```
void Function(int n) {
        int i=1, s=1;
        while( s <= n) {
                i++;
                s= s+i;
                printf("*");
        }
}
```

Solution: Consider the comments in the below function:

```
void Function (int n) {
  int i=1, s=1;
  // s is increasing not at rate 1 but i
  while( s <= n) {
    i++;
    s= s+i;
    printf("*");
  }
}
```

We can define the 's' terms according to the relation $s_i = s_{i-1} + i$. The value oft' increases by 1 for each iteration. The value contained in 's' at the i^{th} iteration is the sum of the first '('positive integers. If k is the total number of iterations taken by the program, then the *while* loop terminates if:

$$1 + 2 + \ldots + k = \frac{k(k+1)}{2} > n \Rightarrow k = O(\sqrt{n}).$$

Problem-24 Find the complexity of the function given below.

```
void function(int n) {
        int i, count =0;
        for(i=1; i*i<=n; i++)
                count++;

}
```

Solution:

```
void function(int n) {
    int i, count =0;
    for(i=1; i*i<=n; i++)
        count++;
}
```

In the above-mentioned function the loop will end, if $i^2 > n \Rightarrow T(n) = O(\sqrt{n})$. This is similar to Problem-23.

Problem-25 What is the complexity of the program given below:

```
void function(int n) {
    int i, j, k , count =0;
    for(i=n/2; i<=n; i++)
        for(j=1; j + n/2<=n; j= j+1)
            for(k=1; k<=n; k= k * 2)
                count++;
}
```

Solution: Consider the comments in the following function.

```
void function(int n) {
    int i, j, k , count =0;
     //outer loop execute n/2 times
    for(i=n/2; i<=n; i++)
        //middle loop executes n/2 times
        for(j=1; j + n/2<=n; j= j+1)
            //inner loop execute logn times
            for(k=1; k<=n; k= k * 2)
                count++;
}
```

The complexity of the above function is $O(n^2logn)$.

Problem-26 What is the complexity of the program given below:

```
void function(int n) {
        int i, j, k , count =0;
        for(i=n/2; i<=n; i++)
                for(j=1; j<=n; j= 2 * j)
                        for(k=1; k<=n; k= k * 2)
                                count++;
}
```

Solution: Consider the comments in the following function.

```
void function(int n) {
    int i, j, k , count =0;
    //outer loop execute n/2 times
    for(i=n/2; i<=n; i++)
        //middle loop executes logn times
        for(j=1; j<=n; j= 2 * j)
            //inner loop execute logn times
            for(k=1; k<=n; k= k*2)
                count++;
}
```

The complexity of the above function is $O(nlog^2n)$.

Problem-27 Find the complexity of the program below.

```
function( int n ) {
        if(n == 1) return;
        for(int i = 1 ; i <= n ; i + + ) {
                for(int j= 1 ; j <= n ; j + + ) {
                        printf("*" );
                        break;
                }
        }
}
```

Solution: Consider the comments in the function below.

```
function( int n ) {
   //constant time
   if( n == 1 ) return;
   //outer loop execute n times
   for(int i = 1 ; i <= n ; i + + ) {
      // inner loop executes only time due to break statement.
      for(int j= 1 ; j <= n ; j + + ) {
         printf("*" );
         break;
      }
   }
}
```

The complexity of the above function is O(n). Even though the inner loop is bounded by n, due to the break statement it is executing only once.

Problem-28 Write a recursive function for the running time T(n) of the function given below. Prove using the iterative method that $T(n) = \Theta(n^3)$.

```
function( int n ) {
        if( n == 1 ) return;
        for(int i = 1 ; i <= n ; i + + )
                for(int j = 1 ; j <= n ; j + + )
                        printf("*");
        function( n-3 );
}
```

Solution: Consider the comments in the function below:

```
function (int n) {
    //constant time
    if( n == 1 ) return;
    //outer loop execute n times
    for(int i = 1 ; i <= n ; i + + )
      //inner loop executes n times
      for(int j = 1 ; j <= n ; j + + )
        //constant time
        printf("*" ) ;
      function( n-3 );
}
```

The recurrence for this code is clearly $T(n) = T(n-3) + cn^2$ for some constant $c > 0$ since each call prints out n^2 asterisks and calls itself recursively on $n - 3$. Using the iterative method we get: $T(n) = T(n-3) + cn^2$. Using the *Subtraction and Conquer* master theorem, we get $T(n) = \Theta(n^3)$.

