Pacific Northwest ACM ICPC Regional Programming Contest Writeup

A Airports

Make a graph with a vertex for each flight, and draw a directed edge from flight i to flight j if it is possible for the plane to service flight i then flight j. Notice that this graph is a DAG, and we want to find the minimum number of paths needed to cover all the vertices.

To solve this problem, make a bipartite graph with two copies of each flight, call the u_1 and u_2. If there was an edge between u and v in the DAG, draw the edge u_1 to v_2. Now, find the max bipartite matching in this graph, and the answer is m - |max matching|.

It is not too hard to see how to convert between any matching in this bipartite graph to a path cover and vice versa. (i.e. if u_1 to v_2 is included in the matching, then we do flight v directly after flight u). Note that unmatched vertices on the left hand of the bipartite graph corresponds to the "end" of a path. Thus, we want to minimize the number of unmatched vertices, which corresponds to maximizing the size of the matching.

B Butterfly Effect

This problem required a bitmask dynamic-programming approach.

Let f(mask,pos,left) denote the maximum probability we can get given that we're currently at event number "pos", with "left" interventions, and "mask" describes the state of positive/negative events that happened before us.

When we intervene, we either want to take the maximum of the dice rolls or take the minimum of the dice rolls.

The probability of getting a max of k on an m sided dice can be computed as $(k^2-(k-1)^2) / m^2$. Consider the following grid:

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

To expand our dynamic programming state, we either choose to intervene or not

intervene, and take the max of both case.

The rest of it is mainly implemenattion details, so refer to the code for more details on the actual recurrence.

C Classy

This problem was a simple sorting problem. It could be solved either by writing a special comparison function following the rules in the problem statement, and calling a sort routine with that comparison function, or by calculating a unique key value for each equivalent class that sorted the same as the classes themselves. The first approach was usually simpler.

D Triangle

This problem was apparently very tricky, largely because of assumptions that were made about the order of the sides. The problem made no such guarantees. The intended solution was to sort the given side lengths for each triangle and verify that they were identical, and then verify that the side lengths make up a right triangle. With sorted side lengths, this was easy, since the hypotenuse needs to be the longest side.

E Excellence

Excellence required a bit of insight but the implementation was straightforward. The insight was that, to solve this problem, always pair the strongest and weakest members, remove them from the pool, and repeat until all teams were formed. This is equivalent to sorting the members by strength, numbering them from 1 to n, and choosing teams by pairs that sum to n+1.

How do we know this works? We can show it by induction. The base case has only two members, (A,B); pairing them is the only possible arrangement and is thus optimal.

Now consider a larger group of members. Our algorithm pairs the best and the worst. If we do not pair the best and the worst, assume we pair the best with someone other than the worst and that this leads to a better overall strength. Without loss of generality let us assume every programmer has a distinct strength value. By pairing the best with someone other than the worst, we've improved the value of the team we just formed---but we've strictly weakened the remaining group (that is, the sorted ordering by strength has members that are weaker or equivalent for every position). There is no way to get a better overall solution from a group that is strictly weaker.

F Falling Blocks

First, note that since the top three rows must be empty before we make a move, we only care about the bottom seven rows.

There are a total of 2^{21} states for the actual board, and |s| states for the next piece, so a total of 2^{21} * |s| states, which is small enough to do a linear time solution.

To solve this, we use depth-first search. The nodes in our graph are pairs of (board state, index). We track for each node whether it is fully explored (and thus has a maximum depth) or whether it is still on the dfs stack (and not fully explored); this is Tarjan's algorithm for determining if a graph has a cycle.

If we ever revisit a state during the depth-first search that is still unexplored (that is, still on the stack), we have a cycle, and can conclude that the answer is infinity. Otherwise, let f[i] denote the max time we can play given we're at state i. Then, f[i] = max(f[child]) + 1, where child is a state we can get from i.

The rest is some tricky bit maniuplation. See the code for more details on specific implementation details. Some solutions also represent the board state slightly differently, so you can look through the other solutions for a variety of ways to approach this.

