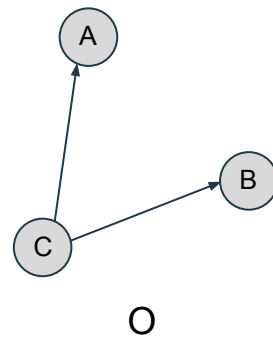
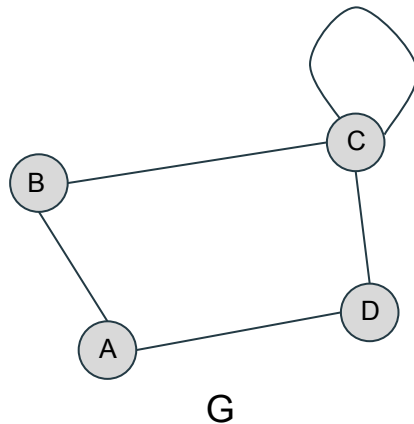
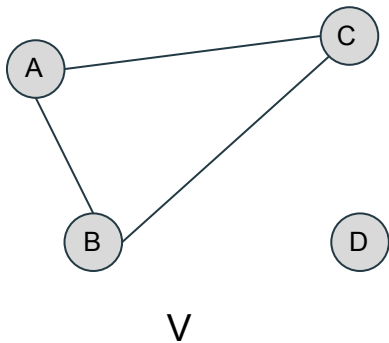


# Графы

# Определение

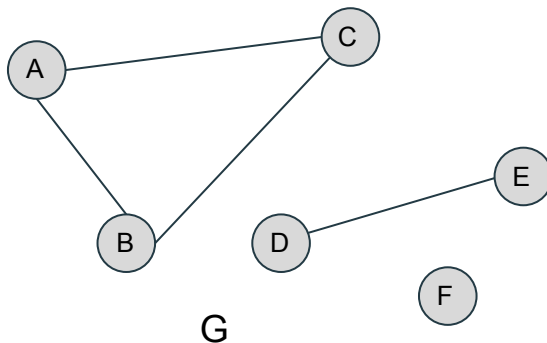
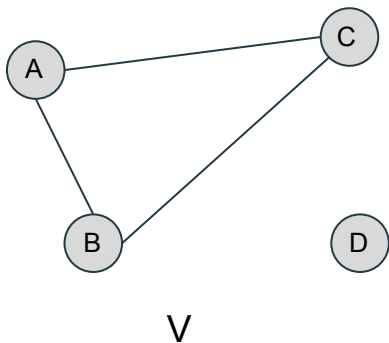
**Граф** - множество вершин и набор связей между ними.



# Связность

Неориентированный граф называется **связным**, если существует путь из каждой его вершины в любую другую.

**Компонента связности** - подмножество вершин графа, такое, что между каждой из них существует путь, но не существует пути из вершины этого множества в вершину не из этого множества.



# Некоторые графы

**Дерево** - связный граф без циклов.

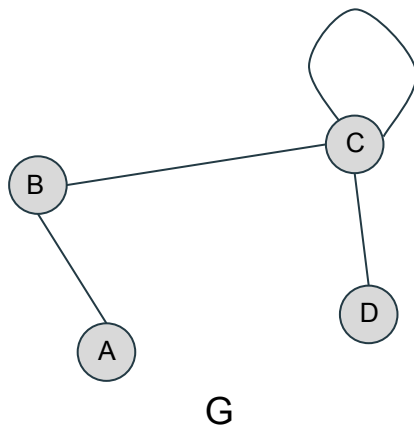
**Лес** - граф без циклов.

**Полный граф** - граф, в котором существуют ребра между всеми парами вершин.

Больше определений можно найти в [интернете](#).

