Programming Assignment 2

ECS 032B FQ 2022

Due: November 20th, 2022

Setup:

Python3.6+ to develop and run your code.

Provided files:

PA2 /

- __init__.py
- recursion.py
- graphs.py
- prims.py
- dijkstras.py
- data.json
- test.py
  - *Run this file to test your code.*
- tests/
  - test_graph.py
  - test_prims.py
  - test_dijkstras.py
  - test_recursion.py

Submission instructions:

Submit all provided files through Gradescope as a zip file.

Do not remove/rename files nor change any naming conventions.

You should receive your grade for correctness automatically.

Rubric:

Correctness = 18 points

TAs will double-check your assignment to ensure that you are adhering the specifications. They may overrule your score if you break the listed restrictions.

Documentation = 2 points

Provide comments in code and good documentation on Gradescope.

A TA will grade this based on correctness.

Extra Credit

+0.2 points for every day you turn in all parts (Python files + Documentation) early. You must submit all parts; the latest day will be counted.

(Maximum: 2 points)

You only need to do either Section 3 or Section 4.

If you choose to do both, you will receive the additional grade as extra credit.

(+4 points)

Section 1: Recursion (recursion.py)

No additional modules to be imported for this section.

You may use any built-in Python functionalities.

```
def howManyGroups(n: int, m: int) -> int:
```

Given an array of n non-unique elements, find the number of unique groups such that each subset within the group has at most m elements.

In other words, split the elements into subsets with a maximum number of m elements to form a group. Then, output the number of different ways, or the number of groups, these elements can be split up.

Approach this problem using recursion.

Think about what the base cases are and how solutions relate to each other.

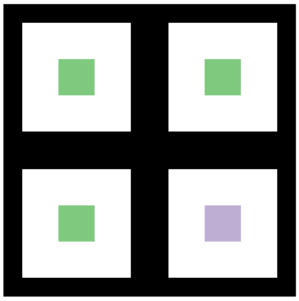


Figure 1. This image shows a visual of `howManyGroups(n = 2, m = 2) = 2`.

In the image above, each row is a different group. For each group, the colors represent which subset the element belongs to. Notice how for each row/group, there can only be a max of `m = 2` blocks/elements that belong to a certain color/subset.

Given an array of `n = 2` elements, there are only 2 groups with subsets that contain less than or equal to `m = 2` elements. The first group has 1 subset, seen in the top row, with 2 elements. The second group has 2 subsets, each with 1 element. There are no other unique groups we can find. Therefore, we return 2.

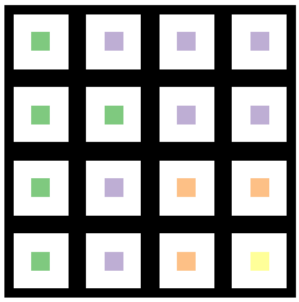
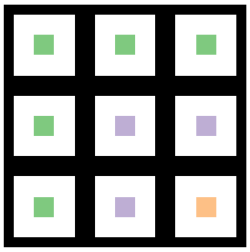
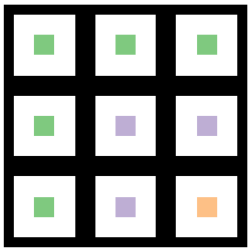
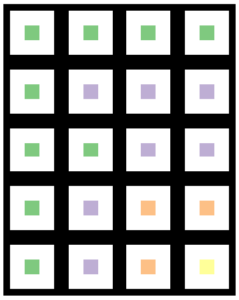
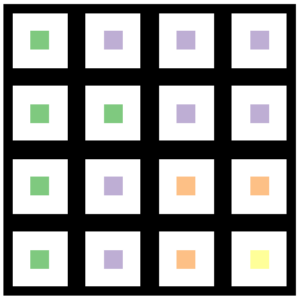
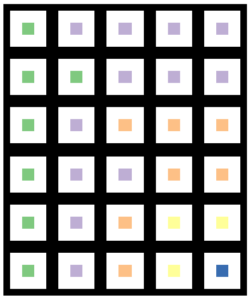
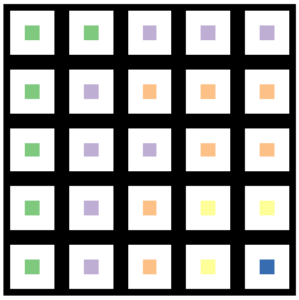
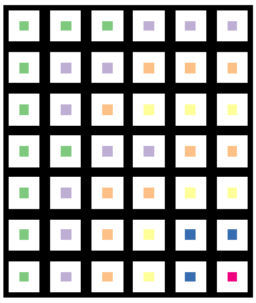
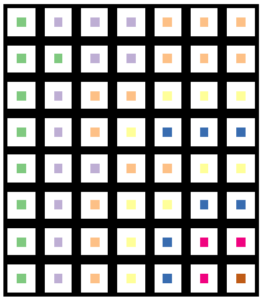


