CSC 6013 final exam

You have four hours to solve all eight questions.

During the exam you are allowed to consult all materials presented in the course, including slides, class notes, recommended text or videos. However, you are not supposed to not access any other sources from the Internet.

While consulting the authorized sources, it is also important to pay attention to time, as this exam has a limited time of four hours.

You have to submit one single pdf with your answers.

In this pdf you can put electronic edited text, photo of handwritten text or annotations, print screen images, or any other materials you believe to be appropriate as your answer.

Remember to number your answers from 1 to 8.

If you have any issues submitting your pdf file, please send an email immediately to bradleym@merrimack.edu and fernandesp@merrimack.edu stating your issue and attaching your pdf file with answers.

1) Linked Lists - Create a Swap Method

Given the LinkedList implementation seen in class (here is a copy and also the link: linkedlist.py), create a method that swaps the node pointed by the *current* pointer by the next node (the *current* will be the formerly next node). This method shall return -1 if the swap is impossible (either the list is empty or the *current* node has no node next), or it should return 0 if the swap was performed.

For example, for a linked list:

20 40 50(current) 60 70

The method swap will change the linked list to

20 40 60(current) 50 70

… and it returns the value 0

Another example, for a linked list:

20 40 50 60 70(current)

The method swap will not change the linked list and it returns the value -1

```python
class Node:
    def __init__(self, d):
        self.Data = d
        self.Next = None

class LinkedList:
    def __init__(self, d=None):
        if (d == None): # an empty list
            self.Header = None
            self.Current = None
        else:
            self.Header = Node(d)
            self.Current = self.Header
    def nextCurrent(self):
        if (self.Current.Next is not None):
            self.Current = self.Current.Next
        else:
            self.Current = self.Header
    def resetCurrent(self):
        self.Current = self.Header
    def getCurrent(self):
        if (self.Current is not None):
            return self.Current.Data
        else:
            return None
    def insertBeginning(self, d):
        if (self.Header is None): # if list is empty
            self.Header = Node(d)
            self.Current = self.Header
        else:                     # if list not empty
            Tmp = Node(d)
            Tmp.Next = self.Header
            self.Header = Tmp
    def insertCurrentNext(self, d):
        if (self.Header is None): # if list is empty
            self.Header = Node(d)
            self.Current = self.Header
        else:                     # if list not empty
            Tmp = Node(d)
            Tmp.Next = self.Current.Next
            self.Current.Next = Tmp
    def removeBeginning(self):
        if (self.Header is None): # if list is empty
            return None
        else:                     # if list not empty
            ans = self.Header.Data
            self.Header = self.Header.Next
            self.Current = self.Header
            return ans
    def removeCurrentNext(self):
        if (self.Current.Next is None): # if there is no node
            return None                 #        after Current
        else:                           # if there is
            ans = self.Current.Next.Data
            self.Current.Next = self.Current.Next.Next
            return ans
    def printList(self,msg="====="):
        p = self.Header
        print("====",msg)
        while (p is not None):
            print(p.Data, end=" ")
            p = p.Next
        if (self.Current is not None):
            print("Current:", self.Current.Data)
        else:
            print("Empty Linked List")
        input("----------------")
```

2) **Python Dictionary - Dealing with Python Dictionaries**

Having Python Dictionaries in mind, answer briefly these questions justifying your answers:

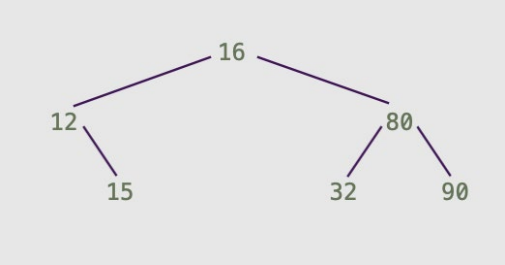
A. Which kind of data structure is a Python dictionary?
B. Which is the command to append a new pair "*key: value*" to a Python dictionary named "*dic*"?
C. Which is the command to change an existing pair "*key: val1*" to become "*key: val2*" for a Python dictionary named "*dic*"?
D. Given a Python dictionary "*dic*" the code below has a for loop that will be executed how many times (how many iterations)?

```
for x in dic:
    print(x)
```

3) **Trees - Tree Traversals**

Given the binary search tree data structure seen in class (here is a copy and also the link: [binSearchTree.py]), you have to create a method *inorderRank* that given a object of this class and a specific number it returns -1 if the number is absent of the tree object, or, otherwise, the position of the number if the tree is traversed using depth-first inorder (consider 0 the first node).

