Дерево отрезков

Denis Bakin

Постановка задачи

Дан массив a из n целых чисел. Нужно поддерживать два типа запросов:

Постановка задачи

Дан массив a из n целых чисел. Нужно поддерживать два типа запросов:

1. **Точечное обновление**: `a[k] = x`
2. **Сумма на отрезке**: посчитать $a_l + a_{l+1} + ... + a_{r-1}$

Постановка задачи

Дан массив a из n целых чисел. Нужно поддерживать два типа запросов:

1. **Точечное обновление**: `a[k] = x`
2. **Сумма на отрезке**: посчитать $a_l + a_{l+1} + ... + a_{r-1}$

Хотим оба запроса за $O(\log n)$

Какие у нас варианты?

Подход	Обновление	Сумма на отрезке
Массив «в лоб»	$O(1)$	$O(n)$
Префиксные суммы	$O(n)$	$O(1)$
Дерево отрезков	$O(\log n)$	$O(\log n)$

Какие у нас варианты?

Подход	Обновление	Сумма на отрезке
Массив «в лоб»	$O(1)$	$O(n)$
Префиксные суммы	$O(n)$	$O(1)$
Дерево отрезков	$O(\log n)$	$O(\log n)$

Дерево отрезков — компромисс, который делает **обе** операции быстрыми

Идея: рекурсивное разбиение

Идея: рекурсивное разбиение

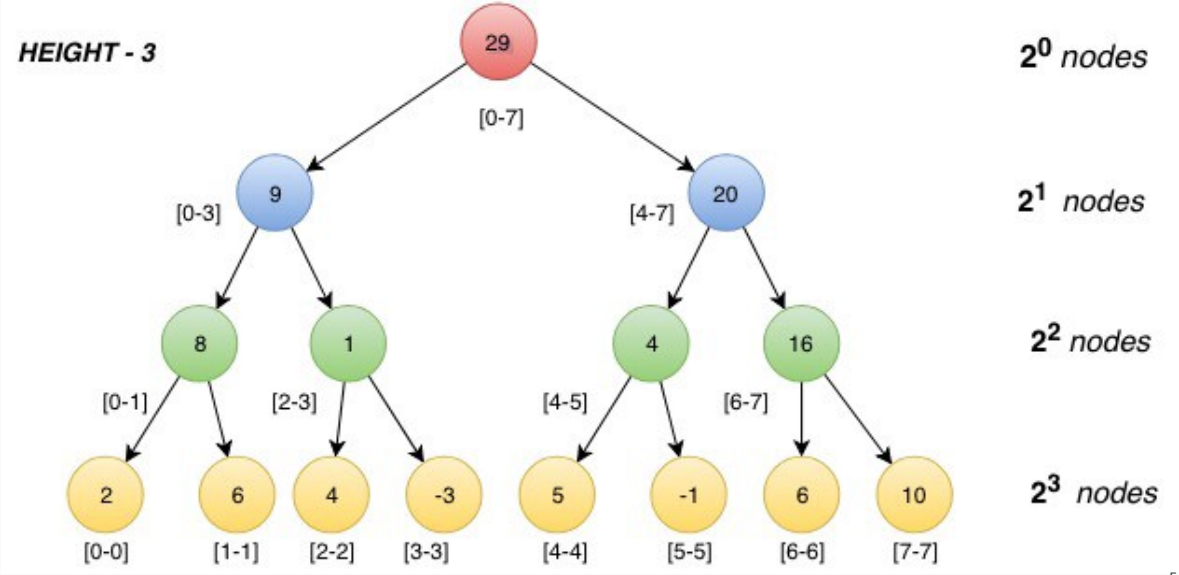
- посчитаем сумму всего массива
- разделим массив пополам, посчитаем сумму на половинах
- каждую половину снова разделим пополам
- продолжаем, пока не дойдём до отрезков длины 1

Идея: рекурсивное разбиение

- посчитаем сумму всего массива
- разделим массив пополам, посчитаем сумму на половинах
- каждую половину снова разделим пополам
- продолжаем, пока не дойдём до отрезков длины 1

Эту последовательность разбиений можно представить в виде **дерева**

Как это выглядит?



