Florida High School Programming Series
2019 Playoff Round

Problem Review by Arup Guha (Problem Setter)

Problem A: ab Knight

The total number of squares on the board is no more than 10,000 and the total number of moves that can be made from any one square is 8. We can view each square on the board as a vertex in a graph and each possible jump as an edge connecting two vertices. Thus, the graph that is induced by this conditions has at most 10,000 vertices and at most 80,000 edges. A breadth first search (BFS)can be used to find the shortest distance (in number of edges) in a graph with unweighted edges, where shortest distance is defined as the number of edges to travel from one vertex to another. A single BFS can find the shortest distance from any one vertex to all of the others. This is what is being requested in the problem. The run-time of a BFS on a graph with V vertices and E edges is O(V+E). Thus, for this problem, this solution is sufficient.

In terms of implementation, it helps to build a DX/DY array using the a and b given. One can either hard-code this or run a set of loops that go through the eight possible directions. In the judge's solution, there are three nested loops - one for the sign of dx, one for the order of a and b and one for the sign of dy. Once this array is filled, the BFS code looks very similar to any BFS on a grid.

Problem B: Energizer Knight

Just like problem A, we can treat the chessboard as a graph, where each square is a vertex and we place edges between pairs of vertices if a jump can be made between the corresponding squares. In this problem, the graph we create is a fixed graph with 64 vertices and a number of edges less than 500. (Note: my solution can be edited to find the exact number of these edges as I do create the graph in it.) The question is asking: how many paths are there between two vertices that use some specified number of edges exactly? Thus, we don't care about shortest distance, rather any path that gets you from point A to point B should count, but only the ones of a specified length.

It's clear that the adjacency matrix of the graph described above stores all the correct answers for paths of length 1. (The answer for this case is always 0 or 1, either there is a single jump between the squares or there isn't.) Now, consider trying to calculate the number of paths between two squares, square x and square y, of length 2. We must have one jump go from x to some middle vertex z, and then a second jump from vertex z to vertex y. We can simply try all possible values of vertex z in this equation:

$$\sum_{z=0}^{63} adj[x][z] * adj[z][y]$$

Basically, the first term in the sum is the number of paths from x to z of 1 jump and the second term is the number of paths from z to y of one jump. Adding over all intermediate vertices z, we get the total number of paths of length 2 between vertex x and vertex y.

More generally, let $M^k[x][y]$ be the number of paths between vertex x and vertex y of length k. Then, using the same logic, for any k = k1 + k2 it follows that:

$$M^k[x][y] = \sum_{z=0}^{63} M^{k1}[x][z] * M^{k2}[z][y]$$

Basically, we can split all paths of length k into the first k1 edges followed by the last k2 edges and sum over all possible locations of where a path could be after k1 edges.

It just so turns out that the definition shown above is exactly the definition for matrix multiplication. Thus, to solve this problem, we simply create the adjacency matrix described previously and raise it to the k power.

Unfortunately, using typical matrix exponentiation, this would take too long, since k can be as large as 10^{18}. (Using the usual algorithm, the run time would be $O(n^3k)$, where n is the number of rows and columns of the matrix and k is the exponent.)

Luckily, we can solve the matrix exponentiation problem faster using the Fast Matrix Exponentiation algorithm. The key observation of this algorithm is that if the power k that we are raising our matrix to is even, then, we can first recursively raise the matrix to the k/2 power and then take that answer and square it. For example, if k = 128, the recursive calls would all go to previous powers of 2 (64, 32, 16, 8, 4, 2 and 1, the base case) before squaring the previous matrix. Note that if our exponent is a perfect power of 2, we just do exactly $\log_2 k$ matrix multiplications to solve the problem. Instead of 128 matrix multiplications, this would result in 7 of them. If the exponent is odd, we do the usual breakdown of $M^k = M^{k-1}M$. Notice that if in one step, the exponent is odd, for the next step it'll be even. Thus, the total number of matrix multiplications is never more than $2\log_2 k$, where k is the exponent. Thus, using the Fast Matrix Multiplication algorithm, our run-time is $O(n^3 \log k)$. For this problem, the worst case has n = 64 and $k = 10^{18}$, for which $n^3 \log k \sim 16{,}000{,}000$, which is fast enough for the given time limit. The last piece of the puzzle is calculating the matrix mod 10007. To accomplish this, we must mod after every step. Since no negatives are being used, we won't run into any issues with negatives

here. (But in other problems, one must be very careful with negatives under mod. The standard fix is to check to see if a value is less than 0 after modding and then immediately add one copy of the mod value.)

