Assignment 1 (26 Rohit Rajbhar)

Q1. What are asymptotic notations?

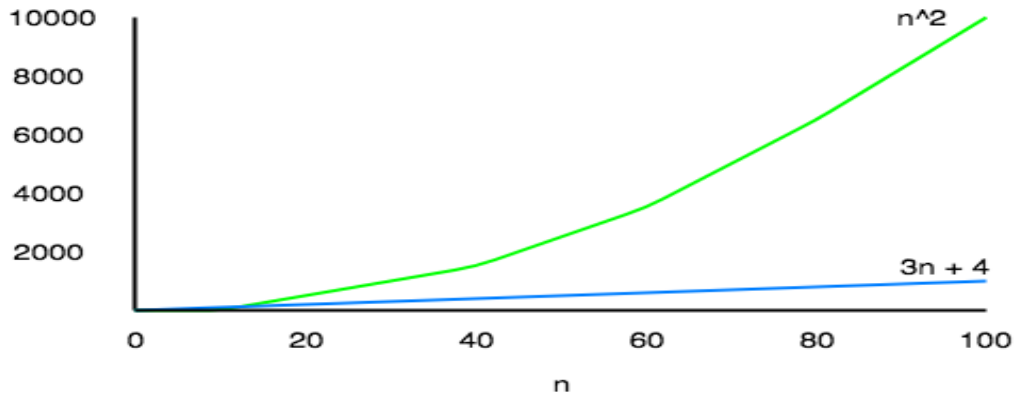
Ans :

- When it comes to analysing the complexity of any algorithm in terms of time and space, we can never provide an exact number to define the time required and the space required by the algorithm, instead we express it using some standard notations, also known as Asymptotic Notations.

- We use three types of asymptotic notations to represent the growth of any algorithm, as input increases:
  a) Big Theta (Θ):Big-Θ notation is like the average value or range within which the actual time of execution of the algorithm will be.
  b) Big Oh(O):It tells us that a certain function will never exceed a specified time for any value of input n.
  c) Big Omega (Ω):This always indicates the minimum time required for any algorithm for all input values, therefore the best case of any algorithm.

- When we analyse any algorithm, we generally get a formula to represent the amount of time required for execution or the time required by the computer to run the lines of code of the algorithm, number of memory accesses, number of comparisons, temporary variables occupying memory space etc. This formula often contains unimportant details that don't really tell us anything about the running time.

- Let us take an example, if some algorithm has a time complexity of T(n) = (n2 + 3n + 4), which is a quadratic equation. For large values of n, the 3n + 4 part will become insignificant compared to the n2 part.



For n = 1000, n2 will be 1000000 while 3n + 4 will be 3004.

Also, When we compare the execution times of two algorithms the constant coefficients of higher order terms are also neglected.

An algorithm that takes a time of 200n2 will be faster than some other algorithm that takes n3 time, for any value of n larger than 200. Since we're only interested in the asymptotic behavior of the growth of the function, the constant factor can be ignored too.

Q2. Describe the growth rate of common functions like 2^n, n^c, n^2, nlogn, n and *log log n*.

Ans:

- There is an order to the functions that we often see when we analyze algorithms using asymptotic notation. If a and b are constants and a < b, then a running time of $0(n^a)$ grows more slowly than a running time of 0(nb). For example, a running time of 0(n), which is 0(n1), grows more slowly than a running time of 0(n2). The exponents don't have to be integers, either. For example, a running time of 0(n2) grows more slowly than a running time of 0(n2vn), which is 0(n2.5).
- Logarithms grow more slowly than polynomials. That is, 0(lgn) grows more slowly than $0(n^a)$ for any positive constant a. But since the value of lg n increases as n increases, 0(lg n) grows faster than 0(1).
- Note that an exponential function an where a > 1, grows faster than any polynomial function nb where b is any constant.
-

Notation	Name	Example
O(2^n)	Exponential	Find all subsets
O(n^c)	Polynomial or Algebraic	Tree-adjoining grammar parsing; maximum matching for bipartite graphs; finding the determinant with LU decomposition
O(n^2)	Quadratic	Multiplying two n-digit numbers by a simple algorithm; simple sorting algorithms, such as bubble sort, selection sort and insertion sort; (worst case) bound on some usually faster sorting algorithms such as quicksort, Shellsort, and tree sort
O(n log n) = O(log n!)	Linearithmic, Loglinear, Performing a fast Fourier transform; Fastest	Performing a fast Fourier transform; Fastest possible comparison sort;

	possible comparison sort; heapsort and merge sort quasilinear, or "nlogn"	heapsort and merge sort
O(n)	Linear	Finding an item in an unsorted list or in an unsorted array; adding two n-bit integers by ripple carry
O(log logn)	Double Logarithmic	Number of comparisons spent finding an item using interpolation search in a sorted array of uniformly distributed values

- The statement f (n) = O(n!) is sometimes weakened to f (n) = O (n^n) to derive simpler formulas for asymptotic complexity. For any k > O and e > O, O(n^e (log n)^k) is a subset of O(n^c+e) for any e > O, so may be considered as a polynomial with some bigger order.

