Mohamed Ihab

7000399

Analysis Assignment 1 Report

Q1.

<u>Code: 1(a,c)</u>

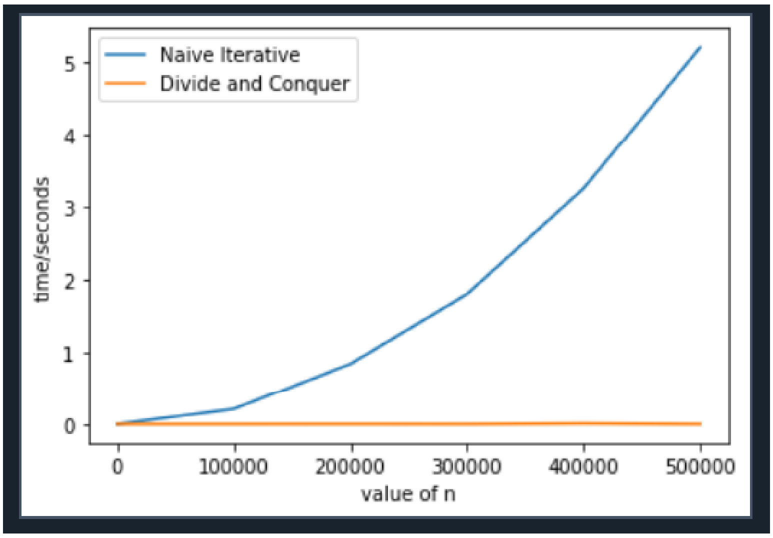
C:\Users\mzaka\Desktop\analysis python\Analysis Q.1.py

Analysis Q.1.py

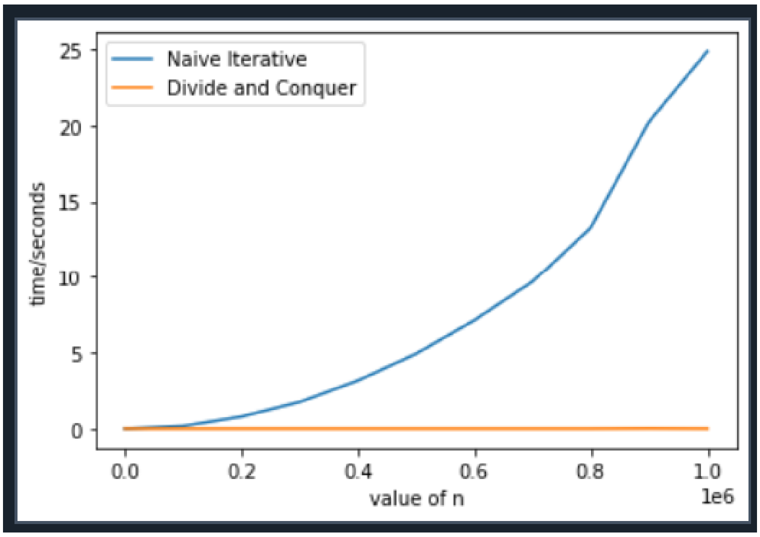
```python
import time
import matplotlib.pyplot as plt

def divideNconquer(a, n):
    if n==0:# trivial case or ending clause to recursion
        return 1
    elif (n%2)==0:# n is even
        half = divideNconquer(a, n//2)
        return half * half
    elif (n%2)!=0:# n is odd
        half = divideNconquer(a, (n-1)//2)
        return a * half * half

def naive(a, n):
    result = 1
    for i in range(n):
        result *= a
    return result


def timeTaken(computingFunction, a, n):
    start_time = time.time()
    computingFunction(a, n)
    end_time = time.time()
    return (end_time - start_time)

nArray = list(range(1, 500002, 100000))
naiveTimeArray = []
dNcTimeArray = []
for n in nArray:
    naiveTime = timeTaken(naive, 2, n)
    dNcTime = timeTaken(divideNconquer, 2, n)
    naiveTimeArray.append(naiveTime)
    dNcTimeArray.append(dNcTime)


plt.plot(nArray, naiveTimeArray, label='Naive Iterative')
plt.plot(nArray, dNcTimeArray, label='Divide and Conquer')
plt.xlabel('value of n')
plt.ylabel('time/seconds')
plt.legend()
plt.show()

```

Output:

Output when n Values from 1 to 5*10^5=500,000 used



Output when n Values from 1 to 10^6=1,000,000 used



Conclusion:

#1(b)

#the theoritically expected for the naive method would be O(n) and for the divide&conquer method would be o(log(n))

#1(d)

As for the results from my code and graphs it looks like the naïve iterative algorithm has a time complexity of O(n) while the complexity of the divide&conquer algorithm looks like an O(log(n))

So, in conclusion the results 1(c) are as what we expected theoretically from 1(b).

Q2.

