Поиск в глубину

Поиск в глубину — удивительный алгоритм. С одной стороны, он очень прост для реализации и довольно прост для понимания, а с другой стороны, имеет столько применений для различных стандартных задач, сколько, пожалуй, не имеет ни один другой алгоритм. Если большинство других алгоритмов решают вполне конкретную задачу, то поиск в глубину сам по себе — скорее шаблон для написания алгоритмов решения для целого набора задач. Я тут постараюсь рассказать применения его для почти всех задач, которые знаю; некоторые из этих применения попроще, некоторые посложнее, надеюсь, что хотя бы часть будет понятна.

Часть I. Элементарная реализация и общие замечания

§1. Реализация. Итак, что такое поиск в глубину. Поиск в глубину — это рекурсивный алгоритм обхода графа. Он принимает на вход некоторую вершину графа и рекурсивно запускает себя для всех ещё не посещённых соседей данной вершины. Таким образом, у нас в каждый момент про каждую вершину мы знаем, были мы в ней уже или ещё нет. При входу в очередную вершину мы помечаем, что мы в ней были, и рекурсивно входим во все соседние с ней вершины, в которых ещё не были. Элементарная реализация поиска в глубину тогда получает следующий вид (считаем, что мы храним граф матрицей смежности):

```
procedure find(i:integer);
begin
was[i]:=1;
for j:=1 to n do
    if (gr[i,j]=1)and(was[j]=0) then
       find(j);
end;
```

Здесь gr — матрица смежности, was — тот самый массив, в котором мы храним, были мы уже в этой вершине или нет ещё.

В принципе, это абсолютно корректный вариант реализации, но я лично предпочитаю другой вариант: как в любых рекурсивных процедурах, я предпочитаю проверять необходимость рекурсивного вызова при *входе* в процедуру, а не *при её вызове* (конечно, пока это возможно). Конкретно, в данном случае я предпочитаю перекинуть проверку $was[j] = 0$ в начало процедуры:

```
procedure find(i:integer);
begin
if was[i]<>0 then
   exit;
was[i]:=1;
for j:=1 to n do
    if gr[i,j]=1 then
       find(j);
end;
```

(обратите внимание, что, конечно, проверка из $was[j]$ превратилась в $was[i]$).

Смысл такого переноса проверок в общем случае рекурсивных процедур (в первую очередь в случае перебора) в том, что 1. если вы вызываете вашу процедуру из нескольких мест в коде, то не надо каждый раз дублировать проверку, 2. нередко в начале процедуры проверка смотрится как-то естественнее (естественнее по максимуму работать с i-ой вершиной, т.к. именно она является параметром процедуры), и иногда эту проверку тут написать проще. С другой стороны, такой перенос приводит к дополнительным затратам времени на вызов функции при выполнении программы, но по-моему в большинстве случаев эти затраты несущественны (время выполнения самой процедуры обычно много больше). Правда, в случае поиска в глубину все эти преимущества почти незаметны, т.к. $was[i] <> 0$ — фактически единственная проверка, которую получается перенести (совершенно понятно, что проверку $gr[i, j] = 1$ переносить невозможно и вообще бессмысленно), к тому же процедура find обычно вызывается максимум из двух мест в коде, поэтому экономии от переноса нет, так что здесь это скорее дело вкуса.

Ещё обращу внимание на то, что во втором варианте написано $was[i] <> 0$, а не $was[i] = 1$. Вообще, это как бы ещё пример общей идеи, что имхо стоит максимально расширять условия в if'ах. В данном случае потом может оказаться, что, зайдя в вершину, в was мы сохраним что-нибудь ещё, не обязательно единицу (например, номер компоненты связности, или «состояние» вершины в том или ином смысле) — но в любом случае ноль будет обозначать непосещённую вершину, а не-ноль — посещённую, поэтому $was[i] <> 0$ будет всегда работать.

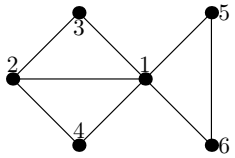
§2. Как вызывать процедуру $find$**.** Конечно, сама по себе процедура ничего не сделает; чтобы её использовать, нужно её вызвать в главной программе (или, естественно, в другой процедуре, где вам понадобилось запустить поиск в глубину, и т.п.). Способ вызова, конечно, зависит от задачи, но наиболее часто используются два варианта: либо нужно просто запустить поиск в глубину из данной вершины и всё — это, естественно, делается просто вызовом $find(i)$, где i — номер этой вершины, либо нужно запустить поиск так, чтобы обойти обязательно все вершины — тогда пишем так:

```
for i:=1 to n do
    if was[i]=0 then
       find(i);
```

Конечно, если сам поиск написан вторым способом (т.е. с проверкой *was* при входе в процедуру), то if из этого кода можно убрать.

И ещё, конечно, нужно не забыть предварительно занулить массив *was*.

§3. Пример работы. Пусть у нас есть граф, показанный на рис. Запустим на нем поиск в глубину из вершины 1. Пусть (в соответствии с тем, как это написано в приведённых выше реализациях) при просмотре соседних вершим мы перебираем вершины в порядке возрастания номеров. Тогда алгоритм поиска будет работать следующим образом (отступ строки соответствует уровню рекурсии):



`find(1)`: запускаем поиск в глубину из вершины 1
Помечаем, что побывали в вершине 1
Просматриваем соседей вершины 1
Нашли соседа: вершину 2
В вершине 2 ещё не были
    `find(2)`: запускаем поиск в глубину из вершины 2
    Помечаем, что побывали в вершине 2
    Просматриваем соседей вершины 2
    Нашли соседа: вершину 1
    В вершине 1 уже были, поиск из вершины 1 не запускаем[1]
    Нашли соседа: вершину 3
    В вершине 3 ещё не были
        `find(3)`: запускаем поиск в глубину из вершины 3
        Помечаем, что побывали в вершине 3
        Просматриваем соседей вершины 3
        Нашли соседа: вершину 1
        В вершине 1 уже были, поиск из вершины 1 не запускаем
        Нашли соседа: вершину 2
        В вершине 2 уже были, поиск из вершины 2 не запускаем
        Соседи вершины 3 закончились, завершаем поиск из вершины 3
    (продолжаем просмотр соседей вершины 2)[2]
    Нашли соседа: вершину 4
    В вершине 4 ещё не были
        `find(4)`: запускаем поиск в глубину из вершины 4
        Помечаем, что побывали в вершине 4
        Просматриваем соседей вершины 4
        Нашли соседа: вершину 1
        В вершине 1 уже были, поиск из вершины 1 не запускаем
        Нашли соседа: вершину 2
        В вершине 2 уже были, поиск из вершины 2 не запускаем
        Соседи вершины 4 закончились, завершаем поиск из вершины 4
    (продолжаем просмотр соседей вершины 2)
    Соседи вершины 2 закончились, завершаем поиск из вершины 2
(продолжаем просмотр соседей вершины 1)
... (нашли соседей: вершины 3 и 4, поиск из них не запускаем, для краткости не описываю это подробно)
Нашли соседа: вершину 5
В вершине 5 ещё не были
    `find(5)`: запускаем поиск в глубину из вершины 5
    Помечаем, что побывали в вершине 5
    Просматриваем соседей вершины 5
    Нашли соседа: вершину 1
    В вершине 1 уже были, поиск из вершины 1 не запускаем
    Нашли соседа: вершину 6
    В вершине 6 ещё не были
        `find(6)`: запускаем поиск в глубину из вершины 6
        Помечаем, что побывали в вершине 6
        Просматриваем соседей вершины 6
        Нашли соседа: вершину 1
        В вершине 1 уже были, поиск из вершины 1 не запускаем
        Нашли соседа: вершину 5
        В вершине 5 уже были, поиск из вершины 5 не запускаем
        Соседи вершины 6 закончились, завершаем поиск из вершины 6
    (продолжаем просмотр соседей вершины 5)
    Соседи вершины 5 закончились, завершаем поиск из вершины 5
(продолжаем просмотр соседей вершины 1)
Нашли соседа: вершину 6
В вершине 6 уже были, поиск из вершины 6 не запускаем
Соседи вершины 1 закончились, завершаем поиск из вершины 1

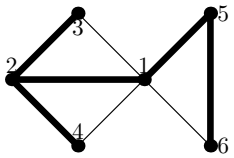
На самом деле очень нетривиально придумать один небольшой пример, который бы полностью характеризовал все особенности поиска в глубину. Поэтому, если вам что-то во внутреннем механизме работы поиска ещё не понятно, порисуйте ещё графы и промоделируйте вручную работу поиска на них.

§4. Дерево поиска в глубину. У каждой вершины, кроме той, из которой был произведён начальный запуск поиска («корня»), можно выделить «родителя» — вершину, из которой мы перешли в данную

[1] Или, если реализовано вторым способом, запускаем, но тут же выходим назад.

[2] Обратите внимание, что это происходит автоматически: работа процедуры `find(3)` завершилась, поэтому продолжается работа программы с того места, откуда была вызвана процедура `find(3)` — а это есть строчка в цикле в процедуре `find(2)`, поэтому просто происходит переход к следующей итерации цикла в `find(2)`, т.е. продолжаем просмотр соседей вершины 2. Поэтому эта строчка здесь и написана курсивом — ей, можно сказать, не соответствует никакая строка исходного текста программы.

вершину. Соединив каждую вершину (кроме корня, конечно) с её родителем, получим подграф исходного графа — *дерево поиска в глубину.*



Задание 1: *докажите, что действительно получится дерево. Точнее, докажите, что в полученном подграфе не будет циклов. Верно ли, что это всегда будет дерево, покрывающее все вершины исходного графа? Как это зависит от того, как мы вызываем поиск в глубину (п. I.2)? На самом деле доказательство не очень тривиально.*

На рис. справа приведено дерево для примера из предыдущего пункта.

Собственно, поиск в глубину — самый, пожалуй, простой алгоритм построения в связном графе *остовного* дерева (т.е. дерева, покрывающего все вершины графа). Если вам зачем-то понадобилось *любое* остовное дерево, пользуйтесь поиском в глубину. Кроме того, дерево поиска в глубину нам ещё пригодится при решении задач ниже.

§5. Оценка сложности. Какова сложность поиска в глубину? Во-первых, замечу, что здесь, как и на любых задачах на графы, 1. сложность принято оценивать как функцию *двух* параметров — количества вершин V и количества рёбер E, и 2. сложность зависит от того, каким образом мы храним граф в памяти.

В первом варианте реализации очевидно, что процедура $find$ ни для какого параметра не будет вызвана дважды (т.е. $find(1)$ будет вызвана максимум один раз за время работы программы, $find(2)$ тоже максимум один раз и т.д.), поскольку перед каждым вызовом мы проверяем, а не были ли мы уже в этой вершине[3]. Поэтому общее количество вызовов будет $O(V)$. Сложность работы каждой процедуры (не считая времени работы рекурсивных вызовов) есть $O(V)$, т.к. в ней просто цикл, поэтому общая сложность поиска в глубину будет $O(V^2)$.

Для второго варианта оценка сложности будет, конечно, в точности такая же: *полноценных* запусков процедуры, т.е. таких, которые не выйдут тут же по первой проверке, будет тоже $O(V)$, а время, потраченное на остальные (на каждый — $O(1)$ времени), можно учесть во времени выполнения цикла в вызывающей процедуре, таким образом сложность $O(V^2)$. Понятно, что вообще в общем случае от переноса проверки в начало процедуры сложность работы алгоритма не изменится, поскольку общее количество действий фактически осталось тем же (проверок будет столько же; добавится только время на вызовы функций, но лишних вызовов будет столько же, сколько и проверок, поэтому сложность не изменится).

Это все относится к случаю, когда граф мы храним матрицей смежности. Но можно хранить граф списком смежных вершин или любым другим способом, позволяющим перебрать соседей вершины за $O($*количество этих соседей*$)$ (например, списком рёбер, отсортированным по первой вершине, или вообще не хранить граф, а вычислять соседние вершины «на лету», как, например, в различных задачах типа хождения коня по шахматному полю — там мы, конечно, не будем хранить ребра вообще, а будем просто перебирать все клетки, на которые можно попасть с текущей). Тогда суммарное время работы всех таких переборов будет $O($*суммарное количество соседей всех вершин*$)$, т.е. $O(E)$, а общее время оставшейся работы будет $O(V)$, т.е. общее время работы алгоритма будет $O(V+E)$. В большинстве случаев $E > V$, поэтому часто говорят, что сложность работы поиска в глубину есть $O(E)$. Это, в общем-то, не совсем корректно, но ошибка обычно не страшна (т.к., например, обычно в ограничениях задачи все-таки $maxE > maxV$, и т.п.). Таким образом, достаточно точно можно сказать, что время работы поиска в глубину на списке смежных вершин есть $O(E)$. Замечу особый случай: если степень вершин графа не превышает некоторой маленькой константы (например, при хождении коня по шахматной доске степени вершин не превышают 8), то $E = O(V)$ и сложность работы алгоритма есть $O(V)$.

