

# Алгоритмы и структуры данных

Лекция 11

Графы - 1

Кандауров Геннадий



education

# Напоминание отметиться на портале

+ ОСТАВИТЬ ОТЗЫВ

vk образование

БлогиЛюдиПрограммаВакансииРасписание

Q<

VK

Техно

Открыт приём заявок!

чт, 8 сентября

Нет занятий

пт, 9 сентября

18:00 Углубленный Py... с3  
Введение в Python, основные  
понятия, тестирование  
Г. Кандауров

сб, 10 сентября

Нет занятий

вс, 11 сентября

Нет занятий

пн, 12 сентября

Нет занятий

Углубленный Python

↓ 0 ↑

Блог для материалов по курсу "Углубленный Python"

57 читателей, 2 топика

ПодписатьсяСоздать топик

Поиск по авторам, заголовку и тексту топика...

Найти

Добро пожаловать на курс!

Углубленный Python

ИзменитьУдалить

Всем привет и добро пожаловать на курс по углубленному изучению Python!

Прямой эфир

МоиВсе

Геннадий Кандауров час назад  
Углубленный Python → Добро пожаловать  
на курс! 0

Екатерина Черкасова 7 дней назад  
Стажировка → Приглашаем мобильных,  
фронтенд- и бэкэнд-разработчиков на  
Weekend Offer! 0

Дарья Вовченко 9 дней назад  
Углубленный Python → Добро пожаловать  
в образовательные проекты VK  
Образование! 0

Дарья Вовченко 9 дней назад  
Разработка веб-сервисов на

## Квиз про прошлой лекции



# Содержание занятия

1. Граф
2. Виды графов
3. Представление графов
4. Обход графа
5. Алгоритм Дейкстры

# Граф



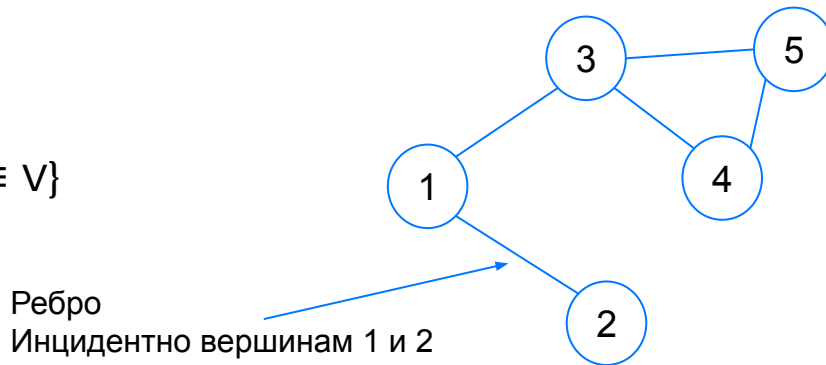
# Граф

**Граф** - совокупность непустого множества вершин  $V$  и множества ребер  $E$ .

$$G = (V, E)$$

$$n = |V|, m = |E|$$

$$V = \{1, 2, \dots, n\}, E \subset \{\{v, u\}, v, u \in V\}$$

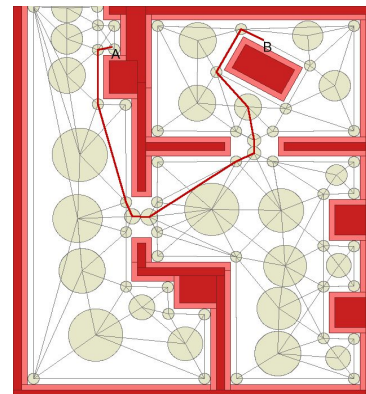
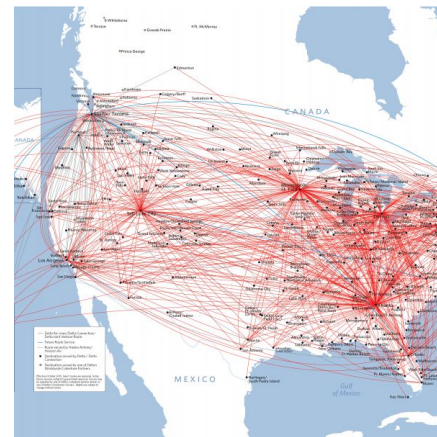
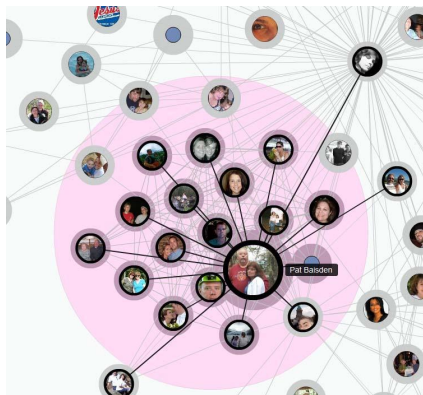
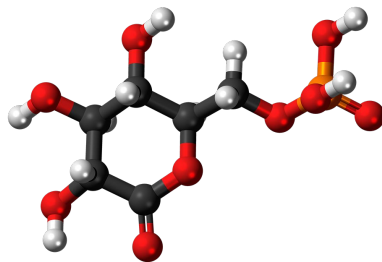


$$V = \{1, 2, 3, 4, 5\}$$

$$E = \{(1, 2), (1, 3), (3, 4), (3, 5), (4, 5)\}, \text{ 1 и 2 - смежные вершины}$$

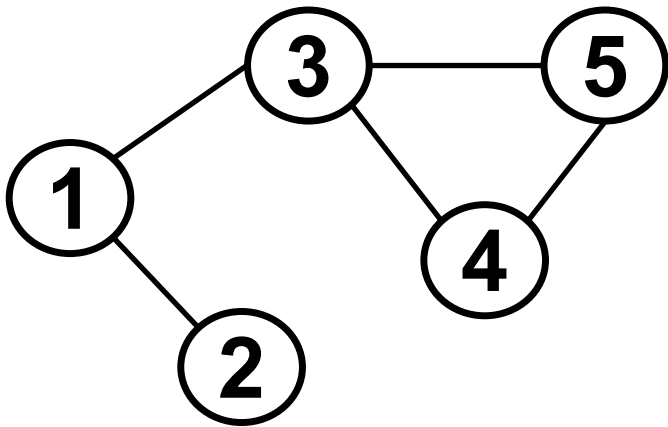
# Граф

- Страницы в интернете с ссылками
- Дороги
- Самолетные маршруты
- Друзья в соцсетях
- Генеалогическое древо
- Химические элементы
- Зависимости в исходниках
- Сетка перемещений в играх
- Печатные платы



# Виды графов

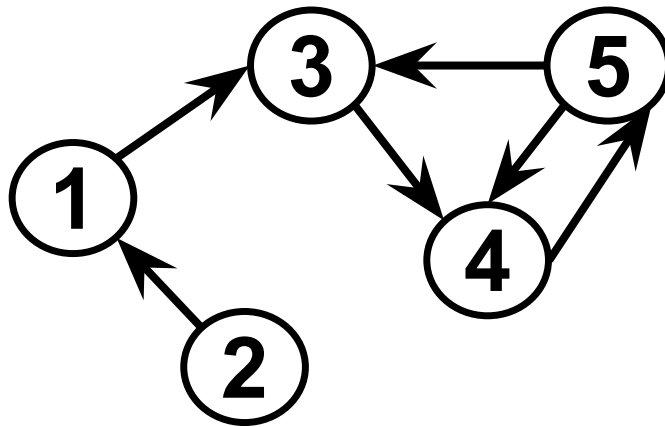
## Неориентированные графы



$$G = (V, E), \text{ где } E \subset \{\{v, u\}, v, u \in V\}$$

Ребра не имеют направлений  
(4,5) и (5, 4) – одно и то же ребро

## Оrientированные графы (орграфы)



$$G = (V, E), \text{ где } E \subset V \times V$$

Ребра (дуги) имеют направления  
(4,5) и (5, 4) – разные дуги



# Мультиграф, псевдограф

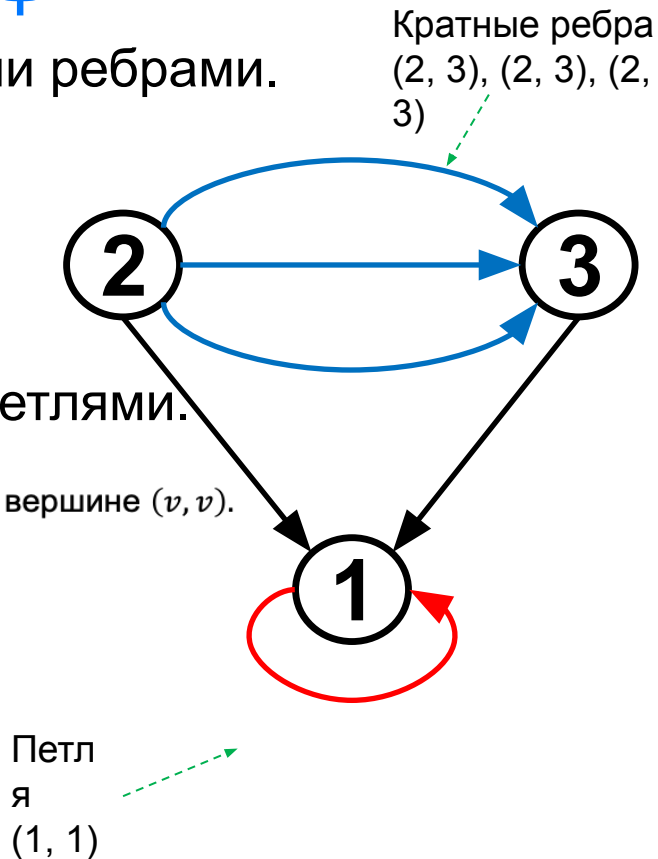
**Мультиграф** — граф с кратными ребрами.

**Кратные рёбра** (параллельные рёбра, мультирёбра) — это два и более **рёбер**, инцидентных одним и тем же двум вершинам.

**Псевдограф** — мультиграф с петлями.

**Петля** — это ребро, инцидентное одной и той же вершине  $(v, v)$ .

**Простой граф** — граф, в котором нет кратных ребер и петель.



# Ориентированный граф

Определение.  $G = (V, E)$ , где  $E \subset V \times V$ , называется *ориентированным графом*.

$V$  – множество *вершин*,

$E$  – множество *ребер*.

Определение. *Петля* – ребро  $(v, v)$ .

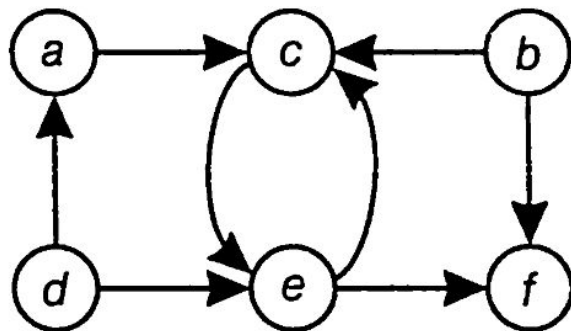
Определение. Пусть  $(v, u)$  – ребро.

Тогда  $v$  – *предок*  $u$ ,

$v$  и  $u$  – *смежные*,

$v$  *инцидентна*  $(v, u)$ ,

$u$  *инцидентна*  $(v, u)$ .



# Неориентированный граф

Определение.  $G = (V, E)$ , где  $E \subset \{\{v, u\}, v, u \in V\}$ ,  
называется *неориентированным графом*.

$V$  – множество *вершин*,

$E$  – множество *ребер*.

Определение.  $G = (V, E, ends)$ ,

где  $V, E$  – множества,

$ends: E \rightarrow \{\{v, u\}, v, u \in V\}$ ,

называется *неориентированным псевдографом*.

# Степень вершины

**Степень вершины**  $\deg v$  – число ребер, инцидентных  $v$ ,  
причем петля добавляет степень 2.

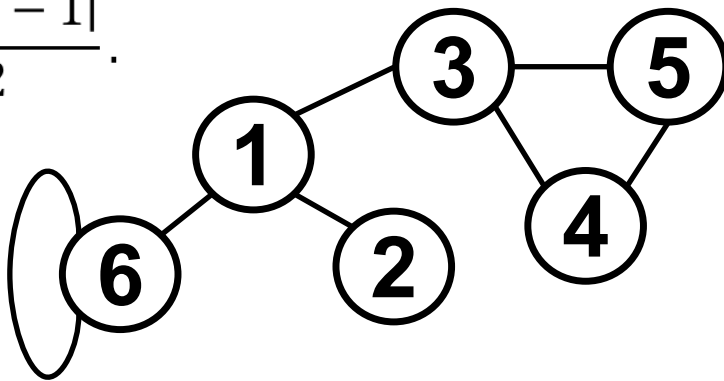
Лемма (о рукопожатиях).  $\sum_{v \in V} \deg v = 2|E|$ .

Доказательство. Индукция по числу ребер.

Следствие 1. Число вершин нечетной степени – четно.

Следствие 2. Число ребер в полном графе равно

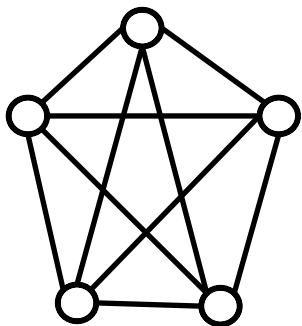
$$\frac{|V||V-1|}{2}.$$



# Регулярный и полный граф

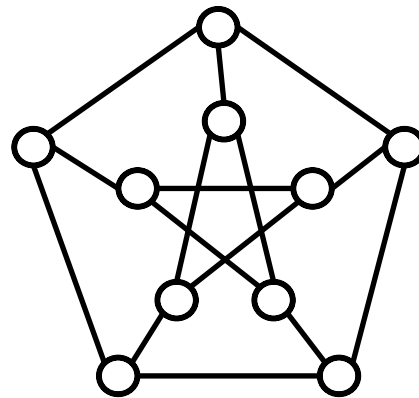
**Регулярный граф** – граф, в котором степени всех его вершин равны.

В таком графе  $|E| = \frac{k|V|}{2}$ .



**Полный граф** – граф, в котором каждая пара вершин смежна (все вершины соединены со всеми).

В таком графе  $|E| = \frac{|V||V-1|}{2}$

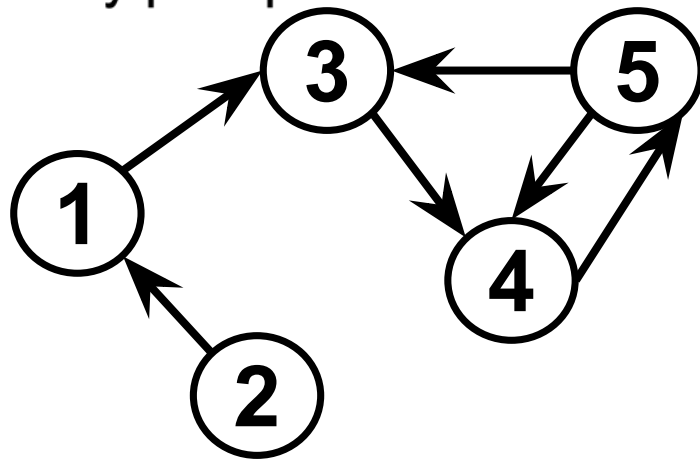


## Граф: представление в памяти

**Матрица смежности** – это матрица  $n \times n$  элементов, в которой значение  $a_{ij}$  равно количеству рёбер из  $i$ -й вершины графа в  $j$ -ю вершину.

$$A = \begin{pmatrix} 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 1 & 0 \end{pmatrix}$$

i



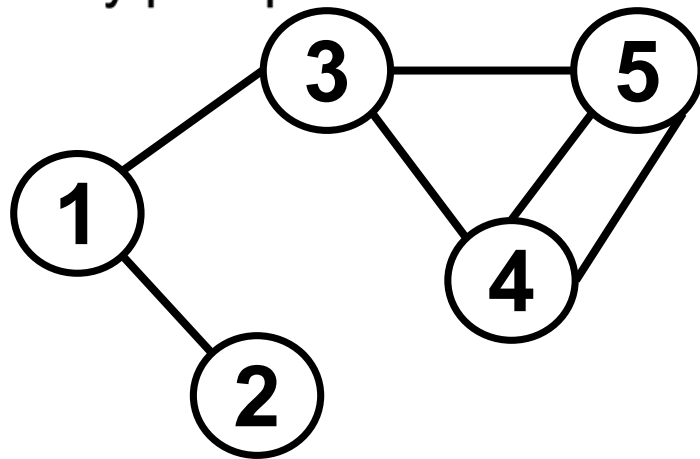
- Требует  $O(|V|^2)$  памяти
- Определение наличия ребра в графе за  $O(1)$
- Эффективна для хранения насыщенных графов ( $|E| = O(|V|^2)$ )

## Граф: представление в памяти

**Матрица смежности** – это матрица  $n \times n$  элементов, в которой значение  $a_{ij}$  равно количеству рёбер из  $i$ -й вершины графа в  $j$ -ю вершину.

$$A = \begin{pmatrix} 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 2 \\ 0 & 0 & 1 & 2 & 0 \end{pmatrix}$$

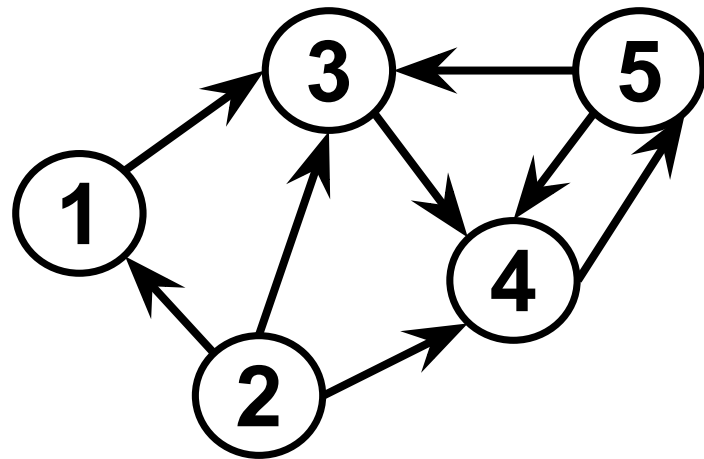
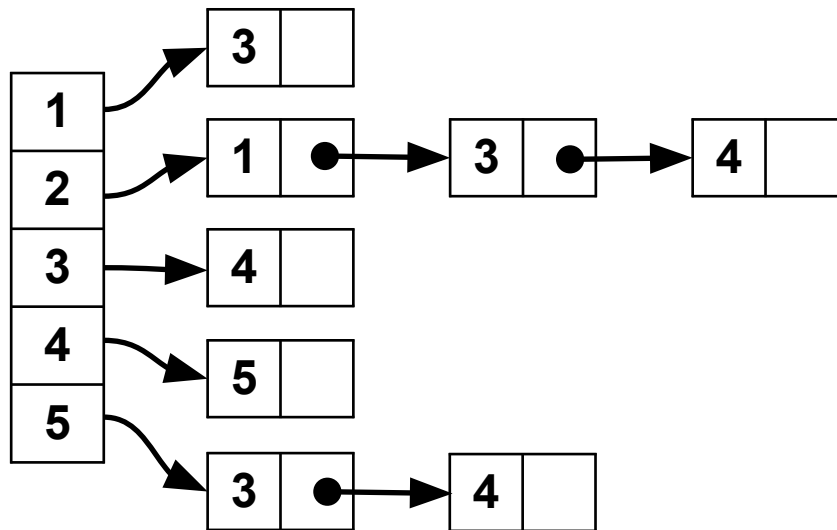
i



- Требует  $O(|V|^2)$  памяти
- Определение наличия ребра в графе за  $O(1)$
- Получение списка смежных вершин за  $O(|V|)$
- Эффективна для хранения насыщенных графов ( $|E| = O(|V|^2)$ )

# Граф: представление в памяти

**Списки смежных вершин** – для каждой вершины хранится список смежных с ней вершин.



- Требует  $O(|V| + |E|)$  памяти
- Определение наличия ребра в графе за  $O(|E|)$
- Получение списка смежных вершин за  $O(|E|)$
- Эффективна для хранения разреженных графов ( $|E| = O(|V|)$ )



# Граф: пути и циклы

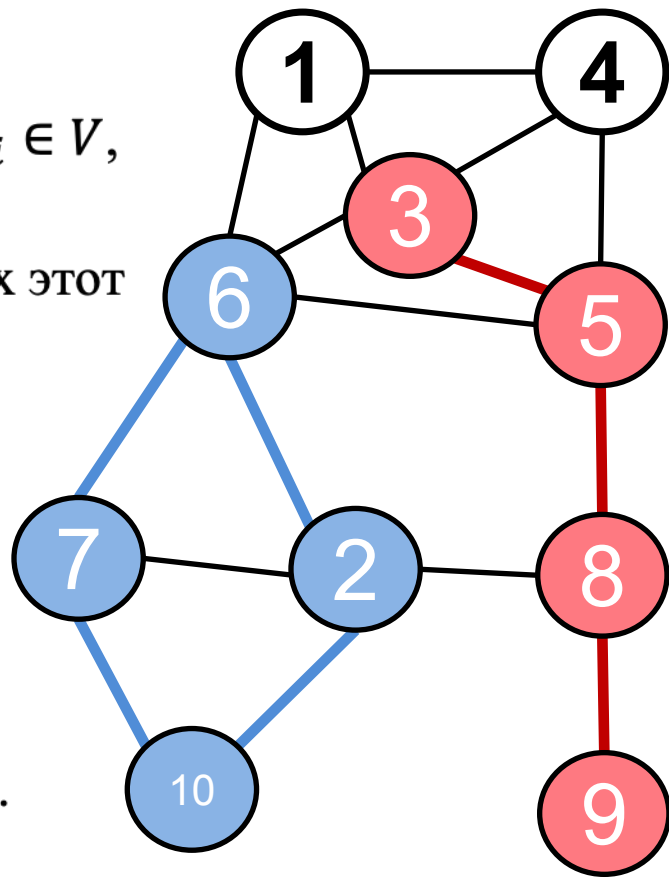
**Путь** – последовательность

$v_0, e_1, v_1, e_2, v_2, \dots, e_k, v_k$ , где  $e_i \in E$ ,  $v_i \in V$ ,  
 $e_i = (v_{i-1}, v_i)$ ,  $k$  – длина пути.

**Длина пути** – кол-во рёбер, задающих этот путь

**Циклический путь (цикл)** в ориентированном графе – путь, в котором  $v_0 = v_k$ .

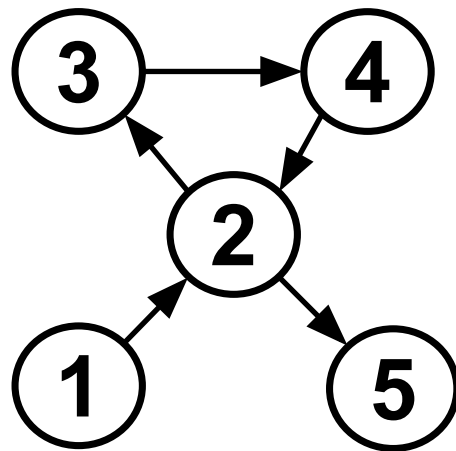
**Циклический путь (цикл)** в неориентированном графе – путь, в котором  $v_0 = v_k$  и  $e_i \neq e_{i+1}$ ,  $e_1 \neq e_k$ .



## Граф: простые пути и циклы

**Простой (вершинно-простой) путь** – путь, в котором каждая из вершин встречается не более одного раза.

Путь (1, 2, 3, 4, 2, 5) не является вершинно-простым.



**Реберно-простой путь** – путь, в котором каждое ребро встречается не более одного раза.

Путь (1, 2, 3, 4, 2, 3, 4, 2, 5) не является реберно-простым.

# Граф: связность

Вершины  $u$  и  $v$  в неориентированном графе **связны**, если существует путь  $u \rightsquigarrow v$ .

Теорема. Связность – отношение эквивалентности.

