INDEX

Lab Sheet	S.No.	Topic	Signature & Marks
Lab Sheet 1	1.	Revise pseudocode for sorting an array (int, float, or char type) using following sorting techniques: ● Selection sort ● Bubble sort ● Merge sort (Recursive) ● Quick sort (Recursive) A. Plot the complexity chart for n=10 to 100. B. Analyze their complexities in best case, average case and worst case.	
	2.	Revise pseudocode for searching within an array (int, float, or char type) using following searching techniques: ● Linear Search ● Binary Search A. Plot the complexity chart for n=10 to 100. B. Analyze their complexities in best case, average case and worst case.	
	3.	You have been given two sorted lists of size M and N. It is desired to find the Kth smallest element out of M+N elements of both lists. Propose and implement an efficient algorithm to accomplish the task. Further, propose and implement an efficient algorithm to accomplish the task considering that elements in both lists are unsorted.	
	4.	You are given a list of n-1 integers and these integers are in the range of 1-n. There are no duplicates in the list. One of the integers is missing in the list. Write an efficient code to find the missing integer.	
	5.	You have been given a sorted array ARR (of size M, where M is very large) of two elements, 0 and 1. It is desired to compute the count of 0s in the array ARR. Propose and implement an efficient algorithm to accomplish the task.	
Lab Sheet 2	6.	Let there be an array of N random elements. We need to sort this array in ascending order. If n is very large (i.e. N= 1,00,000) then Quicksort may be considered as the fastest algorithm to sort this array. However, we can further optimize its performance by hybridizing it with insertion sort. Therefore, if n is	

		small (i.e. N<= 10) then we apply insertion sort to the array otherwise Quick Sort is applied. Implement the above discussed hybridized Quick Sort and compare the running time of normal Quick sort and hybridized quick sort. Run each type of sorting 10 times on a random set of inputs and compare the average time returned by these algorithms.	
Lab Sheet 3	7.	Implement the strassen's multiplication method (using Divide and Conquer Strategy) and naive multiplication method. Compare these methods in terms of time taken using the nXn matrix where n=3, 4, 5, 6, 7 and 8 (compare in bar graph).	
	8.	Implement the multiplication of two N-bit numbers (using Divide and Conquer Strategy) and naive multiplication method. Compare these methods in terms of time taken using N-bit numbers where n=4, 8, 16, 32 and 64.	
	9.	Maximum Value Contiguous Subsequence: Given a sequence of n numbers A(1) ...A(n), give an algorithm for finding a contiguous subsequence A(i) ...A(j) for which the sum of elements in the subsequence is maximum. Example : {-2, 11, -4, 13, -5, 2} → 20 and {1, -3, 4, -2, -1, 6 } → 7.	
	10.	Implement the algorithm (Algo_1) presented below and discuss which task this algorithm performs. Also, analyze the time complexity and space complexity of the given algorithm. Further, implement the algorithm with following modification: replace m = ⌈2n/3⌉ with m = ⌊2n/3⌋, and compare the tasks performed by the given algorithm and modified algorithm.	
Lab Sheet 4	11.	Implement LCS algorithm for A[1 .. n] and B[1 .. l] sequences.	
	12.	Given an array A[1 .. n] of integers, compute the length of a longest increasing subsequence. A sequence B[1 .. l] is increasing if B[i] > B[i − 1] for every index $i \geq 2$. For example, given the array ⟨3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5, 8, 9, 7, 9, 3, 2, 3, 8, 4, 6, 2, 7⟩	
	13.	Given an array A[1 .. n] of integers, compute the length of a longest alternating subsequence. A sequence B[1 .. l] is alternating if B[i] < B[i − 1] for every even index $i \geq 2$, and B[i] > B[i − 1] for every odd index $i \geq 3$. For example, given the array ⟨3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5, 8, 9, 7, 9, 3, 2, 3, 8, 4, 6, 2, 7⟩	
	14.	Given an array A[1 .. n], compute the length of a longest palindrome subsequence of A. Recall that a sequence B[1 .. l] is a palindrome if B[i] = B[l− i + 1] for every index i.	
	15.	Given an array A[1 .. n] of integers, compute the length of a longest convex	

		subsequence of A. A sequence B[1 .. l] is convex if B[i] − B[i − 1] > B[i − 1] − B[i − 2] for every index i ≥ 3.	
Lab Sheet5	16.	Implement MCM algorithm for the given n matrix <M1xM2… .....Mn> where the size of the matrix is Mi=di-1 x di.	
	17.	Implement OBST for given n keys (K1,K2…..Km) whose pi and qi (dummykeys) are given.	
	18.	Implement 0/1 Knapsack problem using dynamic programming.	
Lab Sheet6	19.	Wap to Implement breadth first search algorithm for given graph G.	
	20.	Wap to Implement depth first search algorithm for given graph G.	
	21.	Wap to Implement topological sorting.	
	22.	Wap to find the strongly connected components in a Graph.	
Lab Sheet7	23.	Wap to Implement Prim's algorithm for given graph G.	
	24.	Wap to Implement Kruskal's algorithm for given graph G.	
	25.	Wap to Implement dijkstra algorithm to find single source shortest path.	

Lab Sheet 1

Q1. Revise pseudocode for sorting an array (int, float, or char type) using following sorting techniques:

- **Selection sort**
- **Bubble sort**
- **Merge sort (Recursive)**
- **Quick sort (Recursive)**

A. Plot the complexity chart for n=10 to 100.

B. Analyse their complexities in best case, average case and worst case.

Pseudo Code:

```
#Bubble sort
    bubbleSort(array)
      n = length(array)
      for i = 0 to n-1 do
        for j = 0 to n-i-1 do
          if array[j] > array[j+1] then
            swap(array[j], array[j+1])
          end if
        end for
      end for
    end function
#selection sort
    selectionSort(array)
      n = length(array)
      for i = 0 to n-1 do
        minIndex = i
        for j = i+1 to n-1 do
          if array[j] < array[minIndex] then
            minIndex = j
          end if
        end for
```

```
            swap(array[i], array[minIndex])
        end for
end function
Similarly
#merge sort
#quick sort
#tester function
    Calculate time take and add to result list
        init= timer()
        sorting_func(x)
        run_time =timer()-init
        result.append((i,run_time))
plt.plot(x_axis,y1_axis,label = "Bubble sort")
plt.plot(x_axis,y2_axis,label = "Selection sort")
plt.plot(x_axis,y3_axis,label = "Merge sort")
plt.plot(x_axis,y4_axis,label = "Quick sort")
```

Program Analysis:

- Algorithm for all sorting technique (bubble, selection, merge, and quick sort)
- Bubble sort: sort two adjacent no. for entire list and loop until list will be sorted
- Selection sort: select min of all list elements and place it to beginning to sort it
- Merge sort: divide array in equal half until single element and merge again in sorted form
- Quick sort: select pivot and divide according to min and max each side of pivot.
- A tester function that calculates time required to run sorting technique on randomly generated list from size 10 to 100.
- Add these times to new result list for plotting
- Define all plotting axis.
- Plot the graph by putting time on y axis of different sorting techniques and array size on x axis.
- Add x and y labels with legends.

Calculating time taken by different sorting technique and plotting a line graph of them.

This show time complexity of algorithms like Bubble sort have some curve line as parabola as it take Big-Oh (n ^2) time to execute and other also such as

Selection sort taken O (n ^2)

Merge sort take O(nlogn)

Quick sort take O(nlogn)

This helps us to identify on different size which algorithm will work better than other. Like here quicksort has taken very less time as compared to bubble and selection on large n.

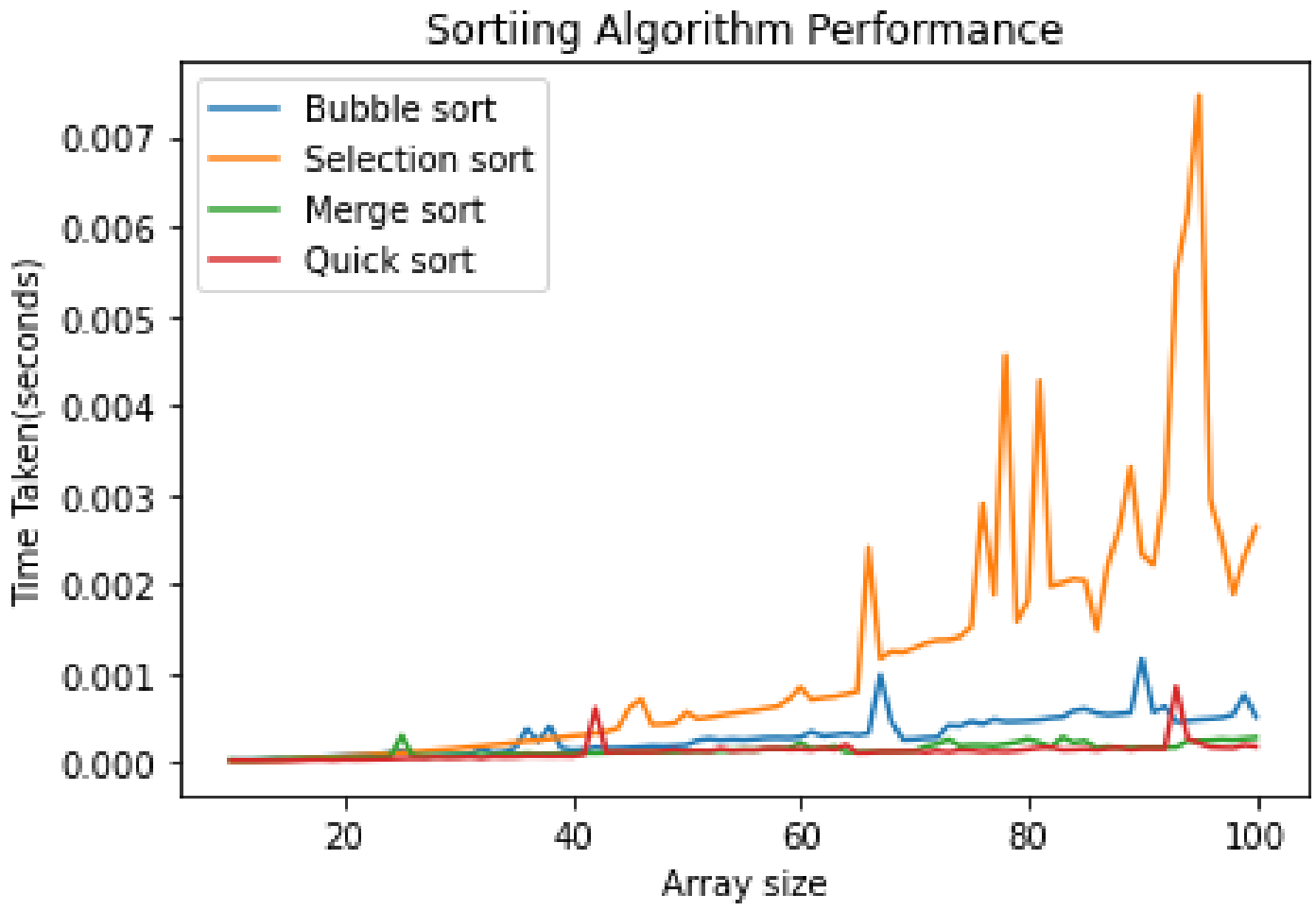
Code:

```
# Importing libraries
from timeit import default_timer as timer
import random
random.seed(4)
import matplotlib.pyplot as plt
print("Gaurav Kumar Chaurasiya")
#bubble sort
def bubble_sort(x):
    for i in range(0,len(x)-1):
        for j in range(i+1,len(x)):
            if x[i]>x[j]:
                c=x[i]
                x[i]=x[j]
                x[i+1]=c
    return(x)
#selection sort
def selection_sort(x):
    for i in range(len(x)):
        for j in range(i+1,len(x)):
            if x[i>x[j]]:
                c=x[j]
                x[i]=x[i+1]
                x[i+1]=c
    return(x)
# Merge sort
def merge_sort(x):
    if len(x)<=1:
        return x
    mid =  len(x)//2
    left =  merge_sort(x[:mid])
    right = merge_sort(x[mid:])
    return merge(left,right)
def merge(left,right):
    result=[]
```

```
    i,j=0,0
    while i < len(left) and i<len(right):
        if left[i]<right[i]:
            result.append(left[i])
            i+=1
        else:
            result.append(right[i])
            i+=1
        result +=left[i:]
        result +=right[i:]
    return result
#quick sort
def quick_sort(x):
    if len(x)<=1:
        return x
    pivot = x[len(x)//2]
    left =[x for x in x if x<pivot]
    middle =[x for x in x if x ==pivot]
    right = [x for x in x if x>pivot]
    return quick_sort(left)+middle+quick_sort(right)

# to calculate time and plot diagram
def tester(sorting_func):
    result=[]
    for i in range(10,101):
        x=[random.randint(1,101) for i in range(1,i)]
        random.shuffle(x)
        init= timer()
        sorting_func(x)
        run_time =timer()-init
        result.append((i,run_time))
    return result
bubble_result = tester(bubble_sort)
selection_result = tester(selection_sort)
merge_result = tester(merge_sort)
quick_result = tester(quick_sort)
```

```
x_axis=[d[0] for d in bubble_result]
y1_axis=[d[1] for d in bubble_result]
y2_axis=[d[1] for d in selection_result]
y3_axis =[d[1] for d in merge_result]
y4_axis =[d[1] for d in quick_result]
plt.plot(x_axis,y1_axis,label = "Bubble sort")
plt.plot(x_axis,y2_axis,label = "Selection sort")
plt.plot(x_axis,y3_axis,label = "Merge sort")
plt.plot(x_axis,y4_axis,label = "Quick sort")
plt.xlabel("Array size")
plt.ylabel("Time Elapsed(seconds)")
plt.title('Algorithm Performance')
plt.legend()
plt.show()
```

Output :



Q2. Revise pseudocode for searching within an array (int, float, or char type) using following searching techniques:

- **Linear Search**
- **Binary Search**

A. Plot the complexity chart for n=10 to 100.

B. Analyse their complexities in best case, average case and worst case.

Pseudo code :

 # Linear search

```
def linear_search(arr, element):
    ans = None
    for i in arr:
        if i == element:
            return arr.index(i)
    return None
```

 # Binary Search

```
def binary_search(array, element):
    low=0
    high=len(array)-1
    while low <= high:
        mid = low + (high - low)//2
        if array[mid] == element:
            return mid
            return mid, (end-start)
        elif array[mid] < element:
            low = mid + 1
        else:
            high = mid - 1
    return None
```

calculating time required to search element and plot them on matplot.

```
        plt.plot(x_axis,y1_axis,label = "Linear Search")
        plt.plot(x_axis,y2_axis,label = "Bineary Search")
```

Program Analysis:

Linear Search: select key element and search in entire list. If found return index else -1

Binary Search: take mid of list and compare to key if it key than return else move to either left or right according to key. and do again same steps

Linear Search take O (N) time. and it length of list is greater and element present in last it will traverse whole list.

