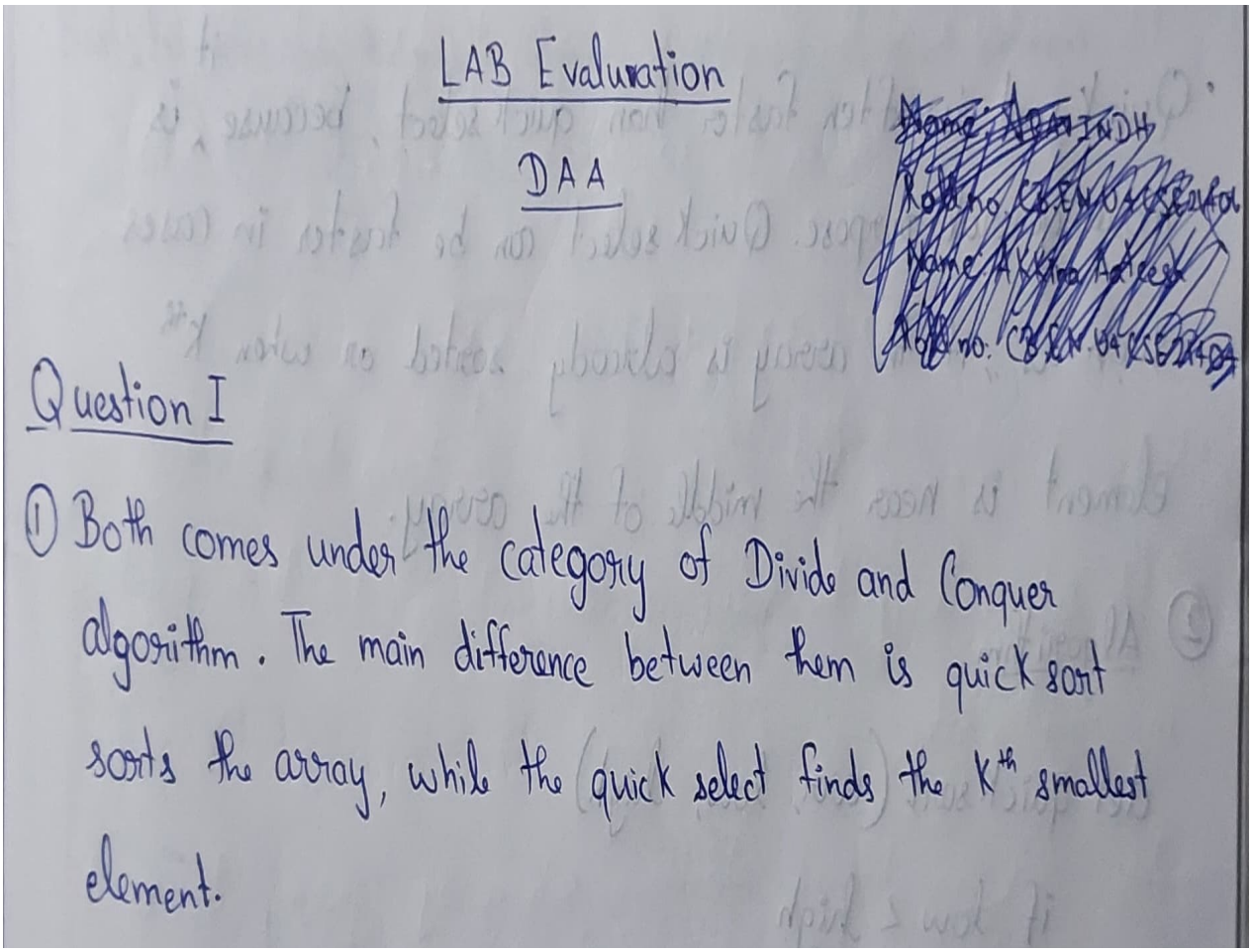
DAA LAB EVALUATION - 1

TEAM MATES	ROLL NO
NS Surya	CB.EN.U4CSE21461
Ramnaresh U	CB.EN.U4CSE21447
ARAVINDH	CB.EN.U4CSE21406

LAB EVALUATION 1

Q1

1)



LAB Evaluation

DAA

Question I

① Both comes under the category of Divide and Conquer algorithm. The main difference between them is quick sort sorts the array, while the quick select finds the k^{th} smallest element.

