Лекция 10. Дерево Фенвика. Разреженная таблица (Sparse Table).

1. Введение

На предыдущей лекции мы изучили **дерево отрезков** – мощную структуру для выполнения запросов на отрезках. Однако для некоторых задач, особенно когда нужны только **операции на префиксе** (например, сумма на префиксе), существуют более простые и эффективные структуры данных.

Дерево Фенвика (Fenwick Tree) и **Разреженная таблица (Sparse Table)** – две такие структуры, которые сочетают в себе простоту реализации и высокую производительность.

2. Дерево Фенвика (Fenwick Tree)

Дерево Фенвика (также известно как **Binary Indexed Tree, BIT**) – это структура данных, которая позволяет выполнять две основные операции:

1. **update(i, delta)** – увеличить элемент a[i] на delta.
2. **prefix_sum(i)** – вернуть сумму элементов на префиксе [0..i].

С их помощью можно легко реализовать:

- sum(l, r) = prefix_sum(r) - prefix_sum(l-1)
- Точно так же можно поддерживать и другие операции (например, минимум), но это менее очевидно.

Преимущества перед деревом отрезков:

- Меньший объём кода.
- Меньшая константа в времени работы.
- Память: O(n) (ровно n элементов).

2.1. Идея дерева Фенвика

Каждый элемент tree[i] хранит сумму элементов на отрезке [i - g(i) + 1 .. i], где g(i) – наибольшая степень двойки, на которую делится i (последний единичный бит в двоичной записи i).

Пример:

i = 12 (двоичное 1100) → g(12) = 4 → tree[12] хранит сумму a[9..12].

Операции:

- **prefix_sum(i)**. Суммируем tree[i], затем вычитаем g(i) и повторяем.
- **update(i, delta)**. Добавляем delta в tree[i], затем увеличиваем i на g(i) и повторяем.

2.2. Вычисление g(i)

g(i) = i & -i – эта операция оставляет только последний единичный бит.

Пример:

```
i = 12  # 1100
g = i & -i  # 1100 & 0100 = 0100 → 4
```

2.3. Реализация дерева Фенвика

```
class FenwickTree:
    def __init__(self, size: int):
        self.n = size
        # Создаём массив для дерева Фенвика размером n+1
        # Индексация с 1 для удобства работы с битовыми операциями
        self.tree = [0] * (self.n + 1)  # Индексация с 1

    def update(self, i: int, delta: int) -> None:
        """Увеличить a[i] на delta (индексация с 1)."""
        # Начинаем с позиции i и поднимаемся вверх по дереву
        while i <= self.n:
            # Добавляем delta к текущему узлу
            self.tree[i] += delta
            # Переходим к следующему узлу: i += младший значащий бит (LSB)
            # Это перемещает нас к узлу, который отвечает за больший отрезок
            i += i & -i
            # Пример: если i=3 (бинарно 011), то i & -i = 1 → i=4
            # если i=4 (бинарно 100), то i & -i = 4 → i=8

    def prefix_sum(self, i: int) -> int:
        """Сумма на префиксе [1..i]."""
        s = 0
        # Начинаем с позиции i и спускаемся вниз по дереву
        while i > 0:
            # Добавляем значение текущего узла к сумме
            s += self.tree[i]
            # Переходим к предыдущему узлу: i -= младший значащий бит (LSB)
            # Это перемещает нас к узлу, который отвечает за предыдущий отре-
зок
            i -= i & -i
            # Пример: если i=7 (бинарно 111), то i & -i = 1 → i=6
            # если i=6 (бинарно 110), то i & -i = 2 → i=4
        return s

    def range_sum(self, l: int, r: int) -> int:
        """Сумма на отрезке [l, r] (индексация с 1)."""
        if l > r:
            return 0
        # Сумма на отрезке [l, r] = сумма на префиксе [1..r] - сумма на пре-
фиксе [1..l-1]
        return self.prefix_sum(r) - self.prefix_sum(l - 1)


# Пример использования:
arr = [1, 3, 5, 7, 9, 11]
n = len(arr)
# Создаём дерево Фенвика для массива из 6 элементов
ft = FenwickTree(n)
```

```
# Инициализируем дерево: добавляем каждый элемент по очереди
for i in range(1, n + 1):
    ft.update(i, arr[i - 1])  # arr[i-1] потому что индексация в arr с 0, а в дереве с 1

# Запрос суммы на отрезке [2, 4]
# В человеческих терминах: элементы с индексами 2,3,4 (значения 3,5,7)
print(ft.range_sum(2, 4))  # 3+5+7=15

# Обновляем элемент с индексом 3: увеличиваем его на 10
# Было a[3]=5, стало 15
ft.update(3, 10)

# Снова запрашиваем сумму на том же отрезке
print(ft.range_sum(2, 4))  # 3+15+7=25
```

2.4. Анализ сложности

- **Построение:** O(n log n) (если делать n вызовов update).
- **update:** O(log n)
- **prefix_sum / range_sum:** O(log n)

Примечание. Можно построить дерево Фенвика за O(n), если инициализировать массив префиксных сумм и затем вычислить tree[i] как сумму на отрезке [i - g(i) + 1 .. i].

