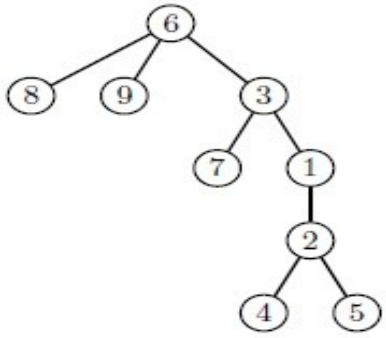
# *Конспект лекции LCA*

1. Определение дерева и его свойства
Дерево — неориентированный связный граф без циклов
Если в дереве n вершин, то в нем n-1 ребер
Дерево называется подвешенным за какую-то вершину, если мы выбрали эту вершину и ориентировали ребра от нее. Эта вершина называется корнем




В подвешенном дереве в каждую вершину (кроме корня) входит ровно одно ребро
Если есть ребро из вершины v в вершину u, то v — родитель u. 3 — родитель 7 и 1.
Возьмем любую вершину (не корень) и «отрубим» то ребро, которое входит в нее. Тогда то дерево, которое «подвешено» на эту вершину — поддерево этой вершины. В нашем случае поддерево вершины 1 — это вершины 2, 4, 5
Уровнем вершины называется расстояние от нее до корня по ребрам. Так, вершина 6 находится на 0 уровне, вершина 3 — на 1 уровне, вершина 7 — на 3 уровне

2. Как хранить дерево?
Удобнее всего в виде списков смежности

```
const int MAXN = 1e6; //максимальный размер дерева для данной задачи
vector<int> tree[MAXN]; //массив списков смежности

void add_egde(int a, int b) { //функция для добавления нового ребра из a в b
	tree[a].push_back(b);
	tree[b].push_back(a);
}
```

3. Как подвесить дерево?
Выбираем корень и запускаем из него обход в глубину или в ширину. В обходе удалим все ребра, ориентированные к корню.

```
void delete_edge(int v, int id) { //функция удаления ребра за O(1)
	swap(tree[v][id], tree[v].back());
	tree[v].pop_back();
}

void dfs(int v, int p) { //обход в глубину от текущей вершины и ее родителя. Для корня номер родителя равен -1
	for (int i = 0; i < int(tree[v].size()); ++i) { //перебираем ребра из v
		int u = tree[v][i]; //номер вершины, в которую ведет ребро
		if (u == p) { //если u - родитель v, то удаляем ребро
			delete_edge(v, i);
			--i; //так как на место i в функции мы поставили какую-то другую вершину, то мы должны еще раз посмотреть tree[v][i]
```

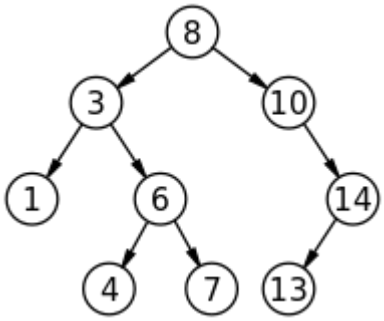
```
            }
            else {
                  dfs(u, v); //v - родитель u
            }
      }
}
```

Однако, можно и не удалять ребра, а просто пропускать их при каждом обходе. Однако, это придется делать и в каждом следующем обходе, и такой подход не позволяет воспользоваться одним из алгоритмов, о котором будет сказано позже.

```
void dfs(int v, int p) {
      for (int u: tree[v]) { //данная форма написания цикла for доступна с C++11. Если не знаете, как его подключить, обратитесь ко мне
            if (u == p) continue;
            dfs(u, v);
      }
}
```

4. Как посчитать уровни?
Заведем глобальный массив `int level[MAXN];`
В dfs будем передавать еще один параметр — текущий уровень (d). Как только мы пришли в вершину, запишем в массив значение d. При переходе к новой вершине запускаемся от d+1

