1. Modify the partition() function in quicksort to implement the following. Take a list as input. This list contains the firstname, lastname and date of birth of n persons. Priority rule for a person:

(i) First letter of firstname

(ii) First letter of lastname

(iii) Date of birth.

In case of clash we look for second rule and if there is a clash then look for the third rule. Print the position of the chosen person in the list according to his/her priority. Show the best case, worst case and average case time taken for 5 element list.

Example: Input Anil Kumar 04/04/2000, Amit Kumar 04/04/2001, Asraf Seikh 08/12/2008
Position of Asraf Seikh is 3

ANSWER

```
import pandas as pd
def custom_compare(person1, person2):
    first_name1, last_name1, dob1 = person1
    first_name2, last_name2, dob2 = person2

    if first_name1[0] < first_name2[0]:
        return -1
    elif first_name1[0] > first_name2[0]:
        return 1
    else:
        if last_name1[0] < last_name2[0]:
            return -1
        elif last_name1[0] > last_name2[0]:
            return 1
        else:
            dob1_date = pd.to_datetime(dob1)
            dob2_date = pd.to_datetime(dob2)
            if dob1_date.day < dob2_date.day:
                return -1
            elif dob1_date.day > dob2_date.day:
                return 1
            else:
                return 0

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

    for j in range(low, high):
        if custom_compare(arr[j], pivot) < 0:
            i += 1
            arr[i], arr[j] = arr[j], arr[i]

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

def quicksort(arr, low, high):
    if low < high:
        pi = partition(arr, low, high)
        quicksort(arr, low, pi - 1)
        quicksort(arr, pi + 1, high)


people = [
    ("Anil", "Kumar", "04/04/2000"),
    ("Amit", "Kumar", "04/04/2001"),
```

```
    ("Asraf", "Seikh", "08/12/2008")
]


quicksort(people, 0, len(people) - 1)


chosen_person = ("Asraf", "Seikh", "08/12/2008")
position = people.index(chosen_person) + 1

print(f"Position of {chosen_person[0]} {chosen_person[1]} is {position}")
```

```
Position of Asraf Seikh is 3
```

2. Modify Bellman Ford algorithm to find longest path from the given source to the destination via a given node. Analyze the time taken for the code written by you.

ANSWER

```
class Graph:

    def __init__(self, vertices):
        self.V = vertices  # No. of vertices
        self.graph = []

    def addEdge(self, u, v, w):
        self.graph.append([u, v, w])

    def printArr(self, dist):
        print("Vertex Distance from Source")
        for i in range(self.V):
            print("{0}\t\t{1}".format(i, dist[i]))

    def BellmanFord(self, src):
        dist = [float("-Inf")] * self.V  # Change initialization to negative infinity
        dist[src] = 0
        for _ in range(self.V - 1):
            for u, v, w in self.graph:
                if dist[u] != float("-Inf") and dist[u] + w > dist[v]:  # Change the condition
                    dist[v] = dist[u] + w
        for u, v, w in self.graph:
            if dist[u] != float("-Inf") and dist[u] + w > dist[v]:  # Change the condition
                print("Graph contains positive weight cycle")  # Change the message
                return

        self.printArr(dist)

if __name__ == '__main__':
  g = Graph(5)
  g.addEdge(0, 1, 1)
  g.addEdge(0, 2, 4)
  g.addEdge(1, 2, 3)
  g.addEdge(1, 3, 2)
  g.addEdge(1, 4, 2)
  g.addEdge(2, 3, 5)
  g.addEdge(4, 3, 3)

g.BellmanFord(0)
```

```
Vertex Distance from Source
0		0
```

1	1
2	4
3	9
4	3

3. You need to create a tertiary tree using linked list. Each node in this tree stores an alphabet (upper case only). You need to check that this tree is a max-heap or not. Show the best-case, worst-case and average case time taken for 10 element list.

The best-case time complexity of the algorithm is O(n). This happens when the tree is already a max-heap.

The worst-case time complexity of the algorithm is O(n log n). This happens when the tree is a complete binary tree, i.e., all levels of the tree are fully filled except possibly the last level, which is filled from left to right.

The average case time complexity of the algorithm is also O(n log n).

