Problem Tutorial: “Awesome Shawarma”

Problem:

Given a tree, find the number of nodes after connecting them, the number of overall bridges will be between L and R.

Prerequisites: DSU on tree, Binary Indexed Tree(BIT)

Explanation:

Let’s define some term $dist$ between two nodes, $dist(a, b)$ is the number of edges in the path between node a and b.

In any tree, all the edges are bridges initially , so there is $n - 1$ bridges and when we connect node a and b with an edge, we can see that the new graph will have cycle, and the number of bridges that remains after the connection is $n - 1 - dist(a, b)$.

We have:

$L \leq n - 1 - dist(a, b) \leq R \Rightarrow$ (Subtract by $n - 1$)

$L - n + 1 \leq -dist(a, b) \leq R - n + 1 \Rightarrow$ (Multiply by -1)

$n - 1 - R \leq dist(a, b) \leq n - 1 - L$

So the problem can be converted to finding the number of nodes the a and b that the distance between them is between $n - 1 - R$ and $n - 1 - L$ and if we managed to create function $solve(X)$ that will take X as input and will output the number of nodes that the distance between them are less or equal to X, so our answer for the problem will be:

$solve(n - 1 - L)$ - $solve(n - 1 - R - 1)$

How to create the $solve$ function?

This problem can be solved using multiple ways, we will discuss "DSU on tree"approach: Please refer to this link for better understanding to "DSU on tree": http://codeforces.com/blog/entry/44351

Let’s root the tree with the node number 1 and define $dep[u]$ to be $dist(1, u)$, i.e the number of edges between the root and u (the depth of node u in the tree). After calculating the dep array, we can define the $dist$ term as: $dist(a, b) = dep[a] + dep[b]$ - 2 * $dep[LCA(a, b)]$ where $LCA(a, b)$ is the lowest common ancestor of a and b.

Now suppose we have some node $base$, we will call the big subtree of $base$ with **heavy** and the other subtrees as **light**, and let’s create Binary Indexed Tree, where its indices are the depth, and the value will be number of nodes having that depth.

If we have node $base$ and its **heavy** subtree, and **light** subtrees, we will follow these steps:

1 - store all depths of the **heavy** subtree in the **BIT** data structure.

2 - For every **light** subtree:

- **a.** Query result for every node in that subtree.
- **b.** Add every node depth to the **BIT**.

step **(a)** can be calculated as:

loop through all **light** subtree nodes u, we need to know the number of nodes v that are stored in the **BIT** and the $dist(u, v) \leq X$, so:

$dep[u] + dep[v]$ - 2 * $dep[LCA(u, v)] \leq X$

but we know that the $LCA(u, v)$ is $base$ (because u and v are in the subtree of $base$ and in different subtrees), so:

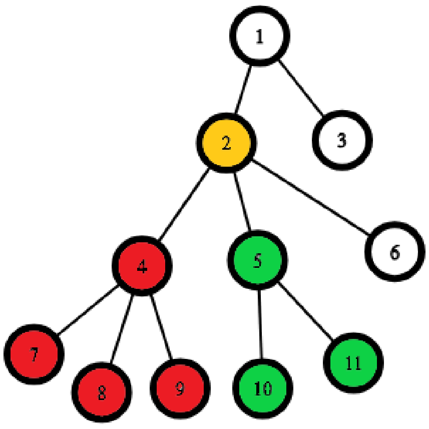
$dep[u] + dep[v]$ - 2 * $dep[base] \leq X \Rightarrow$

$dep[v] \leq X + 2$ * $dep[base]$ - $dep[u]$

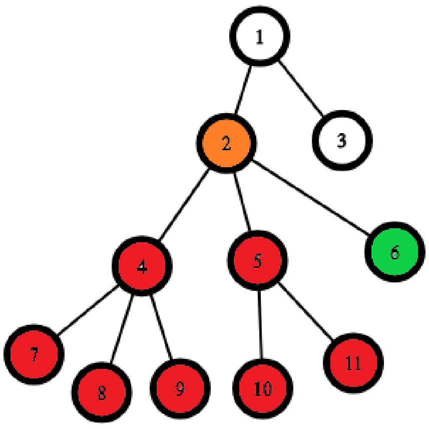
In other words, we need to find the number of nodes that the depth is less or equal to $X + 2 * dep[base]$ - $dep[u]$, and it can be calculated using **BIT**.

step **(b)** is required to calculate the distance between **light** subtrees between each other, not only with **heavy** subtree.

Don't forget to calculate the answer between *base* and all its subtrees.



The orange is *base* node, red is **heavy** and green is **light** subtree, so we calculate the answer between green nodes and red nodes, and after calculating the answer between them, we add the green nodes to the **BIT**.



We calculate the answer between green nodes and red nodes, and after calculating the answer between them, we add the green nodes to the **BIT**, and at the last step, we calculate the answer between *base* and all nodes in its subtree.

Complexity:

- Calculating *dep* array can be done in $O(N)$
- Looping through **light** and **heavy** nodes using DSU on tree can be done in $O(NLog(N))$
- Query and update using **BIT** can be done in $O(Log(N))$

So the total complexity is: $O(N.log^2(N))$

Challenge:

Solve the problem in $O(N.log(N))$ using "DSU on tree".

