The 2020 ICPC Caribbean Finals

Qualifier

Editorial real contest

June 26^{th}, 2021

Problem A: Lost Key

author: Roberto Abreu

First to solve: 5 minutes by UH TOP
Shortest judge/team solution: 284 bytes/592 bytes
Accepted/tried solutions: 40 teams/44 teams

We can find key k with the first m values of a and b by applying the formula:

$$k[i] = \texttt{'a'} + (b[i] - a[i] + 26) \mod 26$$

Now we most use the remaining $n - m$ values of a and b to check if our key meets the formula given in the problem's description. If we found a contradiction, then the answer is -1, otherwise we can conclude we found the key.

Complexity is $\mathcal{O}(n)$.

Alternate solution:

Since the key length, m, is up to 4, it is possible to brute force all possible key values and check whether any of them produce the encrypted text from the plain text.

Complexity is $\mathcal{O}(26^m \cdot n)$.

Problem B: Maximum GCD Sum

author: Carlos Joa

First to solve: 12 minutes by UH TOP
Shortest judge/team solution: 531 bytes/500 bytes
Accepted/tried solutions: 23 teams/28 teams

Per the definition of gcd, any divisor d of set S must satisfy $d \leq gcd(S)$. Hence, it trivially follows that $|S| \times d \leq |S| \times gcd(S)$.

This allows us to iterate over all possible divisors d (from 1 up to $max(A)$) and take the one that maximizes the expression $|S| \times d$, where d divides all numbers in set S. In this analysis, let's name the expression $|S| \times d$ as $score(d)$. Note that we may be considering divisors d that are not equal to $gcd(S)$, so the expression will not result in a correct "score" for these divisors. However, due to the above observation, this does not matter because when we (later) process the divisor that is the gcd of S, that score will always be higher than $score(d)$ that was computed earlier with a smaller d.

The problem boils down to determining, for each divisor d, the maximal set S: we simply pick all multiples of d present in set A. To speed up this computation, for each possible divisor, we precompute a count of how many numbers in list A are multiples of d. Another possible way to precompute these counts is to use sieve, but it is not required to solve this task.

Complexity is $\mathcal{O}(N \cdot sqrt(M) + M)$ where $M = max(A)$. If you use sieve, complexity is $\mathcal{O}(N + M \cdot log(M))$

Problem C: Find the Point

author: Ernesto David Peña Herrera

First to solve: 144 minutes by UH TOP
Shortest judge/team solution: 1723 bytes/1497 bytes
Accepted/tried solutions: 4 teams/10 teams

Let's first suppose we are solving the problem for a one-dimensional grid and we can select a marked cell. The solution to this problem is well know to be reached at the median of all coordinates, and moreover, our solution decreases as we select a cell closer to the median.

Now suppose the same one-dimensional problem, but we cannot select a marked cell. What happens if n is odd, or is even and there are no empty cells in between the two middle values? In this case we want to get as closer as possible to the median, that implies our selected cell must be adjacent to a marked cell, otherwise we can continue moving our cell towards the median value and improving the solution.

Then we can see our two-dimensional problem as two one-dimensional problems, one for the x coordinate and one for the y coordinate. If n is odd and the cell at given coordinates by the median of xs and the median of ys is empty, then that cell gives the best solution. Otherwise we need to check the 4 adjacent cells to the optimal cell (a marked cell), and if those 4 are marked, continue doing that. We can see that we will have at most $4 \cdot n$ candidates to be our solution (all adjacent and unmarked neighbors of the given cells).

We just need to calculate the sum of Manhattan distances from one cell to all others optimally.

Note that given the nature of this distance, the calculations can be done for xs and ys independently and they are symmetrical, so we will only discuss how to do it for xs coordinates. If we have all xs sorted, let $Sl(i)$ be the distance from the $i\text{-}th$ sorted x to all its preceding given coordinates. We can see that:

$$Sl(i) = \begin{cases} 0 & \text{for } i = 1 \\ Sl(i-1) + (i-1) \cdot (X[i] - X[i-1]) & \text{for } i > 1 \end{cases}$$

where $X[i]$ is the $i\text{-}th$ x in X, the ordered list of all the xs.

We will have to construct the list X, and to precompute Sl for every element of X.

When having a coordinate x, we need to find the lowest position i such that $x \leq X[i]$, we can do that with binary search in $\mathcal{O}(\log(n))$. Now we need to calculate the sum of distances from x to all the coordinates on the left side, let's call it $L(x)$.

