Big O Cheat Sheet

Introduction:

Welcome to the "Big-O Complexity Cheat Sheet" repository! This cheat sheet is designed to provide a quick reference guide for understanding the time and space complexity of various algorithms and data structures. As a developer, you will often encounter problems that require efficient solutions, and having a solid understanding of Big O notation is essential for writing performant code.

In this repository, you will find a comprehensive list of common algorithms and data structures, along with their time and space complexities. This will serve as a handy resource for developers, computer science students, and anyone interested in learning more about the fundamental concepts of computer science.

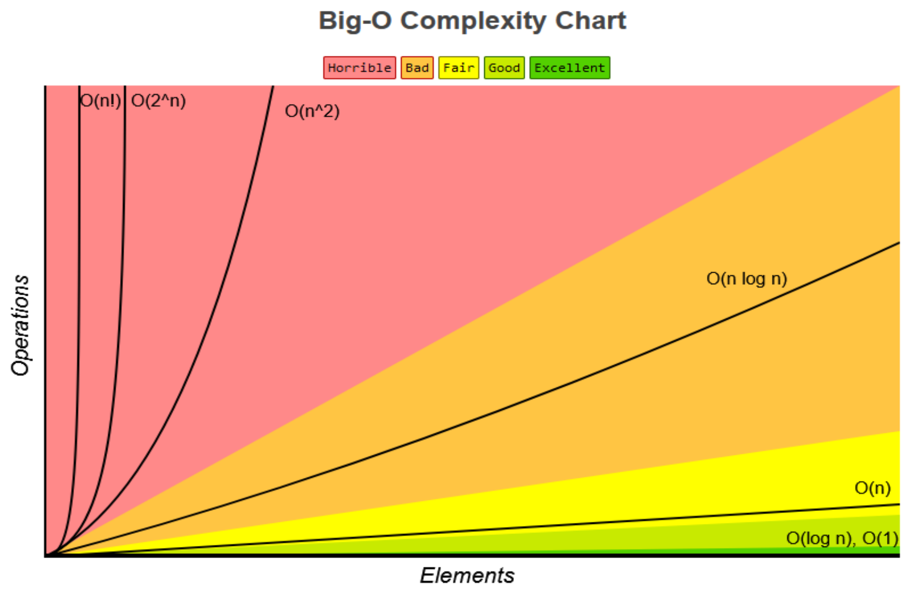
Whether you are preparing for a technical interview or simply want to improve your knowledge of algorithmic complexities, this cheat sheet is the perfect starting point for your journey.

Table of Contents:

TL;DR:

A very useful complexity chart by:

bigocheatsheet.com



Complexity	Description	Example
Constant Time	`O(1)` - Constant, regardless of input size	Accessing an array element by index

Complexity	Description	Example
Logarithmic Time	`O(log n)` - Increases logarithmically with input size	Binary search
Linear Time	`O(n)` - Increases linearly with input size	Iterating through an array (no loops)
Linearithmic Time	`O(n log n)` - Linearly with input size * logarithmic factor	Merge sort
Quadratic Time	`O(n²)` - Increases quadratically with input size	Nested loops (2 loops)
Cubic Time	`O(n³)` - Increases cubically with input size	Triple nested loops (3 loops)
Exponential Time	`O(2^n)` - Increases exponentially with input size	Naive recursive Fibonacci
Factorial Time	`O(n!)` - Increases factorially with input size	Generating all possible permutations

---

Time Complexity:

- **O(1): Constant time.**
    - No matter how large the input, the algorithm will always take the same amount of time to complete.
    - Example: Accessing an element in an array by index.

```
def get_first(my_list):
    return my_list[0]
```

- **O(log n): Logarithmic time.**
    - The input size has a logarithmic effect on the running time of the algorithm.
    - Example: Binary search.

```
# Binary search
```

- **O(n): Linear time.**
    - The input size has a linear effect on the running time of the algorithm.
    - Example: Iterating through an array.

```
# Iterating through an array
def print_all_elements(my_list):
    for element in my_list:
        print(element)
```

- **O(n log n): Log-linear time.**
    - The input size has a log-linear effect on the running time of the algorithm.
    - Example: Merge sort.

```
# Merge sort
```

- **O(n^2): Quadratic time.**
    - The input size has a quadratic effect on the running time of the algorithm.
    - Example: 2 nested for loops.

```
# 2 nested for loops
def print_all_possible_ordered_pairs(my_list):
    for first_item in my_list: # O(n)
        for second_item in my_list: # O(n)
            print(first_item, second_item)
```

- **O(n^3): Cubic time.**
    - The input size has a cubic effect on the running time of the algorithm.
    - Example: Iterating through a 3D array, or 3 nested for loops.

```
# 3 nested for loops -- Also use as last resort for 3D arrays
def naive_matrix_mult(A, B):
    rows_A, cols_A = len(A), len(A[0])
    rows_B, cols_B = len(B), len(B[0])
    if cols_A != rows_B:
        raise ValueError("Matrices cannot be multiplied.")

    C = [[0 for _ in range(cols_B)] for _ in range(rows_A)]
    for i in range(rows_A):
        for j in range(cols_B):
            for k in range(cols_A):
                C[i][j] += A[i][k] * B[k][j]
```

