15-110 Hw4 - Written Portion

Name:

AndrewID:

Complete the following problems in the fillable PDF, or print out the PDF, write your answers by hand, and scan the results. Also complete the programming problems in the starter file hw4.py from the course website. When you are finished, upload your hw4.pdf to **Hw4 - Written** on Gradescope, and upload your hw4.py file to **Hw4 - Programming** on Gradescope. Make sure to check the autograder feedback after you submit!

Don't forget that you can get three bonus points on Hw4 by filling out the midsemester surveys! Check Piazza for links to those surveys and further instructions.

Written Problems

#1 - Best Case and Worst Case - 8pts

Can attempt after Runtime and Big-O Notation lecture

For each of the following functions, describe an input that would result in **best-case efficiency,** then describe an input that would result in **worst-case efficiency**. This generic input must work at **any possible size**; don't answer 1 for `isPrime`, for example.

```
def getEmail(words):
    # words is a list of strings
    for i in range(len(words)):
        if "@" in words[i]:
            return words[i]
    return "No email found"
```

```
def isPrime(num):
    for factor in range(2, num):
        if num % factor == 0:
            return False
    return True
```

What is a generic **best case input** for `getEmail`?

What is a generic **worst case input** for `getEmail`?

What is a generic **best case input** for `isPrime`?

What is a generic **worst case input** for `isPrime`?

#2 - Calculating Big-O Families - 10pts

Can attempt after Runtime and Big-O Notation lecture

For each of the following functions, check the one best-matching **Big-O function family** that function belongs to. You should determine the function family by considering how the number of steps the algorithm takes grows as the size of the input grows.

```
def countEven(L): # n = len(L)
  result = 0
  for i in range(len(L)):
    if L[i] % 2 == 0:
      result = result + 1
  return result
```

- ☐ O(1)
- ☐ O(logn)
- ☐ O(n)
- ☐ $O(n^2)$
- ☐ $O(2^n)$

```
# n = len(L)
def sumFirstTwo(L):
  if len(L) < 2:
    return 0
  return L[0] + L[1]
```

- ☐ O(1)
- ☐ O(logn)
- ☐ O(n)
- ☐ $O(n^2)$
- ☐ $O(2^n)$

```
# n = len(L1) = len(L2)
def allLinearSearch(L1, L2):
  count = 0
  for item in L1:
    # Hint: linear search complexity..?
    if linearSearch(L2, item) == True:
      count = count + 1
  return count
```

- ☐ O(1)
- ☐ O(logn)
- ☐ O(n)
- ☐ $O(n^2)$
- ☐ $O(2^n)$

```
# n = len(L1) = len(L2)
def bothBinarySearch(L1, L2, item):
  # Hint: binary search complexity..?
  result1 = binarySearch(L1, item)
  result2 = binarySearch(L2, item)
  return result1 or result2
```

- ☐ O(1)
- ☐ O(logn)
- ☐ O(n)
- ☐ $O(n^2)$
- ☐ $O(2^n)$

```
# n = len(L); original call has i = 0
def recursiveSum(L, i):
  if i == len(L):
    return 0
  else:
    return L[i] + recursiveSum(L, i+1)
```

- ☐ O(1)
- ☐ O(logn)
- ☐ O(n)
- ☐ $O(n^2)$
- ☐ $O(2^n)$

#3 - Tree Vocabulary - 5pts

Can attempt after Trees lecture

Consider the following tree, implemented in code with our dictionary implementation:

```
t = { "contents" : "A",
      "left"  : { "contents" : "B",
                  "left"  : None,
                  "right" : None },
      "right" : { "contents" : "C",
                  "left"  : { "contents" : "D",
                              "left"  : None,
                              "right" : { "contents" : "E",
                                          "left"  : None,
                                          "right" : None } },
                  "right" : { "contents" : "F",
                              "left"  : None,
                              "right" : None } } }
```

How many **nodes** does this tree have?	
Which nodes are **children** of the node with value "C"?	
What is the value of the **root** of the tree?	
What are the values of the **leaves** of the tree?	
If we ran the first version of the function `countNodes` from lecture on this tree (with leaf base case), what is the **total** number of function calls that would be made?	