For example, for the tree



The method *inorderRank* delivers 2 for the value 16.

Another example, for the same tree, the method *inorderRank* delivers -1 for the value 17.

```python
class Node:
    def __init__(self, d):
        self.Data, self.Left, self.Right = d, None, None

class Tree:
    def __init__(self, d=None):
        if (d == None): # an empty tree
            self.Root = None
        else:
            self.Root = Node(d)
    def insert(self, d):
        def __insertHere__(n, d):
            if (n.Data > d):   # if no node left insert here
                if (n.Left == None):
                    n.Left = Node(d)
                else:          # or try left child
                    __insertHere__(n.Left, d)
            elif (n.Data < d): # if no node right insert here
                if (n.Right == None):
                    n.Right = Node(d)
                else:          # or try right child
                    __insertHere__(n.Right, d)
        if (self.Root == None): # it was an empty tree
            self.Root = Node(d)
        else:
            if (self.Root.Data > d):          # try left child
                if (self.Root.Left == None):  # if empty insert here
                    self.Root.Left = Node(d)
                else:                         # try left subtree
                    __insertHere__(self.Root.Left, d)
            elif (self.Root.Data < d):        # try right child
                if (self.Root.Right == None): # if empty insert here
                    self.Root.Right = Node(d)
                else:                         # try right subtree
                    __insertHere__(self.Root.Right, d)
    def check(self, d):
        def __check__(n, d):
            if (n == None):
                return False
            elif (n.Data == d):
                return True
            elif (n.Data > d):
                return __check__(n.Left, d)
            elif (n.Data < d):
                return __check__(n.Right, d)
        return __check__(self.Root, d)
    def printInorder(self):
        def __visit__(n):
            if (n != None):
                __visit__(n.Left)
                print(n.Data, end=" ")
                __visit__(n.Right)
        print("\n--------")
        __visit__(self.Root)
        print("\n--------")
    def printPreorder(self):
        def __visit__(n, h):
            if (n != None):
                print("---"*h, n.Data)
                __visit__(n.Left, h+1)
                __visit__(n.Right, h+1)
        print("^^^^^^^^^^")
        __visit__(self.Root, 1)
        print("^^^^^^^^^^")
    def printPostorder(self):
        def __visit__(n, h):
            if (n != None):
                __visit__(n.Left, h+1)
                __visit__(n.Right, h+1)
                print("---"*h, n.Data)
        print("==========")
        __visit__(self.Root, 1)
        print("==========")
```

4) **Brute-Force Algorithm - Create the Difference of Two Sets**

Overview: Given two arrays of integers A with len(A) = n and len(B) = m, create a third array C that includes all elements of A that are not in B.

Example: With A = [6, 5, 2, 4] and B = [4, 3, 5], your algorithm should produce C = [6, 2].

Directions/requirements:

a) Write a brute force function that uses nested for loops to repeatedly check if each element in A matches any of the elements in B. If the element from A does not match any element in B, then copy it into the next available slot of array C.

Your function should have two input parameters (the sets A and B) and should return the set difference (the set C) through the return statement. Be sure to explain the inputs and outputs in your comments.

You can use the Python "in" operator to control the for loops, but do not use the Python "in" operator to test to see if an element of one set is in the other set. You may assume that in each array, each element is listed only once (there are no duplicates within the same array). Do not assume that either array is sorted and do not sort any of the arrays at any time. Number each line of code.

b) Trace the algorithm with

A = [20, 40, 70, 30, 10, 80, 50, 90, 60]

B = [35, 45, 55, 60, 50, 40]

c) Using the line numbers that you assigned to each line of code, create a chart showing the cost of each line and the number of times each line is executed in the worst case. Use this chart to conduct your formal asymptotic analysis. What is the Big-Oh class for this algorithm in terms of m and n?

