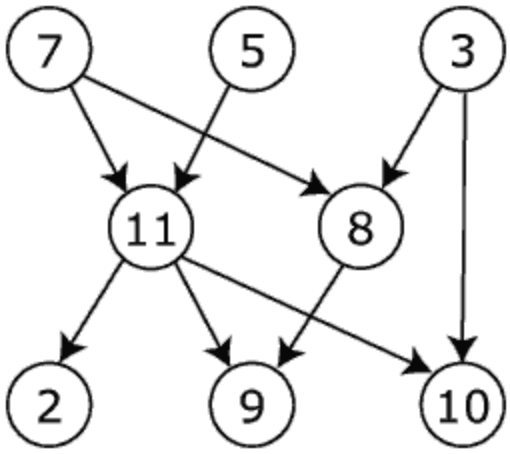
Топологическая сортировка

Топологическая сортировка

Топологическая сортировка — упорядочивание вершин ориентированного графа таким образом, чтобы любое ребро вело от вершины с меньшим номером к вершине с большим номером.



Определение:

Степень или **валентность вершины** графа — количество рёбер графа G, инцидентных вершине v

Определение:

Полустепенью исхода $d^+(v)$ вершины *v* называется число дуг, исходящих из *v*. Аналогично определяется *полустепень захода* $d^-(v)$ вершины *v.*

1) **Лемма:** Топологическая сортировка возможна только на безконтурных графах
   Доказательство: В графе с циклами не существует такого порядка вершин, который является топологической сортировкой
2) **Лемма:** Если в графе отсутствуют циклы, то в нем обязательно есть вершина с нулевой полустепенью захода
   Доказательство: По индукции. Пусть есть граф, 2 вершины и есть путь из v1 в v2. Нет других вариантов, кроме как сделать путь v2->v1 , но это цикл. Добавим 3-ю вершину....Данное доказательство верно для любого конечного количества вершин.

Части́чно упоря́доченное мно́жество (англ. *partially ordered set (также poset)*) — математическое понятие, которое формализует интуитивные идеи упорядочения, расположения элементов в определённой последовательности. Неформально, множество частично упорядочено, если указано, какие элементы *следуют* за какими

Упорядоченное множество (англ. *ordered set*) - множество с заданным отношением порядка.

3) **Лемма:** Порядок вершин в топологической сортировке будет соответствовать частичному порядку
   Доказательство: В графе может существовать более одной вершины с нулевой полустепенью захода. Так же может существовать несколько не связанных друг с другом подграфов
4) Следствие: Топологическая сортировка, в общем случае имеет более одного решения
5) **Лемма:** Если в графе отсутствуют циклы, то он может быть топологически отсортирован
   Доказательство: По индукции. Пусть есть орграф без циклов v0, v1… vn. Существует вершина с нулевой полустепень заходов.u, исключим ее из графа с дугами. Полученный подграф тоже не имеет циклов. Выполним это в рекурсии.

Алгоритм Кана

```
Кан():
    пока вершины не пусто
        v = вершина с количеством заходов = 0
        если v пусто то
            вернуть ложь
        вершины.удалить(v)
        дуги.дуги(v.исходы)
        очередь.добавить(v)
    вернуть истина
```

Какие структуры данных удачные, какие нет для этого алгоритма?
Какой недостаток этого алгоритма?
Вопросы?

Алгоритм Тарьяна

6) **Лемма:** Если в графе отсутствуют циклы, то в нем обязательно есть вершина с нулевой полустепенью исхода
   Доказательство: По индукции. Пусть есть граф, 2 вершины и есть путь из v1 в v2. Нет других вариантов, кроме как сделать путь v2->v1 , но это цикл. Добавим 3-ю вершину....Данное доказательство верно для любого конечного количества вершин.
7) **Лемма:** Если в графе отсутствуют циклы, то он может быть топологически отсортирован
   Доказательство: По индукции. Пусть есть орграф без циклов v0, v1… vn. Существует вершина с нулевой полустепенью исхода u, исключим ее из графа с дугами. Полученный подграф тоже не имеет циклов. Выполним это в рекурсии.

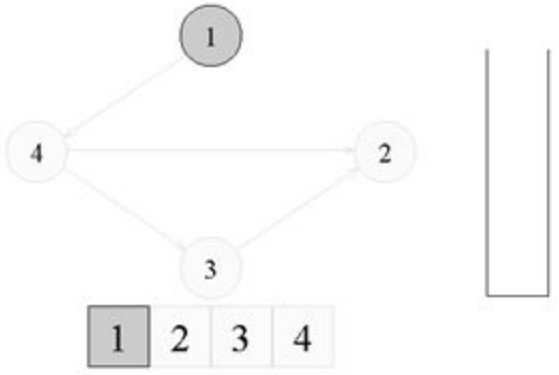
Алгоритм Тарьяна основан на поиске в глубину в графе
Использует 3-х цветную раскраску графа

- Белая вершина (еще не посещали)
- Серая вершина (начали обработку)
- Черная вершина (закончили обработку)

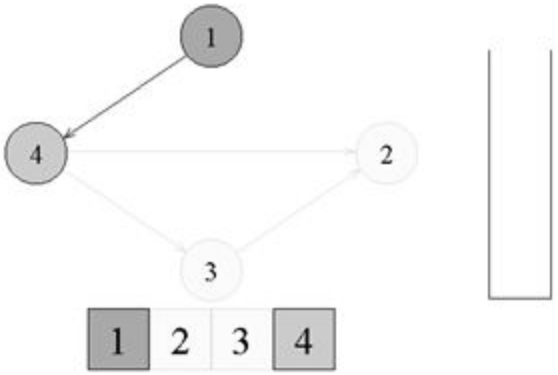
Понадобится стек, для отсортированных вершин

Рассмотрим данный алгоритм на примере:

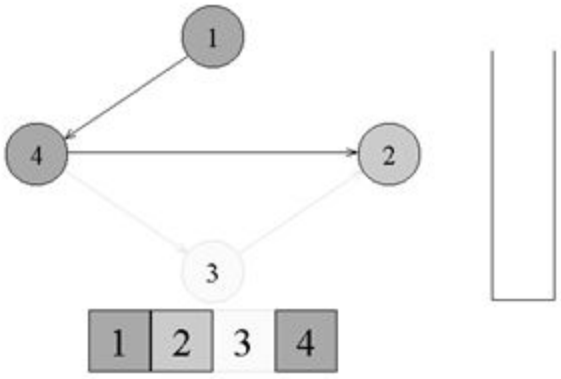
Имеем бесконтурный ориентированный граф. Изначально все вершины белые, а стек пуст. Начнем обход в глубину с вершины номер 1.



