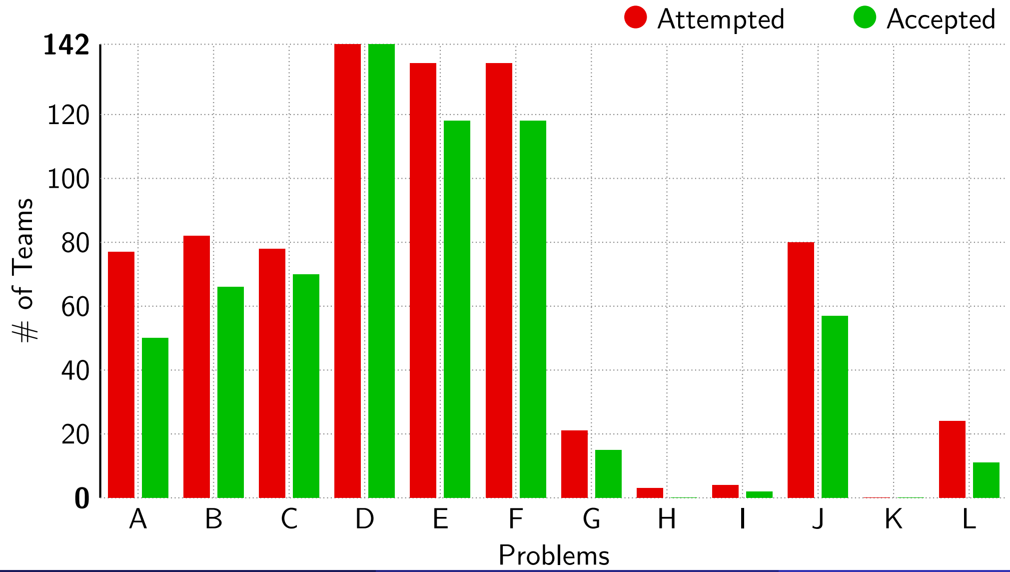
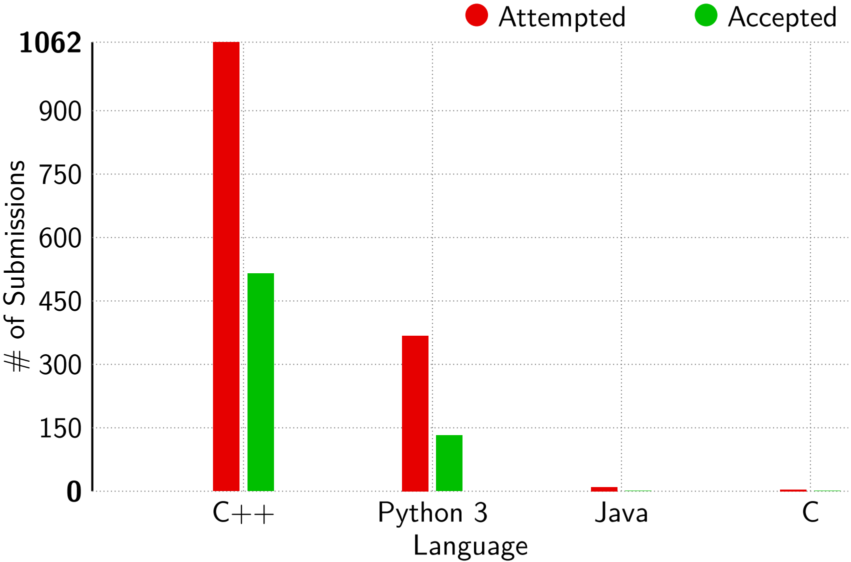
SWERC 2025

Problem Analysis

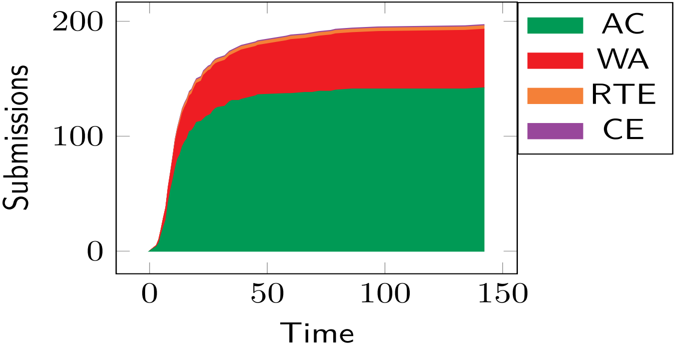
November 23, 2025


Attempted
Accepted
142
120
100
80
60
40
20
0
# of Teams
A
B
C
D
E
F
G
H
I
J
K
L
Problems


Attempted
Accepted
1062
900
750
600
450
300
150
0
# of Submissions
C++
Python 3
Java
C
Language

D: Dungeon Equilibrium

Solved by 142 teams before freeze.
First solved after 3 min by UXT.



D: Dungeon Equilibrium

Statement (summary).

We are given an array $a_1, \ldots, a_n$ $(0 \le a_i \le n)$. A level is *balanced* if, for every monster type x that appears, the total number of occurrences of x in the array is exactly x. We may delete elements. Compute the minimum number of deletions needed to make the level balanced.

D: Dungeon Equilibrium

Key observations.

- Each value x is independent of the others; only its own frequency matters.

D: Dungeon Equilibrium

Key observations.

- Each value x is independent of the others; only its own frequency matters.
- In a balanced array, value x must appear either 0 times or *exactly* x times.

D: Dungeon Equilibrium

Key observations.

- Each value x is independent of the others; only its own frequency matters.
- In a balanced array, value x must appear either 0 times or *exactly* x times.

Algorithm.

1. Count frequencies $f[x]$ for all $x \in [0, n]$.

D: Dungeon Equilibrium

Key observations.

- Each value x is independent of the others; only its own frequency matters.
- In a balanced array, value x must appear either 0 times or *exactly* x times.

Algorithm.

1. Count frequencies $f[x]$ for all $x \in [0, n]$.
2. For each x, add $f[x]$ to the answer if $f[x] < x$, else add $f[x] - x$.

D: Dungeon Equilibrium

Key observations.

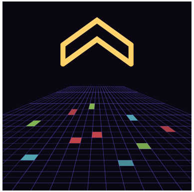
- Each value x is independent of the others; only its own frequency matters.
- In a balanced array, value x must appear either 0 times or *exactly* x times.

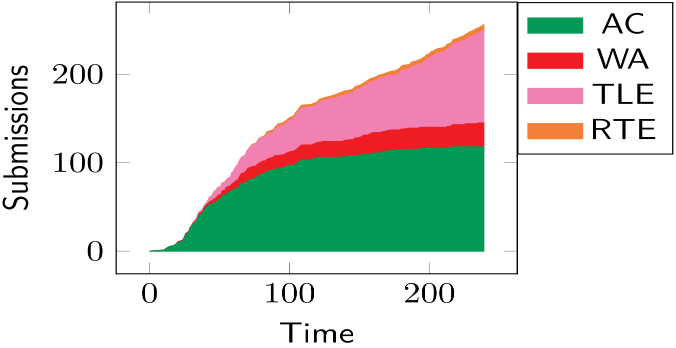
Algorithm.

1. Count frequencies $f[x]$ for all $x \in [0, n]$.
2. For each x, add $f[x]$ to the answer if $f[x] < x$, else add $f[x] - x$.

Complexity. $O(n)$ time (frequency array of size $n+1$). Slower solutions (e.g., $O(n^2)$) are allowed.

Solved by 118 teams before freeze.
First solved after 10 min by TempName.





Statement (summary).

Initially, only cell $(0, 0)$ is black. With a character 4, the 4 cells touching a black cell orthogonally become black. With a character 8, the 8 cells touching a black cell orthogonally or diagonally become black. Given a string s, answer queries (l, r, x, y): is cell (x, y) black after the commands in substring $[l, r]$?

E: Expansion Plan 2

Key observations.

- The order of operations does not matter. Only the number of 4s and 8s in the substring matters.

E: Expansion Plan 2

Key observations.

- The order of operations does not matter. Only the number of 4s and 8s in the substring matters.
- The black cells are the positions we can reach starting from $(0, 0)$, if we can move orthogonally (using a 4) and diagonally (using an 8).

E: Expansion Plan 2

Key observations.

- The order of operations does not matter. Only the number of 4s and 8s in the substring matters.
- The black cells are the positions we can reach starting from $(0, 0)$, if we can move orthogonally (using a 4) and diagonally (using an 8).
- A move with an 8 is equivalent to two moves with a 4, in different directions.

E: Expansion Plan 2

Key observations.

- The order of operations does not matter. Only the number of 4s and 8s in the substring matters.
- The black cells are the positions we can reach starting from $(0, 0)$, if we can move orthogonally (using a 4) and diagonally (using an 8).
- A move with an 8 is equivalent to two moves with a 4, in different directions.
- The grid is symmetrical. We can assume $x, y \geq 0$.

E: Expansion Plan 2

Key observations.

- The order of operations does not matter. Only the number of 4s and 8s in the substring matters.
- The black cells are the positions we can reach starting from $(0, 0)$, if we can move orthogonally (using a 4) and diagonally (using an 8).
- A move with an 8 is equivalent to two moves with a 4, in different directions.
- The grid is symmetrical. We can assume $x, y \geq 0$.

Algorithm.

1. Find the number of 4s and 8s, using prefix sums. Assume there are a 4s and b 8s.

E: Expansion Plan 2

Key observations.

- The order of operations does not matter. Only the number of 4s and 8s in the substring matters.
- The black cells are the positions we can reach starting from $(0,0)$, if we can move orthogonally (using a 4) and diagonally (using an 8).
- A move with an 8 is equivalent to two moves with a 4, in different directions.
- The grid is symmetrical. We can assume $x, y \geq 0$.

Algorithm.

1. Find the number of 4s and 8s, using prefix sums. Assume there are a 4s and b 8s.
2. Let $x := |x|$, $y := |y|$. The necessary and sufficient conditions are
   - $a + 2b \geq x + y$ (number of orthogonal steps);

E: Expansion Plan 2

Key observations.

