Indian National Olympiad in Informatics 2024
Editorial and Results

Indian Computing Olympiad Scientific Committee

May 23, 2024

1 Monsters

Author: Paras Kasmalkar

Preparation: Paras Kasmalkar, Shreyan Ray, Soumyaditya Choudhuri

Abridged statement: There are N Monsters numbered from 1 to N, each of type Fire, Water or Grass represented by a string A of length N. When two Monsters battle each other, one of them wins according to the following rules:

- A Fire Monster always defeats a Grass Monster.
- A Water Monster always defeats a Fire Monster.
- A Grass Monster always defeats a Water Monster.
- In a battle between two Monsters of the same type, either one can win.

Answer Q queries of the following form: if the Monsters $L_j ... R_j$ are standing in a line from left to right, and repeatedly, two adjacent Monsters battle each other with the loser leaving the line, how many Monsters can be the last remaining Monster in at least one valid sequence of events?

Constraints:

- $1 \leq N, Q \leq 10^5$.
- $A_i \in \{\texttt{F}, \texttt{W}, \texttt{G}\}$
- $1 \leq L_j \leq R_j \leq N$

Subtasks:

Constraint for Subtasks 1-9
You only have to answer one query, and this query covers all the Monsters. In other words, $Q = 1$, $L_1 = 1$, $R_1 = N$.

- **Subtask 1** (4 points) There are only Fire Monsters.

- **Subtask 2** (5 points) There are no Grass Monsters.
- **Subtask 3** (7 points) $N = 3$.
- **Subtask 4** (5 points) $N \leq 35$. Further, there are at most 2 pairs of adjacent Monsters who are of different types.
- **Subtask 5** (11 points) $N \leq 35$.
- **Subtask 6** (10 points) $N \leq 80$.
- **Subtask 7** (10 points) $N \leq 400$.
- **Subtask 8** (10 points) $N \leq 1500$.
- **Subtask 9** (16 points) $N \leq 10^5$.

Constraint for Subtasks 10-11
There may be multiple queries.

- **Subtask 10** (6 points) There are no Grass Monsters.
- **Subtask 11** (16 points) No additional constraints.

1.1 Subtask 1 (4 points): $Q = 1$, Only Fire Monsters

For every battle, either of the two monsters can win as they are both of Fire type; thus, in the end, any Monster can be a winner. The answer is simply N.

1.2 Subtask 2 (5 points): $Q = 1$, No Grass Monsters

Water Monsters always defeat Fire Monsters, and are thus at an advantage. No Fire Monster can win if there are any Water Monsters, while any of the Water Monsters can be a winner similar to the first subtask.

Thus, if there are no Water Monsters, the answer is again simply N, otherwise it is the number of Water Monsters.

1.3 Subtask 10 (6 points): No Grass Monsters

We need prefix sums to speed up the solution to subtask 2. We can use prefix sums over the number of Water Monsters to calculate the number of Water Monsters in the range $[L_j, R_j]$. If there is at least one Water Monster, we can output the number of Water Monsters. Otherwise, every Monster in the range is a Fire Monster and any of them can win, so the answer is $R_j - L_j + 1$.

1.4 Subtask 3 (7 points): $Q = 1, N = 3$

One way to solve this subtask is to manually compute the answer for each of the $3^3 = 27$ inputs possible.

A less tedious solution involves dividing the problem into the following cases:

- $A_1 = A_2 = A_3$, i.e. all Monster types are the same : Output $N = 3$ as any of them can win.
- Either $A_1 = A_2$ or $A_1 = A_3$ or $A_2 = A_3$, i.e. there are 2 distinct types of Monsters: Suppose there are only Fire and Water types, then the answer is just the number of Water types. Similarly, in cases of Fire and Grass types, Fire types win, and in cases of Water and Grass types, Grass types win.
- All Monsters are of distinct types: We claim that the middle Monster can never win, while both of the other monsters can end up winning. You can try to see this through the case $A =$ "`FWG`".
  For the Fire Monster to win, Grass first beats Water and then Fire beats Grass.
  For the Grass Monster to win, Water first beats Fire and then Grass beats Water.
  The Water Monster cannot win, as there is no way for it to beat the Grass Monster.
  In general, first the enemy (i.e. the Monster which can defeat it) of the winner is taken out, and then the Monster ends up winning. However, for the middle Monster this is not possible as its enemy and its enemy's enemy are not adjacent.

1.5 Subtask 4 (5 points): $Q = 1, N \leq 35$. Further, there are at most 2 pairs of adjacent Monsters who are of different types.

The solution to this is a compression idea building off the previous subtask of $N = 3$. Monsters which share the same type and are adjacent can behave in essentially identical ways, and thus if one of them can win so can the other.

Thus, we compress the string A to an array of pairs (b_i, f_i) where that pair represents that there are f_i continuous occurrences of the character b_i in the string A. Thus reduces the problem to $N \leq 3$ (but if you don't want to make special cases for $N = 1$ and 2, you can simply append $(\texttt{F}, 0)$ to the end of your array of pairs until it is of size 3).

1.6 Subtasks 5 - 9 (11 + 10 + 10 + 10 + 16 points): $Q = 1, N \leq 35$ to $N \leq 10^5$

Instead of dealing with these subtasks in a linear fashion, we will discuss various approaches and the optimizations one can do to pass certain subtasks. Further, we only discuss about counting the number of winning Fire Monsters. It is trivial to extend any solution to all types of Monsters (as they all have essentially the same behavior in the sense that every type has a type it beats and every type has a type it gets beaten by). We can find the overall answer by adding up the answer for each type.

1.7 Dynamic Programming

Refer to the codes provided for exact details of the transitions.

1.7.1 Approach 1

We will use the idea of dynamic programming over ranges. Let $\texttt{dp}(l, r, i)$ denote whether the Monster numbered i can win in the range $[l, r]$ or not. Then, the answer is simply the number of i such that $\texttt{dp}(1, n, i)$ is `true`.

Consider the last 2 remaining Monsters in the range $[l, r]$, let them be x and y ($x < y$). Every monster in the range $[l, r]$ lost to exactly 1 of x or y directly or indirectly. The crucial observation here is that there exists some index m such that for all $l \leq i \leq m$, every Monster in the range $[l, m]$ lost (directly or indirectly) to x and every Monster in range $[m + 1, r]$ lost (directly or indirectly) to y.