G Racing Gems

Suppose r = 1 (that is the vertical/horizontal velocities are the same).

Then, we can rotate the board by 45 degrees, the path we follow can only go up or right. This can be solved by doing a longest increasing subsequence.

More specifically, given a point (x,y), we turn it into the point (x+y, y-x). If we sort by x+y, the longest increasing subsequence of y-x will give us the max number of gems we can hit.

Now, if r is not one, we can scale the x coordinates by r instead first. Thus, we instead convert the points to (x*r*+*y*, *y*-*x*r).

It was also possible to sort the points in a direction perpendicular to one of the race direction bounds and do a geometric sweep directly; this is equivalent to the approach given above.

H Hilbert Sort

Sort the points into four quadrants as stated: points in the lower-left quadrant

are visited first, followed by points in the upper-left, then upper-right, and finally lower-right. If some points belong to the same quadrant, they are recursively sorted by splitting it into even smaller quadrants. The tricky part is that the first and fourth quadrants are rotated and flipped, so this transformation must be applied at each recursive step.

This problem had a number of distinct ways to solve it. One way was to recurse as described in the statement in the order of the curve, calculating which points were in each quadrant and remapping them according to the rotations and flips described in the problem statement; once a particular level had 1 or fewer points you could emit that point and stop the recursion. Another way was to transform the two-dimensional coordinate into a "distance" along the curve by examining one bit of both coordinates at a time, remapping as needed, and sorting based on this coordinate. Either technique worked fine.

You could turn the coordinates into floating point values and do all arithmetic as floating point, or you could use fixed-point math and pull the relevant bits off by doubling each coordinate and comparing the result against S to determine the quadrants.

Getting the rotations and flips correct was probably the most difficult part of this problem.

I Coverage

Without the one additional tower, this problem would be the same as finding the largest connected component in a graph where two towers are neighbors iff they are at most 2km away from each other. The additional tower enables us to merge a set of connected components; the problem is therefore to identify the biggest merged component we can create. As a preprocessing step, we compute all existing connected components in $O(n^2)$ time.

To identify which sets of components we can merge, let's make the following observations:

1. Placing a new tower and seeing which towers its coverage overlaps (i.e., placing a 1km disc and seeing which existing 1km discs it overlaps) is equivalent to placing a 2km disc and seeing which existing towers are inside it.
2. If it is possible to place a 2km disc that contains a set T of existing towers (where |T| >= 2), it is possible to place a 2km disc that contains T AND has at least 2 towers in T on its boundary.

Therefore we can enumerate all possible merging of components by taking all pairs of existing towers, and for each pair, finding the discs of radius 2km that have both points of that pair on their boundary. Working out the geometry shows that for each pair there are 0, 1, or 2 such discs, which we will call

candidate discs. For each candidate disc, we simply iterate over all the towers to see which ones are inside this disc (and therefore whose components are included in the merged component). This gives us O(n) work per candidate disc, of which there are $O(n^2)$, for a total of $O(n^3)$ work, which isn't quite fast enough.

Or is it? Recall that in the input, it is guaranteed that no two towers will be closer than 1km to each other. Because the towers are in the plane, this means that for every r, there is a maximum count of towers C(r) that can be contained within a circle of radius r. That means if we pick a tower and draw a circle of radius 4km around it, the number of towers that can be inside that circle is bounded above by a constant. Every tower that's not in that circle is too far away from our central tower to generate a candidate disc. That means that the total number of pairs of towers that are capable of generating 1 or 2 candidate discs is actually O(n), not just $O(n^2)$, and so the algorithm we described above actually only does a total of O(n^2) work.

Edge case: The logic above doesn't consider singleton sets of existing towers. In the case that no candidate discs are created in the above process, we simply add a single tower to the largest existing connected component. (The edge case in particular that breaks without this addition is the case where there is only 1 tower in the input.)

Trivia: This problem can be solved in linear time by using a data structure for point location, though the constant factors are awful enough that for the bounds given in the problem statement, the linear time solution is likely to be slower than a well-written quadratic time solution.

