Assignment 3 - Write a program to perform heapsort on an n-length array. You should first construct the heap from the array and then perform heapsort and print the correct sorted sequence of the input array. Also analyse the time complexity of the algorithm by plotting the graph of running time of the algorithm for different values of n and comparing it with nlogn. Give your comments.

```
import time
import numpy as np
import matplotlib.pyplot as plt
import math


def heapify(arr, n, i):
    l = 2 * i + 1  # index value of left child.
    r = 2 * i + 2  # index value of right child.
    # To check if left child of root exists and is greater than root.
    if l < n and arr[l] > arr[i]:
        largest = l
    else:
        largest = i
    # To check if right child of root exists and is greater than root.
    if r < n and arr[r] > arr[largest]:
        largest = r
    if largest != i:    # Changing parent with child when child is greater than parent.
        arr[i], arr[largest] = arr[largest], arr[i]
        heapify(arr, n, largest)    # Using recursion to make array again heap.


# Function to build heap of an array.
def BuildHeap(arr):
    n = len(arr)  # Array length.
    for i in range(n // 2 - 1, -1, -1):
        heapify(arr, n, i)  # Calling heapify to make array a heap.


# Function to perform heap sort.
def heapSort(arr):
    n = len(arr)
    BuildHeap(arr)  # Calling function to build heap of given array.
    # One by one extract elements
    for i in range(n - 1, 0, -1):
        arr[i], arr[0] = arr[0], arr[i]  # swap
        heapify(arr, i, 0)


# Driver code
num = []
tm = []
```

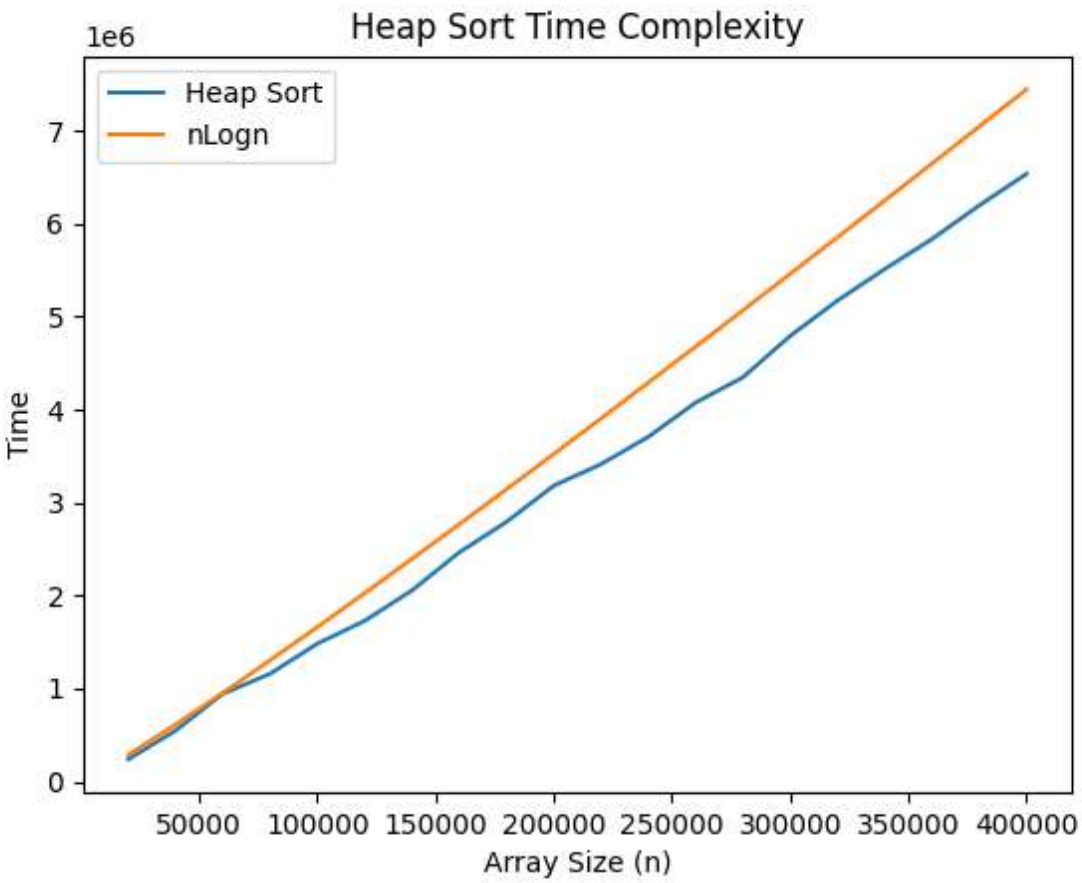
```
y = []
for i in range(1, 21):
    size = i * 20000
    arr = np.random.randint(1, 2000, size)
    n = len(arr)
    num.append(n)
    print("\n\nGiven array is: ")
    for i in range(n):
        print("%d" % arr[i], end=" ")
    start = time.time()
    heapSort(arr)
    end = time.time()
    print("\n\nSorted array is: ")
    for j in range(n):
        print("%d" % arr[j], end=" ")
    tm.append((end - start))
    y.append((n * math.log2(n)))
    print("\n\n--- ", end - start, " nano seconds ---")
plt.title("Heap Sort Time Complexity")
plt.xlabel("Array Size (n)")
plt.ylabel("Time")
plt.plot(num, tm, label="Heap Sort")
plt.plot(num, y, label="nLogn")
plt.legend()
plt.show()
```

