Problem B - Astronomy Problem

Difficulty: Medium-Hard, Hard

Required Knowledge: Geometry, Binary Search, Randomized Algorithm, Hill Climbing

One of the hardest problems of the set. Although there is a surprisingly simple and naive solution that works due to the nature of the problem. The number of stars can be at most 10, and the required precision is only 10^{-5}, thus enabling a randomized solution with hill-climbing or simulated annealing. The following method is able to produce correct answer for all the inputs in judge data. (Note: A lot of things work, no magic numbers for the constants below).

Repeat 100 times:
    Set a multiplying factor m = 0.997
    Choose a random coordinate between (-1000, 1000) as the station, denoted by (x, y)
    Choose a random number between (0, 1000) as the adjustment factor, denoted by t

    Repeat while t > eps (e.g. 1e-9):
        Set station_moved = False
        Repeat 10 times:
            Choose a random direction vector between (0-2π), denoted by theta

            (Move the station to a new coordinate following the direction vector and making a displacement according to the adjustment factor, t.)

            Let dx = x + sin(theta) * t, dy = y + cos(theta) * t

            If (dx, dy) can be a valid coordinate for the station and if the smallest angle made between two stars is more than that of (x, y):
                Set station_moved = True
                Set (x, y) = (dx, dy)

        If not station_moved:
            t = t * m

    Take the max of all the random trials and output it.

There is however another solution with binary search and computational geometry.

Problem C - AVL Trees

Difficulty: Medium

Required Knowledge: DP, FFT/NTT

The problem can be solved simply using DP in O(h^2 * n^2). Let dp[n][h] be the number of ways to have an AVL tree with n nodes and h height. Then, dp[n][h] can be computed using the following method.

```
Let dp[h][n] = 0
For (h1 = 0; h1 < h; h1++): // h1 = height of left subtree
        For (h2 = 0; h2 < h; h2++): // h2 = height of right subtree
                If (max(h1, h2) + 1) == h: // + 1 because one node for root
                        For (n1 = 0; n1 < n; n1++) // n1 = number of nodes in left subtree
                                Let n2 = n - n1 - 1 // number of nodes in right subtree (-1 (root) )
                                dp[h][n] += dp[h1][n1] * dp[h2][n2]
```

Notice that we can use fast fourier transformation or number theoretic transform to calculate dp[h][n] for all n at the same time for a fixed h, h1 and h2. This reduces the complexity to O(h^2 * n log n). A good implementation of FFT is sufficient to get accepted for the given constraints.

Problem D - Block Edit Distance

Difficulty: Easy-Medium, Medium

Required Knowledge: DP, Implementation

The problem is just a simple variation of the edit distance problem. We do have an additional operation here that we can select any substring of S and remove it using *b* cost. We can keep an additional flag in our usual edit distance DP.

DP (i, j, flag) indicates the minimum cost to convert S[i:] (suffix starting from i) and T[j:] where flag denotes whether we are not finished removing a substring from the prefix of S[i:]. Besides the usual transition as in edit distance, flag = 0 means we can start a removal from the prefix of S[i:] and call to the next state DP(i + 1, j, 1) + *b*. If flag = 1, we can continue removal with no additional cost or stop removing, min(DP(i + 1, j, 1), DP(i, j, 0)).

Problem E - Cryptography

Difficulty: Easy

Required Knowledge: Brute force, Bitmasks

x ^ y == y
This means that y must be a submask of x.

Fix x (x <= n) and for a fixed x, loop through all submask of x and just take the maximum.

Complexity: $3^{\log_2 n} = 3^{17} \sim 10^8$ iterations in the worst case, which passes quite comfortably.

Problem F - Independent Set

Difficulty: Medium

Required Knowledge: DP, Expressions

The problem is not as daunting as it seems initially. It can be parsed and solved recursively and the maximum independent set can be computed using simple DP (coupled with a few greedy observations).

