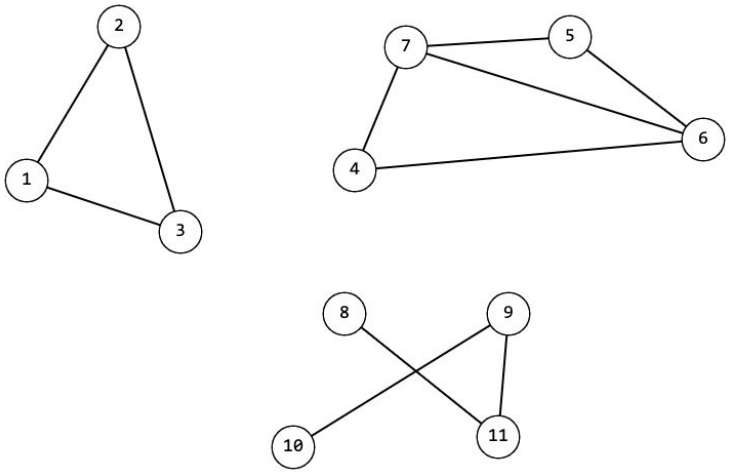
Графы

Семинар

Поиск компонент связности

Дан граф. Необходимо подсчитать количество компонент связности.



Поиск компонент связности

0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0

1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0

1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0

0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0

0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0

0, 0, 0, 1, 1, 0, 1, 0, 0, 0, 0

0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0

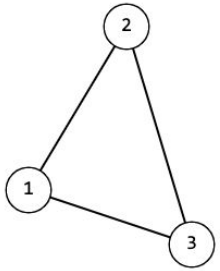
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1

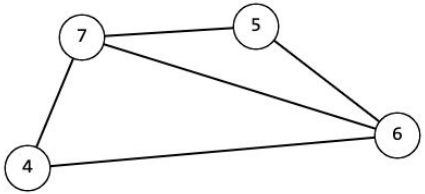
0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1

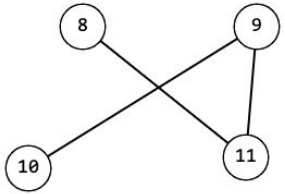
0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0

0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0

1: 2, 3
2: 1, 3
3: 1, 2
4: 6, 7
5: 6, 7
6: 4, 5, 7
7: 4, 5, 6
8: 11
9: 10, 11
10: 9
11: 8, 9

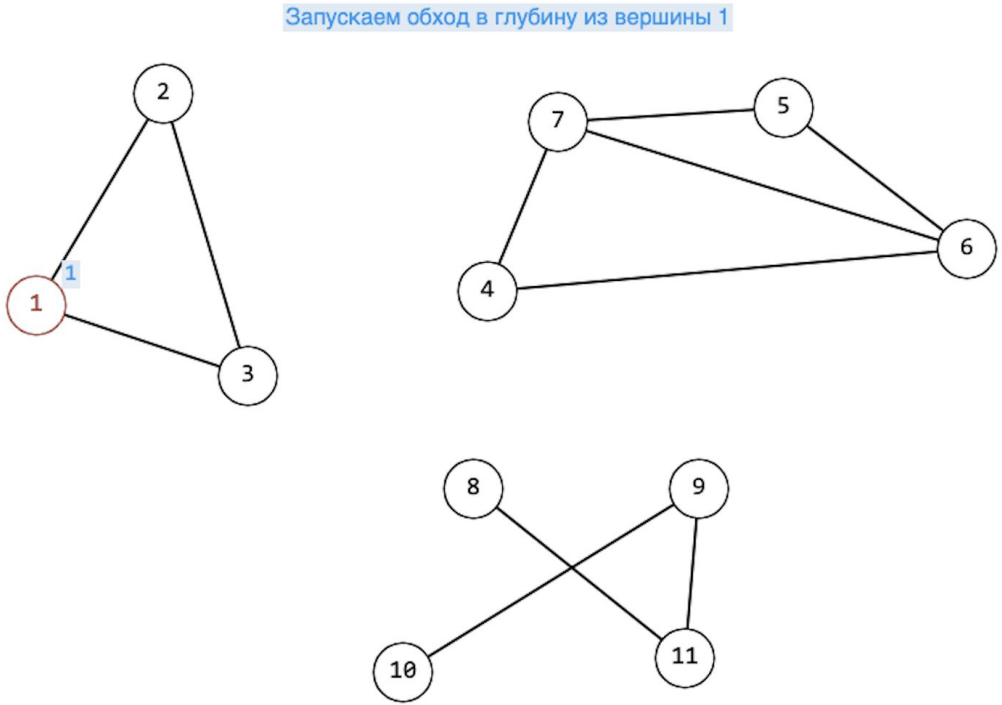




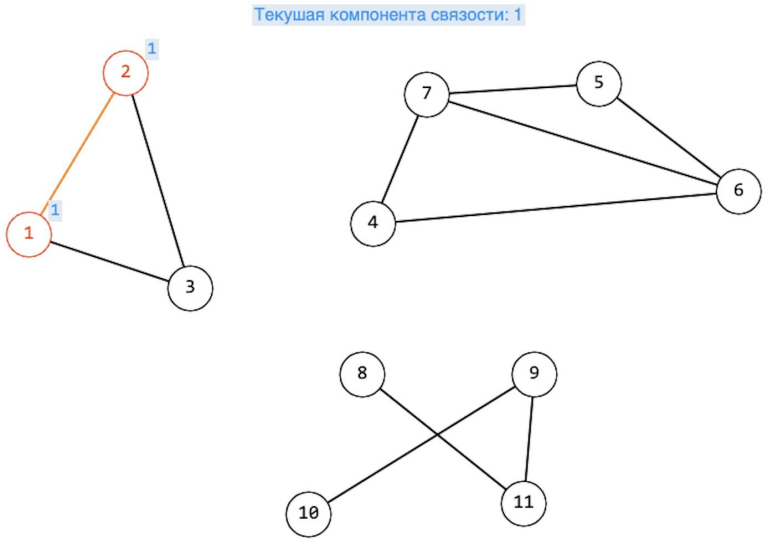


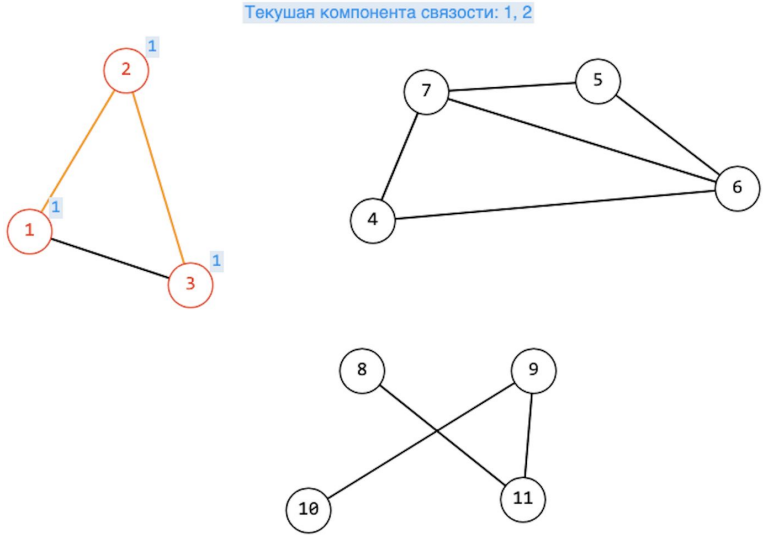
Поиск компонент связности

- Необходимо последовательно запускать dfs от каждой не посещенной вершины.
- Заводим массив visited для отслеживания посещения вершин
- Если после запуска dfs в графе есть вершины, которых нет в массиве visited, то запускаем от любой такой вершины обход в глубину
- После каждого запуска dfs инкрементируем счетчик количества компонент связности.

