Лекция 9. Дерево отрезков. Базовые операции.

1. Введение и мотивация

Дерево отрезков (Segment Tree) – это мощная древовидная структура данных, позволяющая эффективно выполнять различные запросы на отрезках массива (например, сумма, минимум, максимум) и обновлять элементы.

Зачем нужно?

- **Запросы на отрезке.** Найти сумму/минимум/максимум на отрезке [L, R].
- **Обновление элементов.** Изменить значение одного элемента или всех элементов на отрезке.
- **Динамические данные.** Когда массив может изменяться, и нужно быстро отвечать на запросы.

Сложность:

- Построение: O(n)
- Запрос/обновление: O(log n)
- Память: O(n)

Сравнение с другими структурами:

- **Префиксные суммы.** Быстрый запрос суммы (O(1)), но обновление элемента – O(n).
- **Дерево Фенвика.** Проще в реализации, но поддерживает меньше операций (только обратимые, например, сумма).
- **Дерево отрезков.** Универсальнее, поддерживает любые ассоциативные операции (сумма, минимум, максимум и др.).

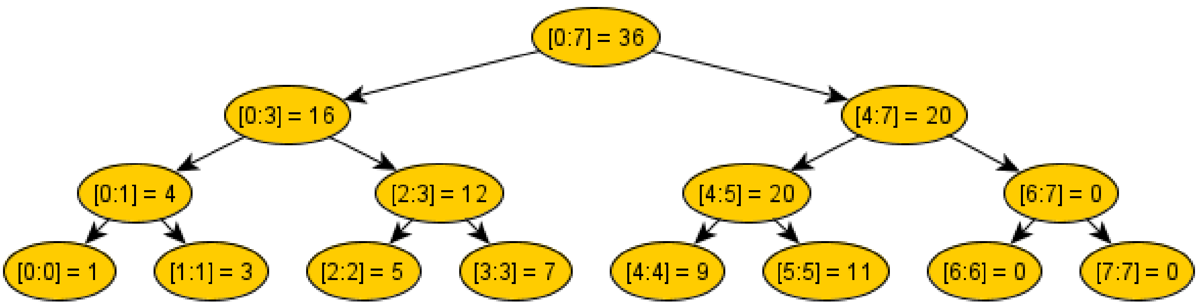
2. Структура дерева отрезков

Дерево отрезков – это полное бинарное дерево, где:

- Каждый лист соответствует элементу массива.
- Каждый внутренний узел хранит результат операции (например, сумму) для отрезка своих потомков.

Пример массива: [1, 3, 5, 7, 9, 11] (n=6)

Представление в виде дерева:



Хранение в массиве:
Для удобства дерево хранится в массиве tree[]:
Корень – tree[1] (индекс 1).
Для узла i:
- Левый ребёнок: 2*i
- Правый ребёнок: 2*i + 1
- Родитель: i // 2

Размер массива tree – 4*n (для гарантии достаточного размера).

3. Построение дерева отрезков

Рекурсивная функция build:
- Если узел – лист (отрезок длины 1), записываем значение элемента.
- Иначе рекурсивно строим левое и правое поддеревья и объединяем результаты.

```
from typing import List, Callable


class SegmentTree:
    def __init__(self, data: List[int], func: Callable[[int, int], int],
neutral: int):
        # Инициализация дерева отрезков
        self.n = len(data)  # Длина исходного массива
        self.func = func  # Функция для объединения значений (сумма, минимум
и т.д.)
        self.neutral = neutral  # Нейтральный элемент для функции (0 для
суммы, inf для минимума)

        # Находим ближайшую степень двойки, не меньшую n
        self.size = 1
        while self.size < self.n:
            self.size *= 2

        # Создаём массив дерева размером 2*size, заполняем нейтральными
элементами
        self.tree = [self.neutral] * (2 * self.size)
        self._build(data)  # Строим дерево

    def _build(self, data: List[int]) -> None:
        """Построение дерева отрезков из исходного массива"""
        # Заполняем листья - элементы с индексами от size до size+n-1
        for i in range(self.n):
            self.tree[self.size + i] = data[i]

        # Заполняем внутренние узлы снизу вверх
        # Идём от последнего внутреннего узла к корню
        for i in range(self.size - 1, 0, -1):
            # Значение узла = функция от левого и правого ребёнка
            self.tree[i] = self.func(self.tree[2 * i], self.tree[2 * i + 1])
```

Пример для суммы:

```python
data = [1, 3, 5, 7, 9, 11]
st = SegmentTree(data, func=lambda a, b: a + b, neutral=0)
# tree = [0, 36, 16, 20, 4, 12, 20, 0, 1, 3, 5, 7, 9, 11, 0, 0]
```

4. Запрос на отрезке

Функция query(l, r) возвращает результат операции на отрезке [l, r] (полуинтервал [l, r)).