J Olympics

Suppose the weights are discrete for a moment, and let's instead ask the question: what's the maximum range for our strength S such that we can determine S exactly?

Let f[E] denote the maximum K such that we can exactly determine S from the set [1, K] given we start with E energy.

Then, the recurrence is f[E] = f[E - Esucc] + f[E - Efail] + 1. (with f[x] = 0 when x <= 0).

The strategy is as follows: first we try to lift f[E - Efail] + 1. If we succeed, we can search an interval of size f[E - Esucc] above us. Otherwise, we can search an interval of size f[E - Efail] below us.

Now back to the original problem with continuous real-valued weights. Without the 25 kg lower bound, we know we can get at least 225/(f[E]+1) of our actual strength. With the 25 kg lower bound, we can get at least 200/f[E] of our actual strength. Thus, we can take the min of these two cases.

Another way to see the solution is to consider the decision tree of all possible lifts you might attempt, depending on the outcomes of earlier lifts. Let f[E] be the total number of these "possible lifts". After performing one lift, the tree splits into a possible world containing f[E-succ] additional lifts, and an alternate world containing f[E-fail] additional lifts. Thus, the total number of lifts is once again given by f[E] = 1 + f[E - Esucc] + f[E - Efail].

If these lifts are spread evenly along the interval from exactly 25 kg to just below 225 kg, your precision will be 200/f[E]. If the lifts are spread evenly from just above 0 kg to just below 225 kg, your precision will be 225/(f[E]+1).

Note that f[E] can overflow (since it grows exponentially), but once f[E] surpasses 400 million, the answer is going to be smaller than 5×10^{-7}, so you can safely print 0.000000

The simplest way to evaluate the dynamic programming was to compute |f[E]| for increasing E. Another approach was to compute the number of distinct success/failure combinations possible for a given energy using binomials as shown in the Python solution.

K Checkers

Checkers is a graph problem. The apparently small board size (26 x 26) masked the existence of test cases that timed out simplistic search strategies.

The key to solving checkers is to realize that, when jumping, black checkers always move in steps of two in each direction. Thus, a black checker starting at (x,y) can only ever reach squares (x+4i,y+4j) (for all integral i and j) and (x+2+4i,y+2+4j); these are only one fourth of the black squares. The only white checkers it can jump are those at (x+1+2i,y+1+2j), so all such white checkers must be of this form.

So the first check is to ensure that all white checkers are on the same parity of black squares (that is, all white checkers must be on squares of (a+2i,b+2j) for fixed a and b) and to only consider black checkers on the other parity (a+1+2i,b+1+2j). Black checkers on the same parity as the white checkers can be ignored.

The black checkers of the other parity fall into two groups according to the value of ((x+y)%4). Each group should be handled separately. Each group consists of the squares that can be mutually reached by jump-style moves.

For any given white checker, there are at most two adjacent black squares that a given black checker can ever reach by jumping moves. Thus, we can treat white checkers as graph edges, and the reachable black squares that are directly adjacent to some white checker as the nodes.

With this graph representation, asking if all the white checkers can be jumped is the same as asking for an Euler path through the white checkers starting at a given black checker. This can be done by ensuring that the graph is connected, and that at most two of the nodes have odd degree, and if any nodes have odd degree one of them must be the node where the capturing black checker lies. In addition, there must be exactly one black checker on all of the nodes of the graph, otherwise that second black checker will interfere with some portion of the capture sequence.

Testing connectivity of a graph is a simple depth-first or breadth-first search. Calculating node degree is a simple linear-time walk of each node.

The overall runtime should be linear in the size of the input graph.

L Millionaire

One should start by converting all the dollar values to happiness units and working with those; converting the final answer back to dollars is straightforward algebra.

