22AIE212 - DESIGN & ANALYSIS OF ALGORITHMS

Lab Sheet 1

Basic iterative programs

Name : Anuvind M P

Roll no : AM.EN.U4AIE22010

---

1. Write a program to find the smallest element of an unsorted array of size N.
   Don't use any built-in functions/ methods like min() supported by certain programming languages.
   a. How many times does your loop execute?
   b. As the elements in the array changes (the size of the array remains same), will there be any change in the number of times the loop executes? What is the minimum and maximum number of times the loop executes?
   c. What is the time complexity of your program?

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 :

```
/LAB 1/Q1.py"
Array :  [3, 6, 8, 3, 6, -1]
Minimum value of array :  -1
```

a. The loop executes ***N*** times. In this case the array has 6 elements, so the loop executes 6 times.
b. 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***.

c. Time complexity : O(N)

---

2. Write a program to find the smallest and the largest element in a sorted array.
   a. Do we need any loops in this program?
   b. What is the time complexity of your program?

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)
```

OUTPUT :

```
/LAB 1/Q2.py"
Smallest element: 1
Largest element: 9
```

a. 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.

b. Time complexity : O(N)

---

3. Write a program to search an element 'k' in an unsorted array of size N.
   a. As the elements in the array changes (the size of the array remains same), will there be any change in the number of times the loop executes? What is the minimum and maximum number of times the loop executes?
   b. What is the time complexity of your program?

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 :

```
/LAB 1/Q3.py"
Element found at index 4
```

a. 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).

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

---

4. We need to search an element ‘k’ in a sorted array of size N.
   a. Will your program for Qn. No 3 work for this case?
   b. Is the program for Qn. No 3 the most efficient one for this? (Hint: There exists a Binary search algorithm)
   c. Write an iterative program to implement Binary search.
   d. As the elements in the array changes (the size of the array remains same), will there be any change in the number of times the loop executes? What is the minimum and maximum number of times the loop executes?
   e. What is the time complexity of your program?

a. 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.
b. 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.
c.

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 :

```
/LAB 1/Q4.py"
element found at index 5
```

d. 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)

e. Time complexity : O(log N)

5. Write an efficient program to find an element in an array which neither the smallest nor the largest. (Hint: you can do this without a loop.)
   a. What is the time complexity of your program?

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 :

```
/LAB 1/Q5.py"
not big/not small : 3
```

a. Time complexity : O(log N)

---

6. Write an efficient program to check if a given number is prime or not.
   a. What is the time complexity of the algorithm?
   b. Show that the problem can be solved in root(n) time.

a.

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))
```

OUTPUT :

```
/LAB 1/Q6.py"
O(n) time complexity
4 is not prime
7 is prime
```

Time Complexity : O(N)

b. 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))**

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))
```

OUTPUT :

```
O(root(n)) time complexity
7 is prime
4 is not prime
```

7. Write an efficient program to find the GCD (also called HCF) of two given numbers.
   a. What is the time complexity of the algorithm?
   b. Find an input that requires maximum number of iterations to solve.

CODE :

```
def HCF(n, m):
    while m:
        n, m = m, n % m
    return n

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

OUTPUT :

```
/LAB 1/Q7.py"
HCF :  4
```

a. **Time Complexity : O(log N)**
   N : lowest number among m and n
b. 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