Problem A: Books For War Victims

Setter: Rayhan Mahmud Shihab
Alternate: Jisan Sheikh

Editorial:

Alter's Editorial: Easiest problem of the problem set. As a book can be shared by at most 3 childrens so we need $\frac{N}{3}$ books. But N may not be divisible all the time in this case less than 3 childrens remaining for them 1 book is enough. So the result should be increased by one. And the final equation should be like $\frac{N}{3} + (bool)\ (N\ \%\ 3)$.

Problem B: Acer Rubrum

Setter: Miloslav Brožek
Alternate: Tahsin Masrur

Editorial: Firstly, let us consider a simpler problem: Let us have similar queries, yet without the "time-stamp", and all nodes are either 0 or 1 (and we would like the sum). Now if this would be path (not a tree), it could be easily solved with segment tree (fenwick tree or similar) ... well obviously - simply ask for answer on segment.

In fact, we can use very similar solution for our simplified problem yet we will divide it into two parts (for now) - the two queries:

To solve query number 1, we can use HLD (Heavy Light Decomposition). If we split the tree in such way, it can be proved you can find path between any two nodes in at most **O(log(N))** jumps. Each such jumps access segment tree exactly once (for more precise overlook refer to the blog), making the complexity of a query **O(log$_2$(N))**. In each such access simply ask for sum either between nodes or between node and head of chain.

To solve query number 2, we shall firstly preprocess tree. Here, we could use something called **Euler Tour Tree**. This is an easy flattening of tree, assigning node the time it dfs comes to recursion (also noting this as **BEGIN$_{time}$**), while also noting **END$_{time}$**, which is time dfs leaves recursion. Now, if you build segment tree over such flattened tree (each node will be on index equal to time you assigned it), you can simply access whole subtree of node **u** as query to segment tree from index **BEGIN$_{time}$** (of node **u**) to **END$_{time}$** (of node **u**). This cost **O(log(N))** per query.

BONUS: As you can see, those two queries above could be "easily" done with two distinct segment trees. Anyway if you closely observe, it is not a problem to merge those trees: If you first recurse into the fattest child, preorder indexing will ensure the chains to be consecutive in the **Euler Tour**

Tree. This might make slightly slower queries (but just a little bit), yet the memory will get much better and so will the preprocess along with coding.

Great - now we shall be able to solve the simplified problem - yet the real problem is a little bit harder. However the tree traverse will remain same - the only change is the accessed data structure. There are multiple ways to achieve the objective. One of the possible data-structures might be **Persistent Segment Tree**. Briefly, this data structure is almost similar as segment tree, yet every update query overwrites all nodes it updates (making it equally efficient as classical segment trees, yet using slightly more memory).

Simply sort all begin/end times (note that as end is "inclusive", you would probably add 1 to it). Now for each time, if the time refers to begin, add **+1** to segment tree(s) (on proper index) and if it refers to end, add **-1**. Now, if you would know the "right" time, you could easily apply the above procedure and find the answer. A little problem (yet very easy to solve) is that real time might not correspond to time of **PST**. As you are preprocessing the times (after sorting), you can [for-example] record the time of **PST** to an index of an array corresponding to real time. Now - by a single sweep - you can preprocess the array to obtain the **PST** time in **O(1)**.

So to sum it up, the time complexity of this solution will be$\mathbf{O(N}\boldsymbol{log(N)+Q}\mathbf{log_2(N)+MAX_TIME)}$ and memory complexity will get to$\mathbf{O(Nlog(N))}$

Problem C: Bomb Disposal Strings

Setter: Rayhan Mahmud Shihab
Alternate: Tahsin Masrur, Miloslav Brožek

Alter's Editorial:
IMO, the hardest part in this problem was finding the critical input. While it's pretty hard to understand from problem description, it's not like there was any wrong or misleading information in the problem statement. It's more like, there just weren't that many clues, but there still were a few.
"The password can be found by **concatenating those two strings in such a way it produces the minimum length**, abiding by some rules."
"Let's say, **one of the strings** is string1 and another one is string2." One of the strings. Not the first one.
"After **all these concatenations**, we can get the **final_string"**
"If there are **multiple concatenations** with minimum length"
Whether it's okay for a problem statement to be like that is debatable, I doubt it's okay to blame the problemsetter for one problem that requires a lot of thinking on what kinds of operations/inputs are valid according to problem statement. But that's all personal opinion, let's get back to the important stuff.