We can brute force over all possible m and try all possible values of x and y. This approach has a time complexity $O(N^5)$ as there are $O(N^2)$ ranges $[l, r]$ and we try $O(N)$ values each of m, x and y. This is sufficient to pass subtask 5. Code

The first optimization we can make is instead of trying all values of x and y, we simply try all values of x and see if we get a y we can win against. If x is of Fire type for instance, we need to only check if there are any y which can be of Fire or Grass types. To do this fast, we will compute the number of winning Fires and Grass beforehand in the state $\texttt{dp}(m + 1, r, i)$ and then check if either is greater than 0. Remember to also try all values of y and make it win against x. This approach takes $O(N^4)$ and it is sufficient to pass Subtask 6. Code

1.7.2 Approach 2

Let $\texttt{dp}(l, r, t)$ be true if and only if a monster of type t can win over the range $[l, r]$. This is easy to calculate in $O(N^3)$ by trying all midpoints similar to Approach 1.

Now, we will compute another dynamic programming state: $\texttt{dp2}(l, r, t)$ which answers the following question: if a Monster of type t wins over the range $[l, r]$, can it be the overall winner? To compute this, our base case is $l = 1, r = N$ and not the usual $l = r$. We compute the DP for smaller ranges based on the DP values of bigger ranges that encompass them. To compute this DP, we can use the values of $\texttt{dp}(l, r, t)$ and try extending the current interval by having the winner fight with a winning type of a neighboring interval. This takes $O(N^3)$ time. Code

1.7.3 Approach 3

Observation

Suppose Monster i can win over the entire array, then there exists at least one sequence of battles such that the Monster i only battles against the Monster who won in the range $[1, i-1]$ and the range $[i + 1, N]$.

Non-Formal Proof

Without loss of generality, assume Monster i is of Fire type. Then, the only bad case is when the winner in the range $[1, i-1]$ is Water type, but Monster i battling some Monsters between the other battles can change the winner type in the range $[1, i-1]$.

However, if Monster i beats some Grass Monster, it only helps Water Monsters in the range $[1, i-1]$ to win. Further, if Monster i beats some Fire Monster, then it might help some Grass Monsters to win; however, if this is true, it also means the winner in the range $[1, i-1]$ could already be Fire type.

Try to come up with a formal proof for the last statement.

Solution

Using the approach above, we can conclude that calculating $\texttt{dp}(l, r, t)$ defined as whether a Monster of type t can win in the range $[l, r]$ is sufficient to calculate the answer. For calculating the answer, a Monster of type i (say it is Fire type) can win if and only if either Grass or Fire types can win in both the ranges $[1, i-1]$ and $[i+1, N]$. This gives us a $O(N^3)$ solution with the slowest step being that of calculating the DP. Code

This can even be sped up to $O(N^2)$. Instead of considering all possible midpoints, just considering the first and last positions of each type in the interval $[l, r]$ to be the midpoint values is sufficient. This reduces the time complexity to $O(6N^2) = O(N^2)$. However, we do not currently have a proof for this. Code

1.8 Greedy Simulation

We can fix a Monster we want to be the winner and then greedily try to create a sequence of battles which are beneficial for it.

Suppose we want Monster i of Fire Type to win. Then, below is the optimal order of choosing which battle we should have next where the best choice is listed first.

1. Battles between adjacent Pokemon of equal types, since these battles don't affect anything.
2. Battles between an adjacent pair of Water Monster and Grass Monster since they will remove Water Monsters, which pose the most danger to our chosen Monster.
3. Battles between an adjacent pair of Fire Monster and Water Monster since they will make the Water Monsters more accessible to the Grass Monsters. (Note that this Fire Monster should not be our chosen Monster.)
4. Battles between an adjacent pair of Fire Monster and Grass Monster. We want to have as few of these battles as possible because we need the Grass Monsters to remain in the line to defeat the Water Monsters.

Naively implementing the algorithm works in $O(N^3)$ as we have to try all N Monsters, then have a total of $N-1$ battles, and each time find the most beneficial adjacent battle by checking $O(N)$ pairs manually.

We can also speed this up to $O(N^2 \cdot \log N)$ by keeping a priority queue of adjacent battles sorted by most beneficial to least beneficial. Then, when a Monster gets defeated, we need to consider the next and previous non-defeated Monsters of this one, make them adjacent, and insert that pair into the priority queue.

1.9 Finding a Necessary and Sufficient Condition

A powerful way of solving this problem is finding a necessary condition for Monster i to win, and then also showing that it is sufficient, or vice versa. This approach can be helpful in many types of problems.

1.9.1 Necessary Condition

For a Fire type Monster i to win, if there is a Water type in the range $[1, i-1]$, there must also be a Grass Type in the range $[1, i-1]$. Similarly, if there is a Water type in the range $[i+1, N]$, there must also be a Grass type in the range $[i+1, N]$.

In general, replace Water by type that can beat the Monster i and Grass by type that Monster i beats. This condition is also sufficient for Monster i to win as we prove below.

1.9.2 Proof of Sufficiency

Suppose we want Monster i to win, who is Fire type. Divide the battles into several stages:

1. Repeatedly, while there is at least one pair of adjacent Fire and Water Monsters (where the Fire Monster in question is not 1), the two battle and the Fire Monster loses. After all possible such battles, if it is possible for 1 to win, every contiguous group of Water Monsters must be adjacent to a Grass Monster.
2. Repeatedly, while there is at least one adjacent pair of Grass and Water Monsters, battle the two and the Water Monster loses. The effect of this is that now there are only Fire and Grass Monsters left.
3. Monster i defeats all other Monsters.

Thus, the condition is sufficient as we see that Monster i is able to win when the condition is true.

1.9.3 Alternative Proof of Sufficiency Using Induction

Suppose we want Monster i to win, who is Fire type. First of all, don't consider equal adjacent Monsters (to make the analysis simpler), further assume that $i = 1$ (so we only need to care about the suffix).

Consider the first position where $A_i = \texttt{G}$, the string up to i then looks like `FWFW....G`. We can reduce this prefix to just `FG` by doing the following inductive steps :

- Base Case : $i = 2$, the string is already `FG`.
- if $A_{i-1} = \texttt{F}$, battle Monster $(i-2)$ and Monster $(i-1)$ (which are `W` and `F` type respectively), thus reduces to a $(i-1)$ case.
- if $A_{i-1} = \texttt{W}$, battle Monster $(i-1)$ and Monster (i) (which are `W` and `G` type respectively), thus reduces to a $(i-1)$ case.
- Hence, by induction, we have reduced the prefix to `FG`.

Now, by a similar inductive casework argument, we can show that the `FG` prefix can always be conserved while N is greater than 3.

