Thank you for participating in BAPC 2025!

This editorial may be updated if errors are found (this version was last updated March 3, 2025). See the most up-to-date version here:

https://www.overleaf.com/read/twbmgksvytqt#7d57ba

# Problem A

We are at $(a, b)$ and we are trying to get to $(n, m)$:

- If $a = n$, we have to go up.
- If $b = m$, we have to go right.
- Otherwise, suppose that $a < b$. Then we should go right (increase $a$ by 1). And if $a > b$ then we should go up (increase $b$ by 1). If $a = b$ then it doesn't matter what we do.

Let's justify this. Suppose without loss of generality that $a \leq b$. Since $a < n$, we will have to go right eventually.

Let's imagine an alternate solution where we don't go right at this step. Let $(a, b')$ (where $b' > b$) be the first step where we do actually go right. The total value collected on the path from $(a, b)$ to $(a, b')$ will be $a(b + (b + 1) + (b + 2) + \ldots + b')$ — but if we were to instead move right immediately, the value would be $ab + (a + 1)((b + 1) + (b + 2) + \ldots + b')$, which is strictly greater.

So we just need to simulate the algorithm above.

Runtime: $O(n + m)$.

# Problem B

Try drawing strings as paths on a grid. So 0 corresponds to moving right, and 1 corresponds to moving up.

As you play around with a few examples, you may start to notice a pattern: the area between the two strings stays the same when we apply $f$. Furthermore, it seems that $(x, y)$ is between strings $s$ and $t$ IF AND ONLY IF $(2x + y + 1, x + y)$ is between $f(s)$ and $f(t)$!

It's pretty straightforward to prove this when $(x, y)$ is a single isolated unit of area. If you iron out the definitions of being "below" or "above" a string, then you can in fact prove this in general. But in contest, just checking a few examples should be strong enough evidence to convince you.

With this fact, we can see that the Fibonacci-like transform makes all units of area between the strings grow further and further apart as you apply $f$. By the time we get to $f^{22}$, in fact, all units of area will be isolated from each other. So $S$ and $T$ will differ only at disjoint pairs of locations, where each pair is just two consecutive indices (this pattern should've stuck out like a sore thumb just from looking at the sample inputs). If you want to prove this, try putting two units of area directly next to each other in $S$ and $T$. Then apply the inverse of $(2x + y + 1, x + y)$ 22 times — we see that the two units of area cannot both fit inside the $300\,000 \times 300\,000$ bounding box we are interested in.

The idea, then, is as follows: for each index $i$, map it back to a coordinate $(x, y)$. Now, with the set of coordinates, find two strings $s$ and $t$ that enclose exactly those coordinates.

It's not immediately clear, in the first part, that $(x, y)$ is uniquely defined from $i$ — each $i$ can be mapped back to infinitely many different coordinates! But: let $(X, Y)$ be the result of applying the transformation $(x, y) \mapsto (2x+y+1, x+y)$ exactly 22 times. Then the contribution of $x$ to the answer will be $F_{46}$, the 46-th Fibonacci number, and the contribution of $y$ will be $F_{45}$. It is known that all two adjacent Fibonacci numbers are coprime. Conclusion: all solutions $(x, y)$ that map to any given index are spaced out by at least $F_{45}$. So, in particular, there is a unique solution $(x, y)$ within the $300\,000 \times 300\,000$ box we care about!

Now you can precompute the linear mapping $(x, y) \mapsto (X, Y)$. Let $v_x$ be the contribution of $x$ to the answer, and define $v_y$ similarly. Suppose we are interested in an index $I$. You can use your favorite GCD algorithm to find $a, b$ such that $av_x + bv_y = 1 \Rightarrow av_x i + bv_y i = i$. Now modding $av_x i$ by $F_{45}$ and modding $bv_y i$ by $F_{46}$ will give a solution within the box we're interested in. Alternatively, as suggested by tourist, Cassini's identity immediately gives the values for $a$ and $b$.

Now, how do we find strings that enclose exactly this area? We can start by getting more specific with our definitions of "above a string" and "below a string":

- $(x, y)$ is *above* a string $s$ if the first prefix of $s$ with $x + 1$ zeros has $y$ or fewer ones.
- $(x, y)$ is *below* a string $s$ if the first prefix of $s$ with $x + 1$ zeros has $y + 1$ or more ones.

