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:
> $1 \leq n, q \leq 2 \cdot 10^5$
> $1 \leq T[i], x \leq 10^9$

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 $Lazy[v]$.

- $Lazy[v]$ oznacza jakiej wartości nie przekazałem jeszcze do poddrzewa v.

Aby szybko wykonywać operację 1) zaktualizuję tylko te konieczne (tzn. te których przedziały bazowe stanowią podział $[a, b]$) i zapiszę im w tablicy $Lazy$ informację o tym, że w przyszłości (przy następnym przejściu przez dany wierzchołek) muszą one przekazać tę informacje do swojego poddrzewa i je zaktualizować.

Użyję do tego funkcji `push`:

```
void push(int v) {
        if (Lazy[v] != 0) {
                Tree[2*v] = Lazy[v]; // aktualizuje wartości w synach
                Tree[2*v+1] = Lazy[v];
                Lazy[2*v] = Lazy[v]; // mówię synom, że one też będą
musiały przekazać
                Lazy[2*v+1] = Lazy[v]; // Lazy[v] dalej do swoich synów
        }
        Lazy[v] = 0; // trzeba oczyścić v z wartości Lazy
}
```

Operację 1) wykonuję funkcją `update`:

```
// v - wierzchołek w którym jestem
// [st,ed] - przedział jaki obejmuje wierzchołek v
// [a,b] - przedział który chcemy zmodyfikować
// val - wartość którą ustawiam na przedziale [a,b]
void update(int v, int st, int ed, int a, int b, int val) {
        if (ed < a || b < st) // przedziały rozłączne
                return;
        else if (a <= st && ed <= b) { // [st,ed] zawiera się w [a,b]
                Tree[v] = val;
                Lazy[v] = val;
        }
        else {
                push(v); // muszę zaktualizować synów przed tym jak do
nich pójdę
                int md = (st + ed) / 2;
                update(2*v, st, md, a, b, val);
                update(2*v+1, md+1, ed, a, b, val);
                Tree[v] = max(Tree[2*v], Tree[2*v+1]);
        }
}
```

Którą wywołuję w main'ie jako `update(1, 0, base-1, a, b, x)`.

Operację 2) funkcją `query`:

```
int query(int v, int st, int ed, int a, int b) {
        if (ed < a || b < st)
                return 0; // może być coś innego, np. -1e9+7
        else if (a <= st && ed <= b)
                return Tree[v];
        else {
                push(v);
                int md = (st + ed) / 2;
                int lewy = query(2*v, st, md, a, b);
                int prawy = query(2*v+1, md+1, ed, a, b);
                return max(lewy, prawy);
        }
}
```

I wywołuję ją jako `query(1, 0, base-1, a, b)`.