Problem A. Mission Hexa

Problem Setter: Irfanur Rahman Rafio
Tester: Rumman Mahmud, Nafis Sadique
Alt Writer: Tariq Hasan Rizu
Category: Geometry, Number Theory, Combinatorics
Total Solved: 16
First to Solve: BUET_Silver_hawks

Hint 1: The beehive is **symmetric** about the center.
Hint 2: When can a single shot hit two cells?
Hint 3: How can you check whether two cells and the center lie on a straight line?
Hint 4: Introduce a **coordinate system** where the central cell is the origin.
Hint 5: Think about **vectors**.
Hint 6: From any cell, movement is possible in exactly 6 directions.
Hint 7: The beehive has a rotational symmetry of order 6.
Hint 8: How many cells are present in layer i?
Hint 9: How can you determine the layer number of a given cell k?
Hint 10: The length of the shortest path from the center to cell k is equal to the layer number of cell k.
Hint 11.1: Represent paths efficiently.
Hint 11.2: Each movement corresponds to one of the six possible directions in the grid.
Hint 11.3: Instead of tracking individual cells, represent a path as a sequence of movements.

Solution:

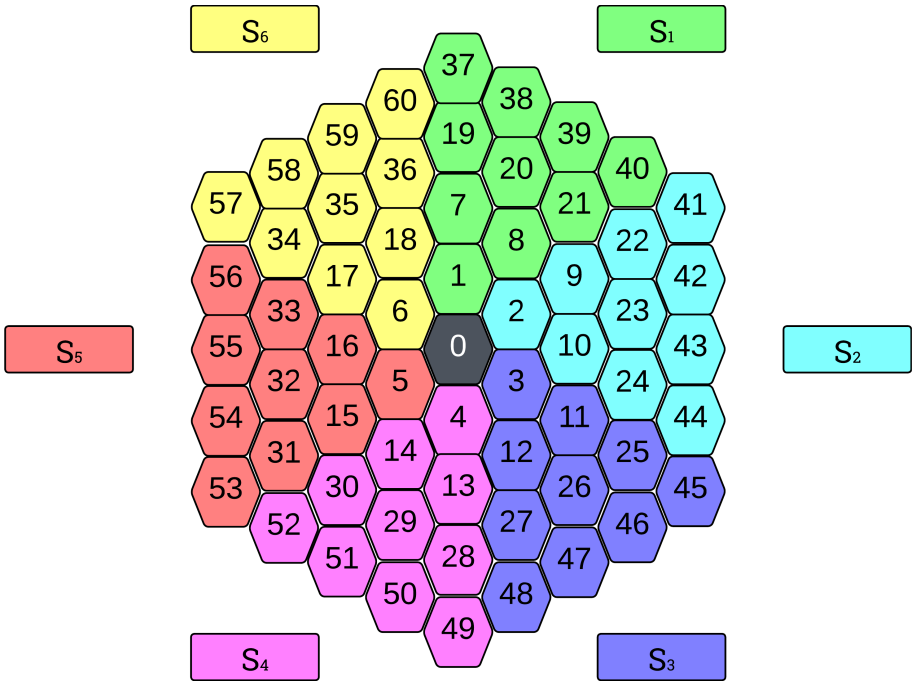
First of all, the beehive is symmetric about the center. So you can ignore the rule that says that the numbering of cells proceeds clockwise for odd-numbered layers and counter-clockwise for even-numbered layers. Without loss of generality, simply assume numbering proceeds clockwise for all layers. This relabeling does not change any geometric property used later.

Each hexagonal cell, including the queen's chamber, has exactly six neighbors. So, there are six possible directions to move from every cell. Define $u_1, u_2, \ldots, u_6$ as the unit vectors representing these six directions, where u_z is the vector connecting the center of cell 0 to the center of cell z. These six unit vectors will be referred to as **step vectors**. For convenience, define $u_0 = u_6$ and, more generally, $u_z = u_{z \bmod 6}$ for any integer z.