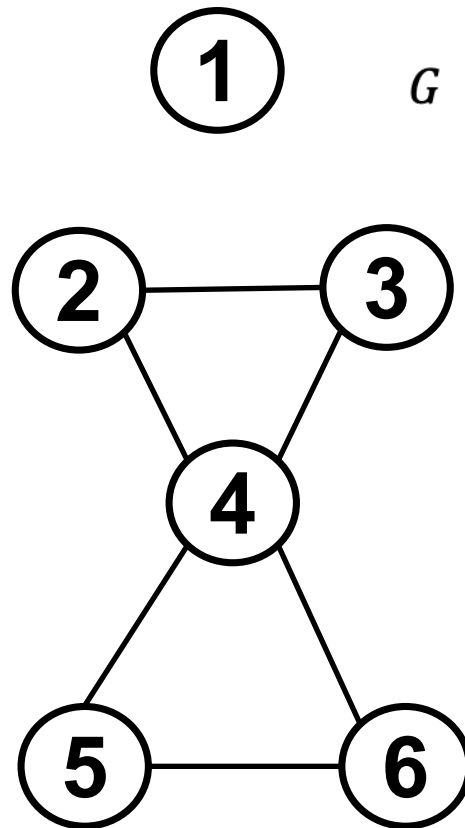
Доказательство. Надо доказать:

Рефлексивность.  $\forall u \in V u \rightsquigarrow u$

Симметричность.  $u \rightsquigarrow v \Rightarrow v \rightsquigarrow u$

Транзитивность.  $v \rightsquigarrow a \wedge a \rightsquigarrow u \Rightarrow v \rightsquigarrow u$

**Компонента связности** – класс эквивалентности отношения СВЯЗНОСТИ.



$V = \{1, 2, 3, 4, 5, 6\}$

$E = \{(2, 4), (2, 3), (4, 3), (4, 5), (5, 6), (6, 4)\}$

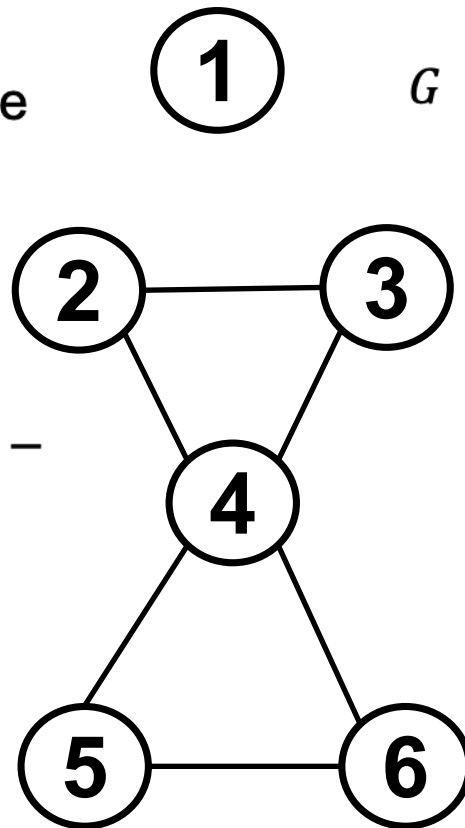
# Граф: сильная связность

Вершины  $u$  и  $v$  в ориентированном графе  $G$  **сильно связны**, если существуют пути  $u \rightsquigarrow v$  и  $v \rightsquigarrow u$ .

**Сильная связность** – отношение эквивалентности.

**Компонента сильной связности (КСС)** – класс эквивалентности отношения сильной связности.

Сколько компонент сильной связности в примере?



$V = \{1, 2, 3, 4, 5, 6\}$

$E = \{(2, 4), (2, 3), (4, 3), (4, 5), (5, 6), (6, 4)\}$

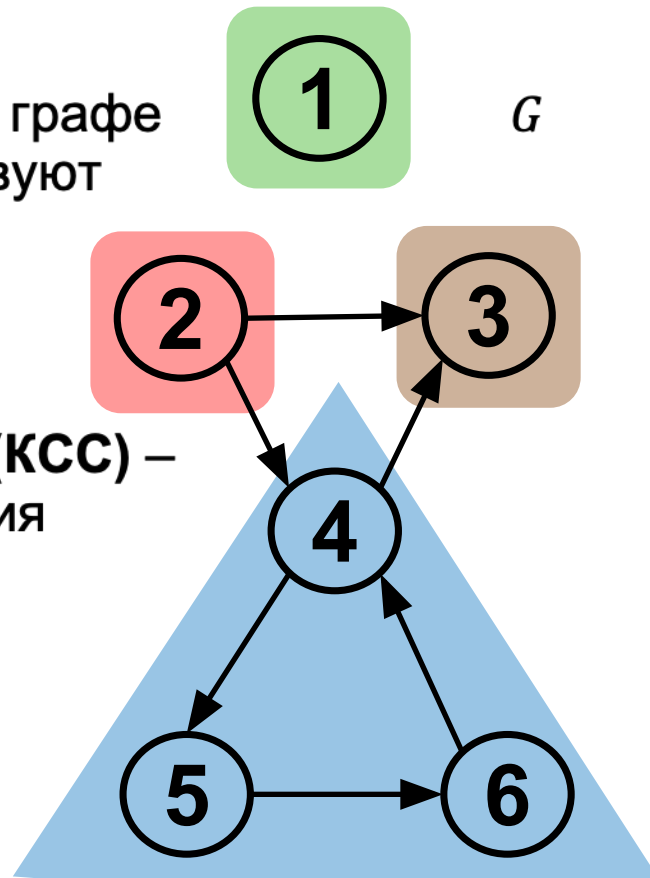
# Граф: сильная связность

Вершины  $u$  и  $v$  в ориентированном графе  $G$  **сильно связны**, если существуют пути  $u \rightsquigarrow v$  и  $v \rightsquigarrow u$ .

**Сильная связность** – отношение эквивалентности.

**Компонента сильной связности (КСС)** – класс эквивалентности отношения сильной связности.

Сколько компонент сильной связности в примере?



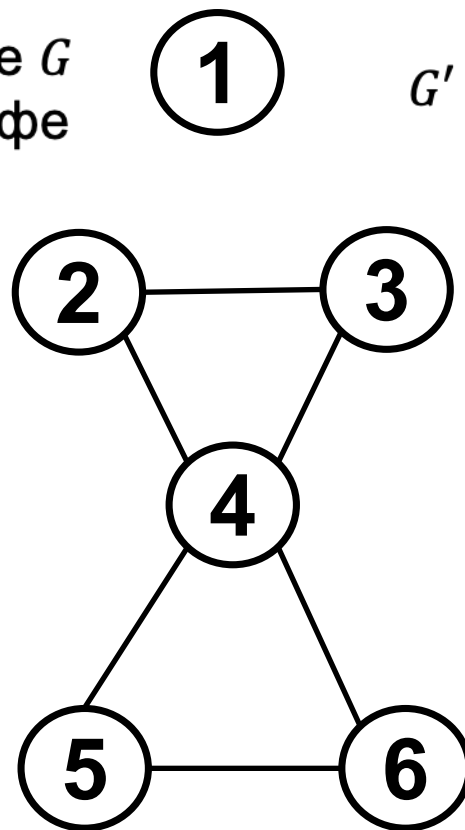
# Граф: слабая связность

Вершины  $u$  и  $v$  в ориентированном графе  $G$  **слабо связны**, если они связны в графе  $G'$ , полученном из  $G$  удалением ориентации ребер и повторяющихся ребер.

**Слабая связность** – отношение эквивалентности.

**Компонента слабой связности** – класс эквивалентности отношения слабой связности.

Сколько компонент слабой связности в примере?

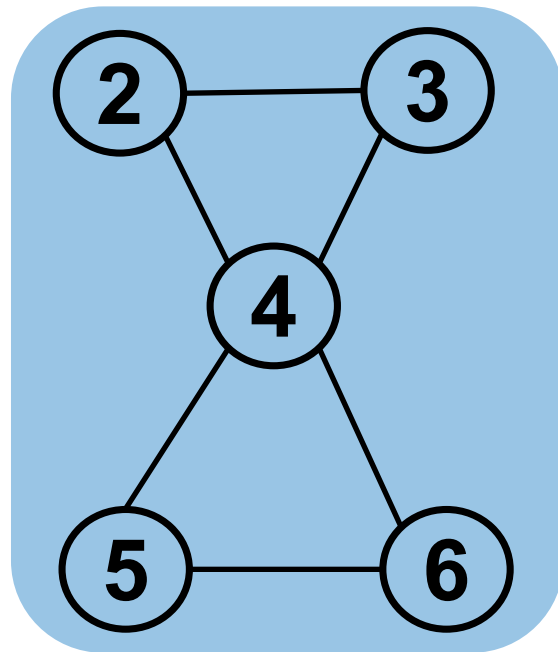


# Граф: слабая связность

Вершины  $u$  и  $v$  в ориентированном графе  $G$  **слабо связны**, если они связны в графе  $G'$ , полученном из  $G$  удалением ориентации ребер и повторяющихся ребер.

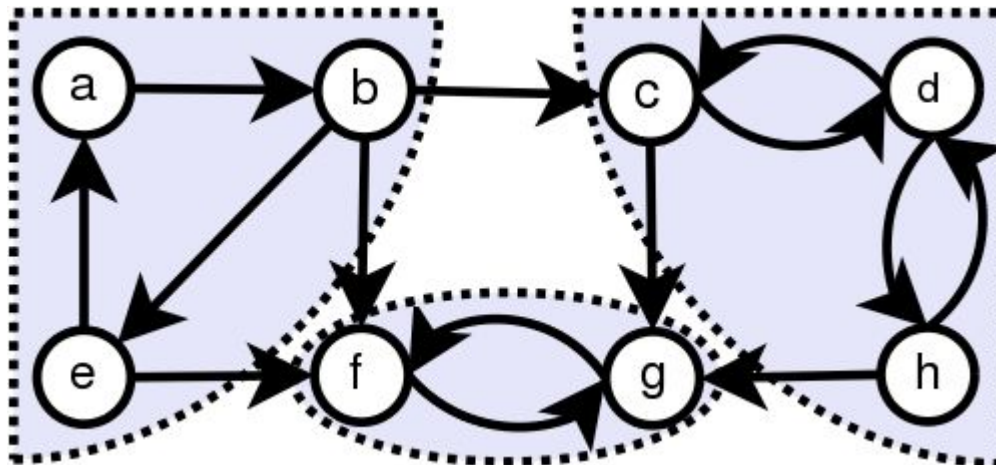
**Слабая связность** — отношение эквивалентности.

**Компонента слабой связности** — класс эквивалентности отношения слабой связности.



## Граф: КСС

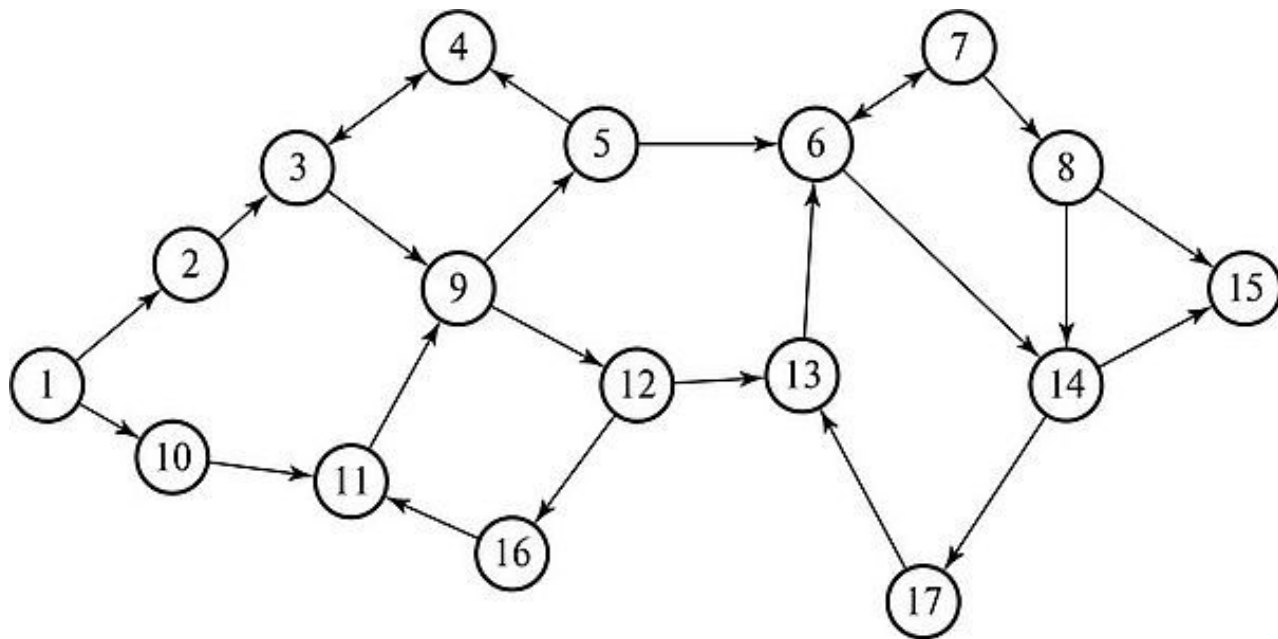
Интересно искать сильно связанные компоненты графа.





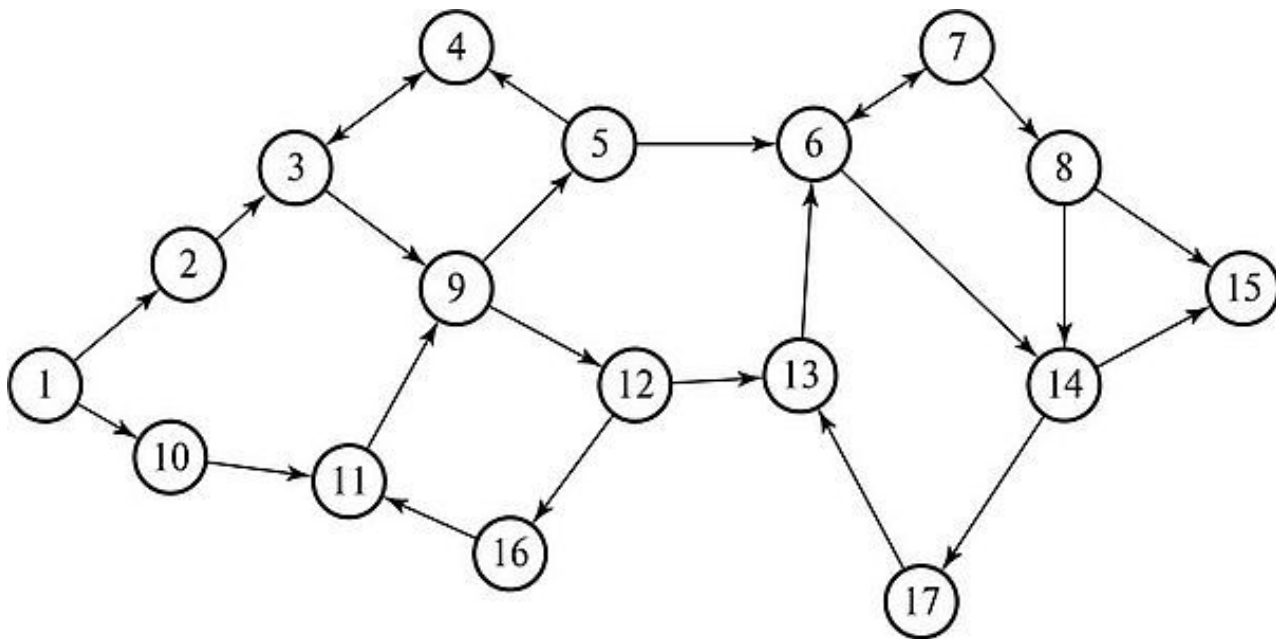
## Граф: КСС

Сколько КСС в этом графе?



