Big O Notation

Big O Notation is used to describe the upper bound of an algorithm's running time. It focuses on the worst-case scenario and gives an idea of the algorithm's efficiency as the input size grows.

Common Rates of Growth

- **Constant:** $O(1)$
- **Logarithmic:** $O(\log n)$
- **Linear:** $O(n)$
- **Linearithmic:** $O(n \log n)$
- **Exponential:** $O(2^n)$
- **Quadratic:** $O(n^2)$
- **Factorial:** $O(n!)$
- **Cubic:** $O(n^3)$

Table of Algorithms, Time, and Space Complexities:

Algorithm	Time Complexity	Space Complexity	Use Case
Sorting Algorithms			
Bubble Sort	$O(n^2)$	$O(1)$	Simple but inefficient; use for small datasets.
Merge Sort	$O(n \log n)$	$O(n)$	Efficient sorting for large datasets.
Quick Sort	$O(n \log n)$ (Average)	$O(\log n)$	Fast sorting; worst-case $O(n^2)$.

Heap Sort	$O(n \log n)$	$O(1)$	In-place, reliable sorting algorithm.
Insertion Sort	$O(n)^2$	$O(1)$	Efficient for small or nearly sorted arrays.
Searching Algorithms			
Linear Search	$O(n)$	$O(1)$	Simple search in unsorted data.
Binary Search	$O(\log n)$	$O(1)$	Fast search in sorted data.
Graph Algorithms			
Depth-First Search (DFS)	$O(V + E)$	$O(V)$	Exploring all vertices in a graph.
Breadth-First Search (BFS)	$O(V + E)$	$O(V)$	Shortest path in unweighted graphs.
Dijkstra's Algorithm	$O((V + E)\log V)$	$O(V)$	Shortest path in weighted graphs.
Bellman-Ford Algorithm	$O(V \times E)$	$O(V)$	Shortest path with negative weights.
Floyd-Warshall Algorithm	$O(V)^3$	$O(V)^2$	All-pairs shortest paths.

Algorithm	Time Complexity	Space Complexity	Use Case
Tree Traversal Algorithms			
Inorder Traversal	$O(n)$	$O(h)$	Traverse BST in non decreasing order.
Preorder Traversal	$O(n)$	$O(h)$	Traverse tree in root-first order.

Postorder Traversal	$O(n)$	$O(h)$	Traverse tree in child-first order.
Level Order Traversal (BFS)	$O(n)$	$O(w)$	Traverse level by level.
Dynamic Programming			
Longest Common Subsequence (LCS)	$O(m \times n)$	$O(m \times n)$	Sequence alignment.
Longest Increasing Subsequence (LIS)	, $O(n\ )^2$ $O(n \log n)$	$O(n)$	Finding the longest increasing subsequence.
Knapsack Problem	$O(n \times W)$	$O(n \times W)$	Optimal selection of items.
Fibonacci Series (Top Down)	$O(n)$	$O(n)$	Compute Fibonacci numbers efficiently.
Matrix Chain Multiplication	$O(n\ )^3$	$O(n\ )^2$	Optimal parenthesization of matrix products.
Advanced Algorithms			
Segment Tree	$O(\log n)$ (query/update)	$O(n)$	Efficient range queries and updates.
Fenwick Tree (Binary Indexed Tree)	$O(\log n)$ (query/update)	$O(n)$	Efficient prefix sum queries.
Mo's Algorithm	$O((n + q)\ n)$	$O(n)$	Efficient range query processing.

Algorithm	Time Complexity	Space Complexity	Use Case
Kruskal's Algorithm (MST)	$O(E \log E)$	$O(V\ )$	Minimum spanning tree in a graph.
Prim's Algorithm (MST)	$O(E \log V\ )$	$O(V\ )$	Minimum spanning tree in a graph.

Union-Find (Disjoint Set)	$O(\alpha(n))$	$O(n)$	Efficient set operations.
Trie (Prefix Tree)	$O(L)$ (insert/search)	$O(L \times n)$	Fast retrieval of strings.
Rabin-Karp Algorithm (String Matching)	$O(n + m)$ (average)	$O(1)$	Pattern searching in a string.

This table provides a structured overview of the algorithms, their complexities, and typical use cases, making it easier to reference the information quickly.

