Problem Analysis

The 2025 ICPC Asia Pacific Championship

This is an analysis of some possible ways to solve the problems of The 2025 ICPC Asia Pacific Championship. Since the purpose of this analysis is mainly to give the general idea to solve each problem, we left several (implementation) details in the discussion for reader's exercise. If some of the terminology or algorithms mentioned below are not familiar to you, your favorite search engine should be able to help. If you find an error, please send an e-mail to icpc-apac-judges@googlegroups.com about it.

Problem Title		Problem Author	Analysis Author
A	Control Towers	Ammar Fathin Sabili	Ammar Fathin Sabili
B	Three-Dimensional Embedding	Mitsuru Kusumoto	Mitsuru Kusumoto
C	Cactus Connectivity	Meng-Tsung Tsai	Jonathan Irvin Gunawan
D	Tower of Hanoi	Kai-Chun Ho	Shogo Murai
E	Minus Operator	Mitsuru Kusumoto	Mitsuru Kusumoto
F	Hold the Star	Fausta Anugrah Dianparama, Pikatan Arya Bramajati, Julian Fernando	Pikatan Arya Bramajati
G	Corrupted File	Pikatan Arya Bramajati	Pikatan Arya Bramajati, Nguyen Thanh Trung
H	Secret Lilies and Roses	Ammar Fathin Sabili	Ammar Fathin Sabili
I	Squares on Grid Lines	Pikatan Arya Bramajati	Pikatan Arya Bramajati
J	Gathering Sharks	Pikatan Arya Bramajati	Pikatan Arya Bramajati
K	Book Sorting	Brian Tsai, Bo-Wei Lin, Kuan-Lin Chen	Shogo Murai
L	Boarding Queue	Alham Fikri Aji	Jonathan Irvin Gunawan
M	Can You Reach There?	Kwee Lung Sin	Kwee Lung Sin, Nguyen Thanh Trung

The contest judges would also like to thank the following for their valuable feedback to the tasks: Bui Hong Duc, Masaki Nishimoto, Prof. Takashi Chikayama, Rama Aryasuta Pangestu, Riku Kawasaki, Vu Hoang Kien, William Gan, and Yui Hosaka.

A Control Towers

Here is one of many ways to solve this problem.

If two or more towers could be placed at the same cell, it is not hard to come up with a DP solution: $DP[k][i][j]$ denotes the number of ways to place the towers where k towers have been placed such that tower k is at (i, j). Naively, there will be $O(r + c)$ transitions for each state. These transitions could be replaced by doing some precomputations on the sum of each row and column of the placement of tower $(k - 1)$. Let $SUM(k)$ be the sum of $DP[k][i][j]$ for all i and j. $SUM(4)$ then should be the final answer.

With minor tweaks, we can force the DP to disallow adjacent towers to be located at the same cell. Our remaining job now is to eliminate the cases where the non-adjacent towers could be at the same cell.

Let A, B, C, and D be the locations of towers 1, 2, 3, and 4. We need to eliminate the cases where $A = C$, or $B = D$, or $A = D$. Recall that by the tweak mentioned previously, the adjacent towers cannot be located at the same cell. Therefore, the last case is independent (and cannot be mixed) to the first two.

For $A = C$ and $B = D$ cases, we can use inclusion–exclusion principle. If $A = C$ but $B \neq D$, it is equivalent to placing three towers instead of four. Same thing goes for $B = D$ but $A \neq C$. For $A = C$ and $B = D$ though, it is equivalent to placing two towers instead. Therefore, we can subtract the answer (of $SUM(4)$) by $2 \times SUM(3)$ and add it by $SUM(2)$.

Lastly for $A = D$ case, observe that A, B, and C must either be located at the same row or at the same column. Therefore, for each row and for each column, we can simply count the number of empty cells, namely p, and subtract the current answer by $p \times (p - 1) \times (p - 2)$.

Time complexity: $O(rc)$

B Three-Dimensional Embedding

In this problem, we are asked to embed a given graph into the cubic space $[0, 400]^3$. This is not straightforward, so let us first consider how to embed it in a differently sized box (not necessarily a cube).

Suppose that one vertex is placed at $(0, 2, 0)$. With a suitable construction, we can attach five polylines to that vertex so that their endpoints lie at $(0, j, 0)$ for $j = 0, 1, \ldots, 4$. We call this arrangement a *gadget*. We can center such a gadget at any position, not necessarily at $(0, 2, 0)$.

Step 1: $3n \times 6 \times m$

One possible embedding is to line up n gadgets horizontally and then make connections between these gadgets in each z-layer (a plane parallel to the xy-plane).

Place the i-th gadget so that its center is at $(1 + 3i, 2, 0)$ for $i = 0, 1, \ldots, n - 1$. For each edge connecting vertices v and w in the original graph, choose one endpoint from the v-th gadget and one endpoint from the w-th gadget, and connect them as follows:

- Suppose we use the z-layer $z = z_0$ for $z_0 \geq 1$.
- From the chosen endpoint in the v-th gadget, extend along the z-axis to $z = z_0$, shift by -1 in the

x-direction, and then shift up to 5 in the y-direction.

- Do the same for the chosen endpoint in the w-th gadget.
- In the plane $z = z_0$, join these two extended endpoints with a straight segment.

By assigning distinct z-layers to different edges, we avoid intersections. This produces an embedding in a box of size $3n \times 6 \times m$.