- The order of operations does not matter. Only the number of 4s and 8s in the substring matters.
- The black cells are the positions we can reach starting from $(0, 0)$, if we can move orthogonally (using a 4) and diagonally (using an 8).
- A move with an 8 is equivalent to two moves with a 4, in different directions.
- The grid is symmetrical. We can assume $x, y \geq 0$.

Algorithm.

1. Find the number of 4s and 8s, using prefix sums. Assume there are a 4s and b 8s.
2. Let $x := |x|$, $y := |y|$. The necessary and sufficient conditions are
   - $a + 2b \geq x + y$ (number of orthogonal steps);
   - $a + b \geq \max(x, y)$ (number of orthogonal steps in the same direction).

E: Expansion Plan 2

Key observations.

- The order of operations does not matter. Only the number of 4s and 8s in the substring matters.
- The black cells are the positions we can reach starting from $(0, 0)$, if we can move orthogonally (using a 4) and diagonally (using an 8).
- A move with an 8 is equivalent to two moves with a 4, in different directions.
- The grid is symmetrical. We can assume $x, y \geq 0$.

Algorithm.

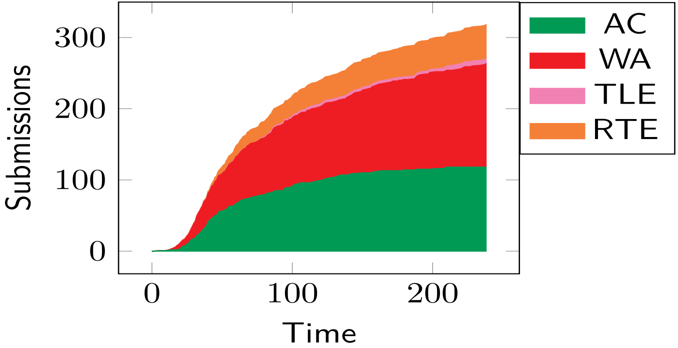
1. Find the number of 4s and 8s, using prefix sums. Assume there are a 4s and b 8s.
2. Let $x := |x|$, $y := |y|$. The necessary and sufficient conditions are
   - $a + 2b \geq x + y$ (number of orthogonal steps);
   - $a + b \geq \max(x, y)$ (number of orthogonal steps in the same direction).

Complexity. $O(n + q)$ time.

F: Factory Table

Solved by 118 teams before freeze.
First solved after 11 min by Segfault go BRRRR.

×	**1**	**2**	**3**	**4**
1	1	2	3	4
2	2	4	6	8
3	3	6	9	12
4	4	8	12	16



Statement (summary).

The k-th unrolled table is $[1 \cdot 1, 1 \cdot 2, \ldots, 1 \cdot k, 2 \cdot 1, 2 \cdot 2, \ldots, 2 \cdot k, \ldots, k \cdot 1, k \cdot 2, \ldots, k \cdot k]$. We are given a subarray of an unrolled table, with length ≥ 2. Find the minimum possible k.

Key observations.

- For each pair of adjacent elements in the subarray, we can determine their row.

F: Factory Table

Key observations.

- For each pair of adjacent elements in the subarray, we can determine their row.
  - If $x < y$, then they are in the same row, with index $y - x$.

F: Factory Table

Key observations.

- For each pair of adjacent elements in the subarray, we can determine their row.
  - If $x < y$, then they are in the same row, with index $y - x$.
  - If $x \geq y$, then y is in row y, and x is in row $y - 1$.

F: Factory Table

Key observations.

- For each pair of adjacent elements in the subarray, we can determine their row.
  - If $x < y$, then they are in the same row, with index $y - x$.
  - If $x \geq y$, then y is in row y, and x is in row $y - 1$.
- If element x is in row r, it is in column x/r.

F: Factory Table

Key observations.

- For each pair of adjacent elements in the subarray, we can determine their row.
    - If $x < y$, then they are in the same row, with index $y - x$.
    - If $x \geq y$, then y is in row y, and x is in row $y - 1$.
- If element x is in row r, it is in column x/r.

Algorithm.

1. For each element, find its row $r[i]$ and its column $c[i]$.

F: Factory Table

Key observations.

- For each pair of adjacent elements in the subarray, we can determine their row.
  - If $x < y$, then they are in the same row, with index $y - x$.
  - If $x \geq y$, then y is in row y, and x is in row $y - 1$.
- If element x is in row r, it is in column x/r.

Algorithm.

1. For each element, find its row $r[i]$ and its column $c[i]$.
2. Among all the $r[i]$ and $c[i]$, output the largest.

F: Factory Table

Key observations.

- For each pair of adjacent elements in the subarray, we can determine their row.
  - If $x < y$, then they are in the same row, with index $y - x$.
  - If $x \geq y$, then y is in row y, and x is in row $y - 1$.
- If element x is in row r, it is in column x/r.

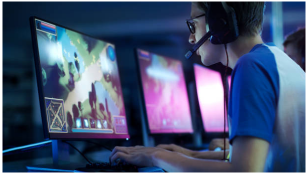
Algorithm.

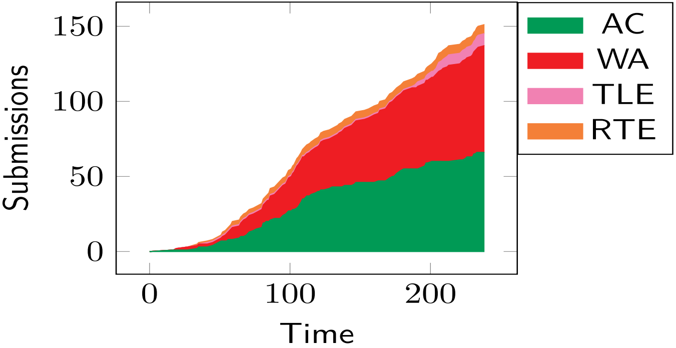
1. For each element, find its row $r[i]$ and its column $c[i]$.
2. Among all the $r[i]$ and $c[i]$, output the largest.

Complexity. $O(n)$ time per test case.

B: Billion Players Game

Solved by 66 teams before freeze.
First solved after 17 min by UniBois.





B: Billion Players Game

Statement (summary).

There is a hidden integer x in the interval $[l, r]$. We are given some offers $a_1, a_2, \ldots, a_n$. For each of them, we can choose to either get $x - a_i$ coins, $a_i - x$ coins, or nothing. Maximize the number of coins in the worst case.

B: Billion Players Game

Optimal strategy. Let's color a_i red if we get $x - a_i$ coins.

B: Billion Players Game

Optimal strategy. Let's color a_i red if we get $x - a_i$ coins.

Lemma

The worst case is either $x = l$ or $x = r$.

B: Billion Players Game

Optimal strategy. Let's color a_i red if we get $x - a_i$ coins.

Lemma

The worst case is either $x = l$ or $x = r$.

Sketch of proof. After fixing the strategy, the number of coins as a function of x is a straight line.

B: Billion Players Game

Optimal strategy. Let's color a_i red if we get $x - a_i$ coins.

Lemma

The worst case is either $x = l$ or $x = r$.

Sketch of proof. After fixing the strategy, the number of coins as a function of x is a straight line.

Lemma

There exists an optimal strategy where a prefix of elements (after sorting) is red, and a suffix of elements is blue.

B: Billion Players Game

Optimal strategy. Let's color a_i red if we get $x - a_i$ coins.

Lemma

The worst case is either $x = l$ or $x = r$.

Sketch of proof. After fixing the strategy, the number of coins as a function of x is a straight line.

Lemma

There exists an optimal strategy where a prefix of elements (after sorting) is red, and a suffix of elements is blue.

Sketch of proof. Use exchange argument. For example, if $y \leq z$, y is white and z is red, then we get $x - z$ coins. If we make y red and z white, we make $x - y$ coins instead.

B: Billion Players Game

Lemma

There exists an optimal strategy where (in addition to the previous properties) at most one element is white.

B: Billion Players Game

Lemma

There exists an optimal strategy where (in addition to the previous properties) at most one element is white.

Sketch of proof. If $x \leq y$, and they are both white, we can make x red and y blue instead.

B: Billion Players Game

Lemma

There exists an optimal strategy where (in addition to the previous properties) at most one element is white.

Sketch of proof. If $x \leq y$, and they are both white, we can make x red and y blue instead.

Algorithm.

- Try all possibilities for the white element. So there are $O(n)$ candidate strategies.

B: Billion Players Game

Lemma

There exists an optimal strategy where (in addition to the previous properties) at most one element is white.

Sketch of proof. If $x \leq y$, and they are both white, we can make x red and y blue instead.

Algorithm.

- Try all possibilities for the white element. So there are $O(n)$ candidate strategies.
- We can test a strategy in $O(1)$: find the number of coins as a function of x using prefix sums, and evaluate it in l and r to check the actual worst case.

B: Billion Players Game

Lemma

There exists an optimal strategy where (in addition to the previous properties) at most one element is white.

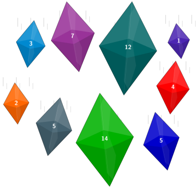
Sketch of proof. If $x \leq y$, and they are both white, we can make x red and y blue instead.

Algorithm.

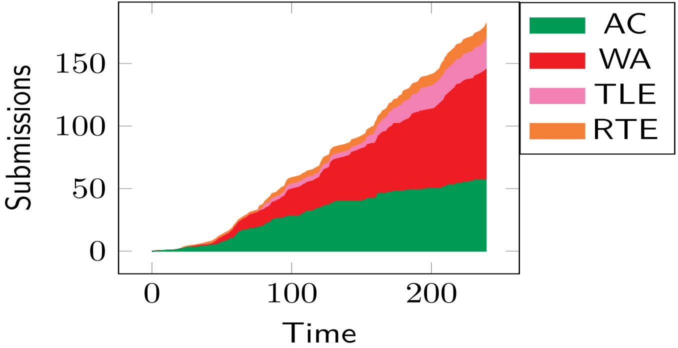
- Try all possibilities for the white element. So there are $O(n)$ candidate strategies.
- We can test a strategy in $O(1)$: find the number of coins as a function of x using prefix sums, and evaluate it in l and r to check the actual worst case.