#4 - Graph Vocabulary - 5pts

Can attempt after Graphs lecture

In class we discussed how a graph can be used to model a social network. Create a social network graph of your own design with the same notation we used in class (ovals for nodes, lines for edges). You can base the graph on whoever you like - fictional characters, celebrities, people in your own life, etc. - but your graph must meet the following requirements:

- The graph should contain at least five labeled nodes
- The graph should contain at least five edges
- The graph should contain an annotation that points out a pair of neighbors
- The graph should contain an annotation describing whether it is weighted or unweighted (your choice, but the annotation must match the graph)
- The graph should contain an annotation describing whether it is directed or undirected (your choice, but the annotation must match the graph)

You can do this with a picture of a physical drawing or an online image editing tool (like Google Drawings). To upload the image on the next page, use the same approach you used on Hw2.

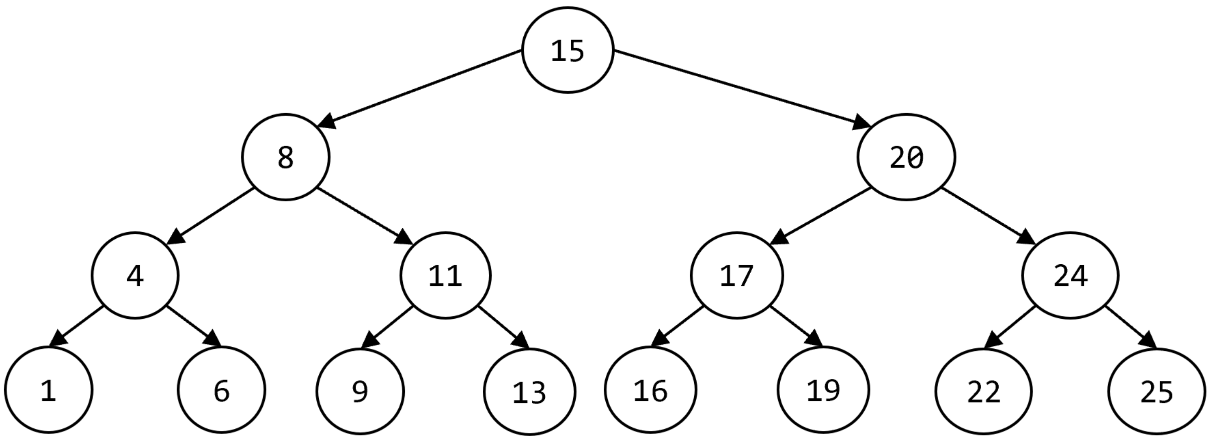
Click here to add your image

If that doesn't work, use a PDF tool to add your image manually.

#5 - Searching a BST - 6pts

Can attempt after Search Algorithms II lecture

Given the Binary Search Tree shown below:



What series of numbers would you visit if you ran a search algorithm that looked for **19**?

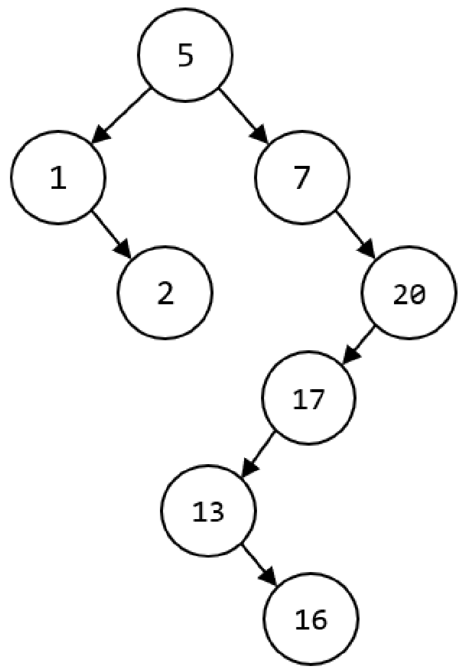
What series of numbers would you visit if you ran a search algorithm that looked for **4**?

What series of numbers would you visit if you ran a search algorithm that looked for **10**?