- quick sort algorithm chooses the pivot element from the array, partition the array around the pivot element, such that all elements smaller than the pivot are on the left side of the pivot and elements larger than pivot on right side of the pivot.
- And then, recursively sort the left and right subarrays.
- While on the other hand, quick select, it follows the initial step of quick sort algorithm, and then if the index of the pivot element is equal to k, then return the pivot element. Otherwise, recursively sort the sub array that contains the k^{th} smallest element.
- Quick sort is often faster than quick select, because it is more general-purpose. Quick select can be faster in cases such as, if the array is already sorted or when k^{th} element is near the middle of the array.

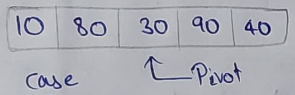
② Algorithm

```
def quicksort (arr, low, high)
    if low < high
        pi = part (arr, low, high)
        quicksort (arr, Low, piindx - 1)
        quicksort (arr, piindx + 1, high)
```

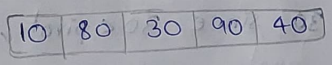
2)

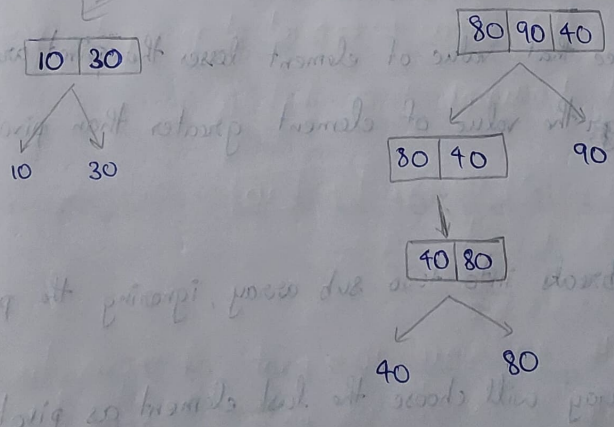
```
def part (arr, low, high):
    pivot = arr[high]
    i = low - 1
    for j in range (low, high):
        if arr[j] <= pivot:
            i = i+1
            arr[i], arr[j] = arr[j], arr[i]
    arr[i+1], arr[high] = arr[high], arr[i+1]
    return i+1
```

Best case Scenario:



- Best happens only the middle element is choosen as pivot element.





- Recurrance Relation for quick sort

$$T(n) = T(k) + T(n-k-1) + O(n)$$

where,

$T(n)$ = is the time it takes to sort an array of size n

k is the no. of element that are smaller than pivot.