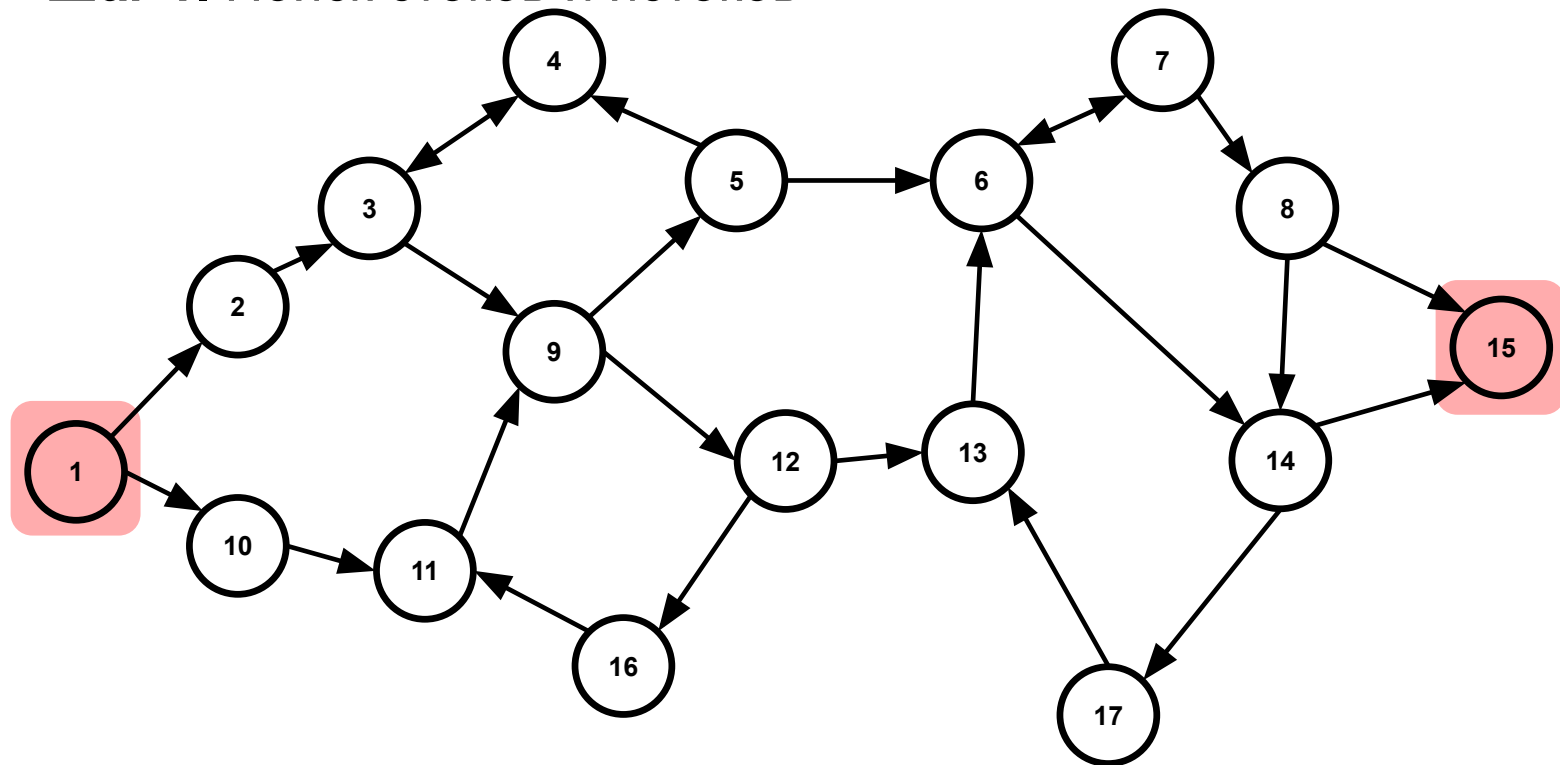
# Граф: КСС

## Шаг 1. Поиск стоков и истоков



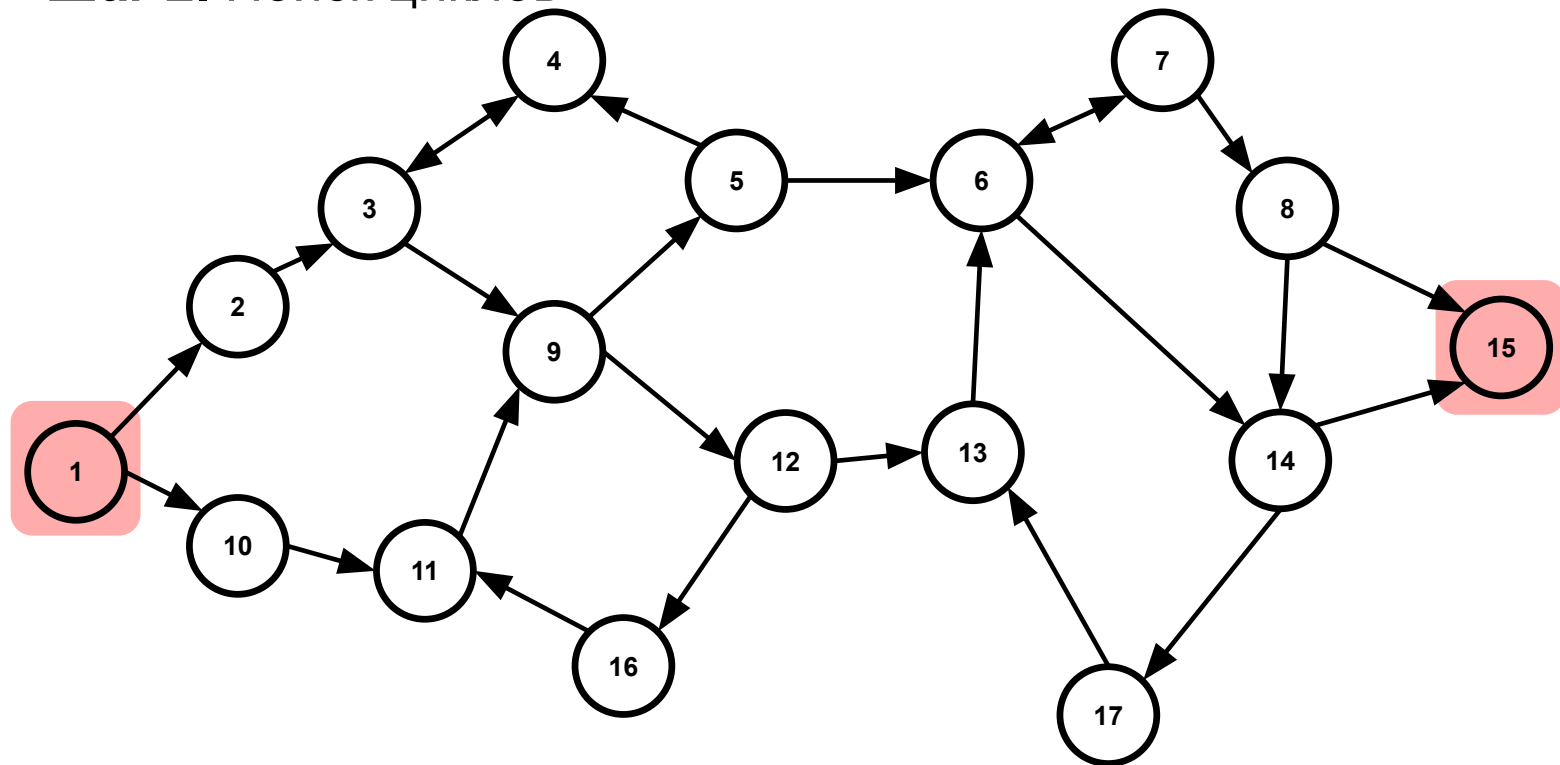
# Граф: КСС

## Шаг 1. Поиск стоков и истоков



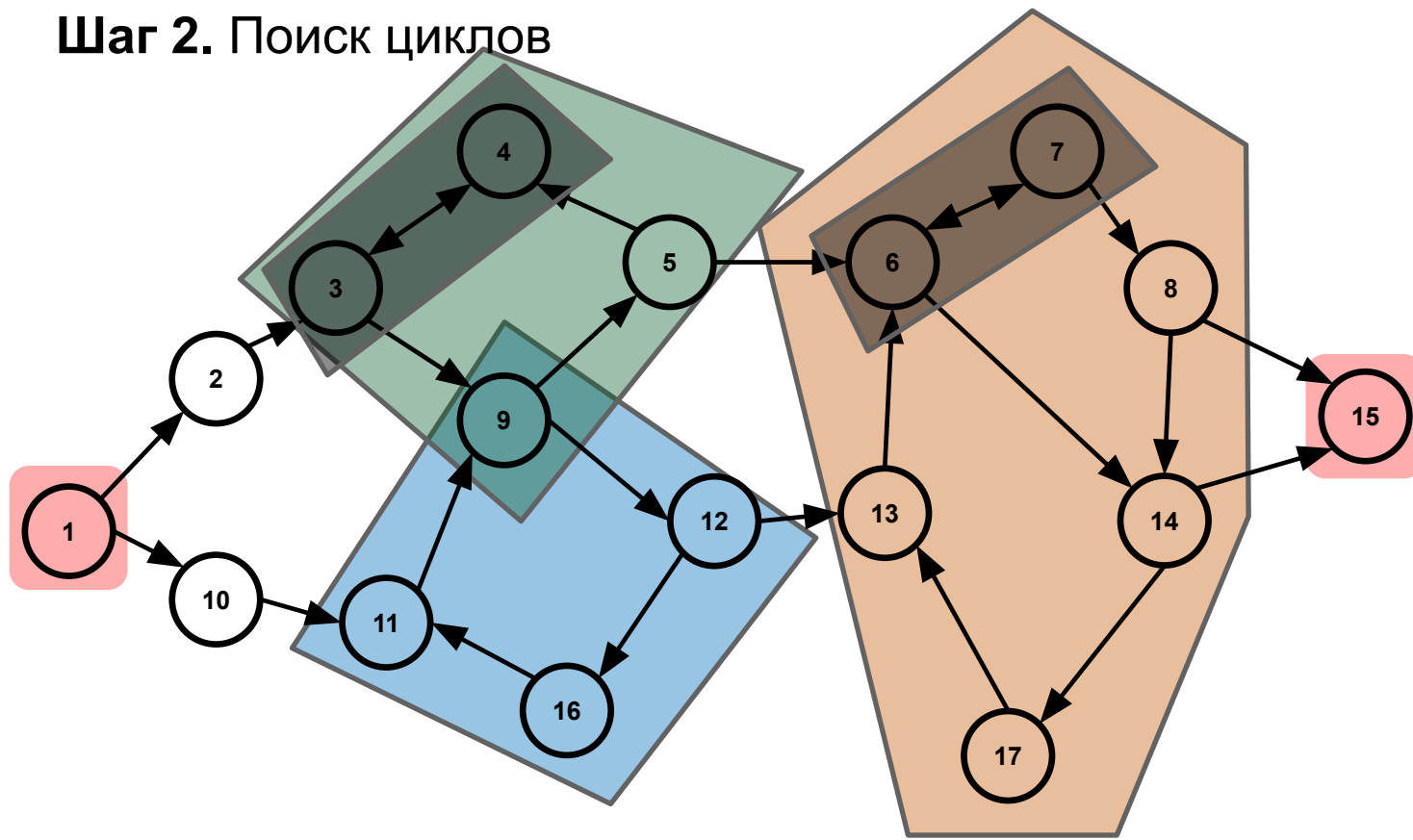
# Граф: КСС

## Шаг 2. Поиск циклов



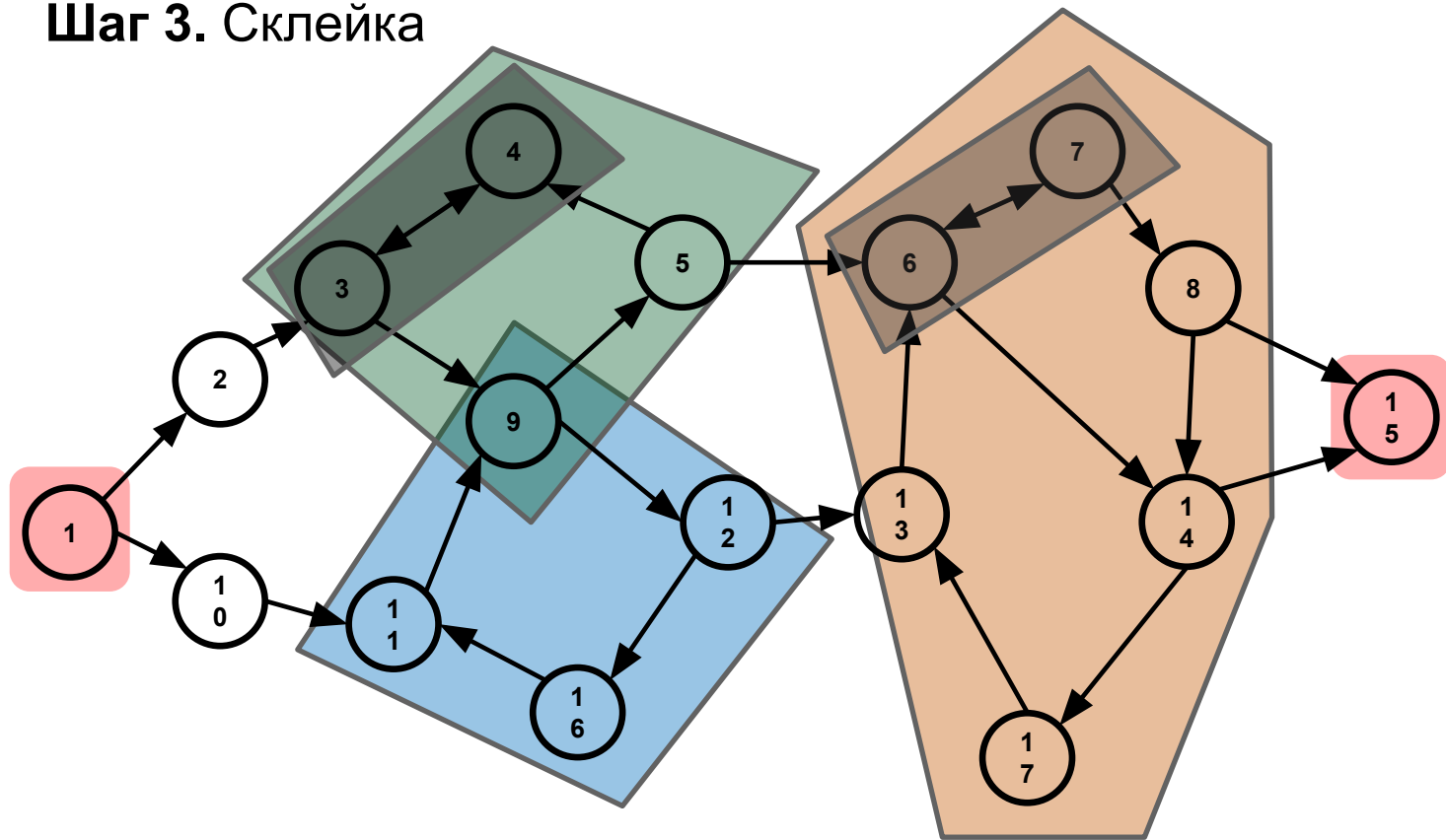
# Граф: КСС

## Шаг 2. Поиск циклов



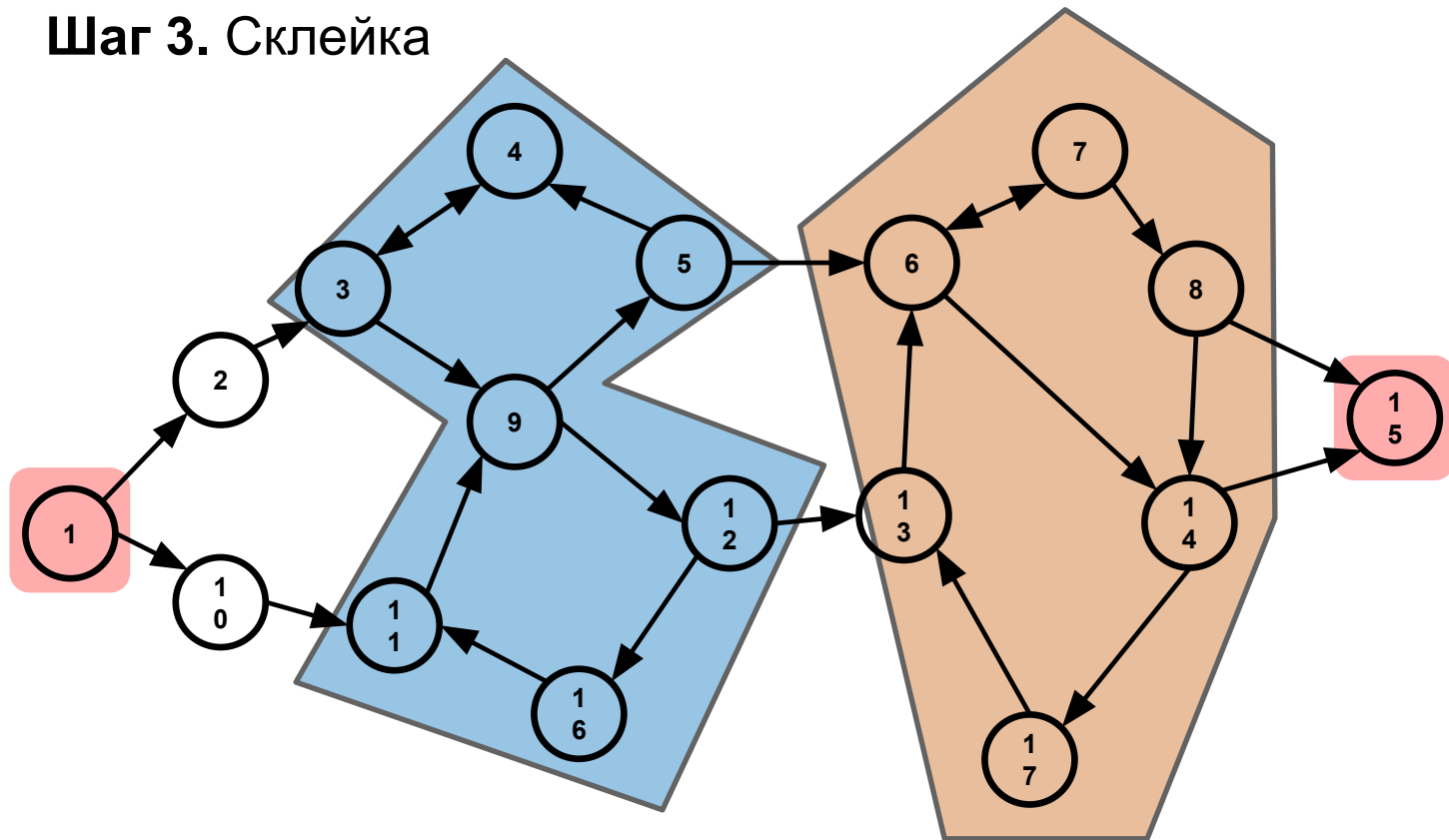
# Граф: КСС

## Шаг 3. Склейка



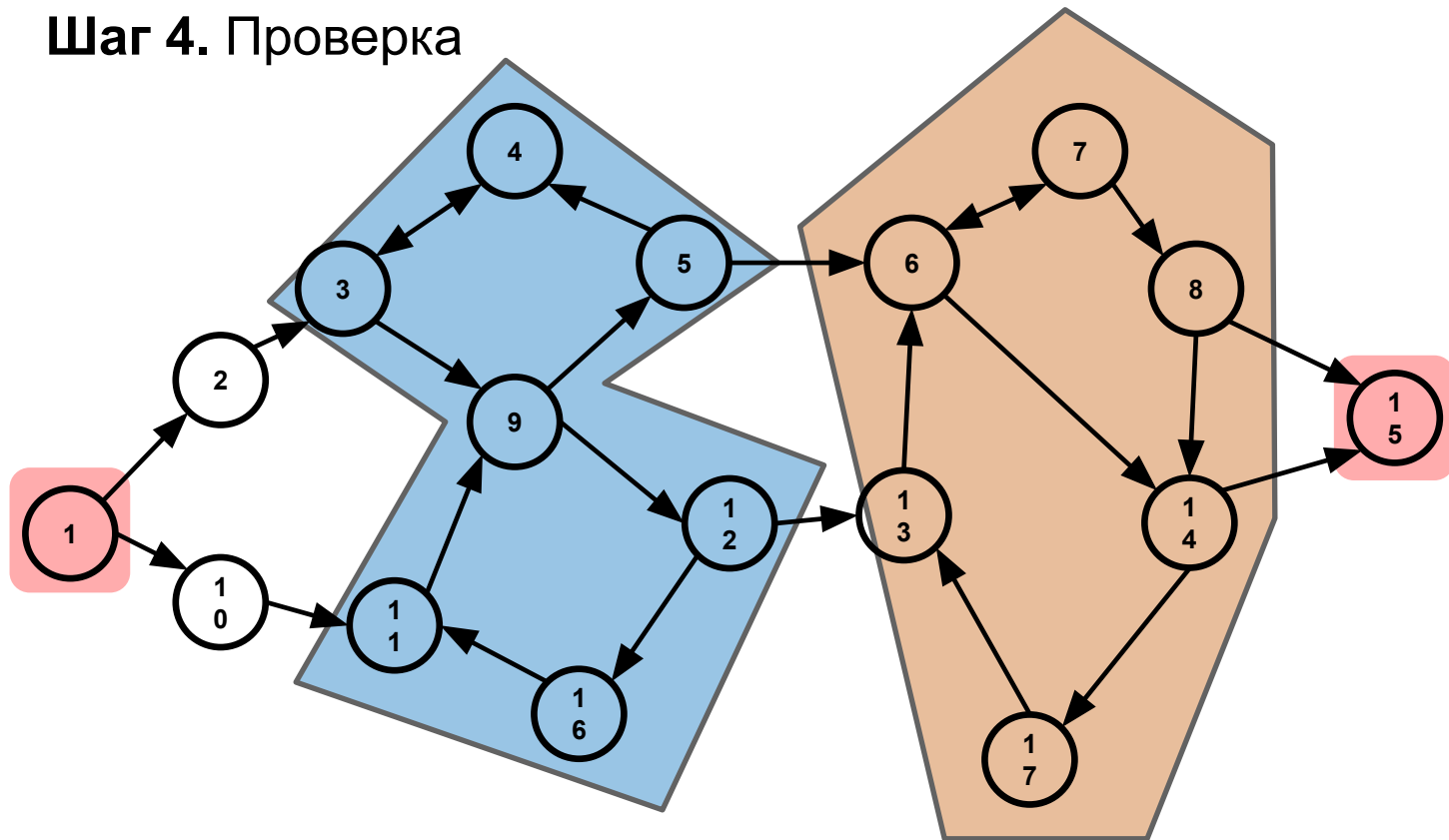
# Граф: КСС

## Шаг 3. Склейка



# Граф: КСС

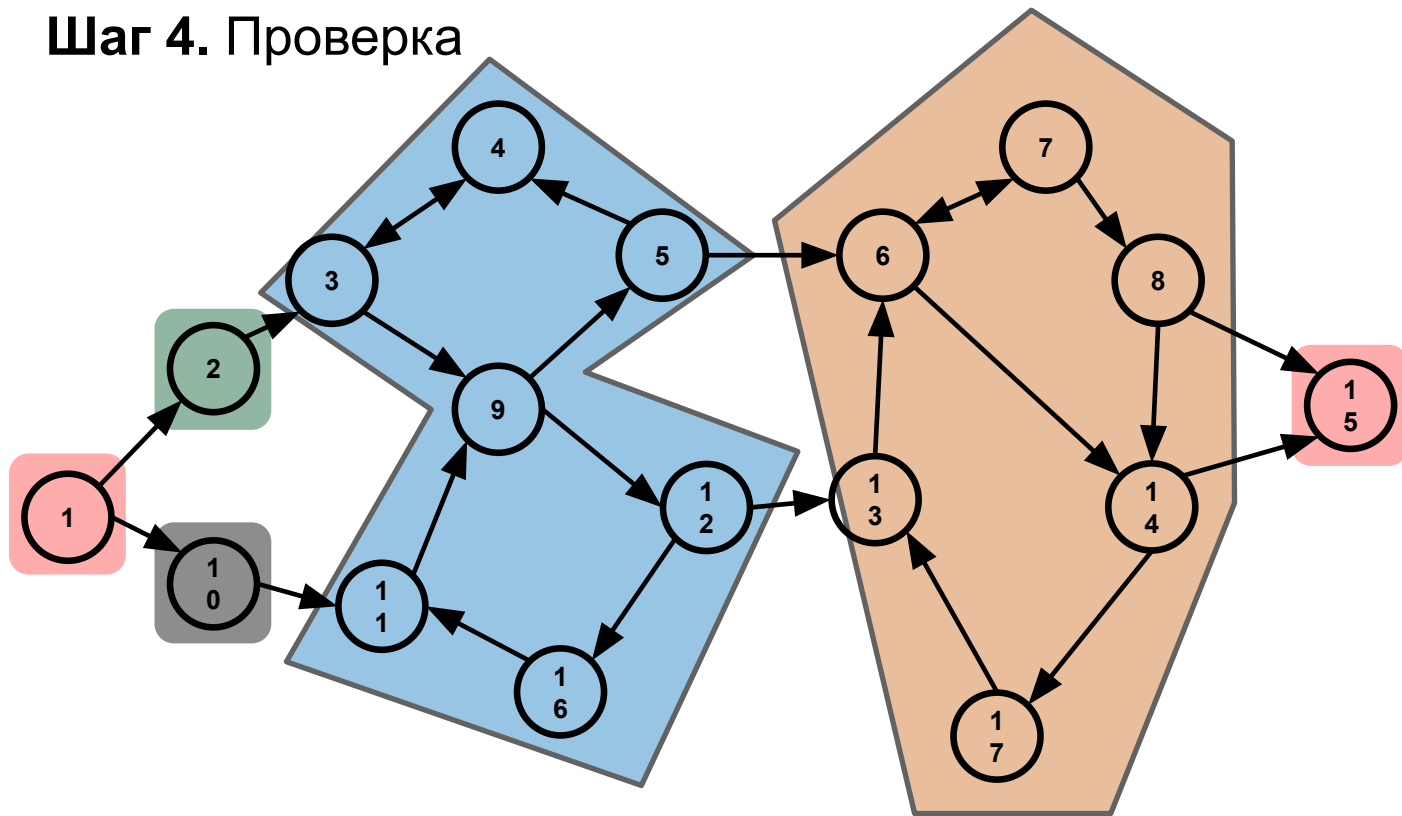
## Шаг 4. Проверка





# Граф: КСС

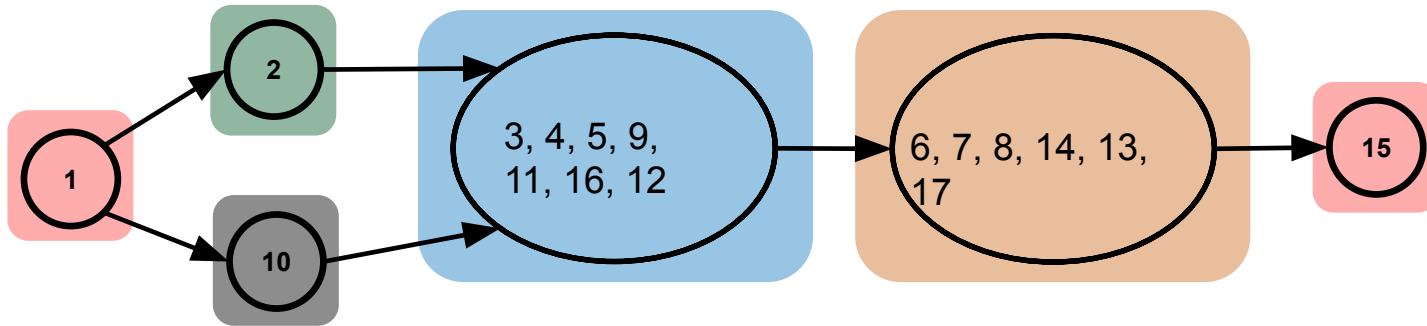
## Шаг 4. Проверка



## Граф: конденсат графа

**Конденсат графа** – это ориентированный граф, в котором вершины соответствуют компонентам сильной связности, а дуги отражают достижимость компонент друг из друга.

Конденсат графа **ацикличен**.



## Граф: обход в глубину

DFS – Depth First Search – обход ориентированного или неориентированного графа, при котором рекурсивно обходятся все вершины, достижимые из текущей вершины.

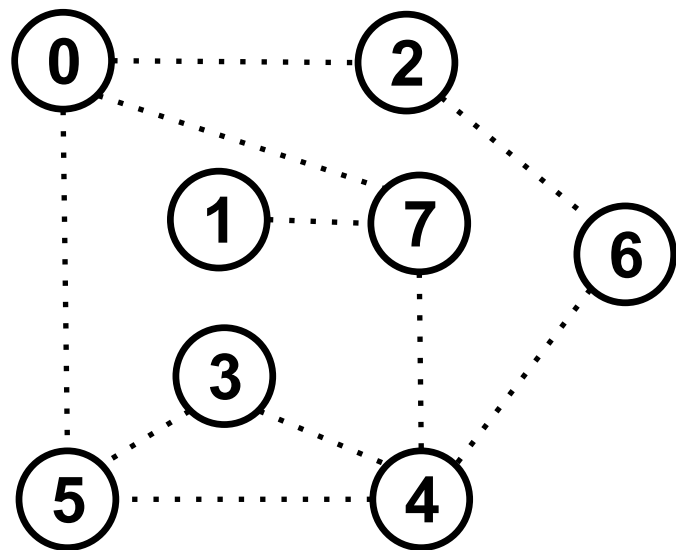
- 1) Выбираем непосещенную вершину  $u$ .
- 2) Запускаем  $\text{dfs}(u)$ :
  - Помечаем  $u$ ,
  - Запускаем  $\text{dfs}(v)$  для всех  $(u, v) \in E$ .
- 3) Повторяем 1) и 2) пока есть непосещенные вершины.