Step 2: $3n \times n/2 \times 19$

We next aim to fit the graph into a more compact (and closer to cubic) space by reducing the number of z-layers used.

Let M be a matching in the graph, meaning that each vertex is incident to at most one edge in M. We will use two consecutive z-layers to embed as many edges as possible. Suppose z_0 and $z_0 + 1$ are two consecutive layers. For the j-th edge (v, w) in M, do the following:

- From one endpoint in the v-th gadget, extend along the z-axis to $z = z_0$, shift by -1 in x, shift to $j + 5$ in y, and then continue the extension by $+1$ in z.
- Do the same for one endpoint in the w-th gadget.
- Connect these polylines in the plane $z = z_0 + 1$.

Polylines produced in this way do not intersect. With a greedy edge-coloring of the graph, we obtain 9 matchings, each embedded by the above method. Thus, we get an embedding in a box of size $3n \times n/2 \times 19$.

Step 3: $9\sqrt{n} \times 3\sqrt{n} \times 7\sqrt{n}$

Finally, we arrange the gadgets in a rectangular grid to further balance the dimensions.

Let d be an integer to be chosen later. Label each gadget as (i, j) for $0 \le i < n/d$ and $0 \le j < d$. Place gadget (i, j) so that its center is at $(1 + 3i, 2 + 5j, 0)$. This arrangement ensures that none of the gadgets overlap.

We then form a matching in a *contracted* graph: for each i, we contract the vertices corresponding to the gadgets $(i, 0), (i, 1), \ldots, (i, d-1)$. The contracted graph has n/d vertices, each of degree at most $5d$. Using a greedy edge-coloring again, we obtain $10d - 1$ matchings. Each pair of gadgets in different columns $(i, \cdot)$ and $(i', \cdot)$ is connected by the method from Step 2.

For gadgets in the same column, say (i, j) and (i, j'), we connect them in a single z-layer $z = z_0$ as follows:

- From an endpoint in gadget (i, j), extend up to z_0 and **shift by** $+1$ **in** x.
- Do the same for gadget (i, j').
- Connect these two extended endpoints in the plane $z = z_0$.

These polylines do not interfere with those connecting gadgets across different columns.

To decide d, note the required space:

- x-dimension: $3n/d$,
- y-dimension: $5d + n/2d$,
- z-dimension: $20d$.

Setting $d = \sqrt{n}/3$ yields a final embedding in a box of size $9\sqrt{n} \times 3\sqrt{n} \times 7\sqrt{n}$, which fits within $[0, 400]^3$ for $n \leq 1600$ because $9\sqrt{1600} = 360 < 400$.

Overall, the time complexity of this construction is $O(n\sqrt{n})$.

C Cactus Connectivity

For any graph G, denote the connectivity value and max-cut of the graph as $cv(G)$ and $mc(G)$, respectively. We are going to prove that $cv(G) = mc(G) + 1$.

Lemma 1. $cv(G) > mc(G)$

Proof. Let S and T be a partition of vertices of G such that the value of $S-T$ cut is the maximum cut. Create a graph H by adding more edges and vertices to each of S and T, so that they each become a clique with at least $(mc(G) + 1)$ vertices. H is a $mc(G)$-edge-connected supergraph of G, but removing all the edges of G from H leaves S and T disconnected. Therefore, $mc(G)$ is not sufficient. □

Lemma 2. $cv(G) \leq mc(G) + 1$

Proof. We are going to prove this by contradiction. Let us assume that $cv(G) > mc(G)+1$. This means that there exists an $(mc(G) + 1)$-edge-connected supergraph of G such that removing all the edges of G from it leaves a disconnected graph. Let that supergraph be F, and A and B be any two connected components of the disconnected graph. The value of $A - B$ cut in F must be at least $(mc(G)+1)$, since otherwise F would not be a $(mc(G) + 1)$-edge-connected graph. Also, since removing all the edges of G from F leaves A and B disconnected, all the edges connecting A and B must also be present in G. This means there exists a cut in G with a value more than $mc(G)$. This is a contradiction. □

Therefore, the crux of this problem is to find the max-cut of the given graph. While finding a max-cut of a general graph is an NP-hard problem, fortunately, there is an easy solution to find a max-cut of a cactus.

Each of the connected components of the given cactus can be solved separately. For each connected component, find any spanning tree. Greedily bicolor the tree so that each of the tree edges connects two vertices of different colors. All the tree edges count toward the value of the max-cut. For each of the non-tree edges, increment the value of the max-cut if they also connect two vertices of different colors.

This solution runs in $O(n + m)$ time.

D Tower of Hanoi

First, consider the solution for the single-query case. Instead of putting the disks back into the rod 1, consider moving all disks from rod 1 to their initial stack, which we now regard as the desired stack. If the largest disk is on the rod 1, it does not need to move. Otherwise, if it needs to move to the rod 2 or 3, all other smaller disks need to move first to the rod 3 or 2, respectively. We can show by induction that moving x disks from a stack to another takes $2^x - 1$ moves. After the largest disk has been moved to its desired stack, consider whether the second-largest disk needs to move or not, and so on.

