Complexity Analysis Cheat Sheet

1. Introduction to Complexity Analysis

Complexity analysis is a crucial aspect of computer science that helps us evaluate the efficiency of algorithms. It provides a theoretical framework to estimate the resources (primarily time and space) an algorithm requires to solve a problem as the input size grows. Understanding complexity analysis allows developers to choose the most efficient algorithms for a given task, especially when dealing with large datasets or performance-critical applications.

Unlike empirical testing, which measures an algorithm's performance on a specific machine with a specific input, complexity analysis provides a machine-independent and input-size-agnostic measure of efficiency. It focuses on the growth rate of an algorithm's resource consumption as the input size (`n`) increases.

Why is Complexity Analysis Important?

- **Predicting Performance:** It helps predict how an algorithm will perform with larger inputs without actually running it.
- **Comparing Algorithms:** It provides a standardized way to compare the efficiency of different algorithms for the same problem.
- **Identifying Bottlenecks:** It helps identify parts of an algorithm that consume the most resources, guiding optimization efforts.
- **Scalability:** It helps understand how an algorithm scales with increasing input size, which is critical for designing robust and efficient systems.

Complexity analysis typically focuses on two main types of resources:

1. **Time Complexity:** The amount of time an algorithm takes to complete its execution as a function of the input size.
2. **Space Complexity:** The amount of memory space an algorithm requires to complete its execution as a function of the input size.

Both time and space complexity are usually expressed using **Big O Notation**, which describes the upper bound or worst-case scenario of an algorithm's resource consumption.

2. Time Complexity

Time complexity quantifies the amount of time taken by an algorithm to run as a function of the length of the input. It's not about measuring the actual execution time in seconds, but rather counting the number of elementary operations an algorithm performs. These elementary operations can include comparisons, assignments, arithmetic operations, function calls, etc. The goal is to understand how the number of operations grows with the input size.

Factors Affecting Time Complexity:

- **Input Size (n):** The primary factor influencing time complexity. As `n` increases, the number of operations typically increases.
- **Operations per Input:** The number of basic operations performed for each unit of input.
- **Algorithm Structure:** Loops, nested loops, recursive calls, and conditional statements all impact the number of operations.

Common Time Complexities (and their implications):

Here's a breakdown of common time complexities, ordered from most efficient to least efficient for large inputs:

- **O(1) - Constant Time:** The execution time remains constant regardless of the input size. This is the most efficient complexity.
    - **Example:** Accessing an element in an array by its index.
- **O(log n) - Logarithmic Time:** The execution time grows logarithmically with the input size. This is very efficient, as the time taken increases very slowly with `n`.
    - **Example:** Binary search in a sorted array.

- **O(n) - Linear Time:** The execution time grows linearly with the input size. If the input doubles, the time taken roughly doubles.
    - **Example:** Traversing a list or array, finding an element in an unsorted array.
- **O(n log n) - Linearithmic Time:** The execution time grows proportionally to `n` times the logarithm of `n`. This is common in efficient sorting algorithms.
    - **Example:** Merge Sort, Quick Sort.
- **O(n^2) - Quadratic Time:** The execution time grows quadratically with the input size. If the input doubles, the time taken quadruples. This is often seen with nested loops.
    - **Example:** Bubble Sort, Insertion Sort, selection sort.
- **O(2^n) - Exponential Time:** The execution time doubles with each additional element in the input. This is highly inefficient and generally impractical for even moderately sized inputs.
    - **Example:** Naive recursive calculation of Fibonacci numbers, solving the Traveling Salesperson Problem using brute force.
- **O(n!) - Factorial Time:** The execution time grows extremely rapidly with the input size. This is the least efficient and typically only feasible for very small inputs.
    - **Example:** Brute-force solutions to problems like the Traveling Salesperson Problem.