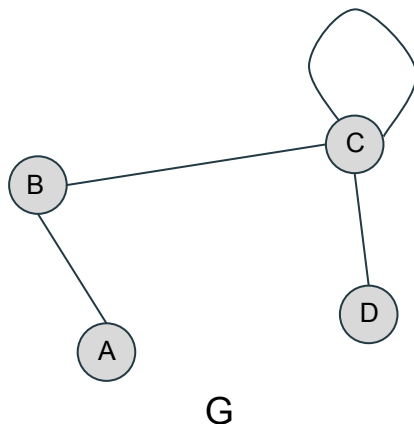
# Матрица смежности

	A	B	C	D
A	0	1	0	0
B	1	0	1	0
C	0	1	1	1
D	0	0	1	0



# Список смежности

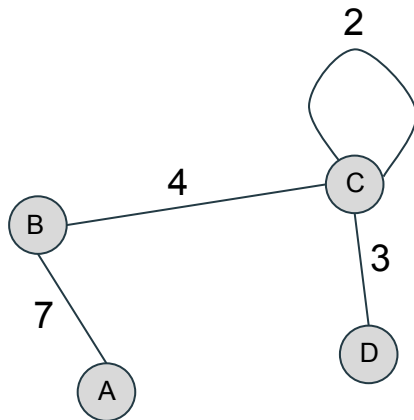
A:	B		
B:	A	C	
C:	B	D	C
D:	C		



Расходуется меньше памяти (для  
неполных графов и для графов, не  
имеющих кратных ребер)!

# Весовая матрица

	A	B	C	D
A	0	7	0	0
B	7	0	4	0
C	0	4	2	3
D	0	0	3	0



# Графы в ЯП

Массив динамических массивов (считываем пары - начало и конец ребра):

```
vector<int> v[N];  
.  
.  
.  
for(int i = 0; i<m; i++){  
    int x, y;  
    cin >> x >> y;  
    v[x].push_back(y);  
    v[y].push_back(x);  
}
```

```
var v: array [1..N] of array of integer;  
.  
.  
.  
for var i := 1 to m do begin  
    read(x, y);  
    setLength(v[x], Length(v[x]) + 1);  
    v[x] [Length(v[x]) - 1] := y;  
    setLength(v[y], Length(v[y]) + 1);  
    v[y] [Length(v[y]) - 1] := x;  
end;
```



# Поиск в глубину C

## Подготовка:

1. Отмечаем все  $n$  (от 1-ой до  $n$ -ой) вершин как непосещенные  
`bool used[100010]; // при "правильном" описании массив заполнен значениями false`
2. Для каждой непосещенной вершины запускаем из нее поиск в глубину (dfs)  
`for(int i = 1; i<=n; i++)  
 if (!used[i]) dfs(i);`

## Поиск в глубину от вершины $x$ (dfs(x)):

1. Отмечаем вершину как посещенную  
`used[x] = true;`
2. Для каждой непосещенной вершины смежной с  $x$  запускаем из нее поиск в глубину (dfs)  
`for(int i = 0; i<v[x].size(); i++)  
 if (!used[ v[x][i] ]) dfs(v[x][i]);`

# Поиск в глубину Pascal

## Подготовка:

1. Отмечаем все  $n$  (от 1-ой до  $n$ -ой) вершин как непосещенные  
`var used: array [1..100010] of boolean;`
2. Для каждой непосещенной вершины запускаем из нее поиск в глубину (dfs)  
`for var i := 1 to n do`  
`if (not used[i]) then dfs(i);`

## Поиск в глубину от вершины $x$ (dfs(x)):

1. Отмечаем вершину как посещенную  
`used[x] := true;`
2. Для каждой непосещенной вершины смежной с  $x$  запускаем из нее поиск в глубину (dfs)  
`for var i := 0 to Length(v[x])-1 do`  
`if (not used[v[x,i]]) dfs(v[x,i]);`

```

const int N = 2e5+10;
vector<int> v[N];
bool used[N];

void dfs(int x){
    used[x] = true;
    for(int i = 0; i<v[x].size(); i++) // перебираем всех соседей
        if (!used[v[x][i]]) dfs(v[x][i]);
}

int main(){
    int n, m; // количество вершин и ребер соответственно
    cin >> n >> m;
    for(int i = 0; i<m; i++){ // перебираем все ребра!
        int x, y; cin >> x >> y;
        //добавляем вершину y в список смежных вершин с x (v[x] расширяется)
        v[x].push_back(y);
        //добавляем вершину x в список смежных вершин с y (v[y] расширяется)
        v[y].push_back(x);
    }
}



for(int i = 1; i<=n; i++) // перебираем все вершины!
    if (!used[i]) dfs(i);