Problem Tutorial: "Baklava Tray"

We can phrase the problem as "Given an infinite sequence of n-sided polygons, each is formed by joining the midpoints of the sides of the latest drawn polygon, and a point is thrown inside, find the expected value of the number of polygons that contain it". Once we solve that problem, we would just need to multiply the answer by 10^4 to count for each fork in the original problem statement.

We will discuss the main idea of the solution. First, we should notice that each time the area of the newly formed polygon will decrease by a factor of α, strictly speaking, the formed polygons will have the following areas : $1, 1/\alpha, /\alpha^2, 1/\alpha^3, ..., 1/(\alpha^\infty)$

Let's denote the area of the ith consructed polygon as A_i , a point has a probability $A_i/A_0 = \alpha^{-i}$ of ending up in the ith polygon, so our answer reduces to calculating $\sum_{i=0}^{\infty} A_i/A_0 = \sum_{i=0}^{\infty} \alpha^{-i} = \frac{1}{1-\alpha}$

This sum can be calculated using geometric series in O(1). In fact, a for loop that is done carefully up to enough limits, this doesn't use too much time and still yields enough precision that will pass.

In order to calculate the α stated earlier, there are two ways, one is by simulating the first step and calculate the ratios. The second way is by deriving a formula, we triangulate the i^{th} polygon into n triangles, where each triangle has two sides of side-length R_i and an angle $2\pi/n$ in between, then the area of the corresponding polygon by the sine-rule is $R_i^2 \sin(2\pi/n) \cdot n/2$. After that, if we draw a line joining the mid-point of the third side and the intersection of the first two sides, we will find that this is an isosceles triangle, and thus this new line is perpendicular to the third side. Using this fact, we can calculate the side-lengths R_{i+1} of the of the smaller polygon using the formula $R_{i+1} = \sqrt{R_i^2 - (x/2)^2}$ where x is the side-length of the third side, which can be calculated with the cosine-rule $x = R_i\sqrt{2 - 2 \cdot \cos(2\pi/n)}$. The final formula is $R_{i+1}^2 = R_i^2(1+\cos(2\pi/n))/2$, and if we are to multiply both sides by $\sin(2\pi/n)\cdot n/2$, we will find that the areas are given by $A_{i+1} = A_i \cdot (1+\cos(2\pi/n))/2$, and thus $\alpha = (1+\cos(2\pi/n))/2 = \sin^2(\pi/n)$.

Here's a short solution: $2 \cdot 10^4/(1 - \cos(2\pi/n)) = 10^4/\sin^2(\pi/n) = (10^2/\sin(\pi/n))^2$

Challenge: if n is larger than 10^3, do you think your code would still have enough precision?

Problem Tutorial: "Coffee"

This is a direct implementation easy problem, all you need to do is to save for each drink its price with the three sizes and then calculate the total cost for each person independently as it is guaranteed that the drinks and persons are unique.

Problem Tutorial: "Dull Chocolates"

The idea to solving this problem is noticing that while the grid is very large, the number of chocolates can at most be 1000. Let's assume our grid can only be up to 1000×1000, in this case the problem would be easy to solve as we can formulate it a direct 2D Sum on a grid. In other words, we maintain a 2D sum of all chocolates, then brute force every end point of the grid, and counting if it's odd or even.

However, since N and M can be up to 10^9 each, the above approach would certainly not pass, instead we can utilize a technique known as grid compression, which can be thought of as a variation of co-ordinate compression. Let's compress all points in ascending order starting from 1 independently for each dimension of the grid. For example, if we have 2 chocolates as $(10^2, 10^3), (10^2+4, 10^3+4)$, they would be compressed as $(1, 1), (2, 2)$. We should also be able to find a given original chocolate given it's compressed value.

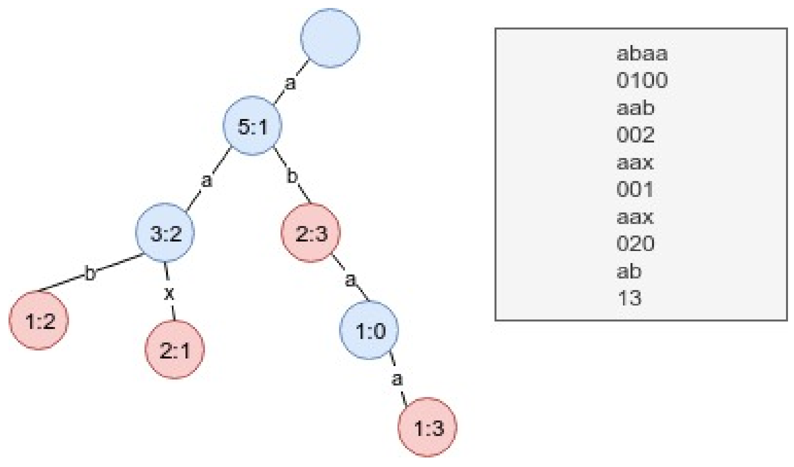
Once the compression is done, and since we can have at most 10^3 chocolates, they would all fit in a $10^3 \times 10^3$ grid, however if we apply the regular approach of brute forcing every end point, we would undercount the answer since end points that didn't appear as chocolates will not be counted, but notice that the answer for a given fixed point on which a chocolate lie is the same till the first point before it on each respective dimension where a chocolate appeared, so we can just multiply by the difference of of dimensions. More technically speaking, if (x_2, y_2) is a compressed chocolate point, it's contribution depending on parity would be, (Original position of (x_2) in the grid - original position of $(x_2 - 1)$ in the grid) $\times$ (Original position of (y_2) in the grid - original position of $(y_2 - 1)$ in the grid). You should account for special cases, where an earlier chocolate did not appear.