Алгоритм:

1. Начинаем с корня.
2. Если текущий отрезок полностью внутри [l, r), возвращаем значение узла.
3. Если текущий отрезок не пересекается с [l, r), возвращаем нейтральный элемент.
4. Иначе рекурсивно запрашиваем левую и правую половины и объединяем результаты.

```python
def query(self, l: int, r: int) -> int:
    """Запрос на полуинтервале [l, r) - l включительно, r не включительно"""
    l += self.size  # Переходим к индексам в дереве
    r += self.size
    res_left = self.neutral  # Результат для левой части
    res_right = self.neutral  # Результат для правой части

    # Поднимаемся по дереву, пока границы не сомкнутся
    while l < r:
        # Если l - нечётный (правый ребёнок), то он не полностью входит в
родительский отрезок
        if l & 1:  # Эквивалентно l % 2 == 1
            res_left = self.func(res_left, self.tree[l])
            l += 1  # Переходим к следующему узлу

        # Если r - нечётный (правый ребёнок)
        if r & 1:
            r -= 1  # Переходим к предыдущему узлу
            res_right = self.func(self.tree[r], res_right)  # Добавляем
справа

        # Поднимаемся на уровень выше
        l //= 2
        r //= 2

    # Объединяем результаты левой и правой частей
    return self.func(res_left, res_right)
```

Пример:

```python
print(st.query(1, 4))  # Сумма элементов [3, 5, 7] = 15
```

5. Обновление элемента

Функция update(i, value) изменяет элемент data[i] на value.

Алгоритм:

1. Находим лист, соответствующий индексу i.
2. Обновляем значение листа.
3. Поднимаемся к корню, пересчитывая значения родителей.

```
def update(self, i: int, value: int) -> None:
    """Обновление элемента на позиции i"""
    i += self.size  # Переходим к индексу в дереве
    self.tree[i] = value  # Обновляем лист

    # Поднимаемся к корню, обновляя родителей
    i //= 2
    while i >= 1:
        # Пересчитываем значение узла на основе детей
        self.tree[i] = self.func(self.tree[2 * i], self.tree[2 * i + 1])
        i //= 2  # Переходим к родителю
```

Пример:

```
st.update(2, 10)  # Было 5, стало 10
print(st.query(0, 3))  # Сумма [1, 3, 10] = 14
```

6. Массовые операции (отложенные обновления)

Иногда нужно обновить не один элемент, а целый отрезок (например, прибавить ко всем элементам x). Наивный подход – O(n), но с **ленивым распространением** (Lazy Propagation) можно за O(log n).

Идея:

- В каждом узле храним «ленивое» значение lazy, которое нужно прибавить ко всем элементам поддерева.
- При запросе или обновлении «проталкиваем» lazy детям.