Figure 2. This image shows a visual of `howManyGroups(n = 4, m = 3) = 4`.

Given an array of four elements, there are `n = 4` instances of groups such that each group have subsets contain less than or equal to `m = 3` elements.

The first group has 2 subsets, with one subset with 1 element, the other with 3 elements. The second group has 2 subsets, each with 2 elements. The third group has 3 subsets, two with 1 element and another with 2 elements. The fourth group has 4 subsets, each with 1 element. There are no other unique groups we can find. Therefore, we return 4.

 n = 3, m = 4, output = 3	 n = 3, m = 3, output = 3
 n = 4, m = 4, output = 5	 n = 4, m = 3, output = 4
 n = 5, m = 4, output = 6	 n = 5, m = 3, output = 5
 n = 6, m = 4, output = 9	 n = 6, m = 3, output = 7

 n = 7, m = 4, output = 11	 n = 7, m = 3, output = 8

For Section 2-4:

No additional modules may be imported unless specified.

You may use any built-in Python data structures and methods (i.e. sort())

Section 2: Graphs (graphs.py)

In this section, you will implement a graph class.

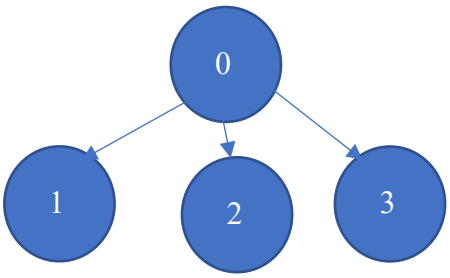
You may use your linked list code from PA1, but you are not required to.

You may use Numpy (https://numpy.org/) which is a Python library that allows you to work with matrices more easily as well as useful for other linear algebra operations.

Assume multigraphs are not permitted. This means there will be no parallel edges. (https://mathworld.wolfram.com/Multigraph.html)

For any traversal methods, to break ties, work in ascending order.

> Example: If you are at vertex 0, and the next vertices to go to are [1,2,3], you would select vertex 1, then vertex 2, then vertex 3. Our test cases will only work with Comparable types.



```
class Graph():
```

```
    def __init__(self):
```
Constructor for a graph.

You may choose to either represent the graph using an adjacency list or matrix.

```
    def addVertex(self, data: Any) -> None:
```
Adds a vertex `data` to the graph.

```
    def removeVertex(self, data: Any) -> None:
```
Remove the vertex `data` from the graph.

Assume the value of `data` is unique within the graph.

```
    def addEdge(self, src:Any, dest:Any, weight: float) -> None:
```
Adds an edge with `weight` from the vertex `src` to the vertex `dest`

```
    def addUndirectedEdge(self, A:Any, B:Any, weight: float) -> None:
```
Adds an undirected edge with `weight` between the vertex `A` and the vertex `B`

```
    def removeEdge(self, src:Any, dest:Any) -> None:
```
Removes all edges from the vertex `src` to the vertex `dest`

```
    def removeUndirectedEdge(self, A:Any, B:Any) -> None:
```
Remove all undirected edge between the vertex `A` and the vertex `B`

```
    def V(self) -> list:
```
Return a list of all vertices.

```
    def E(self) -> list:
```
Return a list of all edges, defined as a list of 3-tuples `(src, dest, weight)`.

```
    def neighbors(self, data: Any) -> list:
```
Returns a list of values of the neighbors of the vertex `data` in the graph.