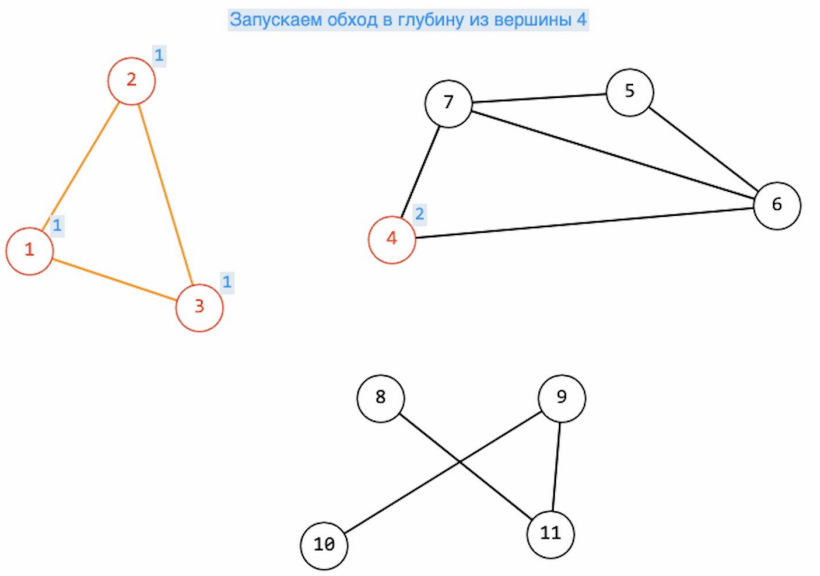


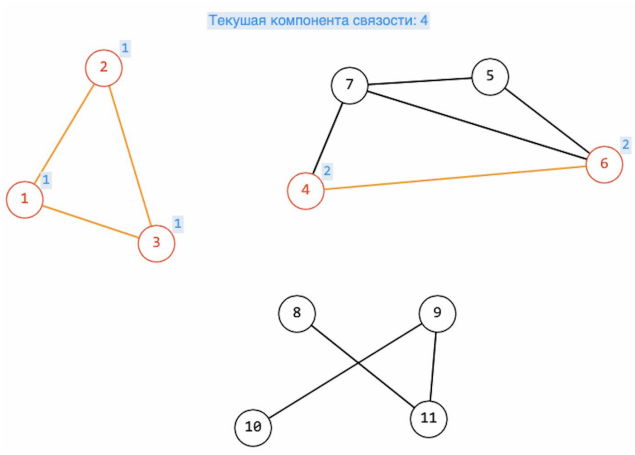
Поиск компонент связности

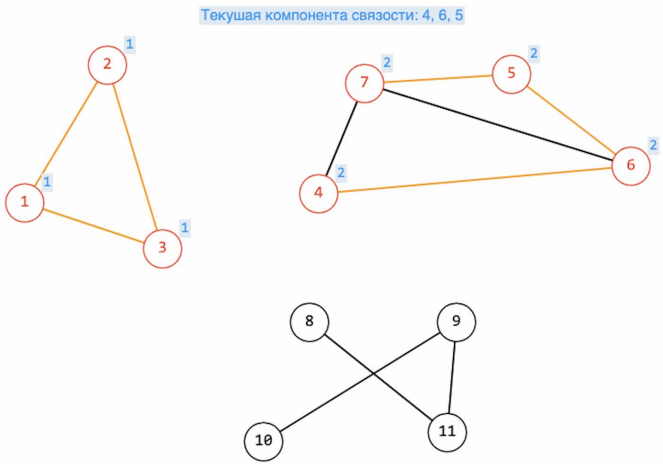




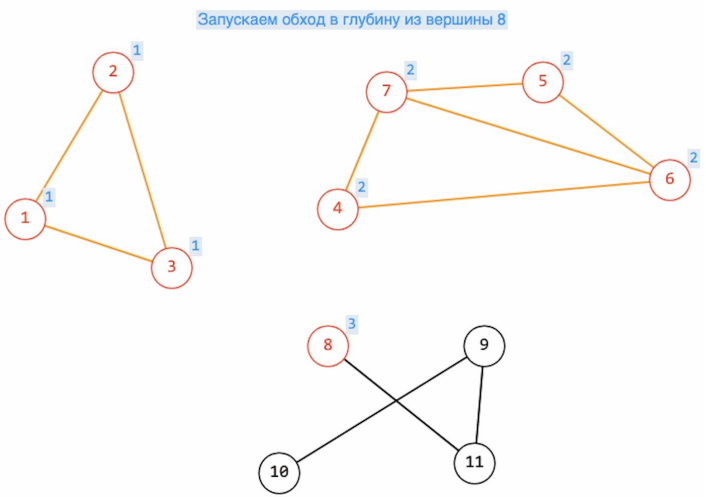
Поиск компонент связности

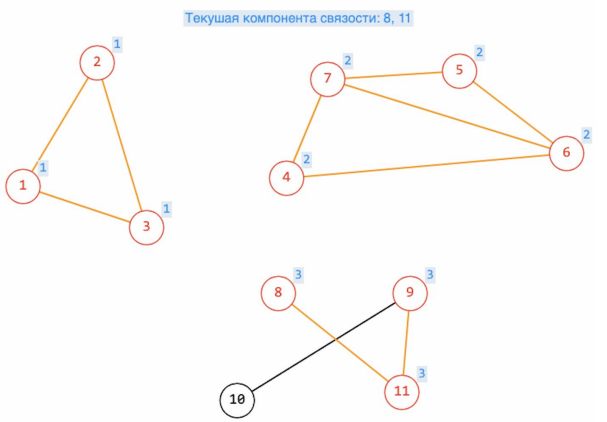


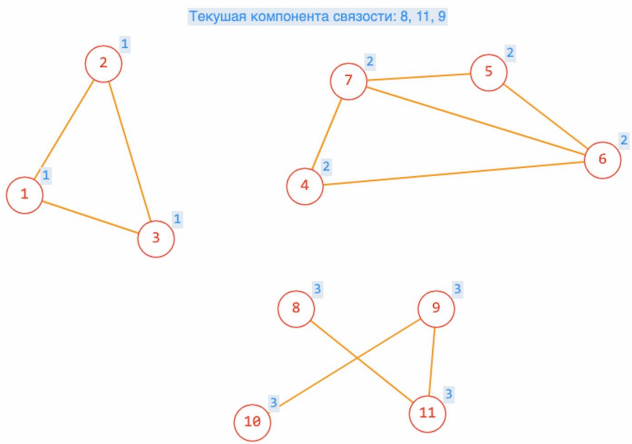




Поиск компонент связности







Поиск компонент связности

```
graph = {
   1: [2, 3],
   2: [1, 3],
   3: [1, 2],
   4: [6, 7],
   5: [6, 7],
   6: [4, 5, 7],
   7: [4, 5, 6],
   8: [11],
   9: [10, 11],
   10: [9],
   11: [8, 9]
}
```

```
function find_connected_components(graph) {
   // инициализируем хеш таблицу для хранения
   // посещенных вершин
   // инициализируем массив для хранения
   // компонент связности
   // в цикле по графу для каждой
   // не посещенной вершины запускаем DFS
   // для каждого запуска создаем массив
   // в котором храним текущий подграф

   return connected_components
}

function dfs(graph, v, visited, component) {
   // отмечаем вершину посещенной
   // добавляем вершину в массив компоненты связности
   // идем в цикле по всем смежным вершинам
   // если какая-то из них не посещена
   // запускаем для нее dfs
}
```

Поиск компонент связности

```
graph = {
   1: [2, 3],
   2: [1, 3],
   3: [1, 2],
   4: [6, 7],
   5: [6, 7],
   6: [4, 5, 7],
   7: [4, 5, 6],
   8: [11],
   9: [10, 11],
   10: [9],
   11: [8, 9]
}
```

```
function find_connected_components(graph) {
   visited = map[int]bool
   for i = 1; i <= length(graph); i++ {
      visited[i] = false
   }
   connected_components = []
   // в цикле по графу для каждой
   // не посещенной вершины запускаем DFS
   // сигнатура обхода немного измениться
   // для каждого запуска создаем массив
   // в котором храним текущий подграф

   return connected_components
}

function dfs(graph, v, visited, component) {
   visited[v] = true
   component.append(v)
   // идем в цикле по всем смежным вершинам
   // если какая-то из них не посещена
   // запускаем для нее dfs
}
```

Поиск компонент связности

```
graph = {
   1: [2, 3],
   2: [1, 3],
   3: [1, 2],
   4: [6, 7],
   5: [6, 7],
   6: [4, 5, 7],
   7: [4, 5, 6],
   8: [11],
   9: [10, 11],
   10: [9],
   11: [8, 9]
}
```

```
function find_connected_components(graph) {
   visited = map[int]bool
   for i = 1; i <= length(graph); i++ {
      visited[i] = false
   }
   connected_components = []
   for i = 1; i <= length(graph); i++ {
      currentNode = graph[i]
      if !visited[currentNode]{
         // для каждого запуска создаем массив
         // в котором храним текущий подграф
         // добавляем этот массив в connected_components
      }
   }

   return connected_components
}

function dfs(graph, v, visited, component) {
   visited[v] = true
   component.append(v)
   // идем в цикле по всем смежным вершинам
   // если какая-то из них не посещена
   // запускаем для нее dfs
}
```

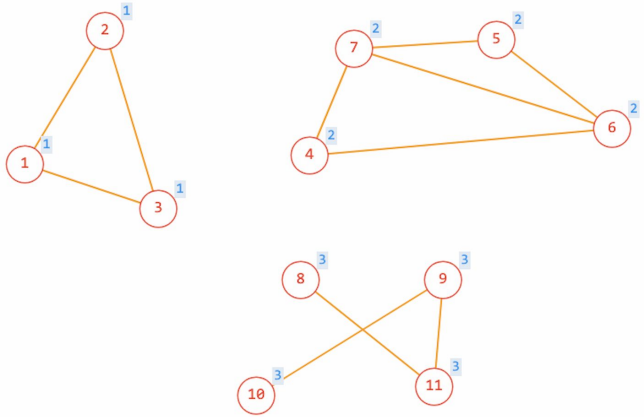
Поиск компонент связности

```
graph = {
   1: [2, 3],
   2: [1, 3],
   3: [1, 2],
   4: [6, 7],
   5: [6, 7],
   6: [4, 5, 7],
   7: [4, 5, 6],
   8: [11],
   9: [10, 11],
   10: [9],
   11: [8, 9]
}
```

```
function find_connected_components(graph) {
   visited = map[int]bool
   for i = 1; i <= length(graph); i++ {
      visited[i] = false
   }
   connected_components = []
   for i = 1; i <= length(graph); i++ {
      currentNode = graph[i]
      if !visited[currentNode]{
         component = []
         dfs(graph, currentNode, visited, component)
         connected_components.append(component)
      }
   }

   return connected_components
}

function dfs(graph, v, visited, component) {
   visited[v] = true
   component.append(v)
   for i in graph[v] {
      if !visited[i] {
         dfs(graph, i, visited, component)
      }
   }
}
```

Поиск компонент связности
Раскраска графа

- Не всегда есть смысл тащить отдельный массив для каждой компоненты
- У нас уже есть хеш мапа **visited** и в ней мы можем в качестве значения хранить “**цвет**” компоненты связности



Раскраска графа

```
function dfs(graph, v, visited, color) {
    visited[v] = color
    for i in graph[v] {
        if visited[i] == 0 {
            dfs(graph, i, visited, color)
        }
    }
}

function find_connected_components(graph) {
    visited = map[int]int
    for i = 1; i <= length(graph); i++ {
        visited[i] = 0
    }
    color = 0
    for i = 1; i <= length(graph); i++ {
        currentNode = graph[i]
        if visited[currentNode] == 0 {
            // помечаем цветом узел
        }
    }

    return visited
}
```

