***All expected numbers of ACs were estimated before the contest.**

A. The One with the Infinite Squares

Problem Setter: Tanzir Islam
Tester: Anik Sarker
Alternate Writer: Mehdi Rahman, Aminul Haq
Category: Computational Geometry, Randomization
Expected no. of AC: 25+

Brief idea:
We need to find the area of the convex hull created by the given points after each point is generated. Let **h** be the maximum number of points on the hull at any moment. We can check if the latest point is inside the current hull in O(log(h)). We compute the convex hull again only if the point is outside the current hull. The hull can be reconstructed in O(hlogh) using Graham Scan or any other algorithm or even in O(h). Let **k** be the number of times we reconstruct hull. Then the final complexity will be: `O(Nlogh + khlog h)`

Due to the randomness of the input the expected value of **h** is `O(log(N))` and the expected value of **k** is `O(h log N)` or $O(\log^2 N)$

So the final expected complexity is: $O(N\log(\log N) + \log^3(N)\log(\log(N)))$

Detailed idea:
The first observation is that the area of intersection of all the squares will be the convex hull of the given points. To prove this, we can take infinitely large squares and place them in such a way that one of its sides coincides with a side of the convex hull. Now, if the hull contains **h** sides and we put **h** infinitely large square intelligently in this way, their area of intersection will only be the convex hull. These sides of the squares are acting as half-planes basically.

Next, point in convex polygon can be checked in logarithmic time using binary search. Alternatively it can be checked linearly. Unless the linear implementation is unreasonably heavy, we have set the TL to let those solutions pass.

Why expected value of h is O(log(N)):
Please check lemma 2.4 of this paper to understand the reasoning behind the expected value of O(logN). But of course, we did not expect you to prove this during the contest. We expected you to have an intuition and experiment on your pc to approximate the number in case you did not know this before.

Why expected value of k is O(h log N):
After the i-th point is generated, let's say we have checked if the given point is inside the current hull. We have either discarded it if it was inside or we have again built the hull. Now the hull has **h** points. Now we look at the hull and ask this question: what is the probability that the latest (i-th) point is on the hull and we had to reconstruct the hull after it was generated? The probability will be $\frac{h}{i}$ as the points are random. The reasoning is: there are in total **i** points right now and only **h** of them are on the hull. So the probability of the latest point being one of those **h** points is $\frac{h}{i}$**.** So,

Expected value of k,

$E(k) = \sum \frac{h_i}{i}$ (where, 1 <= i <= N; h_i is the number of points on hull after the i-th step)

$$=> E(k) \leq h \times \sum \frac{1}{i}$$

$$=> E(k) \leq \log(N) \times harmonic(N)$$

$$=> E(k) \leq \log^2(N)$$

B. Kings Vs Rooks

Problem Setter: Sabit Zahin
Tester: Shahed Shahriar
Alternate Writer: Nafis Sadique, Anik Sarker
Category: Dynamic Programming, Case Analysis, Linear Recurrence, Polynomial Interpolation
Expected no. of AC: 5+

Analysis:
This problem can be solved in multiple ways. It can be solved in O(N) with a tedious DP by case analysis. Perhaps it's easier to write a brute force and try to find patterns from it. There exists a linear recurrence of degree 8 and a polynomial of degree 5 (after excluding n=1 and n=2), which can be found by linear recurrence solvers like berlekamp-massey/reed-sloane or lagrange polynomial interpolation.

C. Almost Palindrome

Problem Setter: Sourav Sen Tonmoy
Tester: Sabit Zahin
Alternate Writer: Raihat Zaman Neloy, Aminul Haq
Category: Hash & Data structure
Expected no. of AC: 5

Analysis:
TL;DR - Hashing along with data structures like segment tree or BIT at a runtime of `Q*Log^2(|S|)`.
Query type 1:
For answering if a segment (L, R) is palindrome or not, we can just check this segment's forward hash and reverse hash. If these two don't match, we need to find the index which can help us make it a palindrome.
For that, we need to find the **Longest common prefix** of S[L...R] and reverse of S[L...R]. These can be efficiently found by storing hash values in data structures like segment tree. Lets say, we found the LCP and its length is X. Let all characters from S[L+X-P] to S[L+X] be the same. So the minimum value of P can be 0. We can remove(hypothetically) the S[L+X-P]-th character and check if the resulting substring is a palindrome or not. The value of P can be found by keeping a set where we store the boundaries of each block of characters who have the same value.