Свойство 1: высота

Свойство 1: высота

На каждом уровне длина отрезка уменьшается **вдвое**

Свойство 1: высота

На каждом уровне длина отрезка уменьшается **вдвое**

$$h = \Theta(\log n)$$

Свойство 1: высота

На каждом уровне длина отрезка уменьшается **вдвое**

$$h = \Theta(\log n)$$

Это ключевой факт — все асимптотики операций будут опираться на него

Свойство 2: память

Сколько всего вершин в дереве?

Свойство 2: память

Сколько всего вершин в дереве?

- 1-й уровень: 1 вершина
- 2-й уровень: ≤ 2 вершин
- 3-й уровень: ≤ 4 вершин
- ...
- последний уровень: $\leq n$ вершин

Свойство 2: память

Сколько всего вершин в дереве?

- 1-й уровень: 1 вершина
- 2-й уровень: ≤ 2 вершин
- 3-й уровень: ≤ 4 вершин
- ...
- последний уровень: $\leq n$ вершин

$$1 + 2 + 4 + ... + n < 2n$$

Свойство 2: память

Сколько всего вершин в дереве?

- 1-й уровень: 1 вершина
- 2-й уровень: ≤ 2 вершин
- 3-й уровень: ≤ 4 вершин
- ...
- последний уровень: $\leq n$ вершин

$$1 + 2 + 4 + ... + n < 2n$$

На практике берут массив размера $4n$ — гарантированно хватает для любого n

Свойство 3: разбиение отрезка

Любой полуинтервал $[l, r)$ разбивается на $O(\log n)$ непересекающихся отрезков, соответствующих вершинам дерева

Свойство 3: разбиение отрезка

Любой полуинтервал $[l, r)$ разбивается на $O(\log n)$ непересекающихся отрезков, соответствующих вершинам дерева

С каждого уровня нам понадобится не более **двух** отрезков

Свойство 3: разбиение отрезка

Любой полуинтервал $[l, r)$ разбивается на $O(\log n)$ непересекающихся отрезков, соответствующих вершинам дерева

С каждого уровня нам понадобится не более **двух** отрезков

Этим свойством мы будем пользоваться при ответе на запрос суммы

Запрос обновления

Нужно обновить значения в вершинах после `a[k] = x`

Запрос обновления

Нужно обновить значения в вершинах после `a[k] = x`

Изменятся только те вершины, в чей отрезок входит позиция k — по одной с каждого уровня, итого $\Theta(\log n)$ штук

Запрос обновления

Нужно обновить значения в вершинах после `a[k] = x`

Изменятся только те вершины, в чей отрезок входит позиция k — по одной с каждого уровня, итого $\Theta(\log n)$ штук

Реализуем рекурсивно:

1. спускаемся в того сына, в чей отрезок входит k
2. после возврата пересчитываем свою сумму как сумму значений в детях

Запрос суммы: три случая

Для каждой вершины в процессе спуска возможны **три случая**:

Запрос суммы: три случая

Для каждой вершины в процессе спуска возможны **три случая**:

1. **Отрезок вершины целиком в запросе** $\Rightarrow$ вернуть записанную сумму
2. **Не пересекается с запросом** $\Rightarrow$ вернуть 0
3. **Пересекается частично** $\Rightarrow$ рекурсивно спуститься в обоих детей

Почему сумма за $O(\log n)$?

«Интересные» отрезки — те, что попадают в случай 3 и порождают новые вызовы

Почему сумма за $O(\log n)$?

«Интересные» отрезки — те, что попадают в случай 3 и порождают новые вызовы

Это в точности отрезки, **содержащие границу запроса**

Почему сумма за $O(\log n)$?

«Интересные» отрезки — те, что попадают в случай 3 и порождают новые вызовы

Это в точности отрезки, **содержащие границу запроса**

- остальные либо целиком внутри (случай 1) — сразу завершаются
- либо целиком снаружи (случай 2) — тоже сразу завершаются

Почему сумма за $O(\log n)$?