Output: Here the graph below shows the time taken by the Heap Sort algorithm for various length of input arrays and the curve follows closely the same path as nlogn when compared.

Time complexity of heapify is O(logn).

Time complexity of BuildHeap() is O(n).

Time Complexity of Heap Sort is O(nlogn).



Assignment 4 - Write a program to implement a priority queue (using Max_Heap). The program should contain the following functions:

a. Maximum(S)

b. Extract_Max(S)

c. Increase_key(S, i, key)

d. Insert (A, key)

Show that each of the above methods takes logn time for running, where n is the problem size.

```
import math
import random
from time import perf_counter_ns
import numpy as np
import matplotlib.pyplot as plt


def Heapify(arr, i):
    l = i * 2 + 1   # Left child of parent at index i.
    r = i * 2 + 2   # Right child of parent at index i.

    # Checking if left child is greater than parent and also index of left child less than array length.
    if l < len(arr) and arr[l] > arr[i]:
        largest = l
    else:
        largest = i
    # Checking if right child is greater than parent and also the index of right child less than array
length.
    if r < len(arr) and arr[r] > arr[largest]:
        largest = r
    # If parent is not largest then exchange the value with largest child.
    if largest != i:
        arr[i], arr[largest] = arr[largest], arr[i]
        Heapify(arr, largest)   # Calling heapify to make heap again.


def Max_Heap(arr):
    heapsize = len(arr) # Array length.
    for i in range(heapsize // 2 - 1, -1, -1):  # Running for loop from last parent to the root node.
        Heapify(arr, i)


def Maximum(arr):
    return arr[0]   # Return root element which is maximum in heap.
```

