Good intervals (`intervals2`)

Una sequenza di numeri $v = [v_1, v_2, \ldots, v_k]$ (per $k \geq 1$) viene definita buona se v_i è un multiplo di i per tutti gli i da 1 a k. Per esempio, $v = [1, 4, 9, 8]$ è una sequenza buona, mentre $v = [1, 3, 6, 12]$ non lo è, perché $v_2 = 3$ non è un multiplo di 2. Vengono date Q query offline che forniscono due interi l e r (1 based), viene chiesto quante sequenze buone esistono nell'intervallo $[l, r]$.

Soluzione in $\mathcal{O}(N \log N + Q \log N)$

Iniziamo creando un array LES, che conterrà per ogni indice i, l'indice k più grande tale che $[v_i, \ldots, v_k]$ è una sequenza buona.

> **Claim 0.0.1** Proprietà di LES
>
> Se $LES[i] = k$, la sequenza $[v_i, \ldots, v_j]$ non è buona, per ogni $k < j \leq N$.
> Se $LES[i] = k$, la sequenza $[v_i, \ldots, v_j]$ è buona, per ogni $i \leq j \leq k$.
> Se $LES[i] = k$, esistono esattamente $k - i + 1$ sequenze buone che cominciano in i.

Tornando alle query, si può notare che se abbiamo una query del tipo $[l, r]$, allora le sequenze buone che hanno inizio in j, dove $j < l$, non influenzano la risposta. Di conseguenza sarà necessario considerare solo le sequenze che hanno inizio dopo l.

La struttura dati che ci permette di risolvere questo problema in modo efficiente è un **lazy segment tree**.

Creiamo un ciclo da $N - 1$ a 0. Ad ogni iterazione, facciamo un update al segment tree aggiungendo 1 all'intervallo $[i, LES[i]]$. Processiamo poi una ad una le query che hanno $l = i$, facendo una query al segment tree con l'intervallo $[l, r]$.

TODO: perchè funziona.

Complessivamente l'algoritmo esegue N update, ciascuno di complessità $\mathcal{O}(\log N)$. Vengono poi eseguite Q query al segment tree, anch'esse di complessità $\mathcal{O}(\log N)$. La complessità finale dell'algoritmo sarà quindi $\mathcal{O}(N \log N + Q \log N)$

```
#include <bits/stdc++.h>
using namespace std;
#pragma GCC optimize("Ofast")


struct segtree {
  int n;
  vector<int> st, lazy;

  void init(int _n) {
    this->n = _n;
    st.resize(4 * n, 0);
    lazy.resize(4 * n, 0);
  }

  int query(int start, int ending, int l, int r, int node) {
    if (lazy[node] != 0) {
      st[node] += lazy[node] * (ending - start + 1);
      if (start != ending) {
        lazy[2 * node + 1] += lazy[node];
        lazy[2 * node + 2] += lazy[node];
      }
      lazy[node] = 0;
    }

        if (start > r || ending < l) {
      return 0;
    }
```

```
    if (start >= l && ending <= r) {
      return st[node];
    }

    int mid = (start + ending) / 2;

    int q1 = query(start, mid, l, r, 2 * node + 1);
    int q2 = query(mid + 1, ending, l, r, 2 * node + 2);

    return q1 + q2;
  }

  void update(int start, int ending, int node, int l, int r, int value) {

        if (lazy[node] != 0) {
      st[node] += lazy[node] * (ending - start + 1);
      if (start != ending) {
        lazy[2 * node + 1] += lazy[node];
        lazy[2 * node + 2] += lazy[node];
      }
      lazy[node] = 0;
    }
        if (start > r || ending < l) {
      return ;
    }



    if (start >= l && ending <= r) {
      st[node] += value * (ending - start + 1);
      if (start != ending) {
        lazy[2 * node + 1] += value;
        lazy[2 * node + 2] += value;
      }
      return;
    }

    int mid = (start + ending) / 2;
    update(start, mid, 2 * node + 1, l, r, value);

    update(mid + 1, ending, 2 * node + 2, l, r, value);

    st[node] = st[node * 2 + 1] + st[node * 2 + 2];

    return;
  }
  int query(int l, int r) {
    return query(0, n - 1, l, r, 0);
  }

  void update(int l, int r, int x) {
    update(0, n - 1, 0, l, r, x);
  }
};


struct query
{
    int id;
    int l,r;

};
int main()
{
    ios_base::sync_with_stdio(0);
    cin.tie(0);
    cout.tie(0);

    int T;
```

```
    cin >> T;

    while(T--)
    {
        int N;
        cin >> N;
        vector<long long> A(N);
        for(int i = 0; i < N; i++) cin >> A[i];

        vector<int> les(N, 0);
        unordered_set<int> to_check;
        for(int i = 0; i < N; i++)
        {
            vector<int> tbd;

            if(!to_check.empty())
            for(auto k : to_check)
            {
                long long div = i-k+1;

                if(A[i]%div)
                {
                    les[k] = i-1;
                    tbd.emplace_back(k);
                    continue;
                }
            }
            for(auto k : tbd) to_check.erase(k);
            to_check.insert(i);
        }
        for(auto i : to_check) les[i] = N-1;



        int Q;
        cin >> Q;

        vector<int> queryAnswers(Q, 0);
        vector<vector<query>> queries(N);

        for(int i = 0; i < Q; i++)
        {
            query q;
            cin >> q.l >> q.r;
            q.id = i;
            q.l--;
            q.r--;

            queries[q.l].push_back(q);
        }



        segtree tree;
        tree.init(N);
        for(int i = N-1; i >= 0; i--)
        {
            tree.update(i, les[i], 1);
            for(auto q : queries[i])
            {
                queryAnswers[q.id] = tree.query(q.l, q.r);
            }
        }

        for(auto i : queryAnswers) cout << i << "\n";

    }
}
```