Data Structures

Arrays

Search: $O(1)$

Insert: $O(n)$

Lookup: $O(n)$

Delete: $O(n)$

Space Complexity: $O(n)$

Pros:

Fast lookups

Fast push/pop operations

Ordered

Cons:

Slow inserts and deletes

Fixed size (if using static array)

Linked Lists

Prepend: $O(1)$

Append:

Lookup

:
$O(n)$
$O(1)$
Delete:
$O(n)$
$O(n)$
Insert:
Space Complexity:
Pros:
$O(n)$

Fast insertion and deletion

Flexible size

Cons:

Slow lookup

More memory overhead due to pointers

Doubly Linked Lists

Prepend:
$O(1)$
$O(n)$
$O(1)$
Insert:
Append:
$O(n)$
$O(1)$
Lookup:

Delete: (once the node is found)

Space Complexity:
Pros:
$O(n)$

Fast insertions/deletions from both ends

Can traverse in both directions

Cons:

- More memory usage due to extra pointer
- Slower lookups

Stacks (LIFO)

Push: $O(1)$

Pop: $O(1)$

Peek: $O(1)$

Lookup: $O(n)$

Space Complexity: $O(n)$

Pros:

- Fast operations
- Simple and easy to implement

Cons:

- Slow lookup

Queues (FIFO)

Enqueue: $O(1)$

Dequeue: $O(1)$

Peek: $O(1)$

Lookup: $O(n)$

Space Compl

exity: d

Pros: **Cons:**

Fast operations Slow lookup $O(n)$

Ordere

Priority Queues (using Binary Heap)

Insert: $O(\log n)$

Delete (Extract Max/Min): $O(\log n)$

Peek (Find Max/Min): $O(1)$

Space Complexity: $O(n)$

Pros:

Fast access to highest/lowest priority element

Cons:

Slower operations than simple queues

Hash Tables

Searc h: $O(1)$

Insert : $O(1)$

Delete: $O(1)$ $O(n)$

$O(1)$

Lookup: (Average Case) or (Worst Case, if collisions are high)

Space Complexity: $O(n)$

Pros:

Extremely fast lookups, inserts, and deletes

Flexible keys

Cons:

Unordered

Performance depends on a good hash function

Potential for hash collisions, leading to O(n) operations in the worst case

Binary Search Trees (BST)

$O(\log n)$ $O(n)$

Search: (Average Case), (Worst Case, if unbalanced) $O(\log n)$ $O(n)$

Insert: (Average Case), (Worst Case)

$O(\log n)$ $O(n)$

Delete: (Average Case), (Worst Case)

Space Complexity: $O(n)$

Ordered structure

Pros:

- Efficient searching, insertion, and deletion

Cons:

- Can become unbalanced, leading to poor performance

Balanced Binary Search Trees (e.g., AVL, Red-Black Tree)

- **Search:** $O(\log n)$
- **Insert:** $O(\log n)$
- **Delete:** $O(\log n)$

Space Complexity: $O(n)$

Pros:

- Guaranteed $O(\log n)$ operations
- Self-balancing

Cons:

- More complex to implement
- Higher memory overhead

Heaps (Min-Heap, Max-Heap)

- **Insert:** $O(\log n)$
- **Delete (Extract Max/Min):** $O(\log n)$
- **Peek (Find Max/Min):** $O(1)$

Space Complexity:

Pros:

$O(n)$

Efficient for priority queue operations

Easy to implement

Cons:

Not suitable for searching specific elements

Not sorted

Fenwick Tree (Binary Indexed Tree)

Update: $O(\log n)$

Space Complexity:

Prefix Sum Query:

Pros:

$O(\log n)$ $O(n)$

Efficient for cumulative frequency tables

Easy to implement

Cons:

Limited to operations like range sum, not general-purpose

Segment Tree

Range Query: $O(\log n)$

Update: $O(\log n)$

Space Complexity:

Pros:

$O(n)$

Handles range queries efficiently

Flexible for various operations (sum, min, max)

Cons:

Complex implementation

Higher memory usage compared to Fenwick Tree

Graphs

Adjacency List Representation:

Space Complexity: $O(V + E)$ $O(V + E)$

Search: using DFS/BFS **Adjacency**

Matrix Representation:

Space Complexity: $O(V)^2$

$O(V)^2$

Search: using DFS/BFS

Pros:

Useful for modeling relationships between entities

Cons:

High space complexity, especially with adjacency matrix

Algorithms

Sorting Algorithms

1. **Bubble Sort**

Time Complexity: $O(n^2)$

Space Complexity:

2. **Selection Sort**

$O(1)$

Complexity:

$O(n^2)$

Space Complexity:

3. **Insertion Sort**

$O(1)$

Time

$O(n)$ $O(n^2)$

Time Complexity: (Best Case), (Worst Case)

Space Complexity:

4. **Merge Sort**

$O(1)$

Time

Complexity:

$O(n \log n)$

Space Complexity:

5. **Quick Sort**

$O(n)$

$O(n \log n)$ $O(n^2)$

Time Complexity: (Average Case), (Worst Case)

Space Complexity:

6. **Heap Sort**

$O(\log n)$

Complexity:

7. **Counting Sort**

$O(1)$

Time Complexity:

$O(n \log n)$

Time Complexity:

$O(n + k)$

$O(k)$ k

Space

Space Complexity: (where is the range of input values) 8.