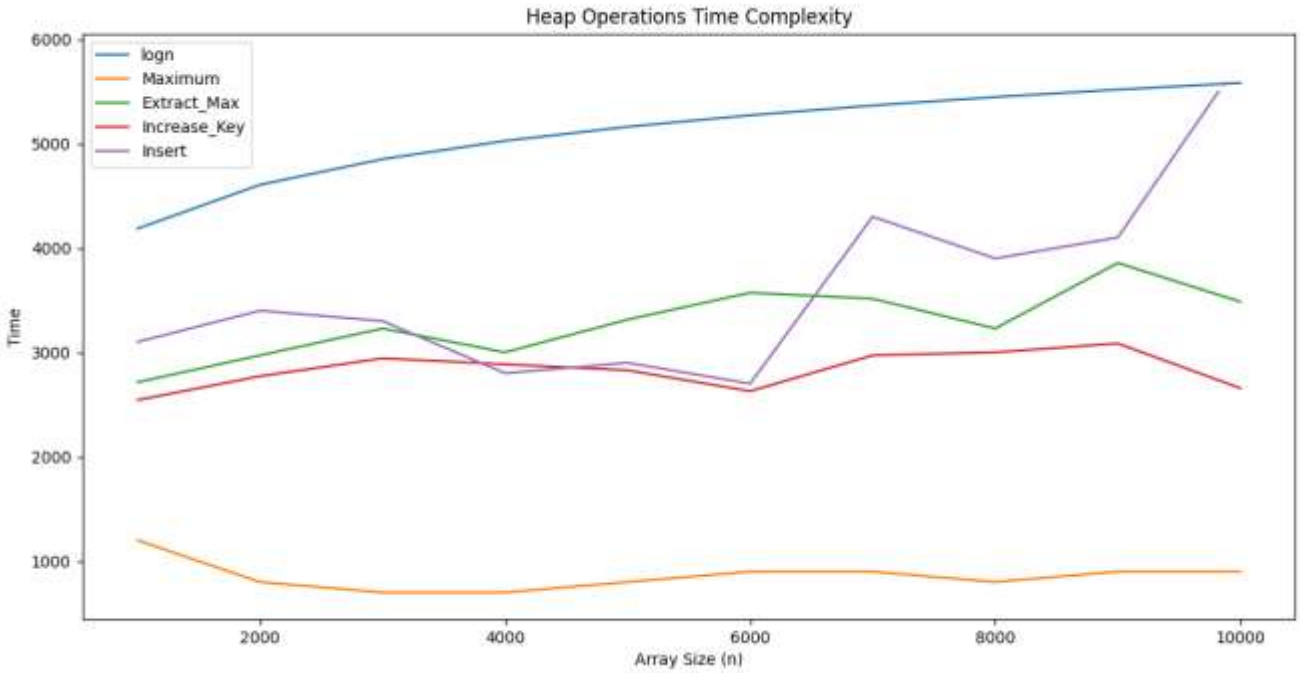
```
def Extract_Max(arr):
    heapsize = len(arr) # Array length.
    if heapsize < 0:
        print("Error: Heap Underflow")
    max = arr[0]    # Extracting root element.
    arr[0] = arr[heapsize - 1]  # Moving last leaf node element to root.
    arr.pop()   # Removing last leaf node from heap.
    Heapify(arr, 0) # Calling heapify to make heap again.
    return max  # Returning maximum element.


def Increase_Key(arr, i, key):
    # Checking if key is smaller than the element at index before inceasing.
    if key < arr[i]:
        print("Error: New key is smaller than current key.")
    arr[i] = key    # Changing element if key is greater.
    # Now comaparing and changing with its children to make it heap again.
    while i > 0 and arr[(i - 1) // 2] < arr[i]:
        arr[i], arr[(i - 1) // 2] = arr[(i - 1) // 2], arr[i]
        i = (i - 1) // 2


def Insert(arr, key):
    arr.append(-1)  # Adding any negative value in array to increase size of array.
    Increase_Key(arr, len(arr) - 1, key)    # Calling increase key to insert the key.


# Arrays for storing values.
t_maximum = []
t_extractmax = []
t_increasekey = []
t_insert = []
x = []
logn = []
for loop in range(1, 11):
    size = loop * 12000  # Length of input array.
    x.append(size)
    logn.append((math.log2(size))*500)   # Logn

    # Creating arraay to perform heap operations.
    arr = [random.randint(100, 900) for i in range(size)]

    # Calling Max Heap function to make a max heap of given array.
    Max_Heap(arr)

    # Maximum function to return maximum element of heap.
```

```
    start = perf_counter_ns()
    Maximum(arr)
    end = perf_counter_ns()
    total = end - start
    t_maximum.append(total)

    # Extract_Max function to extract the maximum element from the heap.
    start = perf_counter_ns()
    Extract_Max(arr)
    end = perf_counter_ns()
    total = end - start
    t_extractmax.append(total)

    # Increase_Key function to increase key value at given index.
    start = perf_counter_ns()
    Increase_Key(arr, random.randint(100, 900), random.randint(900, 1500))  # Increase_Key(Array,
Index, Key)
    end = perf_counter_ns()
    total = end - start
    t_increasekey.append(total)

    # Insert Function to insert a element in heap.
    start = perf_counter_ns()
    Insert(arr, random.randint(100, 900))   # Insert(Array, Key)
    end = perf_counter_ns()
    total = end - start
    t_insert.append(total)

# Plotting values on graph.
plt.title("Heap Operations Time Complexity")
plt.xlabel("Array Size (n)")
plt.ylabel("Time")
plt.plot(x, logn, label="logn")
plt.plot(x, t_maximum, label="Maximum")
plt.plot(x, t_extractmax, label="Extract_Max")
plt.plot(x, t_increasekey, label="Increase_Key")
plt.plot(x, t_insert, label="Insert")
plt.legend()
plt.show()
```

Output: Here the graph below shows the time taken by the various Heap Sort operations algorithm for various length of input arrays.

Time complexity of heapify is O(logn).

Time complexity of BuildHeap() is O(n).