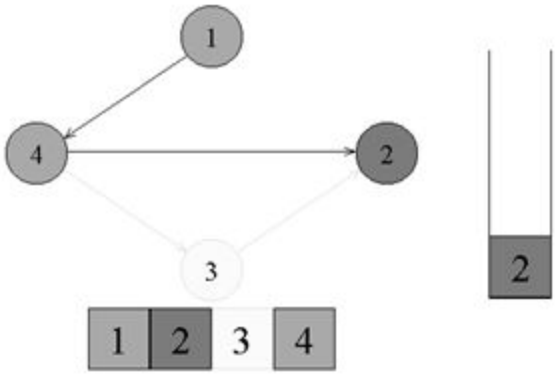
Переходим к вершине номер 1. Красим ее в серый цвет.



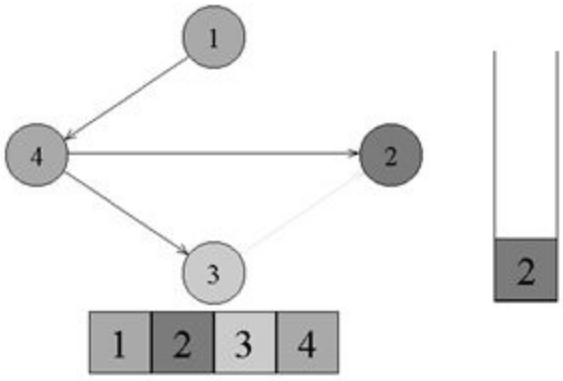
Существует ребро из вершины номер 1 в вершину номер 4. Переходим к вершине номер 4 и красим ее в серый цвет.



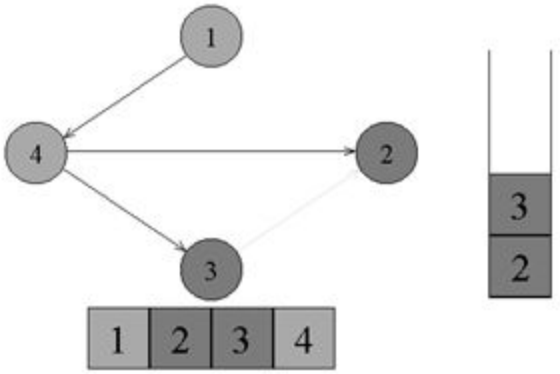
Существует ребро из вершины номер 4 в вершину номер 2. Переходим к вершине номер 2 и красим ее в серый цвет.



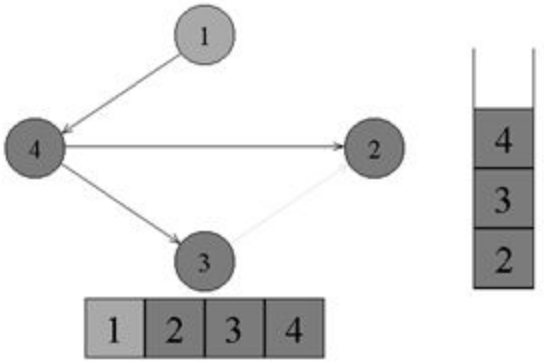
Из вершины номер 2 нет ребер, идущих не в черные вершины. Возвращаемся к вершине номер 4. Красим вершину номер 2 в черный цвет и кладем ее в стек.



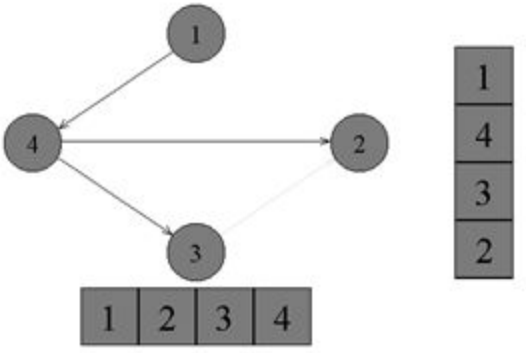
Существует ребро из вершины номер 4 в вершину номер 3. Переходим к вершине номер 3 и красим ее в серый цвет.



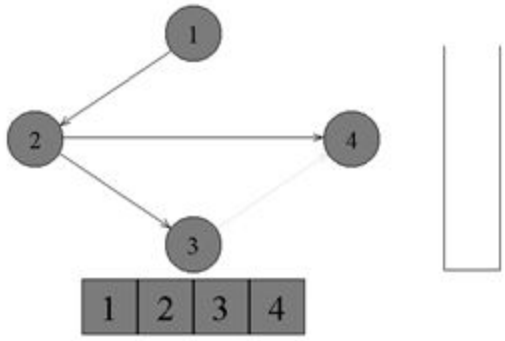
Из вершины номер 3 нет ребер, идущих не в черные вершины. Возвращаемся к вершине номер 4. Красим вершину номер 3 в черный цвет и кладем ее в стек.



Из вершины номер 4 нет ребер, идущих не в черные вершины. Возвращаемся к вершине номер 1. Красим вершину номер 4 в черный цвет и кладем ее в стек.



Из вершины номер 1 нет ребер, идущих не в черные вершины. Красим её в черный цвет и кладем в стек. Обход точек закончен.



По очереди достаем все вершины из стека и присваиваем им номера 1, 2, 3, 4 соответственно. Алгоритм топологической сортировки завершен. Граф отсортирован.

Что будет, если мы начнем с вершины с ненулевой полустепенью захода? Зачем 3-х цветная раскраска?

