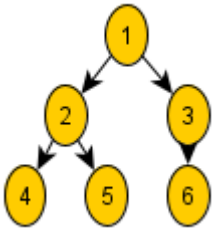
Лекция 13. Наименьший общий предок (LCA), бинарные подъёмы, Фарах-Колтон и Бендер.

1. Введение в задачу LCA

Наименьший общий предок (Lowest Common Ancestor, LCA) двух узлов u и v в корневом дереве – это самый глубокий узел, который является предком как u, так и v.

Пример:



LCA(4, 5) = 2
LCA(4, 6) = 1
LCA(2, 2) = 2

Зачем нужно LCA?

- Нахождение расстояния между двумя узлами: dist(u, v) = depth[u] + depth[v] - 2 * depth[lca].
- Проверка, лежит ли узел u на пути между v и корнем.
- Решение сложных задач на деревьях (например, подсчёт путей с определёнными свойствами).
- Основывается во многих алгоритмах, например, для поиска мостов и точек сочленения.

2. Метод бинарных подъёмов

2.1. Идея метода

Метод бинарных подъёмов позволяет находить LCA за O(log N) на запрос после предобработки за O(N log N).

Для каждого узла предпосчитывается массив up, где:

- up[u][0] – непосредственный родитель u.
- up[u][1] = up[up[u][0]][0] – предок на расстоянии 2.
- up[u][k] = up[up[u][k-1]][k-1] – предок на расстоянии 2^k.

Также хранится массив depth[] – глубина каждого узла.

2.2. Предобработка (DFS)

Перед ответом на запросы необходимо заполнить массивы up и depth с помощью обхода в глубину (DFS).

2.3. Реализация метода бинарных подъёмов

```python
from collections import defaultdict


class LCABinaryLifting:
    """
    Класс для нахождения LCA с помощью метода бинарных подъёмов.
    Подходит для деревьев любого размера, работает за O(log N) на запрос.
    """

    def __init__(self, n: int):
        """Инициализация класса.

        Args:
            n: Количество вершин в дереве
        """
        self.n = n
        self.graph = defaultdict(list)  # Список смежности для хранения
дерева
        # LOG - максимальная степень двойки, которая нам может понадобиться
        self.LOG = n.bit_length()  # log2(n) округлённый вверх
        # up[u][k] - 2^k-й предок вершины u
        self.up = [[-1] * (self.LOG + 1) for _ in range(n + 1)]
        self.depth = [0] * (n + 1)  # Глубина каждой вершины от корня

    def add_edge(self, u: int, v: int):
        """Добавление ребра в дерево.

        Args:
            u: Первая вершина ребра
            v: Вторая вершина ребра
        """
        self.graph[u].append(v)
        self.graph[v].append(u)

    def dfs(self, u: int, p: int):
        """Обход в глубину для предподсчёта массивов up и depth.

        Args:
            u: Текущая вершина
            p: Родитель текущей вершины
        """
        # Запоминаем непосредственного родителя
        self.up[u][0] = p

        # Предподсчёт всех степеней двойки для вершины u
        for i in range(1, self.LOG + 1):
            if self.up[u][i - 1] != -1:
                # 2^i-й предок = 2^(i-1)-й предок от 2^(i-1)-го предка
                self.up[u][i] = self.up[self.up[u][i - 1]][i - 1]
            else:
                self.up[u][i] = -1  # Если вышли за корень
```

```python
        # Рекурсивно обрабатываем всех детей
        for v in self.graph[u]:
            if v != p:  # Исключаем обратное движение к родителю
                self.depth[v] = self.depth[u] + 1  # Увеличиваем глубину
                self.dfs(v, u)  # Рекурсивный вызов для потомка

    def preprocess(self, root: int):
        """Запуск предобработки дерева.

        Args:
            root: Корень дерева
        """
        self.depth[root] = 0  # Глубина корня = 0
        self.dfs(root, -1)  # Запускаем DFS из корня

    def lca(self, u: int, v: int) -> int:
        """Нахождение наименьшего общего предка двух вершин.

        Args:
            u: Первая вершина
            v: Вторая вершина

        Returns:
            Номер вершины - LCA(u, v)
        """
        # Гарантируем, что u не выше v
        if self.depth[u] < self.depth[v]:
            u, v = v, u

        # Поднимаем u до уровня v
        diff = self.depth[u] - self.depth[v]
        for i in range(self.LOG, -1, -1):  # Идём от старших битов к младшим
            if diff >= (1 << i):  # Если можем подняться на 2^i
                u = self.up[u][i]
                diff -= (1 << i)

        # Если после выравнивания u и v совпали - нашли LCA
        if u == v:
            return u

        # Поднимаем обе вершины одновременно, пока их родители не совпадут
        for i in range(self.LOG, -1, -1):
            if self.up[u][i] != self.up[v][i]:  # Пока предки разные -
поднимаемся
                u = self.up[u][i]
                v = self.up[v][i]

        # Теперь u и v - непосредственные дети LCA
        return self.up[u][0]


if __name__ == "__main__":
    n = 6
    edges = [(1, 2), (1, 3), (2, 4), (2, 5), (3, 6)]
```

```
lca_bl = LCABinaryLifting(n)
for u, v in edges:
    lca_bl.add_edge(u, v)
lca_bl.preprocess(1)

print(f"LCA(4, 5) = {lca_bl.lca(4, 5)}")  # 2
print(f"LCA(4, 6) = {lca_bl.lca(4, 6)}")  # 1
print(f"LCA(2, 2) = {lca_bl.lca(2, 2)}")  # 2
```