Problem C: Harkness

This problem is a simulation with custom sorting. The most common bugs would most likely be updating one player's rating and using this new rating before calculating another player's new rating. What must be done is to store the win/loss record of each player as well as a list of each of the original ratings of their opponents. Then, use this information to calculate their new rating. When doing so, make sure to watch for precisions errors. Either using fractions or adding an epsilon to the final rating before truncating should provide the necessary accuracy. Once all of the scores are calculated, then a custom sort must be performed, sorting by rating, breaking ties by the input order (which can be coded as an ID number for each player, based on their appearance in the input).

Problem D: King

This problem is the banger in the set. A king can occupy a 3 x 3 space at most. If we want to minimize kings, we can simply start having them occupy 3 x 3 spaces, starting at the top left and going in row/column order. No simulation is necessary. Rather, we just calculate ceil(r/3)*ceil(c/3), where ceil represents the ceiling function. In code the answer looks like this:

```
int ans = ((r+2)/3)*((c+2)/3);
```

In general, in code, one safe way to accurately calculate ceil(n/k) is `(n+k-1)/k`.

Problem E: Move Notation

In scanning two copies of the board, there will only be two squares that have changed. Mark these two squares. Secondly, we must determine which square the move was from and which square the move was to. It's necessarily the case that the square the move was made from previously had a piece and subsequently did not. It's also clear that the square to which the move was made will have a piece afterwards. Thus, we can just look at the after board and see which of the two squares in question is vacant. This tells us the "from" square and the other square is the "to" square. The last piece of the solution is converting a row-col value (probably two integers in between 0 and 7, inclusive) into the desired output format. One can write a function or inline a formula for this conversion. (The columns are in order, and the rows are in reverse order. We convert from number to column as follows: `(char)(number+'A')`.)

Problem F: One Dimensional k-Rooks

For each rook, it "turns on" at a particular square and turns off an another square. We can call each of these column values "important". Note that there will be at most 200,000 important column values. At any other column value, no change in number of rooks attacking the square, will occur. Thus, we only care about the squares where these changes occur, knowing that in between consecutive important squares, the number of rooks attacking each of those squares is equal.

So, for each rook, we create two events: an on event at the minimum column value it can attack and an off event at the maximum column value it can attack, plus one, signifying the first square that can't be attacked by that rook. Then, we sort the events by column value, breaking ties by if the event is an on event or off event, with off events coming first. We processing an on event, we might create a new maximum number of rooks. We add the number of rooks that turn on at this column and see if this new value is higher than the most we've ever had. If so, we update our max and set our number of squares to 0. When we process an off event, we check to see if the number of rooks is maximal. If so, we add the last interval length (current column minus previous column) to the number of squares being attacked by the maximal number of rooks.

The bulk of the run-time is the O(nlgn) sort of the n events, since sweeping through the events takes O(n) time, for n events.

Problem G: Parallel Movement

We can split up all of the squares into two groups: ones with pieces and ones without. Then, we want to match as many squares from the first group with squares from the second group such that all matching pairs represent moves that the designated piece on the square in the first group can make. Since we know which squares in the first group can match with which squares in the second group, this problem is the Bipartite Matching Problem. One way to solve this problem is to run a Max Flow Algorithm (here are three: Ford-Fulkerson, Edmunds-Karp and Dinitz). We assign each square a vertex number in our graph. We place all vertices representing squares with pieces on the "left" of the graph and all vertcies representing squares without pieces on the "right" of the graph. (In code, there isn't a left or right, but you do have to know which vertex numbers correspond to which group.) Connect an edge with capacity one from the source to each of the left vertices. Connect an edge with capacity one from each of the right vertices to the sink. Then, for each left vertex, calculate all of the locations that piece could move to. For each location that is an unoccupied square, add an edge with capacity one between the left vertex and the corresponding right vertex. The maximum flow through this graph equals the answer to the problem.