Тарьян():

```
    для v из V
        если цвет[v] == белый то
            если не DFS(v) то
                вернуть ложь
    вернуть истина
```

```
DSF(v)
цвет[v] = серый;
для u из v.V
    если цвет[u] == белый то
        если не DFS(u) то
            вернуть ложь
    иначе
        если цвет[u] == серый то
            вернуть ложь

цвет[v] = черный
стек.положить(v)
вернуть истина
```

Алгоритм поиска в глубину с 3-х цветной раскраской применяется для поиска циклов

Вопросы?

Алгоритм Демукрона

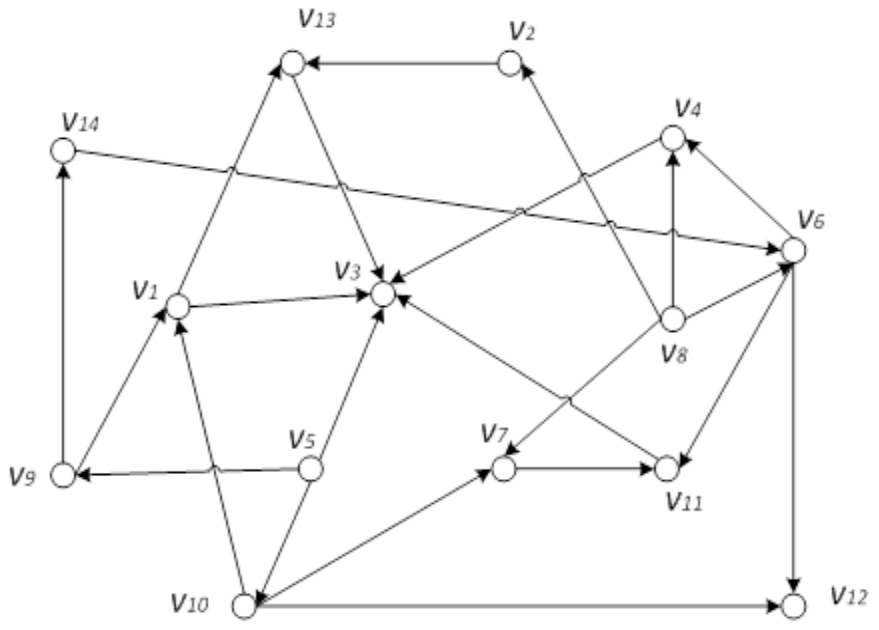
Определение:

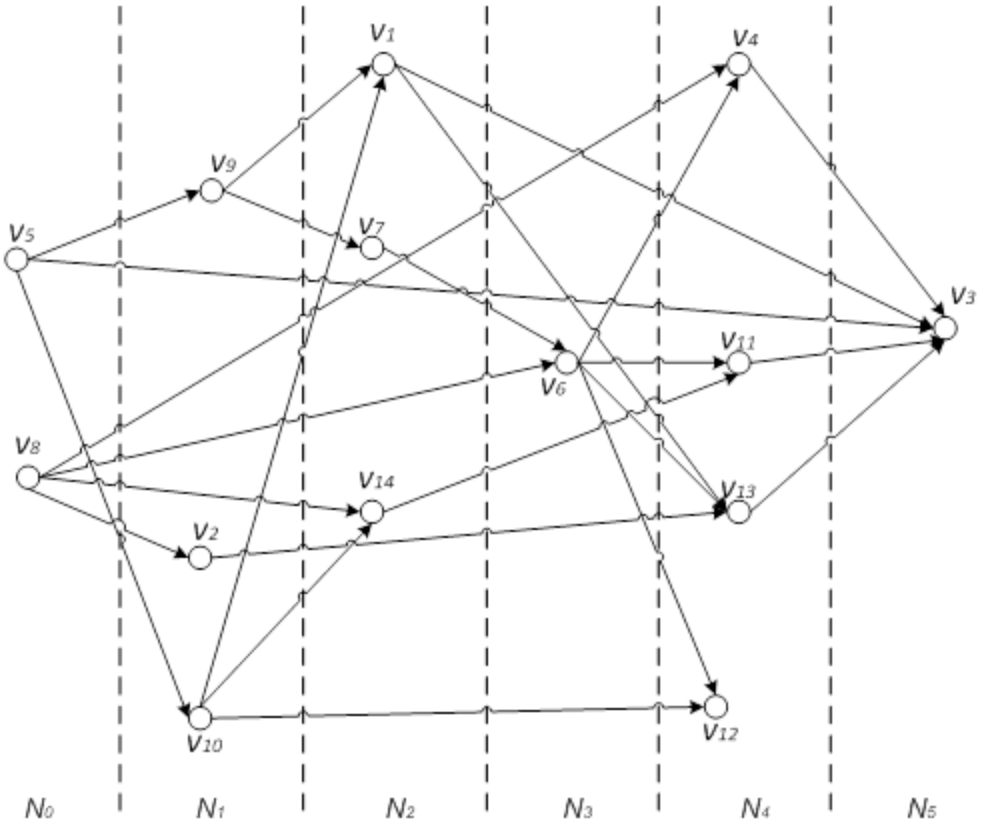
Уровень вершины графа - это натуральное число, определяемое следующим образом:

1) если полустепень захода вершины равна 0, то ее уровень равен 0 и наоборот (т.е. нулевой уровень N0 - это множество всех входов);

2) если множества Ni вершин уровня i определены для всех i < k, то уровень Nk+1 содержит те и только те вершины, предшественники которых принадлежат любому из уровней с номером от 0 до k, причем существует хотя бы один предшественник уровня k.

Уровень вершины графа можно интерпретировать как длину максимального пути от входов графа до этой вершины, посчитанную в количестве пройденных вершин





Общая идея аналогична алгоритму Кана. Нулевой уровень образуют вершины с полустепенью захода, равной 0. Удалив из сети все вершины нулевого уровня и исходящие из них дуги, вновь получим граф, входами которой будут вершины первого уровня исходного графа. Указанный процесс "послойного" удаления вершин следует продолжать до тех пор, пока все вершины исходного графа не будут распределены по уровням.

Алгоритм Демукрона использует матрицу смежности вершин и основан непосредственно на определении уровня вершины и свойствах матрицы смежности. Можно увидеть, что сумма элементов k-го столбца матрицы A равна полустепени захода вершины Vk. Поэтому, просуммировав элементы матрицы по всем столбцам и выбрав вершины, соответствующие столбцам с нулевой суммой, получим множество вершин нулевого уровня - входы сети.