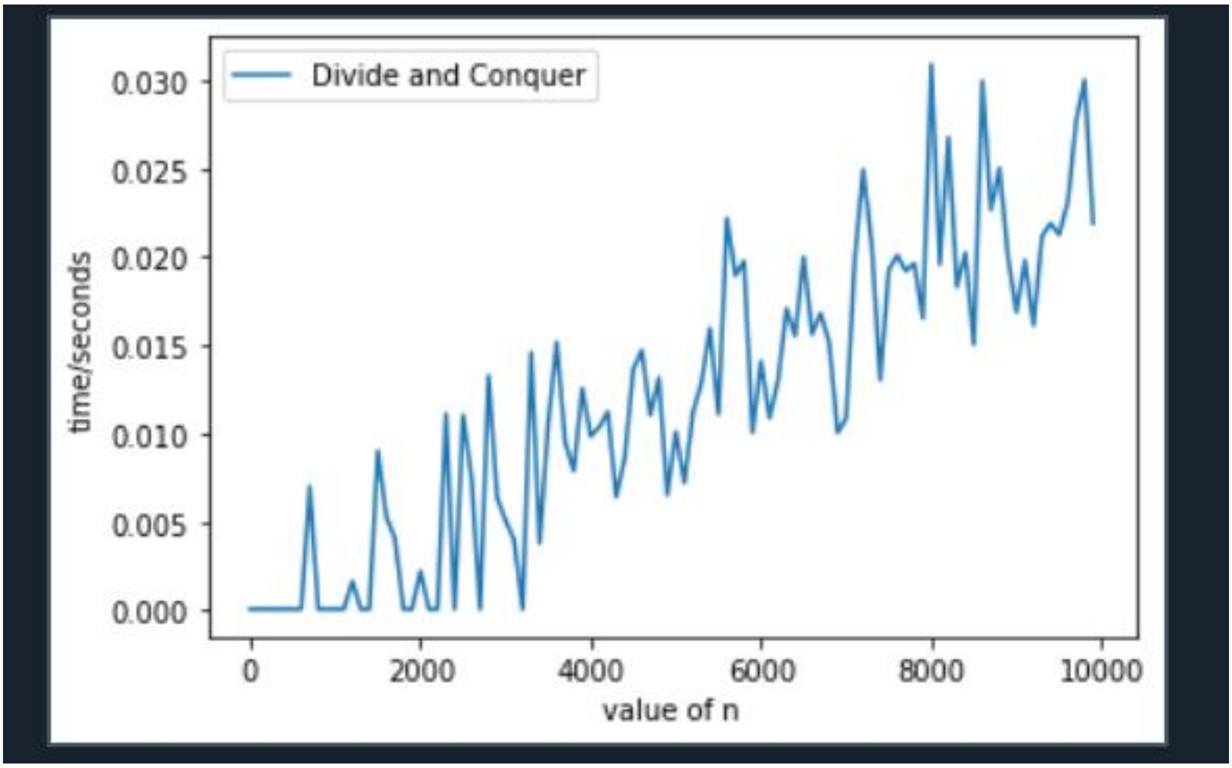
Code:

Analysis Q2.py

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

def mergeSort(array):

    if len(array) > 1:

        midIndex = len(array)//2
        leftHalf = array[:midIndex]
        rightHalf = array[midIndex:]

        mergeSort(leftHalf)
        mergeSort(rightHalf)

        leftIndex = 0 #left array's index
        rightIndex = 0 #right array's index
        i = 0 #index to guide the origin array

        while leftIndex < len(leftHalf) and rightIndex < len(rightHalf):
            if leftHalf[leftIndex] < rightHalf[rightIndex]:
                array[i] = leftHalf[leftIndex]
                leftIndex += 1
            else:
                array[i] = rightHalf[rightIndex]
                rightIndex += 1
            i += 1

        while leftIndex < len(leftHalf):
            array[i] = leftHalf[leftIndex]
            leftIndex += 1
            i += 1

        while rightIndex < len(rightHalf):
            array[i] = rightHalf[rightIndex]
            rightIndex += 1
            i += 1

def binarySearch(array, target):#assuming sorted array
    low = 0
    high = len(array) - 1
    pairsArray = []

    while low < high:
        sum = array[low] + array[high]
        if sum == target:
            pairsArray.append((array[low], array[high]))
            low += 1
            high -= 1
        elif sum < target:
            low += 1
        else:
            high -= 1

    return pairsArray

def getPairsEqualsTarget(inputArray, target):
    mergeSort(inputArray) #sort the array then pass it to binarySearch
    pairs = binarySearch(inputArray, target) #binarySearch gets possible pairs with sum==target
    return pairs

def timeTaken(computingFunction, a, n):
    start_time = time.time()
    computingFunction(a, n)
    end_time = time.time()
    return (end_time - start_time)



nArray = list(range(1, 10000, 100))
pairsTimeArray = []
for n in nArray:
    S = [random.randint(0, n) for i in range(n)]
    pairsTimeTaken = timeTaken(getPairsEqualsTarget, S, n)
    pairsTimeArray.append(pairsTimeTaken)


plt.plot(nArray, pairsTimeArray, label='pairs that sum==s')
plt.xlabel('value of n')
plt.ylabel('time/seconds')
plt.legend()
plt.show()
```

Output:

n Values between 1 and 10000



#2(b)

THEORETICALLY merge sort time complexity is o(nlog(n))

AND binary search time complexity is O(log(n))

BUT the modified binary search we created time

complexity is expected to be higher and possibly

equal to O(n) so I will consider it O(n) theoritically

therefore our program using both of these sequentially

would prove time complexity of O(nlog(n))+O(n) which

AND the complexity is the dominant therefore it is theoretically O(nlog(n))

2(c)

The answer corresponds to our expected time complexity which is overall O(nlog(n))