- If $A_3 = \texttt{W}$, battle Monster 2 and Monster 3 (which are G and W type respectively), thus reduces to a $(N-1)$ case.
- If $A_3 = \texttt{F}$, $A_4 = \texttt{G}$, battle Monster 2 and Monster 3, the string now looks like `FFG.....` Battle Monster 1 and Monster 2 (and make Monster 1 win) and the string again looks like `FG....`, thus reduces to a $(N-2)$ case.
- if $A_3 = \texttt{F}, A_4 = \texttt{W}$, battle Monster 3 and Monster 4, thus reduces to a $(N-1)$ case.
- Hence, by Induction, we can always keep the FG prefix till we are left with either $N = 2$, $A = \texttt{FG}$, or $N = 3, A = \texttt{FGW}$ or $N = 3, A = \texttt{FGF}$.

For all of the 3 cases in the final base cases, we can easily see that the Monster 1 of F type can win. Hence, the condition is sufficient.

1.9.4 Solution

A Fire Monster wins if and only if both its prefix and suffix either have no Water Monsters or at least one Grass Monster. A naive implementation of this check for each Monster yields a $O(N^2)$ solution. This can be sped up to $O(N)$ by performing one left-to-right sweep to find information about all the prefixes, and another right-to-left sweep to find information about all the suffixes.

1.10 Subtask 11 (16 points): No additional constraints

We consider the number of Fire Monsters that cannot win. Subtracting that number from the total number of Fire Monsters gives us the number of Fire Monsters who can win.

A Fire Monster cannot win when either its prefix or its suffix contains a Water Monster but does not contain a Grass Monster. Let W_{min}, W_{max}, G_{min}, G_{max} be the first and last positions among Water and Grass Monsters respectively. A Fire Monster i is bad if and only if $W_{\text{min}} < i < G_{\text{min}}$ (the prefix is bad) or $G_{\text{max}} < i < W_{\text{max}}$ (the suffix is bad). Note that a

Monster cannot be bad under both conditions as long as there exists at least one Grass type, so we don't double count.

We can find all these four positions by either keeping a set of indices where Monsters of a certain type exist, or by calculating the next/previous position where a Monster of a certain type exists using a dynamic programming approach. After we have those values, we can simply subtract the number of Fires in a specific interval, which can be done with prefix sums. The time complexity is $O(N + Q)$.

Be careful while handling the following edge cases: $W_{\mathsf{min}} > G_{\mathsf{min}}$, $W_{\mathsf{max}} < G_{\mathsf{max}}$, there are no Water types in the range $[l, r]$, there are no Grass types in the range $[l, r]$.

2 Fertilizer

Author: Paras Kasmalkar

Preparation: Shreyan Ray, Paras Kasmalkar, Soumyaditya Choudhuri

Abridged statement: There are F fields, numbered from 1 to F. There are N trips, where i-th trip is a range of fields is from L_i to R_i, where $1 \leq L_i \leq R_i \leq F$. You are given Q queries, where the j-th query consists of integers X_j and Y_j. For each query, output the number of fields contained in the union of trips $X_j, X_j + 1 \ldots Y_j$.

Constraints:

- $1 \leq F \leq 10^6$.
- $1 \leq N \leq 5 \cdot 10^5$.
- $1 \leq L_i \leq R_i \leq F$ for all $1 \leq i \leq N$.
- $1 \leq Q \leq 10^6$.
- $1 \leq X_j \leq Y_j \leq Q$ for all $1 \leq j \leq Q$.

Subtasks:

- **Subtask 1** (4 points) $F, Q, N \leq 200$.
- **Subtask 2** (5 points) $F, Q, N \leq 2000$. Further, $X_j = 1$ for all $1 \leq j \leq Q$.
- **Subtask 3** (8 points) $F, Q, N \leq 2000$.
- **Subtask 4** (5 points) $N \leq 10^5$. Further, $R_i < L_{i+1}$ for all $1 \leq i < N$.
- **Subtask 5** (8 points) $N \leq 10^5$. Further, $L_i < L_{i+1}$ and $R_i < R_{i+1}$ for all $1 \leq i < N$.
- **Subtask 6** (10 points) $N, Q \leq 2 \cdot 10^4$.
- **Subtask 7** (8 points) $N, Q \leq 5 \cdot 10^4$. Further, $L_i = R_i$ for all $1 \leq i \leq N$.
- **Subtask 8** (15 points) $N \leq 10^5$. Further, $X_j \leq X_{j+1}$ and $Y_j \leq Y_{j+1}$ for all $1 \leq j < Q$.
- **Subtask 9** (17 points) $N \leq 10^5$.
- **Subtask 10** (19 points) No additional constraints.

2.1 Subtask 1: (4 points) $F, Q, N \leq 200$

For each query, we can iterate through the $O(N)$ trips performed, and for each trip, we can iterate through the $O(F)$ fields covered. For each query, we maintain a counter array for all the fields, which starts at 0. When we simulate each trip, we increment the counter of all the covered fields by 1, and then the answer to the query is the total number of non-zero indices in the counter array.

The time complexity is $O(Q \cdot F \cdot N)$.

2.2 Subtask 2: (5 points) $F, Q, N \leq 2000$. Further, $X_j = 1$ for all $1 \leq j \leq Q$.

In this subtask, all the queries involve a prefix of the trips, so we can incrementally add trips from left to right and for each trip, count the number of newly added fields in $O(F)$, again using the above counter approach.

The time complexity is $O(N \cdot F + Q)$.

2.3 Subtask 3: (8 points) $F, Q, N \leq 2000$

Here, we need to improve the solution to subtask 1. Instead of maintaining a counter array, we maintain a set of all the fields that have not been covered by at least one trip. When we run a query, for each trip we can use the lower bound function to repeatedly remove the first uncovered field at or after L_i, provided that it is not after R_i.

The time complexity is $O(Q \cdot (N + F) \log F)$.

2.4 Subtask 4: (5 points) $N \leq 10^5$. Further, $R_i < L_{i+1}$ for all $1 \leq i < N$.

In this subtask, no two trips share any fields. So, the answer to each query is the sum of numbers of fields covered by each trip. We can compute this efficiently using a prefix sum array.

The time complexity is $O(N + Q)$.