Grid compression can be implemented in variety of ways, one for example is putting all entries in a sorted map, then re-iterating over it, and assigning values in an increasing order.

The complexity of this solution is $\mathcal{O}(N \cdot M \cdot log(N * M))$

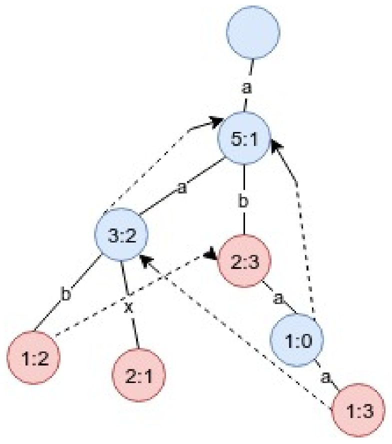
Problem Tutorial: "Exciting Menus"

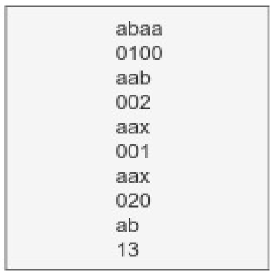
First of all, we need to notice that although we are looking for a substring, this substring must match a prefix of some of the given strings, because if it is not a prefix the popularity part will be zero. So let's insert all the given strings in a 'Trie' and after that, each node of this trie will represent a prefix of one or more of the given strings. The depth of each node will represent its length and the number of strings in which this node is a prefix corresponds to the number of times you passed by this node while inserting the strings, this will be stored in the node as the value *cnt*. So for each node lets keep track of two values $(cnt : max)$ where *cnt* is the popularity of this node and *max* is the maximum value of A^i_k of all substrings matching the prefix ended at this node. Using this definition, the problem would be just to compute for each node u the value $depth(u) \cdot cnt(u) \cdot max(u)$, and return the maximum of all of those. The values *cnt* and *depth* are not hard to compute, however the value *max* is tricky, initially while inserting the strings let's set the *max* value of a node by the maximum possible value of $A^{i_1}_k, \cdots A^{i_r}_k$ corresponding to all the prefixes of this node, for better understating see the following figure representing a trie after inserting some strings in it.



```
abaa
0100
aab
002
aax
001
aax
020
ab
13
```

Please note that the max value of some nodes are not final e.g nodes corresponding to 'a', and 'aa' their max values should be 3 but currently they are not (take a moment to figure out why it should be 3). So what should we do next? First, you should note that all substrings to be chosen $S^i_{j,k}$ are also suffixes of the prefix S^i_k (with node u), so in other words, we are looking for a prefix S^i_k (with a corresponding A^i_k) and a corresponding suffix of that $S^i_{j,k}$ which is another prefix with node v so we can get $|S^i_{j,k}| \cdot popularity(S^i_{j,k}) \cdot A^i_k$, this corresponds to $depth(v) \cdot popularity(v) \cdot max(u)$. Therefore given a node v, we should update the *max* value of all the suffix nodes u by $max(u) := \max(max(u), max(v))$. The question is how to precompute these updates efficiently. If we recall the 'Aho-Corasick algorithm', we will find out that the relation between u and v is that u will reach v using some fail-links (also known as suffix-links), because the fail-links are defined as the maximum suffix of a given prefix that also matches a prefix in the Trie, which is exactly what we are looking for. Knowing this, we will find that the relation between v, u is that $u = fail^k(v)$ for some k. If we use the 'Aho-Corasick algorithm' to build the fail links, and after creating these links our trie should look like this:

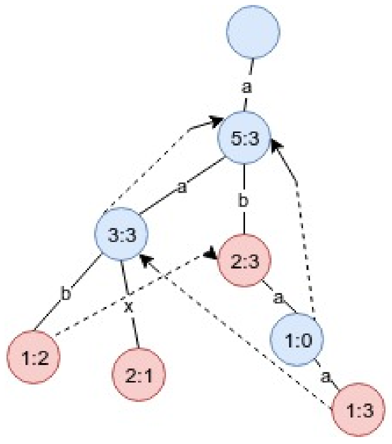


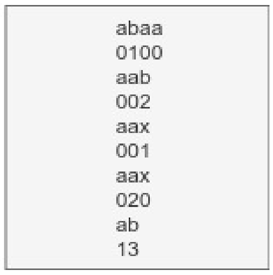


```
abaa
0100
aab
002
aax
001
aax
020
ab
13
```

Note that the nodes with no outgoing suffix-link will fail to the root node by default.

After that each node should maximize between its current max value and max values of all nodes that reach it through some sequence of suffix links, thus we can compute this using BFS from the bottom nodes to top nodes (the BFS needs to take care of processing all nodes with higher depths before nodes of lower depths, to update the max values in the correct order and not missing correct updates), and update the max value of all nodes by max value of the parent in the BFS (that arrived using some suffix-link). Thus we arrive at the final max values of the nodes, and the Trie with the final values of max should be like this:





```
abaa
0100
aab
002
aax
001
aax
020
ab
13
```

After knowing the $depth$, cnt and the final max values of each node calculating the answer will be $\max_u depth(u) \cdot cnt(u) \cdot max(u)$ for all nodes u.