Complexity. $O(n)$ time per test case ($O(n \log n)$ is also possible, for example using ternary search).

J: Jewels Building



Solved by 57 teams before freeze.
First solved after 17 min by TempName.



J: Jewels Building

Statement (summary).

We are given two arrays $a_1, a_2, \ldots, a_n$ and $b_1, b_2, \ldots, b_m$. Determine whether it is possible to turn a into b in a finite number of operations.
Operation: pick a subarray of length $x \geq 1$ containing equal elements, and replace it with x.

J: Jewels Building

More powerful operations. Let the operation in the statement be Operation 1.

J: Jewels Building

More powerful operations. Let the operation in the statement be Operation 1.

Operation 2

Operation 2: pick any subarray of length $x \geq 1$ (not necessarily containing equal elements), and replace it with x.

More powerful operations. Let the operation in the statement be Operation 1.

Operation 2

Operation 2: pick any subarray of length $x \geq 1$ (not necessarily containing equal elements), and replace it with x.
Can be done using operation 1: $[a_l, a_{l+1}, \ldots, a_r] \to [1, 1, \ldots, 1] \to x$.

J: Jewels Building

More powerful operations. Let the operation in the statement be Operation 1.

Operation 2

Operation 2: pick any subarray of length $x \geq 1$ (not necessarily containing equal elements), and replace it with x.
Can be done using operation 1: $[a_l, a_{l+1}, \ldots, a_r] \to [1, 1, \ldots, 1] \to x$.

Operation 3

Operation 3: pick any subarray of length $x \geq 1$ (not necessarily containing equal elements), and replace it with any $y \leq x$.

J: Jewels Building

More powerful operations. Let the operation in the statement be Operation 1.

Operation 2

Operation 2: pick any subarray of length $x \geq 1$ (not necessarily containing equal elements), and replace it with x.

Can be done using operation 1: $[a_l, a_{l+1}, \ldots, a_r] \to [1, 1, \ldots, 1] \to x$.

Operation 3

Operation 3: pick any subarray of length $x \geq 1$ (not necessarily containing equal elements), and replace it with any $y \leq x$.

Can be done using operation 2: $[a_l, a_{l+1}, \ldots, a_r] \to [a_l, a_{l+1}, \ldots, a_{l+y-2}, x - y + 1] \to [y]$.

J: Jewels Building

Reverse order. For every element y in the final array,

J: Jewels Building

Reverse order. For every element y in the final array,

- either it is one of the elements of the initial array (untouched),

J: Jewels Building

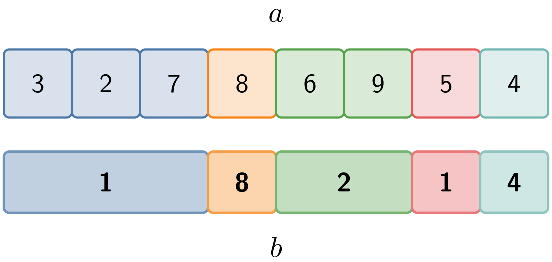
Reverse order. For every element y in the final array,

- either it is one of the elements of the initial array (untouched),
- or it comes from a subarray of the initial array, which must have length $x \geq y$.

J: Jewels Building

Reverse order. For every element y in the final array,

- either it is one of the elements of the initial array (untouched),
- or it comes from a subarray of the initial array, which must have length $x \geq y$.



J: Jewels Building

Algorithm.

- DP (similar to longest common subsequence).

Algorithm.

- DP (similar to longest common subsequence).
- Instead of replacing a subarray of length x with a single value $y \leq x$, we replace a prefix of length y with a single value y and we throw the rest of the subarray.

J: Jewels Building

Algorithm.

- DP (similar to longest common subsequence).
- Instead of replacing a subarray of length x with a single value $y \leq x$, we replace a prefix of length y with a single value y and we throw the rest of the subarray.
- $dp[i][j][k] =$ is it possible to match $a[1,i]$ with $b[1,j]$?

J: Jewels Building

Algorithm.

- DP (similar to longest common subsequence).
- Instead of replacing a subarray of length x with a single value $y \leq x$, we replace a prefix of length y with a single value y and we throw the rest of the subarray.
- $dp[i][j][k] =$ is it possible to match $a[1, i]$ with $b[1, j]$?
  - $k = 1$ when we have just replaced a subarray of length y with a single value y, so we are allowed to discard elements from a.

J: Jewels Building

Algorithm.

- DP (similar to longest common subsequence).
- Instead of replacing a subarray of length x with a single value $y \leq x$, we replace a prefix of length y with a single value y and we throw the rest of the subarray.
- $dp[i][j][k] =$ is it possible to match $a[1,i]$ with $b[1,j]$?
  - $k = 1$ when we have just replaced a subarray of length y with a single value y, so we are allowed to discard elements from a.
  - $k = 0$ otherwise.

J: Jewels Building

Algorithm.

- DP (similar to longest common subsequence).
- Instead of replacing a subarray of length x with a single value $y \leq x$, we replace a prefix of length y with a single value y and we throw the rest of the subarray.
- $dp[i][j][k] =$ is it possible to match $a[1, i]$ with $b[1, j]$?
  - $k = 1$ when we have just replaced a subarray of length y with a single value y, so we are allowed to discard elements from a.
  - $k = 0$ otherwise.
- The transitions require $O(1)$ time.

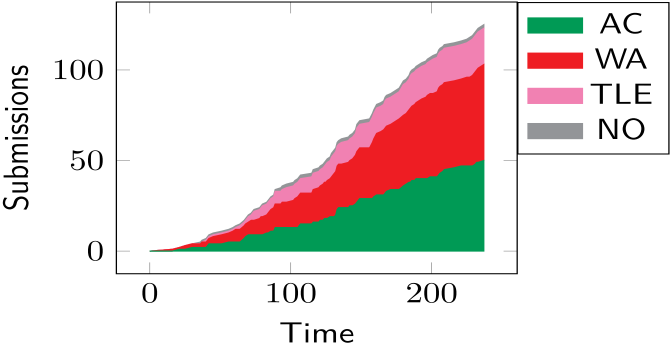
Algorithm.

- DP (similar to longest common subsequence).
- Instead of replacing a subarray of length x with a single value $y \leq x$, we replace a prefix of length y with a single value y and we throw the rest of the subarray.
- $dp[i][j][k] =$ is it possible to match $a[1, i]$ with $b[1, j]$?
  - $k = 1$ when we have just replaced a subarray of length y with a single value y, so we are allowed to discard elements from a.
  - $k = 0$ otherwise.
- The transitions require $O(1)$ time.

Complexity. $O(nm)$ time per test case.

A: Adjusting Drones

Solved by 50 teams before freeze.
First solved after 21 min by ENS de Lyon 3.



A: Adjusting Drones

Statement (summary).

Given an array, on each operation we increase simultaneously by 1 all the elements which are equal to some previous element. After how many operations every element appears at most k times?

A: Adjusting Drones

The order of the elements does not matter. So let's sort the elements in increasing order. Also, we have the freedom to choose the tiebreaker rule (i.e., which elements are increased by 1).

The order of the elements does not matter. So let's sort the elements in increasing order. Also, we have the freedom to choose the tiebreaker rule (i.e., which elements are increased by 1).

Fixed point.

Assume the process stops when all the elements are distinct. Let's find the configuration $b_1, b_2, \ldots, b_n$ at that point. If some elements are equal, the **rightmost** does not increase.

The order of the elements does not matter. So let's sort the elements in increasing order. Also, we have the freedom to choose the tiebreaker rule (i.e., which elements are increased by 1).

Fixed point.

Assume the process stops when all the elements are distinct. Let's find the configuration $b_1, b_2, \ldots, b_n$ at that point. If some elements are equal, the **rightmost** does not increase. From right to left, if the initial value is a_i, then b_i is the smallest $x \geq a_i$ not present in b yet.

A: Adjusting Drones

The order of the elements does not matter. So let's sort the elements in increasing order. Also, we have the freedom to choose the tiebreaker rule (i.e., which elements are increased by 1).

Fixed point.

Assume the process stops when all the elements are distinct. Let's find the configuration $b_1, b_2, \ldots, b_n$ at that point. If some elements are equal, the **rightmost** does not increase. From right to left, if the initial value is a_i, then b_i is the smallest $x \geq a_i$ not present in b yet. If we look at a single element over time, it has values $a_i, a_i + 1, \ldots, b_i, b_i, \ldots, b_i$.

A: Adjusting Drones

t	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
0	3	3	3	4	5	5	6	6	6	6	6	7	7	7	13	13
1	4	4	3	4	6	5	7	7	7	7	6	8	8	7	14	13
2	5	5	3	4	7	5	8	8	8	8	6	9	8	7	14	13
3	6	6	3	4	8	5	9	9	9	9	6	9	8	7	14	13
4	7	7	3	4	9	5	10	10	10	10	6	9	8	7	14	13
5	8	8	3	4	10	5	11	11	11	10	6	9	8	7	14	13
6	9	9	3	4	11	5	12	12	11	10	6	9	8	7	14	13
7	10	10	3	4	12	5	13	12	11	10	6	9	8	7	14	13
8	11	11	3	4	13	5	14	12	11	10	6	9	8	7	14	13
9	12	12	3	4	14	5	15	12	11	10	6	9	8	7	14	13
10	13	13	3	4	15	5	15	12	11	10	6	9	8	7	14	13
11	14	14	3	4	16	5	15	12	11	10	6	9	8	7	14	13
12	15	15	3	4	16	5	15	12	11	10	6	9	8	7	14	13
13	16	16	3	4	16	5	15	12	11	10	6	9	8	7	14	13
14	17	17	3	4	16	5	15	12	11	10	6	9	8	7	14	13
15	18	17	3	4	16	5	15	12	11	10	6	9	8	7	14	13

Binary search.

