Space Complexity

1 Concept of Space Complexity

Space complexity is a fundamental concept in the analysis of algorithms. While time complexity measures how long an algorithm takes to run, **space complexity** measures the total amount of memory (RAM) an algorithm needs to complete its execution based on the size of its input.

For an MSc student, it is important to understand that memory is a finite resource. An algorithm that runs very fast but consumes more memory than the machine possesses will fail to run. Therefore, analyzing space is just as critical as analyzing time.

Mathematically, if n is the size of the input, the space complexity $S(n)$ expresses the memory required as a function of n.

2 Types of Space Usage in Algorithms

When an algorithm runs, it uses computer memory for different purposes. We can categorize this usage into two main types.

2.1 Input Space

This is the memory required to store the input data itself. For example, if you are sorting an array of 1,000 integers, the memory needed to store these 1,000 numbers is the input space. This part usually does not depend on the logic of the algorithm but strictly on the problem definition.

2.2 Auxiliary Space

Auxiliary space is the *extra* space or temporary space used by the algorithm during its execution. This includes memory for:

- Loop counters (e.g., i, j).
- Temporary variables for swapping values.
- Stack space used by recursive function calls.
- Helper data structures like temporary arrays or hash maps.

2.3 Total Space Complexity

The total space complexity of an algorithm is the sum of the input space and the auxiliary space.

$$\text{Total Space} = \text{Input Space} + \text{Auxiliary Space}$$

However, when we talk about space complexity in algorithm analysis (like Big O notation), we often focus implicitly on the **Auxiliary Space**, unless specified otherwise.

3 Components of Space Complexity

Generally, the memory used by a program can be divided into two distinct components.

3.1 Fixed Part

This component is independent of the input size (n). It includes:

- The memory required to store the program's code (instructions).
- Simple variables and constants (e.g., an integer size is 4 bytes, a double is 8 bytes).
- Structure padding and alignment.

Since this does not grow as n grows, it is considered constant space, $O(1)$.

3.2 Variable Part

This component depends directly on the input size n. It includes:

- Dynamic memory allocation (e.g., creating an array of size n).
- Recursion stack space (which grows with the depth of recursion).

This is the part we are most interested in when calculating asymptotic space complexity.

4 Common Space Complexity Classes

4.1 Constant Space: $O(1)$

The algorithm uses a fixed amount of memory regardless of the input size.

- **Example:** Swapping two numbers or checking if a number is even.

4.2 Logarithmic Space: $O(\log n)$

The space grows logarithmically with the input size. This is common in divide-and-conquer algorithms where the recursion depth is $\log n$.

- **Example:** Binary Search (iterative version is $O(1)$, but recursive is $O(\log n)$ due to stack).

4.3 Linear Space: $O(n)$

The space grows directly in proportion to the input size.

- **Example:** Copying an array to another array or simple recursion depth of n.

4.4 Quadratic Space: $O(n^2)$

The space grows with the square of the input.

- **Example:** Creating a 2D matrix of size $n \times n$ (often seen in Dynamic Programming).

4.5 Exponential Space: $O(2^n)$

The space required doubles with every addition to the input size. This is extremely inefficient and impractical for large n.

5 Space Complexity of Iterative Algorithms

Iterative algorithms use loops (for, while) to process data.

5.1 Single Variable Algorithms

If an algorithm processes an array but only uses a few variables (like ‘sum‘, ‘count‘, or loop index ‘i‘), the auxiliary space is constant.

```
// Calculates sum of array
int sum = 0;
for(int i = 0; i < n; i++) {
    sum = sum + arr[i];
}
```

Space Complexity: $O(1)$ auxiliary space.

5.2 Algorithms Using Arrays

If an algorithm creates a new array of size n to store results, the space complexity becomes linear.

```
// Copies array A to B
int B[n];
for(int i = 0; i < n; i++) {
    B[i] = A[i];
}
```

Space Complexity: $O(n)$.