- Запишем суммы элементов столбцов матрицы A в вектор M длины n. При этом элемент m[i] вектора M будет содержать полустепень захода вершины v[i].
- Удалим вершину v[i] и все исходящие из нее дуги.
- Чтобы пересчитать полустепени захода всех вершин сети, оставшихся в ней после удаления вершины v[i], надо из вектора M вычесть i-ю строку матрицы A.

- Обобщим, если на очередном шаге удаляем вершины v[i], … , v[k] , то для определения следующего „слоя" вершин нужно из вектора M вычесть строки матрицы A с номерами i, ... , k и зафиксировать **новые** нулевые элементы вектора M, появившиеся после вычитания.
- Фиксировать следует именно новые нулевые элементы, поскольку элементы вектора M, соответствующие вершинам, лежащим на предыдущих уровнях, стали равными 0 на предыдущих шагах алгоритма.
- Результат топологической сортировки - упорядоченные множества вершин, принадлежащие соответствующему множеству

Пример. Матрица смежности вершин сети имеет следующий вид:

	1	2	3	4	5	6	7	8	9	10	11	12	13	14
1			1										1	
2													1	
3														
4			1											
5			1						1	1				
6				1							1	1	1	
7											1			
8		1		1		1	1							
9	1													1
10	1						1					1		
11			1											
12														
13			1											
14														

Приведем последовательность значений массива M, соответствующую итерациям алгоритма и множества Ni вершин i-го уровня Прочерки соответствуют вершинам, не принадлежащим множеству V1 ("замаскированные" вершины) на соответствующем этапе алгоритма.

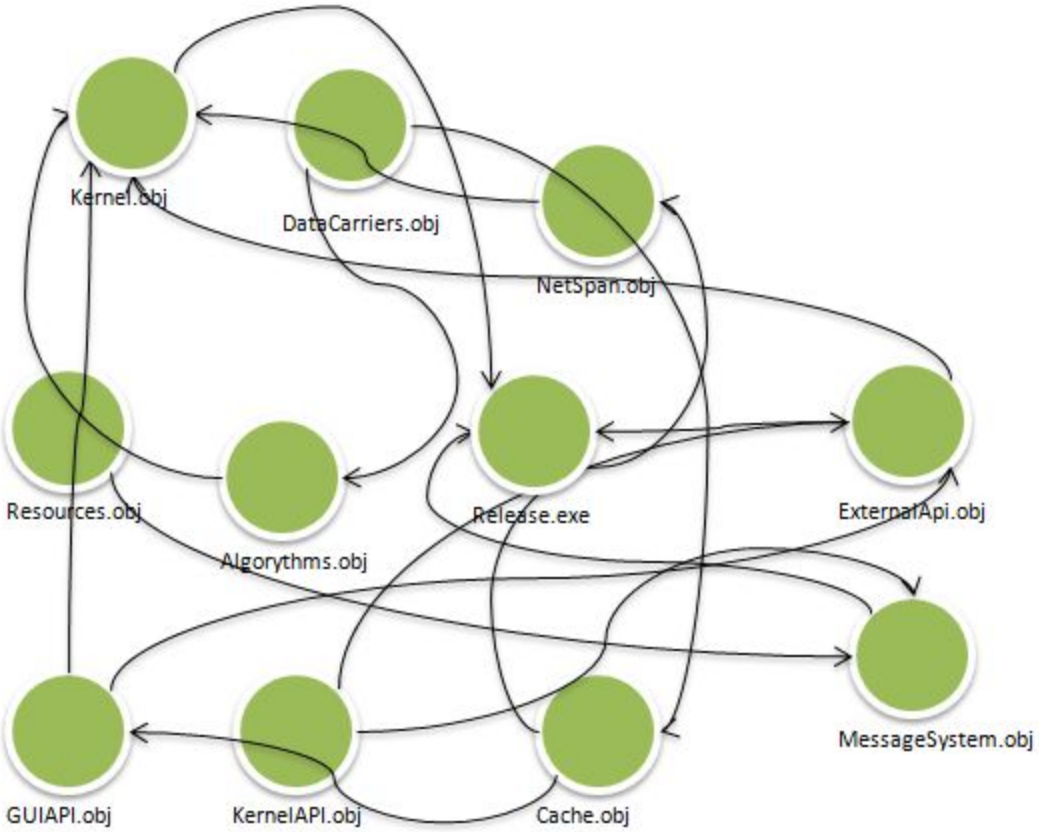
1	2	3	4	5	6	7	8	9	10	11	12	13	14	
2	1	5	2	0	2	2	0	1	1	2	2	3	1	N_0={5,8}
2	0	4	1	—	1	1	—	0	0	2	2	3	1	N_1={2,9,10}
0	—	4	1	—	1	0	—	—	—	2	1	2	0	N_2={1,7,14}
—	—	3	1	—	0	—	—	—	—	1	1	1	—	N_3={6}
—	—	3	0	—	—	—	—	—	—	0	0	0	—	N_4={4,11,12,13}
—	—	0	—	—	—	—	—	—	—	—	—	—	—	N_5={3}

