# Problem A. Another GCD Problem

- Considering the direct computation of $f(k)$, we can find that $f(k) = \gcd(ka_1, a_2 - a_1, \ldots, a_n - a_1)$.
- Therefore, if all elements in $a$ are equal, the answer is infinite. Otherwise, the answer is $ans = \gcd(a_2 - a_1, \ldots, a_n - a_1)$, and the smallest $k$ is $\mathrm{lcm}(a_1, ans)/a_1$.
- Derivation of the value of $f(k)$: Let $t = f(k)$. Clearly $t \mid \gcd(a_2 - a_1, a_3 - a_1, \ldots, a_n - a_1)$. Also, clearly $t \mid ka_1$, thus proven.
- Sufficiency follows from $\gcd(a, b) = \gcd(a, a + b)$.
- Time complexity is $O(n \log n)$.

# Problem B. Ball And Movable Walls

A particle is inside a two-dimensional rectangle, given its initial velocity. The sides perpendicular to the $x$-axis expand outward at a certain speed, while the sides perpendicular to the $y$-axis compress inward at a certain speed. Upon colliding with any side, the component of the particle's velocity perpendicular to that side is completely reversed. Determine the position of the particle after the compression ends.

- Consider $v_x$ and $v_y$ separately. The change in the $y$-coordinate is straightforward, and the end time $T_{\text{end}}$ can be calculated.
- If the speeds of the left and right plates are both 0, the answer can be calculated directly.
- Otherwise, the particle will only collide in the left-right direction $O(\log(T_{\text{end}}))$ times. This is because the distance traveled by the particle from one collision to the next increases by at least 1/20 each time.
- Therefore, collisions in the left-right direction can be simulated by brute force.

Time complexity per test case: $O(\log(T_{\text{end}}))$.

# Problem C. Circular Primes

The simplest approach is to check, for each query and for each number in the corresponding interval, whether it is a circular prime or a partially circular prime. To perform this check, it is necessary to consider all cyclic permutations of the number and test which of them are prime numbers. To simplify the implementation of cyclic permutations, the number can be converted into a `string`. It is also possible to implement a solution without conversion, processing numbers directly via the `int` type.

Next, we notice that during program execution we repeatedly check the same numbers for primality. To avoid repeated computations, we use an array `primes[]`, in which we store the results of already performed checks:

- `primes[i] = 2`, if it has not yet been checked whether the number `i` is prime;
- `primes[i] = 1`, if the number `i` is prime;
- `primes[i] = 0`, if the number `i` is not prime.

Thus, each number is checked for primality at most once. This solution is implemented in the file `circular_40p.cpp`.

We optimize the counting of circular primes. Such numbers are relatively rare. In fact, in the interval from 1 to 1,000,000, there are only 55 circular primes. We pre-write all these numbers into an array

`circular_primes[]`. When `t = 1`, for each query we iterate over this relatively small array and count how many numbers fall into the interval from `a` to `b`. Thus, we avoid checking all numbers in the interval and significantly speed up the solution.

We use prefix counters. We create an additional array `p[]`, where `p[i]` contains the number of circular primes or partially circular primes in the interval from 1 to `i`.

Then the number of desired numbers in the interval from `a` to `b` can be determined by the formula:

$$p[b] - p[a-1].$$

# Problem D. Director and Excitement

### Brief Description

Given two increasing arrays $l$ and $r$ of length $n$, and a fixed constant $k$.

There are $q$ queries. Each query provides an interval $[L, R]$ and a constant $D$, asking whether it is possible to divide these people into $k$ groups such that within each group, there exists a way to assign $x_i \in [l_i, r_i]$ so that the minimum value of $|x_i - x_j| (i \neq j)$ within the group is no less than $D$.

Assume $l_i = r_i$. It can be found that the grouping must be: $[L, L+k, L+2k, \ldots], [L+1, L+1+k, L+1+2k, \ldots], \ldots$. That is, positions with the same modulo $k$ are assigned to the same group.

Assume $k = 1$. The problem is equivalent to checking whether $\forall i < j, (r_j - l_i)/(j-i) \geq D$.

In summary, each query essentially asks whether for all $i < j$ within the interval that have the same modulo $k$, the condition $r_j - jD \geq l_i - iD$ holds. This form can be maintained by various data structures. The standard implementation is described below.

