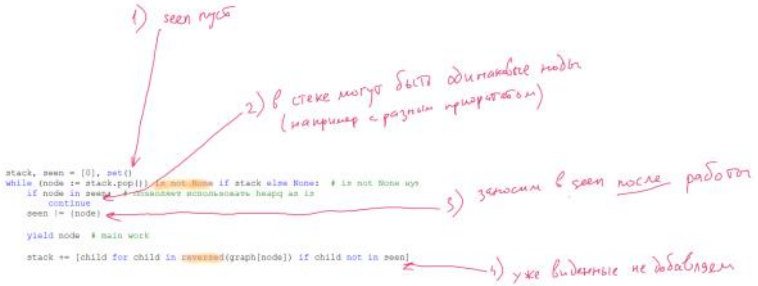
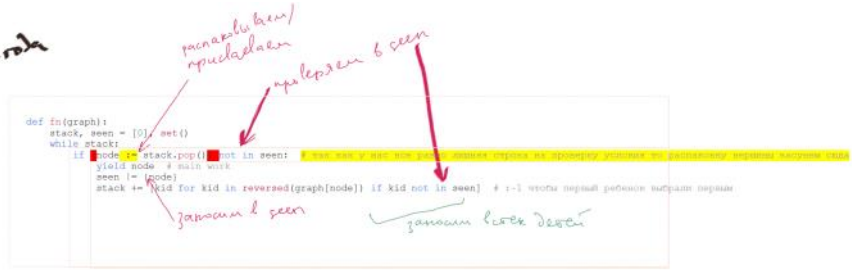
графы II

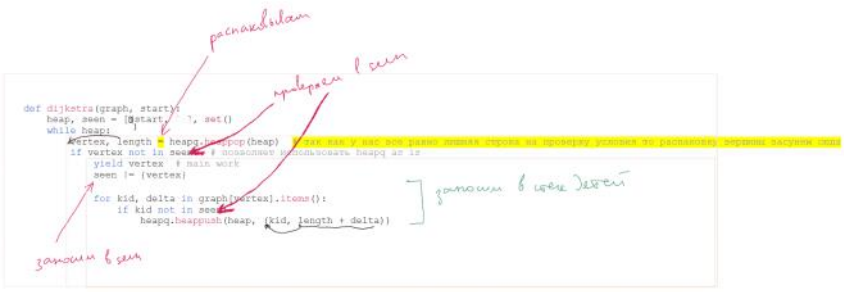
27 января 2022 г. 11:18

ТАБЛИЦА ДЕРЕВЬЕВ И ГРАФОВ



Версия 2022 года



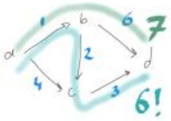


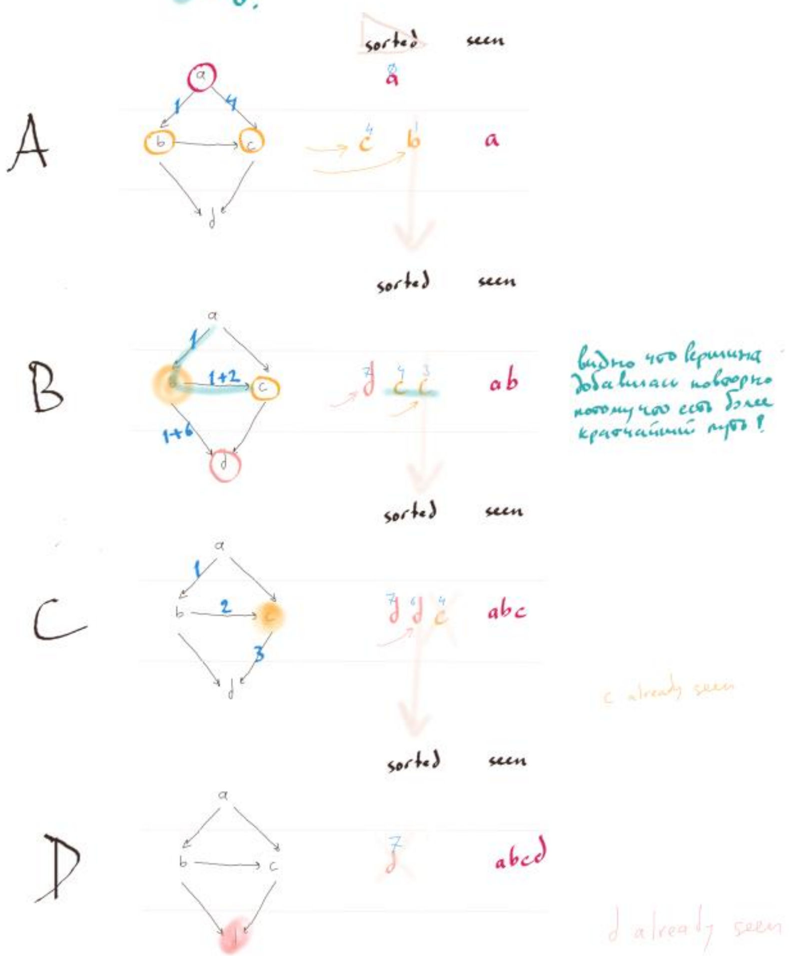
O(E log N)T O(E)S в алгоритме дейкстры повторные вершины допускаются, потому что в эту вершину может быть более кратчайший путь, и одна из дублей вершин пересортировывается в начало кучи

Dijkstra's Algorithm takes $O(E \log N)$. Finding the minimum time required in `signalReceivedAt` takes $O(N)$.

The maximum number of vertices that could be added to the priority queue is E. Thus, push and pop operations on the priority queue take $O(\log E)$ time. The value of E can be at most $N \cdot (N-1)$. Therefore, $O(\log E)$ is equivalent to $O(\log N^2)$ which in turn equivalent to $O(2 \cdot \log N)$. Hence, the time complexity for priority queue operations equals $O(\log N)$.

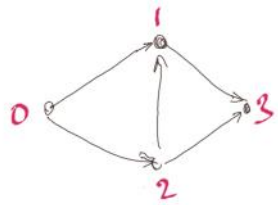
Building the adjacency list will take $O(E)$ space. Dijkstra's algorithm takes $O(E)$ space for priority queue because each vertex could be added to the priority queue $N-1$ time which makes it $N * (N-1)$ and $O(N^2)$ is equivalent to $O(E)$. `signalReceivedAt` takes $O(N)$ space.

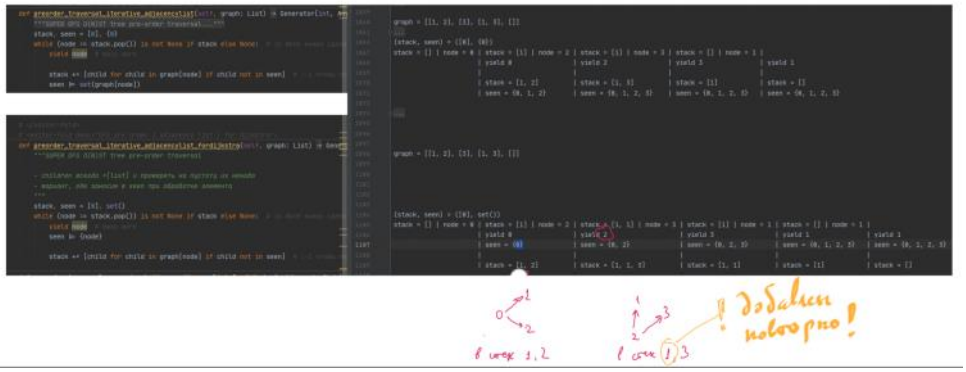




нужно заносить в seen при занесении в стек, так как если я буду заносить в seen только при обработке элемента, то в стеке уже к тому времени могут накопиться несколько дублей (особенно когда ноды ссылаются друг на друга)

- в рекурсивной версии, проверка и установка seen в начале алгоритма (аналогично проверке на None)
- в итеративной версии для Dijkstra (там где возможно добавить одну и ту же вершину дважды, чтобы ее переприоритезировать):
  - seen пустой
  - установка seen тамже где и обработка элемента
  - ребра добавляются только в not seen
- в итеративной версии для DFS/BFS(для traversal, где лучше вершину повторно не добавлять):
  - seen изначально содержит [root]
  - ребра добавляются только в not seen
  - установка seen сразу после добавления в стек/очередь





DFS traversal

вариант, Adjacency list

- is None нужно сделать явную проверку, так как номер первой вершины может быть 0
  - while (node := stack.pop()) is not None if stack else None:
  - if node is None or node in seen: # в рекурсивнйо версии


```
        stack, seen = [0], {0} # manual start vertex
        while (node := stack.pop()) is not None if stack else None: # is None нужно сделать явную проверку, так как номер первой вершины может быть 0
            yield node  # main work

            stack += [kid for kid in graph[node] if kid not in seen]  # все кроме последнего добавляем, последнего добавляем потом, чтобы он был сверху
            seen |= set(graph[node]) # в seen добавляю уже после добавления в стек иначе в стек ничего непопадет
можно было бы сделать оптимизацию и в seen класть не всех детей а тех что not seen(те результат предыдущего шага те приращение которое складывается в stack)


graph_directed = [[1, 2], [2, 3], [3], []]
graph_directed2 = [[1, 2], [3], [1, 3], []]
wilmerlibrary.draw_graph_adjacency_list(graph_directed)


def fn(graph: List):
    """ O(N)TS """
    stack, seen = [0], {0: 0}  # словарь вместо множества только для того чтобы использовать функцию setdefault
    while (node := stack.pop()) is not None if stack else None:  # is None нужно сделать явную проверку, так как номер первой вершины может быть 0
        yield node  # main work
        stack += [seen.setdefault(kid, kid) for kid in graph[node] if kid not in seen]  # :-1 чтобы первый ребенок выбрали первым
```

супер

```
def fn(graph: List):
    """ O(N)TS """
    stack, seen = [0], {0}  # словарь вместо множества только для того чтобы использовать функцию setdefault
    while (node := stack.pop()) is not None if stack else None:  # is None нужно сделать явную проверку, так как номер первой вершины может быть 0
        yield node  # main work
        stack += [seen.add(kid) or kid) for kid in graph[node] if kid not in seen]  # :-1 чтобы первый ребенок выбрали первым


print(list(fn(graph_directed)))
```

рекурсивная версия

```
graph_directed = [[1, 2], [2, 3], [3], []]
graph_directed2 = [[1, 2], [3], [1, 3], []]
wilmerlibrary.draw_graph_adjacency_list(graph_directed)
```

супер

```
def fn(graph, node, seen={0}):
    return [node] + [arr for kid in graph[node] if kid not in seen for arr in fn(graph, seen.add(kid) or kid, seen)] if node else []
```

вариант с полезной работой

```
def fn(graph, node, seen={0}):
    print(node)
    [fn(graph, seen.add(kid) or kid, seen) for kid in graph[node] if kid not in seen]
```

```
def fn(graph, node, seen={0}):
    return [seen.add(node) or node] + [arr for kid in graph[node] if kid not in seen for arr in fn(graph, kid, seen)]
```

```
def fn(graph, node, seen={0: 0}):
    return [seen.setdefault(node, node)] + [arr for kid in graph[node] if kid not in seen for arr in fn(graph, kid, seen)]
```

```
def fn(graph, node, seen={0}):
    result = [node] + [arr for kid in graph[node] if kid not in seen for arr in fn(graph, kid, seen)]
    seen |= set(graph[node]) # в seen добавляю уже после вызова, иначе вызовы дальше не пройдут
    return result

print(list(fn(graph_directed, 0)))
```

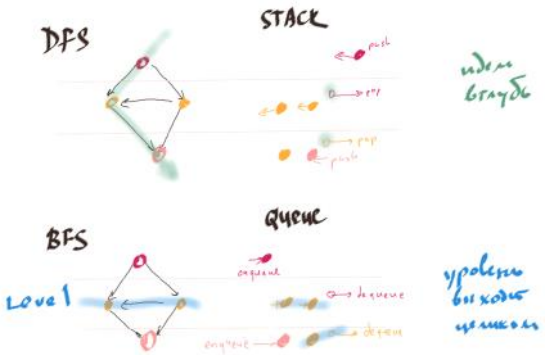
post-order traversal

```
    def traversal(self, graph, start):
        stack, seen = [(start, False)], {start}
        while stack:
            node, again = stack.pop()
            if not again:
                stack += [(node, True)] + [(seen.add(kid) or kid, False) for kid in graph[node] if kid not in seen]
            else:
                yield node
```

интересный вариант алгоритма с удалением нод во врем итерации
result можно заменить на yield

```
    def traversall(self, grid, start):
        stack, seen, result = [start], set(), []
        while stack:
            while grid[node := stack[-1]]:
                stack += grid[node].pop(),
            (res := stack.pop()) not in seen and (result.append(res), seen.add(res))

        return result


adjacency_matrix = {0: [1, 2], 1: [2, 3], 2: [3], 3: []}
print(list(Solution().fn(adjacency_matrix, 0, {0})))
print(list(Solution().traversal(adjacency_matrix, 0)))
print(list(Solution().traversall(adjacency_matrix, 0)))
```

BFS (level-order) traversal



классический подход с заменой стека на очередь

```
    def levelorder_traversal_iterative(self, root: Node) -> Generator[int, Any, None]:
        """SUPER BFS O(N)ST tree level-order traversal

        children всегда =[list] и проверять на пустоту их ненадо
        """
        queue, seen = collections.deque([root]), {root}
        while node := queue.pop() if queue else None:
            yield node.val  # main work

            queue.extendleft([child for child in node.children if child not in seen])
            seen |= set(node.children)
```

```
    def levelorder_traversal_iterative(self, graph):
        queue, seen = collections.deque([0]), {0}
        while (node := queue.popleft()) is not None if queue else None:
            yield node
            queue += [seen.add(kid) or kid for kid in graph[node] if kid not in seen]
```

подход где собираем по уровням

```
graph_directed = [[1, 2], [2, 3], [3], []]
graph_directed2 = [[1, 2], [3], [1, 3], []]
wilmerlibrary.draw_graph_adjacency_list(graph_directed)
```

```
def fn(graph):
    level, seen = [0], {0}
    while level:
        newlevel = []
        for node in level:
            print(node)
            for kid in graph[node]:
                if kid not in seen:
                    newlevel += kid,
                    seen |= {kid}
        level = newlevel
```

```
def fn(graph):
    level, seen = [0], {0}
    while level:
        newlevel = []
        print(level)
        for node in level:
            # print(node)
            newlevel += [kid for kid in graph[node] if kid not in seen]
            seen |= set(graph[node])
        level = newlevel


def fn(graph):
    level, seen = [0], {0: 0} # словарь вместо множества только для того тчобы использовать функцию setdefault
    while level:
        print(level)
        level = [seen.setdefault(kid, kid) for node in level for kid in graph[node] if kid not in seen]
```