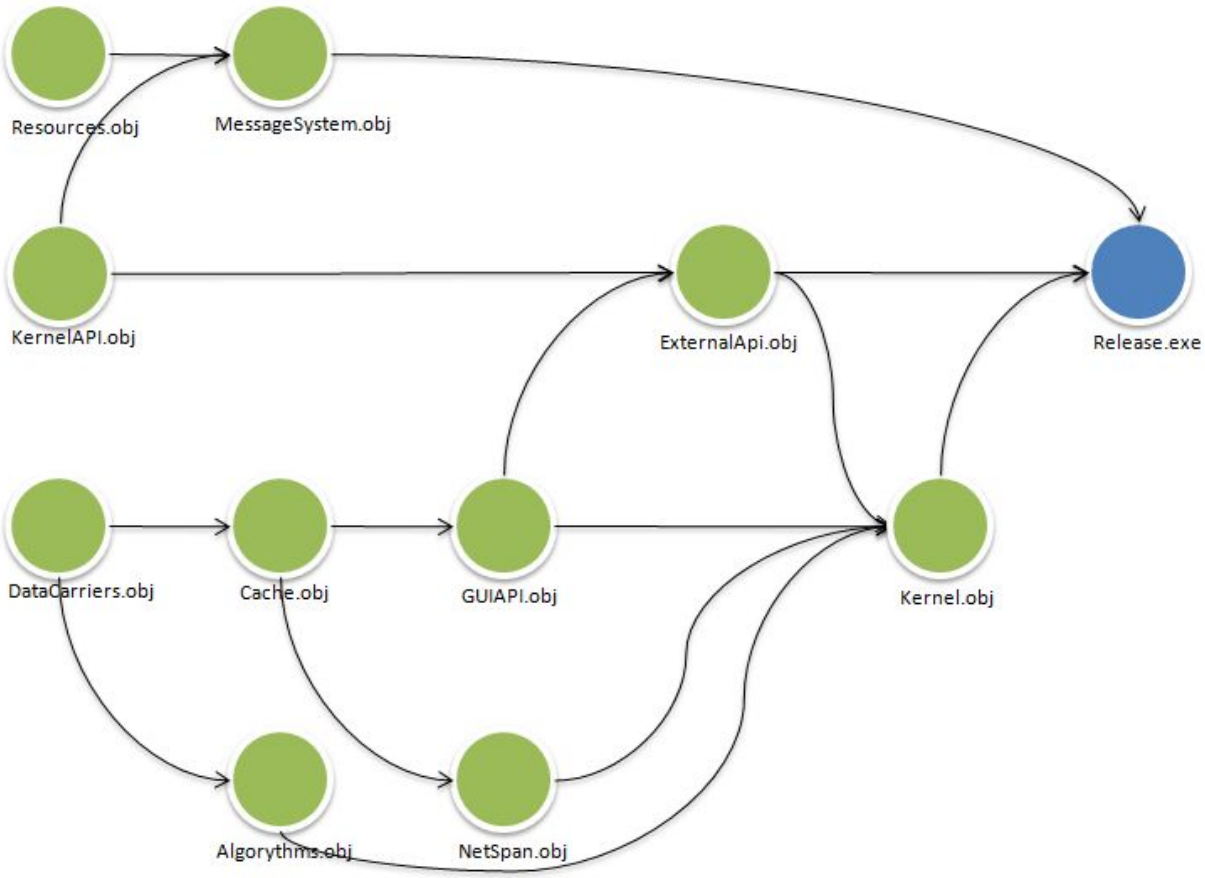
Алгоритм обрабатывает матрицу A смежности вершин графа порядка n. В результате работы алгоритма получаем массив Порядок длины n, i-й элемент которого равен номеру уровня вершины Vi.

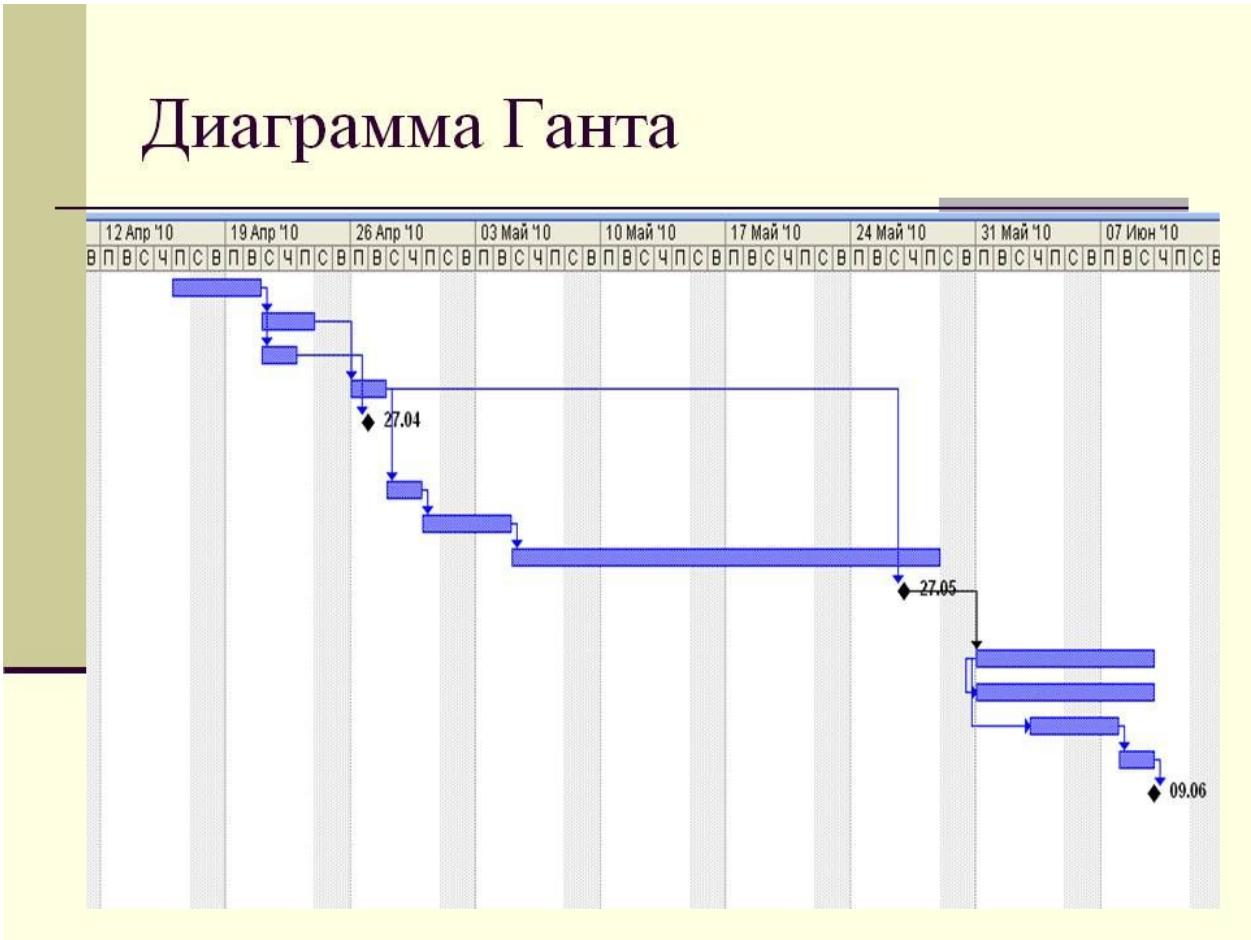
```
Демукрон()
    уровень = 0
    для i = 0 до количество вершин
        m[i] = СуммаСтолбца(A, i);
        v.добавить(i)

    пока v не пусто
        для u из v
            если m[u] == 0 то
              zero.добавить(u)
        если zero пусто
            вернуть ложь
        для u из zero
                Порядок[уровень].добавить(u)
                v.удалить(u)
                ПересчитатьM(u)
        уровень++


ПересчитатьM(u):
    для w из v
        m[w] -= A(u, w)
```

Что не учтено в алгоритме?
Вопросы?

