Задание 1. Пузырьковая сортировка и оценка сложности алгоритмов

Когда мы говорили о сложности пузырьковой сортировки, то намеренно показали наименее оптимальный алгоритм, состоящий из двух вложенных проходов по списку элементов: от первого и до последнего элемента. Алгоритм можно выразить примерно таким кодом:

for i in range(len(data)):  
   for j in range(len(data) - 1):  
       if data[j] < data[j+1]:  
           data[j], data[j+1] = data[j+1], data[j]

Один цикл вложен в другой, во внешнем — n итераций, во внутреннем — n итераций. Получается n раз по n, или n2. Но обратите внимание: после первой итерации самый маленький (лёгкий) элемент всплывает вверх (оказывается в конце списка). На следующей итерации второй по меньшинству элемент тоже всплывёт и станет предпоследним в списке. Получается, что последние (всплывшие) элементы уже отсортированы, а те, что следуют перед ними, — ещё нет.



Другими словами, на каждой итерации внутреннего цикла нужно перебирать не все n элементов, а только ещё не отсортированный «хвост» списка. На первой итерации внутреннего цикла перебираются все элементы, на второй — все, кроме последнего, на третьей — все, кроме последнего и предпоследнего, и так происходит, пока весь список не будет упорядочен. Может показаться, что, действуя таким образом, список можно отсортировать за меньшее количество итераций.

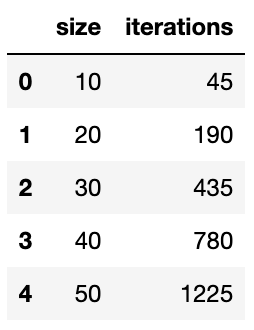
И это действительно так. Однако давайте проведём настоящий научный эксперимент!

**Что нужно сделать**

1. Создайте новый проект в PyCharm и добавьте в него[файл task1.py с кодом](https://drive.google.com/file/d/1mNH7D_VRRvgtiS4oH83T6eGZRWAaa5_u/view?usp=sharing).
2. В месте, помеченном комментарием «Напишите ваш код тут» (это функция bubble\_sort), напишите код пузырьковой сортировки, которая упорядочивает массив data **по убыванию.**
3. Реализуйте экономный вариант сортировки, при которой каждый новый вложенный цикл содержит на одну операцию меньше, чем предыдущий.
4. При этом посчитайте, за какое количество итераций отрабатывает весь алгоритм.
5. Сделайте счётчик итераций возвращаемым значением: вы передаёте в функцию массив, она его сортирует и возвращает количество действий, которое потребовалось, чтобы выполнить алгоритм.

Теперь запустите код. Если вы всё сделали правильно, программа выполнится и напечатает:

OK  
Done!

Кроме того, программа запишет файл bubble.csv, а в него — две колонки:  


Продолжим работу.

1. Запустите Jupyter Notebook и считайте этот файл в Pandas-датафрейме.
2. Добавьте в датафрейм ещё одну колонку, запишите в неё квадраты значений из колонки size (size2).
3. Затем с помощью matplotlib постройте график с двумя линиями: по оси абсцисс — значения size, по оси ординат — значения iterations для первой линии и значения колонки с квадратами size для второй.

Если всё сделано правильно, вы увидите две линии, обе — параболы. Кажется, что та линия, которая находится выше, вот-вот обгонит ту, что ниже. Но это не совсем так. Сейчас значение size варьируется от 0 до 1 000, отрезок небольшой. Но какой бы большой отрезок вы ни взяли — от 0 до 1 000 000, от 0 до 1 000 000 000, от 0 до бесконечности, — графики этих функций будут выглядеть примерно так же. Две линии, одна выше другой. Первая упрямо убегает, вторая не менее упрямо догоняет.

Именно это имеют в виду, когда говорят, что функции имеют один порядок роста. Выходит, вы провели оптимизацию, ускорили алгоритм сортировки, но бесконечности нет до этого дела. Это значит, что при определённом объёме данных (причём не очень большом) оба алгоритма будут работать непозволительно долго. Сложность обоих, несмотря ни на что, — O(n2).

