Структуры данных. Разреженные таблицы, Дерево Отрезков.

ЛШКН 2020, параллель B
Прохоров Михаил
tg: @Aphanasiy

Проблематика

Посчитайте заданную функцию на отрезке в онлайне много раз

Степени принятия

1) Отторжение

   === ВЫ НАХОДИТЕСЬ ЗДЕСЬ ===

2) Научиться считать для идемпотентных функций без изменений
3) Научиться считать для всех функций с изменением в точке
4) Научиться считать для всех функций с изменением на отрезке
5) Порешать завтра утром контест
6) Депрессия

Разреженные таблицы

Часть 2. Научиться считать для идемпотентных функций без изменений

Идемпотентная функция

Идемпотентная операция - это действие, многократное повторение которого эквивалентно однократному.

Примеры:

1) MIN, MAX => MIN(a, a) = a
2) AND, OR => AND(a, a) = a
3) GCD => GCD(a, a) = a

Антипримеры:

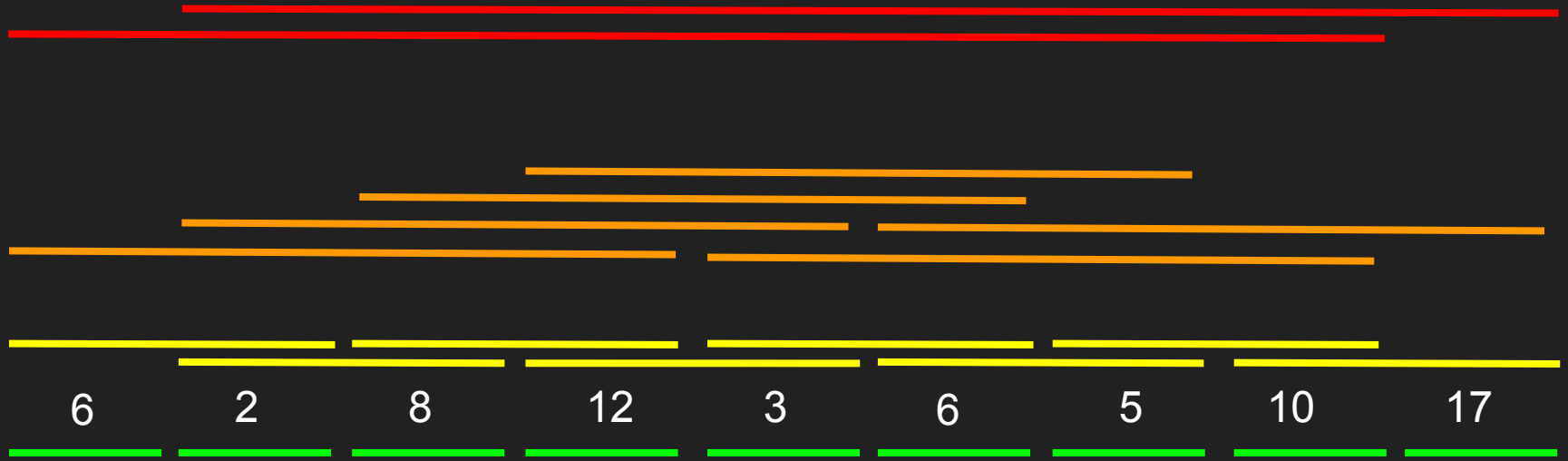
1) SUM => SUM(a, a) = 2 * a != a
2) XOR => XOR(a, a) = a xor a = 0 != a
3) PROD => PROD(a, a) = a^2 != a