Problem H: Peaceful Bishops

We can not place more than one bishop per diagonal. There are 2n-1 diagonals, where n is the number of rows and columns of the board. One way to identify "forward" diagonals in code is to label the rows 0 to n-1 from top to bottom and label the columns 0 to n-1 from left to right. Then, each possible sum row+column represents a forward diagonal. The first one has 1 square (0, 0), the second has 2 squares, n^{th} one has n squares, the $(n+1)^{th}$ one has n-1 squares, … and the $(2n-1)^{th}$ one has 1 square (n-1, n-1). It's impossible to place a bishop in every one of these diagonals because the very first diagonal (top left square only) and very last diagonal (bottom right square only) are on the same backwards diagonal. This means that the maximum number of bishops we can place can't exceed 2n-2. There's always a construction that achieves this as well: place a bishop on every square on the first row. Then you can place bishops in every square on the last row except for the first and last.

Now that we know the answer to the first half of the query, we can write a backtracking solution utilizing the column fact. We first generate all of the valid arrangements of bishops and then we sort them in lexicographical order. Finally we choose the appropriate arrangement.

When backtracking, what we want to do is try placing one bishop per forward diagonal. Our recursive function takes in the current placement of bishops and a value for the diagonal sum which represents the current diagonal we want to place a bishop. Then, we try placing a bishop in each of the possible slots, but only if it doesn't conflict with a previously placed bishop (by backwards diagonal). In addition, if we've skipped placing two bishops, we immediately stop the recursion because the current placement of bishops will lead to no solutions. Without this optimization (backtracking), the code won't run fast enough for an 8 x 8 board.

Problem I: Queen Coverage

An individual queen on a board of size 2000 x 2000 can attack at most 8000 squares. Since there are at most 2000 queens, we can one by one mark each square each queen can attack in 2000 x 8000 = 16,000,000 operations, which runs in time. Thus, we allocate a boolean array the size of the board, and then we go through each queen, marking the squares on its horizontal, vertical, forward diagonal and backward diagonal squares. After marking all of the squares, just count how many are "true" to get the final answer.

An alternative solution marks each row, column, forward and backward diagonal for each queen (so run time of 4 x 2000), in four separate sets. Then, we can go through each square on the board and see if the row, column, forward or backward diagonal for that square was marked.

Problem J: Super Pawn

We can think of the number of squares separating the two pieces in each column as the gap (counting 0 if the pieces are adjacent). If we make a normal forward move, what we are doing is reducing this gap by some number. For now, let's just assume we can make forward moves. Then, we can calculate the gap for each column and when we are playing the game, we must choose to reduce one of those gaps by any value. The loser is the person who can not make a move. This game is exactly the same as the famous game NIM, where two players have several piles of rocks and on each turn, the current player can take 1 or more rocks from a single pile. The solution to the NIM game is as follows: take the bitwise XOR of the number of rocks in each pile. If this value is 0, then the second player wins the game, otherwise the first player does. It turns out that if the XOR value isn't 0, then the first player can find a way to make the XOR value 0. Similarly, in this case, if the second player received an XOR value of 0, all moves he makes will change the XOR value to be non-zero.

Now, what about moving backwards? This doesn't help a player because however many steps one player moves backwards, the opponent can move exactly that many steps forwards, returning the game state to where it used to be before the two moves. Eventually, a player runs out of backwards moves.

Thus, to solve this problem, just take each column difference and bitwise XOR them. If the value is 0, BLACK wins, otherwise WHITE wins.