**Дополнительное необязательное задание**

При желании можете вывести аналитическую формулу, которая связывает число итераций оптимизированной пузырьковой сортировки от размера сортируемого массива. Будет здорово, если вы выведете формулу и пришлёте её на проверку.

Подсказка: неоптимизированный алгоритм делает n проходов по n итераций в каждом, поэтому формула выводится легко: n2. Оптимизированный алгоритм тоже делает n проходов, но в каждом следующем проходе число итераций уменьшается на единицу.

**Что оценивается**

* Алгоритм реализован корректно и в соответствии с заданием: число итераций нового алгоритма совпадает со значениями таблицы в описании задания.
* Файл bubble.csv содержит 2 столбца и 101 строку.
* Критерии для необязательного задания:
  + формула выведена корректно: при подстановке в неё чисел из столбца size таблицы bubble.csv получаются значения из столбца iterations;
  + описан алгоритм вывода формулы.

Задание 2. Задача со скобками (стек)

Классическая задача, которая довольно часто встречается на собеседованиях, в том числе на позицию junior-разработчиков и инженеров данных. Дана последовательность скобок трёх типов: круглые, квадратные и фигурные. Скобки (, [, { называются открывающими, скобки ), ], } называются закрывающими. Последовательность считается сбалансированной, если каждой открывающей скобке соответствует закрывающая того же типа. Например, последовательности (), ([]), {}{}()[], {[()]} считаются сбалансированными, а последовательности {), {}[, ((((()))) — нет.

Задача заключается в том, чтобы для данной строки со скобками определить, сбалансирована она или нет. Задача достаточно просто решается с помощью стека.

Алгоритм таков: вы перебираете последовательность посимвольно. Если попадается открывающая скобка — помещаете её на стек. Если попадается закрывающая скобка, то проверяете следующие варианты.

Если стек пуст, последовательность не сбалансирована. Завершаем алгоритм.

