[CS202]-Algorithms I **2022-23**

Homework: 01

Name: Karan Sunil Kumbhar RollNo: 12140860 email: karansunilk@iitbhilai.ac.in

Collaborators Names:

Solution of problem 1.

a) Time complexity of an algorithm

Definition 1.1. Time complexity is a measure of how efficiently an algorithm uses computing resources, particularly time, to solve a problem. It describes how the running time of an algorithm grows as the size of the input increases. It is typically expressed in terms of the big O notation. The time complexity of an algorithm depends on the number of operations executed by the algorithm and their relative costs. For example, a linear search algorithm has a time complexity of O(n), where n is the size of the input array. This means that the time taken by the algorithm grows linearly with the size of the input array.

There are several factors that contribute to the time complexity of an algorithm, including the number of input elements, the number of steps required to solve the problem, and the execution time of each step. In order to analyze the time complexity of an algorithm, we typically count the number of basic operations (such as comparisons, assignments, and arithmetic operations) that are performed as a function of the input size.

b) Worst case running time of an algorithm

Definition 1.2. Worst case running time of an algorithm refers to the maximum amount of time taken by an algorithm to solve a problem for any input of size n. In other words, it is the running time of the algorithm for the input that causes the most number of operations to be executed.

For example, the worst case running time of a binary search algorithm is O(log n), where n is the size of the input array. This is because the algorithm performs a binary search on the input array and divides the array into two parts at each step, which reduces the search space by half. The worst case scenario is when the item to be searched is not present in the array, and the algorithm has to search the entire array.

c)Best case running time of an algorithm.

Definition 1.3. Best case running time of an algorithm refers to the minimum amount of time taken by an algorithm to solve a problem for any input of size n. In other words, it is the running time of the algorithm for the input that causes the least number of operations to be executed.

For example, the best case running time of a linear search algorithm is O(1), where the item to be searched is present at the first position of the array. In this case, the algorithm performs only one comparison operation and immediately returns the result.

Solution of problem 2.

Line	Code	cost	times
1	SAMPLE(n)	c1	1
2	s = 3	c2	1
3	**for** i = 1 to n **do**	c3	n+1
4	**for** j = i to n **do**	c4	$\sum_{i=1}^{n}(n-i+2)$
5	$s = s^3$	c5	$\sum_{i=1}^{n}(n-i+1)$
6	**end for**	c6	n
7	**end for**	c7	1
8	return s	c8	1

- Time complexity is -

  $T(n) = c_1 \times 1 + c_2 \times 1 + c_3 \times (n+1) + c_4 \times \sum_{i=1}^{n}(n-i+2) + c_5 \times \sum_{i=1}^{n}(n-i+1) + c_6 \times n + c_7 \times 1 + c_8 \times 1$

  $T(n) = (c_4 + c_5) \times \frac{n^2}{2} + (2c_3 + 3c_4 + c_5 + 2c_6) \times \frac{n}{2} + (c_1 + c_2 + c_3 + c_4 + c_7 + c_8)$

- Return value of S -
  So the fifth statement of code repeating for $\frac{n(n+1)}{2}$ (from above equations)
  so the final value of $s = 3^{3^{\frac{n(n+1)}{2}}}$

Solution of problem 3.

The increasing order of asymptotic (Big-O) growth is as follows:

- $4 \log n$
- $\sqrt{n} \log n$
- $2^{\log n} = 4n$
- $4n \log n$
- n^2
- 1.01^n
- $(3/2)^n$
- 5^n
- $n^{\log n}$
- $n!$

Solution of problem 4.

(a) The given statement is False.
The counter example for this statement is as follows:
Let's assume that

$$f(n_1) = 3n^2 + 2n + 1$$
$$f(n_2) = n^2 + n + 1$$
$$g(n) = n^2$$

Here, $f_1(n) = \theta(g(n))$

For constants c_1 and c_2 and n $\geq$$n_0$ where n_0 is a positive integer,

$$0 \leq c_1 g(n) \leq f_1(n) \leq c_2 g(n)$$

For, $c_1 n^2 \leq 3n^2 + 2n + 1 \leq c_2\, n^2$ to be true, c_1 = 3 and c_2 = 6 and n_0 = 1.

Here, $f_2(n) = \theta(g(n))$

For constants c_1 and c_2 and n $\geq$$n_0$ where n_0 is a positive integer,

$$0 \leq c_1 g(n) \leq f_2(n) \leq c_2 g(n)$$

For, $c_1 n^2 \leq n^2 + n + 1 \leq c_2\, n^2$ to be true, c_1 = 1 and c_2 = 3 and n_0 = 1.