5. Предок
Вершина a называется предком для вершины b в подвешенном ориентированном дереве, если из a достижимо b
Заметим, что корень — предок для всех вершин. Также вершина является предком сама для себя. В этом примере 10 — предок 14 и 13. Иначе говоря, a — предок b, если b лежит в поддереве a




6. Времена входа и выхода.
Запустим обход в глубину, как показано в пункте 3. Создадим два глобальных массива

```
int tin[MAXN];
int tout[MAXN];
```

Также сделаем глобальный счетчик текущего времени

```
int t = 0;
```

Теперь при входе в вершину запишем значение t в tin[v], затем увеличим t на 1. После обхода всего поддерева запишем значение t в tout[v] и увеличим t на 1

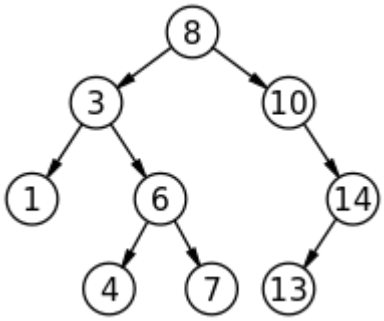
Полученные значения tin[v] и tout[v] называются временами входа и выхода в вершину v и обладают многими полезными свойствами (например, с помощью них можно искать мосты в графе, но это выходит за рамки нашей лекции)
Здесь мы рассмотрим только одно из этих свойств. Но прежде, чем мы это сделаем обобщим то, что мы уже можем сделать с помощью dfs

```
функция dfs(цел v, цел p, цел d):
    level[v] := d
    tin[v] := t
    t := t + 1
    цикл u : ребра из v:
        если u = p:
```

```
            продолжить цикл
        dfs(u, v, d + 1)
    tout[v] := t
    t := t + 1
```

7. Необходимое и достаточное условие, чтобы утверждать, что a — предок b
Утверждение без доказательства (теорема 1):
a предок b тогда и только тогда, когда tin[a] <= tin[b] <= tout[b] <= tout[a]

Убедимся, что это так:




| Вершина | tin | tout |
|---|---|---|
| 8 | 0 | 17 |
| 3 | 1 | 10 |
| 1 | 2 | 3 |
| 6 | 4 | 9 |
| 4 | 5 | 6 |
| 7 | 7 | 8 |
| 10 | 11 | 16 |
| 14 | 12 | 15 |
| 13 | 13 | 14 |

Легко увидеть, что Теорема 1 выполняется для всех вершин

8. Наименьший общий предок.
Наименьшим общим предком (Lowest Common Ancestor) (далее — LCA) для двух вершин a и b называется вершина v такая, что v — предок для a и для b на наибольшем уровне (то есть самая низкая из таких, что в ее поддереве лежат a и b.
LCA(4, 7) = 6
LCA(1, 3) = 3
LCA(7, 1) = 3
LCA(1, 10) = 8

Свойство:
Единственный кратчайший путь по дереву из вершины a в вершину b можно представить в следующем виде
1) Подняться до LCA(a, b)
2) Спуститься до b

Таким образом, умение быстро находить LCA крайне важно для решения задач на деревьях. Так мы сможем быстро найти, к примеру, стоимость маршрута между двумя вершинами дерева, минимальное ребро на этом пути и многое другое

9. Алгоритм «в лоб»
Во время обхода dfs-ом для каждой вершины сохраним её предка в массив parent
Поступил запрос a, b

Версия 1.

```
Пока a и b не на одном уровне:
    Если level[a] < level[b]:
        поменять(a, b)
    a := parent[a]
Пока a не равно b:
    a := parent[a]
    b := parent[b]
```

```
Вернуть a
```

Версия 2.
В этом алгоритме для определения того, что вершина a — предок b, пользуемся Теоремой 1

```
Пока a не предок b:
    a := parent[a]
Вернуть a
```