Раскраска графа

```
graph = {
   1: [2, 3],
   2: [1, 3],
   3: [1, 2],
   4: [6, 7],
   5: [6, 7],
   6: [4, 5, 7],
   7: [4, 5, 6],
   8: [11],
   9: [10, 11],
   10: [9],
   11: [8, 9]
}
```

{1: 1, 2: 1, 3: 1, 4: 2, 5: 2, 6: 2, 7: 2, 8: 3, 9: 3, 10: 3, 11: 3}

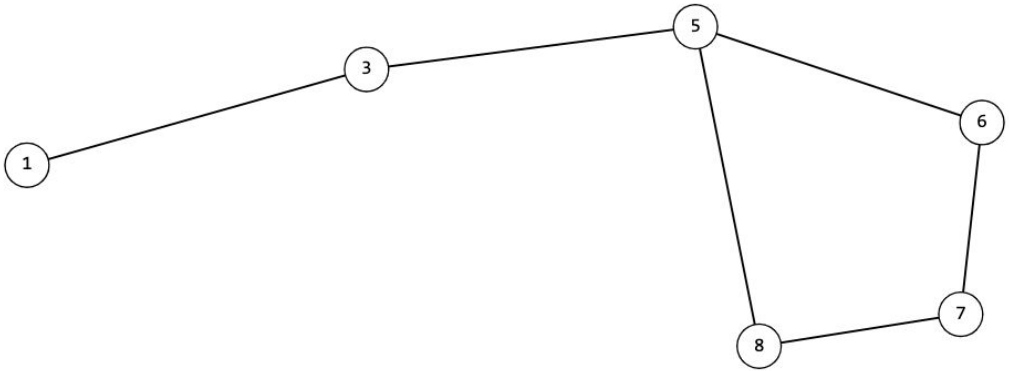
```
function dfs(graph, v, visited, color) {
   visited[v] = color
   for i in graph[v] {
      if visited[i] == 0 {
         dfs(graph, i, visited, color)
      }
   }
}

function find_connected_components(graph) {
   visited = map[int]int
   for i = 1; i <= length(graph); i++ {
      visited[i] = 0
   }
   color = 0
   for node in graph {
      if visited[node] == 0 {
         color++
         dfs(graph, node, visited, color)
      }
   }

   return visited
}
```

Поиск цикла в графе

Дан граф в виде списка вершин. Необходимо понять, есть ли в этом графе цикл

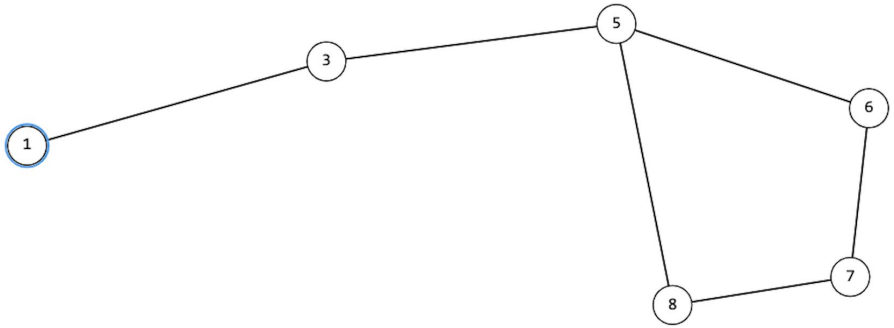
Поиск цикла в графе

- **DFS**: Мы используем поиск в глубину для обхода графа, начиная с каждой вершины. При этом помечаем вершины, которые уже были посещены.
- Обнаружение обратных рёбер: Если в процессе DFS соседская вершина уже была посещена и при этом не является родительской для текущей вершины - в графе есть цикл
- Возврат результата: Если при обходе графа находится цикл, функция возвращает **True**, иначе - **False**.

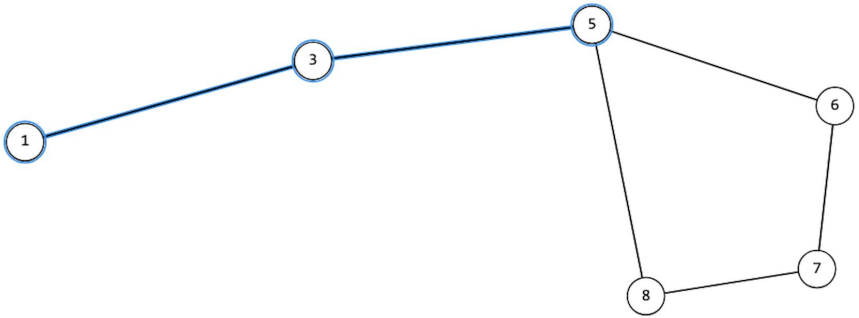


Поиск цикла в графе

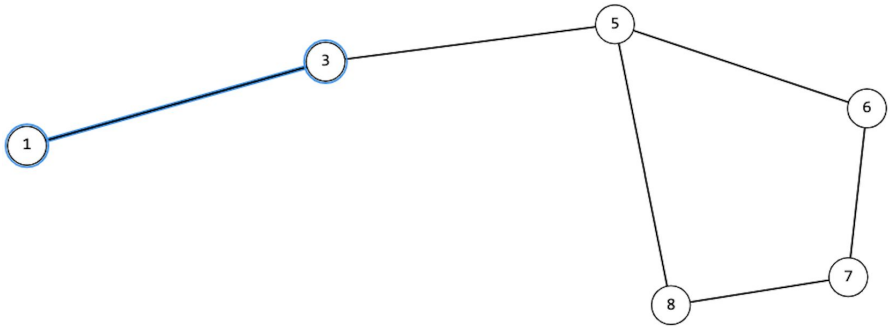
Запускаем обход в глубину из вершины 1



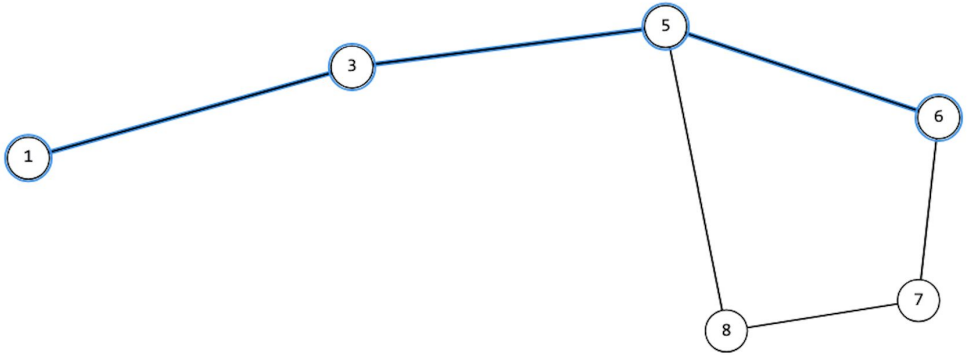
Запускаем обход в глубину из вершины 5



Запускаем обход в глубину из вершины 3

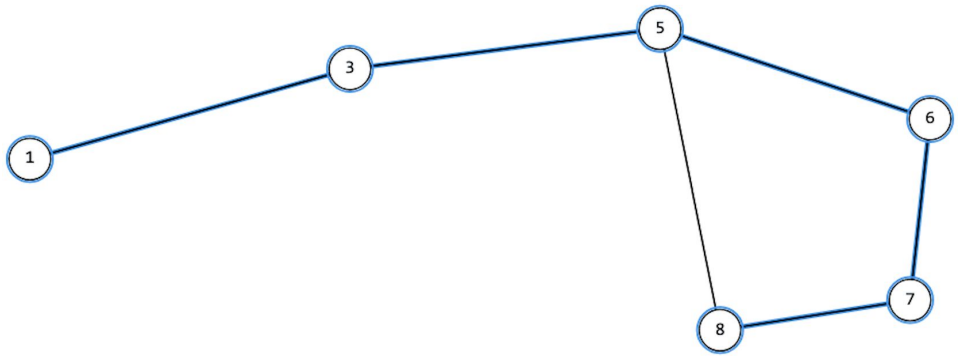


Запускаем обход в глубину из вершины 6

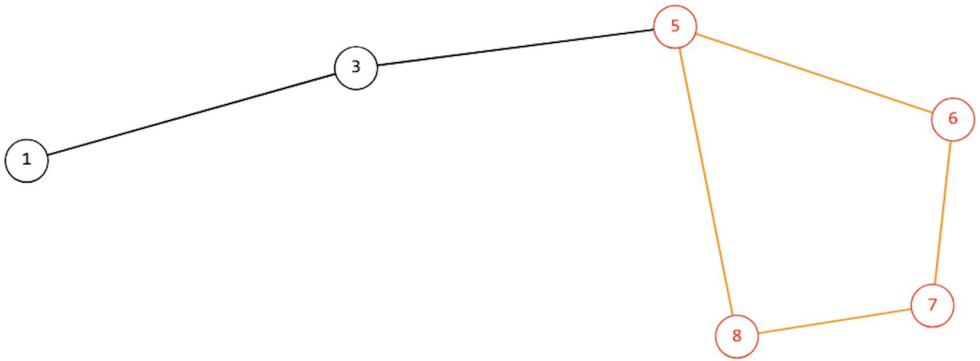


Поиск цикла в графе

Запускаем обход в глубину из вершины 8

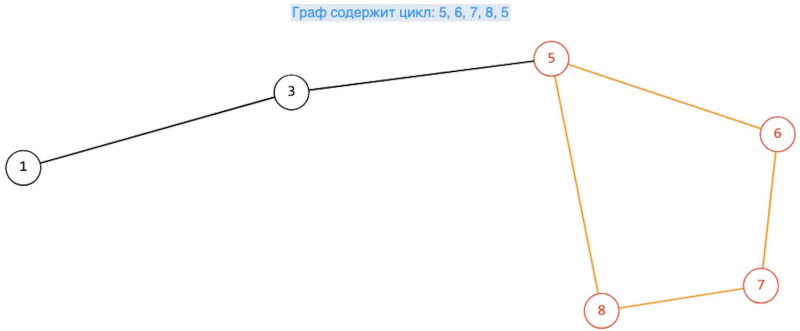


Граф содержит цикл: 5, 6, 7, 8, 5



Поиск цикла в графе

- Теперь при вызове **dfs** мы должны учитывать родительские вершины
- По-прежнему запускаем **dfs** от не посещенных вершин
- Если вершина в которую мы пришли была посещена и при этом не является родительской для текущей вершины ты мы говорим, что в таком графе есть цикл
- вершина **5** не является родительской для **8**, при этом когда мы переходим из 8 в 5, 5 уже находится в массиве **visited**



