# Battle of the Brains Fall 2024 Editorial

Matthew Sheldon, Suraj Mathashery, Mike Nguyen, Quang Ngo,
Dr. Chitturi, Darrin Wiley

October 2024

# Bishop Attack

**Explanation**

There are two components to this problem. The first is determining whether the two bishops are on the same color square. Determining which color square a piece is on can be efficiently calculated as follows:

```
1: function IsOnWhiteSquare($r$, $c$)
2:     return $(r + c)$ % $2 = 1$
```

Note that if both $A$ and $B$ are on different colored squares, then it is impossible. So performing the logical XOR ($\oplus$) of the two pieces is sufficient to determine if they are on different colored squares. This can be calculated in $\mathcal{O}(1)$ time.

Then, we can perform a Breadth-First Search (BFS) to see if it is possible to reach $B$ from $A$. Note that there exists an edge $e$ from $u$ to $v$ in graph $G$ if and only if $u$ and $v$ are on the same diagonal, and $P$ does not lie in between $u$ and $v$. Since we have a constant $8 \times 8$ board, we can note that

$$\begin{aligned} |V| &= 8 \cdot 8 = \boxed{64} \\ |E| &\leq 8^3 = \boxed{512} \end{aligned} \tag{1}$$

BFS takes $\mathcal{O}(V + E)$ time, but since both $V$ and $E$ are not dependent on the input size and are fixed, they can be treated as constants. Thus, this can be performed in $\mathcal{O}(1)$ time.

**Time Complexity**:

$$\mathcal{O}(1) + \mathcal{O}(1) = \boxed{\mathcal{O}(1)}$$

# Two Brains Are Better Than One (Easy)

**Explanation**
Note that the term "cool" really refers to the fact that a number is a perfect square. This can be observed as the only time that a number has an odd number of factors is when one is repeated, which, by definition, makes it a perfect square.

Then, simply iterate over the ratings, determining if they are perfect squares or not, filtering out those that are. This takes $\mathcal{O}(n)$ time as determining whether a number is a perfect square is a constant time operation.

Lastly, iterate over every unique pair of non-perfect-square ratings, determining if their product yields a number which is a perfect square. Let $m$ be the number of non-perfect-square ratings, and $n$ be the number of Comet ratings. The following is an analysis of the time complexity of this step:

$$\sum_{i=1}^{m} i \leq \sum_{i=1}^{m} m \leq \sum_{i=1}^{n} n = n^2 \therefore \mathcal{O}(n^2) \tag{2}$$

Thus, the total running time analysis can be calculated as follows:

**Time Complexity**:

$$\mathcal{O}(n) + \mathcal{O}(n^2) = \boxed{\mathcal{O}(n^2)}$$

# Two Brains Are Better Than One (Hard)

**Explanation**
Since the bounds on the problem have been increased fairly significantly, we need to make some additional observations in order to create a more efficient solution to this problem:

**Observation 1:** We can redefine what it means for a number to be a perfect square
We can note that a number $r$ is a perfect square if and only if all of the exponents in the prime factorization of $r$ are even. More formally...

$$\begin{gathered} r = p_1^{e_1} \cdot p_2^{e_2} \cdot \ldots \cdot p_k^{e_k} \text{ is a perfect square} \\ \Leftrightarrow \\ \forall i,\ 1 \le i \le k \ :\ (p_i \text{ is prime}) \land (e_i \bmod 2 = 0) \end{gathered} \tag{3}$$

**Observation 2:** We can redefine what it means for a team to be "cool"
When pairing two Comets with ratings $r_1 = p_1^{e_1} \cdot p_2^{e_2} \cdot \ldots \cdot p_k^{e_k}$ and $r_2 = p_1^{f_1} \cdot p_2^{f_2} \cdot \ldots \cdot p_k^{f_k}$, we can note that the product of their ratings $R = r_1 \cdot r_2 = p_1^{e_1+f_1} \cdot p_2^{e_2+f_2} \cdot \ldots \cdot p_k^{e_k+f_k}$ must be a perfect square in order to be considered "cool". Thus, we can note that $R$'s prime factorization must also only contain even exponents. Specifically, this will only happen when the odd exponents in $r_1$ and $r_2$ "cancel" each other out.

