Graph traversal, Pt. 1 🏃

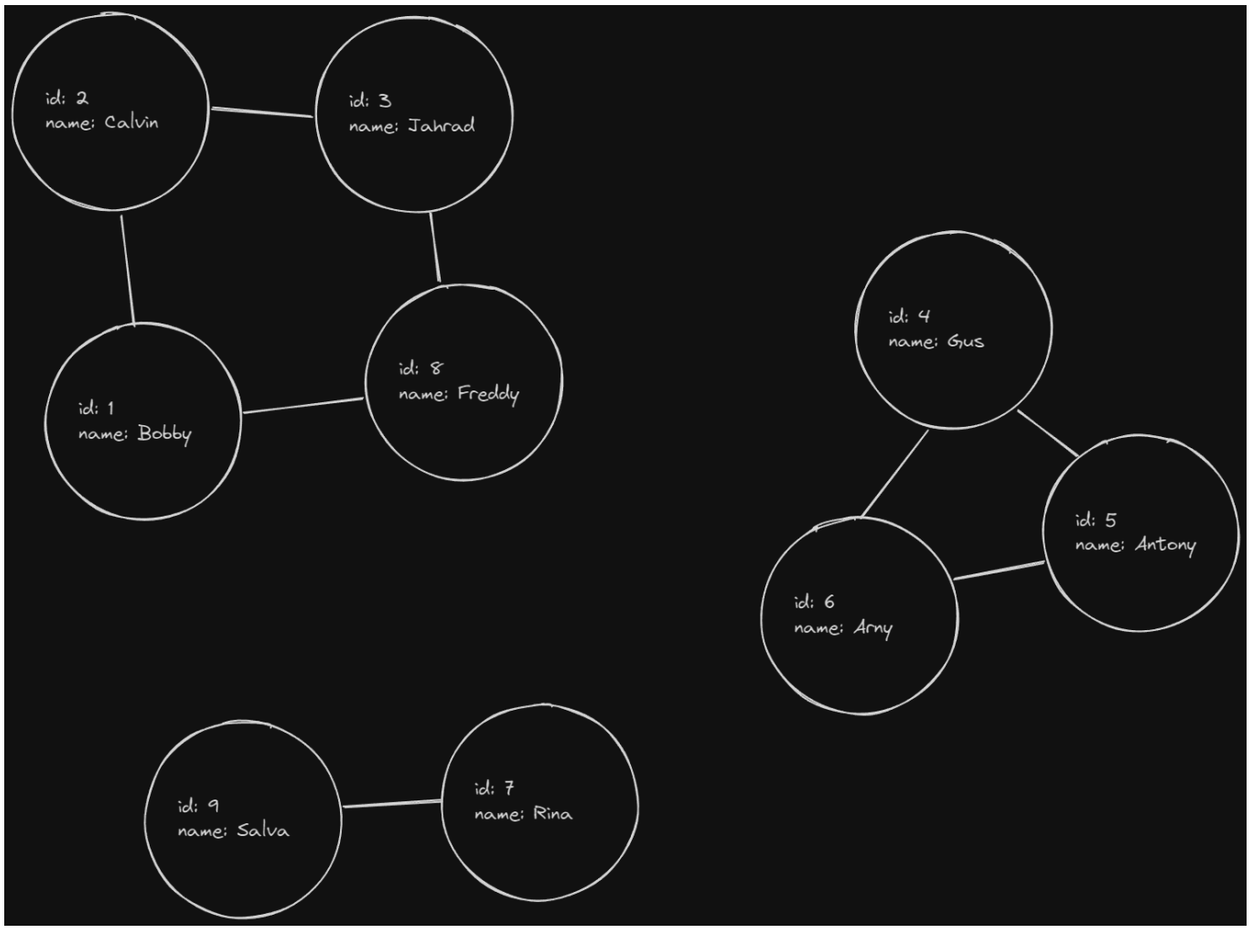
Interlude

Представим, что мы хотим получить имена всех друзей некоторого пользователя. В таком случае необходимо найти пользователя с данными идентификатором и посмотреть с кем он связан в графе, описывающем дружбу между пользователями.

