Advanced Space Complexity in Data Structures and Algorithms (DSA)

Abdelrahman Mohamed Elktob Felefal

July 27, 2025

Contents

1 What is Space Complexity?

Space complexity of an algorithm quantifies the total memory required by the program to execute completely.

Components of Space Usage:

- **Instruction Space:** Memory for compiled instructions.
- **Environmental Stack Space:** Memory for function call stacks.
- **Auxiliary Space:** Extra memory used for computations (temporary variables, structures, etc.)
- **Input Space:** Space for storing inputs (may or may not be counted depending on definition).

Total Space Complexity:

$$S(n) = I(n) + E(n) + A(n)$$

2 Space Complexity Classes

- **Constant Space:** $O(1)$ — uses fixed amount of memory regardless of input.
- **Logarithmic Space:** $O(\log n)$ — binary search, recursion on half inputs.
- **Linear Space:** $O(n)$ — storing input, arrays, hash tables.
- **Polynomial Space:** $O(n^k)$ — matrix algorithms, brute-force combinatorics.
- **Exponential Space:** $O(2^n)$ — naive recursion in problems like TSP, Fibonacci.

3 Space Complexity of Common Algorithms

1. Sorting Algorithms

- Bubble, Insertion, Selection: $O(1)$
- Merge Sort: $O(n)$ (due to merging process)
- Quick Sort: $O(\log n)$ average, $O(n)$ worst (recursion stack)

2. Search Algorithms

- Linear Search: $O(1)$
- Binary Search (iterative): $O(1)$, recursive: $O(\log n)$

3. Graph Algorithms

- BFS: $O(V)$ for visited, $O(V+E)$ total
- DFS: $O(V)$ stack
- Dijkstra (Heap): $O(V)$ space + $O(E)$ for graph
- Floyd-Warshall: $O(n^2)$ matrix

4 Data Structures and Their Space Complexities

Data Structure	Space Complexity
Array (size n)	$O(n)$
Singly Linked List	$O(n)$
Doubly Linked List	$O(n)$
Stack (array or list)	$O(n)$
Queue (array or list)	$O(n)$
Hash Table	$O(n)$
Binary Tree	$O(n)$ (tree) + $O(h)$ recursion
Heap (Binary Heap)	$O(n)$
Trie (n words, m avg length)	$O(n \cdot m)$
Graph (Adjacency Matrix)	$O(n^2)$
Graph (Adjacency List)	$O(n+e)$
Disjoint Set (Union-Find)	$O(n)$

5 Algorithm Design Paradigms and Space

Divide and Conquer

- Often uses recursion stack.
- Example: Merge Sort — $O(n)$ space + $O(\log n)$ stack

Dynamic Programming

- Tabulation: $O(n)$ or $O(n^2)$ depending on DP table.
- Memoization: $O(n)$ call stack + $O(n)$ cache.

Greedy Algorithms

- Typically $O(1)$ to $O(n)$ depending on input storage.

Backtracking

- Space grows with depth of recursion tree.
- Often exponential in worst case.

6 Case Study: Fibonacci

- Naive recursive: $O(2^n)$ time, $O(n)$ stack
- With Memoization: $O(n)$ time, $O(n)$ space
- Iterative DP: $O(n)$ time, $O(n)$ space
- Space-Optimized DP: $O(n)$ time, $O(1)$ space

7 Tips to Optimize Space

- Use in-place updates when possible.
- Reuse arrays instead of creating new ones.
- Convert recursion to iteration to save stack.
- Use bit manipulation to reduce space (e.g., Boolean arrays).

8 Common Pitfalls

- Allocating new arrays in recursive calls unnecessarily.
- Forgetting that stacks/queues take $O(n)$ in space.
- Overuse of hash maps in greedy/DP solutions.

9 Real-World Examples

1. Search Engines

Trie or Ternary Search Trees used: $O(n \cdot l)$

2. Maps/GPS

Graph storage and search algorithms. Space varies with representation.

3. Machine Learning

Matrix operations: $O(n^2)$ or higher in models like transformers.

10 Conclusion

Understanding space complexity allows developers to build efficient, scalable, and memory-conscious applications. It's especially crucial for embedded systems, mobile apps, and real-time processing.