5.3 Nested Structures

Using nested loops does not necessarily increase space complexity unless you are allocating memory *inside* the loops. However, creating a grid or matrix (like in image processing or graph adjacency matrices) results in higher complexity.

```
// Creating a multiplication table
int table[n][n];
```

Space Complexity: $O(n^2)$.

6 Space Complexity of Recursive Algorithms

Recursive algorithms use the system’s call stack to store the state of each function call. This is often the ”hidden” cost of recursion.

6.1 Function Call Stack

Every time a function calls itself, a new ”stack frame” is added to memory. This frame contains local variables and the return address. The space complexity is determined by the **maximum depth** of the recursion tree.

6.2 Space Usage in Simple Recursion

Consider a function to calculate the factorial of n:

```
int factorial(int n) {
    if (n == 0) return 1;
    return n * factorial(n - 1);
}
```

The calls go from n down to 0. The maximum depth is n. **Space Complexity:** $O(n)$.

6.3 Space Usage in Divide and Conquer

In algorithms like Merge Sort, the problem is divided into halves. The recursion depth is not n, but $\log_2 n$. **Space Complexity:** $O(n)$ total (because Merge Sort needs an auxiliary array for merging), but the stack space specifically is $O(\log n)$.

7 Comparison of Algorithms Based on Space Complexity

Here is a quick comparison of common sorting algorithms:

Algorithm	**Space Complexity**	**Type**
Bubble Sort	$O(1)$	Constant (In-place)
Selection Sort	$O(1)$	Constant (In-place)
Insertion Sort	$O(1)$	Constant (In-place)
Quick Sort	$O(\log n)$	Logarithmic (Stack)
Merge Sort	$O(n)$	Linear (Auxiliary Array)
Heap Sort	$O(1)$	Constant (In-place)

8 Time–Space Trade-off

A core principle in computer science is the Time-Space Trade-off. Often, you can reduce the time an algorithm takes by using more memory (space), and vice versa.

- **Example (Hashing):** To search for an item in an unsorted list takes $O(n)$ time. If we use a Hash Table (using $O(n)$ space), we can search in $O(1)$ time. We "paid" with space to "save" time.
- **Example (Dynamic Programming):** In calculating Fibonacci numbers, a simple recursive solution takes exponential time $O(2^n)$. By storing previous results in an array (Memoization), we use $O(n)$ space but reduce the time to $O(n)$.

9 Practical Considerations in Space Complexity

When developing software, theoretical analysis must meet practical reality:

1. **Stack Overflow:** Deep recursion can exceed the fixed size of the call stack (usually a few MBs), causing the program to crash.
2. **Memory Leaks:** In languages like C++, forgetting to free memory (using 'delete' or 'free') causes the variable space to grow indefinitely, eventually consuming all RAM.
3. **Cache Locality:** Algorithms that use space sequentially (like arrays) often run faster than those using scattered space (like linked lists) because of CPU caching, even if the Big O space complexity is the same.

10 Summary of Key Concepts

- **Space Complexity** measures the maximum memory needed relative to input size.
- **Auxiliary Space** is the temporary space used, excluding the input data.
- **Recursion** uses stack memory proportional to the depth of the recursion tree.

- **Trade-off:** We can often improve execution speed by using more memory.
- Ideally, we aim for $O(1)$ or $O(\log n)$ auxiliary space for large datasets.

11 Practice Problems and Exercises

1. **Analyze:** What is the space complexity of an algorithm that creates a new string of length n inside a loop that runs n times?
2. **Compare:** Why does Quick Sort have a better space complexity than Merge Sort in most implementations?
3. **Calculation:** Given an algorithm with an input array of size n and a 2D matrix of size $k \times k$ (where k is a constant independent of n), is the space complexity $O(n^2)$ or $O(n)$?
4. **Conceptual:** Explain why the space complexity of an iterative Fibonacci function is $O(1)$ while the recursive one is $O(n)$.