#6 - Binary Search Tree Efficiency - 6pts

Can attempt after Search Algorithms II lecture

Consider the following binary search tree:



When running binary search on this tree and selecting the item to search for…

What is a **best case input** that could be provided for this particular tree?

What is a **worst case input** that could be provided for this particular tree?

What is the **Big-O runtime** for binary search on trees like this?

#7 - Good Use of Hashing? - 5pts

Can attempt after Search Algorithms II lecture

Recall our discussion of what hash functions are and what they are used for. Below we've listed four different scenarios. Each scenario contains a data set, a hash function, and which values will need to be looked up in the hashtable. Select **all** the scenarios where you will generally be able to look up the given values in **constant time**.

- ☐ Given a set of integer phone numbers, hash a phone number based on the phone number itself. Use the hashtable to look up an individual phone number.

- ☐ Given a set of all the college essays sent to CMU (as strings), hash an essay based on the ASCII value of the first character of the essay ("I want to go to CMU because.." hashes based on "I"). Use the hashtable to look up an individual essay.

- ☐ Given a set of string full names (like "Farnam Jahanian"), hash a name by adding together the numeric ASCII values of all the letters in the name. Use the hashtable to look up an individual name.

- ☐ Given a set of lists of high scores (so each list contains integers), hash a list based on the sum of its scores. Lists can be updated after hashing when new high scores are added. Use the hashtable to look up an individual high-score list.

- ☐ None of the situations described above can be searched in constant time.

#8 - P and NP Identification - 5pts

Can attempt after Tractability lecture

For each of the following problems, identify whether the problem is in the complexity class P, NP, or neither. Please choose just one class (the one that **best** describes the problem), even if multiple classes are technically correct.

Finding the smallest value in a tree	☐ P ☐ NP ☐ Neither
Scheduling final exams for CMU so that there are no conflicts	☐ P ☐ NP ☐ Neither
Determining if an item is in a list	☐ P ☐ NP ☐ Neither
Finding the **best** (fastest) road route through Pittsburgh that takes you over every bridge.	☐ P ☐ NP ☐ Neither
Determining if there is a set of inputs that makes a circuit output 1	☐ P ☐ NP ☐ Neither

#9 - P vs NP - 6pts

Can attempt after Tractability lecture

For each of the following questions choose just one answer, the **best** answer.

Which of the following is the best definition of the complexity class **P**?

- ☐ The set of problems that can be solved in polynomial time
- ☐ The set of problems that can be verified in polynomial time
- ☐ The set of problems we discussed in lecture (Puzzle Solving, Subset Sum, etc)

Which of the following is the best definition of the complexity class **NP**?

- ☐ The set of problems that can be solved in polynomial time
- ☐ The set of problems that **cannot** be solved in polynomial time
- ☐ The set of problems that can be verified in polynomial time
- ☐ The set of problems that **cannot** be verified in polynomial time
- ☐ The set of problems we discussed in lecture (Puzzle Solving, Subset Sum, etc)

