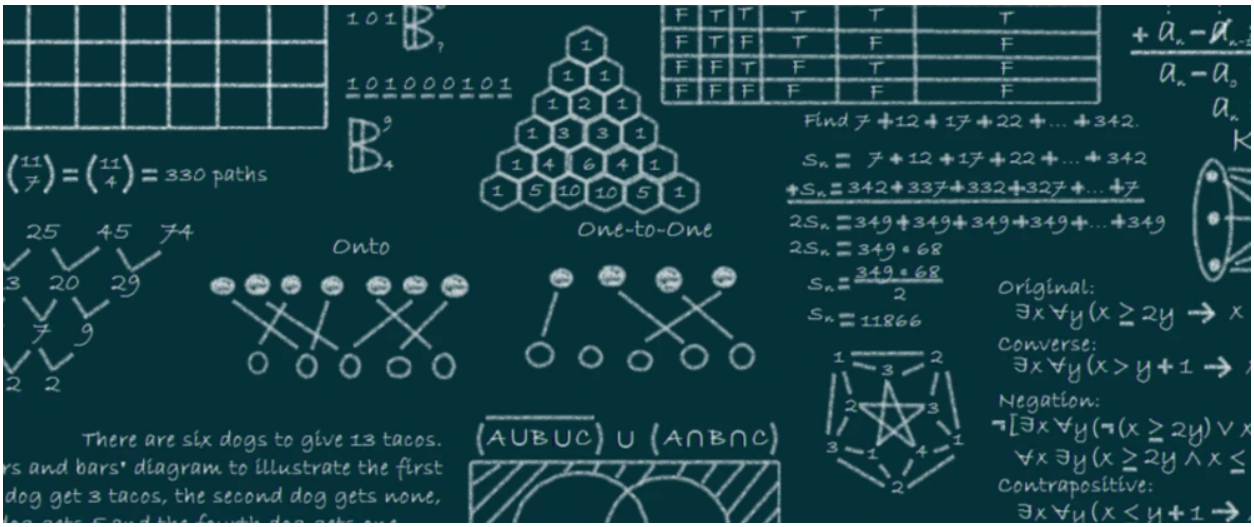


19CSE302: Design and Analysis of Algorithms

Lab Evaluation 1

Team:

Roll Number	Name
CB.EN.U4CSE21425	Joseph Jeffrey J
CB.EN.U4CSE21453	Sanjay Balamurugan

Question 1

1) Difference between Quick Sort and Quick Select algorithms.

Quick Sort	Quick Select
It is mainly used for sorting an array of elements.	It is used for finding the kth smallest element.

Quick Sort	Quick Select
This algorithm selects a "pivot" element from the array which partitions into two sub-arrays.	This algorithm uses the "pivot" element but focuses only on the sub-array that contains the desired kth element.
The sub-arrays are then recursively sorted.	It recursively narrows down the search to one of the sub-arrays until it finds the kth element.
It has an average and best-case time complexity of O(n log n).	It has an average and best-case time complexity of O(n).
It has a worst-case time complexity of O(n^2).	The worst-case time complexity of the Quick Select algorithm is O(n^2).
It is commonly used for sorting data in various applications.	It is commonly used to quickly find specific values, like finding the median of a dataset or selecting the top k elements in a list.

Implementation of Quick Sort

```
def quick_sort(arr):
    if len(arr) <= 1:
        return arr

    pivot = arr[len(arr) // 2]
    left = [x for x in arr if x < pivot]
    middle = [x for x in arr if x == pivot]
    right = [x for x in arr if x > pivot]

    return quick_sort(left) + middle + quick_sort(right)

arr = [3, 6, 8, 10, 1, 2, 1]
sorted_arr = quick_sort(arr)
print("Sorted Array:", sorted_arr)
```

Implementation of Quick Select

```
def partition(arr, low, high):
    pivot = arr[high]
    i = low - 1

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

```
    arr[i + 1], arr[high] = arr[high], arr[i + 1]
    return i + 1

def quick_select(arr, low, high, k):
    if low <= high:
        pivot_index = partition(arr, low, high)
        if pivot_index == k:
            return arr[pivot_index]
        elif pivot_index < k:
            return quick_select(arr, pivot_index + 1, high, k)
        else:
            return quick_select(arr, low, pivot_index - 1, k)

def find_kth_smallest(arr, k):
    if 0 < k <= len(arr):
        return quick_select(arr, 0, len(arr) - 1, k - 1)
    else:
        return None