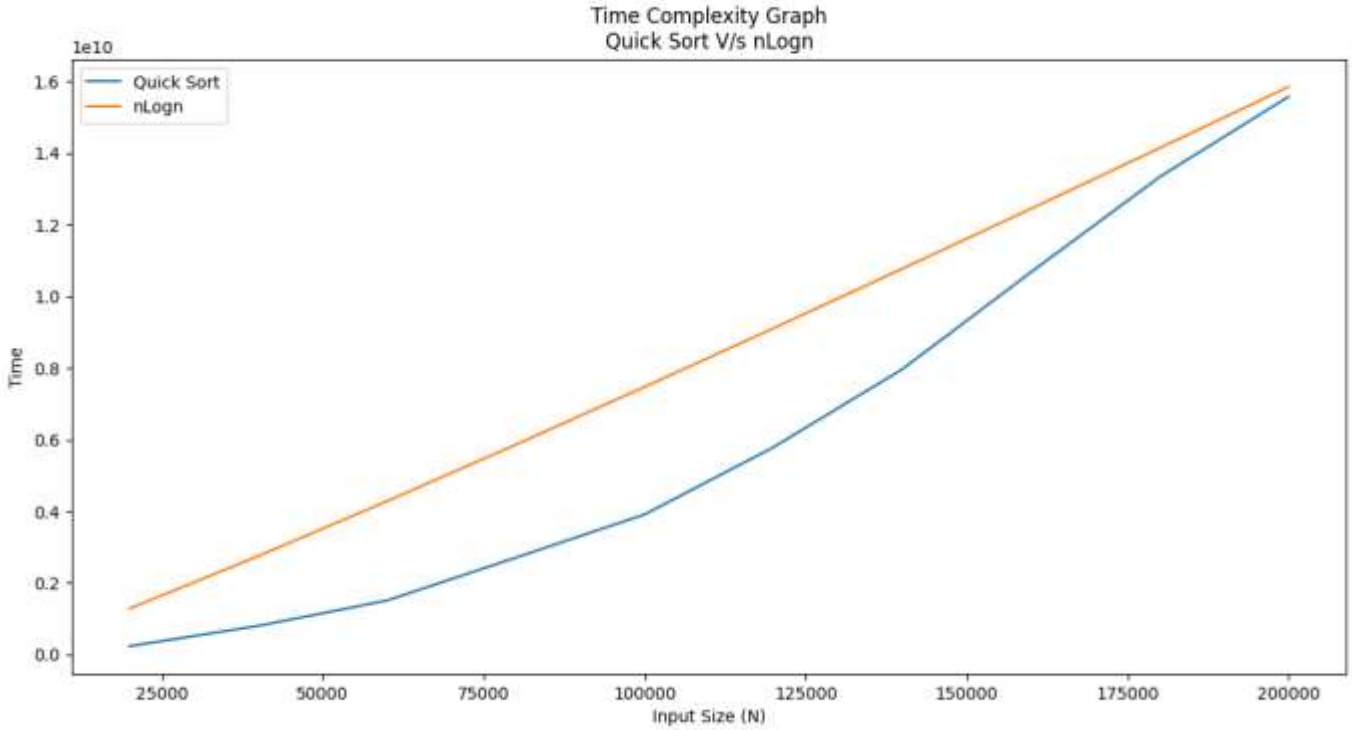
Assignment 5 - Write a program to implement Quick Sort algorithm. Also plot the graph of time complexity for different values of array size 'n'. Compare this with the plot of nlogn.

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


def partition(A, p, r):
    pivot = A[r]
    i = p - 1
    for j in range(p, r):
        if A[j] <= pivot:
            i = i + 1
            A[i], A[j] = A[j], A[i]
    A[i + 1], A[r] = A[r], A[i + 1]
    return i + 1


def quick_sort(A, p, r):
    if p < r:
        q = partition(A, p, r)
        quick_sort(A, p, q - 1)
        quick_sort(A, q + 1, r)