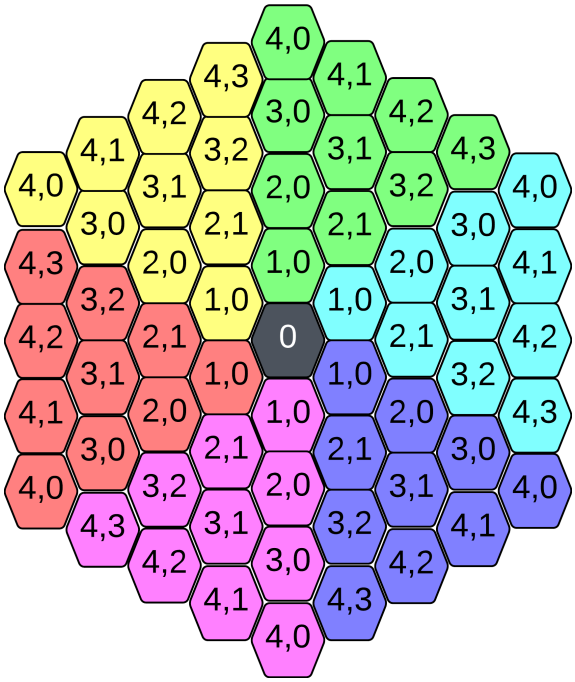
Let the **direction vector** from the center of cell 0 to the center of any other cell i of the beehive be denoted by v_i. Since the step vectors represent all possible directions of movement, every direction vector v_i can be expressed as a weighted sum of the step vectors, where all weights are non-negative integers. For example, one path from cell 0 to cell 20 is $0 \to 1 \to 7 \to 19 \to 20$, and $v_{20} = 3u_1 + u_3$.

It is easy to observe that the beehive is rotationally symmetric and that the six step vectors are equally spaced, each separated by an angular interval of 60°.

There are exactly six cells in layer 1, and their direction vectors are u_1, u_2, u_3, u_4, u_5, and u_6.
Extending these directions outward produces six cells in layer 2 with direction vectors $2u_1, 2u_2, 2u_3, 2u_4, 2u_5$, and $2u_6$.
To obtain the remaining six cells in layer 2, one additional movement is made after turning by 120°. The corresponding direction vectors are $2u_1 + u_3$, $2u_2 + u_4$, $2u_3 + u_5$, $2u_4 + u_6$, $2u_5 + u_1$, and $2u_6 + u_2$.
Generalizing this pattern, you can divide the beehive (excluding the center) into six disjoint **sectors** $S_1, S_2, \ldots, S_6$, as illustrated in the following figure.



In each sector S_z, there are exactly x cells in layer x. The direction vectors of those x cells can be written uniquely in the form $xu_z + yu_{z+2}$, where $y \in \{0, 1, 2, \ldots, x-1\}$. Thus, every cell in the beehive (except the center) can be represented uniquely as (z, x, y), where z is the sector number, x is the layer number, and y is the offset within that layer.



Two cells (z_1, x_1, y_1) and (z_2, x_2, y_2) can be hit by the same laser shot *iff* their direction vectors are **positive scalar multiples** of each other. This happens when $z_1 = z_2$ and $\frac{y_1}{x_1} = \frac{y_2}{x_2}$.

Conversely, the cell (z, x_2, y_2) must be hit by a separate laser shot *iff* there exists no cell (z, x_1, y_1) such that $x_2 = kx_1$ and $y_2 = ky_1$ for some integer $k > 1$. This occurs precisely when x_2 and y_2 are **coprime**.

Thus, in a fixed sector S_z and a fixed layer x, the number of cells that must be hit by distinct laser shots is

exactly the number of integers y such that $0 \leq y < x$ and $\gcd(x, y) = 1$. This number is exactly $\varphi(x)$, where φ denotes **Euler's totient function**.

Since there are six sectors and each sector contributes $\varphi(x)$ independent laser shots for layer x, the total number of laser shots required to kill all guard bees in n layers is $6 \sum_{x=1}^{n} \varphi(x)$.

Adjacency between any two cells in the beehive is bidirectional. So, the paths from cell i to cell 0 and the paths from cell 0 to cell i have a one-to-one correspondence. Each cell in layer x, where $0 < x < n$, is adjacent to at least one cell in layer $(x - 1)$, and to no cell in any lower layer. As a result, the minimum number of steps required to reach layer 0 from layer x is exactly x.

This implies that if cell i can be represented as (z, x, y), then every shortest path from cell 0 to cell i contains exactly x steps.

Since the beehive is rotationally symmetric and symmetric about the center, the value of z does not affect the number of shortest paths and the initial relabeling does not change the answer.

