**1.2.1 Big Oh Notation**

-

A convenient way of describing the growth rate of a function and hence the time complexity of an algorithm.

Let n be the size of the input and *f* (*n*), *g*(*n*) be positive functions of n.

**DEF.Big Oh**. *f* (*n*) is *O*(*g*(*n*)) if and only if there exists a real, positive constant *C*and a positive integer *n*0 such that

*f* (*n*) $\displaystyle \leq$*Cg*(*n*) $\displaystyle \forall$   *n$\displaystyle \geq$n*0

* Note that *O*(*g*(*n*)) is a class of functions.
* The "Oh" notation specifies asymptotic upper bounds
* *O*(1) refers to constant time. *O*(*n*) indicates linear time; *O*(*n*k) (k fixed) refers to polynomial time; *O*(log *n*) is called logarithmic time; *O*(2n) refers to exponential time, etc.

**1.2.2 Examples**

* Let f(n) = n2 + n + 5. Then

-

f(n) is O(n2)

-

f(n) is O(n3)

-

f(n) is not O(n)

* Let f(n) = 3n

-

f(n) is O(4n)

-

f(n) is not O(2n)

* If f1(n) is O(g1(n)) and f2(n) is O(g2(n)), then

-

f1(n) + f2(n) is O(max(g1(n), g2(n)))

**1.2.3 An Example: Complexity of Mergesort**

Mergesort is a divide and conquer algorithm, as outlined below. Note that the function *mergesort* calls itself *recursively*. Let us try to determine the time complexity of this algorithm.

**list** mergesort (**list** L, **int** n);

            {

**if** (n = = 1)

**return** (L)

**else** {

                Split L into two halves L1 and L2 ;

**return** (merge (mergesort (L1, $ {\frac{n}{2}}$), (mergesort (L2, $ {\frac{n}{2}}$))

                    }

             }

Let T(n) be the running time of Mergesort on an input list of size n. Then,

|  |  |  |  |
| --- | --- | --- | --- |
| *T*(*n*) | $\displaystyle \leq$ | *C*1  (if  *n* = 1)    (*C*1 is a constant) |  |
|  | $\displaystyle \leq$ | $\displaystyle \underbrace{2 \space T \space \left(\frac{n}{2}\right)}_{\mbox{ two recursive \space calls}}^{}\,$+ $\displaystyle \underbrace{C_2 n}_{\mbox{cost \space of \space merging}}^{}\,$  (if  *n* > 1) |  |

If n = 2k for some k, it can be shown that

*T*(*n*) $\displaystyle \leq$ 2k*T*(1) + *C*2*k*2k

That is, *T*(*n*) is *O*(*n*log *n*).   
 

|  |
| --- |
| **Figure 1.1:** Growth rates of some functions |
| \begin{figure}\centerline{\psfig{figure=figures/Fgrowthrate.ps}}\end{figure} |

 

|  |
| --- |
| **Table 1.1:** Growth rate of functions |
| |  |  |  |  | | --- | --- | --- | --- | |  | Maximum Problem Size that can be solved in | | | | T(n) | 100 | 1000 | 10000 | |  | Time Units | Time Units | Time Units | |  |  |  |  | | 100 n | 1 | 10 | 100 | |  |  |  |  | | 5 n2 | 5 | 14 | 45 | |  |  |  |  | | n3/2 | 7 | 12 | 27 | |  |  |  |  | | 2n | 8 | 10 | 13 | |  |  |  |  | |

## 1.2.4 Role of the Constant

## The constant C that appears in the definition of the asymptotic upper bounds is very important. It depends on the algorithm, machine, compiler, etc. It is to be noted that the big "Oh" notation gives only asymptotic complexity. As such, a polynomial time algorithm with a large value of the constant may turn out to be much less efficient than an exponential time algorithm (with a small constant) for the range of interest of the input values. See Figure [1.1](http://lcm.csa.iisc.ernet.in/dsa/node7.html#fig:growthrate) and also Table 1.1.