# Граф: обход в глубину

```
def dfs(u, visited)
    visited[u] = True
    for v in u.neighbours:
        if not visited[v]:
            dfs(v, visited)
```

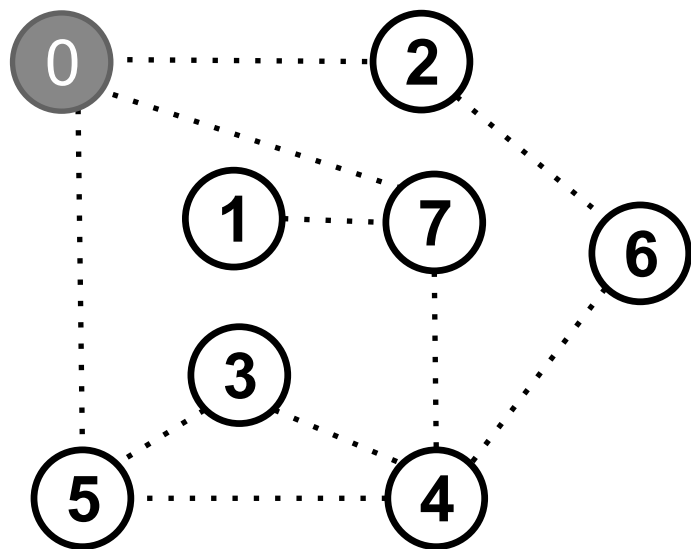
```
def traverse_dfs():
    visited: list[bool] = [False] * n
    for i in range(n):
        if not visited[i]:
            dfs(i, visited)
```

## Граф: обход в глубину



Индекс	0	1	2	3	4	5	6	7
Visited	0	0	0	0	0	0	0	0

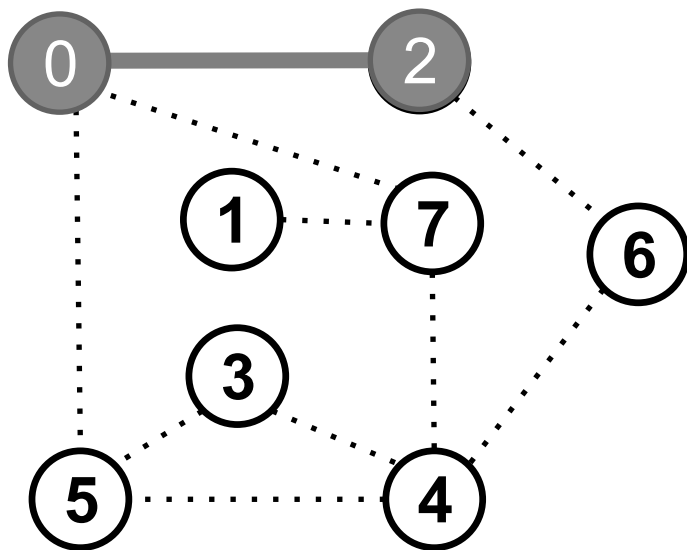
## Граф: обход в глубину



[0] ← Порядок посещения вершин

Индекс	0	1	2	3	4	5	6	7
Visited	1	0	0	0	0	0	0	0

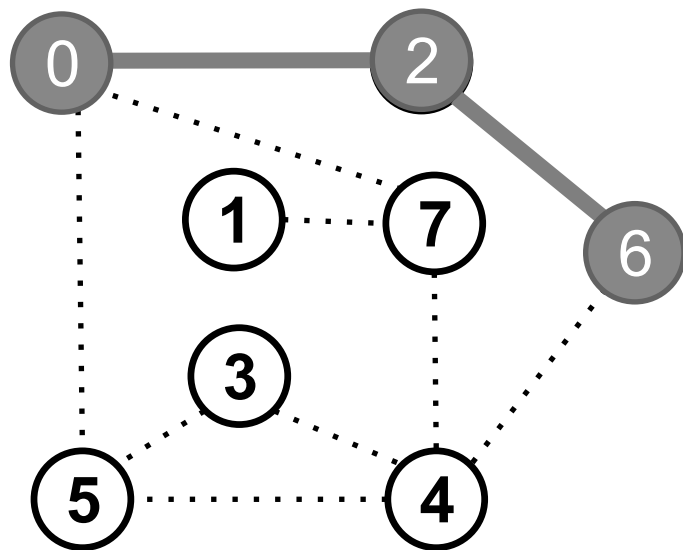
## Граф: обход в глубину



[0, 2]

Индекс	0	1	2	3	4	5	6	7
Visited	1	0	1	0	0	0	0	0

## Граф: обход в глубину

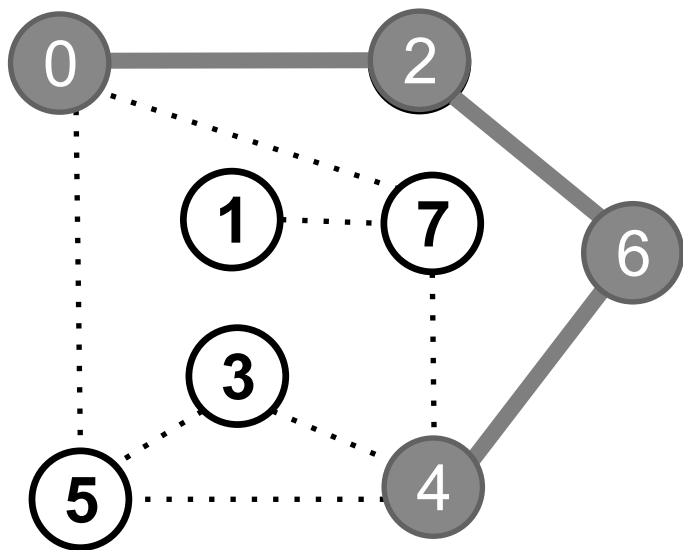


[0, 2, 6]

Индекс	0	1	2	3	4	5	6	7
Visited	1	0	1	0	0	0	1	0



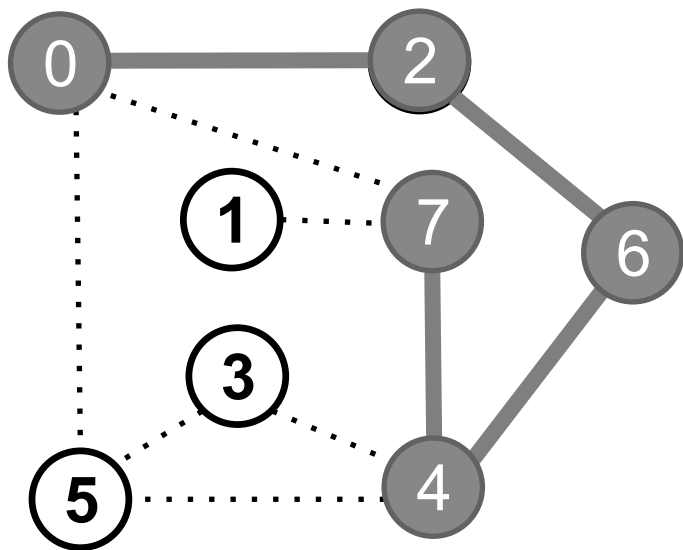
## Граф: обход в глубину



[0, 2, 6, 4]

Индекс	0	1	2	3	4	5	6	7
Visited	1	0	1	0	1	0	1	0

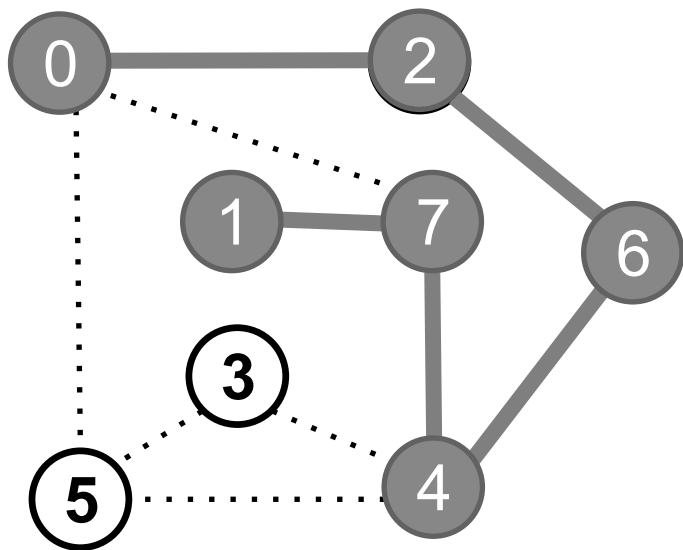
## Граф: обход в глубину



[0, 2, 6, 4, 7]

Индекс	0	1	2	3	4	5	6	7
Visited	1	0	1	0	1	0	1	1

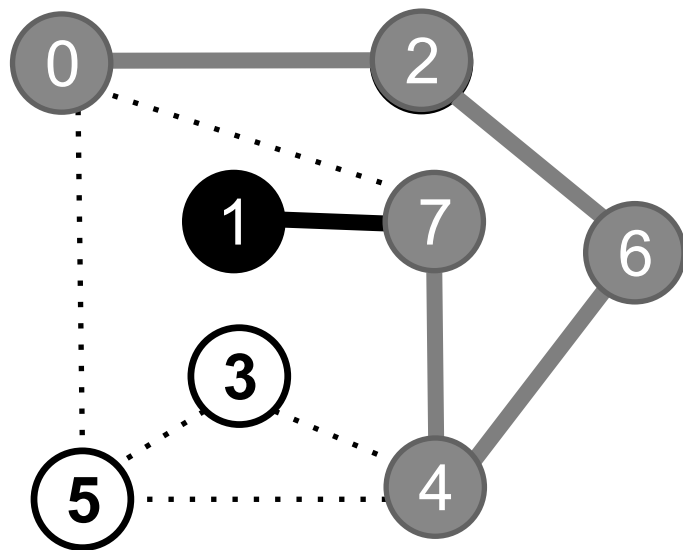
## Граф: обход в глубину



[0, 2, 6, 4, 7, 1]

Индекс	0	1	2	3	4	5	6	7
Visited	1	1	1	0	1	0	1	1

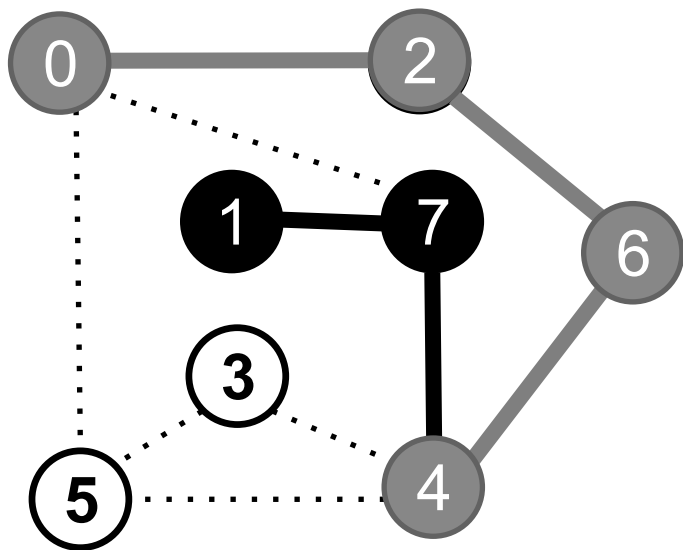
## Граф: обход в глубину



[0, 2, 6, 4, 7, 1]

Индекс	0	1	2	3	4	5	6	7
Visited	1	1	1	0	1	0	1	1

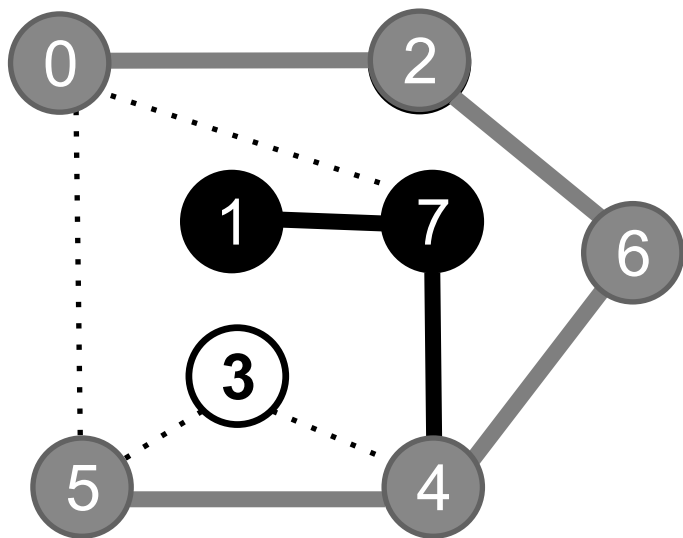
## Граф: обход в глубину



[0, 2, 6, 4, 7, 1]

Индекс	0	1	2	3	4	5	6	7
Visited	1	1	1	0	1	0	1	1

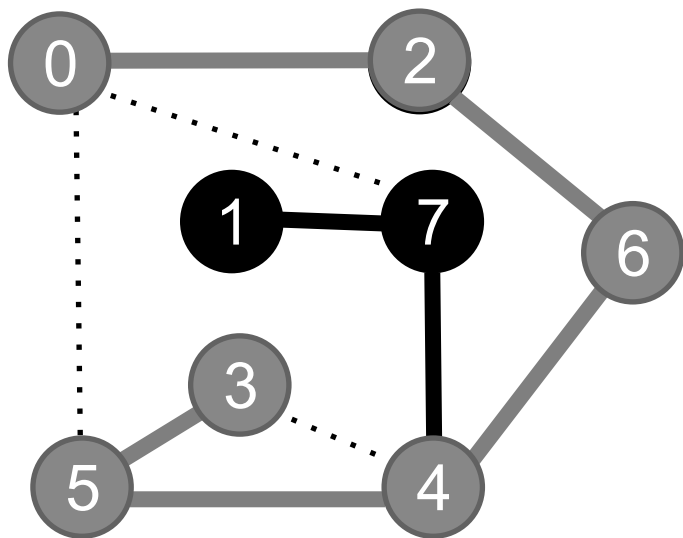
## Граф: обход в глубину



[0, 2, 6, 4, 7, 1, 5]

Индекс	0	1	2	3	4	5	6	7
Visited	1	1	1	0	1	1	1	1

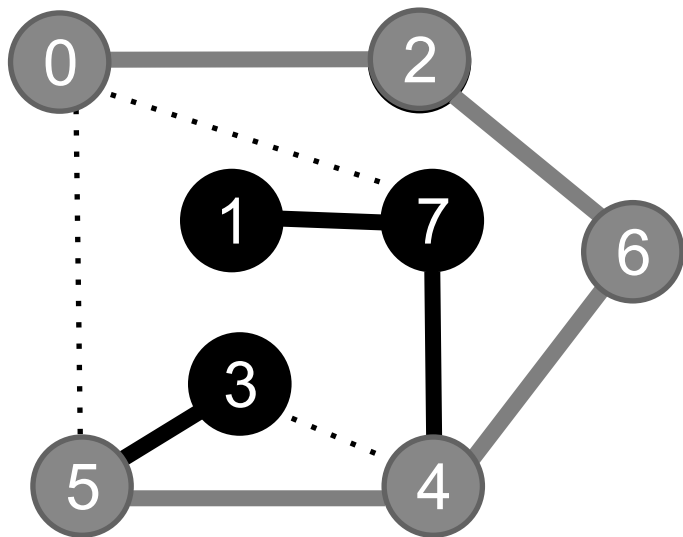
## Граф: обход в глубину



[0, 2, 6, 4, 7, 1, 5, 3]

Индекс	0	1	2	3	4	5	6	7
Visited	1	1	1	1	1	1	1	1

## Граф: обход в глубину

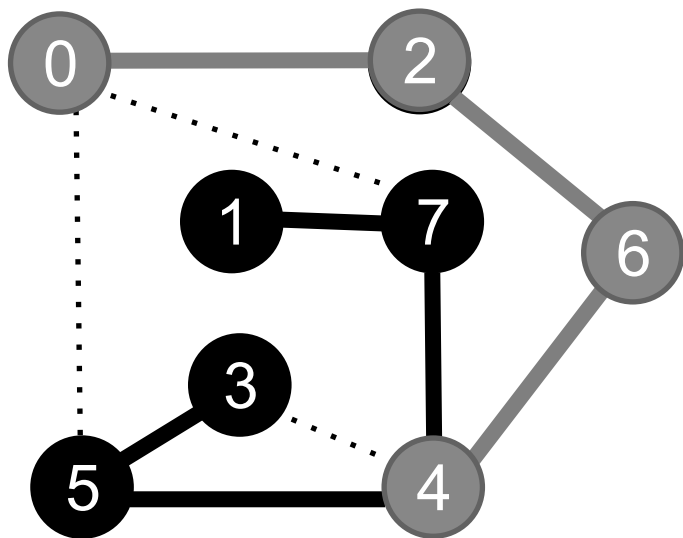


[0, 2, 6, 4, 7, 1, 5, 3]

Индекс	0	1	2	3	4	5	6	7
Visited	1	1	1	1	1	1	1	1



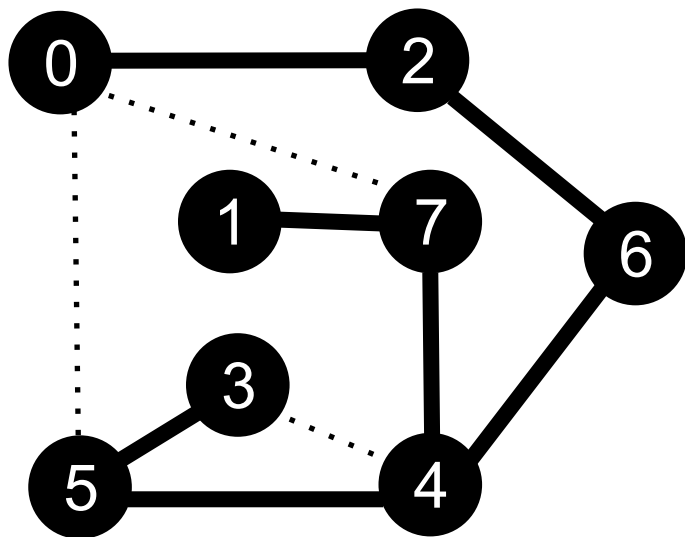
## Граф: обход в глубину



[0, 2, 6, 4, 7, 1, 5, 3]

Индекс	0	1	2	3	4	5	6	7
Visited	1	1	1	1	1	1	1	1

## Граф: обход в глубину



[0, 2, 6, 4, 7, 1, 5, 3]

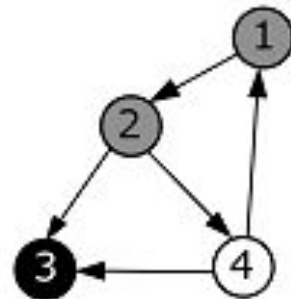
Индекс	0	1	2	3	4	5	6	7
Visited	1	1	1	1	1	1	1	1

## Граф: цвета вершин

Белая – в ней еще не были.

Серая – проходимся текущим вызовом dfs.

Черная – пройдена, итерации завершены.



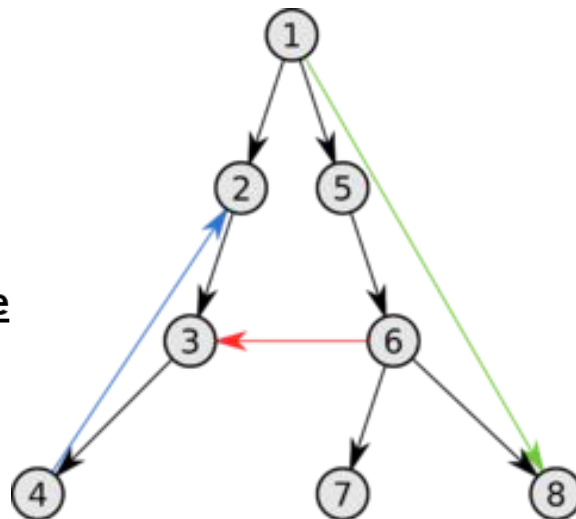
## Подграф предшествования.

$G_P = (V, E_P)$ , где  $E_P = \{(p[u], u)\}$ ,  $p[u]$  – вершина, от которой был вызван  $\text{dfs}(u)$ .

$G_P$  – лес обхода в глубину, состоящий из нескольких деревьев.

# Граф: типы ребер относительно dfs

1. Ребра дерева  $\in G_p$ .
2. Ребра  $(u, v)$ , соединяющие  $u$  с предком  $v$  – обратные ребра.
3. Ребра  $(u, v)$ , соединяющие  $u$  с потомком  $v$  – прямые ребра.
4. Остальные ребра  $(u, v)$  – перекрестные ребра.



Переход в белую вершину в dfs – ребро дерева.

Переход в серую вершину в dfs – обратное ребро.

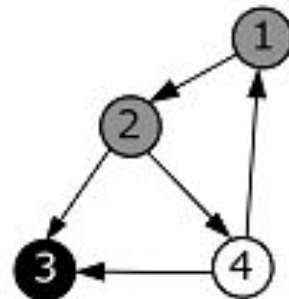
Переход в черную вершину в dfs – прямое или перекрестное.

Последние можно различить по времени входа и выхода.

- 1) ребра дерева
- 2) **обратные** ребра
- 3) **прямые** ребра
- 4) **перекрестные** ребра

## Граф: время входа и выхода

```
dfs( u ) {  
    entry[u] = time++  
    visited[u] = True  
    ... // обработка вершины.  
    leave[u] = time++  
}
```



В дереве dfs вершина  $u$  — предок  $v$  (ребро  $(u, v)$  - прямое), если  $entry[u] < entry[v]$  и  $leave[u] > leave[v]$ .

## Граф: простая лемма

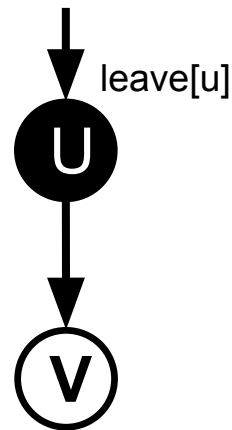
**Лемма.** Не существует момента поиска в глубину такого, в котором существует ребро из черной вершины в белую.

**Доказательство.** От противного. Пусть такое ребро  $(u, v)$  и момент  $time$  существуют. Рассмотрим момент  $leave[u]$ . Этот момент – первый, в котором вершина  $u$  – черная. Т.е.

$$leave[u] \leq time.$$

Следовательно, вершина  $v$  в момент  $leave[u]$  – белая, т.к. она белая в момент  $time$ .

Но это означает, что на момент выхода из вершины  $u$  есть необработанное ребро  $(u, v)$ . Противоречие.



# Граф: лемма о белых путях

**Лемма о белых путях.** Пусть есть некоторый обход dfs в графе  $G$ .

$entry[u]$  и  $leave[u]$  – моменты входа и выхода из вершины  $u$ . Тогда между этими моментами:

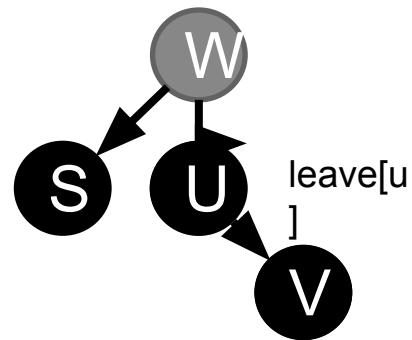
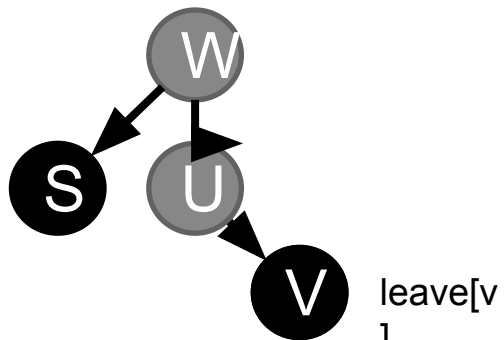
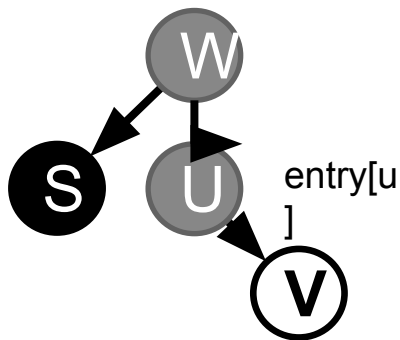
1. Черные и серые вершины  $G \setminus u$  не поменяют свой цвет.
2. Белые вершины  $G \setminus u$  либо останутся белыми, либо станут черными. Причем черными станут те, которые были достижимы из  $u$  по белым путям и только они.

**Доказательство.** Черная вершина останется черной.

Серая вершина останется серой, т.к. находится в стеке рекурсии.

Достижимая белая вершина станет черной. Иначе на пути к ней в момент  $leave[u]$  будет ребро из некоторой черной вершины в некоторую белую, чего не может быть по лемме.