Поиск цикла в графе

```
function has_cycle(graph) {
    // создаем пустой массив
    // для отслеживания посещенных вершин
    // перебираем все вершины графа
    // если вершина еще не посещена
    // запускаем DFS для этой вершины
    // на первом запуске родительская вершина отсутствует
    return false
}

function dfs(graph, vertex, parent, visited) {
    // добавляем текущую вершину в множество посещенных
    // перебираем соседей текущей вершины
    // если сосед не является родительской вершиной,
    // чтобы избежать обратного перехода

    // если сосед уже посещен
    // или dfs для соседа вернул true
    // возвращаем true, так как мы нашли цикл
    return false
}
```

Поиск цикла в графе

```
function has_cycle(graph) {
   visited = []
   for vertex in graph {
      if vertex not in visited {
         // запускаем dfs для этой вершины
         // проверяем, что возвращает dfs
      }
   }
   return false
}


function dfs(graph, vertex, parent, visited) {
      visited.append(vertex)
      for neighbor in graph[vertex] {
            // если сосед не является родительской вершиной,
            // чтобы избежать обратного перехода

            // если сосед уже посещен
            // или dfs для соседа вернул true
            // возвращаем true, так как мы нашли цикл
      }

   return false
}
```

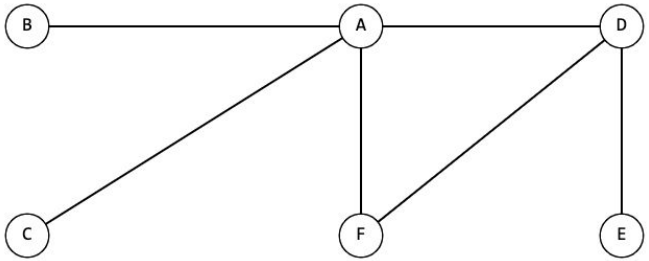
Поиск цикла в графе

```
function has_cycle(graph) {
    visited = []
    for vertex in graph {
        if vertex not in visited {
            if dfs(graph, vertex, null, visited) {
                return true
            }
        }
    }
    return false
}

function dfs(graph, vertex, parent, visited) {
    visited.append(vertex)
    for neighbor in graph[vertex] {
        if neighbor != parent {
            if neighbor in visited or dfs(graph, neighbor, vertex, visited) {
                return true
            }
        }
    }
    return false
}
```

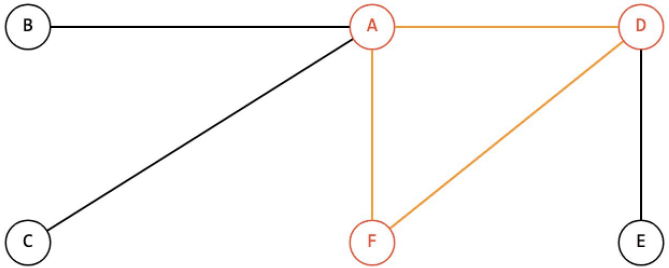
Является ли граф деревом

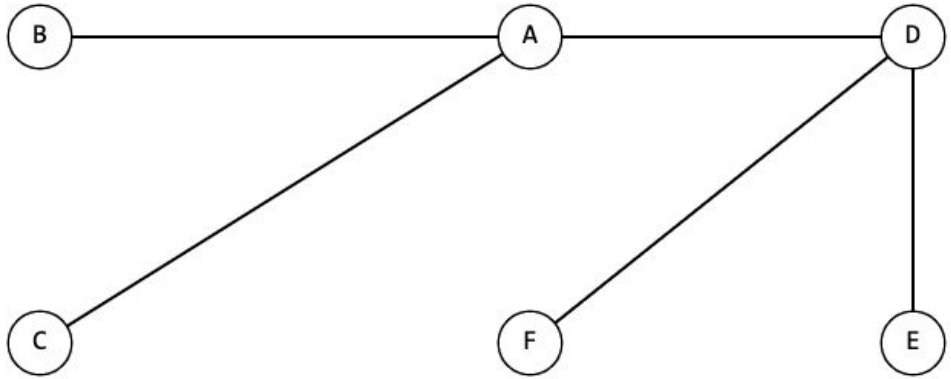
Является ли дерево графом

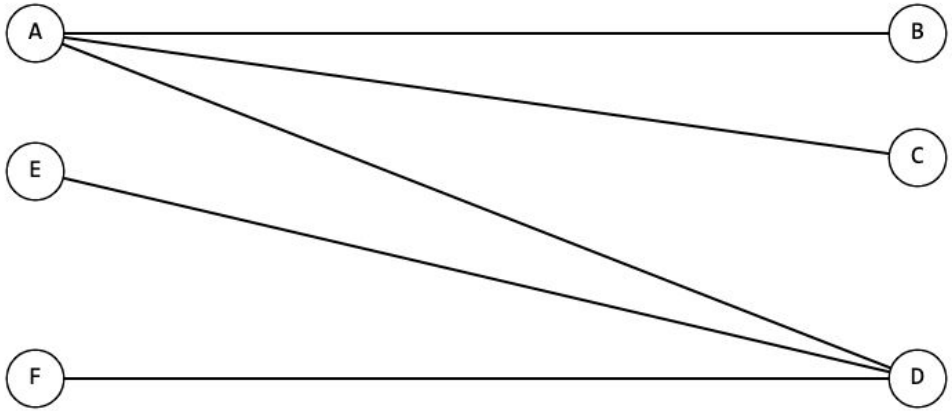
Какие условия должны быть выполнены?



Граф содержит цикл: A, D, F, A

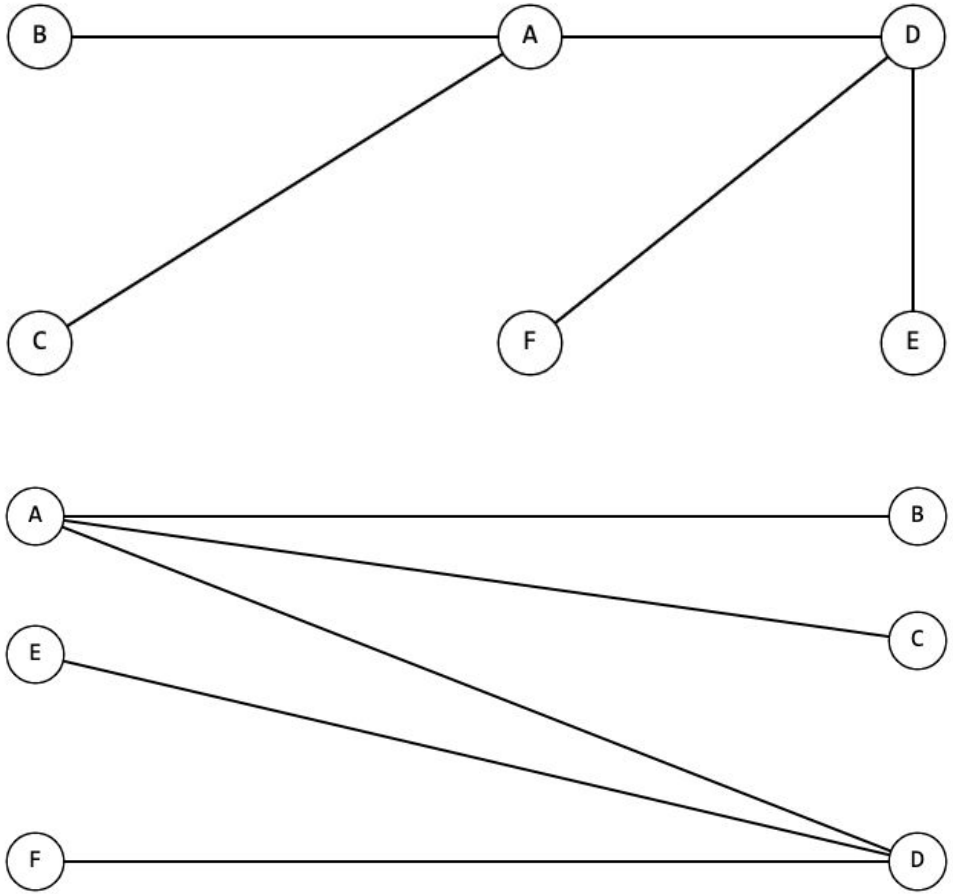






Является ли дерево графом

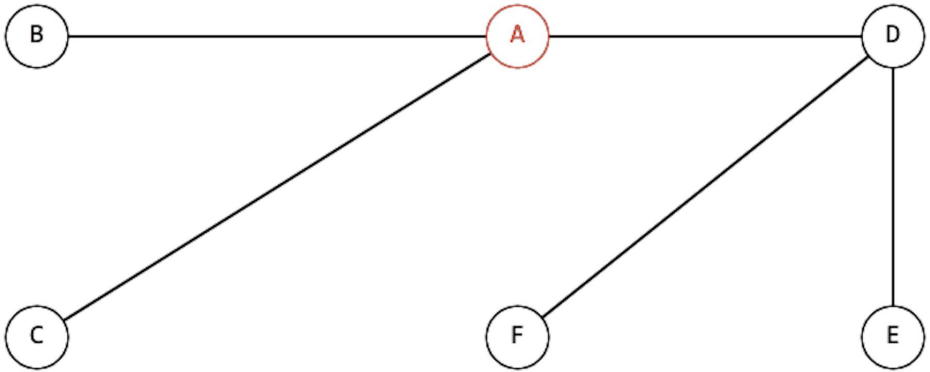
- Граф не должен содержать циклов
- Должен состоять из одной компоненты связности
- Решаем через **BFS**