Оценим эти алгоритмы
Очевидно, что оба алгоритма работают за O(h) (h — высота дерева, то есть его максимальный уровень)
h <= n
Для простоты скажем, что оба алгоритма работают за O(n). Эта оценка достигается, если взять «бамбук» (дерево в виде прямой линии), «согнуть» его пополам и запуститься из двух нижних вершин. Тогда оба алгоритма сделают n/2 шагов.
По памяти оба алгоритма так же тратят O(n)

Такое время ответа на запрос допустимо на случайно сгенерированных деревьях (из высота приблизительно log n) и когда для ответа на задачу нам надо отвечать не на много запросов (менее 100).
В других случаях, мы не будем укладываться в ограничения по времени. Значит, надо учиться ускорять время ответа на запрос, чтобы не подниматься по всему дереву
На самом деле, оба эти алгоритма можно значительно ускорить, оставив при этом ту же идею

10. Предподсчёт. Двоичные подъемы (binary lifting)
Слабое место нашего алгоритма — мы много поднимаемся на 1, хотя было бы здорово уметь «прыгнуть» сразу на несколько вершин вверх. Разумеется, мы не можем сохранить для каждой вершины всех её предков (это заняло бы слишком много времени и памяти). Зато мы можем сохранить, какие вершины выше нас на 2^k (k = 0, 1, … log n) уровней (правда, пока непонятно как и зачем). Это займет всего лишь O(n log n) памяти, что вполне нас устраивает.

Такой прием называется «двоичные подъемы»

Идея:
Пусть мы посчитали двоичные подъемы для всех вершин выше текущей. Посчитаем для текущей.
На 1 вверх ходить мы умеем (это переход в родителя)
Как перейти на 2? Можно перейти на 1 вверх, а там мы уже знаем, кто выше этой вершины еще на 1
Как перейти на 4? Мы уже умеем переходить на 2. Перейдем туда. Там мы, опять же, умеем переходить на 2 (двоичные подъемы для той вершины уже построены).
На 8? Перейти на 4 и опять на 4! И так далее…

Реализация:
Для удобства будем считать, что дерево подвешено за вершину 0, и что если перейти из корня на 2^k вверх, мы снова попадем в корень. Это никак не меняет алгоритм, зато заметно упрощает реализацию

```
const int MAXN = 1e5;
const int MAXLOG = 21;

int up[MAXN][MAXLOG];

int logn;
```

```
void add(int a, int b) {
	up[a][0] = 0;
	for (int i = 1; i <= logn; ++i) {
		up[a][i] = up[up[a][i - 1]][i - 1];
	}
}
```

Как же посчитать logn? Можно это сделать динамикой или бинарным поиском, но зачем? Мы знаем максимальный возможный размер дерева в конкретной задаче. Давайте сделаем logn константой и возьмем «с запасом». 2^20 > 10^6, значит logn = 20 будет вполне достаточно.

Когда запускать эту функцию? В dfs, при просмотре ребра, но перед тем, как запустить dfs от другой вершины этого ребра.

11. Как теперь посмотреть, кто выше нас на k?
Пусть мы стоим в вершине a. Тогда будет работать следующий алгоритм:

```
int jump(int a, int k) {
	for (int i = 0; i <= logn; ++i) {
		if ((k & (1 << i))) {
			a = up[a][i];
		}
	}
	return a;
}
```

Наверное, if заставил вас испугаться, но здесь все очень просто. Так мы определяем значение i-го бита в числе k. Если он равен 1, то мы «прыгаем» на 2^i. Если вы хотите понять, почему работает этот if, то обратитесь ко мне после лекции.