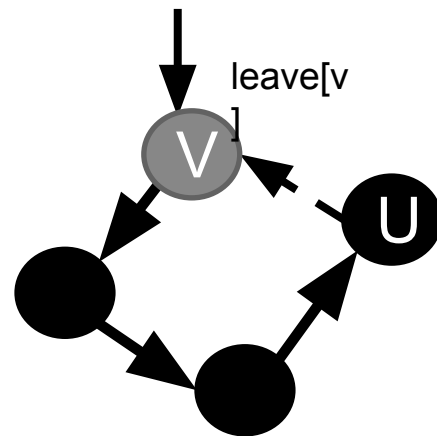
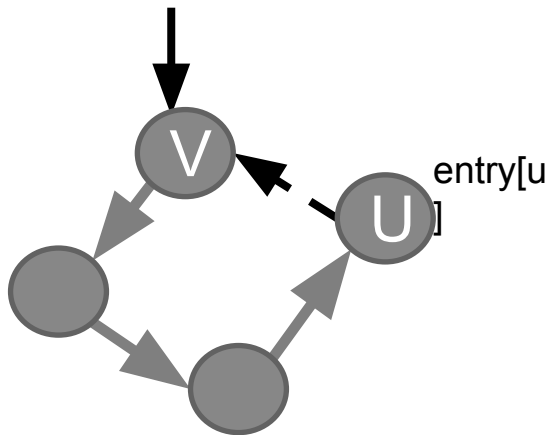
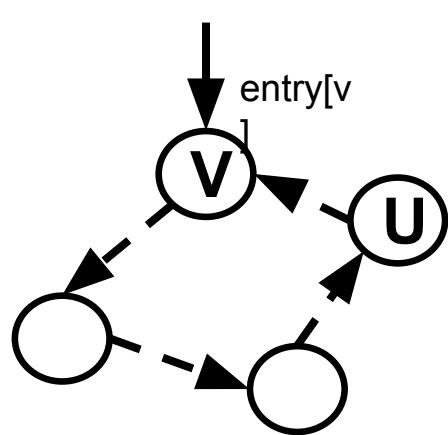
Если вершина стала черной к моменту  $leave[u]$ , значит, она была достижима из  $u$  по белому пути.



# Граф: проверка наличия циклов

**Задача.** Есть ориентированный или неориентированный граф  $G$ . Проверить наличие циклов в графе и, если циклы есть, найти какой-нибудь цикл.

**Решение.** Если в некоторый момент некоторого обхода dfs нашли обратное ребро (ведущее из текущей вершины в серую), то цикл существует. Иначе цикла нет.





## Граф: поиск цикла

Время работы проверки наличия цикла  
–  $O(V + E)$ .

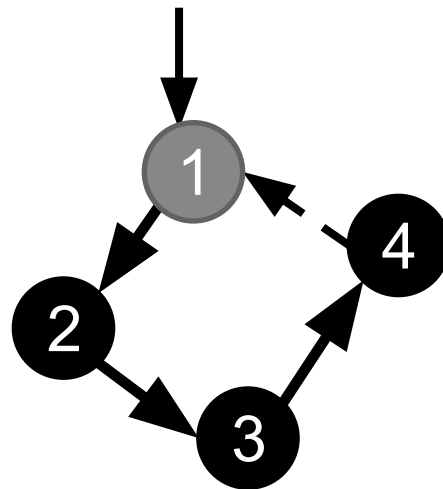
**Задача.** Найти какой-нибудь цикл в графе с циклами.

**Решение.**

Пусть в момент *time* в dfs была найден переход  $(u, v)$  в серую вершину  $v$ .

Цикл восстанавливается по предкам (стек вызовов в момент  $entry[u]$ ):

$v, u, p[u], p[p[u]], \dots, v.$



## Граф: проверка связности

**Задача.** Проверить, является ли неориентированный граф  $G$  связным.

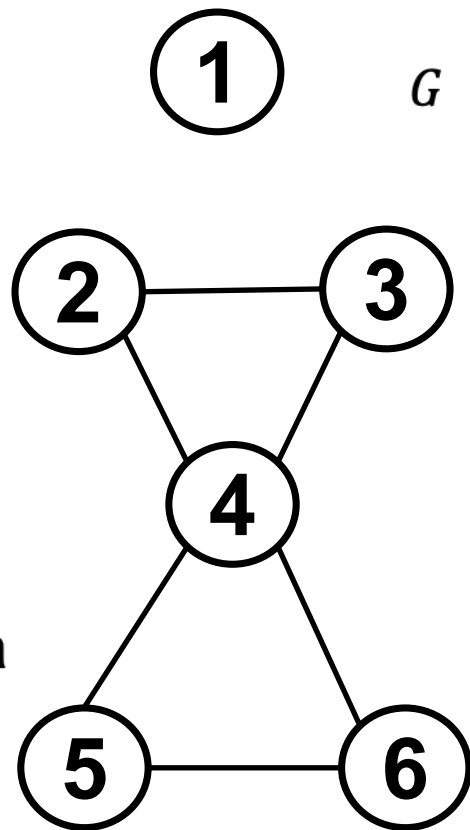
**Решение.**

Запустим  $\text{dfs}(v)$ .

Если после выхода все вершины посетили  $\Leftrightarrow$  связность.

После выхода из  $\text{dfs}(v)$  все вершины можно не проверять на  $\text{visited}[u]$ , если использовать переменную для подсчета числа обработанных вершин.

Время работы  $O(V + E)$ .



## Граф: топологическая сортировка

Определение. *Топологическая сортировка* ациклического графа  $G = (V, E)$  – такое упорядочивание всех вершин  $V$ , что если  $(u, v) \in E$ , то  $u$  располагается до  $v$ .

Формально: упорядочивание

$$\phi: V \rightarrow \{1, \dots, n\}, \phi(u) < \phi(v).$$

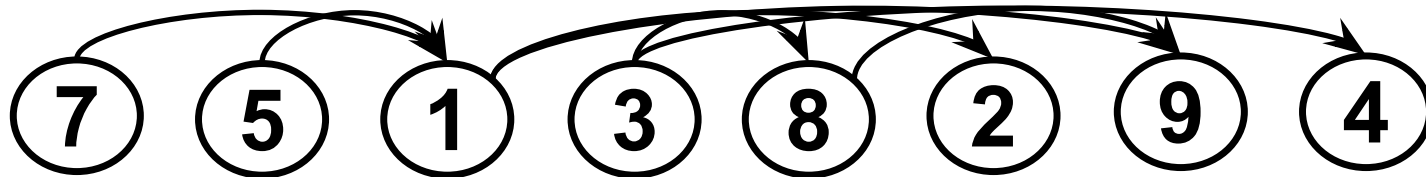
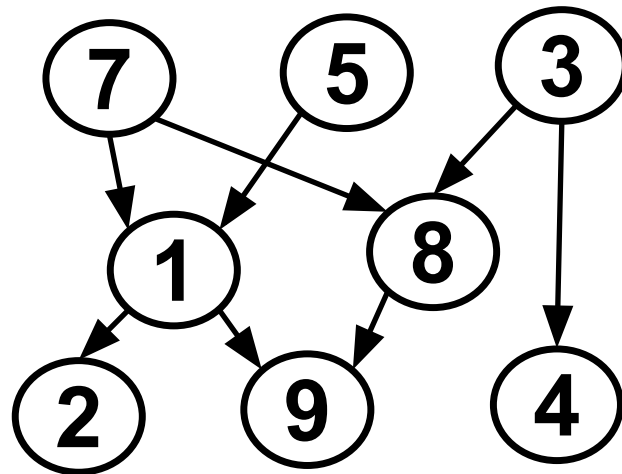
Топологическая сортировка для графов с циклами невозможна.

# Граф: топологическая сортировка

Строится корректная последовательность зависимых действий.

Например:

- Сборка исходников в правильном порядке
- Порядок прохождения обучающих курсов
- Порядок выполнения технологических операций



# Граф: топологическая сортировка

Алгоритм топологической сортировки:

- Запустить DFS, считать *leave*.
- $\phi(v) = |V| + 1 - \textit{leave}[v]$

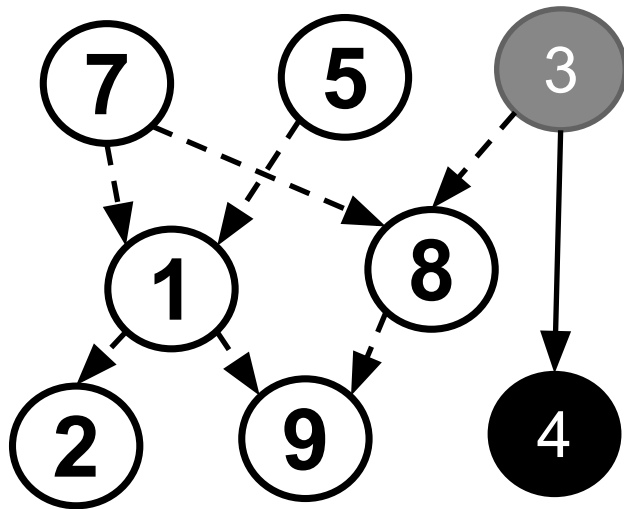
Время работы  $T = O(V + E)$ .

# Граф: топологическая сортировка

```
def dfs(u, visited, sorted_v)
    visited[u] = True
    for v in u.neighbours:
        if not visited[v]:
            dfs(v, visited)
    sorted_v.insert(0, u)

def traverse_dfs():
    sorted_v = []
    visited: list[bool] = [False] * n
    for i in range(n):
        if not visited[i]:
            dfs(i, visited, sorted_v)
```

## Граф: топологическая сортировка

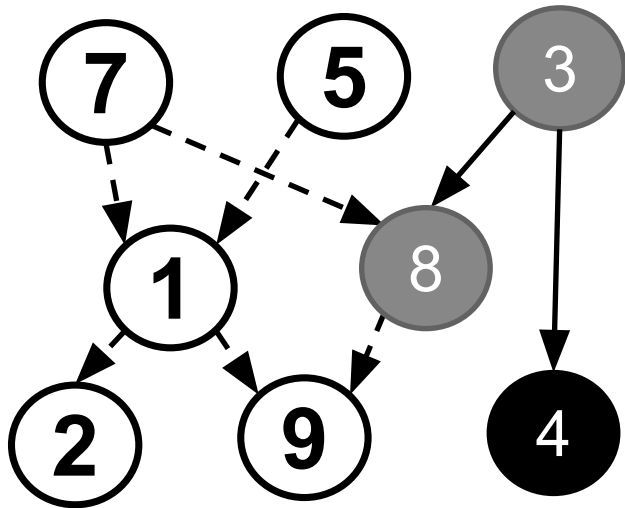


Добавляем элемент в начало списка в момент окраски в чёрный



sorted

## Граф: топологическая сортировка



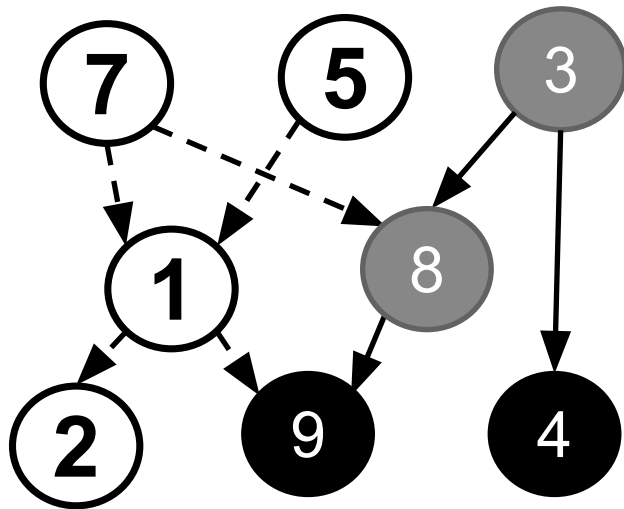
Добавляем элемент в начало списка в момент окраски в чёрный



sorted



## Граф: топологическая сортировка

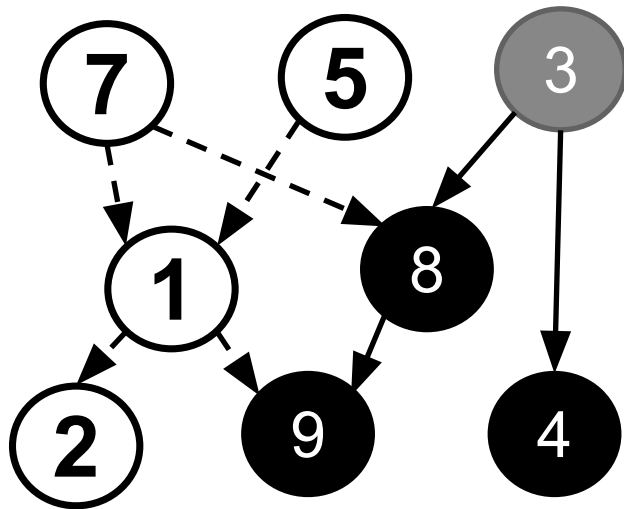


9

4

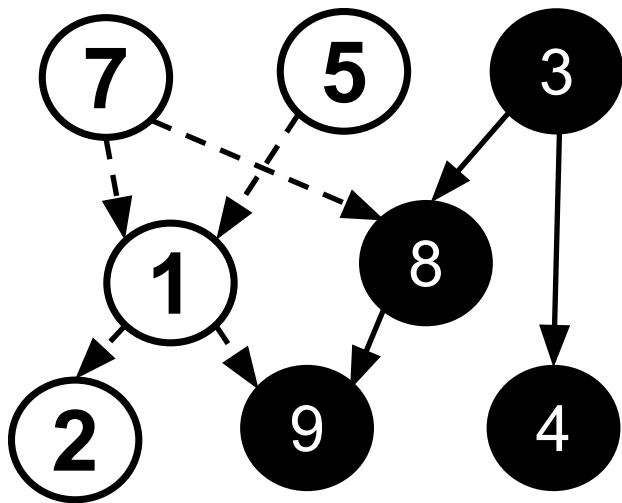
sorted

## Граф: топологическая сортировка



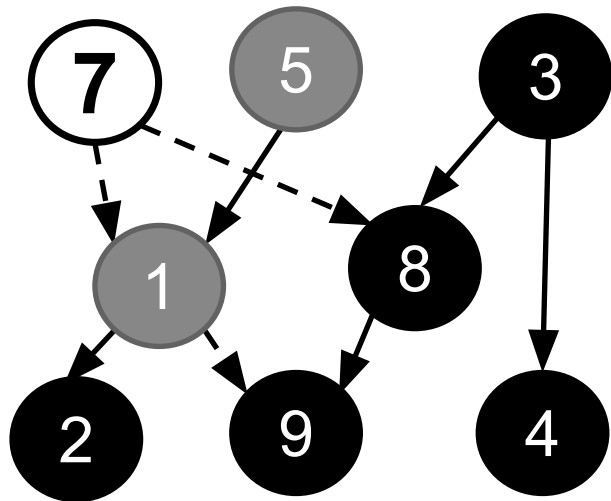
8 9 4 sorted

## Граф: топологическая сортировка



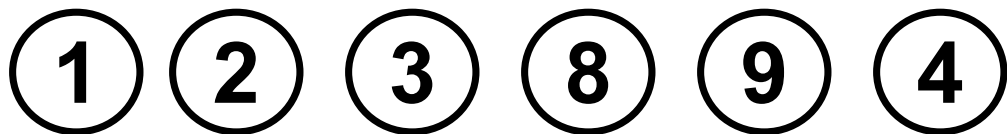
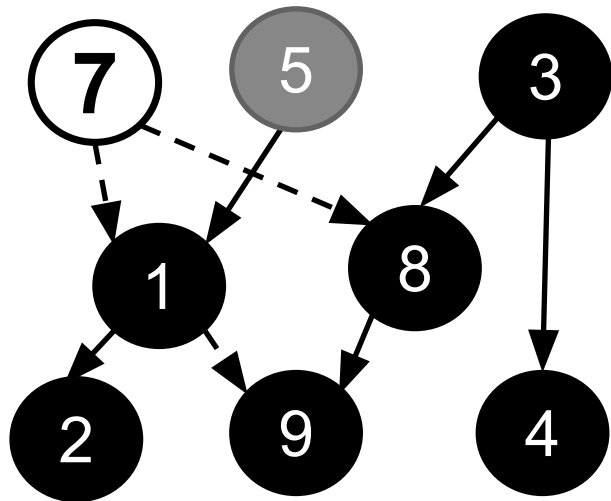
3 8 9 4 sorted

## Граф: топологическая сортировка



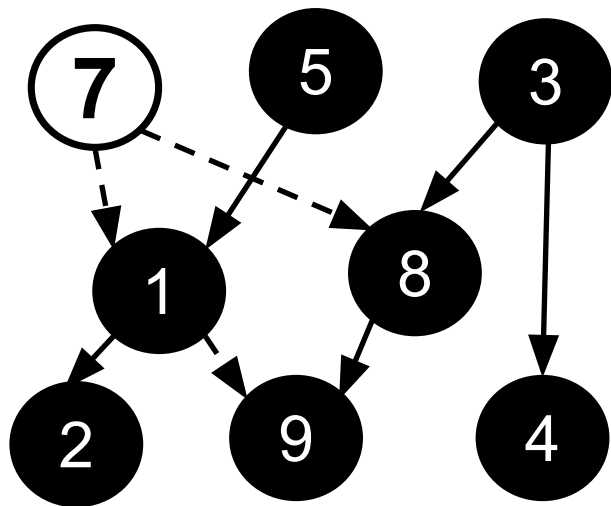
2 3 8 9 4 sorted

## Граф: топологическая сортировка



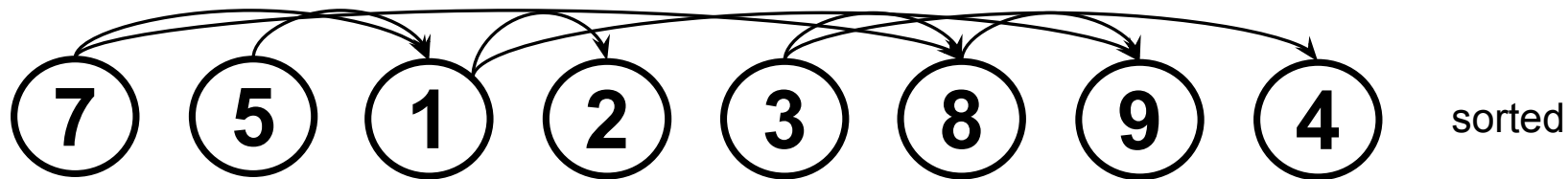
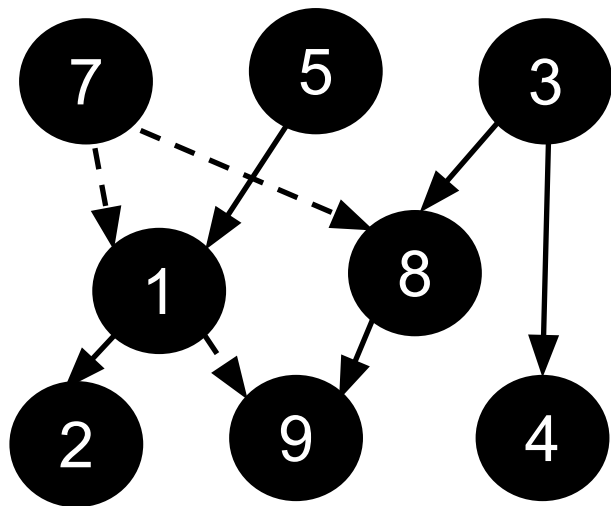
sorted

## Граф: топологическая сортировка



sorted

## Граф: топологическая сортировка



## Граф: топологическая сортировка

Теорема. Пусть  $G$  – ациклический ориентированный граф.  
Тогда существует топологическая сортировка, т.е.

$$\exists \phi: V \rightarrow \{1, \dots, n\},$$

т.ч. для любого  $(u, v) \in E: \phi(u) < \phi(v)$ .

Доказательство. Определим

$$\phi(v) = |V| + 1 - \text{leave}[v].$$

$\text{entry}[v]$  не считаем, только  $\text{leave}[v]$ , начиная с 1.

Докажем, что для так определенного  $\phi(v)$  выполняется  
 $\phi(u) < \phi(v)$  для любого  $(u, v) \in E$ .

Рассмотрим момент  $\text{entry}[u]$ .  $v$  – не серая, т.к. нет циклов.

- 1)  $v$  – белая. Тогда она будет обработана внутри  $\text{dfs}(u)$ .
- 2)  $v$  – черная. Значит, она уже обработана.



## Граф: алгоритм Косарайю

Алгоритм Косарайю (1978г) – алгоритм поиска сильно связных компонент.

Пусть  $G = (V, E)$ .

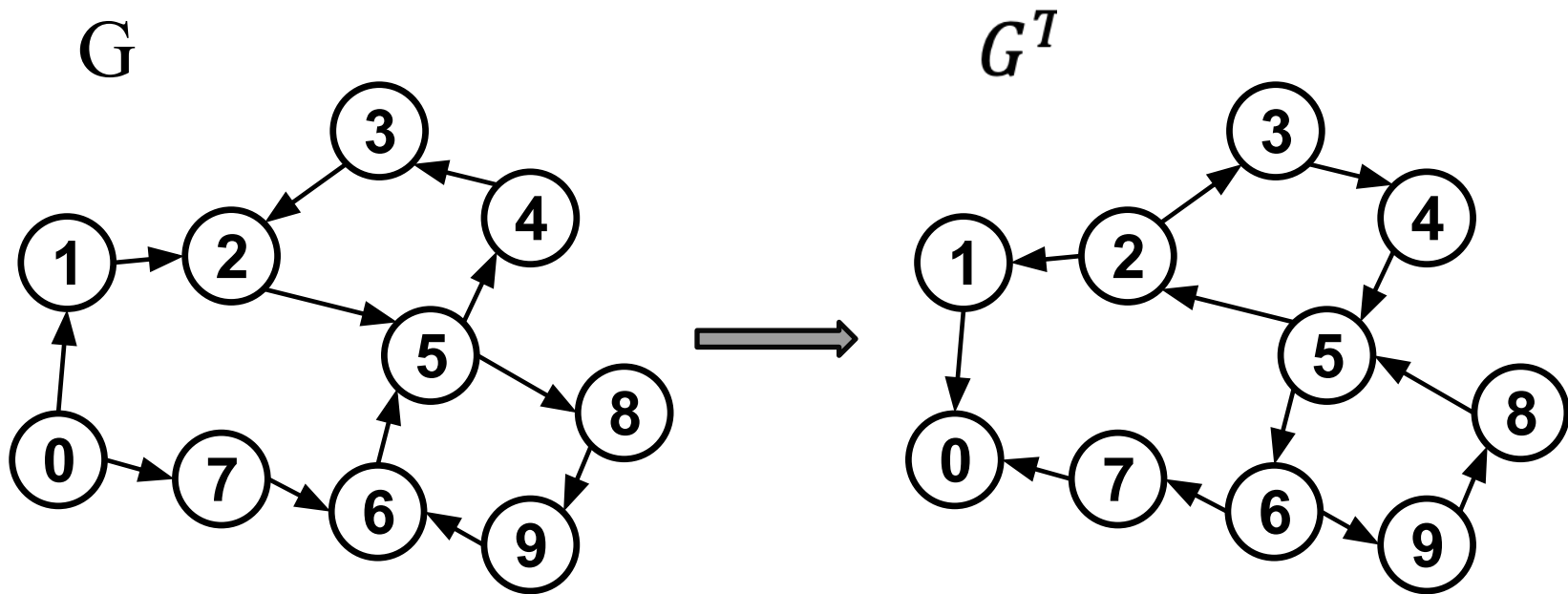
Алгоритм:

- 1) Построим  $H = G^T$  – граф, являющийся инвертированным к  $G$ .
- 2) DFS(H)
- 3) DFS(G), перебирая вершины в MainDFS в порядке убывания  $leave_H$ .

Деревья, полученные запусками dfs на шаге 3 – компоненты сильной связности.