Ближе к делу

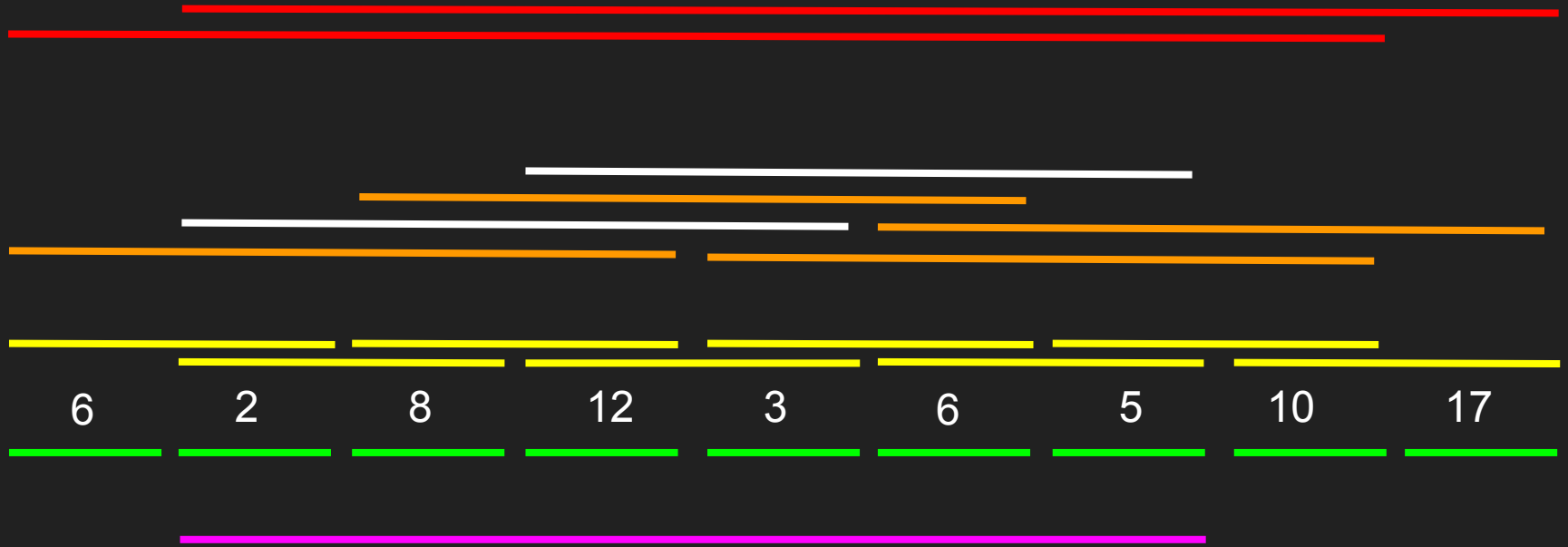
Дан массив чисел.

В онлайне приходят запросы на минимум на отрезке.

Структура разреженной таблицы (инициализация)



Ответ на запрос



Вопросы?

Степени принятия

1) Отторжение
2) Научиться считать для идемпотентных функций без изменений

=== ВЫ НАХОДИТЕСЬ ЗДЕСЬ ===

3) Научиться считать для всех функций с изменением в точке
4) Научиться считать для всех функций с изменением на отрезке
5) Порешать завтра утром контест
6) Депрессия