Применение топологической сортировки


Kernel.obj
DataCarriers.obj
NetSpan.obj
Resources.obj
Algorythms.obj
Release.exe
ExternalApi.obj
MessageSystem.obj
GUIAPI.obj
KernelAPI.obj
Cache.obj


Resources.obj
MessageSystem.obj
KernelAPI.obj
ExternalApi.obj
Release.exe
DataCarriers.obj
Cache.obj
GUIAPI.obj
Kernel.obj
Algorythms.obj
NetSpan.obj

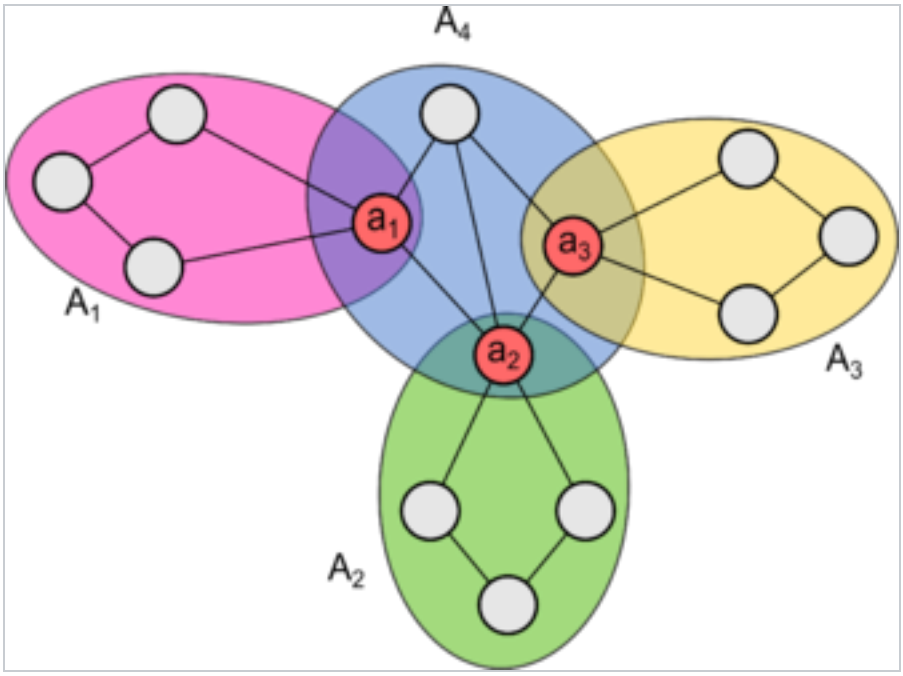


Компоненты связности

Точка сочленения

Определение:

Точка сочленения графа G — вершина, при удалении которой в G увеличивается число компонент связности.

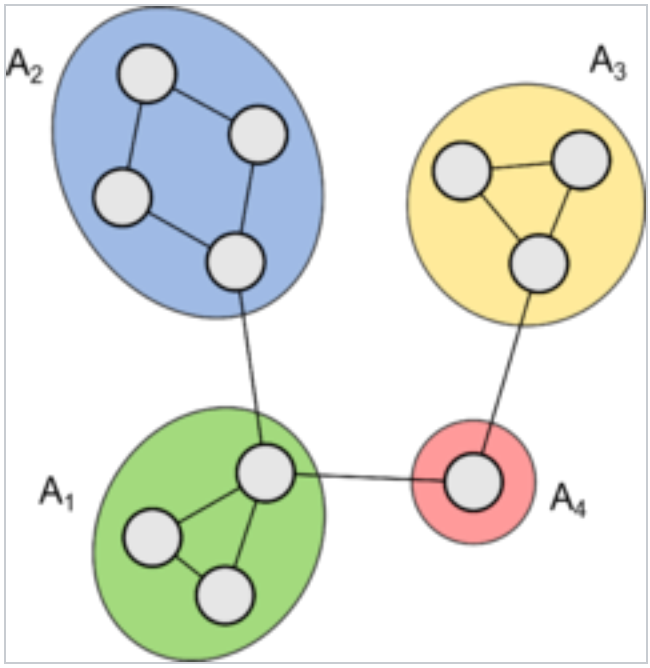


Вершины $a_1 a_1$, $a_2 a_2$, $a_3 a_3$ - точки сочленения графа G

Мост

Определение:

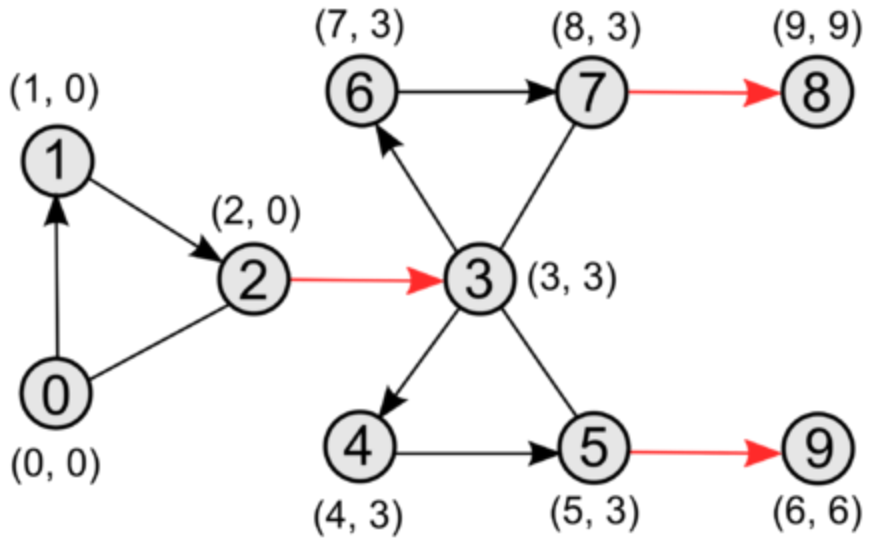
Мост графа G — ребро, при удалении которого в графе G увеличивается число компонент связности.