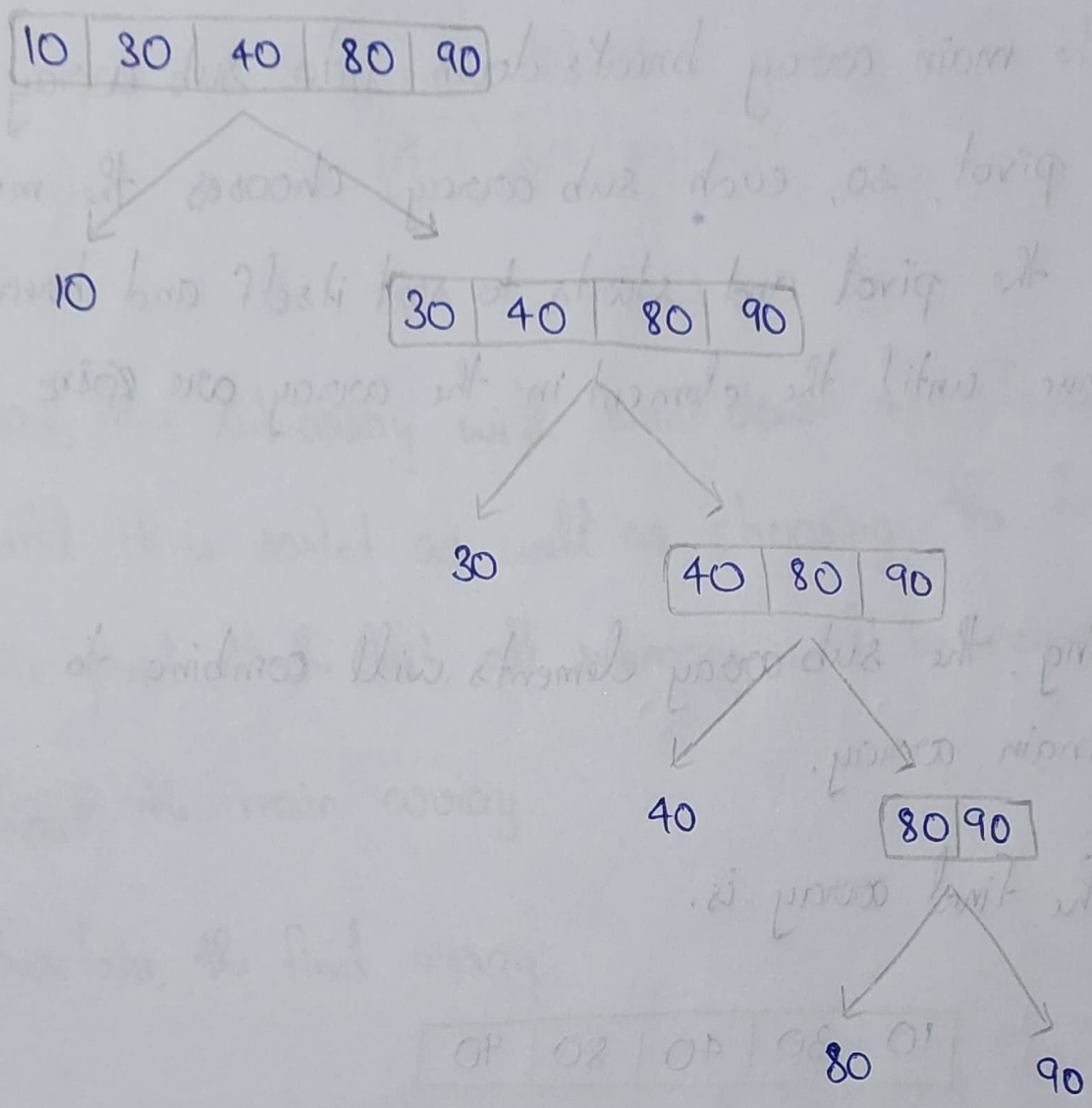
So,

$$T(n) = T(n/2) + T(n/2) + O(n)$$

$$= 2 * T(n/2) + O(n)$$

$$= O(n \log n)$$

- Worst case happens when the given array is already sorted, and 1st element as pivot.



- In this case K=0,

$$T(n) = 0 + T(n-1) + O(n)$$

$$= O(n) + O(n)$$

$$= O(n^2)$$

3)

③ for quick sort, it is, Contact list sorting, Music playlist sorting, etc.

for quick select, it is GPS navigation, order statistics in finance.

④ ① Last element as the pivot

⟹

10	80	30	90	40

↑ Pivot

- It compares with 10, since 10 < pivot, it moves to 80, since 80 > pivot, it swaps.

10	40	30	90	80

- It comparses with 80, since 80 > pivot, it moves to 90, since 90 > pivot, it moves to 30, since 30 < pivot, it swaps.

10	30	40	90	80

- Now, we can see that value of element lesser than pivot are in left side and the value of element greater than pivot are in right side.
- Now, the given breaks into two sub array, ignoring the pivot element
- Now, each subarray will choose the last element as pivot and start to get sort like main array, ~~after sorting, it~~ the sub array will break down into few more subarray, until the array

4)

is sorted, after sorting, it is combined to form the main array which is the array.

