Space Complexity

Space Complexity of an algorithm is total space taken by the algorithm with respect to the input size. Space complexity includes both Auxiliary space and space used by input.

Auxiliary Space is the extra space or temporary space used by an algorithm.

1.Space Complexity on Iterative Algorithms:-

1.

```
A - input array
n - size of input array
Algo(A, n){
      int i;
      for(i=1 to n)
            A[i]=0;
}
```

For the entire program we require n elements stored in array and one extra variable i.
So ,the total space required is **(n+1) cells.** (considering one variable occupy one memory cell)
The n cells are given in the **input** hence **will not be counted**.
So the **extra space** required is only **one cell.**
Space complexity of this program is **O(1).**
If in the above program ,there are a **constant** number of **extra variables** then the space complexity will be **O(c) or O(1).**

2.

```
A - input array
n - size of array
Algo(A, n){
      int i;
      create B[n];          //creating an extra array B
      for(i=1 to n)
            B[i]=A[i];
}
```

In this algo, we create an **extra array B** of size same as input **size n** and variable i occupies one extra cell.
So the total extra space will be (n+1)
Hence, **the space complexity is O(n).**

While calculating the space complexity **don't count the input variables or data structure** since we cannot reduce the space required by input .

3.

```
Algo(A, n){
            int i,j;
            create B[n][n];                     //creating 2-d array
            for(i=1 to n)
                for(j=1 to n)
                  B[i][j]=A[i];
}
```

Here, we create an extra 2D array B of size n*n, and there are two extra variables used i,j .
So total extra space used is **(n^2 +1).**
Space complexity= O(n^2)

2. Space Complexity on Recursive Algorithms:-

Consider a recursive program and let's see with the help of examples how to calculate the space complexity.

1.

```
n - user input
A(n){
      if(n>=1){
            A(n-1);           //calling recursion
            cout<<n;          //print n
      }
}

```

Let us consider n=3 and see how the **recursion tree** looks like.

```
                             A(3)
                          /        \
                     A(2)      Print(3)
                    /        \
                 A(1)     Print(2)
                /      \
            A(0)    Print(1)
```

In order to execute A(3), 4 recursive calls are required (A(3), A(2), A(1), A(0))
For executing A(2),   3 recursive calls are required.(A(2), A(1), A(0))
In general for **A(n)** ,   **(n+1) recursive calls** are required.

Let's see how recursion works and what is the order in which these functions are called and what output is printed.In order to analyse this we need a stack.All the recursive calls of A will be pushed onto the stack.

Initially we start from A(3) and traverse the recursion tree from top to bottom and left to right. Whenever you visit a node and if that node is a function call then push it to the stack or if it is any operation then execute that operation.Whenever we visit a node last time and it is a function call we pop the function call from the stack.

Top

```
stack  A(3)
       A(2) A(3)
       A(1) A(2) A(3)
       A(0) A(1) A(2) A(3)
       A(1) A(2) A(3)          // pop A(0) printed 1
       A(2) A(3)          // pop A(1) printed 2
       A(3)               // pop A(2) printed 3
       empty stack        // pop A(3)
```

Output : 1 2 3

Space complexity is depending on the depth or height of the stack.
Every stack record is going to occupy a **constant space(let k)**
Total recursive calls = n+1
Total space required is= **(n+1)*k**   **space complexity = O(n)**

2. Let's take one more example for better understanding.

```
A(n){
    if(n>=1){
         A(n-1);
         cout<<n;
         A(n-1);
    }
}
```

Let’s trace the recursion tree for A(3).

```
                         A(3)
                    /      \       \
                A(2)     Print(3)  A(2)
           /     \     \
       A(1)   Print(2)  A(1)
   /    \    \         /    \    \
 A(0) Print(1) A(0)  A(0) Print(1) A(0)
```

To solve A(3), total recursive calls = 15
For A(2) , total recursive calls   = 7
For A(1), total recursive calls   = 3
If you observe the pattern 15 = 2^{3+1} -1 ,   (16-1)
                                        7 = 2^{2+1} -1 ,   (8-1)
                                        3 = 2^{1+1} -1   (4-1)
In general we can say , **Total recursive calls for A(n) = 2^{n+1} -1**
Total number of recursive calls are **2^{n+1} -1** but does that mean we need a stack of maximum depth **2^{n+1} -1.**
Let’s see how the stack works.

```
stack  empty
       A(3)                      //push A(3)
       A(2) A(3)                 //push A(2)
       A(1) A(2) A(3)                 //push A(1)
       A(0) A(1) A(2) A(3)       //push A(0)
       A(1) A(2) A(3)                 //pop A(0) & print 1
       A(0) A(1) A(2) A(3)       //push A(0)
       A(1) A(2) A(3)                 //pop A(0) & push A(1)
       A(2) A(3)                 //pop A(1) & print 2
       A(1) A(2) A(3)                 //push A(1)
       A(0) A(1) A(2) A(3)       //push A(0)
       A(1) A(2) A(3)                //pop A(0) & print 1
       A(0) A(1) A(2) A(3)     //push A(0)
       A(1) A(2) A(3)              //pop A(0)
       A(2) A(3)               //pop A(1)
       A(3)                    //pop A(2) & print 3
 // here i have made the recursion tree of one part only you
   can expand the further right half A(2) and get the complete
   output.
```
Output: 1 2 1 3 1 2 1

Even though total recursive calls were **2^{n+1} -1 ,** the stack required only **(n+1) cells** ,assuming that every record is of constant space (k).
Total space required by the program is **(n+1)*k**
Space complexity = O(n)

So, we can conclude that while calculating the space complexity of recursive programs we must check how the stack works and the depth of the stack required to solve the problem instead of blindly following the total number of recursive calls.

