Algorithms are measured in terms of *time* and *space complexity*. The time complexity of an algorithm is a measure of how much time is required to execute an algorithm for a given number of inputs. The time complexity of an algorithm is measured by its rate of growth relative to standard functions. Standard functions are given in Table 1.

Table 1. Standard Time Complexity Functions

Function	Name
c	Constant
$\log N$	Logarithmic
$\log^2 N$	Log-squared
N	Linear
$N \log N$	
N^2	Quadratic
N^3	Cubic
2^N	Exponential

Space complexity is similar to time complexity. The space complexity of an algorithm is a measure of how much storage is required by the algorithm.

Time and space can often be bartered. It is possible to make an algorithm use more space and less time.

Typically, computer scientists are interested in minimizing the time complexity of algorithms. The economics of storage versus the speed of computers are the principal factor determining the focus on time complexity. Memory has decreased in cost at an exponential rate for the past 50 years whereas the cost of central processing unit time has not decreased at that rate. The bottleneck is execution time. Hence, computer scientists focus on the execution time of algorithms.

An algorithm can be characterized by a timing function $T(N)$. $T(N)$ is a measure of how much time is required to execute an algorithm given N values. For example, the timing function for a sort specifies the time required to sort N values. The timing function for an algorithm that solves a system of linear equations specifies the time required solving N linear equations.

If we say that an algorithm is $O(N^2)$, pronounced, oh of N squared, then what we mean is that the timing function for the algorithm will grow no faster than the square of the number of values it processes.

Definition: $T(N) = O(f(N))$ if there are positive *constants* c and n_0 such that $T(N) \le cf(N)$ when $N \ge n_0$.

Steps:

1. Find $T(N)$ for the algorithm.
2. $f(N)$ is the fastest growing term in $T(N)$.
3. Compute the minimum value for c, $c_{mim} = \lim_{n\to\infty} \frac{T(N)}{f(N)}$
4. Find $c = c_{\min} + \Delta$ where Δ is usually some small integer value like 1.

5. Find n_0 by solving $T(n_0) \le cf(n_0)$.

Example: Consider the code fragment

sum=0;
for (*a*=*N*;*a*>0;*a*--) *sum*++;

Line	Code	Cost
1	sum=0;	1
2	a=N;	1
3	**while**(a>0) {	N
4	sum++;	N
5	a--;	N
6	}	1
	Total	**3N+3**

1. $T(N) = 3N + 3$
2. $f(N) = N$
3. $c_{\min} = \lim_{N \to \infty} \frac{3N+3}{N} = 3$
4. $c = 3 + 1 = 4$
5. $3N + 3 = 4N \Rightarrow N = n_0 = 3$

Example:
Find T(N) for the algorithm

```
sum=0;
for (a=0;a<N;a++) sum++;
```

Line	Code	Cost
1	sum=0;	1
2	a=0;	1
3	while (a<N) {	N
4	sum++;	N
5	a++;	N
6	}	1
	Total	**3N+3**

```
int af01(int N)
{    return 3*N+3;
}

int ef01(int N)
{    int a,sum,c=0;
     sum=0;          c++;
     a=0;                c++;
     while (a<N) {       c++;
```

```
        sum++;      c++;
        a++;            c++;
    }               c++;
    return c;
}
```