Space Complexity

Space complexity is the **total amount of memory space used by an algorithm/program including the space of input values for execution**. So to find space-complexity, it is enough to calculate the space occupied by the variables used in an algorithm/program.

Space complexity and Auxilliary space

- Auxiliary space is just a temporary or extra space and it is not the same as space-complexity.
- **Space Complexity = Auxiliary space + Space use by input values**
- The best algorithm/program should have the less space-complexity. The lesser the space used, the faster it executes.

Why do you need to calculate space complexity?

- Similar to Time Complexity, Space-complexity also plays a crucial role in determining the efficiency of an algorithm/program. If an algorithm takes up a lot of time, you can still wait, run/execute it to get the desired output. But, **if a program takes up a lot of memory space, the compiler will not let you run it**.

How to calculate Space Complexity of an Algorithm?

- Example 1

```
#include<stdio.h>
int main()
{
  int a = 5, b = 5, c;
  c = a + b;
  printf("%d", c);
}
```

Output …. Space required more?

CPU Time: 0.00 sec(s), Memory: 1364 kilobyte(s)

```
10
```

No, Actual Space complexity

- **Explanation:** Do not misunderstand space-complexity to be 1364 Kilobytes as shown in the output image.
- In the above program, 3 integer variables are used. The size of the integer data type is 2 or 4 bytes which depends on the compiler.
- Now, lets assume the size as 4 bytes.
- So, the total space occupied by the above-given program is 4 * 3 = 12 bytes.
- Since no additional variables are used, no extra space is required.
- Hence, **space complexity for the above-given program is O(1), or constant.**

Example 2

```
#include <stdio.h>
int main()
{
  int n, i, sum = 0;
  scanf("%d", &n);
  int arr[n];
  for(i = 0; i < n; i++)
  {
    scanf("%d", &arr[i]);
    sum = sum + arr[i];
  }
  printf("%d", sum);
}
```

Output

- In the above-given code, the array consists of n integer elements. So, the space occupied by the array is 4 * n.
- Also we have integer variables such as n, i and sum.
- Assuming 4 bytes for each variable, the total space occupied by the program is 4n + 12 bytes.
- Since the highest order of n in the equation 4n + 12 is n, so **the space complexity is O(n) or linear.**

Example 3

```
/* Recursive function for summing list
  of number */

float rsum (float list [ ], int n)
{
  if(n)
  return rsum (list , n-1 )+ list[n-1];
  return 0;
}
```

Type	Name	Number of bytes
parameter: float	list [ ]	4
parameter: integer	n	2
return address:(used internally)		2(unless a far address)
TOTAL per recursive call		8

$$\boxed{S_{sum}(I)=S_{sum}(n)=8n}$$

Summary

O(1)	Constant Space Complexity occurs when the program doesn't contain any loops, recursive functions or call to any other functions.
O(n)	Linear space complexity occurs when the program contains any loops.

What is the time complexity of the following algorithms?

```
Algorithm abc(a,b,c)
{
  return a+b*c+(a-b-)/a+b+4.0
}
```

```
Algorithm Sum(a,n)
{
  s:=0
      for i:=1 to n do
            s:=s+a[i];
return s;
}
```

Function's growth

- Take the following list of functions and arrange them in ascending order of growth rate. That is, if function g(n) immediately follows function f(n) in your list, it should be the case that f(n) is O(g(n)).
  - $f_1(n) = n^{2.5}$
  - $f_2(n) = \sqrt{2n}$
  - $f_3(n) = n + 10$
  - $f_4(n) = 10^n$
  - $f_5(n) = 100^n$
  - $f_6(n) = n^2 log(n)$

We can start approaching this problem by putting f_4 and f_5 at end of the list, because these functions are exponential and will grow the fastest. $f_4 < f_5$ because $10 < 100$. Other four functions are polynomial and will grower slower then exponential. We can represent f_1 and f_2 as: $n^{2.5} = n^2 * \sqrt{2n}$; and $\sqrt{2n} = 2n^{0.5}$. Now, we can say that out of all polynomial functions f_2 will be the slowest because it has the smallest degree. Morover, and will be bounded by f_3 because it has a higher degree of 1. Furthermore, f_1 and f_6 will be beween exponential f_4 and f_5 and polynomial f_2 and f_3, because polynomial functions grow slower and both f_4 and f_5 and have the highest degree of 2 out of all other polynomial functions. And f_6 will be bounded by f_1 because $f_6 = n^2 log(n)$ and $f_1 = n^2\sqrt{2n}$ and $log(n) = O(\sqrt{2n})$. Therefore the final order will be: $f_2(n) < f_3(n) < f_6(n) < f_1(n) < f_4(n) < f_5(n)$

Solution

- f4 and f5 at end of the list, because these functions are exponential and will grow the fastest. f4 < f5 because 10 < 100.
- Other four functions are polynomial and will grower slower then exponential.
- We can represent f1 and f2 as: n ^2.5 = n 2 ∗ √ 2n; and √ 2n = 2n ^0.5 . Now, we can say that out of all polynomial functions f2 will be the slowest because it has the smallest degree.
- Morover, and will be bounded by f3 because it has a higher degree of 1. Furthermore, f1 and f6 will be beween exponential f4 and f5 and polynomial f2 and f3, because polynomial functions grow slower and both f4 and f5 and have the highest degree of 2 out of all other polynomial functions.
- And f6 will be bounded by f1 because f6 = n 2 log(n) and f1 = n 2 √ 2n and log(n) = O(√ 2n).
- Therefore the final order will be: f2(n) < f3(n) < f6(n) < f1(n) < f4(n) < f5(n)