The number of cells in layer x is $6x$. To obtain (z, x, y) from a given cell index k, first determine the layer number x such that $3(x - 1)x < k \leq 3x(x + 1)$. The integer solution for x is unique and one of the ways to calculate it is using binary search.
Let $p = k - (3(x - 1)x + 1)$ be the zero-based position of cell k within layer x. Since each sector contains exactly x cells in layer x, the sector index is $z = \lfloor p/x \rfloor + 1$, and the offset within that sector is $y = p \bmod x$. Thus, cell i is represented uniquely as (z, x, y).

Using basic geometry, it is easy to verify that for any z, $u_{z+1} = u_z + u_{z+2}$.
Now consider a cell k with representation (z, x, y). Its direction vector is $v_k = xu_z + yu_{z+2} = (x - y)u_z + yu_{z+1}$. Notice that $(x - y) + y = x$. So reaching cell k using a shortest path requires exactly $(x - y)$ steps in the direction u_z and exactly y steps in the direction u_{z+1}.

For any two vectors a and b, $a + b = b + a$. So, the order of your steps does not affect the final position. Out of the x steps in the shortest path, any y steps can be chosen to move in the direction u_{z+1}, with the remaining $(x - y)$ steps taken in the direction u_z. Each such choice produces a distinct shortest path.

So, the number of unique shortest paths from cell 0 to cell (z, x, y) is $\binom{x}{y}$.

Problem B. Boulevard of Broken Cars

Problem Setter: Pritom Kundu
Tester: Nafis Sadique, Irfanur Rahman Rafio
Alter: Gemini
Category: DS
Total Solved: 0
First to Solve: N/A

Solution 1:

First let's solve the problem without any addition queries. We will add the trucks one by one in deceasing order of speed and maintain a set of intervals (l_i, r_i) with color i meaning that for any query y in the interval (l_i, r_i), the fastest truck to reach y is truck i. From this set, we can easily answer for any point y, the fastest truck to reach y in $O(\log n)$ time using `lower_bound()`.

When we add a new truck at position x with speed v, we need to find the interval where this truck is the fastest. **The key observation is the new truck will be the fastest truck in a continuous interval which contains** x. This is because all previously added trucks are faster, so if a truck reaches a point y right of x faster than the new truck, it must also reach all points right of y faster than the new truck. Same applies for points left of x.

To find the interval where the new truck is the fastest, we can do two binary searches. One to find the leftmost point l where the new truck is faster, and one to find the rightmost point r where the new truck is faster. To check if the new truck is faster at a point y, we can find the truck j which is currently the fastest at point y using `lower_bound()`, and compare the time taken by truck j and the new truck to reach point y.

This is done in $O(\log n)$ time for each iteration of binary search for finding the current fastest truck, so adding a new truck takes $O(\log^2 n)$ time. However, we can do faster.

Instead of doing a binary search for the left, we can do a linear scan over the intervals starting from the interval containing x and moving left until we find the first interval I where the new truck is not faster. All intervals we passed are now covered by the new truck and can be removed. We still need to do a binary search for I to find the exact leftmost point l where the new truck is faster. However, we no longer need to do a `lower_bound` for the current fastest truck, because we already know the truck for interval I. Similarly, we can do a linear scan to the right to find the rightmost point r where the new truck is faster.

It may seem on the first glance that this approach is $O(n)$ per addition in the worst case, leading to $O(n^2)$ overall. However, when iterating over the intervals, all intervals that we visit are removed from the set, and at most two new intervals are added per addition. Therefore, over all additions, the total number of intervals visited is $O(n)$, leading to an time complexity of $O(n \log n)$ overall for adding all trucks.

Cool! We can solve the problem without addition queries in $O(n \log n)$ time. Now let's incorporate the addition queries. There are multiple ways to do this.

- We can use the "static to dynamic" technique. See https://codeforces.com/blog/entry/67263. This technique allows us to add addition queries in $O(\log N)$ factor overhead, leading to $O(N \log^2 N)$ overall for adding all trucks and $O(\log^2 N)$ per query, where $N = n + q$.
- We can use sqrt decomposition. We can divide the addition queries into blocks of size K. For each block, we can build the above data structure for all trucks added in that block. Then, for each query, we can check all blocks that finish before query using the data structure for that block, and check all trucks added in the current block using brute force. With $K = \sqrt{N \log N}$, this leads to $O(N\sqrt{N \log N})$ overall, where $N = n + q$. With careful implementation, this should pass within the time limit.

