Question 1 : Circular Primes [10 marks]

The number, 197, is called a *circular prime* because all rotations of the digits: 197, 971, and 719, are themselves prime. There are thirteen such primes below 100: 2, 3, 5, 7, 11, 13, 17, 31, 37, 71, 73, 79, and 97. In this question, we will count these circular primes. Note that we are talking about rotation and NOT permutation of the digits.

A. [**Warm Up**] Write a function `rotations` that will take as arguments a positive integer *n* and return a list of all the rotations for the number with **no repetitions**. The order within the list doesn't matter. [5 marks]

Sample execution:

```
>>> rotations(1)
[1]

>>> rotations(11)
[11]

>>> rotations(101)
[101, 11, 110]

>>> rotations(123)
[123, 231, 312]

>>> rotations(221)
[221, 212, 122]

>>> rotations(1231)
[1231, 2311, 3112, 1123]

>>> rotations(1212)
[1212, 2121]
```

B. [**Count Circular Primes**] Write a function `count_circular_primes` that will take as arguments a positive integer n and return the number of circular primes $\leq n$. [5 marks]

-

Sample execution:

```
>>> count_circular_primes(2)
1

>>> count_circular_primes(4)
2

>>> count_circular_primes(13)
6

>>> count_circular_primes(57)
9

>>> count_circular_primes(100)
13
```

Question 2 : Tyrion's Flight Mania [10 marks]

After trying his luck with the dragons of the *Khaleesi*, Tyrion Lannister has *time-travelled* to the modern day. He is now fixated upon machines that fly common people over large distances – the airplanes. He is quite interested in knowing who owns how many of these airplanes, which are the most popular ones and how they interconnect several modern cities on planet earth. With the help of his friend Lord Varys, Tyrion has managed to get a flight routes dataset. Your task is to use the sacred magic of Python on this flight routes dataset and answer the following questions that Tyrion has posed. And it's in your best interests to do so; because a Lannister always pays his debts!

The flight data is provided in the accompanying file `flight_routes.csv` where each entry contains the following information:

- Airline : 2-character IATA code that uniquely identifies an airline operating the route
- Source airport : 3-character IATA code that uniquely identifies the source airport
- Destination airport : 3-character IATA code that uniquely identifies the destination airport
- Aircraft type(s) : Alphanumeric code(s) to identify particular plane(s) used on the flight route.

Note that (i) Each route is defined by a source-destination pair of airports and several airlines would operate on a route. For a route, there is only **one entry** per airline to indicate that the airline serves that route. (ii) For a given route, an airline could fly different aircrafts depending upon the number of bookings for a particular flight. For example, Singapore Airlines (`SQ`) operates four different aircrafts on the route from Singapore (`SIN`) to Beijing (`PEK`), namely – `772, 77W, 773` and `333`. These are unique codes illustrating different aircrafts.

You do not have to worry about the codes used for airlines, airports or aircrafts and they are to be treated merely as keys.

A. [**Number of Flights**] Help Tyrion to easily find out the number of **direct** flights available if he wants to fly from one city to another. Thus, write a function `get_num_flights` that takes in a source airport, a destination airport and the name of the file containing the flights data. The function should return the number of direct flights available between the source and destination airports. It returns zero, if no direct flights are available.

Note: In reality, one airline could operate several direct flights on a route. However, for simplicity all these flights are represented as a single flight per airline in this dataset. So do not be alarmed if the numbers look too small to be believable. [2 marks]

Sample execution:

```
>>> get_num_flights('VIE','HAM','flight_routes.csv')
1

>>> get_num_flights('SIN','MNL','flight_routes.csv')
3

>>> get_num_flights('SIN','HAV','flight_routes.csv')
0
```

B. [**Airport Hubs**] The airports that act as connecting points for several domestic or international flights are called the *airports hubs*. Needless to say, they are also one of the busiest airports in terms of the number of flights that arrive **and** depart. Write a function `get_top_k_hubs` that takes in a value k and the flights dataset to return the list of top k hubs. The returned list should consists of tuples of the form `(airport,flights_count)` which would be arranged in descending order of the number of flights. If there are airports that serve equivalent number of flights as that of the kth airport, include them in the list as well. Also, if there is any tie within the top-k, then the airports should be listed in an alphabetical order. In case the k is greater than the total number of airports in the data file, your solution should include all the airports.

