Big-O Notation
Analysis of Algorithms
(how fast does an algorithm grow with respect to N)
(Note: Best recollection is that a good bit of this document comes from C++ For You++, by Litvin & Litvin)

The time efficiency of almost all of the algorithms we have discussed can be characterized by only a few growth rate functions:

I. **O(1) - constant time**
   This means that the algorithm requires the same fixed number of steps regardless of the size of the task.

   **Examples** (assuming a reasonable implementation of the task):
   
   A. Push and Pop operations for a stack (containing n elements);
   B. Insert and Remove operations for a queue.

II. **O(n) - linear time**
   This means that the algorithm requires a number of steps proportional to the size of the task.

   **Examples** (assuming a reasonable implementation of the task):
   
   A. Traversal of a list (a linked list or an array) with n elements;
   B. Finding the maximum or minimum element in a list, or sequential search in an unsorted list of n elements;
   C. Traversal of a tree with n nodes;
   D. Calculating iteratively n-factorial; finding iteratively the n-th Fibonacci number.

III. **O(n^2) - quadratic time**
   The number of operations is proportional to the size of the task squared.

   **Examples**:
   
   A. Some more simplistic sorting algorithms, for instance a selection sort of n elements;
   B. Comparing two two-dimensional arrays of size n by n;
   C. Finding duplicates in an unsorted list of n elements (implemented with two nested loops).

IV. **O(log n) - logarithmic time**

   **Examples**:
   
   A. Binary search in a sorted list of n elements;
   B. Insert and Find operations for a binary search tree with n nodes;
   C. Insert and Remove operations for a heap with n nodes.

V. **O(n log n) - "n log n" time**

   **Examples**:
   
   A. More advanced sorting algorithms - quicksort, mergesort

VI. **O(a^n) (a > 1) - exponential time**

   **Examples**:
   
   A. Recursive Fibonacci implementation
   B. Towers of Hanoi
   C. Generating all permutations of n symbols
The best time in the above list is obviously constant time, and the worst is exponential time which, as we have seen, quickly overwhelms even the fastest computers even for relatively small n. Polynomial growth (linear, quadratic, cubic, etc.) is considered manageable as compared to exponential growth.

Order of asymptotic behavior of the functions from the above list:
Using the "<" sign informally, we can say that

\[ \text{O}(1) < \text{O}(\log n) < \text{O}(n) < \text{O}(n \log n) < \text{O}(n^2) < \text{O}(n^3) < \text{O}(a^n) \]

**A word about O(\log n) growth:**

As we know from the Change of Base Theorem, for any a, b > 0, and a, b ≠ 1

\[ \log_a n = \frac{\log_b n}{\log_b a} \]

Therefore,

\[ \log_a n = C \log_b n \]

where C is a constant equal to \( \log_a b \).

Since functions that differ only by a constant factor have the same order of growth, \( \text{O}(\log_2 n) \) is the same as \( \text{O}(\log n) \). Therefore, when we talk about logarithmic growth, the base of the logarithm is not important, and we can say simply \( \text{O}(\log n) \).

**A word about Big-O when a function is the sum of several terms:**

If a function (which describes the order of growth of an algorithm) is a sum of several terms, its order of growth is determined by the fastest growing term. In particular, if we have a polynomial

\[ p(n) = a_k n^k + a_{k-1} n^{k-1} + \ldots + a_1 n + a_0 \]

its growth is of the order \( n^k \):

\[ p(n) = \text{O}(n^k) \]

Example:

\[
\begin{aligned}
\{ \text{perform any statement } S_1 \} & \quad \text{O}(1) \\
\text{for } (i=0; i < n; i++) & \\
\{ & \quad \text{O}(n) \\
\{ & \quad \text{perform any statement(s) } S_2 \} \\
\{ & \quad \text{run through another loop } n \text{ times} \} & \quad \text{O}(n^2) \\
\} \end{aligned}
\]

Total Execution Time: \( 1 + n + n^2 \) therefore, \( \text{O}(n^2) \)