Amortised Analysis

It is a type of analysis in which, instead of finding out the time complexity of per operation in Data Structure, We calculate the total time complexity of 'n' operations and then find out the avg. time complexity i.e. [(total time) / n] per operation.

With the help of amortised analysis, we can show the avg. time of operation would be small as compared to the single operation time.

This type of Analysis is helpful in analyzing the time complexity of :-

1. Hash Tables
2. Splay Trees

3. Disjoint Sets

There are 3 Ways to do the Amortized Analysis:-

1. Aggregate Method
2. Accounting Method
3. Potential method

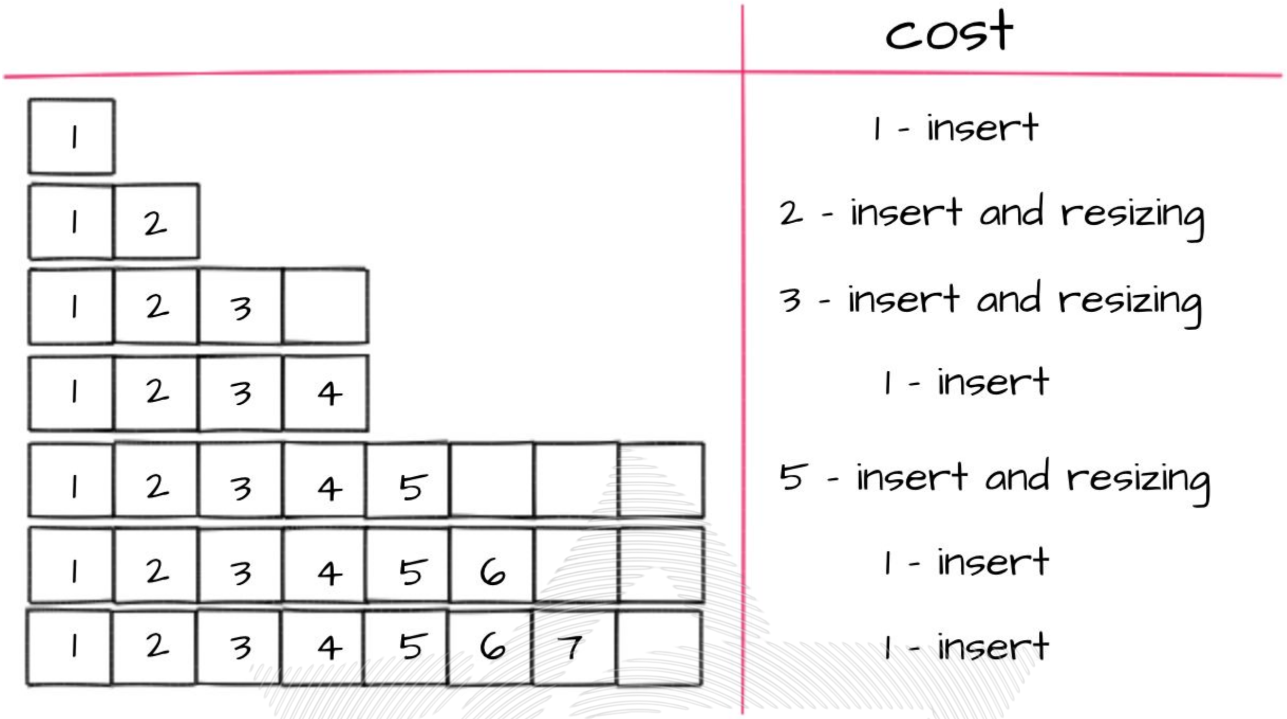
Aggregate Analysis

Using Aggregate Analysis, we can prove that the insertion time in the Dynamic Table scheme is of O(1) which is a great result used in hashing. Also, the concept of dynamic table is used in vectors in C++ , ArrayList in Java.

E.g : **Hash Table Insertions:-**

Operations:-

1. Increase the size of the table (array) whenever it gets full.
2. Allocate memory for the larger table of size double of the old table.
3. Copy the Contents of the old Table into the new table.
4. Free the space of the old table.
5. If the table has space available, we simply insert a new item in available space.



Time Complexity of above Operations :-

Item No.	1	2	3	4	5	6	7	8	9	10	......
Table Size	1	2	4	4	8	8	8	8	16	16	......
Cost	1	2	3	1	5	1	1	1	9	1	......

$$\text{Amortized Cost} = \frac{(1 + 2 + 3 + 5 + 1 + 1 + 9 + 1\ldots)}{n}$$

We can simplify the above series by breaking terms 2, 3, 5, 9.. into two as (1+1), (1+2), (1+4), (1+8)

$$\text{Amortized Cost} = \frac{[\overbrace{(1 + 1 + 1 + 1\ldots)}^{\text{n terms}} + \overbrace{(1 + 2 + 4 + \ldots)}^{\lfloor \text{Log}_2(n-1) \rfloor + 1 \text{ terms}}]}{n}$$

$$<= \frac{[n + 2n]}{n}$$

$$<= 3$$

$$\text{Amortized Cost} = O(1)$$

Worst Case : O(n), if there are n-elements in a table of size ‘n’ , then inserting another element would cause generating a table of size ‘2n’ and then copying all the ‘n’ elements into new table which would cause the time complexity to be **O(n)**.