What is Space Complexity?

Space complexity is a measure of the total amount of memory including the space of input values with respect to the input size, that an algorithm needs to run and produce the result.

Auxiliary Space and Space complexity

- **Auxiliary Space:** It is the temporary or extra space used by an algorithm apart from the input size in order to solve a problem.
- **Space complexity:** It is the total space used by an algorithm in order to solve a problem including the input size. It includes both auxiliary space and space taken by the input size.

  Space complexity = Auxiliary Space + Space taken by the input size

Calculating Space Complexity

The calculation of space complexity is necessary for determining the algorithm's efficiency. However, the space complexity also depends on the programming language, the compiler used, and the machine on which it is executed.

- Let us consider a program for calculating the sum of two numbers:

```
// Function to print the sum of two integers
function SumOfTwoIntegers()
   // Reading integers num1 and num2
   read(num1);
   read(num2);

   // Calculating the sum of two integers
   sum = num1 + num2;
   print("The sum of two integers");
   print(sum);
```

  We have declared three variables 'num1', 'num2',' 'sum', considering them of data type 'int' let the space occupied by 'int' data type be 4 bytes, hence the total space consumed is 4*3 = 12 bytes.

  Hence we can say that the space complexity of the above program is O(1) as 12 is a constant.

- Now, let us consider a program for calculating the sum of the numbers in an array:

```
// Function to calculate the sum of elements of the array
```

```
function sum()
   // Reading n, the size of the array
   read(n);
   // Declaring array A of size n
   A[n];

   // Reading the values of the array A
   for i = 0 to n - 1
      read(A[i]);

   // Calculating the sum of elements of array A
   for i = 0 to n - 1
      sum = sum + A[i];

   print("The sum of elemtents of array is");
   print(sum);
```

We have declared variables ‘n’, ‘sum’ and array of size ‘n’, considering them of datatype ‘int’, let the space occupied by ‘int’ be 4 bytes, hence the total space consumed is 4+4+n*4 bytes. Note that the auxiliary space is O(1) as only the sum variable is declared apart from the input array.

Since, the space consumed is a linear function of ‘n’, the size of the array so the space complexity of the above program is O(n), where n is the size of the array.

- Let’s consider a recursive version of the above program of calculating the sum of the numbers in an array

```
// Recursive function to calculate the sum of elements of array
function calc(A, index)
   if(index < 0) return 0;
   return A[index] + calc(A, index - 1);

// Function to calculate the sum of elements of array
function sum()
   // Reading n, the size of the array
   read(n);
   // Declaring array A of size n
   A[n];

   // Reading the values of the array A
   for i = 0 to n - 1
      read(A[i]);
```

```
// Calculating the sum of elements of array A through recursive function calc()
sum = calc(A, n - 1);

print("The sum of elemtents of array is");
print(sum);
```

The above program is a recursive version of the program to find the sum of numbers in an array. However, due to the recursive calls, we need to consider the space consumption due to function call stack too. The function call stack is responsible for keeping track of the function calls. The function call stack is made up of stack frames - one for each method call.

A stack frame consists of -

- Local variables
- Arguments passed to the function
- Information about caller's stack frame
- The return address of the function

In the above example, the maximum depth of the recursive call goes from n-1 to 0 as -

```
calc(n-1)
   |
calc(n-2)
   |
calc(n-3)
   .
   .
   .
calc(0)
```

Hence, the maximum depth of the recursion is all the way up to n, as we make n recursive calls which is definitely an auxiliary space for our program. So, the overall space complexity is O(n) + O(n) => O(n), where one term is the space consumed due to input size and the other is the auxiliary space due to function call stack.

NOTE: The notations used above for representing the space complexities of the programs have been discussed in detail in the tutorial for time complexity analysis.

The trade-off between Time and Space Complexity

The best algorithm to solve a particular problem is no doubt the one that requires less memory space and takes less time to execute. However, designing such an algorithm is not a trivial task, there can be more than one algorithm to solve a given problem one may require less memory space while the other may require less time to solve the problem.

So, it is quite common to observe a tradeoff between time and space consumed while designing an algorithm, where one needs to be sacrificed for the other. So, if space is a constraint, one might choose an algorithm that takes less space at the cost of more CPU time and vice-versa. Hence, we must choose an algorithm according to the requirements and the environment in which it needs to be executed.