Time complexity :

Big O notation
f(n) = O(g(n)) means
There are positive constants c and k such that:
0<= f(n) <= c*g(n) for all n >= k.

For large problem sizes the dominant term(one with highest value of exponent) almost completely determines the value of the complexity expression. So abstract complexity is expressed in terms of the dominant term for large N. Multiplicative constants are also ignored.

N^2 + 3N + 4 is O(N^2)
since for N>4, N^2 + 3N + 4 < 2N^2 (c=2 & k=4)

O(1)- constant time
This means that the algorithm requires the same fixed number of steps regardless of the size of the task.
Example:
1)a statement involving basic operations
Here are some examples of basic operations.
    -one arithmetic operation (eg., +, *)
    -one assignment
    -one test (eg., x==0)
    -one read(accessing an element from an array)

2) Sequence of statements involving basic operations.

```
statement 1;
statement 2;
..........
statement k;
```

Time for each statement is constant and the total time is also constant: O(1)

O(n)- linear time
This means that the algorithm requires a number of steps proportional to the size of the task.

Examples:
1. Traversing an array.
2. Sequential/Linear search in an array.
3. Best case time complexity of Bubble sort (i.e when the elements of array are in sorted order).

Basic strucure is :

```
for (i = 0; i < N; i++) {
    sequence of statements of O(1)
}
```

The loop executes N times, so the total time is N*O(1) which is O(N).

O(n^2) - quadratic time
The number of operations is proportional to the size of the task squared.

Examples:
1) Worst case time complexity of Bubble, Selection and Insertion sort.

Nested loops:
1.
```
for (i = 0; i < N; i++) {
   for (j = 0; j < M; j++) {
      sequence of statements of O(1)
   }
}
```
The outer loop executes N times and inner loop executes M times so the time complexity is O(N*M)

2.
```
for (i = 0; i < N; i++) {
   for (j = 0; j < N; j++) {
      sequence of statements of O(1)
   }
}
```
Now the time complexity is O(N^2)

3.let's consider nested loops where the number of iterations of the inner loop depends on the value of the outer loop's index.

```
for (i = 0; i < N; i++) {
   for (j = i+1; j < N; j++) {
      sequence of statements of O(1)
   }
}
```
Let us see how many iterations the inner loop has:

Value of i	Number of iterations of inner loop
0	N-1
1	N-2
.....	....
N-3	2
N-2	1
N-1	0

So the total number of times the “sequence of statements” within the two loops executes is :
(N-1)+(N-2)+.....2+1+0 which is N*(N-1)/2 or (1/2)*(N^2) - (1/2)* N
and we can say that it is O(N^2) (we can ignore multiplicative constant and for large problem size the dominant term determines the time complexity)

O(log n) - logarithmic time
Examples:
1. Binary search in a sorted array of n elements.

O(n log n) - "n log n " time
Examples:
1. MergeSort, QuickSort etc.

O(a^n)(a>1)- exponential time
Examples:
1. Recursive Fibonacci implementation
2. Towers of Hanoi

The best time in the above list is obviously constant time, and the worst is exponential time which, as we have seen, quickly overwhelms even the fastest computers even for relatively small n. **Polynomial** growth (linear, quadratic, cubic, etc.) is considered manageable as compared to exponential growth.

Using the "<" sign informally, we can say that the order of growth is
O(l) < O(log n) < O(n) < O(n log n) < O(n^2) < O(n^3) < O(a^n) where a>1

A word about Big-O when a function is the sum of several terms
If a function (which describes the order of growth of an algorithm) is a sum of several terms, its order of growth is determined by the fastest growing term. In particular, if we have a polynomial

$$p(n) = a_k n^k + a_{k-1} n^{k-1} + \ldots\ldots + a_1 n + a_0$$

its growth is of the order n^k:

$$p(n) = O(n^k)$$

Example:

```
{perform any statement S1}                    O(1)
for (i=0; i < n; i++)
{
        {perform any statement(s) S2}         O(n)

        {run through another loop n times}    O(n^2)
}
```

Total Execution Time: O(1) + O(n) +O(n^2)   therefore, O(n^2)

Statements with method calls:
When a statement involves a method call, the complexity of the statement includes the complexity of the method call. Assume that you know that method *f* takes constant time, and that method *g* takes time proportional to (linear in) the value of its parameter *k*. Then the statements below have the time complexities indicated.

```
f(k);  // O(1)
g(k);  // O(k)
```

When a loop is involved, the same rule applies. For example:

```
for (j = 0; j < N; j++) g(N);
```

has complexity (N^2). The loop executes N times and each method call `g(N)` is complexity O(N).

Other Examples

1. for (j = 0; j < N; j++) f(j);
   This is O(N) since f(j) is O(1) and it is executed N times.

2. for (j = 0; j < N; j++) g(j);
   The first time the loop executes j is 0 and g(0) takes "no operations". The next time j is 1 and g(1) takes 1 operations. The last time the loop executes j is N-1 and g(N-1) takes N-1 operations. The total time is the sum of the first N-1 numbers and is $O(N^2)$.