3. Разреженная таблица (Sparse Table)

Разреженная таблица – это структура данных, которая позволяет отвечать на запросы **минимума на отрезке (RMQ)** за O(1) после предобработки за O(n log n).

Ограничение:

- Массив **не должен изменяться** после построения таблицы.
- Операция должна быть **идемпотентной** (f(x, x) = x), например, минимум, максимум, НОД.

3.1. Идея разреженной таблицы

Мы предподсчитываем ответы для всех отрезков длины 2^k для k = 0..log2(n).

st[k][i] – значение операции на отрезке [i, i + 2^k - 1].

Построение:

- st[0][i] = a[i]
- st[k][i] = f(st[k-1][i], st[k-1][i + 2^(k-1)])

Запрос на отрезке [l, r]:

- Находим максимальное k, такое что 2^k <= len = r - l + 1
- Ответ = f(st[k][l], st[k][r - 2^k + 1])

3.2. Реализация разреженной таблицы для минимума

```
import math
from typing import List


class SparseTable:
    def __init__(self, arr: List[int], func=min):
        self.n = len(arr)  # Длина исходного массива
        self.func = func  # Функция для вычисления (min, max, gcd и т.д.)

        # Определяем количество уровней: log2(n) округлённое вверх
        k = self.n.bit_length()

        # Создаём таблицу: k уровней × n элементов
        # st[j][i] будет хранить результат функции на отрезке [i, i + 2^j -
1]
        self.st = [[0] * self.n for _ in range(k)]

        # Массив для быстрого вычисления логарифмов
        self.log = [0] * (self.n + 1)

        # Предподсчет логарифмов - заполняем массив log заранее
        # log[i] = floor(log2(i))
        for i in range(2, self.n + 1):
            self.log[i] = self.log[i // 2] + 1

        # Базовый случай: отрезки длины 1 (2^0 = 1)
        # st[0][i] = значение на отрезке [i, i] (т.е. сам элемент)
        for i in range(self.n):
            self.st[0][i] = arr[i]

        # Заполнение таблицы для больших длин
        for j in range(1, k):  # Для каждого уровня j (длина отрезка 2^j)
            step = 1 << (j - 1)  # 2^(j-1) - половина длины текущего отрезка

            # Проходим по всем начальным позициям, для которых отрезок длины
2^j помещается в массив
            for i in range(self.n - (1 << j) + 1):
                # Объединяем два отрезка половинной длины:
                # [i, i + 2^(j-1) - 1] и [i + 2^(j-1), i + 2^j - 1]
                self.st[j][i] = self.func(self.st[j - 1][i], self.st[j - 1][i
+ step])

    def query(self, l: int, r: int) -> int:
        """Запрос на отрезке [l, r] (индексы с 0)."""
        length = r - l + 1  # Длина запрашиваемого отрезка

        # Находим максимальную степень двойки, не превышающую длину отрезка
        k = self.log[length]

        # Отрезок [l, r] покрывается двумя отрезками длины 2^k:
        # [l, l + 2^k - 1] и [r - 2^k + 1, r]
```

```python
        # Эти отрезки перекрываются, но для идемпотентных операций (min, max) это не проблема
        return self.func(self.st[k][l], self.st[k][r - (1 << k) + 1])


# Пример использования:
arr = [1, 3, 2, 7, 9, 11, 0, 5]
# Создаём разреженную таблицу для поиска минимума
st = SparseTable(arr, func=min)

# Запрос минимума на отрезке [1, 5]
# Элементы: arr[1]=3, arr[2]=2, arr[3]=7, arr[4]=9, arr[5]=11
print(st.query(1, 5))  # min(3, 2, 7, 9, 11) = 2

# Запрос минимума на всём массиве [0, 7]
print(st.query(0, 7))  # min(1, 3, 2, 7, 9, 11, 0, 5) = 0
```

3.3. Анализ сложности

- **Память:** O(n log n)
- **Предподсчёт:** O(n log n)
- **Запрос:** O(1)

4. Сравнение структур

Структура	**Построение**	**Запрос**	**Обновление**	**Память**	**Применение**
Дерево Фенвика	O(n log n)	O(log n)	O(log n)	O(n)	Сумма, обратимые операции
Разреженная табл.	O(n log n)	O(1)	-	O(n log n)	Минимум, максимум, НОД (стат.)
Дерево отрезков	O(n)	O(log n)	O(log n)	O(n)	Любые операции, обновления