Introduction

Disclaimer: this document contains rough outlines of techniques that can be used to solve the problems from the invitational. Nothing in this document is legally binding. Please email utpc@cs.utexas.edu with any errata.

Thanks to everyone who competed in the invitational, both remotely and in-person! "not Benq" was the only team to solve all 12 problems, finishing at the 113th minute mark. 5 teams were able to solve 10 or more problems. Congratulations to everyone who competed!

A: Bump, Set, Spike!

First Solve: 5 min by "AFK from 120 to 150 :(". Solved by: 34 teams

This is an implementation problem intended to be the easiest in the set. For every touch in the rally, check if the touch violates any of the three rules, e.g. to check for duplicate touches, ensure that the previous touch was not by the same player. For this problem, it is very important to read the problem carefully, as there are various edge cases that are important to consider. For example, the ball is considered "hit to the other team" at the end of the rally, so you must check for the gender violation at the end of the rally.

Once a violation is detected, print the team that committed the violation. There is a slight bit of trickiness to this, since the gender rule can be violated by the non-violating team touching the ball.

B: Bump, Set, Spike, Count!

First Solve*: 23 min by "AFK from 120 to 150 :(". Solved by: 18 teams

*The first solve time is distorted since there was an error in the problem statement.

The goal in this problem is equivalent to computing the number of ways to construct a list of size $N$ using the elements $S = \{A1, A2, A3, A4, A5, A6, B1, B2, B3, B4, B5, B6\}$ such no three consecutive elements have the same first character (team) and no two consecutive elements have the same character and number (player). One observation we can make is that in order to check if an element can be added to a list, we only need the value of the last two elements. Therefore, we can first precompute for each ordered triple of elements, whether these elements can be next to each other in the list by checking that they don’t violate any of the rules. Using this information, we can write a recurrence as follows.

$$ f(N, a, b) = \begin{cases} 
1 & \text{if } N = 0 \\
\sum_{c \in S} \begin{cases} 
f(N - 1, b, c) & \text{if } (a, b, c) \text{ is a valid triple} \\
0 & \text{otherwise}
\end{cases} & \text{otherwise}
\end{cases} $$

Using dynamic programming we can compute $f(N)$ for all $a$ and $b$ in $O(N \times 12^2)$ which is fast enough for the bounds given.
C: Candy Consumption

*First Solve: 7 min by "not Benq". Solved by: 19 teams*

We want to calculate the summed length of all possible candy consumption sequences. A candy consumption sequence of length \( K \) consists of \( K - 1 \) unique elements permuted in some order and is terminated by another copy of one of these \( K - 1 \) elements. For example, a candy consumption sequence of size 4 with candies A, B, and C is BACA. We can iterate over each possible length to determine how many candy consumption sequences there are of that length and multiply by this length to find the overall number of candies consumed. Given a candy consumption sequence is length \( K \), there are \((N \binom{K-1}{N})\) ways to select the \( K - 1 \) unique elements, \((K - 1)!\) ways to permute these unique elements, and \( K - 1 \) ways to select the terminating candy (must be one of the \( K - 1 \) unique elements).

Multiplying these values by length and iterating over all possible lengths \( (2 \leq K \leq N + 1) \) we get

\[
\sum_{K=2}^{N+1} \binom{N}{K - 1} \cdot (K - 1)! \cdot (K - 1) \cdot K
\]

This can be calculated in \( O(N) \) time.

D: Chemical Equations

*First Solve: 26 min by "not Benq". Solved by: 17 teams*

We can represent these chemical equations as a DAG (Directed Acyclic Graph) where we represent each compound as a vertex and form a directed, weighted edge from A to B if compound B is needed for form compound A. With this representation, the leaf nodes are base elements and we need to express the desired compound in terms of base elements.

To do this, create a recursive procedure that simplifies a desired compound in terms of the base elements. The base case is when the compound is a leaf, in which case the desired compound is simplified to itself with a quantity of 1. If the compound is not a leaf, we call this procedure on each of its children and then update the quantities in the base element accordingly. For each child, we find the base elements needed to make the child and associated quantities and multiply these quantities by the quantity of the child needed to make the parent compound. We sum up (and combine as needed) all the quantities of base elements from each child to get the final answer.

Because the DAG has no cycles, we are guaranteed this process will terminate. We will call this procedure only once for each compound if we store intermediate results and there are a constant number of edges for each compound, so we can compute the result in \( O(N) \) time.

Note: There was initially an issue with overflow, but the test data was updated in contest to ensure the quantity of each base element fit in a signed 32 bit integer.

