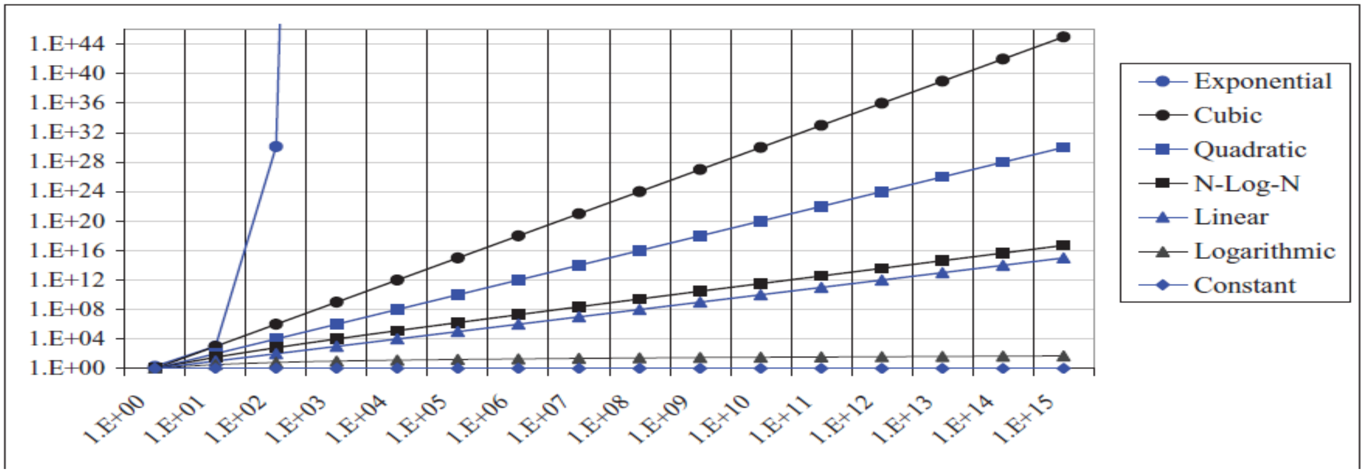
Analysis of Algorithms

Introduction

Functions

Name	Formula	Graph
The Constant Function	$f(n) = C$	
The Logarithm Function	$f(n) = \log(n)$	
The Linear Function	$f(n) = n$	
The N-Log-N Function	$f(n) = nlog(n)$	
The Quadratic Function	$f(n) = n^2$	
The Cubic Function and Other Polynomials	$f(n) = n^3$	
The Exponential Function	$f(n) = e^n$	



NOTES:

[The Logarithm Function]:

1. we can divide n by a until we get a number less than or equal to 1. For example, this evaluation of $\log_3 \mathbf{27}$ is 3, since 27/3/3/3 = 1.

2. log n = $\log_2 \boldsymbol{n}$.

3. Logarithm Rules :

$$\log_b ac = log_b(a) + log_b(c) \qquad \log_b a^c = clog_b(a)$$

$$\log_b(\frac{a}{c}) = log_b(a) - log_b\,(c)$$

[Polynomials Functions]:

1. A polynomial function is a function of the form:

$$f\ (n)\ =\ a_0 + a_1 n + a_2 n_2 + a_3 n_3\ + \cdots + a_d n^d$$

2. A notation that appears again and again in the analysis of data structures and algorithms is the summation, which is defined as :

$$\sum_{i=a}^{b} f\ (i)\ =\ f\ (a) +\ f\ (a+1) +\ f\ (a+2) + \cdots + f\ (b)$$

Summations arise in data structure and algorithm analysis because the running times of loops naturally give rise to summations:

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

Analysis of Algorithms

1) Experimental Studies **2) Asymptotic Notation**

Experimental Studies

If an algorithm has been implemented, we can study its running time by executing it on various test inputs and recording the actual time spent in each execution. While experimental studies of running times are useful, they have three majors limitations:

1) Experiments can be done only on a limited set of test inputs; hence, they leave out the running times of inputs not included in the experiment (and these inputs may be important).
2) We have difficulty comparing the experimental running times of two algorithms unless the experiments were performed in the same hardware and software environments.
3) We have to fully implement and execute an algorithm in order to study its running time experimentally.