Is not difficult to see that:

$$L(x) = \begin{cases} 0 & \text{if } i = 1 \\ Sl(i-1) + (i-1) \cdot (x - X[i-1]) & \text{if } i > 1 \end{cases}$$

By the same analysis we can calculate the distances for the right side and for the ys coordinates as well. Finally we take the minimum over all candidates.

Complexity is $\mathcal{O}(n \cdot \log(n))$.

Bonus: Can you find a $\mathcal{O}(n)$ solution?

Problem D: Bracket sequence

author: Ernesto David Peña Herrera

First to solve: 45 minutes by UH TOP
Shortest judge/team solution: 1384 bytes/1461 bytes
Accepted/tried solutions: 9 teams/10 teams

Let's say $F(x)$ is the number of balanced bracket sequences (BLC from now on) not lexicographically greater than string x. Then the answer to the problem will be given by $F(b)-F(a)+check(a)$, where $check(x)$ returns 1 if x is a balanced bracket sequence, or 0 otherwise.

First let's define the balance of a string as the difference of amount of opening brackets minus the amount of closed brackets. Let's say a string is good if its balance is greater than or equal to 0. You may note that a string is a BLC if all its prefixes are good and its balance is 0.

To calculate $F(x)$ we iterate over all prefixes of x with size $t \in [0, n-1]$ in increasing order and count the amount of BLC not greater than x with that prefix. You may note this is sufficient to count all strings. A special case is when x is itself a BLC. If at some point we reach a prefix which is not a good string, we stop iterating since all the subsequent prefixes will have a closing bracket with no previous opening bracket and thus it won't be a BLC.

For a fixed prefix of size t, we can try to put '(' or ')' at position $t+1$. If $x[t+1] =$ '(' we can only put '(' at position $t+1$ because strings must be not lexicographically greater than x and we should continue to next prefix. Suppose $s[t+1] =$ ')' and b is the balance of prefix with size t, we can put ')' and continue to next prefix or put '(' and count the number of valid suffixes, that is, the number of strings of size $n-t-1$ with balance $-(b+1)$ such that every opening bracket can be matched with a unique closing bracket at a later position. That is the same as calculating the number of strings of size $n-t-1$ with balance $b+1$ such that all it's prefixes are good (if you reverse the suffix).

Let's say $f(n, b)$ is the number of strings of size n with balance b and all its prefixes are good. Then f can be defined by the following recursion:

$$f(n,b) = \begin{cases} 0 & \text{if } b < 0 \text{ or } b > n \\ 1 & \text{if } n = 0 \text{ and } b = 0 \\ f(n-1, b-1) + f(n-1, b+1) & \text{otherwise} \end{cases}$$

where the last case just tries to put '(' or ')' at position n and solve for $n-1$. This can be computed in $\mathcal{O}(n^2)$ with dynamic programming.

$check(x)$ can be implemented in $\mathcal{O}(n)$ by just checking all prefixes of x are good.

The final time complexity is bounded by the computation of function f and it's also $\mathcal{O}(n^2)$, which is enough for this problem constrains.

Bonus: Can you solve it for n up to 10^5?

Problem E: Counting multiples

author: Rubén Alcolea Núñez

First to solve: 78 minutes by UH TOP
Shortest judge/team solution: 4467 bytes/2635 bytes
Accepted/tried solutions: 2 teams/16 teams

This problem is a classical range query problem where we need to solve two kind of queries:

- **q i j:** Within the interval $[i, j]$, find the number of **distinct** multiples of 2 and 3, but not both.
- **u p v:** Update the value of the array at index p with the value v, that is set $A[p] = v$.

Solution 1: MO's Algorithm with updates

The solution involves a slight modification of Mo's Algorithm to handle updates. Mo's algorithm works based in two key ideas:

- All the queries are known before starting to solve them, so we can answer them offline. We order the range queries by the tuple $(\lfloor \frac{l}{B}, r \rfloor)$ with $B = \sqrt{n}$ and compute the answers in this order. It means we split the array into $\sqrt{n}$ blocks of size $\sqrt{n}$. First answer all range queries that start in the first block (sorted by their range ends), then all range queries that start in the second block, an so further.
- The second idea is to use a data structure that represents a specific range $[L, R]$, which can give us the answer to its range quickly. It's also possible to add/remove a single number quickly to modify its range. We will maintain this data structure updated the entire time, starting with the empty range [0,0).

