Lab 1

1. Write a function INSERTION-SORT(A) which implements the following algorithm in Python. Print the sorted list at the output.

```
INSERTION-SORT(A)
1  for j = 2 to A.length
2      key = A[j]
3      // Insert A[j] into the sorted sequence A[1 .. j − 1].
4      i = j − 1
5      while i > 0 and A[i] > key
6          A[i + 1] = A[i]
7          i = i − 1
8      A[i + 1] = key
```

2. Using a pencil and a paper, illustrate the operation of ***INSERTION-SORT*** on the array **A = {31, 41, 59, 26, 41, 58}** i.e show the contents of the list A after every while and for iteration. Verify the illustration using this website by executing the code written in question 1, one step at a time.

3. Rewrite the ***INSERTION-SORT*** procedure to sort into decreasing instead of increasing order. Justify your solution.

4. Consider the searching problem:

   Input: A sequence of **n** numbers A = {a_1; a_2, …. , a_n} and a value **v**.
   Output: An index **i** such that ***v = A[i]*** or the special value **NIL** if **v** does not appear in A.

   Write pseudocode for linear search, which scans through the sequence, looking for **v**. Using a loop invariant, prove that your algorithm is correct. Make sure that your loop invariant fulfills the three necessary properties.

5. Write a Python function **min_mod_tuple(A, k)** which accepts two arguments, Python List A = [a_0, a_1, . . . , a_{n-1}] containing **n** positive integers and positive integer **k**, and returns a Python Tuple ***(i, j)*** of two array indices with 0 ≤ i < j < n that minimizes $(a_i \times a_j)$ mod k.

6. Let us pit a faster computer (computer A) running ***insertion sort*** against a slower computer (computer B) running ***merge sort (A sorting algorithm)***. They each must sort an array of 10 million numbers. Suppose that computer A executes 10 billion instructions per second and computer B executes only 10 million instructions per second, so that computer A is 1000 times faster than computer B in raw computing power. To make the

difference even more dramatic, suppose that the world's craftiest programmer codes insertion sort in machine language for computer A, and the resulting code requires $\mathbf{2}n^2$ instructions to sort **n** numbers. Suppose further that just an average programmer implements merge sort, using a high-level language with an inefficient compiler, with the resulting code taking 50 * n * lg(n) instructions. Note: lg means log to the base 2.

a. ***Calculate (using a pencil and a paper) the time taken for computer A and B to sort 10 million numbers.***
b. Generate a ***n versus time*** plot for the computers A and B using PYTHON. Which of the two computers would you prefer and under what conditions? Use this file for guidance related to plotting. Open it using a jupyter notebook.

7. Solve problem set 0 available on piazza.

Lab 2

1. Find and plot the best and worst case running time T(n) for the Insertion Sort algorithm given below. **Display grid lines and properly label the plots.**

