $O(N)$ flattened plants *B* before p_A,

and if there is a flattened plant *C* such that *A*, *B*, and *C* are equally spaced, look up in the array the number of landings in the candidate frog path through *B* from *C*, increment by 1, and store as the *B*th entry in the row for *A*. If *C* would be outside the field, then enter it as having 2 flattenings. At the same time check to see if the next flattened plant (*D*) would be outside the graph, and if so, you have a completed frog path. To efficiently determine *C*, the same 50MB array as above is needed; a hash table can again be used, with no increase in average-case time complexity, but an increase in worst-case time complexity.

TESTING

The test data contains 25 test cases. Most of data are initially generated by random function, then they are modified by manual work.

Each test case has size *N* (the number of points) in the range between 10 and 5000. Among 25 test cases, 10 test cases have size $N \leq 1000$. The remaining 15 test cases have size $N \geq 2000$. The detail on the test data is summarized in the following table.

Each test case is worth 4 points. A program which implements a cubic time algorithm can solve the test cases within time limit such that their size $N \leq 1000$. An implementation of this algorithm should be able to get the first 10 test cases correct but will likely run out of time on all other cases (scoring 40% of the points).

Testing Data Description FROG

No	points, (r*c)	Description	Solution
1	18, (6 * 7)	Sample data in task	4
2	10, (10 * 10)	Manually designed	5
3	25, (50 * 50)	Manually designed	13
4	50, (10 * 10)	Several Lines + random points	10
5	100, (20 * 20)	modified random point set	10
6	300, (30 * 30)	modified random point set	15
7	500, (55 * 55)	Several Lines + random points	28
8	500, (100 * 100)	Special case for no solution	0
9	1000, (100 * 100)	Several Lines + random points	34
10	1000, (1000 * 1000)	Several Lines + random points	250
11	2000, (50 * 50)	Random (uniform) points	25
12	2000, (100 * 200)	Several Lines + random points	33
13	2000, (1000 * 2000)	Several Lines + random points	333
14	3000, (60 * 60)	Uniformly random points	31
15	3000, (500 * 500)	X shapes and random points	500
16	3000, (5000 * 1)	Horizontal line	20
17	3000, (5 * 1000)	Several Lines + random points	17
18	4000, (100 * 100)	Random points (uniformly)	34
19	4000, (200 * 20)	Very dense points set	200
20	4000, (1000 * 1000)	Several Lines + random points	500
21	4000, (5000 * 5000)	Several Lines + random points	311
22	5000, (100 * 100)	Chess board style	100

23	5000, (1000 * 1000)	Several Lines + random points	334
24	5000, (3000 * 3000)	Irregular linear points	1000
25	5000, (5000 * 5000)	Modified random points	72

BACKGROUND

The problem "The Troublesome Frog" is related to the problem for detecting spatial regularity in images. Spatial regularity detection is an important problem in a number of domains such as computer vision, scene analysis, and landmine detection from infrared terrain images. The AMESCS(All Maximum Equally-Spaced Collinear Subset) problem is defined as follows. Given a set P of n points in E^d, find all maximal equally-spaced, collinear subset of points. Kahng and Robins[1] present an optimal quadratic time algorithm for solving the AMESCS problem.

Reference

[1] A B. Kahng and G. Robins, **Optimal algorithms extracting spatial regularity in images**, Pattern Recognition Letters, 12, 757-764, 1991.