E: Con-cat-nation

*First Solve: 30 min by "Benq". Solved by: 17 teams*

Trying all possible combinations for each cat results in an \( O(N^2) \) solution, which is too slow. With the constraints of the problem, we notice that, when appending an identifier to our own, we only care about its first character and its length (and whether the identifier has any invalid successor characters). Thus for each identifier, we first check if itself is already good enough, then, if not, iterate through every length and first characters, and if combining them would be good enough (of course, we can only combine if we have an id which matches this criteria). This results in a \( O(|\Sigma|\text{maxlen}|n) \) solution.

F: Dominant Conversations

*First Solve: 85 min by "Benq". Solved by: 4 teams*

This can be solved in \( O(n \log(n)) \) using a lazy segment tree.

Basically, you need a lazy segtree that supports the following operations in \( O(\log n) \):
- setting a range
- incrementing a range
- querying a range for sum
- querying a range for sum of the form \( n \times x_n + (n - 1) \times x_{n-1} + ... + x_1 \)

The trickiest one of these is the last query, but you can accomplish it by storing by a fixed multiplier for each segment, e.g. the \( A \times x_n + (A-1) \times x_{n-1} + ... + (A-n+1) \times x_1 \) and then subtracting \( (A-n) \times (x_n + x_{n-1} + ... + x_1) \) using your sum query

Once you have this segtree you solve for each person \( p \) individually, by imagining the array as -1 for those elements \( \neq p \) and 1 for those that equal \( p \). This will break the problem into ranges of -1 and +1. This idea helps, as if we calculate running sums the number of dominating segments ending at a point \( i \) equals the number of prior running sums such that \( (rsum_i - rsum_j > 1) \).

To efficiently operate over these ranges we use our segment tree. In the segment tree, we keep track of the count of each \( rsum \) (index \( ind = \) of \( rsums \) such that \( rsum_j = ind \)) (Note that you can also accomplish this by having the segment tree store each \( rsum \leq ind \), which simplifies the implementation) When operating over a segment of -1, we can use our range increment to accomplish this. The main problem now is calculating the number of dominating segments given this information. Notice how, in the case 1, -1, -1, .. (range of -1) the running sum decreases by one each time. This is where the last query comes in handy. Say the maximum \( rsum_j \) such that \( rsum_i - rsum_j > 1 \) for the 1 is \( x \), and the number of \( x \)'s we have is \( C(x) \), then this sum is

\[
C(-\infty) + C(-\infty + 1) + ... + C(x - 1) + C(x)
\]

adding on the next -1 we get

\[
2C(-\infty) + 2C(-\infty + 1) + ... + 2C(x - 1) + C(x)
\]

then adding one more we get

\[
3C(-\infty) + 3C(-\infty + 1) + ... 3C(x - 2) + 2C(x - 1) + C(x)
\]

now we can break the above into our two queries, namely

\[
3 \times (C(-\infty) + ... + C(x - 3)) + 3C(x - 2) + 2C(x - 1) + C(x)
\]

(or if we used the \( \leq \) representation, we would simply do a range addition)

We then use the set range to clear our segment tree for each step.

As we have a total of \( n \) elements, and for each element we have a constant number of \( \log n \) operations, this is \( O(n \log(n)) \).

G: Leaky Cauldron

First Solve: 58 min by "not Benq". Solved by: 12 teams

To begin with this problem, note that \( N, M, H \) are all relatively small, so a solution that can run in \( O(N \cdot M \cdot H) \) time would suffice to pass.

The first step to solving this problem is to break it up layer by layer. At any given point in time, there are a set number of squares in the grid that have water in them. Taking any water height \( h \), we can simply apply a flood fill algorithm (DFS or BFS) to figure out which squares are covered by water. Essentially, every square that is height \( h \) or less will be covered by water if it can be connected to the center square with a path that contains squares also all height \( h \) or less.

To find the largest rectangle in \( O(NM) \) time can be done in many ways. One solution is to go row by row, keeping track of a running histogram of how many squares above any given square in the row are also filled with water. Then, a linear sweep can be performed on this histogram, with each square being put onto the stack if the top of the stack has height less than the current height. When this condition does not hold,
the stack is repeatedly popped, with the area calculated, until the square can be put onto the stack. This method can be performed in $O(MN)$ time, yet many other algorithms exist.

This is repeated $h + 1$ times for each of the heights, giving us an $O(MNH)$ solution which suffices to pass the constraints of this problem.