2.5 Subtask 5: (8 points) $N \leq 10^5$. Further, $L_i < L_{i+1}$ and $R_i < R_{i+1}$ for all $1 \leq i < N$.

In this subtask, the trips follow an important property. Each field, if it is covered by any trip, will be covered by an interval of trips. That is, for each field z which is covered at least once, there will be some i_1, i_2, such that the trips $i_1, i_1 + 1, \ldots i_2$ are exactly the set of trips which cover field z.

This means that we simply need to count the number of fields which are covered at least once by any trip and lie in the range $L_{X_j}, L_{X_j} + 1, \ldots R_{Y_j}$. This can again be counted using prefix sums.

The time complexity is $O(F + N + Q)$.

2.6 Subtask 6: (10 points) $N, Q \leq 2 \cdot 10^4$

Here, we need to answer each query in $O(N)$ without any log factor. We can sort all the trips once at the beginning of the program in the order of R_i.

Then, for each query, we can iterate through this entire sorted list and disregard any queries which do not belong to the query range $X_j, X_j + 1, \ldots Y_j$.

As we iterate through the trips, we can maintain the answer as follows. Let r be the largest R value we have encountered among any trips under consideration. Then, if we encounter some trip i, there are two cases. If $r < L_i$, then the trip i and any trips following it will not share any fields with the trips prior to trip i, so we can simply add the total number of fields of trip i. Otherwise, if $L_i \leq r$, only the fields $r + 1, r + 2, \ldots R_i$ will be newly added, so we can add $R_i - r$ to the answer.

The time complexity is $O(N \log N + NQ)$.

2.7 Subtask 7: (8 points) $N, Q \leq 5 \cdot 10^4$. Further, $L_i = R_i$ for all $1 \leq i \leq N$.

We can use Mo's algorithm to answer all the queries efficiently. As we move through the queries, we need to maintain a counter array for the number of trips which cover each field.

The time complexity is $O((N + Q)\sqrt{N} + F)$.

2.8 Subtask 8: (15 points) $N \leq 10^5$. Further, $X_j \leq X_{j+1}$ and $Y_j \leq Y_{j+1}$ for all $1 \leq j < Q$.

In this subtask, all query ranges have both endpoints in non-decreasing order. Therefore, if we can design a data structure to answer the three following queries, we can solve the problem with $O(N)$ operations of this data structure:

- Add 1 to an interval.
- Subtract 1 from an interval.
- Count the number of non-zero entries in the entire array.

We can add 1 to an interval whenever our right pointer Y_j advances to a new range, and subtract 1 from an interval whenever our left pointer X_j leaves an existing range. To answer the query at a particular time, we can just run the count query.

While the current data structure is nontrivial to design, an important step to note is that only intervals that are already added to using some type 1 query will be subtracted from during a type 2 query. In other words, the values in our data structure should never be below 0, making 0 the minimum possible value that the data structure can take on at any point in time.

This allows us to change the third type of query into a query of the type: find the minimum element in the array, and how many times it occurs. Then there will be two cases:

- If the minimum element is 0 and it occurs c times, the answer is $N - c$ since there are $N - c$ nonzero elements which must necessarily be greater than 0, since elements cannot end up less than 0.
- If the minimum element is greater than 0, then all N elements are nonzero and should be included in the answer.

Note that the minimum element cannot be less than 0.

Therefore the data structure we need is a range add (note that range subtract is just a case of range add), range minimum, and range minimum count data structure. This can be done using a segment tree with lazy propagation. Each query to this data structure can be handled in $O(\log F)$ time, for a total time complexity of $O(N + F + Q \log F)$ for this approach, which is sufficient to pass this subtask.

2.9 Subtask 9: (17 points) $N \leq 10^5$.

This subtask can be done using the divide and conquer technique. Specifically, we have a recursive function `solve(x, y, queries)` where `x` and `y` represent a range of trips and `queries` is an array of queries to be answered. We call `solve(1, N, {1, 2, ... Q})`.

Within a run of `solve(x, y, queries)`, let $m = \frac{x+y}{2}$. All queries with $Y_j \leq m$ or $X_j \geq m+1$ can be put into `queries_left` or `queries_right` respectively, and solved with recursive calls to `solve(x, m, queries_left)` and `solve(m+1, r, queries_right)`.

This leaves the set of queries that contain the indices m and $m + 1$.

The current call to `solve` only involves the trips in the range `x...y`. We can use coordinate compression to reduce the number of fields under consideration to $O(y - x)$.

For each compressed field `f`, we need to find the first trip equal to or to the right of $m + 1$ that covers this field (let this be called `first_right[f]`) and the first trip equal to or to the left of m that covers this field (let this be called `first_left[f]`). One efficient way to do this is to maintain a set of all the fields, iterate through the trips in increasing or decreasing order, and for each trip extract from the set all fields that are covered by that trip.

Now, a compressed field `f` should be counted by a query j only if `first_left[f]` $< X_j$ and `first_right[f]` $> Y_j$. We can answer all the queries offline using a Fenwick tree and the sweep line technique. If the number of trips currently under consideration is n and the number of queries currently under consideration is q, the time complexity for this single call of `solve` is $O((n + q) \log n)$. The sum of n over all calls of `solve` is $O(N \log N)$ and the sum of q over all calls of `solve` is Q.

The overall time complexity is $O(N \log^2 N + Q \log N)$.

2.10 Subtask 10: (19 points) No additional constraints.

2.10.1 Solution 1

We use the following approach: we iterate through all the fields, and attempt to efficiently increment the answer of all queries for which the current field should be counted.

As we iterate through all the fields $z = 1, 2, \ldots F$, we maintain a set of trips which contain the current field z. That is, initially this set is empty. Whenever we transition from a field z to $z+1$, we add to this set all the trips i with $L_i = z+1$ and remove all the trips i with $R_i = z$.

Now, consider all the queries which will cover at least one trip in this set. Let us think of the queries as if they are points in a 2-D plane. That is, each query j is the point (X_j, Y_j). For any set of trips, the set of queries that contain at least one of them will belong to a region that resembles a staircase, as shown in Figure 1. The diagonal line represents $X = Y$, and each trip z is represented as a purple point (z, z).

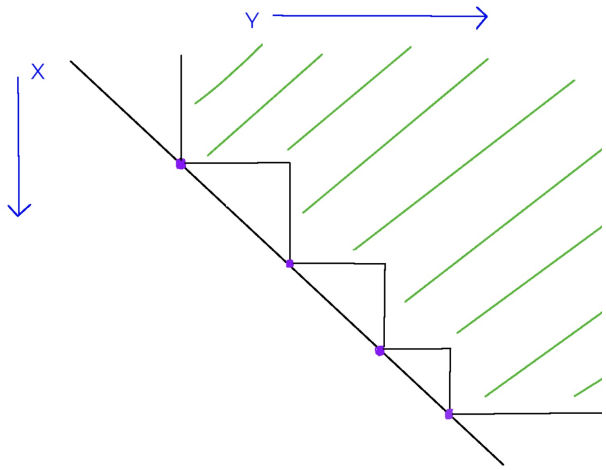

Figure 1: All queries in the shaded region should be incremented by the current field.