arr = [3, 1, 4, 2, 5]
k = 3
print(f"{k}th Smallest Element:", find_kth_smallest(arr, k))
```

2) Time complexity of the algorithms (Best/Worst Cases). Frame the recurrence relations for each of the cases and solve to compute the time complexity

Time Complexity	Quick Select	Quick Sort
Best Case	Time complexity of O(n).	Time complexity of O(n log n).
Worst Case	Time complexity of O(n^2).	Time complexity of O(n^2).

Quick Sort

Recurrence Relation

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

- `T(k)` represents the time taken to recursively sort the left subarray which contains k elements.
- `T(n-k-1)` represents the time taken to recursively sort the right subarray which contains (n-k-1) elements.

- `O(n)` represents the time taken for the partitioning step, where we compare and rearrange elements in the array based on the pivot.

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

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

= O(n*log(n))

Time Complexity is O(n*log(n))

Best Case Scenario

[8, 3, 1, 7, 6, 4, 5, 2]

- Choose pivot as 4
- It divides into 2 sub-array [3, 1, 2] (Less than the pivot) and [8, 7, 6, 5] (More than the pivot).
- We recursively apply the Quick Sort algorithm to the left and right subarrays.
- From the Left Subarray '[3, 1, 2]'
  - Choose pivot as 2
  - it divides into 2 sub-array [1] (Less than the pivot) and [3] (More than the pivot).
- From the Right Subarray '[8, 7, 6, 5]'
  - Choose pivot as 5
  - it divides into 2 sub-array [] (Less than the pivot) and [8, 7, 6] (More than the pivot).
    - Choose pivot as 6
    - it divides into 2 sub-array [] (Less than the pivot) and [8, 7] (More than the pivot).
      - Choose pivot as 7
      - it divides into 2 sub-array [] (Less than the pivot) and [8] (More than the pivot).
- Final sorted array - [1, 2, 3, 4, 5, 6, 7, 8]

Worst Case Scenario

[8, 3, 1, 7, 6, 4, 5, 2]

- For the worst case, always select either the smallest or the largest element as the pivot.
- Choose pivot as 8
- It divides into 2 sub-array [3, 1, 7, 6, 4, 5, 2] (Less than the pivot) and [8] (Pivot itself).
- We recursively apply the Quick Sort algorithm to the left subarray.
- From the Left Subarray '[3, 1, 7, 6, 4, 5, 2]'
  - Choose pivot as 7
  - It divides into 2 sub-array [3, 1, 6, 4, 5, 2] (Less than the pivot) and [7] (Pivot itself).
  - From the Left Subarray '[3, 1, 6, 4, 5, 2]'
    - Choose pivot as 6
    - It divides into 2 sub-array [3, 1, 4, 5, 2] (Less than the pivot) and [6] (Pivot itself).
    - From the Left Subarray '[3, 1, 4, 5, 2]'
    - Continue this process.
- The Final sorted array [8, 7, 6, 5, 4, 3, 2, 1]

Quick Select

Recurrence Relation

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

- `T(n)` is the time complexity to find the kth element in an array of size `n`.
- `T(n/2)` represents the time spent on average in the recursive call for a subarray of size `n/2`.
- `O(n)` represents the time spent on partitioning the array and selecting the pivot.

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

= O(n)

Time Complexity is O(n)

Best Case Scenario

[6, 10, 2, 8, 3, 5, 7, 1, 4, 9]

- Choose the pivot as 5.
- It divides into 2 sub-array **[2, 3, 1, 4]** (Less than the pivot) and **[6, 10, 8, 7, 9]** (More than the pivot).
- We recursively apply the Quick Select algorithm to the left and right subarrays.
- Our goal is to find the 4th Smallest Element (i.e. 4)
- From the Left Subarray '**[2, 3, 1, 4]**'
    - Choose pivot as 3.
    - it divides into 2 sub-array [2,1] (Less than the pivot) and [4] (More than the pivot).