Алгоритм, который позволит это сделать — это самое базовое, что хочется уметь делать с графами — обходить их.

How to 🧐

Допустим, мы хотим вывести имена друзей пользователя `id: 3`.



В графе пользователь `id: 3` напрямую связан с 2 другими людьми, значит у него 2 друга, имена которых можно получить посетив эти вершины.

```
struct User{
  public int Id;
  public string Name;
}

User[] getFriendsOf(int id){
  ans = []
  for(auto v in g[id])   // смотрим на всех друзей пользователя с данным id
    ans.add(g[id][v])    // добавляем друга пользователя с данным id
```

```
  return ans
}

for(auto i in getFriendsOf(3))
  cout << i.Name << endl;      // Calvin, Freddy
```

А что, если хочется узнать всех пользователей, с которыми пользователь `id: 3` может быть знаком через одно или несколько рукопожатий?

Depth first search 🔍

Первый алгоритм, который позволяет обходить графы называется *поиск в глубину*. Не понятно, что хотел искать в графе создатель этого алгоритма, но мы будем искать социальные круги — группы людей, в которых каждый знаком с каждым напрямую или через кого-то.

В теории графов такие социальные круги называются компонентами связности.

```
struct User{
  public int Id;
  public string Name;
}

ans = []
visited = []
User[] getSocialGroupOf(int id){
  visited.add(id);                    // пометка, что мы знаем информацию о пользователе

  for(auto v in g[id]){               // смотрим на всех друзей пользователя с данным id
    if (g[id][v].Id is not in visited) // если мы ещё не знаем информацию о друге, то
            ans.add(g[id][v])             // сделаем то же самое для него
            getSocialGroup(g[id][v].Id)
  }

  return ans
}

for(auto i in getSocialGroupOf(3))
  cout << i.Name << endl;          // Calvin, Jahrad, Bobby, Freddy
```

DFS — алгоритм, с которым можно делать очень много смешных вещей, одной из которых является выделение компонент связности. С помощью **DFS** можно найти все социальные круги в данной социальной сети.

```
struct User{
  public int Id;
  public string Name;
}

ans = []
visited = []
User[] getSocialGroupOf(int id){
  visited.add(id);                    // пометка, что мы знаем информацию о пользователе

  for(auto v in g[id]){               // смотрим на всех друзей пользователя с данным id
    if (g[id][v].Id is not in visited) // если мы ещё не знаем информацию о друге, то
            ans.add(g[id][v])             // сделаем то же самое для него
            getSocialGroup(g[id][v].Id)
  }
```

```
  return ans
}


group = 1;
for(auto user in g){
   if (user.Id is not in visited)
           User[] result = getSocialGroupOf(user.Id)
           cout << result << ' : ' << group;
           group++;
           ans.clear();
}

//   Calvin, Jahrad, Bobby, Freddy : 1
//   Gus, Arny, Antony : 2
//   Salva, Rina : 3
```

Anything else ❓

Я упоминал граф друзей в социальной сети и граф подписчиков. При этом граф подписчиков был ориентированным. Понятно, что мы умеем выделять социальные группы. А что если хочется уметь выделять множество фанатов одного человека ?