```

```

const maxN = 200010;
var used: array [1..maxN] of boolean;
    v: array[1..maxN] of array of integer;
    n,m: integer;
procedure bfs(x: integer);
begin
    used[x] := true;
    for var i:= 0 to Length(v[x]) - 1 do // перебираем всех соседей
        if(not used[v[x][i]]) then dfs(v[x][i]);
    end;
begin
    read(n,m); // количество вершин и ребер соответственно
    for var i := 1 to m do begin // перебираем все ребра!
        var x,y: integer; read(x, y);
        setLength(v[x], Length(v[x]) + 1); // расширяем v[x] на 1 элемент
        v[x] [Length(v[x]) - 1] := y; // добавляем y в список смежных вершин с x
        setLength(v[y], Length(v[y]) + 1); // расширяем v[y] на 1 элемент
        v[y] [Length(v[y]) - 1] := x; // добавляем x в список смежных вершин с y
    end;
    for var i := 1 to n do // перебираем все вершины!
        if (not used[i]) then dfs(i);
    end.

```



# Задачи на поиск в глубину в неориентированных графах

# Подсчет компонент связности (dfs-ом)

Количество компонент связности в **неориентированном графе** = количество вызовов поиска в глубину в **основной программе из непосещенных вершин**

```
int cnt = 0;
for(int i = 1; i<=n; i++) // перебираем все вершины!
    if (!used[i]){        // заходим только в непосещенные ранее
        dfs(i);
        cnt++;
    }
```

# Подсчет вершин в компоненте СВЯЗНОСТИ

Количество вершин в компоненте связности в **неориентированном графе** = количество вызовов поиска в глубину в **ЭТОЙ КОМПОНЕНТЕ СВЯЗНОСТИ**

```
var cnt: integer := 0; // начальное значение счетчика
procedure bfs(x: integer);
begin
    cnt := cnt + 1; // счетчик вершин
    used[x] := true;
    for var i:= 0 to Length(v[x]) - 1 do // перебираем всех соседей
        if(not used[v[x][i] ]) then dfs(v[x][i]);
end;
```

# Список вершин компоненты связности

**Идея №1:** пометим все вершины, лежащие в одной компоненте связности, одним цветом (дадим таким вершинам одинаковые номера).

```
int color[N]; // Массив цветов. Если 0, то вершина еще не посещалась

void dfs(int x, int col){ // col - цвет для текущей компоненты связности
    color[x] = col; // можно использовать вместо used
    for(int i = 0; i < v[x].size(); i++) // перебираем всех соседей
        if (color[v[x][i]] == 0) dfs(v[x][i], col);
}
```



# Список вершин компоненты связности

**Идея №2:** заведем динамический массив динамических массивов для каждой компоненты

```
vector <vector<int> > comp; // Массив компонент
void dfs(int x) {
    used[x] = true;
    comp[comp.size() - 1].push_back(x); // добавляем вершину в последний список компонент
    for(int i = 0; i < v[x].size(); i++) // перебираем всех соседей
        if (!used[v[x][i]] == 0) dfs(v[x][i]);
}
. . .
for(int i = 1; i <= n; i++) { // перебираем все вершины!
    if (!used[i]) {
        vector<int> p; // пустой вектор
        comp.push_back(p); // расширяем список компонент
        dfs(i);
    }
}
```