In the above example, the node 'aa' is the node with maximum answer 18, where $i = 1, j = 3, k = 4$, because $A_4^1 = 3$, and then j is determined by the suffix-link going to the node 'aa' with depth 2 and popularity 3.

Problem Tutorial: "Flipping El-fetiera"

Let's denote our matrix as T. We are essentially trying to calculate the expected value of ones in the matrix after K steps, which is $\mathbb{E}(T_{0,0}, T_{0,1}, \cdots, T_{i,j}, \cdots, T_{N-1,N-1})$ $\forall i, j < N$

By the linearity of expectation property, $\mathbb{E}(T_{0,0}, T_{0,1}, \cdots, T_{i,j}, \cdots, T_{N-1,N-1})$ is equal to the sum $\mathbb{E}(T_{0,0}) + \mathbb{E}(T_{0,1}) + \cdots + \mathbb{E}(T_{i,j}) + ... + \mathbb{E}(T_{N-1,N-1})$, so our problem reduces to calculating for each individual bit the expected value after K steps, and summing all of these. More formally, we would like to calculate $\sum_{i=1}^{N}\sum_{j=1}^{N}\mathbb{E}(i,j)$ where $\mathbb{E}(i,j)$ represents the expected value of the cell (i,j) after K steps.

The probability $p_{i,j}$ of flipping a fixed bit is (number of matrices that can contain the bit) / (all possible matrices), which is :

$$p_{i,j} = \frac{((i+1)\cdot(j+1)\cdot(N-i)\cdot(N-j))}{\sum_{i=0}^{N-1}\sum_{j=0}^{N-1}(N-i)\cdot(N-j)}$$

Once that is fixed we can compute the expected value of a single cell after K steps using various ways, one of them is using dynamic programming, by having $dp[k][s]$ represent that the current bit is s after k steps. We can formulate the recurrence as $dp[k][s] = (1-p_{i,j})\cdot dp[k-1][s] + p_{i,j}\cdot dp[k-1][1-s]$. The complexity will be $\mathcal{O}(N^2 \cdot K)$.

Alternative solution:

Let's define a point-wise binary operator $\odot$ that takes as input a matrix, and a given mask of a sub-matrix to flip. The operator should simulate the flipping operation for the given matrix. If we define the matrix as A, and a mask of a sub-matrix as M, then the operator can be defined as :

$$R_{i,j} = A_{i,j}\cdot(1-M_{i,j}) + M_{i,j}\cdot(1-A_{i,j}) = A_{i,j} + M_{i,j}\cdot(1-2\cdot A_{i,j})$$

It can be thought as taking $A_{i,j}$ with the complement $M_{i,j}$ or the flipped $A_{i,j}$ with $M_{i,j}$. It can be shown also that the operator is associative.

The solution to the problem is simulating all masks for K steps, and then averaging them. (Dividing them by total number of masks).

Let's assume our matrix is X, and our masks are a, b, c and d then our simulation for one step assuming the operator is distributive is

$$\frac{X\odot a + X\odot b + X\odot c + X\odot d}{4} = X\odot\frac{(a+b+c+d)}{4}$$

If we apply the operation for another step we get:

$$X\odot\frac{(a+b+c+d)}{4}\odot\frac{(a+b+c+d)}{4}$$

which is equivalent to: (as the operator is associative)

$$X\odot\frac{(aa+ab+ac+ad+ba+bb+bc+bd+ca+cb+cc+cd+da+db+dc+dd)}{16}$$

In another words, the solution is contributions of all combinations of all possible masks for K times divided by the total number of possible combinations of all masks.. How can we calculate the answer for K times? We effectively want to calculate the kth power under the given defined operation, which can be done by binary exponentiation (as the operator is associative) similar to normal mod power, but instead replacing the $\times$ operator, by our defined one $\odot$. Let B be our initial matrix which contain the values for a single step, as in the contribution of possible masks divided by the count of all possible masks, we are effectively trying to calculate $B\odot B\odot B\odot\cdots\odot B$ for K times which is B^K. Once B^K is calculated, our answer to the problem is summing the cells of the resulting matrix of $X\odot B^K$, where X is our original matrix. The complexity will be $\mathcal{O}(N^2\cdot\log(K))$.

As for B^1, it can be calculated as follows :

$$B_{i,j} = \frac{((i+1)\cdot(j+1)\cdot(N-i)\cdot(N-j))}{\sum_{i=0}^{N-1}\sum_{j=0}^{N-1}(N-i)\cdot(N-j)}$$

It can be thought of as fixing a point for each dimension independently, and choosing two points to contain it, one before (or equal) it, and another after (or equal) it.

We are normalizing by all possible masks in this step because doing so would allow us to prove distributivity of the operator. To prove distributivity, we need to prove that

$$p \odot (\sum_{i=1}^{R} \frac{q_i}{R}) = \frac{1}{R} \sum_{i=1}^{R} p \odot q_i$$

$$p \odot (\sum_{i=1}^{R} \frac{q_i}{R}) = (p + (\sum_{i=1}^{R} \frac{q_i}{R}) \cdot (1-2p)) = \frac{1}{R}(R \cdot p + \sum_{i=1}^{R} q_i \cdot (1-2p)) = \frac{1}{R}\sum_{i=1}^{R}(p + q_i \cdot (1-2p)) = \frac{1}{R}\sum_{i=1}^{R} p \odot q_i$$

Problem Tutorial: “Greatest Chicken Dish”

This queries of this problem will be solved offline, but first, we define some needed terminologies and then we show the main processing to solve it. Later we mention some details related to the side tasks to be made.

