

A Bouncy Castle

Author: Michael Xiang

An answer x is only possible if $x^2 \leq 2n$. As in, we can treat each bed as 2 blocks and try to find the largest square that uses at most $2n$ squares.

So, the answer is $\lfloor \sqrt{2n} \rfloor$.

B Beta Testing

Author: Victoria Huang

This problem asks us to find the total number of distinct rooms we can reach starting from Room 1. We can model the map as a directed graph, where each room is a vertex, and each one-way vent is a directed edge.

To find all reachable vertices, we can perform a very standard graph traversal, like a DFS starting from vertex 1. During the traversal, we must maintain a boolean array or a set of *visited* rooms to ensure we don't revisit the same room or get stuck in an infinite loop (which would occur if the one-way vents form a cycle).

Every time we discover a new, unvisited room, we mark it as visited and increment our reachable room count. Once the traversal finishes, this count will be our final answer.

Since we visit each vertex at most once and traverse each edge at most once, the overall time complexity is $\mathcal{O}(n + m)$.

C Champion's Meeting (Easy)

Author: Michael Xiang

Since all the racers have unique prestige levels, we can use a direct greedy approach. Consider choosing either a_1 and b_1 as the first racer of C. Either $a_1 < b_1$ or $a_1 > b_1$. If $a_1 < b_1$, we should always take a_1 since all possible outcomes with a_1 first are better than all possible outcomes with b_1 first. Similarly, if $a_1 > b_1$, we should always take b_1 first. Once we have chosen a racer, we can remove them, and then compare the new first elements of a and b .

This process continues down the line, each time taking the racer with lower prestige to be the next racer in team C. If one team runs out of racers, just take a racer from the other team.

This results in a $\mathcal{O}(n + m)$ algorithm.

My solution can be found [here](#).

D Anagrams

Author: Nick Robinson

Because the maximum number of letters is very small ($n \leq 8$), the most straightforward solution is to use brute-force backtracking to generate every possible string. You can use a frequency map to track how many of each letter you have available, then write a recursive function that builds strings character by character by choosing a letter, exploring further, and then "un-choosing" it. Every time you form a valid string during this recursion, insert it into a set which automatically filters out duplicate strings caused by identical input letters. Since the absolute maximum number of unique strings you can form from 8 characters is only about 109,600, generating them all and summing up their lengths will easily execute well within the 1-second time limit.

Below is my solution using the C++ next_permutation method: [click here](#).

E Strongest Attack First

Author: Colin Stack

Note that setting your level to ℓ reduces the damage per second of each attack to $\max(0, d_i - \ell)$. Therefore, the total damage dealt during the i -th wave will be $\max(0, d_i - \ell) \times t_i$. Therefore, the total damage from the skeleton's attack is the sum of this quantity over all waves:

$$D(\ell) = \sum_{i=1}^n \max(0, d_i - \ell) t_i$$

Thus, we must find the minimum ℓ such that $D(\ell) < h$. The key observation is that $D(\ell)$ is a monotonically decreasing function; increasing ℓ reduces or doesn't change the damage per wave, so the total damage can never increase as ℓ increases. So, if a certain ℓ permits survival, then all larger ℓ will also permit survival. Conversely, if you die at player level ℓ , then you will also die at any smaller player level ℓ . Because this is true, we can binary search for the minimum level that allows survival.

So, perform a binary search over ℓ in the range $[0, \max(d_i)]$. This upper bound works because if $\ell \geq \max(d_i)$, every attack will deal zero damage and survival is guaranteed. For each ℓ during the search, compute the total damage and if it reaches or exceeds h , then survival is impossible, otherwise the ℓ works.

Each damage check will take $O(n)$ time and the binary search has $O(\log(\max(d_i)))$ iterations. Thus, the total time complexity is $O(n \log(\max(d_i)))$, which is sufficient for the bounds on n and d_i .

F Chippa Rank

Author: Saadhya Vijayvargiya

Each uma must travel a different total distance. If the i th uma starts at placement a_i , then they must run a total distance of $d + a_i$. Since their speed is s_i , their finishing time is

$$t_i = \frac{d + a_i}{s_i}.$$

Chippi Chappa always starts at position $n + 1$, so he must run a distance of $d + n + 1$. If his speed in a query is g , his finishing time is

$$t_c = \frac{d + n + 1}{g}.$$

Chippi Chappa's rank is simply 1 plus the number of standard umas whose finishing time is smaller than t_c .