xaxis = []
yaxis = []
nlogn = []
for x in range(1, 11):
    inputsize = x * 20000
    xaxis.append(inputsize)
    A = [random.randint(100, 500) for i in range(inputsize)]
    print(f'\n\nGiven array: {A}')
    start_time = time.time_ns()
    quick_sort(A, 0, len(A) - 1)
    end_time = time.time_ns()
    yaxis.append(end_time - start_time)
    nlogn.append((inputsize * math.log2(inputsize))*4500)
    print(f'\nSorted array: {A}')
    print(f'\nTime Taken: {end_time - start_time}')
plt.plot(xaxis, yaxis, label="Quick Sort")
plt.plot(xaxis, nlogn, label="nLogn")
plt.xlabel('Input Size (N)')
plt.ylabel('Time')
plt.title("Time Complexity Graph\nQuick Sort V/s nLogn")
plt.legend()
plt.show()
```

Output: Here the graph below shows the time taken by the Quick Sort algorithm for various length of input arrays and the curve follows closely the same path as nlogn when compared.

Best case time complexity of Quick Sort is O(logn).

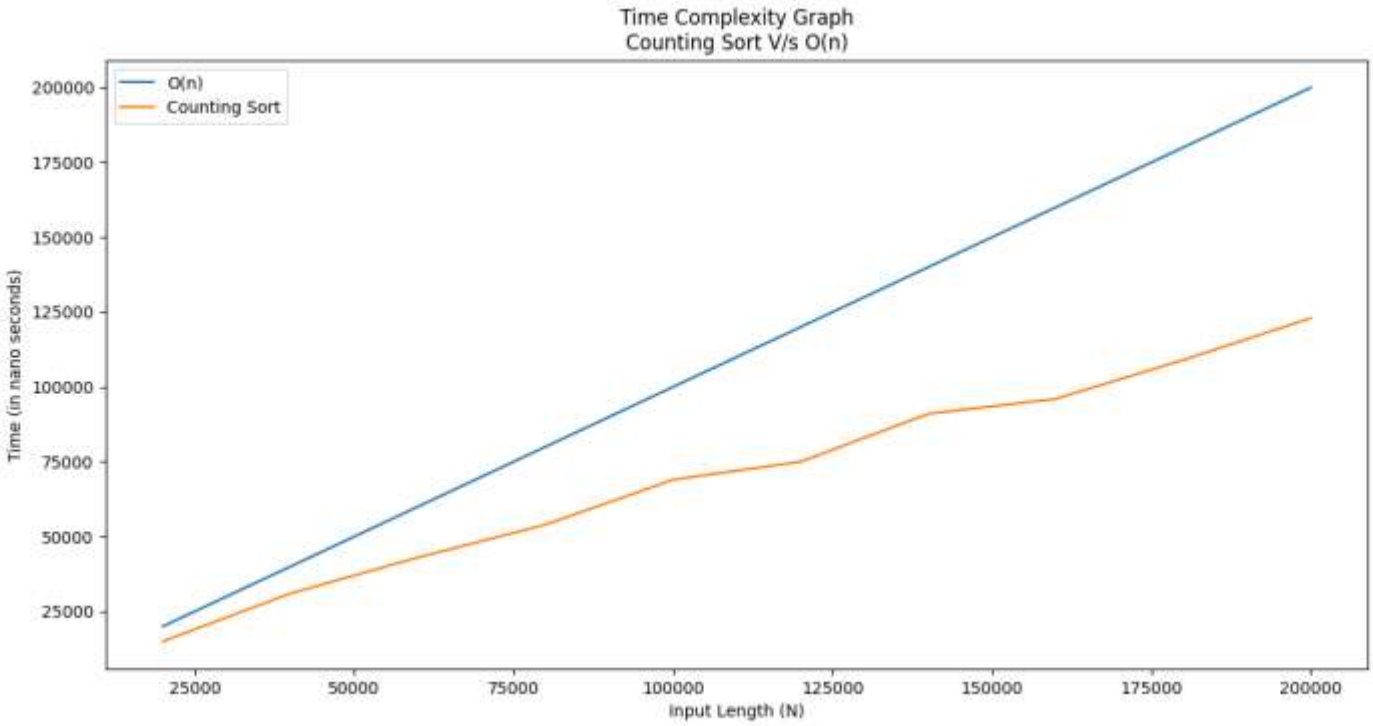
Worst case time complexity of Quick Sort is $O(n^2)$.



Assignment 6 - Write a program to implement the Counting sort algorithm. Verify that it runs in O(n) time for inputs in the range of n.

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


# Function to perform counting sort.
def counting_sort(A, B, k):
    c = [0] * (k + 1)   # Array to store count of each word.
    for j in range(len(A)):
        c[A[j]] += 1    # Adding 1 at index value equal to number in array.
    for i in range(1, k + 1):
        c[i] += c[i - 1]    # Adding count array value with its previous index value.
    for j in range(len(A) - 1, -1, -1):
        B[c[A[j]] - 1] = A[j]   # Putting values in output array B in sorted order.
        c[A[j]] = c[A[j]] - 1
    return B


x = []
t = []
for loop in range(1, 11):
    size = loop * 20000     # Array size.
    A = [random.randint(100, 900) for i in range(size)]
    B = [0 for i in range(len(A))]
    k = max(A)  # Maximum element in array.
    x.append(size)
    print(f'\nGiven Array: {A}')
    start = time.time_ns()
    counting_sort(A, B, k)  # Calling counting sort function.
    t.append((time.time_ns() - start)/10**3)
    print(f'\nSorted Array: {B}')
# Plotting graph
plt.title("Time Complexity Graph\nCounting Sort V/s O(n)")
plt.xlabel('Input Length (N)')
plt.ylabel('Time (in nano seconds)')
plt.plot(x, x, label='O(n)')
plt.plot(x, t, label='Counting Sort')
plt.legend()
plt.show()
```