Then we go back to the original setting in which there are several queries. For a block (consecutive subsequence) of disks, suppose we want to move all disks in the block to the rod i. Then moving all disks in the block requires some steps. Also, suppose that one or more smaller disks are stacked on top of the disks in the block. In order to move all these disks to the rod i, those smaller disks must be first moved to a specific rod, whose index is uniquely determined by the target rod i and the arrangement of the disks in the block. To efficiently process the queries, we maintain two values for a block ($i = 1, 2, 3$): the number of steps required to move all the disks in the block to the rod i, and the rod index where smaller disks first must be moved.

Given two consecutive blocks, we can compute the values for the block formed by concatenating them. Now we can use a segment tree to efficiently computing these values for any (consecutive) subsequences of disks. The total time complexity is $O(n + q \log n)$.

E Minus Operator

In a standard context, the expression $a - b - c$ is interpreted as $(a - b) - c$, reflecting the *left-associative* nature of the minus operator. In the following, we omit parentheses wherever possible for simplicity. For instance, the expression

$$(((x - x) - (x - x)) - (x - x))$$

can be written more compactly as

$$x - x - (x - x) - (x - x).$$

Throughout this simplified representation, parentheses around a variable can only appear in one of the following forms: "$-(x-$", "$-x-$", "$-x)-$", "$-x))-$", "$-x)))-$", and so on. Note that "$-((x-$" cannot appear, because that would make one pair of parentheses redundant.

Nested Case

As a special case, consider the situation where $n = 6$, and the expression takes the form

$$x - (x - (x - (x - \ ? \ \ x \ \ ? \ \ - \ \ ? \ \ x \ \ ? \ \ ))),$$

where "?" indicates unknown parts. Suppose we specifically want to identify how the second-to-last variable is parenthesized. The possible forms are "$(x$", "x", "$x)$", "$x))$", or "$x)))$". To distinguish these forms, the following queries are effective: $S = 111101, 111001, 110001,$ and 100001. Here is a table of the evaluated results under each query:

	111101	111001	110001	100001
$(x$	0	1	0	1
x	1	1	0	1
$x)$	0	0	0	1
$x))$	1	1	1	1
$x)))$	0	0	0	0

From these results, we can devise the following query strategy to identify the parentheses around that variable: First, query $S = 111101$.

- If the response to the query is 0, the possibilities are "$(x$", "$x)$", or "$x)))$". Next, query $S = 111001$. If the response is 1, the form must be "$(x$". Otherwise, query $S = 100001$.
- If the response to the query is 1, then the possibilities are "x" or "$x))$". Next, query $S = 110001$ to distinguish between these two.

General Case

While the previous section focuses on a specific case, the same reasoning can be generalized. Assume we aim to determine the parenthesis patterns for each variable from left to right. For the leftmost variable, there are obviously no parentheses. For the second variable, we might have either "x" or "$(x$", and so forth.

When investigating the parentheses around the i-th variable from the left, suppose we already know the parenthesis arrangement to its left. We can then construct a query in which the variables that directly contributes to the depth of the i-th variable are substituted with 1. For instance, consider a partially known expression:

$$x-(x-x)-(x-(x-x)-(x-x)-(x-(x-(x-x)-x-(x-\quad ?\quad x\quad ?\quad x.$$

We would build a corresponding query string so the result looks like

$$1-(0-0)-(1-(0-0)-(0-0)-(1-(1-(0-0)-0-(1-\quad ?\quad x\quad ?\quad x,$$

which simplifies to

$$1-(1-(1-(1-(1-\quad ?\quad x\quad ?\quad x.$$

This closely mirrors the situation from the nested case above. Using the same method: "$(x$" and "x" can be identified in 2 queries, "$x)$" in 3 queries, "$x))$" in 3 queries, "$x)))$" in 4 queries, "$x))))$" in 4 queries, and so on.

Query Complexity

Now, let's estimate the worst query complexity of the method. Let t be the total number of "$(x$" instances in the expression. Also, suppose k variables appear in the forms "$x)$", "$x))$", etc., and let the j-th of those k variables close r_j parentheses, for $j = 1, \ldots, k$. Note that $r_1 + r_2 + \cdots + r_k = t$.

The total number of queries used by this strategy is

$$2n + \sum_{j=1}^{k} \left\lceil \frac{r_j}{2} \right\rceil .$$

Analyzing this sum requires some care. Let s_0 be the sum of the even r_j values, and s_1 be the sum of the odd r_j values. Also, let k_1 be the number of odd r_j. Then

$$\sum_{j=1}^{k} \left\lceil \frac{r_j}{2} \right\rceil = \frac{s_0}{2} + \frac{s_1 + k_1}{2} = \frac{t}{2} + \frac{k_1}{2} \leq \frac{n}{2},$$

where we used $s_0 + s_1 = t$ and $t + k_1 \leq n$. Hence, the overall method requires at most $2n + \frac{n}{2} = 2.5n$ queries.

F Hold the Star

We can separate the levels into two cases: the case where the star is initially to the left of character m, and the case where it's initially to the right. Let's just solve for the latter case. Doing the other case is exactly the same, with just mirroring the room configuration.

Let's consider just one level. The star is initially to the right of character m. It's not hard to see that it's always optimal to always move the star to the left, and never move the star to the right. However, it's possible for a character to move to the right to pick up the star.