        - Choose pivot as 1.
        - it divides into left sub-array [ ] (Less than the pivot).
        - Choose pivot as 2.
        - it divides into left sub-array [ ] (Less than the pivot).
    - Choose pivot as 4.
    - it divides into 2 sub-array [ ] (Less than the pivot) and [ ] (More than the pivot).
    - Finally the 4th Smallest Element is 4.

Worst Case Scenario

[6, 10, 2, 8, 3, 5, 7, 1, 4, 9]

- For the worst case, always select the largest element as the pivot.
- Choose the pivot as 10.
- It divides into sub-array [6, 2, 8, 3, 5, 7, 1, 4, 9] (Less than the pivot)
- We recursively apply the Quick Select algorithm to the left subarray.

3) List a few use cases of both algorithms

Quick Sort

1. **Database Management**: Itcan be used to sort data in databases efficiently.
2. **File System Sorting**: It helps in quickly listing files and directories in alphabetical order.
3. **Network Routing**: It can be used in network routing algorithms to efficiently sort and process routing tables or data packets.
4. **Search Algorithms**: It can be used as a subroutine in various search algorithms to quickly reorganize data for efficient searching.
5. **Data Analytics**: It is used in data analytics pipelines to preprocess and sort data before running analytical algorithms and generating reports.

Quick Select

1. **Selection Algorithms**: It is used to find the top-k elements in a list or array.
2. **Database Queries**: It is used for finding the kth highest or lowest salary from an employee database.
3. **Partial Sorting**: It is used to identify and isolate the k smallest or largest elements without sorting the entire dataset.
4. **Searching in Unordered Data**: It can be used as a preprocessing step to efficiently search for elements in an unordered dataset. By finding the median or a pivot element, it can narrow down the search space quickly.
5. **Network Routing**: It can be employed in network routing algorithms to find optimal paths or routes based on specific criteria or metrics.

4) What are the different pivot selection methods. Give a demo of atleast 3 methods with appropriate examples.

Three Methods of pivot selection

1. **Random Pivot Selection**
2. M**edian of Three Pivot Selection**
3. **First or Last Element as Pivot**

Example :

1. Random Pivot Selection
    - **[7, 2, 1, 6, 8, 5, 4, 3]**
    - In this method we can consider the pivot at any position.
2. Median of Three Pivot Selection
    - **[5, 2, 1, 6, 8, 4, 7, 3]**
    - We choose the first , middle and last position values as pivot (i.e. 5, 6, 3).
    - So, Taking median (5 + 6 + 3) / 3 = 4.67
    - We round off the value and take 5 as the pivot.
3. **First or Last Element as Pivot**
    - **[3, 6, 1, 2, 8, 4, 7, 5]**
    - In this method we either take the first element or the last element as the pivot.

5) You have to split an unsorted integer array, Wn, of size n into the two subarrays, Uk and Vn−k, of sizes k and n − k, respectively, such that all the values in the subarray Uk are smaller than the values in the subarray Vn−k. You may use the following two algorithms:

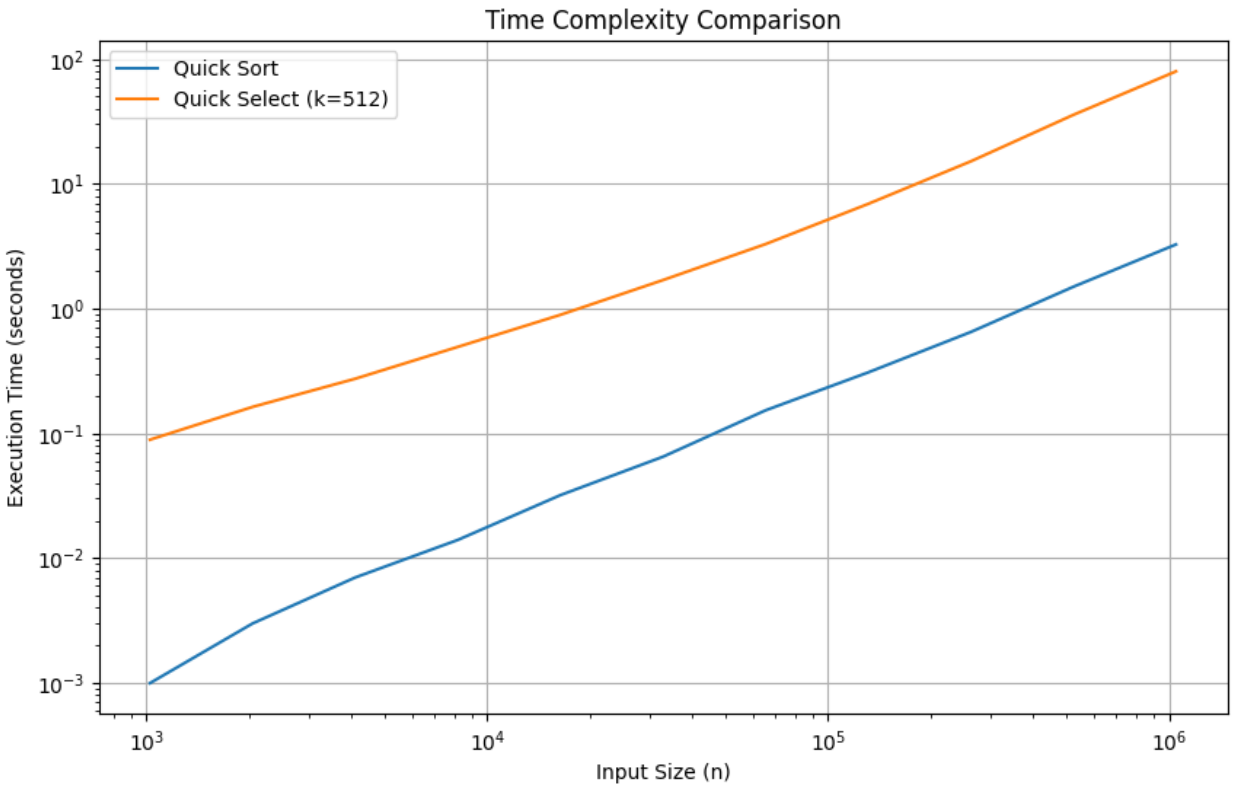
A.) Sort the array Wn with QuickSort and then fetch the smallest k items from the positions i=0,1,...,k−1 of the sorted array to form Uk (you should ignore the time of data fetching comparing to the sorting time)

B.) Directly select k smallest items with QuickSelect from the unsorted array (that is, use QuickSelect k times to find items that

might be placed to the positions i = 0, 1, . . . , k − 1 of the sorted array).