In this problem, the data structure saves the frequency of multiples of 2, 3 and 6 in the current range every time one of these multiples is added or removed. In order to maintain a constant time for operations of adding/removing a multiple, we need to apply coordinate compression to the original values of the array and update operations. We also need to map the coordinates of compressed values to their original values to keep the correct multiplicity of original values. Once we have the information about the multiples, applying inclusion - exclusion, the answer will be

$$multiples_2 + multiples_3 - 2 \times multiples_6$$

.

Assuming that the data structure currently maintains the range $[L, R]$, and the next query covers the range $[l, r]$, then we just need to individually add the parts of the array, that were not covered by the old range but are covered in the new range, to the data structure: $[l, r]/[L, R]$, and remove the parts that were covered before and are now not covered $[L, R]/[l, r]$.

To handle updates we have to add a new dimension to the original version of Mo: time. Because an identically query (l, r) asked multiple times, can give different answers depending on the updates in the time between them. So we represent each query as a tuple $(l, r, time)$, where $time$ represents the number of update operations that needs to be performed before applying this query.

Again, as before, we just order the queries, this time by $(\lfloor \frac{l}{B} \rfloor, \frac{r}{B}, time \rfloor)$ with $B = n^{\frac{2}{3}}$. Now we have blocks of size B for l and r, and we will answer them accordingly. First the queries with $\lfloor \frac{l}{B} \rfloor = 0$ and $\lfloor \frac{r}{B} \rfloor = 0$ (sorted by $time$), then all queries with $\lfloor \frac{l}{B} \rfloor = 0$ and $\lfloor \frac{r}{B} \rfloor = 1$ (sorted by $time$), and so on.

In this new version, if the data structure currently represents the range $[L, R]$ at time point T, and we want to answer the query $[l, r]$ at time point t, then before adding the parts $[l, r]/[L, R]$ and removing the parts $[L, R]/[l, r]$, we need to perform the updates in the time range $(T, t]$ if $T < t$, or undo the updates in reverse if $T > t$.

The time complexity of solution is estimated by taking into account movements of pointers L and R at time point T: $< L, R, T >$.

- **Left pointer:** $\mathcal{O}(Q \cdot B + N)$
- **Right pointer:** $\mathcal{O}(\frac{N^2}{B} + Q \cdot B)$
- **Time point:** $\mathcal{O}(Q \cdot \frac{N}{B} \cdot \frac{N}{B})$

The total time complexity is $O(Q(B + \frac{N^2}{B^2}))$. For $B = N^{\frac{2}{3}}$, the final complexity is $\mathcal{O}(Q \cdot log(Q) + Q \cdot N^{\frac{2}{3}})$.

Solution 2: 2D Data Structure

Let's suppose we have a data structure to store the elements that are multiples of 2 or 3 but not both (tag condition). Our data structure needs to support adding and removing elements, and counting how many different elements exists in a given range. To do so, let's store for each element the position of its previous occurrence in the array (or 0 if that's the first occurrence), then to answer a given query (a, b) we just have to count the number of element from a to b that are less than a, since it will only happen once for each different value. Note that we'll only add elements that meets the tag condition.

The previous data structure can be implemented in $\mathcal{O}(\log^2(n))$ per update and query with *Segment Tree + BBST* or *BIT + BBST*.

The tricky part is to maintain all the pointers updated, to do so we need to store previous and next occurrences for each element as well as some sort of sets carrying out the positions of all occurrences for each element in order to updates the pointers accordingly. Also you may need to do some coordinate compression due to long range of values.

Final complexity is $\mathcal{O}((n+q)\log^2(n))$, where n is the initial number of elements and q is the amount of queries.

Problem F: Antipalindromes

author: Marcelo Fornet Fornés

First to solve: 5 minutes by UH TOP
Shortest judge/team solution: 262 bytes/396 bytes
Accepted/tried solutions: 16 teams/32 teams

Let's make some observations:

- All antipalindromes have even length, then the length of the smallest antipalindrome is 2.
- If a string s of length t is antipalindrome, then s has at least $t/2$ antipalindrome substrings. $\{(s_1 \dots s_t), (s_2 \dots s_{t-1}), \dots (s_{t/2} s_{t/2+1})\}$

You may note that for every antipalindrome, the substring composed by just the two characters in the middle is also antipalindrome, so we can start counting antipalindromes from the center. For every antipalindrome $s_i s_{i+1}$ of length 2, try to expand this antipalindrome's center as much as possible. That is, check if substrings $s_{i-1} \dots s_{i+2}$, $s_{i-2} \dots s_{i+3}$, etc, are also antipalindromes, until we find that $s_{i-t} = s_{i+t+1}$ for some t (ie. substring $s_{i-t} \dots s_{i+t+1}$ is no antipalindrome and neither its further expansions). If at some point our count reaches 10^5, stop counting and output this number.