C++ Code Examples for Time Complexity:

O(1) - Constant Time

```cpp
#include <iostream>
#include <vector>

int main() {
    std::vector<int> arr = {1, 2, 3, 4, 5};
    // Accessing an element by index is O(1)
    std::cout << "First element: " << arr[0] << std::endl;
    return 0;
}
```

O(n) - Linear Time

```cpp
#include <iostream>
#include <vector>

void printElements(const std::vector<int>& arr) {
    for (int i = 0; i < arr.size(); ++i) { // Loop runs 'n' times
        std::cout << arr[i] << " ";
    }
    std::cout << std::endl;
}

int main() {
    std::vector<int> data = {10, 20, 30, 40, 50};
    printElements(data);
    return 0;
}
```

O(n^2) - Quadratic Time

```cpp
#include <iostream>
#include <vector>

void printPairs(const std::vector<int>& arr) {
    for (int i = 0; i < arr.size(); ++i) { // Outer loop runs 'n' times
        for (int j = 0; j < arr.size(); ++j) { // Inner loop runs 'n' times for
each outer loop iteration
            std::cout << "(" << arr[i] << ", " << arr[j] << ") ";
        }
        std::cout << std::endl;
    }
}

int main() {
    std::vector<int> data = {1, 2, 3};
    printPairs(data);
    return 0;
}
```

O(log n) - Logarithmic Time (Binary Search Example)

```cpp
#include <iostream>
#include <vector>
#include <algorithm> // For std::sort

int binarySearch(const std::vector<int>& arr, int target) {
    int low = 0;
    int high = arr.size() - 1;

    while (low <= high) {
        int mid = low + (high - low) / 2;
        if (arr[mid] == target) {
            return mid; // Found
        } else if (arr[mid] < target) {
            low = mid + 1;
        } else {
            high = mid - 1;
        }
    }
    return -1; // Not found
}

int main() {
    std::vector<int> sortedData = {10, 20, 30, 40, 50, 60, 70, 80, 90, 100};
    int target = 70;
    int index = binarySearch(sortedData, target);

    if (index != -1) {
        std::cout << "Target " << target << " found at index: " << index <<
std::endl;
    } else {
        std::cout << "Target " << target << " not found." << std::endl;
    }
    return 0;
}
```

3. Space Complexity

Space complexity quantifies the amount of memory space an algorithm requires to complete its execution as a function of the input size. It includes both the auxiliary space (temporary space used by the algorithm during execution) and the space taken by the input itself. However, typically, when discussing space complexity, we focus on the auxiliary space.

Factors Affecting Space Complexity:

- **Input Size (n):** The size of the input data can directly affect the memory required.

- **Auxiliary Data Structures:** Any additional data structures created by the algorithm (e.g., arrays, stacks, queues, hash maps).
- **Recursion Stack Space:** For recursive algorithms, the space used by the call stack can contribute significantly to space complexity.

Common Space Complexities:

- **O(1) - Constant Space:** The memory usage remains constant regardless of the input size. This is the most memory-efficient.
    - **Example:** Swapping two variables.
- **O(log n) - Logarithmic Space:** The memory usage grows logarithmically with the input size.
    - **Example:** Recursive calls in binary search (due to call stack).
- **O(n) - Linear Space:** The memory usage grows linearly with the input size.
    - **Example:** Creating a copy of an array, using an auxiliary array of the same size as the input.
- **O(n^2) - Quadratic Space:** The memory usage grows quadratically with the input size.
    - **Example:** Creating a 2D array (matrix) of size `n x n`.

C++ Code Examples for Space Complexity:

O(1) - Constant Space

```cpp
#include <iostream>

void swap(int& a, int& b) {
    int temp = a; // Only one temporary variable is used, regardless of a or
b's value
    a = b;
    b = temp;
}

int main() {
    int x = 5, y = 10;
    std::cout << "Before swap: x = " << x << ", y = " << y << std::endl;
    swap(x, y);
    std::cout << "After swap: x = " << x << ", y = " << y << std::endl;
    return 0;
}
```

O(n) - Linear Space

```cpp
#include <iostream>
#include <vector>

std::vector<int> createCopy(const std::vector<int>& original) {
    std::vector<int> copy(original.size()); // Creates a new vector of size 'n'
    for (int i = 0; i < original.size(); ++i) {
        copy[i] = original[i];
    }
    return copy;
}

int main() {
    std::vector<int> myVector = {1, 2, 3, 4, 5};
    std::vector<int> copiedVector = createCopy(myVector);

    std::cout << "Original vector: ";
    for (int val : myVector) {
        std::cout << val << " ";
    }
    std::cout << std::endl;

    std::cout << "Copied vector: ";
    for (int val : copiedVector) {
        std::cout << val << " ";
    }
    std::cout << std::endl;
    return 0;
}
```

4. 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. In complexity analysis, it's used to classify algorithms according to how their running time or space requirements grow as the input size grows. It specifically describes the **upper bound** of the growth rate, representing the worst-case scenario.