```
import random
import time
import matplotlib.pyplot as plt

def quick_sort(arr):
    if len(arr) <= 1:
        return arr
    pivot = arr[random.randint(0, len(arr) - 1)]
    left = [x for x in arr if x < pivot]
    middle = [x for x in arr if x == pivot]
    right = [x for x in arr if x > pivot]
    return quick_sort(left) + middle + quick_sort(right)

def quick_select(arr, k):
    if len(arr) == 1:
        return arr[0]

    pivot = arr[random.randint(0, len(arr) - 1)]
    left = [x for x in arr if x < pivot]
    right = [x for x in arr if x > pivot]
    pivot_count = len(arr) - len(left) - len(right)

    if k < len(left):
        return quick_select(left, k)
    elif k < len(left) + pivot_count:
        return pivot
    else:
        return quick_select(right, k - len(left) - pivot_count)

# Measure execution time for different input sizes
input_sizes = [2**i for i in range(10, 21)]  # Input sizes from 2^10 to 2^20
quick_sort_times = []
quick_select_times = []

for n in input_sizes:
    arr = [random.randint(1, 1000000) for _ in range(n)]
    a = arr.copy()

    # Measure Quick Sort time
    start_time = time.time()
    quick_sort(arr)
    end_time = time.time()
    quick_sort_times.append(end_time - start_time)

    # Measure Quick Select time for k = 512
    k = 512
    start_time = time.time()
    for _ in range(k):
```

```
        quick_select(a, k)
    end_time = time.time()
    quick_select_times.append(end_time - start_time)

# Plot the time complexities
plt.figure(figsize=(10, 6))
plt.plot(input_sizes, quick_sort_times, label='Quick Sort')
plt.plot(input_sizes, quick_select_times, label='Quick Select (k=512)')
plt.xlabel('Input Size (n)')
plt.ylabel('Execution Time (seconds)')
plt.title('Time Complexity Comparison')
plt.legend()
plt.grid(True)
plt.yscale('log')
plt.xscale('log')
plt.show()
```



Time Complexity after considering both QuickSort and QuickSelect with `k` calls, is approximately O(n log n) on average, assuming good random pivot selection.

Question 2

Algorithm:

K-Split Insertion Sort Algorithm:

The K-Split Insertion Sort is a variation of the classic Insertion Sort algorithm designed to efficiently sort an array by dividing it into K subarrays and sorting each subarray separately before merging them back into a single sorted array.

Step 1: Input Parameters

We start with an unsorted array, 'arr,' and a value 'k' that represents the number of subarrays we want to create.

Step 2: Split the Array

First, we calculate the size of each subarray by dividing the total number of elements in 'arr' by 'k.'
If there are any remaining elements when dividing 'arr' into 'k' subarrays, we distribute them evenly among the subarrays.
We initialize a variable 'start' to keep track of where we start creating each subarray.

Step 3: Sorting Subarrays

Now, we loop through 'k' times, creating and sorting each subarray one by one.
For each iteration, we determine the 'end' position of the subarray.
We extract the elements between 'start' and 'end' to form a subarray.
We use the Insertion Sort algorithm to sort this subarray.
Step 4: Merge Sorted Subarrays

After sorting all 'k' subarrays, we merge them back into a single sorted array.
We maintain a separate pointer for each subarray to keep track of the current element

we are considering.
In each iteration, we find the smallest element among the elements pointed to by the 'k' pointers.
We add this smallest element to the final sorted array.
We increment the pointer for the subarray from which we took the smallest element.
We repeat this process until all elements from the subarrays are merged into the final sorted array.
Step 5: Return the Sorted Array

Finally, we return the sorted array, which now contains all elements sorted in ascending order.

Time Complexity Comparison

To compare the time complexities of k-split insertion sort and k-split merge sort with typical insertion sort and merge sort, we need to consider how each of these algorithms behaves in terms of time complexity.

Typical Insertion Sort:

- The time complexity of typical insertion sort is O(n^2), where `n` is the size of the input array. It has a quadratic time complexity because it involves nested loops to compare and swap elements.

Typical Merge Sort:

- The time complexity of typical merge sort is O(n log n), where `n` is the size of the input array. Merge sort uses a divide-and-conquer approach, splitting the array into halves recursively and merging them back together. This results in a time complexity that grows logarithmically with the input size.

Now, let's compare these typical complexities with k-split insertion sort and k-split merge sort:

K-Split Insertion Sort vs. Typical Insertion Sort:

- K-Split Insertion Sort: O(n + (n^2)/k)
- Typical Insertion Sort: O(n^2)

When comparing k-split insertion sort with typical insertion sort, k-split insertion sort can be more efficient for large values of `k` because it divides the sorting task into smaller subarrays and then sorts them individually using insertion sort. However, for small values of `k`, the typical insertion sort may be more efficient.

K-Split Merge Sort vs. Typical Merge Sort:

- K-Split Merge Sort: O(n * log(n/k))
- Typical Merge Sort: O(n log n)

When comparing k-split merge sort with typical merge sort, typical merge sort has a better time complexity. The k-split merge sort introduces an additional factor of `k` in its time complexity, making it less efficient for most practical cases. Typical merge sort is generally more efficient, especially when the input size is sufficiently large.

In summary:

- K-Split Insertion Sort can be more efficient for large values of `k` but less efficient for small values of `k` compared to typical insertion sort.
- Typical Merge Sort is more efficient than K-Split Merge Sort for most practical cases, as it has a better time complexity, especially for large input sizes. K-Split Merge Sort may be useful when you specifically need to divide the sorting task into `k` subarrays for some reason.

Code:

```
def split_array(l,k):
    n = len(l)
    size = n//k
    rem = n%k
    start = 0
    sub = []
    for i in range (k):
        end = start + size
        sub.append(l[start:end])
        start = end

    if rem > 0:
        sub[-1].extend(l[end:])
    return sub
```

```
def insertion_sort(arr):
    for i in range(1, len(arr)):
        key = arr[i]
        j = i - 1
        while j >= 0 and key < arr[j]:
            arr[j + 1] = arr[j]
            j -= 1
        arr[j + 1] = key
```

```
def merge_sorted_arrays(arrays,num_arrays):
    # num_arrays = len(arrays)
    total_elements = sum(len(arr) for arr in arrays)
    pointers = [0] * num_arrays
    result_array = [0] * total_elements

    for t in range(1, total_elements + 1):
        max_element = float("-inf")
        max_element_array_index = -1

        for i in range(num_arrays):
            current_array_length = len(arrays[i])
            if (
                pointers[i] < current_array_length
                and max_element < arrays[i][current_array_length - pointers[i] - 1]
            ):
                max_element = arrays[i][current_array_length - pointers[i] - 1]
                max_element_array_index = i

        result_array[total_elements - t] = max_element
        pointers[max_element_array_index] += 1

    return result_array
```

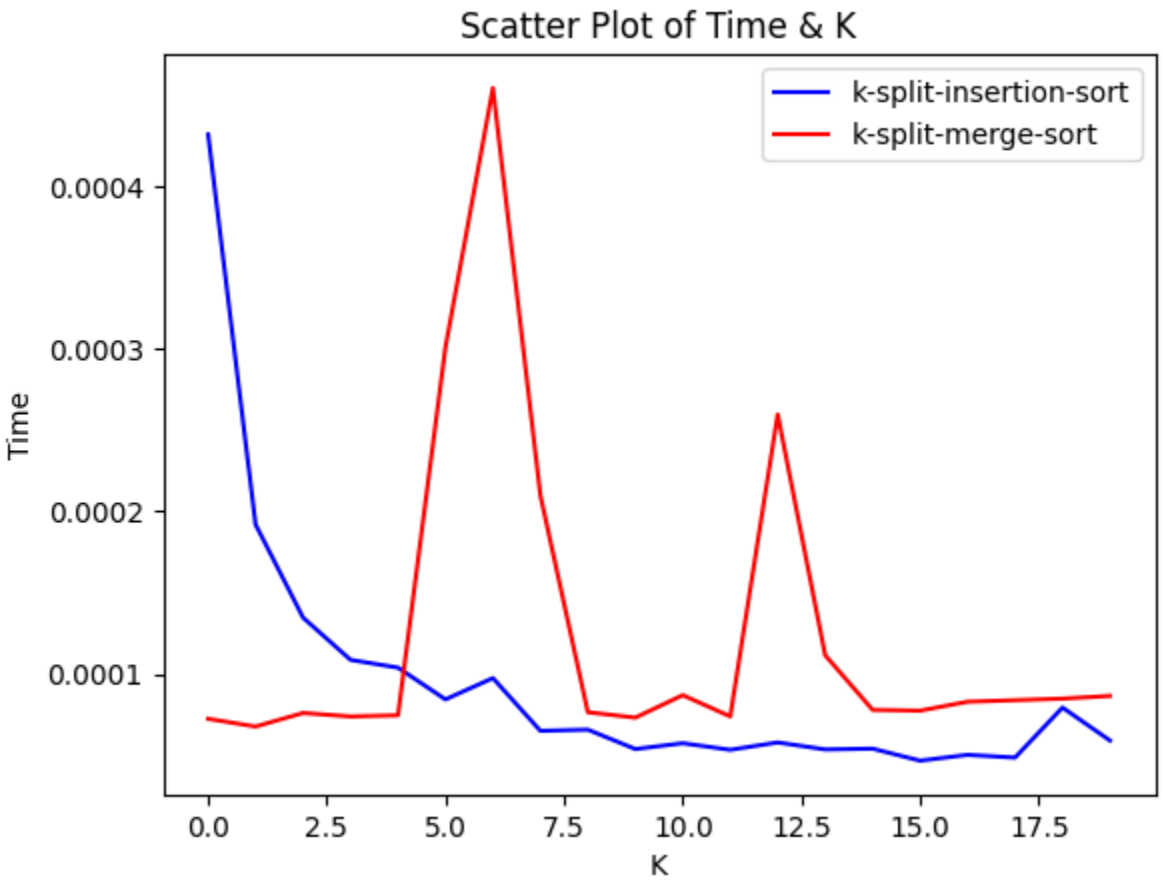
```
def merge_sort(arr):
    if len(arr) > 1:
        mid = len(arr) // 2
        left_half = arr[:mid]
        right_half = arr[mid:]
        merge_sort(right_half)
        i,j,k=0,0,0
        while i < len(left_half) and j < len(right_half):
            if left_half[i] < right_half[j]:
                arr[k] = left_half[i]
                i += 1
            else:
                arr[k] = right_half[j]
                j += 1
```