This gives us an idea: we will try and construct $s$ as the string above all $(x, y)$ that is above as little area as possible. And $t$ will be the string below all $(x, y)$ that encloses as much area as possible. How to construct $s$ and $t$ follows pretty easily from the definitions above.

There's one last wrinkle we need to sort out: what if $s$ goes below $t$ at some point? This can only happen if there are some units of area that are not connected, for example, if $s = 100110$ and $t = 011001$ (try drawing these out!). In this case, all we need to do is reverse the substring of $s$ that's between those two separated units of area.

Runtime: $O(n \log(a_i) + 300\,000)$.

# Problem C

Let's first get a working description of good pairs. Let $x, y$ be a good pair of positive integers.

First note that the highest bits of $x$ and $y$ in binary cannot be the same. Otherwise, the XOR $x \oplus y$ would have that highest bit set to 0, and then $x \oplus y < \max(x, y)$.

Now, without loss of generality, suppose $x < y$. Then, the highest set bit of $x$ must be set to 0 in $y$ — otherwise, that bit would be set to zero in $x \oplus y$. Since no higher bits are changed, this means that $y \geq \max(x, y) > x \oplus y$. On the other hand, if the highest set bit of $x$ is set to 1 in $y$, then $x \oplus y$ will be strictly greater than $y$. So now we have an if-and-only-if condition.

This motivates a DP solution. We insert new elements into our constructed set in increasing order. At each step, we only need to keep track of the highest set bits of the elements in our set.

Let the bitmask of the highest-set-bits of all our elements so far be $b$. Then it is possible to add a new element $x$ with a higher highest-set-bit to our set if and only if $b \& x = 0$. Or in other words, if $b$ is a submask of $\text{NOT}(x)$ — the inverted bitmask of $x$.

Now we can put together the final solution. Read in all the elements. For each $0 \leq x < 2^{22}$ and $0 \leq y < 22$, we need to determine if there exists some $a_i$ that with highest set bit $y$, such that $a_i$ is a

supermask of $x$. This can be done in $O(a_i \log(a_i))$ by reading in all elements first, then doing a scan down (see the model solution for details).

Now we can DP. $dp[i]$ = answer if our good set is represented by bitmask $i$. We can transition from $i \to i \oplus (2^j)$ iff $i\&(2^j) = 0$, and there exists some input element $a_i$ with highest-set-bit $j$ that is a supermask of $i$. Again, please refer to the model solution for details — it's pretty short!

Runtime: $O(a_i \cdot \log(a_i))$.

# Problem D

Solutions for this problem found during testing ranged from digit DP to polynomial interpolation, but the solution described here is beautifully simple, and comes from Thomas Marlowe.

Suppose that there is a robot at position 1 that you want to hit exactly $j$ times, and that you have already fired the gun $i$ times before. Let $dp[i][j]$ be the number of ways to fire the gun in this situation.

$dp[0][j] = 1$ for all $j$, since your only option is to fire the gun $j$ times at $t = 1$.

Otherwise, you can case on the number of times you fire the gun before the robot reaches position $1/2$. Note that the robot moving by $1/2$ if you've fired $i$ times is the same situation as the robot moving by 1 if you've fired $i - 1$ times! So, suppose you fire the gun $k$ times before the robot reaches position $1/2$. The number of ways to do this is $dp[i-1][k] \cdot dp[i-1+k][j-k]$.

You can sum this across all $0 \le k \le j - i$ to compute $dp[i][j]$. You can then compute all dp values $0 \le i \le j \le a_n$ in $O(a_n^3)$.

To get the answer for the problem, then, you just need to multiply $dp[a_i][a_{i+1}]$ for every two adjacent robots.

Runtime: $O(n + (a_n)^3)$.

# Problem E

The important observation is that you can perform an odd operation at most once. Once you perform the odd operation, there will be an even number of odd elements. And the even operation does not affect the parity of any element.

If there are at least 2 indices $1 \le i \le n$ such that $a_i$ is odd and $a_i \ne b_i$, then the answer is `NO` by the above.

Otherwise, there is at most one odd element that is not equal to its counterpart in $b$. We can do the following three cases:

- **Case 1**: don't perform the odd operation at all.
- **Case 2**: perform the odd operation on the odd unequal element, then perform even operations as needed.
- **Case 3**: perform the necessary even operations, then perform the odd operation.

