Order notation is a mathematical method for bounding the performance of an algorithm as its size grows without bound. It allows us to define and compare an algorithm’s performance in a way that is free from uncontrolled influences like machine load, implementation efficiency, and so on.

We provide a brief review of order notation, to ensure a common understanding of its terminology. This is not meant to be a comprehensive discussion of algorithm analysis. In fact, although we use order notation throughout this course, we are often “flexible,” for example, by only measuring certain parts of an algorithm (e.g., seek times) when we know these operations account for the vast majority of the algorithm’s execution time.

### 2.1 Theta Notation

Θ-notation provides asymptotic upper and lower bounds on the efficiency of an algorithm for a particular input size $n$.

$$
\Theta(g(n)) = \{ f(n) : \exists c_1, c_2, n_0 > 0 \mid 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \ \forall \ n \geq n_0 \} \quad (2.1)
$$
Chapter 2. Order Notation

Saying \( f(n) = \Theta(g(n)) \) means \( f(n) \in \Theta(g(n)) \). For example, suppose \( f(n) = \frac{1}{2}n^2 - 3n \). We claim \( f(n) = \Theta(n^2) \). To prove this, we must find \( c_1, c_2, n_0 > 0 \) satisfying the inequality:

\[
 c_1 n^2 \leq \frac{1}{2}n^2 - 3n \leq c_2 n^2
\]  

(2.2)

Dividing through by \( n^2 \) yields:

\[
 c_1 \leq \frac{1}{2} - \frac{3}{n} \leq c_2
\]  

(2.3)

If \( c_2 = \frac{1}{2} \), Eqn. 2.3 is true \( \forall n \geq 1 \). Similarly, if \( c_1 = \frac{1}{14} \), Eqn. 2.3 is true \( \forall n \geq 7 \) (for \( n = 7, \frac{1}{2} - \frac{3}{7} = \frac{1}{14} \)). \( \therefore \) Eqn. 2.3 is true for \( c_1 = \frac{1}{14}, c_2 = \frac{1}{2}, n_0 = 7 \).

Note that there are many possible \( c_1, c_2, n_0 \) that we could have chosen. The key is that there must exist at least one set of \( c_1, c_2, n_0 \) values that satisfy our constraints.

2.2 Big-O Notation

O-notation provides an asymptotic upper bound on the efficiency of an algorithm for a particular input size \( n \).

\[
 O(g(n)) = \{ f(n) : \exists c, n_0 > 0 \mid 0 \leq f(n) \leq c g(n) \ \forall n \geq n_0 \} 
\]  

(2.4)

Note that if an algorithm is \( f(n) = O(n) \), then by definition it is also \( O(n^2), O(n^3) \), and so on. When we say \( f(n) = O(g(n)) \), we normally try to choose \( g(n) \) to define a tight upper bound. Also, \( f(n) = \Theta(g(n)) \iff f(n) = O(g(n)) \), that is, \( \Theta(g(n)) \subset O(g(n)) \).

If \( f(n) = O(g(n)) \) for worst-case input, then \( f(n) = O(g(n)) \) for any input. This is not true for \( \Theta \)-notation. If \( f(n) = \Theta(g(n)) \) for worst case input, it does not imply that \( f(n) = \Theta(g(n)) \) for all input. Certain inputs may provide performance better than \( g(n) \).
2.3 Big-Omega Notation

Ω-notation provides an asymptotic lower bound on the efficiency of an algorithm for a particular input size n.

\[
\Omega(g(n)) = \{ f(n) : \exists c, n_0 > 0 \mid 0 \leq c g(n) \leq f(n) \ \forall \ n \geq n_0 \} \tag{2.5}
\]

As with O-notation, \( f(n) = \Theta(g(n)) \implies f(n) = \Omega(g(n)) \), that is, \( \Theta(g(n)) \subset \Omega(g(n)) \). In fact, \( f(n) = \Theta(g(n)) \iff f(n) = O(g(n)) \) and \( f(n) = \Omega(g(n)) \).