[4 marks]

Sample execution:

```
>>> get_top_k_hubs(1,'flight_routes.csv')
[('SIN', 224)]

>>> get_top_k_hubs(2,'flight_routes.csv')
[('SIN', 224), ('MNL', 144)]

>>> get_top_k_hubs(3,'flight_routes.csv')
[('SIN', 224), ('MNL', 144), ('CGN', 134)]

>>> get_top_k_hubs(4,'flight_routes.csv')
[('SIN', 224), ('MNL', 144), ('CGN', 134), ('CTU', 122)]
```

C. [**Flight Search**] Not all airports are connected by direct flights and sometimes you need to make transfers, typically at the airport hubs. When Tyrion would fly from one city to another, depending upon his mood he would be willing to change flights, or not. Write a function `search_routes` that takes source airport, destination airport, the flights dataset and the **maximum** number of hops as input. The function should return the list of all possible routes in the **increasing order** of number of hops. Each route would be presented as a list of airports. Also no route would involve visiting the same airport twice (i.e. no loops) for obvious reasons. An empty list should be returned if no route is found for the specified maximum number of hops. Your solution should work for any number of maximum hops. Also there should be no duplicate routes in the returned list of routes.

[4 marks]

Sample execution:

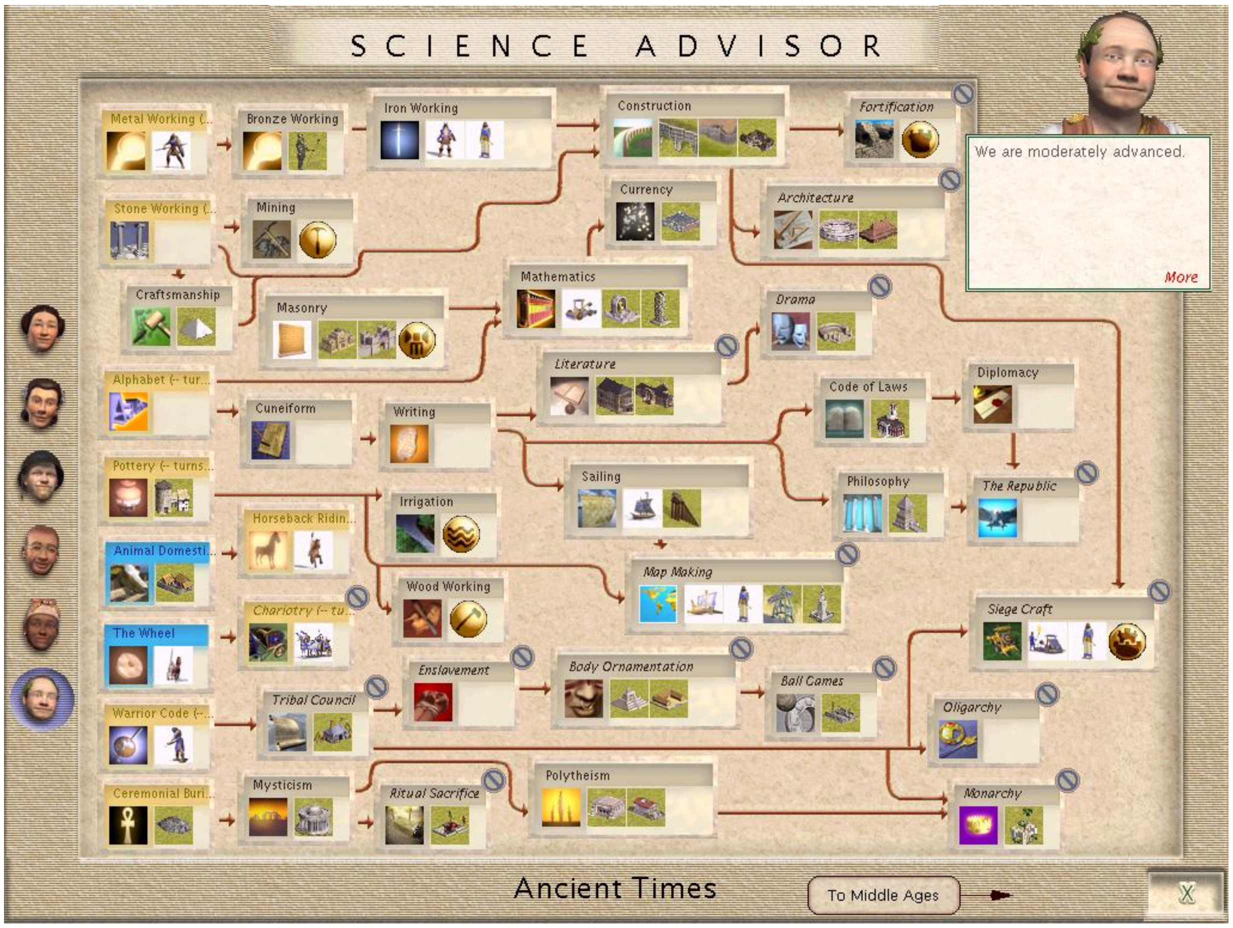
```
>>> search_routes('LED','NBC','flight_routes.csv',1)
[['LED', 'NBC']]

>>> search_routes('LED','NBC','flight_routes.csv',2)
[['LED', 'NBC'], ['LED', 'DME', 'NBC']]

>>> search_routes('LED','NBC','flight_routes.csv',3)
[['LED', 'NBC'], ['LED', 'DME', 'NBC'], ['LED', 'KZN', 'DME', 'NBC'],
['LED', 'KZN', 'SVX', 'NBC'], ['LED', 'UUA', 'DME', 'NBC'],
['LED', 'ASF', 'DME', 'NBC'], ['LED', 'SCW', 'SVX', 'NBC'],
['LED', 'OVB', 'SVX', 'NBC'], ['LED', 'DYU', 'DME', 'NBC'],
['LED', 'DYU', 'SVX', 'NBC'], ['LED', 'LBD', 'DME', 'NBC'],
['LED', 'CSY', 'DME', 'NBC'], ['LED', 'MCX', 'DME', 'NBC'],
['LED', 'SKX', 'DME', 'NBC'], ['LED', 'VOZ', 'DME', 'NBC']]
```