First of all, if we define a contribution array C such that C_L counts how many indices $k \geq i$ satisfy a condition, here C_L will be the number of start points $i \geq L$ that have $j \geq i$ such that $gcd(A_i, \cdots, A_j) = d$ for a fixed value d. In order to answer a query (d, R, L), we can process all possible contributions of points $j \leq R$ with a fixed GCD value d, then we would just output C_L, because the contributions only include points with GCD value d, endpoints $j \leq R$ and C_L answers for all $i \geq L$, which is exactly what the query is asking for. Consequently, we need to maintain a contribution array that computes only the necessary values and then answer all the queries related to these values, then efficiently update the contributions, then answer and so on, until all contributions are processed or all queries are answered.

In order to compute the contributions, we first consider if we fix some endpoint j of a range, and then we move the start point i from j until the start index 1, and at each point we calculate the GCD of the range between i and j, we will find that there are $\mathcal{O}(\log M)$ distinct GCD values, because as i moves, the GCD either stays the same or decreases by dividing by one of its divisors (factor more than 1), and there are at most $\mathcal{O}(\log M)$ such divisors. Knowing this, we can hence define a quadruple (d, j, i_1, i_2), where j is the fixed end point, and $[i_1, i_2]$ defines a range of all possible start points i, such that the GCD of all values in the range $[i, j]$ is exactly equal to d.

There are at most $\mathcal{O}(N \log M)$ such quadruples, these can be easily found by trying all possible endpoints j, and then for each start point i_1 we have, we can compute the one before it using binary search (to find the latest start point x that makes the $gcd(A_x, \cdots, A_j) < gcd(A_{i_1}, \cdots, A_j)$, this would generate the new quadruple $(d, j, x, i_1 - 1)$, then use x as i_1 to get the earlier one and so on. The quadruples will be found in $\mathcal{O}(N \log N \log^2 M)$ time complexity, because each quadruple needs a binary search that computes a gcd operation at each step which costs $\mathcal{O}(\log N \log M)$ time complexity. What remains is to use these quadruples to properly define compute the contributions defined at the beginning.

A small example, as a demonstration of the quadruples, if we are given an initial array $[1, 1, 2, 22, 44, 11, 121, 19]$, then the fixed endpoint 7 ($A_7 = 121$), has the quadruples $(121, 7, 7, 7)$, $(11, 7, 4, 6)$ and $(1, 7, 1, 3)$. If we fix only consider the quadruple $(11, 7, 4, 6)$, we will find that this adds to the contribution array the following values $[3, 3, 3, 3, 2, 1, 0, 0]$ because the only start points are $4, 5, 6$, therefore the start points after indices $1, 2, 3, 4$ are the same three points, and from index 5 there are only two points, and so on. We can generalize this in order to update the contributions if we have a quadruple (d, j, i_1, i_2), then we can update the range $[1, i_1 - 1]$ by incrementing all its values by a fixed value $i_2 - i_1 + 1$, namely the size of the possible starts range, in addition to making a descending-update to the range $[i_1, i_2]$, by adding the values $i_2 - i_1 + 1, i_2 - i_1, \cdots, 1$ to the contribution values. This will be achieved using a segment tree with lazy propagation (will be shown how, later)

Finally, We need to group the queries and quadruples by GCD values, this can be achieved when the quadruples and the queries are sorted lexicographically by GCD value then by endpoint, then the offline algorithm for answering queries would act something similar to the merge function in the merge-sort algorithm, by maintaining two pointers one for quadruples and one for queries. Before answering a query, the quadruples pointer will move to make sure that we processed (updated the contributions) of all quadruples with the same fixed GCD value as the query, and that only quadruples with end values $j \leq R$ are processed, as stated by the early algorithm, then the query pointer is moved to answer the next query and so on. The contributions will get reset in between when the GCD values change.

Summing up all of this together, we can come with the main algorithm:

1. Precompute a sparse array of GCD values.
2. Compute all quadruples.
3. Sort queries and quadruples lexicographically by GCD value, then endpoint.
4. Initialize pointer $quadruple\_idx$
5. For $query\_idx$ in $1, \cdots, Q$
   (a) While quadruples have different GCD then the queries $\Rightarrow$ reset contributions
   (b) While quadruples have the same GCD and end index not greater than the query end index $\Rightarrow$ process the quadruple using a descending-update and normal update operations.
   (c) Compute the query's answer by querying the contribution C_L
   (d) If there is no next query or the next query has different GCD from the last processed quadruple $\Rightarrow$ reset contributions

The time complexity is $\mathcal{O}(N \log^2 M \log N + Q \log Q)$ per test case, where the most intensive part is to compute the quadruples in $\mathcal{O}(N \log^2 M \log N)$ and sorting them and the queries.