- Пусть мы находимся в обходе в глубину, просматривая сейчас все рёбра из вершины v. Тогда, если текущее ребро (v, to) таково, что из вершины to и из любого её потомка в дереве обхода в глубину нет обратного ребра в вершину v или какого-либо её предка, то это ребро является мостом. В противном случае оно мостом не является. (В самом деле, мы этим условием проверяем, нет ли другого пути из v в to, кроме как спуск по ребру (v, to) дерева обхода в глубину.)

Алгоритм поиска мостов

Рассмотрим вершину v или её потомка. Из нее есть обратное ребро в предка v тогда и только тогда, когда найдется такой сын u, что $ret[u] \leq enter[v]$. Если $ret[u] = enter[v]$, то найдется обратное ребро, приходящее точно в v. Если же $ret[u] > enter[v]$, то это означает наличие обратного ребра в какого-либо предка вершины v.



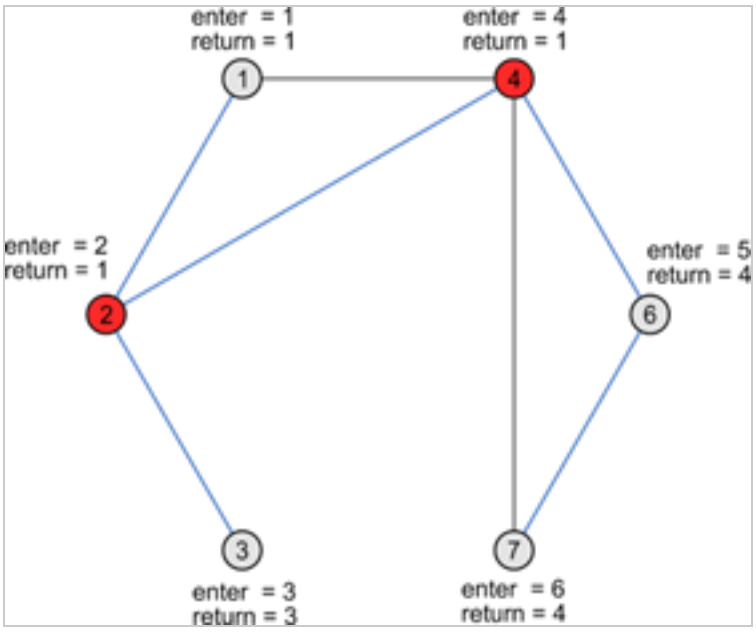
```
Мосты(G):
   для i = 1 до n
      если не посещали[i]
         DFS(i, -1)


DFS(v, p):
  время ++
  вход[v] = время
  вуход[v] = время
  для u из v
    если u != p то
          если посещали[u] то
               выход[v] = min(выход[v], вход[u])
          иначе
               DFS(u, v)
               выход[v] = min(выход[v], выход[u])
               если выход[u] > вход[v]
                    мосты.добавить(v, u)
```

Алгоритм поиска точек сочленения

Запустим обход в глубину из произвольной вершины графа; обозначим её через root. Заметим следующий факт:

- **Пусть мы находимся в обходе в глубину, просматривая сейчас все рёбра из вершины $v \neq \text{root}$. Тогда, если текущее ребро (v, to) таково, что из вершины to и из любого её потомка в дереве обхода в глубину нет обратного ребра в какого-либо предка вершины v, то вершина v является точкой сочленения. В противном случае, т.е. если обход в глубину просмотрел все рёбра из вершины v, и не нашёл удовлетворяющего вышеописанным условиям ребра, то вершина v не является точкой сочленения. (В самом деле, мы этим условием проверяем, нет ли другого пути из v в to)**

Теперь осталось научиться проверять этот факт для каждой вершины эффективно. Для этого воспользуемся "временами входа в вершину", вычисляемыми алгоритмом поиска в глубину.



Красным цветом обозначены точки сочленения

Синим — ребра по которым идет DFS

Пусть tin[u] — время входа поиска в глубину в вершину u. Через up[u] обозначим минимум из времени захода в саму вершину tin[u], времен захода в каждую из вершин p, являющуюся концом некоторого обратного ребра (u,p), а также из всех значений up[v] для каждой вершины v, являющейся непосредственным сыном u в дереве поиска.

Тогда из вершины u или её потомка есть обратное ребро в её предка ⇔ ∃ такой сын v, что up[v]□tin[u]

Таким образом, если для текущей вершины u≠root существует непосредственный сын v: up[v]□tin[u], то вершина u является точкой сочленения, в противном случае она точкой сочленения не является.

```
ТочкиСочленения(G):
   для i = 1 до n
      если не посещали[i]
         DFS(i, -1)

   DFS(v, p):
      время ++
      выход[v] = вход[v] = time
      посещали[v] = истина
      количество = 0
      для u из v
         если u != p то
               если посещали[u] то
                     выход[v] = min(выход[v], вход[u])
               иначе
                     DFS(u, v)
```

```
            количество++
            up[v] = min(выход[v], выход[u])
            если p != -1 and up[u] >= вход[v] то
                точкаСочленения.добавить(v)
    если p == -1 and количество >= 2 то
        точкаСочленения.добавить(v)
```

Вопросы?

Эйлеров цикл

Эйлеров путь (**эйлерова цепь**) в графе — это путь, проходящий по всем рёбрам графа и притом только по одному разу.

Эйлеров цикл — эйлеров путь, являющийся циклом. То есть замкнутый путь, проходящий через каждое ребро графа ровно по одному разу.