ANSWER

```
class Node:
    def __init__(self, key):
        self.key = key
        self.left = None
        self.right = None


def max_heapify(root, i, n):
    largest = i
    left = 2 * i + 1
    right = 2 * i + 2

    if left < n and root[left].key > root[largest].key:
        largest = left

    if right < n and root[right].key > root[largest].key:
        largest = right

    if largest != i:
        root[i], root[largest] = root[largest], root[i]
        max_heapify(root, largest, n)


def build_max_heap(root, n):
    for i in range(n//2-1, -1, -1):
        max_heapify(root, i, n)


def is_max_heap(root, n):
    for i in range(n//2-1, -1, -1):
        if 2 * i + 1 < n and root[i].key < root[2 * i + 1].key:
            return False
        if 2 * i + 2 < n and root[i].key < root[2 * i + 2].key:
            return False
    return True


def check_max_heap(root, n):
    return is_max_heap(root, n)


root = [None] * 10
keys = ['H', 'G', 'F', 'E', 'D', 'C', 'B', 'A']
for i in range(len(keys)):
    root[i] = Node(keys[i])

n = len(keys)
```

```
build_max_heap(root, n)

if check_max_heap(root, n):
    print("This is a max heap.")
else:
    print("This is not a max heap.")
```

```
This is a max heap.
```

4. Consider a list of 10 numbers. Answer the following questions with reference to the same. Compare linear search and binary search in terms of best case, average case and worst case time taken

---

ANSWER

```
count = 0
def linearSearch(array, n, x):
    global count
    for i in range(0, n):
        count += 1
        if array[i] == x:
            return i
    return -1

array = [2, 4, 0, 1, 9,6,7,3,11,17]
x = 2
n = len(array)
result = linearSearch(array, n, x)
if result == -1:
    count += 1
    print("Element not found")
else:
    count += 1
    print("Element found at index:", result)

print("Number of comparisons in linear search:", count)
```

```
Element found at index: 0
Number of comparisons in linear search: 2
```

Linear Search: Worst Case: When looking for an item at the end of an unsorted list, linear search checks every item before finding the desired one. Best Case: Even in the best case (when the item is at the start), linearsearch may still need to check the entire list. Average Case Time: On average, linear search takes longer as it needs to check each item, resulting in a time proportional to the list's length (linear time).

```
count = 0
def binarySearch(arr, l, r, x):
    global count
    while l <= r:
        count += 1
        mid = l + (r - l) // 2
        if arr[mid] == x:
            count += 1
            return mid
        elif arr[mid] < x:
            count += 1
            l = mid + 1
        else:
            count += 1
            r = mid - 1
    return -1

arr = [0,1,2,3,4,6,7,9,11,17]
x = 2
```

```
n = len(arr)
result = binarySearch(arr, 0, n-1, x)
if result == -1:
    count += 1
    print("Element not found")
else:
    count += 1
    print("Element found at index:", result)

print("Number of comparisons in binary search:", count)
```

```
Element found at index: 2
Number of comparisons in binary search: 7
```

Binary Search: Worst Case: If the list is in reverse order, binary search takes a long time as it repeatedly checks the entire list in each step before finding the item. Best Case: Both binary search and merge sort work well when the item is at the beginning of a sorted list, taking very little time (constant time). Average Case Time: Binary search is faster on average because it splits the search space in half with each step, resulting in a quicker search (logarithmic time).

5. Implement merge sort and quick sort. Analyse best case, worst case and average case.

ANSWER

```
count = 0

def mergeSort(arr):
    global count
    if len(arr) > 1:
        count += 1
        mid = len(arr)//2
        L = arr[:mid]
        R = arr[mid:]
        mergeSort(L)
        mergeSort(R)

        i = j = k = 0
        while i < len(L) and j < len(R):
            count += 1
            if L[i] <= R[j]:
                count += 1
                arr[k] = L[i]
                i += 1
            else:
                count += 1
                arr[k] = R[j]
                j += 1
            k += 1

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

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

```
def printList(arr):
    global count
    for i in range(len(arr)):
        count += 1
        print(arr[i], end=" ")
    print()

arr = [12, 11, 13, 5, 6, 7]
print("Given array is")
printList(arr)
mergeSort(arr)
print("\nSorted array is ")
printList(arr)

print("Number of comparisons:", count)
```

```
Given array is
12 11 13 5 6 7

Sorted array is
5 6 7 11 12 13
Number of comparisons: 41
```

```
count = 0

def quickSort(arr, low, high):
    global count
    if low < high:
        count += 1
        pi = partition(arr, low, high)
        quickSort(arr, low, pi - 1)
        quickSort(arr, pi + 1, high)