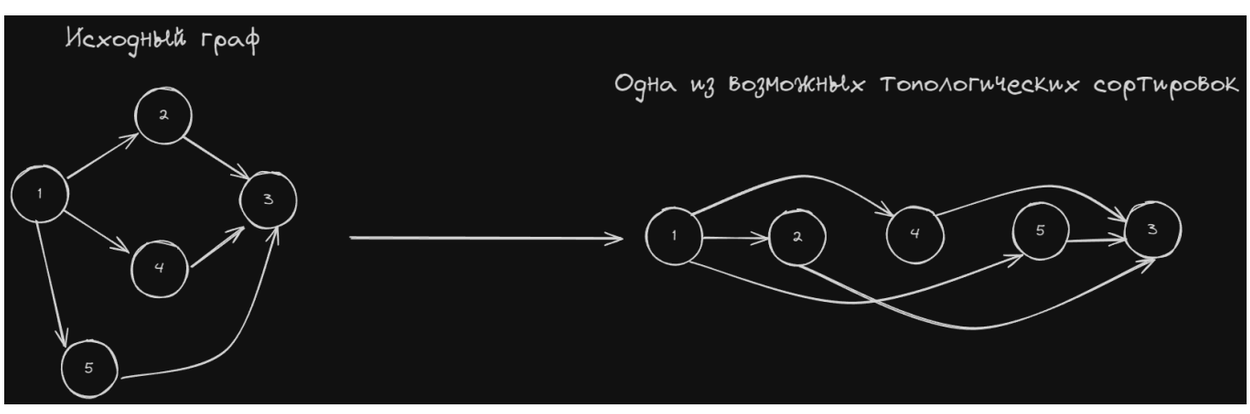
Ещё один важный тип графов — *ациклические графы*. Это графы, в которых нет циклов. Частным случаем такого графа является дерево — это *неориентированный ациклический граф*. Деревья это не совсем интересно, поэтому рассмотрим ориентированные ациклические графы.

Конечно, в таких графах хочется понимать можно ли его таковым называть. Понятно, что это можно сделать с помощью обхода графа. Интереснее уметь отсортировать граф и вот что такое сортировка графа.

Топологической сортировкой графа называется упорядочивание вершин графа в некотором порядке, где стрелочки идут слева направо.

Это отношение порядка позволяет понимать время входа и выхода для компоненты сильной связности ориентированного графа, что дальше будет очень сильно полезно для решения задачи о множестве фанатов.

Topological sort 🌀



Топологическая сортировка — это просто ещё один инструмент для работы с графами, как и обходы, поэтому его просто нужно знать.

Чтобы построить топологическую сортировку, нужно:

- найти в какую вершину не входит ни одно ребро и эту вершину записать в результат
- удалить эту вершину из исходного графа
- повторить предыдущие шаги для полученного графа

Единственное, что не понятно из всего этого как найти вершину, в которую ничего не входит. Можно пойти от обратного: пройдемся по всем не посещенным вершинам и будем записывать вершину, из которой ничего не выходит. Здесь опять же очень удобно использовать рекурсию