Consider the moving characters who take part in delivering the star to character m in a single level. We can divide these moving characters into only four types:

- Type 1: While in his/her initial position, he/she receives the star, and then he/she moves to the left to move the star to another position.
- Type 2: He/she moves to the star's current position, picks up the star, and then does nothing.
- Type 3: He/she moves to the right to the star's current position, and then he/she moves to the left to move the star to another position.
- Type 4: He/she moves to the left to the star's current position, and then he/she moves to the left to move the star to another position.

Using greedy observations, we can obtain multiple properties of the optimal strategy.

- The only possible type 2 character is character m.
- The only possible type 3 character is the first character that moves the star.
- The only possible type 4 character is the first character that moves the star.
- If character m is type 2, then no other character takes part in delivering the star.

Using all properties above, there are only four possible cases for the sequence of types of characters taking part in delivering the star:

1. $[2]$

2. $[3, 1, 1, 1, \ldots, 1, 1, 1]$
3. $[4, 1, 1, 1, \ldots, 1, 1, 1]$
4. $[1, 1, 1, 1, \ldots, 1, 1, 1]$

Notice that type 3 and type 4 also includes type 1. Handling the second and third cases automatically handles the fourth case, so the fourth case can be ignored.

For the second and third cases, it's not hard to see that it's always optimal for the values of s_i to be decreasing over time. For that, we first sort the characters based on their positions from left to right. If we only consider the characters between room r_m and room l_j (inclusive), and we calculate the suffix minimum of s_i, then optimally, the trailing characters of type 1 taking part in this are only the characters that appear in the suffix minimum.

That means, for some l_j, if we have decided which character x we want to be type 3 or type 4, we can know his/her value of s_x, and then the trailing type 1s are just the characters in the suffix minimum with values of s_i smaller than s_x.

Now, let's solve the first, second, and third cases for all levels simultaneously where l_j is to the right of r_m. We can sort the levels based on increasing values of l_j.

For the first case, we can do a simple $O(1)$ math for each l_j.

Now, let's consider the second and third cases. One hard thing is that the suffix minimum changes as l_j moves to the right, because more characters are considered to be a potential type 1. We can handle the changes in the suffix minimum by maintaining the suffix minimum using a stack as l_j gets bigger.

For each character x to the right of r_m, we calculate $dp[x]$ as the cost if the trailing type 1s start from character x. We can calculate these values as we're simulating the suffix minimum stack.

Let's solve for the type 4 character first. Let's say it's character x. He/she must be located to the right of l_j. It can be obtained that finding the optimal type 4 for some l_j is just finding the minimum value of $dp[x]$ for all characters x located to the right of l_j. This can be solved by precomputing the suffix minimum of $dp[x]$.

Now, let's solve for the type 3 character. Let's say it's character x. He/she must be to the left of l_j. It can be obtained that in the optimal strategy:

- If the optimal character x satisfies $r_x > r_m$, then during the time character x goes to the right to pick up the star and goes to the left to deliver the star to the trailing type 1s, he/she must revisit his/her initial position along the way.
- If the optimal character x satisfies $r_x \leq r_m$, then character x only picks up the star and delivers it directly to character m, without the help from other characters.

Because of that, we can calculate the total cost for each possible candidate character x as the following:

- If $r_x > r_m$, its total cost is $dp[x] + 2 \times (l_j - r_x) \times s_x$
- If $r_x \leq r_m$, its total cost is $(2 \times l_j - r_x - r_m) \times s_x$

We can see that for both cases, the cost is linearly dependent on l_j. Because of that, the total cost for each possible candidate character can be made into a line equation with l_j being the variable. To find the optimal line for every given value of l_j efficiently, we maintain these lines using convex hull trick. As we move l_j to the right, we incrementally add more lines to our convex hull. The optimal cost for each level can be calculated using a binary search on these maintained lines.

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

G Corrupted File

Notice that we can "combine" operations: if we pick two bits i and $i+1$ and replace it, and later pick the replaced bit i and $i+1$ (originally at position $i+2$), we can think of it as one big operation: pick bits i, $i+1$, $i+2$, and transform them into 1 if all of the bits are 1, or 0 otherwise.

From this observation, we can rephrase the problem as: Given a bit sequence B. You can select several disjoint subarrays, and replace each subarray with 1 if all the bits in it are 1, or 0 otherwise.

We decompose B into segments, where each segment only contains either bit 0 or bit 1, e.g. for the given bit sequence 11011011, we decompose it into $[11, 0, 11, 0, 11]$.

Notice that doing an operation on a subarray that's completely within a segment is just reducing the length of a segment into any positive length.

What happens if you do an operation on a subarray that contains elements from two or more segments? Since 0 must be present in the subarray, it will be replaced by 0. This operation can be used to delete segments with value 1.

From the observations above, by doing the operations, we can do the following:

- Reduce the length of a segment with value 0 into any positive length.
- Reduce the length of a segment with value 1 into any positive length.
- Combine several adjacent segments into a big segment of value 0, effectively deleting some segments with value 1 in the middle.

We solve the problem greedily: For each i between 1 and number of segments in C, we want to find the minimum prefix of segments in B, which we can transform into segments $C_1, \ldots, C_i$. We iterate each segment in C from left to right while maintaining a pointer in B representing the currently used prefix.

Let denote by c_i the value of segment C_i and k_i its length.

- If $c_i = 0$, we iterate the prefix pointer until it has visited k_i values of 0.
- If $c_i = 1$, we iterate the prefix pointer until we find a segment in B with value 1 and length of at least k_i.