But bubble sort takes O (Nlog N) as it divided list in half at each steps.

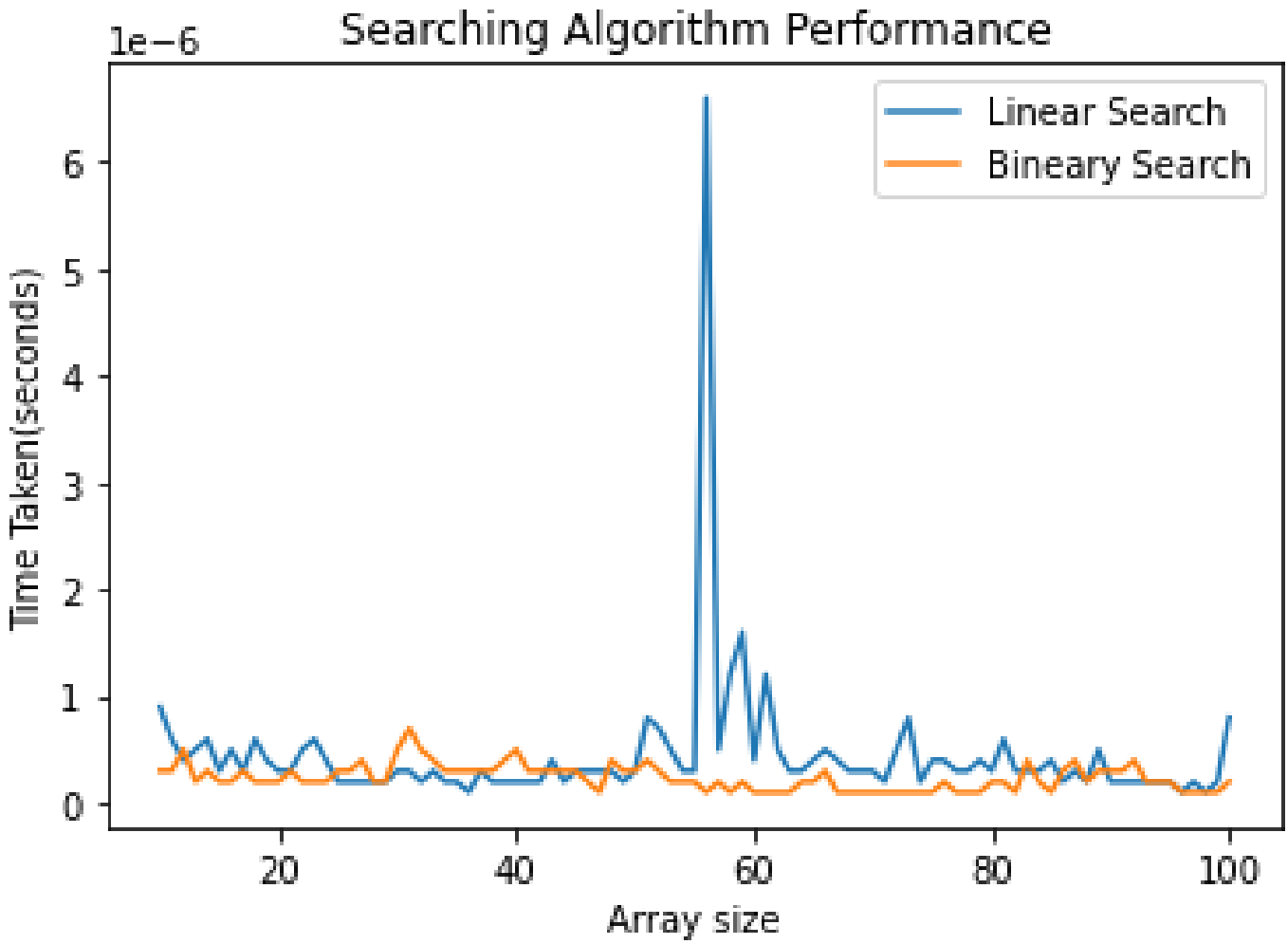
The code shows the time representation over list size how varying it.

Code :

```
from timeit import default_timer as timer
import random
random.seed(4)
import matplotlib.pyplot as plt
#print("Gaurav Kumar Chaurasiya")

# Linear search
def linear_search(arr, element):
    ans = None
    for i in arr:
        if i == element:
            return arr.index(i)
    return None

# Binary Search
def binary_search(array, element):
    low=0
    high=len(array)-1
    while low <= high:
        mid = low + (high - low)//2
        if array[mid] == element:
            return mid
            return mid, (end-start)
```

```
        elif array[mid] < element:
            low = mid + 1
        else:
            high = mid - 1
    return None

def tester(searching_func):
    result=[]
    for i in range(10,101):
        x=[random.randint(1,101) for i in range(1,i)]
        random.shuffle(x)
        init= timer()
        run_time =timer()-init
        result.append((i,run_time))
    return result
linear_result = tester(linear_search)
binary_result = tester(binary_search)

x_axis=[d[0] for d in linear_result]
y1_axis=[d[1] for d in linear_result]
y2_axis=[d[1] for d in binary_result]

plt.plot(x_axis,y1_axis,label = "Linear Search")
plt.plot(x_axis,y2_axis,label = "Bineary Search")
plt.xlabel("Array size")
plt.ylabel("Time Elapsed(seconds)")
plt.title('Searching Algorithm Performance')
plt.legend()
plt.show()
```

Output:


Searching Algorithm Performance
1e−6
Linear Search
Bineary Search
Time Taken(seconds)
0
1
2
3
4
5
6
20
40
60
80
100
Array size

Q3. You have been given two sorted lists of size M and N. It is desired to find the Kth smallest element out of M+N elements of both lists. Propose and implement an efficient algorithm to accomplish the task. Further, propose and implement an efficient algorithm to accomplish the task considering that elements in both lists are unsorted.

Pseudo code:

```
function kth_element(a,b,m,n,k):
    initialize an array c of size m+n with all elements set to 0
     i = 0
    j = 0
    d = 0
     while i < m and j < n:
        if a[i] < b[j]:
            c[d] = a[i]
            i += 1
        else:
            c[d] = b[j]
            j += 1
        d += 1


    while i < m:
        c[d] = a[i]
        d += 1
        i += 1
     while j < n:
        c[d] = b[j]
        d += 1
        j += 1
     return c[k-1]
```

Program Analysis:

- The given program implements the kth_element function that takes in two sorted arrays a and b of sizes m and n respectively, and an integer k. The function merges the two arrays into a sorted array c using the merge operation of merge-sort, and returns the kth element of c.

- The time complexity of the program is O(m+n), which is the time required to merge the two sorted arrays. The program has two while loops to merge the two arrays, which iterate a maximum of m+n times, and hence the time complexity is O(m+n).

- The space complexity of the program is O(m+n), which is the space required to store the merged array c. The program creates an array c of size m+n to store the merged array.

- For Unsorted part first we sort them using quick sort.
- Overall, the given program is a simple and efficient way to find the kth element in two sorted arrays.

Code:

```
# print("Gaurav Kumar Chaurasiya")
def kth_element(a,b,m,n,k):
    c=[0]*(m+n)
    i=j=d=0
    while(i<m and j<n):
        if(a[i]<b[j]):
            c[d]=a[i]
            i+=1
        else:
            c[d]=b[j]
            j+=1
            d+=1
        while(i<m):
            c[d]=a[i]
            d+=1
            i+=1
        while(j<n):
            c[d]=b[j]
            d+=1
            j+=1
    return c[k-1]
a=[2,3,5,7,9]
b=[1,4,6,8,10]
k=5
```

```
print(kth_element(a,b,5,4,k))
```

Output :

7

2 part Code :

```
# UNSORTED PART
def kth_smallest(array,k):
    if len(array) == 1:
        return a[0]
    pivot = array[0]
    lower =[x for x in array if x<pivot]
    equal =[x for x in array if x ==pivot]
    higher = [x for x in array if x>pivot]
    if k<= len(lower):
        return kth_smallest(lower,k)
    elif k>len(lower) + len(equal):
        return kth_smallest(higher,k-len(lower) - len(equal))
    else:
        return equal[0]
m=4
n=4
k=3
array1=[6,7,5,4,6,78,9]
array2=[8,4,6,3,2]
array1.sort()
print(kth_smallest(array1,k))
```

Output :

6

Q4. You are given a list of n-1 integers and these integers are in the range of 1-n. There are no duplicates in the list. One of the integers is missing in the list. Write an efficient code to find the missing integer.

Pseudo Code:

Creating a list using numpy

```
def list_maker(n):
    num_list = np.arange(1, n)
    pop_ele = random.randint(0, n)
    print(f'The actual missing element is:{num_list[pop_ele]}'
```

finding missing value

```
    missing_ele = (n*(n-1))/2 - sum_1
```

Program Analysis:

- This program uses the NumPy library to create a list of numbers from 1 to n, where n is a given integer. It then randomly removes one element from the list and prints the value of the removed element. The program then calculates the sum of the remaining elements in the list and subtracts it from the sum of all numbers from 1 to n, which gives the value of the missing element.
- The program first imports the NumPy library as np and the random library. It then defines a function list_maker(n) that takes an integer n as input, creates a NumPy array of numbers from 1 to n, randomly selects an element from the array, prints its value, deletes it from the array, and returns the modified array and the index of the removed element.
- The program then sets n to 100, calls the list_maker(n) function to create a modified NumPy array num_list and print the value of the removed element, and initializes a variable sum_1 to 0. It then loops through the elements of the modified array num_list, adding each element to sum_1. Finally, the program calculates the expected sum of all numbers from 1 to n using the formula (n*(n-1))/2 and subtracts the actual sum sum_1 from it to obtain the missing element. The program prints the calculated missing element value.

Code :

```
import numpy as np
import random
def list_maker(n):
```

```
    num_list = np.arange(1, n)
    pop_ele = random.randint(0, n)
    print(f'The actual missing element is:{num_list[pop_ele]}')
    num_list = np.delete(num_list, pop_ele)  # Deleting a random index
value
    return num_list, pop_ele


n = 100  # Number of elements in list
num_list, pop_ele = list_maker(n)
sum_1 = 0
for i in num_list:
    sum_1 += i
missing_ele = (n*(n-1))/2 - sum_1
print(f'The calculated missing element from algorithm is:{missing_ele}')
```

Output:

```
The actual missing element is:40
The calculated missing element from algorithm is:40.0
```

Q5. You have been given a sorted array ARR (of size M, where M is very large) of two elements, 0 and 1. It is desired to compute the count of 0s in the array ARR. Propose and

implement an efficient algorithm to accomplish the task.

Pseudo Code:

```
Function  countOnes(arr, low, high): # taking array low and high value as  arguments
   if high >= low:
      mid = low + (high-low)//2
      # check if the element at middle index is last 1
      if ((mid == high or arr[mid+1] == 0) and (arr[mid] == 1)):
         return mid+1
      # If element is not last 1, recur for right side
      if arr[mid] == 1:
         return countOnes(arr, (mid+1), high)
      # else recur for left side
      return countOnes(arr, low, mid-1)
   return 0
```

Program Analysis:

- The countOnes function takes three parameters: arr, low, and high. arr is the input binary array, and low and high are the lower and upper indices of the array to be searched, respectively.

- The function first calculates the middle index of the array using the formula mid = low + (high-low)//2. If the element at the middle index is the last 1 in the array, the function returns the index of the last 1 by adding 1 to the middle index (mid+1). Otherwise, if the element at the middle index is 1 but not the last one, the function recursively calls itself on the right side of the array (countOnes(arr, (mid+1), high)). If the element at the middle index is 0, the function recursively calls itself on the left side of the array (countOnes(arr, low, mid-1)). The function returns 0 if there are no 1s in the array.

