Space Complexity Analysis

Introduction

- The space complexity of an algorithm represents the amount of extra memory space needed by the algorithm in its life cycle.
- Space needed by an algorithm is equal to the sum of the following two components:
    - A fixed part is a space required to store certain data and variables (i.e. simple variables and constants, program size, etc.), that are not dependent on the size of the problem.
    - A variable part is a space required by variables, whose size is dependent on the size of the problem. For example, recursion stack space, dynamic memory allocation, etc.
- Space complexity **S(p)** of any algorithm **p** is **S(p) = A + Sp(I)** Where **A** is treated as the fixed part and **S(I)** is treated as the variable part of the algorithm which depends on instance characteristic **I**.

Note: It's necessary to mention that space complexity depends on a variety of things such as the programming language, the compiler, or even the machine running the algorithm.
To get warmed up, let's consider a simple operation that sums two integers (numbers without a fractional part):

```
def difference(a, b):
    return a + b
```

In this particular method, three variables are used and allocated in memory:

The first integer argument, a; the second integer argument, b; and the returned sum which is also an integer.
In Python, these three variables point to three different memory locations. We can see that the space complexity is constant, so it can be expressed in big-O notation as **O(1)**.
Next, let's determine the space complexity of a program that sums all integer elements in an array:

```
def sumArray(array):
    size = 0
    sum = 0
    for iterator in range(size):
        sum += array[iterator]
    return sum
```

Again, let's list all variables present in the above code:

- **array**
- **size**
- **sum**
- **iterator**

The space complexity of this code snippet is **O(n),** which comes from the reference to the array that was passed to the function as an argument.
Let us now analyze the space complexity for a few common sorting algorithms. This will give you deeper insight into complexity analysis.

Quick-Sort Space Complexity Analysis

Let us consider the various scenarios possible :
Best case scenario: The best-case scenario occurs when the partitions are as evenly balanced as possible, i.e their sizes on either side of the pivot element are either equal or have a size difference of 1 of each other.

- **Case 1:** The case when the sizes of the sublist on either side of the pivot become equal occurs when the subarray has an odd number of elements and the pivot is right in the middle after partitioning. Each partition will have **(n-1)/2** elements.
- **Case 2:** The size difference of 1 between the two sublists on either side of pivot happens if the subarray has an even number, **n**, of elements. One partition will have **n/2** elements with the other having **(n/2)-1**.
- In either of these cases, each partition will have at most **n/2** elements, and the tree representation of the subproblem sizes will be as below:

Worst case scenario:

This happens when we encounter the most unbalanced partitions possible, then the original call takes place **n** times, the recursive call on **n-1** elements will take place **(n-1)** times, the recursive call on **(n-2)** elements will take place **(n-2)** times, and so on.

Based on the above-mentioned cases we can conclude that:

- The space complexity is calculated based on the space used in the recursion stack. The worst-case space used will be **O(n)**.
- The average case space used will be of the order **O(log n).**
- The worst-case space complexity becomes **O(n)** when the algorithm encounters its worst-case when we need to make **n** recursive call for getting a sorted list.

Practice Problems

Problem 1: What is the time & space complexity of the following code:

```
a = 0
b = 0
for i in range(n):
    a = a + i

for j in range(m):
    b = b + j
```

Problem 2: What is the time & space complexity of the following code:

```
a = 0
b = 0
for i in range(n):
    for j in range(n):
        a = a + j

for k in range(n):
    b = b + k
```

Problem 3: What is the time and space complexity of the following code:

```
a = 0
b = 0
for i in range(n):
    j = n
    while j>i:
        a = a + i + j
        j=j-1
```