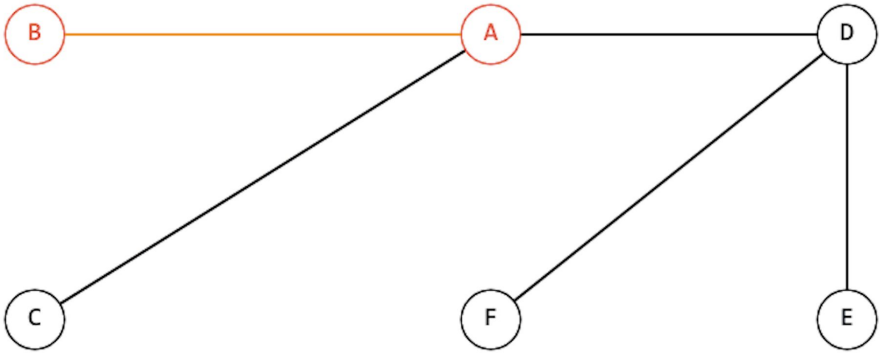
Является ли дерево графом

Начиная с вершины A обходим всех ее соседей, добавляя их в очередь

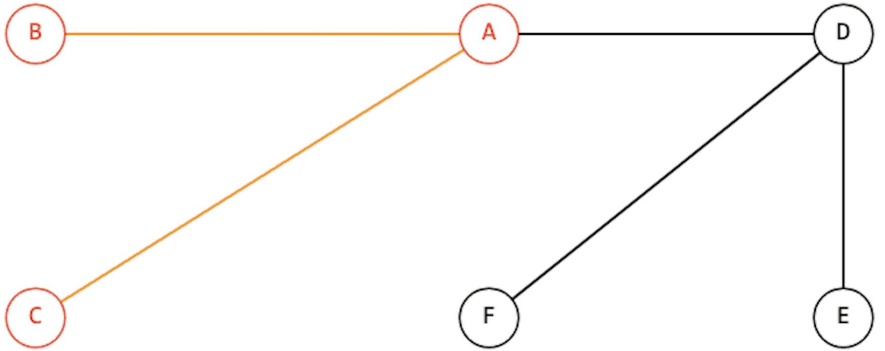
Обход в ширину: A



Обход в ширину: A, B

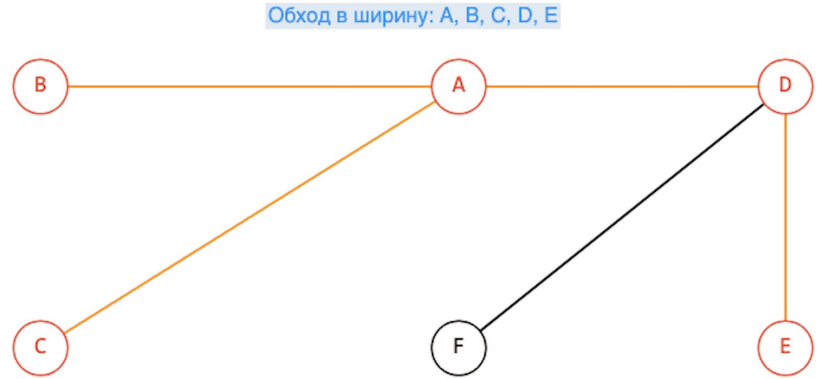


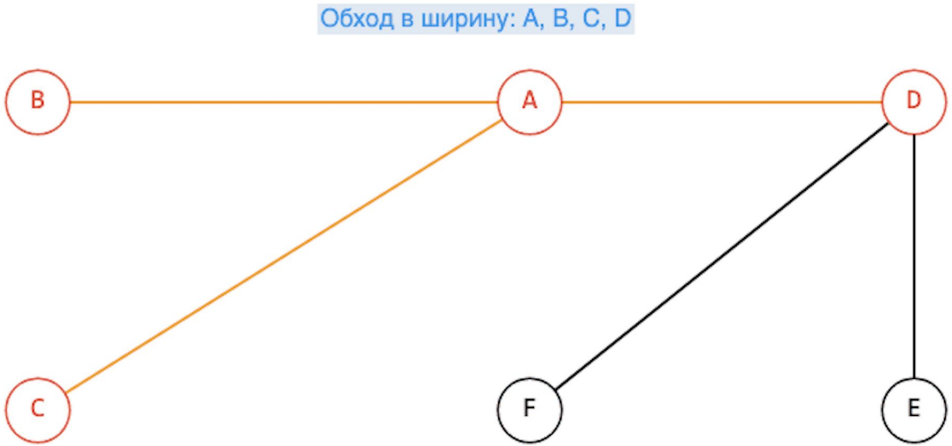
Обход в ширину: A, B, C

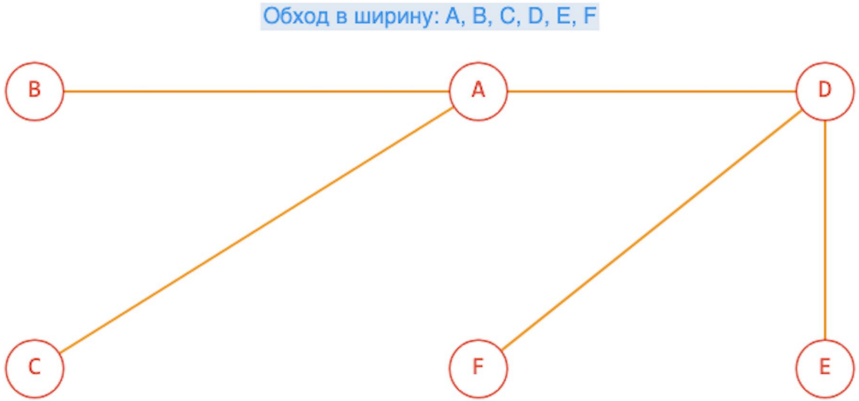


Является ли дерево графом

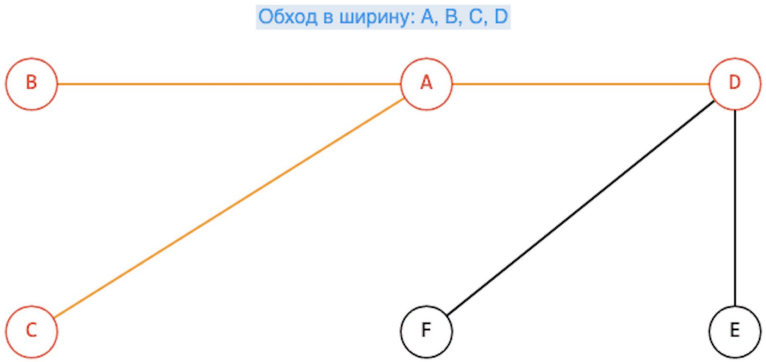
- Далее идем по соседям вершины A
- Записываем в очередь всех соседей вершины D
- Если из вершина F было бы ребро в вершину A и при этом A находится в массиве visited, то граф не является деревом





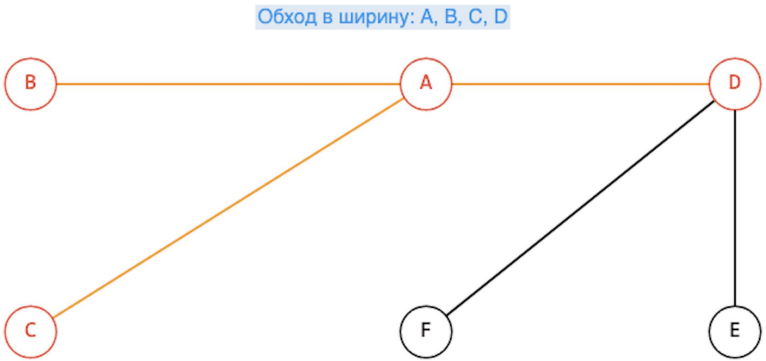


Является ли дерево графом



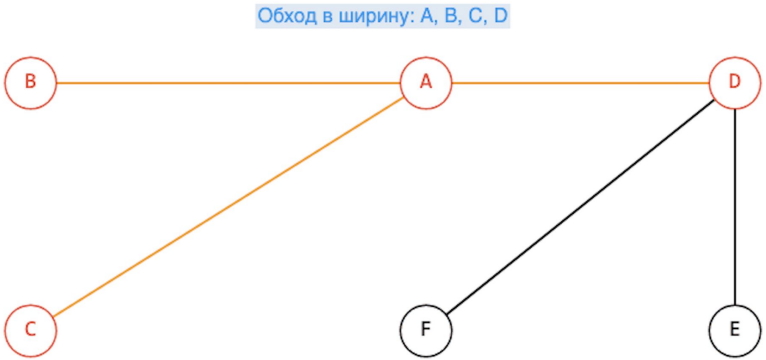

```
function is_tree(graph, start) {
  // создаем множество для отслеживания
  // уже посещенных вершин
  // используем очередь для обхода в ширину
  // и словарь (хэш-таблицу) для отслеживания
  // родительских вершин

  // идем в цикле по нашей очереди
  // заполняя массив visited
  while queue {
    // извлекаем из очереди
    // очередную вершину
    // помечаем вершину посещенной

    // создаем цикл по всем соседним вершинам

      // если сосед не посещен
      // добавляем его в очередь на BFS
      // и в массиве parent запоминаем,
      // что текущая вершина является для
      // нее родительской

      // если уже посещенная вершина не
      // является текущим родителем, значит
      // мы замкнули путь, а значит есть цикл
      // вспоминаем 8 и 5 или в нашем случае F и A
  }
  // Если все вершины достижимы и в графе нет циклов, то это дерево
  return ?
}
```