- **O(2^n): Exponential time.**
    - The input size has an exponential effect on the running time of the algorithm.
    - Example: Iterating through all subsets of a set.

```
# Iterating through all subsets of a set
def print_all_subsets(my_set):
    all_subsets = [[]]
    for element in my_set:
        for subset in all_subsets:
            all_subsets = all_subsets + [list(subset) + [element]]
    return all_subsets

# or

def naive_fibonacci(n):
    if n <= 1:
        return n
    return naive_fibonacci(n - 1) + naive_fibonacci(n - 2)
```

- **O(n!): Factorial time.**
    - The input size has a factorial effect on the running time of the algorithm.
    - Example: Iterating through all permutations of a set.

```
# Iterating through all permutations of a set
def generate_permutations(arr, start=0):
    if start == len(arr) - 1:
        print(arr)
    for i in range(start, len(arr)):
        arr[start], arr[i] = arr[i], arr[start]
        # Recurse
        generate_permutations(arr, start + 1)
        arr[start], arr[i] = arr[i], arr[start]
```

Space Complexity:

- **O(1): Constant space.**
    - The algorithm uses a `constant` amount of memory, regardless of the `input size`.
    - Example: `Iterating` through an `array`.

```
def print_all_elements(my_list):
    for element in my_list:
        print(element)
```

- **O(n): Linear space.**
    - The algorithm uses `linear` amount of memory, proportional to the `input size`.
    - Example: `Iterating` through an `array` and storing the values in a `hash table`.

```
# O(n) space - Storing all elements in a hash table
def reverse_list(arr):
    reversed_arr = []
    for i in range(len(arr) - 1, -1, -1):
        reversed_arr.append(arr[i])
    return reversed_arr
```

- **O(n^2): Quadratic space.**
    - The algorithm uses `quadratic` amount of memory, proportional to the `input size`.
    - Example: `Iterating` through an `array` and storing the values in a `2D array`.

```
# O(n^2) space - Storing all elements in a 2D array
def create_identity_matrix(n):
    identity = [[0 for _ in range(n)] for _ in range(n)]
    for i in range(n):
        identity[i][i] = 1
    return identity
```

- **O(2^n): Exponential space.**
    - The algorithm uses `exponential` amount of memory, proportional to the `input size`.
    - Example: `Iterating` through all subsets of a set.

```
# Exponential Space - O(2^n)
```

```
def power_set(arr):
    result = [[]]
    for item in arr:
        result += [subset + [item] for subset in result]
    return result
```

Common Data Structures:

- ### Array
    - **Time** Complexity:
        - **Access**: `O(1)`
        - **Search**: `O(n)`
        - **Insertion**: `O(n)`
        - **Deletion**: `O(n)`
    - **Space** Complexity: `O(n)`
    - **Description**: An `array` is a data structure that stores a collection of elements. Each element is identified by an index, or key. Arrays are used to store a collection of data, but they are not as flexible as other data structures such as linked lists, stacks, and queues. Arrays are best used when you know exactly what data you need to store, and how you will be accessing it.

- ### Linked List
    - **Time** Complexity:
        - **Access**: `O(n)`
        - **Search**: `O(n)`
        - **Insertion**: `O(1)`
        - **Deletion**: `O(1)`
    - **Space** Complexity: `O(n)`
    - **Description**: A `linked list` is a data structure that stores a collection of elements. Each element is a separate object that contains a `pointer or a link to the next object in that list`. Linked lists are **best** used when you need to *add* or *remove* elements from the beginning of the list.

- ### Stack

    - **Time** Complexity:
        - **Access**: `O(n)`
        - **Search**: `O(n)`
        - **Insertion**: `O(1)`
        - **Deletion**: `O(1)`
    - **Space** Complexity: `O(n)`
    - **Description**: A `stack` is a data structure that stores a collection of elements. A `stack` is a `LIFO` (Last In First Out) data structure, meaning that the last element added to the stack will be the first one to be removed. Stacks are best used when you need to *add* or *remove* elements from the beginning of the list.

- ## Queue

    - **Time** Complexity:
        - **Access**: `O(n)`
        - **Search**: `O(n)`
        - **Insertion**: `O(1)`
        - **Deletion**: `O(1)`
    - **Space** Complexity: `O(n)`
    - **Description**: A `queue` is a data structure that stores a collection of elements. A `queue` is a `FIFO` (First In First Out) data structure, meaning that the first element added to the queue will be the first one to be removed. Queues are best used when you need to *add* or *remove* elements from the end of the list.

- ## Hash Table

    - **Time** Complexity:
        - **Access**: `O(1)`
        - **Search**: `O(1)`
        - **Insertion**: `O(1)`
        - **Deletion**: `O(1)`
    - **Space** Complexity: `O(n)`
    - **Description**: A `hash table` is a data structure that stores a collection of elements. A `hash table` is a `key-value` data structure, meaning that each element is identified by a `key`. A `hash function` is used to compute the index at which an element will be stored. Hash tables are best used when you need to *add*, *remove*, or *access* elements in a collection.