Radix Sort

$O(nk)$ k

Time Complexity: (where is the number of digits)

Space Complexity:

9. **Bucket Sort**

$O(n + k)$

Time Complexity:

$O(n + k)$

Space

Complexity: $O(\log n)$

Space Complexity:

2. **Linear Search**

Search

$O(1)$

Algorithms 1.

Time Complexity:

$O(n)$

Binary Search

Space Complexity:

$O(n + k)$

$O(1)$

Time Complexity:

3. **Breadth-First Search (BFS)**

Time Complexity: $O(V + E)$

Space Complexity: $O(V)$

4. **Depth-First Search (DFS)**

Time Complexity: $O(V + E)$

Space Complexity: $O(V)$

5. **Dijkstra's Algorithm**

$O((V + E)\log V)$

Time Complexity:

Space Complexity: $O(V)$

6. *A Search**

$O((V + E)\log V)$

Time Complexity:

Space Complexity: $O(V)$

7. **Bellman-Ford Algorithm**

Time Complexity: $O(V \times E)$

Space Complexity: (

O(V))

8. **Floyd-Warshall Algorithm**

Time Complexity: $O(V)^3$ **Space Complexity:** $O(V)^2$

Tree Traversal Algorithms

1. **Inorder Traversal**

Time Complexity: $O(n)$ $O(h)$ h

Space Complexity: (where is the height of the tree) 2.

Preorder Traversal

Time Complexity: $O(n)$

Space Complexity: $O(h)$

3. **Postorder Traversal**

Time Complexity: $O(n)$

Space Complexity: $O(h)$

4. **Level Order Traversal (BFS)**

Time Complexity: $O(n)$ $O(w)$ w

Space Complexity: (where is the maximum width of the tree)

Dynamic Programming

1. **Longest Common Subsequence (LCS)**

Time Complexity: $O(m \times n)$ **Space Complexity:** $O(m \times n)$

2. **Longest Increasing Subsequence (LIS)**

$O(n)^2$ $O(n \log n)$

Time Complexity: (or with a binary search approach)

Space Complexity: $O(n)$

3. **Knapsack Problem**

Time Complexity: $O(n \times W)$ (where W is the capacity of the knapsack)

Space Complexity: $O(n \times W)$

4. **Fibonacci Series (Top-Down Approach)**

   Time Complexity: $O(n)$ **Space Complexity:** $O(n)$

5. **Matrix Chain Multiplication**

   Time Complexity: $O(n^3)$ **Space Complexity:** $O(n^2)$

Other Advanced Algorithms

1. **Segment Tree**

   Range Query: $O(\log n)$

   Update: $O(\log n)$

   Space Complexity: $O(n)$

   Use Case: Efficient range queries and updates (sum, min, max).

2. **Fenwick Tree (Binary Indexed Tree)**

   Update: $O(\log n)$

   Prefix Sum Query: $O(\log n)$

   Space Complexity: $O(n)$

   Use Case: Efficient range sum queries.

3. **Mo's Algorithm**

   Time Complexity: $O((n + q)\sqrt{n})$ (where q is the number of queries)

   Space Complexity: $O(n)$

   Use Case: Efficient answering of range queries offline.

4. **Kruskal's Algorithm (Minimum Spanning Tree)**
   - **Time Complexity:** $O(E \log E)$
   - **Space Complexity:** $O(V)$
   - **Use Case:** Finding a minimum spanning tree in a graph.
5. **Prim's Algorithm (Minimum Spanning Tree)**
   - **Time Complexity:** $O(E \log V)$
   - **Space Complexity:** $O(V)$
   - **Use Case:** Finding a minimum spanning tree in a graph.
6. **Union-Find (Disjoint Set)**
   - **Find:** $O(\alpha(n))$ (Inverse Ackermann function)
   - **Union:** $O(\alpha(n))$
   - **Space Complexity:** $O(n)$
   - **Use Case:** Efficient set operations.
7. **Trie (Prefix Tree)**
   - **Insert:** $O(L)$ (where L is the length of the word)
   - **Search:** $O(L)$
   - **Space Complexity:** $O(L \times n)$
   - **Use Case:** Fast retrieval of keys in a large dataset of strings.
8. **Rabin-Karp Algorithm (String Matching)**
   - **Time Complexity:** $O(n + m)$ (Average Case)
   - **Space Complexity:** $O(1)$
   - **Use Case:** Pattern searching in a string.