Big O Notation

Contents

[O types 2](#_Toc416265389)

[O(logn) 2](#_Toc416265390)

[O(n) 2](#_Toc416265391)

[O(nlogn) 2](#_Toc416265392)

[O(n^2) 2](#_Toc416265393)

[O(2^n) 2](#_Toc416265394)

# O types

## O(logn)

Algorithms based on binary trees are often O(logn). This is because a perfectly balanced binary search tree has logn layers, and to search for any element in a binary search tree requires traversing a single node on each layer.  
  
The binary search algorithm is another example of a O(log n) algorithm. In a binary search, one is searching through an ordered array and, beginning in the middle of the remaining space to be searched, whether to search the top or the bottom half. You can divide the array in half only logn times before you reach a single element, which is the element being searched for, assuming it is in the array.

## O(n)

Algorithms of efficiency n require only one pass over an entire input. For instance, a linear search algorithm, which searches an array by checking each element in turn, is O(n). Often, accessing an element in a linked list is O(n) because linked lists do not support random access.

## O(nlogn)

Often, good sorting algorithms are roughly O(nlogn). An example of an algorithm with this efficiency is merge sort, which breaks up an array into two halves, sorts those two halves by recursively calling itself on them, and then merging the result back into a single array. Because it splits the array in half each time, the outer loop has an efficiency of logn, and for each "level" of the array that has been split up (when the array is in two halves, then in quarters, and so forth), it will have to merge together all of the elements, an operations that has order of n.

## O(n^2)

A fairly reasonable efficiency, still in the polynomial time range, the typical examples for this order come from sorting algorithms, such as the selection sort example on the previous page.

## O(2^n)

The most important non-polynomial efficiency is this exponential time increase. Many important problems can only be solved by algorithms with this (or worse) efficiency. One example is factoring large numbers expressed in binary; the only known way is by trial and error, and a naive approach would involve dividing every number less than the number being factored into that number until one divided in evenly. For every increase of a single digit, it would require twice as many tests.