Here, $f_1(n) - f_2(n) = 3n^2 + 2n + 1 - (n^2 + n + 1) = n^2 + n \neq O(1)$, since there isn't any constant c 0 and n $\geq$$n_0$ for which $0 \leq n \leq c$.

(b) The given statement is True.

Here, $f(n) = O(g(n))$
So, for constants c_1 and c_2 and n $\geq$$n_0$ where n_0 is a positive integer,

$$0 \leq f(n) \leq cg(n)$$

Similarly for equation $f(n) + g(n)$ when we add g(n) to the above equation, we get

$$0 \leq g(n) \leq f(n) + g(n) \leq c_1 g(n)$$

Here, $f(n) \leq cg(n)$ so,

$$0 \leq g(n) \leq f(n) + g(n) \leq (c+1)g(n)$$

When compared with $0 \leq c_1 g(n) \leq f_1(n) \leq c_2 g(n)$, we get c_1 = 1 and c_2 = (c+1).

Therefore, f(n) + g(n) = $\theta(g(n))$

(c) The given statement is True.

$$\text{Here, } f(n) = O(g(n))$$
$$\text{So, for constants } c_1 \text{ and } c_2 \text{ and n} \geq n_0 \text{ where } n_0 \text{ is a positive integer, } 0 \leq f(n) \leq cg(n)$$
$$0 \leq (\tfrac{1}{c})f(n) \leq g(n)$$
$$0 \leq c_1 \text{ f(n)} \leq g(n)$$

When compared with

$$cf(n) \leq g(n)$$

we get $c_1 = 1\frac{}{c}$
Therefore, g(n) = $\Omega(f(n))$.

(d) The given statement is True.
Here, we can say that

$$0 \leq b^n \leq f(n) = 1 + b + b^2 + ... + b^n \qquad (i)$$

f(n) is in Geometric Progression. So, from the summation of geometric progression

$$1 + b + b^2 + ... + b^n = \frac{b^{n+1} - 1}{b - 1}$$

When b is greater than 1,

$$\frac{b^{n+1} - 1}{b - 1} \leq \frac{b^{n+1}}{b - 1} = \frac{b}{b - 1} b^n \qquad (ii)$$

Comparing (i) and (ii), we get

$$b^n \leq f(n) = 1 + b + b^2 + ... + b^n \leq \frac{b}{b - 1} b^n$$

Comparing it with,

$$0 \leq c_1 g(n) \leq f_1(n) \leq c_2 g(n)$$

we get c_1 = 1 and c_2 = $\frac{b}{b-1}$
Therefore, f(n) = 1 + b + b^2 + ... + b^n is in $\theta(b^n)$.

Solution of problem 5.

a) h(n) = O(log(n))
Answer :- **False**
Justification :-
(as log function is increasing function in given domain)

$$\text{for } n \geq 0$$

$$n! \geq n$$
$$\log(n!) \geq \log(n)$$

but for O(log(n)) we need some positive constant n_0,such that for all $n \geq n_0$

$$0 \leq h(n) \leq c * log(n)$$

where c is positive constant, so we are not able to find such n_0. This proves that given statement is False

b) h(n) = (n log(n))
Answer :- **False**
Justification :-
we know that -

$$n \log n = \log n^n$$
$$n! \leq n^n$$

where n is positive integer ($n > 0$)
so by applying log on both side equality will not change(as log is increasing function in given domain)

$$\log(n!) \leq log(n^n)$$
$$\log(n!) \leq nlog(n)$$

but for $\Omega(log(n))$ we need some positive constant n_0 , such that for all $n \geq n_0$

$$0 \leq c_1 * n \log n \leq h(n) \leq c_2 * n \log n$$

So, from above condition we are not able to find such n_0,This contradict given statement and so given statement is false

c) h(n) = o(n^2)
Answer :- **True**
Justification :-
We know that, for any positive constant c

$$0 \leq h(n) < c * o(g(n))$$

where $n \geq n_0$ and n_0 is positive integer,

$$\implies c * n^2 > log(n!)$$

but from question b,

$$\log(n!) \leq n \log n$$
$$\implies c * n^2 > n \log n$$
$$\implies c * n > \log n$$

$\implies$ this is true for all positive constant c, for all $n_0 > 0$
so, this proves that above statement is True

Solution of problem 6. :

(a) T(n) = 16 T($\frac{n}{2}$) + n^3

$\Longrightarrow$ **By Master Theorem**

$$\text{a = 16, b = 2, f(n) } = n^3$$
$$n^{\log_b a} = n^{\log_2 16} = n^4$$
$$f(n) = O(n^{4-\epsilon}) \text{ for } \epsilon = 1$$