We consider a vertex B a neighbor of vertex A if and only if the vertex A points to B.

```
    def dft(self, src: Any) -> list:
```
Perform depth-first traversal starting from the vertex with the value `src`

Return a list of the values of the vertices you visited in order.

```
    def bft(self, src) -> list:
```
Perform breadth-first traversal starting from the vertex with the value `src`

Return a list of the values of the vertices you visited in order.

```
    def isDirected(self) -> bool
```
Checks whether the graph is a directed graph.

An undirected graph must only have edges of equal weight that point in both directions.

```
    def isCyclic(self) -> bool:
```
Checks whether the graph has a cycle.

Since each edge would connect the two vertices in both directions for an undirected graph, such a graph is cyclic if a cyclic path exists beyond these two vertices.

```
    def isConnected(self) -> bool:
```
Checks whether the graph is (weakly) connected.

```
    def isTree(self) -> bool:
```
Checks whether the graph is a tree.

Section 3: Minimum Spanning Tree (prims.py)

In this section, you will be implementing a Prim's MST algorithm using the `Graph` class using the following pseudocode.

```
function prim(graph);
Let s be some vertex from graph.V;
Initialize a set of edges T as {};
Initialize U as {s} and V as graph.V - U;
while U != graph.V
  Let (u, v) be the lowest cost edge where u ∈ U and v ∈ V - U;
  T = T U {(u, v)};
  U = U U {v};
return T
```

`def prim(graph: Graph) -> list`

This function will perform Prim's algorithm to form a minimum spanning tree.
The input `graph` must be a weighted undirected graph.
You will return a list of edges.

`def runPrim() -> list`

This function will perform Prim's algorithm on the following task:

> You are hired by a company that has distribution centers in various cities across the U.S.
> The company want to create a road system that can directly service these centers, such that any center can be reached by following the new road system.
> The company wants you to figure out what roads should be built to connect these centers.
> They have provided you with the coordinates (x, y) of these centers in **data.json**.
> You need to provide them with what the roads should be built.

$$cost(loc_1, loc_2) = \sqrt{(loc_1.x - loc_2.x)^2 + (loc_1.y - loc_2.y)^2}$$

Section 4: Single Source Shortest Path (dijkstras.py)

In this section, you will be implementing Dijkstra's algorithm using the `Graph` class using the following pseudocode.

```
function dijkstras(graph, src);
Initialize a priority queue Q = {src};
Initialize a mapping of vertex cost C as {v=inf; v ∈ V \ src};
  cost[src] = 0;
Initialize explored vertices E as {v=False; v ∈ V \ src};
while Q.size != 0
  Let v be the connecting vertex with the lowest cost from Q;
  If v has not been explored
    Look the the neighbors N of v;
    Update cost of n ∈ N if this update is less;
    Add n to Q;
return cost
```

You may use the queue library.

(See: https://docs.python.org/3/library/queue.html#queue.PriorityQueue)

`def dijkstras(graph: Graph, src: Any) -> dict`

This function will perform Dijkstra's algorithm from a start vertex `src`.

You will return cost dictionary with keys as the vertex and the values as the costs.

`def runDijkstras() -> dict`

This function will perform Dijkstra's algorithm on the following task:

You are hired by a company that has distribution centers in various cities across the U.S. The company want to create a road system that can directly service these centers, such that any center can be reached by following the new road system.

They want to decide which center to choose as their headquarters, and what the distance is to each center.

They have provided you with the coordinates (x, y) of these centers in **data.json**.

$$cost(loc_1, loc_2) = \sqrt{(loc_1.x - loc_2.x)^2 + (loc_1.y - loc_2.y)^2}$$

It will return a dictionary of the cost dictionaries starting from every center.