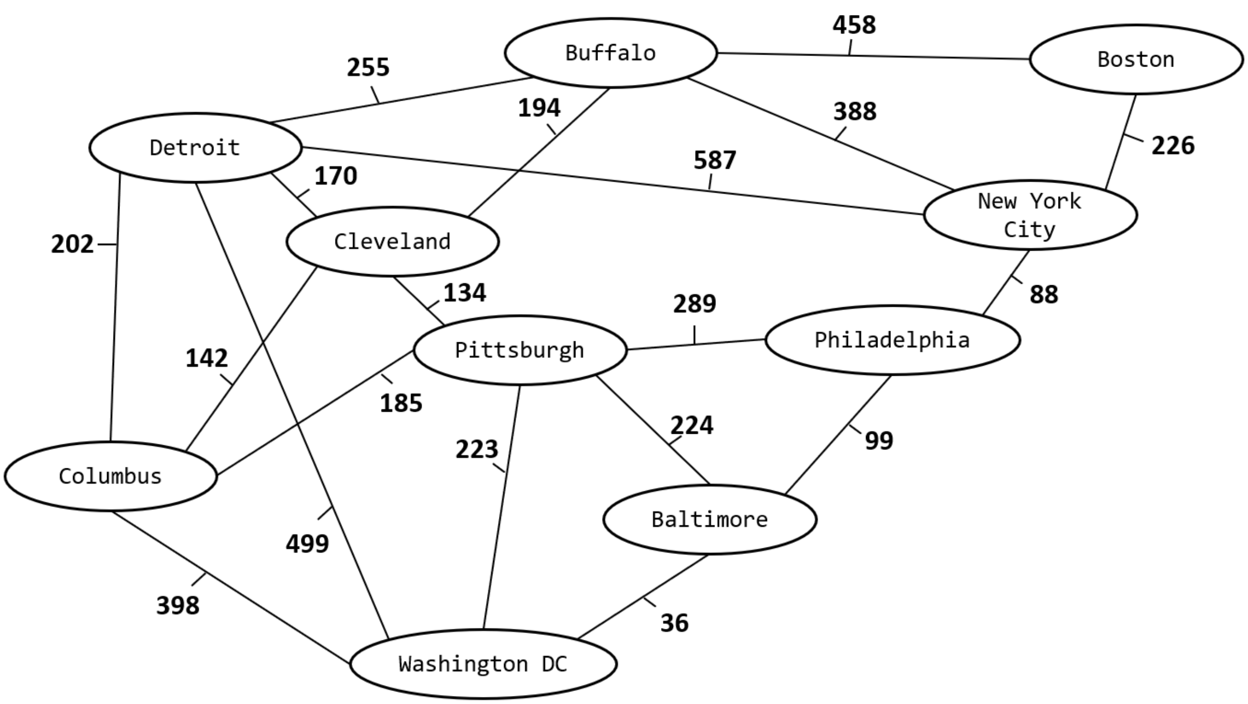
Why does it matter whether or not P = NP? Choose the best answer.

- ☐ If they are the same, we'll be able to solve hard and useful problems a lot faster
- ☐ If they are the same, we'll need to change how we implement some adversarial algorithms, like encryption, to keep them from being broken easily
- ☐ If they are not the same, we can spend less time trying to invent super-fast solutions to hard but useful problems in NP
- ☐ All of the above

#10 - Heuristics - 5pts

Can attempt after Tractability lecture

We want to apply the Travelling Salesperson algorithm to the graph shown here to find a short route that visits each city once, but we want to use a **heuristic** to get the answer quickly. Our heuristic is this: for each choice point, rank the neighbor cities that have not yet been visited based on the weight of the edge leading to them, then choose the lowest-weight edge (the shortest distance).



Say we want to start in New York City and visit each city once (returning to New York City at the end). What path would this heuristic generate?

#11 - Recognizing Data Structures - 5pts

Can attempt after Graphs lecture

For each of the following types of data, choose the single data structure that would be the **best/most natural choice** to represent the data

Carnegie Mellon's organizational structure: ie, departments within each college, and majors within each department	☐ 1D List ☐ 2D List ☐ Dictionary ☐ Tree ☐ Graph
A chess board that has pieces located at specific row-column positions	☐ 1D List ☐ 2D List ☐ Dictionary ☐ Tree ☐ Graph
A set of chores you need to do over the weekend	☐ 1D List ☐ 2D List ☐ Dictionary ☐ Tree ☐ Graph
The subway map for London	☐ 1D List ☐ 2D List ☐ Dictionary ☐ Tree ☐ Graph
A deck of flashcards with words on one side and definitions on the other	☐ 1D List ☐ 2D List ☐ Dictionary ☐ Tree ☐ Graph

#12 - Optimizing for Search - 4pts

Can attempt after Search Algorithms II lecture

You have been given a very large dataset of temperatures (represented as floats), and your task is to find the most extreme temperatures that fall into a given temperature range (such as 40 degrees to 50 degrees, or 75.7 degrees to 78.2 degrees). To do this, you want to store the data in a data structure so that, given any range, you'll be able to:

- find the smallest value in the structure that falls in that range
- find the largest value in the structure that falls in that range

You want to optimize how quickly you can run the algorithm shown above, assuming the data structure has already been created. In other words, **you don't know what range you'll need to check when you create the structure.**

Choose the best search algorithm + data structure combination for the task. There might be multiple correct answers; you only need to choose one per question. Note that you should pick the best search algorithm **for this prompt**, not the best search algorithm generically!