- The maximum number of occurences of one element does not increase after one operation (if value x appeared $\leq k$ times, it produces $\leq k-1$ copies of $x+1$, and at most one other copy of $x+1$ survives).

A: Adjusting Drones

Binary search.

- The maximum number of occurences of one element does not increase after one operation (if value x appeared $\leq k$ times, it produces $\leq k-1$ copies of $x+1$, and at most one other copy of $x+1$ survives).
- So we can binary search the answer. Using the process described earlier, we can retrieve the configuration after m moves in $O(n)$.

A: Adjusting Drones

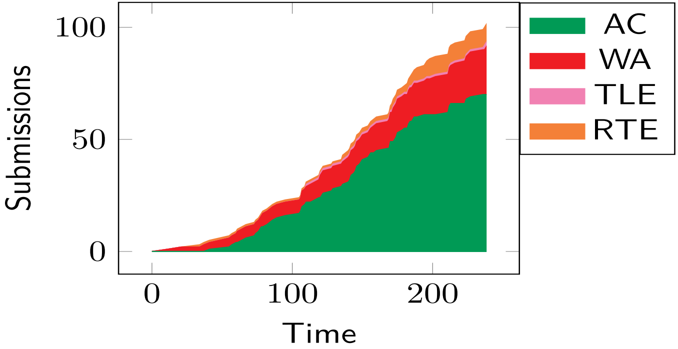
Binary search.

- The maximum number of occurences of one element does not increase after one operation (if value x appeared $\leq k$ times, it produces $\leq k-1$ copies of $x+1$, and at most one other copy of $x+1$ survives).
- So we can binary search the answer. Using the process described earlier, we can retrieve the configuration after m moves in $O(n)$.

Complexity. $O(n \log n)$ time (there also exist “magical” $O(n)$ solutions!)

C: Chamber of Secrets 2

Solved by 70 teams before freeze.
First solved after 41 min by Ctrl+Alt+DelETHe.



Statement (summary).

We have n arrays of length m built as follows:

1. start from a secret key $[p_1, p_2, \ldots, p_{nm/2}]$ (a permutation);
2. concatenate p with itself;
3. split the resulting array into n consecutive blocks, i.e., disjoint subarrays of length m;
4. shuffle these arrays.

Find a possible secret key $[p_1, p_2, \ldots, p_{nm/2}]$.

Case 1: n is even.

- The subarrays in the first half correspond exactly to the subarrays in the second half.

C: Chamber of Secrets 2

Case 1: n is even.

- The subarrays in the first half correspond exactly to the subarrays in the second half.
- Each subarray appears twice. We can rearrange and print them in any order.

C: Chamber of Secrets 2

Case 1: n is even.

- The subarrays in the first half correspond exactly to the subarrays in the second half.
- Each subarray appears twice. We can rearrange and print them in any order.

5	6	3	4	1	2	5	6	3	4	1	2

Case 2: n is odd.

- Split each subarray into two halves.

Case 2: n is odd.

- Split each subarray into two halves.
- We have constraints of type "the first half must be followed by the second half". These constraints make a cycle.

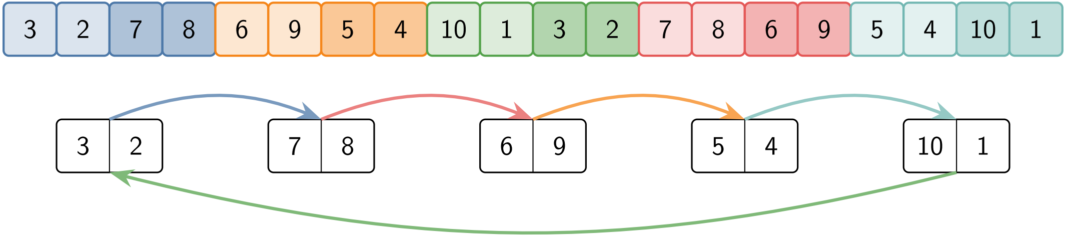
C: Chamber of Secrets 2

Case 2: n is odd.

- Split each subarray into two halves.
- We have constraints of type “the first half must be followed by the second half”. These constraints make a cycle.
- Visit the subarrays in the cycle, starting from any of them.

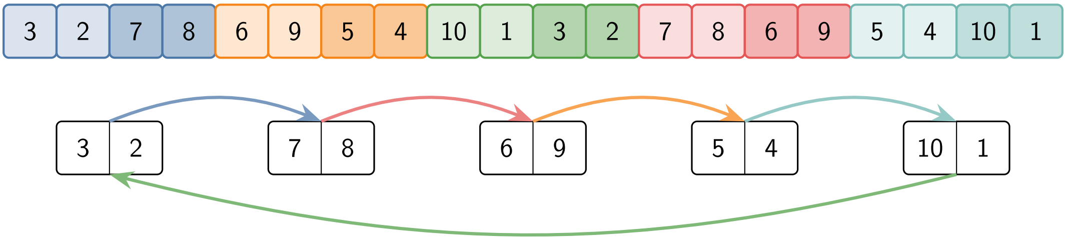
Case 2: n is odd.

- Split each subarray into two halves.
- We have constraints of type "the first half must be followed by the second half". These constraints make a cycle.
- Visit the subarrays in the cycle, starting from any of them.



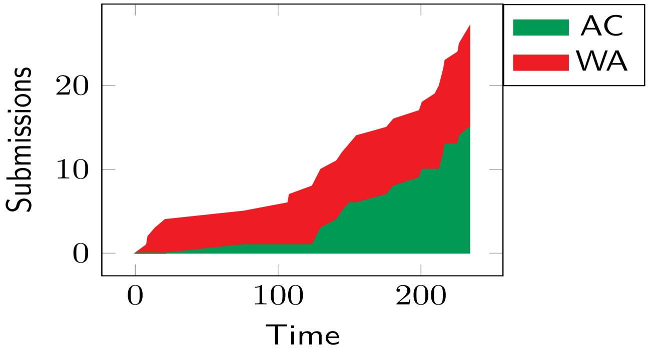
Case 2: n is odd.

- Split each subarray into two halves.
- We have constraints of type "the first half must be followed by the second half". These constraints make a cycle.
- Visit the subarrays in the cycle, starting from any of them.



Complexity. $O(nm)$ time per test case.

Solved by 15 teams before freeze.
First solved after 76 min by Team 2.



G: Git Gud

Statement (summary).

We want to make our skill $\geq n$ using a limited amount of robocoins. We can perform operations (y, l) with cost l and additional cost 1000 if y is larger than the previous one. The operation makes our skill $y + l$ if it was exactly y.

G: Git Gud

Checker. As a warm-up, think about how we would write the checker for this problem.

G: Git Gud

Checker. As a warm-up, think about how we would write the checker for this problem. We can store the set of all the possible skills (initially, all integers in $[1, n]$), and perform operations in $O(1)$ (we have to remove y and insert $y + l$).

Checker. As a warm-up, think about how we would write the checker for this problem.
We can store the set of all the possible skills (initially, all integers in $[1, n]$), and perform operations in $O(1)$ (we have to remove y and insert $y + l$).
As a consequence, the optimal value of l is the smallest one such that $y + l$ is already in the set.
Since we know how to determine the optimal l for a fixed y, let's denote the operations only using y.

Checker. As a warm-up, think about how we would write the checker for this problem.
We can store the set of all the possible skills (initially, all integers in $[1, n]$), and perform operations in $O(1)$ (we have to remove y and insert $y + l$).
As a consequence, the optimal value of l is the smallest one such that $y + l$ is already in the set.
Since we know how to determine the optimal l for a fixed y, let's denote the operations only using y.

Naive strategy 1.

Use operations $1, 2, \ldots, n - 1$. Unfortunately, they are too "increasing".

G: Git Gud

Checker. As a warm-up, think about how we would write the checker for this problem.
We can store the set of all the possible skills (initially, all integers in $[1, n]$), and perform operations in $O(1)$ (we have to remove y and insert $y + l$).
As a consequence, the optimal value of l is the smallest one such that $y + l$ is already in the set.
Since we know how to determine the optimal l for a fixed y, let's denote the operations only using y.

Naive strategy 1.

Use operations $1, 2, \ldots, n - 1$. Unfortunately, they are too "increasing".

Naive strategy 2.

Use operations $n - 1, n - 2, \ldots, 1$. Unfortunately, they are too long.

G: Git Gud

Better strategy.

- Fix our skill modulo 2 using operations $n-1, n-3, n-5, \ldots$

G: Git Gud

Better strategy.

- Fix our skill modulo 2 using operations $n-1, n-3, n-5, \ldots$
- Fix our skill modulo 4 using operations $n-2, n-6, n-10, \ldots$

G: Git Gud

Better strategy.

- Fix our skill modulo 2 using operations $n-1, n-3, n-5, \ldots$
- Fix our skill modulo 4 using operations $n-2, n-6, n-10, \ldots$
- $\ldots$

G: Git Gud

Better strategy.

- Fix our skill modulo 2 using operations $n-1, n-3, n-5, \ldots$
- Fix our skill modulo 4 using operations $n-2, n-6, n-10, \ldots$
- $\ldots$

The cost is $O(n \log n)$ ($\approx \log_2 n$ layers, $O(n)$ per layer). We use very few “increasing” operations (1 per layer): we can try to use more (and maybe use less layers).

G: Git Gud

Better strategy.

- Fix our skill modulo 2 using operations $n-1, n-3, n-5, \dots$
- Fix our skill modulo 4 using operations $n-2, n-6, n-10, \dots$
- $\dots$

The cost is $O(n \log n)$ ($\approx \log_2 n$ layers, $O(n)$ per layer). We use very few "increasing" operations (1 per layer): we can try to use more (and maybe use less layers).

Generalization.

- Fix our skill modulo 3 using operations $n-2, n-5, n-8, \dots, n-1, n-4, n-7, \dots$

G: Git Gud

Better strategy.