We need to perform the following operations: insert a trip, erase a trip, and efficiently increment all the queries which belong to the current staircase region. To maintain the set of trips, we can use the C++ `std::set`. When we add or remove a trip, a rectangular region will be added to or removed from the staircase, as shown in Figure 2 and Figure 3.

We do not have the time to increment each query one by one, for each field. Instead, we look at how queries are affected only when trips are added or removed. Whenever we start looking at a field z and insert rectangular regions representing trips with $L_i = z$, we can perform a 2-D range addition of $-(z-1)$ over each such region. Similarly, whenever we are about to move from z to $z+1$ and erase rectangular regions representing trips with $R_i = z$, we perform a 2-D range addition of $+z$ over each such region.

Now, to answer any query j, we can simply look at the total amount added to the point

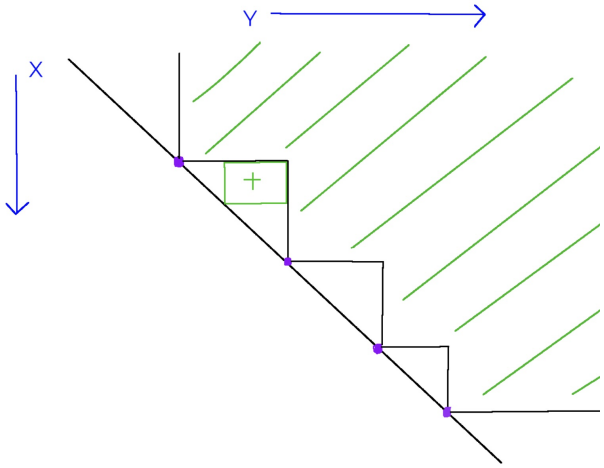

Figure 2: Inserting a trip

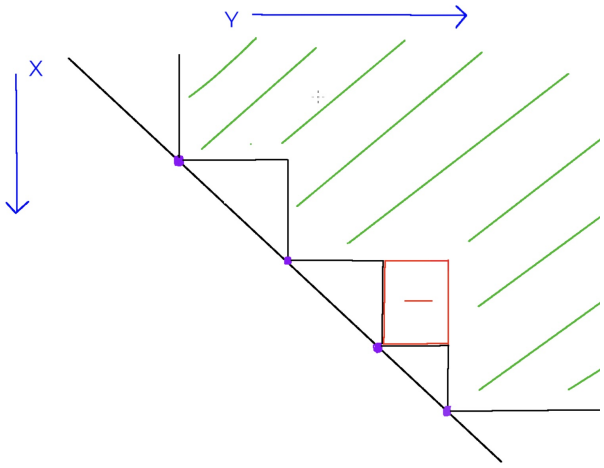

Figure 3: Erasing a trip

(X_j, Y_j). To see why this works, observe that the set of fields which should be counted by query j can be divided into a set of intervals (let's say of some size c_j), $[l_1, r_1], [l_2, r_2], \ldots [l_{c_j}, r_{c_j}]$. Each interval represents a contiguous period of time when the staircase contained (X_j, Y_j). The answer to this query is then $(r_1 - l_1 + 1) + (r_2 - l_2 + 2) + \ldots + (r_{c_j} - l_{c_j} + 1)$.

For each interval k, the above algorithm adds $-(l_k - 1)$ over an area containing (X_j, Y_j) when the staircase starts to cover query j, and adds r_k over an area containing (X_j, Y_j) when the staircase stops covering query j, so in total the amount added to the point (X_j, Y_j) is the total number of fields which query j should count.

We can perform these 2-D range addition and point query operations offline using a Fenwick tree and the sweep-line technique.

The time complexity is $O((N + Q) \log N + F)$.

2.10.2 Solution 2

Another approach to this problem involves answering queries offline, in increasing order of Y_j.

Consider that we have a data structure that maintains an array Z over the indices $1, 2, \ldots F$ and can perform the following operations efficiently:

- Given some range bounds l and r and a value x, set the Z value at each index in the range $l, l+1, \ldots r$ to x
- Given some value x (which is up to F), count the number of indices i such that $Z_i \geq x$.

Initially, this data structure has the value -1 at all indices. We can iterate through all the trips in the order $1, 2, \ldots N$. Whenever we reach some index i, we set all values in the range $L_i, L_i + 1, \ldots R_i$ to i. Notice that now, for any f the value Z_f represents the latest encountered trip which covers the field f, so a query (X_j, Y_j) will cover f only if $X_j \leq Z_f$, so to answer a query we simply need to find the number of values in the array more than or equal to X_j, which can be done by the second count operation of this data structure.

We now describe how to implement this data structure efficiently. We can divide the array into contiguous homogenous intervals, and maintain it as a set of pairs of left end and value. (Another way to implement this is to use a map.) Simultaneously, we maintain a frequency array of the values in a Fenwick tree to answer the count queries in $O(\log F)$.

Whenever we perform a range set operation, some intervals in the set will be erased completely, up to two intervals (containing the endpoints of the range) will be partially adjusted, and one new interval will be inserted. At each stage, only one new interval is inserted and each interval that is erased completely must have been inserted at some prior stage, so the total number of removals is $O(N)$.

The time complexity is $O((N + Q) \log F)$.

3 Trees

Author: Paras Kasmalkar

Preparation: Aryan Maskara, Vishesh Saraswat, Paras Kasmalkar, Shreyan Ray, Soumyaditya Choudhuri

Abridged Statement: There are two undirected edge-weighted trees, tree X and tree Y, both with N nodes numbered $1 \dots N$. The i-th edge in tree X is between nodes A_i and B_i with weight C_i. The j-th edge in tree Y is between nodes U_j and V_j with weight W_j. $\text{cost}_T(p, q)$ denotes the largest edge weight occurring on the unique path between p and q in tree T. Find the number of pairs of nodes (p, q) with $1 \le p < q \le N$ and $\text{cost}_X(p, q) \le \text{cost}_Y(p, q)$.

