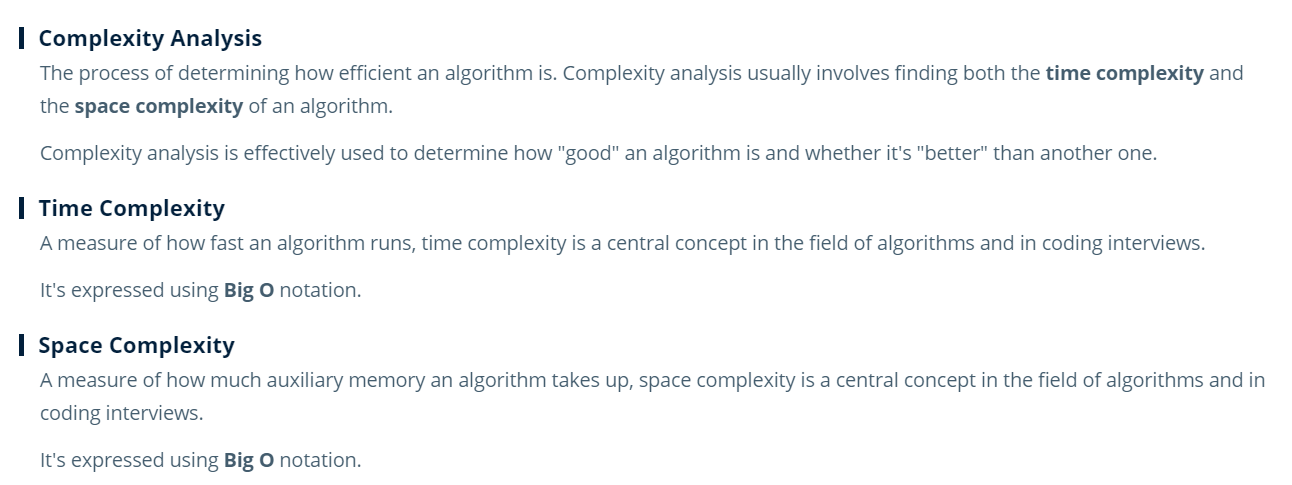
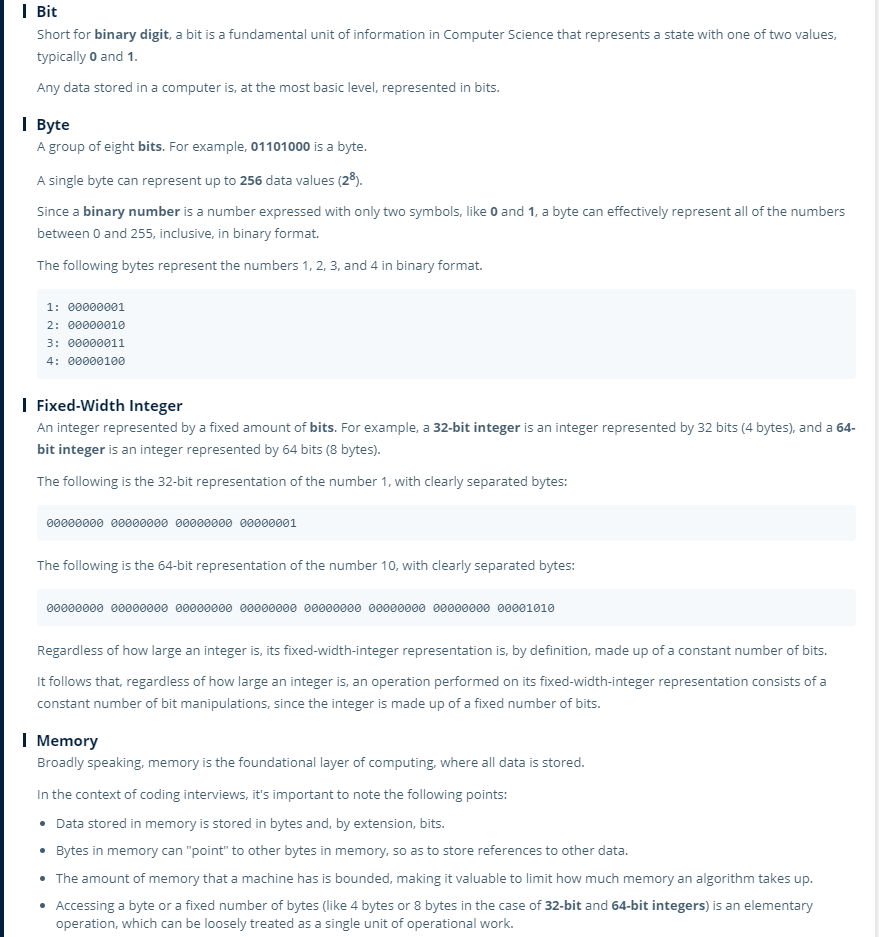
**COMPLEXITY ANALYSIS**



**MEMORY**



**Big O Notation:**

* It is the measure of an algorithm’s speed w.r.t growth of the input size
* Using Big O notation , we can express the time complexity

Example:

Let’s take ‘a’ is an array of length N, we are having below three functions

F1(a) 🡪 1 + a[0] : it fetch first element of the array and returns the value added to 1 🡪 O(1)