- Fix our skill modulo 2 using operations $n-1, n-3, n-5, \dots$
- Fix our skill modulo 4 using operations $n-2, n-6, n-10, \dots$
- $\dots$

The cost is $O(n \log n)$ ($\approx \log_2 n$ layers, $O(n)$ per layer). We use very few "increasing" operations (1 per layer): we can try to use more (and maybe use less layers).

Generalization.

- Fix our skill modulo 3 using operations $n-2, n-5, n-8, \dots, n-1, n-4, n-7, \dots$
- Fix our skill modulo 9 using operations $n-6, n-18, n-30, \dots, n-3, n-12, n-21, \dots$

G: Git Gud

Better strategy.

- Fix our skill modulo 2 using operations $n-1, n-3, n-5, \dots$
- Fix our skill modulo 4 using operations $n-2, n-6, n-10, \dots$
- $\dots$

The cost is $O(n \log n)$ ($\approx \log_2 n$ layers, $O(n)$ per layer). We use very few "increasing" operations (1 per layer): we can try to use more (and maybe use less layers).

Generalization.

- Fix our skill modulo 3 using operations $n-2, n-5, n-8, \dots, n-1, n-4, n-7, \dots$
- Fix our skill modulo 9 using operations $n-6, n-18, n-30, \dots, n-3, n-12, n-21, \dots$
- $\dots$

G: Git Gud

Better strategy.

- Fix our skill modulo 2 using operations $n-1, n-3, n-5, \dots$
- Fix our skill modulo 4 using operations $n-2, n-6, n-10, \dots$
- $\dots$

The cost is $O(n \log n)$ ($\approx \log_2 n$ layers, $O(n)$ per layer). We use very few "increasing" operations (1 per layer): we can try to use more (and maybe use less layers).

Generalization.

- Fix our skill modulo 3 using operations $n-2, n-5, n-8, \dots, n-1, n-4, n-7, \dots$
- Fix our skill modulo 9 using operations $n-6, n-18, n-30, \dots, n-3, n-12, n-21, \dots$
- $\dots$

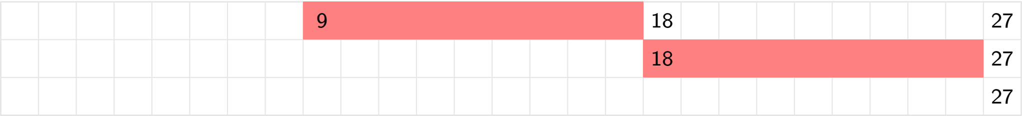
Now we have $\approx \log_3 n$ layers and 2 "increasing" operations per layer.

G: Git Gud

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24		26	27
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21		23	24		26	27
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18		20	21		23	24		26	27
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15		17	18		20	21		23	24		26	27
1	2	3	4	5	6	7	8	9	10	11	12		14	15		17	18		20	21		23	24		26	27
1	2	3	4	5	6	7	8	9		11	12		14	15		17	18		20	21		23	24		26	27
1	2	3	4	5	6		8	9		11	12		14	15		17	18		20	21		23	24		26	27
1	2	3		5	6		8	9		11	12		14	15		17	18		20	21		23	24		26	27
	2	3		5	6		8	9		11	12		14	15		17	18		20	21		23	24		26	27
	2	3		5	6		8	9		11	12		14	15		17	18		20	21		23	24			27
	2	3		5	6		8	9		11	12		14	15		17	18		20	21			24			27
	2	3		5	6		8	9		11	12		14	15		17	18			21			24			27
	2	3		5	6		8	9		11	12		14	15			18			21			24			27
	2	3		5	6		8	9		11	12			15			18			21			24			27
	2	3		5	6		8	9			12			15			18			21			24			27
	2	3		5	6			9			12			15			18			21			24			27
	2	3			6			9			12			15			18			21			24			27
		3			6			9			12			15			18			21			24			27

		3			6			9			12			15			18			21			24			27
		3			6			9			12			15			18						24			27
		3			6			9						15			18						24			27
					6			9						15			18						24			27
					6			9						15			18									27
					6			9									18									27
								9									18									27

G: Git Gud



G: Git Gud

Generalization.

If we use modulo m, we have $\approx \log_m n$ layers, each with cost $\approx n$, and $m-1$ "increasing" operations per layer, each with cost 1000.

G: Git Gud

Generalization.

If we use modulo m, we have $\approx \log_m n$ layers, each with cost $\approx n$, and $m-1$ "increasing" operations per layer, each with cost 1000.

We should choose $m \approx \sqrt[3]{n}$. So the total cost is around

G: Git Gud

Generalization.

If we use modulo m, we have $\approx \log_m n$ layers, each with cost $\approx n$, and $m - 1$ "increasing" operations per layer, each with cost 1000.
We should choose $m \approx \sqrt[3]{n}$. So the total cost is around

$$(\log_m n)(n + 1000m) = 3(n + 1000\sqrt[3]{n}) < 4n$$

if $n = 250\,000$.

G: Git Gud

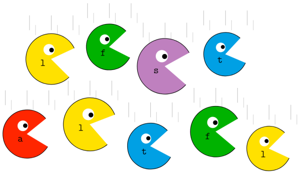
Generalization.

If we use modulo m, we have $\approx \log_m n$ layers, each with cost $\approx n$, and $m-1$ "increasing" operations per layer, each with cost 1000.
We should choose $m \approx \sqrt[3]{n}$. So the total cost is around

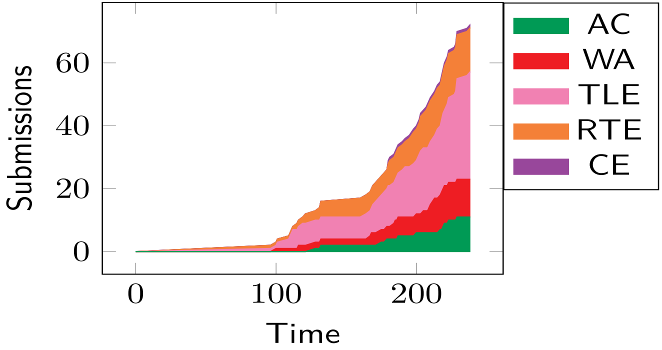
$$(\log_m n)(n + 1000m) = 3(n + 1000\sqrt[3]{n}) < 4n$$

if $n = 250\,000$.
If n is smaller, we can pretend it is $250\,000$.

L: LFS



Solved by 11 teams before freeze.
First solved after 128 min by Northern Spy.



L: LFS

Statement (summary).

Given a string s, answer q queries: what is the maximum length of a substring appearing with maximum frequency in the substring $[l, r]$?

L: LFS

Key observations. Let $\alpha = 26$ (size of the alphabet).

L: LFS

Key observations. Let $\alpha = 26$ (size of the alphabet).

- The maximum frequency is reached by a substring which is 1 character long.

L: LFS

Key observations. Let $\alpha = 26$ (size of the alphabet).

- The maximum frequency is reached by a substring which is 1 character long.
- A substring which ends with a character c can be extended to the right if and only if $s[r] \neq c$, and every occurrence of c in $[l, r]$ is followed by the same character d.

L: LFS

Key observations. Let $\alpha = 26$ (size of the alphabet).

- The maximum frequency is reached by a substring which is 1 character long.
- A substring which ends with a character c can be extended to the right if and only if $s[r] \neq c$, and every occurrence of c in $[l, r]$ is followed by the same character d.
- Make an edge for every such pair (c, d) of characters. There are $O(\alpha)$ such pairs (for each c, there exists at most one valid d).

Algorithm.

- Switch point from some position i: next position j such that the following character is different (i.e., $s[i] = s[j]$ but $s[i+1] \neq s[j+1]$).

Algorithm.

- Switch point from some position i: next position j such that the following character is different (i.e., $s[i] = s[j]$ but $s[i+1] \neq s[j+1]$).
- For each character, create prefix sums (to count the number of occurrences), an array to find the next occurrence, and an array to find the switch point.

Algorithm.

- Switch point from some position i: next position j such that the following character is different (i.e., $s[i] = s[j]$ but $s[i+1] \neq s[j+1]$).
- For each character, create prefix sums (to count the number of occurrences), an array to find the next occurrence, and an array to find the switch point.
- For each candidate edge, check whether the switch point is $> r$.

L: LFS

Algorithm.

- Switch point from some position i: next position j such that the following character is different (i.e., $s[i] = s[j]$ but $s[i+1] \neq s[j+1]$).
- For each character, create prefix sums (to count the number of occurrences), an array to find the next occurrence, and an array to find the switch point.
- For each candidate edge, check whether the switch point is $> r$.
- Find the largest connected component.

L: LFS

Algorithm.

- Switch point from some position i: next position j such that the following character is different (i.e., $s[i] = s[j]$ but $s[i+1] \neq s[j+1]$).
- For each character, create prefix sums (to count the number of occurrences), an array to find the next occurrence, and an array to find the switch point.
- For each candidate edge, check whether the switch point is $> r$.
- Find the largest connected component.
- If we process the queries offline (increasing r), the implementation might become easier.

L: LFS

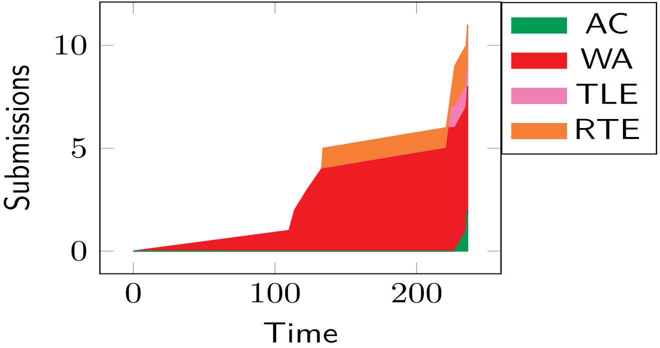
Algorithm.

- Switch point from some position i: next position j such that the following character is different (i.e., $s[i] = s[j]$ but $s[i+1] \neq s[j+1]$).
- For each character, create prefix sums (to count the number of occurrences), an array to find the next occurrence, and an array to find the switch point.
- For each candidate edge, check whether the switch point is $> r$.
- Find the largest connected component.
- If we process the queries offline (increasing r), the implementation might become easier.