По-видимому, сейчас при использовании поиска в глубину в большинстве случаев не стоит использовать матрицу смежности, т.к. нередки задачи с ограничениями типа $V \leqslant 10\,000$, $E \leqslant 100\,000$, так что $O(E)$-алгоритм пройдёт, а $O(V^2)$ — нет. Но, как всегда, выбор способа хранения графа в каждой задаче свой. Анализируйте ограничения и соответственно выбирайте способ хранения графа.

§6. Дополнительные замечания. Нередко при обсуждении элементарного поиска в глубину сразу дают ещё кучу информации, например, деление вершин на *три* класса: непосещённые, обрабатываемые сейчас и уже обработанные (вместо, как у нас, двух классов — в которых мы ещё не побывали и в которых мы уже побывали), классификация рёбер, времена входа/выхода и т.п. Но на самом деле они бывают нужны *далеко не во всех* применениях поиска в глубину, поэтому я буду говорить о них только тогда, когда они понадобятся.

Часть II. Простые задачи, решаемые поиском в глубину

В этом разделе почти везде будем рассматривать только неориентированные графы.

§1. Компоненты связности графа. Довольно очевидно доказывается, что поиск в глубину, запущенный из некоторой вершины графа, обойдёт все вершины из той компоненты связности, в которой находится наша вершина.

[3]Конечно, подразумевается, что вызов процедуры $find$ из главной программы сделан соответствующим образом — чтобы не получилось, что процедура будет дважды запущена с одним и тем же параметром

Докажем это. Действительно, пусть есть вершина, лежащая в той же компоненте[4] связности, но до которой мы не дошли. Раз она лежит в той же компоненте связности, то есть путь из начальной вершины в неё. В начальной вершине пути мы точно побывали (мы ведь оттуда запустились), в конечной вершине нет (по предположению). Тогда очевидно, что вдоль пути найдутся две *соседние* вершины u и v такие, что в u мы побывали, а в v нет. Раз они соседние в нашем пути, то они соединены ребром. Тогда спрашивается: что мы делали, когда просматривали соседей вершины u? Почему не пошли в v? Противоречие, следовательно, мы действительно обойдём всю соответствующую компоненту связности.

Примечание: Как видите, доказательство от противного. Мне кажется, что это довольно хорошо подчёркивает саму суть поиска в глубину: сразу далеко не очевидно, что оно работает. Попытки в лоб доказать, что оно работает, не пройдут. Но тем не менее оно действительно работает.

Естественно, мы считаем, что перед запуском массив *was* по крайней мере в пределах этой компоненты связности был заполнен нулями, иначе, конечно, ничего не получится.

Задание 2: *понимаете, почему может ничего не получиться, если массив was изначально неправильно заполнен? Заодно что вы можете сказать по поводу проверки* ориентированного *графа на связность?*

Таким образом, чтобы проверить, что граф (неориентированный!) связен, поступаем очень просто: запускаем поиск в глубину из любой вершины и после этого проверяем, что мы побывали во всех вершинах:

```
fillchar(was,sizeof(was),0);
find(1);
for i:=1 to n do
    if was[i]=0 then
       writeln('Non-connected!');
```

Пусть теперь нам надо найти все компоненты нашего (возможно, не связанного) графа. Т.е. нам надо посчитать их, а в специальный массив для каждой вершины записать номер соответствующей компоненты. Я думаю, это тоже вполне понятно. Запускаемся из первой вершины, потом ищем ещё непосещённую вершину и запускаемся из неё и т.д. Правда, некоторую сложность может составить реализация этого дела, но, если немного подумать, то можно написать так:

```
procedure find(i:integer);
var j:integer;
begin
if was[i]<>0 then
   exit;
was[i]:=nc;
for j:=1 to n do
    if gr[i,j]=1 then
       find(j);
end;
```

```
begin
nc:=0;
fillchar(was,sizeof(was),0);
for i:=1 to n do
    if was[i]=0 then begin
       inc(nc);
       find(i);
    end;
end.
```

Здесь nc — глобальная переменная, хранящая количество уже найденных компонент связности. Номер компоненты связности, как я и обещал выше, мы записываем прямо в массив *was*: компоненты мы нумеруем, начиная с 1, поэтому проблем не возникает. $was[i] = 0$ обозначает, что вершина i ещё не была посещена, иначе вершина была посещена, а $was[i]$ — номер её компоненты связности. Поэтому в процедуре мы делаем именно $was[i] := nc$, а не 1, как раньше.

Ещё обратите внимание, что тут *обязательно* нужна проверка `if was[i]=0` в главной программе, даже несмотря на то, что соответствующая проверка написана *в начале* процедуры. Действительно, проверка в главной программе нужна, чтобы не увеличить число компонент лишний раз.

И наконец заметьте, что, конечно, массив *was* мы инициализируем один раз за программу.

§2. Проверка графа на двудольность. N-дольным называется неориентированный граф, вершины которого можно раскрасить в N цветов так, что концы любого ребра будут разноцветными, т.е. не будет рёбер, ведущих из одного цвета в тот же самый цвет. В таком случае группы вершин одного цвета называются *долями* графа. В частности, двудольный граф — граф, вершины которого можно разбить на две группы-доли так, что каждое ребро будет вести из одной доли в другую.

Нередко двудольные графы являются двудольными по своей природе, т.е. нередко сама природа вершин разных долей разная: классический пример — первая доля — люди (работники), вторая — работы, которые они могут выполнять, ребро между человеком и работой есть, если он умеет её выполнять. Очевидно, что тут ребра в пределах одной доли совершенно бессмысленно. В таких случаях обычно сразу во входном файле задаётся граф так, что он не может оказаться недвудольным, и вообще вопрос о проверке графа на двудольность бессмысленен. Но бывает так, что дан просто граф, а надо проверить, является ли он двудольным. Именно такую задачу мы и будем рассматривать здесь. Одновременно с проверкой на двудольность мы сразу будем находить его доли.

[4]Кстати, грамматический вопрос: какого рода слово «компонента»? Вроде очевидно женского, но не является ли это ошибкой, ведь нормальное слово-то — компонент (прибора, например)? На самом деле в соответствующих словарях чётко зафиксировано выражение *компонента связности*, *компонента вектора* и т.п., так что в математике это вроде не ошибка.

Задание 3: *может ли эта задача иметь несколько решений? Т.е. может ли быть так, что разбиение вершин графа на доли неоднозначно? Попробуйте сформулировать как можно более простой критерий, который отвечает на этот вопрос. Только, прежде чем читать дальше, ответьте на это задание.*

Итак, нам дан граф. Давайте попробуем его покрасить. Возьмём первую вершину и покрасим её в какой попало цвет (т.е. отнесём её к какой попало доле). Тогда сразу понятно, как надо красить соседние с ней вершины. Покрасим их как надо. После этого понятно, как надо красить соседние с ними вершины и т.д. (Довольно сильно напоминает волновой алгоритм.) Так будет продолжаться до тех пор, пока не случится одно из двух:

1. Возникнет противоречие, т.е. мы должны будем покрасить одну и ту же вершину в разные цвета одновременно или должны будем *перекрашивать* уже покрашенную вершину. Что это обозначает? Единственный произвол, который мы делали, состоял в выборе цвета самой первой вершины. Очевидно, что, если мы попробуем другой вариант цвета первой вершины, то противоречие сохранится, просто цвета всех покрашенных вершин инвертируются. Тогда очевидно, что граф не двудольный.

2. Нам будет нечего делать, т.е. мы покрасили несколько вершин, противоречий нет, но ни у одной из уже покрашенных вершин нет непокрашенных соседей. Что это значит? Одно из двух: либо мы покрасили весь граф — круто, задача решена, ответ положительный (**Контрольный вопрос 4:** *ответ на какой вопрос? :)*). Либо есть ещё непокрашенные вершины. Но ясно, что тогда они находятся в *другой* компоненте связности и потому их можно красить *независимо* от уже покрашенных. Возьмём любую из ещё непокрашенных вершин и покрасим её как попало и т.д., продолжая как описано выше. Опять либо возникнет противоречие, тогда граф точно не двудольный, т.к. на это противоречие влиял только самый последний произвол, а его инвертировать опять бессмысленно (а предыдущие выборы, которые мы делали, не имеют теперь значения), либо опять будет нечего красить — аналогично либо все покрасили, либо переходим к третьей компоненте и т.д.

Таким образом в конце концов мы или покрасим весь граф, или придём к выводу, что граф не двудольный. Прежде чем обсуждать реализацию, обсудим ещё небольшой теоретический вопрос.

Можно ли придумать какой-нибудь критерий двудольности графа? Давайте подумаем, когда «затыкается» наш алгоритм. Когда обнаруживает противоречие, т.е. одну и ту же вершину пытается сразу покрасить и в белый, и в чёрный цвет. Говоря по другому, когда у одной ещё непокрашенной вершины находятся два *разноцветных* соседа. Что это обозначает? До сих пор все было нормально, т.е. на каждом ребре цвет чередовался, поэтому цвета обозначают фактически «слои» графа: до вершин одного цвета от начальной мы добираемся за чётное число шагов (рёбер), до вершин второго цвета — за нечётное. Если же появилось противоречие, значит, нашлась вершина, до которой мы можем добраться и за чётное, и за нечётное количество шагов. Это обозначает, что появился *цикл нечётной длины*: от начальной вершины до неё самой можно добраться за *нечётное* количество шагов. Очевидно, что в двудольном графе не может быть циклов нечётной длины: в любом цикле вершины разных долей чередуются, и потому, чтобы вернутся в начальную вершину, надо сделать *чётное* количество шагов. Поэтому ясно, что, раз наш алгоритм нашёл-таки такой цикл, то граф точно недвудольный. А теперь заметим самое главное: если циклов нечётной длины в графе точно нет, то наш алгоритм в принципе не сможет «заткнуться», т.е. он корректно раскрасит граф, т.е. граф двудольный. Таким образом, мы доказали это утверждение в обе стороны: граф двудольный тогда и только тогда, когда в нем нет циклов нечётной длины.

Все то же самое, но по-другому изложенное (это изложение, может быть, сложнее с ходу понять, и тем более не понятно, как до него дойти, но зато оно очень хорошо помогает разложить все по полочкам, чтобы быть уверенными, что мы нигде ничего не сглючили):

Теорема (о двудольности графа)**:** *граф двудольный тогда и только тогда, когда в нем нет циклов нечётной длины.*

Доказательство:

$\Rightarrow$ Пусть граф двудольный. Тогда в нем вдоль каждого цикла цвета вершин чередуются, поэтому цикл обязательно имеет чётную длину.

$\Leftarrow$ Пусть в графе нет циклов нечётной длины. Запустим вышеприведённый алгоритм. Он может остановиться, найдя противоречие, только если найдёт цикл нечётной длины, что невозможно. Следовательно, он корректно раскрасит граф, значит, граф двудольный. чтд.

Обратите внимание, что доказательство в одну сторону сильно отличается от доказательства в обратную. Ещё обратите внимание, что отсюда очевидно следует, что дерево (и вообще лес) — двудольный граф.

Как теперь реализовать этот алгоритм? Напрашивающаяся идея — поиск в ширину, он же волновой алгоритм. Вполне можно.

Задание 5: *реализуйте этот алгоритм с помощью поиска в ширину.*

Но если немного подумать, то подойдёт *любой* обход графа, который переходит из одной вершины в другую только по рёбрам и обходит весь граф. Например, вполне подойдёт поиск в глубину; поскольку

поиск в глубину реализовать обычно проще, чем в ширину, то обычно проверяют граф на двудольность с помощью поиска в глубину.

Итак, каждый раз, когда находим новую вершину, будем её красить в нужный цвет.

```
procedure find(i:integer);
var j:integer;
begin
for j:=1 to n do
    if gr[i,j]=1 then begin
       if was[j]=was[i] then begin
          ok:=false;
          exit;
       end;
       if was[j]<>0 then
          continue;
       was[j]:=3-was[i];
       find(j);
       if not ok then
          exit;
    end;
end;

...
fillchar(was,sizeof(was),0);
ok:=true;
for i:=1 to n do
    if was[i]=0 then begin
       was[i]:=1;
       find(i);
       if not ok then
          break;
    end;
```

Итак, что тут. Массив was опять используем для хранения дополнительной информации: в данном случае цвета вершины (1 или 2). Может быть, логичнее его было бы назвать как-нибудь по-другому. В процедуре $find$, когда находим очередного соседа текущей вершины, смотрим: если он того же цвета, что и мы, то облом, иначе если он уже покрашен, то туда не сунемся, иначе красим ($3 - was[i]$ даёт как раз нужный цвет) и запускаем поиск из этой вершины. Обратите внимание, что красим вершину (т.е. заполняем $was[j]$) мы *до* входа в процедуру $find(j)$, поэтому в начале процедуры ничего вообще не делаем. В главной программе теперь найдя ещё непокрашенную вершину, красим её (обязательно! т.к. не красим её в самой процедуре) и запускаемся. Обратите внимание, как сделана работа с переменной ok, которая хранит, не наши ли мы ещё противоречия.