Solution 2:

Note that, we can represent the time taken by each truck as a piecewise linear function which is V shaped with a value of 0 at the position of the truck. Our operations are adding such functions and querying the minimum value at a point.

There are a couple data structures to do this.

- It is easy to see that any two such functions intersect at most once (as long as we handle the case when two trucks have the same position separately). Therefore, we can use a Dynamic Li Chao Tree to maintain the minimum of all such functions at any point. Adding a new truck takes $O(\log M \log N)$ and answering a query takes $O(\log M)$, where M is the range of x coordinates and $N = n + q$.
- We can also use a segment tree with each node containing a dynamic convex hull to maintain the minimum of all functions. The segment tree should be built on the compressed x coordinates of all trucks and queries. Adding a new truck at position x with speed v involves adding two lines (one with slope $-v$ to the left of x and one with slope v to the right of x).

  Answering a query at position y involves querying all segment tree nodes covering y and finding the minimum value among all lines in those nodes. Both adding a truck and answering a query take $O(\log^2 N)$ time, where $N = n + q$. This approach needs careful implementation to ensure precision and efficiency.

Problem C. Gas Reservoir

Problem Setter: Mohammad Ashraful Islam
Tester: Raihat Zaman Neloy
Alt Writer: Tariq Hasan Rizu
Category: Graph Theory, Ad Hoc
Total Solved: 129
First to Solve: BUET_Silver_hawks

The solution can be broken down into two main phases:

1. **Component Labeling:** We must identify all unique 3D connected components of gas cells (.). Each cell (x, y, z) belonging to the same reservoir should be assigned a unique `reservoir_id`, and we should store the total `size` (number of cells) of each reservoir. This can be achieved using any graph traversal algorithm:
   - **Breadth-First Search (BFS):** Start from an unvisited gas cell and explore its 6 neighbors.
   - **Depth-First Search (DFS):** Recursively visit adjacent gas cells.
   - **Disjoint Set Union (DSU):** Treat each gas cell as a node and perform `unite` operations for all adjacent gas cells. The size of the component is maintained within the DSU root.
2. **2D Projection and Summation:** After labeling, we simulate drilling at every possible surface coordinate (x, y).
   - For a fixed (x, y), iterate through all depths $z \in [0, Z-1]$.
   - Collect the `reservoir_id` of every gas cell encountered in the column.
   - Use a set or a boolean array to ensure each reservoir's size is added to the total for that column **only once**.
   - The answer is the maximum sum obtained across all (x, y) positions.

Complexity

- **Time Complexity:** $O(T \times (X \cdot Y \cdot Z))$. Identifying components takes linear time relative to the number of cells. The projection phase takes $O(X \cdot Y \cdot Z)$ because we visit each cell in the grid once.
- **Space Complexity:** $O(X \cdot Y \cdot Z)$ to store the grid and the component IDs.

Problem D. Save the Wonderland

Problem Setter: Raihat Zaman Neloy
Tester: Mukit Hasan
Alt Writer: Pritom Kundu
Category: Graph Theory, Greedy, Binary Search
Total Solved: 3
First to Solve: GreenForces

The objective is to find the minimum integer radius R such that K guards can protect all N nodes in a tree. A guard at node U protects node V if the shortest path distance $dist(U, V) \leq R$.

Given $N, K \leq 10^5$, we require an efficient approach, typically $O(N \log N)$ or $O(N)$.

1 Monotonicity and Binary Search

The problem satisfies the property of monotonicity: if all nodes can be covered with radius R, they can also be covered with any radius $R' > R$. Therefore, we can binary search for the minimum R in the range $[0, N]$. For each candidate R, we need a check function to determine the minimum number of guards required.

2 Greedy Strategy

To calculate the minimum guards for a fixed R, we use a bottom-up greedy approach. In a tree, the most constrained nodes are the leaves. To maximize the coverage of a guard intended for a leaf, that guard should be placed as high as possible in the tree (towards the root), specifically at the R-th ancestor of that leaf.

2.1 State Representation

During a post-order DFS, each node u maintains two values:

- d_{uncov}: The distance from u to the farthest node in its subtree that is not yet covered by any guard.
- d_{guard}: The distance from u to the nearest guard located within its subtree.