Q3. Explain the various asymptotic functions that are used to measure space and time complexity.

Ans: We use three types of asymptotic functions to represent the growth of any algorithm, as input increases:

1) Big Theta(Θ): Describes Average case scenario. Represents most realistic-time complexity of an algorithm. Theta notation encloses the function from above and below.
   Since it represents the upper and the lower bound of the running time of an algorithm, it is used for analyzing the average-case complexity of an algorithm.
   $1 < log(n) < sqrt(n) < n < n\,x\,log\,(n) < n2 < n3 < \ldots < n^n$

2) Big Oh(O):Describes Worst case Scenario. Represents upper bound running time complexity of an algorithm.
   The various upper bound of time complexity and their hierarchy in Mathematical terms is as follows : While finding O(n) we have to find out the closest one.

$1 < log(n) < sqrt(n) < n < n\,x\,log\,(n) < n2 < n3 < \ldots. < n$

3) Big Omega (Ω):Describes the Best case Scenario Represents the lower bound time complexity of an algorithm It represents the fastest time / behavior in which an algorithm can run.

$$1 < log\{n) < sqrt(n) < n < n\,x\,log(n) < n2 < nl < .... < n$$

The best case time complexity is Ω(1),as it takes constant time to execute.

4) Functions:

#O(1) - Constant time:

O(1) describes algorithms that take the same amount of time to compute regardless of the input size.

For instance, if a function takes the same time to process ten elements and 1 million items, then we say that it has a constant growth rate or O(1). Let's see some cases.

Examples of constant runtime algorithms:

- Find if a number is even or odd.
- Check if an item on an array is null.
- Print the first element from a list.
- Find a value on a map.

#O(n) - Linear time

Linear running time algorithms are widespread. These algorithms imply that the program visits every element from the input.

Linear time complexity *O(n)* means that the algorithms take proportionally longer to complete as the input grows.

Examples of linear time algorithms:

- Get the max/min value in an array.
- Find a given element in a collection.
- Print all the values in a list.

#O(n^2) - Quadratic time

A function with a quadratic time complexity has a growth rate of n^2. If the input is size 2, it will do four operations. If the input is size 8, it will take 64, and so on.

Here are some examples of quadratic algorithms:

- Check if a collection has duplicated values.
- Sorting items in a collection using bubble sort, insertion sort, or selection sort.
- Find all possible ordered pairs in an array.

#O(n^c) - Polynomial time

Polynomial running is represented as O(n^c), when c > 1. As you already saw, two inner loops almost translate to O(n^2) since it has to go through the array twice in most cases. Are three nested loops cubic? If each one visit all elements, then yes!

Usually, we want to stay away from polynomial running times (quadratic, cubic, n^c, etc.) since they take longer to compute as the input grows fast. However, they are not the worst.

#O(log n) - Logarithmic time

Logarithmic time complexities usually apply to algorithms that divide problems in half every time.
For instance, let's say that we want to look for a book in a dictionary. As you know, this book has every word sorted alphabetically. If you are looking for a word, then there are at least two ways to do it:

Algorithm A:

1. Start on the first page of the book and go word by word until you find what you are looking for.

Algorithm B:

1. Open the book in the middle and check the first word on it.
2. If the word you are looking for is alphabetically more significant, then look to the right. Otherwise, look in the left half.
3. Divide the remainder in half again, and repeat step #2 until you find the word you are looking for.

#O(n log n) - Linearithmic

Linearithmic time complexity it's slightly slower than a linear algorithm. However, it's still much better than a quadratic algorithm (you will see a graph at the very end of the post).
Examples of Linearithmic algorithms:

- Efficient sorting algorithms like merge sort, quicksort, and others.
-

O(2^n) - Exponential time

Exponential (base 2) running time means that the calculations performed by an algorithm double every time as the input grows.
Examples of exponential runtime algorithms:

- Power Set: finding all the subsets on a set.
- Fibonacci.

- Travelling salesman problem using dynamic programming.