If at any point the prefix pointer goes outside B, then it is impossible to transform B into C.

Also, since we cannot make a segment of value 0 disappear, if the first (or last) segment of B has value 0 while the first (or last) segment of C has value 1, then it is impossible.

Otherwise, it is possible.

Time complexity: $O(m+n)$.

H Secret Lilies and Roses

If you are nowhere to the solution, take this hint: "What if you already know how many roses there are?" If you cannot go further, this is another hint: "Once you know how many roses there are, you do not need to make any query again!"

Indeed, if there are k roses, then the answer is simply "`answer` k". The proof is simple. If there are l_k lilies among the leftmost k flowers, that means there are $k - l_k$ roses among them. Therefore, the number of roses among the other flowers (a.k.a. r_k) will be $k - (k - l_k) = l_k$. This also proves that the answer actually always exists for any configuration of the flowers.

So now we have reduced this problem into finding the number of roses among the n flowers. At a glance, this might be easily solved with a binary search by finding the smallest j such that the "`multi` j" is a non-zero value. However, this strategy will not work as we can have the following flower configurations (with R's denoting roses and L's denoting lilies):

$$[RRR\cdots RRR]\ L\ [\text{any}]\ R\ [LLL\cdots LLL]$$

Particularly, the result of making a multiply query on the first block of roses or the last block of lilies is always 0. Only if we make a multiply query on the "middle" block part will result a non-zero response, and note that this part could be very narrow.

One possible solution is to use a binary search with the type query instead: We go to the right if it is rose, and left if it is lily. Assuming that there is a "dummy" rose flower as flower 0 and a "dummy" lily flower as flower $n+1$, this binary search allows us to find any position x separating a rose and then a lily (i.e. flower x is rose and flower $x+1$ is lily).

Finally, we first make "`multi` x" query to get $l_x \times r_x$. And then, we may make "`multi` $x-1$" and "`multi` $x+1$" queries, as the first will definitely increase the number of the roses on the right (i.e. $l_{x-1} \times r_{x-1} = l_x \times (r_x + 1)$) and the second will definitely increase the number of the lilies on the left (i.e. $l_{x+1} \times r_{x+1} = (l_x + 1) \times r_x$). With these three equations, it is not hard to find both values of l_x and r_x even if any of them is 0, and finally know the number of roses among the n flowers. But actually, asking two of these three equations is already enough to determine the two variables.

Query complexity: $\lceil \log_2 n \rceil + 2$

I Squares on Grid Lines

Define the following:

- Corner point: A point that lies on a corner of a cell.
- Horizontal point: A point that lies on a horizontal side of a cell and is not a corner point.
- Vertical point: A point that lies on a vertical side of a cell and is not a corner point.
- Center point: A point that's the center of a cell's square shape.
- Outer border: The perimeter of the $n \times n$ grid.
- Unlimited area: A value for the square area such that there are an infinite number of valid squares with that area.

First, notice that for a square whose vertices are all corner points and not all of which are at the outer border, its area is an unlimited area.

Next, we can obtain that for a square whose area isn't an unlimited area, there are only two cases:

1. Its vertices in order are a horizontal point, then a vertical point, then a horizontal point, then a vertical point.
2. All of its vertices are integer points at the outer border.

We can prove this by seeing that if any two consecutive square vertices are both horizontal or both vertical, it turns out that that would only occur in a square that's able to be translated along the grid lines, which would result in an unlimited area if there's still room to translate it in the grid.

From those aforementioned two cases, we obtain that every single possible square whose area isn't an unlimited area must have a center at a corner point or a center point.

Let's calculate for every possible valid square whose center is at a corner point. One corner point with the most number of possible squares is point $(\lfloor \frac{n}{2} \rfloor, \lfloor \frac{n}{2} \rfloor)$. Let's call that point the pivot. Consider every possible valid square for that pivot. Consider one quadrant of the plane that contains every point (x, y) satisfying $0 \leq x \leq \lfloor \frac{n}{2} \rfloor$ and $0 \leq y < \lfloor \frac{n}{2} \rfloor$. Exactly one vertex of the square must lie inside that quadrant. We can use that vertex as a unique identifier. If the Euclidean distance from the vertex in that quadrant to the pivot is d, then the square's area is $2d^2$.

We can iterate over every single line segment that's a horizontal or vertical side of a cell in that quadrant. For each line segment, we can calculate the lower-bound and upper-bound for the area of a square whose bottom left vertex lies on that segment. In other words, we calculate the interval of possible areas for each line segment.

Remember that we've only been calculating for squares whose centers are at the pivot. However, we can use these squares to calculate for squares centered at other corner points. For each line segment we calculated previously, we can determine how many possible corner points that the pivot can be translated to such that the square doesn't go outside the outer border. We determine that number based on how far away the line segment is to the pivot.

While using this pivot, we can also calculate the areas of squares whose bottom left vertex is a corner point in this quadrant to account for unlimited areas obtained from squares whose centers are corner points.

For valid squares whose centers are center points, we calculate area intervals and unlimited areas again using a slightly different pivot $(\lfloor\frac{n-1}{2}\rfloor+\frac{1}{2},\lfloor\frac{n-1}{2}\rfloor+\frac{1}{2})$. This time, the quadrant cuts off some cell sides in half, so its line segment must be cut off accordingly.