2.4. Анализ сложности

- **Память:** O(N log N) для хранения массива up.
- **Предобработка:** O(N log N) – один DFS.
- **Запрос LCA:** O(log N).

3. Сведение LCA к RMQ (Range Minimum Query)

3.1. Эйлеров обход и минимум глубины

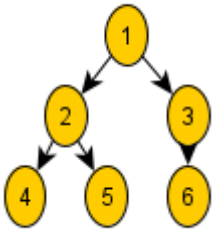
LCA можно свести к задаче поиска минимума на отрезке (RMQ) с помощью **эйлерова обхода** (Euler Tour).

- Выписываем вершины в порядке обхода в глубину.
- Запоминаем **глубину** каждой вершины в этом обходе.
- Запоминаем **первое вхождение** (first_occurrence) каждой вершины в эйлеровом обходе.

Тогда LCA(u, v) – это вершина с **наименьшей глубиной** на отрезке эйлерова обхода между first_occurrence[u] и first_occurrence[v].

Пример:

Дерево:



Эйлеров обход: [1, 2, 4, 2, 5, 2, 1, 3, 6, 3, 1]

Глубины: [0, 1, 2, 1, 2, 1, 0, 1, 2, 1, 0]

first_occurrence: [1:0, 2:1, 4:2, 5:4, 3:7, 6:8]

LCA(4, 5): отрезок в обходе от индекса 2 до 4 → [4, 2, 5]. Минимальная глубина = 1 у вершины 2.