**Observation 3:** When two ratings $r_1$ and $r_2$ cancel each other out
Suppose we filtered $r_1$ and $r_2$ down to the product of only those pairs of prime numbers raised to exponents where the exponent is odd. Hence, the prime numbers and their corresponding exponents that cause us to not be able to evenly split $r$ into $\sqrt{r} \cdot \sqrt{r}$. Let us denote this filtered version as...

$$F(r) = \prod_{i=1}^{k} p_i^{e_i \bmod 2} \tag{4}$$

Therefore, we can note that $F(r_1)$ must be equivalent to $F(r_2)$ for $r_1$ and $r_2$ to cancel each other out. Consider if $F(r_1) \neq F(r_2)$. This would mean that there exists at least one prime factor that has an odd exponent in one rating but not in the other (thus the other one has an even number of that prime factor). Since the sum of an odd number and an even number is an odd number, these two values would not cancel each other out as we would be unable to evenly split $R$ into $\sqrt{R} \cdot \sqrt{R}$. Thus, we can conclude that $F(r_1) = F(r_2)$ is sufficient and necessary for $r_1$ and $r_2$ to cancel each other out.

**Observation 4:** Calculating the number of "cool" teams
Suppose we calculated the frequency for all values of $i$ where $i = F(r_j)$ for all $1 \le j \le m$, where $m$ is the number of ratings that are not perfect squares. Denote an individual frequency as $F_i$. Let $k$ be the number of unique values of $F_i$. Thus, the number of unique "cool" teams that can be formed is as follows:

$$\sum_{i=1}^{k} \binom{F_i}{2} = \sum_{i=1}^{k} \frac{F_i(F_i - 1)}{2} \tag{5}$$

**Time Complexity**:

$$\mathcal{O}(n) + \mathcal{O}(m\sqrt{r_{max}}) + \mathcal{O}(k) \le \boxed{\mathcal{O}(n\sqrt{r_{max}})}$$

# Cylinder Volume

**Explanation**
We can note that a solution to this problem exists if we have at least $1 + 2 + \cdots + k$ total height and radii. If there is leftover height and/or radii, in order to add an additional height/radii to the topmost cylinder, then we must have at least $k$ more of either height or radii since we must not invalidate the constraint of the problem. With this in mind, there are a couple of important observations to make with regard to this problem:

**Observation 1:** The bounds on $h$, $r$, and $k$ are all really large.
This would indicate that we need to store them with 64-bit integers, rather than 32-bit integers. Since $1 \leq k \leq 10^9$, we should note that a $\mathcal{O}(k)$ solution to finding the sum of the first $k$ elements will not be fast enough, especially since $1 \leq t \leq 5 \cdot 10^5$. With this, we can note that there is a $\mathcal{O}(1)$ short cut to calculate this:

$$\sum_{i=1}^{k} i \equiv \boxed{\frac{(k)(k+1)}{2}} \tag{6}$$

**Observation 2:** The input and output can be quite large
Since $1 \leq h, r \leq 10^{18}$ and $1 \leq k \leq 10^9$, we can note that an individual line of input can have at most 47 characters (18 for $h$, 18 for $r$, 9 for $k$, and 2 spaces to separate the symbols).

Since $1 \leq t \leq 5 \cdot 10^5$, then there may be, at most, $2.35 \cdot 10^7 + 6$ symbols in an entire test case (6 symbols for $t$, and $2.35 \cdot 10^7$ symbols for the $t$ test cases).