- Therefore, the final array is

10	30	40	80	90

(ii) First element as pivot

⟹

10	80	30	90	40

↑ Pivot

- It compares with 40, since 40 > pivot, it moves to 90, since 90 > pivot, it moves to 30, since 30 > pivot, it moves to 80, since 80 > pivot, it moves to 10.
- From the above scenario we could see that, pivot is the least element.
- So now, the main array breaks into a sub array ignoring the pivot element, so the sub array is

80	30	90	40

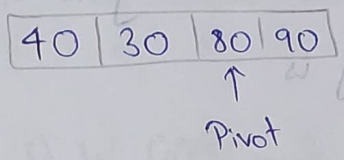
↑ Pivot

- It compares with 40, since 40 < pivot it swaps.

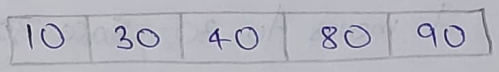
40	30	90	80

↑

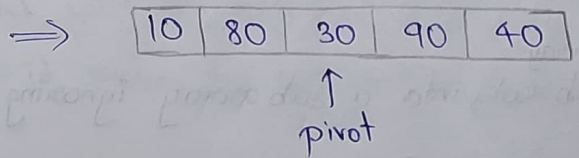
- It compares with 40, since 40 < pivot, it moves to 30, since 30 < pivot, it moves to 90, since 90 > pivot, it swaps.



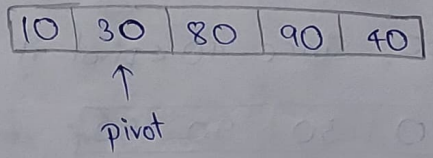
- Now, this subarray will break down into fewer subarray until it is sorted as well as choosing the first element as pivot. At the last, all subarray will be combined to form the main array.
- Therefore, the final array,



(iii) Middle element as pivot.



- It compares with 10, since 10 < pivot, it moves to 80, since 80 > pivot it swaps.



- It compares with 40, since 40 > pivot, it moves to 90, it compares with 90, since 90 > pivot, it moves to 80, since 80 > pivot, it moves to 30.

- So, now, the main array breaks down into sub array ignoring the pivot, so, each sub array chooses the mid element as the pivot and starts to sort itself and break will continue until the element in the array are sorting.
- After sorting, the sub array elements will combine to form the main array.
- Therefore the final array is.

10	30	40	80	90

5)

⑤ So, to choose the Algo b/w A or B, we first need to calculate the time consuming of each Algo: which is

A: 10.24 ms for 1024 selection

B: 1.024 ms for 512 selection

$= 1.024 \times 512 = 512.288$

Since, the processing time for Algo A is faster than Algo B.

Q2)

Question:

The Time complexity of sorting n data items with insertion Sort is $O(n^2)$; that is, the processing time of insertion Sort is $c_s \cdot n^2$ where c_s is a constant scale factor. The time complexity of merging k subarrays pre-sorted in ascending order and containing n items in total is O(k . n). The processing time is $c_m(k-1) \cdot n$ because at each step t = 1, 2, . . . , n you compare k current top items in all the subarrays (that is, (k − 1) comparisons), pick up the maximum item, and place it in position (n − t + 1) in the final sorted array. You save time by splitting the initial array of size n into k smaller subarrays, sorting each subarray by insertion Sort, and merging the sorted subarrays in a single sorted array. Let the scale factors c_s and c_m be equal: $c_s = c_m = c$. Analyse whether you can accelerate the sorting of n items in the following way:

- Split the initial array of size n into k subarrays of size n/k. The last subarray may be longer to preserve the total number of items n, but do not go into such detail in your analysis.
- Sort each subarray separately by Insertion Sort.
- Merge the sorted subarrays into a final sorted array. If the acceleration is possible, then find the optimum value of k giving the best acceleration and compare the resulting time complexity of the resulting sorting algorithm to that of Insertion Sort and Merge Sort.