супер

```
def fn(graph):
    level, seen = [0], {0}  # словарь вместо множества только для того тчобы использовать функцию setdefault
    while level:
        print(level)
        level = [seen.add(kid) or kid for node in level for kid in graph[node] if kid not in seen]

print(list(fn(graph_directed)))
```

рекурсивная версия

```
    def fn(self, graph, level, seen):
        return [level] + self.fn(graph, [seen.add(kid) or kid for node in level for kid in graph[node] if kid not in seen], seen) if level else []
print(list(Solution().fn(graph_directed, [0], {0})))
```

почти Dijkstra - алгоритмы допускающие дубли

O(N)TS

```
graph_directed = [[1, 2], [2, 3], [3], []]
graph_directed2 = [[1, 2], [3], [1, 3], []]
wilmerlibrary.draw_graph_adjacency_list(graph_directed)


def fn(graph):
    stack, seen = [0], set() # manual start vertex
    while stack:
        if (node := stack.pop()) not in seen:  # так как у нас все равно лишняя строка на проверку условия то распаковку вершины засунем сюда
            yield node  # main work
            seen |= {node}
            stack += [kid for kid in reversed(graph[node]) if kid not in seen]  # :-1 чтобы первый ребенок выбрали первым


def fn(graph):
    stack, seen = [0], {}
    while stack:
        if (node := stack.pop()) not in seen:  # так как у нас все равно лишняя строка на проверку условия то распаковку вершины засунем сюда
            yield seen.setdefault(node, node)  # main work
            stack += [kid for kid in reversed(graph[node]) if kid not in seen]  # :-1 чтобы первый ребенок выбрали первым
```

супер

```
def fn(graph):
    stack, seen = [0], set()
    while stack:
        if (node := stack.pop()) not in seen:  # так как у нас все равно лишняя строка на проверку условия то распаковку вершины засунем сюда
            yield seen.add(node) or node  # main work
            stack += [kid for kid in reversed(graph[node]) if kid not in seen]  # :-1 чтобы первый ребенок выбрали первым



print(list(fn(graph_directed)))
```

рекурсивная версия

```
    def fn(self, graph, node, seen=set()):
        if node not in seen:
            seen |= {node}
            return [node] + [arr for kid in reversed(graph[node]) if kid not in seen for arr in self.fn(graph, kid, seen)]


graph_directed = [[1, 2], [2, 3], [3], []]
print(list(Solution().fn(graph_directed, 0)))
```

можно и не проверять in seen при обходе детей, все равно ф-ия на входе проверяет, но тогда возможны None вызовы функции

```
    def fn(self, graph, node, seen=set()):
        if node not in seen:
            seen |= {node}
            return [node] + [arr for kid in graph[node] for arr in self.fn(graph, kid, seen) or []]
```

если нужно просто выполнить какуюлибо работу на графе (б удет много пустых вызовов)

```
    def fn(self, graph, node, seen=set()):
        if node not in seen:
            print(seen.add(node) or node)
            [self.fn(graph, kid, seen) for kid in graph[node]]
```

Dijkstra - алгоритмы допускающие дубли

O(N log N)TS

- в кучу заносить надо сначала дистанцию чтобы по ней сортировалось а потом уже координаты вершины

```
def dijkstra(graph, start):
    heap, seen = [(0, start)], set()
    while heap:
        length ,vertex, = heapq.heappop(heap)  # так как у нас все равно лишняя строка на проверку условия то распаковку вершины засунем сюда
        if vertex not in seen:  # позволяет использовать heapq as is
            yield vertex  # main work
            seen |= {vertex}

            for kid, delta in graph[vertex].items():
                if kid not in seen:
                    heapq.heappush(heap, (length + delta,kid))

def dijkstra(graph, start):
    heap, seen = [(0, start)], set()
    while heap:
        length ,vertex = heapq.heappop(heap)  # так как у нас все равно лишняя строка на проверку условия то распаковку вершины засунем сюда
        if vertex not in seen:  # позволяет использовать heapq as is
            yield vertex  # main work
            seen |= {vertex}

            [heapq.heappush(heap, (length + delta,kid)) for kid, delta in graph[vertex].items() if kid not in seen]

graph_weighed = {'a': {'b': 1, 'c': 4}, 'b': {'d': 6, 'c': 2}, 'c': {'d': 3}, 'd': {}}  # graph_weighed = dict(a=dict(b=1, c=4), b=dict(d=6, c=2), c=dict(d=3), d=dict())
print(list(dijkstra(graph_weighed, 'a')))



def dijkstra(graph, start):
    heap, seen = [(0, start)], {}
    while heap:
        length ,vertex = heapq.heappop(heap)  # так как у нас все равно лишняя строка на проверку условия то распаковку вершины засунем сюда
        if vertex not in seen:  # позволяет использовать heapq as is
            yield seen.setdefault(vertex, vertex)  # main work
            [heapq.heappush(heap, (length + delta, kid)) for kid, delta in graph[vertex].items() if kid not in seen]
```

супер

```
def dijkstra(graph, start):
    heap, seen = [(0, start)], set()
    while heap:
        length ,vertex = heapq.heappop(heap)  # так как у нас все равно лишняя строка на проверку условия то распаковку вершины засунем сюда
        if vertex not in seen:  # позволяет использовать heapq as is
            yield seen.add(vertex) or vertex  # main work
            [heapq.heappush(heap, (length + delta,kid)) for kid, delta in graph[vertex].items() if kid not in seen]
```

ненастоящая рекурсия, потому что передает heap

```
def fn(self, graph, heap=[], seen=set()):
    length ,vertex= heapq.heappop(heap)
    while heap and vertex in seen:
        length, vertex = heapq.heappop(heap)
    if vertex not in seen:
        seen |= {vertex}
        [heapq.heappush(heap, (length + delta, kid)) for kid, delta in graph[vertex].items() if kid not in seen]
        return [vertex] + self.fn(graph, heap, seen)
    else:
        return []
```

```
graph_weighed = {'a': {'b': 1, 'c': 4}, 'b': {'d': 6, 'c': 2}, 'c': {'d': 3}, 'd': {}}
print(list(Solution().fn(graph_weighed, [('a', 0)])))
```

вывести только листовые ноды

```
def fn(node, stay):
    return [arr for kid in graph[node] if stay.pop(kid, None) for arr in fn(kid, stay)] or [node]
fn(0, collections.Counter(graph.keys() - {0}))
```

```
adjacency_matrix = {0: [1, 2], 1: [2, 3], 2: [3], 3: []}
print(Solution().task(adjacency_matrix))
```

WWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWW

алгоритмы graph land получаются только заменой процедуры получения соседей

```
def neighbors(self, matrix, point):
    for y, x in ((point[0] - 1, point[1]), (point[0] + 1, point[1]), (point[0], point[1] - 1), (point[0], point[1] + 1)):
        if 0 <= y < len(matrix) and 0 <= x < len(matrix[0]) and matrix[y][x] not in (-1, 0) AND matrix[y][x] not in (WALL,GATE) :
            yield y, x  # point(y, x)
def neighbors(self, matrix, point):
    for dy, dx in ((-1, 0), (1, 0), (0, -1), (0, 1)):
        if 0 <= (resy := point[0] + dy) < len(matrix) and 0 <= (resx := point[1] + dx) < len(matrix[0]) AND matrix[y][x] not in (WALL,GATE):
            yield resy, resx  # point(y,x)
```

O(N*M)TS, **graph land** DFS

```
def neighbors(self, matrix, point):
    for dx, dy in ((-1, 0), (1, 0), (0, -1), (0, 1)):
        if 0 <= (resx := point[0] + dx) < len(matrix[0]) and 0 <= (resy := point[1] + dy) < len(matrix):  # and graph[y_new][x_new] != WALL):
            yield resx, resy  # point(x,y)


def fn(matrix, point=(0, 0), seen={(0, 0)}):
    yield point
    [fn(matrix, seen.add(kid) or kid, seen) for kid in self.neighbors(matrix, point) if kid not in seen]

def traversal(matrix, start=(0, 0)):
    stack, seen = [start], {start}
    while stack and (point := stack.pop()) is not None:
        yield point
        stack += [(seen.add(kid) or kid) for kid in self.neighbors(matrix, point) if kid not in seen]
```

```
mat = [[0, 0, 0], [0, 1, 0], [0, 0, 0]]
print_matrix(mat)
print(list(Solution().fn(mat)))
```

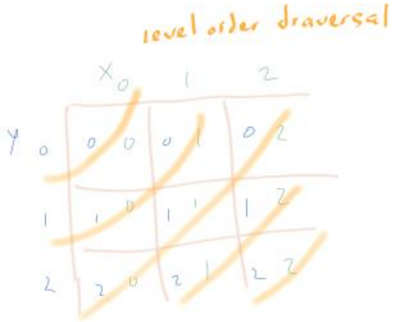
time complexity

. The breadth-first search takes at most m×nm \times nm×n steps to reach all rooms, therefore the time complexity is O(mn)O(mn)O(mn). But what if you are doing breadth-first search from kkk gates?

O(N*M)TS, **graph land** BFS (level order traversal)

if matrix represent physical area, traversal all cells

- ничем не отличается от обычного BFS
  - только работаетм не с нодой а с точкой
  - отдельная процедура нахождения соседей



итеративный

```
    def neighbors(self, matrix, point):
        for dx, dy in ((-1, 0), (1, 0), (0, -1), (0, 1)):
            if 0 <= (resx := point[0] + dx) < len(matrix[0]) and 0 <= (resy := point[1] + dy) < len(matrix):  # and graph[y_new][x_new] != WALL):
                yield resx, resy  # point(x,y)

    def traversal1(self, matrix, start=(0, 0)):
        level, seen = [start], {start}
        while level:
            yield level
            level = [seen.add(kid) or kid for point in level for kid in self.neighbors(matrix, point) if kid not in seen]



    def traversal(self, matrix, start=(0, 0)):
        queue, seen = collections.deque([start]), {start}
        while queue and (point := queue.popleft()) is not None:
            yield point
            queue += [seen.add(kid) or kid for kid in self.neighbors(matrix, point) if kid not in seen]


mat = [[0, 0, 0], [0, 1, 0], [0, 0, 0]]
print_matrix(mat)
print(list(Solution().traversal(mat)))
```

рекурсивный

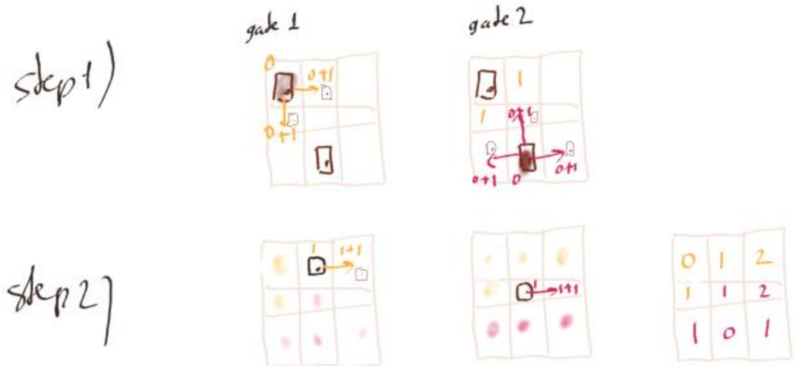
```
    def traversal(self, matrix, start=(0, 0)):
        def fn(level, seen):
            return [level] + fn([seen.add(kid) or kid for point in level for kid in self.neighbors(matrix, point) if kid not in seen], seen) if level else []

        return fn([start], {start})
```

в graph land лучше точку всегда представлять как (y,x) а не (x,y)

- так как итерация по матрице сначала всегда идет по строкам
- и также доступ к элементам матрице тоже идет как matrix[y][x]

суперрешение (одновременно идем от всех целевых нод одновременно)

- идем сразу от всех гейтов, те от каждого гейта во все четыре стороны
- какой гейт быстрее заполнит ПУСТУЮ(еще не заполненную другим гейтом) ячейку




```
    def wallsAndGates(self, rooms: List[List[int]]) -> None:
        """ O(?)TS """
        gates = [(y, x) for y, row in enumerate(rooms) for x, col in enumerate(row) if not col]
        for gatey, gatex in gates:  # идем сразу от всех гейтов, те от каждого гейта во все четыре стороны
            for y, x in ((gatey - 1, gatex), (gatey + 1, gatex), (gatey, gatex - 1), (gatey, gatex + 1)):
                if 0 <= y < len(rooms) and 0 <= x < len(rooms[0]) and rooms[y][x] == 2147483647:  # какой гейт быстрее заполнит ПУСТУЮ(еще не заполненную другим гейтом) ячейку
                    rooms[y][x] = rooms[gatey][gatex] + 1
                    gates += (y, x),
```

WWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWW

seen ненужен, если я использую модификацию исходной матрицы in place

- это модицицирует все алгоритмы

O(N*M)TS, **graph land** DFS

```
mat = [[0, 0, 0], [0, 1, 0], [0, 0, 0]]
WALL = 1

mat[0][0] = WALL
def fn(matrix, y=0, x=0):
    setget = lambda a, b: matrix[a].__setitem__(b, WALL) or (a, b)
    print(y, x)
    [fn(matrix, *setget(i, j)) for i, j in ((y + 1, x), (y - 1, x), (y, x + 1), (y, x - 1)) if 0 <= i < len(matrix) and 0 <= j < len(matrix[0]) and matrix[i][j] != WALL]


mat[0][0] = WALL
def fn(matrix, y=0, x=0):
    print(y, x)
    for i, j in ((y + 1, x), (y - 1, x), (y, x + 1), (y, x - 1)):
        if 0 <= i < len(matrix) and 0 <= j < len(matrix[0]) and matrix[i][j] != WALL:
            matrix[i][j] = WALL
            fn(matrix, i, j)

def traversal(matrix, start=(0, 0)):
    setget = lambda a, b: matrix[a].__setitem__(b, WALL) or (a, b)
```

```
    stack = [setget(*start)]
    while stack and (point := stack.pop()) is not None:
        print(y := point[0], x := point[1])
        stack += [setget(i, j) for i, j in ((y + 1, x), (y - 1, x), (y, x + 1), (y, x - 1)) if 0 <= i < len(matrix) and 0 <= j < len(matrix[0]) and matrix[i][j] != WALL]

def traversall(matrix, start=(0, 0)):
    stack, matrix[start[0]][start[1]] = [start], WALL
    while stack and (point := stack.pop()) is not None:
        print(y := point[0], x := point[1])
        for i, j in ((y + 1, x), (y - 1, x), (y, x + 1), (y, x - 1)):
            if 0 <= i < len(matrix) and 0 <= j < len(matrix[0]) and matrix[i][j] != WALL:
                matrix[i][j] = WALL
                stack += (i, j),




DEB = traversal(mat)
```