«Интересные» отрезки — те, что попадают в случай 3 и порождают новые вызовы

Это в точности отрезки, **содержащие границу запроса**

- остальные либо целиком внутри (случай 1) — сразу завершаются
- либо целиком снаружи (случай 2) — тоже сразу завершаются

Каждая граница содержится в $O(\log n)$ отрезках $\Rightarrow$ итог $O(\log n)$

Соглашения о реализации

- работаем с **полуинтервалами** $[l, r)$, а не с отрезками $[l, r]$
- индексация массива с нуля
- индексация дерева с **единицы** (как в куче):
  - корень — `tree[1]`
  - дети вершины v — `tree[2v]` и `tree[2v + 1]`
- размер массива — $4n$

Реализация: интерфейс класса

```
class SegmentTree {
public:
    SegmentTree(const std::vector<int64_t>& a)
        : n(a.size()), tree(4 * n) {
        Build(1, 0, n, a);
    }
    void Update(size_t pos, int64_t value) {
        Update(1, 0, n, pos, value);
    }
    int64_t Sum(size_t l, size_t r) {
        return Sum(1, 0, n, l, r);
    }

private:
    size_t n;
    std::vector<int64_t> tree;
    // ... приватные методы
};
```

Реализация: интерфейс класса

```
class SegmentTree {
public:
    SegmentTree(const std::vector<int64_t>& a)
        : n(a.size()), tree(4 * n) {
        Build(1, 0, n, a);
    }
    void Update(size_t pos, int64_t value) {
        Update(1, 0, n, pos, value);
    }
    int64_t Sum(size_t l, size_t r) {
        return Sum(1, 0, n, l, r);
    }

private:
    size_t n;
    std::vector<int64_t> tree;
    // ... приватные методы
};
```

Реализация: построение

```
void Build(size_t v, size_t l, size_t r,
           const std::vector<int64_t>& a) {
    if (r - l == 1) {       // лист
        tree[v] = a[l];
        return;
    }
    size_t mid = (l + r) / 2;
    Build(2 * v, l, mid, a);
    Build(2 * v + 1, mid, r, a);
    tree[v] = tree[2 * v] + tree[2 * v + 1];
}
```

Каждая вершина обрабатывается ровно один раз $\Rightarrow$ построение за $O(n)$

Реализация: точечное обновление

```
void Update(size_t v, size_t l, size_t r,
            size_t pos, int64_t value) {
    if (r - l == 1) {
        tree[v] = value;
        return;
    }
    size_t mid = (l + r) / 2;
    if (pos < mid) {
        Update(2 * v, l, mid, pos, value);
    } else {
        Update(2 * v + 1, mid, r, pos, value);
    }
    tree[v] = tree[2 * v] + tree[2 * v + 1];
}
```

Спускаемся в одного сына $\Rightarrow O(\log n)$ вершин на пути

Реализация: сумма на отрезке

```
int64_t Sum(size_t v, size_t l, size_t r,
            size_t ql, size_t qr) {
    if (qr <= l || r <= ql) return 0;       // случай 2
    if (ql <= l && r <= qr) return tree[v]; // случай 1
    size_t mid = (l + r) / 2;               // случай 3
    return Sum(2 * v, l, mid, ql, qr)
         + Sum(2 * v + 1, mid, r, ql, qr);
}
```

Три случая в коде один к одному соответствуют тому, что мы обсуждали

Сложность операций

Операция	Сложность
Построение	$O(n)$
Точечное обновление	$O(\log n)$
Сумма на отрезке	$O(\log n)$
Память	$O(n)$

Сложность операций

Операция	Сложность
Построение	$O(n)$
Точечное обновление	$O(\log n)$
Сумма на отрезке	$O(\log n)$
Память	$O(n)$

И обновление, и запрос за $O(\log n)$ — то, чего мы хотели!

Какие операции можно использовать?