Таким образом, за O(log n) мы научились переходить сразу на k вершин наверх

12. Простой алгоритм нахождения LCA
Мы уже умеем смотреть, является ли вершина a предком b
Тогда мы можем бинарным поиском найти наивысшую вершину c такую, что c — предок a, но не предок b. LCA(a, b) = parent[c], то есть LCA(a, b) = up[c][0]
Заметим, что если a предок b, то такой вершины c не будет.
Но тогда LCA(a, b) = a. Обработаем этот случай отдельно

Реализация:

```
int lca(int a, int b) {
	if (is_ancestor(a, b)) {
		return a;
	}
	int L = 0;
	int R = n + 1;
	int M;
	while (L + 1 != R) {
		M = (L + R) / 2;
		int v = jump(a, M);
		if (is_ancestor(v, b)) {
			R = M;
		}
		else {
			L = M;
		}
	}
	return up[jump(a, R)][0];
}
```

Этот алгоритм работает за O(log^2 n) на запрос. То есть, он позволяет отвечать на очень большое количество запросов в рамках одного теста. Но, оказывается, и его можно значительно упростить и ускорить.
13. Стандартный алгоритм нахождения LCA

Здесь мы разберем только модификацию 2-го алгоритма из пункта 9, но иногда бывает полезно уметь писать и быструю версию 1-го алгоритма из пункта 9 (в контесте есть задача, где этот способ значительно проще). Попробуйте написать её сами или обратитесь ко мне.

Будем делать другую версию бинарного поиска. Идем вниз по степеням двойки. Если мы поднимемся вверх на 2^i, то мы попадем в вершину v. Если v — предок b, то не будем подниматься вверх, а иначе поднимемся.
Утверждается, что мы попадем как раз в вершину с из пункта 12 (попробуйте доказать это сами).

Алгоритм получился очень простым:

```
int lca(int a, int b) {
	if (is_ancestor(a, b)) {
		return a;
	}
	for (int i = logn; i >= 0; --i) {
		int v = up[a][i];
		if (!is_ancestor(v, b)) {
			a = v;
		}
	}
	return up[a][0];
}
```

15. Как посчитать что-то на пути?
Мы можем посчитать какую-либо функцию на пути вверх от вершины на степень двойки одновременно с двоичными подъемами. Затем найдем LCA(a, b) = e. В jump будем возвращать результат этой функции при подъеме до LCA. Обозначим функцию как #. Тогда функция на пути будет равна:
jump(a, level[a] — level[e]) # jump(b, level[b] — level[e])

16. Оценка и сравнение алгоритмов
Обозначим за m количество запросов. Тогда и 1 и 2 модификация алгоритмов из пункта 9 работают за O(n log n + m log n) = O((n + m) log n)
При этом они требуют O(n log n) памяти
При размере дерева 10^5 мы можем ответить примерно на 10^6 запросов!
Чем отличаются модификации алгоритмов 1 и 2 из пункта 9?
Алгоритм первый
Поднимаемся до одного уровня, затем одновременно поднимаемся вверх
Алгоритм не требует подсчета времен входа и выхода
Алгоритм второй
Поднимаемся от одной вершины наверх, смотрим, являемся ли мы предком
Алгоритм требует подсчета времен входа и выхода, но имеет очень мальнькую константу

Таким образом, если вы можете посчитать времена входа и выхода — лучше писать второй алгоритм. Он работает примерно в 2 раза быстрее на больших тестах.
Если такой возможности нет, или это затруднительно, то удобнее будет написать первый алгоритм.

17. Промежуточные итоги

| | Предподсчет | Время на запрос | Память | Умеем считать функцию |
|---|---|---|---|---|
| «Лобовой» | n | n | n | Да |
| Простые подъемы | n log n | log^2 n | n log n | Да |
| Стандартные подъемы | n log n | log n | n log n | Да |

18. Эйлеров обход и его свойства
Существует и более быстрый алгоритм для нахождения LCA, но он не позволяет считать некоторые функции (например, минимум)
Для того, чтобы понять его, нужно ввести определение Эйлерова обхода дерева
Эйлеровым обходом дерева называется обход по следующему правилу:

добавить вершину в обход
Для каждого сына:
  запустить обход из сына
  добавить вершину в обход

Таким образом, мы получили дерево в виде массива чисел:
для каждых двух соседних вершин в массиве есть ребро между ними.