O(N*M)TS, **graph land** BFS (level order traversal)

```
grid = [[0, 0, 0], [0, 1, 0], [0, 0, 0]]


WALL = 1

grid[0][0] = 1
def fn(level=[(0, 0)]):
    setget = lambda a, b: grid[a].__setitem__(b, WALL) or (a, b)
    print([(y, x) for y, x in level])
    return level and [fn([setget(i, j) for x, y in level for i, j in ((y + 1, x), (y - 1, x), (y, x + 1), (y, x - 1)) if 0 <= i < len(grid) and 0 <= j < len(grid[0]) and grid[i][j] != WALL])]


print(fn())


def traversall(grid, x0=0, y0=0):
    setget = lambda a, b: grid[a].__setitem__(b, WALL) or (a, b)
    level = [setget(x0, y0)]
    while level:
        print([(y, x) for y, x in level])
        level = [setget(i, j) for y, x in level for i, j in ((y + 1, x), (y - 1, x), (y, x + 1), (y, x - 1)) if 0 <= i < len(grid) and 0 <= j < len(grid[0]) and grid[i][j] != WALL]


def traversal(grid, start=(0, 0)):
    setget = lambda a, b: grid[a].__setitem__(b, WALL) or (a, b)
    queue = collections.deque([setget(*start)])
    while queue and (point := queue.popleft()) is not None:
        print(y := point[0], x := point[1])
        queue += [setget(i, j) for i, j in ((y + 1, x), (y - 1, x), (y, x + 1), (y, x - 1)) if 0 <= i < len(grid) and 0 <= j < len(grid[0]) and grid[i][j] != WALL]


print(traversal(grid))
```

почти Dijkstra - алгоритмы допускающие дубли

```
mat = [[0, 0, 0], [0, 1, 0], [0, 0, 0]]
WALL = 1


def fn(matrix, y=0, x=0):
    if 0 <= y < len(matrix) and 0 <= x < len(matrix[0]) and matrix[y][x] != WALL:  # not in seen or not wall
        matrix[y][x] = WALL  # aka add in {seen}
        fn(matrix, y + 1, x), fn(matrix, y - 1, x), fn(matrix, y, x + 1), fn(matrix, y, x - 1)


def traversal(matrix, start=(0, 0)):
    stack = [start]
    while stack and (point := stack.pop()) is not None:
        if 0 <= (y := point[0]) < len(matrix) and 0 <= (x := point[1]) < len(matrix[0]) and matrix[y][x] != WALL:  # not in seen or not wall
            matrix[y][x] = WALL  # aka add in {seen}
            stack += (y + 1, x), (y - 1, x), (y, x + 1), (y, x - 1)


DEB = traversal(mat)
```

примеры всех типов решений 1

итеративное

```
        def neighbors(self, matrix, point):
            for y, x in ((point[0] - 1, point[1]), (point[0] + 1, point[1]), (point[0], point[1] - 1), (point[0], point[1] + 1)):
                if 0 <= y < len(matrix) and 0 <= x < len(matrix[0]) and matrix[y][x] not in (-1, 0) and matrix[y][x] == '1':  # go to the land only
                    yield y, x  # point(y, x)
        def numIslands(self, grid: List[List[str]]) -> int:
            """ O(N*M)TS """
            lands, result = [(y, x) for y, row in enumerate(grid) for x, col in enumerate(row) if col == '1'], 0
            for land in lands:
                if grid[land[0]][land[1]] == '1': # будем начинать обход только с реальной земли тк она могла изменится за предыдущий DFS  (для точго чтобы сразу посчитать результат)
                    stack, seen, result = [land], {land}, result + 1
                    while stack and (point := stack.pop()) is not None:
                        grid[point[0]][point[1]] = '0' # пометим землю океаном
                        stack += [(seen.add(kid) or kid) for kid in self.neighbors(grid, point) if kid not in seen]

            return result



    def numIslands(self, grid: List[List[str]]) -> int:
        """ O(N)TS """
        result = 0
        for landy, landx in itertools.product(range(len(grid)), range(len(grid[0]))):
            if grid[landy][landx] == '1':
                stack, result = [(landy, landx)], result + 1
                while stack and (point := stack.pop()) is not None:
                    if 0 <= (y := point[0]) < len(grid) and 0 <= (x := point[1]) < len(grid[0]) and grid[y][x] == '1':  # not in seen or not wall
                        grid[y][x] = '0'  # aka add in {seen}
                        stack += (y + 1, x), (y - 1, x), (y, x + 1), (y, x - 1)
        return result


    def numIslands(self, grid: List[List[str]]) -> int:
        """ O(N*M)TS """
        result, setget = 0, lambda a, b: grid[a].__setitem__(b, '0') or (a, b)

        for landy, landx in itertools.product(range(len(grid)), range(len(grid[0]))):
            if grid[landy][landx] == '1':
                stack, result = [setget(landy, landx)], result + 1
                while stack and (point := stack.pop()) is not None:
                    y, x = point
                    stack += [setget(i, j) for i, j in ((y + 1, x), (y - 1, x), (y, x + 1), (y, x - 1)) if 0 <= i < len(grid) and 0 <= j < len(grid[0]) and grid[i][j] == '1']

        return result
```

рекурсивное

```
    def neighbors(self, matrix, point):
        for y, x in ((point[0] - 1, point[1]), (point[0] + 1, point[1]), (point[0], point[1] - 1), (point[0], point[1] + 1)):
            if 0 <= y < len(matrix) and 0 <= x < len(matrix[0]) and matrix[y][x] not in (-1, 0) and matrix[y][x] == '1':  # go to the land only
                yield y, x  # point(y, x)
    def numIslands(self, grid: List[List[str]]) -> int:
        """ O(N*M) TS """

        def fn(point=(0, 0), seen={(0, 0)}):
            grid[point[0]][point[1]] = '0'
            [fn(seen.add(kid) or kid, seen) for kid in self.neighbors(grid, point) if kid not in seen]

        lands, result = [(y, x) for y, row in enumerate(grid) for x, col in enumerate(row) if col == '1'], 0
        for land in lands:
            if grid[land[0]][land[1]] == '1': # будем начинать обход только с реальной земли тк она могла изменится за предыдущий DFS  (для точго чтобы сразу посчитать результат)
                result += 1
                fn(land, {land})

        return result
```

```
    def numIslands(self, grid: List[List[str]]) -> int:
        """ O(N*M) TS """

        def fn(y=0, x=0):
            if 0 <= y < len(grid) and 0 <= x < len(grid[0]) and grid[y][x] == '1':  # not in seen or not wall
                grid[y][x] = '0'  # aka add in {seen}
                fn(y + 1, x), fn(y - 1, x), fn(y, x + 1), fn(y, x - 1)

        return sum(fn(landy, landx) or 1 for landy in range(len(grid)) for landx in range(len(grid[0])) if grid[landy][landx] == '1')
```

```
    def numIslands(self, grid: List[List[str]]) -> int:
        """ O(N) TS """
        result, setget = 0, lambda a, b: grid[a].__setitem__(b, '0') or (a, b)

        def fn(y=0, x=0):
            [fn(*setget(i, j)) for i, j in ((y + 1, x), (y - 1, x), (y, x + 1), (y, x - 1)) if 0 <= i < len(grid) and 0 <= j < len(grid[0]) and grid[i][j] == '1']

        return sum(fn(*setget(landy, landx)) or 1 for landy, landx in itertools.product(range(len(grid)), range(len(grid[0]))) if grid[landy][landx] == '1')
```

примеры всех типов решений 2

```
    def maxAreaOfIsland(self, grid: List[List[int]]) -> int:
        """ O(N*M) TS """

        def fn(y, x):
            if 0 <= y < len(grid) and 0 <= x < len(grid[0]) and grid[y][x] == 1:
                grid[y][x] = 0
                return 1 + (fn(y + 1, x) or 0) + (fn(y - 1, x) or 0) + (fn(y, x + 1) or 0) + (fn(y, x - 1) or 0)

        return max([fn(y, x) or 1 for y in range(len(grid)) for x in range(len(grid[0])) if grid[y][x] == 1], default=0) # for case if no ilands
```

вынесем функцию

```
    def maxAreaOfIsland(self, grid: List[List[int]]) -> int:
        """ O(N*M) TS """
        getset = lambda y, x: grid[y].__setitem__(x, 0) or 1 if 0 <= y < len(grid) and 0 <= x < len(grid[0]) and grid[y][x] == 1 else 0

        def fn(y, x):
            return getset(y, x) and 1 + fn(y + 1, x) + fn(y - 1, x) + fn(y, x + 1) + fn(y, x - 1)

        return max([fn(y, x) for y in range(len(grid)) for x in range(len(grid[0])) if grid[y][x] == 1], default=0)
```

- используется фишка удаления обрабатываемой ячейки во время итерации
  - этим самым она как бы помечается как seen, так как в словаре ее уже небудет
- используется фишка того что всю матрицу загоняем в словарь, и получается что если x±1 или y±1 ячейки нет в словаре то она не в границах и поэтому уже явно на границы проверять ненадо

```
    def maxAreaOfIsland(self, grid: List[List[int]]) -> int:
        """ O(N*M) TS """
        grid = {(y, x): val for y, row in enumerate(grid) for x, val in enumerate(row)}

        def fn(point):
            y, x = point
            return grid.pop(point, 0) and 1 + fn((y + 1, x)) + fn((y - 1, x)) + fn((y, x + 1)) + fn((y, x - 1))

        return max(map(fn, set(grid)))
```

- не используем кортеж, и ненадо распаковывать координаты
- сразу удалим обработанную ячейку (как бы пометим seen)

```
    def maxAreaOfIsland(self, grid: List[List[int]]) -> int:
        """ O(N*M) TS """
        grid = {(y, x): grid[y][x] for y in range(len(grid)) for x in range(len(grid[0]))}

        def fn(y, x):
            return grid.pop((y, x), 0) and 1 + fn(y + 1, x) + fn(y - 1, x) + fn(y, x + 1) + fn(y, x - 1) # это позволяет возвращать ноль в base case

        return max(fn(y, x) for y, x in grid.copy()) # чтобы не было ошибки мутирования во время итерации
```

оригинальный подход, обходим матрицу только в двух направлениях тока слева направо и сверху вниз

- смотрим только наверхнюю и левую ячейку (те на предыдущие, которые уже прошли) и если они вдруг едёнички то вычтем сразу двойку так как до этого в этом направлении не вычитали


```
class Solution:
```

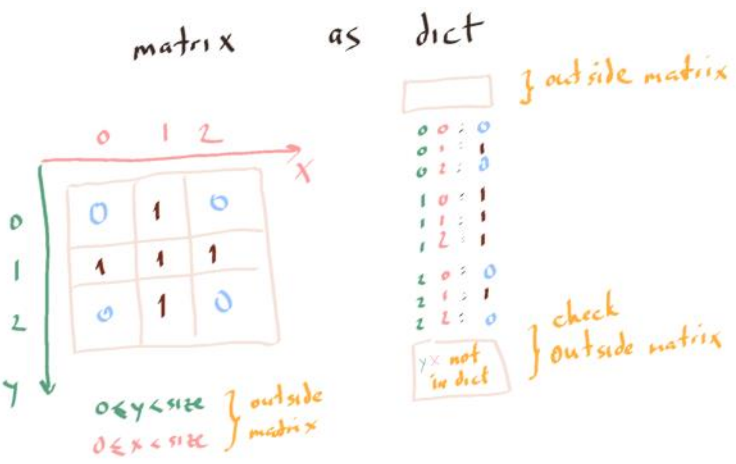
итерация с вызовом функции вынесена в одну строку с генератором

```
    def islandPerimeter(self, grid: List[List[int]]) -> int:
        """ O(N*M) TS"""

        def fn(y, x):
            return 4 + (-2 if 0 < y and grid[y - 1][x] == 1 else 0) + (-2 if 0 < x and grid[y][x - 1] == 1 else 0)

        return sum(fn(y, x) for y in range(len(grid)) for x in range(len(grid[0])) if grid[y][x] == 1)
```

wwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwww

представим матрицу как словарь

- seen ненужен, я просто удаляю обработанную ячейку из словаря
- проверка на границы матрицы ненужны, (y,x) просто не будет в словаре



алгоритмы обхода всегда начинают с определенной вершины

а не перебирают все ячейки матрицы
пожтому стартовой ячейкой лучше указать ту с содержимым == 1

но в данном кейсе (где представили граф как словарь) мы можем перебрать все вообще ячейки потому что вернется все равно 0