Constraints:

- $1 \le N \le 10^5$.
- $1 \le A_i, B_i \le N$, $A_i \ne B_i$, $1 \le C_i \le 10^9$ for all $1 \le i \le N - 1$.
- $1 \le U_j, V_j \le N$, $U_j \ne V_j$, $1 \le W_j \le 10^9$ for all $1 \le j \le N - 1$.

Subtasks:

(The *Lines* constraint means that $(A_i, B_i) = (U_i, V_i) = (i, i + 1)$.)

- **Subtask 1** (4 points) $N \le 200$, *Lines*, $C_1 \le C_2 \le \ldots \le C_{N-1}$, $W_1 \le W_2 \le \ldots \le W_{N-1}$.
- **Subtask 2** (5 points) $N \le 200$.
- **Subtask 3** (8 points) $N \le 2000$.
- **Subtask 4** (9 points) *Lines*, $C_1 \le C_2 \le \ldots \le C_{N-1}$, $W_1 \le W_2 \le \ldots \le W_{N-1}$.
- **Subtask 5** (15 points) *Lines*
- **Subtask 6** (10 points) $(A_i, B_i) = (U_i, V_i)$, $C_1 \le C_2 \le \ldots \le C_{N-1}$, all W_j are equal.
- **Subtask 7** (14 points) $(A_i, B_i) = (U_i, V_i)$, $C_1 \le C_2 \le \ldots \le C_{N-1}$, $W_1 \le W_2 \le \ldots \le W_{N-1}$.
- **Subtask 8** (35 points) No additional constraints.

3.1 Subtask 1: (4 points) $N \le 200$, Lines, $C_1 \le C_2 \le \ldots \le C_{N-1}$, $W_1 \le W_2 \le \ldots \le W_{N-1}$

As both trees are lines with nodes in the order $1, 2, ...N$, for any (p, q) with $1 \le p < q \le N$ we have $\text{cost}_X(p, q) = C_{q-1}$ and $\text{cost}_Y(p, q) = W_{q-1}$. So, we can check all possible pairs (p, q) and count the number of pairs which satisfy the condition.

The time complexity is $O(N^2)$.

3.2 Subtask 2: (5 points) $N \leq 200$

In this subtask, for each tree we can find the maximum weight between every pair of nodes using the Floyd-Warshall algorithm, and again count the number of pairs which satisfy the condition.

The time complexity is $O(N^3)$.

3.3 Subtask 3: (8 points) $N \leq 2000$

We can improve the time complexity by running depth first search or breadth first search, once from each node, to find the maximum weight between every pair of nodes.

The time complexity is $O(N^2)$.

3.4 Subtask 4: (9 points) Lines, $C_1 \leq C_2 \leq \ldots \leq C_{N-1}$, $W_1 \leq W_2 \leq \ldots \leq W_{N-1}$.

In the solution of subtask 1, notice that if we fix the number q, for any number p such that $1 \leq p < q$, the value of $\text{cost}_X(p, q)$ remains constant at C_{q-1} and the value of $\text{cost}_Y(p, q)$ remains constant at W_{q-1}.

This means that for each value of q, either all of the possibilities of (p, q) will satisfy the condition or none of them will satisfy the condition. If the condition is satisfied, we can update the answer by $q - 1$, for the $q - 1$ different possibilities of p.

The time complexity is $O(N)$.

3.5 Subtask 5: (15 points) Lines

To check whether a pair (p, q) satisfies the condition, we only need information about the maximum values of C_i and W_i over the interval $p \leq i \leq q - 1$.

Observe that for a given edge i, if $C_i \leq W_i$, then the value C_i is essentially obsolete because in any interval that contains this edge, the maximum edge weight will be at least W_i. Similarly, if $C_i > W_i$, the value W_i is obsolete. Let $Z_i = \max(C_i, W_i)$, and let the **type** of edge i be Y if $C_i \leq W_i$, and X if $C_i > W_i$.

Note that in the case where $C_i = W_i$, it is important that the edge is of type Y. For any pair of nodes (p, q), if $\text{cost}_X(p, q) = \text{cost}_Y(p, q)$, the pair must be counted in the answer. So, we can break the tie between C_i and W_i by considering C_i to be smaller.

We can iterate through all possible values of p from $N - 1$ to 1 in decreasing order. As we do this, we will maintain information about how the maximum values change as we increase q from $p + 1$ to N. We can use a stack to store the list of all visited indices i for which Z_i is greater than all Z_j with $p \leq j < i$. (Again, we break ties between equal C and W values by

considering the C value to be smaller.)

We can maintain such a stack as follows. Initially, let the stack be empty. Then, as we iterate through all i (such that $1 \leq i \leq N-1$) in decreasing order, while the stack is not empty and the top edge in the stack is smaller than the i-th edge, we pop from the stack. Then, we push the i-th edge to the top of the stack.

At any time, for some number q, (p, q) satisfies the original condition only if the largest weight edge in the stack to the left of q is of type Y. By keeping track of the sum of number of nodes between every adjacent pair of edges in the stack where the left edge is of type Y, we can find the number of valid q. To get the final answer, we need to add up the number of valid q over all the steps of this algorithm.

The time complexity is $O(N)$.

3.6 Subtask 6: (10 points) $(A_i, B_i) = (U_i, V_i)$, $C_1 \leq C_2 \leq \ldots \leq C_{N-1}$, all W_j are equal.

In this subtask, $\text{cost}_Y(p, q)$ is equal to W_1 for all pairs (p, q). So, we only need to count the number of (p, q) with $\text{cost}_X(p, q) \leq W_1$. We can do this by computing the set of connected components in the subgraph of tree X consisting of only those edges i with $C_i \leq W_1$.

Within a connected component of size s, all pairs of nodes satisfy the condition and there are $\binom{s}{2} = \frac{s(s-1)}{2}$ pairs of nodes. We can add up this quantity over all components to find the answer.

The time complexity is $O(N)$.

3.7 Subtask 7: (14 points) $(A_i, B_i) = (U_i, V_i)$, $C_1 \leq C_2 \leq \ldots \leq C_{N-1}$, $W_1 \leq W_2 \leq \ldots \leq W_{N-1}$.

In this subtask, the two trees have the same $N-1$ edges, and crucially, the edge weights are in the same relative order in both trees.

In tree X, imagine that there are initially no edges, and we iterate through the edges in increasing order of edge weight and add them to the tree one by one. We can observe that for some pair of nodes (p, q), the largest-weighted edge between them in the original tree is the edge which when added to the tree creates a path between p and q for the first time. If the newly added edge e connects two different connected components c_1 and c_2, the number of pairs (p, q) for which $\text{cost}_X(p, q)$ is defined by edge e is $\text{size}(c_1) \cdot \text{size}(c_2)$. Of course, such a property also holds in tree Y.