А теперь сделаем следующий фокус: будем добавлять в обход не просто номер вершины, а пару <высота, номер>

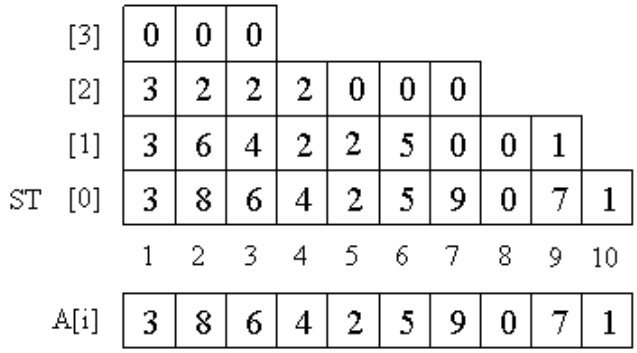
Возьмем две различные вершины a и b. Найдем любые их вхождения в обход. Посмотрим на отрезок между этими вхождениями
Теорема 2 (без доказательства):
Найдем на этом отрезке минимум. Пусть это пара <d, c>.
Утверждается, что LCA(a, b) = c

Таким образом, мы свели задачу LCA к другой известной задаче RMQ (Range Minimum Query, минимум на отрезке)

19. Решение RMQ
Вы уже знаете решение этой задачи за O(log n) на запрос с предподсчетом за O(n). Такая асимптотика достигается при использовании Дерева Отрезков (которое обсуждалось на прошлой сессии), Декартова Дерева, Дерева Фенвика или другой похожей структуры данных. Сейчас мы будем обсуждать решение этой задачи за O(1) на запрос и предподсчетом за O(n log n). Стоит отметить, что мы решаем задачу RMQ±1, а она имеет решение за O(1) на запрос и предподсчетом за O(n). То есть идеальный по асимптотике алгоритм. Но в рамках этой лекции он рассмотрен не будет. Желающие могут ознакомиться с ним по ссылке:
http://e-maxx.ru/algo/lca_linear
Кстати, любая задача RMQ за O(n) сводится к RMQ±1, то есть она решаема за O(1) на запрос и O(n) на предподсчет в онлайне. Но это уже совсем другая история.

20. Sparse Table (разрешенная таблица)
Эта структура данных позволяет вычислять функции вида «максимум», «минимум» и другие (строго говоря, обладающие свойствами ассоциативности, коммутативности и идемпотентности).

Описание структуры:
Структура представляет из себя двумерный массив пар.
Элемент на позиции i, j будет равен минимуму на промежутке [j; j + 2^i). Следующий слой можно построить из предыдущего, а всего достаточно log n + 1 слоёв.
Таким образом, первый слой представляет собой исходный массив, а каждый следующий «динамически» пересчитывается из предыдущего.




Реализация построения:
Примечание: (1 « i) = 2^i. Это операция побитового сдвига

```
pair<int, int> sparse[MAXLOG][MAXN];
vector<pair<int, int> > rmq_arr;

void build_sparse() {
	int m = rmq_arr.size();
	for (int j = 0; j < n; ++j) {
		sparse[0][j] = rmq_arr[j];
	}
	for (int i = 1; i <= logn; ++i) {
		for (int j = 0; j + (1 << i) <= m; ++j) {
			sparse[i][j] = min(sparse[i - 1][j], sparse[i - 1][j + (1 << (i - 1))]);
		}
	}
}
```