O(N*M)TS, **graph land** DFS

```
class Solution:


    def traversal(self, grid):
        grid, groups = collections.OrderedDict(((y, x), (val, False)) for y, row in enumerate(grid) for x, val in enumerate(row)), []
        while grid:
            (y, x), (val, seen) = grid.popitem()
            if val:
                if seen:
                    groups[-1] += (y, x),
                else:
                    groups += [[(y, x)]]
                for pt in ((y + 1, x), (y - 1, x), (y, x + 1), (y, x - 1)):
                    if dot := grid.get(pt):
                        grid[pt] = (dot[0], True)
                        grid.move_to_end(pt)

        print(groups)




    def traversal(self, grid, start=(0, 0)):
        grid = {(y, x): val for y, row in enumerate(grid) for x, val in enumerate(row)}
        stack = [grid.pop(start) and start]
        while stack and (point := stack.pop()):
            print(y := point[0], x := point[1])
            stack += [pt for pt in ((y + 1, x), (y - 1, x), (y, x + 1), (y, x - 1)) if grid.pop(pt, False)]  # if land==1



    def maxAreaOfIsland11(self, grid: List[List[int]]) -> int:
        """ O(N*M)TS """

        def fn(y, x):
            print(y, x)
            [fn(*pt) for pt in ((y + 1, x), (y - 1, x), (y, x + 1), (y, x - 1)) if grid.pop(pt, False)]

        grid = {(y, x): val for y, row in enumerate(grid) for x, val in enumerate(row)}
        [fn(*pt) for pt in grid.copy() if grid.pop(pt, False)]  # pop needed as seen

    def maxAreaOfIsland1(self, grid: List[List[int]]) -> int:
        """ O(N*M)TS """

        def fn(y, x):
            print(y, x)
            return [(y, x)] + [arr for pt in ((y + 1, x), (y - 1, x), (y, x + 1), (y, x - 1)) if grid.pop(pt, False) for arr in fn(*pt)]

        grid = {(y, x): val for y, row in enumerate(grid) for x, val in enumerate(row)}
        return fn(grid.pop((0, 0)) and 0, 0)  # pop needed as seen
```

```
# grid = [[0, 1, 0], [0, 1, 0], [0, 1, 0]]
grid = [[0, 0, 1], [1, 0, 0], [0, 0, 0]]
print_matrix(grid)
# print(Solution().maxAreaOfIsland(grid))
print(Solution().traversal(grid))
```

O(N*M)TS, **graph land** BFS (level order traversal)

```
grid = [[1, 0, 0], [1, 1, 0], [1, 0, 0]]
WALL = 0


grid[0][0] = WALL # aka seen
def fn(level=[(0, 0)]):
    print(level)
    level and fn([pt for y, x in level for pt in ((y + 1, x), (y - 1, x), (y, x + 1), (y, x - 1)) if grid.pop(pt, False)])


# grid1 = {(y, x): val for y, row in enumerate(grid) for x, val in enumerate(row)}

def traversal(grid, x0=0, y0=0):
```

```
    grid = {(y, x): val for y, row in enumerate(grid) for x, val in enumerate(row)}
    level = [grid.pop((0, 0)) and (0, 0)]
    while level:
        print([(y, x) for y, x in level])
        level = [pt for y, x in level for pt in ((y + 1, x), (y - 1, x), (y, x + 1), (y, x - 1)) if grid.pop(pt, False)]


print_matrix(grid)
print(traversal(grid))
print_matrix(grid)
```

почти Dijkstra - алгоритмы допускающие дубли

```
    def traversal(self, grid, start=(0, 0)):
        stack, grid = [start], {(y, x): val for y, row in enumerate(grid) for x, val in enumerate(row)}
        while stack and (point := stack.pop()):
            if grid.pop(point, False):
                print(y := point[0], x := point[1])
                stack += (y + 1, x), (y - 1, x), (y, x + 1), (y, x - 1)
```

```
    def maxAreaOfIsland0(self, grid: List[List[int]]) -> int:
        """ O(N*M)TS """

        def fn(y, x):
            return grid.pop((y, x), 0) and 1 + fn(y + 1, x) + fn(y - 1, x) + fn(y, x + 1) + fn(y, x - 1)

        grid = {(y, x): grid[y][x] for y in range(len(grid)) for x in range(len(grid[0]))}
        return max(fn(y, x) for y, x in grid.copy())

    def maxAreaOfIsland1(self, grid: List[List[int]]) -> int:
        """ O(N*M)TS """

        def fn(y, x):
            if (val := grid.pop((y, x), None)) is not None:
                print(y, x)
                fn(y + 1, x), fn(y - 1, x), fn(y, x + 1), fn(y, x - 1)

        grid = {(y, x): val for y, row in enumerate(grid) for x, val in enumerate(row)}
        [fn(y, x) for y, x in grid.copy()]

    def maxAreaOfIsland2(self, grid: List[List[int]]) -> int:
        """ O(N*M)TS """

        def fn(y, x):
            if (val := grid.pop((y, x), None)) is not None:
                print(y, x)
                return [(y, x)] + fn(y + 1, x) + fn(y - 1, x) + fn(y, x + 1) + fn(y, x - 1)
            return []

        grid = {(y, x): val for y, row in enumerate(grid) for x, val in enumerate(row)}
        return fn(0, 0)

    def maxAreaOfIsland3(self, grid: List[List[int]]) -> int:
        """ O(N*M)TS """

        def fn(y, x):
            return grid.pop((y, x), []) and [(y, x)] + fn(y + 1, x) + fn(y - 1, x) + fn(y, x + 1) + fn(y, x - 1)

        grid = {(y, x): (val or 10) for y, row in enumerate(grid) for x, val in enumerate(row)}
        return [fn(y, x) for y, x in grid.copy()]
```

Dijkstra - алгоритмы, нет весов ребер(длина каждого ребра=1)

```
    def dijkstra(self, graph, start):  # weighted graph
        heap, grid = [(0, *start)], {(y, x): 1 - val for y, row in enumerate(graph) for x, val in enumerate(row)}
        while heap:
            length, y, x = heapq.heappop(heap)
            if grid.pop((y, x), False):
                yield (y, x)
                [heapq.heappush(heap, (length + 1, *kid)) for kid in ((y + 1, x), (y - 1, x), (y, x + 1), (y, x - 1)) if kid in grid]


# graph_weighed = {'a': {'b': 1, 'c': 4}, 'b': {'d': 6, 'c': 2}, 'c': {'d': 3}, 'd': {}}
# print(list(Solution().dijkstra(graph_weighed, 'a')))
maze = [[0, 0, 1, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 1, 0], [1, 1, 0, 1, 1], [0, 0, 0, 0, 0]]
print(list(Solution().dijkstra(maze, (0, 4))))
```

WWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWW

topologicl sort - топологическая сортировка (post-order DFS с повторными нодами в стеке)

TS topological sort / топологическое упорядочивание/сортировка

на вход не дана стартовая вершина графа, потому я должен перебрать все его вершины, через несколько запусков DFS (вершина может быть вообще несвязанной)

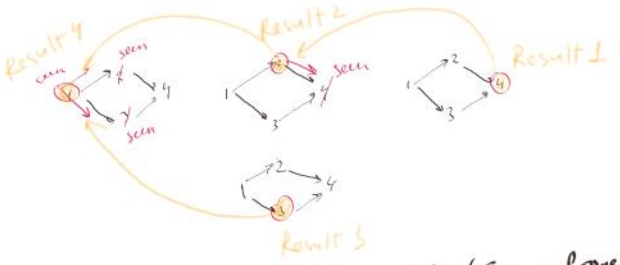
- рекурсивный алгоритм - DFS post-order
- в нескольких DFS один общий seen, чтобы не брать подграф уже обработанный в предыдущем DFS
- если первый вызов DFS пришолся на конец графа то он вернет только хвост
- если вызов DFS пришолся на середину графа, то он пройдет от середины графа до конца и вернет вершины в обратном порядке (те начиная с хвостовой)
- так как post-order traversal то в итоге нужно весь результат перевернуть[::-1]

алгоритм определяет **стоковую** вершину

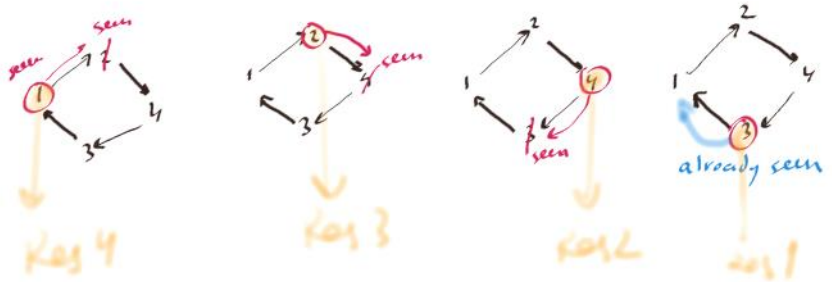
истоковую вершину можно определить следуя по ребрам назад

- если вы продолжите следовать по входящим ребрам назад из произвольной вершины ориентированного ациклического графа, вы обязательно достигнете истоковой вершины. (В противном случае вы породите цикл, что невозможно.)
- В качестве альтернативы: прослеживание не по входящим, а по исходящим ребрам в доказательстве леммы 8. 7 показывает, что каждый граф DAG имеет минимум одну стоковую вершину, и мы можем наполнить топологическое упорядочение справа налево извлеченными подряд стоковыми вершинами.

Post order traversal Acyclic graph (без повторений вершин в цепке)



Post order traversal Cyclic graph (без повторений вершин в цепке)



ниже примеры неправильны так как из первой вершины вызов разветвляется на два

```
DFS-Loop (graph G)
- mark all nodes unexplored
- current_label = n   [to keep track of ordering]
- for each vertex v ∈ G:   [in some precise DFS can]
    - if v not yet explored
        -DFS(G,v)
```

```
DFS(graph G, start vertex s)
- mark s explored
- for every edge (s,v):
    - if v not yet explored
        -DFS(G,v)
- set f(s) = current_label
- current_label --
```

алгоритм работает даже с циклами
НО невозможно топологически упорядочить вершины графа, содержащего ориентированный цикл.

топологических упорядочений может быть несколько

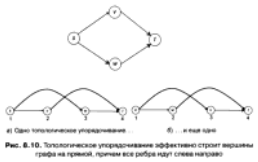


Рис. 8.10. Топологическое упорядочивание эффективно строит вершины графа на прямой, причем все ребра идут слева направо

```
class Solution():

    def topo_sort1(self, graph: List) -> List[int]:
        """ O(E*V)TS"""

        def fn(node, seen, result):
            if node not in seen:  # is None нужно сделать явную проверку, так как номер первой вершины может быть 0
                seen |= {node}
```

```
            [fn(kid, seen, result) for kid in graph[node] if kid not in seen]
            result += [node]  # присваиваю уже на выходе из рекурсии (те сначала конечную вершину)

        seen, result = set(), []
        [fn(node, seen, result) for node in graph]
        return result[::-1]
```

алгоритм **допускающий повторения** вершин в стеке

```
    def topo_sort(self, graph: List) -> List[int]:
        """ O(E*V)TS"""

        def fn(node, seen):
            if node not in seen:  # is None нужно сделать явную проверку, так как номер первой вершины может быть 0
                seen |= {node}
                return [arr for kid in graph[node] if kid not in seen for arr in fn(kid, seen)] + [node] # post-order DFS

        seen = set() # в нескольких DFS один общий seen, чтобы не брать подграф уже обработанный в предыдущем DFS
        return list(itertools.chain.from_iterable(fn(node, seen) or [] for node in graph))[::-1] # так как если начну со стока то к началу не продвинусь
```

обычный рекурсивный DFS (без повторений)

```
    def topo_sort(self, graph: List) -> List[int]:
        """ O(E*V)TS"""

        def fn(node, seen):
            return [arr for kid in graph[node] if kid not in seen for arr in fn(seen.add(kid) or kid, seen)] + [node] if node is not None else []

        seen = set()
        return list(itertools.chain.from_iterable(fn(node, seen.add(node) or seen) for node in graph if node not in seen))[::-1] # в функции нету входной проверки на seen, поэтому ее лучше не вызывать с уже виденной нодой
```

обычный итеративный DFS (без повторений)

```
    def traversal(self, graph):
        seen, result = set(), []
        for start in graph:
            if start not in seen:
                stack = [(seen.add(start) or start, False)]
                while stack:
                    node, again = stack.pop()
                    if not again:
                        stack += [(node, True)] + [(seen.add(kid) or kid, False) for kid in graph[node] if kid not in seen]
                    else:
                        result += node,
        return result[::-1]
```

вместо seen, здесь держу множество с оставшимися для итерации нодами (решение короче на одну строку)
можно заменить на словарь stay.pop(kid, False)

```
    def traversal(self, graph):
        stay, result = set(graph), []
        while stay:
            stack = [(stay.pop(), False)]
            while stack:
                node, again = stack.pop()
                if not again:
                    stack += [(node, True)] + [(stay.discard(kid) or kid, False) for kid in graph[node] if kid in stay]
                else:
                    result += node,
        return result[::-1]
```

~~обычный итеративный DFS (с повторениями)~~

```
~~    def traversal(self, graph):~~
~~        seen, result = set(), []~~
~~        for start in graph:~~
~~            if start not in seen:~~
~~                stack = [start]~~
~~                while stack:~~
~~                    if (node := stack.pop()) not in seen:~~
~~                        result += [seen.add(node) or node]~~
~~                        stack += [kid for kid in reversed(graph[node]) if kid not in seen]~~
~~        return result[::-1]~~
```

~~то же что и предыдущий но убрана одна проверка (короче на одну строку)~~

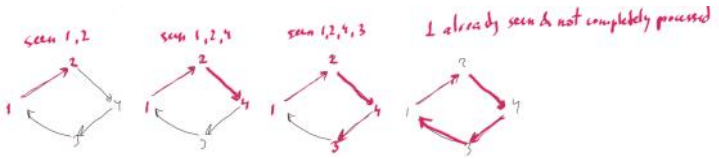
```
~~    def traversal(self, graph):~~
~~        seen, result = set(), []~~
~~        for start in graph:~~
~~            stack = [start]~~
~~            while stack:~~
~~                if (node := stack.pop()) not in seen:~~
~~                    result += [seen.add(node) or node]~~
~~                    stack += [kid for kid in reversed(graph[node]) if kid not in seen]~~
~~        return result[::-1]~~
```

```
g1 = {4: [], 3: [4], 2: [4], 1: [2, 3]}
g2 = {4: [3], 3: [1], 2: [4], 1: [2]}
g = {4: [3], 3: [2], 2: [4], 1: [2, 3]}
wilmerlibrary.draw_graph_adjacency_dict(g1)
print(Solution().topo_sort(g1))
```

обнаружение циклов через toposort (post-order DFS с повторными нодами в стеке) (только для directed graph)

- в обычном алгоритме обхода с повторениями, на самом деле можно добавлять детей повторно
- надо отключить оптимизацию: обычно оптимизируют тем что не добавляют уже seen детей
    - тогда вызов такого уже seen попадет снова в функцию и я смогу определить цикл

_




```
    def topo_sort2(self, graph: List) -> List[int]:
        """ O(E*V)TS"""

        def fn(node, seen, fin):
            if node not in seen:  # is None нужно сделать явную проверку, так как номер первой вершины может быть 0
                seen |= {node}
                return [arr for kid in graph[node] for arr in fn(kid, seen, fin) or []] + [fin.add(node) or node]  # post-order DFS
            (node in seen and node not in fin) and print("CYCLE")

        seen, fin = set(), set()  # в нескольких DFS один общий seen, чтобы не брать подграф уже обработанный в предыдущем DFS
        return list(itertools.chain.from_iterable(fn(node, seen, fin) or [] for node in reversed(graph) if node not in seen))[::-1]  # а здесь наоборот нужна проверка на seen
```