These cases are exhaustive: it is impossible to perform the odd operation between two even operations, since the number of even elements will change after the odd operation.

Runtime: $O(n)$.

# Problem F

The structure of prerequisites described makes a rooted tree.

Let $dp[i]$ be the answer for the subtree rooted at node $i$. If children$(i)$ are the direct children of $i$ in the tree, then $dp[i] = \max\left(0, a_i + \sum_{x \in \text{children}(i)} \max(dp[x], 0)\right)$ — we either use 0 nodes, or we use this node and some of its children.

The answer for a type-2 query for a node $c$, then, will be $\max\left(1, \sum_{c' \text{ an ancestor of c}} dp[c']\right)$. Note that $c$ is part of the optimal plan (I say "the," since you can prove that the optimal solution is unique: exercise to the reader) if and only if all $dp[c']$ are non-negative. A negative dp value at an ancestor corresponds to that ancestor not being included in the solution for the parent of that ancestor. On the other hand, if all ancestors have a non-negative value, that means that all ancestors are included in the optimal solutions for their parents.

Now for type-1 queries. For simplicity, suppose we perform a type-1 query on a node $v$, incrementing its value by **exactly 1**. If all dp-values of $v$'s ancestors are non-negative, then the *set of nodes* included in the optimal solution does not change (the optimal profit made will increase by 1, but we use the same nodes), and the dp values of all ancestors of $v$ (including $v$ itself) increase by 1.

Otherwise, let $v'$ be the deepest ancestor of $v$ with a negative dp value. Then the dp-values of all nodes between $v$ and $v'$ (inclusive) will increase by exactly 1, but no other dp values will change. Why? At best, the dp value of $v'$ will become 0. See the definition of our DP above and convince yourself that this is true.

Finally note that in type-2 queries, we don't care about the exact dp-value of a node if it is non-negative. So we will just pretend that all dp-values are capped at 0. The algorithm to handle type-1 queries can then be described:

```
let x = (input parameter)
while x > 0 && (some ancestor v' of v has negative dp value):
    let incr = min(x, -dp[v'])
    x -= incr
    dp[v'] += incr
```

To implement all this efficiently, we can use a Fenwick tree or your favorite range-sum/point-update data-structure. Use an Euler tour over the tree. For type-2 queries, we just need to check if the sum of all values between $v$ and the root is negative. For type-1 queries, we can binary search over the ancestors of $v$ to find the first negative one, add to its dp-value, and repeat until we either run out of negative ancestors, or we no longer have any more to add to them.

For the updates from type-1 queries, every node gets incremented to zero at most once. So amortized, we only need to do $n + q$ node updates over all queries.

As a few bonus ideas:

- Benq's solution uses a top tree (correct me if the linked article isn't referencing the right data structure), and also works when type-1 queries have negative values.
- PurpleCrayon's solution, as pointed out by tourist, uses a DSU to more efficiently keep track of the closest negative ancestor. This lets us chop off a log factor from the solution.

Runtime: $O((n + q)\log^2(n))$.

# Problem G

Two grids are commutative if and only if at least one of the following holds:

- One of the grids is empty.
- The two grids are equal.

If the first condition holds, then both $G_1 \circ G_2$ and $G_2 \circ G_1$ will be empty. The second condition is clear.

What if neither of these conditions hold? Let $(r, c)$ be any cell where $G_1[r][c] \neq G_2[r][c]$. Without loss of generality suppose that $G_1[r][c]$ is filled in. Then in $G_1 \circ G_2$, the tiny grid at $(r, c)$ will be a tiny copy of $G_2$. But in $G_2 \circ G_1$, that tiny grid will be empty. Since we assumed $G_1$ is nonempty, this means $G_1$ and $G_2$ are not commutative.

We can then just, for example, just create (for example, in C++) a `map<string,int>` to keep track of how many of each type of grid there is. When we read in a new grid:

- If the new grid is empty, add the number of grids previously read to the answer.
- Otherwise, add the number of equal grids previously read, and the number of empty grids previoiusly read to the answer.

Runtime: $O(nm^2 \log(n))$ or $O(n(m^2 + \log(n)))$, depending on how you check grid equality.

# Problem H

The root node will always be 1. Now note that since we swap the left and right child at each step, the odd-numbered nodes and the even-numbered nodes will be in different subtrees of the root. Furthermore, in both of these subtrees, we insert the new nodes in order! For example, in the even subtree, the nodes inserted in this subtree will be $2, 4, 6, 8, \ldots$.