Problem-29 Determine Θ bounds for the recurrence relation: $T(n) = 2T\left(\frac{n}{2}\right) + nlogn$

Solution: Using Divide and Conquer master theorem, we get $O(nlog^2n)$.

Problem-30 Determine Θ bounds for the recurrence: $T(n) = T\left(\frac{n}{2}\right) + T\left(\frac{n}{4}\right) + T\left(\frac{n}{8}\right) + n$

Solution: Substituting in the recurrence equation, we get: $T(n) \leq c1 * \frac{n}{2} + c2 * \frac{n}{4} + c3 * \frac{n}{8} + cn \leq k * n$, where k is a constant. This clearly says $\Theta(n)$.

Problem-31 Determine Θ bounds for the recurrence relation: $T(n) = T(\lceil n/2 \rceil) + 7$.

Solution: Using Master Theorem we get: $\Theta(logn)$.

Problem-32 Prove that the running time of the code below is $\Omega(logn)$.

```
void Read(int n) {
        int k = 1;
        while( k < n )
                k = 3*k;
}
```

Solution: The *while* loop will terminate once the value of 'k' is greater than or equal to the value of 'n'. In each iteration the value of 'k' is multiplied by 3. If i is the number of iterations, then 'k' has the value of 3^i after i iterations. The loop is terminated upon reaching i iterations when $3^i \geq n$

$\therefore\ i \geq \log_3 n$, which shows that $i = \Omega(logn)$.

Problem-33 Solve the following recurrence.

$$T(n) = \begin{cases} 1, & if\ n = 1 \\ T(n-1) + n(n-1), & if\ n \geq 2 \end{cases}$$

Solution: By iteration:

$$T(n) = T(n-2) + (n-1)(n-2) + n(n-1)$$

$$\ldots$$

$$T(n) = T(1) + \sum_{i=1}^{n} i(i-1)$$

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

$$T(n) = 1 + \frac{n((n+1)(2n+1)}{6} - \frac{n(n+1)}{2}$$

$$T(n) = \Theta(n^3)$$

Note: We can use the *Subtraction and Conquer* master theorem for this problem.

Problem-34 Consider the following program:

```
Fib[n]
if(n==0) then return 0
else if(n==1) then return 1
else return Fib[n-1]+Fib[n-2]
```

Solution: The recurrence relation for the running time of this program is: $T(n) = T(n-1) + T(n-2) + c$. Note $T(n)$ has two recurrence calls indicating a binary tree. Each step recursively calls the program for n reduced by 1 and 2, so the depth of the recurrence tree is $O(n)$. The number of leaves at depth n is 2^n since this is a full binary tree, and each leaf takes at least $O(1)$ computations for the constant factor. Running time is clearly exponential in n and it is $O(2^n)$.

Problem-35 Running time of following program?

```
function(n) {
        for(int i = 1; i <= n ; i + + )
                for(int j = 1 ; j <= n ; j+ = i )
                        printf(" * ");
}
```

Solution: Consider the comments in the function below:

```
function (n) {
   //this loop executes n times
   for(int i = 1;  i <= n ; i + + )
      //this loop executes j times with j increase by the rate of i
      for(int j = 1 ; j <= n ; j+ = i )
         printf( " * " );
}
```

In the above code, inner loop executes *n/i* times for each value of *i*. Its running time is $n \times (\sum_{i=1}^{n} n/i) = O(nlogn)$.

Problem-36 What is the complexity of $\sum_{i=1}^{n} log\, i$?

Solution: Using the logarithmic property, $logxy = logx + logy$, we can see that this problem is equivalent to

$$\sum_{i=1}^{n} logi = log\,1 + log\,2 + \cdots + log\,n = log(1 \times 2 \times \ldots \times n) = log(n!) \leq log(n^n) \leq nlogn$$

This shows that the time complexity = O(*nlogn*).

