Assignment - I

Date:
Page No:

1. Explain the asymptotic notation with suitable examples.

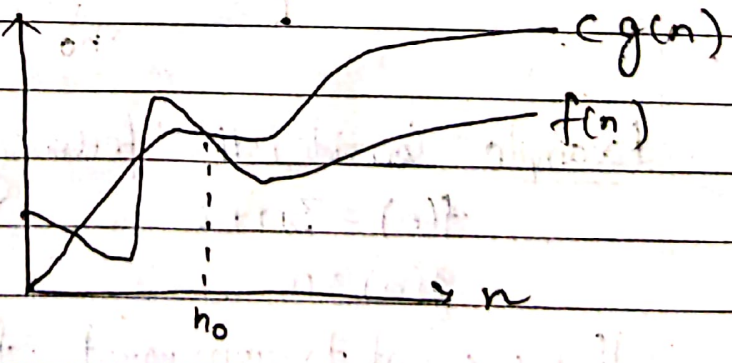
Asymptotic notation are mathematical tools to represent the time complexity of algorithm for asymptotic analysis. The main idea of asymptotic analysis is to have a measure of the efficiency of algorithm.

There are mainly three asymptotic notations:

i) Big-O-Notation (O-notation)
ii) Omega Notation (Ω-notation)
iii) Theta Notation (θ-notation)

Big O-Notation: Big O notation represents the upper bound of the running time of an ~~alog~~ algorithm. It gives the worst-case complexity.

$O(g(n)) = \{ f(n)$: there exist positive constant c and n_0 such that $0 \leq f(n) \leq cg(n)$ for all $n \geq n_0 \}$



Example: Consider the following f(n) and g(n)

$f(n) = 3n+2$

$g(n) = n$.

If we want to represent f(n) as O(g(n)) then it

must satisfy $f(n) <= cg(n)$ for all values of $c>0$ and $n_0 >= 1$

$f(n) <= cg(n)$

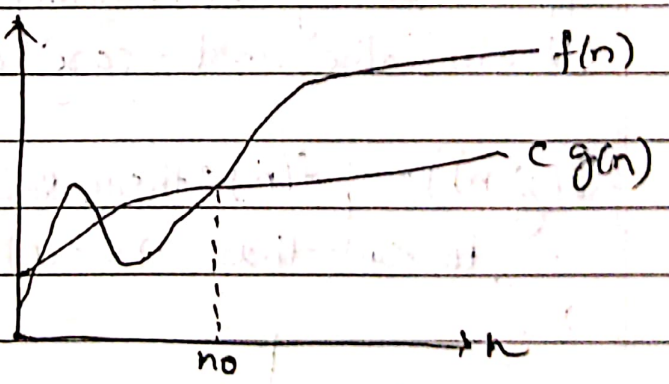
$\Rightarrow 3n+2 <= cn$

Above condition is always true for all $c=4$ and $n \geq 2$

So, $3n+2 = O(n)$

Omega Notation (Ω) : Omega notation represent the lower bound of the running time of an algorithm. It provides the best case complexity of an algorithm.

$\Omega(g(n)) = \{f(n)$: there exist positive constant c and n_0 such that $0 \leq cg(n) \leq f(n)$ for all $n \geq n_0\}$



Example: Consider the following f(n) and g(n)

$f(n) = 3n+2$

$g(n) = n$.

If we want to represent $f(n)$ as $\Omega(g(n))$ then it must satisfy $f(n) >= cg(n)$ for all values of $c>0$ and $n_0 >= 1$

$f(n) >= cg(n)$

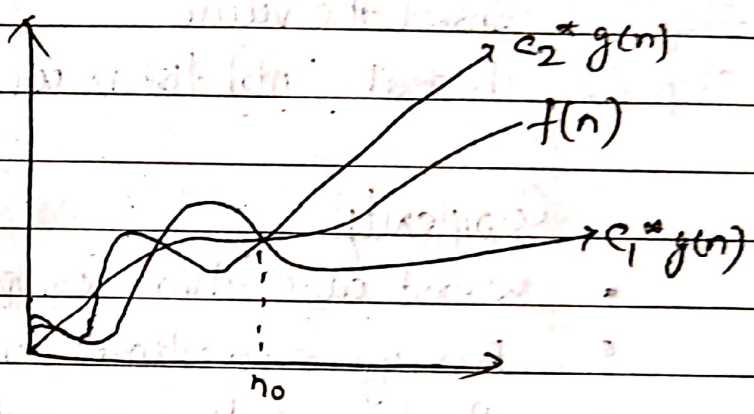
Above condition is always True for all values of $c=1$ and $n >= 1$

Date:
Page No:

By using Big Omega notation we can represent the time complexity as, $3n+2 = \Omega(n)$

Theta Notation (Θ): It represent the upper and the lower bound of the running time of an algorithm it is used to analysing the average-case complexity of an algorithm.

