1. ***O*(*f*(*n*)):** *O*( ) is pronounced "Oh of" and suggests "Order of." The function *f*(*n*) characterizes the amount of time required to execute a particular algorithm. In particular, *f*(*n*) characterizes the rate of growth. Function *f*(*n*) is a measure of how much time is required to execute an algorithm. The argument, *n*, represents the number of input values processed by the algorithm. *O*(*f*(*n*)) is a way of characterizing the rate of growth for the time required to execute an algorithm.

   The function *O*(*f*(*n*)) is a ceiling on the rate of growth. The function *f*(*n*) grows no faster than *O*(*f*(*n*)).

   1.1. The argument *n*, is the number of inputs accepted by an algorithm.
   1.2. Function *f*(*n*) characterizes the rate of growth required by an algorithm to process n inputs.
   1.3. The function *O*(*f*(*n*)) is a ceiling on the rate of growth.

   Definition: $T(n) = O(f(n))$ if there are positive *constants* *c* and n_0 such that $T(n) \le cf(n)$ when $n \ge n_0$.
2. ***W*(*g*(*n*)):** *W*( ) is pronounced "Omega of." Function *W*( ) is similar to O( ) but specifies a floor to the rate of growth for an algorithm. *W*( ) specifies the best performance that can be expected from an algorithm. An algorithm always runs slower that the expression given by *W*( ). Function *g*(*n*) characterizes the rate of growth of the timing function for an algorithm. Function *g*(*n*) has the same function as function f(*n*).

   Definition: $T(n) = \Omega(g(n))$ if there are positive *constants* *c* and n_0 such that $T(n) \ge cf(n)$ when $n \ge n_0$.
3. *Q*(*h*(*n*)): *Q*( ) is pronounced "theta of." Function *Q*( ) describes the condition when the rate of growth for both the ceiling function O( ) and the floor function Ω( ) match.

   Definition: $T(n) = \Theta(h(n))$ if and only if $T(n) = O(h(n))$ and $T(N) = \Omega(h(n))$ when $n \ge n_0$.

Rules:

1. If $T_1(n) = O(f(n))$ *and* $T_2(n) = O(g(n))$, then
   1.1. $T_1(n) + T_2(n) = \max(O(f(n), O(g(n)))$
   1.2. $T_1(n) * T_2(n) = O(f(n) * g(n))$
2. If $T(n)$ is a polynomial of degree *k*, then $T(n) = \Theta(n^k)$.

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

1. Find $T(n)$.
2. Select $f(n)$
3. Find $c_{\min}$ and c
4. Find n_0

1. Find $T(n)$

Consider the code fragment

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

Line	Code	Cost
1	*sum*=**0;**	1
2	*a*=**0;**	1
3	**while (***a*<*N***) {**	$\sum_{a=0}^{N-1} 1$
4	*sum*++**;**	same as 3
5	*a*++**;**	same as 3
6	**}**	1

Open form: $T(n) = 3\sum_{a=0}^{N-1} 1 + 3$

Closed form: $T(n) = 3n + 3$

2. Select $f(n)$
   Let $f(n) = n$
3. Find $c_{\min}$ and c

$$c_{\min} = \lim_{n\to\infty} \frac{T(n)}{f(n)}$$

$$c_{\min} = \lim_{n\to\infty} \frac{3n+3}{n} = 3$$

$$c = c_{\min} + \partial$$

c=3+1=4

4. Find n_0

Solve $T(n) = cf(n)$

$3n+3=4n$

$n=3$

Function $f(n)$	**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

Table 1. Sample values for various $f(n)$

n	**O(*n*)**	**O(*n* log *n*)**	**O(*n***2)**	**O(*n***3)**	**O(2*****n*)**
1	10	0	1	2	2
2	20	2	4	12	4
3	30	5	9	36	8
4	40	8	16	80	16
5	50	12	25	150	32
6	60	16	36	252	64
7	70	20	49	392	128
8	80	24	64	576	256
9	90	29	81	810	512
10	100	33	100	1000	1024

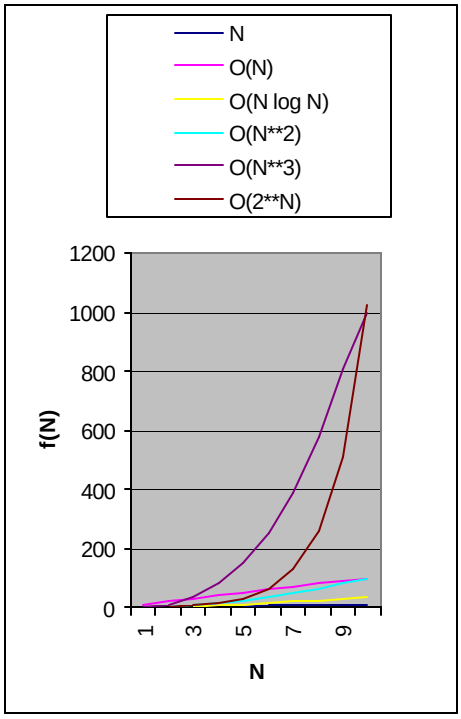

Figure 1. Sample values for various $f(n)$

Goal: We want to verify our analytical estimation of time complexity. Our solution is to design two functions. One function produces time complexity using our analytical model and the other function computes an empirical value.

```
int analytical(int n)
{    return 3*n+3;
}

int empirical(int N)
{    int t=0;
     int a,sum;
     sum=0;              t++;
     a=0;                t++;
     while (a<n) {       t++;
          sum++;         t++;
          a++;           t++;
     }                   t++;
     return t;
}
```