Эйлеров граф — граф, содержащий эйлеров цикл.

Полуэйлеров граф — граф, содержащий эйлеров путь

Степень вершины — количество рёбер графа *G*, *инцидентных* вершине *x*.

Для орграфов бывают **входящая полустепень** и **исходящая полустепень**

Существование эйлерова цикла и эйлерова пути

В неориентированном графе

Согласно теореме, доказанной Эйлером, эйлеров цикл существует тогда и только тогда, когда граф связный или будет являться связным, если удалить из него все изолированные вершины, и в нём отсутствуют вершины нечётной степени.

Эйлеров путь в графе существует тогда и только тогда, когда граф связный и содержит не более двух вершин нечётной степени. Ввиду леммы о рукопожатиях, число вершин с нечётной

степенью должно быть четным. А значит эйлеров путь существует только тогда, когда это число равно нулю или двум. Причём когда оно равно нулю, эйлеров путь вырождается в эйлеров цикл.

В ориентированном графе

Ориентированный граф G=(V,A) содержит эйлеров цикл тогда и только тогда, когда он сильно связан или среди его компонент сильной связности только одна содержит ребра (а все остальные являются изолированными вершинами) и для каждой вершины графа её входящая степень **indeg** равна её исходящей степени **outdeg**, то есть в вершину входит столько же ребер, сколько из неё и выходит: **indeg(v)=outdeg(v)**

Так эйлеров цикл является частным случаем эйлерова пути, то очевидно, что ориентированный граф G=(V,A) содержит эйлеров путь тогда и только тогда, когда он содержит либо эйлеров цикл, либо эйлеров путь, не являющейся циклом. Ориентированный граф G=(V,A) содержит эйлеров путь, не являющейся циклом, тогда и только тогда, когда существуют две вершины **p in V** и **q in V** (начальная и конечная вершины пути соответственно) такие, что их полустепени захода и полустепени исхода связаны равенствами
indeg(q) = outdeg (q)+1 и**indeg(q) = outdeg (q)-1**, а все остальные вершины имеют одинаковые полустепени исхода и захода: **indeg(v)=outdeg(v)**

Поиск эйлерова пути в графе

Можно всегда свести задачу поиска эйлерова пути к задаче поиска эйлерова цикла. Действительно, предположим, что эйлерова цикла не существует, а эйлеров путь существует. Тогда в графе будет ровно 2 вершины нечётной степени. Соединим эти вершины ребром, и получим граф, в котором все вершины чётной степени, и эйлеров цикл в нём существует. Найдём в этом графе эйлеров цикл (алгоритмом, описанным ниже), а затем удалим из ответа несуществующее ребро.

Поиск эйлерова цикла в графе

Алгоритм Флёри

Алгоритм был предложен Флёри в 1883 году.

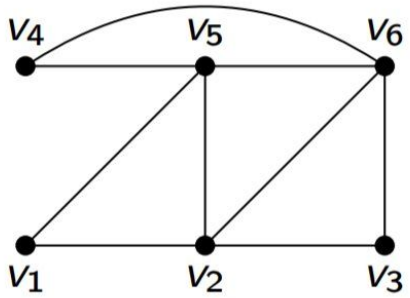
Пусть задан граф **G=(V,E)**. Начинаем с некоторой вершины **p in V** и каждый раз вычеркиваем пройденное ребро. Не проходим по ребру, если удаление этого ребра приводит к разбиению графа на две связные компоненты (не считая изолированных вершин), т.е. необходимо проверять, является ли ребро мостом или нет.

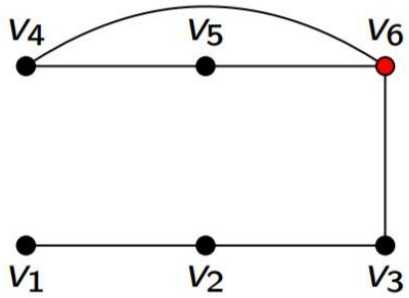
Этот алгоритм неэффективен: время работы оригинального алгоритма $O(|E|^2)$. Если использовать более эффективный алгоритм для поиска мостов[4], то время выполнения

можно снизить до **O(|E|(log |E|)^3 log |E|)**, однако это всё равно медленнее, чем другие алгоритмы.

Алгоритм может быть распространен на ориентированные графы.

Рассмотрим граф, изображенный на рис. 4 (он эйлеров в силу теоремы Эйлера о циклах) и найдем в нем эйлеров цикл. v1 v2 v3 v4 v5 v6 v1 v2 v3 v4 v5 v6 Рис. 4





Пусть на шаге 1 выбрана вершина v1. На выборе на шаге 2 ограничение никак не сказывается; пусть выбрано ребро (v1, v5). На двух следующих итерациях ограничений на выбор по-прежнему не возникает; пусть выбраны ребра (v5, v2) и (v2, v6). Тогда текущим графом становится граф, изображенный на рис. 4 справа (текущая вершина — v6). На следующей итерации нельзя выбрать ребро (v6, v3) из-за ограничения; пусть выбрано ребро (v6, v5). Дальнейший выбор ребер определен однозначно (текущая вершина всегда будет иметь степень 1), так что в итоге будет построен следующий эйлеров цикл: v1 → v5 → v2 → v6 → v5 → v4 → v6 → v3 → v2 → v1

Алгоритм на основе циклов

Будем рассматривать самый общий случай — случай ориентированного мультиграфа, возможно, с петлями. Также мы предполагаем, что эйлеров цикл в графе существует (и состоит хотя бы из одной вершины). Для поиска эйлерова цикла воспользуемся тем, что эйлеров цикл — это объединение всех простых циклов графа. Следовательно, наша задача — эффективно найти все циклы и эффективно объединить их в один.

Реализовать это можно, например, так, рекурсивно:

```
ВсеЦиклы(v)
1. пока есть цикл, проходящий через v, (ищем его через DFS) и
    добавляем все вершины найденного цикла в массив циклы (сохраняя
порядок обхода)
    удаляем цикл из графа
2. идем по элементам массива циклы
```

каждый элемент **циклы**[i] добавляем к ответу
из каждого элемента рекурсивно вызываем себя: ВсеЦиклы(циклы[i])