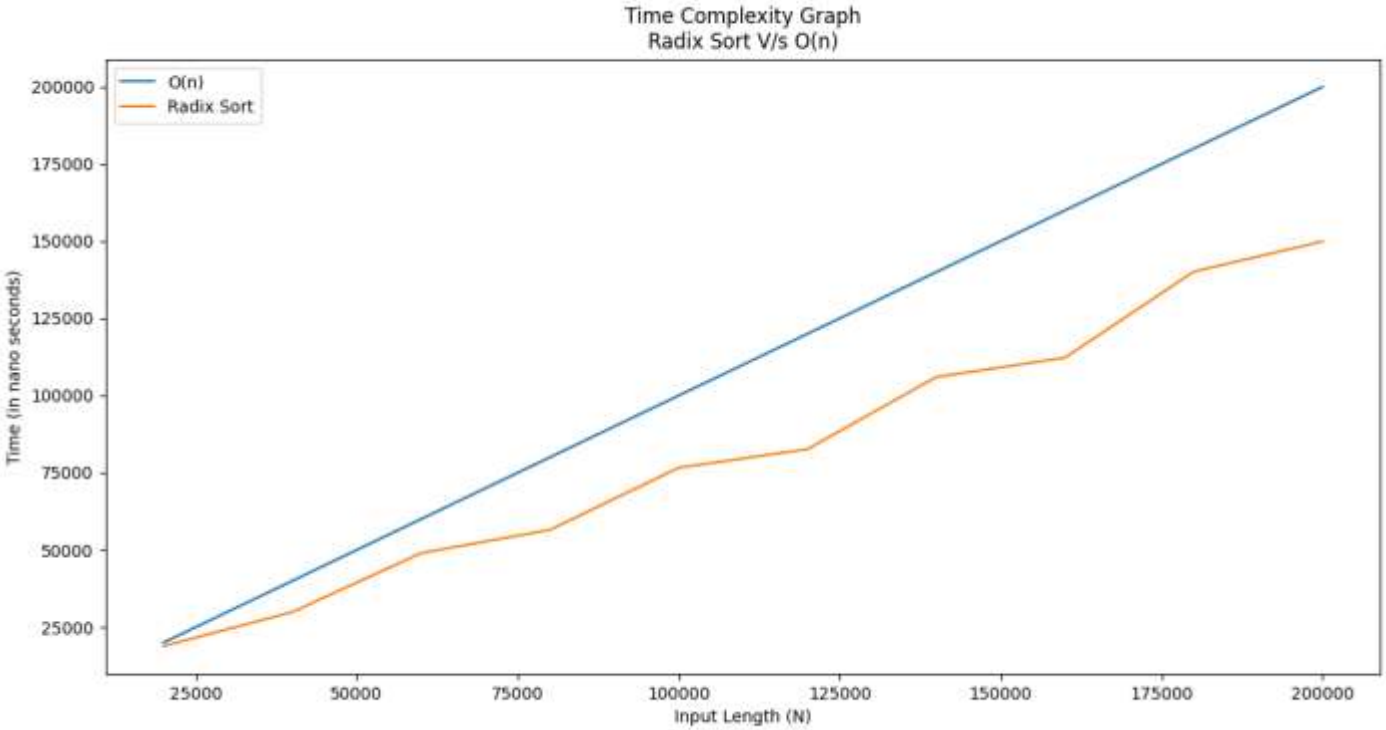
Output: Here the graph below shows the time taken by the Counting Sort algorithm for various length of input arrays and the curve follows closely the same path as O(n) when compared.



Assignment 7 - Write a program to implement the Radix sort algorithm on a d-digit number (where d is a fixed integer). Verify that the algorithm takes linear time.

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

def counting_sort(arr, exp):        # Function to perform counting sort.
    n = len(arr)
    c = [0] * 10    # Array to store count of each word.
    b = [0] * n     # Array to store output.
    for j in range(n):
        index = arr[j] // exp
        c[index % 10] += 1  # Taking modulus to get digit at preferred ones, tens, hundreds.. positions.
    for x in range(1, 10):
        c[x] += c[x-1]      # Adding count array value with its previous index value.
    for i in range(n-1, -1, -1):
        index = arr[i] // exp
        b[c[index % 10] - 1] = arr[i]
        c[index % 10] -= 1
    for i in range(n):
        arr[i] = b[i]   # Putting values in array in sorted order according to digits of number.

def radix_sort(arr):
    m = max(arr)    # Maximum value in array.
    exp = 1     # Taking variable to help in passing the digit of numbers.
    while m / exp > 1:
        counting_sort(arr, exp)     # Calling counting sort function.
        exp *= 10   # Increasing value with multiple of 10.