Complexity. $O((n+q)\alpha)$ time (additional $O(\log n)$ factors with low constant are allowed).

I: Isaac's Queries

Solved by 2 teams before freeze.
First solved after 235 min by Segfault go BRRRR.



Statement (summary).

There is a hidden array, generated randomly. For each subarray $[l, r]$, we want to find the most significant bit of $a_l \oplus a_{l+1} \oplus \ldots \oplus a_r$. We can ask the answer for some $[l, r]$, with cost $\frac{1}{r-l+1}$. Find all the answers with limited total cost.

I: Isaac's Queries

Let $g(l, r)$ be the answer for the subarray $[l, r]$.

Let $g(l, r)$ be the answer for the subarray $[l, r]$.

Saving queries (I).

If $g(l, m) \neq g(m+1, r)$, then $g(l, r) = \max(g(l, m), g(m+1, r))$. Otherwise, we know that $g(l, r) < g(l, m)$.

I: Isaac's Queries

Let $g(l, r)$ be the answer for the subarray $[l, r]$.

Saving queries (I).

If $g(l, m) \neq g(m+1, r)$, then $g(l, r) = \max(g(l, m), g(m+1, r))$. Otherwise, we know that $g(l, r) < g(l, m)$.

Saving queries (II).

We can generalize for m outside $[l, r]$.

I: Isaac's Queries

Let $g(l, r)$ be the answer for the subarray $[l, r]$.

Saving queries (I).

If $g(l, m) \neq g(m+1, r)$, then $g(l, r) = \max(g(l, m), g(m+1, r))$. Otherwise, we know that $g(l, r) < g(l, m)$.

Saving queries (II).

We can generalize for m outside $[l, r]$.
If $x < l$ and $g(x, l-1) \neq g(x, r)$, then $g(l, r) = \max(g(x, l-1), g(x, r))$.

I: Isaac's Queries

Let $g(l,r)$ be the answer for the subarray $[l,r]$.

Saving queries (I).

If $g(l,m) \neq g(m+1,r)$, then $g(l,r) = \max(g(l,m), g(m+1,r))$. Otherwise, we know that $g(l,r) < g(l,m)$.

Saving queries (II).

We can generalize for m outside $[l,r]$.
If $x < l$ and $g(x,l-1) \neq g(x,r)$, then $g(l,r) = \max(g(x,l-1), g(x,r))$.
If $y > l$ and $g(l,y) \neq g(r+1,y)$, then $g(l,r) = \max(g(l,y), g(r+1,y))$.

I: Isaac's Queries

Let $g(l, r)$ be the answer for the subarray $[l, r]$.

Let $g(l, r)$ be the answer for the subarray $[l, r]$.

Solution 1.

For r from 1 to n, for l from 1 to i, find $g(l, r)$ using the previous queries if possible, otherwise ask it.

I: Isaac's Queries

Let $g(l, r)$ be the answer for the subarray $[l, r]$.

Solution 1.

For r from 1 to n, for l from 1 to i, find $g(l, r)$ using the previous queries if possible, otherwise ask it.

Cost of Solution 1.

In order to ask $g(l, r)$, it must be less than $g(1, r), g(2, r), \ldots, g(l-1, r)$. So the probability of asking it is $O(1/l)$. So the average cost is

Let $g(l, r)$ be the answer for the subarray $[l, r]$.

Solution 1.

For r from 1 to n, for l from 1 to i, find $g(l, r)$ using the previous queries if possible, otherwise ask it.

Cost of Solution 1.

In order to ask $g(l, r)$, it must be less than $g(1, r), g(2, r), \ldots, g(l-1, r)$. So the probability of asking it is $O(1/l)$. So the average cost is

$$O\left(\sum_{1\leq l\leq r\leq n} \frac{1}{l(r-l+1)}\right) = O\left(\left(\sum_{1\leq k\leq n} \frac{1}{k}\right)^2\right) = O(\log^2 n)$$

Let $g(l, r)$ be the answer for the subarray $[l, r]$.

Solution 1.

For r from 1 to n, for l from 1 to i, find $g(l, r)$ using the previous queries if possible, otherwise ask it.

Cost of Solution 1.

In order to ask $g(l, r)$, it must be less than $g(1, r), g(2, r), \ldots, g(l-1, r)$. So the probability of asking it is $O(1/l)$. So the average cost is

$$O\left(\sum_{1 \leq l \leq r \leq n} \frac{1}{l(r-l+1)}\right) = O\left(\left(\sum_{1 \leq k \leq n} \frac{1}{k}\right)^2\right) = O(\log^2 n)$$

In practice, the average cost is ≈ 20.

I: Isaac's Queries

In the following solution, we apply prefix XOR. So instead of $a_l \oplus a_{l+1} \oplus \ldots \oplus a_r$ we are interested in $a_l \oplus a_r$.

I: Isaac's Queries

In the following solution, we apply prefix XOR. So instead of $a_l \oplus a_{l+1} \oplus \ldots \oplus a_r$ we are interested in $a_l \oplus a_r$.

Solution 2.

Let's build a trie containing the a_i. Specifically, each node corresponds to a prefix of the binary representation, and contains positions i such that a_i has that prefix. It is not possible to recover the a_i uniquely, but we can set bits to 0 without loss of generality if they do not impact any queries.

I: Isaac's Queries

In the following solution, we apply prefix XOR. So instead of $a_l \oplus a_{l+1} \oplus \ldots \oplus a_r$ we are interested in $a_l \oplus a_r$.

Solution 2.

Let's build a trie containing the a_i. Specifically, each node corresponds to a prefix of the binary representation, and contains positions i such that a_i has that prefix. It is not possible to recover the a_i uniquely, but we can set bits to 0 without loss of generality if they do not impact any queries.
Then, in order to calculate the answer for some (l, r), we have to find the LCA of the corresponding nodes.

I: Isaac's Queries

In the following solution, we apply prefix XOR. So instead of $a_l \oplus a_{l+1} \oplus \ldots \oplus a_r$ we are interested in $a_l \oplus a_r$.

Solution 2.

Let's build a trie containing the a_i. Specifically, each node corresponds to a prefix of the binary representation, and contains positions i such that a_i has that prefix. It is not possible to recover the a_i uniquely, but we can set bits to 0 without loss of generality if they do not impact any queries.
Then, in order to calculate the answer for some (l, r), we have to find the LCA of the corresponding nodes.
Suppose we are in some node of the trie containing k values. We want to recurse into two children, depending on the next bit. This is possible by asking the query.

I: Isaac's Queries

In the following solution, we apply prefix XOR. So instead of $a_l \oplus a_{l+1} \oplus \ldots \oplus a_r$ we are interested in $a_l \oplus a_r$.

Solution 2.

Let's build a trie containing the a_i. Specifically, each node corresponds to a prefix of the binary representation, and contains positions i such that a_i has that prefix. It is not possible to recover the a_i uniquely, but we can set bits to 0 without loss of generality if they do not impact any queries.

Then, in order to calculate the answer for some (l, r), we have to find the LCA of the corresponding nodes.

Suppose we are in some node of the trie containing k values. We want to recurse into two children, depending on the next bit. This is possible by asking the query.

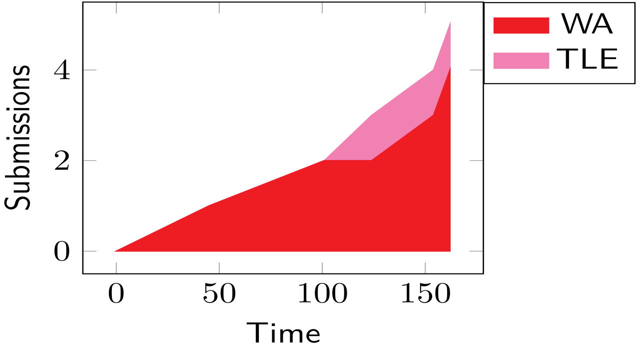
Specifically, suppose that a node contains positions $p_1, p_2, \ldots, p_k$. In order to recurse, we need to ask queries between some pairs. We choose this pairs greedily, by calculating a minimum spanning tree (the edges have cost $\frac{1}{p_j - p_i + 1}$).

Cost of Solution 2.

The same analysis of Solution 1 gives an upper bound. However, this solution is much more efficient (average cost ≈ 9).

H: Hyper Smawk Bros

Not solved before freeze.



Statement (summary).

You and Bob want to make an integer $n \leq 0$. You alternately subtract an integer in $[1, m]$ from n, but it must be different from the one just chosen by the opponent. Given n, m, determine whether you can force a win. Answer multiple test cases.

H: Hyper Smawk Bros

Let the starting number be N.

H: Hyper Smawk Bros

Let the starting number be N.

Losing positions.

Let (n, l) $(0 \leq m)$ be losing if you lose if it's your turn, the current integer is n, and you cannot subtract l (if l is 0, you can subtract everything).

H: Hyper Smawk Bros

Let the starting number be N.

Losing positions.

Let (n, l) $(0 \leq m)$ be losing if you lose if it's your turn, the current integer is n, and you cannot subtract l (if l is 0, you can subtract everything).

Evil n.

Let n be evil if $(n, 0)$ is losing (i.e., you lose even without constraints on the number you subtract).

H: Hyper Smawk Bros

Let the starting number be N.

Losing positions.

Let (n, l) $(0 \le m)$ be losing if you lose if it's your turn, the current integer is n, and you cannot subtract l (if l is 0, you can subtract everything).

Evil n.

Let n be evil if $(n, 0)$ is losing (i.e., you lose even without constraints on the number you subtract).

Bound on the evil n (I).

Claim: there are $O(N/m)$ evil n. Specifically, let $v_1, v_2, \ldots, v_k$ be the evil $n \le N$. Then, $v_i - v_{i-1} \ge m + 1$ (otherwise you would be able to move from an evil n to another, contradiction).