To compare these times without using floating point arithmetic, we cross multiply:

$$\frac{d + a_i}{s_i} < \frac{d + n + 1}{g}$$

which is equivalent to

$$g(d + a_i) < s_i(d + n + 1).$$

Rearranging gives

$$\frac{d + a_i}{s_i} < \frac{d + n + 1}{g} \iff g < \frac{s_i(d + n + 1)}{d + a_i}.$$

Define

$$x_i = \frac{s_i(d + n + 1)}{d + a_i}.$$

Then uma i finishes before Chippi Chappa exactly when $g < x_i$. Therefore, for a query speed g , the number of umas that beat Chippi Chappa is the number of x_i values strictly greater than g .

We can precompute all x_i values and sort them. For each query, we perform a binary search to count how many x_i are greater than g . If k umas finish before Chippi Chappa, his rank is $k + 1$.

Computing the x_i values takes $\mathcal{O}(n)$ time and sorting them takes $\mathcal{O}(n \log n)$. Each query can then be answered with a binary search in $\mathcal{O}(\log n)$ time, giving a total complexity of $\mathcal{O}((n + q) \log n)$.

G Tyson's Taunt

Author: Aiyaz Mostofa

First, consider that the answer is the number of subarrays with mean $\geq a$, minus the number of subarrays with mean $> b$. These subproblems can be solved using the same underlying algorithm.

Looking at the formula for the mean of a sequence x ,

$$\bar{x} = \sum_i \frac{x_i}{n}$$

Rearranging, we get,

$$\left(\sum_i x_i \right) - n\bar{x} = 0$$

Which we rewrite as,

$$\sum_i (x_i - \bar{x}) = 0$$

So if we want to see if an array has a mean $\geq a$, then we see if,

$$\sum_i (x_i - a) \geq 0$$

Similarly, if we want to see if an array has a mean $> b$, then we see if,

$$\sum_i (x_i - b) > 0$$

Since everything is an integer, this is equivalent to,

$$\sum_i (x_i - b) \geq 1$$

We've now reduced the subproblem(s) to finding the number of subarrays of s with a sum $\geq m$, given m . Let s' be the cumulative sum version of s , with 0 attached at the beginning (the same setup as prefix sums).

For each r , try to find the number of subarrays that have endpoint r and have a sum $\geq m$. Using a similar approach to prefix sums, we need to find the number of elements in $s'[0 \dots r-1]$ that have value $\leq s'_r - m$. You can use a fenwick tree, segment tree, or order statistic tree to achieve this.

Time complexity: $O(n \log n)$

H Red Combo

Author: Zafir Nasim

A naive approach is trying to find where the cuts are. You track the range of elements seen, and if it matches $1 - p$ or $p - n$ for some p , you assume a cut and validate the two sub-problems. This is too slow; if a cut only takes one element off one side, you end up with $O(N^2)$.

Given the constraint on N we need an $O(N)$ solution. Trying out some permutations, you might realize the process is actually pretty tolerant. The smallest "NO" case is $[2, 4, 1, 3]$, where no neighbors are numerically adjacent. It's clear this permutation can't be made with the process because somehow for all ranges it cuts it then sends the two parts off so they aren't adjacent. Reversing this idea, $[1, 3, 4, 2]$ is valid because you can merge adjacent neighboring numbers: combining 3 and 4, then merging with 2 (which is adjacent to $3 - 4$), then 1 (which is adjacent to $2 - 4$). If everything merges into a single $1 - N$ range, we know we have a valid permutation (consider undoing the merges, it's equivalent to the cutting process).

Algorithm: We use a stack to merge these ranges as we go:

1. Push the current number as a range $[x, x]$ onto the stack.
2. While the top range is numerically adjacent to the one below it (like $[1, 2]$ and $[3, 4]$), pop them both and push the merged range.
3. If you finish with exactly one range $[1, N]$, output YES.

We add N elements and merge up to $N - 1$ times so this is $O(N)$.

I Champion's Meeting (Hard)

Author: Michael Xiang

Define ω to be a symbol larger than any other prestige level. For any two sequences X and Y , define $X \leq Y$ to mean that X is lexicographically at most Y . Also, let XY denote the concatenation of two sequences X and Y .

