Drzewo Przedział-Przedział

Struktura danych, która pozwala na aktualizowanie wartości w tablicy w danym przedziale oraz sprawdzanie wartości tablicy w danym przedziale w czasie $O(\log n)$.

Standardowy problem:

> Dana jest tablica $T[0, n-1]$. Chcemy q razy wykonywać dwa typy operacji:
>
> 1. Ustaw x na przedziale $T[a, b]$. Formalnie: $T[i] := x$, dla $i \in [a, b]$
> 2. Zwróć maksymalną wartość na przedziale $T[a, b]$. Formalnie: $\max\{T[a], T[a+1], \ldots, T[b]\}$
>   Limity:
> $1 \le n, q \le 2 \cdot 10^5$
> $1 \le T[i], x \le 10^9$

Oczywiście te zadanie można rozwiązać naiwnie: Dla każdego zapytania przechodzić pętlą `for` po tablicy T. Niestety takie rozwiązanie działa w złożoności $O(nq)$. Potrzebujemy czegoś szybszego.

Z pomocą przychodzi drzewo przedziałowe z tak zwaną *lazy propagation*.
Lazy propagation polega na tym, aby nie zmieniać czegoś, co jeszcze nie musi być zmienione.
Dla każdego wierzchołka v w drzewie będziemy trzymali dodatkową informację $L[v]$.
$L[v]$ oznacza jakiej wartości nie przekazałem jeszcze synom v. Aby szybko wykonywać operację 1) część wierzchołków w drzewie nie zostanie zaktualizowana. Zaktualizuję tylko te konieczne i zapiszę im w tablicy L informację o tym, że muszą one przekazać tę informacje swoim synom w przyszłości (przy następnym przejściu przez dany wierzchołek).

Użyję do tego funkcji `lazypush`:

```
void lazypush(int v) {
        if (Lazy[v] != 0) {
                Tree[2*v] = Lazy[v];
                Lazy[2*v] = Lazy[v];
                Tree[2*v+1] = Lazy[v];
                Lazy[2*v+1] = Lazy[v];
        }
        Lazy[v] = 0;
}
```

Operację 1) wykonuję funkcją `insert`:

```
void insert(int v, int lw, int rw, int L, int R, int val) {
        if (rw < L || R < lw)
                return;
        else if (L <= lw && rw <= R) {
                Tree[v] = val;
                Lazy[v] = val;
        }
        else {
                lazypush(v);
                int mid = (lw + rw) / 2;
                insert(2*v, lw, mid, L, R, val);
                insert(2*v+1, mid+1, rw, L, R, val);
                Tree[v] = max(Tree[2*v], Tree[2*v+1]);
        }
}
```

Operację 2) funkcją `query`:

```
int query(int v, int lw, int rw, int L, int R) {
        if (rw < L || R < lw)
                return -INF;
        else if (L <= lw && rw <= R)
                return Tree[v];
        else {
                lazypush(v);
                int mid = (lw + rw) / 2;
                int Lson = query(2*v, lw, mid, L, R);
                int Rson = query(2*v+1, mid+1, rw, L, R);
                return max(Lson, Rson);
        }
}
```