Assuming that a single character requires 8 bits $\equiv$ 1 byte to store, then the input data may be, at most

$$2.35 \cdot 10^7 + 5 \text{ byte} \approx \boxed{22.5 \text{ MB}} \tag{7}$$

Similarly, since there will be at most $t \cdot 10$ characters output by the program, the output may be at most

$$5 \cdot 10^6 \text{ byte} \approx \boxed{4.8 \text{ MB}} \tag{8}$$

Therefore, programs need to make use of fast I/O operations to be able to handle all input and output within the time constraint.

**Time Complexity**:

$$t \cdot \mathcal{O}(1) = \boxed{\mathcal{O}(t)}$$

# Good BoB

**Explanation**
First, calculate the maximum total score as:

$$s_{max} = u + d + i + c + m + p + 2b \tag{9}$$

Then, as you are reading in the strings, check for which one it is, and set some flag corresponding to that string to be true. Once all strings are read in for a given BoB, simply run through the different cases that were presented for how to increase/decrease the score and change the score accordingly.

Once you have determined the score of the BoB, iterate through the cases where a different output is required and print the output accordingly.

**Time Complexity**:

$$t \cdot n \cdot \mathcal{O}(1) = \boxed{\mathcal{O}(t \cdot n)}$$

# Chromosomal Similarity Frequency

**Explanation**
This problem simplifies down to the classical Longest Common Substring problem but adds the need to count the frequency of strings of that length. This can be accomplished by maintaining the current value for the length of the longest common substring, and then maintaining a second value for the number of times that we have seen a string of that length. When we find a new substring with a larger length, we reset the frequency count to 1. Using dynamic programming, we can solve this in $\mathcal{O}(c \cdot e)$ time. A further space optimization can be made to solve this with $\mathcal{O}(1)$ memory by only maintaining the two most recent rows of the DP table (rather than the whole $c \cdot e$ matrix).

Then, to handle the wrap-around case without double-counting strings, we can re-perform the modified longest common substring algorithm described above with a couple of further modifications:

1. The two strings being compared will be $D = C[c-(e-1)\dots c] + C[0\dots e-1]$ In other words, the last $e-1$ characters of $C$ concatenated with the first $e-1$ characters of $C$.
2. Our value for the length of the longest common substring must start out as the length found in the initial run-through.
3. We only consider substrings that could not have been found in the initial run-through of the two strings. Let $i$ and $j$ be the indices in $D$ and $E$, respectively, that are being compared. Thus, $i$ indicates the ending position of the common substring in $D$. Similarly, let $\ell$ be the length of the current common substring ending at index $j$ in $E$. Thus, $i-\ell+1$ is the starting position of the substring in $D$. Then, we only consider a string of length $\ell_{max}$ to be unique if both of the following are true:
   (a) $i-\ell+1 \leq e-2$
   (b) $i \geq e-1$

   In other words, the starting index appears within the first $e-1$ characters (a), and the ending index appears within the last $e-1$ characters (b).

This takes $\mathcal{O}(2e \cdot e) = \mathcal{O}(e^2)$ time to calculate as $|D| = 2e-2$.

**Time Complexity**:

$$\mathcal{O}(c \cdot e) + \mathcal{O}(e^2) = \boxed{\mathcal{O}(c \cdot e)}$$

# Pairs Apart

**Explanation**
We can observe that $|M[i] - M[j]|$ is maximum among all pairs $i, j$ when $i$ is the student with the highest, un-grouped score and $j$ is the student with the lowest, un-grouped score. After making the first pair, we can note that this property still holds as those elements $i$ and $j$ are no longer considered as they are grouped.

Thus, simply sort the list of integers, and pair $(0, n)$, $(1, n-1)$, $(2, n-2)$, $\ldots$, $(k, n-k)$. At each pair, print their sum, per the problem description.

**Time Complexity**:

$$\mathcal{O}(n \lg n) + \mathcal{O}(n) = \boxed{\mathcal{O}(n \lg n)}$$

# A Big Bang Theory

**Explanation**
Notice if we construct an array $c$ where $c_i = \gcd(a_i, b_i)$, then the problem can be reworded as follow:

Given an array $c$ and a list of queries $x$. For each $x_i$, count the number of element in $c$ that is multiple of $x_i$.