To play Millionaire optimally, you want to determine how far you can go before you should take your winnings so far and quit, refusing to answer any further questions. The key observation is, once you've made it to a particular question, there's no additional history you need to worry about. So you can solve it backwards: assuming you make it to the final question, should you try to answer it or quit? If you quit, you receive the penultimate prize. If you try, the possible outcomes are success and failure, and your expected happiness is a weighted average of the two. Having determined whether to try or quit at the final question, now work backwards to the penultimate question and so on all the way back to the first question.

While a recursive solution risks overflowing the program stack, the same idea can be implemented by scanning over the questions backwards.

M Magic Trick

Magic Trick was intended to be fairly easy, but many people had difficulty because they accumulated more than one error for a single given number, rather than calculating the number of input values that lead to errors. The overall solution was just to iterate over the values from 1 to 100, and follow the steps. If any error occurred at any step, that number was considered a "mess up" and no further processing on that particular number was necessary.

N Egg Drop

Egg Drop was a lot simpler than many coders made it out to be. The intent was just to keep track of the lowest floor the egg could definitely be dropped from safely, and the highest floor the egg could definitely not be dropped from safely. At the end, the output was just one floor higher than the lowest floor the egg could be definitely dropped from safely, and one floor lower than the highest floor the egg would definitely break.

O Grid

Grid was a standard breadth-first search; there were no complicating factors. The most common mistake was getting confused between width and height at various points of the code.

P Complexity

Complexity was originally entitled Simplicity, and it was problematic for many coders. The intended solution was just to identify the two most common characters, and delete all the rest. This could be done by accumulating character counts for every possible character, sorting that array by frequency, and returning the length of the input string minus the two highest frequencies from this array.

Q Excellence

(See E Excellence above.)

R Class Time

Class Time was a problem that requiring sorting a set of names by last name and then by first name. There were to approaches. The first was to write a custom comparison function that found and compared the last names, and if equal, compared just the first names. The second was to use a Schwartzian tranformation by reversing the first and last names, sorting, and then reversing them again on output.

S Surf

To solve surf, execute dynamic-programming-based optimization on the topologically sorted graph of waves.

We sort the waves by start time; you can never catch an earlier wave after a later wave. Then, we scan the set of waves, for every wave considering every other potential predecessor wave and choosing the one that permits us to catch

the current wave with a maximum value of happiness. Then we scan all the waves and find the one with the best total happiness; that's our optimal result.

Unfortunately this won't run in time with 300,000 waves because it considered every possible predecessor wave. We can turn this into an O(n log on) solution by building a list sorted by the start times and end times of each wave. We scan this list, accumulating the greatest happiness we can achieve at each point in time. Whenever we encounter a wave start, we store the current happiness value in that wave. Whenever we encounter a wave end, we add the happiness stored in that wave to the happiness we get from taking that wave; if the sum is greater than our current happiness, we adopt the new happiness as our current happiness. This solution runs very fast even in Python.

T Triangle

(See D Triangle above.)

U Blur

Following the basic instructions in Blur would not in general give you a working solution because floating point arithmetic has inaccuracies that cause computed values that are algebraically equal to be represented by slightly different values.

The easiest solution was to avoid floating point altogether; instead of calculating the average, just calculate the sum, using integers which are exact. The sum is just the average multiplied by the count of elements, so uniqueness is preserved. Since the maximum number of blurs was 9, and 9^9 is less than 2^{31}, normal integers were sufficient and 64-bit longs were not needed.

V Gears

To solve Gears, you had to determine if the input gear was connected to the output gear, what the parity of the path length was, and if the graph of connected gears had any odd-length cycles (in which case all gears would be locked in place). Once all these conditions were checked, the final speed ratio is simply the size of the input gear divided into the size of the output gear (and you needed Euclid's algorithm to factor out any common divisors).

To check for odd cycles, simply create a two-coloring of the graph. Mark all the nodes initially unknown; mark the first as clockwise. As you encounter each node, mark it the opposite of the adjacent node unless the new node is already marked; if it is marked inconsistently, you've found an odd cycle.

Of course unconnected portions of the graph can have odd cycles; only the

connected component that includes the initial gear would prevent the initial gear from turning.