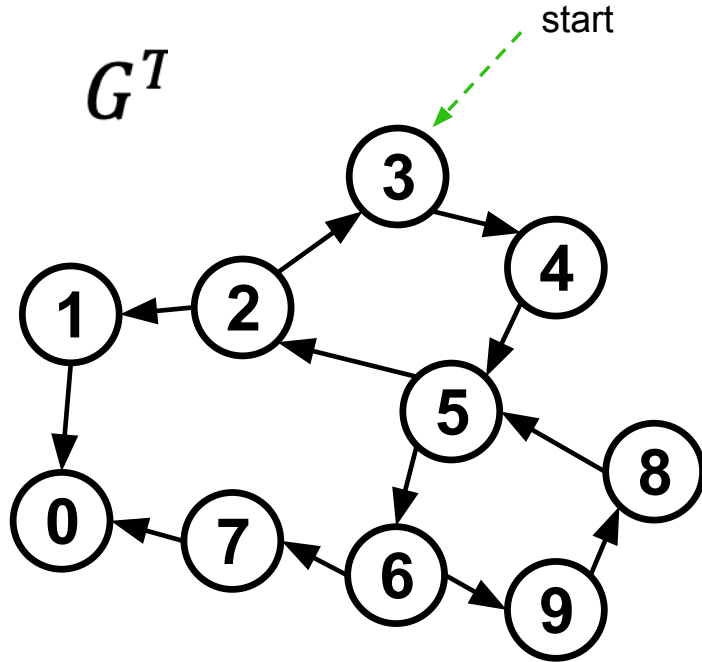
# Граф: алгоритм Косарайю

1) Построим  $H = G^T$  – граф, являющийся инвертированным к  $G$ .



# Граф: алгоритм Косарайю

2) Обходим в глубину граф  $H = G^T$ , для всех вершин запоминаем  $leave_H$



$leave_H: 0$

Visited

0	0
1	0
2	0
3	0
4	0
5	0
6	0
7	0
8	0
9	0

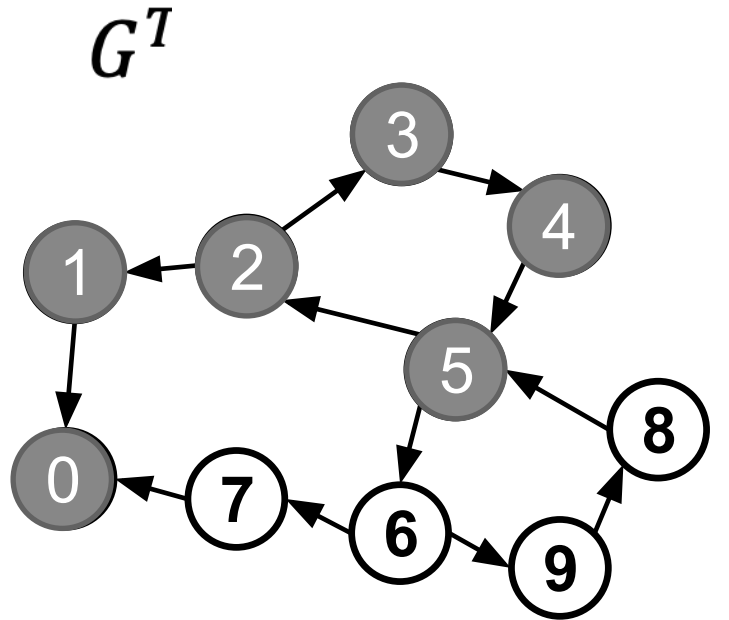
Order

0	
1	
2	
3	
4	
5	
6	
7	
8	
9	

Порядок обхода:

# Граф: алгоритм Косарайю

2) Обходим в глубину граф  $H = G^T$ , для всех вершин запоминаем  $leave_H$



Порядок обхода: 3 4 5 2 1 0

$leave_H: 0$

Visited

0	1
1	1
2	0
3	1
4	1
5	1
6	0
7	0
8	0
9	0

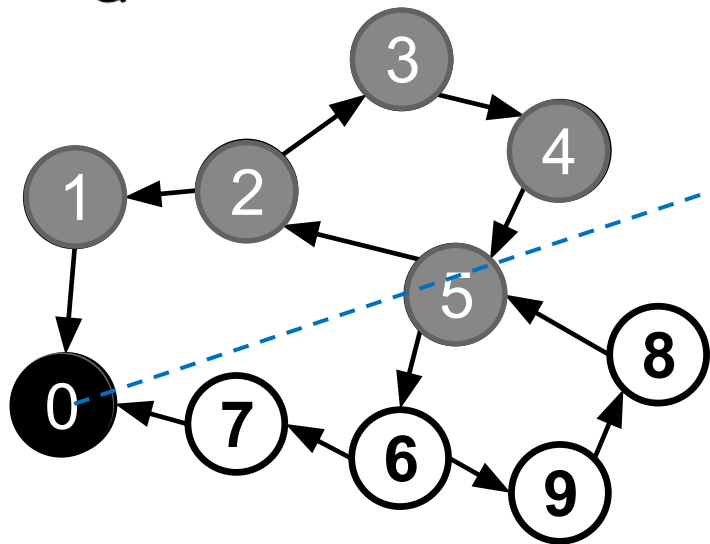
Order

0	
1	
2	
3	
4	
5	
6	
7	
8	
9	

# Граф: алгоритм Косарайю

В момент, когда вершина становится чёрная, пишем  $\text{order}[\text{leave}_H] = v$

$G^T$



Порядок обхода: 3 4 5 2 1 0

$\text{leave}_H: 0$

Visited

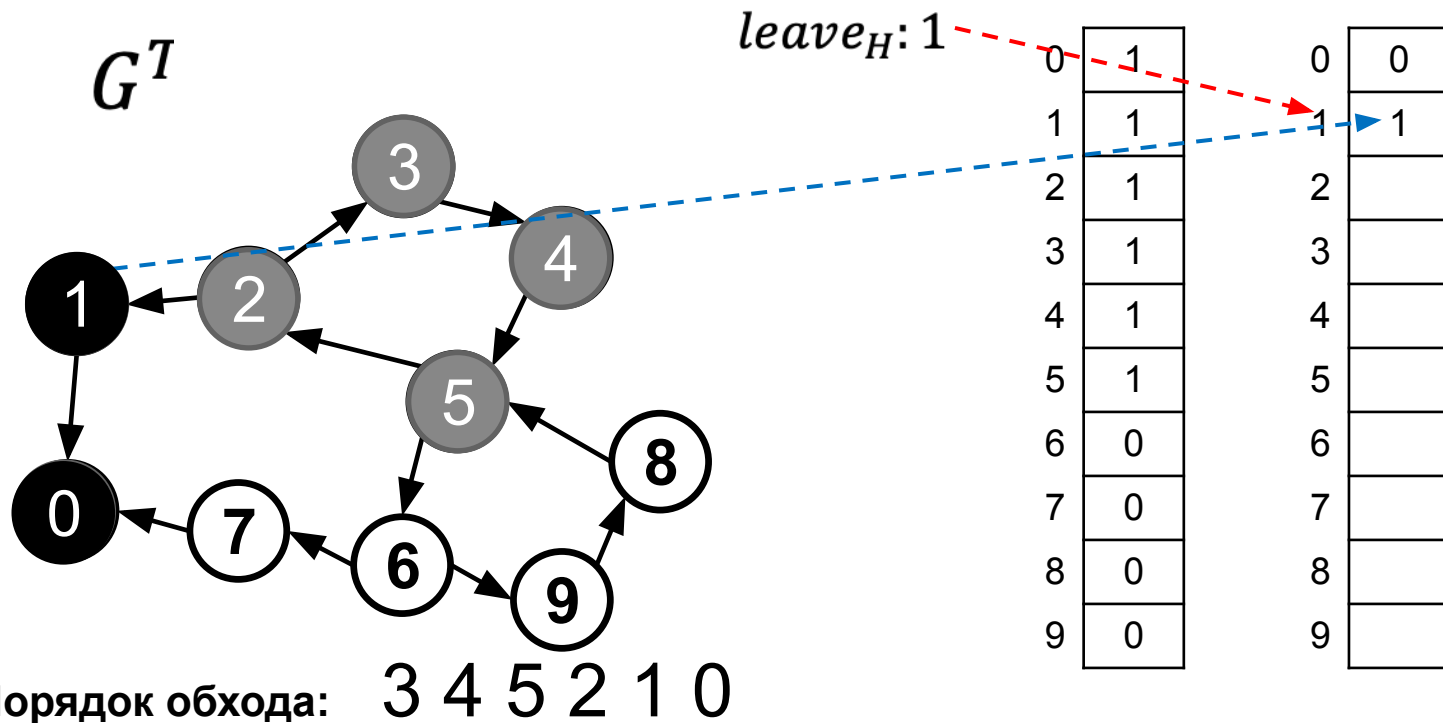
0	1
1	1
2	1
3	1
4	1
5	1
6	0
7	0
8	0
9	0

Order

0	0
1	
2	
3	
4	
5	
6	
7	
8	
9	

# Граф: алгоритм Косарайю

В момент, когда вершина становится чёрная, пишем  $\text{order}[\text{leave}_H] = v$

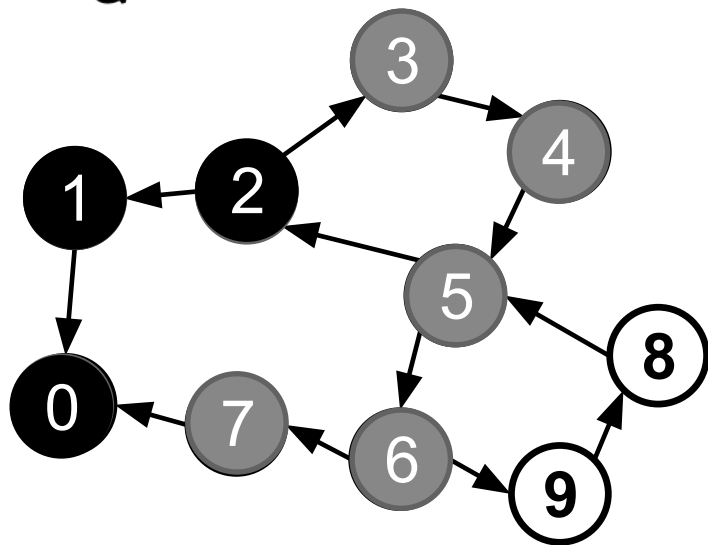


# Граф: алгоритм Косарайю

В момент, когда вершина становится чёрная, пишем  $\text{order}[\text{leave}_H] = v$

$G^T$

$\text{leave}_H: 3$



Порядок обхода: 3 4 5 2 1 0 6 7

Visited

0	1
1	1
2	1
3	1
4	1
5	1
6	1
7	1
8	0
9	0

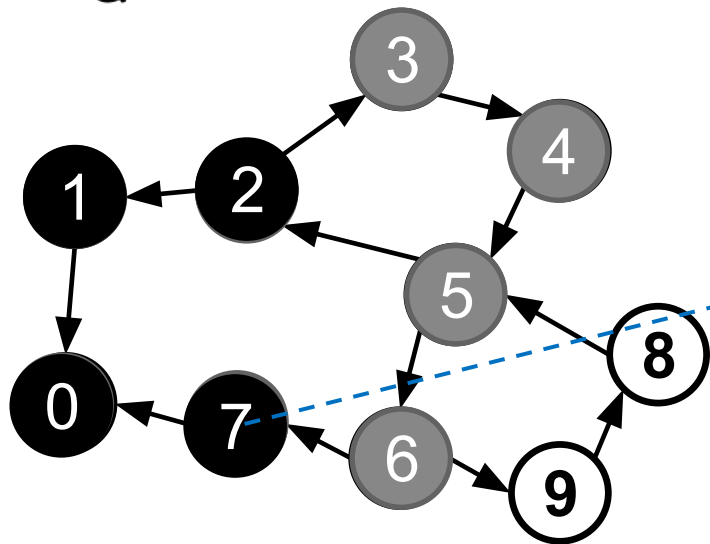
Order

0	0
1	1
2	2
3	
4	
5	
6	
7	
8	
9	

# Граф: алгоритм Косарайю

В момент, когда вершина становится чёрная, пишем  $\text{order}[\text{leave}_H] = v$

$G^T$



Порядок обхода: 3 4 5 2 1 0 6 7

$\text{leave}_H: 3$

Visited

0	1
1	1
2	1
3	1
4	1
5	1
6	1
7	1
8	0
9	0

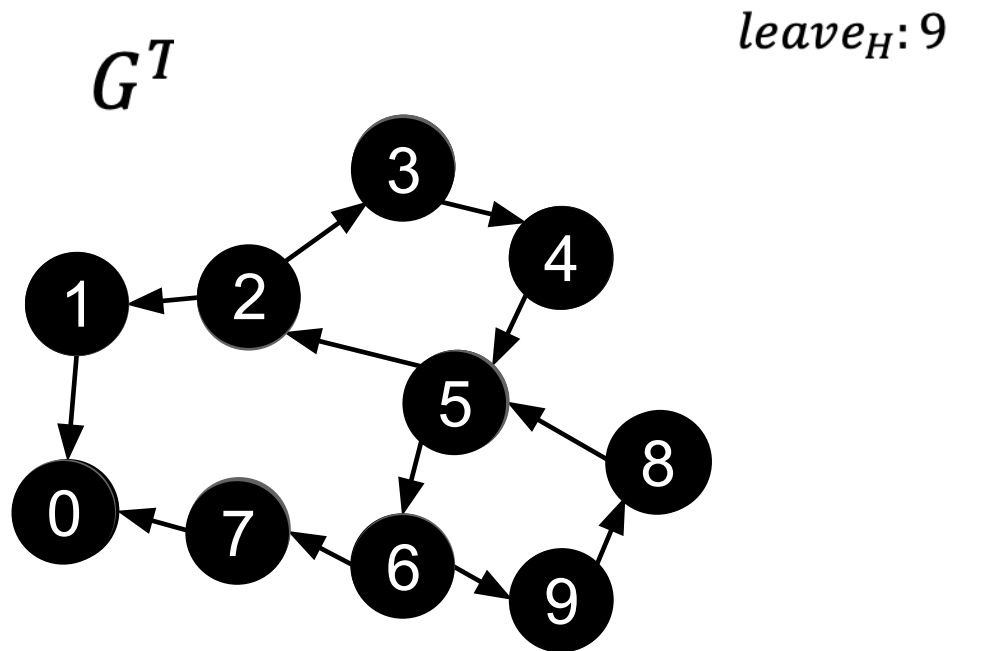
Order

0	0
1	1
2	2
3	7
4	
5	
6	
7	
8	
9	



# Граф: алгоритм Косарайю

По сути в Order получилась топологическая сортировка



Visited

0	1
1	1
2	1
3	1
4	1
5	1
6	1
7	1
8	1
9	1

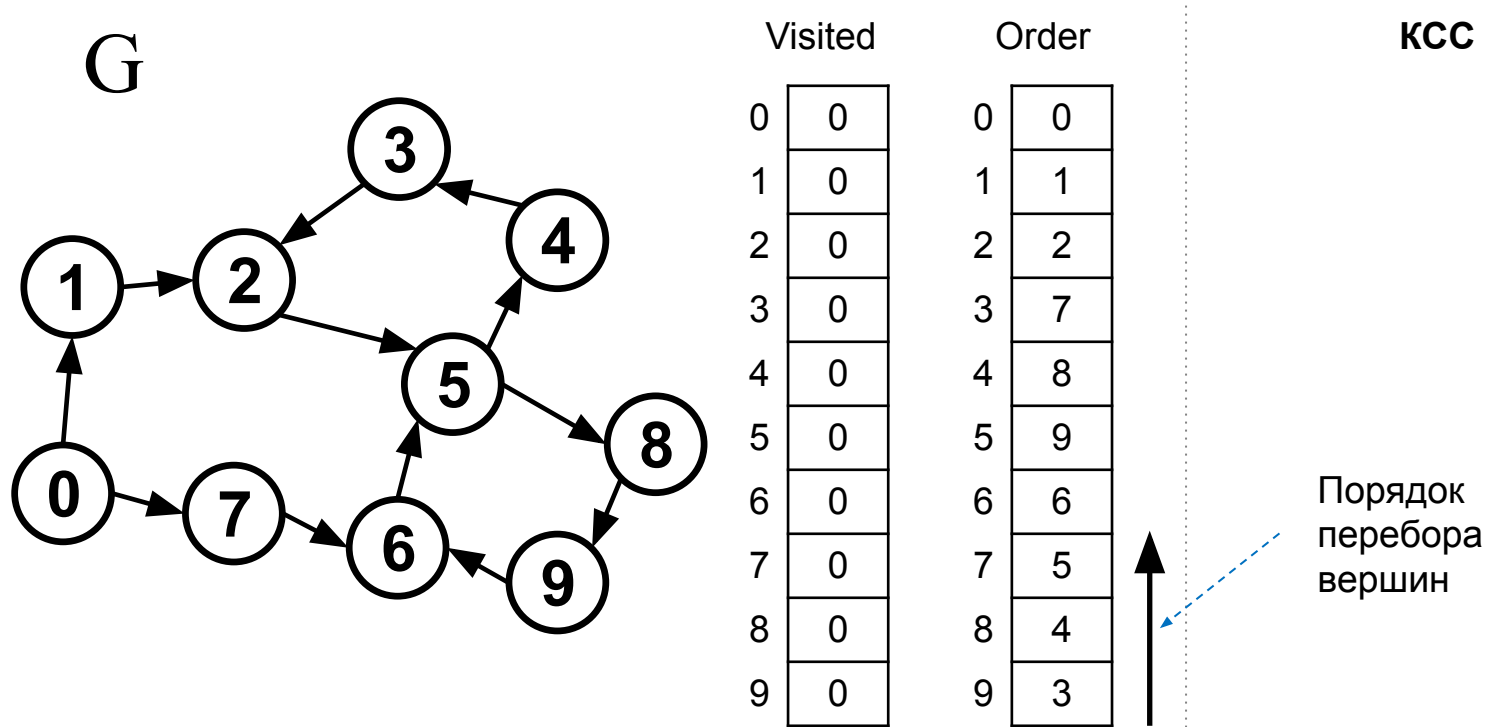
Order

0	0
1	1
2	2
3	7
4	8
5	9
6	6
7	5
8	4
9	3

Порядок обхода: 3 4 5 2 1 0 6 7 9 8

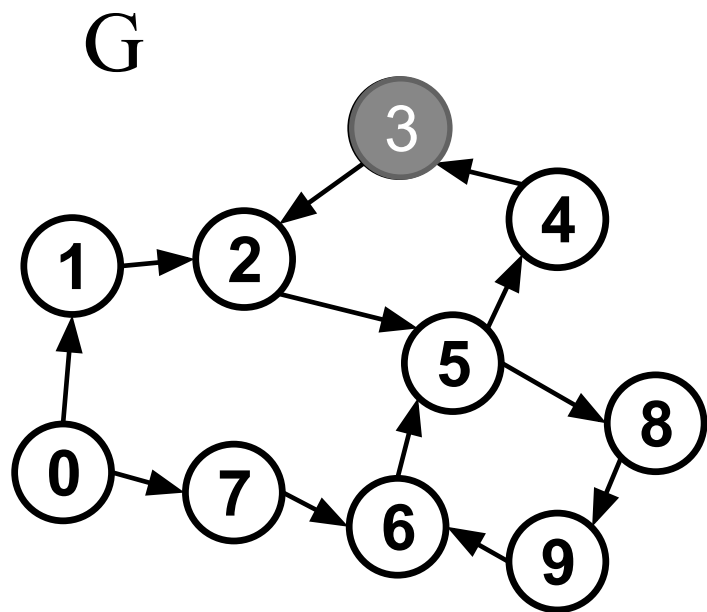
## Граф: алгоритм Косарайю

3) Обходим в глубину исходный граф  $G$  перебирая вершины в MainDFS в порядке убывания  $leave_H$ .



# Граф: алгоритм Косарайю

Начинаем с вершины 3, у нее самое большое  $leave_H$



Visited

0	0
1	0
2	0
3	1
4	0
5	0
6	0
7	0
8	0
9	0

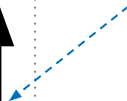
Order

0	0
1	1
2	2
3	7
4	8
5	9
6	6
7	5
8	4
9	3

КСС

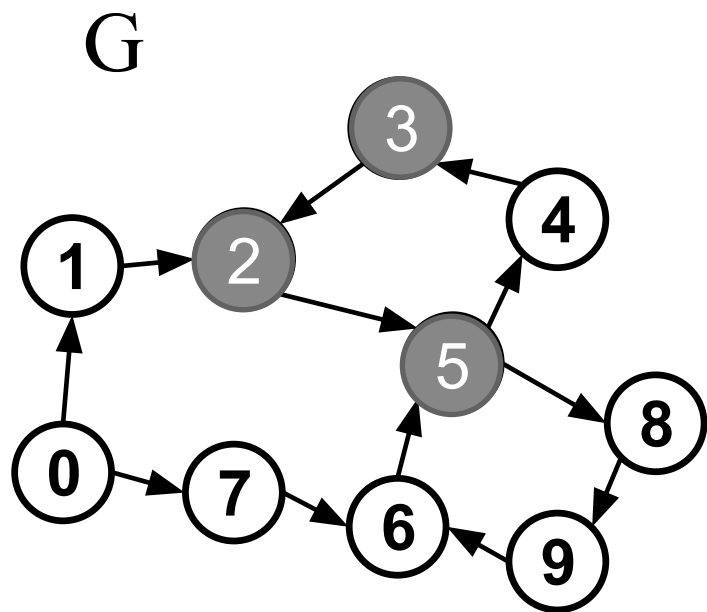
КСС1: 3

Порядок  
перебора  
вершин



# Граф: алгоритм Косарайю

Начинаем с вершины 3, у нее самое большое  $leave_H$



Visited

0	0
1	0
2	1
3	1
4	0
5	1
6	0
7	0
8	0
9	0

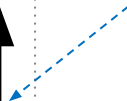
Order

0	0
1	1
2	2
3	7
4	8
5	9
6	6
7	5
8	4
9	3

КСС

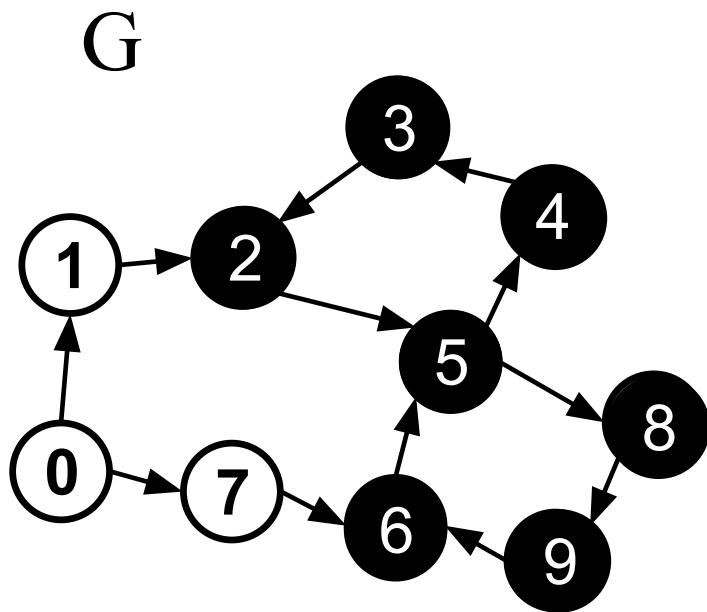
КСС1: 3 2 5

Порядок  
перебора  
вершин



# Граф: алгоритм Косарайю

Пропускаем вершины 4, 5, 6, 9, 8, так как уже посетили их.



Visited

0	0
1	0
2	1
3	1
4	1
5	1
6	1
7	1
8	1
9	1

Order

0	0
1	1
2	2
3	7
4	8
5	9
6	6
7	5
8	4
9	3

КСС

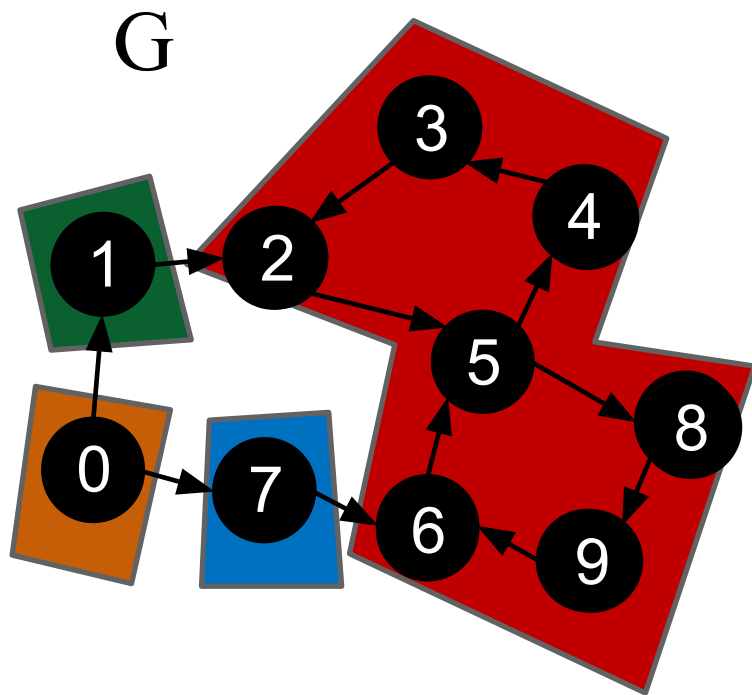
КСС1: 3 2 5 8 9 6 4

КСС2: 7

Уже посетили

# Граф: алгоритм Косарайю

В итоге получаем список КСС.