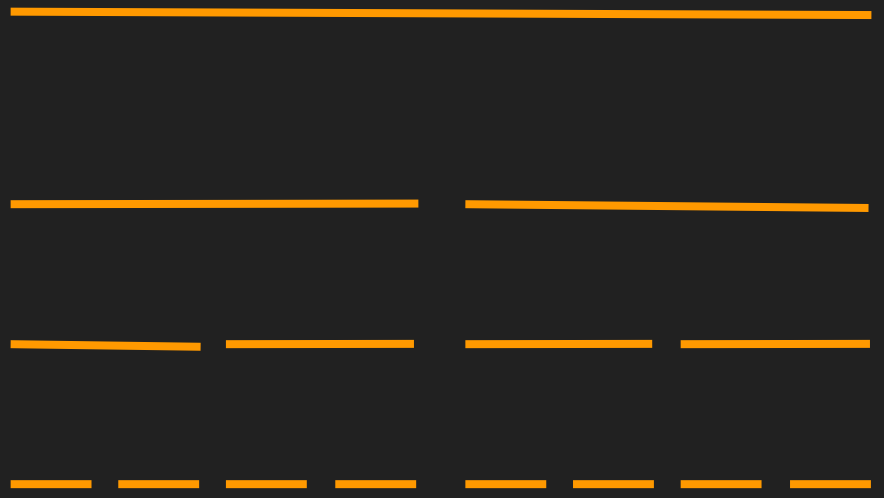
Дерево отрезков

Часть 3. Научиться считать для всех функций с изменением в точке

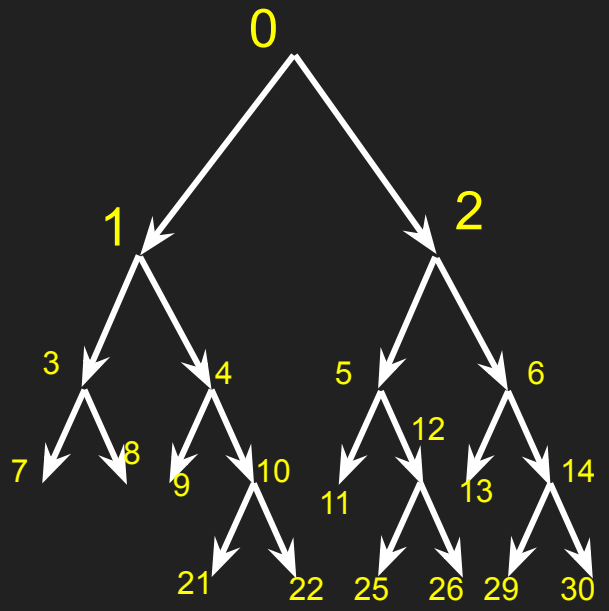
Что умеет ДО (Дерево Отрезков)

1. Запрос на отрезке за O(log n)
2. Запрос на изменение в точке за O(log n)
3. Запрос на изменение на отрезке за O(log n)

Как выглядит ДО



Как хранить ДО?



- Если наша вершина V отвечает за полуинтервал [L, R), то:
- Левый сын имеет номер 2V + 1 и отвечает за полуинтервал [L, (L + R) / 2)
- Правый сын имеет номер 2V + 2 и отвечает за полуинтервал [(L + R) / 2, R)
- Если R - L = 1, то наша вершина - бездетный лист.

Но ты не можешь называть структуру Дерево Отрезков, а писать его на полуинтервалах!

Могу.

Инициализация работает за линию!

$$n + n/2 + n/4 + n/8 + \ldots \sim 2n$$

Сколько линий нужно, чтобы ~~вкрутить лампочку~~ хранить дерево?

Двух точно мало,
Трёх может быть не достаточно.
Больше четырёх точно не понадобится.

Кодинг тайм: Инициализация

```
int t[4 * MAX_V];

void build(const vector<int>& base,
     int v = 0, int L = 0, int R = N) {

    if (R - L == 1) {
        t[v] = base[L];
        return;
    }
    build(base, 2 * v + 1, L, (L + R) / 2);
    build(base, 2 * v + 2, (L + R) / 2, R);
    t[v] = t[2 * v + 1] # t[2 * v + 2];
    return;
}
```

После этих строк я начал программировать ДО быстрее. Дедовский метод...

```
int left_son(int v)
    return 2 * v + 1;

int right_son(int v)
    return 2 * v + 2;

void relax(int v) {
    t[v] = t[2 * v + 1] # t[2 * v + 2]
    // ‘#’ - это считаемая в ДО операция. +, *, и т.д.
}
```

Кодинг тайм: Инициализация

```
int t[4 * MAX_V];

void build(const vector<int>& base,
     int v = 0, int L = 0, int R = N) {

    if (R - L = 1) {
        t[v] = base[L];
        return;
    }
    int MID = L + (R - L) / 2 // защита от переполнения (!)
    build(base, left_son(v), L, M);
    build(base, right_son(v), M, R);
    relax(v);
    return;
}
```