Therefore,

$$T(n) = \theta(n^4)$$

(b) T(n) = 4 T($\frac{n}{2}$) + n^2

$\Longrightarrow$ **By Master Theorem**

$$\text{a = 4, b = 2, f(n) } = n^2$$
$$n^{\log_b a} = n^{\log_2 4} = n^2$$
$$f(n) = \theta(n^2)$$

Therefore,

$$T(n) = \theta(n^2 \log n)$$

(c) T(n) = 3 T($\frac{n}{2}$) + n^2

$\Longrightarrow$ **By Master Theorem**

$$\text{a = 3, b = 2, f(n) } = n^2$$
$$n^{\log_b a} = n^{\log_2 3} = n^{1.584}$$
$$f(n) = \Omega(n^{1.5+\epsilon}) \text{ for } \epsilon = 0.5$$

Therefore,

$$T(n) = \theta(n^2)$$

(d) T(n) = 2 T($\frac{n}{4}$) + $n^{0.58}$

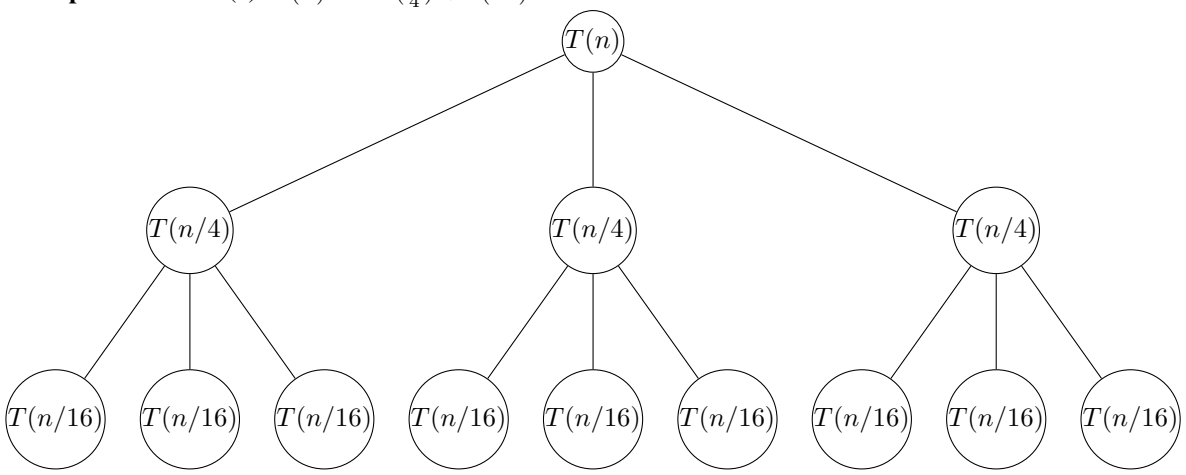
$\Longrightarrow$ **By Master Theorem**

$$\text{a = 2, b = 4, f(n) } = n^{0.58}$$
$$n^{\log_b a} = n^{\log_4 2} = n^{0.50}$$
$$f(n) = \Omega(n^{0.58+\epsilon}) \text{ for } \epsilon = 0.08$$

Therefore,

$$T(n) = \theta(n^{0.58})$$

Solution of problem 7. (a) $T(n) = 3T(\frac{n}{4}) + \theta(n^2)$



The recursion tree has $\log_4 n$ levels. Therefore, the total time of all levels is:

$= cn^2 + \frac{3}{16}cn^2 + \cdots + \frac{3}{16}^{\log_4(n-1)}cn^2 + \theta(n^{log_4 3})$

$= \sum_{i=0}^{\log_4(n-1)}(\frac{3}{16})^i + \theta(n^{\log_4 3})$

$< \sum_{i=0}^{\infty}(\frac{3}{16})^i + \theta(n^{\log_4 3})$

$= \frac{1}{1-\frac{3}{16}}cn^2 + \theta(n^{\log_4 3})$

$= \frac{16}{13}cn^2 + \theta(n^{\log_4 3})$

$= O(n^2)$

(b) T(n) = 8T(n/2) + $n^3 \log n$

On comparing above equation with aT(n/b) + f(n), we get
a = 8, b = 2 and f(n) = $n^3 \log n$
Now, $n^{\log_b(a)} = n^{\log_2(8)} = n^3$.
$f(n) = \Omega(n^3)$,but $f(n) \neq \Omega(n^{1+\epsilon})$ for any $\epsilon > 0$ that is f(n) is not polynomially larger.
Therefore, master theorem cannot be applied. An asymptotic upper bound for this recurrence is n^4.