Visited

0	1
1	1
2	1
3	1
4	1
5	1
6	1
7	1
8	1
9	1

Order

0	0
1	1
2	2
3	7
4	8
5	9
6	6
7	5
8	4
9	3

КСС

KCC1: 3 2 5 8 9 6 4

KCC2: 7

KCC3: 1

KCC4: 0

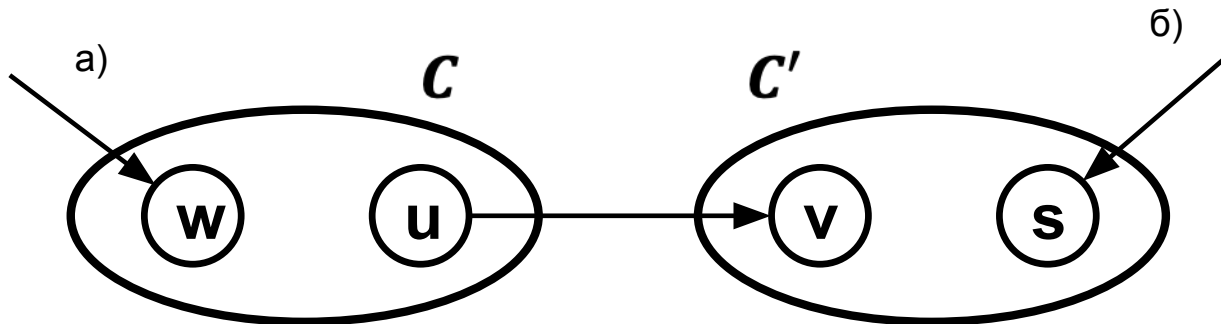
# Граф: алгоритм Косарайю

Пусть  $C$  – КСС. Обозначим  $\text{leave}[C]$  – максимальное время выхода  $\text{leave}[v]$ ,  $v \in C$ .

Лемма. Пусть  $C, C'$  – две различные КСС, и есть ребро  $(u, v)$  между ними,  $u \in C, v \in C'$ . Тогда  $\text{leave}[C] > \text{leave}[C']$ .

Доказательство леммы. а) Первой была достигнута КСС  $C$  – вершина  $w$ . Тогда в момент входа в  $C$  вся компонента  $C'$  – белая и достижима из  $C$ . По лемме о белых путях в момент  $\text{leave}[w] > \text{leave}[C']$ .

б) Первой была достигнута КСС  $C'$ . В этом случае вся компонента  $C'$  будет пройдена до обхода  $C$ , т.к. не существует пути из  $C'$  в  $C$ . То есть  $\text{leave}[C] > \text{leave}[C']$ .



## Граф: алгоритм Косарайю

Теорема. Деревья, полученные в п. 3 алгоритма Косарайю, – КСС.

Доказательство.  $\Leftarrow$

Пусть вершины  $s$  и  $t$  из одной КСС.

Тогда существуют пути  $s \rightsquigarrow t$  и  $t \rightsquigarrow s$ . Значит, вершины  $s$  и  $t$  попадут в одно дерево в шаге 3).

Это следует из следующего рассуждения:

Пусть  $v$  – первая вершина из цикла  $s \rightsquigarrow t \rightsquigarrow s$  в обходе DFS( $G$ ). Тогда в момент  $entry[v]$  вершины  $s$  и  $t$  достижимы из  $v$  по белым путям. По лемме о бп они будут обработаны в  $dfs(v)$ .



## Граф: алгоритм Косарайю

Продолжение доказательства теоремы.  $\Rightarrow$

Рассмотрим дерево  $T$  – дерево обхода dfs на этапе 3.

Докажем, что  $T$  – компонента сильной связности.

Пусть  $T$  содержит два или более различных КСС и пусть  $C$  – первая КСС в обходе dfs. Существуют ребро  $(v, u)$ , пройденное dfs.  $v \in C$  и  $u \in C_2$ .  $C, C_2$  – различные КСС.

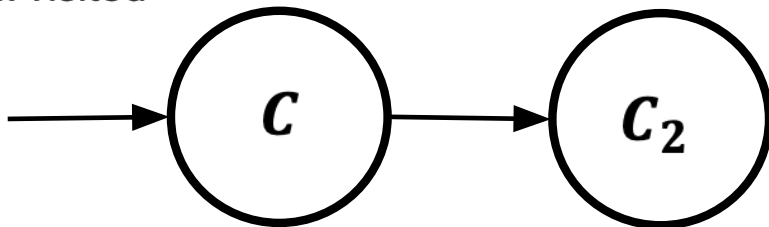
$leave_H[C] > leave_H[C_2]$  по построению  $T$ .

Но по лемме  $leave_H[C] < leave_H[C_2]$ , т.к. ребро  $(u, v) \in H$ .

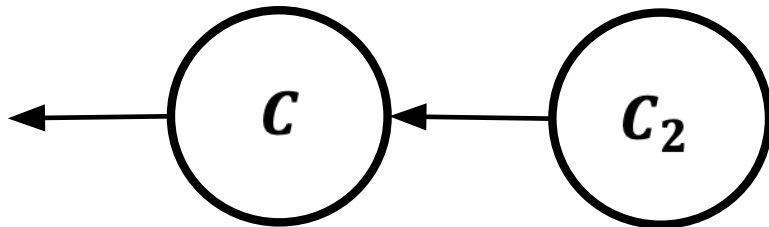
Противоречие.

## Граф: алгоритм Косарайю

Если можем пройти по ребру при обходе dfs в п. 3, значит вершины из  $C_2$  не отмечены как visited



Но такого быть не может, потому что при обходе инвертированного графа по лемме  $leave_H[C] < leave_H[C_2]$



В п.3 перебираем вершины в порядке убывания  $leave_H$ . Следовательно, вершины из  $C_2$  будут отмечены как visited до начала обхода  $C$ .

# Граф: обход в ширину

BFS – Breadth First Search – обход в ширину.

Обход, при котором вершины обходятся в порядке увеличения расстояния от стартовой вершины.

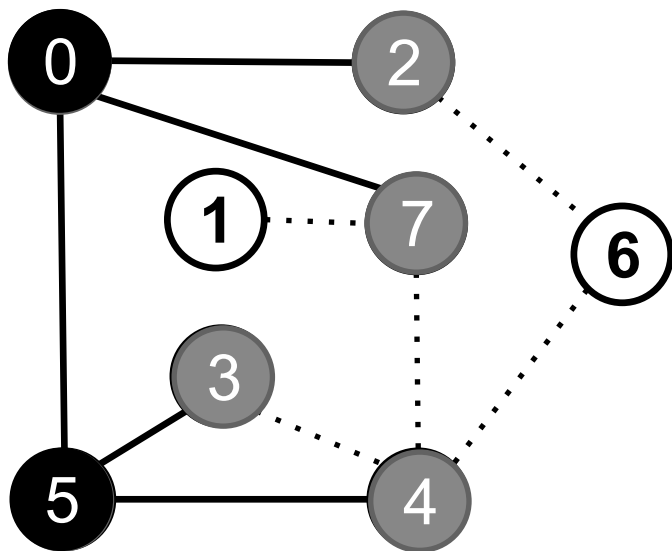
Обход, при котором вершины обходятся «по слоям».

Как и в обходе в ширину деревьев используется очередь.

```
vector<bool> visited;
void bfs( int u ) {
    std::queue<int> q;
    q.push( u ); visited[q] = true;
    while( !q.empty() ) {
        v = q.front(); q.pop();
        for( w: (v, w) ∈ E ) {
            if( !visited[w] ) {
                visited[w] = true;
                q.push( w );
            }
        }
    }
}

int MainBFS() {
    visited.assign( n, false );
    for( int i = 0; i < n; ++i )
        if( !visited[i] )
            bfs( i );
}
```

## Граф: обход в ширину

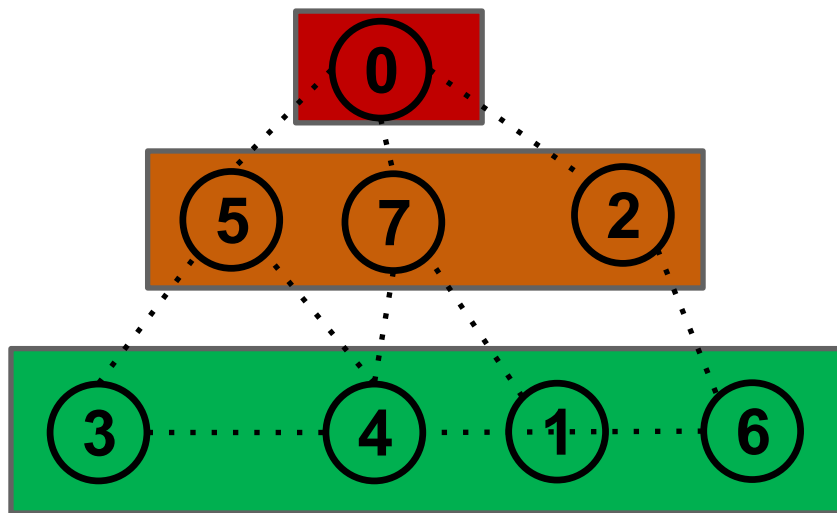


Queue: 7 2 3 4

Индекс	0	1	2	3	4	5	6	7
Visited	1	0	1	1	1	1	0	1

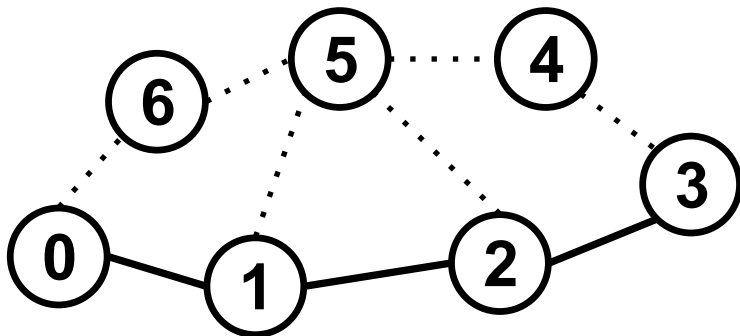
# Граф: обход в ширину

BFS - обход по «слоям»



Вершины в одном слое, если у них одинаковое расстояние от стартовой вершины.

# Граф: поиск кратчайших путей



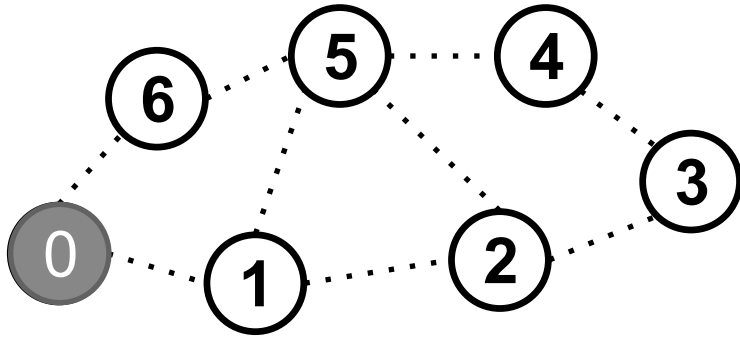
Алгоритм:

1. Обходим граф в ширину.
2. Для непосещенных вершин запоминаем кратчайший путь.

Сложность  $O(V + E)$

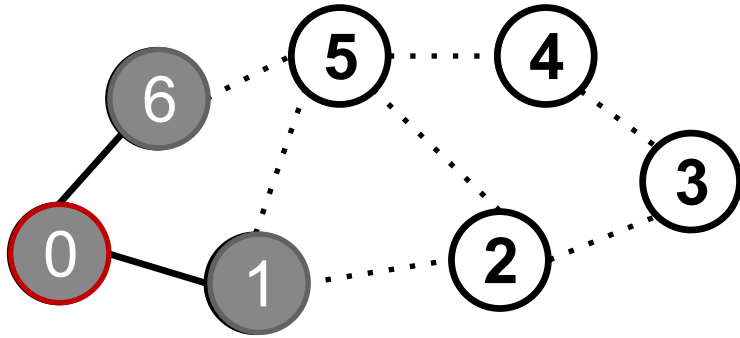
```
vector<int> r, pi;
void bfs( int s ) {
    r.assign( n, INT_MAX );
    std::queue<int> q;
    q.push( s ); r[s] = 0; pi[s] = -1;
    while( !q.empty() ) {
        v = q.front(); q.pop();
        for( w: (v, w) ∈ E )
            if( r[w] > r[v] + 1 ) {
                r[w] = r[v] + 1;
                pi[w] = v;
                q.push( w );
            }
    }
}
```

# Граф: поиск кратчайших путей



	r	pi	Queue
0	0	-1	0
1			
2			
3			
4			
5			
6			

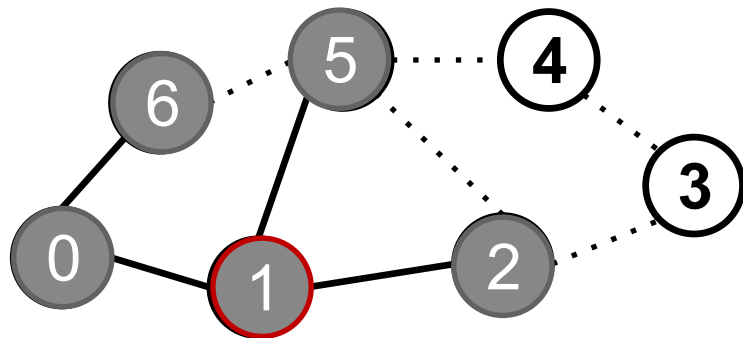
# Граф: поиск кратчайших путей



	r	pi	Queue
0	0	-1	1 6
1	1	0	
2			
3			
4			
5			
6	1	0	



# Граф: поиск кратчайших путей

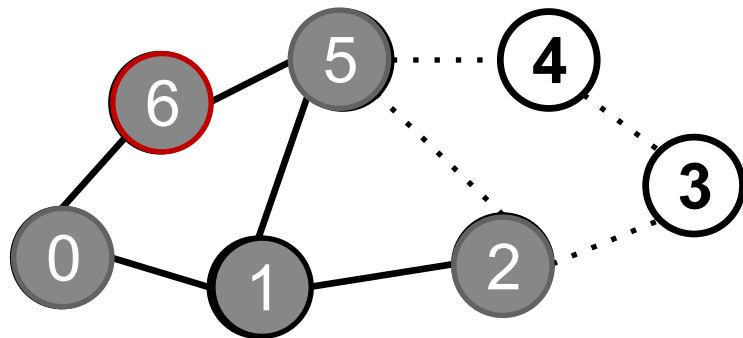


	r
0	0
1	1
2	2
3	
4	
5	2
6	1

	pi
0	-1
1	0
2	1
3	
4	
5	1
6	0

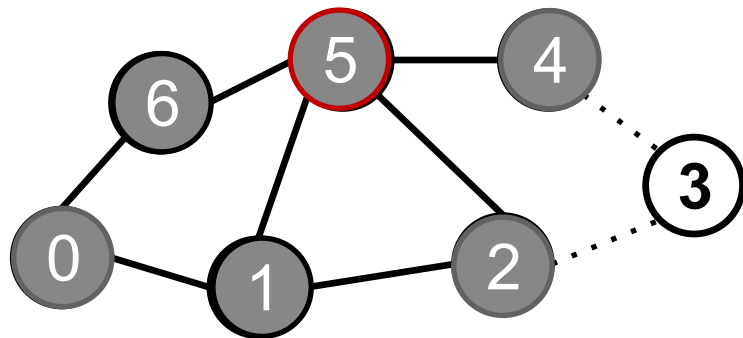
Queue  
6 5 2

# Граф: поиск кратчайших путей



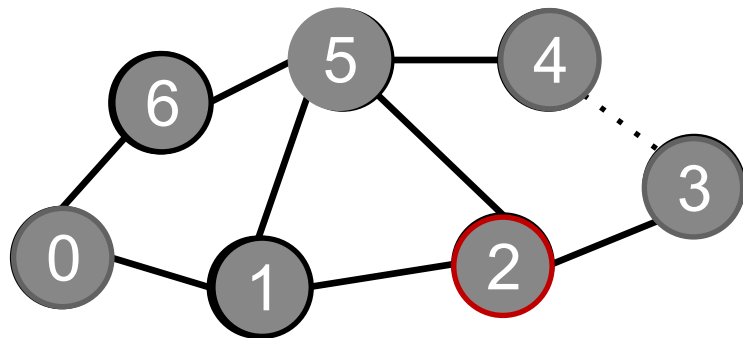
	r	pi	Queue
0	0	-1	5 2
1	1	0	
2	2	1	
3			
4			
5	2	1	
6	1	0	

# Граф: поиск кратчайших путей



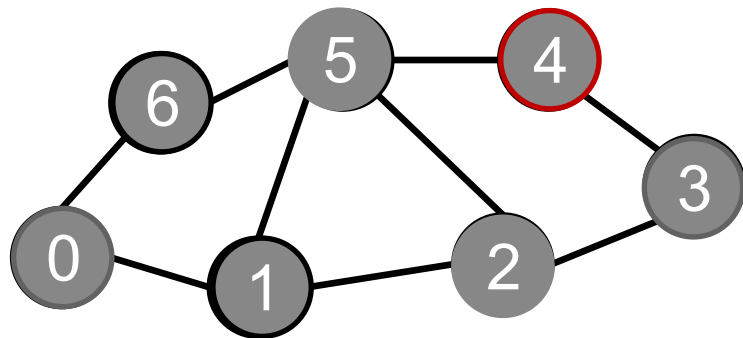
	r	pi	Queue
0	0	-1	2 4
1	1	0	
2	2	1	
3			
4	3	5	
5	2	1	
6	1	0	

# Граф: поиск кратчайших путей



	r	pi	Queue
0	0	-1	4 3
1	1	0	
2	2	1	
3	3	2	
4	3	5	
5	2	1	
6	1	0	

# Граф: поиск кратчайших путей

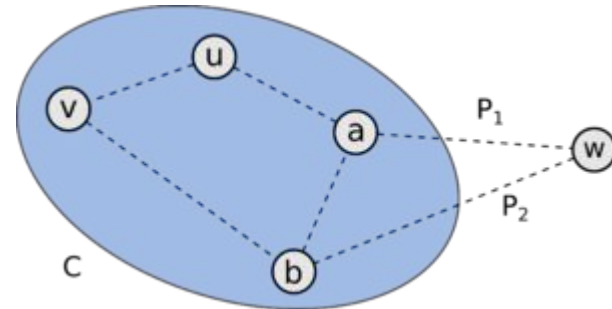


	r	pi	Queue
0	0	-1	3
1	1	0	
2	2	1	
3	3	2	
4	3	5	
5	2	1	
6	1	0	

## Граф: реберная двусвязность

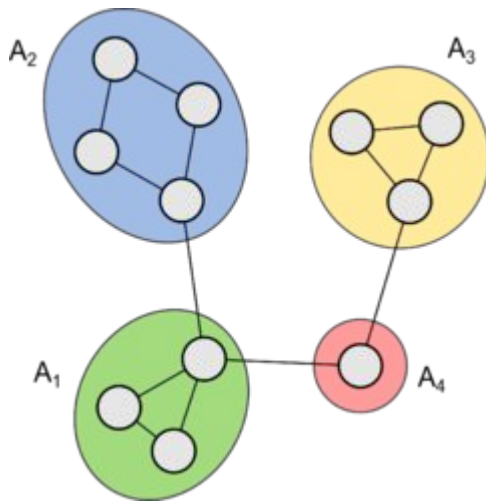
Вершины  $u$  и  $v$  в неориентированном графе  $G$  **реберно двусвязны**, если между этими вершинами существуют два реберно непересекающихся пути.

**Реберная двусвязность** – отношение эквивалентности.



## Граф: реберная двусвязность

Вершины неориентированного графа разбиваются на **компоненты реберной двусвязности**.



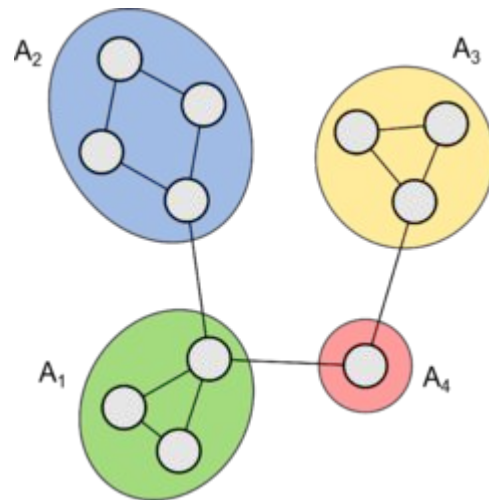
# Граф: мост

**Мост** – ребро, связывающее две различные компоненты реберной двусвязности.

**Мост** – ребро, при удалении которого компонента связности распадается.

**Мост** – ребро  $(u, v)$ , лежащее на любом пути, соединяющем  $u$  и  $v$ .

Все три определения эквивалентны.

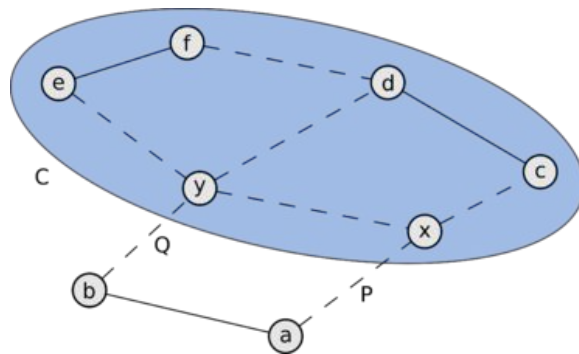




# Граф: вершинная двусвязность

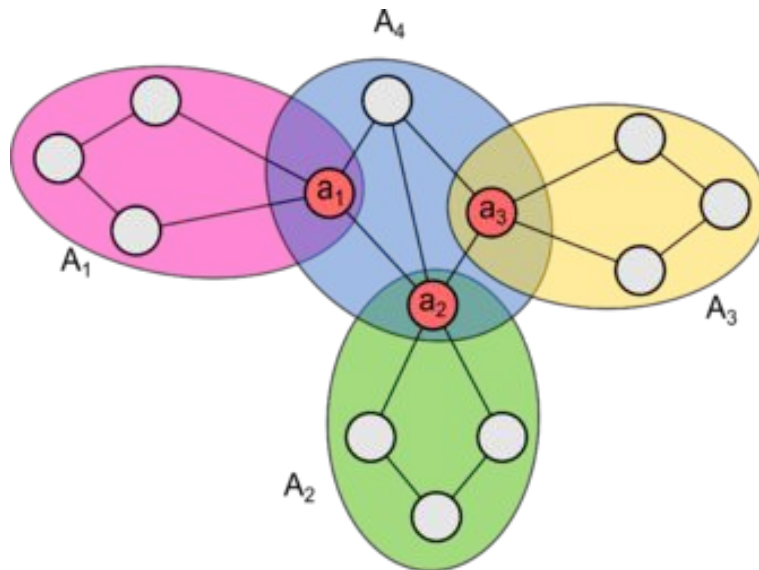
Два ребра в неориентированном графе  $G$  **вершинно двусвязны**, если существуют вершинно непересекающиеся пути, соединяющие их концы.