**1.2.5 Worst Case, Average Case, and Amortized Complexity**

* **Worst case Running Time**: The behavior of the algorithm with respect to the worst possible case of the input instance. The worst-case running time of an algorithm is an upper bound on the running time for any input. Knowing it gives us a guarantee that the algorithm will never take any longer. There is no need to make an educated guess about the running time.
* **Average case Running Time**: The expected behavior when the input is randomly drawn from a given distribution. The average-case running time of an algorithm is an estimate of the running time for an "average" input. Computation of average-case running time entails knowing all possible input sequences, the probability distribution of occurrence of these sequences, and the running times for the individual sequences. Often it is assumed that all inputs of a given size are equally likely.
* **Amortized Running Time** Here the time required to perform a sequence of (related) operations is averaged over all the operations performed. Amortized analysis can be used to show that the average cost of an operation is small, if one averages over a sequence of operations, even though a simple operation might be expensive. Amortized analysis guarantees the average performance of each operation in the worst case.

1.

For example, consider the problem of finding the minimum element in a list of elements.

Worst case = O(n)

Average case = O(n)

2.

Quick sort

Worst case = O(n2)

Average case = O(n log n)

3.

Merge Sort, Heap Sort

Worst case = O(n log n)

Average case = O(n log n)

4.

Bubble sort

Worst case = O(n2)

Average case = O(n2)

5.

Binary Search Tree: Search for an element

Worst case = O(n)

Average case = O(log n)

**1.2.6 Big Omega and Big Theta Notations**

The $ \Omega$notation specifies asymptotic lower bounds.

**DEF.** **Big Omega**. *f* (*n*) is said to be $ \Omega$(*g*(*n*)) if $ \exists$a positive real constant C and a positive integer *n*0 such that

*f* (*n*) $\displaystyle \geq$*Cg*(*n*)   $\displaystyle \forall$  *n* $\displaystyle \geq$*n*0

An Alternative Definition : *f* (*n*) is said to be $ \Omega$(*g*(*n*)) iff $ \exists$a positive real constant C such that

*f* (*n*) $\displaystyle \geq$*Cg*(*n*)  for infinitely many values of *n*.

The $ \Theta$notation describes asymptotic tight bounds.

**DEF.** **Big Theta**. *f* (*n*) is $ \Theta$(*g*(*n*)) iff $ \exists$positive real constants C1 and C2 and a positive integer n0, such that

*C*1*g*(*n*) $\displaystyle \leq$*f* (*n*) $\displaystyle \leq$*C*2*g*(*n*)   $\displaystyle \forall$ *n* $\displaystyle \geq$*n*0

**1.2.7 An Example:**

Let *f* (*n*) = 2*n*2 + 4*n* + 10. *f* (*n*) is O(n2). For,

*f* (*n*) $ \leq$ 3*n*2$ \forall$  *n$ \geq$* 6

Thus, C = 3 and *n*0 = 6

Also,

*f* (*n*) $ \leq$ 4*n*2$ \forall$  *n$ \geq$* 4

Thus, *C* = 4 and *n*0 = 4

*f* (*n*) is O(n3)

In fact, if *f* (*n*) is *O*(*n*k) for some *k*, it is *O*(*n*h) for *h* > *k*

*f* (*n*) is not *O*(*n*).

Suppose $ \exists$ a constant C such that    
                            2*n*2 + 4*n* + 10 $\displaystyle \leq$*Cn$\displaystyle \forall$n$\displaystyle \geq$n*0   
This can be easily seen to lead to a contradiction. Thus, we have that:

*f* (*n*) is $ \Omega$(*n*2) and *f* (*n*) is $ \Theta$(*n*2)

## 1.2.4 Role of the Constant

The constant C that appears in the definition of the asymptotic upper bounds is very important. It depends on the algorithm, machine, compiler, etc. It is to be noted that the big "Oh" notation gives only asymptotic complexity. As such, a polynomial time algorithm with a large value of the constant may turn out to be much less efficient than an exponential time algorithm (with a small constant) for the range of interest of the input values. See Figure [1.1](http://lcm.csa.iisc.ernet.in/dsa/node7.html#fig:growthrate) and also Table 1.1.