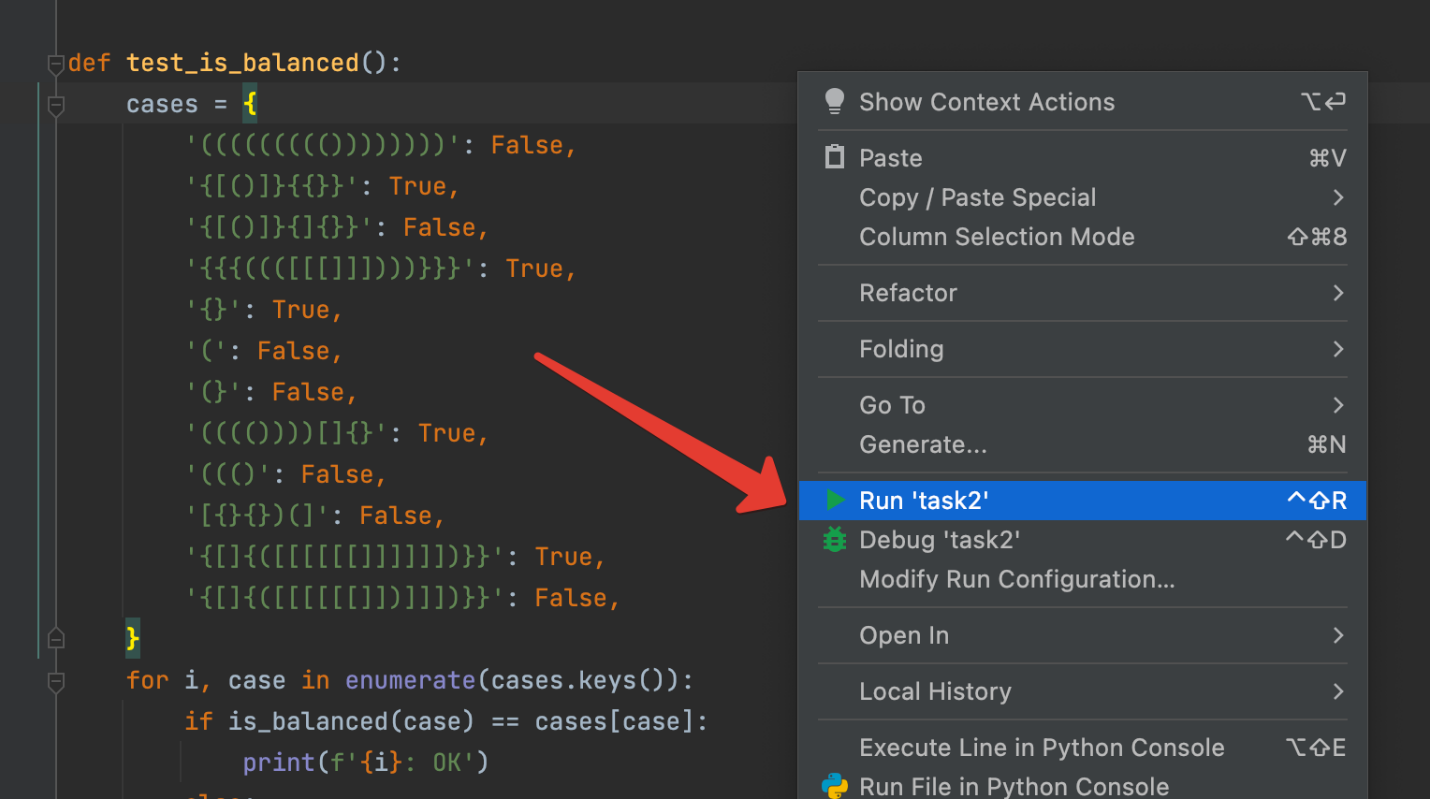
Если стек не пуст, то текущая закрывающая скобка должна быть того же типа, что и открывающая скобка на вершине стека. Другими словами, снимаем открывающую скобку со стека и проверяем, что текущая закрывающая скобка имеет тот же тип. Если это не так, последовательность не сбалансирована. Завершаем алгоритм.

Если вы дошли до самого конца строки, ваш стек должен быть пуст. Вы складывали в него открывающие скобки, потом, когда встречали закрывающую, доставляли скобку из стека и проверяли, что они одного типа. Очевидно, если последовательность сбалансирована, то, когда вы переберёте все скобки, в стеке ничего не останется. Это — заключительное условие сбалансированности. Так что если в конце у вас пустой стек — всё хорошо, но если стек не пуст — беда: последовательность не сбалансирована.

**Что нужно сделать**

1. Скачайте [файл task2.py](https://drive.google.com/file/d/1mAZdH3ncZcUn1Dij-7OWv3XXChh1Ke09/view?usp=sharing) и добавьте его в проект, который создали при выполнении предыдущего задания.
2. В месте, помеченном как «Напишите ваш код тут» (функция is\_balanced), напишите код, который реализует алгоритм (описан выше), проверяющий, сбалансирована последовательность скобок в строке или нет. Функция должна возвращать True, если последовательность сбалансирована, и False в противном случае.
3. Стек реализуйте с помощью списков Python. Как это сделать, мы рассказывали на занятиях.

Когда напишете код, запустите программу на выполнение. Запустить в PyCharm отдельный файл (а не весь проект) можно так: кликните правой кнопкой мыши в любом месте редактора кода, в появившемся всплывающем меню выберите пункт Run task2. Файл запустится.



Если всё сделано правильно, ваш код выполнится, и программа напечатает такой вывод:

0: OK  
1: OK  
2: OK  
3: OK  
4: OK  
5: OK  
6: OK  
7: OK  
8: OK  
9: OK  
10: OK  
11: OK

Получили такой вывод? Отправляйте код на проверку.