Какой-то ужас тут получается. Поэтому имхо логичнее перенести всю работу в начало процедуры, а — внимание! — в процедуру будем передавать *дополнительный* параметр — цвет, в который надо покрасить эту вершину.

```
procedure find(i,c:integer);
var j:integer;
begin
if was[i]<>0 then begin
   if was[i]<>c then
      ok:=false;
   exit;
end;
was[i]:=c;
for j:=1 to n do
    if gr[i,j]=1 then begin
       find(j,3-c);
       if not ok then
          exit;
    end;
end;

...
fillchar(was,sizeof(was),0);
ok:=true;
for i:=1 to n do
    if was[i]=0 then begin
       find(i,1);
       if not ok then
          break;
    end;
```

Теперь работа процедуры $find$ имхо более очевидна: она пытается покрасить вершину i в цвет c. Во-первых, если вершина уже покрашена, то надо только посмотреть, в тот ли цвет (обратите внимание, что в прошлом варианте была проверка $was[i] = was[j]$, а теперь $was[i] <> c$), и выйти. Иначе красим и смотрим соседей.

Ещё замечу, что, если в случае, когда граф недвудольный, нужно сделать что-то простое и завершить работу программы (например, вывести '`No solution`' и выйти), то можно с ok не возиться, а просто сделать что надо:

```
procedure outno;
begin
...
halt;
end;


procedure find(i,c:integer);
var j:integer;
begin
if was[i]<>0 then begin
   if was[i]<>c then outno;
   exit;
end;
was[i]:=c;
for j:=1 to n do
    if gr[i,j]=1 then begin
       find(j,3-c);
    end;
end;
```

Ещё замечу, что можно переменную ok убрать, а процедуру $find$ сделать функцией, возвращающей boolean. Можете попробовать это реализовать. Наконец, если гарантируется, что граф двудольный, надо только его доли найти, то ok вообще не нужна.

Задание 6: *А почему также нельзя проверять граф на трехдольность?*

§3. Проверка, является ли граф деревом. Как вы знаете, деревом называется связный граф без циклов (все ещё рассматриваем только неориентированные графы), лесом называется произвольный, т.е. не обязательно связный, граф без циклов. Как проверить, является ли граф деревом или лесом? В принципе, понятно: проверить, что циклы в графе отсутствуют, можно просто запустив поиск в глубину и посмотрев, не придём ли мы когда-нибудь в ту вершину, где мы уже побывали.

Конкретно: если нам надо проверить, является ли граф деревом, то запустимся из первой вершины. Если хоть раз вернёмся в вершину, где мы уже побывали, то граф точно не дерево. Иначе в конце проверим, верно ли, что мы побывали во всех вершинах. Если да, то граф связен, а отсутствие циклов мы уже проверили — ок. Иначе не дерево.

Если нам надо проверить, является ли граф лесом, то все аналогично, только аналогично поиску всех компонент связности закончив поиск в глубину из первой вершины, запускаемся из первой ещё не посещённой и т.д. — такой же цикл, как и при поиске компонент связности.

Задание 7: *Напишите эти две программы. Тщательно потестируйте их. Переберите все возможные подлые случаи. Представьте, что вы — жюри на некоторой олимпиаде и даёте участникам такую задачу. Вы знаете, как её решать, поэтому можете продумать, какие тут подлости возможны — на них и делайте тесты. Например, очевидно, надо оттестировать связные и несвязные графы; деревья, леса, несвязные графы, у которые первая компонента является/не является деревом; линейные структуры (т.е. первая вершина связана со второй, вторая — с третьей и т.д.) и разветвлённые деревья; длинные циклы, пустые графы и т.д.*

При написании программы почти наверняка вы столкнётесь с (одним) большим подводным камнем. Осознайте его, поймите, почему ваша программа не работает, и исправьте программу так, чтобы она работала.

Текст программы тут я приводить не буду, приведу только в ответах. У этой задачи есть подсказка; порешав задачу, посмотрите подсказку до ответа.

§4. Нахождение эйлерова пути и цикла. Я думаю, вы знаете, что такое эйлеров цикл — это цикл, который проходит по каждому ребру ровно один раз. Аналогично, эйлеров путь — это путь, который по каждому ребру проходит ровно один раз (но, в отличии от цикла, может начинаться и заканчиваться в разных вершинах). Я также думаю, что вы знаете критерий наличия эйлерова цикла и эйлерова пути в графе. Действительно, если в графе есть эйлеров цикл, то в при движении по нему в каждую вершину мы входим ровно столько же раз, сколько выходим. За время прохода по всему циклы мы прошли по все рёбрам, инцидентным данной вершине, следовательно, степень каждой вершины *должна быть* чётна (мы пока все ещё рассматриваем неориентированные графы). Совершенно аналогично, если в графе есть эйлеров путь, то степени только двух вершим могут быть нечётны — это будут начало и конец нашего пути: из начала мы вышли на один раз больше, чем вошли в него, с концом пути все наоборот. Ещё, очевидно, надо поставить некоторое условие на связность графа. Мы будем дальше считать граф связным, но это не есть *необходимое* условие.

Задание 8: *Попробуйте сформулировать это условие абсолютно точно, т.е. указать, какое условие на связность графа надо добавить к условию на степени вершин, чтобы получить критерий существования эйлерова цикла/пути в графе, такой, что, если он выполняется, то путь/цикл точно есть, иначе точно нет.*

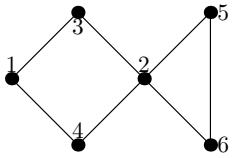
Обратите внимание, что вышеприведённые рассуждения не доказывают *существования* цикла/пути, если эти условия выполняются. Существование цикла мы будем доказывать построением алгоритма, который будет решать эту задачу. Наш алгоритм будет находить цикл/путь в любом связном графе, удовлетворяющим условию на степени вершин (что делать для несвязного графа — это ваше задание).

Итак, пусть граф связен и степени всех его вершин чётны. Построим эйлеров цикл. Пожалуй, тут проще будет привести сам алгоритм, а потом объяснить, почему он работает. Итак, запустим поиск в глубину, но *не* будем контролировать возвращение в уже посещённые вершины: будем допускать сколько угодно раз приходить в одну и ту же вершину (очевидно, что в общем случае эйлеров цикл будет через каждую вершину проходить по нескольку раз, поэтому ясно, что без этого изменения поиск в глубину не поможет). Зато будем стирать ребра из графа, как только мы по ним прошли. Ясно, что тогда алгоритм до бесконечности работать не будет. Когда будем *выходить* из вершины, будем выводить её номер в выходной файл:

```
procedure find(i:integer);
var j:integer;
begin
for j:=1 to n do
    if gr[i,j]=1 then begin
       gr[i,j]:=0;
       gr[j,i]:=0;
       find(j);
    end;
write(i,' ');
end;
```

Обратите внимание, что стираем ребра мы *двумя* присваиваниями, ведь каждому ребру в матрице смежности соответствуют две единички.

Утверждается, что после работы этого алгоритма (точнее, после выполнения команды $find(1)$) на экран будет выведена последовательность вершин, которая образует эйлеров цикл. Чтобы понять это лучше, пожалуй, стоит разобрать простой пример. Рассмотрим граф, показанный справа — в нем, очевидно, есть эйлеров цикл. Как будет работать наш алгоритм?



find(1)
нашли соседа — вершину 3, стираем ребро 1–3
    find(3)
    нашли соседа — вершину 2, стираем ребро 3–2
        find(2)
        нашли соседа — вершину 4, стираем ребро 2–4
            find(4)
            нашли соседа — вершину 1, стираем ребро 4–1
                find(1)
                (все ребра из вершины 1 уже стёрты, поэтому никаких соседей не находим)
                writeln(1);
                завершаем процедуру find(1)
            (больше никуда из вершины 4 не идём, т.к. все ребра уже стёрты)
            writeln(4);
            завершаем процедуру find(4)
        (продолжаем поиск из вершины 2)
        нашли соседа — вершину 5, стираем ребро 2–5
            find(5)
            нашли соседа — вершину 6, стираем ребро 5–6
                find(6)
                нашли соседа — вершину 2, стираем ребро 6–2
                    find(2)
                    (больше никуда из вершины 2 не идём, т.к. все ребра уже стёрты)
                    writeln(2);*
                    завершаем процедуру find(2)
                (продолжаем поиск из вершины 6, но ничего больше не находим)
                writeln(6);
                завершаем процедуру find(6)
            (больше никуда из вершины 5 не идём, т.к. все ребра уже стёрты)
            writeln(5);
            завершаем процедуру find(5)
        (продолжаем поиск из вершины 2, но ничего больше не находим)
        writeln(2);
        завершаем процедуру find(2)
    (продолжаем поиск из вершины 3, но ничего больше не находим)
    writeln(3);
    завершаем процедуру find(3)
(продолжаем поиск из вершины 1, но ничего больше не находим)
writeln(1);
завершаем процедуру find(1)

(пометка звёздочкой будет использоваться ниже)

Все. Вывели следующую последовательность на экран:

1 4 2 6 5 2 3 1

Это действительно эйлеров цикл, но, если сравнить с тем, как мы ходили по графу, то выглядит это очень странно: цикл получается какой-то каракатицей, проходя по рёбрам в обратную сторону по сравнению с тем, как мы по ним ходили при поиске в глубину. Я не буду строго доказывать, что этот алгоритм корректно находит цикл; пожалуй, самый лучший способ проверить его работу — это вручную промоделировать его работу на разных графах, стараясь придумать случай поподлее. Только скажу идею обоснования корректности работы. Текста много, но по-моему, он простой: я просто расписываю все подробно и повторяю по несколько раз :)

Итак, мы запустились из первой вершины v_1 и пошли в её соседа v_2, стерев по пути ребро. В результате степень как v_1, так и v_2, стала нечётной, т.к. изначально по условию они были чётными. Но это обозначает,

что степень вершины v_2 теперь точно не равна нулю — значит, у неё есть *ещё как минимум один* сосед v_3, в который мы можем пойти. Пойдя в него, мы сотрём ребро v_2–v_3 и степень вершины v_2 опять станет чётной, зато степень v_3 станет нечётное. Значит, и у неё *есть ещё как минимум один сосед* v_4, в который мы и пойдём. И так далее, в каждый момент степени всех вершин будут чётными, за исключением первой v_1 и последней, в которую мы только что зашли, v_k. У этих двух вершин степени будут нечётными. Но это будет обозначать, что у *каждой* текущей вершины будет *ещё как минимум один сосед* и мы сможем идти так до бесконечности?! Что-то тут не так, мы же выяснили, что алгоритм бесконечно работать не может, в конце концов просто ребра кончатся... Значит... о! значит, в какой-то момент мы вернёмся в начальную вершину v_1! При этом мы сотрём ребра вдоль целого цикла и потому степени всех вершин будут чётными. Если степень вершины v_1 не ноль, то мы пойдём дальше, и совершенно аналогичными рассуждениями можно будет доказать, что мы опять вернёмся в неё... И так далее, до тех пор, пока на очередном возвращении в v_1 её степень не станет равной нулю. Тогда мы её и выведем в выходной файл.

Внимание! Это, пожалуй, конец основной идеи всех рассуждений. Мы доказали, что *всегда первой выведенной в выходной файл вершиной будет та, с которой мы начали*; с самого начала это имхо было весьма не очевидно. Если вы это ещё не осознали, попробуйте ещё раз это продумать; может быть, порисуйте примеры графов и посмотрите. Ещё раз кратко повторю основную идею: в каждый момент времени в каждую вершину в графе мы успели поровну раз войти и выйти, кроме самой первой вершины и той вершины, в которой мы находимся. Поэтому у всех вершин степени остались чётными, кроме этих двух вершин — если они не совпадают, то у них степени нечётные. Но тогда из текущей вершины мы *можем* пойти дальше. Значит, мы остановимся только эти две вершины будут совпадать, т.е. когда (в очередной раз) вернёмся в первую вершину. Значит, именно первую вершину мы и выведем первой.