2.4 Insertion Sort

As a practical example, consider measuring the performance of insertion sort, where performance is represented by the number of statements executed within the sort. An insertion sort of an array \( A[1 \ldots n] \) of size \( n \) pushes each element \( A[2] \) through \( A[n] \) to the left, into its sorted position in the front of the array.

\[
\text{insertion_sort}(A, n)
\]

**Input:** \( A[\] \), array of integers to sort; \( n \), size of \( A \)

for \( j = 2 \) to \( n \) do

\[
\begin{align*}
\text{key} &= A[j] \\
i &= j - 1 \\
\text{while } i \geq 1 \text{ and } A[i] > \text{key} \\
\text{do}
\begin{align*}
i &= i - 1
\end{align*}
\end{align*}
\]

\( A[i + 1] = \text{key} \)

end

The numbers to the right of each statement represent a count of the number of times the statement is executed. These are based either on \( n \), the size of the array, or \( t_j \), the number of executions on the \( j \)-th pass through the inner while loop. We can simply sum up the execution counts to get an overall cost \( T(n) \) for an insertion sort on an array of size \( n \).

\[
T(n) = n + (n - 1) + (n - 1) + \sum_{j=1}^{n-1} t_j + \sum_{j=1}^{n-1} (t_j - 1) + \sum_{j=1}^{n-1} (t_j - 1) + (n - 1) \tag{2.6}
\]

Suppose we encountered the best-case input, an already sorted array. In this case \( t_j = 1 \ \forall \ j = 2, \ldots, n \), since we never enter the body of the while loop (because \( A[i] \leq A[i + 1] \ \forall \ i = 1, \ldots, n \)). The cost in this case is:
Chapter 2. Order Notation

\[ T(n) = n + (n - 1) + (n - 1) + \sum_{j=1}^{n-1} 1 + \sum_{j=1}^{n-1} 0 + \sum_{j=1}^{n-1} 0 + (n - 1) \]

\[ = 4n - 3 + \sum_{i=1}^{n-1} 1 \]

\[ = 4n - 3 + (n - 1) \]

\[ = 5n - 4 \]

(2.7)

This shows best-case performance of \( \Theta(n) \).

In the worst case, \( A \) is reverse sorted, and the while loop must be executed \( j \) times for the \( j \)-th pass through the for loop. This means \( t_j = j \). To compute total cost, we need to solve for \( \sum_{j=1}^{n-1} j \) and \( \sum_{j=1}^{n-1} (j - 1) \). Recall:

\[ \sum_{j=1}^{n-1} j = \frac{n(n - 1)}{2} \]

(2.8)

\[ \sum_{j=1}^{n-1} (j - 1) = \sum_{j=1}^{n-1} j - \sum_{j=1}^{n-1} 1 \]

\[ = \frac{n(n - 1)}{2} - (n - 1) \]

(2.9)

\[ = \frac{n(n - 3)}{2} + 1 \]

Given these sums, the total cost \( T(n) \) can be calculated.

\[ T(n) = n + (n - 1) + (n - 1) + \frac{n(n - 1)}{2} + \frac{n(n - 3)}{2} + 1 + \frac{n(n - 3)}{2} + 1 + (n - 1) \]

\[ = \frac{3n^2}{2} + n - 1 \]

(2.10)

The \( n^2 \) term in \( T(n) \) dominates all other terms as \( n \) grows large, so \( T(n) = \Theta(n^2) \) for worst case input. Given this, the worst case \( T(n) = O(n^2) \) as well, so we often say that insertion sort runs in \( O(n^2) \) in the worst case.

In the average case, we could argue that each element in \( A \) would need to move about half the distance from its starting position to the front of \( A \). This means \( t_j = \frac{j}{2} \). Although we don’t provide the details here, calculating \( T(n) \) shows that, on average, it is also \( \Theta(n^2) \).

In spite of insertion sort’s \( \Theta(n^2) \) average case performance, it is one of the fastest sorts, in absolute terms, for small arrays. For example, insertion sort if often used in Quicksort to sort a partition once its length falls below some pre-defined threshold, because this is faster than recursively finishing the Quicksort.