Example 2: Binary Search

```
//-------------------------------------------------------------------------------------------------
//class List defines the attributes of a list of integers.
//Integers are stored in ascending order.  Element L[0] is a sentinel.  The sentinel
//is the smallest integer.  The sentinel is not in the list.
//-------------------------------------------------------------------------------------------------
class List {
    int size;       //Number of available elements
    int N;          //Number of occupied elements
    int* L;         //Points to storage for the list
public:
    List(int sz=100);
    ~List()
    class FullException { };
    class EmptyException { };
    bool IsFull(void);
    bool IsEmpty(void);
    void Insert(int key);
    void Remove(int key);
    int Search(int key)
};
```

```
Line    Code
1       int List::Search(int key)
2       {  int lo=1, hi=N;
3          while (lo<=hi) {
4             int m=(lo+hi)/2;
5             if (key==L[m]) return m;
6             if (key<L[m])
7                hi=m-1;
8             else
9                lo=m+1;
10         }
11         return 0;
12      }
```

L[0]	L[1]	L[2]	L[3]	L[4]	L[5]	L[6]	L[7]	L[8]
	11	22	33	44	55	66	77	88

lo, hi, lo<=hi, m key L[m] key==L[m] key<L[m] m-1 key>L[m] m+1

lo	*hi*	*lo*<=*hi*	*m*	*key*	*L*[*m*]	*key* == *L*[*m*]	*key* < *L*[*m*]	*m*-1	*key* > *L*[*m*]	*m*+1
1	8	yes	4	33	44	no	yes	3		
1	3	yes	2	33	22	no	no		yes	3
3	3	yes	3	33	33	yes				

lo	*hi*	*lo*<=*hi*	*m*	*key*	*L*[*m*]	*key* == *L*[*m*]	*key* < *L*[*m*]	*m*-1	*key* > *L*[*m*]	*m*+1
1	8	yes	4	34	44	no	yes	3		
1	3	yes	2	34	22	no	no		yes	3
3	3	yes	3	34	33	no			yes	4
4	3	no								

Algorithm Analysis of Binary Search

Line	Code	Cost
1	**int** *List*::*Search*(**int** *key*)	0
2	{ **int** *lo*=**1**, *hi*=*N*;	2
3	**while** (*lo*<=*hi*) {	*k*
4	**int** *m*=(*lo*+*hi*)/**2**;	3*k*
5	**if** (*key*==*L*[*m*]) **return** *m*;	*k*
6	**if** (*key*<*L*[*m*])	*k*
7	*hi*=*m*-**1**;	2*k*
8	**else**	0
9	*lo*=*m*+**1**;	2*k*
10	}	1
11	**return 0**;	0
12	}	0
	Total	***8k+3***

Let *k* be the number of times the while-loop on lines 3 through 10 iterates.

The number of elements that could match the `key` is halved every iteration. Eventually a single element is selected. If the last element does not match the `key` then the while-loop terminates. Variable *lo* exceeds variable *hi*.

1. Let n be the number of elements in the list to search. Let $n = n_0$ the number of elements to search initially.
2. The number of elements to search after the first iteration is no more than $n_1 = \frac{n_0}{2}$.
3. In a similar way the number of elements to search after the second iteration is no more than $n_2 = \frac{n_1}{2} = \frac{n_0}{2^2}$.
4. In general we define the recurrence relation $n_{i+1} = \frac{n_i}{2}, n_0 = n$.
5. The solution to the recurrence relation is $n_i = \frac{n}{2^i}$
6. We know that eventually the number of elements to search will be reduced to one on, say, the k^{th} iteration, $n_k = 1$.
7. Further, $n_k = \frac{n}{2^k} = 1$
8. $1 \le \frac{n}{2^k}$
9. $2^k \le n \Rightarrow k \le \log_2 n$

The largest value of *k* is $\log_2 n$

1. $T(n) = 8\log_2 n + 3$
2. $f(n) = \log_2 n$

3. $c_{\min} = \lim_{n\to\infty} \frac{T(n)}{f(n)} = \lim_{n\to\infty} \frac{8\log_2 n + 3}{\log_2 n} = 8,\ c = 9$
4. $T(n_0) \le cf(n_0)$
   4.1. $8\log_2 n_0 + 3 \le 9\log_2 n_0$
   4.2. $\log_2 n_0 \ge 3$
   4.3. $n_0 \ge 2^3$