если нужно просто обнаружить цикл а не возвращать ноды

```
        def fn(node):
            if node not in seen:
                seen.add(node), [fn(kid) for kid in graph[node]], fin.add(node)
            (node in seen and node not in fin) and cycle.add(1)

        seen, fin, cycle = set(), set(), set()
        [fn(node) for node in graph.copy() if node not in seen]
        return len(cycle) == 1
```

```
g = {4: [], 3: [4], 2: [4], 1: [2, 3]}
g22 = {4: [3], 3: [1], 2: [4], 1: [2]}
g2 = {4: [3], 3: [2], 2: [4], 1: [2, 3]}
wilmerlibrary.draw_graph_adjacency_dict(g)
# print(Solution().topo_sort(g1))
print(list(Solution().topo_sort2(g)))
```

алгоритм обнаружения циклов кана

```
        graph, inp = collections.defaultdict(list), collections.Counter({k: 0 for k in range(numCourses)})
        [ (inp.update({b: 1}), graph[a].append(b)) for a, b in prerequisites]

        topo = [i for i in range(numCourses) if inp[i] <= 0]
        for node in topo:  # массив мутирует во время обхода
            [ inp.update({kid: -1}) for kid in graph[node]]
            topo += [kid for kid in graph[node] if inp[kid] <= 0]  # добавим в стек вершины с нулями (если они образовались на текущем цикле)
        return len(stack) == numCourses
```

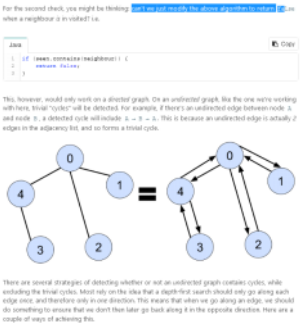
```
graph, inp = collections.defaultdict(list), collections.Counter({k: 0 for k in range(numCourses)})
[( inp.update({b: 1}) , graph[a].append(b)) for a, b in prerequisites]

topo = [i for i in range(numCourses) if inp[i] <= 0]
for node in topo:  # массив мутируем во время обхода
    for kid in graph[node]:
        inp[kid] -= 1
        inp[kid] <= 0 and topo.append(kid)
return len(topo) == numCourses
```

```
graph, inp = collections.defaultdict(list), collections.Counter({k: 0 for k in range(numCourses)})
[(inp.update({b: 1}), graph[a].append(b)) for a, b in prerequisites]

topo = [k for k, v in inp.items() if v <= 0]
[inp.update({kid: -1}) or inp[kid] == 0 and topo.append(kid) for node in topo for kid in graph[node]]
return len(topo) == numCourses
```

обнаружение циклов через обычный DFS(с повторными нодами в стеке) для ненаправленных графов

- если ребенок собирается ходить в ноду которая уже в seen (то значит это цикл)

мы не можем просто проверить ноду на seen(чтобы детектировать цикл) так как у двух
нод есть связи в обе стороны



можно после прохождения ребра в прямом направлении:
удалить обратное ребро (чтобы мы по нему больше не смогли пройти)

- The first strategy is to simply delete the opposite direction edges from the adjacency list. In other words, when we follow an edge A → B, we should lookup Bs adjacency list and delete A from it, effectively removing the opposite edge of B → A.

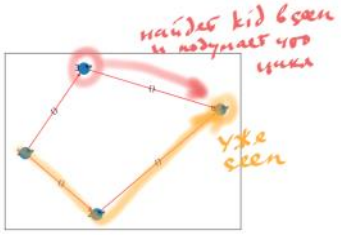
можно после прохождения ребра в прямом направлении:
поместить ноду в словарь parent[куда]=[ноды откуда пришли]

- The second strategy is, instead of using a seen set, to use a seen map that also keeps track of the "parent" node that we gotto a node from. We'll call this map parent. Then, when we iterate through the neighbours of a node, we ignore the "parent" node as otherwise it'll be detected as a trivial cycle (and we know that the parent node has already been visited by this point anyway). The starting node (0 in this implementation) has no "parent", so put it as -1.
- At first, it's a little more difficult to understand why this strategy even works. A good way to think about it is to remember that like the first approach, we just want to avoid going along edges we've already been on (in the opposite direction). The parent links prevent that, aseach node is only entered for exploration once. So, imagine you're walking through a maze, with the condition that you're not allowed to go back alongany path you've already been on. If you still somehow end up somewhere you were previously, there must have been a cycle!

здесь исходится из предположения что мы знаем начало графа
когда находимся в ноде, то мы должны исключить обратный путь к паренту

```
def fn(node, parent, seen, cycle):
    if node not in seen:
        (seen - {parent}) & set(grid[node]) and cycle.add(node)
        seen.add(node), [fn(kid, node, seen, cycle) for kid in grid[node] if kid != parent]

return fn(0, -1, seen := set(), cycle := set()) or (len(cycle) == 0 and len(seen)==n) # также проверим на связность
```

для направленного графа не работает, потому что в одну и ту же ноду может быть несколько путей (а алгоритм подумает что это цикл)

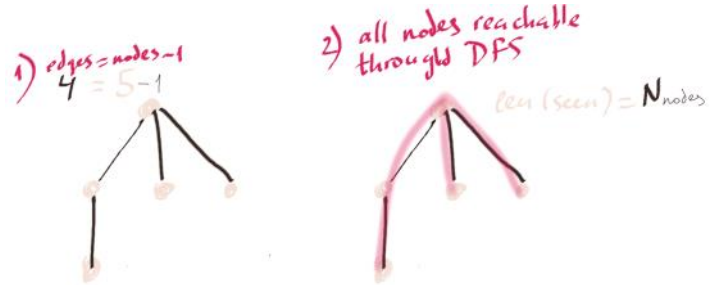


если граф не дерево то значит в нем есть циклы

дерево - это ненаправленный граф без циклов

- в дереве **ровно** N-1 ненаправленных ребер (ни больше не меньше)
- в дереве все ноды связаны (те граф связанный)
  - количество ребер можно просто посчитать
  - а достижимость всех нод можно проверить через DFS (посчитав потом ноды в seen)

так как корневая вершина известна то нужно сделать всего один DFS



здесь исходится из предположения что мы знаем начало графа

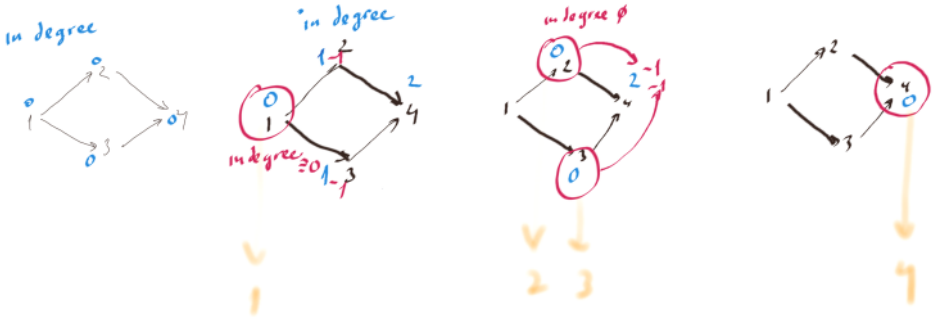
```
class Solution:

    def validTree(self, n: int, edges: List[List[int]]) -> bool:
        """ O(N+E) """
        grid = collections.defaultdict(list)
```

```
        [(grid[a].append(b), grid[b].append(a)) for a, b in edges]

        def fn(node, seen):
            [fn(seen.add(kid) or kid, seen) for kid in grid[node] if kid not in seen]

        return len(edges) == n - 1 and (fn(0, seen := {0}) or len(seen) == n)


#print(Solution().validTree(n=5, edges=[[0, 1], [1, 2], [2, 3], [1, 3], [1, 4]]))
print(Solution().validTree(n=5, edges=[[0, 1], [0, 2], [0, 3], [1, 4]]))  # print(Solution().validTree(4, [[0, 1], [2, 3]]))
```

алгоритм кана kahn (только для DAG) для toposort и обнаружения циклов

по сути это BFS

- итеративно идем сразу в правильном порядке топологической сортировки
- если у вершины ноль входящих ребер значит она первая
  - и после обработки текущей вершины уменьшаем количество входящих ребер ее детей
  - и далее продолжаем обход от тех детей у которых 0-входящих ребер




```
class Solution:

    def topo_sort(self, graph):
        """ O(E*V)TS """
        inp = collections.Counter({k: 0 for k in graph})
        [inp.update({kid: 1}) for node in graph for kid in graph[node]]   # сделаем матрицу количества входящих ребер

        min_ = min(inp.values())
        inp = {k: v - min_ for k, v in inp.items()}

        stack = [k for k, v in inp.items() if v <= 0]
        while stack and (node := stack.pop()):
            [inp.update({kid: -1}) for kid in graph[node]]
            stack += [kid for kid in graph[node] if inp[kid] <= 0]  # добавим в стек вершины с нулями (если они образовались на текущем цикле)
            yield node


    def topo_sort(self, graph):
        """ O(E*V)TS """
        inp = collections.Counter({k: 0 for k in graph})
        [inp.update({kid: 1}) for node in graph for kid in graph[node]]  # сделаем матрицу количества входящих ребер

        topo = [k for k, v in inp.items() if v <= 0]
        for node in topo:  # массив мутирует во время обхода
            [inp.update({kid: -1}) for kid in graph[node]]
            topo += [kid for kid in graph[node] if inp[kid] <= 0]
        return topo  # цикл если len(topo)!=len(graph)



    def topo_sort(self, graph):
        """ O(E*V)TS """
        inp = collections.Counter({k: 0 for k in graph})
        [inp.update({kid: 1}) for node in graph for kid in graph[node]]   # сделаем матрицу количества входящих ребер

        topo = [k for k, v in inp.items() if v <= 0]
        for node in topo: # массив мутирует во время обхода
            for kid in graph[node]:
                inp[kid] -= 1
                inp[kid] <= 0 and topo.append(kid)
        return topo # цикл если len(topo)!=len(graph)
```

- функция update складывает значения каунтеров по одному и тому же ключу
- мутирование массива во время итерации
- update **OR** a==b **AND** append
  - так как update всегда возвращает None то выполнится следующий оператор
  - если условие следующего оператора TRUE то выполнится последний оператор

```
    def topo_sort(self, graph):
        """ O(E*V)TS """
        inp = collections.Counter({k: 0 for k in graph})
        [inp.update({kid: 1}) for node in graph for kid in graph[node]]  # сделаем матрицу количества входящих ребер

        topo = [k for k, v in inp.items() if v <= 0]
        [inp.update({kid: -1}) or inp[kid] == 0 and topo.append(kid) for node in topo for kid in graph[node]]
        return topo  # цикл если len(topo)!=len(graph)



g1 = {4: [], 3: [4], 2: [4], 1: [2, 3]}
g2 = {4: [3], 3: [1], 2: [4], 1: [2]}
g = {4: [3], 3: [2], 2: [4], 1: [2, 3]}
wilmerlibrary.draw_graph_adjacency_dict(g)
# print(Solution().topo_sort(g1))
print(list(Solution().topo_sort(g)))
```

если в алгоритме кана остались непройденные вершины значит существует цикл

Approach 3: Topological Sort

Intuition

Actually, the problem is also known as topological sort problem, which is to find a global order for all nodes in a DAG (*Directed Acyclic Graph*) with regarding to their dependencies.

A linear algorithm was first proposed by Arthur Kahn in 1962, in his paper of "Topological order of large networks". The algorithm returns a topological order if there is any. Here we quote the pseudo code of the Kahn's algorithm from wikipedia as follows:

```
L = Empty list that will contain the sorted elements
S = Set of all nodes with no incoming edge

while S is non-empty do
    remove a node n from S
    add n to tail of L
    for each node m with an edge e from n to m do
        remove edge e from the graph
        if m has no other incoming edges then
            insert m into S

if graph has edges then
    return error   (graph has at least one cycle)
else
    return L   (a topologically sorted order)
```

To better understand the above algorithm, we summarize a few points here:

- In order to find a global order, we can start from those nodes which do not have any prerequisites (i.e. indegree of node is zero), we then incrementally add new nodes to the global order, following the dependencies (edges).
- Once we follow an edge, we then remove it from the graph.
- With the removal of edges, there would more nodes appearing without any prerequisite dependency, in addition to the initial list in the first step.
- The algorithm would terminate when we can no longer remove edges from the graph. There are two possible outcomes:
  - 1). If there are still some edges left in the graph, then these edges must have formed certain cycles, which is similar to the deadlock situation. It is due to these cyclic dependencies that we cannot remove them during the above processes.
  - 2). Otherwise, i.e. we have removed all the edges from the graph, and we got ourselves a topological order of the graph.

Algorithm

Following the above intuition and pseudo code, here we list some sample implementations.

wwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwww

проверить что граф полностью связанный/connected

- можно просто тем что при обычном обходе мы видели все ноды в seen
- The first check is straightforward. If the graph is fully connected, then every node must have been seen. This means that allnodes must be in the seen set at the end. Because a set removes duplicates, and the only values going into it were valid nodenumbers, then we know that the graph was fully connected if, and only if, the seen set contains n values at the end.

Java

```java
// Return true iff the depth first search discovered ALL nodes.
return seen.size() == n;
```

матрица смежности лучше чем список смежности когда число ребер больше чем число нод

- adjacency matrix
- adjacency list
- linked representation
- edges list

- adjacent (immediate neighbours) of a given node

- we'd only use an adjacency matrix if we know that the number of edges is substantially higher than the number of nodes

Before we move onto actually carrying out the depth-first search, let's quickly reassure ourselves that an **adjacency list** was the best graph representation for this problem. The other 2 choices would have been an **adjacency matrix** or a **linked representation**.

- An **adjacency matrix** would be an acceptable, although not ideal, representation for this problem. Often, we'd only use an adjacency matrix if we *know* that the number of edges is substantially higher than the number of nodes. We have no reason to believe that is the case here. Approach 2 will also provide some useful insight into this.
- A **linked representation**, where you make actual nodes objects, would be an overly complicated representation and could suggest to an interviewer that you have a limited understanding of adjacency lists and adjacency matrices. They are not commonly used in interview questions.