Since both trees have the same $N-1$ edges, the two trees will have the same set of connected components at every stage of the above algorithm (where stage is defined by the number of

edges added to that particular tree).

Consider that the edges in both trees are interleaved together and sorted by edge weight (while breaking ties in such a way that edges from tree X occur before edges from tree Y) and we add these edges to the trees in this order. If two components c_1 and c_2 are merged in tree X before they are merged in tree Y, we have identified size$(c_1) \cdot$ size(c_2) pairs of nodes which satisfy the condition, so we can add $c_1 \cdot c_2$ to the answer. We can find such pairs of nodes whenever we add an edge from tree X but the corresponding edge from tree Y has not been added.

To perform this quickly, we can efficiently add edges to a tree and maintain information about its connectivity using the small-to-large technique. We can maintain a list of nodes for every connected component. At the start, each node belongs to its own component. When two components are merged, we need to transfer all the nodes from the list of one component to the list of the other component. We always move nodes from the smaller list to the bigger list. If we do this, whenever a node is moved to another list, the size of the list it belongs to at least doubles, so each node will be moved at most $\lfloor \log_2 N \rfloor$ times.

The data structure which uses this principle to maintain connectivity about a graph is called the disjoint set union (DSU) or union find data structure.

The time complexity is $O(N \log N)$.

3.8 Subtask 8: (35 points) No additional constraints.

We use a similar approach to the previous subtask: we interleave the edges and sort them by weight, and add them to the trees in this order. Whenever an edge of tree Y is added and connects some two components c_1 and c_2, all the pairs of nodes (p, q) which satisfy the condition in the problem statement have been merged in tree X already. We need to find the number of such pairs at every stage to compute the answer.

We use the small-to-large technique from the previous subtask but maintain some more information. For every component c of tree Y, we group the nodes in it by their present component ID in tree X. We can maintain a set of pairs of component ID in tree X and number of occurrences. We only include those components of X with at least one occurrence, so this set of pairs has size less than or equal to the number of nodes in c.

Consider that we are merging two components c_1 and c_2 in tree Y, and assume without loss of generality that size$(c_1) \leq$ size(c_2). We can iterate through all the pairs in the set of c_1. If we find some pair whose component ID is also present in the set of c_2, we can multiply the number of occurrences in these pairs and add this quantity to the final answer. When we move all the pairs from the set of c_1 to the set of c_2, we need to merge all such pairs with the same tree X component ID by adding their frequencies together and leaving only a single pair in the set.

Whenever two components in tree X are merged, we move all the nodes from the list of the smaller component to the list of the bigger component. As we do this, we can update the component ID of all the moved nodes, which will cause some stored pairs in the sets of components of tree Y to be incorrect. If x nodes are moved to another list in tree X, some $O(x)$ pairs of component ID and frequency will be incorrect, so we can fix them one by one. As this happens, some pairs in the same component of tree Y may have the same tree X component ID, so we again need to merge all such pairs.

Because of the small-to-large principle, we only move elements to new sets $O(N \log N)$ times, and if we use a balanced binary search tree such as the C++ `std::set` or `std::map`, each move operation can be done in $O(\log N)$.

The time complexity is $O(N \log^2 N)$.

4 Contest Report

The contest was scheduled to begin at 2:00 pm IST, but was delayed due to minor technical difficulties to 2:10 pm IST. The contest began successfully at 2:10 pm IST.

There were two notable issues with the problems during in the contest:

- Sample testcases for Problem 3, "Trees", were incorrectly uploaded to Problem 2, "Fertilizer". This was corrected 21 minutes into the contest, at 2:31 pm IST.

  This issue did not affect any solutions submitted to the platform, or any solutions tested locally using the samples in the problem statement. The only affected feature was the "Run Code" feature on the editor on the platform.

- Sample testcase 1 for Problem 1, "Monsters", was either missing or failed to render on the contest platform. This was corrected 25 minutes into the contest, at 2:35 pm IST.

  This issue did not affect any solutions submitted to the platform, or any solutions tested locally using the samples in the problem statement. The only affected feature was the "Run Code" feature on the editor on the platform.

Since both issues did not affect any contest submissions or any local testing, there was no contest extension made in either case.

There was also a minor issue:

- In the statement of Problem 1, "Monsters", in some instances "operation" was inadvertently used instead of "thought experiment". No clarifications regarding this were raised during the contest, and this issue wasn't detected until after the end of the contest. The problem statement has since been corrected on Codedrills.

The scientific committee will work to avoid such errors in the future.

5 Scientific Committee

The problem setting committee consisted of, in alphabetical order:

- Aryan Maskara, International Institute of Information Technology, Hyderabad
- Jishnu Roychoudhury, Princeton University
- Paras Kasmalkar, University of Wisconsin-Madison
- Shreyan Ray, Indian Institute of Technology, Kharagpur
- Soumyaditya Choudhuri, Carnegie Mellon University
- Vishesh Saraswat, International Institute of Information Technology, Hyderabad

6 Cutoffs

6.1 Medals

Medal	Cutoff	Medalists
Gold	156	10
Silver	128	20
Bronze	78	30

6.2 Qualification to IOITC 2024

Grade	Male Cutoff	Female Cutoff
12	141	71
11	141	71
10	123	71
9 and below	113	71

Female students with a score of 53 or above are eligible to participate in the EGOI Team Selection Tests, which will be conducted prior to the IOI Training Camp. The four members who are selected for the EGOI team will also qualify for the IOI Training Camp, in case they are not already included in the cutoffs above.

7 Statistics and Anonymized Ranking

A total of 191 contestants participated in this contest, obtaining a total of 93 distinct scores out of 300 possible points. This section features the breakdown of results by problem, subtask, and over the whole contest.

7.0 Overall Statistics

- Number of participants: 191
- Minimum score: 0
- Maximum score: 300
- Number of participants with positive score: 182
- Number of participants with full score: 1
- Mean score: 60.05
- Median score: 38
- Number of distinct scores: 93

7.1 Monsters: Problem Statistics

- Minimum score: 0
- Maximum score: 100
- Number of participants with positive score: 163
- Number of participants with full score: 7
- Mean score: 31.58
- Median score: 16
- Number of distinct scores: 16

The following table summarizes the number of participants who solved each subtask.