Итак, что же дальше? А дальше, после того, как мы вывели вершину, про неё можно забыть: у неё степень точно ноль (т.к. мы не смогли никуда дальше пойти), поэтому в неё мы никогда больше не вернёмся (точнее, можем вернуться, но только откатываясь при выходе из рекурсии). Дальше мы будем откатываться по рекурсии назад и выводить все вершины по пути. Но они точно будут связаны рёбрами в исходном графе, т.к. мы по этим рёбрам шли вперёд. Значит, пока мы выводим корректный путь. Дальше в очередной момент мы выведем некоторую вершину u и откатимся до вершины v'_1, из которой будет куда пойти ещё (как вершина 2 в нашем примере); саму вершину v'_1 мы ещё не выведем к этому моменту. Мы пойдём в её соседа v'_2 и... опять попадём в такую же ситуацию, как уже было: у вершин v'_1 и v'_2 степени нечётные, а у остальных чётные. Поэтому из v'_2 мы пойдём куда-нибудь ещё и т.д.; остановиться мы сможем *только в* v'_1. (Обратите внимание, что степень *самой начальной* вершины v_1 уже давно ноль, и поэтому в ней мы, конечно, не сможем остановиться — не зря мы про неё забыли). Значит, мы выведем v'_1. Возникает вопрос: а корректно ли? Соединена ли она ребром с той вершиной, которую мы вывели перед этим? Да, конечно. Т.к. перед этим мы вывели u, а u — это та вершина, в которую мы в своё время, давным-давно, пошли из v'_1: ведь мы сравнительно недавно откатились из u в v'_1. Значит, вывод v'_1 корректен (если вы совсем запутались, то проследите это на нашем примере: тут $v'_1 = 2$, $u = 4$, а обсуждаем мы корректность вывода 2 в операторе, помеченном звёздочкой. Вообще, переводите все рассуждения на наш пример, он, по-моему, довольно хорошо иллюстрирует тут все, о чем я говорю).

Так что наш вывод все ещё будет корректным. Далее мы опять будем откатываться назад до тех пор, пока не откатимся в вершину, откуда будет куда идти, и т.д. — а там опять все будет аналогично и т.д.... Наконец, *последней* мы выведем ту же вершину, что и первой вывели: действительно, ведь мы из главной программы запустили $find(v_1)$, поэтому *последняя* процедура find, из которой мы выйдем, будет именно эта, и последней выведенной вершиной будет v_1; а выше мы видели, что первой выведенной будет она же — т.е. действительно цикл замкнётся.

Более-менее понятно, что алгоритм работает. Правда, не уверен, что вышеприведённые рассуждения можно превратить в *строгое доказательство* (т.е. раскрыть «и т.д.» так, чтобы все было строго); может быть, строго все доказывается методом от противного — если честно, не знаю... Но идея, я надеюсь, ясна.

А тогда ясно и то, что нужно делать для случая поиска эйлерова *пути* и для работы в ориентированных графах. А именно, для эйлерова пути надо просто найти одну из двух вершин с нечётной степенью (пусть вершину v_1) и запуститься из неё. Аналогичными рассуждениями можно объяснить, что первая вершина, которую мы выведем, будет *другая* вершина с нечётной степенью, и что то, что мы выведем действительно путь в графе, и он закончится в вершине v_1.

В ориентированном графе несколько хитрее. Во-первых, там критерий немного другой: для цикла там надо требовать равенства входящей и выходящей степени для каждой вершины (т.е. равенства количеств входящих и выходящих рёбер). Во-вторых, поскольку мы выводим путь «каракатицей», то идти по рёбрам в поиске в глубину надо *навстречу* стрелкам, чтобы окончательный путь шёл как положено. Рассуждения, объясняющие корректность, проводятся аналогично. Ещё не забудем, что удалять обратное ребро тут не надо (т.е. когда идём из i в j, надо стирать только ребро $i \to j$, а $j \to i$ не надо).

Задание 9: *Напишите и оттестируйте алгоритм поиска эйлерова цикла в ориентированном графе.*

(На самом деле, конечно, я надеюсь, что вы напишите *все* алгоритмы, которые тут обсуждаются, но это — особо важное задание :))

Задание 10: *(немного более сложное задание). Подумайте, как искать эйлеров* путь *в ориентированном графе. А именно, каковы критерии существования пути в ориентированном графе? Как надо писать алгоритм? Почему он будет работать? Напишите и, конечно, потестируйте его.*

Ещё отмечу, что всё это работает и для случая кратных рёбер, петель и т.д. (петля увеличивает степень соответствующей вершины на два). Алгоритм даже не придётся менять, кроме того, что надо аккуратнее хранить граф и стирать ребра (т.е., например, в матрице смежности хранить *число* рёбер между вершинами, которое может быть и больше 1 — в случае кратных рёбер — и стирать ребро уменьшением соответствующего элемента матрицы смежности).

А теперь немного обсудим сложность этого алгоритма. В той его реализации, которая приведена выше, сложность оценить непросто, но, пожалуй, можно так. Время работы одной процедуры, не считая рекурсивных вызовов, будет $O(V)$. Всего вызовов процедур будет E, ведь именно столько вершин мы в итоге выведем. Поэтому все работает за $O(VE)$.

Но, если подумать, то ясно, что алгоритм на самом деле делает кучу лишней работы. Действительно, если в $find(i)$ мы уже дошли до вершины j, то точно все предыдущие ребра мы уже стёрли. Тогда, когда если мы в очередной раз запустим $find(i)$, нам не надо будет перебирать все вершины сначала, можно начинать с $j+1$. (Речь не идёт о том, что нам делать, когда мы *вернёмся* на тот уровень рекурсии, где мы дошли до вершины j, а о том, что на более глубоком уровне рекурсии мы можем опять запустить $find(i)$). Можно, например, в особом массиве хранить, на какой вершине мы остановились, просматривая соседей i-ой (т.е. в $cur[i]$ будем хранить, какого последнего соседа у i мы смотрели), и изменить цикл в $find$ на что-нибудь типа

```
while cur[i]<n do begin
      inc(cur[i]);
      if gr[i,cur[i]]<>0 then begin
         gr[i,cur[i]]:=0;
         gr[cur[i],i]:=0;
         find(cur[i]);
      end;
end;
```

Теперь вроде должно бы работать быстрее (типа за $O(V^2)$; но этот код я не продумывал до конца, вдруг здесь есть какие-нибудь подводные камни), но по-моему ещё проще написать все, если хранить граф списком соседних вершин (вообще, все основанное на поиске в глубину будет быстрее работать на списке соседних вершин — я уже говорил про это). Я, пожалуй, не буду приводить здесь соответствующей реализации (тут надо быть осторожным с удалением обратных рёбер, т.е. когда идёте из вершины i в j, удалить не только ребро $i \to j$, но и ребро $j \to i$; как следствие, для ориентированных графов, где удалять второе ребро не надо, тут все проще). Тем не менее это позволяет добиться времени работы $O(E)$, как и всех остальных алгоритмов на поиске в глубину (т.е. лишней работы мы тут делать не будем, только ходить по рёбрам — по каждому по разу — и выводить вершины).

Часть III. Поиск в глубину для ориентированных графов: топологическая сортировка и смежные вопросы

На первый взгляд может показаться, особенно после прочтения первой части этого текста, что поиск в глубину для *ориентированных* графов ничего хорошего не даст. Но это не так. Как ни странно, но ориентированные графы — пожалуй, наиболее часто используемая арена для поиска в глубину. В частности, топологическая сортировка ациклического графа очень часто бывает нужна, наверное, намного чаще, чем проверка графа на двудольность/на дерево...

В общем, в этом разделе мы будем рассматривать ориентированные графы.

§1. Топологическая сортировка. *Ациклическим* называется ориентированный граф, в котором нет (ориентированных, конечно) циклов. **Контрольный вопрос 11:** *Понимаете ли вы, что этот класс намного шире, чем деревья? Т.е. что ациклический граф — это не только ориентированное дерево?*

Самая основная задача, которая может быть поставлена на ациклических графах — это *топологическая сортировка* (сокращённо топсорт) такого графа. Несмотря на страшное название, на самом деле это просто. Задача состоит в том, чтобы пронумеровать вершины графа так, чтобы все ребра шли из вершин с меньшим номером в вершину с бóльшим. Или, говоря по-другому, расположить вершины на прямой так, чтобы все ребра шли слева направо. Классическая легенда задачи гласит: есть N дел, которые надо сделать. При этом про каждое дело известно, какие дела надо сделать обязательно *до* него (и, соответственно, известно, какие надо сделать *после*). Требуется определить порядок, в котором нужно делать эти дела.

Очевидно, что, если в графе есть циклы, то задача решения не имеет. Оказывается, если циклов нет, то задача имеет решение; как вы уже, наверное, догадались, доказывать мы это будем предъявлением соответствующего алгоритма.

На самом деле есть *два* довольно существенно разных алгоритма решения этой задачи. Первый вообще не использует идеи поиска в глубину[5], довольно очевиден идейно, но не очень тривиален в реализации. Он приведён в «Искусстве программирования для ЭВМ» Кнута и потому я буду называть его методом Кнута (он используется редко, и я не знаю, есть ли у него стандартное название. Кстати, когда говорят «алгоритм топологической сортировки», обычно понимают не этот, а второй). Состоит этот алгоритм в следующем: в любом ациклическом графе точно есть «сток», т.е. вершина, из которой не выходит ни одного ребра (т.е. вершина, у которой *исходящая степень* равна нулю). Очевидно, её можно поставить последней. После этого как бы мы ни ставили остальные вершины, проблем с рёбрами, идущими в *эту* вершину, не возникнет: они в любом случае будут идти направо. Тогда эту вершину вместе со всеми входящими в неё рёбрами (а исходящих, мы помним, нет) можно выкинуть из графа. Получившийся граф также будет ациклическим, и в нем тоже обязательно будет сток. Поставим его на предпоследнее место в массиве и выкинем из графа. И так далее. Очевидно, что в итоге все получится: поскольку каждую вершину мы ставили только тогда, когда она становилась стоком, то проблем ни с какими рёбрами не получится.

Кстати, я надеюсь, очевидно, почему в любом ациклическом графе обязательно найдётся сток? Если его нет, то встанем в произвольную вершину. Из неё есть выходящее ребро — пойдём по нему. Дальше опять найдётся выходящее ребро — и т.д. Гулять по графу мы сможем до бесконечности, а это обозначает, что вершины обязательно повторятся — найдётся цикл. Противоречие.

Таким образом, алгоритм: находим какой-нибудь сток, ставим его на последнее место в выходном массиве, выкидываем его (а на самом деле просто помечаем в соответствующем массиве, что он выкинут, и в будущем при расчёте исходящей степени не учитываем его). Дальше находим новый сток и т.д., пока не расставим все вершины. Если на очередном шагу стока не найдётся, значит, граф не ациклический. Заметьте, что мы таким образом доказали, что любой ациклический граф можно оттопсортить.

В простейшей реализации этот алгоритм будет работать за $O(V^3)$: действительно, каждый поиск стока занимает $O(V^2)$: перебрать все вершины и для каждой посчитать исходящую степень — а таких поисков надо сделать V. Но можно на самом деле все ускорить, и в итоге получить программу, работающую за $O(E)$ (как всегда, при условии, что граф хранится списком смежных вершин).

Задание 12: *Напишите эту программу. Попробуйте придумать, как можно заставить её работать быстрее. На самом деле это делается в два этапа: во-первых, заметим, что каждый раз пересчитывать исходящую степень не надо, а на втором этапе вам пригодится структура данных очередь или стек. Попробуйте придумать это сами. Если не придумаете, то смотрите подсказки, которые есть по этому заданию (впрочем, как и по некоторым другим). Обязательно потом прочитайте подсказки и ответ; в частности, там будет пример хранения графа списком смежных вершин.*

Кстати, замечу, что, очевидно, вместо стоков можно точно также использовать истоки — вершины, у которых *входящая* степень равна нулю, и вообще, задача топологической сортировки обладает соответствующей симметрией: если обратить все ребра, то искомый порядок вершин тоже просто обратится.

Вышеприведённый алгоритм на самом деле очень интересен, а две идеи его ускорения весьма красивы. Но перейдём ко второму алгоритму, который напрямую использует поиск в глубину. Давайте, как и в первом алгоритме, будем заполнять выходной массив справа налево, т.е. от бóльших номеров к меньшим. Подумаем, когда можно поставить некоторую вершину? Очевидно, лишь после того, как поставлены все вершины, в которые из нашей идут ребра. Получаем следующую процедуру, ставящую вершину i в выходной массив:

перебрать все выходящие из i ребра и поставить в выходной массив вершины, в которые эти ребра идут.
После этого поставить нашу.

Но как мы будем ставить эти самые вершины, «в которые эти ребра идут»? Очевидно, *рекурсивным вызовом!* Только не забудем проверить, а вдруг эта вершина *уже* поставлена в выходной массив. А тогда это есть вылитый поиск в глубину:

```
procedure put(i:integer);
begin
if was[i]<>0 then exit;
was[i]:=1;
for j:=1 to n do
    if gr[i,j]<>0 then
       put(j);
записать вершину i в выходной массив;
end;
```

Итак, ещё раз. Процедура *put* ставит вершину i в выходной массив. Прежде чем туда её поставить, она пытается поставить туда все вершины, которые должны идти после i-ой (напомню, что массив мы

[5]Ну, конечно, можно, наверное, найти какие-нибудь идейные сходства у этих двух алгоритмов, и даже, может быть, можно, сильно помучившись, попытаться объявить, что в каком-то смысле они одинаковы... Но, может быть, это вообще верно для *любых* двух решений одной и той же задачи? :)

заполняем с конца); естественно, это делается рекурсивным вызовом. После того, как это выполнено, можно непосредственно поместить i в выходной массив.