Let $M$ be the largest value in array $c$. Notice that number of multiple of some arbitrary value $x$ that actually can contribute to the answer is $\lfloor \frac{M}{x} \rfloor$.

Let $C(x)$ be function returning occurrence of $x$ in array $c$. Notice that we can count number of multiple of some value $v$ by iterate through all multiples of $v$ as follow:

```
1: int total = 0;
2: for int $x = v$; $v \leq M$; $x$ += $v$ do
3:     total += $C(x)$;
```

Lastly, notice that if we apply the above procedure to all $v$ that are less than or equal $M$, then overall time complexity will be as follow:

$$M + \frac{M}{2} + \frac{M}{3} + ... + 1 = O(M \log M)$$

**Time Complexity**:

$$\boxed{\mathcal{O}(N + M \lg M)}$$

# All The Rage

**Explanation**
First, imagine only one person is infected. The day that everybody else is infected will be 1 + the distance to that one person. This reduces the problem to finding the distance from all nodes to that one person. This can be solved with a breadth-first search (Start from the first node and search outwards expanding the nodes in the order they are seen). Can this be generalized to multiple infected people?

Yes, it can!! If we do a breadth-first search with multiple initial nodes, we will find the distance from each node to the closest initially infected node, the result that we want.

**Time Complexity**:

$$\boxed{\mathcal{O}(N)}$$

# Comet Camp Crisis

**Explanation**
First observation one can make about this problem is the fact that the queries can be processed in an offline manner. In other words, it is possible for one to reorder the order of the queries to facilitate efficient processing.

Let $V$ be some arbitrary vertex such that all nodes in its subtree is blocked. Notice that the answer for $V$ can be either of the following:

- A path from some node $x$ not in V's subtree to V's parent.
- A path from 2 arbitrary nodes $x$ and $y$ that are not in V's subtree. In this case, LCA of $x$ and $y$ will be either an ancestor of $V$ or will be some node $z$ such that there exists a node that is ancestor of $V$ and has $z$ in its subtree.

With that in mind, we will propose one solution to solve this problem.

To begin with, we will precalculate the following values for each subtrees in the tree using DFS:

- `round_dist`: The longest distance between any 2 nodes within the subtree.
- `root_dist`: The longest distance between the root of the subtree and any node within the subtree.

With those values computed, we can now move on to pre-calculate `ret[i]`, denoting the longest path between any 2 nodes in the tree if every nodes within subtree rooted at $i$ is blocked using a second DFS.

In this DFS, we need to keep track of the following parameters or an arbitrary subtree rooted at $x$:

- `max_single_path`: Longest path from a node not in subtree of $x$ to the parent of $x$.
- `max_dual_path`: Longest path between any 2 nodes that are not in the subtree rooted at $x$.

Notice that if our DFS transitions from a node $x$ to one of its children $y$, then y's `max_single_path` will be `max(max_single_path[x], max_sibling_single_path)`, where `max_sibling_single_path` is `max(root_dist[j])` for all node $j$ that are sibling with $y$.

As for `max_dual_path`, it will be maximum of either its parent's `max_dual_path`, the largest `root_dist` of 2 of its sibling (if any), largest `root_dist` of any of its sibling (if any), sum of largest `root_dist` among that of its sibling and its parent's `max_single_path`, or the largest `round_dist` among all its sibling.

The transition for both `max_dual_path` and `max_single_path` from a node to its child can be done using `std::multiset` or any data structures with similar functionality.

Another way to solve this problem will be to utilize a segment tree that supports lazy propagation on the tree in which nodes are sorted based on preorder traversal. Refer to nhatquangtm1805's implementation for more details.

**Time Complexity**:

$$\boxed{\mathcal{O}(n \log n + q)}$$