EGOI 2026 Editorial - Fox Families

Task Author: Tomohisa Hachiya, Yuki Tanaka

In this problem, you are give a graph with nodes representing intervals. Two nodes are connected connected if and only if the corresponding intervals are contained in each other. The task is to track the number of connected components while the graph is extended with new intervals step by step.

Subtask 1 $N \leq 100$

In this subtask, we can construct and maintain the graph explicitly. Whenever a new interval is added, we go through all already existing intervals and check the containment condition. For every containment found, we add a new edge to the graph. After updating the graph this way, we compute the number of connected components using depth first search (DFS) after each update.

Since the number of containments is $\mathcal{O}(N^2)$, the size of the graph is also $\mathcal{O}(N^2)$. Since we run a DFS after each update, the total complexity of this approach is $\mathcal{O}(N^3)$.

Subtask 2 $N \leq 2000$

In this subtask, we can maintain the connected components using a disjoint set union (DSU) data structure, initialized with N separate components. Additionally, we maintain the number of components, initially 0. When adding interval i, we first increment the number of connected components. Then, we go through all already existing intervals and check the containment condition. Whenever two intervals are contained in each other but do not yet belong to the same component, the corresponding components are merged and the number of components is reduced by 1. The total time complexity is $\mathcal{O}(N^2)$ (ignoring the almost-constant factor induced by the DSU).

Subtask 3 $R_i - L_i \leq 2$

The solution for this subtask is similar to Subtask 2: we will also maintain the connected components using a DSU. However, we can not afford checking all other intervals whenever adding a new one.

Instead, the key observation here is that for each interval, there is only a small number of other intervals that could contain it or be contained in it. To find all connected intervals for some $[L, R]$, it suffices to check the intervals with left endpoints $L-1, L-2, L$ and $L+1$. Since the intervals are guaranteed to be disjoint, for each left endpoint, there are at most 2 intervals starting there (the interval with lenght 1 and the interval with length 2).

Thus, we store all already-added intervals by their left endpoint and run the solution of Subtask 2, with the faster containment check that only checks the close intervals after each update.

Runtime: $\mathcal{O}(N)$ (again ignoring the almost-constant DSU factor)

Subtask 4 $L_i < L_(i+1)$

From now on, we will no longer use a DSU data structure, but instead exploit the special structure of the graph.

In this subtask, containment of intervals i and j with $i < j$ reduces to just $R_j \leq R_i$. A newly added interval will thus never contain any previously existing intervals. It will be contained in all the intervals that have a larger or equal right endpoint.

Therefore, it suffices to store the **largest right endpoint** of each component: This is enough to check whether a newly added interval will join this component.

The solution then works as follows: maintain a sorted stack of largest right endpoints (one for each component). Whenever a new interval arrives, check whether it is contained in the component on the top of the stack (by checking $R_{\text{new}} \leq \text{maxR}_{\text{top}}$). If it is contained, pop the component from the stack and continue with the new top of the stack. Note that the stack will always be sorted by right endpoint, thus when the new interval is not contained in the topmost component, it is not contained in any component. After popping all components that are connected to the new node, push the newly merged component to the stack. Then, the size of the stack equals the number of components.

Since every interval will be pushed to the stack once and popped at most once, the total runtime of this approach is $\mathcal{O}(N)$.

Full Solution

For the full solution, we generalize the idea from the previous subtask to two dimensions.

An interval $[L, R]$ is *not* part of a component c if and only if for every interval [l, r] in c, it holds that either $L < l \wedge R < r$ ([L, R] is 'more left' than $[l, r]$), or $L > l \wedge r > 0$ ([L, R] is 'more right' than $[l, r]$). In all other cases, $[l, r]$ and $[L, R]$ are included in each other.

Thus, for every component, it suffices to keep track of four numbers: the *minimum right endpoint* (minR), *maximum right endpoint* (maxR), *minimum left endpoint* (minL) and the *maximum right endpoint* (maxL). Then, a new interval $[L, R]$ is *not* part of a component, if either $L < \text{minL} \wedge R < \text{minR}$ (it is 'more left' than all intervals in the component) or $L > \text{maxL} \wedge R > \text{maxR}$ (it is 'more right' than all intervals in the component). Note that if there are both intervals that are 'more right' and some intervals that are 'more left', the interval will be included

in the component. This can be formally proven by an exhaustive case distinction on the concrete relations between L, R, minL, maxL, minR and maxR.

We maintain a C++ `set` with all components, sorted by their minimal left endpoint. When adding a new interval $[L, R]$, we merge it in two steps: First, we merge it with all components that are connected and have $L \geq \text{minL}$, and then we merge it with all components that are connected and have $L < \text{minL}$.

For the first step, we search in the set for the first component with a minimal left endpoint that is at least L. We check whether this component is connected to $[L, R]$ and if yes, remove the component and repeat. Once we reach the first component that is not connected we stop - at this point the interval is already 'more' left than the checked component, thus there is no need to check other components that are more 'to the right'.

Symmetrically, for the second step, we repeatedly search for the maximal right endpoint that is less than L, do the merge check, and repeat.

At the end, we insert the newly merged component into the set. The number of components is then the size of the set.

In Python, the solution is more involved since there is no primitive `set`. One possibility is to manually implement a binary search tree.