**Подсказка**

* Чтобы быстро найти открывающую скобку того же типа, что и некоторая заданная закрывающая скобка (пригодится, когда будете сравнивать текущую закрывающую скобку со скобкой на вершине стека), удобно использовать вот такой словарь:

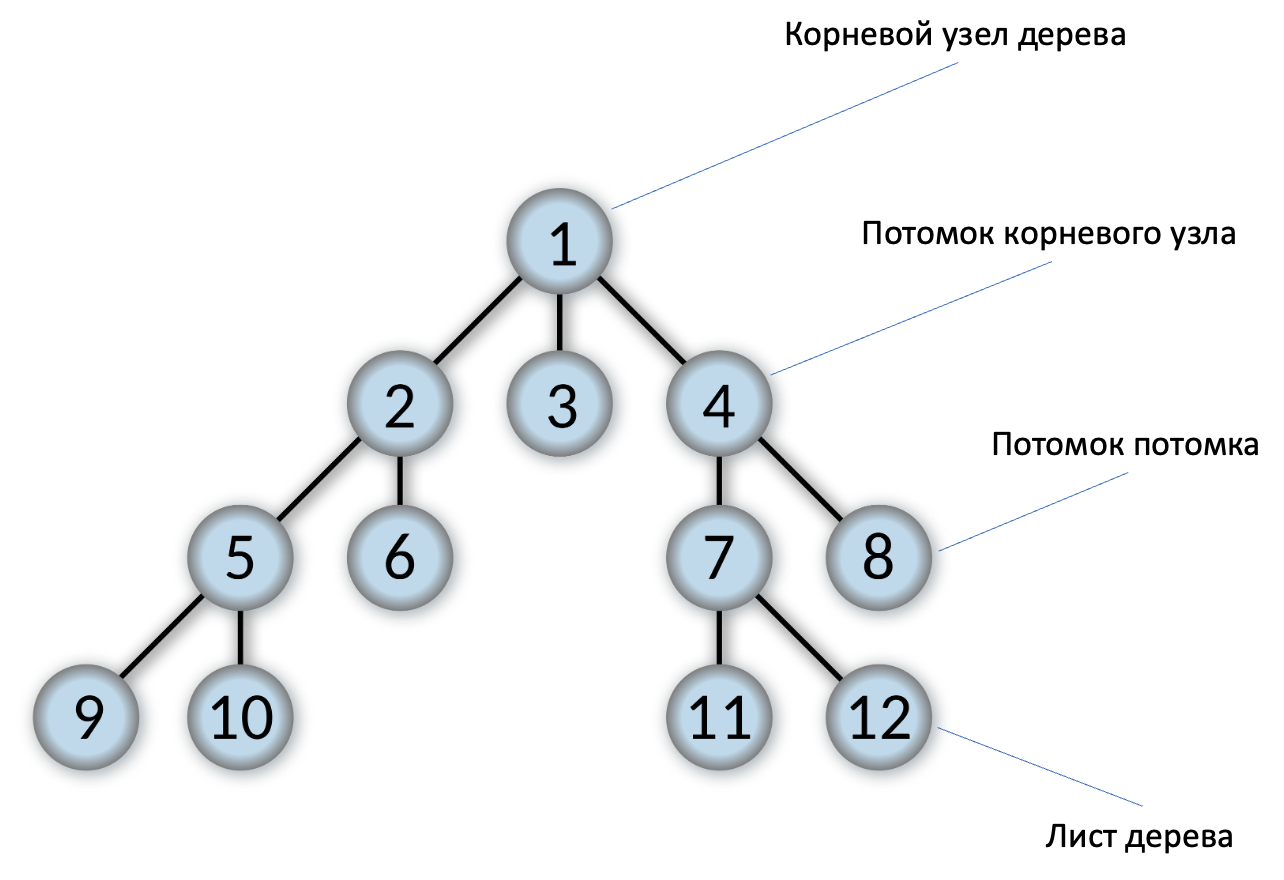
pairs = {  
        ')': '(',  
        ']': '[',  
        '}': '{'  
    }

**Что оценивается**

* Алгоритм реализован корректно и в соответствии с заданием: функция возвращает True, если последовательность сбалансирована, и False в противном случае.
* При запуске программы вы получаете 11 строк со словом ОК.