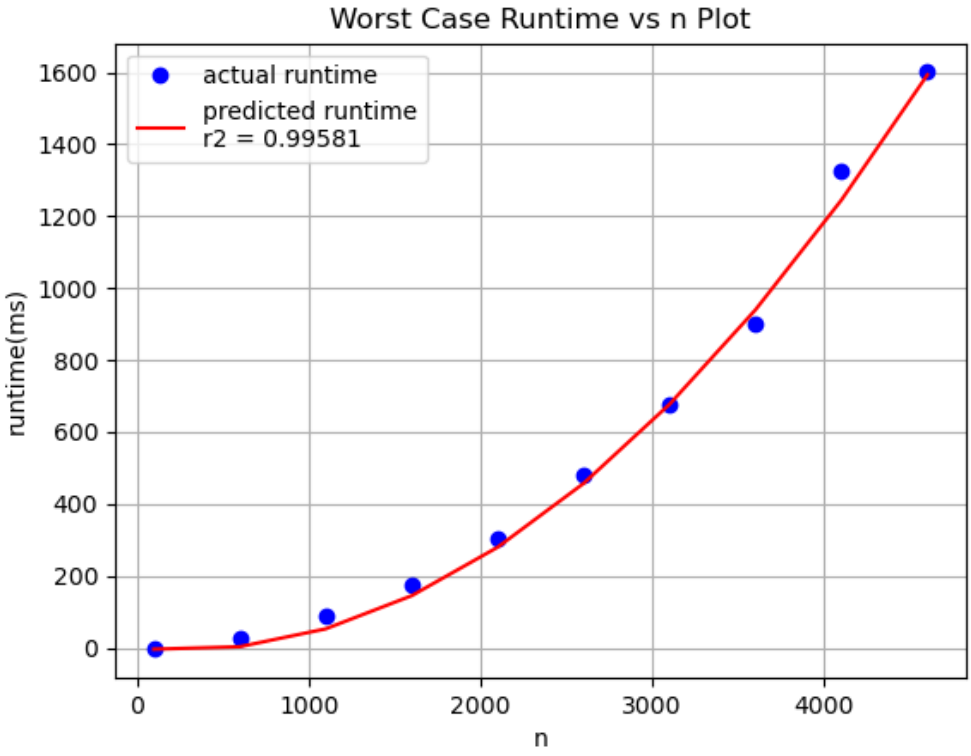
```
INSERTION-SORT(A)
1  for j = 2 to A.length
2      key = A[j]
3      // Insert A[j] into the sorted sequence A[1 .. j − 1].
4      i = j − 1
5      while i > 0 and A[i] > key
6          A[i + 1] = A[i]
7          i = i − 1
8      A[i + 1] = key
```

A. Calculate and plot the **theoretical** best and worst case running times.
B. Write the function **foo()** to calculate and plot the **practical** best and worst case running time. The last 3 arguments of this function represent the start value of n, stop value of n and gap between consecutive values of **n**.

1. Use the **time.time()** to calculate the running time. Use **f-strings** to get a readable output at the console window.
2. To plot one point of T(n) corresponding to any n, generate and apply insertion sort on 10 completely random lists of size n. Calculate the cumulative runtime and average it over 10 trials as shown in the figure below.
3. Generate a curve that fits the best case and worst case curves. How well does it fit? Use **polyfit()** from **numpy** library for curve-fitting and implement **coefficient of determination** R^2 to see how well the curve fits. (see the formula and the worst case plot below)

```
7   def foo(case, start, stop, gap):
8
9
10
11  foo('worstcase', start = 100, stop = 4000,   gap = 500)
12  foo('bestcase' , start = 100, stop = 400000, gap = 50000)
```

```
n = 1100 trialno = 1 runtime = 78.196ms
n = 1100 trialno = 2 runtime = 82.226ms
n = 1100 trialno = 3 runtime = 93.778ms
n = 1100 trialno = 4 runtime = 78.105ms
n = 1100 trialno = 5 runtime = 85.015ms
n = 1100 trialno = 6 runtime = 93.727ms
n = 1100 trialno = 7 runtime = 78.112ms
n = 1100 trialno = 8 runtime = 93.722ms
n = 1100 trialno = 9 runtime = 82.654ms
n = 1100 trialno = 10 runtime = 78.106ms
avg worstcase runtime over 10 trials for n = 1100 is 84.364ms
n = 1600 trialno = 1 runtime = 171.837ms
n = 1600 trialno = 2 runtime = 187.458ms
n = 1600 trialno = 3 runtime = 171.782ms
n = 1600 trialno = 4 runtime = 171.906ms
n = 1600 trialno = 5 runtime = 203.006ms
n = 1600 trialno = 6 runtime = 171.906ms
n = 1600 trialno = 7 runtime = 187.384ms
n = 1600 trialno = 8 runtime = 187.456ms
n = 1600 trialno = 9 runtime = 171.835ms
n = 1600 trialno = 10 runtime = 171.879ms
```



$$R^2 = 1 - \frac{\sum_i (y_i - p_i)^2}{\sum_i (y_i - \mu)^2}$$

Y_i are measured values
P_i are predicted values
μ is mean of measured values

C. Compare the worst case plot in step B with the one plotted by computers of any 3 friends.

2. Consider sorting n numbers stored in array A by first finding the smallest element of A and exchanging it with the element in A[1]. Then find the second smallest element of A, and exchange it with A[2]. **Write pseudocode and code for this algorithm**. Give the best-case and worst-case running times of this algorithm in theeta-notation.