#O(n!) - Factorial time

Factorial is the multiplication of all positive integer numbers less than itself. For instance:

5! = 5 x 4 x 3 x 2 x 1 = 120

It grows pretty quickly:

20! = 2,432,902,008,176,640,000

As you might guess, you want to stay away, if possible, from algorithms that have this running time!

Examples of O(n!) factorial runtime algorithms:

- Permutations of a string.
- Solving the traveling salesman problem with a brute-force search

Q4. Write algorithm and find the space and time complexity of the following data structures operations:

1. Array
    a. Insertion
    b. Deletion
    c. Traversal
    d. Search (Linear)
    e. Bubble Sort
2. Stack (Implement stack data structure using array)
    a. Traversal
    b. Insertion
    c. Deletion
3. Queue (Implement Queue data structure using array)
    a. Traversal
    b. Insertion
    c. Deletion

ANS:

A)Array:

1. Insertion

Algorithm -

To sort an array of size n in ascending order:

1: Iterate from arr[1] to arr[n] over the array.

2: Compare the current element (key) to its predecessor.

3: If the key element is smaller than its predecessor, compare it to the elements before. Move the greater elements one position up to make space for the swapped element.

Space and time complexity-
Worst Case Time Complexity [Big-O]: O(n2)
Best Case Time Complexity [Big-omega]: O(n)
Average Time Complexity [Big-theta]: O(n2)
Space Complexity: O(1)

2.Deletion:
Steps

1. Back=1
2. While (Back<M) repeat 3 and 4
3) Reg[Back]= Reg[Back+1]
4) Back= Back+1
5) M=M-1
6) End

Space and time complexity-
It takes O(n) time to find the element you want to delete. Then in order to delete it, you must shift all elements to the right of it one space to the left.
This is also O(n) so the total complexity is linear.

3.Traversal :

Step 1 : [Initialization] Set I = LB.
Step 2 : Repeat Step 3 and Step 4 while I < = UB.
Step 3 : [processing] Process the A[I] element.
Step 4 : [Increment the counter] I = I + 1.
Step 5: Exit.

the complexity of this algorithm is O(n).

4.Linear Search (Array A, Value x)

Step 1: Set i to 1
Step 2: if i > n then go to step 7
Step 3: if A[i] = x then go to step 6
Step 4: Set i to i + 1
Step 5: Go to Step 2
Step 6: Print Element x Found at index i and go to step 8
Step 7: Print element not found
Step 8: Exit

Space and time complexity-
insertion sort, on average, takes O (n 2) O(n^2) O(n2) time.
space complexity of O (1) O(1) O(1).

5.Bubble Sort:
Algorithm-

```
1. Flag= false, i=0;
2. While(i<N and(flag= false)
     repeat steps 3 to 5.
3. flag= true; i=i+1
4. for(j=1) to N-1 repeat steps 5
5. if (LIST[j]>List[J+1])
then {
Temp=List[j];
List[j]=List[j+1]
List[j+1]=Temp;
flag=false;
}
{
end of j loop
}
{
end of while loop
}
End.

Space complexity:
Worst complexity: n^2
Average complexity: n^2
Best complexity: n
Space complexity: 1
```

B)Stack

1.Traversal:

We can't traverse through stack. Stacks are a type of container adaptor, specifically designed to operate in a LIFO context (last-in first-out), where elements are inserted and extracted only from one end of the container. Elements are pushed/popped from the "back" of the specific container, which is known as the top of the stack.

It is not intended for stack to show this behavior, for this we have other containers

2.Insertion

Algorithm:

1) IF TOP = MAX then
   Print “Stack is full”;
   Exit;
   2) Otherwise

TOP: = TOP + 1; /*increment TOP*/
STACK (TOP):= ITEM;
3) End of IF
4) Exit

the time complexity to insert an element from a stack equal to O(1)
space complexity is: O(1)

3.Deletion:

1) IF TOP = 0 then
   Print “Stack is empty”;
   Exit;
2) Otherwise
   ITEM: =STACK (TOP);
   TOP:=TOP – 1;
3) End of IF
4) Exit

the time complexity to delete an element from a stack equal to O(1)
space complexity is: O(1)

C)Queue:

1.Insertion:

```
procedure enqueue(data)

   if queue is full
      return overflow
   endif

   rear ← rear + 1
   queue[rear] ← data
   return true

end procedure
```

The time complexity for insertion is O(1) while deletion is O(n)
(in the worst case) for a single operation. The amortized costs for both are O(1) since having to delete n elements from the queue still takes O(n) time.
space complexity is: O(1)

2.Deletion:

```
procedure dequeue

   if queue is empty
```

```
    return underflow
  end if

  data = queue[front]
  front ← front + 1
  return true

end procedure
```

time: O(1)
Size: O(1)