Big-O Complexity Analysis and Crucial Rules for Time Complexity

November 21, 2024

Big-O Complexity Analysis of Nested Loops

The given code is:

```
for (int i = 1; i <= n; ++i) {          // Outer loop
    cout << i;
    Sum = 0;
    for (int j = 1; j <= i; ++j) {      // Inner loop
        Sum++;
        cout << i;
    }
}
```

Outer Loop Analysis

The outer loop runs from $i = 1$ to $i = n$, incrementing by 1 in each iteration. The total number of iterations for the outer loop is:

$$\text{Outer loop iterations} = n.$$

Inner Loop Analysis

For each value of i, the inner loop runs from $j = 1$ to $j = i$, incrementing by 1 in each iteration. The number of iterations of the inner loop depends on the current value of i, and is given by:

$$\text{Inner loop iterations (for a fixed } i) = i.$$

Total Iterations

The total number of iterations for the statement `cout << i;` inside the inner loop is the sum of the iterations of the inner loop for all values of i in the outer loop:

$$\text{Total inner loop iterations} = \sum_{i=1}^{n} i.$$

Summing Up

The summation $\sum_{i=1}^{n} i$ is a well-known arithmetic series:

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}.$$

Thus, the total number of iterations for the inner loop is $\frac{n(n+1)}{2}$, which simplifies to $O(n^2)$.

Time Complexity

- The **outer loop** contributes $O(n)$. - The **inner loop**, when combined across all iterations, contributes $O(n^2)$. Thus, the overall time complexity of the nested loops is:

$$O(n^2).$$

—

Crucial Rules and Boundaries for Big-O Analysis

The following are essential rules and boundaries to help you analyze time complexities (Big-O) of various algorithms.

Basic Rules for Loops

Rule 1: Single Loops - A loop that runs from 1 to n (or similar bounds) has a time complexity of:

$$O(n)$$

Example:

```
for (int i = 1; i <= n; ++i) {
    // O(1) work
}
```

Rule 2: Nested Loops - For nested loops, multiply the complexities of the individual loops: - Outer loop: $O(n)$ - Inner loop: $O(n)$ - Total: $O(n \times n) = O(n^2)$ Example:

```
for (int i = 1; i <= n; ++i) {
    for (int j = 1; j <= n; ++j) {
        // O(1) work
    }
}
```

Rule 3: Logarithmic Loops - A loop that grows or shrinks exponentially (e.g., $i = i \times 2$ or $i = i/2$) has a time complexity of:

$$O(\log_2 n) \text{ or simply } O(\log n)$$

Example:

```
for (int i = 1; i <= n; i = i * 2) {
    // O(1) work
}
```

Rule 4: Combination of Linear and Logarithmic Loops - If one loop is $O(n)$ and another nested loop is $O(\log n)$, the total complexity is:

$$O(n \log n)$$

Example:

```
for (int i = 1; i <= n; ++i) {
    for (int j = 1; j <= n; j = j * 2) {
        // O(1) work
    }
}
```

Summation Rules

Rule 5: Arithmetic Summations - For summing over a series like $1+2+3+\ldots+n$:

$$\text{Sum} = \frac{n(n+1)}{2} \implies O(n^2)$$

Rule 6: Geometric Summations - For a geometric series like $1+2+4+\ldots+n$ (where each term doubles):

$$\text{Sum} = 2^0 + 2^1 + 2^2 + \ldots + 2^k = 2^{k+1} - 1$$

If $2^k = n$, then:

$$\text{Sum} = O(n)$$

Logarithmic Properties

Rule 7: Logarithm Growth - A logarithmic function grows slowly:

$$O(\log n) < O(\sqrt{n}) < O(n).$$

Logarithms appear in divide-and-conquer algorithms like mergesort, quicksort, or binary search.

Rule 8: Common Bases of Logarithms - Logs with different bases differ by a constant factor:

$$O(\log_2 n) = O(\log_{10} n) = O(\ln n).$$

Ignore the base when analyzing time complexity.

Key Patterns in Nested Loops

Rule 9: Dependent Loops - If an inner loop's range depends on the outer loop's variable:

```
for (int i = 1; i <= n; ++i) {
    for (int j = 1; j <= i; ++j) {
        // O(1) work
    }
}
```

The total number of iterations is:

$$1 + 2 + 3 + \ldots + n = \frac{n(n+1)}{2} \implies O(n^2).$$

Rule 10: Logarithmic Inner Loop - If the inner loop doubles each iteration:

```
for (int i = 1; i <= n; ++i) {
    for (int j = 1; j <= i; j = j * 2) {
        // O(1) work
    }
}
```

Total complexity:

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

Recursion Rules

Rule 11: Divide and Conquer - Divide-and-conquer algorithms break a problem into smaller subproblems, solve them, and combine results. For example:

$$T(n) = 2T(n/2) + O(n).$$

Using the *Master Theorem*, the complexity is:

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

Rule 12: Master Theorem For a recurrence of the form:

$$T(n) = aT\left(\frac{n}{b}\right) + O(n^d),$$

- Compare $\log_b a$ (work per subproblem) and d (work outside recursion): 1. If $\log_b a > d$: $O(n^{\log_b a})$. 2. If $\log_b a = d$: $O(n^d \log n)$. 3. If $\log_b a < d$: $O(n^d)$.

Other Important Cases

Rule 13: Constant Time - If the loop executes a fixed number of times, the complexity is:

$$O(1).$$

Rule 14: Nested Multiplicative Loops - When nested loops multiply instead of iterating linearly:

```
for (int i = 1; i <= n; i = i * 2) {
    for (int j = 1; j <= n; j = j * 2) {
        // O(1) work
    }
}
```

Total iterations:

$$O(\log n \times \log n) = O((\log n)^2).$$

Best, Worst, and Average Cases

Rule 15: Three Cases for Complexity - Analyze the input: - **Best Case**: Fewest iterations. - **Worst Case**: Maximum iterations. - **Average Case**: Typical input behavior.

Amortized Analysis

Rule 16: Dynamic Array Resizing - Doubling the size of an array leads to amortized $O(1)$ insertion time: - Resize cost is spread over many operations.