At each step we either increase our count or discard some center, then the final complexity is $\mathcal{O}(n+p)$, where n is the size of the initial string and $p = \min(10^5, \#\, antipalindromes)$.

Problem G: Evolution

author: Alexander Bestard

First to solve: 8 minutes by 3N1?M4
Shortest judge/team solution: 328 bytes/399 bytes
Accepted/tried solutions: 52 teams/55 teams

This problem can be solved by just simulating the described process. We can compute the state at iteration i from state at iteration $(i-1)$ in $\mathcal{O}(n)$.

Since we need to do this for m iterations final complexity is $\mathcal{O}(n \cdot m)$.

Problem H: Round table

author: Ernesto David Peña Herrera

First to solve: 95 minutes by UH TOP
Shortest judge/team solution: 573 bytes/13776 bytes
Accepted/tried solutions: 1 teams/2 teams

Single-Case Solution(Bonus):

An approach to solve this problem would be to apply Inclusion-Exclusion Principle on the amount of enemies sitting together. Thus we directly obtain the following formula:

$$f(n) = \sum_{k=0}^{n} (-1)^k \cdot \binom{n}{k} \cdot 2^k \cdot 2n \cdot (2n-k-1)!$$

Where k is the amount of fixed pairs of enemies sitting together. We add or subtract depending on k's parity. The $\binom{n}{k}$ factor is the number of ways of selecting k pairs of enemies. From now on, we'll consider the k fixed pairs like k persons occupying 2 consecutive seats each one, and for each pair there are two ways of sitting them together, so we need to multiply by 2^k. Now we need to count the ways to distribute the final $2n-k$ persons and there are in total $(2n-k)!$, but we need to divide by $2n-k$ to remove all circular permutations and then multiply by $2n$ to count the real number of circular permutations since there are actually $2n$ persons rather than $2n-k$.

The complexity for computing the previous formula is $\mathcal{O}(n \cdot \log(n))$ or $\mathcal{O}(n)$, depending on how you calculate modular inverses for factorials. But since the problem asks to solve t different test cases, actual complexity is $\mathcal{O}(t \cdot n)$ or $\mathcal{O}(t \cdot n \cdot \log(n))$, which unfortunately is not enough for getting AC.

Multi-Case Solution:

The idea is to transform the problem into a "simpler" one. Let $F(n)$ be the number of ways of matching $2n$ seats into pairs, in such a way that no two matched seats are adjacent. For $F(n)$ we allow the first seat to be matched with the last one. The solution to the original problem will be given by:

$$f(n) = (F(n) - F(n-1)) \cdot 2^n \cdot n!$$

where $(F(n) - F(n-1))$ is the number of ways to match seats taking into account that first seat must not be matched with the last one (remember that table is circular). We multiply by 2^n because for each of the n pairs of persons there are two ways to place them on a pair of matched seats, and multiply by $n!$ to count the number of ways to distribute the n pairs of persons among the n pairs of matched seats.

Now let's calculate $F(n)$. It is easy to see that $F(0) = 1$ and $F(1) = 0$. Suppose our table is a line where all the matched seats are located, for example suppose the table a, b, a, c, b, c for $n = 3$, same letter stands for a matched pair of seats. Let's try to add a new pair of matched seats. To avoid over-counting, one of the them will always be in the last position of the line, and we need to check the valid positions for the other. Following the same example, we have $*, a, *, b, *, a, *, c, *, b, *, c, d$, that means we have 6 (the amount of *s) ways to place the other seat d. In a more general sense there are $2(n-1) \cdot F(n-1)$ ways to do this.

With this approach we are not counting seatings like a, d, a, b, c, b, c, d because $F(3)$ does not count the distribution a, a, b, c, b, c since there are two matched adjacent seats. The same happens with a, b, c, b, a, c since $F(2)$ does not count a, b, b, a as a valid distribution.