- Finally, the code initializes an input binary array arr and calls the countOnes function with low and high indices set to 0 and len(arr)-1, respectively. The output is the index of the last 1 in the array, which is printed to the console.
- In terms of complexity, the time complexity of the countOnes function is O(log n) since the function performs a binary search on the input array, where n is the number of elements in the array. The space complexity is O(1) since the function uses only a constant amount of memory to store the indices and the middle element. Overall, the given code is an efficient way to find the index of the last occurrence of 1 in a sorted binary array**.**

Code :

```
def countOnes(arr, low, high):
    if high >= low:
        mid = low + (high-low)//2
    # check if the element at middle index is last 1
    if ((mid == high or arr[mid+1] == 0) and (arr[mid] == 1)):
        return mid+1
    # If element is not last 1, recur for right side
    if arr[mid] == 1:
        return countOnes(arr, (mid+1), high)
    # else recur for left side
    return countOnes(arr, low, mid-1)
    return 0


arr = [1, 1, 1, 1, 1, 0, 0, 0]
print("Count of 1's in given array is", countOnes(arr, 0, len(arr)-1))
```

Output:

```
Count of 1's in given array is 5
```

Lab Sheet 2

Q6. Let there be an array of N random elements. We need to sort this array in ascending order. If n is very large (i.e. N= 1,00,000) then Quicksort may be considered as the fastest algorithm to sort this array. However, we can further optimize its performance by hybridizing it with insertion sort. Therefore, if n is small (i.e. N<= 10) then we apply insertion sort to the array otherwise Quick Sort is applied. Implement the above discussed hybridized Quick Sort and compare the running time of normal Quick sort and hybridized quick sort. Run each type of sorting 10 times on a random set of inputs and compare the average time returned by these algorithms

Program Analysis:

1. The code imports the necessary libraries, which are time, random, and matplotlib.pyplot.
2. The quick_sort() function takes an array as input and recursively sorts it using the quicksort algorithm.
3. The insertion_sort() function takes an array as input and sorts it using the insertion sort algorithm
4. The hybrid_quick_sort() function takes an array as input and uses the quicksort algorithm for large arrays (with more than 10 elements) and the insertion sort algorithm for small arrays (with 10 or fewer elements). It first checks if the length of the array is

less than or equal to 10. If so, it calls the insertion_sort() function to sort the array. Otherwise, it uses the quicksort algorithm to sort the array.
5. The code generates a random array of length 10000 using the random library.
6. The code generates a random array of length 10000 using the random library.
7. The code then measures the time taken to run quick_sort() and hybrid_quick_sort() 10 times each on the same array. For each sorting algorithm, it measures the time taken to run the sorting algorithm 50 times, and takes the average time taken.
8. Finally, the code uses matplotlib.pyplot to plot a bar graph comparing the average times taken by quick_sort() and hybrid_quick_sort(). The x-axis shows the 10 runs of the sorting algorithms, and the y-axis shows the time

Pseudo Code:

```
function quick_sort(arr):
    if length of arr is less than or equal to 1:
        return arr
    else:
        pivot = arr[0]
        left = []
        right = []
        for i from 1 to length of arr - 1:
            if arr[i] < pivot:
                append arr[i] to left
            else:
                append arr[i] to right
        return concatenate (quick_sort(left), [pivot], quick_sort(right))


function insertion_sort(arr):
    for i from 1 to length of arr - 1:
        key = arr[i]
        j = i - 1
        while j >= 0 and arr[j] > key:
            set arr[j+1] to arr[j]
            decrement j by 1
        set arr[j+1] to key
    return arr


function hybrid_quick_sort(arr):
    if length of arr is less than or equal to 10:
        return insertion_sort(arr)
    else:
        pivot = arr[0]
        left = []
        right = []
```

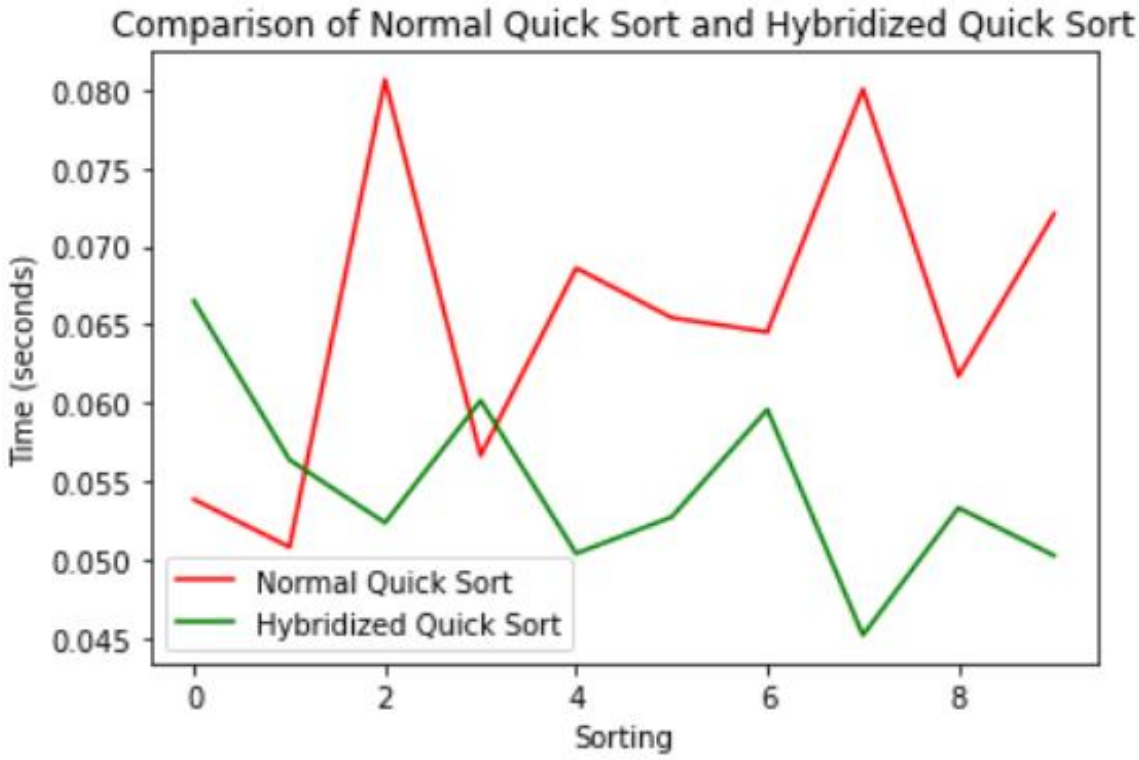
```
    for i from 1 to length of arr - 1:
      if arr[i] < pivot:
        append arr[i] to left
      else:
        append arr[i] to right
    return concatenate (hybrid_quick_sort(left), [pivot], hybrid_quick_sort(right))


set arr to a list of 10000 random integers between 0 and 10000 inclusive
set quick_sort_times to an empty list
repeat 10 times do the following:
  set quick_sum to 0
  repeat 50 times do the following:
    set start_time to the current time
    call quick_sort function with arr as argument
    set end_time to the current time
    add (end_time - start_time) to quick_sum
  append (quick_sum/50) to quick_sort_times
same for hybrid quick sort
set hybrid_quick_sort_times to an empty list

plot the graph of quick_sort_times and hybrid_quick_sort_
```

Code:

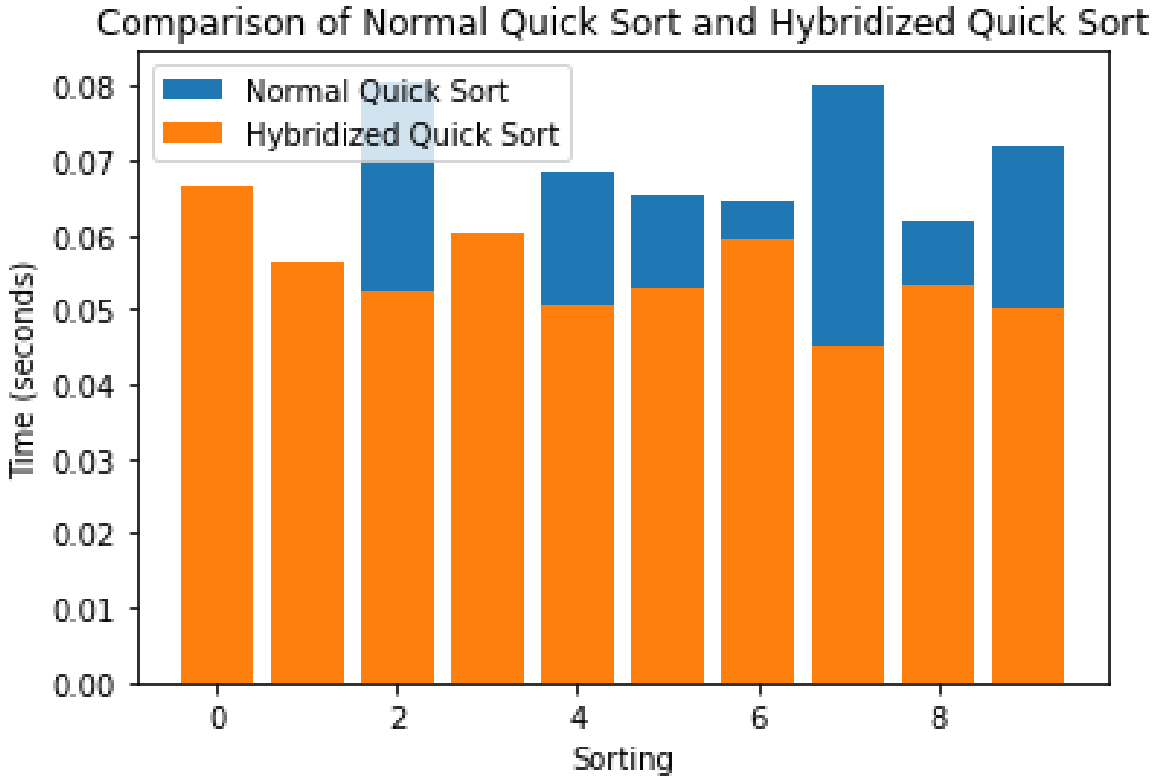
```
import time
import random
import matplotlib.pyplot as plt
print("Gaurav Kumar Chaurasiya")
def quick_sort(arr):
    if len(arr) <= 1:
        return arr
    else:
        pivot = arr[0]
        left = []
        right = []
```

```
        for i in range(1, len(arr)):
            if arr[i] < pivot:
                left.append(arr[i])
            else:
                right.append(arr[i])
        return quick_sort(left) + [pivot] + quick_sort(right)
def insertion_sort(arr):
    for i in range(1, len(arr)):
        key = arr[i]
        j = i - 1
        while j >= 0 and arr[j] > key:
            arr[j + 1] = arr[j]
            j -= 1
        arr[j + 1] = key
    return arr

def hybrid_quick_sort(arr):
    if len(arr) <= 10:
        return insertion_sort(arr)
    else:
        pivot = arr[0]
        left = []
        right = []
        for i in range(1, len(arr)):
            if arr[i] < pivot:
                left.append(arr[i])
            else:
                right.append(arr[i])
        return hybrid_quick_sort(left) + [pivot] +
hybrid_quick_sort(right)

arr = [random.randint(0, 10000) for _ in range(10000)]
# print(arr)
quick_sort_times = []
for i in range(10):
```

```
    quick_sum=0
    for j in range(50):
        start_time = time.time()
        quick_sort(arr)
        end_time = time.time()
        quick_sum+=(end_time - start_time)
    quick_sort_times.append(quick_sum/50)


hybrid_quick_sort_times = []
for i in range(10):
    hybrid_sum=0;
    for j in range(50):
        start_time = time.time()
        hybrid_quick_sort(arr)
        end_time = time.time()
        hybrid_sum+=(end_time-start_time)
    hybrid_quick_sort_times.append(hybrid_sum/50)


plt.plot(range(10), quick_sort_times, label="Normal Quick
Sort",c="r")
plt.plot(range(10), hybrid_quick_sort_times, label="Hybridized
Quick Sort",c="g")
plt.title("Comparison of Normal Quick Sort and Hybridized Quick
Sort")
plt.xlabel("Sorting")
plt.ylabel("Time (seconds)")
plt.legend()
plt.show()
```




```
plt.bar(range(10), quick_sort_times, label="Normal Quick Sort")

plt.bar(range(10), hybrid_quick_sort_times, label="Hybridized Quick
Sort")

plt.title("Comparison of Normal Quick Sort and Hybridized Quick
Sort")

plt.xlabel("Sorting")

plt.ylabel("Time (seconds)")

plt.legend()

plt.show()
```


Comparison of Normal Quick Sort and Hybridized Quick Sort
Normal Quick Sort
Hybridized Quick Sort
Time (seconds)
0.00
0.01
0.02
0.03
0.04
0.05
0.06
0.07
0.08
0
2
4
6
8
Sorting

\

Lab Sheet 3

Q8. Implement the multiplication of two N-bit numbers (using Divide and Conquer Strategy) and naive multiplication method. Compare these methods in terms of time taken using N-bit numbers where n=4, 8, 16, 32 and 64.