H: Martian Patrol

First Solve: 113 min by "not Benq". Solved by: 1 teams

Before reading this, note that the problem statement omitted a constraint on “buddies,” in that they need to be able to possibly meet while going in the same direction (in terms of left-right). (an alternative way to phrase this would be that is possible for the two to walk together for a non-infinitesimal amount of time, which was the original inspiration)

When reading the problem statement, you probably noticed it is very complicated. Lets go part by part when solving this.

First lets just look at the sadness requirement, and try to find some hidden patterns. The main indication that this might be fruitful is the sheer flexibility the algorithm is given, its seems like it would either be easy or impossible for two martians to be sad.

For simplicity, lets notate $a_l < b_l$ to represent that the starting point of $a$ is to the left of the starting point of $b$ (and similarly for $a_r < b_r$ in terms of ending points). Note that I will do all of the proofs assuming that we are starting on the left side, but starting on the right side is symmetric.

In that case between any two martians, without loss of generality we have three cases in terms of the relative arrangement

Case 1: $a_l < a_r < b_l < b_r$

a and b can’t be buddies, as even if their paths intersect they can only intersect like this:

\[
\begin{array}{c}
\text{A}_l \\
\text{A}_r \\
\text{B}_l \\
\text{B}_r \\
\end{array}
\]

in which case if a and b only meet if they are going in opposite directions, so they can’t be buddies.

Case 2: $a_l < b_r < a_l < b_r$
I posit that this arrangement is always sad. Essentially, I can always choose a time such that $A_l$ and $B_l$ meet at their lowest common ancestor, in which case $B$ is always to the right or equal to $A$. $A$’s path can have a length of at most $N - 1 - Y$, and $B$ similarly has length at most of $N - 1 - X$, so whichever one has a shortest path to their lowest common ancestor can wait to leave until they will arrive at the lca at the same time, at which point they will be equal until they diverge later.

Case 3: $a_l < b_r < b_l < a_r$

Obviously at the endpoints the two have swapped sides, so this arrangement can never be sad.

Now we come to the big step: the structure of the tree doesn’t actually matter. We can determine if an arrangement of nodes is valid by just comparing the position of the nodes, so we can compress this “binary search tree” into a sorted array. Now we see how to deal with the extra room, as each placement of the extra room shares the same answer, we can find the answer for one tree and multiply it by the number of places to place the extra room. Now the question becomes:

Given an array of size $n + 1$, what are the number of ways to place intervals on this array such that no interval covers another’s endpoint?

Or alternatively, given an array of size $n + 1$, what are the number of ways to place copies of the characters “(”, “0”, “)” in this array, such that when the zeros are removed the parenthesis statement is valid. This has become a fairly standard counting problem. Now we sum over each of the possible number of soldiers:

$$\sum_{k=0}^{\lfloor (n+1)/2 \rfloor} \frac{1}{k+1} \binom{2 * k}{k} \binom{n + 1}{n + 1 - 2 * k}$$

and our final answer is:

$$(\text{number of valid places for the secret room}) \sum_{k=0}^{\lfloor (n+1)/2 \rfloor} \frac{1}{k+1} \binom{2 * k}{k} \binom{n + 1}{n + 1 - 2 * k}$$

all calculated with modular arithmetic. This solution is $O(n \log n)$ if you calculate modular inverses with modular exponentiation, though you can speed it up to $O(n)$ by calculating inverses in a smart way. Either passes.

I: Pigeon X

First Solve: 38 min by ”unordered_cartographers”. Solved by: 16 teams

In this problem, we wish to minimize the cost of pairing up pigeons with seed where the cost of pairing a particular pigeon to a particular seed is the shortest distance from the pigeon to the seed using corridors in the mansion. Therefore, the first thing we can do is compute all the costs of pairing pigeons with seed using an All Pairs Shortest Path algorithm such as Floyd-Warshall. Now that we have all these paths, the problem is a vanilla minimum cost bipartite matching problem which can be solved using algorithms such as Minimum Cost Maximum Flow or the Hungarian Algorithm.
J: Red Light, Green Light

First Solve: 29 min by "Random Projections". Solved by: 13 teams

Given a specific speed $S$, we can calculate how long it takes for Jay to get from home to work using a slightly modified Dijkstra’s algorithm. First, the “length” of each edge is translated from the distance given in the problem statement to the time it takes to traverse the road given the speed $S$. Next, when removing a node $u$ from the top of the min-heap and considering a neighbor $v$, the total time to reach the neighbor isn’t as simple as $d[u] + l[u][v]$, as we must also account for the color of the light at that moment (here, $d[u]$ gives the minimum distance to $u$ and $l[u][v]$ gives the amount of time it takes to get from $u$ to $v$ given a speed $S$). What we really need is something along the lines of $\text{first green}(d[u], u, v) + l[u][v]$, where $\text{first green}$ is a function that returns the first time greater than equal to $d[u]$ that the light for road $(u,v)$ is green. This function can be implemented in a straightforward manner.