H: Hyper Smawk Bros

Structure of losing positions and solution in $O(t(N+m))$.

If (n, l) is losing, then $(n+l, l')$ with $l' \neq l$ is winning. This means that the number of losing l for a fixed n is either 0, 1, or $m+1$. Let's call n "good", "neutral", and "evil", respectively.

H: Hyper Smawk Bros

Structure of losing positions and solution in $O(t(N+m))$.

If (n, l) is losing, then $(n+l, l')$ with $l' \neq l$ is winning. This means that the number of losing l for a fixed n is either 0, 1, or $m+1$. Let's call n "good", "neutral", and "evil", respectively. Note that the losing states are $O(N+m)$ in total.

H: Hyper Smawk Bros

Structure of losing positions and solution in $O(t(N + m))$.

If (n, l) is losing, then $(n + l, l')$ with $l' \neq l$ is winning. This means that the number of losing l for a fixed n is either 0, 1, or $m + 1$. Let's call n "good", "neutral", and "evil", respectively. Note that the losing states are $O(N + m)$ in total.

Even m.

For even m, only the multiples of $m + 1$ are evil. In order to win, you can ensure that the last two moves have sum $m + 1$. Since $m + 1$ is odd, the last move will never be the same as the second last.

Structure of losing positions and solution in $O(t(N+m))$.

If (n, l) is losing, then $(n+l, l')$ with $l' \neq l$ is winning. This means that the number of losing l for a fixed n is either 0, 1, or $m+1$. Let's call n "good", "neutral", and "evil", respectively. Note that the losing states are $O(N+m)$ in total.

Even m.

For even m, only the multiples of $m+1$ are evil. In order to win, you can ensure that the last two moves have sum $m+1$. Since $m+1$ is odd, the last move will never be the same as the second last.

From now, let's only consider odd m.

More structure.

There are three types of losing states.

More structure.

There are three types of losing states.

- (n, l) with $f(n) = 0$.

H: Hyper Smawk Bros

More structure.

There are three types of losing states.

- (n, l) with $f(n) = 0$.
- $(n, f(n))$.

H: Hyper Smawk Bros

More structure.

There are three types of losing states.

- (n, l) with $f(n) = 0$.
- $(n, f(n))$.
- $(n, f(n)/2)$ with $f(n) = m + 1$.

More structure.

There are three types of losing states.

- (n, l) with $f(n) = 0$.
- $(n, f(n))$.
- $(n, f(n)/2)$ with $f(n) = m + 1$.

This is a consequence of the slow solution described above. If s is evil:

More structure.

There are three types of losing states.

- (n, l) with $f(n) = 0$.
- $(n, f(n))$.
- $(n, f(n)/2)$ with $f(n) = m + 1$.

This is a consequence of the slow solution described above. If s is evil:

- the losing states (s, l) propagate to n in $[s + 1, s + m]$;

H: Hyper Smawk Bros

More structure.

There are three types of losing states.

- (n, l) with $f(n) = 0$.
- $(n, f(n))$.
- $(n, f(n)/2)$ with $f(n) = m + 1$.

This is a consequence of the slow solution described above. If s is evil:

- the losing states (s, l) propagate to n in $[s + 1, s + m]$;
- only the losing state $(s + \frac{m+1}{2}, \frac{m+1}{2})$ can propagate to $(s + m + 1, \frac{m+1}{2})$, and in that case $s + m + 2$ is evil.

More structure.

There are three types of losing states.

- (n, l) with $f(n) = 0$.
- $(n, f(n))$.
- $(n, f(n)/2)$ with $f(n) = m + 1$.

This is a consequence of the slow solution described above. If s is evil:

- the losing states (s, l) propagate to n in $[s + 1, s + m]$;
- only the losing state $(s + \frac{m+1}{2}, \frac{m+1}{2})$ can propagate to $(s + m + 1, \frac{m+1}{2})$, and in that case $s + m + 2$ is evil.

So $s + m + 1$ cannot be good (it is either neutral or evil).

H: Hyper Smawk Bros

More structure.

There are three types of losing states.

- (n, l) with $f(n) = 0$.
- $(n, f(n))$.
- $(n, f(n)/2)$ with $f(n) = m + 1$.

This is a consequence of the slow solution described above. If s is evil:

- the losing states (s, l) propagate to n in $[s + 1, s + m]$;
- only the losing state $(s + \frac{m+1}{2}, \frac{m+1}{2})$ can propagate to $(s + m + 1, \frac{m+1}{2})$, and in that case $s + m + 2$ is evil.

So $s + m + 1$ cannot be good (it is either neutral or evil).

Bound on the evil n (II).

Let $v_1, v_2, \ldots, v_k$ be the evil $n \leq N$. Then, $m + 1 \leq v_i - v_{i-1} \leq m + 2$.

H: Hyper Smawk Bros

n	losing last moves l (if $m = 9$)
21	[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
22	[1]
23	[]
24	[]
25	[]
26	[5]
27	[6]
28	[]
29	[8]
30	[9]
31	[5]
32	[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
33	[]
34	[2]
35	[3]

H: Hyper Smawk Bros

Our goal.

Let $v_1, v_2, \ldots, v_k$ be the evil $n \leq N$. We want to find them in $O(k) = O(N/m)$. In this way, we can solve the problem for all m in $O(\sum_{1 \leq m \leq M} N/m) = O(N \log M)$.

Our goal.

Let $v_1, v_2, \ldots, v_k$ be the evil $n \leq N$. We want to find them in $O(k) = O(N/m)$. In this way, we can solve the problem for all m in $O(\sum_{1 \leq m \leq M} N/m) = O(N \log M)$.

Finding v_{i+1}.

We have $v_1, v_2, \ldots, v_i$, and we want to find v_{i+1} in $O(1)$. It is enough to check whether $v_i + m + 1$ is evil. It's easier to check whether it is **not** evil, i.e., whether there exists a winning move.

H: Hyper Smawk Bros

Finding v_{i+1}.

We have $v_1, v_2, \ldots, v_i$, and we want to find v_{i+1} in $O(1)$. It is enough to check whether $v_i + m + 1$ is evil. It's easier to check whether it is **not** evil, i.e., whether there exists a winning move.

Finding v_{i+1}.

We have $v_1, v_2, \ldots, v_i$, and we want to find v_{i+1} in $O(1)$. It is enough to check whether $v_i + m + 1$ is evil. It's easier to check whether it is **not** evil, i.e., whether there exists a winning move.

Let's start from some n which satisfies $v_j \leq n \leq v_{j+1}$. A winning move must ensure that the opponent cannot move to an evil state (necessary condition). So there are three candidate winning moves.

Finding v_{i+1}.

We have $v_1, v_2, \ldots, v_i$, and we want to find v_{i+1} in $O(1)$. It is enough to check whether $v_i + m + 1$ is evil. It's easier to check whether it is **not** evil, i.e., whether there exists a winning move.

Let's start from some n which satisfies $v_j \leq n \leq v_{j+1}$. A winning move must ensure that the opponent cannot move to an evil state (necessary condition). So there are three candidate winning moves.

1. Subtracting $n - v_j + 1$ (so that the opponent is too far from v_{j-1}; this can only work if $v_j - v_{j-1} = m + 2$).

Finding v_{i+1}.

We have $v_1, v_2, \ldots, v_i$, and we want to find v_{i+1} in $O(1)$. It is enough to check whether $v_i + m + 1$ is evil. It's easier to check whether it is **not** evil, i.e., whether there exists a winning move.

Let's start from some n which satisfies $v_j \leq n \leq v_{j+1}$. A winning move must ensure that the opponent cannot move to an evil state (necessary condition). So there are three candidate winning moves.

1. Subtracting $n - v_j + 1$ (so that the opponent is too far from v_{j-1}; this can only work if $v_j - v_{j-1} = m + 2$).
2. Subtracting $\frac{n-v_j}{2}$ (so that the opponent would have to subtract $\frac{n-v_j}{2}$ again to reach v_j).

H: Hyper Smawk Bros

Finding v_{i+1}.

We have $v_1, v_2, \ldots, v_i$, and we want to find v_{i+1} in $O(1)$. It is enough to check whether $v_i + m + 1$ is evil. It's easier to check whether it is **not** evil, i.e., whether there exists a winning move.

Let's start from some n which satisfies $v_j \leq n \leq v_{j+1}$. A winning move must ensure that the opponent cannot move to an evil state (necessary condition). So there are three candidate winning moves.

1. Subtracting $n - v_j + 1$ (so that the opponent is too far from v_{j-1}; this can only work if $v_j - v_{j-1} = m + 2$).
2. Subtracting $\frac{n-v_j}{2}$ (so that the opponent would have to subtract $\frac{n-v_j}{2}$ again to reach v_j).
3. Subtracting $\frac{n-v_{j-1}}{2}$ (so that the opponent would have to subtract $\frac{n-v_{j-1}}{2}$ again to reach v_{j-1}).

H: Hyper Smawk Bros

Finding v_{i+1} efficiently.

The previous observation provides a recursive function which can call itself up to 3 times.

Finding v_{i+1} efficiently.

The previous observation provides a recursive function which can call itself up to 3 times.
The recursive function answers the question "can we win starting from $(n, n - v_j)$?" (i.e., it is forbidden to make the move which would win immediately). This is equivalent to "is n good or neutral?"

Finding v_{i+1} efficiently.

The previous observation provides a recursive function which can call itself up to 3 times.
The recursive function answers the question "can we win starting from $(n, n - v_j)$?" (i.e., it is forbidden to make the move which would win immediately). This is equivalent to "is n good or neutral?"
We want to make the average number of recursive calls less than 1, so that the function terminates in $O(1)$.

H: Hyper Smawk Bros

Finding v_{i+1} efficiently.

The previous observation provides a recursive function which can call itself up to 3 times.
The recursive function answers the question "can we win starting from $(n, n - v_j)$?" (i.e., it is forbidden to make the move which would win immediately). This is equivalent to "is n good or neutral?"
We want to make the average number of recursive calls less than 1, so that the function terminates in $O(1)$.