```
            k += 1

        while i < len(left_half):
            arr[k] = left_half[i]
            i += 1
            k += 1

        while j < len(right_half):
            arr[k] = right_half[j]
            j += 1
            k += 1
```

```
import random
import time
import matplotlib.pyplot as plt

random.seed=42
l=[random.randint(1,101) for i in range(50)]
ll=[]+l

time_taken=[]

for i in range(1,21):
    start_time=time.time()
    k=split_array(l,i)
    for j in k:
        insertion_sort(j)
    end_time=time.time()
    time_taken.append((end_time-start_time))

time_taken_merge=[]
for i in range(1,21):
    start_time_merge=time.time()
    k=split_array(l,i)
    for j in k:
        merge_sort(j)
    end_time_merge=time.time()
    time_taken_merge.append((end_time_merge-start_time_merge))


data1 = time_taken
data2 = time_taken_merge
indices = list(range(len(time_taken)),)

plt.plot(indices, data1, label='k- split-insertion-sort', color='blue')
plt.plot(indices, data2, label='k split-merge-sort', color='red')
plt.xlabel('K')
plt.ylabel('Time')
plt.title('Scatter Plot of Time & K')
plt.legend()
```

```
plt.show()
```

Output:



Observation:

For an unsorted list of size `100` running the algorithm multiple time we have come to the conclusion that the optimum value for `k` is in the range `10 to 15`.

Question 3

Question 3A

Algorithm:

1. **Initialization**:
   - The class initializes an empty `graph` dictionary to store the vertices and their adjacent edges and a counter `edge_count` to keep track of the number of edges.
2. **Adding Vertices**:
   - The `add_vertex` method allows you to add vertices to the graph. It checks if the vertex already exists in the graph and adds it if not.
3. **Adding Weighted Edges**:
   - The `add_edge` method adds weighted edges between two vertices. It adds the edge in both directions (undirected graph) and increments the `edge_count`.
4. **Getting Edges in Decreasing Weight Order**:
   - The `get_edges_in_decreasing_order` method extracts all edges from the graph along with their weights, sorts them in decreasing order of weight, and returns the sorted list.
5. **Removing Edges to Build MST**:
   - The core of the MST algorithm is implemented in the `if __name__ == "__main__":` block.
   - It first creates an instance of the `WeightedGraph` and adds vertices and edges to it.
   - It retrieves the edges in decreasing order of weight and stores them in the `dec_edge` list.
   - It iterates through the sorted edges:
     - It temporarily removes the edge from the graph using the `remove_edge` method.
     - It checks if removing the edge disconnects the graph by performing a depth-first search (DFS) starting from a specific vertex (in this case, "A"). If the DFS result changes (the graph becomes disconnected), it adds the edge back.
     - This process continues until all edges have been checked.
   - The result is an MST stored in the `graph` attribute of the `WeightedGraph` instance.

Code:

```
class WeightedGraph:
    def __init__(self):
        self.graph = {}
        self.edge_count = 0

    def add_vertex(self, vertex):
        if vertex not in self.graph:
            self.graph[vertex] = []

    def add_edge(self, vertex1, vertex2, weight):
        if vertex1 in self.graph and vertex2 in self.graph:
            self.graph[vertex1].append((vertex2, weight))
            self.graph[vertex2].append((vertex1, weight))
            self.edge_count += 1

    def get_edges_in_decreasing_order(self):
        edges = []
        for vertex, neighbors in self.graph.items():
            for neighbor, weight in neighbors:
                edges.append((vertex, neighbor, weight))
        edges.sort(key=lambda edge: edge[2], reverse=True)
```