2.2 Decision Rules

At each node u, we first aggregate the states from all children v:

$$d_{uncov}(u) = \max(\{d_{uncov}(v) + 1\}) \tag{1}$$

$$d_{guard}(u) = \min(\{d_{guard}(v) + 1\}) \tag{2}$$

We then apply the following transitions:

1. **Coverage Check:** If $d_{uncov}(u) + d_{guard}(u) \leq R$, the farthest uncovered node is within range of an existing guard in a sibling branch. We set $d_{uncov}(u) = -\infty$.
2. **Guard Placement:** If $d_{uncov}(u) = R$, we are forced to place a guard at u to cover the leaf at distance R. We increment the guard count, set $d_{guard}(u) = 0$, and $d_{uncov}(u) = -\infty$.
3. **Root Handling:** If $d_{uncov}(root) \geq 0$ after the traversal, one additional guard is placed at the root.

3 Complexity

- **Time Complexity:** The binary search takes $O(\log N)$ steps. Each check function is a single DFS traversal taking $O(N)$. The total complexity is $O(N \log N)$.
- **Space Complexity:** $O(N)$ is required to store the adjacency list and the recursion stack.

Problem E. Love Marriage

Problem Setter: Irfanur Rahman Rafio
Tester: Rafid Bin Mostofa
Alt Writer: Rafid Bin Mostofa
Category: Probability, Range Query, Greedy
Total Solved: 8
First to Solve: GodFathers Inc.

Hint 1: What are the requirements for a girl to be Shanto's wife?
Hint 2: Use **events** and **conditional probabilities**.
Hint 3: What information do you need to determine the conditional probabilities?
Hint 4: Can you **preprocess** some information to answer each query efficiently?
Hint 5: Can you determine the optimal value of k for type 2 queries with a greedy approach?

Solution:

Consider the type 1 queries. To compute the probability that a girl becomes Shanto's wife, first analyze the necessary conditions.

It is easy to observe that the requirements for any girl to be Shanto's wife are as follows:

- Shanto must reach her (not go to sleep before chatting with her).
- After chatting with her, Shanto must select her as his favorite girl.
- No subsequent girl may replace her as Shanto's favorite girl.

Define a **leader** as a girl whose happiness value is *strictly greater than* that of all previous girls. Similarly, define a **co-leader** as a girl whose happiness value is *greater than or equal to* that of all previous girls.

Now, define the following events:

- A_i: Shanto chats with the i-th girl.
- B_i: The i-th girl becomes Shanto's favorite girl.
- C_i: The i-th girl becomes Shanto's wife.

Trivially, $P[A_1] = 1$.

Shanto goes to sleep if he chats with k consecutive girls without finding a leader. For any $i > 1$, let j be the maximum index such that $1 \leq j < i$ and the j-th girl is a leader. Then Shanto chats with the i-th girl if and only if he chats with the j-th girl and there are fewer than k girls between them. So,

$$P[A_i] = \begin{cases} 0, & \text{if } P[A_j] = 0, \\ 0, & \text{if } i - j > k, \\ 1, & \text{otherwise.} \end{cases}$$

After Shanto chats with the i-th girl, she will surely become his favorite girl if she's a leader. If she's a coleader, she'll become his favorite girl if she wins her toss, which has a probability of $\frac{1}{2}$. If she's neither a leader, nor a coleader, there is no way she can become his favorite. So,

$$P[B_i \mid A_i] = \begin{cases} 1, & \text{if the } i\text{-th girl is a leader,} \\ \frac{1}{2}, & \text{if the } i\text{-th girl is a co-leader,} \\ 0, & \text{otherwise.} \end{cases}$$

After becoming Shanto's favorite girl, the i-th girl becomes Shanto's wife if and only if Shanto doesn't chat with any leader after her and all co-leaders after her lose their tosses.

Let j be the index of the most recent leader at or before i, let $r = \min(j + k, n)$, and let c be the number of co-leaders in the range $[i + 1, r]$. Then

$$P[C_i \mid A_i \text{ and } B_i] = \begin{cases} 0, & \text{if there exists a leader in } [i+1, r], \\ \left(\frac{1}{2}\right)^c, & \text{otherwise.} \end{cases}$$

Combining all three conditions, you can obtain

$$P[C_i] = P[A_i] \times P[B_i \mid A_i] \times P[C_i \mid A_i \text{ and } B_i].$$

All required quantities can be computed efficiently using **prefix-based preprocessing**, without explicitly simulating the process.