Problem-37 What is the running time of the following recursive function (specified as a function of the input value *n*)? First write the recurrence formula and then find its complexity.

```
function(int n) {
        if(n <= 1) return;
        for (int i=1 ; i <= 3; i++ )
                f(⌈n/3⌉);
}
```

Solution: Consider the comments in the below function:

```
function (int n) {
   //constant time
   if(n <= 1) return;
   //this loop executes with recursive loop of n/3 value
   for (int i=1 ; i <= 3; i++ )
      f(⌈n/3⌉);
}
```

We can assume that for asymptotical analysis $k = \lceil k \rceil$ for every integer $k \geq 1$. The recurrence for this code is $T(n) = 3T(\frac{n}{3}) + \Theta(1)$. Using master theorem, we get $T(n) = \Theta(n)$.

Problem-38 What is the running time of the following recursive function (specified as a function of the input value n)? First write a recurrence formula, and show its solution using induction.

```
function(int n) {
        if(n <= 1) return;

        for (int i=1 ; i <= 3 ; i++ )
                function (n - 1).
}
```

Solution: Consider the comments in the function below:

```
function (int n) {
   //constant time
   if(n <= 1) return;
   //this loop executes 3 times with recursive call of n-1 value
   for (int i=1 ; i <= 3 ; i++ )
      function (n - 1).
}
```

The *if* statement requires constant time [O(1)]. With the *for* loop, we neglect the loop overhead and only count three times that the function is called recursively. This implies a time complexity recurrence:

$$T(n) = c, if\ n \leq 1;$$
$$= c + 3T(n-1), if\ n > 1.$$

Using the *Subtraction and Conquer* master theorem, we get $T(n) = \Theta(3^n)$.

minimum 1 time. Therefore, T(n) = O($\sqrt{n}$) and T(n) = Ω(1).

Problem-55 In the following C function, let $n \geq m$. How many recursive calls are made by this function?

```
int gcd(n,m){
        if (n%m ==0)
            return m;
        n = n%m;
        return gcd(m,n);
}
```

(A) $\Theta(log_2^n)$
(B) $\Omega(n)$
(C) $\Theta(log_2 log_2^n)$
(D) $\Theta(n)$

Solution: No option is correct. Big O notation describes the tight upper bound and Big Omega notation describes the tight lower bound for an algorithm. For m = 2 and for all n = 2^i, the running time is O(1) which contradicts every option.

Problem-56 Suppose $T(n) = 2T(n/2) + n$, T(O)=T(1)=1. Which one of the following is false?
(A) $T(n) = O(n^2)$
(B) $T(n) = \Theta(nlogn)$
(C) $T(n) = Q(n^2)$
(D) $T(n) = O(nlogn)$

Solution: (C). Big O notation describes the tight upper bound and Big Omega notation describes the tight lower bound for an algorithm. Based on master theorem, we get $T(n) = \Theta(nlogn)$. This indicates that tight lower bound and tight upper bound are the same. That means, O($nlogn$) and Ω($nlogn$) are correct for given recurrence. So option (C) is wrong.

Problem-57 Find the complexity of the below function:

```
function(int n) {
    for (int i = 0; i<n; i++)
        for(int j=i; j<i*i; j++)
            if (j %i == 0){
                for (int k = 0; k < j; k++)
                    printf(" * ");
            }

}
```

Solution:

```
function(int n) {
    for (int i = 0; i<n; i++)              // Executes n times
        for(int j=i; j<i*i; j++)           // Executes n*n times
            if (j %i == 0){
                for (int k = 0; k < j; k++)    // Executes j times = (n*n) times
                    printf(" * ");
            }
}
```

Time Complexity: $O(n^5)$.

Problem-58 To calculate 9^n, give an algorithm and discuss its complexity.

Solution: Start with 1 and multiply by 9 until reaching 9^n.

