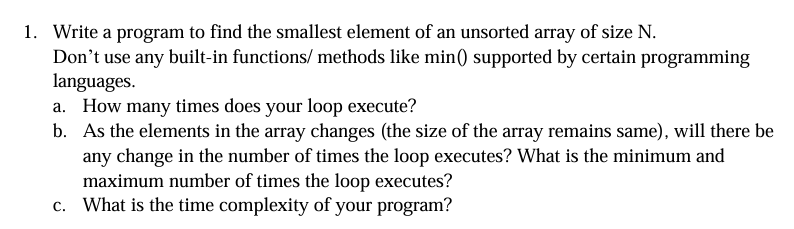
**22AIE212 - DESIGN & ANALYSIS OF ALGORITHMS**

**Lab Sheet 1**

**Basic iterative programs**

Name : Anuvind M P

Roll no : AM.EN.U4AIE22010



**CODE :**

def minArray(array):

    smallest = array[0]

    for i in array:

        if i<smallest:

            smallest = i

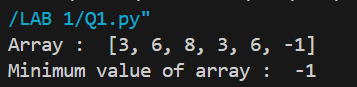
    return smallest

array = [3,6,8,3,6,-1]

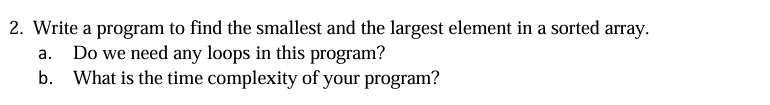
print("Array : ", array)

print("Minimum value of array : ", minArray(array))

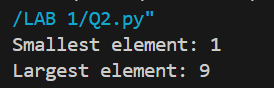
**OUTPUT :**



1. The loop executes ***N*** times. In this case the array has 6 elements, so the loop executes 6 times.
2. No, the number of times the loop executes does not change as the elements in the array change. The minimum and maximum number of times the loop executes are both equal to the size of the array, which is ***N***.
3. **Time complexity : O(N)**



**OUTPUT :**



**CODE :**

def smallest\_largest(array):

    return array[0], array[-1]

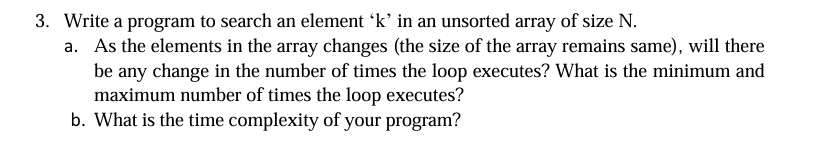
array = [1,2,3,4,5,6,7,8,9]

smallest, largest = smallest\_largest(array)

print("Smallest element:", smallest)

print("Largest element:", largest)

1. No we don’t need any loops for this program. We can simply return the first and last elements of the array using indexing. Since the array is sorted the first and last elements represents the smallest and largest elements respectively.
2. **Time complexity : O(N)**



**CODE :**

def linearSearch(array, k):

    for i in range(len(array)):

        if array[i] == k:

            return f"Element found at index {i}"

    return False

array = [5,8,9,4,7,3]

k = 7

print(linearSearch(array, k))

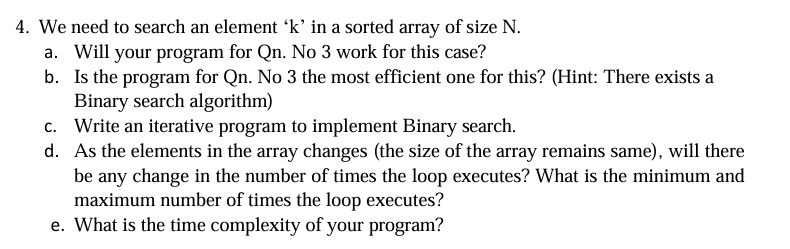
**OUTPUT :**



1. As the elements in the array change assuming the size of the array remains the same, there won't be any change in the number of times the loop executes.

The minimum number of times the loop executes is 1 (if the element 'k' is found at the first position), and the maximum number of times is N (if the element 'k' is not found in the array).