Задание 3. Поиск в ширину (очереди)

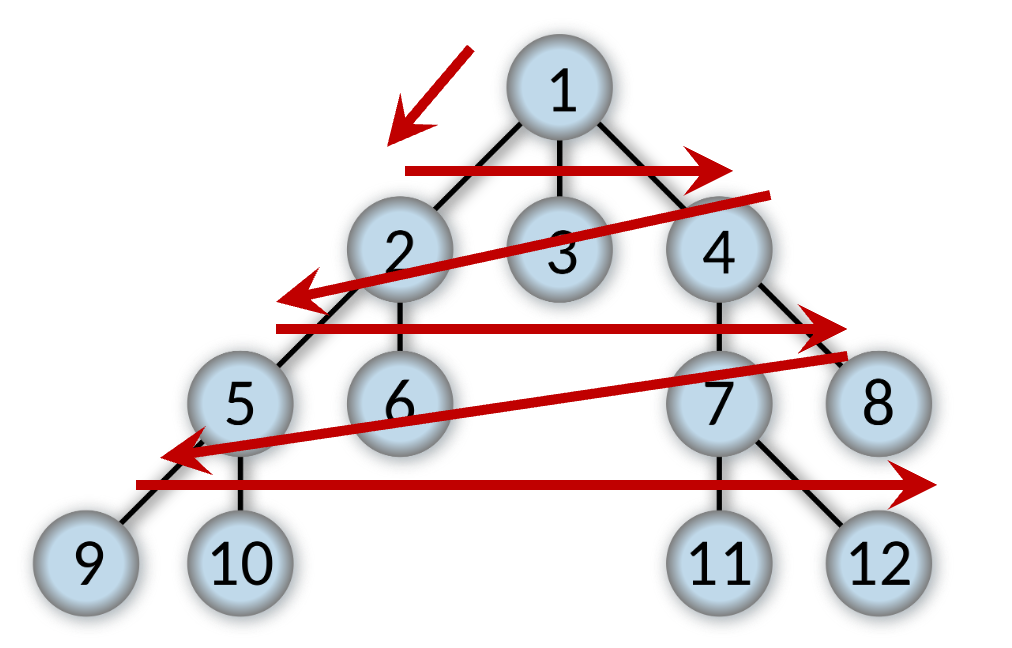
Одна из важных структур данных, которую мы не рассматривали в занятиях, — это дерево. Дерево представляет собой набор связанных узлов. У дерева есть корневой узел — это самый первый (верхний) узел дерева. У корневого узла есть потомки, у потомков, в свою очередь, есть свои потомки и так далее.



Узлы, у которых нет потомков, называют листовыми узлами, или просто листьями. Деревья используются во многих операциях: от поиска до управления иерархиями данных. Вообще говоря, вы уже использовали деревья, когда изучали и тренировали классификаторы Decision Tree и Random Forest. В общем, деревья — штука нужная.

Дерево, в отличие от списка, — это пространственная структура. Когда вы обходите плоский список, в большинстве случаев вы перебираете элементы в цикле слева направо. Гораздо реже — справа налево. С деревом всё сложнее: прогуляться по узлам дерева, как по плоскому списку, не получится, нужно учитывать связи между узлами. Или, как иногда говорят, топологию.

Обходить дерево можно по-разному. Один из популярных алгоритмов — **поиск в ширину.** Суть в следующем: вы начинаете обход дерева с заданной вершины, потом обходите всех её потомков, потом всех потомков первого потомка, потом всех потомков второго потомка и так далее. Задача — обойти всё дерево, посетив каждый узел не больше одного раза.