$\Theta(g(n)) = \{ f(n) :$ there exist positive constant c_1, c_2 and n_0 such that $0 \leq c_1 * g(n) \leq f(n) \leq c^2 * g(n)$ for all $n \geq n_0 \}$



Example: Consider the following $f(n)$ and $g(n)$...

$f(n) = 3n+2$

$g(n) = n$.

If we want to represent $f(n)$ as $\Theta g(n)$ then it must satisfy $c_1 g(n) <= f(n) <= c_2 g(n)$ for all value of $c_1 > 0$, $c_2 > 0$ and $n_0 >= 1$.

$c_1 g(n) <= f(n) <= c_2 g(n)$

$\Rightarrow c_1 n <= 3n+2 <= c_2 n$.

Above condition is always true of all values of $c_1 = 1$, $c_2 = 4$ and $n >= 2$.

$3n+2 = \Theta(n)$.

Date:
Page No:

2. Write down the algorithm for insertion Sort along with its complexity. Also list characteristic of Insertion sort.

Algorithm

Step 1. If it is the first element, it is already sorted. return 1
Step 2. pick next element
Step 3 Compare with all elements in the ~~sub~~-sorted sub-list
Step 4. Shift all the elements in the sorted sub-list that is greater than the value to be sorted
Step 5. Insert the value
Step 6. Repeat until list is sorted.

Complexity

- Worst case time complexity : $O(n^2)$
- Average case time complexity : $O(n^2)$
- Best case time complexity : $O(n)$

Characteristic of Insertion Sort

- It is efficient for smaller data set, but very inefficient for large list.
- Insertion sort is adaptive, that mean it reduce its total number of steps if given a partially sorted list, hence it increases its efficiency.
- Its space complexity is less. It requires a single additional memory space
- Overall time complexity of insertion sort is $O(n^2)$

Date:
Page No:

3. Define Hashing, Hash table, Hash Function, List different type of hash function. Explain any Hast function with example.

Hashing : Hashing is a technique of process of mapping keys, value into the hash table by using hash function.

Hash function : Hash function is a function that maps any big number of string to a smaller integer value.

Hash Table : Hash table is a data structure that store some information, and the information is basically two main components, i.e., key and value.

Types of Hash Functions = i) Mid Square hash function
ii) Division hash function
iii) Folding hash function.

Division has function

In this the hash function is dependent upon the remainder of a division.

forexample : If the record 52, 68, 99, 84 is to be placed in a hash table and let us take the table size is 10.

Date:
Page No:

Then, h(key) = record % tablesize

2 = 58 % 10

8 = 68 % 10

9 = 99 % 10

4 = 84 % 10

0	
1	
2	52
3	
4	84
5	
6	
7	
8	68
9	99

4. Find the address of integer type element A[2,1] in Row and column major order. Array if the base address is 2000. Array of [3x3].

W = Integer Size = 2

B = Base address = 2000

M = Numbers of row = 3

N = Number of column = 3

I = Row Subset of element = 2

J = colum subset of element = 1

LR = Lower row index = 0 1

LC = lower column index = 0 1

Row Major: Address of [2][1] = $B + W*((I-LR)*N + (J-LC))$

$= 2000 + 2((2-1)3 + (1-1))$

$= 2000 + 2x3$

$= 2006.$

Column Major: Address of [2][1] = $B + W*((J-LC)*M + (I-LR))$

$= 2000 + 2*((1-1)*3 + (2-1))$

$= 2000 + 2*(1)$

$= 2002.$

Date:
Page No:

5 Define Searching and its types. Differentiate between Linear and Binary Search.

- Searching is a technique of selecting specific data from a collection of data based on some condition.
- The process of finding an element from a list of elements is known as searching.

Types of Searching.

i) Linear Search
ii) Binary Search.

Linear Search : which simply check the values in sequence until the desired value is found.

Binary Search : which require a sorted input list, and checks for the value in the middle of the list repeatedly discarding the half of the list which contains values which are definitely either all large or smaller so than the desired value.