Recall that Shanto goes to sleep if he chats with k consecutive girls without encountering any leader. Equivalently, $P[A_i] = 1$ if and only if, among the girls indexed from 1 to i, there is no contiguous segment of length k that contains no leader. Therefore, the computation of $P[A_i]$ reduces to checking whether the longest prefix segment without a leader has length at most $k - 1$. This can be achieved by maintaining a prefix maximum of happiness values and a prefix maximum of consecutive non-leaders.

All other required quantities are computed using similar prefix-based techniques. In particular, prefix maximums of happiness values are used to classify leaders and co-leaders, and prefix counts of co-leaders allow the number of co-leaders in any interval to be obtained in constant time. Consequently, each probability $P[C_i]$ can be evaluated independently in $O(1)$ time after linear-time preprocessing.

For type 2 queries, the additional task is to determine an optimal value of k. Notice that after becoming Shanto's favorite girl, it is optimal for a girl if Shanto chats with as few additional girls as possible. Therefore, for any fixed i, the optimal value of k is the minimum value for which Shanto reaches the i-th girl. Any larger value of k can only introduce more competitors and cannot increase the probability.

Problem F. Morning Walk

Problem Setter: Monirul Hasan
Tester: Irfanur Rahman Rafio
Alt Writer: Raihat Zaman Neloy
Category: Geometry, Basic Physics
Total Solved: 184
First to Solve: IUT_Kamikaze

The key insight is that meetings on a circular trail happen **periodically**.

Relative Speed and Period

Consider walker i with speed v_i. The relative speed between Kabul and this walker is:

$$\text{relative\_speed} = |v_0 - v_i|$$

Whether they walk in the same direction ($v_i > 0$) or opposite directions ($v_i < 0$), after their first meeting at time t_i, they will meet again after traveling a relative distance of exactly L meters (one complete lap around the circular trail).

The time between consecutive meetings is:

$$\text{period} = \frac{L}{|v_0 - v_i|}$$

Counting Total Meetings

Given that the first meeting occurs at time t_i and Kabul walks for a total of T seconds, the time remaining after the first meeting is $(T - t_i)$ seconds.

The number of additional meetings is:

$$\left\lfloor \frac{T - t_i}{\text{period}} \right\rfloor$$

Therefore, the total number of meetings with walker i is:

$$\boxed{1 + \left\lfloor \frac{(T - t_i) \cdot |v_0 - v_i|}{L} \right\rfloor}$$

The time complexity is $O(N)$ per test case.

Problem G. Nonogram

Problem Setter: Pritom Kundu
Tester: Nafis Sadique, Irfanur Rahman Rafio
Alt Writer: Rafid Bin Mostofa
Category: Greedy, Ad Hoc
Total Solved: 1
First to Solve: BUET_Silver_hawks

First let's try to solve the problem without Bob's constraints, ie. let's assume all cells are `?`.

We can find a solution by greedily placing the blocks from left to right. Since $\sum b_i \leq n - k + 1$, we can always place all blocks with at least one space between them. After placing all blocks, we can fill the remaining cells with `?`. This will always give us a valid solution. In fact, this will give us the "leftmost" valid solution, meaning that for any other valid solution, the position of the i^{th} block will be at the same or a later position. Let's call this solution S_{left}.

Similarly, we can find the "rightmost" valid solution S_{right} by placing the blocks from right to left. Let's call this solution S_{right}.

Now, consider the i^{th} block. Suppose in S_{left}, it starts at position l_i and ends at position r_i. Similarly, in S_{right}, it starts at position l'_i and ends at position r'_i. We make the following two observations:

- If $(l'_i \leq r_i)$, then the cells from l'_i to r_i must be filled with `1` in any valid solution. This is because the i^{th} block must occupy these cells in both S_{left} and S_{right} (and hence in any valid solution).
- All cells between l_i and r'_i can potentially be filled with `1` in some valid solution. To see this, consider moving blocks one by one, one position to the right from S_{left} until we reach S_{right}. The i^{th} block will occupy all positions between l_i and r'_i during this process. Thus, these cell should be marked as `?` unless they are already marked as `1` from the previous observation.

Now, let's try to incorporate Bob's constraints. The `0`s divide the nonogram into segments of `?`s. Each `?` will be part of exactly one such contiguous segment of `?`s. If the i^{th} cell is `?`, let g_i be the length of the segment it is part of.