x = []
t = []
for loop in range(1, 11):
    size = loop * 20000     # Array size.
    arr = [random.randint(1, 900) for i in range(size)]
    x.append(size)
    print(f'\nGiven Array: {arr}')
    start = time.time_ns()
    radix_sort(arr)     # Calling Radix Sort function.
    t.append((time.time_ns() - start)/10**3.6)
    print(f'\nSorted Array: {arr}')
plt.title("Time Complexity Graph\nRadix Sort V/s O(n)")
plt.xlabel('Input Length (N)')
plt.ylabel('Time (in nano seconds)')
plt.plot(x, x, label='O(n)')
plt.plot(x, t, label='Radix Sort')
plt.show()
```

Output: Here the graph below shows the time taken by the Radix Sort algorithm for various length of input arrays and the curve follows closely the same path as O(n) when compared.

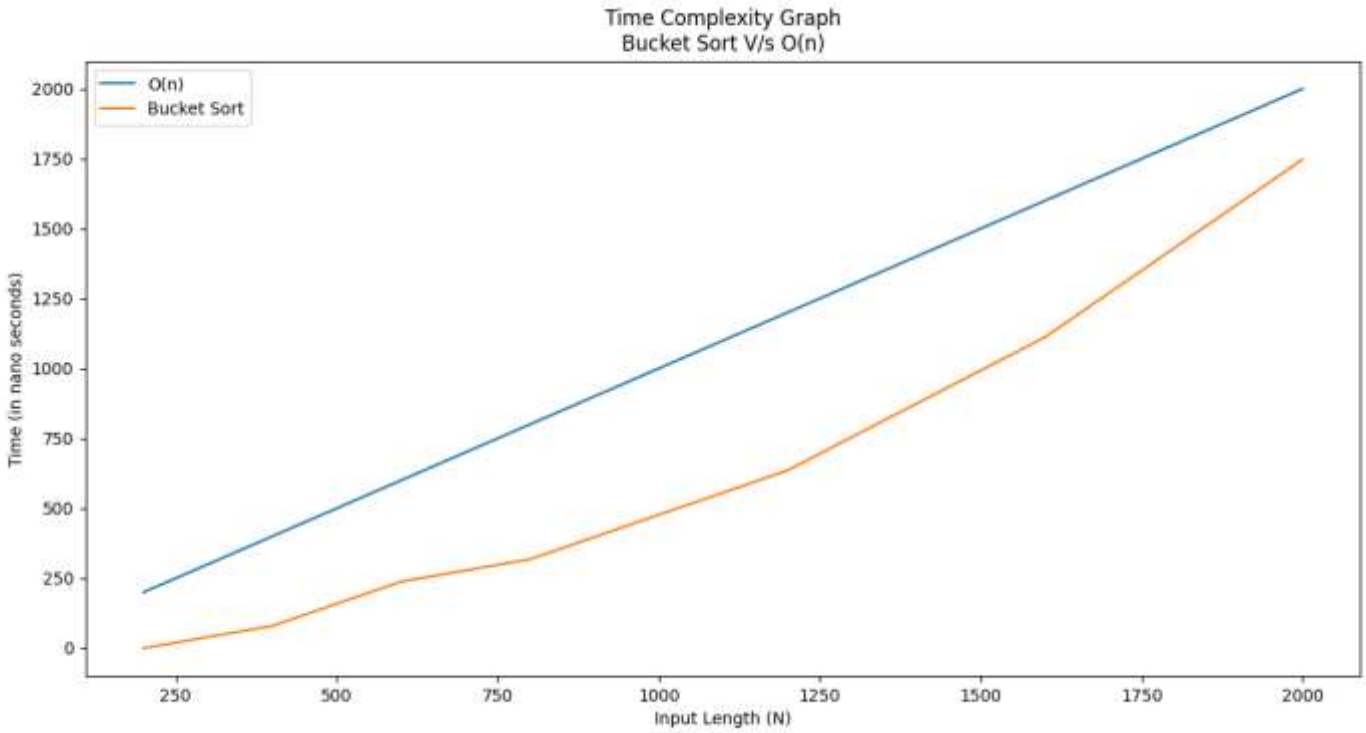


Assignment 8 - Write a program to implement the Bucket sort algorithm. Verify that it runs in linear time for inputs coming from a uniform distribution.

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


def insertion_sort(arr):
    for i in range(1, len(arr)):
        key = arr[i]    # Taking element from array for comparison.
        j = i - 1
        while j >= 0 and arr[j] > key:
            arr[j + 1] = arr[j] # Replacing if key is less than index value.
            j = j-1
        arr[j + 1] = key
    return arr


def bucket_sort(arr):
    B = []  # Creating empty array.
    for i in range(10):
        B.append([])    # Adding empty array in B as a bucket.
    for j in arr:
        index = int(10*j)
        B[index].append(j)  # Inserting values in bucket.
    for i in range(10):
        B[i] = insertion_sort(B[i]) # Calling insertion sort to sort the bucket.
    k = 0
    for x in range(10):
        for y in range(len(B[x])):
            arr[k] = B[x][y]    # Adding elements from bucket in a array after sorting.
            k += 1
    return arr


x = []
t = []
for loop in range(1, 11):
    size = loop * 200    # Array size.
    A = [random.uniform(0.001, 0.999) for i in range(size)]
    x.append(size)
    print(f'\nGiven array: {A}')
    start = time.time_ns()
    bucket_sort(A)  # Calling bucket sort function.
    t.append((time.time_ns() - start)/10**4)
    print(f'\nSorted array: {A}')
```

```
# Plotting graph
plt.title("Time Complexity Graph\nBucket Sort V/s O(n)")
plt.xlabel('Input Length (N)')
plt.ylabel('Time (in nano seconds)')
plt.plot(x, x, label='O(n)')
plt.plot(x, t, label='Bucket Sort')
plt.legend()
plt.show()
```

Output: Here the graph below shows the time taken by the Bucket Sort algorithm for various length of input arrays of uniform distribution and the curve follows closely the same path as O(n) when compared.