1. **Time Complexity : O(N)**



1. Program for Q3 will give correct results. But it won’t be the most efficient algorithm to use here because it won’t take advantage of the fact that the array is sorted.
2. No, the program for Q3 is not the most efficient one for searching an element in a sorted array. Binary search algorithm is more efficient for this purpose as it takes advantage of the fact that the array is sorted and reduces the search space by half with each iteration.

**CODE :**

def binarySearch(array, k):

    l = 0

    u = len(array)-1

    while l <= u :

        mid = (l+u)//2

        if array[mid] == k:

            return f"element found at index {mid}"

        elif array[mid] < k:

            l = mid + 1

        else:

            u = mid - 1

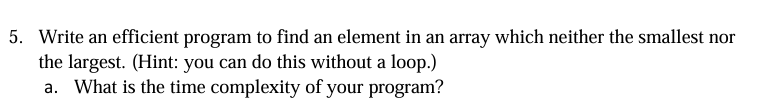
array = [1,2,3,4,5,6,7,8,9]

print(binarySearch(array, 6))

**OUTPUT :**



1. As the elements in the array change (assuming the size of the array remains the same), there won't be any change in the number of times the loop executes. The minimum number of times the loop executes is 1 (if the element 'k' is found at the middle position), and the maximum number of times is log2(N) (if the element 'k' is not found in the array)
2. **Time complexity : O(log N)**



**CODE :**

def mid(array):

    array.sort()

    if len(array) < 3:

        return None

    else:

        return array[1]

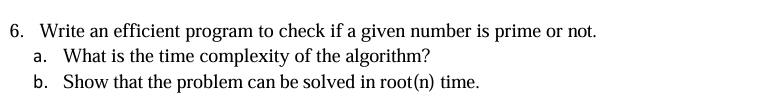
array = [3,6,9,8,4,2,5,4,7,8,9]

print("not big/not small :", mid(array))

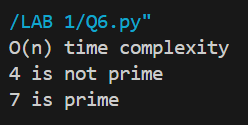
**OUTPUT :**



1. **Time complexity : O(log N)**



**OUTPUT :**



**CODE :**

def checkPrime(n):

    for i in range(2, n//2+1):

        if n % i == 0:

            return f"{n} is not prime"

    return f"{n} is prime"

print("O(n) time complexity")

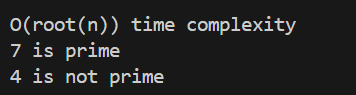
print(checkPrime(4))

print(checkPrime(7))

**Time Complexity : O(N)**

1. When we check if a number n is prime, we only need to check divisibility up to its square root. Because if n is not a prime number, it will have at least one factor smaller than or equal to its square root. So, we only need to check divisibility up to root(n) . This reduces the number of checks and makes the algorithm faster, with a time complexity of **O(root (N))**

**OUTPUT :**



**CODE :**

#root(n) time complexity problem

import math

def checkPrime(n):

    if n == 2:

        return f"{n} is prime"

    if n % 2 == 0:

        return f"{n} is not prime"

    sqrt\_n = int(math.sqrt(n)) + 1

    for i in range(3, sqrt\_n, 2):

        if n % i == 0:

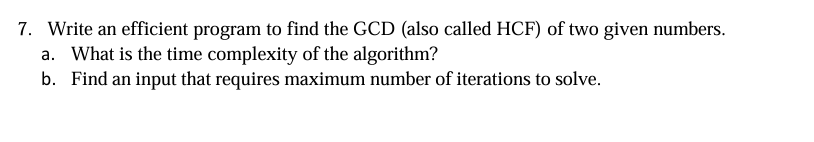
            return f"{n} is not prime"

    return f"{n} is prime"

print("\nO(root(n)) time complexity")

print(checkPrime(7))

print(checkPrime(4))



**CODE :**

def HCF(n, m):

    while m:

        n, m = m, n % m

    return n

print("HCF : ", HCF(4, 8))

**OUTPUT :**



1. **Time Complexity : O(log N)**

*N : lowest number among m and n*

1. Consecutive fibonacci numbers would require the maximum number of iterations since they are coprime

Eg: (144,233) : log(144) = 7 iterations

(6765,10946) : log(6765) = 13 iterations