1. We do not need the first call, because $v_{j-1} + m + 1$ is neutral (proved before).

Finding v_{i+1} efficiently.

The previous observation provides a recursive function which can call itself up to 3 times.
The recursive function answers the question "can we win starting from $(n, n - v_j)$?" (i.e., it is forbidden to make the move which would win immediately). This is equivalent to "is n good or neutral?"
We want to make the average number of recursive calls less than 1, so that the function terminates in $O(1)$.

1. We do not need the first call, because $v_{j-1} + m + 1$ is neutral (proved before).
2. The second call happens with probability $\lesssim 1/2$ ($n - v_j$ must be even, and the first case must not apply).

H: Hyper Smawk Bros

Finding v_{i+1} efficiently.

The previous observation provides a recursive function which can call itself up to 3 times.
The recursive function answers the question "can we win starting from $(n, n - v_j)$?" (i.e., it is forbidden to make the move which would win immediately). This is equivalent to "is n good or neutral?"
We want to make the average number of recursive calls less than 1, so that the function terminates in $O(1)$.

1. We do not need the first call, because $v_{j-1} + m + 1$ is neutral (proved before).
2. The second call happens with probability $\lesssim 1/2$ ($n - v_j$ must be even, and the first case must not apply).
3. The third call happens with probability $\lesssim 3/8$ ($n - v_{j-1}$ must be even, the first case must not apply, and the second call must not succeed).

H: Hyper Smawk Bros

Finding v_{i+1} efficiently.

The previous observation provides a recursive function which can call itself up to 3 times.
The recursive function answers the question "can we win starting from $(n, n - v_j)$?" (i.e., it is forbidden to make the move which would win immediately). This is equivalent to "is n good or neutral?"
We want to make the average number of recursive calls less than 1, so that the function terminates in $O(1)$.

1. We do not need the first call, because $v_{j-1} + m + 1$ is neutral (proved before).
2. The second call happens with probability $\lesssim 1/2$ ($n - v_j$ must be even, and the first case must not apply).
3. The third call happens with probability $\lesssim 3/8$ ($n - v_{j-1}$ must be even, the first case must not apply, and the second call must not succeed).

So the function calls itself $\lesssim 7/8$ times on average. This is fast enough for all m (it can be tested locally).

K: Keygen 3

Not solved before freeze.

K: Keygen 3

Statement (summary).

Print all the bitonic permutations of length n with m cycles. If there are more than 2000 of them, print 2000 of them.

K: Keygen 3

Minor modification.

For convenience, let's find "anti-bitonic" permutations instead (decreasing, then increasing).

K: Keygen 3

Minor modification.

For convenience, let's find "anti-bitonic" permutations instead (decreasing, then increasing). There is a bijection between bitonic and anti-bitonic permutations.

```
vector<int> f(vector<int> p) {
    int n = p.size();
    reverse(p.begin(), p.end());
    for (auto &u : p) u = n - u + 1;
    return p;
}
```

K: Keygen 3

Let's divide into three cases:

K: Keygen 3

Let's divide into three cases:

1. $n \leq 18$;

K: Keygen 3

Let's divide into three cases:

1. $n \leq 18$;
2. $n \geq 19$, $n - m \geq 10$;

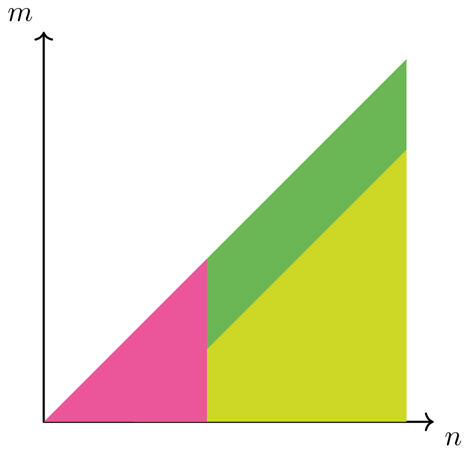
K: Keygen 3

Let's divide into three cases:

1. $n \leq 18$;
2. $n \geq 19$, $n - m \geq 10$;
3. $n \geq 19$, $n - m \leq 9$.

Let's divide into three cases:

1. $n \leq 18$;
2. $n \geq 19$, $n - m \geq 10$;
3. $n \geq 19$, $n - m \leq 9$.



K: Keygen 3

$n \leq 18$.

We can find all the anti-bitonic permutations and test them in $O(n \cdot 2^n)$.

K: Keygen 3

$n \leq 18$.

We can find all the anti-bitonic permutations and test them in $O(n \cdot 2^n)$.

Transforming a solution to (n, m) into a solution to $(n+1, m+1)$.

Just append $n+1$ at the end.

K: Keygen 3

$n \leq 18$.

We can find all the anti-bitonic permutations and test them in $O(n \cdot 2^n)$.

Transforming a solution to (n, m) into a solution to $(n+1, m+1)$.

Just append $n+1$ at the end.

Transforming a solution to (n, m) into a solution to $(n+1, m)$.

Append $n+1$ at the end, and swap p_1 with $p_1 + 1$.

K: Keygen 3

$n \leq 18$.

We can find all the anti-bitonic permutations and test them in $O(n \cdot 2^n)$.

Transforming a solution to (n, m) into a solution to $(n+1, m+1)$.

Just append $n+1$ at the end.

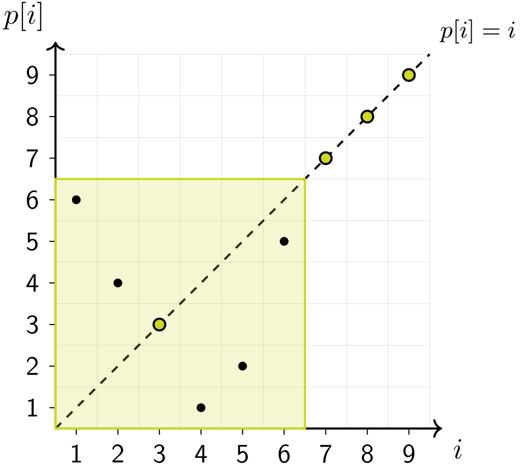
Transforming a solution to (n, m) into a solution to $(n+1, m)$.

Append $n+1$ at the end, and swap p_1 with $p_1 + 1$.

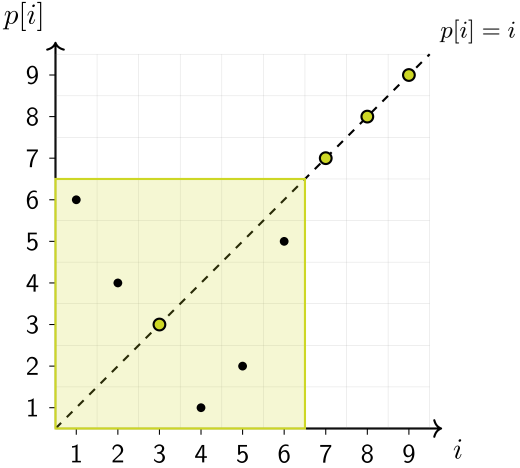
$n \geq 19, n - m \geq 10$.

First, solve $(18, i)$ for $1 \leq i \leq 8$ (for each i, we have ≥ 2000 solutions). Transform these solutions into solutions to (n, m).

K: Keygen 3



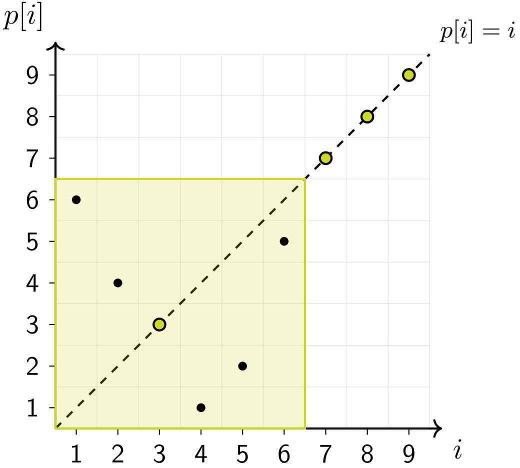
K: Keygen 3



$n \geq 19, n - m \leq 9$.

$p_1, p_2, \ldots, p_n$ must have at least $n - 9$ cycles, so it must have at least $n - 18$ fixed points (i.e., $p_i = i$).

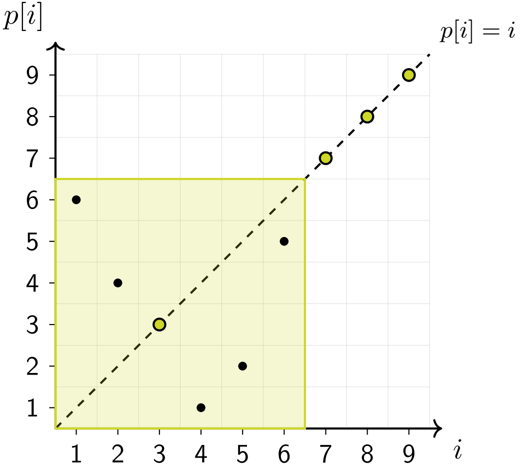
K: Keygen 3



$n \geq 19, n - m \leq 9.$

$p_1, p_2, \ldots, p_n$ must have at least $n - 9$ cycles, so it must have at least $n - 18$ fixed points (i.e., $p_i = i$).
The permutation structure is $[p_1, \ldots, p_1 + 1, p_1 + 2, \ldots, n]$: the highlighted square has at most 1 fixed point, so there are at most $n - p_1 + 1$ fixed points.

K: Keygen 3



$n \geq 19, n - m \leq 9.$

$p_1, p_2, \ldots, p_n$ must have at least $n - 9$ cycles, so it must have at least $n - 18$ fixed points (i.e., $p_i = i$).
The permutation structure is $[p_1, \ldots, p_1 + 1, p_1 + 2, \ldots, n]$: the highlighted square has at most 1 fixed point, so there are at most $n - p_1 + 1$ fixed points.
Therefore, $p_1 \leq 19$, and we can iterate over all such permutations.