# Data Structure.

Data structures are the collection of data and the relationships among data.

# Complexity Analysis.

Two types of complexity and both refers to the space-time complexity.

* Time Complexity.

How fast a program or an algorithm runs.

* Space Complexity.

How much memory or space an algorithm uses up.

* Time-space Complexity.

Time and Space complexities are important concepts in data structures and algorithms. These complexities are calculated to find the efficient algorithm which uses less time to execute in the least memory space possible.

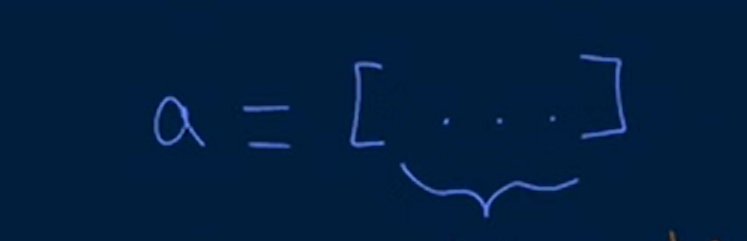
# Memory.

| Data Type | Memory (bytes) | Range | Format Specifier |
| --- | --- | --- | --- |
| short int | 2 | -32,768 to 32,767 | %hd |
| unsigned short int | 2 | 0 to 65,535 | %hu |
| unsigned int | 4 | 0 to 4,294,967,295 | %u |
| int | 4 | -2,147,483,648 to 2,147,483,647 | %d |
| long int | 4 | -2,147,483,648 to 2,147,483,647 | %ld |
| unsigned long int | 4 | 0 to 4,294,967,295 | %lu |
| long long int | 8 | -(2^63) to (2^63)-1 | %lld |
| unsigned long long int | 8 | 0 to 18,446,744,073,709,551,615 | %llu |
| signed char | 1 | -128 to 127 | %c |
| unsigned char | 1 | 0 to 255 | %c |
| float | 4 | 1.2E-38 to 3.4E+38 | %f |
| double | 8 | 1.7E-308 to 1.7E+308 | %lf |
| long double | 16 | 3.4E-4932 to 1.1E+4932 | %Lf |

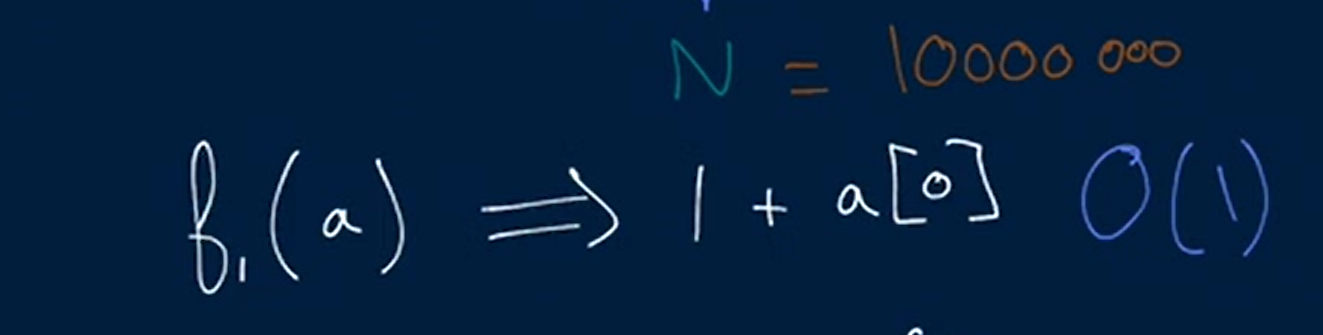
* One memory slot store 8 bit = 1 byte.

# Bog O Notation.

Big O refers to the worst casecomplexity.



* Constant Time complexity – O (1).



**O (1). Constant time. Time complexity will remain constant irrespective of the size of the array.**

* **Linear Time Complexity O (N).**

****

**O (N). Linear time. Linear Time Complexity will be linear as the size of the array increase.**

* **Quadratic O (n\*n)**

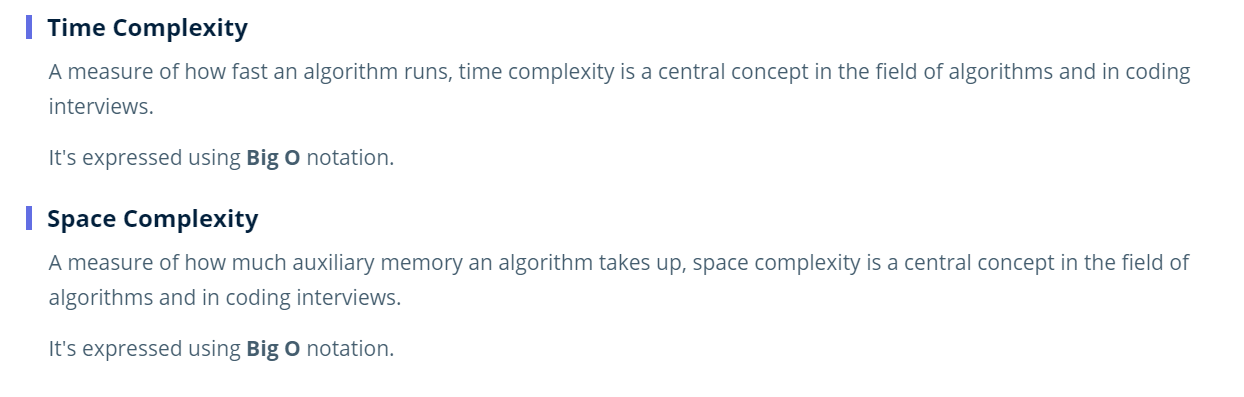


In Pair function with each element make pair other element of array. So i.e.