Basically, we have to try two concatenations. a + b and b + a (a and b are input strings).
Now how to find the final string of a concatenation? Say, we are trying a + b. According to the rules, we first have to find the longest prefix of b that has frequency >= 2 in b. This can be done by KMP

algorithm. You basically take a new string c = b and discard first character of c i;e c is b without its first character. Then you try to find b as pattern in c. The longest prefix of b that you can find in c as pattern is the answer.
Then you have to check whether that longest prefix exists in a too. If it does, you remove that part from b and add the remaining to a. If it does not, you add the full part of a to b.

Now, after finding the final string, the next task is pretty straightforward. Finding number of substrings in a large string is straightforward with Palindromic Tree (also known as errtree). You can also use Manacher's algorithm for that too.
References:
http://adilet.org/blog/25-09-14/
https://www.geeksforgeeks.org/palindromic-tree-introduction-implementation/
https://www.quora.com/What-is-Manachers-algorithm-How-can-we-use-it-to-find-all-palindromic-sub-strings-in-a-string

Problem D: Easy Math

Setter: Hasinur Rahman
Alternate: Md. Imrul Hassan, Tahsin Masrur.

Editorial:
(^ stands for "to the power" here)
a^b **mod** m As **a** and **m** is not always coprime, Fermat's Little Theorem will not work directly. That's why we have to think in a different way. For example , a = 5 , b = 567 , m = 125 .

$a^b = 5^{567}$

$= 5^{5*100 + 6*10 + 7*1}$

$= (5^{100})^5.(5^{10})^6.(5^1)^7$

Now you can ask that how can we get $5^{10^{\wedge}100}$. 10^{100} is a large number too . To solve this issue we shall observe a pattern: $5^{10} = (5^1)^{10}$, $5^{100} = (5^{10})^{10}$

So, First we calculate a = a mod m .Then,we just go from right most digit of **b** to left most digit and calculate x = (a^10^i)^b[i] and multiply it with answer.

Problem E: Mr. Tom and Relief Goods

Setter: Jisan Sheikh
Alternate: Md. Imrul Hassan, Emrul Chowdhury.

Editorial: Firstly we consider the box as a stack. For any query of first type we have a shirt with size **A** and we push this shirt to the box considered as stack. So this shirt is at the top of the box . For each query of second type first we have to determine whether there is any shirt with size **A** or not. If there exist any shirt with size **A** then we have to find how many elements are there from top to the position containing **A** and this will be the answer. If there is multiple **A** choose the toppest one. And this part will be erased. But if there is no shirt with size **A** then no change will occur.

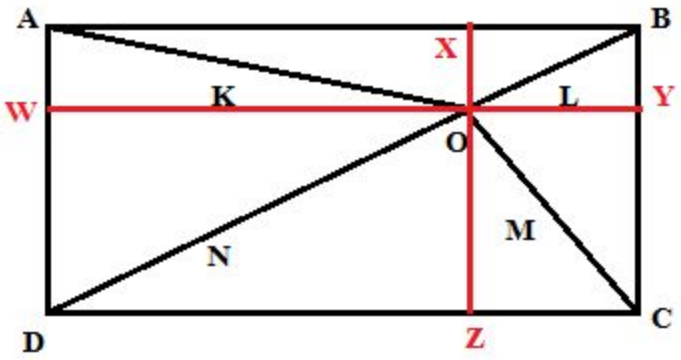
Problem F: Cake Controversy

Setter: Avik Sarkar
Alternate: Md. Imrul Hassan

Editorial: You are given 4 types of Query. In each query , if you will be given -
1) L, M, N and need to find out K . Then K = sqrt(L^2 - M^2 + N^2)
2) K, M, N and need to find out L . Then L = sqrt(K^2 + M^2 - N^2)
3) K, L, N and need to find out M . Then M = sqrt(-K^2 + L^2 + N^2)
4) K, L, M and need to find out N . Then N = sqrt(K^2 - L^2 + M^2)

Proof:

Let **ABCD** is a Rectangle. Honey Drop is at **O**. Draw **4** perpendicular line from **O** on **4** side of the rectangle.
Let they are, **OW** perpendicular on **AD**, **OX** perpendicular on **AB**, **OY** perpendicular on **BC**, **OZ** perpendicular on **CD**.