Finally, we mention quickly how the segment tree queries are processed in a way that supports the descending-update. Each node will store three values sum, $toadd$, $toadd\_dec$, where sum is the actually processed sum of the node, $toadd$ is the lazy variable showing what is the value needed to be added to the range (standard range-sum), and $toadd\_dec$ is the lazy variable counting how many times we need to make the descending-update to this node. There is an additional lazy flag for resetting the node.

The key idea is to realize that we can make the descending-update with an offset x for a node with the range [a, b], meaning that we add the values $[x+b-a+1, \cdots, x+2, x+1]$ to the range. Then, In order to descendingly-update the two children nodes with ranges $[a, c]$ and $[c+1, b]$, we then make descending-update on the left node with offset $x+b-c$ and on the right node with the offset x. However, the descending-update with an offset x on some node can be dissolved into two updates, namely descending-update with offset 0 and a standard range-sum update of value x.

Eventually, the propagation of the lazy variables needs a summary update to the value sum, this can be acheived by the formula

$$sum := sum + toadd\_dec \cdot \frac{(siz-1) \cdot siz}{2} + toadd \cdot siz$$

because of the triangular sum $toadd\_dec \cdot (1 + 2 + \cdots + siz) = toadd\_dec \cdot \frac{(siz-1) \cdot siz}{2}$ and the constant value sum $toadd \cdot siz$. After that the $toadd\_dec$ is propagated as is to the children, and the left node's lazy value $toadd$ is updated by $toadd\_dec \cdot right.siz$ because of the additional offset.

Problem Tutorial: "Hawawshi Decryption"

First of all, the range $[A, B]$ is small, so we can try all possible value of R_0 in this range, in each of these tries we need to compute if $R_n = X$ will be the case for some n such that $n < N$ (checking the smallest such n is sufficient). The baseline solution would try all the elements of the sequence, but this would time out because it has a time complexity $\mathcal{O}(|B - A|p)$ per test case, therefore we need to look on fewer elements of the sequence to achieve a faster solution. The main task would then be: given R_0, we need to solve for the smallest n such that $R_n = X$.

To solve this new reduction, we introduce S_n such that $S_n = R_n - v$ and $S_n = aS_{n-1}$, then by substituting in the pseudo-random generator, we can get that $v = av + b$ hence $v(1 - a) = b$. We can then conclude, if such v exists, that $S_n = a^n \cdot S_0$ which implies $R_n = a^n(R_0 - v) + v$ where $v = b(1 - a)^{-1}$; such v exists only if $a \neq 1$, otherwise if $a = 1$, then this needs to be handled differently, we find in that case that R_n is an arithmetic sequence, such that $R_n = R_0 + nb$.

In order to find the smallest n such that $R_n = X$, we handle three cases. In case if $a = 1$, then we can solve $n = (X - R_0)b^{-1}$ as a special case. Otherwise, in the second case we use the other equation, $a^n = (X - v)(R_0 - v)^{-1}$ to solve for n, this is done using the discrete logarithm. However, there's an additional third (special) case needs to be handled, if $R_0 = v$ (in such case $(R_0 - v)^{-1}$ is not defined), then by definition of R_1 and the equation of $v = av + b$ above, we can deduce that $R_1 = aR_0 + b = av + b = v = R_0$, which means that the sequence is a fixed point (that $R_i = R_0$ for all i). The solution of this second special case, is either $n = 0$ if $R_0 = R_n = X$ or no solution if $X \neq R_0$.

The discrete logarithm intended was the *baby step giant step* algorithm, which operates in time complexity $\mathcal{O}(\sqrt{p})$, so the overall time complexity is $\mathcal{O}(|B - A|\sqrt{p})$ per test case, using a hash/unordered map. The implementation needs to take care of finding the smallest solution of n.

======

Another solution, whose idea was proposed by the winning team (rephrased by the author), is to define an operator $\odot$ over the set $\mathbb{Z}_p \times \mathbb{Z}_p$, the operator is defined by $(x, y) \odot (a, b) = (ax \mod p, ay + b \mod p)$, with an identity element $(1, 0)$. This operator can be shown to be associative as well, so we can easily make power operations without special handling. Additionally a function $f : \mathbb{Z}_p \times \mathbb{Z}_p \to \mathbb{Z}_p$ is defined by $f((x, y)) = (xR_0 + y) \mod p$.

The choice of this operator particularly simulates the linear recurrence, this can be shown by induction by proving that $f((a, b)^n) = f((a, b) \odot \cdots \odot (a, b)) = R_n$. The induction basis $n = 0$ is proved using the identity element (since $(a, b)^0 = (1, 0)$ by definition) by applying the definition $f((1, 0)) = 1 \cdot R_0 + 0 = R_0$. The induction hypothesis assumed is $f((a, b)^n) = R_n$ means that $(x_nR_0 + y_n) \mod p = R_n$ where $(x_n, y_n) = (a, b)^n$. The induction step is then proved using:

$$f((a, b)^{n+1}) = f((x_n, y_n) \odot (a, b)) = f((x_na \mod p, (y_na + b) \mod p))$$

then this can be reduced to:

$$f((a, b)^{n+1}) = (x_naR_0 + y_na + b) \mod p = (a(x_nR_0 + y_n) + b) \mod p = (aR_n + b) \mod p = R_{n+1}$$

Using this proved proposition, we can restate the problem as solving for n for the equation $f((a, b)^n) = R_n = X$, this can be done using the *baby step giant step* directly without handling any special cases. The main difference here is that the identity element is $(1, 0)$ instead of 1 and the *multiplication* operator is the newly defined operator. Also, the operator is associative which is an implicit assumption made by the *baby step giant step* algorithm, because of the operation assuming $a^{xy} = (a^x)^y$. This solution has the same time complexity $\mathcal{O}(|B - A|\sqrt{p})$ per test case.