То же можно сказать немного по-другому: процедура *убеждается*, что i-я вершина уже стоит в выходном массиве. Если стоит, то ок, иначе процедура помещает её туда с соблюдением всех мер предосторожности. А именно, прежде чем поместить вершину в массив, она перебирает все вершины, которые должны идти после i-ой и *убеждается* (рекурсивным вызовом, конечно), что эти вершины там уже стоят. Таким образом, после вызова $put(j)$ j-я вершина точно будет в выходном массиве. Массив was здесь фактически как раз обозначает, находится ли вершина уже в выходном массиве или нет.

(На всякий случай замечу довольно очевидную на мой взгляд вещь: здесь у нас, вообще говоря, есть *две* схемы нумерации вершин: одна — так, как они заданы во входном файле, вторая — искомая при топологической сортировке. Везде под номеров вершины я понимаю, конечно, номер её так, как она задана во входном файле.)

Как реализовать последнюю строчку в приведённой выше процедуре? Очевидно. Заведём глобальный массив out, в котором будем формировать результат сортировки, и счётчик pos, который будет указывать, какую позиция мы сейчас будем заполнять (т.е. первую свободную позицию при движении справа налево). Изначально $pos = n$: заполнение массива начинаем справа. Тогда получаем следующий алгоритм топологической сортировки (для единообразия переименовал процедуру put в $find$):

```
procedure find(i:integer);
begin
if was[i]<>0 then exit;
was[i]:=1;
for j:=1 to n do
    if gr[i,j]<>0 then
       find(j);
out[pos]:=i;
dec(pos);
end;
```

```
...
fillchar(was,sizeof(was),0);
pos:=n;
for i:=1 to n do
    find(i);
```

Обратите внимание, что вызывать поиск в глубину приходится циклом: *ясно*, что, запустившись из случайной вершины, мы не обязательно обойдём *весь* граф, даже если он связен (а тем более если он несвязен). Можно это же сказать и по-другому: чтобы оттопсортить граф, нам надо *убедиться*, что все его вершины стоят в выходном массиве. Поэтому надо запустить процедуру $find$ из всех вершин.

Задание 13: *(простое) Так ли ясно? Приведите пример связного ориентированного графа, на котором однократно запущенный поиск в глубину не обойдёт все вершины. Не забудьте, что связный ориентированный граф — это такой, который будет связным, если забыть про ориентацию его рёбер. Вспомните доказательство того, что поиск в глубину в неориентированном графе обходит всю компоненту связности, и поймите, почему это доказательство не работает в случае ориентированного графа.*

Обратите внимание, как просто. Десяток строк — и решена такая нетривиальная задача. И думать почти ничего не надо, т.к. это просто поиск в глубину. Но идеи очень глубокие, они ещё активно всплывут в динамическом программировании, например. (Приведённый выше «алгоритм Кнута» тоже, наверное, можно реализовать так коротко, только там думать надо...)

Сложность, как и всегда у поиска в глубину, у приведённого выше алгоритма $O(V^2)$, а, если граф хранить списком соседних вершин, то $O(E)$.

§2. Проверка графа на ацикличность. Как проверить граф на ацикличность? На самом деле все очень просто. Кажется, точно также, как проверять неориентированный граф на то, является ли он лесом, только, может быть, ещё проще. Встанем в произвольную вершину и пойдём поиском в глубину. Если хоть раз вернёмся туда, где уже были, значит, граф точно не ациклический. Хотя... Нет! Ничего подобного!

Задание 14: *Приведите пример ациклического графа, в котором мы при поиске в глубину попадём в вершину, в которой уже были раньше.*

Что же делать? Пожалуй, я могу предложить два варианта. Первый: на самом деле, видимо, цикл найдётся, если мы вернёмся в ту вершину, которую ещё *не закончили* обрабатывать. Т.е. теперь введём *три* состояния вершин: в которой мы ещё не были, которую мы начали обрабатывать, но ещё не закончили, и которую мы уже обработали (их часто называют, соответственно, белыми, серыми и чёрными). Т.е. теперь массив was кроме значений 0 и 1 будет принимать ещё значение 2: вершины, которую мы обработали до конца; это значение мы будем устанавливать на выходе из процедуры $find$. Если немного подумать, то мы нашли цикл тогда и только тогда, когда вернулись в *серую* вершину:

```
procedure find(i:integer);
begin
if was[i]=1 then
   граф не ациклический
if was[i]<>0 then
   exit;
```

```
was[i]:=1;
for j:=1 to n do
    if gr[i,j]<>0 then
       find(j);
was[i]:=2;
end;
```

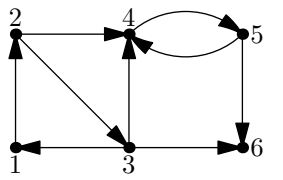
Действительно, в каждый момент «серые» вершины (у которых $was = 1$) образуют *путь* в графе, в точности соответствующий стеку вызовов процедуры $find$. Если мы вернулись в одну из них, то мы точно нашли цикл. Если немного подумать, то видимо верно обратное: что, если граф не ациклический, то мы хотя бы один цикл найдём. Можете над этим подумать, мне это как-то с ходу не очевидно.

Есть второй, совершенно тупой, алгоритм проверки графа на ацикличность: запустим вышеприведённый алгоритм для топологической сортировки (использующий поиск в глубину, а не «алгоритм Кнута»). Он *в любом случае* выдаст какую-то последовательность вершин. Если граф ацикличный, то это будет решение задачи о топсорте. Если граф не ацикличный, то это *точно* не будет решением задачи о топсорте, т.к. в неацикличных графах она решения не имеет. Значит, можно просто проверить, верно ли, что все ребра идут слева направо в полученном графе, и, если да, то граф ациклический, иначе нет.

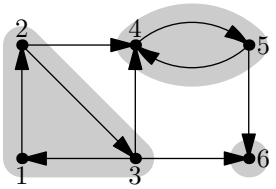
Задание 15: *Попробуйте понять, почему в вышеприведённой идее нельзя в лоб заменить алгоритм топсорта с помощью поиска в глубину на «алгоритм Кнута». Но «алгоритм Кнута» также несложно приспособить для проверки графа на ацикличность; придумайте, как.*

Все эти алгоритмы тоже, конечно, работают за $O(V^2)$ или $O(E)$ в зависимости от способа хранения графа.

§3. Компоненты сильной связности. Говорят, что две вершины u и v находятся в одной компоненте сильной связности ориентированного графа, если существует (ориентированный, конечно) путь как из u в v, так и назад. Несложно видеть, что, если u и v находятся в одной компоненте сильной связности и v и w тоже, то и u и w тоже находятся в одной компоненте сильной связности (это свойство называется *транзитивностью*). Поэтому можно разбить все вершины на компоненты сильной связности так, что каждая вершина будет ровно в одной компоненте; на рисунке справа приведён пример ориентированного графа и разбиения его на компоненты сильной связности (здесь их три).



Задание 16: *Зачем требовать транзитивность? Давайте я попробую определить* компоненты слабой связности *следующим образом: две вершины u и v находятся в одной компоненте слабой связности, если хотя бы в одну сторону есть путь, т.е. или есть путь из u в v, или назад, или и туда и туда. Имеет ли смысл такое определение? Т.е. сумеете ли вы в любом графе разбить все вершины на компоненты слабой связности?*



Замечу, что в ациклическом графе каждая вершина является отдельной компонентой сильной связности, поскольку наличие двух вершин в одной компоненте сильной связности очевидно обозначает наличие цикла.

Итак, поставим задачу нахождения сильносвязных компонент графа. Есть известный алгоритм их поиска. Я не знаю, как до него можно додуматься самим и с трудом представляю, как его доказать (в Кормене есть двухстраничное доказательство его, которое я с большим трудом осознал), поэтому я его доказывать не буду.

Итак, алгоритм следующий. Сначала запустим алгоритм топологической сортировки поиском в глубину — он выдаст нам некоторую последовательность вершин (конечно, в случае неациклического графа она не будет решением задачи о топсорте, но какая-то последовательность вершин получится). После этого сделаем следующее: запустим поиск в глубину ещё раз, но будем двигаться по *инвертированным* рёбрам, а во внешнем цикле поиска будем просматривать вершины в том порядке, в котором нам их выдал топсорт. Т.е. обратим направление каждого ребра графа и запустим поиск в глубину, перебирая во внешнем цикле вершины в этом особом порядке. (Ясно, что скорее всего вы не будете инвертировать ребра, а просто в поиске будете писать условие типа $gr[j,i] <> 0$ вместо $gr[i,j] <> 0$). Утверждается, что те «компоненты связности», которые вы найдёте при втором поиске, как раз и будут сильносвязными компонентами исходного графа.

```
procedure find(i:integer);
var j:integer;
begin
if was[i]<>0 then
   exit;
was[i]:=1;
for j:=1 to n do
    if gr[i,j]<>0 then
       find(j);
ts[pos]:=i;
dec(pos);
end;
```

```
procedure find2(i:integer);
var j:integer;
begin
if was[i]<>0 then
   exit;
was[i]:=nc;
for j:=1 to n do
    if gr[j,i]<>0 then
       find2(j);
end;
```

```
...
fillchar(was,sizeof(was),0);
pos:=n;
for i:=1 to n do
    find(i);
fillchar(was,sizeof(was),0);
nc:=0;
for i:=1 to n do
    if was[ts[i]]=0 then begin
       inc(nc);
       find2(ts[i]);
    end;
```

Ещё раз обратите внимание, как алгоритм состоит из двух частей: первые четыре строки в основном алгоритме — топсорт, остальное в точности повторяет поиск обычных компонент связности в неориентированно графе. Обратите внимание, что для них используются *разные* процедуры find, поскольку, хотя обе процедуры реализуют поиск в глубину, но они делают разные вещи в дополнение к собственно поиску.

Ещё замечу, что тут стандартная ошибка — в рекурсивном вызове во второй процедуре вызвать первую, а не вторую, т.е. написать

```
procedure find2(i:integer);
...
    if gr[j,i]<>0 then
       find(j);
```

Всегда, когда у вас в программе есть две процедуры с похожими именами, следите, чтобы их не перепутать. Особенно если эти процедуры рекурсивны — не перепутайте, где какую функцию в рекурсивном вызове вызывать. Даже более общее утверждение: если есть два похожих куска кода, особенно внимательно просмотрите, не сглючили ли вы где-нибудь. Например, если у вас два поиска в глубину по двум *разным* графам, не перепутайте графы внутри процедур, не перепутайте рекурсивные вызовы и т.д. Особенно это важно, если один блок вы получаете копирование из другого.

Ещё обратите внимание, что во втором поиске используется $gr[j,i] <> 0$: поиск идёт в инвертированном графе.

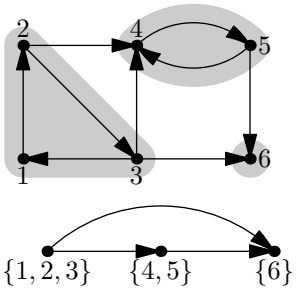
В общем, как я уже сказал, я не знаю, как до этого алгоритма можно дойти, и не знаю, как его легко доказать. Поэтому имхо этот алгоритм нужно примерно запомнить. Собственно, запоминать тут нечего: два поиска в глубину, причём второй — в инвертированном графе. Единственное, что не очевидно, как запомнить, — это то, в каком порядке стоит перебирать вершины при втором поиске в глубину. Но это легко восстанавливается: представьте себе *ациклический* граф. В нем, очевидно, каждая компонента сильной связности — это отдельная вершина. Очевидно, что такие компоненты мы получим, только если будем второй поиск запускать в порядке, который мы получим из топсорта (не забудьте, что второй поиск — в инвертированном графе!). Конечно, это не очевидно, а надо представить себе ациклический оттопсорченный граф в уме и просто прикинуть, какие последствия будут у разных порядков выбора вершин. Если все ещё не понятно, то нарисуйте оттопсорченный ациклический граф и попробуйте в нем позапускать поиск в глубину в инвертированном графе, перебирая вершины слева направо, справа налево и т.п.

В общем, по-моему, этот алгоритм довольно легко можно запомнить.

Задание 17: *(нетривиальное) Не очевидно, что не работает такой алгоритм: оттопсортим граф и пойдём поиском в глубину в* ***не****инвертированном графе* справа налево*, т.е. от последних вершин к первым. Придумайте контрпример к этому алгоритму (конечно, придумать контрпример проще, чем доказать корректность :)).*

§4. Конденсация графа. Пусть у нас есть ациклический граф. Выделим в нем компоненты сильной связности и сожмём каждую в одну вершину. Т.е. рассмотрим новый граф: в нем каждая вершина отвечает компоненте сильносвязности начального графа, ребра, шедшие между вершинами из разных компонент, сохранены (если при этом получаются парные ребра, то оставляют только одно из них), а ребра, шедшие в пределах одной компоненты, не сохраняются. То, что получится, называется конденсацией этого графа.