5) **Master Method**

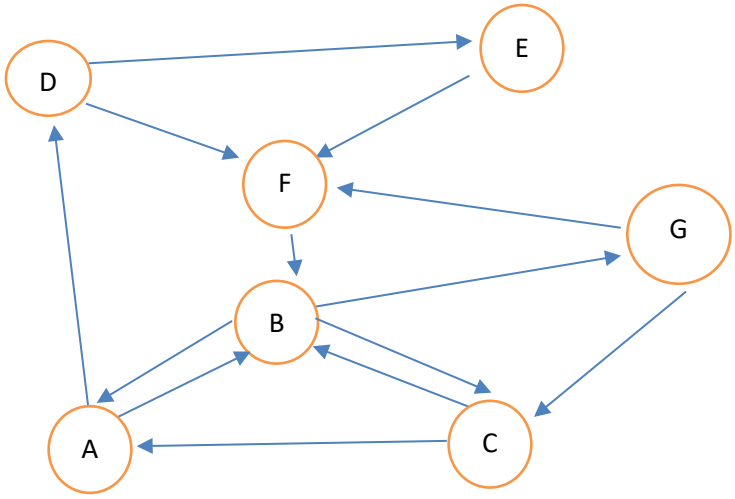
Use the master method to determine the Big-Oh class for an algorithm whose worst-case performance is given by each of these recurrence relations. For each recurrence relation identify the values of a, b, and d; show the result of the relevant inequality; and give the Big-Oh class in terms of the actual numbers in the problem.

a) $T(n) = 4T\left(\frac{n}{2}\right) + n^3$

b) $T(n) = 4T\left(\frac{n}{2}\right) + n^2$

c) $T(n) = 4T\left(\frac{n}{2}\right) + 1$

#6) Breadth First Search and Depth First Search



a) Represent this directed graph using an adjacency matrix.

b) Represent this directed graph using adjacency lists. Arrange the neighbors of each vertex in alphabetical order.

c) Breadth First Search - Show the steps of a breadth first search with the graph given above using the technique given in the class notes. Use the adjacency lists representation that you created in part b. Start at vertex A. Show the contents of the queue each time that a node is added to the queue or removed from the queue. Draw the resulting tree - all the nodes labeled with their processing order and the edge that caused each node to be added to the queue. Finally, list the nodes in the order they are processed. [Give a queue, a tree, and a processing list (as in the worksheet in module 5).]

d) Depth First Search - Show the steps of a depth first search with the graph given above using the technique given in the class notes. Use the adjacency lists representation that you created n part b. Start at vertex A. Draw a doubly linked data structure (like we did in class) that indicates the logic of why each node was processed when it was. Draw the resulting tree - all the nodes labeled with their processing order and the edge that caused each node to be added to the processed. Finally, list the nodes in the order they are processed. [Give a doubly linked data structure, a tree, and a processing list.]

7) Decrease-and-Conquer Algorithm – Maximum Element in Array

a) Write a recursive decrease-and-conquer algorithm to calculate the maximum element in a non-empty array of real numbers. Your algorithm should work by comparing the last element in the array with the maximum of the "remaining front end" of the array.

For example, to find the largest element in the array <5, 13, 9, 10> your algorithm should call itself to find the maximum of <5, 13, 9> and return either 10 or the result of the recursive call, whichever is larger.

Parameters and return: Your function call should call should be

Maximum(A, right)

where the two input parameters are the array and right index. With these input parameters, the function should return the maximum array element from A[0] to A[right]. Using a return statement, return the value of the array element, not the index where it occurs in the array.

Do not use Python's built-in max() function.

Do not rearrange the elements of the array by sorting or partially sorting them.

Do not use any loops.

You can assume that the array has at least one element in it.

b) Trace your algorithm with

A = [17, 62, 49, 73, 26, 51]

For each recursive call , show the parameters involved in the call, show the value returned by the call, show the comparison between the returned value and the element of the array that it gets compared to, and show the result of this comparison.

c) Write a recurrence relation for the number of comparisons of array elements that are performed for a problem of size n. Include the recursive part and the stopping state.

d) Then perform asymptotic analysis using back substitution to determine the Big-Oh class for this algorithm.

8) **Divide-and-Conquer Algorithms – Mergesort and Quicksort**

a) For each of these two sorting algorithms, what is its Big-Oh class in the worst case? Explain briefly.

b) For each of these two sorting algorithms, what is its Big-Oh class in the average case? Explain briefly.

c) Trace the mergesort algorithm for the following array of values.

A = [127, 48, 62, 198, 17, 51, 209, 52]

Rather than keep track of the values of individual variables, follow the "tree format" that was used in the class notes to trace the mergesort algorithm.

d) Trace the quicksort algorithm with Lomuto partitioning for the same array of values.

A = [127, 48, 62, 198, 17, 51, 209, 52]

Indicate the pivots in red as was done in the class notes.