Problem Tutorial: "Ice-cream Knapsack"

First let's try to find what is the minimum number of calories we can achieve, clearly it's the caloric number of the Kth item in the list when sorted in ascending order by caloric value. Let's denote the minimal number of calories we can achieve as P. Once that is fixed, let's put the happy values of all elements with calories $\leq P$ in a maximum heap, and greedily pop the first K values. The sum of these K values is the maximum total happiness we can achieve.

Problem Tutorial: "Journey to Jupiter"

One of the main difficulties of this problem is just that it is 3d-geometry, however, once it's imagined the solution should come by relatively easily by using some geometrical definitions.

If we assume that the points A, B, C are the new points after the plane's rotation, then we can deduce that they all lie in the same plane whose normal vector is the given vector $\mathbf{N}$. Using the two alternatives equations of the vector (cross) product, we can then deduce that:

$$A \times B = (A_xB_y - A_yB_x, A_zB_x - A_xB_z, A_yB_z - A_zB_y) = |A||B|\sin(\theta_{AB})\frac{\mathbf{N}}{|\mathbf{N}|}$$

$$= (|A||B|\sin(\theta_{AB})\frac{N_x}{|\mathbf{N}|}, |A||B|\sin(\theta_{AB})\frac{N_x}{|\mathbf{N}|}, |A||B|\sin(\theta_{AB})\frac{N_x}{|\mathbf{N}|})$$

This gives three equations in the three unknowns B_x, B_y, B_z, however, we can't use all the three to solve for the point B because the three equations are linearly dependent (hence, they don't have a unique solution), this can be shown by observing the rank of the Gauss-Jordan reduction of the equations, or by Cramer's rule. Picking any two of these will be linearly independent though. Any two equations are linearly independent because of the given constraint $A_x, A_y, A_z \neq 0$, otherwise the two chosen equations needed to be picked up carefully.

Additionally, by using the two alternative equations of the dot product:

$$A \odot B = A_xB_x + A_yB_y + A_zB_z = |A||B|\cos(\theta_{AB})$$

We get the third linearly independent equation in the three unknowns B_x, B_y, B_z. The same operation can be repeated for the point C by replacing B by C. Since the three points form an equilateral triangle, we can deduce that $\theta_{AB} = -\theta_{AC} = \frac{2\pi}{3}$ and $|A| = |B| = |C| = \frac{L}{\sqrt{3}}$.

The equations can be solved using Cramer's rule of Gauss-Jordan elimination. The main solution used Cramer's rule, by expanding the determinants using the $(O(n!))$ definition since $n = 3$ here.

====

Rotation matrix solution :

The spacecraft can only rotate along its principal axis : Yaw ($\vec{u}$ axis), pitch ($\vec{v}$ axis) and roll ($\vec{w}$ axis). Let's decompose the movement into three independent phases :

In the first phase the simulator rotates along the yaw axis by θ. The triangle stays in the (XY) Plane but the tree points were rotated by theta, let's call the corresponding new points A' B' C'.

The second phase is the rotation along the pitch axis (Transverse Axis) by ϕ. The triangle is no longer in the XY axis and the three points rotate again. The new points are A", B" and C" where A" is given as input.

The last phase is a rotation along the roll axis (longitudinal axis) by ψ. In this phase only B" and C" will rotate.

If a,b,c are the new points corresponding to the final position of the spacecraft we can write :

$$\begin{bmatrix} a \\ b \\ c \end{bmatrix} = [R(u,\theta)] \times [R(v,\phi)] \times [R(w,\psi)] \times \begin{bmatrix} A \\ B \\ C \end{bmatrix}.$$

Knowing $u, v, w, \theta, \phi, \psi$ we can calculate R

θ and ϕ can be calculated directly knowing A and A": If the spherical coordinates of A are R,0,0, the spherical coordinates of A" are R,θ,ϕ. (The spherical frame must be oriented in the same way as the planes principal axis).

$\vec{u}$ direction corresponds to the Z axis, $\vec{u} =$ 0,0,1.

$\vec{v}$ corresponds to the direction of the cross product of (OA') and (Z).

$\vec{w}$ corresponds to the direction of (OA").

ψ can be calculated from the new normal vector and A", B". Left as an exercice to the reader.

=====

Another solution, let the plane of the triangle ABC be P, $\overrightarrow{A} = OA$, and $\overrightarrow{M}$ be the cross product $\overrightarrow{A} \times \overrightarrow{N}$. Then by definition of the cross product, $\overrightarrow{M}$ is perpendicular on both vectors. Since that $\overrightarrow{N}$ is perpendicular on the plane P, $\overrightarrow{M}$ is perpendicular on $\overrightarrow{N}$, and $\overrightarrow{M}$ is co-planar with $\overrightarrow{A}$, then the vector $\overrightarrow{M}$ exists in the plane P.

