Лекция 4.
Битовые операции. Дерево Фенвика

179 Школа, Овчинников Андрей

26 октября 2022 г.

Аннотация

Перед тем, как вы прочитаете то, что написано ниже обращу внимание на то, что в данном конспекте могут быть допущены ошибки и опечатки, если вы находите подобное, то пишите мне в личку или группу в телеграмм с упоминанием (*то есть через* @). Буду рад, если вы поможете довести конспект до хорошего безошибочного состояния.

1 Битовые операции

Все мы знаем о том, что любое число представимо в двоичной системе счисления. Собственно, таким же образом они представляются в памяти компьютера. Более того, большинство языков программирования позволяют напрямую, и самое главное быстро работать с этим представлением. Сегодня мы поговорим о том, как рабоать с типами целых чисел, *такими как: unsigned int, unsigned long long и т.п.*

1.1 О количестве битов в разных типах

Очевидно, что для разных типов используется разное количество битов для хранения. Укажем основные:

- **int** - 32 бита (отдельно стоит помнить про старший знаковый бит и избегать работы с ним)
- **unsigned int** - 32 бита (можете спокойно работать со всеми битами)
- **long long** - 64 бита (отдельно стоит помнить про старший знаковый бит и избегать работы с ним)
- **unsigned long long** - 64 бита (можете спокойно работать со всеми битами)

1.2 О стандартных операциях

Перечислим все стандартные операции, которые можно производить над битами и числами:

1.2.1 Побитовые И, ИЛИ, ИСКЛЮЧАЮЩЕЕ ИЛИ

&, |, ^ - побитовые *и, или, исключающее или* соответственно. Всё это бинарные операци (a & b, a | 10 и т.п., все числа которые вы указываете в коде, естественно записаны вами в десятичной системе и уже компилятор перегоняет их в память компьютера в двоичном виде)

1.2.2 Инвертирование битов числа

~ - инвертирует каждый бит числа (0 заменяет на 1, а 1 на 0). Следует осторожно использовать, так как при воздействии на знаковые типы, так же изменяет знаковый бит. Оператор унарный и пишется слева от числа или переменной, к которой применяется.
Например, $(\ ((unsigned)1)) == 4294967295 - 1$.

1.2.3 Побитовый сдвиг влево и вправо

$<<, >>$ - битовые сдвиги, сдвигает агрумент слева на количество битов указанное справа (слева может быть как переменная, так и число, аналогично справа). Если в при сдвиге в позицию должно придти значение, которого не было (например младший бит у (1 « 1)), то на позицию встаёт значение 0.
Например,

- $(1 << 1) \ == \ 2 \ == \ 10_2$
- $(4 >> 2) \ == \ (100_2 >> 2) \ == \ 1_2 \ == \ 1_{10}$

Так же слежует помнить о том, что при битовых операциях, результат будет представлен в типе левого операнда, то есть $(1 << 31)$ - будет равен INT_MIN, так как 1 - приводится к типу int. а если вы хотите что бы ваш результат не был таковым, то нужно привести 1 (или переменную) к нужному типу $((unsigned)1 << 31), ((longlong)1 << 31), ((unsignedlonglong)1 << 31)$.

1.2.4 О важном аспекте битовых операций

Очень важно помнить, что у большинства битовых операций очень маленький приоритет, как следствие следует прописывать все операции в скобочках (если вы конечно не помните все приоритеты).

2 Дерево Фенвика

Рассмотрим знакомую задачу. Пусть данн массив из n чисел. Требуется научиться считать сумму на отрезке и изменять элемент.

Заметим, что если мы научимся находить сумму на суффиксе массива, то тогда запрос получения суммы можно будет предстваить, как запрос суммы на суффиксе с l минус запрос суммы на суффиксе с r (если запросы на полуинтервалах).

2.1 О дереве Фенвика

Я не смог придумать что-то лучше чем это.

2.2 Код на сумму

```
struct Fenw
{
    int size;
    vector<long long> tree;

    Fenw(int n)
    {
        this->size = n;
        this->tree.resize(n + 1);
    }

    void
    update(vector<int> &array, int position, int value)
    {
        int p_value = array[position];
        array[position] = value;
        for (; position < this->size; position = (position | (position + 1))) {
            this->tree[position] += (value - p_value);
        }
    }

    long long
    get(int from)
    {
        long long result = 0;
        for (; from > -1; from = (from & (from + 1)) - 1) {
                    result += this->tree[from];
        }
        return result;
    }

    long long
    sum(int l, int r)
    {
        return (get(r) - get(l - 1)); //если с l по r включительно
    }
};
```