Answer:

We know that implementing the traditional insertion sort takes a time complexity of $O(n^2)$ for the worst case and the average case. So we have been asked to improve the traditional algorithm by implementing a hybrid sort which combines insertion sort and certain merging techniques which reduces the time complexity drastically as we will see in the graphs which we have implemented for the sake of comparison. So lets get started with implementing our hybrid sort. For this we have 3 functions which allows us to solve the function with a lesser time complexity. The first function is the split_array function which helps us to split the array into k parts with the average size being n/k. The time complexity of this split function is O(n).After this we implement the insertion sort function on each of these arrays of size n/k . The time complexity is $O(k.(n/k)^2)$.

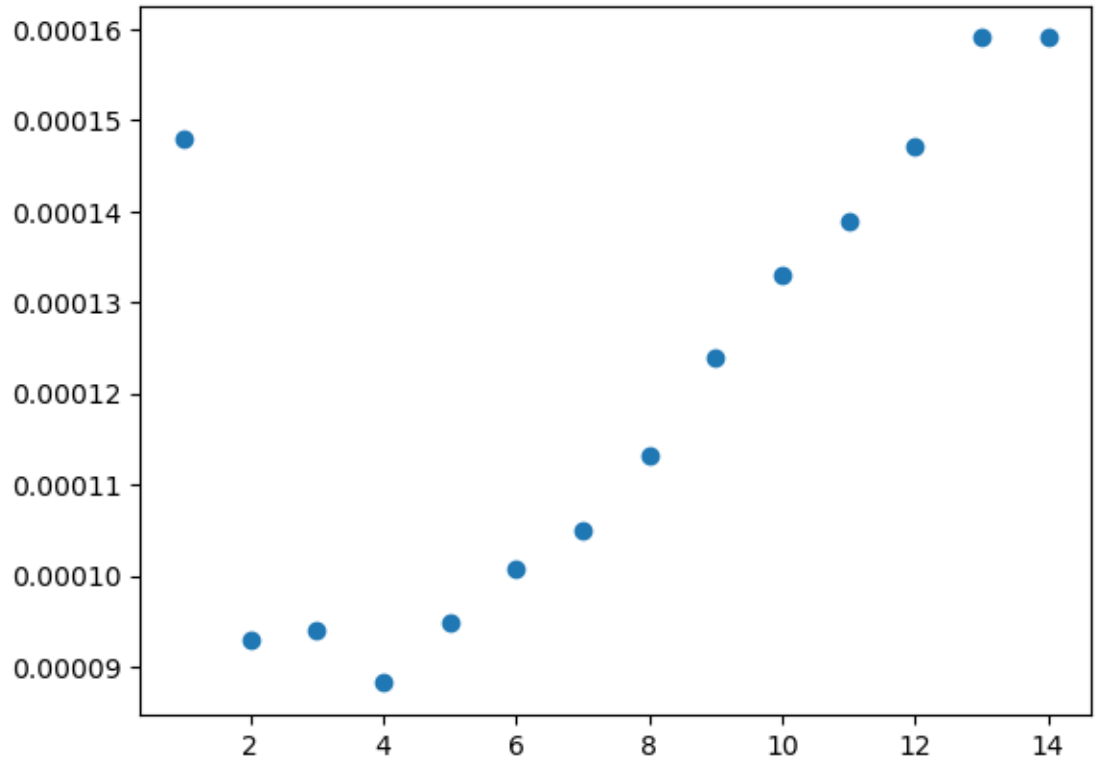
Here itself we see that the time complexity of the insertion sort operation has been drastically reduced. After this we take each of the sorted arrays and merge them using a technique of having pointers to each of the arrays which helps us in reducing the time complexity to O(k.n) as we are only comparing the top pointer of the subarrays and moving each one if the condition is satisfied. We are not iterating subarray by subarray due to which the time complexity doesn't shoot up. When we combine these different functions we get a hybrid sort which is far better than the original insertion sort as we can see from the graph below.