After doing all that, we get every single area interval for every line segment, each with its number of possible translated squares. We can do a line sweep through these intervals while also iterating the queries in ascending order to get the total number of squares for every query. Don't forget to handle the case where a queried area is among the calculated unlimited areas.

The number of line segments we need to iterate is $O(n^2)$. The number of unlimited areas is $O(n^2)$.

Time complexity: $O((n^2+q)\log n)$

J Gathering Sharks

Let's renumerate the sharks such that shark i is the shark initially in the group numbered i. We calculate a new array pos, where $pos[i]$ is the location of shark i.

Executing a command is like merging two adjacent elements. Suppose the elements at indices x and y ($x<y$) are merged. Its merged result will have a position of $pos[x]$ and an index of y. Notice that, after the merge, its relative order with the rest of the elements remains the same, so its exact new index doesn't matter. We can see this operation as the element at index x erasing the element at index y, with a cost of $|pos[x]-pos[y]|$.

We can solve this problem using dynamic programming. Let $dp[l][r]$ ($l\le r$) be the minimum time required to have all sharks from $l+1$ to r get erased by shark l. The base case is $dp[l][r]=0$ if $l=r$.

Let's figure out the transitions. For some pair (l,r) ($l\le r$), let's consider only the sharks from l to r. Consider the sharks among them that are directly erased by shark l. Let p be the last such shark to be erased. Then it must hold that:

- Every shark from $l+1$ to $p-1$ must be erased by sharks between l and $p-1$.
- Every shark from $p+1$ to r must be erased by sharks between p and r.

That means, if we fix some value of p, the minimum total time is equal to $dp[l][p-1]+dp[p][r]+|pos[l]-pos[p]|$.

For each $l\le r$, we brute force every single possible value of p ($l+1\le p\le r$) to get the minimum time.

Time complexity: $O(n^3)$

K Book Sorting

Let the sequence $a_1,a_2,\ldots,a_n$ be the "inverse permutation" of p. That is, $p_{a_i}=i$ for $i=1,\ldots,n$. Also, let $\mathrm{inv}(x_1,\ldots,x_m)$ denote the inversion number of the sequence $x_1,\ldots,x_m$ (number of pairs of indices $1\le i<j\le m$ such that $x_i>x_j$).

We first consider what the optimal strategy for sorting the books looks like. We can observe that, in an optimal strategy, each book is either

- moved to the leftmost position exactly once,
- moved to the rightmost position exactly once, or
- swapped with an adjacent book zero or more times.

Then, using two numbers $1 \leq l \leq r \leq n$, an optimal strategy is represented as follows:

- Move books labeled $l-1, l-2, \ldots, 1$ to the leftmost position, in this order.
- Move books labeled $r+1, r+2, \ldots, n$ to the rightmost position, in this order.
- Sort books labeled $l, l+1, \ldots, r$ using swap actions.

Here the total number of operations equals $l - 1 + n - r + \mathrm{inv}(a_l, a_{l+1}, \ldots, a_r)$. This can be computed by finding the maximum value of $f(l, r) = r - l - \mathrm{inv}(a_l, a_{l+1}, \ldots, a_r)$.

Let $S_r = \{1 \leq i \leq r \mid \forall i < j \leq r, f(i, r) > f(j, r)\}$. We note that $\{r\} \in S_r$ holds. Suppose we sort the values in S_r in descending order: $r = s_{r,1} > s_{r,2} > \cdots > s_{r,k_r}$. Then we can show $f(s_{r,j}, r) = j - 1$. It follows from $f(r, r) = 0$ and $f(s_{r,j+1}, r) - f(s_{r,j}, r) = 1$ (otherwise, another index between $s_{r,j+1}$ and $s_{r,j}$ would have been present in S_r). Thus, if we can track S_r for each r, we can easily compute the answer.

Here, let us consider how S_{r+1} can be constructed from S_r.

- If a_{r+1} is larger than all of $a_1, \ldots, a_r$, $f(l, r+1) = f(l, r) + 1$ for all l, thus $S_{r+1} = S_r \cup \{r+1\}$.
- Otherwise, we handle the effect of "inversions" one by one for each l with $a_l > a_{r+1}$. A single inversion $a_l > a_{r+1}$ reduces the value of $f(1, r+1), f(2, r+1), \ldots, f(l, r+1)$ by 1 compared with the case in which this inversion had not been present. We can see that this eliminates the largest element in S_{r+1} that is not larger than l.

In the overall process of computing $S_1, \ldots, S_n$, each index r will be added to S exactly once and removed at most once. Elimination of the largest element that is not larger than l can have no effects if all the elements in S is larger than l. However, in this case, we can ignore such l in the subsequent elimination processes to keep the runtime efficient.

If we maintain S and elimination candidates using balanced binary trees (such as `std::set`), the overall algorithm runs in $O(n \log n)$ time.

L Boarding Queue

For each i ($1 \leq i \leq p$), consider what the state of the boarding queue would be by the time you occupy traveler i's initial location. There would have been $p - i$ boarding steps. Travelers 1 to $p - i$ would have left the boarding queue, while traveler x (for $x > p - i$) would occupy traveler $(x - (p - i))$'s initial location.

Therefore, we can simply solve this problem by iterating i from 1 to p. Consider all the travelers j who are initially adjacent to traveler i. If $j + (p - i) \leq N$, then traveler $j + (p - i)$ will be adjacent to you after $p - i$ boarding steps. Insert $j + (p - i)$ to a data structure.