создать граф из матрицы смежности (сериализация leetcode)

```python
class Node:

    def __init__(self, val=0, children=None):
        self.val = val
        self.children = children or []

    def __repr__(self):
        return f"{self.val}"


class Solution:


    def make_graph(self, adj_list):
        """ O(N)TS"""
        graph = {i: Node(i) for i in range(1, len(adj_list) + 1)}  # 1-indexed
        for idx, vertex in enumerate(adj_list, start=1):
            for kid in vertex:
                graph[idx].children += graph[kid],
        return graph[1]

каждый элемент -это нода и ее дети

t1 = Solution().make_graph([[2, 4], [1, 3], [2, 4], [1, 3]])
```

вспомогательные функции для вывода матрицы

```python
def print_matrix2(matrix, yt=None, xt=None, y2=None, x2=None):
    # печать матрицы просто как есть
    for y, row in enumerate(matrix):
        for x, val in enumerate(row):
            if (x, y) == (xt, yt):
                print(f"({val})", end="\t")
                continue
            if (x, y) == (x2, y2):
                print(f"[{val}]", end="\t")
                continue
            print(val, end="\t")
        print("\n")
    print("\n")


def print_matrix3(matrix, yt=None, xt=None, y2=None, x2=None):
    max_y, max_x = max(y for (y, x), v in matrix.items()), max(x for (y, x), v in matrix.items())
    for y in range(max_y + 1):
        for x in range(max_x + 1):
            val = matrix.get((y, x))
            if (x, y) == (xt, yt):
                print(f"({val})", end="\t")
                continue
            if (x, y) == (x2, y2):
                print(f"[{val}]", end="\t")
```

```
            continue
        print(val, end="\t")
    print("\n")
print("\n")
```

создать граф из списка ребер

```
class Node:

    def __init__(self, val=0, children=None):
        self.val = val
        self.children = children or []

    def __repr__(self):
        return f"{self.val}"


class Solution:

    def make_graph(self, edges):
        """ O(N)TS"""
        graph = collections.defaultdict(Node)  # 1-indexed
        for vert_from, vert_to in edges:
            graph[vert_from].children += graph[vert_to],
        return graph[1]


gl = (Solution().make_graph(edges=[[1, 2], [5, 1], [1, 3], [1, 4]]))
wilmerlibrary.draw_graph_nodetree(gl)
```

как создать матрицу смежности из списка ребер (для НЕнаправленного графа)

```
graph = collections.defaultdict(list)
[(graph[vert_from].append(vert_to), graph[vert_to].append(vert_from)) for vert_from, vert_to in edges]
```

```
graph = collections.defaultdict(list)
[graph[a].add(b) for a, b in prerequisites]
```

прием когда неохота возвращать список

- тогда добавляю в сам итератор чтобы не было ошибки итерации по None

```
def fn(self, graph, node, seen=set()):
    if node not in seen:
        seen |= {node}
        return [node] + [arr for kid in graph[node] for arr in self.fn(graph, kid, seen) or []]
```

как избавится от лишних проверок на None

нужно добавить в словарь дефолтный None

```
def cloneGraph(self, node: 'Node', seen={}) -> 'Node':
    """ O(N)TS """
    if node and node not in seen: # Node-это объект поэтому "is not None" не требуется
        seen[node] = Node(node.val)
        seen[node].neighbors = [self.cloneGraph(kid, seen) for kid in node.neighbors]
    return seen.get(node)

def cloneGraph(self, node: 'Node', seen={None:None}) -> 'Node': # как избавится от лишних проверок на None, (нужно добавить в словарь что None уже видели)
    """ O(N)TS """
    if node not in seen: # Node-это объект поэтому "is not None" не требуется
        seen[node] = Node(node.val)
        seen[node].neighbors = [self.cloneGraph(kid, seen) for kid in node.neighbors]
    return seen[node]
```

интересный способ положить в стек только если нету в seen

```
def shortestDistance(self, maze: List[List[int]], start: List[int], destination: List[int]) -> int:
    """ O(N*M*logMN)T O(M*N)S """
    grid_src = {(y, x): 1 - val for y, row in enumerate(maze) for x, val in enumerate(row)}
    heap, grid = [(0, *tuple(start))], grid_src.copy()
    while heap:
        dist, y, x = heapq.heappop(heap)
        if [y, x] == destination:
            return dist
        if grid.pop((y, x), False):
            for dy, dx in ((1, 0), (-1, 0), (0, 1), (0, -1)):
                y_, x_, gap = y, x, dist
                while grid_src.get((y_ + dy, x_ + dx)):
                    y_, x_, gap = y_ + dy, x_ + dx, gap + 1
                grid.get((y_, x_)) and heapq.heappush(heap, (gap, y_, x_)) # optimization
```

как определить есть ли путь между двумя вершинами

```
def validPath(self, n: int, edges: List[List[int]], source: int, destination: int) -> bool:
    """ O(N)TS """
    graph = collections.defaultdict(list)
    [(graph[vert_from].append(vert_to), graph[vert_to].append(vert_from)) for vert_from, vert_to in edges]

    def fn(gr, node, seen):
        return max((fn(graph, seen.add(kid) or kid, seen) for kid in gr[node] if kid not in seen), default=0) if node != destination else 1

    return fn(graph, source, set({source}))
```

так как граф может разветвятся то счетчик лучше обернуть в объект (аналогично seen)

```
def canVisitAllRooms1(self, rooms: List[List[int]], node=0, seen=None, cnt=None) -> bool:
    """ O(E+V)TS """
    seen, cnt = seen or set({0}), cnt or {0: 0}
    cnt[0] += 1
    return int(cnt[0] == len(rooms)) or max((self.canVisitAllRooms(rooms, seen.add(kid) or kid, seen, cnt) for kid in rooms[node] if kid not in seen), default=0)
```

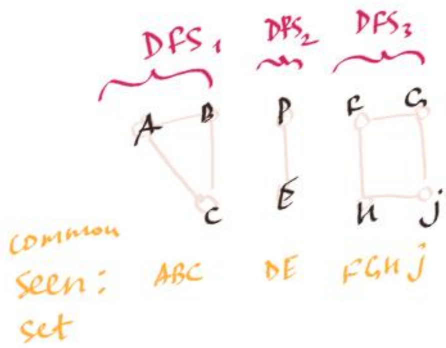
более короткая проверка цикла

```
def fn(graph: List):
    """ O(N)TS """
    stack, seen = [0], {0}  # словарь вместо множества только для того тчобы использовать функцию setdefault
    while (node := stack.pop()) is not None if stack else None:  # is None нужно сделать явную проверку, так как номер первой вершины может быть 0
        yield node  # main work
        stack += [(seen.add(kid) or kid) for kid in graph[node] if kid not in seen]  # :-1 чтобы первый ребенок выбрали первым
```

```python
def fn(graph: List):
    """ O(N)TS """
    stack, seen = [0], {0}  # словарь вместо множества только для того тчобы использовать функцию setdefault
    while stack and (node := stack.pop()) is not None:  # is None нужно сделать явную проверку, так как номер первой вершины может быть 0
        yield node  # main work
        stack += [(seen.add(kid) or kid) for kid in graph[node] if kid not in seen]  # :-1 чтобы первый ребенок выбрали первым
```

- компоненты связности ненаправленного графа (просто нужно сделать несколько DFS)

- In an undirected graph, a connected component is a subgraph in which each pair of vertices is connected via a path. So essent ially, all vertices in a connected component are reachable from one another.

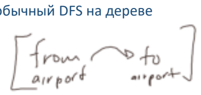


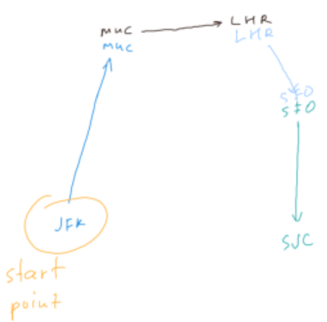
общий seen на все вызовы DFS

```python
    def countComponents(self, n: int, edges: List[List[int]]) -> int:
        """ O(E+V)TS """
        grid, seen = {k: [] for k in range(n)}, set()  # это нужно на случай отдельно стоящих нод
        [(grid[a].append(b), grid[b].append(a)) for a, b in edges]

        def fn(node):
            [fn(seen.add(kid) or kid) for kid in grid[node] if kid not in seen]

        seen = set()
        return sum(fn(seen.add(node) or node) or 1 for node in grid if node not in seen)


# print(Solution().countComponents(n=5, edges=[[0, 1], [1, 2], [3, 4]]))
print(Solution().countComponents(4, [[2, 3], [1, 2], [1, 3]]))
```

интересная задача Эйлерова пути

O(N)TS обычный DFS на дереве





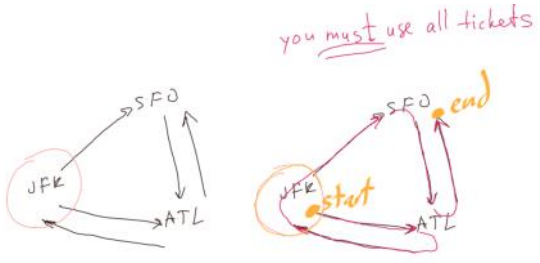

```python
    def findItinerary(self, tickets: List[List[str]]) -> List[str]:  # pylint: disable=invalid-name
        """ O(N)T O(N)S """
        START = "JFK"

        # (1) reconstruct graph
        graph = {}
        for start, end in tickets:
            graph[start] = end

        # (2) DFS graph traversal
        stack = [START]
        while node := stack.pop() if stack else None:
            yield node

            if graph.get(node):
                stack += [graph[node]]
```

O(N)TS обычный DFS на графе (с учетом посещенных вершин)

```python
    def findItinerary(self, tickets: List[List[str]]) -> List[str]:  # pylint: disable=invalid-name
        """ O(N)T O(N)S """
        START = "JFK"

        # (1) reconstruct graph
        graph = {n: [] for n in itertools.chain.from_iterable(tickets)}  # get all nodes from graph
```

```
        for start, end in tickets:
            graph[start] += [end]

        # (2) DFS graph traversal
        stack, seen = [START], {START}
        while node := stack.pop() if stack else None:
            yield node

            stack += [child for child in graph[node] if child not in seen]
            seen |= set(graph[node])
```

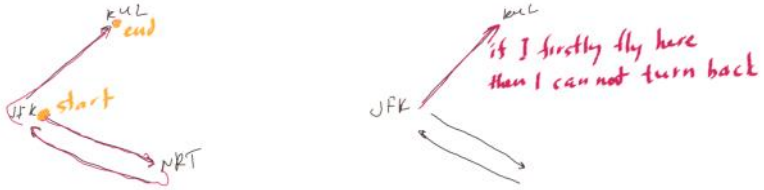
O(N)TS DFS но с учетом пройденных ребер (а не вершин)




```
    def findItinerary(self, tickets: List[List[str]]) -> List[str]:  # pylint: disable=invalid-name
        """ O(N)T O(N)S """
        START = "JFK"

        # (1) reconstruct graph
        graph = {n: [] for n in itertools.chain.from_iterable(tickets)}  # get all nodes from graph
        for start, end in tickets:
            graph[start] += [end]

        # (2) DFS graph traversal
        result, seen = [], set()

        def helper(node, parent):
            nonlocal seen, result
            if not node or (parent, node) in seen:
                return
            result += [node]
            seen |= {(parent, node)}
            for child in sorted(graph[node]): # sorted because condition
                if (node, child) not in seen:
                    helper(child, node)

        helper(START, None)
        return result
```

O(N)TS DFS но с учетом зависимостей (топологическое упорядочивание)

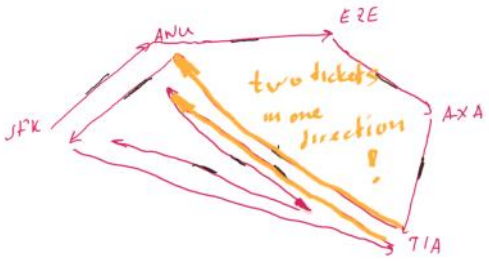
- но с учетом пройденных ребер (а не вершин)
- в отличие от топологической сортировки, здесь topo_sorted_arr могут быть вершины повторно
- в отличие от топологической сортировки, я знаю стартовую вершину, поэтому ненадо их перебирать




```
    def findItinerary(self, tickets: List[List[str]]) -> List[str]:  # pylint: disable=invalid-name
        """ O(N)T O(N)S """
        START = "JFK"

        # (1) reconstruct graph
        graph = {n: [] for n in itertools.chain.from_iterable(tickets)}  # get all nodes from graph
        for start, end in tickets:
            graph[start] += [end]

        # (2) DFS graph traversal
        seen, topo = set(), []  # topo can be any size, diff from toposort

        def helper(node, parent):
            nonlocal seen, topo
            if node is None or (parent, node) in seen:  # is None нужно сделать явную проверку, так как номер первой вершины может быть 0
                return
            seen |= {(parent, node)}

            for child in sorted(graph[node]):  # can add: if child not in seen
                if (node, child) not in seen:
                    helper(child, node)

            topo += [node]  # присваиваю уже на выходе из рекурсии (те сначала конечную вершину)

        helper(START, None)
        return topo[::-1]
```

O(N)TS DFS считаем даже одинаковые ребера

Те ввожу глобальный счетчик всех билетов и из него удаляю израсходованные



промежуточный вариант

```
    def findItinerary(self, tickets: List[List[str]]) -> List[str]:  # pylint: disable=invalid-name
        """ O(N)T O(N)S """
        START = "JFK"

        # (1) reconstruct graph
        graph = {n: {} for n in itertools.chain.from_iterable(tickets)}  # get all nodes from graph
        for start, end in tickets:
            if graph[start].get(end):
                graph[start][end] += 1
            else:
                graph[start][end] = 1

        print(graph)
        # (2) DFS graph traversal
        topo = []  # topo can be any size, diff from toposort

        def helper(node, parent):
            nonlocal topo
            if node is None:  # is None нужно сделать явную проверку, так как номер первой вершины может быть 0
                return
```

```
            if parent and graph[parent].get(node) < 1:
                return
            if parent:
                graph[parent][node] -= 1

            for child in sorted(graph[node].keys()):  # can add: if child not in seen
                helper(child, node)

            topo += [node]  # присваиваю уже на выходе из рекурсии (те сначала конечную вершину)

        helper(START, None)
        return topo[::-1]
```

```
    def findItinerary(self, tickets: List[List[str]]) -> List[str]:  # pylint: disable=invalid-name
        """ O(N)T O(N)S """
        START = "JFK"

        # (1) reconstruct graph
        graph = {n: [] for n in itertools.chain.from_iterable(tickets)}  # get all nodes from graph
        for start, end in tickets:
            graph[start] += [end]

        # (2) DFS graph traversal
        counter, topo = collections.Counter([(a, b) for a, b in tickets]), []

        def helper(node, parent):
            nonlocal counter, topo
            if node is None or (parent and counter[(parent, node)] < 1):  # analogously if seen
                return

            counter[(parent, node)] -= 1  # throwing away one ticket

            for child in sorted(graph[node]):
                helper(child, node)

            topo += [node]  # присваиваю уже на выходе из рекурсии (те сначала конечную вершину)

        helper(START, None)
        return topo[::-1]
```