Consider a static divide-and-conquer structure similar to a segment tree. For a node $x$ representing the interval $[l, r]$, precompute $val_x$ which denotes the maximum feasible $D$ within $[l, r]$.

- $val_x$ can be obtained by first taking the minimum of $val_{ls_x}$ and $val_{rs_x}$, then constructing convex hulls for $[l, mid]$ and $[mid+1, r]$ respectively and performing queries.
- If binary search is used for tangent line queries on the convex hull, the time complexity for this part is $O(n \log n \log D)$. If two-pointers are used, it becomes $O(n \log n)$.
- For a query $[L, R]$, similar to segment tree queries, we can insert all queries into nodes where the interval straddles the midpoint (i.e., $l \leq L \leq mid < R \leq r$). When recursively reaching a whole interval, directly return $val_x$.
- For queries inserted at a particular node, since $D$ is fixed, we can separately compute the convex hulls for $[l, mid]$ and $[mid+1, r]$, query for the extreme values at slope $k = D$, and then compare these extreme values.
- Each query is inserted into $O(\log n)$ nodes, and each node requires a binary search for the extreme value. Therefore, the time complexity for this part is $O(qk \log^2 n)$.

The total time complexity is $O(n \log n + qk \log^2 n)$ or $O(n \log n \log W + qk \log^2 n)$. In practice, the time limit is relatively lenient, and any solution with logarithmic complexity should be acceptable.

# Problem E. Exposing The Foxes

While reading the input file, the $x$-coordinates of each segment are swapped if necessary so that the condition $x_1 \leq x_2$ holds, after which the pairs $\{x_1, 1\}$ and $\{x_2, 2\}$, corresponding to the beginning and

the end of the segment, are written into vector $x$. The same is done for the $y$-coordinates, writing them into vector $y$.

To examine vertical lines, sorting is performed: `sort(x.begin(), x.end())`, and then in a loop over the elements of vector $x$, either one is added to counter $c$, or one is subtracted from it, depending on whether the beginning or the end of a segment is encountered.

Horizontal lines are processed analogously. In the end, the largest value attained by counter $c$ is printed.

# Problem F. Force Levels

First, it is convenient to consider the solution for a single adaptation query, that is, for the case $q = 1$. It is necessary to process the days in chronological order as they are given, keeping track of when the last Force increase occurred, the current Force level, and the total Force increase so far. When processing the next increase, update the total Force increase as well as the current Force level to account for how many times skills were practiced. The answer is the difference between the total Force increase and the Force level remaining at the end. Since the increase days are already sorted by time, processing one increase takes $O(1)$, so the total complexity is $O(n)$.

If considered naively, one could now process all $n$ increases for each of the $q$ adaptations and answer the corresponding query, but the complexity of such an approach is too high and equals $O(qn)$. The problem can be solved more efficiently by preprocessing and storing the data necessary to answer each query, which takes $O(N)$, and then answering each query using binary search over $N$, which gives complexity $O(Q \log N)$. Thus, the total complexity of this algorithm is $O(N + Q \log N)$, which is sufficient for the maximum score.

Using the two-pointers technique, the problem can also be solved in $O(N+Q)$ time, although this was not required for the maximum score. The key idea of this solution is to iterate in a single loop simultaneously over the increases and over the queries, performing the corresponding operations. Besides the better time complexity, the memory complexity of this algorithm is also lower than that of the binary-search-based solution described above.

# Problem G. Efficient Summation

The first approach was written by dls during problem testing. It is similar to the traditional min25 sieve, where the last prime number is handled separately during the search. The complexity is completely incalculable, but it passes. However, this approach is extremely difficult to write.

Let's provide a more reasonable approach: Consider storing all states during the min25 $n^{1-\epsilon}$ search and performing a high-dimensional suffix sum on them. However, you will find that this undercounts when the maximum prime factor exponent of the query $x$ is 1. Thus, we can consider a sqrt-decomposition on the maximum prime factor of $x$. Let the threshold be $B$. For maximum prime factors $p \leq B$, recompute the high-dimensional suffix sum each time. Otherwise, the problem can be transformed into a two-dimensional point query problem where one dimension is within $n/B$. By offline processing the larger dimension and maintaining it with a block data structure that supports $O(1)$ addition and $O(\sqrt{n})$ query, and since the other dimension has size $\sqrt{n}$, the query complexity per instance is $O(n^{1/4})$.