We claim that if $X\omega \leq Y\omega$, there exists a lexicographically minimal interleaving of X and Y such that the first element comes from the first element of X .

If $X = Y$, any interleaving that starts with an element from Y can be symmetrically transformed into one that starts with an element from X (just swap all X s with Y s and vice versa).

Otherwise, $X\omega < Y\omega$. Let P be the longest common prefix between $X\omega$ and $Y\omega$. We can then write

$$X = P\alpha X' \quad Y = P\beta Y'$$

where $\alpha < \beta$, and β could be ω .

Among all possible lexicographically minimal interleavings of X and Y , choose one where the first element of X appears as early as possible. For the sake of contradiction, assume that there are $t \geq 1$ elements from Y that appear before the first element of X .

If $t > |P|$, the first $|P| + 1$ elements form $P\beta$. However, the first $|P| + 1$ elements could have been taken from X , which would produce $P\alpha$. Since this is also a valid prefix of the final interleaving and $P\alpha < P\beta$, the selected interleaving is not minimal.

Thus, $t \leq |P|$. The first t elements of X and Y are the same, so we can swap their sources and still produce a valid interleaving. For the first t elements, instead of taking from Y , take them from X . For the first t elements that originally came from X , take them from Y instead. This preserves the relative order inside both sequences and does not change the final interleaving, contradicting the assumption that the earliest possible position of the first element of X was not the first position.

Thus, there must exist a lexicographically minimal interleaving starting with the first element of X .

Solution 1

To accomplish this, we can run a 2-pointer approach. Initialize i and j to 1. Set $a[n+1] = b[m+1] = \omega = k+1$. Then, find the smallest value of x such that $a[i : i+x] \neq b[j : j+x]$ using binary search. This can be done by finding the minimal such x such that the string hashes of those 2 substrings are different. Then, just compare $a[i+x]$ and $b[j+x]$. The result of this comparison is the same as comparing the augmented suffixes $a[i : \omega]$ and $b[j : \omega]$. Choose the best element, then increment the corresponding pointer (if the suffixes are the same, take from either one). This results in $O(n+m)$ binary searches, for a total of $O((n+m) \log(n+m))$ time.

ChatGPT's can be found [here](#).

Solution 2

Let A_i and B_j be the suffixes of a and b starting at indices i and j .

Consider the array

$$U = (a_1, \dots, a_n, k+1, b_1, \dots, b_m, k+2)$$

Comparing $A_i \omega \leq B_j \omega$ is the exact same as comparing the suffixes of U starting at indices i and $n+1+j$, respectively:

- If A_i and B_j differ before one of them ends, both comparisons stop at the first differing prestige level.
- If the difference reaches exactly one of $k+1$ or $k+2$, the array with a longer real suffix would be considered smaller, which is expected.
- If the real suffixes are equal, then the comparison reaches $k+1$ and $k+2$. This algorithm takes from a by default, which is safe here.

We can then use a 2-pointer approach, each time taking the better of a_i and b_j by comparing the ranks of i and $n+1+j$. Computing all ranks of this array can be done with a suffix array in $O((n+m) \log(n+m))$ time.

My solution can be found [here](#).

J Building Bridges

Author: Warith Rahman

In both Alice and Bob's version of the problem, we are essentially constructing a *Maximum* Spanning Tree. This can be solved the exact same way as the normal Minimum Spanning Tree problem, just by processing the edges in decreasing order instead. Note that the graph is complete (all pairs of vertices are connected), so using Prim's Algorithm continuously considers all n neighbors of an added vertex, devolving into $\mathcal{O}(n^2 \log n)$.

We'll use a modified version of Kruskal's Algorithm. Note that if $g \mid a_i$ and $g \mid a_j$, then $g \mid \gcd(a_i, a_j)$. Iterating from each g from $g = \max a$ to $g = 1$, we'll connect any vertices i and j such that g divides both a_i and a_j . If a particular g connects vertices i and j , then a lower $g' \mid g$ won't connect vertices anyway, so we never consider extraneous edges. As soon as our DSU detects that we reduced the vertices to a single component, we store this g , and we get $\beta = g^2$, since $\gcd(w^2, z^2) = \gcd(w, z)^2$.

With $m = \max a$, the time complexity of the solution is approximately $\mathcal{O}(m \log m)$. My code is [here](#).