Pseudo Code:

```
# Naive multiplication method
function multiply(num1, num2):
    mult=0
    for i in range(num2):
        mult=mult+num1;
     return(mult)


# Divide and Conquer multiplication method
function divide_and_conquer_multiply(num1, num2):
    n = max(len(str(num1)), len(str(num2)))
    if n <= 2: # Base case for small numbers
        return num1 * num2

    # Dividing the numbers into half
    a = int(str(num1)[:n//2] if len(str(num1)) > n//2 else 0)
    b = int(str(num1)[n//2:] if len(str(num1)) > n//2 else num1)
    c = int(str(num2)[:n//2] if len(str(num2)) > n//2 else 0)
    d = int(str(num2)[n//2:] if len(str(num2)) > n//2 else num2)

    # Recursive calls
    ac = divide_and_conquer_multiply(a, c)
    bd = divide_and_conquer_multiply(b, d)
    ad_bc = divide_and_conquer_multiply(a+b, c+d) - ac - bd

    # Multiplying the parts and combining the results
    return (ac * 10**(2*(n//2))) + (ad_bc * 10**(n//2)) + bd
```

```
loop n in bit_lengths:
    loop j in range(n):
        k = random.randint(0, 9)
        l = random.randint(0, 9)
        num1 += str(k)
        num2 += str(l)
```

Program Analysis:

- Naïve multiplication is simply used loop to add n to m times (simple multiplication) eg : 2*3 = 2+2+2=6
- Divide and conquer Mulitplication divide the no. into half until reach to smallest digit and then simply multipy them to get multiplication
- Two function multiplication and DCM do above things
- Then calculated time to run them and
- Plot them by putting time on y axis and number of bits on x axis

Code:

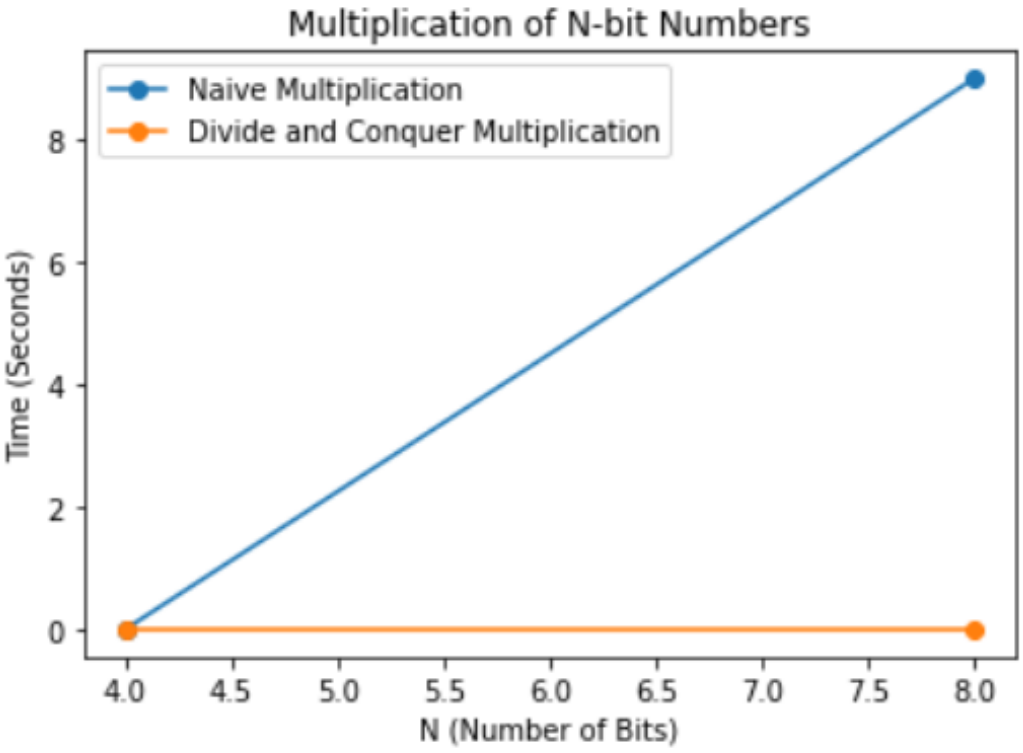
```
import random
import time
import matplotlib.pyplot as plt

# Naive multiplication method
def multiply(num1, num2):
    mult=0
    for i in range(num2):
        mult=mult+num1;

    return(mult)

# Divide and Conquer multiplication method
def divide_and_conquer_multiply(num1, num2):
    n = max(len(str(num1)), len(str(num2)))
    if n <= 2: # Base case for small numbers
        return num1 * num2

    # Dividing the numbers into half
    a = int(str(num1)[:n//2] if len(str(num1)) > n//2 else 0)
```

```
    b = int(str(num1)[n//2:] if len(str(num1)) > n//2 else num1)
    c = int(str(num2)[:n//2] if len(str(num2)) > n//2 else 0)
    d = int(str(num2)[n//2:] if len(str(num2)) > n//2 else num2)

    # Recursive calls
    ac = divide_and_conquer_multiply(a, c)
    bd = divide_and_conquer_multiply(b, d)
    ad_bc = divide_and_conquer_multiply(a+b, c+d) - ac - bd

    # Multiplying the parts and combining the results
    return (ac * 10**(2*(n//2))) + (ad_bc * 10**(n//2)) + bd

# bit_lengths = [4, 8, 16, 32, 64]
bit_lengths = [4, 8]
naive_times = []
dc_times = []

for n in bit_lengths:
    num1 = ''
    num2 = ''
    for j in range(n):
        k = random.randint(0, 9)
        l = random.randint(0, 9)
        num1 += str(k)
        num2 += str(l)

    start_time = time.time()
    result1 = multiply(int(num1), int(num2))
    print("result1",result1)
    end_time = time.time()
    naive_times.append(end_time - start_time)

    start_time = time.time()
    result2 = divide_and_conquer_multiply(int(num1), int(num2))
```

```
    print("result2",result1)
    end_time = time.time()
    dc_times.append(end_time - start_time)

# Plot the results
plt.plot(bit_lengths, naive_times, 'o-', label='Naive Multiplication')
plt.plot(bit_lengths, dc_times, 'o-', label='Divide and Conquer
Multiplication')
plt.title('Multiplication of N-bit Numbers')
plt.xlabel('N (Number of Bits)')
plt.ylabel('Time (Seconds)')
plt.legend()
plt.show()
```

Output:

```
result1 42795
result2 42795
result1 1451726918091650
result2 1451726918091650
```



Q 7. Implement the strassen's multiplication method (using Divide and Conquer Strategy) and naive multiplication method. Compare these methods in terms of time taken using the nXn matrix where n=3, 4, 5, 6, 7 and 8 (compare in bar graph).

Program Analysis:

- The program compares the time taken by two multiplication methods - Divide and Conquer and Naive.
  - The time and space complexity
    - Time Complexity: O(n^3) because three nested loop
    - Space Complexity: O(n^2) takes a new matrix of size n x n to store the result.
  - As its complexity is cubic So, to reduce this Strassen's comes with new way for matrix mulitplcation
  - Strassen's Matrix Multiplication:
    - Time complexity
      - T(n) = **7**T(n/2) + O(n^2) #while naïve muliti take 8 multiplcation
  - By applying the Master Theorem to this recurrence relation, we obtain:
    - T(n) = O(n^log2(7)) = O(n^2.81)
  - The space complexity of Strassen's algorithm is O(n^2)

Pseudo Code:

```
function strassen_multiply(A, B):
n = A.shape[0]
if n == 1:
   return A * B
else:
    mid = n // 2
A11 = A[:mid, :mid]
A12 = A[:mid, mid:]
A21 = A[mid:, :mid]
A22 = A[mid:, mid:]
B11 = B[:mid, :mid]
B12 = B[:mid, mid:]
B21 = B[mid:, :mid]
B22 = B[mid:, mid:]
P1 = strassen_multiply(A11 + A22, B11 + B22)
P2 = strassen_multiply(A21 + A22, B11)
```

P3 = strassen_multiply(A11, B12 - B22)

P4 = strassen_multiply(A22, B21 - B11)

P5 = strassen_multiply(A11 + A12, B22)

P6 = strassen_multiply(A21 - A11, B11 + B12)

P7 = strassen_multiply(A12 - A22, B21 + B22)

C = np.zeros((n, n))

C[:mid, :mid] = P1 + P4 - P5 + P7

C[:mid, mid:] = P3 + P5

C[mid:, :mid] = P2 + P4

C[mid:, mid:] = P1 - P2 + P3 + P6

return C

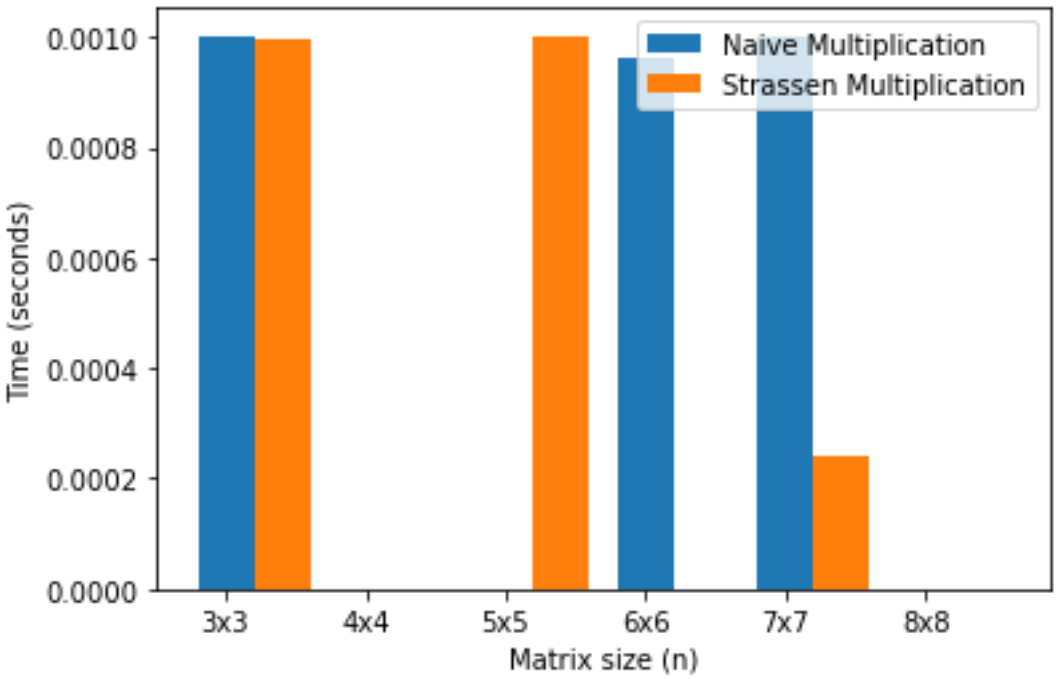
Code:

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

def strassen_multiply(A, B):
    n = A.shape[0]
    if n == 1:
        return A * B
    else:
        mid = n // 2
        A11 = A[:mid, :mid]
        A12 = A[:mid, mid:]
        A21 = A[mid:, :mid]
        A22 = A[mid:, mid:]
        B11 = B[:mid, :mid]
        B12 = B[:mid, mid:]
        B21 = B[mid:, :mid]
        B22 = B[mid:, mid:]
        P1 = strassen_multiply(A11 + A22, B11 + B22)
```

```
        P2 = strassen_multiply(A21 + A22, B11)
        P3 = strassen_multiply(A11, B12 - B22)
        P4 = strassen_multiply(A22, B21 - B11)
        P5 = strassen_multiply(A11 + A12, B22)
        P6 = strassen_multiply(A21 - A11, B11 + B12)
        P7 = strassen_multiply(A12 - A22, B21 + B22)
        C = np.zeros((n, n)) #to store result
        C[:mid, :mid] = P1 + P4 - P5 + P7
        C[:mid, mid:] = P3 + P5
        C[mid:, :mid] = P2 + P4
        C[mid:, mid:] = P1 - P2 + P3 + P6
        return C

def naive_multiply(A, B):
    n = A.shape[0]
    C = np.zeros((n, n))
    for i in range(n):
        for j in range(n):
            for k in range(n):
                C[i][j] += A[i][k] * B[k][j]
    return C

n_values = [3, 4, 5, 6, 7, 8]

# Time taken by methods
naive_times = []
strassen_times = []
for n in n_values:
    A = np.arange(16).reshape(4,4)
    B = np.arange(16,32).reshape(4,4)
    start_time = time.time()
    naive_multiply(A, B)
    end_time = time.time()
```

```
    naive_times.append(end_time - start_time)
    start_time = time.time()
    strassen_multiply(A, B)
    end_time = time.time()
    strassen_times.append(end_time - start_time)

# Plot the results
plt.bar(n_values, naive_times, width=0.4, label='Naive Multiplication')
plt.bar([n+0.5 for n in n_values], strassen_times, width=0.4,
label='Strassen Multiplication')
plt.xlabel('Matrix size (n)')
plt.ylabel('Time (seconds)')
plt.xticks(n_values, [f"{n}x{n}" for n in n_values])
plt.legend()
plt.show()
```

Output:



Q9. Maximum Value Contiguous Subsequence: Given a sequence of n numbers A(1) ...A(n), give an algorithm for finding a contiguous subsequence A(i) ...A(j) for which the sum of elements in the subsequence is maximum. Example : {-2, 11, -4, 13, -5, 2} → 20 and {1, -3, 4, -2, -1, 6 } → 7.

Pseudo Code:

```
total_max_sum = array[0] #assign total and current to first
current_sum = array[0]

loop  for i in range(1, len(array)): #stating with 2 element till end
    current_sum = max(array[i], current_sum + array[i])
    total_max_sum = max(total_max_sum, current_sum)
```

Program Analysis:

This algorithm is also known as **Kadane's Algorithm**

As the loop iterate for size of array or list. Then

- Time Complexity: O(n)

No extra data structure is used (only variables) so

- Space Complexity: O(1)

Code:

```
def max_contiguous_subsequence(array):
    total_max_sum = array[0] #assign total and
current to first
    current_sum = array[0]

    for i in range(1, len(array)): #stating with 2
element till end
```

```
            current_sum = max(array[i], current_sum +
    array[i])
            total_max_sum = max(total_max_sum,
    current_sum)


        return total_max_sum
```

Output:

```
In [6]: arr1 = [-2, 11, -4, 13, -5, 2]
        print(max_contiguous_subsequence(arr1))

        20

In [7]: arr2 = [1, -3, 4, -2, -1, 6]
        print(max_contiguous_subsequence(arr2))

        7
```

Q10. Implement the algorithm (Algo_1) presented below and discuss which task this algorithm performs. Also, analyse the time complexity and space complexity of the given algorithm. Further, implement the algorithm with following modification: replace m = ⌈2n/3⌉ with m = ⌊2n/3⌋, and compare the tasks performed by the given algorithm and modified algorithm.

Algo_1(A [0 ... n-1])

{ if n = 2 and A[0] > A[1]

swap A[0] ↔ A[1]

else if n > 2

m = ⌈2n/3⌉

Algo_1 (A [0 .. m − 1])

Algo_1 (A [n – m .. n − 1])

Algo_1 (A [0 .. m − 1]) }

Program Analysis:

Algo_1 used ceil value of (2n/3).

Algo_1, performs the task of sorting an array of numbers in non-decreasing order. It uses a divide-and-conquer approach to recursively divide the array into smaller subarrays, sort them individually, and then merge them back together.

Time Complexity:

$$T(n) = 3T(2n/3) + O(1)$$

Using the Master Theorem, time complexity of Algo_1 is O(n log(n)).

The space complexity is O(log(n)) due to the recursion stack.

And the modified Algo has change only from ceil value to floor value . so time and space complexity will remain same.

Pseudo Code:

```
if n == 2 and A[left] > A[right]:


    A[left], A[right] = A[right], A[left]


  elif n > 2:
#       m = (2 * n + 2) // 3  # Ceiling division
    m= math.ceil(2*n/3)
```

algo_1(A, left, left + m - 1)

algo_1(A, right - m + 1, right)

algo_1(A, left, left + m - 1)

Code:

```
Import math
def algo_1(A, left, right):
    n=right - left + 1
    if n == 2 and A[left] > A[right]:

        A[left], A[right] = A[right], A[left]

    elif n > 2:
#         m = (2 * n + 2) // 3  # Ceiling division
        m= math.ceil(2*n/3)
        algo_1(A, left, left + m - 1)
        algo_1(A, right - m + 1, right)
        algo_1(A, left, left + m - 1)

# Example usage:
array = [1, 5, 3, 9, 2, 7, 4, 6, 8]
algo_1(array, 0, len(array) - 1)
print(array)

def modified_algo_1(A, left, right):
    n=right - left + 1
    if n == 2 and A[left] > A[right]:

        A[left], A[right] = A[right], A[left]

    elif n > 2:
        m = math.floor((2 * n) / 3)
        modified_algo_1(A, left, left + m - 1)
        modified_algo_1(A, right - m + 1, right)
        modified_algo_1(A, left, left + m - 1)

# Example usage:
sequence = [1, 5, 3, 9, 2, 7, 4, 6, 8]
modified_algo_1(sequence, 0, len(sequence) - 1)
print(sequence)
```

Output:

In [3]:
```
def algo_1(A, left, right):
    n=right - left + 1
    if n == 2 and A[left] > A[right]:

        A[left], A[right] = A[right], A[left]

    elif n > 2:
#         m = (2 * n + 2) // 3  # Ceiling division
        m= math.ceil(2*n/3)
        algo_1(A, left, left + m - 1)
        algo_1(A, right - m + 1, right)
        algo_1(A, left, left + m - 1)

# Example usage:
array = [1, 5, 3, 9, 2, 7, 4, 6, 8]
algo_1(array, 0, len(array) - 1)
print(array)
```

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

```
        modified_algo_1(A, left, left + m - 1)

# Example usage:
sequence = [1, 5, 3, 9, 2, 7, 4, 6, 8]
modified_algo_1(sequence, 0, len(sequence) - 1)
print(sequence)
```

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

Lab Sheet 4

Q11. Implement LCS algorithm for A[1 .. n] and B[1 .. l] sequences.

Program Analysis:

- LCS(longest Common Subsequence) is a sequence that is generated by deleting(finding common btw them) some characters (possibly 0) from the string without altering the order of the remaining characters.

- In this program we have taken 2 2-D array one for LCS size and other for storing direction to get subsequence.

- Then applying the condition :
  - if x[i] == y[j]:
  - c[i][j] = c[i-1][j-1] + 1

  otherwise
  - c[i][j] = max(c[i][j-1],c[i-1][j])

- then taking last of cost table value at index and traverse back to arrow by joining at diagonal arrow
- hence printing the size and longest sub sequence along with its arrow table

Pseudo Code:

```
for i in range(1,m+1):
    for j in range(1,n+1):
        if x[i-1] == y[j-1]:
            c[i][j] = c[i-1][j-1] + 1
            b[i][j] = "↖"
        else:
            if c[i-1][j] >= c[i][j-1]:
                c[i][j] = c[i-1][j]
                b[i][j] = "↑"
            else:
                c[i][j] = c[i][j-1]
                b[i][j] = "←"
```

Code:

```
p = "TGAALUXRALVF"
q = "AGMAUURAXVT"
def LCS(x,y):
    m = len(x)
    n = len(y)
    b = [["" for j in range(n+1)] for i in range(m+1)]
    c = [[0 for j in range(n+1)] for i in range(m+1)]

    for i in range(1,m+1):
        for j in range(1,n+1):
            if x[i-1] == y[j-1]:
                c[i][j] = c[i-1][j-1] + 1
                b[i][j] = "↖"
            else:
                if c[i-1][j] >= c[i][j-1]:
                    c[i][j] = c[i-1][j]
                    b[i][j] = "↑"
                else:
                    c[i][j] = c[i][j-1]
                    b[i][j] = "←"

    index = c[m][n]
    LCS = [""] * (index+1)
    LCS[index] = ""

    i = m
    j = n
    while i > 0 and j > 0:
        if b[i][j] == "↖":
            LCS[index-1] = x[i-1]
```

```
            i -= 1
            j -= 1
            index -= 1
        elif b[i][j] == "↑":
            i -= 1
        else:
            j -= 1

    return ("".join(LCS), c[m][n], b)
lcs_string, lcs_length, arrow_matrix = LCS(p,q)
print("Longest Common Subsequence:", lcs_string)
print("Length of LCS:", lcs_length)
print("Arrows matrix:")
for row in arrow_matrix:
    print(row)
```

Output:

```
Longest Common Subsequence: GAURAV
Length of LCS: 6
Arrows matrix:
['', '', '', '', '', '', '', '', '', '', '', '']
['', '↑', '↑', '↑', '↑', '↑', '↑', '↑', '↑', '↑', '↑', '↖']
['', '↑', '↖', '←', '←', '←', '←', '←', '←', '←', '←', '↑']
['', '↖', '↑', '↑', '↖', '←', '←', '←', '↖', '←', '←', '←']
['', '↖', '↑', '↑', '↖', '↑', '↑', '↑', '↖', '←', '←', '←']
['', '↑', '↑', '↑', '↑', '↑', '↑', '↑', '↑', '↑', '↑', '↑']
['', '↑', '↑', '↑', '↑', '↖', '↖', '←', '↑', '↑', '↑', '↑']
['', '↑', '↑', '↑', '↑', '↑', '↑', '↑', '↑', '↖', '←', '←']
['', '↑', '↑', '↑', '↑', '↑', '↑', '↖', '←', '↑', '↑', '↑']
['', '↖', '↑', '↑', '↖', '↑', '↑', '↑', '↖', '←', '←', '←']
['', '↑', '↑', '↑', '↑', '↑', '↑', '↑', '↑', '↑', '↑', '↑']
['', '↑', '↑', '↑', '↑', '↑', '↑', '↑', '↑', '↑', '↖', '←']
['', '↑', '↑', '↑', '↑', '↑', '↑', '↑', '↑', '↑', '↑', '↑']
```

Q12. Given an array A[1 .. n] of integers, compute the length of a longest increasing subsequence. A sequence B[1 .. l] is increasing if B[i] > B[i − 1] for every index i ≥ 2. For example, given the array

⟨3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5, 8, 9, 7, 9, 3, 2, 3, 8, 4, 6, 2, 7⟩

Program Analysis:

- First apply the **Brute force Technique**:
  - By balancing two pointers previous and current goes through whole array and compare the current value with previous one if it greater them all increase the previous pointer to current and do again on rest of elements using recursion.
  - Time and space complexity will be
    - O(n ^2)

- Second apply the **Memoization technique Dp**:
  - It stores the value to reduce calculation and provide value at run to again compute the same value
    Time and space complexity will be
    - O (n ^2)

- And lastly apply the **Recursive binary algorithm**:
  - It starts with first and continue to make LIS by replacing the correct value to it and return the size at last

  For example:
  - [3,7,5,8,9,1] →[3,7]→[5,7,8,9]→[1,5,7,8,9]
  - Time and space complexity will be
    - O (n log n) → n for full iteration log n for binary
  - Space will be O(n) storing subsequence

Pseudo Code:

```
    for i in range (1, n):
       if p[i] > ans[-1]:
           ans.append(p[i])
       else:
           index = bisect_left(ans, p[i]) #using binary search find
correct path of element in list
           ans[index] = p[i]

   return len(ans)
```

Code:

```
p= [3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5, 8, 9, 7, 9, 3, 2, 3, 8, 4, 6, 2, 7]
```

Brute force Technique:

```
def LIS(p,size,curr,prev):
    if(curr==size):
        return 0

    take =0
    if(prev == -1 or p[curr] > p[prev]):
        take=1+LIS(p,size,curr+1,curr)

    notTake=0 + LIS(p,size,curr+1,prev)
    return max(take,notTake)
size=len(p)
LIS(p,size,0,-1)
```

In [16]:
```
def LIS(p,size,curr,prev):

    if(curr==size):
        return 0

    take =0
    if(prev == -1 or p[curr] > p[prev]):
        take=1+LIS(p,size,curr+1,curr)

    notTake=0 + LIS(p,size,curr+1,prev)
    return max(take,notTake)
```

In [17]:
```
size=len(p)
LIS(p,size,0,-1)
```

Out[17]: 6

Memoization technique Dp:

```
def LISMem(p,size,curr,prev,dp):

    if(curr==size):
        return 0
    if(dp[curr][prev+1]!= -1):
        return dp[curr][prev+1]

    take =0
    if(prev == -1 or p[curr] > p[prev]):
        take=1+LISMem(p,size,curr+1,curr,dp)

    notTake=0 + LISMem(p,size,curr+1,prev,dp)

    dp[curr][prev+1] = max(take,notTake)
    return dp
```

In [36]:
```
def LISMem(p,size,curr,prev,dp):

    if(curr==size):
        return 0
    if(dp[curr][prev+1]!= -1):
        return dp[curr][prev+1]

    take =0
    if(prev == -1 or p[curr] > p[prev]):
        take=1+LISMem(p,size,curr+1,curr,dp)

    notTake=0 + LISMem(p,size,curr+1,prev,dp)

    dp[curr][prev+1] = max(take,notTake)
    return dp
```

```
6
```

Recursive binary algorithm:

```
from bisect import bisect_left


def LISoptimal(n,p):
    if n == 0:
        return 0


    ans = [p[0]]


    for i in range(1, n):
        if p[i] > ans[-1]:
            ans.append(p[i])
        else:
            index = bisect_left(ans, p[i]) #using binary search
find correct path of element in list
            ans[index] = p[i]


    return len(ans)


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


print (LISoptimal(len(p), p))
```

In [37]:
```
from bisect import bisect_left

def LISoptimal(n,p):
    if n == 0:
        return 0

    ans = [p[0]]

    for i in range(1, n):
        if p[i] > ans[-1]:
            ans.append(p[i])
        else:
            index = bisect_left(ans, p[i]) #using binary search find correct path of element in list
            ans[index] = p[i]

    return len(ans)

p=[3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5, 8, 9, 7, 9, 3, 2, 3, 8, 4, 6, 2, 7]
print(LISoptimal(len(p), p))
```

```
6
```

Q13. Given an array A[1 .. n] of integers, compute the length of a longest alternating subsequence. A sequence B[1 .. l] is alternating if B[i] < B[i − 1] for every even index i ≥ 2, and B[i] > B[i − 1] for every odd index i ≥ 3. For example, given the array

⟨3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5, 8, 9, 7, 9, 3, 2, 3, 8, 4, 6, 2, 7⟩

your algorithm should return the integer 17, because 3, 1, 4, 1, 5, 2, 6, 5, 8, 7, 9, 3, 8, 4, 6, 2, 7 is a longest alternating subsequence (one of many).

Program Analysis:

Longest Alternating Subsequence is

X1 < X2 > X3 < X4 > X5 < .... xn or
X1 > X2 < X3 > X4 < X5 > .... Xn

Taking an 2 D array (dp) storing ...

dp[i][0] *= Length of the longest alternating subsequence ending at index i and last element is greater than its previous element*

dp[i][1] *= Length of the longest alternating subsequence ending at index i and last element is smaller than its previous element*

Recursive Formulation:

dp [i][0] = max (dp[i][0], dp[j][1] + 1);
for all j < i and A[j] < A[i]

dp [i][1] = max (dp [i][1], dp [j][0] + 1);
for all j < i and A[j] > A[i]

Time Complexity: $O(N^2)$
Auxiliary Space: O(N), since N extra space has been taken

Optimized code :

- Declare two integers inc and dec equal to one
- Run a loop for i [1, N-1]
  - If arr[i] is greater than the previous element then set inc equal to dec + 1
  - Else if arr[i] is smaller than the previous element then set dec equal to inc + 1
- Return maximum of inc and dec

Time Complexity: O(N)
Auxiliary Space: O(1)

Pseudo Code:

```
if (arr[j] < arr[i] and dp[i][0] < dp[j][1] + 1):
    dp[i][0] = dp[j][1] + 1
```

```
        # If arr[i] is smaller, then
        if(arr[j] > arr[i] and dp[i][1] < dp[j][0] + 1):
            dp[i][1] = dp[j][0] + 1
```

Optimized:

```
        if (arr[i] > arr[i-1]):
            inc = dec + 1
        elif (arr[i] < arr[i-1]):
            dec = inc + 1
```

Code:

```
def Max(a, b):
    if a > b:
        return a
    else:
        return b
def DAS(arr, n):
#     dp[i][0] = last element is greater than its previous element
#     dp[i][1] = last element is smaller than its previous element
    dp = [[0 for i in range(2)]
           for j in range(n)]
    for i in range(n):
        dp[i][0], dp[i][1] = 1, 1

    ans = 1

    for i in range(1, n):
        for j in range(0, i):
#             print(dp)
            # If arr[i] is greater, then
            if (arr[j] < arr[i] and dp[i][0] < dp[j][1] + 1):
                dp[i][0] = dp[j][1] + 1
```

```
            # If arr[i] is smaller, then
             if(arr[j] > arr[i] and dp[i][1] < dp[j][0] + 1):
                dp[i][1] = dp[j][0] + 1

        if (ans < max(dp[i][0], dp[i][1])):
            ans = max(dp[i][0], dp[i][1])

    return ans
```

Output:

```
In [19]: arr = [3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5, 8, 9, 7, 9, 3, 2, 3, 8, 4, 6, 2, 7]
         n = len(arr)

         print("Length of Longest alternating subsequence is", DAS(arr, n))

         Length of Longest alternating subsequence is 17
```

Optimized Code :

```
def LAS(arr, n):

    inc = 1
    dec = 1

    for i in range(1, n):
        if (arr[i] > arr[i-1]):
            inc = dec + 1

        elif (arr[i] < arr[i-1]):
            dec = inc + 1

    return max(inc, dec)
```

```
In [24]: print(LAS(arr, n))

         17
```

Q14. Given an array A[1 .. n], compute the length of a longest palindrome subsequence of A. Recall that a sequence B[1 .. l] is a palindrome if B[i] = B[l− i + 1] for every index i.

Program Analysis:

The LCS represents the longest sequence of characters that appear in the same order in both the original string and its reverse. Since the reverse of a string is essentially the original string read backward, any common sequence found between the two must be symmetrical around the center of the string. This symmetrical property is a characteristic of palindromes, which are sequences that read the same forward and backward.

if the longest common subsequence (LCS) between a string and its reverse is found, it will be a palindrome.

Time and space complexity is same as of LCS.

Presudo Code:

```
reverse_s = s[::-1]  # Reverse the input string
    return longest_common_subsequence(s, reverse_s)
```

Code:

```
#LCS code
def longest_common_subsequence(s1, s2):
    m = len(s1)
    n = len(s2)

    dp = [[0] * (n + 1) for _ in range(m + 1)]

    for i in range(1, m + 1):
        for j in range(1, n + 1):
            if s1[i - 1] == s2[j - 1]:
                dp[i][j] = dp[i - 1][j - 1] + 1
            else:
                dp[i][j] = max(dp[i - 1][j], dp[i][j - 1])



    lcs_length = dp[m][n]
    lcs = [''] * lcs_length
    i = m
```

```
    j = n
    while i > 0 and j > 0:
        if s1[i - 1] == s2[j - 1]:
            lcs[lcs_length - 1] = s1[i - 1]
            i -= 1
            j -= 1
            lcs_length -= 1
        elif dp[i - 1][j] > dp[i][j - 1]:
            i -= 1
        else:
            j -= 1

    return ''.join(lcs)



def longest_palindrome_subsequence(s):
    reverse_s = s[::-1]  # Reverse the input string
    return longest_common_subsequence(s,
reverse_s),len(longest_common_subsequence(s, reverse_s))
```

Output:

```
    return ''.join(lcs)


def longest_palindrome_subsequence(s):
    reverse_s = s[::-1]  # Reverse the input string
    return longest_common_subsequence(s, reverse_s),len(longest_common_subsequence(s, reverse_s))
```

```
: s = "Gaurav Kumar Chaurasiya"
lps = longest_palindrome_subsequence(s)
print(lps)
```

```
('aauahauaa', 9)
```

Q15. Given an array A[1 .. n] of integers, compute the length of a longest convex subsequence of A. A sequence B[1 .. l] is convex if $B[i] - B[i-1] > B[i-1] - B[i-2]$ for every index $i \geq 3$.-

Program Analysis:

Longest convex subsequence (acc to q):

$$B[i] - B[i-1] > B[i-1] - B[i-2]$$

Eg.:

A = [1, 3, 5, 2, 4, 6, 8]. One of the convex subsequences of A is [1, 3, 5, 8].

5-3 >= 3-1 and 8-5 >=5-3

Can be solve using dynamic programming, dp table, where dp[i] represents the length of the longest convex subsequence ending at index i

Table updating rule:

For each j from 1 to i-2, if A[i] - A[i-1] > A[i-1] - A[j], then dp[i] = max(dp[i], dp[j] + 1).

Time Complexity will be O(n^2) and space complexity will be O(n).

Pseudo Code:

```
for i in range(2, n):
    for j in range(1, i-1):
        if A[i] - A[i-1] > A[i-1] - A[j]:
            dp[i] = max(dp[i], dp[j] + 1)
```

Code:

```
def longest_convex_subsequence(A):
    n = len(A)
    dp = [1] * n

    for i in range(2, n):
        for j in range(1, i-1):
            if A[i] - A[i-1] > A[i-1] - A[j]:
                dp[i] = max(dp[i], dp[j] + 1)

    return max(dp)
A = [3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5, 8, 9, 7, 9, 3, 2, 3, 8, 4, 6, 2,
7]
```

```
result = longest_convex_subsequence(A)
print(result)
```

Output:

```
def longest_convex_subsequence(A):
    n = len(A)
    dp = [1] * n

    for i in range(2, n):
        for j in range(1, i-1):
            if A[i] - A[i-1] > A[i-1] - A[j]:
                dp[i] = max(dp[i], dp[j] + 1)

    return max(dp)
```

```
A = [3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5, 8, 9, 7, 9, 3, 2, 3, 8, 4, 6, 2, 7]
result = longest_convex_subsequence(A)
print(result)
```

```
9
```

Lab Sheet 5

Q16. Implement MCM algorithm for the given n matrix <M1xM2… Mn> where the size of the matrix is Mi=di-1 x di.

Program Analysis:

- MCM goal is to find the optimal parenthesization of the matrices that minimizes the total number of scalar multiplications required.
- The MCM algorithm fill the table M in a bottom-up manner. It iterate over the chain length l from 2 to n, representing the length of the subchains
- For each subchain length l, we consider all possible ways to split the subchain into two parts and calculate the minimum number of scalar multiplications required for each split.
    - M[i][j] = min(M[i][k] + M[k+1][j] + di-1 x dk x dj)
- Time Complexity : O(n^3)
- Space Complexity : O(n^2) (table store)
-

Pseudo Code:

```
for l in range(2, n+1):
    for i in range(1, n-l+2):
        j = i + l - 1
        m[i][j] = 9999
        for k in range(i, j):
            cost = m[i][k] + m[k+1][j] + d[i-1] * d[k] * d[j]
            if cost < m[i][j]:
                m[i][j] = cost
                s[i][j] = k
return m[1][n], s
```

Code:

```
def matrix_chain_multiplication(d):
```

```
    n = len(d)-1  # Number of matrices
    m = [[0] * (n+1) for _ in range(n+1)]
    s = [[0] * (n+1) for _ in range(n+1)]
    for l in range(2, n+1):
        for i in range(1, n-l+2):
            j = i + l - 1
            m[i][j] = 9999
            for k in range(i, j):
                cost = m[i][k] + m[k+1][j] + d[i-1] * d[k] * d[j]
                if cost < m[i][j]:
                    m[i][j] = cost
                    s[i][j] = k
    return m[1][n], s
def print_optimal_parentheses(s, i, j):
    if i == j:
        print("M", i, end=" ")
    else:
        print("(", end="")
        print_optimal_parentheses(s, i, s[i][j])
        print_optimal_parentheses(s, s[i][j]+1, j)
        print(")", end="")
matrix_sizes = [30,1,40,10,25]  # Sizes of the matrices

cost, splits = matrix_chain_multiplication(matrix_sizes)

print("Optimal cost:", cost)
print("Optimal parentheses:", end="")
print_optimal_parentheses(splits, 1, len(matrix_sizes)-1)
```

Output:

In [2]:
```
def matrix_chain_multiplication(d):
    n = len(d)-1  # Number of matrices
    m = [[0] * (n+1) for _ in range(n+1)]
    s = [[0] * (n+1) for _ in range(n+1)]

    for l in range(2, n+1):
        for i in range(1, n-l+2):
            j = i + l - 1
            m[i][j] = 9999
            for k in range(i, j):
                cost = m[i][k] + m[k+1][j] + d[i-1] * d[k] * d[j]
                if cost < m[i][j]:
                    m[i][j] = cost
                    s[i][j] = k
    return m[1][n], s

def print_optimal_parentheses(s, i, j):
    if i == j:
        print("M", i, end=" ")
    else:
        print("(", end="")
        print_optimal_parentheses(s, i, s[i][j])
        print_optimal_parentheses(s, s[i][j]+1, j)
        print(")", end="")
```

In [3]:
```
matrix_sizes = [30,1,40,10,25]  # Sizes of the matrices

cost, splits = matrix_chain_multiplication(matrix_sizes)

print("Optimal cost:", cost)
print("Optimal parentheses:", end="")
print_optimal_parentheses(splits, 1, len(matrix_sizes)-1)
```

```
Optimal cost: 1400
Optimal parentheses:(M 1 ((M 2 M 3 )M 4 ))
```

Q 17) Implement OBST for given n keys (K1,K2…..Km) whose pi and qi (dummy keys) are given.

Program Analysis:

- OBST objective is to minimize the total weighted search time, taking into account both the key search and the unsuccessful search costs.
- OBST follows bottom-up approach. For each subtree length l, we consider all possible roots within the subtree and calculate the minimum cost for each root.
    - C[i][j] = min(C[i][r-1] + C[r+1][j]) + ∑(P[k] + Q[k]) for k = i to j
- Time complexity : O(n^3) (n = no. of keys.)
- Space complexity : O(n ^2)

Pseudo code:

```
for l in range(1, n+1):
     for i in range(1, n-l+2):
         j = i + l - 1
         cost[i][j] = sys.maxsize
         for k in range(i, j+1):
             c = cost[i][k-1] + cost[k+1][j]
             if c < cost[i][j]:
                 cost[i][j] = c
                 root[i][j] = k
         cost[i][j] += sum(probabilities[i-1:j+1])

    return cost[1][n], root
```

Code :

```
def optimal_bst(keys, probabilities, dummy_probabilities):
    n = len(keys)
    cost = [[0] * (n+2) for _ in range(n+2)]
    root = [[0] * (n+2) for _ in range(n+2)]

    for i in range(1, n+2):
        cost[i][i-1] = dummy_probabilities[i-1]
        root[i][i-1] = i

    for l in range(1, n+1):
```

```
        for i in range(1, n-l+2):
            j = i + l - 1
            cost[i][j] = sys.maxsize
            for k in range(i, j+1):
                c = cost[i][k-1] + cost[k+1][j]
                if c < cost[i][j]:
                    cost[i][j] = c
                    root[i][j] = k
            cost[i][j] += sum(probabilities[i-1:j+1])

    return cost[1][n], root

def construct_obst(keys, root, i, j, level):
    if i > j:
        return None

    if i == j:
        return {"key": keys[i-1], "level": level, "left": None, "right":
None}

    k = root[i][j]
    node = {"key": keys[k-1], "level": level}

    node["left"] = construct_obst(keys, root, i, k-1, level+1)
    node["right"] = construct_obst(keys, root, k+1, j, level+1)

    return node
keys = ['K1', 'K2', 'K3', 'K4']  # Keys
probabilities = [0.1, 0.2, 0.3, 0.4]  # Probabilities of keys
dummy_probabilities = [0.2, 0.05, 0.1, 0.15, 0.2]  # Probabilities of
dummy keys

cost, root = optimal_bst(keys, probabilities, dummy_probabilities)
```

```
# Construct the OBST
obst = construct_obst(keys, root, 1, len(keys), 1)
print("Optimal cost:", cost)
def print_tree(node):
    if node["left"]:
        print_tree(node["left"])
    print("Level", node["level"], "-", node["key"])
    if node["right"]:
        print_tree(node["right"])

print_tree(obst)
```

OUTPUT :

In [5]:
```
keys = ['K1', 'K2', 'K3', 'K4']  # Keys
probabilities = [0.1, 0.2, 0.3, 0.4]  # Probabilities of keys
dummy_probabilities = [0.2, 0.05, 0.1, 0.15, 0.2]  # Probabilities of dummy keys

cost, root = optimal_bst(keys, probabilities, dummy_probabilities)
# Construct the OBST
obst = construct_obst(keys, root, 1, len(keys), 1)
print("Optimal cost:", cost)
def print_tree(node):
    if node["left"]:
        print_tree(node["left"])
    print("Level", node["level"], "-", node["key"])
    if node["right"]:
        print_tree(node["right"])

print_tree(obst)
```

```
Optimal cost: 3.0
Level 3 - K1
Level 2 - K2
Level 1 - K3
Level 2 - K4
```

Q 18) Implement 0/1 Knapsack problem using dynamic programming.

Program Analysis:

- 0/1 Knapsack algorithm fills the table K in a bottom-up manner. It iterate over the items i from 1 to n and the weights j from 0 to W.
- Include the item i: If the weight of the current item is less than or equal to the current weight j, we calculate the maximum value achievable by including the item i. It can be obtained by adding the value of the current item v[i] to the maximum value achieved by considering the remaining weight j-w[i] and items i-1, i.e., K[i-1][j-w[i]].
- Exclude the item i: We consider the maximum value achievable by excluding the current item, which is given by the value obtained for the previous item i-1 at the same weight j, i.e., K[i-1][j].
- Time Complexity: O(nW) (n is the number of items and W is the weight capacity.)
- Space Complexity: O(nW) since we use an (n+1) x (W+1)

Pseudo code:

```
for i in range(1, n+1):
        for j in range(M + 1):
            if W[i-1] <= j:
                if P[i-1] + V[i-1][j-W[i-1]] > V[i-1][j]:
                    V[i][j] = P[i-1] + V[i-1][j-W[i-1]]
                else:
                    V[i][j] = V[i-1][j]
            else:
                V[i][j] = V[i-1][j]
```

Code:

```
def knapsack(W, P, M):
    n = len(W)
    V = [[0] * (M + 1) for _ in range(n + 1)]

    for i in range(1, n+1):
        for j in range(M + 1):
            if W[i-1] <= j:
                if P[i-1] + V[i-1][j-W[i-1]] > V[i-1][j]:
                    V[i][j] = P[i-1] + V[i-1][j-W[i-1]]
```

```
                else:
                    V[i][j] = V[i-1][j]
            else:
                V[i][j] = V[i-1][j]

    selected_items = []
    i, w = n, M
    while i > 0 and w > 0:
        if V[i][w] != V[i - 1][w]:
            selected_items.append(i - 1)
            w -= W[i - 1]
        i -= 1

    selected_items.reverse()

    print(" Dp table : ")
    for i in V:
        print(i)
    return V[n][M], selected_items
weights = [2, 3, 4, 5]  # Weights of the items
profit = [4,8,9,11]  # profit of the items
capacity = 5  # Knapsack capacity

max_value, selected_items = knapsack(weights, profit, capacity)

print("Max Value:", max_value)
print("Selected Items:", selected_items)
```

OUTPUT:

```
    selected_items = []
    i, w = n, M
    while i > 0 and w > 0:
        if V[i][w] != V[i - 1][w]:
            selected_items.append(i - 1)
            w -= W[i - 1]
        i -= 1

    selected_items.reverse()

    print(" Dp table : ")
    for i in V:
        print(i)
    return V[n][M], selected_items
```

```
]: weights = [2, 3, 4, 5]  # Weights of the items
   profit = [4,8,9,11]  # profit of the items
   capacity = 5  # Knapsack capacity

   max_value, selected_items = knapsack(weights, profit, capacity)
   print("Max Value:", max_value)
   print("Selected Items:", selected_items)
```

```
 Dp table :
[0, 0, 0, 0, 0, 0]
[0, 0, 4, 4, 4, 4]
[0, 0, 4, 8, 8, 12]
[0, 0, 4, 8, 9, 12]
[0, 0, 4, 8, 9, 12]
Max Value: 12
Selected Items: [0, 1]
```

Lab Sheet 6

Q 19) Wap to Implement breadth first search algorithm for given graph G.

Program Analysis :

- BFS color initially all vertex white color (unvisited) It then selects a starting vertex and assigns it the gray color (discovered and added for exporation).
- a loop where it dequeues a vertex from the front of the queue and explores its adjacent vertices. For each unvisited adjacent vertex, the algorithm assigns it the gray color, adds it to the queue, and marks its predecessor. This process continues until all reachable vertices have been visited and explored.
- Time Complexity: O(V + E) (V is number of vertices E is the number of edges)
- Space Complexity : O(V) (maintain queue)

Pseudo Code :

```
for vertex in graph:
        visited[vertex] = "white"


    queue = deque([start])
    visited[start] = "gray"


    while queue:
        vertex = queue.popleft()
        print("Visited:", vertex)


        for neighbor in graph[vertex]:
            if visited[neighbor] == "white":
                queue.append(neighbor)
                visited[neighbor] = "gray"


        visited[vertex] = "black"
```

Code :

```
# from collections import deque
def bfs(graph, start):
    visited = {}
    for vertex in graph:
        visited[vertex] = "white"

    queue = deque([start])
    visited[start] = "gray"

    while queue:
        vertex = queue.popleft()
        print("Visited:", vertex)

        for neighbor in graph[vertex]:
            if visited[neighbor] == "white":
                queue.append(neighbor)
                visited[neighbor] = "gray"

        visited[vertex] = "black"

graph = {
    'A': ['B', 'C'],
    'B': ['A', 'D'],
    'C': ['A', 'D', 'E'],
    'D': ['B', 'C', 'E'],
    'E': ['C', 'D']
}

start_vertex = 'A'
print("BFS traversal:")
bfs(graph, start_vertex)
```

OUTPUT :

```
# from collections import deque
def bfs(graph, start):
    visited = {}
    for vertex in graph:
        visited[vertex] = "white"

    queue = deque([start])
    visited[start] = "gray"

    while queue:
        vertex = queue.popleft()
        print("Visited:", vertex)

        for neighbor in graph[vertex]:
            if visited[neighbor] == "white":
                queue.append(neighbor)
                visited[neighbor] = "gray"

        visited[vertex] = "black"

graph = {
    'A': ['B', 'C'],
    'B': ['A', 'D'],
    'C': ['A', 'D', 'E'],
    'D': ['B', 'C', 'E'],
    'E': ['C', 'D']
}

start_vertex = 'A'
print("BFS traversal:")
bfs(graph, start_vertex)
```

```
BFS traversal:
Visited: A
Visited: B
Visited: C
Visited: D
Visited: E
```

Q 20) Wap to Implement depth first search algorithm for given graph G.

PROGRAM ANALYSIS :

- DFS stores the time of discovery (start /end) this provides information about the order in which vertices are visited and allows for further analysis of the graph structure.

- Traverse all vertex of graph with coloring with one vertex to its deep .

- Time Complexity : O(V + E)
- Space Complexity : O(V)

PSEUDO CODE :

```
time += 1
    visited[vertex]["color"] = "gray"
    visited[vertex]["start_time"] = time
    print("Visited:", vertex)

    for neighbor in graph[vertex]:
        if visited[neighbor]["color"] == "white":
            time = dfs_visit(graph, neighbor, visited, time)

    time += 1
    visited[vertex]["color"] = "black"
    visited[vertex]["end_time"] = time

    return time
```

CODE :

```
def dfs(graph):
    visited = {}
    time = 0
    for vertex in graph:
        visited[vertex] = {"color": "white", "start_time": 0,
"end_time": 0}
```

```
    for vertex in graph:
        if visited[vertex]["color"] == "white":
            time = dfs_visit(graph, vertex, visited, time)

    print("DFS traversal:")
    for vertex in visited:
        print("Vertex:", vertex, "- Start Time:",
visited[vertex]["start_time"], "- End Time:",
visited[vertex]["end_time"])

def dfs_visit(graph, vertex, visited, time):
    time += 1
    visited[vertex]["color"] = "gray"
    visited[vertex]["start_time"] = time
    print("Visited:", vertex)

    for neighbor in graph[vertex]:
        if visited[neighbor]["color"] == "white":
            time = dfs_visit(graph, neighbor, visited, time)

    time += 1
    visited[vertex]["color"] = "black"
    visited[vertex]["end_time"] = time

    return time

graph = {
    'A': ['B', 'C'],
    'B': ['A', 'D'],
    'C': ['A', 'D', 'E'],
    'D': ['B', 'C', 'E'],
    'E': ['C', 'D']
}
```

```
dfs(graph)
```

OUTPUT :

```
    for vertex in graph:
        visited[vertex] = {"color": "white", "start_time": 0, "end_time": 0}
    for vertex in graph:
        if visited[vertex]["color"] == "white":
            time = dfs_visit(graph, vertex, visited, time)
    print("DFS traversal:")
    for vertex in visited:
        print("Vertex:", vertex, "- Start Time:", visited[vertex]["start_time"], "- End Time:", visited[verte

def dfs_visit(graph, vertex, visited, time):
    time += 1
    visited[vertex]["color"] = "gray"
    visited[vertex]["start_time"] = time
    print("Visited:", vertex)
    for neighbor in graph[vertex]:
        if visited[neighbor]["color"] == "white":
            time = dfs_visit(graph, neighbor, visited, time)

    time += 1
    visited[vertex]["color"] = "black"
    visited[vertex]["end_time"] = time
    return time

graph = {
    'A': ['B', 'C'],
    'B': ['A', 'D'],
    'C': ['A', 'D', 'E'],
    'D': ['B', 'C', 'E'],
    'E': ['C', 'D'] }
dfs(graph)
```

```
Visited: A
Visited: B
Visited: D
Visited: C
Visited: E
DFS traversal:
Vertex: A - Start Time: 1 - End Time: 10
Vertex: B - Start Time: 2 - End Time: 9
Vertex: C - Start Time: 4 - End Time: 7
Vertex: D - Start Time: 3 - End Time: 8
Vertex: E - Start Time: 5 - End Time: 6
```

Q 21) Wap to Implement topological sorting.

PROGRAM ANALYSIS :

- Topological sorting is an algorithm used to linearly order the vertices of a directed acyclic graph (DAG) in such a way that for every directed edge (u, v), vertex u comes before vertex v in the ordering.
- The topological sorting algorithm follows a depth-first search (DFS) approach. After visiting all the adjacent vertices of a vertex, it adds that vertex to the front of a result list. The result list thus represents the topological ordering of the vertices.
- Time Complexity: O(V + E)
- Space Complexity: The space complexity is O(V)

PSEUDO CODE :

```
visited = set()
    stack = []


    def dfs(vertex):
        visited.add(vertex)


        for neighbor in graph[vertex]:
            if neighbor not in visited:
                dfs(neighbor)


        stack.append(vertex)


    for vertex in graph:
        if vertex not in visited:
            dfs(vertex)


    sorted_vertices = stack[::-1]
    return sorted_vertices
```

CODE :

```
from collections import defaultdict
def topological_sort(graph):
    visited = set()
```

```
    stack = []

    def dfs(vertex):
        visited.add(vertex)

        for neighbor in graph[vertex]:
            if neighbor not in visited:
                dfs(neighbor)

        stack.append(vertex)

    for vertex in graph:
        if vertex not in visited:
            dfs(vertex)

    sorted_vertices = stack[::-1]
    return sorted_vertices

graph = {
    'A': ['B', 'C'],
    'B': ['D'],
    'C': ['D', 'E'],
    'D': [],
    'E': ['F'],
    'F': []
}
print("Topological Sorting:")
result = topological_sort(graph)
print(result)

# Create the result graph using the sorted vertices
result_graph = defaultdict(list)
for vertex in result:
```

```
    for neighbor in graph[vertex]:
        result_graph[vertex].append(neighbor)

print("\nResult Graph:")
for vertex in result_graph:
    print(vertex, "->", result_graph[vertex])
```

OUTPUT :

```
print("Topological Sorting:")
result = topological_sort(graph)
print(result)

# Create the result graph using the sorted vertices
result_graph = defaultdict(list)
for vertex in result:
    for neighbor in graph[vertex]:
        result_graph[vertex].append(neighbor)

print("\nResult Graph:")
for vertex in result_graph:
    print(vertex, "->", result_graph[vertex])
```

```
Topological Sorting:
['A', 'C', 'E', 'F', 'B', 'D']

Result Graph:
A -> ['B', 'C']
C -> ['D', 'E']
E -> ['F']
B -> ['D']
```

Q 22) Wap to find the strongly connected components in a Graph.

PROGRAM ANALYSIS :

Strongly Connected Components (SCCs) are subsets of vertices in a directed graph where every vertex in the subset is reachable from every other vertex.

1. Perform a DFS traversal of the graph, keeping track of the discovery time and low link value for each vertex. Initialize an empty stack and a visited set.
2. For each vertex v in the graph:
    - If v has not been visited, recursively call the DFS subroutine.
    - During the DFS, assign a unique discovery time to v and set its low link value to the discovery time.
3. While backtracking from the recursion stack, update the low link value for each vertex. The low link value of a vertex v is the minimum of its own discovery time and the low link values of its adjacent vertices that are still on the recursion stack.
4. If the low link value of a vertex v is equal to its discovery time, it means that v is the root of a strongly connected component. Pop vertices from the stack until v is reached, and add the popped vertices to the SCC.
5. Repeat steps 2-4 until all vertices have been visited.

Time Complexity: O(V + E)

Space Complexity: O(V)

PSEUDO CODE :

```
visited.add(vertex)
        stack.append(vertex)


        for neighbor in graph[vertex]:
            if neighbor not in visited:
                dfs_scc(neighbor)


        if stack[-1] == vertex:
            scc = []
            while stack and stack[-1] != vertex:
                scc.append(stack.pop())
            if stack:
```

```
                scc.append(stack.pop())
            sccs.append(scc)

    for vertex in graph:
        if vertex not in visited:
            dfs_scc(vertex)
```

CODE :

