UTIPC 2019 Solutions

Ethan Arnold, Arnav Sastry

April 2019

Introduction

Disclaimer: this document contains rough outlines of techniques that can be used to solve these problems. Nothing in this document is legally binding. Please email arnavsastry+icpc@gmail.com or masonsbro456@gmail.com with errata.

Thanks to all the teams that came to UT in person and those that competed remotely. We had a lot of outstandingly fast submissions at the beginning of the contest, with team Jonathan Shoemaker being the first to solve 10 problems at around the 78 minute mark. After that, the race to the top slowed down slightly, as competitors who had also done the Codeforces round an hour earlier got ramped up and teams approached the three hardest problems: B, I and L. In the end, team "U-Rah-Rah! Wisconsin!" completed the set at the 148 minute mark and secured first place. 8 teams overall solved all of the problems, congratulations!

1. U-Rah-Rah! Wisconsin!, University of Wisconsin-Madison
2. Yuusha de aru, University of Central Florida
3. UBC BAD, University of British Columbia
4. Jonathan Shoemaker, Massachusetts Institute of Technology
5. cuom1999, Southern Methodist University
6. i did codeforces first, University of Waterloo
7. Felipe Mota, Federal University of Campina Grande
8. Inno, Innopolis University

A: Painting Pips

First Solve: 3 min by Yuusha de aru. Solved by: 45 teams

If X_i is the number rolled on the i-th die, Alice's payout is $X = \prod_i X_i$. We want to maximize $E[X] = E[\prod_i X_i]$. Since the X_i are independent, this is the same as $\prod_i E[X_i]$. But $E[X_i]$ will simply be the number of pips painted on the i-th die, divided by 6. It doesn't matter which sides she paints the pips on.

So if A_i is the number of pips on the i-th die, we just need to maximize $1/6^N \prod_i A_i$ subject to the constraint that $\sum_i A_i = M$. To do this, we should make the A_i as close together as possible: if $A_i - A_j > 1$ for some i, j, we can get a better answer by decrementing A_i and incrementing A_j. So we should greedily paint a pip on the die that currently has the fewest pips. This solution is $O(n)$.

If math isn't your thing, this problem can also be solved with dynamic programming. Let $E(i, j)$ be the maximum expected value Alice can get using i dice and j pips. Then the recurrence is

$$E(i, j) = \max_{0 \le k \le j} E(i - 1, j - k) \cdot \frac{k}{6}.$$

This solution is $O(nm^2)$, which easily runs in time.

B: Serious Business

First Solve: 1 hour, 49 min by U-Rah-Rah! Wisconsin!. Solved by: 16 teams

First, notice that we can change the problem to the following: given R, find the number of lucky serial numbers less than or equal to R. If we let this value be $f(R)$, the final answer is $f(R) - f(L-1)$.

To compute $f(R)$, we use dynamic programming. What does our state need to include? In other words, what information do we need in order to determine if a number is lucky? A number is lucky if the number of even-sum substrings is odd. A substring sum can be represented as a difference of two prefix sums (or two suffix sums), so if we keep track of the length of the number we've built so far, the parity of the last digit, and the number of prefixes (or suffixes) that have even sum, we can determine the number of prefixes (or suffixes) that have odd sum as well. Letting the number of even prefixes be A and the number of odd prefixes be B, the number of even-sum substrings is $\binom{A}{2} + \binom{B}{2}$.

It would be too much information to store the number of even-sum prefixes, though (there would be at least N^2 states where N is the length of R). Note that all we care about is the parity of $\binom{A}{2}$ and $\binom{B}{2}$ in the above expression, and that the parity of the function $\binom{K}{2}$ repeats with period 4. So we can just store the number of even prefixes modulo 4, deduce the number of odd prefixes modulo 4, and compute the parity of $\binom{A}{2} + \binom{B}{2}$.

It is easier to actually build up a suffix instead, and to add a few more pieces of state: whether we are allowing zero as a digit, and whether the number that we've built so far is less than or equal to the corresponding suffix of R. The final time complexity will be linear with a large constant factor (usually 80, 160, or 320 depending on implementation details).

There is also a simple solution based on a pattern: half the 1-digit numbers are lucky, all of the 2-digit numbers are lucky, half the 3-digit numbers are lucky, none of the 4-digit numbers are lucky. This period then repeats modulo 4.

C: Permutations on the Road: Bob

First Solve: 30 min by Eric Price. Solved by: 32 teams

There are many methods to solve this problem. Here is one approach that builds the solution from left to right.

Let P be Alice's chosen permutation and P_i be the i-th element (1-indexed) of P. Consider $\text{inv}(1, n) - \text{inv}(2, n)$. This gives the number of inversions that involve P_1. More specifically, this gives the number of values in $P_2, P_3, \ldots, P_n$ that are less than P_1. Let's keep a working set S of values we haven't used yet. We'll also define $f(i) = \text{inv}(i, n) - \text{inv}(i+1, n)$. Clearly, P_1 is equal to the $f(1)$-th smallest element of S. Then we remove that element from S and compute the rest of the permutation.