Search Algorithm:

- ☐ Linear Search
- ☐ Binary Search
- ☐ Hashed Search
- ☐ Random Search

Data Structure:

- ☐ Sorted List of degrees
- ☐ Dictionary mapping degree->count
- ☐ Binary Search Tree of degrees
- ☐ Graph connecting close degrees

Programming Problems

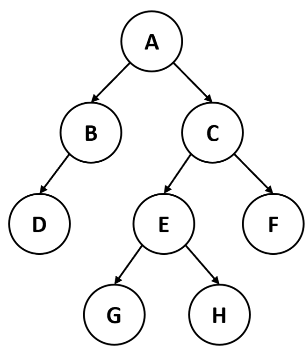
For each of these problems (unless otherwise specified), write the needed code directly in the Python file in the corresponding function definition.

All programming problems may also be checked by running 'Run current script' on the starter file, which calls the function `testAll()` to run test cases on all programs.

#1 - `getLeftmost(t)` - 5pts

Can attempt after Trees lecture

Write the function `getLeftmost(t)` that takes a binary tree in our dictionary format and returns the contents of the **leftmost** child of that tree. This is the child we reach if we keep moving down and left from the root node until we cannot go left any further. For example, in the tree:



Which is represented as the dictionary:

```
t = { "contents" : "A",
      "left" : { "contents" : "B",
          "left" : { "contents" : "D", "left" : None, "right" : None},
          "right" : None },
      "right" : { "contents" : "C",
          "left" : { "contents" : "E",
              "left" : { "contents" : "G", "left" : None, "right" : None },
              "right" : { "contents" : "H", "left" : None, "right" : None } },
          "right" : { "contents" : "F", "left" : None, "right" : None } } }
```

We go from A to B, then from B to D, then we can't go left any further. `"D"` is the content of the leftmost node and is returned when we call the function on `t`.

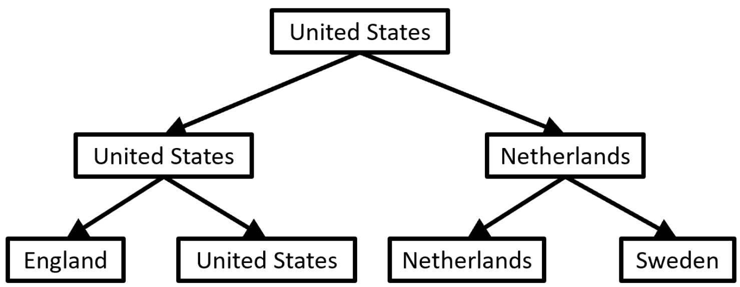
Hint: you can solve this using recursion, or you can just use a while loop.

#2 - `getInitialTeams(bracket)` - 10pts

Can attempt after Trees lecture

We can represent a tournament bracket from a sports competition as a binary tree. To do this, store the winning team as the root node. Its children are the winning team again, as well as the second-place team. In general, every node represents the winner of a match, and its two children are the two teams that competed in that match.

For example, the following bracket represents the last two rounds of the Women's World Cup in 2019.



In our binary tree dictionary format, this would look like:

```
t1 = { "contents" : "United States",
       "left" : { "contents" : "United States",
           "left"  : { "contents" : "England", "left" : None, "right" : None },
           "right" : { "contents" : "United States", "left" : None, "right" : None}},
       "right" : { "contents" : "Netherlands",
           "left"  : { "contents" : "Netherlands", "left" : None, "right" : None },
           "right" : { "contents" : "Sweden", "left" : None, "right" : None } } }
```

Write the function `getInitialTeams(bracket)` which takes a tournament bracket and returns a list of all the teams that participated in that tournament. For example, if the function is called on the tree above it might return `[ "England", "United States", "Netherlands", "Sweden" ]`. You will need to implement this function **recursively** to access all the nodes. We recommend that you start by looking at the `sumNodes` and `listValues` examples from the slides.