This modified Dijkstra’s will tell us whether Jay can reach his workplace in under $T$ minutes given a speed $S$. Notice that if Jay can reach his workplace in under $T$ seconds given a speed $S$, then he can reach with any speed less than $S$ as well. The boolean function mapping from speed to whether work is reachable in $T$ minutes is monotonic, and thus we can perform a binary search over the real numbers to find the slowest speed that still allows Jay to reach his workplace in time.

K: Undetermined Determinants

First Solve: 14 min by "not Benq". Solved by: 5 teams

The formula for determinant of an $n \times n$ is given by (from Wikipedia)

$$\det(A) = \sum_{\sigma \in S_n} \left( \text{sgn}(\sigma) \prod_{i=1}^{n} a_{i,\sigma_i} \right)$$

Here, $S_n$ is the symmetric group on $n$ elements, and so the sum is computed over all permutations $\sigma$ of the set $\{1, 2, ..., n\}$. $\sigma_i$ gives the value in the $i$th index in the permutation $\sigma$, and $\text{sgn}(\sigma)$ gives the parity of the permutation $\sigma$, evaluating to 1 or $-1$.

Using this definition in the context of finding the determinant of adjacency matrices of trees, we can make the observation that if the tree has a perfect matching, then the permutation that reflects this perfect matching will contribute 1 or $-1$ to the determinant sum, depending on the parity of $N/2$. If this is confusing, think about how to construct a permutation such that the product in the equation above becomes 1. We can also observe that if there is no perfect matching, then the determinant will be 0. Finally, realize that a tree can have at most one perfect matching. To convince yourself of this, use a greedy approach to construct the perfect matching. Putting this all together, we get that trees with no perfect matching will have determinant 0, and trees with a perfect matching will have determinant 1 or $-1$ depending on the parity of $N/2$.

Thus, the problem reduces to counting the number of labeled trees on $N$ vertices that have a perfect matching. Say we fix a perfect matching and want to count how many trees have that perfect matching. We can treat each pair of matched nodes as one single node, and count the number of ”compressed” trees there are on these $N/2$ nodes. By Cayley’s formula, this is given by $(N/2)^{(N/2-2)}$. Now, for each compressed tree, we must count the number of ways to expand it, i.e. for each endpoint of each edge, choose which of the two nodes in that compressed node the edge should be incident on. There are a total of $2 \times (N/2 - 1) = N - 2$ such edge endpoints, and for each we have 2 choices, giving us a total of $2^{(N-2)}$ choices in expansion. Thus, for a fixed matching, there are a total of $(N/2)^{(N/2-2)} \times 2^{(N-2)}$ labeled trees. We then multiply by the number of matchings to get the total number of labeled trees of size $N$ with perfect matching.

The final answer can then be constructed as follows:

- If $N$ is odd, the answer is 0.
- If $N/2$ is even, the answer is the total number of labeled trees with perfect matching multiplied by the inverse of $N^{(N-2)}$ (total number of labeled trees) computed modulo 998244353.
- If $N/2$ is odd, the answer is the same as above but negative.
L: Uzbek Trek

*First Solve: 19 min by "HS 1". Solved by: 16 teams*

The problem is essentially asking us to compute the number of ways to achieve a sum of \( N \) using integral powers of \( P \). Note that it is possible to reuse a particular power of \( P \) as many times as needed.

In order to simplify the problem, first assume \( P = 2 \). Then, notice that if \( N \) is odd, then it must require the power \( 2^0 = 1 \) in its sum. If \( N \) is even, then it can either have the power \( 2^0 = 1 \) in its sum or not. If it does not, then every power in the sum is of the form \( 2^k \) where \( k > 0 \). Hence the following recurrence suffices to count the number of ways to achieve \( N \) through sums of powers of 2.

\[
f(N) = \begin{cases} 
0 & x = 0 \\
(f(N - 1)) & \text{if } x \text{ odd} \\
(f(N - 1) + f(N/2)) & \text{if } x \text{ even}
\end{cases}
\]

Finally, this can be easily generalized to arbitrary \( P \) as follows.

\[
f(N) = \begin{cases} 
0 & x = 0 \\
(f(N - (N \mod P))) & \text{if } x \not\equiv 0 \mod P \\
(f(N - P) + f(N/P)) & \text{if } x \equiv 0 \mod P
\end{cases}
\]

This can be implemented using dynamic programming in \( O(N/P) \) time which should pass all the test cases.