Справа приведён пример ориентированного графа с выделенными компонентами сильной связности (то же, что и выше, в теме про компоненты сильной связности), и его конденсации. Поскольку компонент сильной связности три, то и вершин в конденсированном графе три, соответствующие этим самым компонентам.



Задание 18: *Докажите, что конденсация произвольного графа является ациклическим графом.*

Как построить конденсацию графа? Ну в принципе довольно очевидно. Ищем сильносвязные компоненты, а потом строим новый граф (т.е. заводим новую матрицу смежности или т.п.) и добавляем в него ребра, просто пробегаясь по рёбрам старого графа и те, которые идут в пределах одной компоненты, игнорируем, а те, которые идут из одной компоненты в другую, добавляем к получающемуся графу. Все просто.

В чем смысл конденсации? Ну например вот в чем. Пусть у нас есть набор объектов, про которые известно, что их можно в каком-то смысле упорядочить, и пусть про некоторые пары объектов дано утверждение вида «первый объект *меньше или равен* второго» (а про остальные пары ничего не известно). Ясно, что этому соответствует ориентированный граф. Несложно понять, что компоненты сильной связности такого графа — это множества объектов такие, которые *точно* должны быть равны: ведь каждый из них получается меньше или равен сам себя. Тогда логично эти объекты объединить в одну вершину, ведь они все равны между собой. Осталось определить отношения между такими «классами», т.е. про некоторые классы сказать, что «объекты одного класса меньше или равны объектов другого класса». Если пользоваться только теми отношениями, которые нам даны с самого начала, и не пытаться получить дополнительных следствий из них, то то что нам надо — это как раз и есть конденсация начального графа.

Замечу, что если бы с самого начала были даны условия вида «один объект *строго меньше* второго», то граф был бы обязан быть ациклическим. Ясно, что в этом случае задача конденсации большого смысле не

имеет (конденсация ациклического графа есть сам этот граф, т.к. каждая компонента сильной связности тут состоит из одной вершины), но может иметь смысл задача поиска какого-нибудь порядка объектов, не противоречащего этим условиям — а это делает топсорт.

Ещё замечу, что можно поставить задачу так: базируясь на данных нам условиях об объектах, перечислить *наибольшее* возможное количество пар объектов, про которые мы можем сделать аналогичное утверждение. Здесь придётся пользоваться транзитивностью отношения «меньше» (или «меньше или равно», в зависимости от того, какие условия нам даны): т.е. тем, что, если $a < b$, а $b < c$, то $a < c$. Т.е., если нам даны условия

$$a < b, \quad b < c, \quad b < d,$$

то мы должны сказать, что $a < b$, $a < c$, $a < d$, $b < c$, $b < d$, но про c и d ничего сказать не можем. Такая задача называется *транзитивным замыканием графа*, она не решается ни одним из рассмотренных выше алгоритмов, для неё есть отдельный алгоритм, про который мы поговорим как-нибудь в другой раз — к поиску в глубину он имеет мало отношения.

§5. Времена входа и выхода. Очень часто при обсуждении поиска в глубину вводят такие понятия, как времена входа и выхода. А именно, заведём глобальный счётчик «времени» t и каждый раз, когда мы первый раз начинаем обработку очередной вершины (входим в неё), и каждый раз, когда заканчиваем обработку вершины (выходим из неё) будем увеличивать t на единицу, а полученное значение записывать как «время входа» или «время выхода» в/из этой вершины, соответственно. Т.е. получим следующий алгоритм:

```
procedure find(i:integer);
begin
if was[i]<>0 then
   exit;
was[i]:=1;
inc(t);
s[i]:=t;
for i:=1 to n do
    if gr[i,j]=1 then
       find(j);
inc(t);
f[i]:=t;
end;
```

```
...
fillchar(was,sizeof(was),0);
t:=0;
for i:=1 to n do
    find(i);
```

Здесь s — массив времён входа (т.е. $s[i]$ — время входа в i-ю вершину), а f — аналогичный массив времён выхода. Перед запуском поиска в глубину занулим переменную t.

Обратите ещё раз внимание, что при работе этого алгоритма каждый элемент массива s будет установлен *ровно* один раз (т.е. каждый элемент будет установлен: не будет вершин, у которых время входа останется неопределённым — и никакой элемент не будет переписываться: не будет такого, что мы сначала в $s[i]$ запишем одно значение, а потом туда же другое), поскольку каждую вершину мы обрабатываем ровно один раз. Аналогично с массивом f.

Ещё обратите внимание, что массив *was* теперь не нужен, вместо него можно использовать массив s (но не f!), только, конечно, тогда s нужно будет предварительно занулить. Или, что то же самое, времена входа можно хранить в массиве *was*, а не в s (как раньше мы номер компоненты связности хранили в *was*). Здесь я использую s только для наглядности.

Но в реальных задачах времена входа и выхода, как правило, не бывают нужны. Они часто помогают в теоретических рассуждениях, но реально их вычислять обычно не нужно; более того, как мне кажется, нередко понять алгоритм проще, если не обращаться к понятию времён входа/входа. Например, часто при описании алгоритмов употребляется фраза «отсортируем вершины по временам выхода» (реже по временах входа), но это не значит, что нужно вычислять эти времена, а потом писать QSort или что-нибудь подобное. Все делается намного проще.

Задание 19: *Как надо правильно и проще всего сортировать вершины по временам выхода? Аналогично по временам входа.*

Обязательно прочитайте ответ к этой задаче! (Конечно, только после того, как порешаете сами).

Часть IV. Мосты и точки сочленения

В этой теме рассматриваем опять только неориентированные графы.

Мостом в неориентированном графе называется ребро, при удалении которого количество компонент связности графа увеличивается.

Точкой сочленения в неориентированном графе называется вершина, при удалении которой количество компонент связности графа увеличивается.

Задание 20: *Постоянно, когда формулирую определения, хочется сказать «. . . количество компонент связности увеличивается на одну». Можно ли так говорить, т.е. будут ли получающиеся определения эквивалентны тем, что даны выше?*

Задание 21: *Есть ли какая-нибудь простая связь между мостами и точками сочленения? Т.е. верно ли, что а) каждый конец моста — это точка сочленения? б) каждая точка сочленения является концом некоторого моста? в) может быть, ещё что-нибудь придумаете?*

§1. Перекрёстные ребра. Одним из важнейших свойств поиска в глубину является следующее. В дереве поиска в глубину (как, конечно, и вообще в любом дереве с корнем) для любой вершины можно определить её предков — т.е. те вершины, которые лежат по дороге от корня дерева к этой вершине, — а также можно определить потомков данной вершины — т.е. те вершины, на пути от которых до корня лежит наша. Т.е. вершина u является потомком v тогда и только тогда, когда v является предком u.

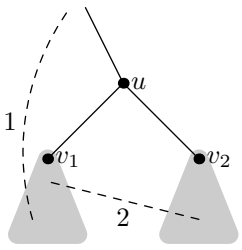
Тогда оказывается, что дерево поиска в глубину так устроено, что *любое ребро графа* соединяет две вершины, одна из которых является предком другой в этом дереве (а, соответственно, вторая является потомком первой). Если представить себе дерево подвешенным за корень, то все ребра идут сверху вниз (или, что то же самое, снизу вверх — граф-то неориентированный), возможно, пропуская несколько вершин (в смысле, что не обязательно соединяют вершину с её непосредственным потомком, а, возможно, с более дальним). Говоря по-другому, в графе отсутствуют *перекрёстные ребра*, т.е. ребра, которые идут «поперёк» дерева, из одного поддерева в совсем другое: каждое ребро идёт из вершины в поддерево этой вершины (или наоборот). Об этом свойстве чаще всего говорят как о свойстве отсутствия перекрёстных рёбер в дереве поиска в глубину.

Может быть, это не очень понятно сформулировано — порисуйте графы и попробуйте понять. Это на первый взгляд очень удивительное свойство, и по началу вообще не верится, что такое дерево вообще может существовать, но порисуйте графы и вы убедитесь в этом. (Вообще, как надо рисовать тут графы? Ясно, что надо рисовать не какой попало граф, а с определённой целью, пытаясь подобрать контрпример — тогда, может быть, намного чётче станет ясно, почему так происходит и, может быть, вы даже сумеете сами это доказать).

Задание 22: *Докажите это свойство.*

Этот факт нередко рассказывают сразу, как только начинают рассказывать поиск в глубину, но, как мы видели, он, как правило, не нужен нигде, кроме текущего раздела — мостов и точек сочленения.

§2. Поиск точек сочленения. Основная идея поиска точек сочленения состоит в том, что максимум, на что может распасться граф при удалении вершины — это поддеревья (дерева поиска в глубину) ниже этой вершине и весь оставшийся граф. Соответственно, вершина *не* является точкой сочленения тогда и только тогда, когда её можно «обойти» из любого поддерева, т.е. если из какой-нибудь вершины ниже этой есть ребро куда-то в оставшуюся часть дерева. (Точнее, если у нашей вершины несколько потомков в дереве поиска в глубину, то ребро должно быть из каждого поддерева с корнем в этих потомках.)



На рис. справа приведён пример ко всему, о чем говорилось выше. Здесь некоторая вершина u, её сыновья в дереве поиска в глубину v_1 и v_2, серым условно показаны поддеревья с корнями v_1 и v_2 (т.е. множества всех потомков v_1 и v_2). Дерево показано подвешенным, т.е. сверху где-то (не показан) — корень, а все рёбра дерева идут вниз. Свойство отсутствия перекрёстных рёбер утверждает, что в графе не могут существовать ребра типа ребра 2, т.е. идущие из одного поддерева в другое. А вот ребро типа 1 может существовать (при условии, что оно идёт в вершину, являющуюся предком u). Именно оно и будет тем «обходным» ребром, которое позволит поддереву с вершиной в v_1 не отделиться, когда удалим вершину u. Т.е. чтобы проверить, не является ли вершина u точкой сочленения, надо проверить, есть ли ребра типа 1 для всех её сыновей, т.е. есть ли ребра, идущие из всех поддеревьев её сыновей, вверх выше u.

А проверить это довольно просто. Ведь такие ребра не могут идти никуда, кроме как в предков вершины u. А они (предки) образуют прямой путь от корня дерева поиска в глубину до вершины u и упорядочены в этом дереве по порядку. Если для каждой вершины ввести её глубину как расстояние (в дереве поиска в глубину) от корня до этой вершины, то все вершины в пути от корня до u получатся идущими в порядке увеличения глубины. Тогда, если подумать, то понятно, что, для проверки наличия обходного ребра достаточно для каждой вершины уметь находить *вершину с наименьшей глубиной, в которую идут ребра из поддерева с корнем в нашей вершине*. Т.е., например, для v_1 найдём вершину с наименьшей глубиной (т.е. самую близкую к корню), в которую идут ребра или из самой v_1, или из её потомков — т.е. из всего соответствующего серого треугольника. Таким ребром будет или ребро 1, или ребра, которые идут ещё выше — короче говоря, если найденная таким образом вершина будет лежать выше, чем u, то это будет обозначать наличие обходного ребра, иначе такого обходного ребра нет. Таким образом мы уже получили алгоритм поиска точек сочленения, но все-таки код алгоритма я приведу в два этапа.

Итак, сначала научимся находить такие вершины. Точнее, для каждой вершины мы научимся находить собственно *глубину* самой неглубокой вершины, в которую идут ребра из нашей вершины или из её потомков. Находить это легко — эта глубина равна минимуму между глубинами всех вершин, куда идут ребра конкретно из нашей вершины и между ответами на такую же задачу для всех сыновей. Т.е., если мы

уже вычислили такую минимальную глубину для всех сыновей, то взяв минимум этих значений и учтя все рёбра, который напрямую из текущей вершину куда-то идут, получим ответ — такую минимальную глубину — для нашей вершины. Легко реализовать это так, чтобы $find$ стала функцией, которая будет возвращать этот самый ответ. Кроме того, для удобства будем в $find$ передавать глубину обрабатываемой вершины (чтобы не вычислять её каждый раз заново) и будет эту самую глубину сохранять прямо в массиве was. При этом потребуется глубину корня считать единичной (а в других случаях можно было бы считать её и нулевой).

```
function find(i,d:integer):integer;
var j,min,t:integer;
begin
was[i]:=d;
min:=n+1;
for j:=1 to n do
    if (gr[i,j]<>0) then begin
       if was[j]=0 then begin
          t:=find(j,d+1);
          if t<min then
             t:=min;
       end else begin
           if was[j]<min then
              min:=was[j];
       end;
    end;
find:=min;
end;
```

Обратите внимание, что я опять перенёс проверку $was[j] = 0$ внутрь цикла. Параметр d — глубина текущей вершины, поэтому вызываем этот алгоритм, конечно, командой `if was[...]=0 then find(...,1)`.

Контрольный вопрос 23: *Что должно быть на месте многоточия?*

Что за муть тут понаписана? Все дополнения к стандартному поиску в глубину здесь делятся на две части. Во-первых, возня с d. Параметр d при первом запуске (т.е. для корня) равен единице, а для каждого сына увеличивается на один (за счёт того, что написано $find(j, d + 1)$, таком образом он действительно отражает глубину вершину, поэтому по окончании поиска в глубину в was действительно лежат глубины вершин.