21. Как теперь найти минимум на отрезке?
Для начала, зададим другой вопрос: как найти границы отрезка, то есть как найти индексы вхождения вершин a и b в обход? На самом деле, это можно сделать жадно. Например, пройтись по обходу и для каждой вершины сохранить ее первое вхождение.
Теперь, пусть L и R — границы отрезка. Найдем его длину. len = R — L + 1. Найдем логарифм двоичный от этой длины (пока не важно, каким образом).
x = log2(len)
Тогда возьмем два элемента sparse[x][L] и sparse[x][R-2^x+1]. Они пересекаются и затрагивают концы отрезка [L; R], таким образом если мы возьмем минимум от этих двух элементов, это и будет ответ

22. Как найти логарифм числа?
У нас все числа маленькие (не более 2 * n). Значит, мы можем предподсчитать логарифмы динамически:

```
int log2[MAXN];

int main() {
	log2[1] = 0;
	for (int i = 2; i < MAXN; ++i) {
		log2[i] = log2[i / 2] + 1;
	}
}
```

23. Сравнение стандартных алгоритмов нахождения LCA

| Название | Предподсчет | Запрос | Память | Онлайн | Рассмотрен в лекции |
|---|---|---|---|---|---|
| Двоичные подъемы | O(n log n) | O(log n) | O(n log n) | да | да |
| Sparse Table | O(n log n) | O(1) | O(n log n) | да | да |
| ФКБ | O(n) | O(1) | O(n) | да | нет |
| Тарьяна | O(n) | O(α(n)) | O(n) | нет | нет |

24. Трюки на деревьях

Трюк 1

В какого из сыновей надо пойти из вершины a, чтобы попасть в вершину b (a — предок b)?
Можно посчитать двоичные подъемы и подняться из b на level[b] — level[a] — 1. Тогда получим O(n log n) предподсчет и O(log n) на запрос
Но можно посчитать только времена входа-выхода и сделать бинпоиск по времени входа-выхода у детей a
Отмечу, что этот прием работает только если мы удалим ребро в родителя, иначе оно нам все портит

Трюк 2

Какая будет высота дерева (или какая-нибудь другая похожая функция), если подвесить его за вершину v?
Это можно легко посчитать в обходе для всех вершин одновременно. Глубину будем считать рекурсивно, а при переходе говорить, что у нас уже есть поддерево с глубиной (Текущая_максимальная_глубина + 1), если мы не идем в это поддерево или (Вторая_максимальная_глубина + 1), если мы идем в поддерево с максимальной глубиной
То есть нам нужно найти первый и второй максимум, а это делается за линейное время.

Трюк 3

Если веса написаны на ребрах, то нам не обязательно считать сумму на пути во время нахождения LCA. Мы можем в dfs сохранить сумму от корня до текущей вершины. Тогда ответ на запрос «найти сумму на пути от a до b» будет равен: sum[a] + sum[b] — 2 * sum[LCA(a, b)]
Этот прием позволяет искать сумму на пути без двоичных подъемов

Трюк 4

Операция xor (исключающее или) обладает следующими свойствами:
a xor a = 0
0 xor a = a
a xor 0 = a
Опираясь на пункт 3, попробуйте догадаться, как найти xor на пути очень простым способом. Задача на это есть на контесте

25. Где почитать про нахождение LCA?

1. http://e-maxx.ru/algo/lca (здесь показано сведение к RMQ и нахождение минимума с помощью ДО)
2. http://e-maxx.ru/algo/lca_simpler (двоичные подъемы, но код плохой)
3. https://neerc.ifmo.ru/wiki/index.php?title=Метод_двоичного_подъема (двоичные подъемы)
4. http://e-maxx.ru/algo/lca_linear_offline (алгоритм Тарьяна. Не был расскзан на лекции. Обладает почти идеальным временем работы, но на запросы нельзя отвечать «онйлайн». То есть мы сначала получаем все запросы, а лишь потом на них отвечаем)

26. Какие задачи на LCA можно порешать?
http://codeforces.com/problemset/problem/813/C
http://acm.timus.ru/problem.aspx?space=1&num=2109