In this approach, we query $\text{inv}(i, n)$ for all $1 \leq i < n$, for a total of $n-1$ queries. Implementing the above procedure naively with a dynamic array (`vector` in C++ or `ArrayList` in Java) runs in $O(n^2)$ which is fast enough for $n = 1\,000$. This can easily be improved to $O(n \log n)$ using data structures.

As it turns out, this problem did not require bitmask DP or the simplex algorithm.

D: Recess Rocks

First Solve: 4 min by UBC PEF. Solved by: 59 teams

Consider the student with the least number of rocks. In order to protect their fragile ego, we should place them at the front of the line. In any other case, they will be able to see another student who has more rocks than them, and they will cry. Any other student with the same number of rocks is guaranteed to cry, so we might as well put them at the end.

This motivates the general strategy: create a prefix of students who don't cry by taking as many students with distinct rock counts and place the rest at the end to cry. An equivalent formulation would be to just sort the list of students and see which ones cry. The overall answer is the number of students minus the number of students with distinct rock types. Both of these methods run in $O(n \log n)$.

E: Social Dancers

First Solve: 39 min by The Extraordinarily Excellent Tantalizing Hotshots. Solved by: 35 teams

The long-ish statement boils down to the following. There are l leads and f follows. Each dancer knows some subset of 3 different dances. Dancers will only pair up if, for every dance either both of them know it or both of them do not know it. Assuming the dancers pair up optimally and m random songs are played, what is the expected number of dancers?

For a given lead, they will only pair with a follow who knows the exact same set of dances that they do. It does not matter which dancer pairs with which follow, just how many pairs there are for each subset of dance styles, the minimum of the number of leads and follows with that subset. Then for each of the three songs, we can compute how many pairs can dance to that song. Divide by 3 to get the average number of dances for a single song, and multiply by m since all styles are independent. Overall complexity is $O(n)$.

F: Permutations on the Road: Alice

First Solve: 12 min by U-Rah-Rah! Wisconsin!. Solved by: 29 teams

Consider a pair of 1-indexed indices $l < r$ where $P_l > P_r$. There are $l \cdot (n-r+1)$ subarrays which contain both l and r, so this inversion contributes $l \cdot (n - r + 1)$ to the overall sum. A naïve solution runs in $O(n^2)$ and sums this value for all l and r. This is too slow with the given input.

To speed this up, fix r. The contribution to the answer of a fixed r is

$$\sum_{l<r, P_l>P_r} l \cdot (n - r + 1) = (n - r + 1) \sum_{l<r, P_l>P_r} l.$$

There are several approaches to computing that sum quickly. The easiest is to create a data structure which supports point updates and prefix sum queries (like a Fenwick tree or segment tree). Then, process r from left to right. After computing the answer for r, set $f[P_r] = r$ in the data structure. Computing the sum of all l is now a single range query in your data structure. The sort and implementation means this runs in a fast $O(n \log n)$.

Fun aside: The maximum answer (for input of a descending permutation of length 100 000) is on the order of $4 \cdot 10^{18}$, which is getting close to half the limit of a signed 64-bit integer. Luckily we found this bound to be sufficient in pruning out $O(n^2)$ solutions. Originally this problem was going to be writing the grader for Permutations on the Road: Bob, but we had trouble distinguishing between fast $O(n^3)$ solutions and $O(n^2 \log n)$ solutions. As a challenge, try writing the grader for that problem if $n = 10^5$ instead.

G: Elevators

First Solve: 3 min by Network Connectivity Problems. Solved by: 60 teams

This is a straightforward implementation problem, and the easiest in the set. One approach is to copy the table into a bunch of if-statements, but a shorter implementation uses the fact that the elevator requirements for each zone are linear in the number of floors, with only a varying slope depending on the zone. For example, to get the number of elevators for a residential building, divide the number of floors by 5, rounding up.

H: Rounded Work

First Solve: 20 min by Orange Juice. Solved by: 34 teams

Fix a denominator B, and write the value of round(A/B) for $A = 1, 2, \ldots, N$. First there will be $\lfloor (B-1)/2 \rfloor$ 0s, followed by B 1s, then B 2s, and so on. We can sum these with the triangular number formula in $O(1)$ time. To compute the answer, sum the values for all B and then divide the result by N^2. The runtime is $O(B)$.