If removal of the S[L+X-P]-th character does not produce a palindrome we can check again by removing the S[R-X]-th character.
Query type 2:
We will need to replace characters in the index. To add the update feature in hash, we will need some kinds of data structure to maintain. For that, we can use segment tree, BIT or even BST if anyone can code it during the contest. Then it will be a simple update operation in those DS.

D. Winning Arrangements

Problem Setter: Nafis Sadique
Tester: Hasnain Heickal
Alternate Writer: Md. Mahbubul Hasan, Sabit Zahin
Category: Dynamic Programming, Counting
Expected no. of AC: 1-2

Analysis:
A different problem
There are N players standing in an arrangement, one of them is you. We know the strength of each player including you. The rules are the same, N-1 matches played, neighboring players are chosen and the loser leaves. Winner gains one strength. If you are given the choice of determining the matches, can you guarantee a win?

Approaching the different problem

The easiest way to determine if a win is possible is to just play with your neighbors as long as they are not stronger than you. If both of them are stronger, then you lose, otherwise you play with them, defeat them and continue.

We can even have a deterministic order, let's say we will always play with the left neighbor first, as long as we can. Once we find there are no left neighbors or the left neighbor can't be defeated with our current strength, we start playing with the neighbors to the right. This way is one of the optimal ways to play.

Slower solution

Based on this approach, we can generate all the arrangements that you can win if played optimally. We generate it by placing players to your left and right while maintaining a valid winnable state. The state can be defined with just a few parameters, a) your strength, b) number of players already placed left and right of you, c) strength of the left next player, who you can't beat yet.

From your strength and already placed players, you know the number of players that we can win against but haven't been placed yet. Initially there aren't any players to the left or right to you now. Everytime our strength increases we need to add the players with the new strength to our hand.

We can place K of them to the left and then place a stronger player who we can't beat even after placing the K players. Then we keep placing players to the right until we can win against the stronger left player. Now we can place players to the left again. We have to keep track of our player's strength and the number of players placed on the left or right and iterate through the stronger player to be placed on the left. This can be done in `O(n^3)` or `O(n^4)`. Since the strengths can be very large, we need to use some sort of indexing system, where we jump from one strength to another one.

Actual Solution

If you look carefully, then you can notice that we really don't have to place a stronger player to the left manually. All we have to do is when we are at the right side placing players and we jump to a new strength, we place one of the players of the new strength to the left and then we are back at the state where we can place on either left or right. Otherwise we keep placing on the right. This way we can achieve `O(n^2)`.

E. Alice, Bob and the array

Problem Setter: Mohammad Ashraful Islam
Tester: Md. Imran Bin Azad
Alternate Writer: Tanzir Islam Pial
Category: Greedy, Ad Hoc
Expected no. of AC: 750

Analysis:
Trivial Problem:
Given an array $\{a_1, a_2, a_3, \ldots, a_N\}$, If we ask you to find the maximum consecutive subarray sum you can find it easily.

In this problem:
Choose two indices i and j and reverse the whole range in a single operation. You can perform this operation as many times as you want. Target is to maximize the maximum consecutive subarray sum with the minimum number of operations.

Solution:
Corner Case :
All the numbers in the array is negative: Max($a_1, a_2, a_3, \ldots, a_N$) will be the answer.

General Case:
Max Sum that can be achieved = Sum of all positive integers.
Minimum number of moves required = Number of Contiguous Negative segments - 1 (if neg segment at the end of the array) - 1 (neg seg at the beginning of the array)

Why will this work?: Figure it yourself.

F. Co Primes

Problem Setter: Tanveer Muttaqeen
Tester: Tanzir Islam Pial
Alternate Writer: Rezwan Mahmud Tonmoy, Mohammad Ashraful Islam
Category: Number Theory, Principle of Inclusion Exclusion
Expected no. of AC: 50+

Analysis:
Let's assume a $\geq$ b.
It can be proved that gcd(a, b) = gcd(a - b, b)
Now,
gcd(a+i, b+i) = gcd(a+i-b-i, b+i) = gcd(a-b, b+i).