Choosing an appropriate $B$ allows passing this problem. The time limit is set quite generously and does not emphasize constant optimization.

# Problem H. How Many Rounds Are Expected?

Use the potential method to analyze. Each set only depends on its own size and is independent of the states of other sets. Thus, define $f(a)$ as the potential function for a set containing $a$ elements.

- Let the current state have sets with element counts $b_1, b_2, \ldots, b_{n_0}$. All possible pairing schemes form a set $S$. After applying the $i$-th pairing scheme, the element counts of the sets become $b_1^i, b_2^i, \ldots, b_{n_i}^i$.

- Then we have $1 + \sum_{i=1}^{n_0} f(b_i) = \frac{1}{|S|} \sum_{i=1}^{|S|} \sum_{j=1}^{n_i} f(b_j^i)$. The answer to the original problem is $f(m) - \sum_{i=1}^{n} f(a_i)$.
- Solve for $f(a)$ using Gaussian elimination. When constructing equations for $f(a)$, it suffices to consider the change in the element count of a set containing $a$ elements after one operation.

# Problem I. Infinite Hexagonal Grids

Given a black-and-white hexagonal grid graph. In each operation, you can choose a hexagon and flip the colors of its adjacent hexagons (without flipping itself). Determine whether one grid can be transformed into another through a sequence of such operations.

- Conclusion: The answer is YES if and only if for each $x$ coordinate (and similarly for each $y$ coordinate and each $z$ coordinate), the parity of the number of black cells is the same in both grids.
- Necessity: Obviously, a single operation does not change the parity of the number of black cells with the same $x$ coordinate (or $y$ or $z$ coordinate). (From the diagram, each coordinate involves exactly two cells being flipped.)
- Sufficiency: The problem is essentially equivalent to transforming a given grid into an all-white grid (which can be achieved via an operation similar to XOR).
- Consider constructing the solution from the outermost layer inward, layer by layer. Each time, the outermost layer (where $x$, $y$, or $z$ attains its maximum) also forms a hexagon.
- A straightforward construction method is: first flip the black cells at the six corners, then independently handle the black cells on the six edges. From the condition above, the number of black cells on each edge is even, so they can all be flipped to white.

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

# Problem J. Judge of Gensokyo

Given a string $s$ consisting only of the characters '`o`', '`v`', and '`w`', where '`w`' is treated as '`vv`'. Find the longest palindromic substring of this string.

A natural idea is to first replace every '`w`' with two '`v`' (let the resulting string be $s'$), then run Manacher's algorithm.

This approach has two issues:

1. The longest valid substring in $s'$ may not correspond to the longest valid substring in $s$.
2. A substring in $s'$ might not correspond to any valid substring in $s$. For example, the substring "`wov`" in $s$ becomes "`vvov`" in $s'$. Its longest palindromic substring might be "`vov`", but this does not represent any substring in $s$.

The first issue can be resolved by enumerating each palindrome center and finding the longest possible radius.

For the second issue, we can consider repeatedly removing '`w`' from both ends until the first non-`w` character is encountered at each end. This yields the longest palindromic substring in $s$ that corresponds to the current palindrome center.

Thus, we enumerate each palindrome center and its maximum palindrome radius in $s'$, and then repeatedly shrink the w's to find the longest palindromic substring that actually exists in $s$. Time complexity: $O(|s|)$.

# Problem K. Knapsack And Greedy Algorithm

Given a $W_{\text{lim}}$, construct a 0/1 knapsack instance with as few items as possible, and with item volumes and values as small as possible, such that for every knapsack capacity $W$ from 2 to $W_{\text{lim}}$, the greedy algorithm fails to produce the correct solution.

The unique construction method (assuming $n = W_{lim}$):

- Item volumes: $1, n, n-1, \ldots, 2$
- Item values: $n, n^2-1, n^2-n-2, \ldots, n+1$

First, prove that after sorting, the item volumes must be $1, n, n-1, \ldots, 2$.

For $n = 2$, the only feasible scheme is volumes $1, 2$ (the additional 1 after 1,2 is useless).

- Next, consider induction. Assume that for $n-1$, the unique scheme is volumes $1, n-1, \ldots, 2$. Then for capacity $n$, the greedy algorithm clearly obtains the optimal solution.
- Due to uniqueness, we cannot insert an item with volume $\leq n-1$. Hence, we can only insert $n$ after 1, resulting in volumes $1, n, n-1, \ldots, 2$.