Let's denote as i the depth of nesting pairs of matched seats, around a fixed seat; for example, for a, d, a, b, c, b, c, d, the first d in the a, d, a context has $i = 1$; for a, b, c, b, a, c, the first c in the a, b, c, b, a context has $i = 2$. For every i from 1 to $n-1$, we are going to consider the seat we are trying to place and its context as one single seat (composed by $2i + 1$ adjacent seats), and take a look at how many ways there are to place that big seat along with the remaining $2(n - i - 1) + 1$ seats. You may note there are $[2(n - i - 1) + 1] \cdot F(n - i - 1)$ ways to do so (using the same analysis with the *s, in this case we can place our big element of $(2 \cdot \mathrm{i} + 1)$ seats between all the other seats except in the last position, because we force that seat and we have to multiply it by $F(n - (i + 1))$ because the $2 \cdot i$ matched seats are in our big element with one of the *n-th* pair, and we cannot count any of these i+1 pairs of seats).

The resulting expression for $F(n)$, for $n > 1$ is the following:

$$F(n) = 2(n-1) \cdot F(n-1) + \sum_{i=1}^{n-1} [2(n - i - 1) + 1] \cdot F(n - i - 1)$$

With the above formula we can compute all $F(n)$ in $\mathcal{O}(n^2)$ and answer $f(n)$ in $\mathcal{O}(1)$, but this does not improve our previous solution in the worst case. Let's consider $j = n - i$, and after some algebra we get:

$$F(n) = 2(n-1) \cdot F(n-1) + \sum_{j=1}^{n-1} (2j - 1) \cdot F(j - 1)$$

Let's say $G(1) = 1$ and for $n > 1$:

$$G(n) = G(n-1) + (2n-1) \cdot F(n-1)$$

then:

$$F(n) = 2(n-1) \cdot F(n-1) + G(n-1)$$

That way we can easily keep track of $G(n-1)$ as we compute $F(n)$. For the next step we would be able to calculate $G(n)$ and $F(n+1)$ in $\mathcal{O}(1)$.

The final time complexity is $\mathcal{O}(n+t)$. As a note, we need to check separately the special case of $f(1) = 0$.

Problem I: Weighted components

author: Marcelo Fornet Fornés

First to solve: 38 minutes by UH TOP
Shortest judge/team solution: 2396 bytes/2378 bytes
Accepted/tried solutions: 5 teams/6 teams

First, let's sort all elements of the matrix and process the queries in increasing order of k. Also, let's use a disjoint-set data structure to maintain connected components and their sums. When processing a query with value k, we are going to add to our data structure all cells whose values are less or equal to $2k$, that we haven't added so far. Now we have all information we need to answer that query since we don't care about cells with values greater than $2k$.

When adding a cell to our data structure, we first create a node (single connected component) for that cell and record its sum as the value on that cell. Then, we should connect this single node to all adjacent and previously existing components, using the disjoint-set.

We claim that if there exists a connected component with sum greater or equal to k, then the answer is `"YES"`, otherwise the answer is `"NO"`.

The second fact is pretty obvious. Let's prove the first one. Select some connected component whose sum is at least k. If that component has at least one cell greater or equal to k, we're done. Now assume all cell's elements are less than k. Pick up one cell and put it in a set S, we can safely select any of its adjacent cells (in the same connected component) and add it to S, since both of them are less than k and their sum won't surpass $2k$. If their sum is at least k we're done, otherwise we can follow the same strategy and we have proved that by adding one single element at the time, we won't get a sum larger than $2k$. Since this component's sum is at least k, eventually we'll get a sum in range $[k, 2k]$.

Complexity is $\mathcal{O}((n \cdot m) \cdot log(n \cdot m) + q \cdot \log(q))$ since we need to sort all queries and cells.

Problem J: Coin

author: Marcelo Fornet Fornés

First to solve: 196 minutes by UH++
Shortest judge/team solution: 1155 bytes/1611 bytes.
Accepted/tried solutions: 2 teams/3 teams

If nodes 1 and N are directly connected by an edge the answer is -1, since player B can not lock node N.

Let's try to check if player B can prevent player A from reaching node N for a fixed value of k. For this, let's check if it's possible for player A to reach node N instead. These are complementary problems and solving one of them, we get the other's answer. You may note the following observations:

- It is possible to reach N from all its adjacent nodes.
- If node u is not adjacent to N, it is possible to reach N from u if there are at least $k+1$ adjacent nodes to u that can reach node N.