Complexity: O(n)

Problem G- 3D Knights

Difficulty: Medium

Required Knowledge: Backtracking, CSP, Heuristics

A simple variation of the knight's tour (not closed) problem in 3D. Can be solved similarly as in 2D using heuristics like Warnsdorff' rule. A simple backtracking solution would be to start from the starting cell and recursively try out each of the neighboring cells if not visited. We can use a heuristic similar to the least constraining value heuristic to optimize the search. The idea is to try out each of the unvisited neighbors as in plain brute force, but sort and order the neighbors first based on the count of unvisited neighbors reachable from those neighbors.

In other words, let D(x, y) be the count of unvisited neighbors reachable from cell(x, y) in one move. Let the last cell of the knight's journey be (kx, ky). Let V be the set of unvisited neighbors reachable from (kx, ky) in one move. For each (x, y) in V, sort and arrange them in non-decreasing order of D(x,y). Pick each of them as the next cell of the knight's journey in that order and enumerate recursively in the same manner. Don't forget to mark the next cell as visited and also to stop the search when a solution is found.

Problem H - Perfect Lodging

Difficulty: Medium

Required Knowledge: Graph Matching, Randomized Algorithm

Not much to discuss here, just a very straightforward problem of finding maximum matching in an undirected general graph. We can use the Edmond's Blossom Algorithm or Gaussian Elimination on the Tutte matrix to find the maximum matching. Other randomized algorithms can also make it if done properly.

Problem I - Hungry Queen

Difficulty: Medium

Required Knowledge: Data structures, STL

We know the current position of the queen at any moment, and we also know the coordinate of the next cell, the next pawn which the queen must capture in order to continue the journey. If the queen cannot reach the next pawn in one move (not in the same row, column or diagonal), the game is over. Otherwise, all that's left to do is check whether another pawn lies in between the two cells. We can use sets in STL to track the pawns in each row, column and diagonals of the chessboard. We can then use lower_bound and upper_bound on sets to efficiently find the previous and next active pawn from any cell following the row, column or diagonal (depends on where the destination is). If there is no intermediate pawn continue the journey and remove the pawn from sets.

Problem J - Trip Expenses

Difficulty: Easy

Required Knowledge: Shortest paths, Floyd Warshall

This is just a simple variation of the floyd warshall. Distance between two nodes is defined as the cheapest path first, and as a tiebreaker smaller number of nodes for the same cheapest path. Just run a floyd warshall on pairs (the first one being distance and the second one number of nodes in between) and do what is asked.

Problem K - Hey Better Bettor

Difficulty: Hard

Required Knowledge: Probabilities, Gambler's Ruin, Ternary Search

The most beautiful and probably the hardest problem of the set. Understand that in the optimal strategy, and in any position, the only thing that matters are two things.

1. How much can you afford to lose before its best to quit?
2. How much should you win and then stop being greedy and quit?

Let us think of it like this. You have two types of coins, the number of coins of the first type indicates how many times can you afford to lose, denoted by *L*. Similarly the number of coins of the second type indicates how many times should you win before quitting, denoted by *W*. If the number of coins of the first type reaches 0, you immediately stop playing, losing some amount. Similarly if the number of coins of the second type reaches 0, you gain some amount and immediately stop playing.

Now, for a fixed *L* and *W*, we can calculate the probability of us gaining some amount and the probability that we lose some amount. The probability to gain some amount can be solved easily since we basically transformed the problem into Gambler's ruin problem. Let this probability be *p*, and the probability to lose, $q = 1 - p$.

So our expected winning amount for a fixed *L* and *W,* denoted by $E(L, W) = p*W - q*L*x$

All that's left to do is find an optimal *L* and *W.* We can fix L or W (loop through 0 to 10^5) and use ternary search on the other variable or simple use a nested ternary search on both. The reason why ternary search will work in this case is left as an exercise for the reader.

Problem L - Sums of 2 and 3

Difficulty: Giveaway

Required Knowledge: DP

Just a simple DP, to generate the sequence known as Padovan Sequence. dp[n] = dp[n-2] + dp[n-3].

Problem A - Agrarian Reform

Difficulty: Hard

Required Knowledge: Graphs, Matching, Mincost Maxflow

Construct the graph, and run hungarian algorithm or mincost maxflow. Take care to order the nodes using topological sort beforehand. Finally you will need to output the matching.