Расширенная реализация:

```
from typing import List, Callable


class LazySegmentTree:
    def __init__(self, data: List[int], func: Callable[[int, int], int],
neutral: int):
        # Аналогично базовому дереву
        self.n = len(data)
        self.func = func
        self.neutral = neutral
        self.size = 1
        while self.size < self.n:
            self.size *= 2
        self.tree = [self.neutral] * (2 * self.size)
        self.lazy = [0] * (2 * self.size)  # Массив для хранения отложенных
операций
        self._build(data)

    def _build(self, data: List[int]) -> None:
        """Построение дерева (аналогично базовой версии)"""
        for i in range(self.n):
            self.tree[self.size + i] = data[i]
```

```
        for i in range(self.size - 1, 0, -1):
            self.tree[i] = self.func(self.tree[2 * i], self.tree[2 * i + 1])

    def _apply(self, i: int, value: int, len: int) -> None:
        """Применение отложенной операции к узлу i, который покрывает len
элементов"""
        # Для суммы: значение узла увеличивается на value * количество
элементов
        self.tree[i] += value * len

        # Если узел не лист, сохраняем отложенную операцию для детей
        if i < self.size:
            self.lazy[i] += value

    def _push(self, i: int, len: int) -> None:
        """Проталкивание отложенной операции из узла i к его детям"""
        if self.lazy[i] != 0:  # Если есть отложенная операция
            # Применяем операцию к обоим детям
            self._apply(2 * i, self.lazy[i], len // 2)
            self._apply(2 * i + 1, self.lazy[i], len // 2)
            self.lazy[i] = 0  # Сбрасываем отложенную операцию

    def update_range(self, l: int, r: int, value: int) -> None:
        """Прибавление value ко всем элементам на отрезке [l, r)"""
        l += self.size
        r += self.size
        l0, r0 = l, r  # Сохраняем исходные позиции для пересчёта
        len_seg = 1  # Длина отрезка текущего узла (начинаем с 1)

        # Первый проход: применяем операцию к покрывающим отрезкам
        while l < r:
            if l & 1:  # l - правый ребёнок
                self._apply(l, value, len_seg)
                l += 1
            if r & 1:  # r - правый ребёнок
                r -= 1
                self._apply(r, value, len_seg)
            l //= 2
            r //= 2
            len_seg *= 2  # На каждом уровне длина отрезка удваивается

        # Второй проход: пересчитываем значения родителей
        self._pull(l0 // 2)
        self._pull((r0 - 1) // 2)

    def _pull(self, i: int) -> None:
        """Пересчёт значений родителей после обновления"""
        len_seg = 2  # Длина отрезка для пересчёта (2 ребёнка)
        while i >= 1:
            # Пересчитываем значение с учётом отложенных операций
            self.tree[i] = self.func(self.tree[2 * i], self.tree[2 * i + 1])
+ self.lazy[i] * len_seg
            i //= 2
            len_seg *= 2
```

```python
    def query_range(self, l: int, r: int) -> int:
        """Запрос на отрезке [l, r) с учётом отложенных операций"""
        l += self.size
        r += self.size

        # Шаг 1: Проталкивание отложенных операций для левой границы
        # Стек для запоминания пути от левой границы к корню
        stack_left = []
        i = l
        len_seg = 1  # Длина отрезка на текущем уровне

        # Поднимаемся от левой границы к корню, запоминая путь
        while i > 1:
            i //= 2  # Переходим к родителю
            len_seg *= 2  # Длина отрезка удваивается
            stack_left.append((i, len_seg))

        # Проталкиваем отложенные операции в обратном порядке (от корня к
листьям)
        while stack_left:
            node, seg_len = stack_left.pop()
            self._push(node, seg_len)

        # Шаг 2: Проталкивание отложенных операций для правой границы
        # Стек для запоминания пути от правой границы к корню
        stack_right = []
        j = r - 1  # Используем r-1, так как полуинтервал [l, r)
        len_seg = 1

        # Поднимаемся от правой границы к корню, запоминая путь
        while j > 1:
            j //= 2
            len_seg *= 2
            stack_right.append((j, len_seg))

        # Проталкиваем отложенные операции в обратном порядке
        while stack_right:
            node, seg_len = stack_right.pop()
            self._push(node, seg_len)

        # Шаг 3: Выполнение запроса (аналогично базовой версии)
        res_left = self.neutral
        res_right = self.neutral

        while l < r:
            if l & 1:  # l - правый ребёнок
                res_left = self.func(res_left, self.tree[l])
                l += 1
            if r & 1:  # r - правый ребёнок
                r -= 1
                res_right = self.func(self.tree[r], res_right)
            l //= 2
            r //= 2

        return self.func(res_left, res_right)
```

Пример:

```
lst = LazySegmentTree([1, 3, 5, 7, 9, 11], func=lambda a, b: a + b,
neutral=0)
lst.update_range(1, 4, 2)  # Прибавить 2 к элементам [3, 5, 7]
print(lst.query_range(0, 6))  # Сумма [1, 5, 7, 9, 9, 11] = 42
```

7. Другие операции

Дерево отрезков поддерживает любые **ассоциативные** операции (удовлетворяющие свойству (a ∘ b) ∘ c = a ∘ (b ∘ c)).

Примеры:

- **Минимум:** func = min, neutral = inf
- **Максимум:** func = max, neutral = -inf
- **Произведение:** func = lambda a, b: a * b, neutral = 1
- **НОД:** func = math.gcd, neutral = 0

Реализация для минимума:

```
import math

data = [1, 3, 5, 7, 9, 11]
st_min = SegmentTree(data, func=min, neutral=math.inf)
print(st_min.query(1, 4))  # min(3, 5, 7) = 3
```

8. Оптимизации и замечания

- **Рекурсивная vs Итеративная реализация.** Итеративная обычно быстрее и избегает ограничений рекурсии.
- **Память.** Размер массива 4*n гарантирует достаточность.
- **Индексация с 1.** Упрощает вычисления детей и родителей.
- **Нейтральный элемент.** Важен для корректной работы (например, 0 для суммы, inf для минимума).