Now, for triangle **OBX** and **OAX** we will have,
OA^2 - AX^2 = OB^2 - BX^2 [= OX^2]
=> **AX^2 - BX^2 = OA^2 - OB^2**
And, for triangle **ODZ** and **OCZ** we will have,
OD^2 - DZ^2 = OC^2 - ZC^2 [= OZ^2]
=> **DZ^2 - ZC^2 = OD^2 - OC^2**

Now, **AX = DZ** and **BX = CZ** . So put values in any one eqn.
OD^2 - OC^2 = OA^2 - OB^2

Hence, we have any three of them and we can find out the rest one.

Problem G: Designing a Data Structure

Setter: Emrul Chowdhury
Alternate: Avik Sarkar

Editorial: This problem can be solved in many ways. One of the way of solving this problem is using deque and multiset.

- Insert Front X : Push the element in front of the deque, also insert X into the multiset.
- Insert Back X : Push the element in back of the deque, also insert X into the multiset.
- Erase Front : Erase one occurence of the element from multiset and pop from front of the deque.
- Erase Back : Erase one occurence of the element from multiset and pop from back of the deque.
- Get Front : Print the front element of the deque.
- Get Back: Print the back element of the deque.
- Get Max: Print the first element of the multiset.
- Get Min: Print the last element of the multiset.

Problem H: Minions and War

Setter: Emrul Chowdhury
Alternate: Md. Imrul Hassan, Miloslav Brožek, Tahsin Masrur

Editorial: We need to make all element equal in this problem and also find minimum cost. This problem can be solved in many was. Let describe author's solution.

Firstly, count the frequency of every element of the array. Let's calculate the total sum of the array. Initialize two variable with 0. Let call them **cur** and **tot**. Then, let's iterate every possible element **i**, from **1 to 1000000**, for every **i**:

- Add **frequency[i] × i** to **cur**, and **frequency[i]** to **tot**.
- Now, the calculation part for **i**, we need to increase **(tot × i) - cur** for making **tot** elements equal to **i**, so we need **P × ((tot × i) - cur)** here. On the other hand, we need to decrease **(sum - cur) - ((n - tot) × i)** for making **n-tot** elements equal to **i**, so we need **Q × ((sum - cur) - ((n - tot) × i))** cost here. So we need total **P × ((tot × i) - cur) + Q × ((sum - cur) - ((n - tot) × i))** cost to make every element equal to **i**.

The minimum cost among every possible **i** is the answer.

Problem I : Mr. Savior

Setter: Avik Sarkar
Alternate: Miloslav Brožek

Editorial:

Tag**: 0/1 BFS, Dijkstra**

Run a 0/1 BFS from the '$' notation. When a '#' encountered level value will increase by 1. When a '*' encountered don't move in that way. Otherwise, keep the current level value as similar as the previous level.
Figure out the lowest among the sentinel grid values.
Tutorial: https://pravalsinghal18.wordpress.com/2016/07/31/0-1-bfs/

This problem can also be solved by Dijkstra's Shortest Path Algorithm.

Problem J: Poga The War Stopper

Setter: Tahsin Mashrur
Alternate: Md. Imrul Hassan, Miloslav Brožek

Editorial:
https://docs.google.com/document/d/1dQxYjp-CvgIddG1VjanGgDY_eEOXwq4kRA1LF5T3bxk/edit

Problem K: Mr. X and His String

Setter: Moudud Khan Shahriar
Alternate: Tahsin Masrur

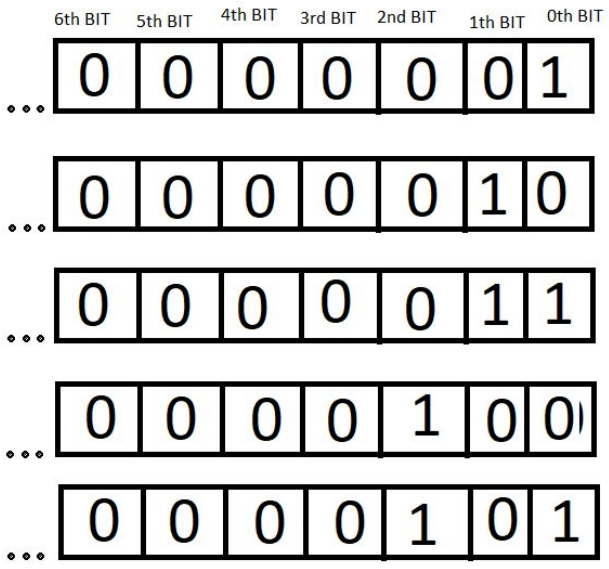
Alter's Editorial:
Let's try to solve the problem for M = 1 first. We can take all substrings as segments, calculate value of string for taking each substring, find expected value overall. But this becomes $O(n^2)$. Let's see if we can reduce the time complexity. Suppose we want to calculate the value each character contributes individually. So, we need to know the probability of each character being in a substring. Imagine the left

