## Introduction

An algorithm is a set of steps to accomplish a task.

Knowledge of algorithm help use to get desired result faster by applying appropriate algorithm.

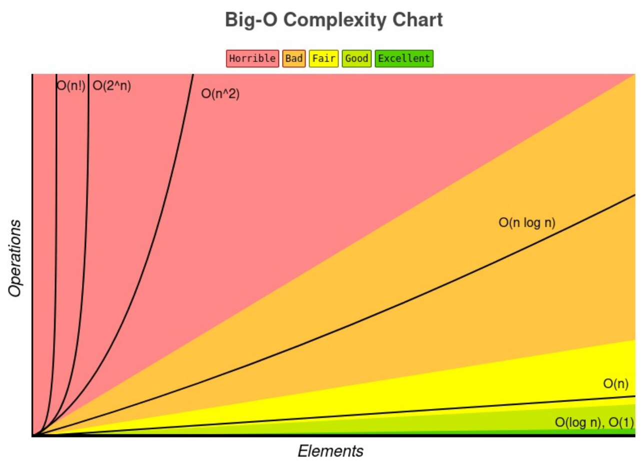
Algorithm should be efficient in solving problems. Efficiency is measured in two parameters. First is Time-Complexity, how quick result can be provided by algorithm. Second is Space-Complexity, how much ram is used by the algorithm to provide the result.

- Time-Complexity is represented by function T(n) – time required versus the input size n.
- Space-Complexity is represented by function S(n) – space required versus the input size n.

## Big-O Notation

*“Big O notation is a mathematical notation that describes the limiting behavior of a function when the argument tends towards a particular value or infinity. It is a member of a family of notations invented by Paul Bachmann, Edmund Landau, and others, collectively called Bachmann–Landau notation or asymptotic notation.”*

— Wikipedia’s definition of Big O notation



Common Data Operation Chart

| **Data Structure** | **Time Complexity** | | | | | | | | **Space Complexity** |
|---|---|---|---|---|---|---|---|---|---|
| | **Average** | | | | **Worst** | | | | **Worst** |
| | **Access** | **Search** | **Insertion** | **Deletion** | **Access** | **Search** | **Insertion** | **Deletion** | |
| Array | Θ(1) | Θ(n) | Θ(n) | Θ(n) | O(1) | O(n) | O(n) | O(n) | O(n) |
| Stack | Θ(n) | Θ(n) | Θ(1) | Θ(1) | O(n) | O(n) | O(1) | O(1) | O(n) |
| Queue | Θ(n) | Θ(n) | Θ(1) | Θ(1) | O(n) | O(n) | O(1) | O(1) | O(n) |
| Singly-Linked List | Θ(n) | Θ(n) | Θ(1) | Θ(1) | O(n) | O(n) | O(1) | O(1) | O(n) |

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| Doubly-Linked List | Θ(n) | Θ(n) | Θ(1) | Θ(1) | O(n) | O(n) | O(1) | O(1) | O(n) |
| Skip List | Θ(log(n)) | Θ(log(n)) | Θ(log(n)) | Θ(log(n)) | O(n) | O(n) | O(n) | O(n) | O(n log(n)) |
| Hash Table | N/A | Θ(1) | Θ(1) | Θ(1) | N/A | O(n) | O(n) | O(n) | O(n) |
| Binary Search Tree | Θ(log(n)) | Θ(log(n)) | Θ(log(n)) | Θ(log(n)) | O(n) | O(n) | O(n) | O(n) | O(n) |
| Cartesian Tree | N/A | Θ(log(n)) | Θ(log(n)) | Θ(log(n)) | N/A | O(n) | O(n) | O(n) | O(n) |
| B-Tree | Θ(log(n)) | Θ(log(n)) | Θ(log(n)) | Θ(log(n)) | O(log(n)) | O(log(n)) | O(log(n)) | O(log(n)) | O(n) |
| Red-Black Tree | Θ(log(n)) | Θ(log(n)) | Θ(log(n)) | Θ(log(n)) | O(log(n)) | O(log(n)) | O(log(n)) | O(log(n)) | O(n) |
| Splay Tree | N/A | Θ(log(n)) | Θ(log(n)) | Θ(log(n)) | N/A | O(log(n)) | O(log(n)) | O(log(n)) | O(n) |
| AVL Tree | Θ(log(n)) | Θ(log(n)) | Θ(log(n)) | Θ(log(n)) | O(log(n)) | O(log(n)) | O(log(n)) | O(log(n)) | O(n) |
| KD Tree | Θ(log(n)) | Θ(log(n)) | Θ(log(n)) | Θ(log(n)) | O(n) | O(n) | O(n) | O(n) | O(n) |