# Поиск наличия цикла

```
var cikl : boolean := false;
procedure bfs(x, y: integer); // y - вершина из которой приходим
begin
    used[x] := true;
    for var i:= 0 to Length(v[x]) - 1 do // перебираем всех соседей
        if(not used[v[x][i]]) then dfs(v[x][i], x) // идем по ребру (v[x][i], x)
        else
            if (y != v[x][i])
            then cikl := true;
// вершина не посещена и мы идем не в ту вершину, из которой в эту вершину пришли
end;
. . .
for var i := 1 to n do // перебираем все вершины!
    if (not used[i])
    then dfs(i, 0);
// идем из несуществующей вершины с номером 0 (или -1, если нумеруем с 0)
```

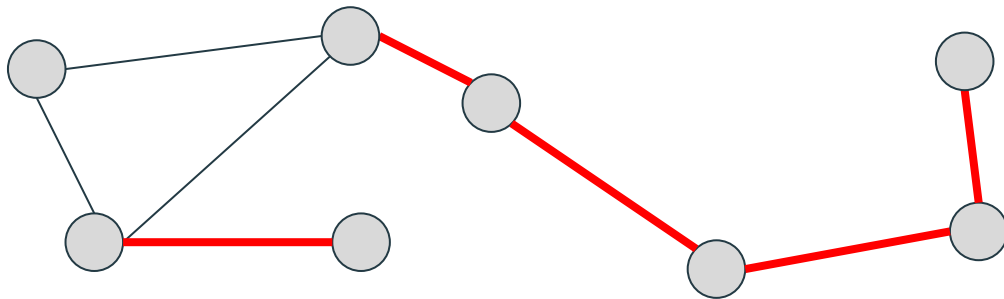
# Является ли граф деревом?

**Дерево** - связный граф без циклов.

Т.е. проверяем наличие цикла и считаем количество компонент связности!

# Поиск мостов в графе

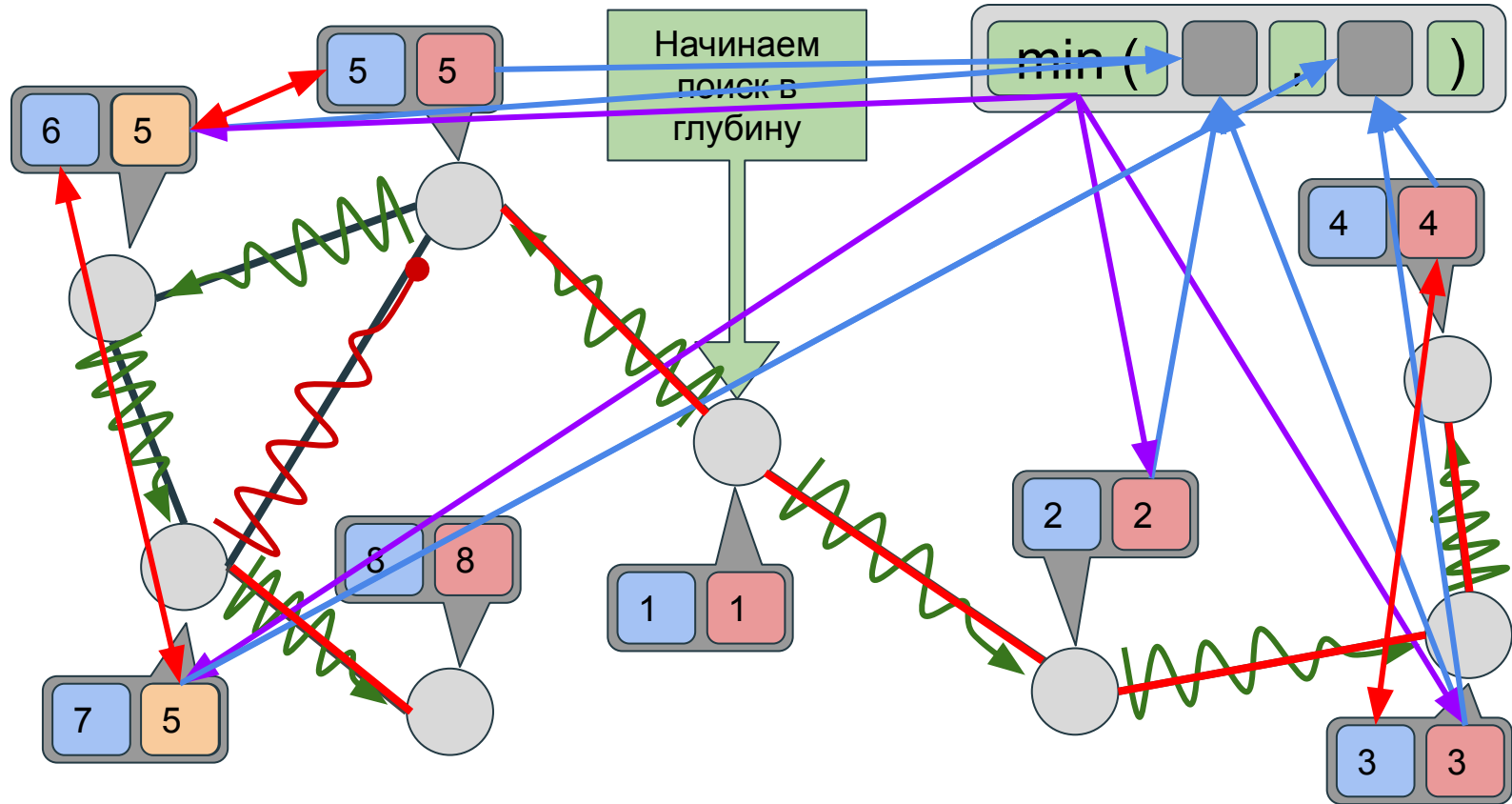
**Мост** - ребро графа, при удалении которого количество компонент связности графа увеличивается.

