# 15-110 - Quiz4 - 11/03/2021

Name: ______________________________ andrewID: __________________________

- This quiz tests material from weeks 1-8 of the course (primarily weeks 7-8).
- You have **20 minutes** to take the quiz.
- If you have a clarification question, raise your hand and a proctor will come help you.
- You must complete the quiz **individually**. You may refer to paper notes during the quiz, but do not communicate with anyone else.

## 1. Free Response - Big-O Runtime [30pts]

Write the Big-O runtime (in the worst case) of each of the following programs in the top of the box to its right. In the space below the box **show your work** with a short explanation of the parts of the program you used to calculate the runtime.

For example, if a program is O(n) because the third line does n actions, you could write 'Line 3 takes n steps'. The explanation does not need to be super detailed; just mention the core ideas.

**Note:** `list.count` runs in O(n) time. All other built-in functions shown here run in O(1).

```
1: def f1(lst): # n = len(lst)
2:   count = 0
3:   i = 0
4:   mid = len(lst) // 2
5:   while i < mid:
6:     if lst[i] == lst[i + mid]:
7:       count = count + 1
8:     i = i + 1
9:   return count
```

| **Big-O of f1:** | O(n) |
|---|---|
| Line 5 takes O(n/2) steps which is equivalent to O(n) (also fine to just say O(n)) | |

```
1: def f2(lst): # n = len(lst)
2:   res = 0
3:   for i in range(len(lst)):
4:     for j in range(len(lst)):
5:       c = lst.count(lst[i])
6:       if lst[i] < lst[j]:
7:         res = res + c
8:   return res
```

| **Big-O of f2:** | O(n^3) |
|---|---|
| - Line 3, 4, nesting loops iterates O(n^2) times<br>- Line 5 has a cost of O(n) because of lst.count which is a built in function | |

## 2. Code Writing - Graphs [34pts]

Write the function `getShortEdges(g, limit)` which takes a weighted, directed graph in our dictionary format and a limit (an integer) and returns a 2D list holding information about short edges. A 'short edge' is one where the edge weight is strictly smaller than the limit provided as a parameter.

Each of the inner lists in the 2D list returned value holds information about one edge in the format `[node1, node2, weight]`. The first two elements are the two neighbors that form the edge, and the third is the edge weight between them.

For example, given the graph:

```
g = { "A" : [ [ "C", 30 ] ],
      "B" : [ [ "D", 15 ] ],
      "C" : [ [ "B", 25 ], [ "D", 5 ] ],
      "D" : [ [ "C", 20 ] ],
      "E" : [ [ "C", 10 ] ]
    }
```

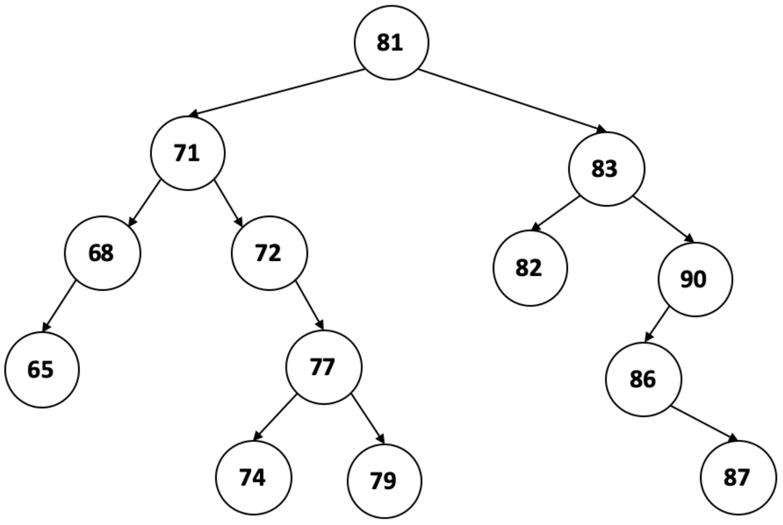
If we call `getShortEdges(g, 20)`, it will return
`[ [ "B", "D", 15 ], [ "C", "D", 5 ], [ "E", "C", 10 ] ]`.

The graph is directed, so you do not need to worry about edges repeating. You also do not need to worry about the order of the edges in the result list.

```
def getShortEdges(g, limit):
    results = []
    for node in g:
        for neighbor in g[node]:
            weight = neighbor[1]
            if weight < limit:
                element = [node] + neighbor
                results.append(element)
    return results
```

## 3. Short Answer - Search Algorithms [16pts]

Consider the following tree:



What nodes in the tree would the **binary search** algorithm visit if you searched for the value **74**? List the nodes in their visited order.

| |
|---|
| 81, 71, 72, 77, 74 |

What nodes in the tree would the **binary search** algorithm visit if you searched for the value **85**? List the nodes in their visited order.

| |
|---|
| 81, 83, 90, 86 |

## 4. Short Answer - Tractability [20pts]

Select **all** of the following statements that are **true**.

- ☐ Problem statement: check whether a list of numbers contains a set of numbers that sum to a given target number. This problem is in the complexity class **P**.
- ☐ To show that something is in the complexity class **NP**, we just need to be able to check a possible solution to that problem in polynomial time.
- ☐ It is possible for a problem to be in both **P** and **NP** at the same time.
- ☐ To prove that **P** = **NP**, we have to prove that **every** problem that could ever be designed can be solved in polynomial time.
- ☐ To prove that **P** != **NP**, we have to find just one problem in **NP** that can **never** be solved in polynomial time.