```
        return edges

    def remove_edge(self, vertex1, vertex2):
        if vertex1 in self.graph and vertex2 in self.graph:
            self.graph[vertex1] = [(v, w) for v, w in self.graph[vertex1] if v != vertex2]
            self.graph[vertex2] = [(v, w) for v, w in self.graph[vertex2] if v != vertex1]

    def dfs(self, start_vertex):
        visited = set()
        stack = [start_vertex]
        result = []

        while stack:
            vertex = stack.pop()
            if vertex not in visited:
                visited.add(vertex)
                result.append(vertex)
                stack.extend(neighbor for neighbor, _ in self.graph[vertex] if neighbor not in visited)

        return result

    def display_graph(self):
        for vertex, neighbors in self.graph.items():
            print(f"Vertex {vertex}: {neighbors}")

if __name__ == "__main__":
    g = WeightedGraph()

    g.add_vertex("A")
    g.add_vertex("B")
    g.add_vertex("C")
    g.add_vertex("D")

    g.add_edge("A", "B", 10)
    g.add_edge("A", "C", 6)
    g.add_edge("A", "D", 5)
    g.add_edge("C", "D", 4)
    g.add_edge("B", "D", 15)


    dec_edge=g.get_edges_in_decreasing_order()
    print("The original Graph: ",g.graph)
    for _ in range(g.edge_count):
        edge=dec_edge[0]
        (X,Y,Z)=edge
        original=len(g.dfs("A"))
        g.remove_edge(X,Y)
        new=len(g.dfs("A"))
        if original!=new:
            g.add_edge(X,Y,Z)
        dec_edge.pop(0)
        dec_edge.pop(0)

    print("The MST          : ",g.graph)
```

Output:

```
The original Graph:  {'A': [('B', 10), ('C', 6), ('D', 5)], 'B': [('A', 10), ('D', 15)], 'C': [('A', 6), ('D', 4)], 'D': [('A', 5), ('C', 4
The MST          :  {'A': [('B', 10), ('D', 5)], 'B': [('A', 10)], 'C': [('D', 4)], 'D': [('A', 5), ('C', 4)]}
```

Question 3B

Algorithm:

1. **Initialization**:
   - Create an empty dictionary called `graph` within the `Graph` class to represent the graph.
2. **Adding Edges**:
   - Implement the `add_edge` method to add directed edges to the graph. If the source vertex `u` already exists in the graph, append the destination vertex `v` to its adjacency list. If not, create a new key-value pair with `u` as the key and `[v]` as the value in the `graph` dictionary.

3. **Depth-First Search (DFS)**:
   - Implement a depth-first search (`dfs`) method that takes a starting vertex `v`, a set of visited vertices, and a stack as parameters.
   - Start with the initial vertex `v`, mark it as visited, and explore its adjacent vertices recursively.
   - Push the visited vertices onto the stack in the order they are visited.
4. **Transposing the Graph**:
   - Implement the `transpose` method to create a new graph (`g`) that is the transpose of the original graph. In the transpose graph, all edges are reversed.
   - Iterate through each vertex `u` in the original graph's `graph` dictionary. For each edge from `u` to `v`, add an edge from `v` to `u` in the transpose graph.
5. **Kosaraju's Algorithm**:
   - Implement the `kosaraju` method to find the strongly connected components (SCCs) of the graph.
   - Create an empty stack and a set to keep track of visited vertices.
   - Perform a DFS traversal of the original graph, pushing vertices onto the stack after visiting all their descendants.
   - Transpose the graph to obtain the transpose graph.
   - Initialize another empty set to track visited vertices and an empty list to store SCCs.
   - While the stack is not empty:
     - Pop a vertex `u` from the stack.
     - If `u` is not visited:
       - Initialize an empty list `scc`.
       - Perform a DFS traversal on the transpose graph starting from `u`, adding visited vertices to the `scc` list.
       - Append the `scc` list to the list of SCCs.
6. **Printing Edges and Transpose Graph**:
   - Implement `print_edges` and `print_transpose` methods to print the edges of the original graph and the edges of the transpose graph, respectively.
   -

Code:

```
class Graph:
    def __init__(self):
        self.graph = {}

    def add_edge(self, u, v):
        if u in self.graph:
            self.graph[u].append(v)
        else:
            self.graph[u] = [v]

    def dfs(self, v, visited, stack):
        visited.add(v)
        for i in self.graph.get(v, []):
            if i not in visited:
                self.dfs(i, visited, stack)
        stack.append(v)

    def transpose(self):
        g = Graph()
        for u in self.graph:
            for v in self.graph[u]:
                g.add_edge(v, u)
        return g
```