Кодинг тайм: Изменение в точке

```
void change(int val, int pos, int v = 0, int L = 0, int R = N) {
    if (R - L = 1) {
        t[L] = val;
        return
    }
    int MID = L + (R - L) / 2;
    if (pos < MID) { // Не забываем про полуинтервалы (!)
        change(val, pos, left_son(v), L, M);
    } else {
        change(val, pos, right_son(v), M, R);
    }
}
```

Кодинг тайм: Изменение в точке

```
void change(int val, int pos, int v = 0, int L = 0, int R = N) {
    if (R - L = 1) {
        t[L] = val;
        return
    }
    int MID = L + (R - L) / 2;
    if (pos < MID) { // Не забываем про полуинтервалы (!)
        change(val, pos, left_son(v), L, M);
    } else {
        change(val, pos, right_son(v), M, R);
    }
    relax(v); // Не забываем обновлять значение в вершине!!!
}
```

Кодинг тайм: Запрос на отрезке

```
int query(int l, int r, int v = 0, int L = 0, int R = N) {
    if (l <= L && R <= r) { // Подотрезок в запросе
        return t[v];
    } else if (r <= L || R <= l) { // Подотрезок не в запросе
        return NEUTRAL; // Для суммы - 0, для произведения - 1.
    }
    int MID = L + (R - L) / 2;
    return query(l, r, left_son(v), L, M) #
           query(l, r, right_son(v), M, R);
    // А как же релаксация?
    // Мы ничего не меняли, она тут не нужна
}
```

Почему работает быстро?

1) В Дереве отрезков логарифм уровней
2) Если мы обновляемся, то значит, что у нас рекурсия запустится не более логарифма раз
3) Если мы делаем запрос, то на каждом уровне мы задействуем не более четырёх подотрезков. Если в подотрезке есть граница, то подотрезок больше не делится. Это значит, что на каждом уровне поделится не более двух подотрезков (по количеству границ запроса), а значит в итоге мы получим логарифмическую сложность.

Степени принятия

1) Отторжение
2) Научиться считать для идемпотентных функций без изменений
3) Научиться считать для всех функций с изменением в точке

=== ВЫ НАХОДИТЕСЬ ЗДЕСЬ ===

4) Научиться считать для всех функций с изменением на отрезке
5) Порешать завтра утром контест
6) Депрессия

Дерево отрезков с массовыми операциями

Часть 4. Научиться считать для всех функций с изменением на отрезке

Идея

Чисто в теории нам ничего не мешает сделать отложенные операции.

То есть если мы должны применить операцию к отрезку, давайте запомним, что мы её должны сделать в другом массиве. Когда в каком-то запросе надо будет спуститься ниже, делаем push.

Кодинг тайм: Оптимизация релаксаций

```
int t[4 * MAX_V];
int r[4 * MAX_V];

void push(v) {
    t[v] #= r[v];
    r[left_son(v)] #= r[v];
    r[right_son(v)] #= r[v];
}

void relax(int v) {
    t[v] = t[left_son(v)] # t[right_son(v)] # r[v];
}
```

Кодинг тайм: Запрос на отрезке

```
int query(int chval, int v = 0, int L = 0, int R = N) {
    if (???)
    // Блин, как программировать?

    // Ничего не понятно…

    // Давайте на примерах, а то чот жесть...

}
```


Наташа нас не покормила
И тапочки тоже спрятала...
Поэтому мы тебе этот слайд
КОТИЗМ
Выкручивайся, как хочешь...

Грязный хак

Если у нас не хватает памяти, можно инициализировать дерево на ходу.