3. for (j = 0; j < N; j++) g(k);
   Each time through the loop g(k) takes k operations and the loop executes N times. Since you don't know the relative size of k and N, the overall complexity is O(N * k).

So why should we bother about time complexity?
Suppose time taken by one operation=1 micro sec.
Problem size N=100

Complexity	Time
N	1 micro sec.
N^2	0.01 sec.
N^4	100 sec.
2^N	4x10^16 years

Lets look at the problem for computing Fibonacci numbers :

A recursive solution:
```
public long Fib1(long n){
if ((n == 1 )||(n==2)) return 1;
return Fib1(n-1) + Fib1(n-2);
}
```

Running Time Analysis:
Let T(n) be the number of steps needed to compute F(n) (nth Fibonacci number)
We can see that
T(n) = T(n-1) + T(n-2) + 1 ....(i)
T(1) = T(2) = 1
T(n) is exponential in n. It takes approximately 2^(0.7n) steps to compute F(n). The proof is out of the scope of this course. F(200) will take about 2^(140) steps which is even more than the life of universe!!!!

What takes so long?
The same subproblems get solved over and over again!

$$\begin{tikzcd}
 & & & F(5) \arrow[dl] \arrow[drr] & & & \\
 & & F(4) \arrow[dl] \arrow[dr] & & & F(3) \arrow[dl] \arrow[dr] & \\
 & F(3) \arrow[dl] \arrow[dr] & & F(2) & F(2) & & F(1) \\
F(2) & & F(1) & & & &
\end{tikzcd}$$

A better solution:
Solve F1, F2, ..., Fn. Solve them in order and save their values!

```
Function Fib2(n){
    Create an array fib[1....n]
    fib[1] = 1
    fib[2] = 1
    for i = 3 to n:
        fib[i] = fib[i-1] + fib[i-2]
    return fib[n]
}
```

The time complexity of this algorithm is O(n). The number of steps required is proportional to n. F(200) is now reasonable so is F(2000) and F(20000).

Moral: The right algorithm makes all the difference

Some Important Recurrence Relations :

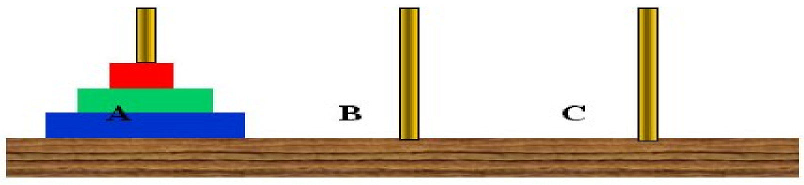
Recurrence	Algorithm	Big-Oh Solution
$T(n) = T(n/2) + O(1)$	Binary Search	$O(\log n)$
$T(n) = T(n-1) + O(1)$	Sequential Search	$O(n)$
$T(n) = T(n-1) + O(n)$	Selection Sort (other n^2 sorts in worst case)	$O(n^2)$
$T(n) = 2*T(n/2) + O(n)$	Merge Sort & Quicksort	$O(n \log n)$

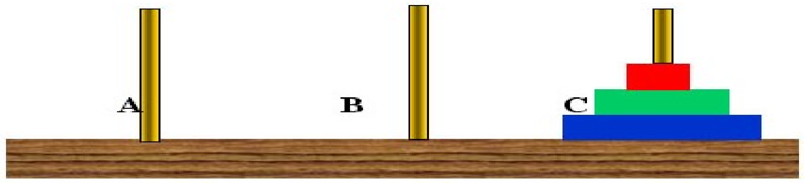
Tower of Hanoi

The **Tower of Hanoi** is a mathematical puzzle. It consists of three rods, and a number of disks of different sizes which can slide onto any rod. The puzzle starts with the disks neatly stacked in order of size on one rod, the smallest at the top, thus making a conical shape.

The objective of the puzzle is to move the entire stack to another rod, obeying the following rules:

- Only one disk may be moved at a time.
- Each move consists of taking the upper disk from one of the rods and sliding it onto another rod, on top of the other disks that may already be present on that rod.
- No disk may be placed on top of a smaller disk.





We want to write a recursive method, THanoi(n,A,B,C) which moves n disks from peg A to peg C using peg B for intermediate transfers.

Observation: THanoi(n, A, B, C) is equivalent to performing following tasks:

THanoi(n-1, A, C, B) (which means moving n-1 disks from A to B using C)
Transferring the largest disk from peg A to peg C
THanoi(n-1, B, A, C) (which means moving n-1 disks from B to C using A)

Stopping condition: n = 1

The time complexity of above algorithm can be determined using following recurrence relation.
Let T(n) be the number of steps required to solve the puzzle for n disks. It is clearly evident from the above observation that the soluiton for n disks is equivalent to- solving the puzzle two times for n-1 disks and a single step involving transfer of disk from starting 'peg' to final 'peg' which takes constant time.
Thus,
$T(n) = T(n-1) + T(n-1) + O(1)= 2*T(n-1) + O(1)$
The solution to this recurrence relation is exponential in n and so T(n) is of exponential order. The proof is out of scope of this course.

This is an example where recursion is much easier to formulate than a loop based solution.