These observations directly gives us a method to solve the problem. Let's do a *BFS*, starting from all adjacent nodes to N. At any moment we only keep in the queue nodes that can reach node N. Also we will only add a new node v to the queue the $(k+1)$-*th* time that an adjacent node from the queue tries to add v to the queue. For this we just need to keep a count for every node, and increase it every time an adjacent node tries to add v. In the end, if node 1 was in the queue at some point, then player A will be able to reach N.

Finally, to get the minimum k we can do a binary search over the value of k (it is easy to see why binary search works in this case).

Complexity is $\mathcal{O}((n+m)\log(n))$.

Problem K: String operations

author: Alberto Rosales

First to solve: 42 minutes by The Last Dance
Shortest judge/team solution: 2576 bytes/2703 bytes
Accepted/tried solutions: 6 teams/12 teams

We will build a *Trie* containing every string in the array A. To answer all queries efficiently, we will add two sets of indices (C and F) to every node of the *Trie*.

- C_u will store the indices of strings that visited node u during its insertion process in the *Trie*.
- F_u will store the indices of strings that finished its insertion process in node u of the *Trie*.

Given that we will modify the *Trie* structure, we define $insert(s, index)$ as a function that inserts a string s in the *Trie* and adds $index$ to the set C of every node visited in the process; it also adds $index$ to the set F of the final node.

Similarly, we define $delete(s, index)$ as a function who simulates the insertion process of string s in the *Trie* and removes $index$ from the set C of every node visited in the process and also from set F of the final node.

So, before we can answer any query, we do $insert(a_i, i)$ for $1 \leq i \leq n$. Once all strings are inserted in the *Trie*, we can answer all queries like this:

1. Type 1:
   - $delete(a_i, i)$
   - $a_i = s$
   - $insert(a_i, i)$
2. Type 2: We will define a function $has\_prefix(s, i, j)$ that returns $True$ if the interval $[i, j]$ contains some string that is prefix of s and $False$ otherwise.
   To check this, we simulate the insertion process of string s in the *Trie*.
   Suppose we are currently in node u of the *Trie*, we need to check if there exists some index $i \leq x \leq j$ in F_u, which can be done with binary search if the sets are maintained ordered. If no such index is found on any visited node, the answer is $False$; otherwise it is $True$.
   If at any moment we cannot make a transition, we return $False$ immediately.
3. Type 3: We will define a function $is\_prefix(s, i, j)$ that returns $True$ if s is prefix of some string of some string in interval $[i, j]$ and $False$ otherwise.
   To check this, we also simulate the insertion process of string s in the *Trie*.
   Suppose the node u is the final node after the insertion process of s, we need to check if there exists some index $i \leq x \leq j$ in C_u, which can be done similarly as

in Query of type 2.

If at any moment we cannot make a transition, we return False immediately.

Complexity of the $insert$, $delete$, has_prefix, and is_prefix functions are $\mathcal{O}(S \cdot log\, N)$ where S is the length of the string to be inserted, removed or queried, and N is the number of strings in the array. The $log\, N$ factor is due to the need to keep sets C and F sorted as well as the binary searches on those sets. At first, it seems that doing $N + Q$ operations on the $Trie$ would be too expensive, but it turns out that the amortized complexity of all operations is bounded by $\mathcal{O}(L \cdot logN)$, the sum of lengths of all strings in the input (present initially in array A as well as the ones in all queries). Since L is no greater than $5 * 10^5$, this solution should run within the given time limit.

Alternate solution: We may replace the $Trie$ data structure with $Prefix\ Hashes$, but time complexity remains the same.

Problem L: Competition

author: Alberto Rosales

First to solve: 33 minutes by Team3C-1 [VC]
Shortest judge/team solution: 648 bytes/928 bytes
Accepted/tried solutions: 14 teams/32 teams

We can model the problem as a tournament graph, a directed graph where each pair of (distinct) vertices u and v are connected by exactly one directed edge, either from u to v or v to u.

The problem of finding a set of three contestants without a clear winner is the same as finding a (directed) cycle of length 3 in this tournament graph. In any tournament graph, three vertices will either:

- form a cycle, which corresponds to the situation where there is no clear winner.
- will contain a vertex that has edges leading to the other two vertices. In this case, this vertex corresponds to the contestant with most wins against the other two contestants.