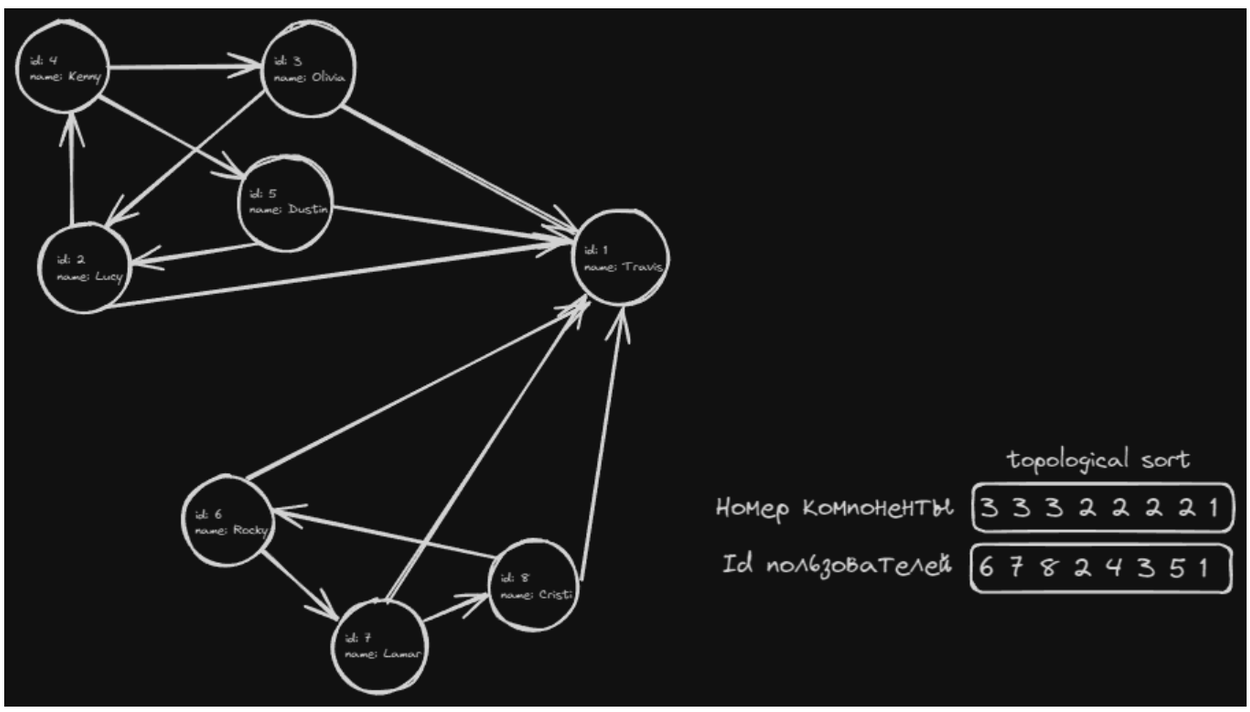
```
visited = []
topsort = []

int[] topsort(int entry){
  visited.addFirst(entry)

  for(auto v : g[entry])                // идем по всем не посещенным вершинам
    if (g[entry][v] is not in visited)  // и помечаем их как посещенные
      topsort(g[entry][v])
                                       // когда нибудь мы встретим вершину из которой
некуда
  topsort.add(entry)                    // идти и её запишем
}

cout << topsort
```

Видно, что топологическая сортировка, как бы она страшно не звучала, это просто DFS с небольшой модификацией.

Отвечая на вопрос о том как найти множество подписчиков одного человека, как один из вариантов: топологическая сортировка без последнего элемента.

Strongly connected components 🔗

Ну из дискретки понятно, что это отношение связности на ориентированных графах. Выделение таких компонент решает задачу о всех фанатских группах (как со всеми социальными группами).



В этом графе можно выделить 3 компоненты сильной связности: $[2,3,4,5], [6,7,8], [1]$.
Каждая такая компонента является фанатской группой пользователя `id: 1` за исключением его

самого.

Чтобы выделить компоненты сильной связности нужно:

- топологически отсортировать граф
- пройтись дфсом по обратным ребрам графа в порядке топологической сортировки

Обратные ребра нужны, чтобы не выйти дфсом из компоненты сильной связности, а топологическая сортировка определяет порядок (время) входа в компоненту связности, именно поэтому в этом контексте использование топологической сортировки на не ациклическом графе оправдано.

```
topsortOrder = []
visited = []

void topsort(int id){
  visited.add(id)

  for(auto v in g[id])
    if (g[id][v].Id is not in visited)
      topsort(g[id][v].Id)

  topsortOrder.add(id)
}

ans = []
User[] getFanGroups(int id){
   visited.add(id)

   for(auto v in reversedG[id])
     if (g[id][v].Id is not in visited){
       ans.add(g[id][v])
       getFanGroups(g[id][v].Id)
     }

  return ans
}

for(auto user in g)
  topsort(user.Id)

topsortOrder.reverse()
visited.clear()

for(auto i in topsortOrder)
  if (i is not in visited)
    auto res = getFanGroups(i)
    ans.clear();
    cout << res
```