Szablon drzewa przedziałowego

Twoim zadaniem jest zaimplementowanie szablonu drzewa przedziałowego parametryzowanego dwoma typami: `Value` oraz `Modifier`.

```
template<typename Value, typename Modifier>
class SegmentTree {
public:
    SegmentTree(std::size_t n, Value value = {});
    SegmentTree(const std::vector<Value>& values);
    Value query(std::size_t begin, std::size_t end) const;
    void update(std::size_t begin, std::size_t end, Modifier modifier);
};
```

Typ `Value` służy do reprezentacji elementów ciągu i dostarcza:

- konstruktor domyślny, który tworzy element neutralny;
- operator dodawania `operator+` (agregacja elementów).

Typ `Modifier` służy do reprezentacji modyfikacji, które możemy aplikować do ciągu, i dostarcza:

- konstruktor domyślny, który tworzy neutralny modyfikator;
- operator mnożenia `operator*` (składanie modyfikatorów);
- operator funkcyjny: `Value operator()(Value v)` (działanie na elementach ciągu).

Zakładamy, że operacje na elementach typu `Value` i `Modifier` działają w czasie stałym. Instancja drzewa utrzymuje ciąg elementów $a_0, \ldots, a_{n-1}$ typu `Value`. Klasa dostarczać powinna dwa konstruktory:

- `SegmentTree(std::size_t n, Value value = {})`: inicjalizacja ciągiem $a_0 = \ldots = a_{n-1} =$ `value`. Domyślnie `value` jest elementem neutralnym.
- `SegmentTree(const std::vector<Value>& values)`: inicjalizacja ciągiem elementów `values`.

Konstruktory powinny działać w czasie $\mathcal{O}(n)$. Ponadto, jak przystało na drzewo przedziałowe, klasa powinna dostarczać metody operujące na przedziałach:

- `Value query(std::size_t i, std::size_t j) const`: zwraca wartość $a_i + \ldots + a_{j-1}$;
- `void update(std::size_t i, std::size_t j, Modifier m)`: aktualizuje elementy $a_i, \ldots, a_{j-1}$ wg wzoru $a_k := m(a_k)$.

W obydwu metodach można założyć, że zachodzi $0 \leqslant i \leqslant j \leqslant n$. W przypadku gdy $i = j$, metoda `query` powinna zwracać element neutralny, a metoda `update` nie mieć żadnego efektu. Obydwie metody powinny działać w czasie $\mathcal{O}(\log n)$.

Aby wykonanie zadania było możliwe, typy `Value` i `Modifier` spełniają kilka algebraicznych własności. Niech v_1, v_2, v_3 będą typu `Value`, a v_ε domyślną wartością typu `Value`. Niech m_1, m_2, m_3 będą typu `Modifier`, a m_ε domyślną wartością typu `Modifier`. Zachodzą następujące własności:

1. Element neutralny dodawania wartości: $v_1 + v_\varepsilon = v_\varepsilon + v_1 = v_1$
2. Element neutralny składania modyfikatorów: $m_1 * m_\varepsilon = m_\varepsilon * m_1 = m_1$
3. Łączność dodawania wartości: $v_1 + (v_2 + v_3) = (v_1 + v_2) + v_3$
4. Łączność składania modyfikatorów: $m_1 * (m_2 * m_3) = (m_1 * m_2) * m_3$
5. Neutralny modyfikator nie ma efektu: $m_\varepsilon(v_1) = v_1$
6. Zgodność działania modyfikatora ze składaniem: $(m_1 * m_2)(v_1) = m_1(m_2(v_1))$
7. Rozdzielność działania modyfikatora względem sumowania wartości: $m_1(v_1 + v_2) = m_1(v_1) + m_1(v_2)$

Zwróć uwagę, że przemienność nie musi zachodzić, a aplikacja modyfikatora do elementu neutralnego może dać inny element (tzn. może się zdarzyć, że $m_1(v_\varepsilon) \neq v_\varepsilon$).

Testowanie rozwiązania

Na Satori należy wysłać pojedynczy plik nagłówkowy zawierający implementację szablonu `SegmentTree`. Nadesłany plik zostanie skompilowany razem z programem testującym. Nazwa pliku nie ma znaczenia. Poniżej załączony jest przykładowy program, który implementuje zapytania o maksimum i dodawanie na przedziale. Zwróć uwagę, że w tym przypadku zarówno agregacja jak i modyfikacje są przemienne.

```
#include "solution.hpp" // Nagłówek z Twoim rozwiązaniem.

#include <bits/stdc++.h>

struct MaxValue {
    MaxValue(int v = 0) : value(v) {}

    friend MaxValue operator+(MaxValue l, MaxValue r) {
        return std::max(l.value, r.value);
    }

    friend std::ostream& operator<<(std::ostream& out, MaxValue v) {
        return out << v.value;
    }

    int value;
};

struct PlusModifier {
    PlusModifier(int a = 0) : add(a) {}

    friend PlusModifier operator*(PlusModifier l, PlusModifier r) {
        return l.add + r.add;
    }

    MaxValue operator()(MaxValue v) const {
```

```
        return v.value + add;
    }

    int add;
};

template<class T>
void makeQueries(const T& tree) {
    std::cout << tree.query(3, 3) << ' ';
    std::cout << tree.query(0, 2) << ' ';
    std::cout << tree.query(0, 3) << ' ';
    std::cout << tree.query(6, 10) << ' ';
    std::cout << tree.query(4, 6) << std::endl;
}

int main() {
    SegmentTree<MaxValue, PlusModifier> tree(10);

    makeQueries(tree);

    tree.update(2, 5, 5);
    makeQueries(tree);

    auto treeCopy = tree;

    tree.update(4, 7, 3);
    makeQueries(tree);

    tree.update(4, 6, 10);
    makeQueries(tree);
    makeQueries(treeCopy);
}
```

Uruchomienie programu z poprawną implementacją szablonu powinno wypisać:

```
0 0 0 0 0
0 0 5 0 5
0 0 5 3 8
0 0 5 3 18
0 0 5 0 5
```