Достаточно, чтобы операция была **ассоциативной**:

$$(a \star b) \star c = a \star (b \star c)$$

Какие операции можно использовать?

Достаточно, чтобы операция была **ассоциативной**:

$$(a \star b) \star c = a \star (b \star c)$$

Тогда в каждой вершине: $\mathsf{tree}[v] = \mathsf{tree}[2v] \star \mathsf{tree}[2v+1]$

Какие операции можно использовать?

Достаточно, чтобы операция была **ассоциативной**:

$$(a \star b) \star c = a \star (b \star c)$$

Тогда в каждой вершине: $\text{tree}[v] = \text{tree}[2v] \star \text{tree}[2v+1]$

Подходящие операции:

- сумма, минимум, максимум
- НОД, НОК
- побитовые `AND`, `OR`, `XOR`
- произведение по модулю

Новая задача: операции на отрезке

А что если мы хотим не только менять отдельные элементы, но и **присваивать значение всем элементам отрезка**?

Новая задача: операции на отрезке

А что если мы хотим не только менять отдельные элементы, но и **присваивать значение всем элементам отрезка**?

```
assign(l, r, x):  для всех i из [l, r): a[i] = x
```

Новая задача: операции на отрезке

А что если мы хотим не только менять отдельные элементы, но и **присваивать значение всем элементам отрезка**?

```
assign(l, r, x):  для всех i из [l, r): a[i] = x
```

Если делать «в лоб» через точечные обновления — изменим до $O(n)$ элементов на запрос

Новая задача: операции на отрезке

А что если мы хотим не только менять отдельные элементы, но и **присваивать значение всем элементам отрезка**?

```
assign(l, r, x):  для всех i из [l, r): a[i] = x
```

Если делать «в лоб» через точечные обновления — изменим до $O(n)$ элементов на запрос

Вся выгода от структуры теряется...

Идея: отложенные изменения

Идея: отложенные изменения

Будем **«помечать» поддерево** меткой $\text{lazy}_v = x$:

«всем элементам этого отрезка должно быть присвоено x, но я ещё не успел этого сделать»

Идея: отложенные изменения

Будем **«помечать» поддерево** меткой $\text{lazy}_v = x$:

«всем элементам этого отрезка должно быть присвоено x, но я ещё не успел этого сделать»

Например, запрос «присвоить x на всём массиве» — просто ставим метку в **корне**, ничего не меняя!

Идея: отложенные изменения

Будем **«помечать» поддерево** меткой $\text{lazy}_v = x$:

«всем элементам этого отрезка должно быть присвоено x, но я ещё не успел этого сделать»

Например, запрос «присвоить x на всём массиве» — просто ставим метку в **корне**, ничего не меняя!

По-английски эта техника называется *lazy propagation*

Как обращаться с метками?

Когда позже понадобятся правильные значения детей — выполним **«проталкивание»** (`Push`):

Как обращаться с метками?

Когда позже понадобятся правильные значения детей — выполним **«проталкивание»** (`Push`):

1. пересчитаем сумму текущего отрезка по метке
2. передадим метку детям
3. сбросим метку у себя

Как обращаться с метками?

Когда позже понадобятся правильные значения детей — выполним **«проталкивание»** (`Push`):

1. пересчитаем сумму текущего отрезка по метке
2. передадим метку детям
3. сбросим метку у себя

Спускаться рекурсивно сразу не нужно — `Push` сработает у детей в нужный момент

Как обращаться с метками?