Subtask	Solves
1	163
2	134
3	109
4	74
5	58
6	58
7	57
8	51
9	47
10	69
11	7

7.2 Fertilizer: Problem Statistics

- Minimum score: 0
- Maximum score: 100
- Number of participants with positive score: 151
- Number of participants with full score: 2
- Mean score: 18.79
- Median score: 18
- Number of distinct scores: 21

The following table summarizes the number of participants who solved each subtask.

Subtask	Solves
1	144
2	133
3	126
4	71
5	40
6	10
7	26
8	9
9	3
10	2

7.3 Trees: Problem Statistics

- Minimum score: 0
- Maximum score: 100
- Number of participants with positive score: 92
- Number of participants with full score: 1
- Mean score: 9.68
- Median score: 0
- Number of distinct scores: 15

The following table summarizes the number of participants who solved each subtask.

Subtask	Solves
1	92
2	44
3	34
4	61
5	3
6	22
7	10
8	1

7.4 Anonymized Ranking

Rank	Total Score	Monsters	Fertilizer	Trees
1	300	100	100	100
2	265	100	100	65
3	229	100	64	65
4	198	84	64	50
5	181	100	31	50
6	173	84	39	50
7	168	68	64	36
8	165	58	81	26
9	159	84	39	36
10	156	84	49	23
11	153	84	46	23
12-13	149	84	39	26
12-13	149	84	23	42
14	148	100	31	17
15-16	147	58	39	50
15-16	147	58	39	50
17	146	100	23	23
18	144	78	39	27
19-24	141	84	31	26
19-24	141	84	31	26
19-24	141	84	31	26
19-24	141	84	31	26
19-24	141	84	31	26
19-24	141	84	31	26
25	133	84	31	18
26	130	84	23	23
27	129	48	31	50
28-30	128	84	31	13
28-30	128	84	31	13
28-30	128	84	31	13
31-34	127	78	23	26
31-34	127	84	39	4
31-34	127	68	23	36
31-34	127	84	39	4
35	123	78	31	14
36	120	84	18	18
37	119	84	9	26
38	117	58	23	36
39	115	84	18	13
40	113	78	18	17
41	111	84	18	9

42	110	78	28	4
43	109	78	31	0
44	107	84	23	0
45	106	58	31	17
46	105	100	5	0
47-48	102	84	18	0
47-48	102	84	18	0
49	101	78	23	0
50	98	58	31	9
51-52	96	78	18	0
51-52	96	78	18	0
53	90	22	41	27
54	84	84	0	0
55	83	78	5	0
56	81	16	39	26
57	80	62	18	0
58-60	78	78	0	0
58-60	78	21	31	26
58-60	78	78	0	0
61-62	71	27	18	26
61-62	71	27	31	13
63	68	27	23	18
64	66	62	4	0
65-67	63	27	23	13
65-67	63	27	23	13
65-67	63	27	23	13
68	61	15	23	23
69	58	22	23	13
70-71	57	22	31	4
70-71	57	9	31	17
72-73	56	0	43	13
72-73	56	16	31	9
74-75	53	27	13	13
74-75	53	22	31	0
76-78	52	16	23	13
76-78	52	21	18	13
76-78	52	21	31	0
79-80	51	16	18	17
79-80	51	15	23	13
81	50	22	15	13
82-83	48	21	23	4
82-83	48	4	31	13
84	46	15	18	13
85	45	4	28	13
86	42	15	18	9

87-90	40	22	18	0
87-90	40	9	18	13
87-90	40	22	18	0
87-90	40	9	18	13
91-93	39	21	18	0
91-93	39	9	26	4
91-93	39	21	18	0
94-96	38	15	23	0
94-96	38	16	18	4
94-96	38	16	18	4
97	37	15	18	4
98	36	0	23	13
99	35	22	0	13
100-104	34	16	18	0
100-104	34	16	18	0
100-104	34	16	18	0
100-104	34	16	18	0
100-104	34	16	18	0
105-106	33	15	18	0
105-106	33	11	18	4
107-108	32	9	23	0
107-108	32	22	10	0
109-110	31	0	18	13
109-110	31	0	18	13
111-112	30	0	13	17
111-112	30	4	26	0
113-114	29	16	0	13
113-114	29	16	4	9
115-124	27	9	18	0
115-124	27	9	18	0
115-124	27	9	18	0
115-124	27	9	18	0
115-124	27	9	18	0
115-124	27	9	18	0
115-124	27	9	18	0
115-124	27	9	18	0
115-124	27	9	18	0
115-124	27	9	18	0
115-124	27	9	18	0
125-126	26	0	26	0
125-126	26	4	18	4
127	25	16	9	0
128	24	16	4	4
129	23	0	23	0
130-138	22	4	18	0
130-138	22	4	18	0

130-138	22	4	18	0
130-138	22	4	18	0
130-138	22	4	18	0
130-138	22	4	18	0
130-138	22	4	18	0
130-138	22	4	5	13
130-138	22	4	18	0
139-140	21	21	0	0
139-140	21	21	0	0
141	20	16	0	4
142-148	18	0	18	0
142-148	18	0	18	0
142-148	18	0	18	0
142-148	18	0	18	0
142-148	18	0	18	0
142-148	18	0	18	0
142-148	18	0	18	0
142-148	18	0	18	0
149-150	17	9	4	4
149-150	17	11	6	0
151-157	16	16	0	0
151-157	16	16	0	0
151-157	16	16	0	0
151-157	16	16	0	0
151-157	16	16	0	0
151-157	16	16	0	0
151-157	16	16	0	0
158-160	15	15	0	0
158-160	15	15	0	0
158-160	15	11	4	0
161-162	13	9	4	0
161-162	13	4	0	9
163	12	4	4	4
164	11	11	0	0
165	10	0	10	0
166-169	9	9	0	0
166-169	9	9	0	0
166-169	9	9	0	0
166-169	9	9	0	0
170-171	8	4	4	0
170-171	8	0	4	4
172	6	0	6	0
173-182	4	4	0	0
173-182	4	4	0	0
173-182	4	0	4	0
173-182	4	4	0	0

173-182	4	0	4	0
173-182	4	4	0	0
173-182	4	4	0	0
173-182	4	4	0	0
173-182	4	4	0	0
173-182	4	4	0	0
183-191	0	0	0	0
183-191	0	0	0	0
183-191	0	0	0	0
183-191	0	0	0	0
183-191	0	0	0	0
183-191	0	0	0	0
183-191	0	0	0	0
183-191	0	0	0	0
183-191	0	0	0	0