3.2. Реализация сведения LCA к RMQ

```python
from collections import defaultdict


class LCAtoRMQ:
    """
    Класс для нахождения LCA через сведение к задаче RMQ.
    Использует эйлеров обход и разреженную таблицу.
    """

    def __init__(self, n: int):
        """Инициализация класса.

        Args:
            n: Количество вершин в дереве
        """
        self.n = n
        self.graph = defaultdict(list)  # Список смежности
        self.euler = []  # Эйлеров обход дерева
        self.depth = []  # Глубины вершин в порядке эйлерова обхода
        # first_occurrence[u] - первое вхождение вершины u в эйлеровом обходе
        self.first_occurrence = [-1] * (n + 1)

    def add_edge(self, u: int, v: int):
        """Добавление ребра в дерево."""
        self.graph[u].append(v)
        self.graph[v].append(u)

    def dfs(self, u: int, p: int, d: int):
        """DFS для построения эйлерова обхода.

        Args:
            u: Текущая вершина
            p: Родитель
            d: Текущая глубина
        """
        # Запоминаем первое вхождение вершины
        self.first_occurrence[u] = len(self.euler)
        # Добавляем вершину в эйлеров обход
        self.euler.append(u)
        self.depth.append(d)  # И её глубину

        # Рекурсивно обходим детей
        for v in self.graph[u]:
            if v != p:
                self.dfs(v, u, d + 1)  # Спускаемся в потомка
                # Возвращаясь, снова добавляем текущую вершину
                self.euler.append(u)
                self.depth.append(d)

    def preprocess(self, root: int):
        """Запуск предобработки."""
        self.dfs(root, -1, 0)  # Строим эйлеров обход
```

```
        self.build_sparse_table()  # Строим разреженную таблицу

    def build_sparse_table(self):
        """Построение разреженной таблицы для RMQ."""
        m = len(self.depth)  # Длина эйлерова обхода
        k = m.bit_length()  # Количество уровней в таблице

        # st[j][i] - индекс минимума на отрезке [i, i + 2^j - 1]
        self.st = [[0] * m for _ in range(k)]
        # log[i] = floor(log2(i)) - для быстрого определения степени
        self.log = [0] * (m + 1)

        # Предподсчёт логарифмов
        for i in range(2, m + 1):
            self.log[i] = self.log[i // 2] + 1

        # Базовый случай: отрезки длины 1
        for i in range(m):
            self.st[0][i] = i  # Минимум на отрезке [i, i] - сам элемент

        # Заполнение таблицы для больших длин
        for j in range(1, k):
            step = 1 << (j - 1)  # 2^(j-1)
            for i in range(m - (1 << j) + 1):
                # Берём минимум из двух половин
                left = self.st[j - 1][i]
                right = self.st[j - 1][i + step]
                # Сравниваем глубины, а не значения вершин
                if self.depth[left] < self.depth[right]:
                    self.st[j][i] = left
                else:
                    self.st[j][i] = right

    def rmq(self, l: int, r: int) -> int:
        """Запрос минимума на отрезке [l, r] в массиве depth.

        Returns:
            Индекс элемента с минимальной глубиной
        """
        length = r - l + 1
        k = self.log[length]  # Максимальная степень двойки <= length

        # Два перекрывающихся отрезка длины 2^k
        left_idx = self.st[k][l]
        right_idx = self.st[k][r - (1 << k) + 1]

        # Возвращаем индекс с меньшей глубиной
        if self.depth[left_idx] < self.depth[right_idx]:
            return left_idx
        else:
            return right_idx
```

```
    def lca(self, u: int, v: int) -> int:
        """Нахождение LCA через RMQ."""
        # Находим первые вхождения вершин в эйлеровом обходе
        l = self.first_occurrence[u]
        r = self.first_occurrence[v]

        # Гарантируем, что l <= r
        if l > r:
            l, r = r, l

        # Находим индекс вершины с минимальной глубиной между
first_occurrence[u] и first_occurrence[v]
        idx = self.rmq(l, r)

        # Возвращаем саму вершину из эйлерова обхода
        return self.euler[idx]


if __name__ == "__main__":
    n = 6
    edges = [(1, 2), (1, 3), (2, 4), (2, 5), (3, 6)]

    lca_rmq = LCAtoRMQ(n)
    for u, v in edges:
        lca_rmq.add_edge(u, v)
    lca_rmq.preprocess(1)

    print(f"LCA(4, 5) = {lca_rmq.lca(4, 5)}")  # 2
    print(f"LCA(4, 6) = {lca_rmq.lca(4, 6)}")  # 1
```

4. Алгоритм Фарах-Колтона и Бендера (RMQ ±1)

4.1. Идея алгоритма

Алгоритм Фарах-Колтона и Бендера решает **статический RMQ** за O(1) на запрос с предобработкой O(N), если разница между соседними элементами равна ±1 (как в массиве глубин эйлерова обхода).

Шаги алгоритма:

1. Разбиваем массив на блоки размера log(N) / 2.
2. Для каждого блока предподсчитываем минимум и **тип блока** (маску переходов ±1).
3. Для всех возможных типов блоков предподсчитываем ответы на все возможные запросы внутри блока.
4. Над минимумами блоков строим разреженную таблицу для ответа на запросы между блоками.