Linear Search	Binary Search
• The elements don't need to be arranged in sorted order	• the elements must be arranged in sorted order
• It can be implement on any data structure like array, linked list, etc.	• It can be implemented only on those data structure that have two way traversal

• It is based on the sequentially approach	• It is based on divide and conquer approach.
• prefferable for small sized data set.	• prefferable for large size data set.
• less efficient in case of large data.	• Most efficient in case of large data.
• Worst case scenario O(n)	• Worst case scenario $O(\log_2 n)$
• It can implemented on both a single or multidimentional array	• It can only be implemented on a multidimentional array

6. Write down the algorithm for Merge Sort. Implement merge sort on 95, 6, 46, 15, 57, 35, 46, 4, 0

Step 1 : Start
step 2 : declare array and left, right, mid variable.
Step 3 : perform merge function.
if left > right
return
mid = (left + right) / 2.
mergesort (array, left, mid)
mergesort (array, mid+1, right)
merge (array, left, mid, right)
Step 4 : stop.

```
def mergeSort(arr):
    if len(arr) > 1:
        mid = len(arr)//2
        L = arr[:mid]
        R = arr[mid:]
        mergeSort(L)
        mergeSort(R)
        i = j = k = 0
        while i < len(L) and j < len(R):
            if L[i] <= R[j]:
                arr[k] = L[i]
                i += 1
            else:
                arr[k] = R[j]
                j += 1
            k += 1
        while i < len(L):
            arr[k] = L[i]
            i += 1
            k += 1
        while j < len(R):
            arr[k] = R[j]
            j += 1
            k += 1
arr = [95, 6, 46, 15, 57, 35, 46, 4, 0]

mergeSort(arr)
print(arr)
```

output : [0, 4, 6, 15, 35, 46, 46, 57, 95]

Date:
Page No:

7. Write down the algorithm of Binary Search. Implement Binary Search on 12, 14, 20, 28, 43, 45, 72. Search given item 45 in the array.

Algorithm :

Step 1. Compare x with the middle element

Step 2. If x matches with the middle element, we return the mid index

Step 3. Else if x is greater than the mid element; then x can only lie in the right (greater) half subarray after the mid element. Then we apply the algorithm again for the right half.

Step 4. Else if x is smaller, the target x must lie in the left (lower) half. So we apply the algorithm for the left half.

```
def binarySearch(arr, l, r, x):
    if r >= l:
        mid = l + (r-l)//2
        if arr[mid] == x:
            return mid
        elif arr[mid] > x:
            return binarySearch(arr, l, mid-1, x)
        else:
            return binarySearch(arr, mid+1, r, x)
    else:
        return -1.
arr = [12, 14, 20, 28, 43, 45, 72]
x = 45
```

```
result = binarySearch(arr, 0, len(arr)-1, x)
if result != -1:
    print("Element is present at index %d" %result)
else:
    print("Element is not present in array")
```

8 Implement Bubble Sort on the given Array:
74, 56, 45, 36, 16 Show all passes and their steps.

```
def bubbleSort(arr):
    n = len(arr)
    for i in range(n):
        for j in range(0, n-i-1):
            if arr[j] > arr[j+1]:
                arr[j], arr[j+1] = arr[j+1], arr[j]
arr = [74, 56, 45, 36, 16]
bubbleSort(arr)
print("Sorted Array is: ")
for i in range(len(arr)):
    print("%d" %arr[i], end=" ")
```

Initial	74	56	45	36	16	Initial Unsorted Array
Step1	74	56	45	36	16	Compare 1st and 2nd (swap)
Step 2	56	74	45	36	16	Compare 2nd and 3rd (swap)
Step 3	56	45	74	36	16	Compare 3rd and 4th (swap)
Step 4	56	45	36	74	16	Compare 4th and 5th (swap)

Date:
Page No:

Step 5 | 56 | 45 | 36 | 16 | 74 | Repeat Step 1-5

Step 6 | 56 | 45 | 36 | 16 | 74 | Compare 1st and 2nd (swap)

Step 7 | 45 | 56 | 36 | 16 | 74 | Compare 2nd and 3rd (swap)

Step 8 | 45 | 36 | 56 | 16 | 74 | Compare 3rd and 4th (swap)

Step 9 | 45 | 36 | 16 | 54 | 74 | Compare 4th and 5th (Dont swap)

Step 10 | 45 | 36 | 16 | 54 | 74 | Repeat step 1-5

Step 11 | 45 | 36 | 16 | 54 | 74 | Compare 1st and 2nd (swap)

Step 11 | 36 | 45 | 16 | 54 | 74 | Compare 2nd and 3rd (swap)

Step 12 | 36 | 16 | 45 | 54 | 74 | Compare 3rd and 4th (Dont swap

Step 13 | 36 | 16 | 45 | 54 | 74 | Compare 4th and 5th (Dont Swap

Step 14 | 36 | 16 | 45 | 54 | 74 | Repeat step 1-5

Step 15 | 36 | 16 | 45 | 54 | 74 | Compare 1st and 2nd (swap)

Step 16 | 16 | 36 | 45 | 54 | 74 | Dont swap further.

| 16 | 36 | 45 | 54 | 74 | - Final sorted array.