O(n) for splitting

$O(k.(n/k)^2)$ for insertion sort

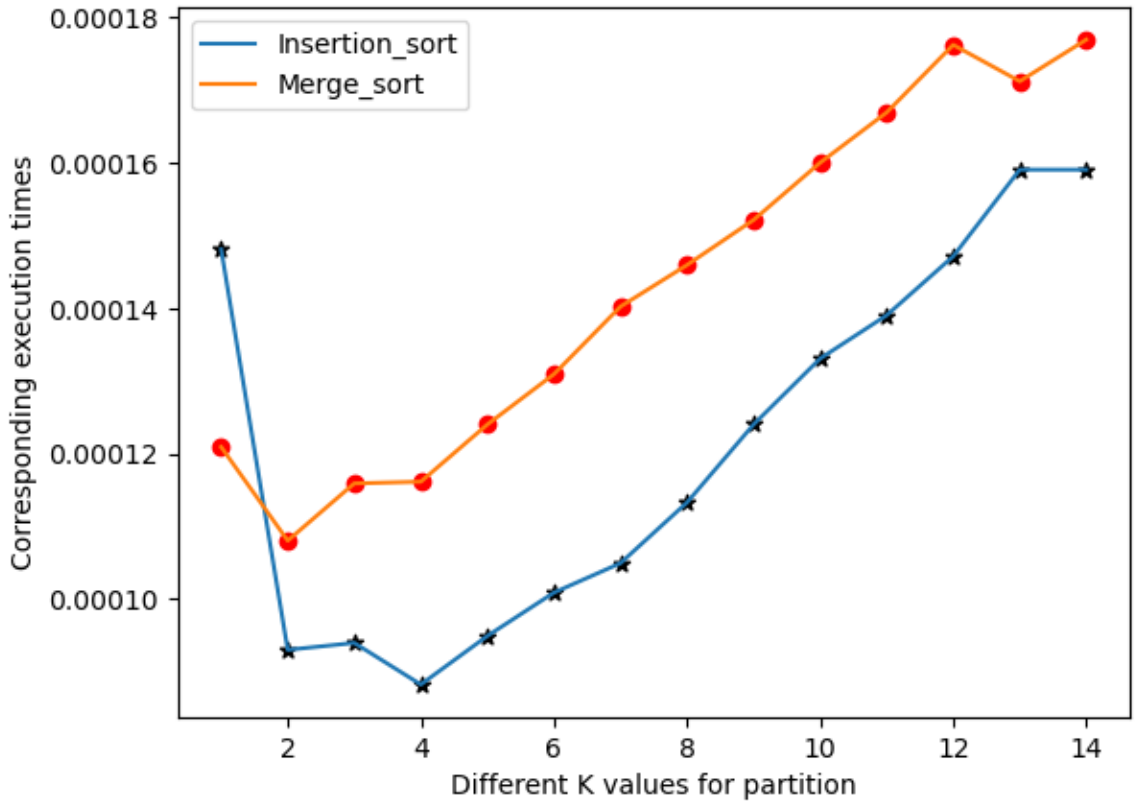
O(k.n) for merging

Since the dominant term is the insertion sort term the final time complexity is $O(k.(n/k)^2)$. This is what has drastically reduced the time for execution . From a lot of experimentation we have found out that for n=100 the best k value is 4 for n=1000 it is 30. Hence the best k value is something like n/25 for the right optimization.



After this we compare insertion sort with merge sort and the interesting result which we have obtained is that in the case of application of traditional insertion sort and traditional merge sort the time taken for merge sort is far lesser than that of insertion sort rather when we start splitting the arrays, we see that insertion sort takes lesser amount of time and quite a find as it shows the various intricacies of these data structures and these sorting algorithms.

`



Question 3.A

The problem

The problem is to prove whether the given algorithm can give the minimum spanning tree for a given connected graph, by removing the edges that do not disconnect the graph. The unremoved edges give the edges of the MST.