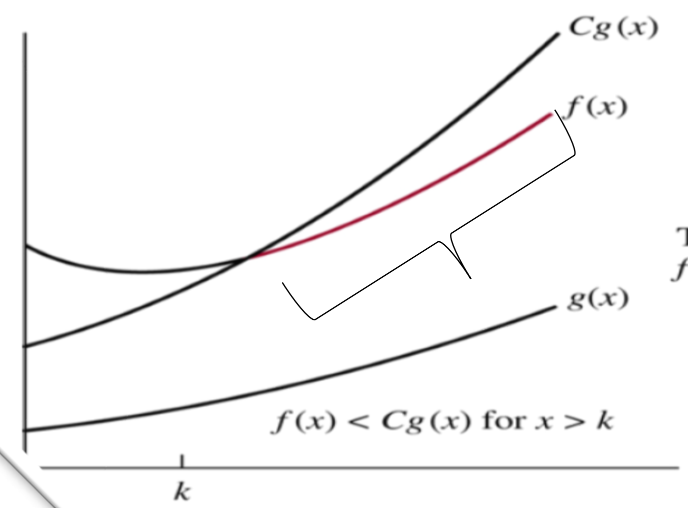
Asymptotic Notation

Big-O Notation

Let f and g be functions from the set of integers or the set of real numbers to the set of real numbers. We say that $f(x)\ is\ O\big(g(x)\big)\ \ or\ O\big(g(x)\big) = f(x)$: if there are constants C and k such that:

$$|f(x)| \leq C|g(x)|\ \ (x > k) \rightarrow \frac{|f(x)|}{|g(x)|} \leq C \quad (x > k)$$

It is defined as upper bound and upper bound on an algorithm is the most amount of time required (the worst-case performance).



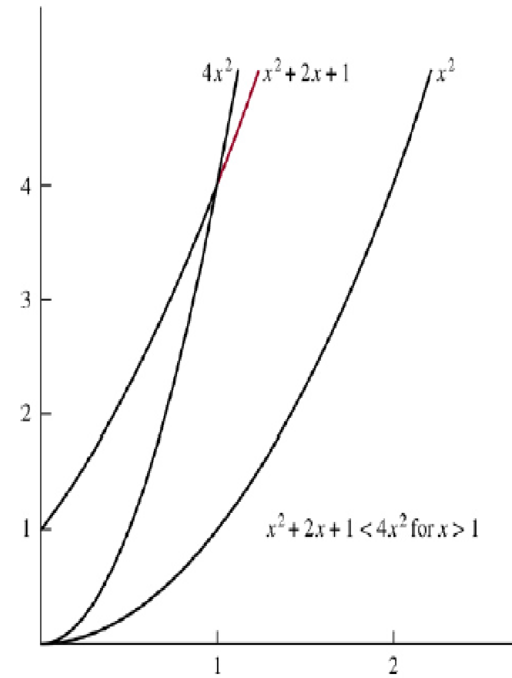
The part of the graph of $f(x)$ that satisfies $f(x) < Cg(x)$ is shown in color.

Example: *Show that* $f(x) = x^2 + 2x + 1$ *is* $O(x^2)$

SOLUTION:

When $x > 1$ we know that :
$x \le x^2$ and $1 \le x^2$ then:
$0 \le\ x^2 + 2x + 1 \le\ x^2 + 2x^2 + x^2$
$= 4x^2$

Then $C = 4$ and $K = 1$



The part of the graph of $f(x) = x^2 + 2x + 1$ that satisfies $f(x) < 4x^2$ is shown in color.

<u>**Example:**</u> Show that the function $n^2$is not $O(n)$

Suppose that $n^2 = O(n)$ then $n^2 \le\ Cn \rightarrow n < C$
Then the C must be larger than n but C is constant and n choose any value for C then you will notice that n won't satisfy the condition.

<u>**Some Important Big-O Results:**</u>

1) $f(x) =\ \sum_{i=0}^{n} a_i x^i = a_0 + a_1 x + \cdots + a_n x^n$ then $f(x) = O(x^n)$

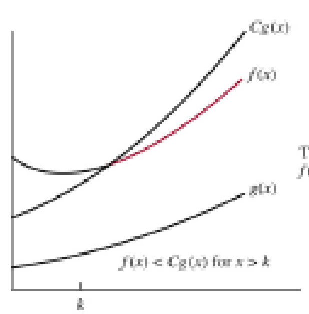
2) $\log(n!) = O\big(n\log(n)\big)$ 3) $n\log(n) = O(\log\,(n!))$

4)$f_1(x) = O\big(g_1(x)\big)$ *and* $f_2(x) = O\big(g_2(x)\big)$ *then* $(f_1 + f_2)(x) = O(MAX(|g_1(x)|, |g_2(x)|))$

5) $f_1(x) = O\big(g_1(x)\big)$ *and* $f_2(x) = O\big(g_2(x)\big)$ *then* $(f_1 f_2)(x) = O(|g_1\,(x)||g_2\,(x)|)$

Big-Omega Ω

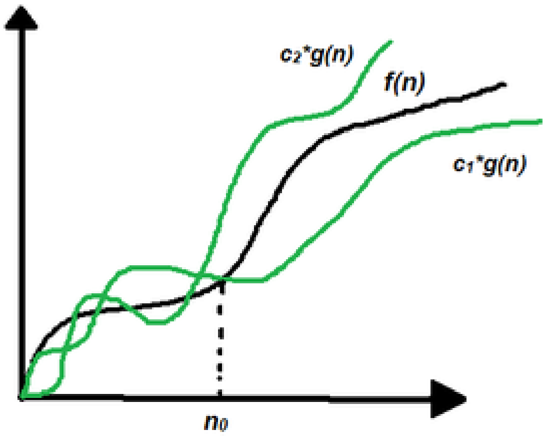
Omega notation represents the lower bound of the running time of an algorithm.