**Вершинная двусвязность** – отношение эквивалентности на ребрах.



# Граф: вершинная двусвязность

Ребра неориентированного графа разбиваются на **компоненты вершинной двусвязности**.

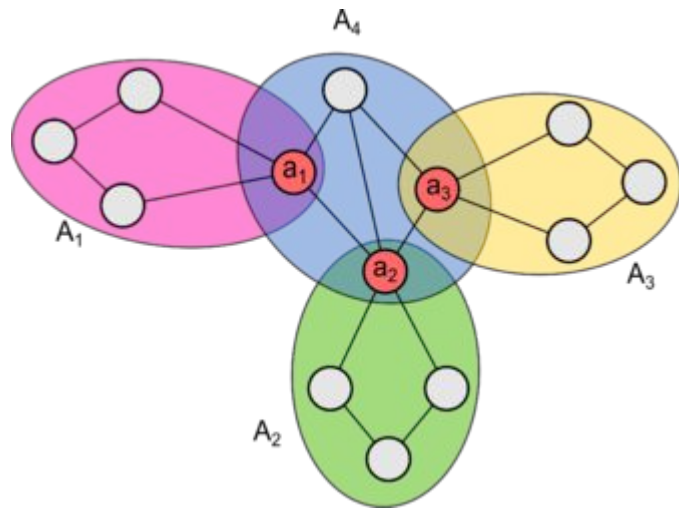


## Граф: точка сочленения

**Точка сочленения** – вершина, при удалении которой вместе с инцидентными ей ребрами, компонента связности распадается.

**Точка сочленения** – вершина, инцидентная рёбрам, принадлежащим двум или более компонентам вершинной двусвязности.

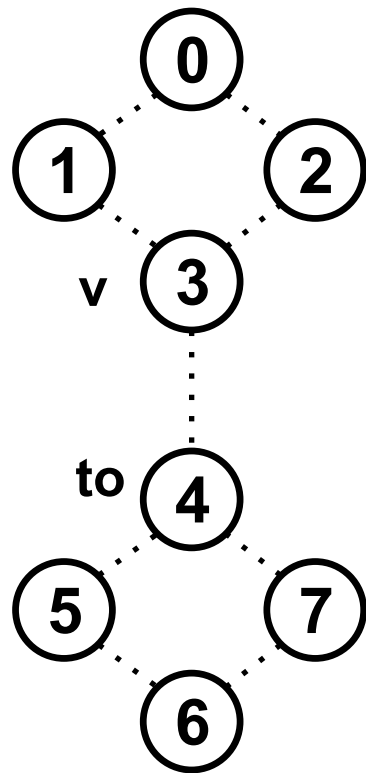
Эти определения эквивалентны.



## Граф: поиск мостов

Если из вершины **to** или ее потомком нет ребра в вершину **v** или в ее родителей, то ребро **(v, to)** – мост.

Мы этим условием проверяем, нет ли другого пути из **v** в **to**, кроме как спуск по ребру **(v, to)** дерева обхода в глубину.



# Граф: поиск мостов

**Проход по дереву поиска в обратную сторону**

Пропускаем ребро

**Критерий обратного ребра**

Берем минимальное время захода

**Ребро дерева поиска**

Обрабатываем всех потомков  
вершины to

Берем минимальное время захода  
из всех потомков to

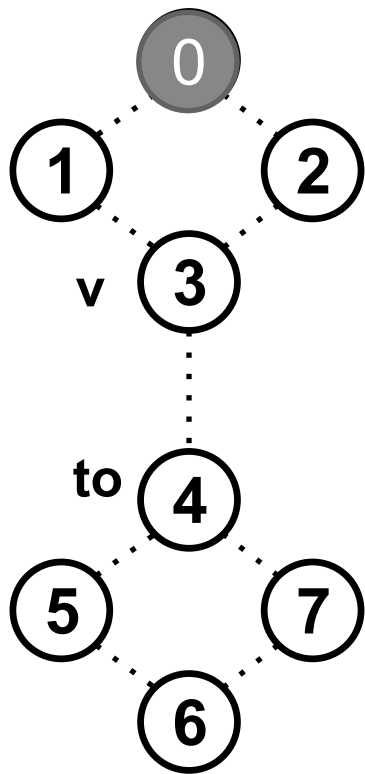
**Критерий моста**

Если время входа в to и в любого  
из её потомком больше времени  
входа в v.

```
void dfs( int v, int p = -1 ) {
    used[v] = true;
    entry[v] = lowest[v] = time++;
    for( int i = 0; i < g[v].size(); ++i ) {
        int to = g[v][i];
        if( to == p )
            continue;
        if( used[to] )
            lowest[v] = min( lowest[v], entry[to] );
        else {
            dfs( to, v );
            lowest[v] = min( lowest[v], lowest[to] );
            if( lowest[to] > entry[v] )
                IS_BRIDGE( v, to );
        }
    }
}

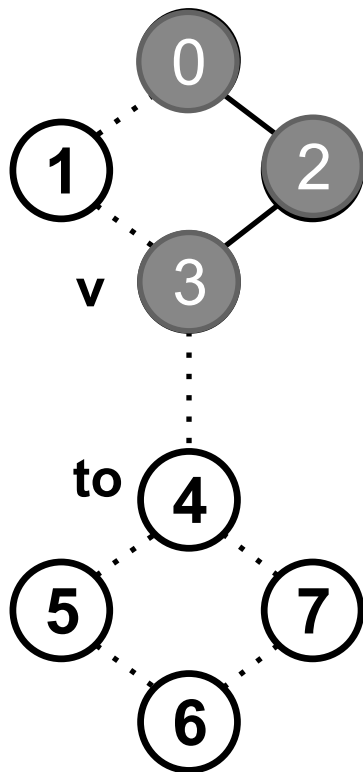
void find_bridges() {
    time = 0;
    for( int i = 0; i < n; ++i )
        used[i] = false;
    for( int i = 0; i < n; ++i )
        if( !used[i] )
            dfs( i );
}
```

# Граф: поиск мостов



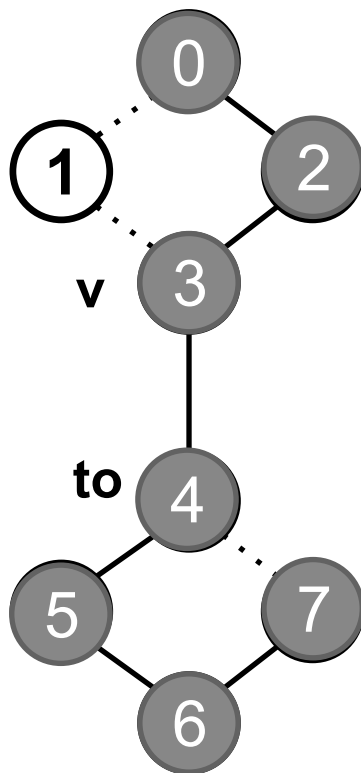
	used		entry		lowest
0	1	0	0	0	0
1	0	1		1	
2	0	2		2	
3	0	3		3	
4	0	4		4	
5	0	5		5	
6	0	6		6	
7	0	7		7	

# Граф: поиск мостов



	used		entry		lowest
0	1	0	0	0	0
1	0	1		1	
2	1	2	1	2	1
3	1	3	2	3	2
4	0	4		4	
5	0	5		5	
6	0	6		6	
7	0	7		7	

# Граф: поиск мостов



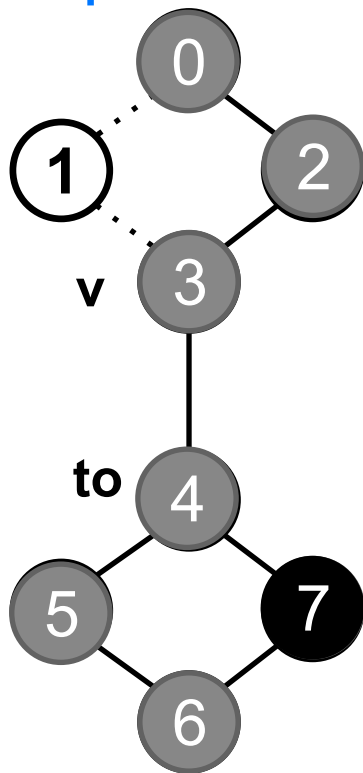
	used		entry		lowest
0	1	0	0	0	0
1	0	1		1	
2	1	2	1	2	1
3	1	3	2	3	2
4	1	4	3	4	3
5	1	5	4	5	4
6	1	6	5	6	5
7	1	7	6	7	3

```
void dfs( int v, int p = -1 ) {
    used[v] = true;
    entry[v] = lowest[v] = time++;
    for( int i = 0; i <
g[v].size(); ++i ) {
        int to = g[v][i];
        if( to == p ) continue;
        if( used[to] )
            lowest[v] = min(
lowest[v], entry[to] );
        else {
            dfs( to, v );
            lowest[v] = min(
lowest[v], lowest[to]);
            if( lowest[to] >
entry[v] )
                IS_BRIDGE( v, to );
        }
    }
}

void find_bridges() {
    time = 0;
    for( int i = 0; i < n; ++i )
        used[i] = false;
    for( int i = 0; i < n; ++i )
        if( !used[i] )
            dfs( i );
}
```



# Граф: поиск мостов

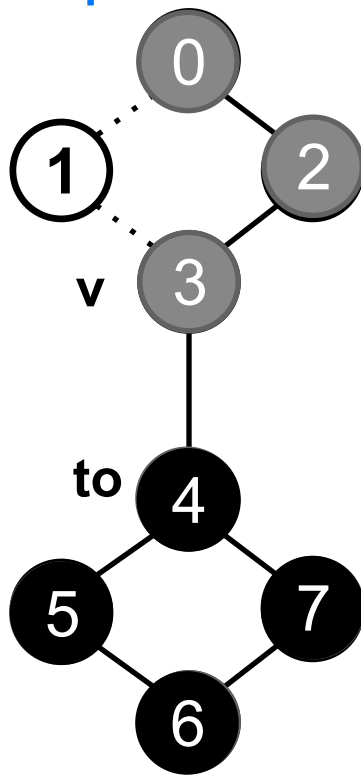


	used		entry		lowest
0	1	0	0	0	0
1	0	1		1	
2	1	2	1	2	1
3	1	3	2	3	2
4	1	4	3	4	3
5	1	5	4	5	4
6	1	6	5	6	3
7	1	7	6	7	3

```
void dfs( int v, int p = -1 ) {
    used[v] = true;
    entry[v] = lowest[v] = time++;
    for( int i = 0; i <
g[v].size(); ++i ) {
        int to = g[v][i];
        if( to == p ) continue;
        if( used[to] )
            lowest[v] = min(
lowest[v], entry[to] );
        else {
            dfs( to, v );
            lowest[v] = min(
lowest[v], lowest[to]);
            if( lowest[to] >
entry[v] )
                IS_BRIDGE( v, to );
        }
    }
}

void find_bridges() {
    time = 0;
    for( int i = 0; i < n; ++i )
        used[i] = false;
    for( int i = 0; i < n; ++i )
        if( !used[i] )
            dfs( i );
}
```

# Граф: поиск мостов



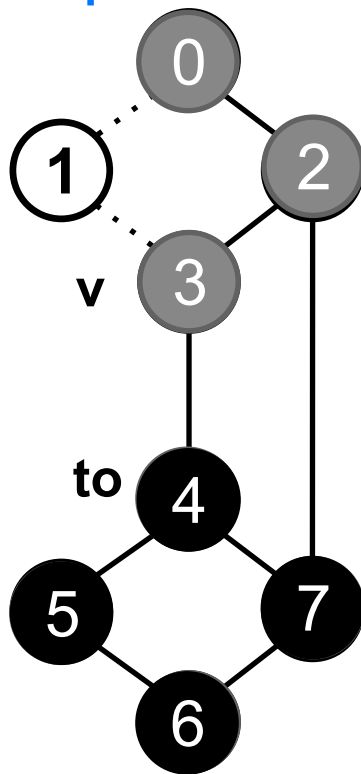
	used		entry		lowest
0	1	0	0	0	0
1	0	1		1	
2	1	2	1	2	1
3	1	3	2	3	2
4	1	4	3	4	3
5	1	5	4	5	3
6	1	6	4	6	3
7	1	7	5	7	3

lowest[to] = 3  
entry[v] = 2  
lowest[to] > entry[v]  
Мост найден

```
void dfs( int v, int p = -1 ) {
    used[v] = true;
    entry[v] = lowest[v] = time++;
    for( int i = 0; i <
g[v].size(); ++i ) {
        int to = g[v][i];
        if( to == p ) continue;
        if( used[to] )
            lowest[v] = min(
lowest[v], entry[to] );
        else {
            dfs( to, v );
            lowest[v] = min(
lowest[v], lowest[to]);
            if( lowest[to] >
entry[v] )
                IS_BRIDGE( v, to );
        }
    }
}

void find_bridges() {
    time = 0;
    for( int i = 0; i < n; ++i )
        used[i] = false;
    for( int i = 0; i < n; ++i )
        if( !used[i] )
            dfs( i );
}
```

# Граф: поиск мостов



	used		entry		lowest
0	1	0	0	0	0
1	0	1		1	
2	1	2	1	2	1
3	1	3	2	3	2
4	1	4	3	4	1
5	1	5	4	5	1
6	1	6	5	6	1
7	1	7	6	7	1

lowest[to] = 1  
entry[v] = 2  
lowest[to] < entry[v]  
Моста нет

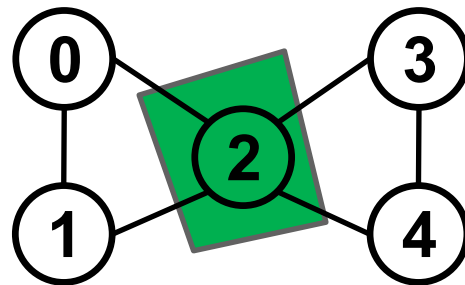
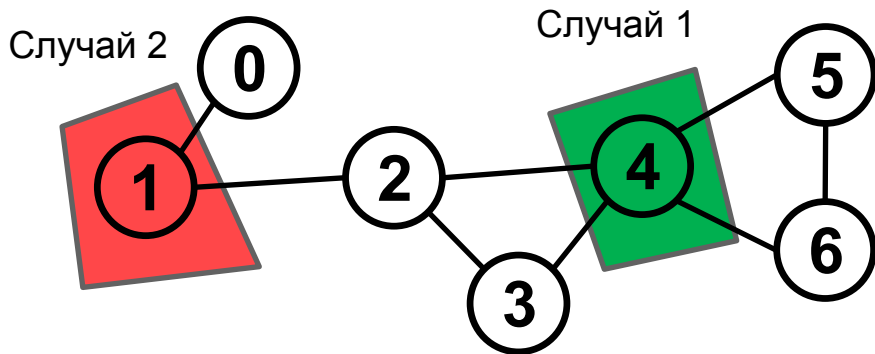
```
void dfs( int v, int p = -1 ) {
    used[v] = true;
    entry[v] = lowest[v] = time++;
    for( int i = 0; i <
g[v].size(); ++i ) {
        int to = g[v][i];
        if( to == p ) continue;
        if( used[to] )
            lowest[v] = min(
lowest[v], entry[to] );
        else {
            dfs( to, v );
            lowest[v] = min(
lowest[v], lowest[to]);
            if( lowest[to] >
entry[v] )
                IS_BRIDGE( v, to );
        }
    }
}

void find_bridges() {
    time = 0;
    for( int i = 0; i < n; ++i )
        used[i] = false;
    for( int i = 0; i < n; ++i )
        if( !used[i] )
            dfs( i );
}
```

# Граф: поиск точек сочленения

Рассматриваются два случая:

- 1) Вершина  $v$  не корень. Если из вершины  $to$  или ее потомком нет ребра в *родителей*  $v$ , то вершина  $v$  – **точка сочленения**.
- 2) Вершина  $v$  – корень. Тогда она является **точкой сочленения**, если в дереве обхода в глубину имеет более одного потомка.



# Граф: поиск точек сочленения

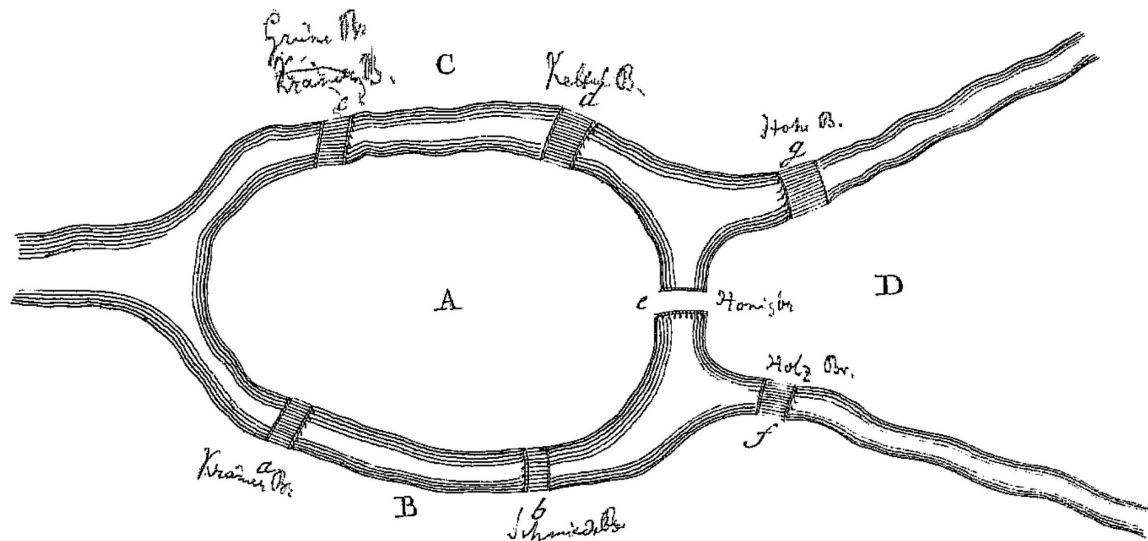
```
void dfs( int v, int p = -1 ) {
    used[v] = true;
    entry[v] = lowest[v] = time++;
    int children = 0;
    for( int i = 0; i < g[v].size(); ++i ) {
        int to = g[v][i];
        if( to == p )
            continue;
        if( used[to] )
            lowest[v] = min( lowest[v], entry[to] );
        else {
            dfs( to, v );
            lowest[v] = min( lowest[v], lowest[to] );
            if( lowest[to] >= entry[v] && p != -1 )
                IS_CUTPOINT( v, to );
            children++;
        }
    }
    if( children > 1 && p == -1 )
        IS_CUTPOINT( v );
}

void find_cutpoints() {
    time = 0;
    for( int i = 0; i < n; ++i )
        used[i] = false;
    for( int i = 0; i < n; ++i )
        if( !used[i] )
            dfs( i );
}
```

# Эйлеров граф

**Задача Эйлера:** Найти путь (цикл), проходящий по всем ребрам графа один раз.

Кёнигсбергский мосты:



# Эйлеров граф

**Эйлеров путь** – это путь, проходящий по всем ребрам графа, притом по одному разу.

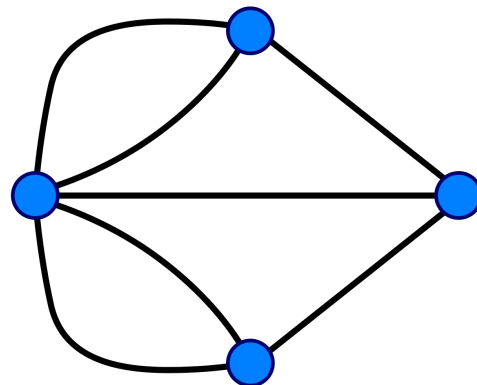
**Эйлеров цикл** – это замкнутый эйлеров путь.

**Эйлеров граф** – это граф, содержащий эйлеров цикл.

**Полуэйлеров граф** – это граф, содержащий эйлеров путь.

$G$  – связный неориентированный граф.

- Эйлеров путь существует тогда и только тогда, когда в  $G$  не более двух нечетных вершин и они являются началом и концом пути.
- Эйлеров цикл существует тогда и только тогда, когда в  $G$  все вершины четные.

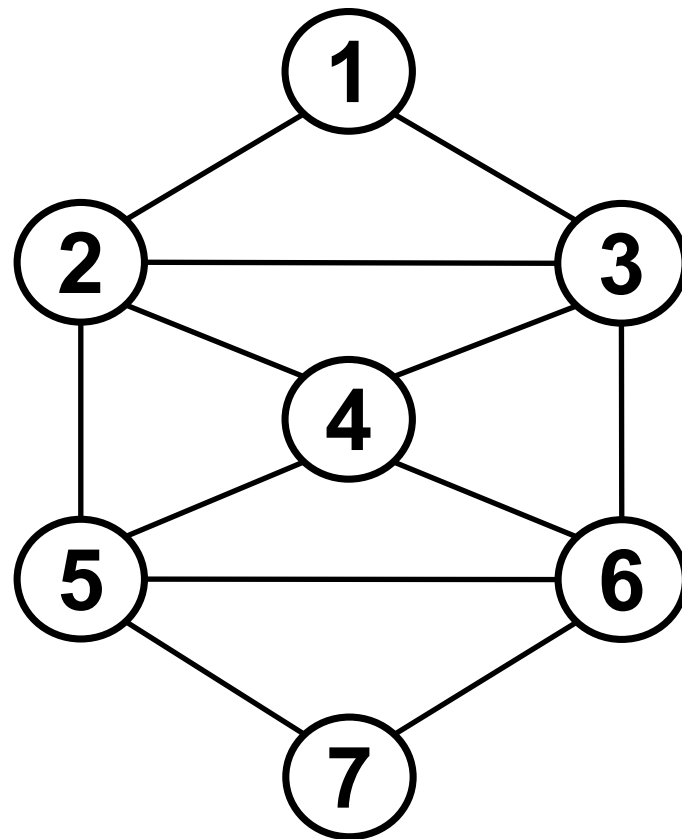


# Эйлеров граф

## Поиск эйлерова пути

Алгоритм запускаем из вершины с нечетной степенью

1. Перебираем все ребра, выходящие из  $V$ 
  1. Удаляем это ребро из графа
  2. Вызываем п.1 для второго конца этого ребра
2. Добавляем  $V$  в ответ





## Домашнее задание #08

- в процессе



# Напоминание отметиться на портале Vol 2

+ ОСТАВИТЬ ОТЗЫВ ПОСЛЕ ЛЕКЦИИ

vk образование

БлогиЛюдиПрограммаВакансииРасписание

Q<

VK

Техно

Открыт приём заявок!

чт, 8 сентября

Нет занятий

пт, 9 сентября

18:00 Углубленный Py... с3  
Введение в Python, основные  
понятия, тестирование  
Г. Кандауров

сб, 10 сентября

Нет занятий

вс, 11 сентября

Нет занятий

пн, 12 сентября

Нет занятий

Углубленный Python

↓ 0 ↑

Блог для материалов по курсу "Углубленный Python"

57 читателей, 2 топика

ПодписатьсяСоздать топик

Поиск по авторам, заголовку и тексту топика...

Найти

Добро пожаловать на курс!

Углубленный Python

ИзменитьУдалить

Всем привет и добро пожаловать на курс по углубленному изучению Python!

Прямой эфир

МоиВсе

Геннадий Кандауров час назад  
Углубленный Python → Добро пожаловать  
на курс! 0

Екатерина Черкасова 7 дней назад  
Стажировка → Приглашаем мобильных,  
фронтенд- и бэкэнд-разработчиков на  
Weekend Offer! 0

Дарья Вовченко 9 дней назад  
Углубленный Python → Добро пожаловать  
в образовательные проекты VK  
Образование! 0

Дарья Вовченко 9 дней назад  
Разработка веб-сервисов на

Спасибо за  
внимание