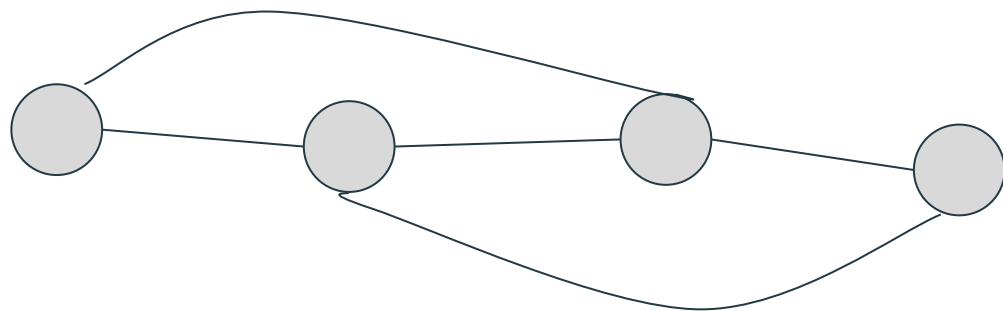




# Поиск мостов в графе

**Идея:** Запустим поиск в глубину с какой-либо вершины. Для каждой вершины  $v$  будем запоминать время входа (порядковый номер обхода) в вершину  $\text{tin}[v]$ . А также время подъема  $\text{tup}[v]$ , равное минимуму из времени входа в саму вершину  $v$ , времен входа во все вершины, являющиеся концами обратного ребра в дереве поиска, а также всех значение  $\text{tup}[p_i]$ , где  $p_i$  - сын  $v$  в дереве поиска.

Пусть мы находимся в вершине  $v$ . Запустим поиск в глубину в какую-либо смежную не посещенную вершину  $u$ . Тогда, после окончания поиска, если  $\text{tin}[v] < \text{tup}[u]$  (время входа в вершину  $v$  меньше чем время подъема из вершину  $u$ ), то ребро  $(v, u)$  - является мостом.







# Задачи на поиск в глубину в ориентированных графах



Пока я не дописал этот раздел :)

Как искать циклы, компоненты сильной связности и т.д. в ориентированных графах можно посмотреть [тут](#)