In the third sample execution above, there is no entry such as `['LED','KZN','LED','NBC']` because although valid, it has a loop. For routes that have the same number of hops, the sequence in which they appear in the list doesn't matter. Note that since this is a real world data, it may not be perfect – it is possible that it may contain an airport from which there are no outgoing flights.

Question 3 : Learning to Make Games [10 marks]

After learning how to program, you decided to take up an internship at a game company, where they are doing a remake of the classic strategy game *Civilization*. Civilization is a empire-building game where the gamer controls a civilization and attempts to lead it to greatness.

One core aspect of the game is *research*, where the player can invest research points to try to unlock in-game abilities via research. A typical research tree is shown here:



We are moderately advanced.

Basically, the technologies are organized with a set of dependencies. One may only attempt to research and unlock more advanced technologies only after the pre-requisite technologies have been researched. Your job is to implemented this technology tree as a new class `TechTree`. An empty `TechTree` can be constructed by passing a string `name` which would be the name of the `TechTree`. A `TechTree` supports the following methods.

- `get_name()` returns the name of the technology tree.
- `add_tech(tech)` adds a new technology `tech` into the technology tree and returns `True` if the operation is successful, or `False` if it fails because the `tech` is already in the tree.
- `add_dependency(parent,child)`: adds a parent technology as a dependency for the child technology, i.e. child can only be unlocked if the parent is already unlocked and returns `True` if the operation is successful. This methods returns `False` if the operation fails because either parent or child are not in the tree. The parameters `parent` and `child` are both strings identifying the corresponding tech.

- `get_parents(tech)` return the list of direct pre-requisites for the specified technology. Or `False` if `tech` is not in the tree. The sequence of parents in the returned list doesn't matter. Also if the `tech` has no parents i.e. it is the root node, `get_parents(tech)` should return an empty list.
- `get_ancestors(tech)` return the full list of pre-requisites technologies for the specified technology. Or `False` if `tech` is not in the tree. The sequence of ancestors in the returned list doesn't matter. Also if the `tech` has no ancestors i.e. it is the root node, `get_ancestors(tech)` should return an empty list.
- `unlock(tech)` attempts to unlock the specified technology. This will succeed and return `True` only if all the pre-requisites have already been unlocked. Returns `False` if the tech is already unlocked or if the pre-requisites have not been unlocked.
- `is_unlocked(tech)` returns `True` if the `tech` is unlocked and `False` otherwise.
- `has_loop()` returns `True` if there is a problem with the technology tree, i.e. there is a loop (which means that the tree is bugged since it would be impossible to unlock the technologies in the loop). Method returns `False` if the tech tree is "normal", i.e. no loops.

You are welcome to define additional helper methods if you think they are helpful.
Sample execution (newlines added for readability):

```
>>> tt = TechTree("civilization")
>>> tt.add_tech("metal working")
>>> tt.add_tech("stone working")
>>> tt.add_tech("bronze working")
>>> tt.add_tech("iron working")
>>> tt.add_tech("construction")
>>> tt.add_tech("mining")
>>> tt.add_tech("craftsmanship")
>>> tt.add_dependency("metal working","bronze working")
>>> tt.add_dependency("bronze working","iron working")
>>> tt.add_dependency("iron working","construction")
>>> tt.add_dependency("stone working","mining")
>>> tt.add_dependency("stone working","craftsmanship")
>>> tt.add_dependency("craftsmanship","construction")
>>> tt.add_dependency("stone working","construction")

>>> tt.get_parents("mining")
['stone working']

>>> tt.get_ancestors("mining")
['stone working']

>>> tt.get_parents("construction")
['iron working', 'craftsmanship', 'stone working']
```

```
>>> tt.get_ancestors("construction")
['stone working', 'craftsmanship', 'iron working', 'bronze working', 'metal working']

>>> tt.is_unlocked("stone working")
False

>>> tt.unlock("stone working")
True

>>> tt.is_unlocked("stone working")
True

>>> tt.is_unlocked("construction")
False

>>> tt.unlock("construction")
False

>>> tt.is_unlocked("construction")
False

>>> tt.has_loop()
False

>>> tt.add_dependency("construction","stone working")
>>> tt.has_loop()
True
```