Measuring Algorithmic Time Complexity from Source Code Implementations in the [Project Name – Feature]

All Authors Name[1] and Max Angelo Dapitilla Perin[1]

Bohol Island State University – Bilar Campus, Bohol, Philippines
`xxxx@bisu.edu.ph, maxangelo.perin@bisu.edu.ph`

Abstract. This template demonstrates how to document algorithmic time complexity for a specific software project feature in CS 311 – Algorithms and Complexity. Students are expected to (i) identify core algorithms in their source code, (ii) express their time complexity using Big-O, Big-Ω, and Big-Θ notation, and (iii) discuss the design implications for performance and scalability.

Keywords: Algorithms and Complexity · Time Complexity · Source Code Analysis · Data Structures

1 Introduction

Algorithms and data structures form the core of computer science and directly influence how software systems perform when the input size grows. In CS 311 – Algorithms and Complexity, students are trained to reason not only about whether a program produces the correct output, but also about how efficiently it does so in terms of time and space resources. As real-world applications increasingly handle large amounts of data, understanding algorithmic time complexity becomes essential for designing scalable and responsive systems.

The [Project Name – Feature] is a functional component of a larger software project developed in the Bachelor of Science in Computer Science curriculum. This feature operates on application data through operations such as searching, sorting, updating, or traversing records. Although the feature is already implemented and working, its performance characteristics are often not explicitly documented. Without a clear analysis of time complexity, developers may unintentionally deploy algorithms that behave poorly under heavy load or large input sizes.

This study focuses on measuring algorithmic time complexity directly from the source code of the [Project Name – Feature]. Instead of treating the implementation as a "black box", the analysis inspects loops, conditional statements, recursion, and data structure operations to derive mathematical expressions for the running time $T(n)$ in terms of the input size n. These expressions are then simplified into standard asymptotic notations such as Big-O, Big-Ω, and Big-Θ to provide an abstract view of performance growth.

Specifically, the objectives of this report are: (i) to identify the critical functions and methods in the [Project Name – Feature] that dominate running time, (ii) to derive corresponding time complexity equations from the source code, and (iii) to interpret the results in the context of expected input sizes and possible refactoring options. Through this analysis, the report aims to demonstrate how theoretical concepts from CS 311 apply concretely to an actual software artifact developed by students, thereby strengthening the link between algorithm design and practical software engineering.

2 Methodology

2.1 Identifying Input Size and Basic Operations

Let n denote the input size relevant to your feature (for example, the number of records, nodes, or items processed). Choose one basic operation to count, such as a key comparison or an assignment.

2.2 Template for Time Complexity Equations

Example: Nested Loop

```
for (i = 1; i <= n; i++) {
    for (j = 1; j <= n; j++) {
        // basic operation
    }
}
```

$$T(n) = \sum_{i=1}^{n}\sum_{j=1}^{n} 1 = \sum_{i=1}^{n} n = n^2 \tag{1}$$

$$= \Theta(n^2). \tag{2}$$

So we can write:

$$T(n) = O(n^2), \quad T(n) = \Omega(n^2), \quad T(n) = \Theta(n^2).$$

Example: Logarithmic Loop

```
while (n > 1) {
    n = n / 2;
    // basic operation
}
```

Number of iterations $\approx \log_2 n$:

$$T(n) = 3\log_2 n + 10 = \Theta(\log n),$$

so

$$T(n) = O(\log n), \quad T(n) = \Omega(\log n), \quad T(n) = \Theta(\log n).$$

2.3 General Polynomial Example

If

$$T(n) = 5n^3 + 2n^2 + 7,$$

then

$$T(n) = O(n^3), \quad T(n) = \Omega(n^3), \quad T(n) = \Theta(n^3).$$

3 Results and Discussion

This section reports the results of the time-complexity computations for the key functions or modules of the [Project Name – Feature] and interprets what they imply for performance.

3.1 Sample Derived Functions

For illustration, suppose the analysis produced the following equations:

– **Search operation** (`searchUsers()`):

$$T_{\text{search}}(n) = 4n + 12,$$

hence

$$T_{\text{search}}(n) = O(n), \quad T_{\text{search}}(n) = \Omega(n), \quad T_{\text{search}}(n) = \Theta(n).$$

– **Sort operation** (`sortRecords()`):

$$T_{\text{sort}}(n) = \frac{n^2 - n}{2},$$

so

$$T_{\text{sort}}(n) = O(n^2), \quad T_{\text{sort}}(n) = \Omega(n^2), \quad T_{\text{sort}}(n) = \Theta(n^2).$$

– **Index-building operation** (`buildIndex()`):

$$T_{\text{index}}(n) = 2n\log_2 n + 5n,$$

which yields

$$T_{\text{index}}(n) = O(n\log n), \quad T_{\text{index}}(n) = \Omega(n\log n), \quad T_{\text{index}}(n) = \Theta(n\log n).$$

3.2 Summary Table of Asymptotic Behaviour

After presenting your own table, discuss which operations become bottlenecks for large n and what algorithmic or data-structure changes could improve performance in the [Project Name – Feature].

Table 1. Sample summary of time-complexity results. Replace rows with your own functions.

Function/Module	Derived $T(n)$	Big-O	Big-Θ (Big-Ω)
`searchUsers()`	$4n + 12$	$O(n)$	$\Theta(n)$
`sortRecords()`	$\dfrac{n^2 - n}{2}$	$O(n^2)$	$\Theta(n^2)$
`buildIndex()`	$2n \log n + 5n$	$O(n \log n)$	$\Theta(n \log n)$

4 Conclusion

Summarize the main insights from your computed time complexities, relate them to realistic input sizes for the [Project Name – Feature], and state recommendations for improving or maintaining performance.

References

1. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms. 3rd edn. MIT Press (2009)
2. Sedgewick, R., Wayne, K.: Algorithms. 4th edn. Addison-Wesley (2011)