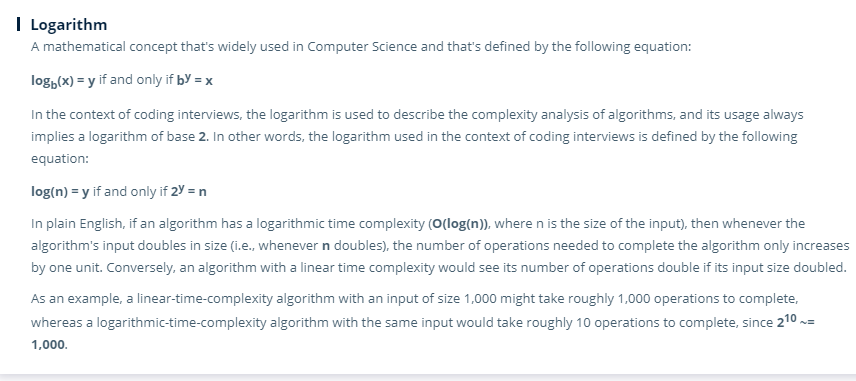
F2(a) 🡪 sum(a) : it iterate through each element and calculate total sum 🡪 O(n)

F3(a) 🡪 pair(a) : it return a pair of all elements in array. So that it iterate through each element twice. 🡪 O(n^2 )

* Big O notation is the notation for expressing the asymptotic analysis on complexity of algorithm that how the execution behaves/grows when the size of the input N grows.
* Even though we consider each operations into count, constant number of operations are always brought down to 1 as in big O just like F1(a). In F1(a) , actually it involves reading of two integers and adding them. This can involve multiple operations based on whether we are reading a 32 bit or 64 bit integer/finding the memory address it resides etc. But when N grows to infinity , these numbers are negligible and can be considered as constant. So we can express it as O(1)
* Similarly if a function F4 calls all the above three functions then its complexity can be expressed as O(1+n+n^2) which in turn is O(n^2) since when N grows to infinity, the other two (1 and n) will be relatively small when compared n^2 which will be gigantic
* Some of the main Big O notations are :
* O(1)
* O(log(n))
* O(n)
* o(nlogn)
* O(n^2),O(N^3),O(N^4) ..etc
* O(2^n)
* O(n!)
* Let’s say we have an algorithm that consumes two arrays of length n and m ( a(n) and b(m) ) . Now if this algorithm merges these two arrays and then do the same operation as of F3(a) in the first example, then big O notation of this algorithm becomes O(N^2 + M^2).
* Let’s say another algorithm have a big O notation of O(N^2 + 2M) , this can be re written as O(N^2 + M). If you see, we haven’t removed M , since M and N are two different values and can not be compared.



**Logarithm:**



Logb(x)=y iif by=x

* Usually in mathematics , if we say log(x) without specifying a base , then base defaults to 10. But in computer science, if we say log(N) without a base , then base defaults to 2.
* So in general:

Log(1) 🡪 2y=1 🡪 y=0

Log(2) 🡪 2y=2 🡪 y=1

Log(4) 🡪 2y=4 🡪 y=2

Log(8) 🡪 2y=8 🡪 y=3

Log(16) 🡪 2y=16 🡪 y=4

…….

Log(256) 🡪 2y=16 🡪 y=8

…….

Log(N)=Y iif 2y=N 🡪 which means as N doubles, complexity grows as log(N), i.e Y only by a tiny 1. Which shows us O(log(N)) < O(N) but greater than O(1)

**Example:**

1. Let’s take an array of size 8:

A[8] = [0,1,2,3,4,5,6,7]

Suppose an algorithm that consumes this array and cuts the array in to half and choose only the left (or right) half for next set of operations . Then cuts the resultant sub array again into half choose only the left (or right) half. After 3 cuts (if it chooses all left halves), we will be having final subset as :

[0]

This in turn implies , 3 elementary operations performed by our algorithm on an array of length 8 , i.e our algorithm is of complexity log(8) 🡪 O(log(N))

1. Binary tree search on a perfectly balanced binary tree is of complexity O(log(N))

**Arrays:**



1. There are static and dynamic arrays. In Python Array implementation is dynamic in nature
2. Let’s take an array of 64 bit integers of length 4🡪 a=[1,2,3,4]
   * Here 64 bit means , 8 slots of 8 byte memory addresses a number will occupy
   * So as a whole , array occupies 32 slots (4\*8). (say from some slot3 to slot34)
   * So to access a[2],
     + array will look for starting memory address slot , say its slot3.
     + Then OS will check how many bytes/bits/memory slots that 1 element take up. Here it is 64 bit/8bytes/8slots
     + Now it will check what index we are trying to access , i.e 2
     + Then finally it will traverse directly to that index by applying formula starting slot + index \*(slots each element occupied) 🡪 3+(2\*8) = slot 19
3. Time and space complexities of different operations on an array (T for time complexity S for space complexity) :
   * + Initializing a list of size N 🡪 O(N) TS
     + Accessing/Updating a value at a given index 🡪 O(1) TS
     + Inserting a value at the beginning or middle OR popping out a value at beginning or middle 🡪 O(N) T , O(1) S
     + Inserting a value at end for static array 🡪 O(N) T , O(1) S
     + Inserting a value at end for dynamic array 🡪 Amortized complexity of O(1)
     + Popping out a value at the end of an array 🡪 O(1) ST
     + Just traversing an array 🡪 O(N) T , no space complexity

**Linked List:**

* Linked list doesn’t require back to back memory slots allocated unlike arrays.
* Singly linked list will store an element and a pointer to next element in continues memory slots so that every time when you add an element it requires only two continues memory slots to be allocated. On the other hand Doubly linked list requires 3 continues memory slots one for previous pointer , one for element and one for next pointer