Time Complexity: There are $n - 1$ multiplications and each takes constant time giving a $\Theta(n)$ algorithm.

Problem-59 For Problem-58, can we improve the time complexity?

Solution: Refer to the *Divide and Conquer* chapter.

Problem-60 Find the time complexity of recurrence $T(n) = T(\frac{n}{2}) + T(\frac{n}{4}) + T(\frac{n}{8}) + n$.

Solution: Let us solve this problem by method of guessing. The total size on each level of the recurrance tree is less than n, so we guess that $f(n) = n$ will dominate. Assume for all $i < n$ that $c_1 n \le T(i) < c_2 n$. Then,

$$\left(\frac{1}{4}n\right)^2 + \left(\frac{1}{3}n\right)^2 + \left(\frac{1}{3}\right)n^2 + \left(\frac{4}{9}\right)n^2 = \frac{625}{1296}n^2 = \left(\frac{25}{36}\right)^2 n^2$$

Similarly the amount of work at level k is at most $\left(\frac{25}{36}\right)^k n^2$.

Let $\alpha = \frac{25}{36}$, the total runtime is then:

$$\begin{aligned}
T(n) &\leq \sum_{k=0}^{\infty} \alpha^k n^2 \\
&= \frac{1}{1-\alpha} n^2 \\
&= \frac{1}{1-\frac{25}{36}} n^2 \\
&= \frac{1}{\frac{11}{36}} n^2 \\
&= \frac{36}{11} n^2 \\
&= O(n^2)
\end{aligned}$$

That is, the first level provides a constant fraction of the total runtime.

Problem-62 Rank the following functions by order of growth: $(n + 1)!$, $n!$, 4^n, $n \times 3^n$, $3^n + n^2 + 20n$, $(\frac{3}{2})^n$, $n^2 + 200$, $20n + 500$, 2^{lgn}, $n^{2/3}$, 1.

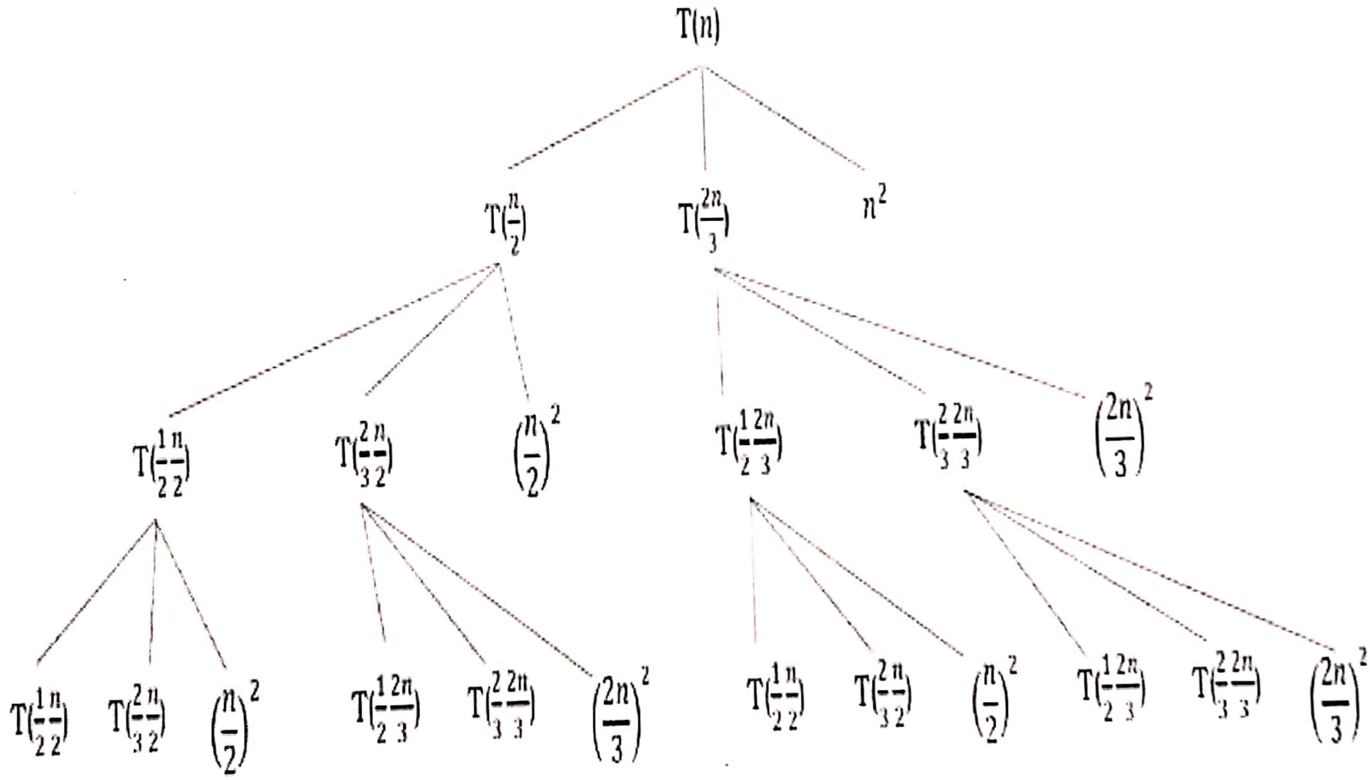
Solution:

$$c_1\frac{n}{2} + c_1\frac{n}{4} + c_1\frac{n}{8} + kn \quad \le T(n) \le \quad c_2\frac{n}{2} + c_2\frac{n}{4} + c_2\frac{n}{8} + kn$$
$$c_1 n(\frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \frac{k}{c_1}) \quad \le T(n) \le \quad c_2 n(\frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \frac{k}{c_2})$$
$$c_1 n(\frac{7}{8} + \frac{k}{c_1}) \quad \le T(n) \le \quad c_2 n(\frac{7}{8} + \frac{k}{c_2})$$

If $c_1 \ge 8k$ and $c_2 \le 8k$, then $c_1 n = T(n) = c_2 n$. So, $T(n) = \Theta(n)$. In general, if you have multiple recursive calls, the sum of the arguments to those calls is less than n (in this case $\frac{n}{2} + \frac{n}{4} + \frac{n}{8} < n$), and $f(n)$ is reasonably large, a good guess is $T(n) = \Theta(f(n))$.

Problem-61 Solve the following recurrence relation using the recursion tree method: $T(n)=T(\frac{n}{2}) + T(\frac{2n}{3}) + n^2$.

Solution: How much work do we do in each level of the recursion tree?



In level 0, we take n^2 time. At level 1, the two subproblems take time:

$$\left(\frac{1}{2}n\right)^2 + \left(\frac{2}{3}n\right)^2 = \left(\frac{1}{4}+\frac{4}{9}\right)n^2 = \left(\frac{25}{36}\right)n^2$$

At level 2 the four subproblems are of size $\frac{1}{2}\frac{n}{2}$, $\frac{2}{3}\frac{n}{2}$, $\frac{1}{2}\frac{2n}{3}$ and $\frac{2}{3}\frac{2n}{3}$ respectively. These two subproblems take time:

Function	Rate of Growth	
$(n+1)!$	$O(n!)$	
$n!$	$O(n!)$	
4^n	$O(4^n)$	
$n \times 3^n$	$O(n3^n)$	
$3^n + n^2 + 20n$	$O(3^n)$	
$(\frac{3}{2})^n$	$O((\frac{3}{2})^n)$	Decreasing rate of growths
$4n^2$	$O(n^2)$	
4^{lgn}	$O(n^2)$	
$n^2 + 200$	$O(n^2)$	
$20n + 500$	$O(n)$	
2^{lgn}	$O(n)$	
$n^{2/3}$	$O(n^{2/3})$	
1	$O(1)$	

Problem-63 Find the complexity of the below function:

```
function(int n) {
    int sum = 0;
    for (int i = 0; i<n; i++)
       if (i>j)
          sum = sum +1;
       else {
            for (int k = 0; k < n; k++)
                sum = sum -1;
        }
    }
}
```

Solution: Consider the worst-case.