Является ли дерево графом




```
function is_tree(graph, start) {
  visited = []
  queue = [start]
  parent = {start: null}

  while queue {
    // извлекаем из очереди
    // очередную вершину

    // помечаем вершину посещенной
    // создаем цикл по всем
    // соседним вершинам
    for neighbor in graph[vertex] {
      // если сосед не посещен
      // добавляем его в очередь на BFS
      // и в массиве parent запоминаем,
      // что текущая вершина является для
      // нее родительской

        // Если уже посещенная вершина не
        // является текущим родителем, значит
        // мы замкнули путь, а значит есть цикл
    }
  }

  // Если все вершины достижимы и в графе нет циклов, то это дерево
  return ?
}
```

Является ли дерево графом



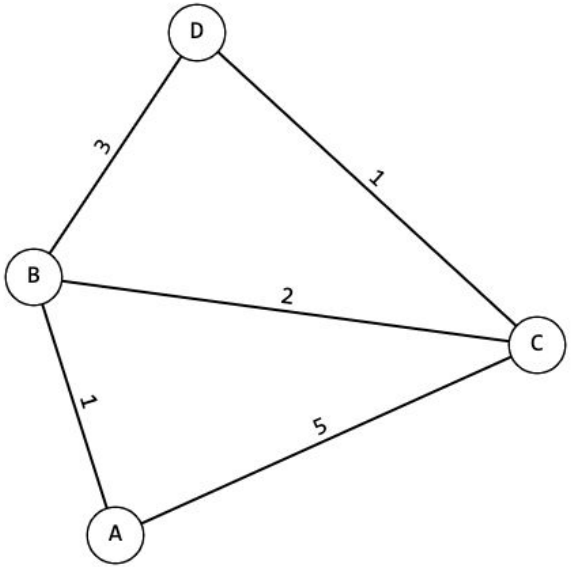

```
function is_tree(graph, start) {
    visited = []
    queue = [start]
    parent = {start: null}

    while queue {
        vertex = queue.pop()
        visited.append(vertex)
        for neighbor in graph[vertex] {
            // для узлов F и A
            if neighbor not in visited {
                queue.append(neighbor)
                parent[neighbor] = vertex
            } else {
                if parent[vertex] != neighbor:
                    return False
            }
        }
    }

    return length(visited) == length(graph)
}
```

Алгоритм Дейкстры

Поиск кратчайшего пути

Алгоритм Дейкстры

Необходимо найти кратчайший путь из заданной точки

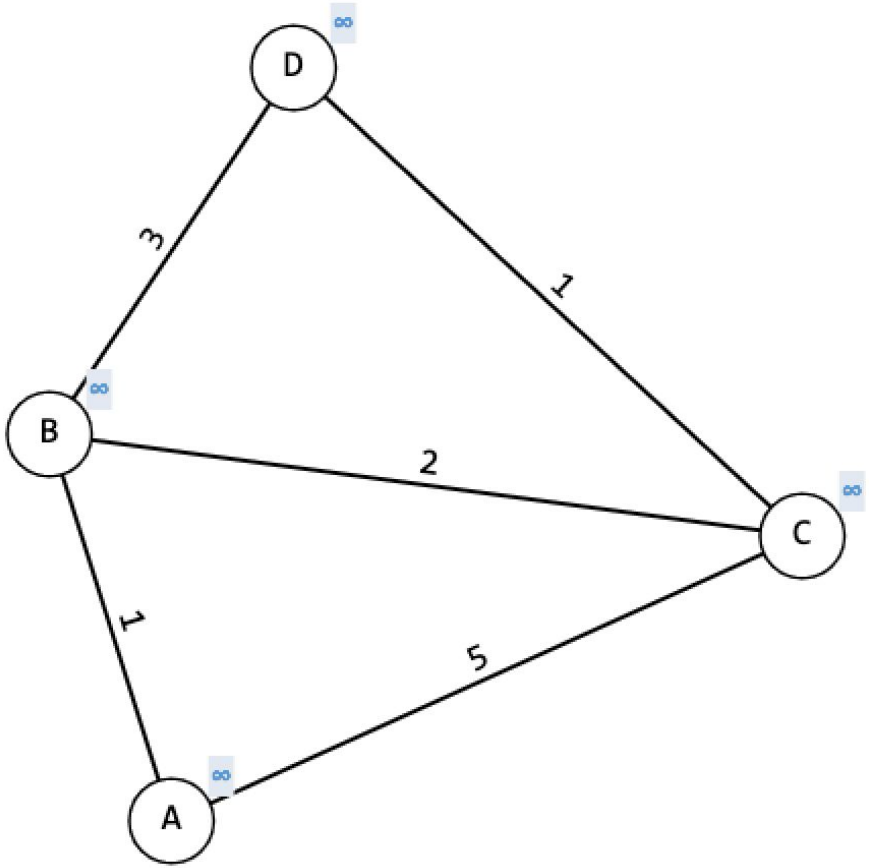


	A	B	C	D
A	0,	1,	5,	0
B	1,	0,	2,	3
C	5,	2,	0,	1
D	0,	3,	1,	0

```
graph = {
    'A': {'B': 1, 'C': 5},
    'B': {'A': 1, 'C': 2, 'D': 3},
    'C': {'A': 5, 'B': 2, 'D': 1},
    'D': {'B': 3, 'C': 1}
}
```

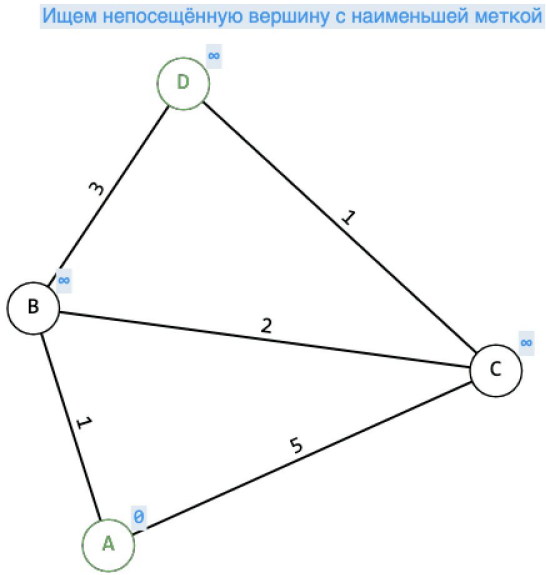
Алгоритм Дейкстры

- Каждой вершине проставляем значения - минимальный вес ребер, которые надо пройти, чтобы попасть в нее из стартовой
- Пока не начали проход по графу эти значения имеют максимально возможные значения
- Стартовая вершина имеет значение 0
- Наша задача при проходе обновлять эти значения

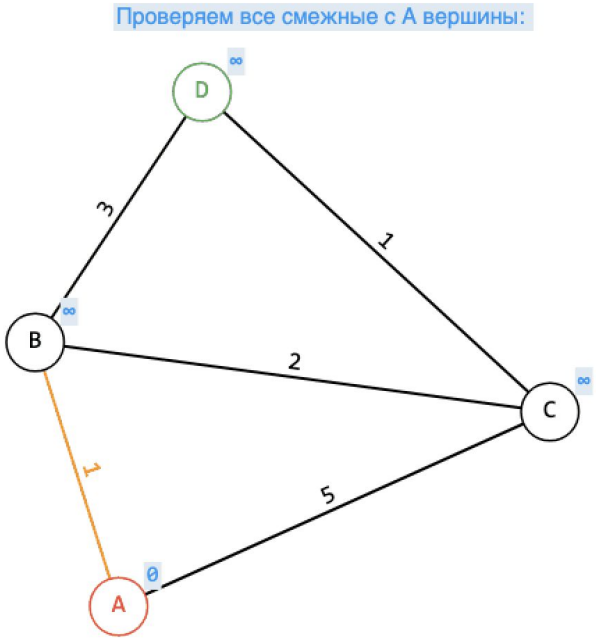


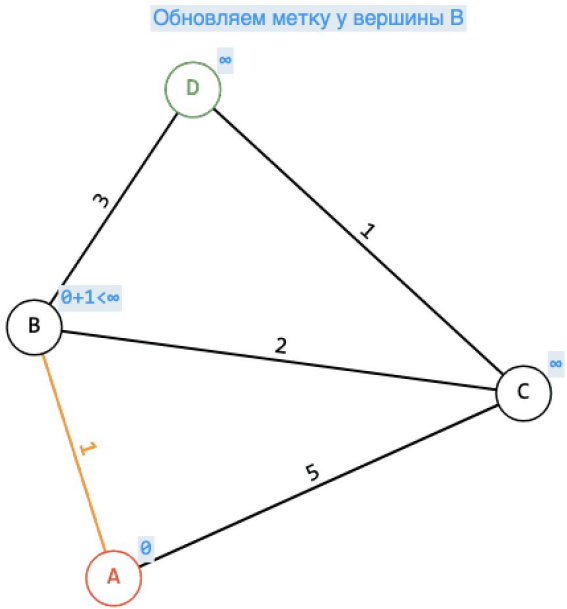
Алгоритм Дейкстры

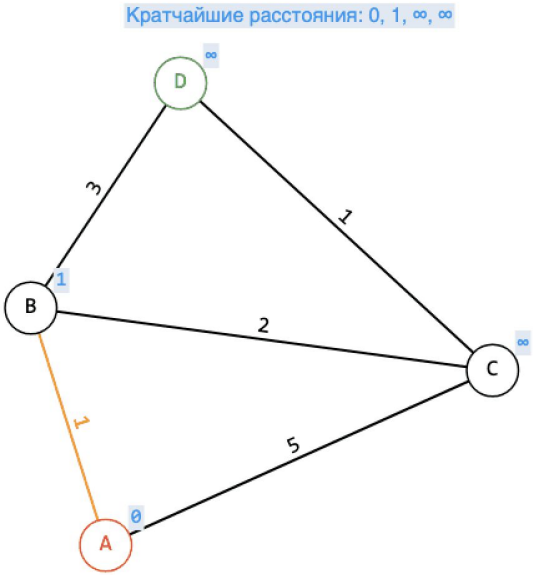
- Добавим вспомогательную функцию поиска вершины с наименьшим весом **vertexWithMinWeight**
- Будем запускать ее пока есть непосещенные вершины