## Array Sorting Algorithm

| **Algorithm** | **Time Complexity** | | | **Space Complexity** |
|---|---|---|---|---|
| | **Best** | **Average** | **Worst** | **Worst** |
| Quicksort | Ω(n log(n)) | Θ(n log(n)) | O(n^2) | O(log(n)) |
| Mergesort | Ω(n log(n)) | Θ(n log(n)) | O(n log(n)) | O(n) |
| Timsort | Ω(n) | Θ(n log(n)) | O(n log(n)) | O(n) |
| Heapsort | Ω(n log(n)) | Θ(n log(n)) | O(n log(n)) | O(1) |
| Bubble Sort | Ω(n) | Θ(n^2) | O(n^2) | O(1) |
| Insertion Sort | Ω(n) | Θ(n^2) | O(n^2) | O(1) |
| Selection Sort | Ω(n^2) | Θ(n^2) | O(n^2) | O(1) |
| Tree Sort | Ω(n log(n)) | Θ(n log(n)) | O(n^2) | O(n) |
| Shell Sort | Ω(n log(n)) | Θ(n(log(n))^2) | O(n(log(n))^2) | O(1) |
| Bucket Sort | Ω(n+k) | Θ(n+k) | O(n^2) | O(n) |
| Radix Sort | Ω(nk) | Θ(nk) | O(nk) | O(n+k) |
| Counting Sort | Ω(n+k) | Θ(n+k) | O(n+k) | O(k) |

| | | | | |
|---|---|---|---|---|
| Cubesort | `Ω(n)` | `Θ(n log(n))` | `O(n log(n))` | `O(n)` |

## Complexity analysis of algorithms

1. **Worst Case Complexity**: It is the complexity of solving the problem for the worst input of size n. It provides the upper bound for the algorithm. This is most used analysis used.
2. **Average Case Complexity**: It is the complexity of solving the problem on an average. We calculate the time for all possible inputs and then take an average on it.
3. **Best Case Complexity**: It is the complexity of solving the problem for the best input of size n.

## Constant Time (1)

An algorithmis said to run in constant time if the output is produced in constant time regardless of the input size.

Examples:

1. Accessing n$^{th}$ element of an Array.
2. Push and pop of a stack.
3. Add and remove of a queue.
4. Accessing an element of Hash-Table

## Linear Time O(n)

An algorithm is said to run in linear time if the execution time of the algorithm is directly proportional to the input size.

Examples:

1. Array operations like search element, find min, find max, etc.
2. Linked list operation like traversal, find min, find max, etc.

## Logarithmic Time O(log n)

An Algorithm is said to run in logarithmic time if the execution time of the algorithm is proportional to the logarithm of the input size. Each step of an algorithm, a significant portion (eg. half portion) of the input is pruned / rejected out without traversing it.

Example: Binary search algorithm, we will read about this algorithm in this book.

## N-LogN Time O(nlog(n))

An algorithm is said to run in nlogn time if the execution time of an algorithms is proportional to the product of input size and logarithm of the input size. In these algorithm, each the input is divided into half (or some proportion) and each time the input is divide into half (or some proportion) and each portion is processed independently.

Examples:

1. Merge-Sort
2. Quick-Sort
3. Heap-Sort

## Quadratic time O($n^2$)

An algorithm is said to run is in quadratic time if the execution time of the an algorithm is proportional to the square of the input size. In these algorithm each element is compared with all the other elements.

Examples:

1. Bubble-Sort
2. Selection-Sort
3. Insertion-Sort

## Exponential Time O($2^n$)

In these algorithms, all possible subset of elements of input date are generated.

### Factorial Time O(n!)

In these algorithm, all possible permutation of elements of input date are generated.

## Deriving the Runtime Function of an Algorithm

### Constants

Each statement takes a constant time to run. Time Complexity is O(1).

### Loops

The running time of a loop is a product of running time of product of running time of the statement inside a loop and number of iterations in loop. Time Complexity is O(n).

### Nested Loop

The running time of a nested loop is a product of running time of the statement inside loop multiplied by a product of the size of all the loops. Time Complexity is $O(n^c)$. Where c is number of loops.

### Consecutive Statement

Just add the running times of all the consecutive statements.

### If-Else Statement

Consider the running time of the larger of if block or else block and ignores the other block.

### Logarithmic Statement

If each iteration the input size is decreased by a constant factor. Time Complexity = O(log n)