```
def find_scc(graph):
    visited = set()
    stack = []
    sccs = []
    timestamps = {}

    def dfs(vertex, time):
        nonlocal visited
        nonlocal stack
        nonlocal sccs
        nonlocal timestamps

        visited.add(vertex)
        stack.append(vertex)
        timestamps[vertex] = {"start": time, "end": None}

        for neighbor in graph[vertex]:
            if neighbor not in visited:
                time += 1
                time = dfs(neighbor, time)

        timestamps[vertex]["end"] = time
        time += 1

        return time
```

```
    # Perform DFS on each unvisited vertex
    time = 1
    for vertex in graph:
        if vertex not in visited:
            time = dfs(vertex, time)

    # Print the graph before finding SCCs
    print("Graph before finding SCCs:")
    for vertex in graph:
        for neighbor in graph[vertex]:
            print(vertex, "->", neighbor, "- Start Time:",
timestamps[vertex]["start"], "- End Time:", timestamps[vertex]["end"])



    visited = set()
    stack = []
    sccs = []

    def dfs_scc(vertex):
        nonlocal visited
        nonlocal stack
        nonlocal sccs

        visited.add(vertex)
        stack.append(vertex)

        for neighbor in graph[vertex]:
            if neighbor not in visited:
                dfs_scc(neighbor)

        if stack[-1] == vertex:
            scc = []
            while stack and stack[-1] != vertex:
```

```
                scc.append(stack.pop())
            if stack:
                scc.append(stack.pop())
            sccs.append(scc)

    for vertex in graph:
        if vertex not in visited:
            dfs_scc(vertex)

    # Print the graph after finding SCCs
    print("\nGraph after finding SCCs:")
    for vertex in graph:
        for neighbor in graph[vertex]:
            print(vertex, "->", neighbor)

    return sccs

graph = {
    'A': ['B'],
    'B': ['C', 'D'],
    'C': ['A'],
    'D': ['E'],
    'E': ['F'],
    'F': ['D']
}
print("Strongly Connected Components:")
result = find_scc(graph)
for scc in result:
    print(scc)
```

OUTPUT :

```
Strongly Connected Components:
Graph before finding SCCs:
A -> B - Start Time: 1 - End Time: 11
B -> C - Start Time: 2 - End Time: 10
B -> D - Start Time: 2 - End Time: 10
C -> A - Start Time: 3 - End Time: 3
D -> E - Start Time: 5 - End Time: 9
E -> F - Start Time: 6 - End Time: 8
F -> D - Start Time: 7 - End Time: 7

Graph after finding SCCs:
A -> B
B -> C
B -> D
C -> A
D -> E
E -> F
F -> D
['C']
['F']
['E']
['D']
['B']
['A']
```

Q 23) Wap to Implement Prim's algorithm for given graph G.

PROGRAM ANALYSIS :

Prim's algorithm is a greedy algorithm used to find the minimum spanning tree (MST) of a connected weighted graph. The algorithm starts with an arbitrary vertex and iteratively adds the minimum weight edge that connects the current MST to a new vertex, ensuring that no cycles are formed.

1. Initialize an empty MST and a set of visited vertices.
2. Select an arbitrary vertex to start the MST.
3. While there are unvisited vertices:
    - Find the minimum weight edge that connects a visited vertex to an unvisited vertex.
    - Add the minimum weight edge and the new vertex to the MST.
    - Mark the new vertex as visited.
4. Repeat step 3 until all vertices are visited.

Time Complexity: O(E log V)

Space Complexity: O(V)

PSEUDO CODE :

```
for neighbor, cost in graph[start_vertex]:
        heapq.heappush(heap, (cost, start_vertex, neighbor))


    while heap:
        cost, src, dest = heapq.heappop(heap)
        if dest not in visited:
            visited.add(dest)
            min_spanning_tree.append((src, dest, cost))
            for neighbor, n_cost in graph[dest]:
                if neighbor not in visited:
                    heapq.heappush(heap, (n_cost, dest, neighbor))


    return min_spanning_tree
```

CODE :

```
import heapq
```

```
def prim(graph):
    start_vertex = next(iter(graph))
    visited = set([start_vertex])
    min_spanning_tree = []
    heap = []

    # Add edges from the starting vertex to the priority queue
    for neighbor, cost in graph[start_vertex]:
        heapq.heappush(heap, (cost, start_vertex, neighbor))

    while heap:
        cost, src, dest = heapq.heappop(heap)
        if dest not in visited:
            visited.add(dest)
            min_spanning_tree.append((src, dest, cost))
            for neighbor, n_cost in graph[dest]:
                if neighbor not in visited:
                    heapq.heappush(heap, (n_cost, dest, neighbor))

    return min_spanning_tree


graph = {
    'A': [('B', 2), ('D', 1)],
    'B': [('A', 2), ('C', 3), ('D', 2)],
    'C': [('B', 3), ('D', 4)],
    'D': [('A', 1), ('B', 2), ('C', 4)]
}

print("Minimum Spanning Tree:")
result = prim(graph)
for edge in result:
```

```
    print(edge)
```

OUTPUT :

```
import heapq

def prim(graph):
    start_vertex = next(iter(graph))
    visited = set([start_vertex])
    min_spanning_tree = []
    heap = []

    # Add edges from the starting vertex to the priority queue
    for neighbor, cost in graph[start_vertex]:
        heapq.heappush(heap, (cost, start_vertex, neighbor))

    while heap:
        cost, src, dest = heapq.heappop(heap)
        if dest not in visited:
            visited.add(dest)
            min_spanning_tree.append((src, dest, cost))
            for neighbor, n_cost in graph[dest]:
                if neighbor not in visited:
                    heapq.heappush(heap, (n_cost, dest, neighbor))

    return min_spanning_tree
graph = {
    'A': [('B', 2), ('D', 1)],
    'B': [('A', 2), ('C', 3), ('D', 2)],
    'C': [('B', 3), ('D', 4)],
    'D': [('A', 1), ('B', 2), ('C', 4)]
}
print("Minimum Spanning Tree:")
result = prim(graph)
for edge in result:
    print(edge)
```

```
Minimum Spanning Tree:
('A', 'D', 1)
('A', 'B', 2)
('B', 'C', 3)
```

Q 24) Wap to Implement Kruskal's algorithm for given graph G.

PROGRAM ANALYSIS :

Kruskal's algorithm is a greedy algorithm works by considering the edges of the graph in ascending order of their weights and adding them to the MST if they do not create a cycle.

1. Sort all the edges of the graph G in non-decreasing order of their weights.
2. Initialize an empty MST.
3. Iterate through the sorted edges:
    - If adding the current edge to the MST does not create a cycle, add it to the MST.
    - To check for the presence of a cycle, use a disjoint-set data structure (such as Union-Find) to keep track of the connected components.
4. Repeat step 3 until all vertices are included in the MST or all edges have been considered.

Time Complexity: O(E log E)

Space Complexity: O(V + E)

PSEUDO CODE :

```
  for neighbor, cost in graph[start_vertex]:
        heapq.heappush(heap, (cost, start_vertex, neighbor))

    while heap:
        cost, src, dest = heapq.heappop(heap)
        if dest not in visited:
            visited.add(dest)
            min_spanning_tree.append((src, dest, cost))
            for neighbor, n_cost in graph[dest]:
                if neighbor not in visited:
                    heapq.heappush(heap, (n_cost, dest, neighbor))
    return min_spanning_tree
```

CODE :

```
import heapq
def prim(graph):
```

```
    start_vertex = next(iter(graph))
    visited = set([start_vertex])
    min_spanning_tree = []
    heap = []

    # Add edges from the starting vertex to the priority queue
    for neighbor, cost in graph[start_vertex]:
        heapq.heappush(heap, (cost, start_vertex, neighbor))

    while heap:
        cost, src, dest = heapq.heappop(heap)
        if dest not in visited:
            visited.add(dest)
            min_spanning_tree.append((src, dest, cost))
            for neighbor, n_cost in graph[dest]:
                if neighbor not in visited:
                    heapq.heappush(heap, (n_cost, dest, neighbor))
    return min_spanning_tree
graph = {
    'A': [('B', 2), ('D', 1)],
    'B': [('A', 2), ('C', 3), ('D', 2)],
    'C': [('B', 3), ('D', 4)],
    'D': [('A', 1), ('B', 2), ('C', 4)]
}
print("Minimum Spanning Tree:")
result = prim(graph)
for edge in result:
    print(edge)
```

OUTPUT :

```
import heapqd
def prim(graph):
    start_vertex = next(iter(graph))
    visited = set([start_vertex])
    min_spanning_tree = []
    heap = []

    # Add edges from the starting vertex to the priority queue
    for neighbor, cost in graph[start_vertex]:
        heapq.heappush(heap, (cost, start_vertex, neighbor))

    while heap:
        cost, src, dest = heapq.heappop(heap)
        if dest not in visited:
            visited.add(dest)
            min_spanning_tree.append((src, dest, cost))
            for neighbor, n_cost in graph[dest]:
                if neighbor not in visited:
                    heapq.heappush(heap, (n_cost, dest, neighbor))
    return min_spanning_tree
graph = {
    'A': [('B', 2), ('D', 1)],
    'B': [('A', 2), ('C', 3), ('D', 2)],
    'C': [('B', 3), ('D', 4)],
    'D': [('A', 1), ('B', 2), ('C', 4)]
}
print("Minimum Spanning Tree:")
result = prim(graph)
for edge in result:
    print(edge)
```

```
Minimum Spanning Tree:
('A', 'D', 1)
('A', 'B', 2)
('B', 'C', 3)
```

Q 25) Wap to Implement dijkstra algorithm to find single source shortest path.

PROGRAM ANALYSIS :

Dijkstra's algorithm is a popular graph traversal algorithm used to find the shortest path from a single source vertex to all other vertices in a weighted graph (non-negative edge weights)

1. Initialize the distance from the source vertex to all other vertices as infinity, except the distance from the source vertex to itself, which is set to 0.
2. Create a priority queue to store vertices based on their distance values. Initialize it with the source vertex and its distance value.
3. While the priority queue is not empty:
    - Extract the vertex with the minimum distance value from the priority queue.
    - For each neighbor of the extracted vertex:
        - Calculate the distance from the source vertex to the neighbor via the extracted vertex.
        - If the newly calculated distance is smaller than the current distance, update the distance value.
        - Add the neighbor to the priority queue.
4. Repeat step 3 until all vertices are processed or the destination vertex is reached (in case you are only interested in finding the shortest path to a specific destination).
5. The distances obtained after the algorithm terminates represent the shortest path from the source vertex to all other vertices. Additionally, you can track the parent nodes during the algorithm's execution to reconstruct the shortest path from the source to any specific vertex.

Time Complexity: O((V + E) log V)

Space Complexity: O(V)

PSEUDO CODE :

```
while heap:
        current_distance, current_vertex = heapq.heappop(heap)

        if current_distance > distances[current_vertex]:
            continue

        print("Visiting vertex:", current_vertex)
```

```
        print("Current distance:", current_distance)
        print("Updated distances:", distances)
        print()

        for neighbor, weight in graph[current_vertex]:
            distance = current_distance + weight
            if distance < distances[neighbor]:
                distances[neighbor] = distance
                heapq.heappush(heap, (distance, neighbor))

    return distances
```

CODE :

```
import heapq
def dijkstra(graph, source):
    distances = {vertex: float('inf') for vertex in graph}
    distances[source] = 0

    heap = [(0, source)]
    heapq.heapify(heap)

    while heap:
        current_distance, current_vertex = heapq.heappop(heap)

        if current_distance > distances[current_vertex]:
            continue

        print("Visiting vertex:", current_vertex)
        print("Current distance:", current_distance)
        print("Updated distances:", distances)
        print()
```

```
        for neighbor, weight in graph[current_vertex]:
            distance = current_distance + weight
            if distance < distances[neighbor]:
                distances[neighbor] = distance
                heapq.heappush(heap, (distance, neighbor))

    return distances

graph = {
    'A': [('B', 2), ('C', 4)],
    'B': [('C', 1), ('D', 7)],
    'C': [('D', 3)],
    'D': [('E', 2)],
    'E': []
}
source_vertex = 'A'
distances = dijkstra(graph, source_vertex)

print("Shortest distances from vertex", source_vertex)
for vertex, distance in distances.items():
    print(vertex, "->", distance)
```

OUTPUT :

```
graph = {
    'A': [('B', 2), ('C', 4)],
    'B': [('C', 1), ('D', 7)],
    'C': [('D', 3)],
    'D': [('E', 2)],
    'E': []  }
source_vertex = 'A'
distances = dijkstra(graph, source_vertex)

print("Shortest distances from vertex", source_vertex)
for vertex, distance in distances.items():
    print(vertex, "->", distance)
```

```
Visiting vertex: A
Current distance: 0
Updated distances: {'A': 0, 'B': inf, 'C': inf, 'D': inf, 'E': inf}

Visiting vertex: B
Current distance: 2
Updated distances: {'A': 0, 'B': 2, 'C': 4, 'D': inf, 'E': inf}

Visiting vertex: C
Current distance: 3
Updated distances: {'A': 0, 'B': 2, 'C': 3, 'D': 9, 'E': inf}

Visiting vertex: D
Current distance: 6
Updated distances: {'A': 0, 'B': 2, 'C': 3, 'D': 6, 'E': inf}

Visiting vertex: E
Current distance: 8
Updated distances: {'A': 0, 'B': 2, 'C': 3, 'D': 6, 'E': 8}

Shortest distances from vertex A
A -> 0
B -> 2
C -> 3
D -> 6
E -> 8
```