I: Cactus Shoppe

First Solve: 1 hour, 17 min by Cameron Reynoldson Fan Club. Solved by: 12 teams

The first part of this problem is to figure out which vertices and edges are included for a given q_i. Clearly if $q_i \mid f_i$, then vertex i is included. Similarly, if there's an edge (a, b) and $q_i \mid f_a$ and $q_i \mid f_b$, then the edge (a, b) is included in the final graph as well. The above expression can be condensed into $q_i \mid \gcd(f_a, f_b)$.

If the input were a tree, then every edge would have to connect two separate components, and the answer would be the number of used vertices minus the number of used edges. As it turns out, cacti are quite like trees, they just have some cycles added. Because of the condition that each edge can belong to at most one cycle, the only way an edge does not connect two previously disconnected components is if the whole cycle is included. A cycle is included if and only if every edge on that cycle is included. That is, if $e_1, e_2, \ldots, e_k$ are the edges on a cycle, and $f(e_i)$ is the gcd of the values of the endpoint of the edge, the cycle is included if $q_i \mid \gcd(f(e_1), f(e_2), \ldots)$. Since each edge appears in at most one cycle, there are at most m cycles (the real bound is closer to $m - n + 1$), so we can compute the vertex values, edge values, and cycle values in $O(n + m)$ time.

For a fixed set of included vertices V, edges E, and cycles C, we get a delightfully simple formula for the number of connected components:

$$|V| - |E| + |C|.$$

The last bit of this problem is to compute each of these counts. For a given q_i, we need to count the number of vertices, edges, and cycles, This can be done in $O(\max f_i \log \max f_i)$ time by computing the Dirichlet convolution on the frequencies of the above arrays. In other words, for each value that the gcd can take, change the answer for every multiple of that value, similar to the sieve of Eratosthenes. Then each query can be answered in $O(1)$ time.

During the contest, teams found an alternate solution that works for general graphs. For each possible input, create a graph G_x which contains only vertices and edges from G where each f_i is divisible by x. There are at most 240 divisors for a number under the given bounds, so this graph has at most $240n$ nodes and $240m$ edges. Now run your favorite connected component counting algorithm on each of these graphs independently to pre-compute the answers.

J: General Knight

First Solve: 8 min by The Extraordinarily Excellent Tantalizing Hotshots. Solved by: 47 teams

The simplest solution to this problem is to iterate over all squares in the order given and see if they have the correct distance from the knight. Let Δc be the difference in columns between the knight and a square and Δr be the difference in rows. Then the knight can move to that square if $\Delta c = a$ and $\Delta r = b$ or $\Delta c = b$ and $\Delta r = a$. By traversing in column-first order, you don't need to do any additional sorting of the output.

K: Buffon's Needle

First Solve: 8 min by unordered_cartographers. Solved by: 44 teams

We are given a bunch of line segments, and we want to find how many of them intersect a vertical line $x = k$ for integer k. The y coordinates in the input don't matter, and the coordinates are all small in absolute value. We can just check, for each line segment, for each vertical line it might cross, whether the two x coordinates are on different sides of the line.

Another different approach to figure out if x_1 and x_2 crossed any line was to cast them to ints and see if they were different. However, casting a double to an integer rounds towards 0, so the needle $(-0.5, 0), (0.5, 0)$ had both x_1 and x_2 round to 0, even though they cross 0.

L: Neural Networks

First Solve: 1 hour by U-Rah-Rah! Wisconsin!. Solved by: 15 teams

First, let's find an equivalent condition for a neural network being interesting. It is clear that it is enough for each node to have at least one incoming edge and at least one outgoing edge. We can rephrase this as saying that, for each layer of *edges* (between two adjacent layers of nodes), every node in the two layers of nodes must have at least one incident edge in this layer of edges. Since all these layers of edges are independent problems, we can count the ways to fill out each one, and then multiply those results.

This subproblem is as follows: how many bipartite graphs there are with A nodes on the left and B nodes on the right such that there are no nodes of degree 0? An equivalent formulation is, how many 0/1-matrices are there with A rows and B columns that have no empty rows or columns? This can be seen as the adjacency matrix of the bipartite graph.

First, how many graphs are there where each of the nodes on the right has an edge? There are $(2^A - 1)^B$ ways: each of B nodes can independently take as its neighbors any subset of the A left nodes, except for the empty set.

The above formula overcounts the answer, as some of the nodes on the left may not have edges if none of the nodes on the right connect to them. We can use the inclusion-exclusion principle to correct for this. The answer is

$$\sum_{i=0}^{A} (-1)^i \binom{A}{i} \left(2^{A-i} - 1\right)^B.$$

The above sum can be computed in $O(A \log B)$ time by precomputing factorials and inverse factorials and using fast exponentiation. Summing over all the layers, the total time complexity is $O(S \log S)$, where S is the sum of the values in the input array.

Shout-out to the team named Network Connectivity Problems for solving a problem about Network Connectivity.

This was the only problem solved in a language outside of C++ / Java / Python. A team solved it in Rust.