Given Algorithm

"Initially all the edges are unmarked. Let e be an unmarked edge with the highest weight. If removing e does not disconnect the graph, remove it; otherwise, mark it. Repeat this process until the graph consists of marked edges only. Then, output the current graph as an MST of the original graph."

Approach to the Problem

The important computation involved in this algorithm is determining if the chosen edge if removed disconnects the graph or not.

The equipped approach to detect such edges is to perform a DFS which counts the no. of connected vertices, then compare this no. with the no. of the connected vertices in the graph.

To make the resulting tree as a minimum spanning tree, edges are sorted in descending order according to their weights and then checked if they form a bridge or not. This ensure the left-out edges that are essential to keep the graph connected are minimum in weights.

Time Complexity Analysis

Sorting of the edges according to their weights take a complexity of O(E * log E).

Then for each edge removal a DFS traversal is performed, and the count is noted. The time complexity for DFS is O(V + E). Performing it over E edges takes

O(E * (V+E)) time complexity

In total, including the sorting, the time complexity for the given algorithm is:

O(E * log E + E * (V+E)).

Result and Observations

The given algorithm indeed finds the MST of a graph, verified with our custom test cases.

One such test case is:

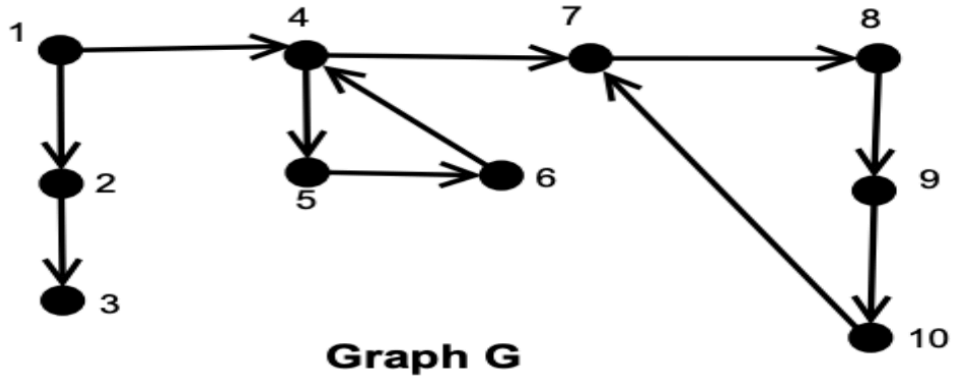
```
vertices = ['A', 'B', 'C', 'D', 'E', 'F', 'G']

edges = [('A', 'B', 2), ('A', 'C', 3), ('A', 'D', 5), ('B', 'D', 7), ('B', 'E', 9), ('C', 'D', 6), ('C', 'F', 8), ('D', 'E', 4), ('D', 'F', 6), ('D', 'G', 4), ('E', 'G', 1), ('F', 'G', 8)]
```

The naïve version of the algorithm performs poor when compared to Kruskal's and Prim's algorithms, but a modified version of Reverse Delete Algorithm works better than its counterparts.

Question 3.B

The Problem

The problem is to implement the given algorithm and hence, provide the strongly connected components of the following graph:



Graph G

Given Algorithm

Input: Directed Graph G
Output: Strongly Connected Components of G.

Step 1: Run DFS(G) and compute the finish time for all vertices.
Step 2: Find G^T and sort the vertex set of G in decreasing order based on its finish time.
Step 3: Run DFS(G^T) from the vertex which has maximum finish time. Do this step repeatedly until all the vertices in G^T are visited at least once (this gives you the collection of SCC.
Step 4: Each forest in DFS(G) is a SCC.

Approach to the Problem

Implementation of the problem is done exactly as mentioned in the given algorithm. The algorithm equips DFS traversal extensively on both given graph and the transpose of the given graph. Discovery time and finish time for each vertices is implemented in DFS to conveniently sort the vertices in a stack fashion.

Time Complexity Analysis

DFS runs on O(E + V), sorting on the finish time for each vertices takes O(V * log V).