Когда позже понадобятся правильные значения детей — выполним **«проталкивание»** (`Push`):

1. пересчитаем сумму текущего отрезка по метке
2. передадим метку детям
3. сбросим метку у себя

Спускаться рекурсивно сразу не нужно — `Push` сработает у детей в нужный момент

Если у ребёнка уже была старая метка — она просто **перезаписывается** (важно для экономии работы)

Push: реализация

```
void Push(size_t v, size_t l, size_t r) {
    if (lazy[v] == -1) return;          // метки нет
    tree[v] = (r - l) * lazy[v];        // применили метку у себя
    if (r - l > 1) {                    // если есть дети
        lazy[2 * v] = lazy[v];          // передали метку детям
        lazy[2 * v + 1] = lazy[v];
    }
    lazy[v] = -1;                       // сбросили свою метку
}
```

Push: реализация

```
void Push(size_t v, size_t l, size_t r) {
    if (lazy[v] == -1) return;          // метки нет
    tree[v] = (r - l) * lazy[v];        // применили метку у себя
    if (r - l > 1) {                    // если есть дети
        lazy[2 * v] = lazy[v];          // передали метку детям
        lazy[2 * v + 1] = lazy[v];
    }
    lazy[v] = -1;                       // сбросили свою метку
}
```

`lazy = -1` — соглашение «метки нет» (если значения неотрицательные)

Запрос Assign

```
void Assign(size_t v, size_t l, size_t r,
            size_t ql, size_t qr, int64_t value) {
    Push(v, l, r);
    if (qr <= l || r <= ql) return;
    if (ql <= l && r <= qr) {
        lazy[v] = value;
        Push(v, l, r);
        return;
    }
    size_t mid = (l + r) / 2;
    Assign(2 * v, l, mid, ql, qr, value);
    Assign(2 * v + 1, mid, r, ql, qr, value);
    tree[v] = tree[2 * v] + tree[2 * v + 1];
}
```

Запрос Assign

```
void Assign(size_t v, size_t l, size_t r,
            size_t ql, size_t qr, int64_t value) {
    Push(v, l, r);
    if (qr <= l || r <= ql) return;
    if (ql <= l && r <= qr) {
        lazy[v] = value;
        Push(v, l, r);
        return;
    }
    size_t mid = (l + r) / 2;
    Assign(2 * v, l, mid, ql, qr, value);
    Assign(2 * v + 1, mid, r, ql, qr, value);
    tree[v] = tree[2 * v] + tree[2 * v + 1];
}
```

Те же три случая, что и в Sum — но теперь с Push в начале

Запрос Sum (с lazy)

```
int64_t Sum(size_t v, size_t l, size_t r,
            size_t ql, size_t qr) {
    Push(v, l, r);
    if (qr <= l || r <= ql) return 0;
    if (ql <= l && r <= qr) return tree[v];
    size_t mid = (l + r) / 2;
    return Sum(2 * v, l, mid, ql, qr)
         + Sum(2 * v + 1, mid, r, ql, qr);
}
```

Запрос Sum (с lazy)

```
int64_t Sum(size_t v, size_t l, size_t r,
            size_t ql, size_t qr) {
    Push(v, l, r);
    if (qr <= l || r <= ql) return 0;
    if (ql <= l && r <= qr) return tree[v];
    size_t mid = (l + r) / 2;
    return Sum(2 * v, l, mid, ql, qr)
         + Sum(2 * v + 1, mid, r, ql, qr);
}
```

Главное отличие — вызов Push в самом начале

Запрос Sum (с lazy)

```
int64_t Sum(size_t v, size_t l, size_t r,
            size_t ql, size_t qr) {
    Push(v, l, r);
    if (qr <= l || r <= ql) return 0;
    if (ql <= l && r <= qr) return tree[v];
    size_t mid = (l + r) / 2;
    return Sum(2 * v, l, mid, ql, qr)
         + Sum(2 * v + 1, mid, r, ql, qr);
}
```

Главное отличие — вызов `Push` в самом начале

Инвариант: после `Push` в текущей вершине `tree[v]` корректно

Итоги

- **Дерево отрезков** — структура для запросов на отрезках за $O(\log n)$
- Каждая вершина хранит «значение» на своём отрезке
- Высота $\Theta(\log n)$, память $O(n)$
- Хранится в массиве как куча: корень — `tree[1]`, дети — $2v$ и $2v + 1$
- Обобщается на любую **ассоциативную** операцию (min, max, gcd, xor, ...)
- **Отложенные изменения** позволяют делать операции и на **отрезке** за $O(\log n)$