The part of the graph of $f(x)$ that satisfies $f(x) < Cg(x)$ is shown in color.

Theta notation Θ

Theta notation encloses the function from above and below. Since it represents the upper and the lower bound of the running time of an algorithm, it is used for analyzing the average-case complexity of an algorithm.



Theta notation

Properties of Asymptotic Notations:
https://www.geeksforgeeks.org/analysis-of-algorithms-set-3asymptotic-notations/

Complexity of Algorithms

- **Time complexity**: time required for solving the problem.
- **Space complexity**: memory required for solving the problem.

Time Complexity

Definition

(Time Complexity) of an algorithm is expressed in terms of the number of basic operations used by the algorithm when the input has a particular size.

Notes:

1) We defined Time Complexity in number of steps, not absolute time, not to be machine dependent.
2) Not all operations are basic; e.g :

```
x = MatrixMultiplcation(A, B);
x = 0;
```

3) Not all basic operations take same execution time.

```
x = 3 * 4;
x = 3 + 4;
```

General

The best is to obtain an exact expression for complexity T (number of steps) as a function of n (problem size): $T = T(n)$ or $T = \Theta(f(n))$ or $T = O(f(n))$

Example

```
xmax = list[0];
for(i=1; i<n; i++)
        if (list[i]>xmax)
            xmax = list[i];
```

$T_1 = 2(n-1) + 1 = 2n - 1 = \theta(n)$

NOTE:

Sometimes T is a random variable (not deterministic) and in this case :

- Worst-case complexity: max(T).
- Best-case complexity: min(T).
- Average-case complexity: E[T]

For example, linear search is random variable as we don't know when we will terminate the search process (We will not talk about Random variables in this session).

Compare

To compare two algorithms, you must use the same step definition.

$$T_1(n) = n^2 + 10n \qquad = \Theta\left(n^2\right)$$

$$T_2(n) = 20n \qquad = \Theta(n).$$

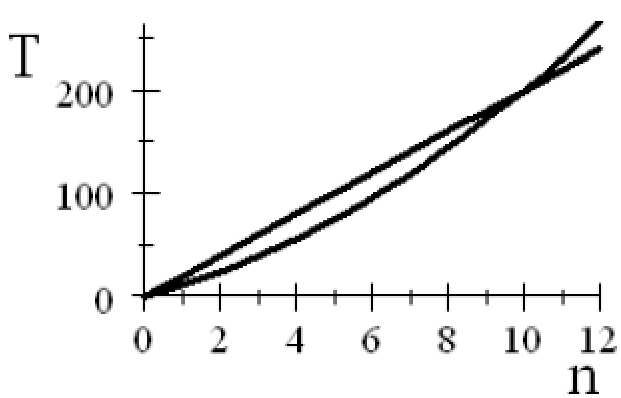


TABLE 1 Commonly Used Terminology for the Complexity of Algorithms.	
Complexity	*Terminology*
$\Theta(1)$	Constant complexity
$\Theta(\log n)$	Logarithmic complexity
$\Theta(n)$	Linear complexity
$\Theta(n \log n)$	$n \log n$ complexity
$\Theta(n^b)$	Polynomial complexity
$\Theta(b^n)$, where $b > 1$	Exponential complexity
$\Theta(n!)$	Factorial complexity

Difference Between Big oh, Big Omega and Big Theta :

Big O	Big Ω	Big Θ
It is like (<=) rate of growth of an algorithm is less than or equal to a specific value.	It is like (>=) rate of growth is greater than or equal to a specified value.	It is like (==) meaning the rate of growth is equal to a specified value.
The upper bound of algorithm is represented by Big O notation. Only the above function is bounded by Big O. Asymptotic upper bound is given by Big O notation.	The algorithm's lower bound is represented by Omega notation. The asymptotic lower bound is given by Omega notation.	The bounding of function from above and below is represented by theta notation. The exact asymptotic behavior is done by this theta notation
Upper Bound	Lower Bound	Tight Bound
It is defined as upper bound and upper bound on an algorithm is the most amount of time required (*the worst-case performance*).	It is defined as lower bound and lower bound on an algorithm is the least amount of time required (*the most efficient way possible, in other words best case*)	It is define as tightest bound and tightest bound is the best of all the worst case times that the algorithm can take.
Mathematically: Big Oh is 0 <= f(n) <= Cg(n) for all n >= n0	Mathematically: Big Omega is 0 <= Cg(n) <= f(n) for all n >= n0	Mathematically – Big Theta is 0 <= C2g(n) <= f(n) <= C1g(n) for n >= n0