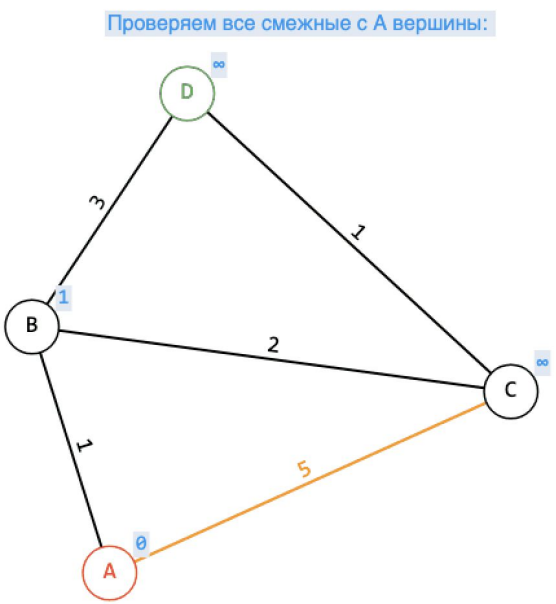
Алгоритм Дейкстры

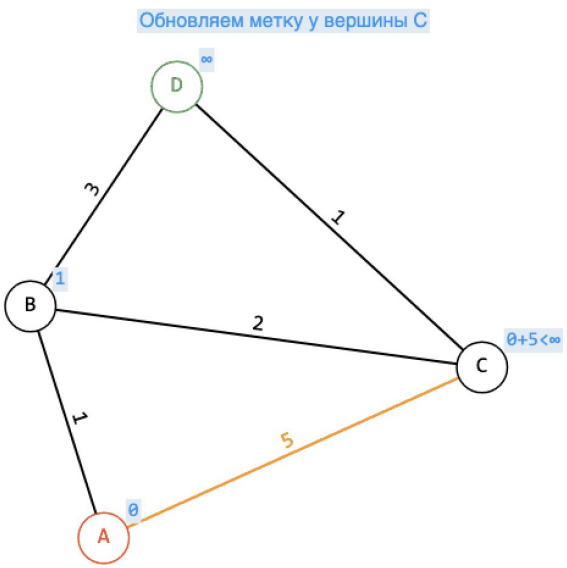


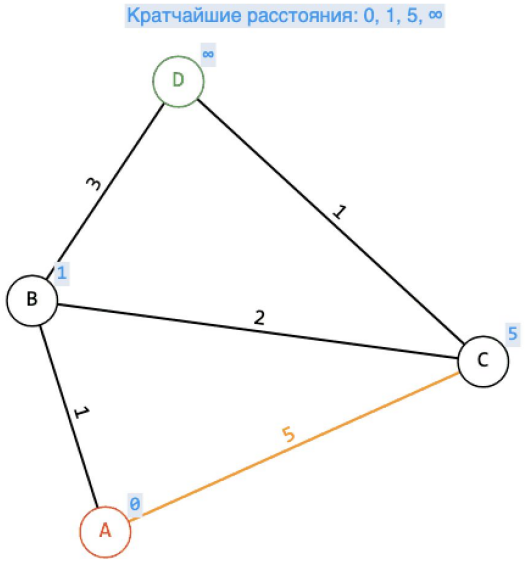




Алгоритм Дейкстры

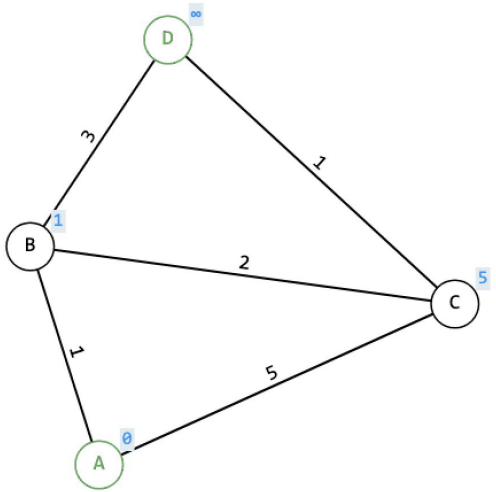




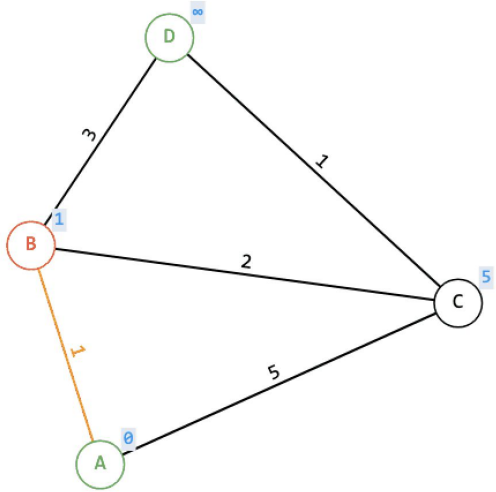


Алгоритм Дейкстры

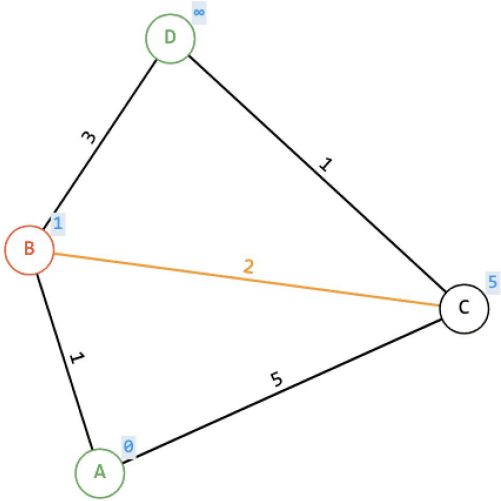
Ищем непосещённую вершину с наименьшей меткой



Проверяем все смежные с B вершины:

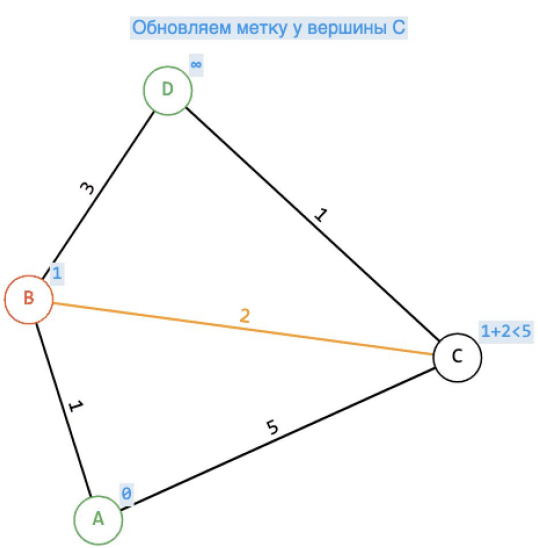


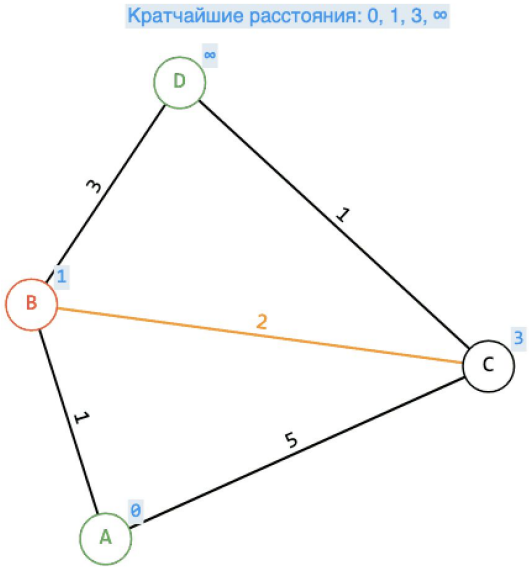
Проверяем все смежные с B вершины:

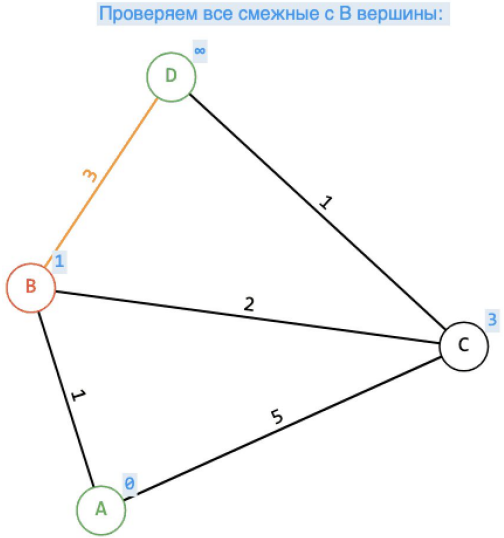


Алгоритм Дейкстры

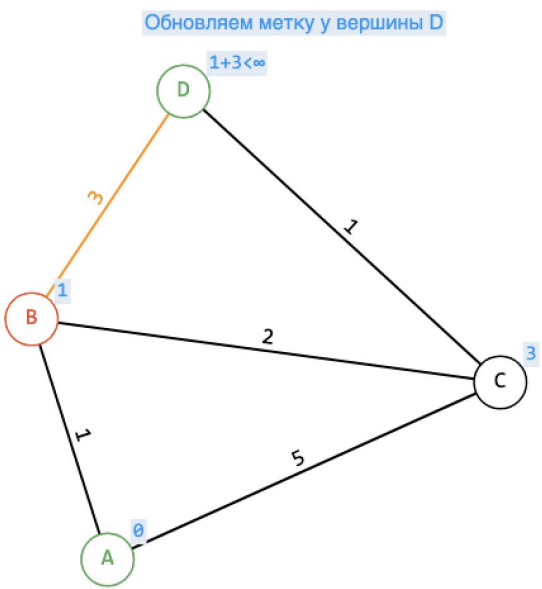
Обновляем для вершины C значение

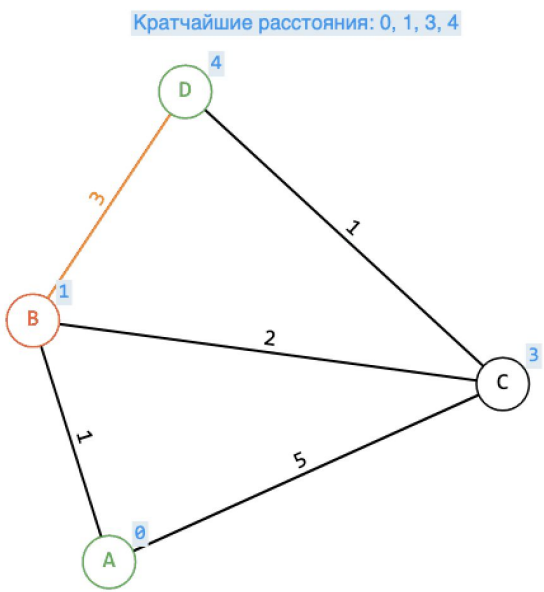


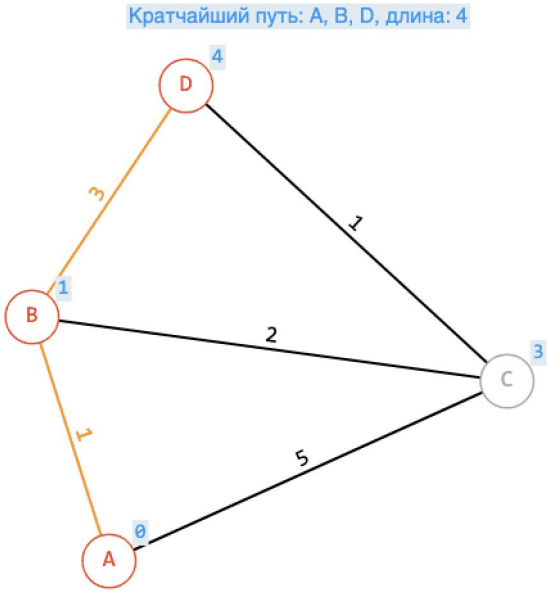




Алгоритм Дейкстры



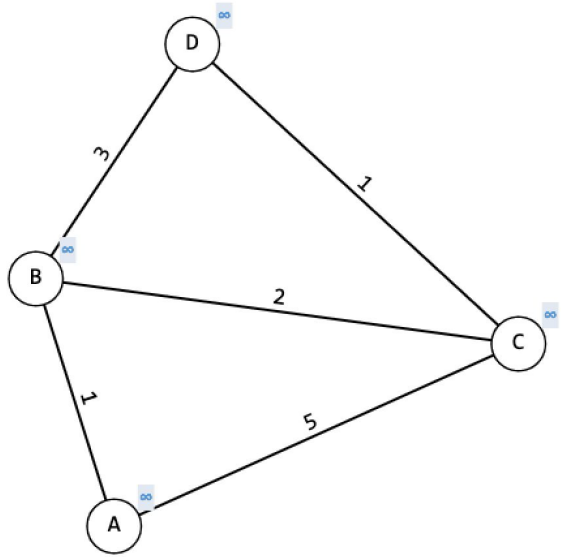




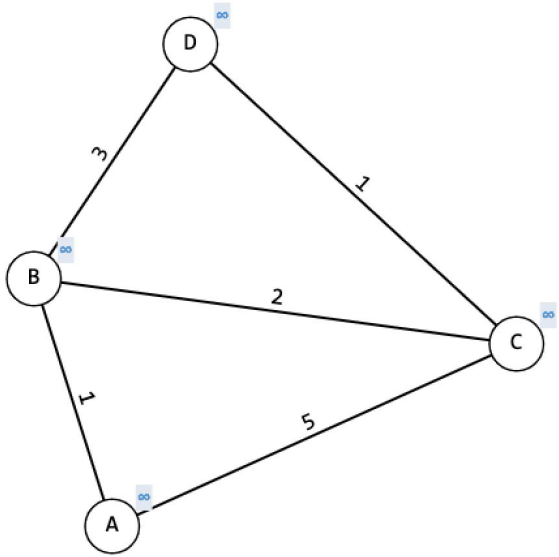
Алгоритм Дейкстры

```
// вспомогательная функция для нахождения вершины с минимальным расстоянием
function vertexWithMinWeight(graph, visited) {
    index = “”
    dist_min = INF
    for vertex in graph {
        // если текущее расстояние меньше минимального и вершина не посещена
        if dist[vertex] < dist_min and !visited[vertex] {
            dist_min = dist[vertex]
            // обновляем индекс вершины с минимальным значением
            index = vertex
        }
    }
    return index
}
```

```
function dijkstra(graph, start) {
    // Определяем бесконечность как очень большое число
    INF = MAX_INT
    // Создаем словарь для отслеживания посещенных вершин
    // Для каждой вершины проставляем false
    visited = map[string]bool

    // Создаем словарь для хранения расстояний до каждой вершины в графе
    // Каждой вершине проставляем MAX_INT
    dist = map[string]int

    // Расстояние до начальной вершины равно 0
    dist[start] = 0

    // Пока есть не посещенные вершины
    while false in visited {
        // Находим вершину с минимальным расстоянием
        u = vertexWithMinWeight(graph, )
        // Перебираем всех соседей вершины u
        for v in graph[u] {
            // Если есть ребро и вершина v не посещена
            if graph[u][v] != 0 and !visited[v] {
                // Обновляем расстояние до вершины v
                dist[v] = min(dist[v], dist[u] + graph[u][v])
            }
        }
        // Помечаем вершину u как посещенную
        visited[u] = true
    }
    // Возвращаем словарь с расстояниями до всех вершин от начальной вершины
    return dist
}
```



```
function dijkstra(graph, start) {
  // Определяем бесконечность как очень большое число
  INF = MAX_INT
  // Создаем словарь для отслеживания посещенных вершин
  visited = map[string]bool
  for vertex in range(graph) {
    visited[vertex] = false
  }
  // Создаем словарь для хранения расстояний до каждой вершины в графе
  dist = map[string]int
  for vertex in range(graph) {
    dist[vertex] = INF
  }
  // Расстояние до начальной вершины равно 0
  dist[start] = 0

  // Пока есть не посещенные вершины
  while false in visited {
    // Находим вершину с минимальным расстоянием
    u = vertexWithMinWeight(graph, visited)
    // Перебираем всех соседей вершины u
    for v in graph[u] {
      // Если есть ребро и вершина v не посещена
      if graph[u][v] != 0 and !visited[v] {
        // Обновляем расстояние до вершины v
        dist[v] = min(dist[v], dist[u] + graph[u][v])
      }
    }
    // Помечаем вершину u как посещенную
    visited[u] = true
  }
  // Возвращаем словарь с расстояниями до всех вершин от начальной вершины
  return dist
}
```



Немного про технологии

Всё про стажировку:
https://internship.vk.company/internship

Golang

- SQL (PostgreSQL, MySQL)
- Clickhouse
- Kafka
- Docker, kubernetes
- Понимание принципов SOLID
- git

PHP

- SQL (PostgreSQL, MySQL)
- Nginx
- NoSQL (Mongodb)
- Понимание принципов SOLID, ООП
- git

Java

- Java 17
- Spring
- PostgreSQL
- NoSQL (Redis, Cassandra)
- Clickhouse
- Понимание принципов SOLID
- ООП
- git

C++

- C++ 17
- CMake
- Boost
- Понимание принципов SOLID, ООП
- Linux
- git

Python

- SQL
- знание основных сетевых web-протоколов,
  как минимум HTTP(S)
- Django, JS, CSS, HTML
- Понимание принципов SOLID, ООП
- docker
- git

ML

- Python (Numpy, scikit-learn, OpenCV, Pandas)
- Deep Learning фреймворки pytorch/jax
- git

Data Scientist

- **SQL** на уровне написания сложных запросов
- Знания в области математической статистики, эконометрики, понимание основных алгоритмов ML
- Знание **Python** (аналитический и **ML** стек)

Devops

- Kubernetes
- docker
- Модули конфигураций puppet/ansible/helm
- Bash, Python
- git

Frontend

- Java script (ES6, type script)
- React
- Доп инструменты Grunt/Gulp/Webpack
- HTML + CSS
- git

Android

- Java (многопоточность, RxJava), Kotlin
- Основные компоненты Android (Activity, Fragments, Services) жизненный цикл
- Основные контейнеры для отрисовки view (LinearLayout, Constraint)
- Базовые понимания для RecyclerView - для чего нужен
- Работа с сетью (Retrofit, OkHttp)
- Понимание принципов SOLID, ООП
- git

ios

- ObjC
- Swift
- UIKit, SnapKit
- Понимание принципов SOLID
- git

Всем спасибо

И хороших выходных:)