Next, consider the item values. The values must satisfy:

$$\begin{cases} \bullet\ \nu_1 > \frac{\nu_n}{n} > \frac{\nu_{n-1}}{n-1} > \ldots > \frac{\nu_2}{2}. \\ \bullet\ \forall 1 < i < n,\ \nu_i + \nu_1 < \nu_{i+1},\ \nu_1 < \nu_2. \end{cases}$$

It can be observed that, given the second condition and that item 1 is first, the first condition naturally holds. Therefore, we need to find the smallest $\nu_1$ such that:

$$\nu_n = \nu_1 + 1 + (n-2)(\nu_1 + 1) < n \cdot \nu_1$$

which simplifies to $\nu_1 > n - 1$. Hence, $\nu_1 = n$. Then the values $\nu_2, \nu_3, \ldots, \nu_n$ can be derived accordingly.

# Problem L. Let's Avoid Obstacles

Given an $n \times m$ grid with $k$ obstacles. Consider a path that starts from some row in the first column and ends at some row in the $m$-th column, without passing through any obstacles. For each obstacle, there are two ways to pass it: from above or from below.

For all $2^k$ possible choices (one for each obstacle), compute the minimum path length.

- Simulate according to the description. Perform BFS while recording the passing state for each obstacle.
- State compression can be used to record the passing states of obstacles. Time complexity: $O(nm2^k)$.

# Problem M. Managing Team of Robots

## 1 Subtasks 1 and 2

The simplest possible method is the method of "sequentially multiplying by 2, starting from the first element". Starting from the second element of the sequence and proceeding to the last, we examine the elements one by one and multiply by 2 until the current element becomes greater than or equal to the previous one. This is a kind of greedy algorithm.

In subtasks 1 and 2, there is a condition that all elements of the sequence are powers of two, so by applying the operation (in subtask 1) 1 time, (in subtask 2) at most 18 times to each element, we can make the sequence nondecreasing. Therefore, treating the maximum size $A_i$ as a constant, the problem can be solved with time complexity $O(N)$.

## 2 Subtask 3

The problem can be solved with the same time complexity as in subtasks 1 and 2. However, since there is no condition that all elements are powers of two, the result of applying the doubling operation may increase each element to a value not exceeding twice the previous element. Therefore, up to $18 + i$ operations may be applied to the $i$-th element, giving time complexity $O(N^2)$.

## 3 Subtask 4

The solutions for subtasks 1, 2, and 3 have low time complexity, but the numbers may become too large, so they cannot be applied directly to subtasks 4, 5, and 6. Therefore, other observations are needed.

First, in subtask 4, since each element is either 2 or 3, we can consider the possible cases. Let us consider two adjacent elements $A_{i-1}$ and $A_i$. The case where it is necessary to apply the operation to $A_i$ occurs only when $A_{i-1} = 3, A_i = 2$. When $A_{i-1}$ and $A_i$ have other values, $A_{i-1} \leq A_i$, so there is no need to apply the operation separately.

Moreover, after applying the operation to $A_i$ in the case above, $A_i$ becomes 4, so the value of $A_i$ becomes greater than all subsequent elements $(A_{i+1}, A_{i+2}, \dots)$. Therefore, all subsequent elements must be doubled. After all subsequent elements have been doubled, the same logic can be applied to them. Thus, by applying this sequentially for $i = 2, 3, \dots, N$, the problem can be solved. The time complexity is $O(N)$.

## 4 Subtask 5

In subtask 5, since the array is initially sorted in decreasing order, an observation similar to that of subtask 4 can be made.

## 5 Subtask 6

As in the observations of the previous subtasks, where the relation between two adjacent elements $A_{i-1}$ and $A_i$ was used, solving the full problem also uses this relation. More important than the actual value of an element is how many times the operation must be applied so that it becomes greater than or equal to the previous element.

Let $M_i$ be the smallest integer $x$ such that $A_{i-1} \leq A_i \times 2^x$. Now applying the «doubling» operation to index $i$ means decreasing $M_i$ by 1 and increasing $M_{i+1}$ by 1. The goal is to make all $M_1, \dots, M_N$ nonpositive. Therefore, while traversing the array $M$ starting from the second element, if an element has a positive value, we apply the operation the required number of times. The time complexity is $O(N)$.