Обход дерева в ширину

Реализовывать обход дерева в ширину удобно с помощью очереди и специального множества «посещённых узлов». Очередь обеспечивает порядок обхода, а множество посещённых узлов хранит узлы, которые алгоритм уже просмотрел.

Алгоритм:

1. Помещаем вершину, с которой начинается обход, в очередь и во множество посещённых узлов.
2. Пока очередь непуста, выполняем следующее:
   1. Извлекаем элемент из очереди.
   2. Для всех потомков элемента проверяем, есть ли потомок во множестве посещённых вершин. Если нет, помещаем его в список посещённых вершин и в очередь. Повторяем шаг 2.

Получается (см. картинку), что сперва мы обходим всех потомков первого узла (попутно помещая их потомков в очередь), потом потомков второго узла, потом четвёртого и так далее.

Алгоритм достаточно прост. Впрочем, многое зависит от способа представления дерева. Один из распространённых способов — это списки смежности. Если говорить о Python, то списки смежности удобно представлять в виде словаря, ключами в котором являются узлы дерева, а значениями — тоже узлы, являющиеся потомками узла-ключа. Так, дерево на картинке представляется в следующем виде:

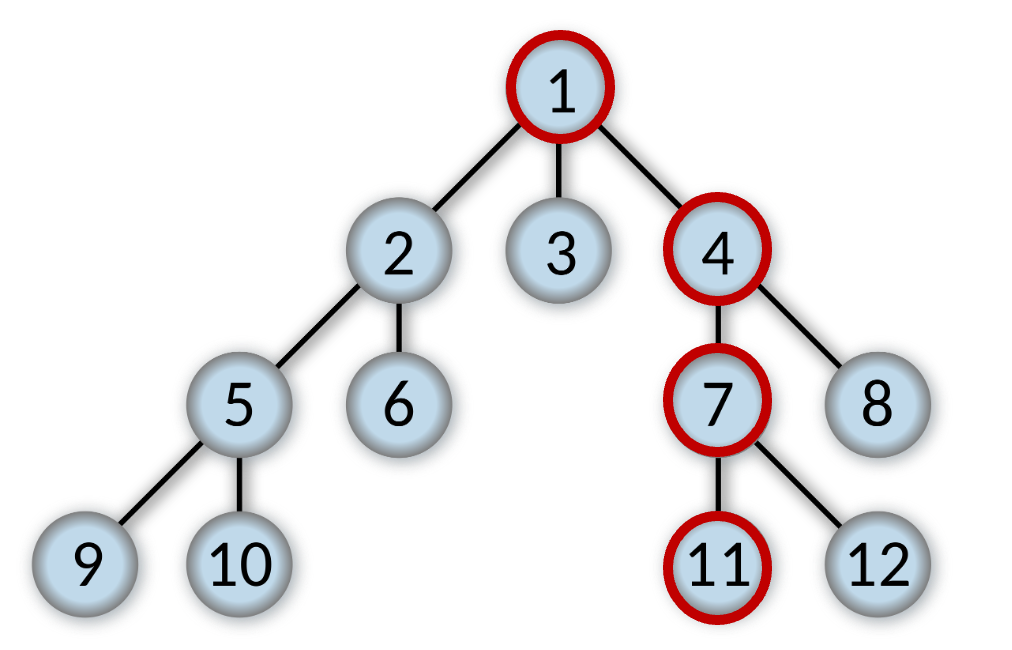
tree = {  
   1: [2, 3, 4],  
   2: [5, 6],  
   3: [],  
   4: [7, 8],  
   5: [9, 10],  
   6: [],  
   7: [11, 12],  
   8: [],  
   9: [],  
   10: [],  
   11: [],  
   12: []  
}

Найти всех потомков данной вершины в такой структуре очень легко. Например, tree[2] —  список всех потомков вершины 2.