4.2. Реализация алгоритма Фарах-Колтона и Бендера

```
from collections import defaultdict
from typing import List, Dict


class RMQFarachColtonBender:
    """
    Реализация алгоритма Фарах-Колтона и Бендера для RMQ ±1.
    Обеспечивает O(1) время запроса с O(N) предобработкой.
    """

    def __init__(self, arr: List[int]):
        """Инициализация класса.

        Args:
            arr: Входной массив со свойством ±1 (разность соседних элементов
= ±1)
        """
        self.arr = arr
        self.n = len(arr)
        # Размер блока ~ log(n)/2
        self.block_size = max(1, self.n.bit_length() // 2)
        self.block_count = (self.n + self.block_size - 1) // self.block_size

        # Минимумы для каждого блока
        self.block_min = [10 ** 9] * self.block_count
        # Битовые маски для типов блоков (кодируют последовательность ±1)
        self.block_mask = [0] * self.block_count
        self.log = [0] * (self.block_count + 1)

        self.precompute_blocks()
        self.build_sparse_table()

    def precompute_blocks(self):
        """Предподсчёт минимумов и масок для каждого блока."""
        for i in range(self.block_count):
            start = i * self.block_size
            end = min(start + self.block_size, self.n)
            min_val = 10 ** 9

            # Находим минимум в блоке
            for j in range(start, end):
                if self.arr[j] < min_val:
                    min_val = self.arr[j]
            self.block_min[i] = min_val

            # Строим битовую маску для последовательности ±1 в блоке
            mask = 0
            for j in range(start + 1, end):
                # Если рост значения - ставим 1, иначе 0
                if self.arr[j] > self.arr[j - 1]:
                    mask |= (1 << (j - start - 1))
            self.block_mask[i] = mask
```

```
def build_sparse_table(self):
    """Построение разреженной таблицы над минимумами блоков."""
    k = self.block_count.bit_length()
    self.st = [[0] * self.block_count for _ in range(k)]

    # Предподсчёт логарифмов
    for i in range(2, self.block_count + 1):
        self.log[i] = self.log[i // 2] + 1

    # Базовый случай - сами блоки
    for i in range(self.block_count):
        self.st[0][i] = i  # Индекс блока с минимальным значением

    # Заполнение таблицы
    for j in range(1, k):
        step = 1 << (j - 1)
        for i in range(self.block_count - (1 << j) + 1):
            left = self.st[j - 1][i]
            right = self.st[j - 1][i + step]
            # Выбираем блок с меньшим минимумом
            if self.block_min[left] < self.block_min[right]:
                self.st[j][i] = left
            else:
                self.st[j][i] = right

def query_block(self, block_idx: int, l: int, r: int) -> int:
    """Запрос минимума внутри одного блока.

    Args:
        block_idx: Индекс блока
        l: Левый край запроса (глобальный индекс)
        r: Правый край запроса (глобальный индекс)

    Returns:
        Индекс минимального элемента в блоке на отрезке [l, r]
    """
    start = block_idx * self.block_size
    end = min(start + self.block_size, self.n)

    # Простой перебор - так как блоки маленькие, это O(log n)
    min_val = 10 ** 9
    min_idx = l
    for i in range(l, r + 1):
        if self.arr[i] < min_val:
            min_val = self.arr[i]
            min_idx = i
    return min_idx

def rmq(self, l: int, r: int) -> int:
    """Основной метод запроса RMQ.

    Args:
        l: Левый край отрезка
        r: Правый край отрезка
```

```
        Returns:
            Индекс минимального элемента на отрезке [l, r]
        """
        # Определяем блоки, в которые попадают концы отрезка
        block_l = l // self.block_size
        block_r = r // self.block_size

        min_idx = l  # Начинаем с левого конца

        # Случай 1: запрос полностью внутри одного блока
        if block_l == block_r:
            return self.query_block(block_l, l, r)

        # Случай 2: запрос покрывает несколько блоков

        # Обрабатываем левый частичный блок
        left_end = min((block_l + 1) * self.block_size - 1, self.n - 1)
        left_min_idx = self.query_block(block_l, l, left_end)
        if self.arr[left_min_idx] < self.arr[min_idx]:
            min_idx = left_min_idx

        # Обрабатываем правый частичный блок
        right_start = block_r * self.block_size
        right_min_idx = self.query_block(block_r, right_start, r)
        if self.arr[right_min_idx] < self.arr[min_idx]:
            min_idx = right_min_idx

        # Обрабатываем полные блоки между ними
        if block_l + 1 <= block_r - 1:
            len_blocks = (block_r - 1) - (block_l + 1) + 1
            k = self.log[len_blocks]

            # Находим минимальный блок среди средних
            left_block = self.st[k][block_l + 1]
            right_block = self.st[k][block_r - 1 - (1 << k) + 1]
            block_min_idx = left_block if self.block_min[left_block] <
self.block_min[right_block] else right_block

            # Находим конкретный элемент в минимальном блоке
            start = block_min_idx * self.block_size
            end = min(start + self.block_size, self.n)
            for i in range(start, end):
                if self.arr[i] == self.block_min[block_min_idx]:
                    if self.arr[i] < self.arr[min_idx]:
                        min_idx = i
                    break

        return min_idx


if __name__ == "__main__":
    arr = [0, 1, 2, 1, 2, 1, 0, 1, 2, 1, 0]  # Глубины из эйлерова обхода
    rmq_fcb = RMQFarachColtonBender(arr)
```

```
idx = rmq_fcb.rmq(2, 4)  # Индексы в эйлеровом обходе
print(arr[idx])  # 1 (минимум на отрезке)
```

5. Сравнение методов

Метод	**Предобработка**	**Запрос**	**Память**	**Примечания**
Бинарные подъёмы	O(N log N)	O(log N)	O(N log N)	Универсальный, простой
LCA → RMQ (Sparse Table)	O(N log N)	O(1)	O(N log N)	Быстрый запрос
Фарах-Колтон и Бендер	O(N)	O(1)	O(N)	Оптимально по памяти и времени