Это называется Неявное Дерево Отрезков (идейно похоже на Декартово Дерево)

```cpp
#include <iostream>
#include <vector>


using namespace std;


const int V_MAX = 1e6 + 7; // Инициализируем максимум для более удобного дебага
int N;

int t[4 * V_MAX];
int a[4 * V_MAX];


int left_son(int v) {
    return 2 * v + 1;
}

int right_son(int v) {
    return 2 * v + 2;
}

void push(int v, int L, int R) {  // В случае массовых операций, нам надо сделать push.
    t[v] += a[v] * (R - L); // Сначала применяем изменения к себе
    a[left_son(v)] += a[v]; // А потом пушим в левого
    a[right_son(v)] += a[v]; //              и правого сыновей.
    a[v] = 0;                //  Не забываем обнулить счётчик пуша.
    return;
}

void relax(int v, int L, int M, int R) {
    t[v] = t[left_son(v)] +  a[left_son(v)] * (M - L) + // Просчитываем сыновей с их изменениями
           t[right_son(v)] + a[right_son(v)] * (R - M);
}
```

```
void build(const vector<int>& base, int v = 0, int L = 0, int R = N) {
    a[v] = 0;   // Инициализируем счётчики изменений.
    if (R - L == 1) {
        t[v] = base[L];
        return;
    }
    int M = (L + R) / 2;
    build(base, left_son(v), L, M);
    build(base, right_son(v), M, R);
    relax(v, L, M, R); // В релакс передаём больше переменных, чтобы пересчитаться от сыновей.
    return;
}


void add(int l, int r, int h, int v = 0, int L = 0, int R = N) {
    if (l <= L && R <= r) { //  Проверяем, что отрезок полностью лежит в запросе
        a[v] += h; // Лениво модифицируем весь отрезок
        return;
    } else if (r <= L || R <= l) {
        return; // Если не лежит, то просто выходим.
    } // Если пересекается или запрос вложен в отрезок, то делим на два и начинаем заново.
    push(v, L, R);
    int M = (L + R) / 2;
    add(l, r, h, left_son(v), L, M);
    add(l, r, h, right_son(v), M, R);
    relax(v, L, M, R);
}
```

```
int query(int l, int r, int v = 0, int L = 0, int R = N) {
    /*
    cout << "V: " << v << "|" << "[L, R): " << "[" << L << ", " << R << ")" <<
                                "[l, r): " << "[" << l << ", " << r << ")"<< endl;
    //*/ // Многострочный комментарий, который можно раскомментировать, добавив / в начало ;)
    // Здесь мы делали дебажный вывод, когда дебажили
    if (l <= L && R <= r) {  // Проверка, на то, что вершина в запросе
        return t[v] + a[v] * (R - L);  // Возвращаем ответ с отложенными операциями
    }
    if (r <= L || R <= l) { // Проверяем, что вершина не в запросе
        return 0; // Возвращаем нейтральный элемент.
    }                   // Иначе
    push(v, L, R);      //       пропихиваем изменения в сыновей
    int M = (L + R) / 2;
    int ans = query(l, r, left_son(v), L, M) +  // Считаем ответ
              query(l, r, right_son(v), M, R);
    relax(v, L, M, R);  // На всякий случай релаксируемся.
                        // Если всё было сделано хорошо, то помочь не должно, но тем не менее
    return ans;
}

void print_tree(int agg = 0, int v = 0, int L = 0, int R = N) {
    // Это обход дерева для дебага.
    agg += a[v];
    if (R - L == 1) {
        cout << L << ": " << t[v] + agg << endl;
        return;
    }
    int M = (L + R) / 2;
    print_tree(agg, left_son(v), L, M);
    print_tree(agg, right_son(v), M, R);
}
```

```cpp
int main() { // Ну и, соответственно, главная функция.
    cin >> N;
    vector<int> base(N);
    for (auto &i : base) {
        cin >> i;
    }
    build(base);
    int q;
    cin >> q;
    for (int i = 0; i < q; ++i) {
        char c;
        cin >> c;
        if (c == '+') {
            int l, r, h;
            cin >> l >> r >> h;
            add(l - 1, r, h);
        } else if (c == '?') {
            int l, r;
            cin >> l >> r;
            cout << query(l - 1, r) << endl;
        }
        cout << "PRINT TREE:" << endl;
        print_tree();
        cout << " === === ===" << endl;
    }

}
```