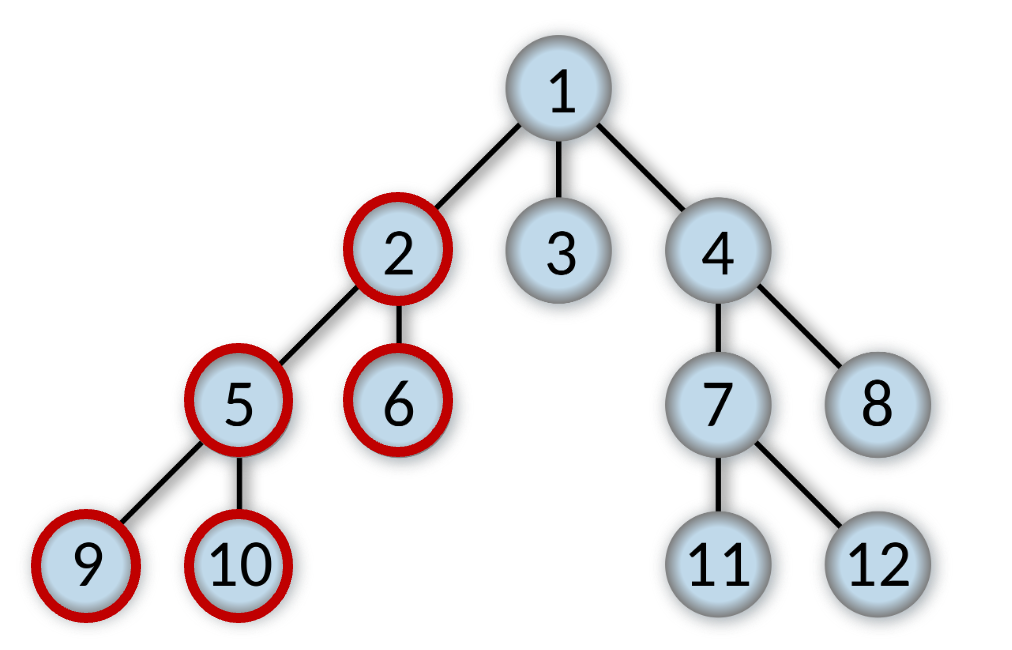
**Что нужно сделать**

1. Скачайте [файл task3.py.](https://drive.google.com/file/d/1iv0LLGeuesTuis9v_H3uZ3rztniH_lqc/view?usp=sharing" \t "_blank)
2. Найдите в нём место, помеченное комментарием «Напишите ваш код тут». Это будет функция bfs (от Breadth-first search — поиск в ширину), которая принимает параметры:
   1. tree — дерево, заданное списками смежности;
   2. start — номер вершины, с которой начинать обход (в примере на картинке — число от 1 до 12);
   3. desired — номер искомой вершины (в примере на картинке — число от 1 до 12).
3. Реализуйте код поиска узла по алгоритму обхода в ширину. Найти узел — значит пройти по дереву от начального узла до искомого. Поиск продолжается до тех пор, пока:
   1. искомый узел не найден;
   2. не осталось ни одной непосещённой вершины (узел не найден).

Функция должна возвращать True, если поиск удачен, и False в противном случае. Например, если начать поиск из вершины 1 (корневой узел) и искать узел 11, то поиск будет удачен. А вот если начать из вершины 2 и попытаться найти вершину 11, то ничего не получится.



Поиск вершины 11 из вершины 1 (найдена)



Поиск вершины 11 из вершины 2 (не найдена, потому что достичь вершины не удалось)

Если вы всё сделаете правильно, то при запуске кода увидите 6 строк со словом OK. Пришлите этот файл с кодом на проверку.

**Подсказки**

1. Для реализации очереди используйте collection.deque:
   1. поставить элемент в начало очереди можно методом appendleft;
   2. взять первый элемент из конца очереди можно методом pop.
2. В качестве множества «посещённых» вершин рекомендуется использовать set в Python.

**Что оценивается**

* Алгоритм реализован корректно и в соответствии с заданием: функция возвращает True, если поиск удачен, и False в противном случае.
* При запуске программы вы получаете 6 строк со словом OK.