The first observation still holds, but the second observation doesn't work anymore because when shifting to the right, a block migh skip over a segment of `?`s entirely, because the length of the segment is less than the length of the block. For example, consider the nonogram `??0?0??` with the clue sequence [2]. The leftmost solution is `1100000` and the rightmost solution is `0000011`. Unfortunately, the second observation fails for the third `?` in the second segment. because when we cannot place the block of length 2 in the second segment of length 1.

However, it is easy to fix this issue. We can simply apply the second observation to segments which are long enough to fit the block. In other words, we modify the second observation as follows:

- A cell between l_i and r'_i which are part of a segment of length at least b_i (ie. $g_j \geq b_i$), can be covered by the i^{th} block in some valid solution. Thus, these cell should be marked as `?` unless they are already marked as `1` from the previous observation.

This leads to a $O(nk)$ solution, because for each of the k blocks, we might have to iterate through $O(n)$ cells to apply the two observations. To improve this, we note two things, the ranges for the first observation are disjoint, so looping through all of them will take $O(n)$ time overall. Secondly, instead of looping through all cells for the second observation, we note that for each cell, we just need to find the shortest block that can cover it.

This can be done in various ways, for example using a sparse table or a segment tree. But the simplest way is to process the cells left to right, and maintain a active set of blocks that can cover the current cell. We can use a multiset for this. When we reach a cell, we add all blocks that can start at this cell to the multiset, and remove all blocks that end just before this cell from the multiset. The minimum element in the multiset will be the length of the shortest block that can cover this cell. If this length is less than or equal to the length of the segment of `?` containing this cell, this cell cannot be covered and must be marked as `0`. Otherwise, it can be marked as `?` unless it is already marked as `1` from the first observation.

This leads to an overall time complexity of $O(n \log k)$.

Problem H. Optimal Balancing Strategy

Problem Setter: Tariq Hasan Rizu
Tester: Shahjalal Shohag, Irfanur Rahman Rafio
Alt Writer: Muhiminul Islam Osim, Rafid Bin Mostofa
Category: Number Theory, Sorting, Greedy
Total Solved: 52
First to Solve: CoU_Reignite

Key Observations

It is always possible to satisfy all three requirements, especially the third one, because any number can be reduced to 1, which is both evenly even and oddly odd. Also, it is always optimal to choose a **prime number** as p. For example, $72 = 2^3 \cdot 3^2$ can be made evenly even by dividing by 3 twice, with total cost $3 + 3 = 6$, instead of dividing by 9.

We split each operation into:

- **Ascend:** remove a prime factor p from A_i and send it to the sky at cost p,
- **Descend:** take any prime factor from the sky and multiply it into A_j at cost 0. Since $a, b \geq 1$, there is always at least one valid j such that A_j is available to receive either any even or any odd prime factor.

Thus, the optimal answer is the sum of the costs of all ascend operations.

Solution Approach

First, we construct an array B of pairs. Let,

$$B_i = (f_i, s_i)$$

For the i-th pair, the first element represents the cost of making A_i evenly even and the second element represents the cost of making A_i oddly odd. Thus, the optimal answer is the sum of the first values for a elements and the second values for b elements, the rest can use either value.

Suppose the optimal solution picks $x \geq a$ elements from the first components and $y = n - x \geq b$ elements from the second components. Our goal is to sort the array B such that the total cost,

$$answer = f_1 + f_2 + \cdots + f_x + s_{x+1} + \cdots + s_n.$$

Consider two adjacent elements B_i and B_{i+1}. We will swap them if and only if

$$f_1 + f_2 + \cdots + f_i + s_{i+1} + s_{i+2} + \cdots + s_n > f_1 + f_2 + \cdots + f_{i-1} + f_{i+1} + s_i + s_{i+2} + s_{i+3} + \cdots + s_n$$

$$\Rightarrow \quad f_i + s_{i+1} > s_i + f_{i+1}$$

$$\Rightarrow \quad f_i - s_i > f_{i+1} - s_{i+1}$$

Repeatedly applying this adjacent swap rule sorts the array by non-decreasing $(f_i - s_i)$, which by the exchange argument produces an optimal arrangement.

Algorithm

1. Factorize each A_i and compute f_i and s_i.
2. Sort indices by $(f_i - s_i)$ in non-decreasing order.
3. For all valid $x \in [a, n-b]$:
   - Assign the first x indices as evenly even.
   - Assign the remaining $n - x$ indices as oddly odd.
   - Compute the total cost,
   $$\mathrm{c} = \sum_{i \le x} f_i + \sum_{i > x} s_i.$$
4. Choose the assignment with the minimum total cost.