1st element of array has to iterate N time to make pair.

2nd element of array has to iterate N time to make pair and so on.

This will increase the time for that algorithm. So the time complexity will be **O (n\*n)**

****

# Arrays.

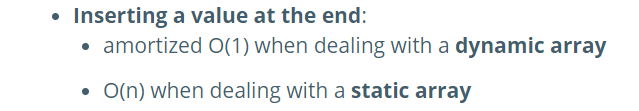
Types of arrays.

1. Dynamic Arrays
2. Static Arrays

**Static Arrays Time Complexity.**

A linear collection of data values that are accessible at numbered indices, starting at index 0. The following are an array's standard operations and their corresponding time complexities:

1. Get => O (1) ST
2. Set => O (1) ST
3. Init => O (N) ST
4. Traverse => O (N) T, O (1) S
5. Copy => O (N) ST
6. Inserting => **O (N) T, O (1)**



0x1234 {1, 2, 3} 0x1238

Now insert after the 2 in that array we have to shift the element 3 to the next address 0x1238 but our array is static. It doesn’t matter we want to put element at beginning, end or the middle we have to shift all other elements but we don’t have the space because we have static array. What option we have, we can say the OS for enough memory for the insertion of the new element then we have to copy the entire array to the new array. You have to perform N operation to make room for the N + 1 element no matter you want to insert at beginning, end or middle so it will take O (N) T and O (1) S. because the previous memory will be free it will not always O (N) because it is a static array.

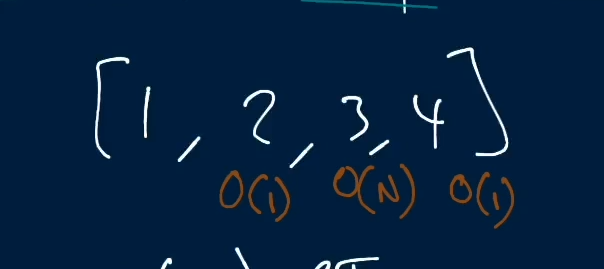
1. Removing at beg => O (N) ST
2. Removing at mid => O (N) ST
3. Removing at end => O (1) ST

**Dynamic Arrays Time Complexity.**

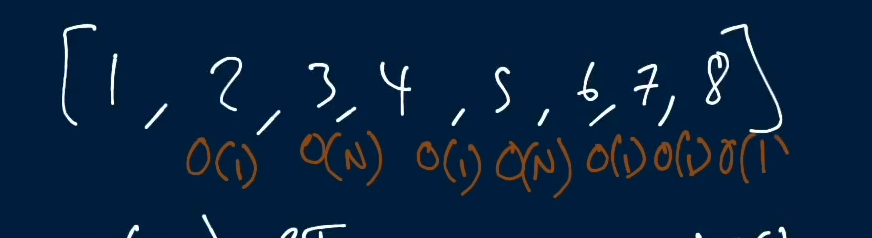
Dynamic array under the hood allows us the faster insertion at the end of the array. Dynamic array can change in size in cpp and java Dynamic arrays are vector and array lists but in other languages like python and JavaScript standard arrays under the hood are the dynamic arrays.

**How it works.**

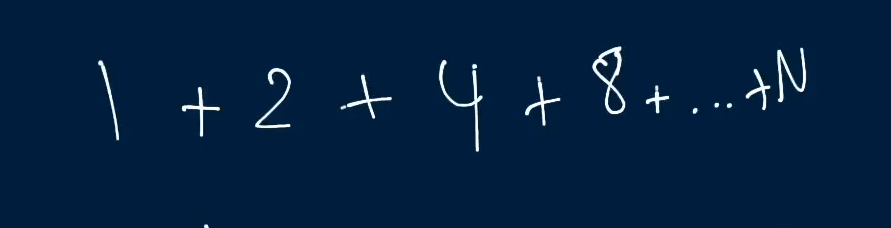
Your OS allocate twice as much memory as you’re asking for the array like if we want to insert one element OS will allocate two element space {1, \_\_\_} now if it will fill the space like {1, 2} then here it comes the static array problem then we have to copy the whole array to the ne array like if we want to inert 3 after 2 {1, 2, 3, \_\_} OS will allocate the double of the size of the previous now the insertion of the 3will have Time complexity of O (N) then the elements have Time complexity of O (1) it continues the procedure.

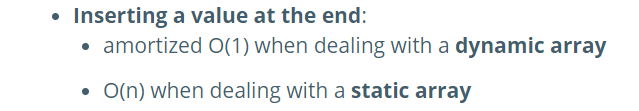


Now if we want to put another element,



Time Complexity will be **O (1)** because we are doubling the array when length of array is 1, 2, 4, and 8 and we keep doubling until we reach N. Because we have so many constant time insertion as compared to O (N) we can take it as the O (1) TS AS constant time.





1. Inserting at end => O (1) ST
2. Inserting at beg => O (N) T, O (1) S
3. Inserting at middle => O (N) T, O (1) S
4. Removing at beg => O (N) T, O (1) S
5. Removing at mid => O (N) T, O (1) S
6. Removing at end => O (1) ST