So we need to find the number of integers from b to b+m who are coprime with a-b.
Let, x = (a-b)
Let x have p prime factors who can be found in O(sqrt(x))

We can backtrack all possible combinations of the factors in 2^p. In each combination, we calculate the LCM of picked factors and either add or subtract the number of integers divisible by the LCM from our answer. The addition or subtraction will be decided based on the parity of the set (Inclusion Exclusion Principle).

Final complexity per test case: `O(`2^p` x log(p) + sqrt(x))`
Where p is the maximum number of prime factors of an integer in the given range.

G. Strange Typewriter

Problem Setter: Tanmoy Datta
Tester: Raihat Zaman Neloy
Alternate Writer: Sabit Zahin, Sourav Sen Tonmoy
Category: String data structure
Expected no. of AC: 40

Analysis:
This problem has a lot of different solutions -

Judge's solution:

TL;DR - Trie + KMP, Z or Hash + DP or Bfs with a complexity of `O(|s| * sqrt(X))` where X is the summation of lengths of all the key strings.

We have a constraint that the total number of characters of the N keystrings will be 10^6. So, if we look closely, we can divide all the keystrings in two groups,

(1) Light group with string length less than sqrt(10^6). Number of this type of string can be large. Lets say, if all the strings have length less than 10, then we will have a maximum of 10^5 key strings.

(2) Heavy group with string length greater than sqrt(10^6). We will have at most 1000.

Now, we can solve each group separately. Because, if all the strings have length of 1000, then we will have at most 1000 different strings.
For light groups, build a trie, so the max depth of the trie will be 1000. Now, on the main string, from each we can do a search query and find occurrences.
For heavy groups, we can do any kind of string matching as at most we will get 1000 strings. KMP, Z or even Hash can work here.

After finding the occurrences, we can now do DP or BFS whatever is feasible to code in order to find the shortest path.

Tester’s solution:

TL;DR - Aho Corasick + DP or Bfs with a complexity of `O(|s| * sqrt(X))` where X is the summation of lengths of all the key strings.

We can use Aho Corasick to find the occurrences. A simple build of Aho corasick will work. While traversing the main string in Aho, we have to climb up using the failure chain to find the occurrences. But if we look closely, simple climbing can produce TLE as failure function always points to the maximum matched suffix. So, we need to change the failure function in a way so that it points to a full string.

After we have built the efficient Aho Corasick with modified failure link, then we can use anyone of DP or BFS to calculate the shortest path

Alters’ solution:

TL;DR - Hash + DP or Bfs with a complexity of `O(|s| * sqrt(X))` where X is the summation of lengths of all the key strings.

If we observe all the key strings, then we can see there will be at most 1000 unique lengths of strings. So, we will keep track which lengths occurred in the input, and hash all the strings. Then, we can simply iterate over the lengths we have got and do hash matching to find the occurrences.
Now, like the other solutions, here we can also do either DP or BFS to find the shortest path.

H. Recursion, Lambda and Parameter

Problem Setter: Md Mahbubul Hasan
Tester: Rezwan Mahmud
Alternate Writer: Anik Sarker, Shahed Shahriar
Category: Computational Geometry
Expected no. of AC: 10+

Analysis: Let’s imagine the problem in a 2d plane. (Pi, Ri) are points on this 2d plane. Any weighted combination of the points are just points inside the convex hull defined by the points. Similarly the other direction is also true. Any point inside the convex hull is achievable. Now let’s focus on the queries. If we transform the query to the 2d plane, it will become a “x = k” or “y = k” line. So our problem becomes, given a “x = k” line, what's the maximum y intersection with the convex hull of the input points. This can be solved using binary search on the convex hull.

I. Lattice Triangle

Problem Setter: Shahriar Manzoor
Tester: Derek Kisman
Alternate Writer: Md Mahbubul Hasan, Md. Imran Bin Azad
Category: Computational Geometry, Algebra
Expected no. of AC: 175

Analysis: Do you know the pick's theorem? This gives a relation between the number of lattice points inside, the number of lattice points on the border and area. Using this formula you can easily solve the problem.

J. COVID - 19

Problem Setter: Hasnain Heickal
Tester: Md. Imran Bin Azad
Alternate Writer: Sabit Zahin
Category: Ad Hoc
Expected no. of AC: ∞

Analysis: Just do it!