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

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

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

def printList(arr):
    for i in range(len(arr)):
        print(arr[i], end=" ")
    print()

arr = [12, 11, 13, 5, 6, 7]
n = len(arr)

print("Given array is")
printList(arr)

quickSort(arr, 0, n - 1)
```

```
print("\nSorted array is ")
printList(arr)

print("Number of comparisons:", count)
```

```
Given array is
12 11 13 5 6 7

Sorted array is
5 6 7 11 12 13
Number of comparisons: 18
```

6. You have an array containing known set of numbers {1,2,3}. Example array: 1,2,1,3,1,1,2,2,3,1,2...

   Write an efficient code to sort these numbers in O(N) time.

ANSWER

```
def counting_sort(arr):
  count = [0,0,0]
  for i in arr:
    count [i-1] += 1
  sorted_arr = []
  for i in range(3):
    sorted_arr.extend([i+1]*count[i])
  return sorted_arr

print(counting_sort([1,2,1,3,1,1,2,2,3,1,2]))
```

```
[1, 1, 1, 1, 1, 2, 2, 2, 2, 3, 3]
```

7. You have an array containing only zeros and ones. Data is available in chunks. Sort the array in O(N) time. Given array:
   00000000000000111111111111111111100000000000000011111111111000000001111111111

ANSWER

```
def sorting_bin(arr):
  count_0=0
  count_1=1
  for i in arr:
    if i == 1:
      count_1 += 1
    else:
      count_0 += 1
  sorted_arr = ( [0]*count_0 + [1]*count_1 )
  return sorted_arr

print(sorting_bin ([0,0,0,0,0,1,1,1,1,1,1,1,1,1,1,1,0,0,0,0,0,0,0,0,1,1,1,1,1,1,1,1,1,1]))
```

```
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
```

8. You have a unsorted list of ten numbers. Which searching algorithm you are going to apply. You have two options

(i) Linear search

(ii) Sorting followed by binary search.

(You can use either bubble , selection or insertion sort). If you choose binary search which sorting algorithm you choose and why? Write codes with count variable for each case.

ANSWER

I will choose option (ii) Sorting followed by binary search. I will use counting sort for sorting. The reason behind choosing counting sort is that it has a linear time complexity of O(n), making it the most efficient sorting algorithm for this particular case.

If the given unsorted list of ten numbers contains a specific value, it would be best to use the linear search algorithm. Linear search algorithm is a simple searching algorithm that scans through every element in the list sequentially until the desired element is found, or the end of the list is reached. The time complexity of linear search is O(n), where n is the number of elements in the list.

Here is a code implementation of linear search:

```
def linear_search(arr, target):
    count = 0
    for i in range(len(arr)):
        count += 1
        if arr[i] == target:
            return i, count
    return -1, count

numbers = [21, 46, 33, 81, 72, 11, 28, 79, 35, 18]
target = 72

result = linear_search(numbers, target)
if result[0] != -1:
    print(f"The target {target} is found at index {result[0]} in {result[1]} iterations.")
else:
    print(f"The target {target} is not found in the list in {result[1]} iterations.")
```

```
The target 72 is found at index 4 in 5 iterations.
```

On the other hand, if the list was already sorted, it would be more efficient to use binary search instead of linear search.

Binary search algorithm is a more advanced searching algorithm that halves the search space at each step by comparing the middle element of the search space with the target value. If the middle element is equal to the target value, the search is successful. If the middle element is less than the target value, the search continues in the right half of the search space. If the middle element is greater than the target value, the search continues in the left half of the search space. The time complexity of binary search is O(log n).

If we were to use binary search, we could use any sorting algorithm such as bubble sort, selection sort, or insertion sort to sort the list before applying the binary search algorithm. Here is a code implementation of binary search using insertion sort as the sorting algorithm:

```
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 binary_search(arr, target):
    count = 0
    low = 0
    high = len(arr) - 1
    while low <= high:
        count += 1
        mid = (high + low) // 2
        if arr[mid] == target:
            return mid, count
        elif arr[mid] < target:
            low = mid + 1
        else:
            high = mid - 1
    return -1, count

numbers = [21, 46, 33, 81, 72, 11, 28, 79, 35, 18]
target = 72
```

```
target = 72

insertion_sort(numbers)

result = binary_search(numbers, target)
if result[0] != -1:
    print(f"The target {target} is found at index {result[0]} in {result[1]} iterations.")
else:
    print(f"The target {target} is not found in the list in {result[1]} iterations.")
```

```
The target 72 is found at index 7 in 2 iterations.
```