Now, the key observation to solve this problem: In a tournament graph, if there is a cycle of any length, there exists a cycle of length 3. The proof is based on the following algorithm to find such a 3-length cycle:

1. Find any cycle $c = u_1, u_2, \ldots, u_k$, where there are edges $u_i \to u_{i+1}$ (for $1 \leq i \leq k$) and $u_k \to u_1$.
2. If $k = 3$, return c.
3. Otherwise, take 3 consecutive vertices in c, say x, y, z.
   (a) If there is an edge $z \to x$, return (x, y, z)
   (b) Else, remove vertex y from c and repeat from step 2 onward.

Final complexity: $\mathcal{O}(N^2)$.

Alternate solution: given the constraint on this problem, it is possible to solve this problem by finding a triple of vertices u, v, and w such that there are edges $u \to v$, $v \to w$, and $w \to u$.

The brute force solution looks as follows:

```
for u := 1 to N
   for v := 1 to N
      for w := 1 to N
         if there are edges u -> v, v -> w, and w -> u
            return (u, v, w)
```

Complexity is $\mathcal{O}(N^3)$. Since N is up to 2000, this should time out. To speed this up, we can use bitset intersection to replace the innermost loop, saving us a factor of 32 or 64 (depending on the machine word size). We have two bitsets that we want to intersect:

1. the bitset corresponding to the set of contestants that v defeats
2. the bitset that corresponds to the set of contestants that u loses to

Problem M: Even split

author: José Carlos Gutiérrez

First to solve: 173 minutes by UH TOP
Shortest judge/team solution: 5249 bytes/3907 bytes
Accepted/tried solutions: 1 teams/1 teams

Let S be the sum of all the values in the matrix. If the sum is odd there is no solution, otherwise let $T = \frac{S}{2}$. The problem asks to find a path such that the sum of all elements to the right of it (as well as the sum of all elements to the left of it) is equal to T.

Let $P_{x,y}$ be a valid partial path starting in the top left corner and ending in the vertex (x, y). We define $V(P_{x,y})$ as the sum of all elements to the right of the path. Let $W(x, y) = \{V(P_{x,y}) | P_{x,y}$ is a partial path ending at $(x, y)\}$. This means that $W(x, y)$ is a set containing all values v such that there is a path ending at (x, y) whose sum of all elements to the right of this path is v.

Let $B(x, y)$ be the sum of all elements in the cells of the rectangle that spans from (x, y) to (n, m). For a partial path $P_{x,y}$ to be a potential answer, it is necessary that $V(P_{x,y}) \leq T$, because while the path is extended, the sum of the elements to its right is non-decreasing. On the other side, the remaining elements should be enough to achieve the target sum T, so the following must be true as well: $T \leq V(P_{x,y}) + B(x, y)$. This gives the following inequality:

$$T - B(x, y) \leq V(P_{x,y}) \leq T$$

.

We can reduce the candidate paths among those ending at (x, y) to be those in the set $W'(x, y) = W(x, y) \cap [T - B(x, y), T]$.

Suppose we compute for every (x, y) the set $W'(x, y)$. If at any point this sets has two consecutive elements, p and $p+1$, then there is a path that splits the matrix evenly. To see this property, start by creating a path that starts at (x, y) and ends in (n, m) going first to the right as much as possible and then down until the end. This suffix path adds 0 to any partial path ending at (x, y). We can modify this suffix path to change one cell from the left to the right, one step at a time. If we keep the current sum to the right of this suffix as s we can include cells while $s + p + 1 < T$.

When $s + p + 1 \geq T$, two options are possible. Either $s + p + 1 = T$ or $s + p = T$. This is because s increases by 0, 1 or 2 at every step. Since at (x, y), we know a path with value p and another with value $p + 1$, we pick the one that fits the current suffix found.

Now let's discuss how to compute the set $W'(x, y)$. This can be computed from the set $W'(x-1, y)$ and $W'(x, y-1)$. While there are not two consecutive values in this set, it holds that this set is formed by "consecutive" numbers with the same parity, so we can efficiently maintain it updated by only storing the first and the last element.

This is a constructive solution that suggests how to compute the path after we find if it exists or not.

Problem N: Arithmetic Mean

author: Marcelo Fornet Fornés

First to solve: 3 minutes by FreesTyle
Shortest judge/team solution: 300 bytes/450 bytes
Accepted /tried solutions: 57 teams/71 teams

For every triple i, j, k (order matters), out of the four elements of input, check if $i + j = 2 \cdot k$. If at least one triple meets previous condition the answer is `"YES"`, otherwise the answer is `"NO"`.