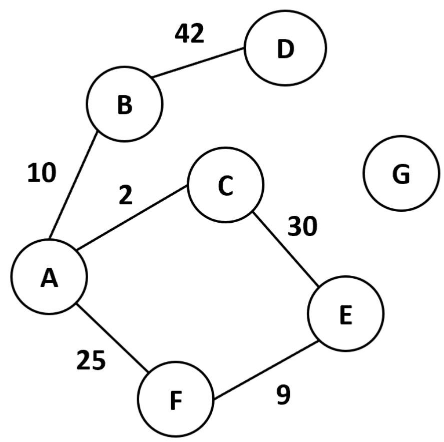
Hint 1: how can we get all of the teams to show up in the list exactly once? Every team occurs at the very beginning of the tournament, in the first set of matches. In the tree, this is represented by the **leaves,** so you should not include values on non-leaf nodes.

Hint 2: make sure the **type** you return is the same in both base and recursive cases!

#3 - `largestEdge(g)` - 10pts

Can attempt after Graphs lecture

We often want to find the **largest edge weight** in a graph. This can help us identify useful information, like the most congested street in a city or the two gas stops that are farthest apart on a highway. Write the function `largestEdge(g)` that takes a weighted graph in our dictionary format and returns a list holding two elements - the two endpoints of the edge with the largest weight in the graph. For example, in the graph:



Which is represented as the dictionary:

```
g = { "A" : [ [ "B", 10 ], [ "C",  2 ], [ "F", 25 ] ],
      "B" : [ [ "A", 10 ], [ "D", 42 ] ],
      "C" : [ [ "A",  2 ], [ "E", 30 ] ],
      "D" : [ [ "B", 42 ] ],
      "E" : [ [ "C", 30 ], [ "F",  9 ] ],
      "F" : [ [ "A", 25 ], [ "E",  9 ] ],
      "G" : [ ] }
```

The largest edge has the weight `42`. That edge is between the nodes B and D, so if we call the function on that graph, it will return `[ "B", "D" ]` (or `[ "D", "B" ]` - the order doesn't matter).

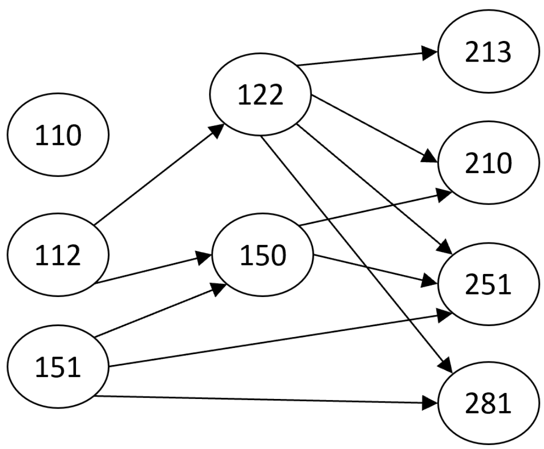
To find the largest edge, modify the find-most-common/find-largest-item pattern we've discussed several times in class. Iterate over each of the nodes in the graph, then for each node iterate over each of that node's neighbors to visit each edge.

Note: to make this easier, you are guaranteed that all edge weights will be **positive** and there will be at least one edge in the graph.

#4 - `getPrereqs(g, course)` - 5pts

Can attempt after Graphs lecture

College course prerequisites are notoriously complicated. However, we can make them a little easier to understand by representing the course dependency system as a **directed graph**, where the nodes are courses and an edge leads from course A to course B if A is a prerequisite of B. For example, the core Computer Science courses (almost) produce the following prereq graph:



Which would be represented in code as:

```
g = { "110" : [],
      "112" : ["122", "150"],
      "122" : ["213", "210", "251", "281"],
      "151" : ["150", "251", "281"],
      "150" : ["210", "251"],
      "213" : [],
      "210" : [],
      "251" : [],
      "281" : [] }
```

Write the function `getPrereqs(g, course)` that takes a directed graph (in our adjacency list dictionary format, without weights) and a string (a course name) and returns a list of all the immediate prerequisites of the given course. If we called `getPrereqs` on our graph above and `"210"`, for example, the function should return `["122", "150"]`.

Hint: you can't just return the neighbors of the course, because the edges are going in the opposite direction! Instead, iterate over all the nodes to find those that have the course as a neighbor. Construct a new list out of these nodes as the result.