After all p iterations, print the number of distinct integers in the data structure. With a data structure that supports constant-time insertion, this solution runs in $O(r \times c + n)$ time.

M Can You Reach There?

We can solve this problem by considering several cases:

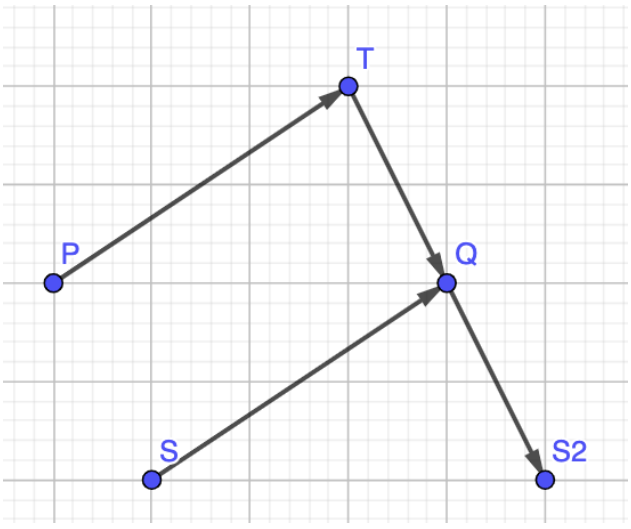
1. $n = 1$,
2. $n = 2$,
3. $n \geq 3$ and not all marked points are on one line, and
4. $n \geq 3$ and all marked points are on one line.

M.1 $n = 1$

From the starting point S and a marked point P, let R be the reflection of S with respect to P. We can show that we can reach any point on the segment SR.

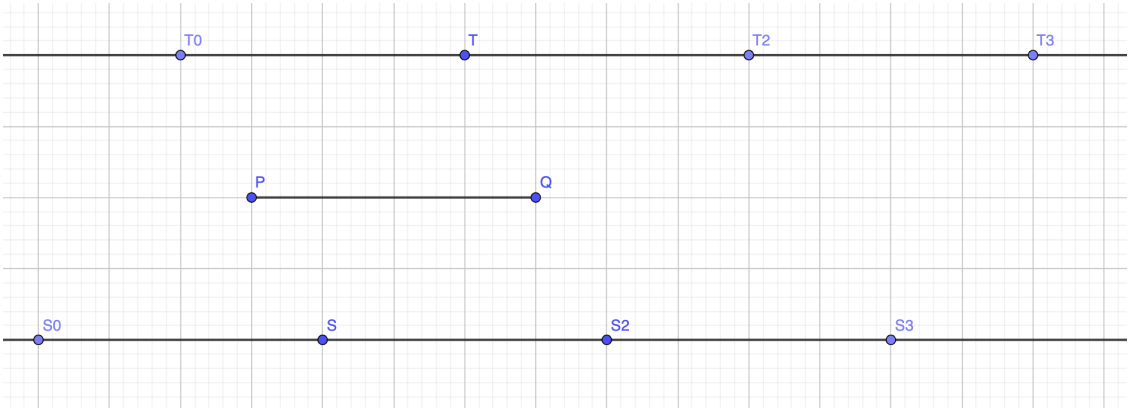
M.2 $n = 2$

Let P and Q be the two marked points. Consider the illustration below:

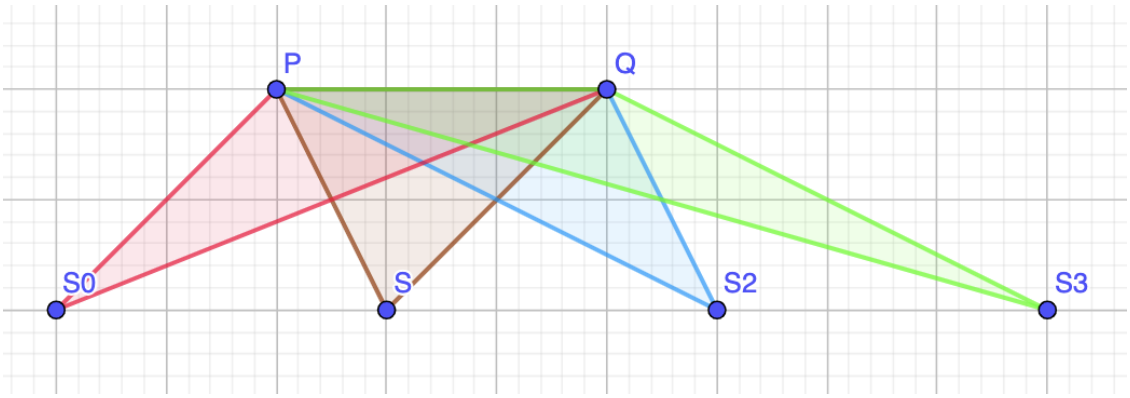


We can move from S to T and from T to S_2. We can prove that $\overrightarrow{PQ} = \overrightarrow{SS_2}$, i.e. we can effectively translate any arbitrary point X with vector $\overrightarrow{PQ}$. Similarly, we can also translate any arbitrary point X with vector $\overrightarrow{QP}$.