Complexity is $\mathcal{O}(1)$.

Problem O: Polygon

author: Ernesto David Peña Herrera

First to solve: N/A
Shortest judge/team solution: 4585 bytes/ N/A bytes
Accepted/tried solutions: 0 teams/0 teams

Let's consider the triangulation as a graph, where triangles are the nodes and diagonals are the edges, that is, two triangles are adjacent in our graph if they share a diagonal.

Any triangulation of a polygon with n vertices is composed by exactly $n-2$ triangles. Proof to this fact can be done with induction over the number of vertices of the polygon: Find a diagonal and split the polygon into two pieces, then assume hypothesis for both pieces and merge them.

From input specification we know there are $n-3$ diagonals, so we do not need to prove this fact, but it can be done with the same inductive analysis as before. So we have a graph with $n-2$ vertices and $n-3$ edges. We should note our graph is connected, it's pretty obvious but again it can be proved the same as before. Thus, our graph is a tree.

Then, we just need to build the tree, locate for every point the triangle(node) that contains it, and answer every query with an LCA data structure. For the first two tasks we are going to build a binary search tree with the following properties:

- Every node represents a piece of our polygon, and hence, the set of diagonals that triangulate it.
- The root of this search tree represents the whole polygon.
- The leaves represent the triangles.
- For every node in the search tree, other than the leaves, we are going to pick a diagonal and split the polygon that node represents in two pieces, and these will be the two children for current node.
- Let's say a and b are the amount of vertices of the two pieces after splitting by some diagonal. We are going to pick the diagonal in such a way that $|a-b|$ is minimized, and we are going to also store that selected diagonal in the node's data.

After building this search tree, we can easily get all triangles and build the original tree along with its associated LCA data structure. Also, we can locate for every point, its containing triangle, by just traversing the search tree from the root until we reach a leave, always picking the side that encloses the point (by checking which side of the diagonal contains the point). This search will be proportional to the search tree's height.

Let's prove this is efficient enough. Selecting the diagonal for each node (of the search tree) can be implemented in linear time on the amount of vertices of the piece of polygon. Next thing to note is that the selected diagonal will be an edge of the centroid for tree that represents the triangulation of current piece of polygon. To prove this, suppose you picked a diagonal (edge in the tree) and then consider the path from this edge to the centroid; you may note that any edge (diagonal in the polygon) on this path will improve the value of $|a - b|$. This applies until you reach a centroid's edge.

Another interesting property is that every node in the tree has degree at most 3. Let's consider centroid's degree:

1. $deg(centroid) \leq 1$: This is a trivial case, the tree has either 1 or 2 vertices.
2. $deg(centroid) = 2$: After removing any of the centroid's edges, we get 2 trees with at most half of the vertices. This is actually the best scenario we can get.
3. $deg(centroid) = 3$: We are going to select the edge connecting the centroid with its subtree with the greater amount of vertices. The worst scenario arrives when all 3 subtrees have 1/3 of the vertices.

Let's just focus on the worst scenario, after removing the diagonal corresponding to one of the three centroid's edges, we get two pieces of polygons, with 1/3 and 2/3 of the vertices respectively. So the recursion to build the search tree has the following complexity:

$$T(n) = T(\frac{n}{3}) + T(\frac{2n}{3}) + \mathcal{O}(n)$$

This complexity will be given by the sum for each vertex, of the amount of nodes of the search tree that contains the vertex. The worst case is reached when the vertex is always in the 2/3 side. Then that vertex will be in at most $log_{3/2}(n)$ nodes of the search tree. And then:

$$T(n) = \mathcal{O}(n \cdot \log_{3/2}(n))$$

Finally, locating a point's triangle has a complexity of $\mathcal{O}(\log_{3/2}(n))$ since it is proportional to the search tree's height. LCA construction is $\mathcal{O}(n \cdot \log(n))$ and each query is $\mathcal{O}(\log(n))$.

Final complexity will be: $\mathcal{O}((n + q) \cdot (\log_{3/2}(n) + \log(n)))$, where q is the amount of queries to answer.

Although this solution solve queries in an online way, the problem can be solved offline by locating at the beginning the triangles of each point with a sweep line technique.

Bonus: Can you solve it for a non convex polygon?