3. Solve problem set 1.

Lab 3

1. Plot the worst case runtime against the size of the list to be sorted using the MergeSort algorithm covered in the theory class.

2. Write a recursive function Sum(L) to sum all the **n** elements of list L.

A. Assume that the sum of n elements of list L is the sum of 1st element and n-1 elements. Recurse on the n-1 elements.

B. Use tree recursion to find the sum of all elements of the list L i.e. divide the list L into two halves, recurse and find the sum.

3. Theoretically, find the runtime of the recursive function you wrote in part a and b in question 2.

4. We can express insertion sort as a recursive procedure as follows. In order to sort A[1…n], we recursively sort A[1… n-1] and then insert A[n] into the sorted array A[1… n-1]. Write a recurrence for the running time of this recursive version of insertion sort and code it as well.

5. You are a game show host. Your guest thinks of a whole number k. Your job is to find the number in O(lg k) steps. You can only ask the question from the guest in this format: “My guess is i. Am I correct? If not, is k > i?”.

6. Emergency service requests are usually dealt with on a first-come first-served basis. In order to request a service, you WhatsApp your CNIC (as the key associated with your complaint) and your complaint.

(a) You could choose a linked list, a double linked list, a static array or a dynamic array as a data structure to store these requests. Which one would you choose and why? Briefly write down the pros and cons of each data structure. Also what API would you offer to the operators of the emergency services?

(b) However the emergency services quickly realize that such simple automation would not do. In order to allow for extreme emergencies, the incharge should be able change the order of applications. Hence the emergency services requires you requests you to change the API to

make complaint_list():
constructs an empty list of complaints in O(1) time
add complaint(x):
adds a complaint with CNIC x to the end of the list of complaints in O(n) time remove
complaint():
removes the complaint from the head of the list of complaints in O(n)
interchange_complaints(x, y):

interchanges the order of complaints with CNIC x and y in O(lg n)

What data structures will you propose and why? Please note that it is up to you to decide whether the list of complaints will be represented by a list or something else. You may propose one or multiple data structures and associated algorithms to construct this API.

Lab 4

1. Referring back to the searching problem (Lab1 , Q4), observe that if the sequence A is sorted, we can check the midpoint of the sequence against and eliminate half of the sequence from further consideration. The binary search algorithm repeats this procedure, halving the size of the remaining portion of the sequence each time. Write code, either iterative or recursive, for **binary search.**

2. Observe that the while loop of the INSERTION -SORT procedure uses a linear search to scan (backward) through the sorted subarray A[1…j-1] . Can we use a binary search instead to improve the overall worst-case running time of insertion sort to O(n lg n)? If yes, code the resulting algorithm.

3. Consider sorting n numbers stored in array A by first finding the smallest element of A and exchanging it with the element in A[1] . Then find the second smallest element of A, and exchange it with A[2] . Continue in this manner for the first n-1 elements of A. Write pseudocode for this algorithm, which is known as **selection sort**. What loop invariant does this algorithm maintain? Why does it need to run for only the first n-1 elements, rather than for all n elements? Give the best-case and worst-case running times of selection sort.