So the two subtrees of the root have exactly the same structure as the root: all the nodes are just multiplied by 2 (and possibly also 1 is subtracted). This gives us some very nice bit of recursion: given the number for any node, we can figure out what the path from the root to that node is. And given the description of a path starting at the root, we can figure out what node it ends at, or determine that the path leaves the heap.

Using these two subroutines naively is fast enough. It was also possible to derive exactly the relation between the parent's index and its two children (see mwen's or Agastya's solutions if you are interested).

Runtime: $O(|s| \log(n))$.

# Problem I

Once you get past the statement, the problem naturally becomes two separate checks:

- For each day, it must be impossible to order the students such that for some $i$, the $i$-th student presenting that day has enthusiasm strictly less than $i$.

  Suppose we sort the students presenting that day: if for some $i$, the $i$-th student in sorted order has enthusiasm strictly less than $i$ then the answer is `NO` by the above.

And if there is no $i$ such that $e_i < i$, then I claim that it is possible for Ezra to order the students on this day. Suppose for the sake of contradiction, that there were some ordering where a student had enthusiasm strictly less than their position in line. Then, if we were to sort the students by repeatedly swapping adjacent students, then one of those unfortunate students would have to end up having $e_i < i$ in the sorted order too — contradiction.

(This explanation was a bit terse: if you are confused, try reading this, or let us know where your confusion is, and we will try to update the editorial).

- It must be impossible for students to pick events such that on some day, there are not events for everyone to present a unique event.

  We can just check this by a greedy simulation. The worst-case for any given day is if all students presenting on that day choose the events that expire the furthest in the future (try and prove this if you like!). Simulate the students picking these events in the worst way possible using your favorite BBST or other similar data structure (e.g. `std::set` in C++).

  If on any day there are not enough events to present, we abort and output `NO`.

Runtime: $O((n+m)\log(m))$.

# Problem J

The optimal solution for each building is to go all the way to the left, use the teleporter, then go all the way to the right. It's clear that this is necessary (going all the way to the left is the only way to visit building 1, ditto for building $n$). And sufficient.

We can then keep prefix sums and suffix sums. In the prefix sums, $pf[i]$ = "how many $i' < i$ have $h_{i'} > h_{i'+1}$" — this is the energy we need to expend to get to building 1 from building $i$. Similarly, in the suffix sums, $pf[i]$ = "how many $i' > i$ have $h_{i'} > h_{i'-1}$."

The answer for building $i$ is then $\max(pf[i], sf[i])$.

Runtime: $O(n)$.

# Problem K

Let $dp[i][j]$, where $0 \le i \le 1$ and $0 \le j < 2^n$, be the minimum number of trips for FJ to be on side $i$ of the river (0 =left side, 1 =right side), where $j$ is the bitmask of the set of cows on the **opposite** side.

If we were to run a naive BFS on this, we could solve the problem in $O(n^2 2^n + 3^n)$ — since for each $j$, the edges going out of $j$ go to the submasks of the complement of $j$. But this is too slow.

Instead, the idea is to add in some dummy states. $dp[i][j][k]$, where $0 \le k \le n$, we can imagine as being a state where FJ is still deciding on which cows to bring across. Let's describe the transitions explicitly, then hopefully everything will be clear:

- $dp[i][j][0]$ means that FJ, so far, has not decided to bring any cows across with him yet. The edges going out of this node will be:
  - Transition for a cost of 0 to $\rightarrow dp[i][j \oplus 2^x][1]$, where $x$ is some cow on the same side of the river as FJ (so $j \& 2^x = 0$). This transition corresponds to FJ deciding to take cow $x$ along with him.

– Transition for a cost of 1 to $\rightarrow dp[i \oplus 1][(2^n - 1) \oplus b][0]$. This transition corresponds to FJ finalizing the list of cows he takes with him and making a trip across the river.

Basically, we can either decide to take another cow with us, or to just go across. We need similar transitions for each $k \geq 1$, which are pretty much the same as what we described above. The total number of edges coming out of each state will be $O(n)$.

Our starting state should be $(i, j, k) = (0, 0, 0)$. Our ending state will be $(i, j, k) = (1, 0, 0)$. Now we just need an 0-1 BFS (don't store the edges explicitly or you might get MLE!).

Runtime: $O(n^2 2^n)$.