Во-вторых, возня с min и возвращаемым значением функции $find$. Функция должна вернуть ответ на нашу задачу, т.е. наименьшую из глубин вершин, в которые ведут ребра из нашей вершины или из её потомков. Вычисляем его мы следующим образом. Изначально присваиваем ему бесконечно большое значение, которое в данном случае равно $n + 1$ (n — количество вершин в графе; очевидно, что никакая глубина не может превосходить n). Далее перебираем всех соседей текущей вершины. Могут быть два варианта. Если $was[j] = 0$, то в этой вершине мы ещё не были. Поэтому идём в неё, вызывая рекурсивно $find$, и, более того, j становится сыном i, поэтому в поиске минимума надо учесть все ребра, выходящие из его поддерева. Но это делается легко: как раз вызов $find(j, \dots)$ и вычислит минимум по всем рёбрам, выходящим из поддерева, осталось нам сравнить его с min и при необходимости наш текущий минимум подкорректировать. Если же $was[j] \neq 0$, то в вершине j мы уже были и потому осталось лишь узнать её глубину и сравнить с текущим минимумом. Но её глубину мы уже знаем — она уже лежит в $was[j]$, поэтому просто сравниваем. (Обратите внимание, что в этом случае нам нужна именно глубина самой j, а не ответ на задачу для j, т.к. j не становится сыном i.)

Замечу, что это — один из сравнительно простых примеров динамического программирования на дереве. Ещё замечу, что именно из-за того, что все равно надо рассматривать эти два случая, я и перенёс проверку $was[j] = 0$ из начала процедуры сюда.

Итак, раз теперь понятно, как работает эта функция, осталось научиться определять, является ли текущая вершина точкой сочленения. Это уже просто. При удалении текущей вершины число компонент связности может увеличиться, только если какой-то сын отвалится вместе со своим поддеревом. Но про каждого сына мы знаем, отвалится ли он: если ответ на нашу задачу (т.е. минимальная глубина вершин, куда идут ребра из этого сына или его потомков) меньше, чем глубина d текущей вершины, то не отвалится, иначе отвалится. Т.е. осталось лишь добавить проверку $t \geqslant d$:

```
function find(i,d:integer):integer;
var j,min,t:integer;
    ok:boolean;
begin
was[i]:=d;
min:=n+1;
ok:=false;
for j:=1 to n do
    if (gr[i,j]<>0) then begin
       if was[j]=0 then begin
          t:=find(j,d+1);
          if t<min then
             t:=min;
          if t>=d then
             ok:=true;
       end else begin
           if was[j]<min then
              min:=was[j];
       end;
    end;
find:=min;
if ok then
   вершина i --- точка сочленения
end;
```

Т.е. если отвалится хотя бы один сын, то вершина i — точка сочленения. Обратите внимание, что проверка выглядит именно как $t \geqslant d$, а не $t > d$: если ребро идёт прямо в i-ую вершину, то все равно поддерево отвалится.

Отмечу только, что особой обработки требует корень дерева поиска в глубину. Можете подумать, почему вышеприведённый алгоритм тут не годиться, а я скажу, что корень является точкой сочленения тогда и только тогда, когда он имеет более одного сына. Окончательную версию алгоритма я приводить не буду, замечу только, что узнать в процедуре $find$, является ли текущая вершина корнем, можно легко: у корня и только у него $d = 1$.

§3. Поиск мостов. Можно пытаться искать мосты аналогично. Очевидно, что все мосты войдут в дерево поиска в глубину. Более того, ребро дерева будет мостом тогда и только тогда, когда нет обходных вокруг него рёбер, т.е. когда для нижнего конца этого ребра функция $find$ возвращает значение не меньше, чем глубина этого нижнего конца.

Задание 24: *Додумайте этот алгоритм и напишите его.*

Но для поиска мостов есть намного более простой алгоритм. Запустим поиск в глубину в нашем графе и ориентируем все ребра в том направлении, в котором мы их в первый раз просматривали, т.е. все ребра дерева — вниз, от корня, а все остальные ребра — вверх, к корню. Тогда несложно понять (и сложнее доказать :)), что достаточно в полученном графе найти сильносвязные компоненты: мостами в исходном графе будут как раз те рёбра, которые идут из одной сильносвязной компоненты в другую. (Нарисуйте пример и проверьте!) Поиск сильносвязных компонент делается, как мы уже знаем, двумя поисками в глубину, но мы уже один сделали, поэтому можно воспользоваться его результатами для запуска второго поиска. Таким образом, за два поиска в глубину ищутся мосты.

Задание 25: *Додумайте этот алгоритм и напишите его.*

Часть V. Задачи на поиск в глубину

Конечно, я предполагаю, что вы напишите все обсуждавшиеся алгоритмы, решите все вышеприведённые задачи и напишите и программы по этим задачам.

Задание 26: *Дан граф. Вывести несколько путей так, чтобы по каждому ребру проходил ровно один путь и при этом число путей было минимальным. В каждом пути вершины могут повторяться, но ребра нет. (В частности, если в графе существует эйлеров цикл или путь, то надо просто его вывести).*

Задание 27: *Дан граф, в котором точно существует эйлеров цикл (т.е. степени всех вершин чётны). Для каждого ребра i дано число x_i. Для каждого эйлерова цикла в этом графе введём его «штраф» следующим образом. Штраф за одно ребро i будет равен $k - x_i$, где k — порядковый номер этого ребра в цикле. Штраф цикла равен сумме штрафов за все ребра. Требуется найти эйлеров цикл с минимальным штрафом.*

Задание 28: *Дан неориентированный граф. Ориентировать его (т.е. каждое ребро ориентировать в одно из двух возможных направлений) так, чтобы он стал ациклическим. Если решения не существует, то сообщить это.*

Задание 29: *Дан неориентированный граф. Ориентировать его так, чтобы он стал сильносвязным. Если решения не существует, то сообщить это.*

Задание 30: *Дан частично ориентированный граф (т.е. некоторые ребра изначально ориентированные, некоторые нет). Ориентировать неориентированные ребра так, чтобы он стал ациклическим. Если решения не существует, то сообщить это.*

Задание 31: *(Задача 172 с acm.sgu.ru) В некоторой школе проводятся «экзамены по выбору». Каждый ученик должен выбрать два экзамена из данного списка и сдать их. Известно, какой ученик какие экзамены выбрал. Школа хочет выделить ровно два дня под эти экзамены, при этом в первый день провести часть экзаменов, а остальные экзамены — и только их — во второй. Конечно, при этом должно получиться так, чтобы каждый школьник мог сдать выбранные им экзамены в разные дни. Требуется составить такое расписание экзаменов (т.е. распределить экзамены по дням) или сообщить, что это невозможно.*

Часть VI. Подсказки по заданиям

Естественно, сначала порешайте задачи сами, а потом только читайте подсказки :) Подсказки даю не по всем заданиям.

Посказка 7: Подводный камень, с которым вы столкнётесь — это то, что из каждой вершины вы будете пытаться пойти в вершину-родителя текущей вершины и программа будет считать, что она нашла цикл, хотя на самом деле это — не цикл. Самый простой способ, который я знаю, чтобы избежать этой проблемы — это передавать в процедуру find дополнительный параметр — вершину-родителя текущей вершины — и перед рекурсивным вызовом проверять, не является ли та вершина, куда мы пытаемся пойти, родителем текущей.

Посказка 8: Могут существовать несвязные графы, в которых эйлеров цикл все-таки существует.

Посказка 12: Идея номер один состоит в следующем: будем в отдельном массиве для каждой вершины хранить её текущую исходящую степень. Тогда при удалении вершины будет достаточно пробежаться по всем входящим в неё рёбрам и уменьшить на единицу исходящую степень соответствующих вершин; сами исходящие ребра можно не удалять и даже не хранить (храним только входящие). Сток искать тогда можно просто пробегаясь по этому массиву и ища в нем нули. Идея два состоит в том, чтобы, в добавок ко всему вышесказанному, хранить все стоки в отдельном массиве (по принципу стека, например). Когда мы уменьшаем исходящую степень очередной вершины, то посмотрим: если степень стала нулевой, то вершина стала стоком и мы её заносим в этот массив. Теперь не надо на каждом шагу пробегаться по всему массиву степеней в поисках нулей — у нас есть отдельный массив, в котором хранятся вершины с нулевой исходящей степенью. Реализация и дополнительные комментарии в ответах (но сначала попробуйте сами написать!).

Посказка 15: Попробуйте понять, что получится, если алгоритм Кнута запустить на графе, в котором есть циклы?

Посказка 16: Рассмотрите следующий граф: три вершины, два ребра: из первой во вторую и из третьей во вторую.

Посказка 19: Подумайте, что же, собственно, делает алгоритм топологической сортировки в плане времён выхода?

Посказка 22: Пусть есть перекрёстное ребро. Рассмотрите тот его конец, в который мы при поиске в глубину попали раньше. При просмотре его соседей мы должны будем наткнуться на другой конец этого же ребра. Пойдём ли мы в него?

Посказка 26: Ясно, что число путей будет как минимум равно половине от числа вершин с нечётной степенью; пусть это число равно K, оно всегда чётно. Добавим в наш граф $K/2$ рёбер, попарно (произвольным образом) соединив наши нечётные вершины. Что дальше?

Посказка 27: На самом деле это почти что задача-шутка.

Часть VII. Ответы

Ответ 1: То, что не будет циклов, видимо, можно доказывать многими способами. Приведу идею самого простого из пришедших мне сейчас в голову доказательств. Пронумеруем все вершины в том порядке, в котором мы их находили. В дереве поиска в глубину из каждой вершины u выходим несколько (ноль или больше) рёбер в её «сыновья» — вершины, которые мы нашли из u, значит, их номера больше, чем у u, — а также ровно одно ребро в «предка» — вершину, *из* которой мы нашли u (такие ребра есть у всех вершин, кроме корня) — номер этой вершины меньше, чем у u. Пусть есть цикл. Рассмотрим в нем вершину с наибольшим номером. В неё входят *два* ребра, принадлежащие этому циклу, и потому идущие и вершин с *меньшими* номерами, чем у нашей. Противоречие.

В случае связного графа это всегда будет остовное дерево графа, т.е. покрывающее все вершины. Но в случае несвязного графа это будет или остовное дерево одной компоненты связности — если запускаем просто $find(i)$, — или остовный лес, покрывающий все вершины — если запускаем вторым из перечисленных в п. I.2 способом.

Ответ 2: Ясно, что, если в текущей компоненте связности некоторые элементы массива was изначально были ненулевыми, то в эти вершины мы не войдём. Скорее всего, это немного не то, что мы хотели (хотя, конечно, все зависит от задачи. Могут быть задачи, хотя я таких с ходу не припомню, в которых нам именно это и надо).

Ответ 3: Да, конечно, количество решений равно 2^k, где k — количество компонент связности графа. В пределах одной компоненты есть два способа раскраски, отличающиеся инвертацией всех вершин.

Вообще, есть элементарная неоднозначность: можно инвертировать все вершины сразу и получить новое решение — значит, решение *всегда* неоднозначно. Но даже если решения, отличающиеся инвертацией *всех* вершин, считать одинаковыми, то все равно в несвязных графах решение неоднозначно.

Ответ 4: Конечно, на вопрос «является ли данный граф двудольным».

Ответ 5: Ну что-нибудь в следующем стиле (конечно, поиск в ширину я реализую очередью)

```
var q:array[1..n] of integer;
    was:array[1..n] of integer;
    l,r:integer;
    cur:integer;
    j:integer;
...
fillchar(was,sizeof(was),0);
l:=1;r:=1;
q[1]:=1;
was[1]:=1;
while l<=r do begin
      cur:=q[l];
      inc(l);
      for j:=1 to n do
          if (gr[cur,j]<>0)and(was[j]=0) then begin
             was[j]:=3-was[cur];
             inc(r);
             q[r]:=j;
          end;
end;
```

Массив q — очередь, массив was — номера доли. Это работает для связного графа, в противном случае ещё нужен внешний цикл с проверкой was. Надеюсь, тут ошибок немного.

Ответ 6: Ну понятно, почему :) Для двудольности, покрасив одну вершину, мы тут же знаем, как красить соседние с ней, т.к. есть всего два варианта, а один из них уже занят. В трехдольности так не получится.

Ответ 7: Собственно, в подсказке я уже сказал, как надо все делать. Осталось привести пример программы.

```
procedure find(i,p:integer);
begin
if was[i]<>0 then begin
   не дерево!
   exit;
end;
was[i]:=1;
for i:=1 to n do
    if (gr[i,j]=1)and(j<>p) then
       find(j,i);
end;
```

Дополнительный параметр p здесь — номер вершины-предка. Вызываем эту процедуру из главной программы, конечно, передавая в качестве вершины-предка номер несуществующей вершины, например, ноль, если нумеруем вершины с единицы.