решение 2021 (обычный topoSort через DFS но с видоизмененным seen)

- всегда начинается с аэропорта JFK
- билетов в одном направлении может быть несколько
- нужно в seen заносить не вершины, а ребра (parent,node)
- этот алгорим работает из за того что DFS должен сначала дойти до конца
- также очень важно условие что путь существует (так как иначе он просто начнет добавлять в маршрут оставшиеся ребра, из которых нельзя вернуться назад)

```
class Solution:

    def findItinerary(self, tickets: List[List[str]]) -> List[str]:
        """ O(E logE/V )T O(E+V)S """

        def fn(node, parent):
            if (parent, node) in lost:
                lost.update(((parent, node): -1)) or not lost.get((parent, node)) and lost.pop((parent, node))
                return [arr for kid in sorted(grid[node]) if (node, kid) in lost for arr in fn(kid, node)] + [node]

        lost, grid = collections.Counter(map(tuple, tickets + [(None, 'JFK')])), {n: [] for n in itertools.chain.from_iterable(tickets)}
        [grid[a].append(b) for a, b in tickets]
        return fn("JFK", None)[::-1]


# print(Solution().findItinerary(tickets=[["MUC", "LHR"], ["JFK", "MUC"], ["SFO", "SJC"], ["LHR", "SFO"]]))
    # ["JFK","MUC","LHR","SFO","SJC"]
print(Solution().findItinerary(tickets=[["JFK", "SFO"], ["JFK", "ATL"], ["SFO", "ATL"], ["ATL", "JFK"], ["ATL", "SFO"]]))
    # ["JFK","ATL","JFK","SFO","ATL","SFO"]
print(Solution().findItinerary([["JFK", "KUL"], ["JFK", "NRT"], ["NRT", "JFK"]]))
    # ["JFK","NRT","JFK","KUL"]
```

итоговый вариант

Eulerian trail (or Eulerian path)

- In graph theory, an Eulerian trail (or Eulerian path) is a trail in a finite graph that visits every edge exactly once (allowing for revisiting vertices).
- Eulerian circuit or Eulerian cycle is an Eulerian trail that starts and ends on the same vertex.

O(E)T Hierholzer алгоритм нахождения эйлерова пути

- мы пройдем все ребра один раз а некоторые ноды дважды
- мы останавливаемся когда дойдем до вершины у которой все ребра уже разведаны (даже не заходим на эту вершину а останавливаемся заранее)
  - мы можем хранить список неразведанных ребер
  - либо мы можем удалять уже разведанные ребра на исходном графе

O(E logE/V)T

- O(E)T потому что мы проходим через все ребра
- O(logE/V) потому что в каждой ноде нам надо сортировать ребра (по условию задачи об аэропорте)

O(E+V)S потому что мы храним граф как вершины и ребра

- Time complexity: $O(|E| \log \frac{|E|}{|V|})$ where $|E|$ is the number of edges (flights) in the input.

  As one can see from the above algorithm, during the DFS process, we would traverse each edge once. Therefore, the complexity of the DFS function would be $|E|$.

  However, before the DFS, we need to sort the outgoing edges for each vertex. And this, unfortunately, dominates the overall complexity.

  It is though tricky to estimate the complexity of sorting, which depends on the structure of the input graph.

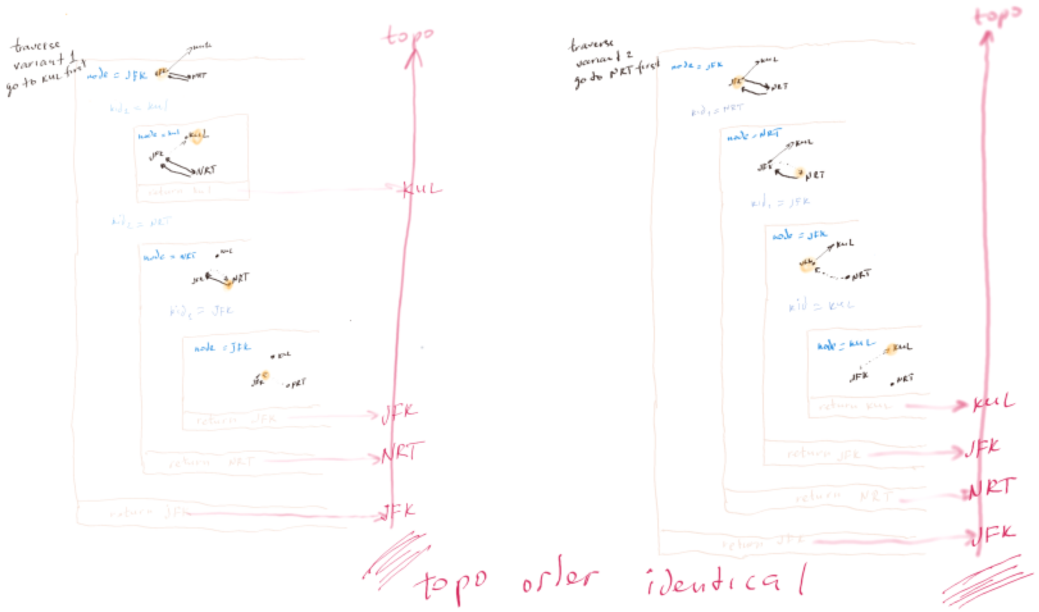
  In the worst case where the graph is not balanced, i.e. the connections are concentrated in a single airport. Imagine the graph is of star shape. In this case, the #K airport would assume half of the flights (since we still need the return flight). As a result, the sorting operation on this airport would be exceptionally expensive, i.e. $N \log N$, where $N = \frac{|E|}{2}$, and this would be the final complexity as well, since it dominates the rest of the calculation.

  Let us consider a less bad case, or an average case, where the graph is less cluttered, i.e. each node has the equal number of outgoing flights. Under this assumption, each airport would have $\frac{|E|}{|V|}$ number of flights (still we need the return flights). Again, we can plug it into the $N \log N$ minimal sorting complexity, in addition, this time, we need to take into consideration all airports, rather than the superficial (#K) in the above case. As a result, we have $|V| \cdot (N \log N)$, where $N = \frac{|E|}{|V|}$. If we expand the formula, we will obtain the complexity of the average case as $O(|V| \frac{|E|}{|V|} \log \frac{|E|}{|V|}) = O(|E| \log \frac{|E|}{|V|})$.

- Space complexity: $O(|V| + |E|)$ where $|V|$ is the number of airports and $|E|$ is the number of flights.

  In the algorithm, we use the graph, which would require the space of $|V| + |E|$.

- Time Complexity: $\mathcal{O}(|E|\log\frac{|E|}{|V|})$ where $|E|$ is the number of edges (flights) in the input.
  - As one can see from the above algorithm, during the DFS process, we would traverse each edge once. Therefore, the complexity of the DFS function would be $|E|$.
  - However, before the DFS, we need to sort the outgoing edges for each vertex. And this, unfortunately, dominates the overall complexity.
  - It is though tricky to estimate the complexity of sorting, which depends on the structure of the input graph.
  - In the worst case where the graph is not balanced, i.e. the connections are concentered in a single airport. Imagine the graph is of star shape, in this case, the JFK airport would assume half of the flights (since we still need the return flight). As a result, the sorting operation on this airport would be exceptionally expensive, i.e. $N\log N$, where $N = \frac{|E|}{2}$. And this would be the final complexity as well, since it dominates the rest of the calculation.
  - Let us consider a less bad case, or an average case, where the graph is less clustered, i.e. each node has the equal number of outgoing flights. Under this assumption, each airport would have $\frac{|E|}{2\cdot|V|}$ number of flights (still we need the return flights). Again, we can plug it into the $N\log N$ minimal sorting complexity. In addition, this time, we need to take into consideration all airports, rather than the superhub (JFK) in the above case. As a result, we have $|V|\cdot(N\log N)$, where $N = \frac{|E|}{2\cdot|V|}$. If we expand the formula, we will obtain the complexity of the average case as $\mathcal{O}(\frac{|E|}{2}\log\frac{|E|}{2\cdot|V|}) = \mathcal{O}(|E|\log\frac{|E|}{|V|})$
- Space Complexity: $\mathcal{O}(|V| + |E|)$ where $|V|$ is the number of airports and $|E|$ is the number of flights.
  - In the algorithm, we use the graph, which would require the space of $|V| + |E|$.
  - Since we applied recursion in the algorithm, which would incur additional memory consumption in the function call stack. The maximum depth of the recursion would be exactly the number of flights in the input, i.e. $|E|$.
  - As a result, the total space complexity of the algorithm would be $\mathcal{O}(|V| + 2\cdot|E|) = \mathcal{O}(|V| + |E|)$.



~~вариант с множественными вершинами в стеке непрокатил~~

```python
    def findItinerary_BAD(self, tickets: List[List[str]]) -> List[str]:
        """ O(E logE/V )T O(E+V)S """
        grid = collections.defaultdict(list)
        [grid[a].append(b) for a, b in sorted(tickets)]  # сортируем аэропорты только в одном месте

        def fn(node, parent=None):
            (g := grid.get(parent)) and g.remove(node)
            return [arr for kid in grid[node] for arr in fn(kid, node)] + [node]

        return fn('JFK', None)[::-1]

    def findItinerary_BAD(self, tickets: List[List[str]]) -> List[str]:
        """ O(E logE/V )T O(E+V)S """
        grid = collections.defaultdict(list)
        [grid[a].append(b) for a, b in sorted(tickets)[::-1]]  # сортируем аэропорты только в одном месте

        def fn(node, parent=None):
            return [arr for kid in grid[node] for arr in fn(grid[node].remove(kid) or kid, node)] + [node]  # при удалении во время итерации сбиваются выбаваемые элементы

        return fn('JFK', None)[::-1]
```

вариант с обычным DFS при добавлении сразу всех детей в seen (через удаление ребер графа)

```python
    def findItinerary1(self, tickets: List[List[str]]) -> List[str]:
        """ O(E logE/V )T O(E+V)S """
        grid = collections.defaultdict(list)
        [grid[a].append(b) for a, b in sorted(tickets)[::-1]]  # сортируем аэропорты только в одном месте

        def fn(node, parent=None):
            res = []
            while grid[node] and (kid := grid[node].pop()):
                res += fn(kid, node)
            return res + [node]  # при удалении во время итерации сбиваются выбаваемые элементы

        return fn('JFK', None)[::-1]

    def findItinerary(self, tickets: List[List[str]]) -> List[str]:
        """ O(E logE/V )T O(E+V)S """
        grid = collections.defaultdict(list)
        [grid[a].append(b) for a, b in sorted(tickets)[::-1]]  # сортируем аэропорты только в одном месте

        def fn(node, parent, res):
            while grid[node]:
                fn(grid[node].pop(), node, res)
            res.append(node)  # при удалении во время итерации сбиваются выбаваемые элементы

        return fn('JFK', None, res := []) or res[::-1]


#print(Solution().findItinerary(tickets=[["MUC", "LHR"], ["JFK", "MUC"], ["SFO", "SJC"], ["LHR", "SFO"]]))
#print(Solution().findItinerary(tickets=[["JFK", "SFO"], ["JFK", "ATL"], ["SFO", "ATL"], ["ATL", "JFK"], ["ATL", "SFO"]]))
print(Solution().findItinerary([["JFK", "KUL"], ["JFK", "NRT"], ["NRT", "JFK"]]))  # ["JFK","NRT","JFK","KUL"]
```

итеративная версия post-order DFS

```python
    def findItinerary(self, tickets: List[List[str]]) -> List[str]:
        """ O(E logE/V )T O(E+V)S """
        grid, stack, result = collections.defaultdict(list), ['JFK'], []
        [grid[a].append(b) for a, b in sorted(tickets)[::-1]]  # сортируем аэропорты только в одном месте

        while stack:
            while grid[node := stack[-1]]:
                stack += grid[node].pop(),
            result += stack.pop(),  # post-order: pop after kids

        return result[::-1]
```

wwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwww

когда не seen (множество уже виденных нод)
a stay (множество оставшихся для обхода нод)

