CS 372
ADVANCED ALGORITHMS
Spring 2017

ASSIGNMENT 1
Due Date: *Friday, March 31st, 2017, 23:59*

Assignment Submission: *Turn in your assignment by the due date through LMS. Prepare a Python file (.py) that has all functions described in the assignment. Name the file as '<your first name>_<your last name>_ assignment1.py'. Put your name and student ID to the left-top corner of the file as comment. Also, prepare another file (Word, PDF,TXT) that has the algorithm descriptions. Name the file as '<your first name>_<your last name>_ assignment1.<word,pdf,txt>'.*

All work in questions must be your own; you must neither copy from (including online resourses) nor provide assistance to anybody else. If you need guidance for any question, talk to the instructor or TA.

QUESTION 1 (35 points)

You have two bags of balls. In the first bag, there are blue (B), yellow (Y) and green (G) balls. In the second bag, there are white (W), red (R) and orange (O) balls. You start drawing balls from first bag with replacement (you put the ball back to the bag before drawing a new one). After an unknown number (**N**= 0 or more) of draws from the first bag, you switch the bags and start drawing balls from the second bag with replacements (and continue forever). For example, if N=5, you may get a sequence as follows:

YGGBYROWWRROWR…

Develop **a very efficient algorithm** and implement in Python to find out when the bag switch happens (e.g., return 5 in the example above).

To simply your implementation, we will assume that you get a finite list of outcomes (outcomes=['Y', 'G', 'G', 'B', 'Y', 'R', 'O', 'W', 'W', 'R', 'R', 'O', 'W', 'R']) as an input. **However, you are NOT allowed to use the length of the list (len(outcomes)) in your implementation. If you use it in your implementation, you will get no credit for this question.**

The reason for this restriction is that we continue drawing balls forever in actual case and effectively we get an infinite sequence. So, there is not a limited range with a specific length. In this question, we just assume that you have a list as an input to simplify your implementation.

If no balls is drawn from the first bag and we immediately started drawing from the second bag, your code should ***return 0***. If all outcomes in the input list are from the first bag, it should ***return None***.

What to return:

- Pseudo-code of your algorithm that should clearly describe your solution
- Complexity analysis of our algorithm.
- Python function (function name and input parameter that you have to use is given below) that implements your algorithm. Note that if you do a recursive implementation, you are not required to call this function recursively in your implementation. You can create other recursive functions that are called within this function. But, the top-level function that we will use to test your solution is this one.

def bag_switch (outcomes) :

// your code goes here

For example:

outcomes=['Y', 'G', 'G', 'B', 'Y', 'R', 'O', 'W', 'W', 'R', 'R', 'O', 'W', 'R']

bag_switch (outcomes)

QUESTION 2 (40 points)

You are given a list **(*NUMBERS)*** of ***N*** integers (do not have to be distinct numbers). In this question, you will develop an efficient ***recursive*** solution to divide this list into 3 parts (***NUMBERS***$_{left}$, ***NUMBERS***$_{middle}$, ***NUMBERS***$_{right}$) such that:

- Numbers in ***NUMBERS***$_{left}$ & ***NUMBERS***$_{right}$ are sorted
- All numbers in ***NUMBERS***$_{left}$ <= ***NUMBERS***$_{middle}$ <= ***NUMBERS***$_{right}$
- Lengths (size) of ***NUMBERS***$_{left}$ & ***NUMBERS***$_{right}$ are maximized

If you partition the list based on these rules, we only need to sort ***NUMBERS***$_{middle}$ to get a sorted ***NUMBERS*** (***NUMBERS***$_{middle}$ is the smallest possible partition of the NUMBERS that enables this)

For example, assume that NUMBERS is [1,3,4,6,8,10,13,15,11,9,12,13,15,17,20,25]. For this list, the partitions are:

NUMBERS$_{left}$: [1,3,4,6,8]

NUMBERS$_{middle}$: [10,13,15,11,9,12,13].

NUMBERS$_{right}$: [15,17,20,25]

In this example, if you add 10,13,15 to the left partition, the left partition is still sorted but there are numbers that are smaller on the remaining part of the list so the second condition above is violated. If we add 8 to the middle partition, both first and second conditions are satisfied but the left partition would not have the maximum possible size.

If we only sort middle partition after partitioning, complete list will be sorted. This is the smallest sub-array with this property.

In this question, **you are required to develop a recursive (divide-and-conquer) algorithm**. Like any recursive algorithm, you can, of course, find an iterative solution. In fact, iterative

solution makes a lot more sense in this question due to recursive depth, but, to practice with recursive algorithms, you will develop a recursive solution.

Your implementation should return a tuple of the ***beginning*** and the ***end*** index of $\boldsymbol{NUMBERS}_{middle}$ in *NUMBERS list*. In the example above, since then index of '10' in NUMBERS is 5 and the index of '13' is 11, the function you develop will return (5,11) (This is effectively the range in the list that you need to sort to get a fully sorted list). If all numbers in the input list *NUMBERS* is already sorted, $NUMBERS_{middle}$ is an empty list. In this case, return (None, None)

What to return:

- Pseudo-code of your algorithm that should clearly describe your solution.
- Complexity analysis of our algorithm.
- Python function (function name and input parameter that you have to use is given below) that implements your algorithm. Note you are not required to call this function recursively in your implementation. You can create other recursive functions that are called within this function. For example, you can create separate recursive functions to find left and right partitions and call them within the top-level function partition_list() that we will use to test your solution .

```
def partition_list (NUMBERS) :
    // your code goes here
```

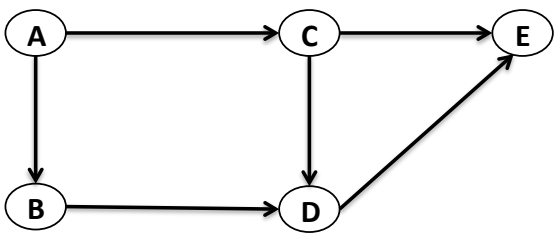
For example:

NUMBERS= [1,3,4,6,8,10,13,15,11,9,12,13,15,17,20,25]

partition_list (NUMBERS)

QUESTION 3 (25 points)

You are given a directed graph as an input where nodes are cities and edges are one-directional roads. However, this is just a partial graph and only shows one-way of roads; so, we ended up with **a directed acyclic graph (DAG).** Graph is represented as an array of tuples where each tuple represents an edge (road). For example, the graph in the figure below is represented as G=[(A,B),(A,C),(B,D),(C,D),(C,E),(D,E)].



Given two nodes, ***start*** and ***end***, develop and implement **a recursive algorithm** to find the number of possible paths (routes) from ***start*** to ***end.*** For example, in the graph given above, if

the ***start*** is A and the ***end*** is D, your code should return 2. If the ***start*** is A and the ***end*** is E, it should return 3. If the ***start*** and the ***end*** are same, return 1.

Note that your solution has to be a recursive algorithm. Create a graph representation of input G, similar to the example codes in class, and work on it.

What to return:

- Pseudo-code of your algorithm that should clearly describe your solution.
- Python function (function name and input parameters that you have to use is given below) that implements your algorithm. Note you are not required to call this function recursively in your implementation. You can create other recursive functions that are called within this function.

```
def number_of_paths (G, start,end) :
    // your code goes here
```

For example:

G=[(A,B),(A,C),(B,D),(C,D),(C,E),(D,E)],

start='A'

end='D'

number_of_paths (G, start,end)

NOTE: When we load your .py code and call the functions as described above with various input combinations, the functions should run and produce the result without any need for modification.