Solution of problem 8.

The divide and conquer algorithm pseudo code is as follows:

```
function hIndex(C,n):
lower_Bound = 0
upper_Bound = n-1
hIndex = 0

while lower_Bound <= upper_Bound:
    midValue = (lower_Bound + upper_Bound) // 2

    if C[midValue] >= midValue+1:
        hIndex = midValue+1
        lower_Bound = midValue+1
    else:
        upper_Bound = midValue-1

return hIndex
```

Solution of problem 9.

(a) Brute force algorithm

The brute force algorithm for finding the pair of elements $A[i]$, $B[j_index]$ such that their absolute difference is the smallest over all such pairs would be to iterate over all possible pairs of elements in A and B, calculate their absolute difference, and keep track of the minimum absolute difference found so far. The algorithm can be described as follows:

1. Initialize `minDiff` to be a very large number.
2. For each element `a` in A:
   (a) For each element `b` in B:
      i. Calculate the absolute difference `diff = abs(a - b)`.
      ii. If `diff` is less than `minDiff`, update `minDiff` to be `diff` and remember the pair (a, b) that resulted in this difference.
3. Return the pair of elements (a, b) with minimum absolute difference.

The running time of the brute force algorithm is $O(n^2)$, since it involves iterating over all possible pairs of elements in A and B.

(b) Improved algorithm

To improve the running time of the algorithm, we can sort the arrays A and B in increasing order and then use a variant of the merge algorithm from merge sort to find the pair of elements with the smallest absolute difference. The algorithm can be described as follows:

1. Sort the arrays A and B in increasing order.
2. Initialize two pointers i_index and j_index to point to the first element of A and B respectively.

3. Initialize `minDiff` to be a very large number.
4. While i is less than the length of A and j_index is less than the length of B:
   (a) Calculate the absolute difference `diff = abs(A[i] - B[j_index])`.
   (b) If `diff` is less than `minDiff`, update `minDiff` to be `diff` and remember the pair $(A[i], B[j_index])$ that resulted in this difference.
   (c) If $A[i]$ is less than $B[j_index]$, increment i by 1.
   (d) Otherwise, increment j_index by 1.
5. Return the pair of elements $(A[i], B[j_index])$ with minimum absolute difference.

To prove the correctness of the algorithm, we can observe that if $A[i]$ is less than $B[j_index]$, then moving the pointer i to the next element in A can only increase the absolute difference. Similarly, if $B[j_index]$ is less than $A[i]$, then moving the pointer j_index to the next element in B can only increase the absolute difference. Therefore, the only way to decrease the absolute difference is to move the pointer corresponding to the smaller element. Since we start with the first elements in A and B and move the pointer corresponding to the smaller element, we are guaranteed to find the pair of elements with the smallest absolute difference

The running time of the algorithm is $O(n \log n)$, since it involves sorting the arrays A and B in increasing order, which takes $O(n \log n)$ time, and then iterating over the sorted arrays, which takes $O(n)$ time.

Solution of problem 10. :
Here is the python code for calculating time of excution of sorting alogrithm and there plot.

```
# importing required libraries.
import random
from time import time
import matplotlib.pyplot as plt
import math

# insertionSort
def insertionSsort(array):
    for i_index in range(1, len(array)):
        key = Array[i_index]
        j_index = i_index - 1
        while j_index >= 0 and array[j_index] > key:
            array[j_index + 1] = array[j_index]
            j_index -= 1
        array[j_index + 1] = key
    return array

# mergeSort
def mergeSort(array):
    if len(array) > 1:
        mid = len(array) // 2
        leftHalf = array[:mid]
        rightHalf = array[mid:]

        mergeSort(leftHalf)
        mergeSort(rightHalf)

        i_index = j_index = k_index = 0

        while i_index < len(leftHalf) and j_index < len(rightHalf):
            if leftHalf[i_index] < rightHalf[j_index]:
                array[k_index] = leftHalf[i_index]
                i_index += 1
            else:
                array[k_index] = rightHalf[j_index]
                j_index += 1
            k_index += 1

        while i_index < len(leftHalf):
            array[k_index] = leftHalf[i_index]
            i_index += 1
            k_index += 1

        while j_index < len(rightHalf):
            array[k_index] = rightHalf[j_index]
            j_index += 1
            k_index += 1
    return array
```

```
# quick sort
def quick_sort(array):
    if len(array) <= 1:
        return array

    pivot = array[len(array) // 2]
    left = [x for x in array if x < pivot]
    middle = [x for x in array if x == pivot]
    right = [x for x in array if x > pivot]

    return quick_sort(left) + middle + quick_sort(right)

# Heap sort
def heap_sort(array):
    # Build max heap
    n = len(array)
    for i_index in range(n // 2 - 1, -1, -1):
        heapify(array, n, i_index)

    # Extract elements from heap one by one
    for i_index in range(n - 1, 0, -1):
        array[i_index], array[0] = array[0], array[i_index] # swap
        heapify(array, i_index, 0)

    return array

def heapify(array, n, i_index):
    largest = i_index # Initialize largest as root
    list = 2 * i_index + 1 # left = 2*i_index + 1
    r = 2 * i_index + 2 # right = 2*i_index + 2

    # Check if left child of root exists and is greater than root
    if list < n and array[list] > array[largest]:
        largest = list

    # Check if right child of root exists and is greater than root
    if r < n and array[r] > array[largest]:
        largest = r

    # Change root, if needed
    if largest != i_index:
        array[i_index], array[largest] = array[largest], array[i_index
            ] # swap

        # Heapify the root.
        heapify(array, n, largest)

# Finding time required for executing.

list = [10, 20, 50, 100, 200, 500, 800, 1000, 5000, 10000]
rTime = []
```