Да, ещё не забудьте, что для проверки, является ли граф деревом, надо запустить $find(1)$ и проверить, что вы побывали во всех вершинах, а для проверки, является ли граф лесом, надо пробежаться по всем вершинам и запускать $find$ оттуда, где ещё не бывали.

Ответ 8: Окончательный критерий — если в графе степени всех вершин чётны плюс все компоненты связности, кроме, может быть, одной, состоят из одной вершины (т.е. это связный граф и ещё несколько изолированных вершин).

Ответ 9: Ну что тут писать-то? Все сказано в абзаце перед задачей.

Ответ 10: Критерий такой: ровно одна вершина с исходящей степенью на единицу больше входящей, и ровно одна — со входящей, на единицу большей исходящей; у остальных эти степени должны быть равны.

Ну и обычные условия на связность. Пишется так же, как и эйлеров цикл в орграфе, только сначала надо найти ту самую вершину, где входящая на единицу больше исходящей (именно так, т.к. мы пойдём по инвертированным рёбрам!)

Ответ 11: Ясно, что могут быть ациклические графы — не деревья. Например, три вершины и ребра $1 \to 2$, $1 \to 3$ и $2 \to 3$. Если снимем ориентацию с рёбер, то могут появиться циклы, которых раньше не было из-за того, что ребра были ориентированы.

Ответ 12: Приведу код, только сначала несколько комментариев про хранение графа списком смежных вершин. Буду использовать настоящие списки, т.е.[6]

```
type tv=record
         v:integer;
         next:pv;
       end;
     pv=^tv;
var gr:array[1..maxN] of pv;
```

Здесь tv — очередной элемент списка, хранящий одно ребро (т.е. одну смежную вершину):. v — номер этой вершины, pv — указатель на следующее ребро (на следующий элемент типа tv), или nil, если такого ребра нет. gr хранит граф: для каждой вершины — указатель на первое ребро, *в*ходящее в эту вершины (или nil, если таких рёбер нет).

В данной задаче мы будем хранить только входящие ребра, т.к. исходящие нам не нужны (я говорил об этом в подсказке). В других случаях для ориентированного графа могут понадобиться два массива отдельно для входящих и исходящих рёбер; для неориентированного графа, конечно, нужен один массив.

Алгоритм:

```
...
var st:array[1..maxN] of integer;
    nst:integer;
    d:array[1..maxN] of integer;
    u,v:integer;
    n,m:integer;
    nv:pv;
    ans:array[1..maxN] of integer;
    pos:integer;

begin
...
fllchar(gr,sizeof(gr),0);
fillchat(d,sizeof(d),0);
read(f,n,m);
for i:=1 to m do begin
    read(f,u,v);
    new(nv);
    nv^.v:=u;
    nv^.next:=gr[v];
    gr[v]:=nv;
    inc(d[u]);
end;
nst:=0;
for i:=1 to n do
    if d[i]=0 then begin
       inc(nst);
       st[nst]:=i;
    end;
pos:=n;
for i:=1 to n do begin
    {должно быть nst>0}
    v:=st[nst];
    dec(nst);
    ans[pos]:=v;
    dec(pos);
    nv:=gr[v];
    while nv<>nil do begin
          dec(d[nv^.v]);
          if d[nv^.v]=0 then begin
             inc(nst);
             st[nst]:=nv^.v;
          end;
          nv:=nv^.next;
    end;
end;
```

st — массив (стек) стоков; nst — количество элементов в нем (т.е. количество стоков в текущем графе). d — массив исходящих степеней (т.е. $d[i]$ — исходящая степень i-ой вершины). ans — массив-ответ, pos — позиция в этом массиве, куда мы должны будем поставить очередную вершину.

Сначала считываем граф. Я специально привожу этот текст, чтобы вы видели, как хранить граф списком смежных вершин. Считаем, что граф задан списком рёбер: т.е. во входном файле сначала количества вершин (n) и рёбер (m), а потом по два числа на строке, задающие две вершины — откуда и куда идёт ребро. Поэтому считываем сначала эти количества, а потом сами ребра. Каждое ребро $u \to v$ надо добавить в список рёбер, входящих в вершину v, т.е. в список $gr[v]$. Посмотрите, как это делается. Тут небольшая путаница с тем, что ребро идёт из вершины u, поэтому приходится писать $nv.v := u$, но это мелочи. Может быть, можно было придумать более хорошие имена полям и переменным. Обратите внимание, что, как всегда при вставке в список, мы вставляем в его начало, а не в конец. Заодно параллельно считаем в массиве d исходящие степени.

После этого формируем начальный список стоков st, пробегаясь по массиву d и ища там нули.

Далее основная часть. Мы должны n раз подряд взять сток, поставить его в выходной массив и удалить его из графа. Каждый раз сток точно найдётся, т.к. граф ациклический, поэтому все время должно быть $nst > 0$. Берём очередной сток (конечно, последний из массива st — его проще удалять, чем первый), удаляем его из массива st (командой $dec(nst)$ просто), ставим в выходной массив и пробегаемся по входящим рёбрам, обратите внимание как. Для каждого ребра просто уменьшаем на единицу исходящую степень соответствующей вершины и, если она стала стоком, заносим её в массив st. Частая ошибка здесь — забыть написать `nv:=nv^.next`, чтобы перейти к следующему ребру. Это вам не for, который переменную цикла автоматически увеличивает.

Ответ 13: Например, граф с двумя вершинами и одним ребром $2 \to 1$ связен, но при запуске поиска в глубину $find(1)$ во вторую вершину мы не попадём.

Ответ 14: Три вершины, три ребра: $1 \to 2$, $1 \to 3$, $2 \to 3$: запустившись $find(1)$, мы два раза попробуем попасть в третью вершину.

[6] Замечу, что это очень синтаксически странная конструкция: я использую идентификатор pv до того, как объяснил, что он значит. Паскаль такое допускает при выполнении двух условий: во-первых, все должно быть в одной «секции» `type`, во-вторых, должен быть определённый порядок: то ли сначала определён tv, потом pv, то ли наоборот, я сейчас точно не помню. Если этот код не компилится, поменяйте их местами.

Ответ 15: Напрямую заменить нельзя, т.к. алгоритм Кнута не всегда даст какую-то последовательность вершин. Но наоборот: он даст какую-то последовательность вершин тогда и только тогда, когда граф ациклический. Т.е. приспособить алгоритм Кнута можно следующим образом: запускаем его и, если он нормально завершается, то граф ациклический, иначе нет. А что значит нормально завершается? Единственное, что ему может помешать — может оказаться, что в очередной момент $nst = 0$, т.е. в текущем графе нет стоков. Несложно понять, что это будет тогда и только тогда, когда граф не ациклический. Таким образом, может в алгоритм Кнута добавить одну проверку внутрь цикла и получить алгоритм проверки графа на ацикличность (а массив ans тогда, конечно, не нужен будет).

Ответ 16: Рассмотрим тот граф, который приведён в подсказке: три вершины, два ребра: $1 \to 2$ и $3 \to 2$. В соответствии с нашим определением «компонент слабой связности» вершины 1 и 2 должны лежать в одной компоненте, 2 и 3 тоже, а 1 и 3 нет (т.к. ни от 1 до 3, ни от 3 до 1 добраться нельзя). Поэтом такое определение бессмысленно в том смысле, что вершины не получается разбить на компоненты слабой связности. Ясно, что проблема именно в том, что нарушается требование транзитивности[7].

Ответ 17: Пример графа, для которого такой алгоритм не работает: три вершины, ребра $1 \to 3$, $3 \to 1$ и $1 \to 2$. Если запустим первый поиск в глубину из вершины 1, то результат «топсорта» будет именно порядок 1, 2, 3, и, пойдя в неинвертированном графе справа налево, запустившись первым же запуском $find(3)$, мы посетим все три вершины, что неправильно. Как «на пальцах» объяснить, чем таким этот алгоритм отличается от верного, я не знаю.

Ответ 18: Пусть в конденсации есть цикл. Но тогда возьмём две вершины этого цикла — пусть это вершины 1 и 2. В конденсации по этому циклу можно дойти и из 1 в 2, и из 2 в 1. Тогда возьмём в изначальном графе две вершины $1'$ и $2'$ из компонент сильной связности, соответствующих вершинам 1 и 2 конденсации. Несложно показать, что тогда и из $1'$ в $2'$ и в обратную сторону можно дойти в начальном графе, что противоречит тому, что они лежат в разных компонентах сильной связности.

Ответ 19: Несложно видеть, что топсорт как раз и сортирует вершины по времени выхода, только в порядке убывания. Он ведь на последнее место в выходном массиве ставит вершину, из которой мы вышли первой, и т.д. Поэтому сортировать вершины по времени выхода надо аналогично топсорту. По времени входа сортировать тоже аналогично, только ставить вершину в выходной массив надо в начале процедуры $find$. Это чем-то аналогично сортировке подсчётом.

Ответ 20: Для мостов можно. Для точек сочленения нет, т.к. может оказаться, что при удалении вершины старая компонента связности распадается сразу на три или ещё больше. Обратите также внимание, что нельзя говорить «мост — это такое ребро, при удалении которой граф распадётся на *две* компоненты связности» и аналогично для точек сочленения, т.к. не понятно, что это значит для несвязных графов, и в наиболее логичной трактовке для несвязных графов это неверно.

Ответ 21: Я такой связи не знаю. а) нет, т.к. конец моста может оказаться висящей вершиной. Вообще, например, в графе с двумя вершинами и одним ребром (1–2) один мост, но ни одной точки сочленения. б) Очевидно, нет, и несложно придумать контрпример.

Ответ 22: Пусть есть такое ребро. Рассмотрим тот конец его, в который мы попали раньше — пусть это вершина u. К моменту, когда мы попали в него, во второй конец (v) мы ещё на заходили. Мы будем просматривать соседей вершины u и наткнёмся на v. Если к этому моменту мы все ещё не были в v, то мы в неё пойдём, v станет сыном и потому потомком u и ребро не будет перекрёстным. В противном случае мы успели побывать в вершине v, пока обрабатывали других соседей вершины u, значит, v — потомок u и ребро все равно не перекрёстное.

Ответ 23: Ну понятно, переменная внешнего цикла, в котором мы запускаем поиск в глубину. См. п. I.2.

Ответ 24: Приводить алгоритм тут не буду, пишите сами :)

Ответ 25: Аналогично предыдущему. Мне кажется, что, если вы хорошо освоились с поиском в глубину, то придумать и написать *этот* алгоритм труда не должно составить.

Ответ 26: Если добавить $K/2$ рёбер, как это сказано в подсказке, то степени всех вершин станут чётными. Поэтому найдём в графе эйлеров цикл и потом из него выкинем те ребра, которые мы раньше добавили. Цикл распадётся на $K/2$ путей, которые и будут ответом на нашу задачу, т.к. меньше путей получить нельзя.

Бонусное задание: а почему именно на $K/2$? Вдруг у нас несколько удаляемых рёбер в цикле будут идти подряд?

Ответ 27: Задача-шутка. Несложно доказать, что веса всех эйлеровых циклов одинаковы и равны $n(n+1)/2 - S$, где S — сумма всех x_i, поэтому выводим любой цикл.

Ответ 28: Очень просто на самом деле, и даже поиск в глубину тут ни при чем. Просто ориентируем все ребра от вершины с меньшим номером к вершине с бóльшим.

[7] На самом деле, пусть про некоторые пары вершин (или вообще любых объектов), сказано, что эти пары «хорошие». Тогда, чтобы вершины можно было разбить на «компоненты хорошести», т.е. на группы такие, что в пределах каждой группы все пары хорошие, а между группами — нет, необходимо и достаточно выполнения трёх условия: рефлексивности (что каждая вершина сама с собой образует хорошую пару), симметричности (что если u и v хорошая пара, то и v и u тоже), и транзитивности (если u и v хорошая пара, и v и w тоже, то и u и w тоже).

Вообще, любой ациклический граф, как мы знаем, можно оттопсортить, и наоборот, любой граф, который можно оттопсортить (т.е. для которого задача топологической сортировки имеет решение), является ациклическим. Поэтому можно просто задать произвольный порядок вершин и ориентировать ребра в соответствии с этим порядком, и, более того, таким образом можно получить *любой* ациклический граф, который вообще можно получить из данного неориентированного.

Ответ 29: Решения не будет существовать тогда и только тогда, когда в графе есть мосты. Если их нет, то ориентировать надо также, как при поиске мостов, т.е. ребра дерева поиска — от корня, остальные ребра — к корню. (На самом деле даже искать мосты не обязательно).

Ответ 30: Выкинем все неориентированные ребра и оттопсортим полученный граф. Если это невозможно, то задача решения не имеет (почему?). Потом вернём все неориентированные ребра и ориентируем их в соответствии с полученным в топсорте порядком.

Бонусное задание: сравните эту задачу с задачей 28.

Ответ 31: Постройте граф: вершины — экзамены, ребра — ученики. Осталось проверить его на двудольность.