Complexity

Factorization of each A_i may take up to $O(\sqrt{A_i})$. Sorting takes $O(n \log n)$. Therefore, overall time complexity in the worst case is $O\left(\sum_{i=1}^{n} \sqrt{A_i} + n \log n\right)$ and space complexity is $O(n)$.

Problem I. Two Strings Attached

Problem Setter: Shahjalal Shohag
Tester: Rafid Bin Mostofa, Irfanur Rahman Rafio
Alt Writer: Sharif Minhazul Islam
Category: Z-function, Data Structures, Fenwick Tree
Total Solved: 0
First to Solve: N/A

We split all pairs into three disjoint cases depending on how i compares to $n - j + 1$.

Case 1: $i = n - j + 1$ (Equal prefix/suffix lengths)

So this implies that $j = n - i + 1$. So our condition becomes:

$$a[1 \dots i] + b[n - i + 1 \dots n] \\ = a[n - i + 1 \dots n] + b[1 \dots i]$$

The equality simplifies to:

- $a[1 \dots i] = a[n - i + 1 \dots n]$ (this means that i is a border of a)
- $b[1 \dots i] = b[n - i + 1 \dots n]$ (this means that i is a border of b)

So for every i, if both a and b have a border of length i, then every $j = n - i + 1$ is valid.

We can compute borders using Z-function in $O(n)$.

Case 2: $i < n - j + 1$ (Prefix shorter than suffix)

This is the main part of the problem.

Our condition is:

$$a[1 \dots i] + b[j \dots n] \\ = a[j \dots n] + b[1 \dots i]$$

Please carefully look at the following example for $i = 4$ and $j = 7$ ($n = 14$, $n - j + 1 = 8$):

$$a = a\ b\ c\ d\ \mathrm{g}\ \mathrm{h}\ a\ b\ c\ d\ p\ q\ r\ s$$
$$b = h\ i\ j\ k\ \mathrm{e}\ \mathrm{f}\ p\ q\ r\ s\ h\ i\ j\ k$$

So you will see that the equality is equivalent to:

1. The prefix $a[1 \dots i]$ must match the substring $a[j \dots j + i - 1]$
2. The suffix $a[j + i \dots n]$ must match $b[j \dots n - i]$
3. $b[n - i + 1 \dots n]$ must match $b[1 \dots i]$ (this means that i is a border of b)

Let:

$$\textit{prefix\_match}[i] = \text{length of the longest common prefix (LCP) of } a[1 \dots n] \text{ and } a[i \dots n]$$
$$\textit{suffix\_cross\_match}[i] = \text{length of the LCP of } \textit{reverse}(a)[1 \dots n] \text{ and } \textit{reverse}(b)[i \dots n]$$

For each fixed i, we need to first make sure there is a border of length i in b, and then find all j such that:

$$j \leq n - i$$
and
$$j + \mathit{prefix\_match}[j] \geq n - \mathit{suffix\_cross\_match}[i+1] + 1$$

We can solve this part using Z-function, offline processing, and Fenwick Tree in $O(n \log n)$.

Case 3: $i > n - j + 1$ (Prefix longer than suffix)

Instead of deriving symmetric logic again, we apply a key transformation.

If we reverse both strings:

$$a' = \mathit{reverse}(a), \quad b' = \mathit{reverse}(b)$$

Then:

$$(i > n - j + 1) \text{ on } (a, b) \iff (i < n - j + 1) \text{ on } (b', a')$$

So we can just reuse the above logic.

Problem J. C-Style String Length

Problem Setter: Shahriar Manzoor
Tester: Mahmodol Hassan Monna
Alt Writer: Irfanur Rahman Rafio
Category: String, Ad Hoc
Total Solved: 288
First to Solve: IUT_Kamikaze

We need to simulate how C's `strlen()` works after decoding escape sequences.

Since the string contains only \ and `0`, scan it from left to right:

- `0` → normal character, length $+= 1$
- \\ → one backslash, length $+= 1$
- \`0` → NUL character, stop immediately (do not count it)
- Trailing \ with no next character → **INVALID**

If a NUL is produced, output the current length. If the scan finishes without errors, output the total length; otherwise print `INVALID`.