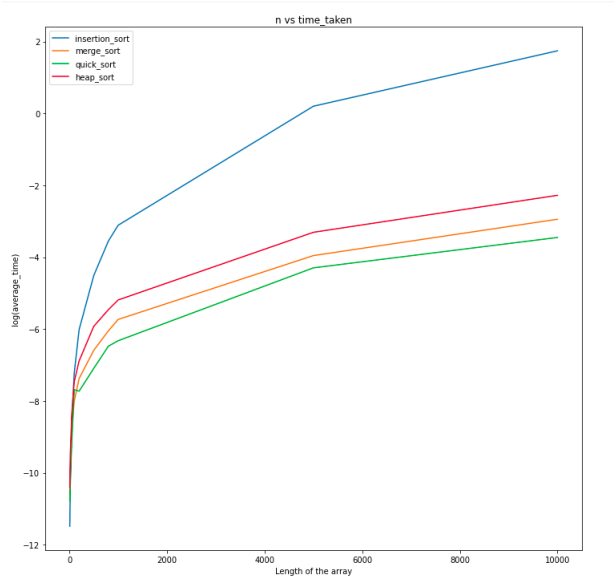
```
for i_index in list:
  n = list(range(i_index))
  sum = [0,0,0,0]
  for j_index in range(3):
    array = random.sample(n,k_index = i_index)
    t0 = time()
    a = insertion_sort(array)
    t1 = time()
    sum[0]+= t1-t0
    t0 = time()
    a = mergeSort(array)
    t1 = time()
    sum[1]+= t1-t0
    t0 = time()
    a = quick_sort(array)
    t1 = time()
    sum[2]+= t1-t0
    t0 = time()
    a = heap_sort(array)
    t1 = time()
    sum[3]+= t1-t0
  sum = [i_index/3 for i_index in sum]

  rTime.append(sum)

# ploting the graph n vs time of execution.
plt.figure(figsize =(12,12))
plt.plot( list, [math.log(i_index[0]) for i_index in rTime])
plt.plot(list,  [math.log(i_index[1]) for i_index in rTime] )
plt.plot(list,  [math.log(i_index[2]) for i_index in rTime] )
plt.plot(list,  [math.log(i_index[3]) for i_index in rTime] )
plt.legend(["insertion_sort","mergeSort","quick_sort","heap_sort"])
plt.title("n vs runTime")
plt.ylabel("log(average_time)")
plt.xlabel("Length of the array")
plt.show()
```

Here is the figure obtain by above code.

It should be noted that various factors such as hardware limitations, the specific implementation, and input data characteristics can affect the actual running times of the algorithms. Hence, the results obtained from executing the code provided may not be representative of all possible scenarios.

Nevertheless, the plot generated by the code provides a general overview of how the algorithms perform concerning their running times for different input sizes. As shown in the plot, Insertion Sort is faster than the other algorithms for smaller input sizes (up to 200). However, Insertion Sort's running time increases dramatically as the input size grows, while Merge Sort, Quick Sort, and Heap Sort remain relatively stable. For larger input sizes (e.g., 5000 and 10000), the running times of Merge Sort, Quick Sort, and Heap Sort are significantly faster than that of Insertion Sort.

In summary, the plot suggests that Merge Sort, Quick Sort, and Heap Sort are more efficient than Insertion Sort for



larger input sizes, which aligns with the expected theoretical complexity of these algorithms.