- НЕНУЖНО ДЕЛАТЬ is not None потому что в stay зашиты одни еденички
- используя красивый .pop() можно за один шаг проверить что нужно итерировать вершину и исключить из дальнейшей итерации

```
class Solution:

    # DFS
    def traversal1(self, graph) -> int:
        def fn(node, stay):
            print(node)
            [fn(kid, stay) for kid in graph[node] if stay.pop(kid, False)]

        fn(0, collections.Counter(graph.keys() - {0}))  # Counter(range(3))

    def traversal2(self, graph) -> int:
        def fn(node, stay):
            return [node] + [arr for kid in graph[node] if stay.pop(kid, False) for arr in fn(kid, stay)]

        return fn(0, collections.Counter(graph.keys() - {0}))

    def traversal(self, graph):
        stack, stay = [0], collections.Counter(graph.keys() - {0})
        while stack and (node := stack.pop()) is not None:
            yield node
            stack += [kid for kid in graph[node] if stay.pop(kid, False)]

    # BFS
    def traversal4(self, graph):
        level, stay = [0], collections.Counter(graph.keys() - {0})
        while level:
            yield level
            level = [kid for node in level for kid in graph[node] if stay.pop(kid, False)]

    def traversal5(self, graph) -> int:
        def fn(level, stay):
            return [level] + fn([kid for node in level for kid in graph[node] if stay.pop(kid, False)], stay) if level else []

        return fn([0], collections.Counter(graph.keys() - {0}))

    def traversal6(self, graph) -> int:
        queue, stay = collections.deque([0]), collections.Counter(graph.keys() - {0})
        while queue and (node := queue.popleft()) is not None:
            yield node
            queue += [kid for kid in graph[node] if stay.pop(kid, False)]

    # post order
    def traversal7(self, graph, start=0):
        stack, stay = [(start, False)], collections.Counter(graph.keys() - {start})
        while stack:
            node, again = stack.pop()
            if not again:
                stack += [(node, True)] + [(kid, False) for kid in graph[node] if stay.pop(kid, False)]
            else:
                yield node

    def traversal8(self, grid, start=0):
        stack, stay = [start], collections.Counter(grid.keys())
        while stack:
            while grid[node := stack[-1]]:
                stack += grid[node].pop(),
            if stay.pop(res := stack.pop(), False):
                yield res

    # с повторными вершинами в стеке
    def traversal9(self, graph):
        stack, stay = [0], collections.Counter(graph.keys())
        while stack:
            if stay.pop(node := stack.pop(), False):
                yield node
                stack += [kid for kid in reversed(graph[node]) if kid in stay]

    def traversal10(self, graph):
        def fn(node, stay):
            if stay.pop(node, False):  # НЕНУЖНО ДЕЛАТЬ is not NOne потому что в stay зашиты одни еденички
                print(node)
                [fn(kid, stay) for kid in graph[node]]

        return fn(0, collections.Counter(graph.keys()))

    def traversal11(self, graph):
        def fn(node, stay):
            if stay.pop(node, False):  # НЕНУЖНО ДЕЛАТЬ is not NOne потому что в stay зашиты одни еденички
                print(node)
                return [node] + [arr for kid in reversed(graph[node]) if kid in stay for arr in fn(kid, stay)]

        return fn(0, collections.Counter(graph.keys()))

    # дейкстра
    def dijkstra(self, graph, start):  # weighted graph
        heap, stay = [(0,start)], collections.Counter(graph.keys())
        while heap:
            length ,node= heapq.heappop(heap)
            if stay.pop(node, False):
                yield node
                [heapq.heappush(heap, (length + gap,kid)) for kid, gap in graph[node].items() if kid in stay]

    # topo
    def topo_sort(self, graph: List) -> List[int]:
        def fn(node):
            return [arr for kid in graph[node] if stay.pop(kid, False) for arr in fn(kid)] + [node] if node is not None else []  # post-order

        stay = collections.Counter(graph.keys())
        return list(itertools.chain.from_iterable(fn(node) for node in graph if stay.pop(node, False)))[::-1]


# grid = [[1, 0, 1], [1, 1, 0], [0, 0, 0]]
# print_matrix(grid)
# print(Solution().traversal(grid))

adjacency_matrix = {0: [1, 2], 1: [2, 3], 2: [3], 3: []}
print(list(Solution().traversal(adjacency_matrix)))  # 0 1 2 3  # graph_weighed = {'a': {'b': 1, 'c': 4}, 'b': {'d': 6, 'c': 2}, 'c': {'d': 3}, 'd': {}}  # print(list(Solution().dijkstra(graph_weighed, 'a')))  # a b c d
```

wwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwww

O(N)TS Disjoint Set Union (DSU) data structure

Union-Find - островки, в каждом острове все вершины не образуют цикла

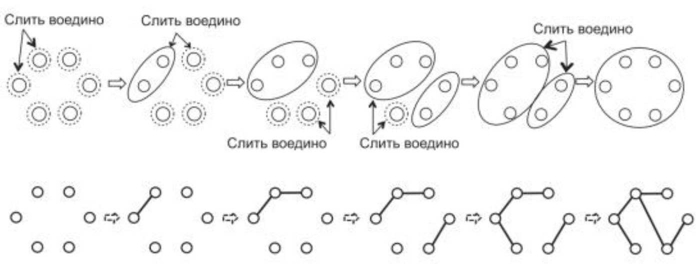
- если вдруг я захочу добавить ребро между любыми двумя вершинами в одном острове (те эти вершины уже были в этом острове) то значит будет цикл

Notes on page "MST minimum spanning tree / минимальное остовное дерево графа 1)Prim's HEAP 2)Kruskal's UnionFind DisjointSet"

Clustering (Kruskal's)

https://en.wikipedia.org/wiki/Disjoint-set_data_structure

MST minimum spanning tree / минимальное остовное дерево графа
1)Prim's HEAP
2)Kruskal's UnionFind DisjointSet



Complexity Analysis

- Time Complexity: $O(N\alpha(N)) \approx O(N)$, where N is the number of vertices (and also the number of edges) in the graph, and α is the Inverse-Ackermann function. We make up to N queries of `dsu.union`, which takes (amortized) $O(\alpha(N))$ time. Outside the scope of this article, it can be shown why `dsu.union` has $O(\alpha(N))$ complexity, what the Inverse-Ackermann function is, and why $O(\alpha(N))$ is approximately $O(1)$.
- Space Complexity: $O(N)$. The current construction of the graph (embedded in our `dsu` structure) has at most N nodes.

Report Article Issue

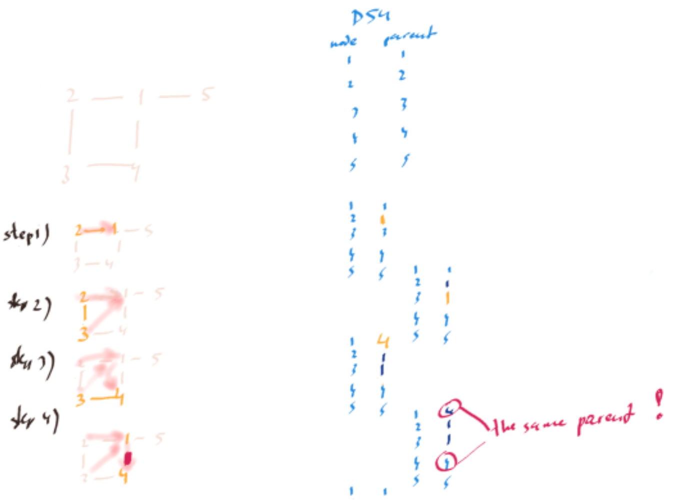
две операции: find и union

- `dsu.find(node x)`, which outputs a unique id so that two nodes have the same id if and only if they are in the same connected component, and:
- `dsu.union(node x, node y)`, which draws an edge `(x, y)` in the graph, connecting the components with id `find(x)` and `find(y)` together.

Python

```
# parent initialized as (x -> x)
function find(x):
    while parent[x] != x: #While x isn't the leader
        x = parent[x]
    return x

function union(x, y):
    parent[find(x)] = find(y)
```

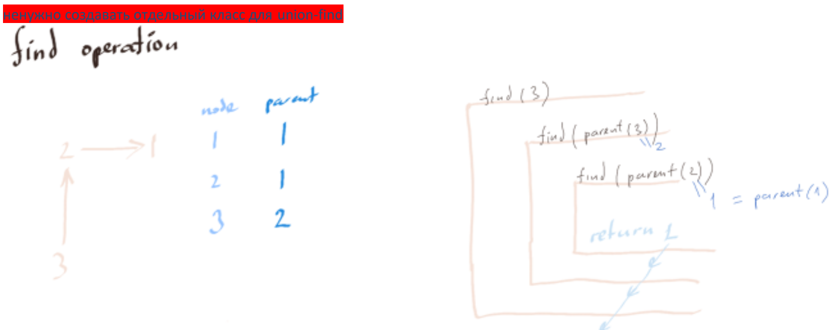
```python
class DSU:
    def __init__(self):
        self.par = range(1001)
    def find(self, x):
        if self.par[x] != x:
            self.par[x] = self.find(self.par[x])
        return self.par[x]
    def union(self, x, y):
        self.par[self.find(x)] = self.find(y)
```



c

```python
class DisjointSet:

    def __init__(self):
        # table idx:parent if idx's node
        # массив заполнен по умолчанию индексами те нода==сама себе родитель
        self.parent = list(range(10000))

    def find(self, val):
        if (par := self.parent[val]) != val:
            self.parent[val] = self.find(par)
        return self.parent[val]  # return updated parent

    def union(self, x, y):
        if (a := self.find(x)) != (b := self.find(y)):
            self.parent[a] = b
            return True
        return False


class Solution:

    def findRedundantConnection(self, edges: List[List[int]]) -> List[int]:
        """ O(N)TS """
        dis_set = DisjointSet()
        for i, j in edges:
            if not dis_set.union(i, j):
                return i, j
```

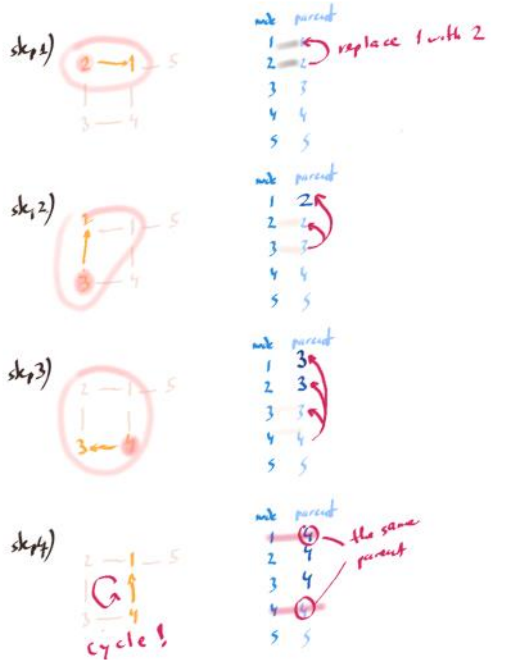
ненужно создавать отдельный класс для union-find




```python
class Solution:

    def findRedundantConnection(self, edges: List[List[int]]) -> List[int]:
        """ O(N)TS """
        parent = list(range(10000))
        find = lambda x: x if parent[x] == x else find(parent[x])
        union = lambda x, y: (x := find(x), y := find(y)) and parent.__setitem__(x, y) or x != y

        return next((i, j) for i, j in edges if not union(i, j))


print(Solution().findRedundantConnection(edges=[[1, 2], [2, 3], [3, 4], [1, 4], [1, 5]]))
```

O(E)T O(V)S представление таблицы парентов в виде строки
облегчает обработку если даны ребра (представление графа в виде ребер)

- таблицу parent-ов представляем в виде строк, тогда в ней будет легко сделать замену сразу нескольких элементов
- в этой таблице у всех связанных элементов сразу пишется номер группы к которой они принадлежат (а не граф который надо разматывать чтобы добраться до парента)
- тогда легче сразу заменить этот номер группы у всех элементов в группе (в обычной версии достаточно заменить родителя в одном месте)




```
class Solution:


    def findRedundantConnection(self, edges):
        """ O(N)TS """
        parent = ''.join(map(chr, range(10000)))
        for i, j in edges:
            if parent[i] == parent[j]:
                return i, j
            parent = parent.replace(parent[i], parent[j])  # все время заменяем строку на новую, а не in place


print(Solution().findRedundantConnection(edges=[[1, 2], [2, 3], [3, 4], [1, 4], [1, 5]]))  # print(Solution().findRedundantConnection(edges=[[1, 2], [1, 3], [2, 3]]))
```

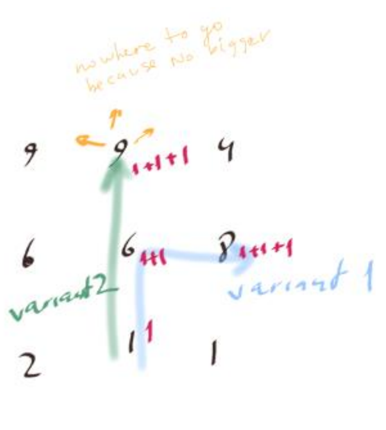
нахождение пути между вершинами

```
def fn(node, target, mult, stay):
    if node == target:
        return mult
    return next(filter(None, (fn(kid, target, mult * grid[node][kid], stay) for kid in grid[node].keys() if stay.pop(kid, False))), None) # если не сделать filter то поймается первое пустое значение и дальше итерация не продолжится
```

wwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwww

если нужно сравнить несколько вариантов то нужно как то без {seen}
так как в условии сказано что мы можем даже повторно проходить по некторым числам, то нам ненужен массив seen
  - варианты: это когда на развилке после прохождения первой ноды мы можем пойти вверх(1-й вариант) или влево например (2-й вариант)
  - если был бы seen то второй вариант не смог бы пройти по уже хоженной ноде (так как в предыдущем варианте мы по этой ноде прошли)

- нету seen но итерация все-равно закончится так как последовательность не может возрастать бесконечно
- словарь grid нужен только чтобы мы не вышли заграницы матрицы




```
def longestIncreasingPath(self, matrix: List[List[int]]) -> int:
```

```python
""" O(N*M)TS """

@functools.cache
def fn(y, x):
    if (val := grid.get((y, x))) is not None:
        return 1 + max([fn(*kid) for kid in ((y + 1, x), (y - 1, x), (y, x + 1), (y, x - 1)) if grid.get(kid, -math.inf) > val], default=0)

grid = {(y, x): val for y, row in enumerate(matrix) for x, val in enumerate(row)}
return max(fn(*pt) or 0 for pt in grid)
```

filter(next) чтобы не выбрать пустые вызовы DFS

```python
def fn(node, target, mult, stay):
    if node == target:
        return mult
    return next(filter(None, (fn(kid, target, mult * grid[node][kid], stay) for kid in grid[node].keys() if stay.pop(kid, False))), None) # если не сделать filter то поймается первое пустое значение и дальше итерация не продолжится
```

как выбрать не только листовые ноды но и все пути к ним

```python
def binaryTreePaths(self, root):
    "tail recursion, None-not allowed"
    return [[root.val] + path for kid in (root.left, root.right) if kid for path in self.binaryTreePaths(kid)] or [[root.val]] ~~if root else [[]]~~

def binaryTreePaths(self, root):
    """ using set , None-allowed"""
    if not root:
        return {()}

    return {(root.val,) + path for path in self.binaryTreePaths(root.left) | self.binaryTreePaths(root.right)} or {(root.val)}
```

```python
    def fn(node): # работает только для ацикличного графа (те дерева)
        return [[node] + path for kid in graph[node] for path in fn(kid)] or [[node]]
    return fn(0)
```

- DFS алгоритмы для деревьев подходят и для DAG (без циклов) так как нам ненужен seen
- backtracking лучше подходит для циклических графов