Key Characteristics of Big O Notation:

- **Worst-Case Scenario:** Big O notation typically represents the upper bound of an algorithm's performance, meaning the maximum time or space it could take for any given input size.
- **Asymptotic Analysis:** It focuses on the behavior of the algorithm as the input size `n` approaches infinity, ignoring constant factors and lower-order terms.
- **Growth Rate:** It describes how quickly the running time or space requirements grow relative to the input size.

Rules for Big O Notation:

1. **Ignore Constant Factors:** `O(2n)` is `O(n)`, `O(5n^2)` is `O(n^2)`. Constants don't significantly affect the growth rate for large `n`.
2. **Ignore Lower-Order Terms:** `O(n^2 + n)` is `O(n^2)`. For large `n`, the `n^2` term dominates the `n` term.
3. **Combine Different Steps:** If an algorithm performs several steps sequentially, the overall complexity is determined by the step with the highest order.
    - Example: A function with an `O(n)` loop followed by an `O(n^2)` nested loop will have an overall complexity of `O(n^2)`.
4. **Multiply for Nested Operations:** For nested loops or recursive calls, multiply the complexities.
    - Example: A loop running `n` times, and inside it another loop running `n` times, results in `O(n * n) = O(n^2)`.

Visualizing Growth Rates:

(Refer to the `time_complexity.png` and `space_complexity.png` images for visual representation of these growth rates.)

5. Analyzing Code Complexity (C++ Examples)

Let's analyze the complexity of some C++ code snippets.

Example 1: Sum of Array Elements

```cpp
#include <iostream>
#include <vector>

int sumArray(const std::vector<int>& arr) {
    int sum = 0; // O(1) - one assignment
    for (int i = 0; i < arr.size(); ++i) { // Loop runs 'n' times
        sum += arr[i]; // O(1) - one addition and one assignment
    }
    return sum; // O(1) - one return
}

int main() {
    std::vector<int> data = {1, 2, 3, 4, 5};
    std::cout << "Sum: " << sumArray(data) << std::endl;
    return 0;
}
```

Time Complexity Analysis:

- The initialization `int sum = 0;` takes constant time, `O(1)`.
- The `for` loop iterates `arr.size()` times. If `arr.size()` is `n`, the loop runs `n` times.
- Inside the loop, `sum += arr[i];` involves one addition and one assignment, both `O(1)` operations.
- The `return sum;` statement is `O(1)`.

Combining these, the dominant factor is the loop that runs `n` times. Therefore, the time complexity is **O(n)**.

Space Complexity Analysis:

- The `sum` variable takes `O(1)` space.

- The loop variable `i` takes `O(1)` space.
- The input `arr` is passed by constant reference, so it doesn't contribute to auxiliary space.

Thus, the auxiliary space used is constant. The space complexity is **O(1)**.

Example 2: Matrix Multiplication

```
 #include <iostream>
#include <vector>

// Assuming square matrices of size N x N
std::vector<std::vector<int>> multiplyMatrices(
    const std::vector<std::vector<int>>& mat1,
    const std::vector<std::vector<int>>& mat2,
    int N) {

    std::vector<std::vector<int>> result(N, std::vector<int>(N, 0)); // O(N^2)
space for result matrix

    for (int i = 0; i < N; ++i) { // Outer loop: N iterations
        for (int j = 0; j < N; ++j) { // Middle loop: N iterations
            for (int k = 0; k < N; ++k) { // Inner loop: N iterations
                result[i][j] += mat1[i][k] * mat2[k][j]; // O(1) operations
            }
        }
    }
    return result;
}

int main() {
    int N = 3;
    std::vector<std::vector<int>> matrixA = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}};
    std::vector<std::vector<int>> matrixB = {{9, 8, 7}, {6, 5, 4}, {3, 2, 1}};

    std::vector<std::vector<int>> matrixC = multiplyMatrices(matrixA, matrixB,
N);

    std::cout << "Resultant Matrix (C = A * B):" << std::endl;
    for (int i = 0; i < N; ++i) {
        for (int j = 0; j < N; ++j) {
            std::cout << matrixC[i][j] << " ";
        }
        std::cout << std::endl;
    }
    return 0;
}
```

Time Complexity Analysis:

- There are three nested `for` loops, each running `N` times.

- The operations inside the innermost loop (`result[i][j] += mat1[i][k] * mat2[k][j];`) are `O(1)` .
- The total number of operations is approximately `N * N * N = N^3` .