4. In this problem, you will implement a **doubly linked list**, supporting some additional constant-time operations. Each node `x` of a doubly linked list maintains an `x.prev` pointer to the node preceding it in the sequence, in addition to an `x.next` pointer to the node following it in the sequence. A doubly linked list `L` maintains a pointer to `L.tail`, the last node in the sequence, in addition to `L.head`, the first node in the sequence. For this problem, doubly linked lists should not maintain their length.
    a. Given a doubly linked list as described above, code the algorithms to implement the following sequence operations

        ```
        insert first(x), insert last(x), delete first(), delete last()
        ```

5. A question related to implementation of set interface will be updated soon…… (Be patient!!)

Lab 5

1. Determine the output of the following code without using the computer. For the following code, which of the following expressions evaluates to an integer?
    - b.z
    - b.z.z
    - b.z.z.z

```
8   class A:
9       z = -1
10      def f(self, x):
11          return B(x-1)
12
13  class B(A):
14      n = 4
15      def __init__(self, y):
16          if y:
17              self.z = self.f(y)
18          else:
19              self.z = C(y+1)
20
21  class C(B):
22      def f(self, x):
23          return x
24
25  a   = A()
26  b   = B(1)
27  b.n = 5
28  print(C(2).n)
29  print(a.z == C.z)
30  print(a.z == b.z)
```

2. Write a function that takes in a linked list and returns the sum of all its elements.

3. Write a function `store_digits` that takes in an integer `n` and returns a linked list where each element of the list is a digit of `n`.

4. Write a function that takes in a Python list of linked lists and multiplies them element-wise. It should return a new linked list.

5. Implement a function `two_list` that takes in two lists and returns a linked list. The first list contains the values that we want to put in the linked list, and the second list contains the number of each corresponding value. Assume both lists are the same size and have a length of 1 or greater. Assume all elements in the second list are greater than 0.

Lab 6

1. Implement a ***linear time*** algorithm ***Radix Sort*** to sort n integers from the range $[-n^2, \ldots, n^3]$.

2. Melon Usk has discovered a Martian civilization and is trying to learn their language. He needs your help compiling a Martian dictionary. Melon has a list of words in Martian, which—by an incredible coincidence—uses the same alphabet and in the same order as English. However, the Martians alphabetize words differently from Earthlings: instead of sorting first by the first letter, then the second letter, and so on, they sort by the letters in some other order. All Martian words have the same length.

   Your task is to sort Melon's dictionary, given a list of n Martian words **wordlist** and the sorting order of indices order. Each string in **wordlist** has the same length **k** and consists of lowercase letters. order is a permutation of the numbers 0 through k − 1, indicating the precedence of each index (words are first ordered by **word[order[0]]**, and among the words with the same **order[0]**'th letter, they are ordered by **word[order[1]]**, and so on).

   For example, if order is [0,1,...,k-1], you should sort alphabetically in the normal sense. If order is [k-1,...,0,1], you should sort the reverse of the words alphabetically (maybe the Martians read right to left). If order=[0,2,1], then these words are in the correct order:

   bag beg big bar bat bet bit rag rat

a. Describe an algorithm to sort a Martian dictionary as described above, given wordlist and order. Each word is of length k. Argue its correctness and determine its runtime in terms of both n and k. Solutions which are asymptotically faster, when the length of each word k is held constant, will receive more points.
b. Implement **martian sort** using your algorithm from part 1. Hint: Use a modified form of **radix sort**.

3. Solve pset 3.

Lab 7

1. Write a code to implement a **Binary Tree T** with traversal order studied in the class.

2. Write a function **cumulative_mul** that mutates the Tree **T** so that each node's label becomes the product of its label and all labels in the subtrees rooted at the node.

3. You are given a binary tree T with n nodes, where each node has a unique integer item between 0 and n − 1. Give an O(n) time algorithm that takes as input such a binary tree T, and outputs an array A of length n, where A[i] stores the size of the subtree rooted at the node with item i. Code your algorithm, and prove its time complexity.
   For a tree **T**, you may assume that you can obtain the root of the tree **T.root** in constant time. For a tree node **x**, you may obtain its left child (x.left) or right child (x.right) in constant time (these fields will store the value None if the child does not exist). You may also obtain the item of node x **(x.item)** in constant time.

4. Given the first n natural numbers, [1, 2, ..., n], we can find a sequence in which these numbers should be inserted into an empty AVL Tree such that we would require 0 rotations to maintain height balance. Given some positive integer n, describe and implement an O(n) time algorithm to produce such a sequence. Your output should be an array containing all the natural numbers up to n (inclusive) in the order they should be inserted.