```
    def kosaraju(self):
        stack = []
        visited = set()

        for u in self.graph:
            if u not in visited:
                self.dfs(u, visited, stack)

        gt = self.transpose()

        visited = set()
        sccs = []

        while stack:
            u = stack.pop()
            if u not in visited:
                scc = []
                gt.dfs(u, visited, scc)
                sccs.append(scc)

        return sccs

    def print_transpose(self):
        for u in self.graph:
            for v in self.graph[u]:
                print(f"{v} -> {u}")

    def print_edges(self):
        for u in self.graph:
            for v in self.graph[u]:
                print(f"{u} -> {v}")

g = Graph()
g.add_edge(1, 2)
g.add_edge(2, 3)
g.add_edge(1, 4)
g.add_edge(4, 5)
g.add_edge(5, 6)
g.add_edge(6, 4)
g.add_edge(4, 7)
g.add_edge(7, 8)
g.add_edge(8, 9)
g.add_edge(9, 10)
g.add_edge(10, 7)

print("\nEdges:")
g.print_edges()


sccs = g.kosaraju()
print()
print("Strongly Connected Components:")
for i, scc in enumerate(sccs):
    print(f"SCC {i + 1}: {scc}")



print("\nTranspose Graph:")
g.print_transpose()
```

Output:

```
Edges:
1 -> 2
1 -> 4
2 -> 3
4 -> 5
4 -> 7
5 -> 6
6 -> 4
7 -> 8
8 -> 9
9 -> 10
10 -> 7

Strongly Connected Components:
SCC 1: [1]
```

```
SCC 2: [5, 6, 4]
SCC 3: [8, 9, 10, 7]
SCC 4: [2]
SCC 5: [3]

Transpose Graph:
2 -> 1
4 -> 1
3 -> 2
5 -> 4
7 -> 4
6 -> 5
4 -> 6
8 -> 7
9 -> 8
10 -> 9
7 -> 10
```

Question 3C

```
#include <bits/stdc++.h>
using namespace std;

const int MAX_NODES = 100001;

int u[MAX_NODES];         // Array to store u values
int v[MAX_NODES];         // Array to store v values
int val[MAX_NODES];       // Array to store values
int n, m, k;              // Number of nodes, number of edges, and threshold
vector<int> adj[MAX_NODES];  // Adjacency list for the graph
stack<int> dfsStack;      // Stack for DFS traversal
bool visited[MAX_NODES];  // Array to keep track of visited nodes
int componentSize[MAX_NODES]; // Array to store component sizes
int componentCount = 0;   // Counter for connected components

// Depth-First Search (DFS) for traversal
void dfs(int node) {
    visited[node] = true;
    for (int neighbor : adj[node]) {
        if (!visited[neighbor]) {
            dfs(neighbor);
        }
    }
    dfsStack.push(node);
}

// Second DFS for counting component sizes
void dfs2(int node) {
    componentSize[componentCount]++;
    visited[node] = true;
    for (int neighbor : adj[node]) {
        if (!visited[neighbor]) {
            dfs2(neighbor);
        }
    }
}

// Function to check if there's a component with size at least k
bool check(int limit) {
    componentCount = 0;
    memset(visited, false, sizeof(visited));
    for (int i = 1; i <= n; i++) {
        adj[i].clear(); // Clear the adjacency list
    }
    for (int i = 1; i <= m; i++) {
        if (val[u[i]] >= limit && val[v[i]] >= limit) {
            adj[u[i]].push_back(v[i]); // Add edges to the graph
        }
    }
    for (int i = 1; i <= n; i++) {
        if (!visited[i]) {
            dfs(i);
        }
    }
    memset(visited, false, sizeof(visited));
    while (!dfsStack.empty()) {
        int cur = dfsStack.top();
```

```
        dfsStack.pop();
        if (!visited[cur]) {
            dfs2(cur);
            componentCount++;
        }
    }
    int largestComponentSize = *max_element(componentSize + 1, componentSize + componentCount + 1);
    fill(componentSize + 1, componentSize + componentCount + 1, 0); // Reset component sizes
    return largestComponentSize >= k;
}

int main() {
    cin >> n >> m >> k; // Input the number of nodes, edges, and threshold
    for (int i = 1; i <= n; i++) cin >> val[i]; // Input values for each node
    for (int i = 1; i <= m; i++) cin >> u[i] >> v[i]; // Input edges

    int left = 1, right = 1e9; // Binary search range
    int answer = -1; // Initialize the answer
    while (left <= right) {
        int mid = (left + right) / 2;
        if (check(mid)) {
            answer = mid;
            left = mid + 1;
        } else {
            right = mid - 1;
        }
    }
    cout << answer << endl; // Output the answer
    return 0;
}
```