Therefore, the time complexity is **O(N^3)**.

Space Complexity Analysis:

- The `result` matrix is created with size `N x N` , which requires `O(N^2)` space.
- Input matrices `mat1` and `mat2` are passed by constant reference, so they don't contribute to auxiliary space.

Thus, the auxiliary space used is proportional to `N^2` . The space complexity is **O(N^2)**.

Example 3: Recursive Factorial (with Space Complexity consideration)

```
#include <iostream>

long long factorial(int n) {
    if (n == 0) { // Base case
        return 1;
    }
    return n * factorial(n - 1); // Recursive step
}

int main() {
    int num = 5;
    std::cout << "Factorial of " << num << " is: " << factorial(num) <<
std::endl;
    return 0;
}
```

Time Complexity Analysis:

- The function makes `n` recursive calls until it reaches the base case `n=0` .
- Each call involves a constant number of operations (comparison, multiplication, function call).

Therefore, the time complexity is **O(n)**.

Space Complexity Analysis:

- Each recursive call adds a new stack frame to the call stack. For `factorial(n)` , there will be `n+1` stack frames (from `n` down to `0`).

- Each stack frame stores local variables and return addresses.

Therefore, the space complexity due to the call stack is **O(n)**.

6. Best, Average, and Worst Case Complexity

When analyzing algorithms, it's common to consider different scenarios:

- **Worst-Case Complexity:** The maximum time or space an algorithm will take. This is the most common measure using Big O notation, as it provides an upper bound on performance.
- **Average-Case Complexity:** The expected time or space an algorithm will take, averaged over all possible inputs of a given size. This is often harder to calculate as it requires knowledge of the distribution of inputs.
- **Best-Case Complexity:** The minimum time or space an algorithm will take. This is usually not very useful in practice, as it represents an ideal scenario that may rarely occur.

For example, in a linear search algorithm:

- **Best Case:** O(1) - The element is found at the first position.
- **Worst Case:** O(n) - The element is found at the last position or not found at all.
- **Average Case:** O(n) - On average, the element is found somewhere in the middle.

7. Amortized Analysis (Optional)

Amortized analysis is a method for analyzing the time complexity of a sequence of operations, where the average cost of an operation is considered over a series of operations, rather than focusing on the worst-case cost of a single operation. It's particularly useful for data structures where occasional operations are very expensive, but most operations are cheap.

Example: Dynamic Array (like `std::vector` in C++)

When you add elements to a `std::vector`, most `push_back` operations are `O(1)`. However, when the vector's capacity is exceeded, it needs to reallocate a larger block of memory and copy all existing elements to the new location, which is an `O(n)`

operation. If we only consider the worst-case of `push_back`, it would be `O(n)`. However, over a sequence of `n` `push_back` operations, the total cost is `O(n)`, making the amortized cost per `push_back` operation `O(1)`.

Conclusion

Complexity analysis is an indispensable tool for any computer scientist or software engineer. By understanding time and space complexity, and utilizing Big O notation, you can effectively evaluate, compare, and optimize algorithms. This knowledge is crucial for writing efficient, scalable, and high-performing C++ applications, especially as input sizes grow and resource constraints become more critical. Always strive to choose algorithms with lower growth rates for better performance in the long run.

Resources

[1] GeeksforGeeks. "Complete Guide On Complexity Analysis". Available at: https://www.geeksforgeeks.org/dsa/complete-guide-on-complexity-analysis/ [2] Wikipedia. "Computational complexity". Available at: https://en.wikipedia.org/wiki/Computational_complexity [3] GeeksforGeeks. "Time Complexity and Space Complexity". Available at: https://www.geeksforgeeks.org/dsa/time-complexity-and-space-complexity/ [4] W3Schools. "DSA Time Complexity". Available at: https://www.w3schools.com/dsa/dsa_timecomplexity_theory.php [5] Wikipedia. "Big O notation". Available at: https://en.wikipedia.org/wiki/Big_O_notation [6] GeeksforGeeks. "Big O Notation Tutorial - A Guide to Big O Analysis". Available at: https://www.geeksforgeeks.org/analysis-algorithms-big-o-analysis/ [7] Simplilearn. "Big O Notation: Time Complexity & Examples Explained". Available at: https://www.simplilearn.com/big-o-notation-in-data-structure-article