Let D be the point of intersection of the two lines $\overrightarrow{M}$ and AB. Since that $\overrightarrow{A}$ is perpendicular on $\overrightarrow{M}$, then the length of OD can easily be calculated, since AOD is a right-angle triangle, where the angle OAD is the same as OAB, which is $\frac{\pi}{6}$. Using simple trigonometry, the formula can be obtained as $|\overrightarrow{D}| = |\overrightarrow{A}| \tan(\frac{\pi}{6})$. The vector $\overrightarrow{M}$ can then be scaled by $\frac{|\overrightarrow{D}|}{|\overrightarrow{M}|}$ to obtain $\overrightarrow{D}$. Finally, in order to obtain the point B, we then scale the vector $\overrightarrow{AD}$ to be $\overrightarrow{AB}$. Without loss of generality, this can be repeated using $-\overrightarrow{M}$ to obtain the point C. In short,

$$B = \frac{1}{2}(\frac{N \times A}{|N|}\sqrt{3} - A), C = \frac{1}{2}(\frac{A \times N}{|N|}\sqrt{3} - A)$$

Problem Tutorial: "Khoshaf"

First of all, let's see when the sum of elements in some interval $[L, R]$ become divisible, by 3. This happens when the accumulative sum of elements in the range $[0, R] \mod 3$ is the same as the accumulative sum of elements in range $[0, L-1] \mod 3$. Because the subtraction of both accumulative sums will then be equal to $0 \mod 3$. The other case is when an accumulative sum is divisible by 3.

After stating this fact our problem now is reduced to the following: In how many ways we can fill an array with numbers from the given range so that the array has X prefixes with accumulative sum mod 3 equals 2, Y prefixes with accumulative sum mod 3 equals 1 and Z prefixes with accumulative sum mod 3 equals 0, so that the total number of pairs of prefixes with the same value is exactly $K - Z$ (we subtracted Z here because any prefix with value 0 adds 1 to the number of intervals divisible by 3 without the need to pair up with another prefix). At this point a simple dynamic programming solution can be used as follows:

$dp[x][y][z][idx][k][r]$ = the answer to the problem if you already filled the first idx cells of the array and so far you constructed x prefixes with value 2, y prefixes with value 1 and z prefixes with value 0, and the remaining value of K is k and the current accumulative sum mod 3 is r.

The only problem with the above solution is that there is no way you can create the above dp table with given constraints. On the first look, you need to create a table with dimensions $[10^4][10^4][10^4][10^4][10^4][3]$ which is very huge. Let's not give up and try to analyze our dp more. First of all, do we really need to keep idx in the state? the answer is of course 'No' because you always can get the value of this parameter form the sum of $x + y + z$. Great we droped one dimension and we still can drop another one, which is 'k' because the value of k can also obtained from x, y, and z using a pretty simple formula $k = K - [x \cdot (x-1) + y \cdot (y-1) + z \cdot (z+1)]/2$. Now our dimensions are $[x][y][z][3] = [10^4][10^4][10^4][3]$. Now the final optimization, does the value of these parameters need to grow up to $N(10^4)$? No, because as we said if we have c prefixes with the same value they will add $c \cdot (c-1)/2$ intervals divisible by 3. so the value of any of these parameters don't have to be greater than $\mathcal{O}(\sqrt{K})$, so you can calculate the exact maximum value of them or use some approximate upper bound, for example 150 is a very good upper bound (since $150 \cdot 149/2 \geq 1.1 \cdot 10^4 \geq N$) and thus our dimensions are now $[150][150][150][3]$ which is enough to pass. In this solution, however, one needs to take care of the constant operations, using the remainder % operation or division more than necessary might make a huge factor and thus cause TLE.

Thus the final dp formula is given by:

$$dp[x][y][z][r] = \begin{cases} 1 & \text{if } x + y + z = N, [x \cdot (x-1) + y \cdot (y-1) + z \cdot (z+1)]/2 = K \\ 0 & \text{if } x + y + z = N, [x \cdot (x-1) + y \cdot (y-1) + z \cdot (z+1)]/2 \neq K \\ 0 & \text{if } [x \cdot (x-1) + y \cdot (y-1) + z \cdot (z+1)]/2 > K \end{cases}$$

Otherwise,

$$dp[x][y][z][r] = \sum_{w=0}^{2} dp[x+\mathbf{1}(w=2)][y+\mathbf{1}(w=1)][z+\mathbf{1}(w=0)][(r+w) \mod 3]\cdot(count(w,R)-count(w,L-1))$$

where $count(w, X) = \lceil \max(0, X - w + 1)/3 \rceil = \lfloor (\max(0, X - w + 1) + 2)/3 \rfloor$ counts the possible numbers that are equal to $w \mod 3$ and $\leq X$. Additionally, the notation $\mathbf{1}(P)$ is equal to 1 if P is true, and 0 otherwise.

Problem Tutorial: "Looking for Taste"

The key idea to solving this problem is noticing that $K \geq 20$ and $A_i \leq 10^6 < 2^{20}$. The following constraints mean that each number can be represented with at most 20 bits and that we can always pick at least 20 numbers.

Let's enumerate all bits from 1 to 20, for each one, let's pick any arbitrary element that has the ith bit set to 1. Since we can always pick 20 elements, it will always be possible to do so for ith bit if there exists an element with it turned on. This means we can always have the ith bit set to 1 if and only if there exists an element with the ith bit set to 1. Therefore, we can just bitwise-OR all the numbers and be done!