Lab 8

1. Code a new function ***subtree_mean(node)*** that computes the average of all the keys in the subtree rooted at node. This function should be a **O(1)** function.

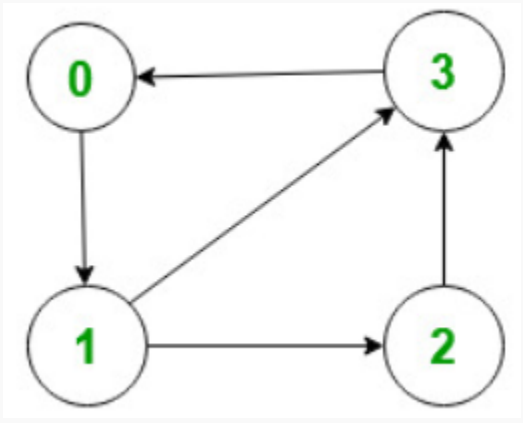
2. A watch shop is maintaining its inventory of **n** watches using AVL trees. The keys are the prices of the watches being sold. Given a price **p**, design a function to return the **k** most-expensive watches whose price is less than **p** efficiently?

3. Car customers come to a showroom to buy the most reliable car within their budget. Each car has a price (the price is in the integer number of millions and varies from Rs. 1M-10M) and a reliability number (between 0 and 10, with 10 being the most reliable). You have been hired by a car showroom to design an ADT using a data structure that supports the following operations in O(lg n): buy(car) (adds a car to the inventory), sell(car) (removes the car from the inventory), find_car(budget) (which gives you the most reliable car within your budget i.e. the most reliable car with price ≤ budget), car_options(reliability) which finds for you the cheapest car that has a reliability ≥ reliability. Suggest and **code** a data structure and show how it supports the above operations in O(lg n) time.

Lab 9

1. Given the adjacency list and number of vertices and edges of a graph, the task is to represent the adjacency list for a directed graph.

Input: *V = 4, edges[ ][ ] = {{0, 1}, {1, 2}, {1, 3}, {2, 3}, {3, 0}}*



Output: *0 -> 1*
1 -> 2 3
2 -> 3
3 -> 0

2. Consider the undirected graph G described by the following adjacency matrix, where Adj [i] [j] = 1 indicates an edge from vertex i to vertex j:

```
#        0  1  2  3  4  5
Adj = [[0, 0, 1, 1, 0, 1], # 0
       [0, 0, 0, 1, 1, 0], # 1
       [1, 0, 0, 0, 0, 1], # 2
       [1, 1, 0, 0, 0, 1], # 3
       [0, 1, 0, 0, 0, 0], # 4
       [1, 0, 1, 1, 0, 0]] # 5
```

a. Draw the graph represented by the above adjacency matrix.

b. What is the adjacency list representation of G? Express this as a Python dictionary where the element at i is a list containing i's neighbors in increasing order

c. Perform breadth-first search on G, starting at node 0. Visit the neighbors of each vertex in increasing order. What order are vertices first visited? Print them at the output.

d. Perform depth-first search on G, starting at node 0. Visit the neighbors of each vertex in increasing order. What order are vertices first visited? Print them at the output.

e. Which single edges can you remove from G to make a disconnected graph? For each one, what are the connected components after removing the edge? Describe each connected component as a list of indices, e.g. [1,3] if the component contains vertices 1 and 3.

Lab 10

1. Modify Dijkstra's algorithm to account for nodal delays in determining shortest path. Nodal delay refers to the weight of a node. What you've studied in theory till now has dealt with graphs in which edges have weights, but nodes do not. In this lab problem, each node has a weight that adds to the cost of the paths passing through that node. Storing these weights in a list with length equal to the number of nodes will make this problem easier.

Lab 11

1. Implement a hash table using open-addressing. Use linear probing. Rehash when hash table is 66% full.
2. Use quadratic probing.