Assignment 9 - Write a program to implement the Knapsack Problem with the following two variations.

a). Fractional version

```
def fractional_knapsack(value, weight, capacity):
    # index = [0, 1, 2, ..., n - 1] for n items
    index = list(range(len(value)))
    # Calculating ratios of values to weight.
    ratio = [v / w for v, w in zip(value, weight)]
    # Sorting index according to value-to-weight ratio in decreasing order
    index.sort(key=lambda i: ratio[i], reverse=True)
    max_v = 0   # Variable to store maximum value that can be taken
    item_fraction = [0] * len(value)    # Variable to store fractions of items that should be taken.
    for i in index:    # Using loop to check conditions for selecting items.
        if weight[i] <= capacity:
            item_fraction[i] = 1
            max_v += value[i]
            capacity -= weight[i]
        else:
            item_fraction[i] = capacity / weight[i]
            max_v += value[i] * capacity / weight[i]
            break

    return max_v, item_fraction


# Driver code.
n = int(input('Enter number of items: '))
value = input('Enter the values of items in order: ').split()
value = [int(v) for v in value]
weight = input('Enter the weights of items in order: ').split()
weight = [int(w) for w in weight]
capacity = int(input('Enter Knapsack Maximum Weight: '))
# Calling fractional knapsack function.
max_v, item_fraction = fractional_knapsack(value, weight, capacity)
print('Maximum value that can be taken:', max_v)    # Print max value that can be taken.
print('Fraction in which the items should be taken:', item_fraction)    # Print fractions of items
selected.
```

Output:

```
Enter number of items: 3
Enter the values of items in order: 100  60  120
Enter the weights of items in order: 20  10  30
Enter Knapsack Maximum Weight: 50
Maximum value that can be taken: 240.0
Fraction in which the items should be taken: [1, 1, 0.6666666666666666]
```

b). Binary (0/1) version

```
def _0_1_knapsack(values, weights, capacity):
    number_of_items = len(values)
    table = []
    for i in range(number_of_items):
        table.append([None] * capacity)
    for j in range(capacity):
        table[0][j] = 0
    for i in range(1, number_of_items):
        for j in range(capacity):
            if weights[i-1] > j:
                table[i][j] = table[i-1][j]
            else:
                table[i][j] = max(table[i-1][j], table[i-1][j-weights[i-1]] + values[i-1])
    return table[-1][-1]


values = [100, 60, 120]
weights = [20, 10, 30]
capacity = 50
print("Values: ", values)
print("Weights: ", weights)
print("Knapsack maximum capacity: ", capacity)
print("\nMaximum value taken: ", _0_1_knapsack(values, weights, capacity))
```

Output:

```
Values:  [100, 60, 120]

Weights:  [20, 10, 30]

Knapsack maximum capacity:  50


Maximum value taken:  160
```

Assignment 10 - Write a program to find the minimum spanning tree in a given graph using the following methods.

a). Prim's Method

```
INF = 9999999
G = [[0, 5, 9, 7, 0],
    [5, 0, 10, 0, 8],
    [9, 10, 0, 20, 25],
    [7, 0, 20, 0, 11],
    [0, 8, 25, 11, 0]]
N = len(G)
selected_node = [0, 0, 0, 0, 0]
no_edge = 0
selected_node[0] = True
print("Edge : Weight\n")
while no_edge < N - 1:
   minimum = INF
   a = 0
   b = 0
   for m in range(N):
      if selected_node[m] == True:
         for n in range(N):
            if (not selected_node[n]) and G[m][n]:
               # not in selected and there is an edge
               if minimum > G[m][n]:
                  minimum = G[m][n]
                  a = m
                  b = n
   print(str(a) + " - " + str(b) + " : " + str(G[a][b]))
   selected_node[b] = True
   no_edge += 1
```

Output:

```
Edge    :    Weight
0 - 1    :    5
0 - 3    :    7
1 - 4    :    8
0 - 2    :    9
```

b). Kruskal's Method

```
def find(g, node):
    if g[node] < 0:
        return node
    else:
        temp = find(g, g[node])
        g[node] = temp
        return temp


def union(g, a, b, d, ans):
    ta = a
    tb = b
    a = find(g,a)
    b = find(g,b)
    if a == b:
        pass
    else:
        ans.append([ta, tb, d])
        if g[a] < g[b]:
            g[a] = g[a] + g[b]
            g[b] = a
        else:
            g[b] = g[a] + g[b]
            g[a] = b


weight_data = [[1, 2, 28], [2, 3, 16], [2, 7, 14], [3, 4, 12], [1, 6, 10], [7, 5, 24], [6, 5, 25], [7, 4, 18], [5, 4,
22]]
n = len(weight_data)
weight_data = sorted(weight_data, key=lambda weight_data: weight_data[2])
ans = []
g = [-1] * (n+1)
for u, v, d in weight_data:
    union(g, u, v, d, ans)
ans = sorted(ans, key=lambda ans: ans[2])
print("Vertices", ": ", "Cost")
for item in ans:
    for i,j,k in [item]:
        print(i, " - ", j, " : ", k)
```

Output:

Vertices : Cost

1 - 6 : 10

3 - 4 : 12

2 - 7 : 14

2 - 3 : 16

5 - 4 : 22

6 - 5 : 25