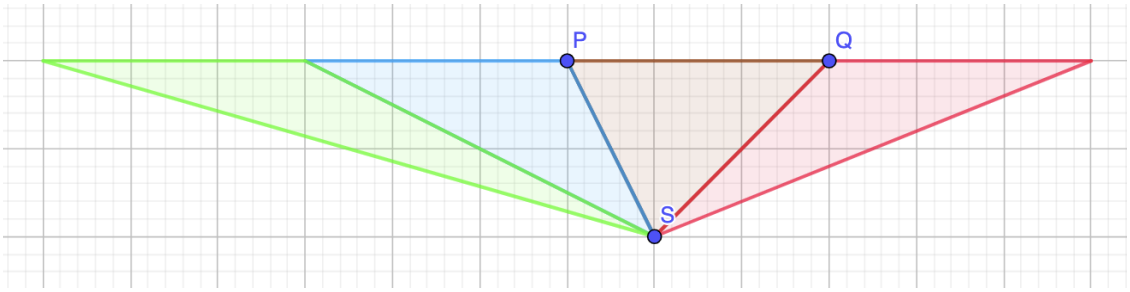
Let's consider the 2 lines: l_1 which goes through T and parallel to the segment PQ, and l_2 which goes through S and parallel to the segment PQ. On these two lines, we can visit all the points that that are translation of S (or T) by a multiple of vector $\overrightarrow{PQ}$. It can be proven that no other points on l_1 and l_2 can be visited.



It is trivial to show that the points "outside" of the region between lines l_1 and l_2 cannot be reached. How about the points "inside" the region between lines l_1 and l_2?



From point S_i, we can visit all the points inside triangle S_iPQ. We can also translate the points in each of the triangles S_iPQ by a multiple of vector $\overrightarrow{PQ}$, more precisely, translate triangle S_iPQ by vector S_iS:



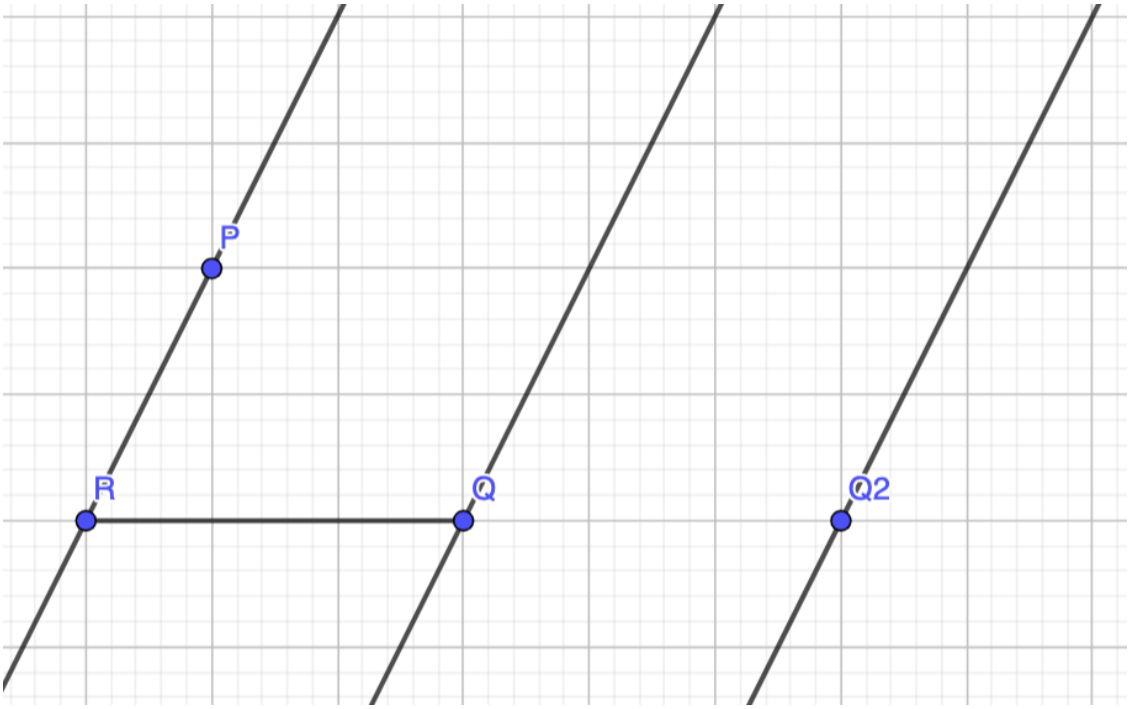
From this, we can show that all the points between lines l_1 and l_2 (exclusive) are reachable.

Note that there is a corner case where S or T is on the line PQ.

M.3 $n \geq 3$

When all marked points are collinear, we can solve it similarly to case $n = 2$.

When not all marked points are collinear, let P, Q and R be three non-collinear marked points:



From any starting point S, we can reach point R (using marked point R). Let l_0 be the line going through P and Q, and l_1 be the line going through Q and parallel with l_0. Using marked points P and Q, we can reach all points between lines l_0 and l_1, excluding line l_1.

Let Q_2 the translation of Q by vector $\overrightarrow{RQ}$, and let l_2 be the line going through Q_2. Using marked points R and

Q, we can translate any point by vector $\overrightarrow{RQ}$. Thus, we can reach all points between lines l_1 and l_2 (excluding line l_2).

Continue to apply the above translation, we can show that we can reach every point in the plane.