and right end of a substring. The left end of a substring that contains i-th character can be in any position from 1 to i (for 1-indexed string), and the right pointer can be in any position from i to n. So the number of such substrings is i*(n-i+1). Since there can be n*(n+1)/2 substrings in a string, the probability of the i-th character being in a substring becomes p = 2*i*(n-i+1)/(n*(n+1)).
The probability of not being is just 1-p.
Now, for m > 1, we can create a recurrence relation, E(m, ch) = p*E(m, ch) + (1-p)*E(m, ch), where E(m, ch) finds expected value ch character contributes for a certain m. This then can easily be solved with DP.

Problem L: Playing With BITs

Setter: Jisan Sheikh
Alternate: Emrul Chowdhury

Editorial: Simply you have to find how many numbers are there in the range l to r whose kth bit is 1. And sometimes you have to update. So you can use any suitable data structure for this purpose. As there is only 61 bit you can easily update your data with **O(61 * log(n))** and query as well.
Initial frame for the sample input is



As first query is 1 2 4
Then the frame will be like
...0001
...0100
...0011
...0100

...0101
Next query is 2 4 4 3
...0001
...0100
...0011
...0100 (Elements in range marked as red color and 3rd bit is marked as green color . you can easily see that there is no element in the range whose 3rd bit is 1 so answer is 0.)
...0101
Next query is 2 2 4 2
...0001
...0100
...0011
...0100
...0101
Here the answer is 2 cause there is two elements in the range whose 2nd bit is 1.
As there no more query of second type we won't simulate the remaining queries.

Problem M: Sabiha and the War Victims

Setter: Md. Shahidul Islam
Alternate: Miloslav Brožek

Editorial:
First compress the numbers, so that their values are at most n. Maintain a frequency array **cnt[]**, where **cnt[i]** represents the frequency of number **i**.

Suppose, when you are at vertex **v**, you know the frequency of each numbers in the subtree of vertex v. Then how can you calculate the result for that vertex? If the size of subtree of vertex **v** is **S,** then you can find the answer by following cases:

i) Case - **If S is odd** and two or more numbers has odd frequency, or no value has odd frequency then answer is 0.
ii) Case - **If S is even** and one or more numbers has odd frequency frequency then answer is 0.
iii) In other cases, you can permute the numbers to find the answer.

How to find the answer by permuting the numbers? If subtree size of a vertex **v** is **S,** then you only need to permute floor(S/2) numbers of any side(left or right), the other side will be same as this side.

So answer is:

$$\frac{(S/2)!}{(\frac{cnt[1]}{2})!\ .(\frac{cnt[2]}{2})!\ldots\ldots\ldots\ldots\ldots\ .(\frac{cnt[n]}{2})!}$$

Let R = $(\frac{cnt[1]}{2})!\ .(\frac{cnt[2]}{2})!\ldots\ldots\ldots(\frac{cnt[n]}{2})!$, initially R = 1

You need to pre-calculate the modular factorial and inverse modular factorial value to do the calculation fast.

Now, question is how to calculate the answer when we move to another vertex **u**?

When we move from vertex v to its child vertex u, cnt[c[u]] will increase by 1. When we move from vertex v up to its parent, then cnt[c[v]] will decrease by 1 (or, will remain same if v is a heavy vertex).

So, when we update the cnt[] value of a number we need to divide R by previous factorial of its cnt[] value, and multiply R by the factorial of its current cnt[] value.

To do all the operation efficiently and achieve O(nlogn) complexity you can maintain DSU on tree. When you go down, add +1 to the cnt value of that vertex and when go up add -1 to the cnt value of that vertex, if it is light or keep unchanged if it is heavy.

Overall time complexity is O(nlogn)

Implementation: [author's solution]