Amortized analysis refers to determining the time-averaged running time for a sequence of operations. It is different from average case analysis, because amortized analysis does not make any assumption about the distribution of the data values, whereas average case analysis assumes the data are not "bad" (e.g., some sorting algorithms do well *on average* over all input orderings but very badly on certain input orderings). That is, amortized analysis is a worst-case analysis, but for a sequence of operations rather than for individual operations.

The motivation for amortized analysis is to better understand the running time of certain techniques, where standard worst case analysis provides an overly pessimistic bound. Amortized analysis generally applies to a method that consists of a sequence of operations, where the vast majority of the operations are cheap, but some of the operations are expensive. If we can show that the expensive operations are particularly rare we can *change them* to the cheap operations, and only bound the cheap operations.

The general approach is to assign an artificial cost to each operation in the sequence, such that the total of the artificial costs for the sequence of operations bounds the total of the real costs for the sequence. This artificial cost is called the amortized cost of an operation. To analyze the running time, the amortized cost thus is a correct way of understanding the overall running time – but note that particular operations can still take longer so it is not a way of bounding the running time of any individual operation in the sequence.

When one event in a sequence affects the cost of later events:

- One particular task may be expensive.
- But it may leave data structure in a state that the next few operations become easier.

Example: Let us consider an array of elements from which we want to find the k^{th} smallest element. We can solve this problem using sorting. After sorting the given array, we just need to return the k^{th} element from it. The cost of performing the sort (assuming comparison based sorting algorithm) is O(*nlogn*). If we perform n such selections then the average cost of each selection is O(*nlogn/n*) = O(*logn*). This clearly indicates that sorting once is reducing the complexity of subsequent operations.

1.28 Algorithms Analysis: Problems & Solutions

Note: From the following problems, try to understand the cases which have different complexities (O(*n*), O(*logn*), O(*loglogn*) etc.).

Problem-21 Find the complexity of the below recurrence:

$$T(n) = \begin{cases} 3T(n-1), if\ n > 0, \\ 1, \qquad otherwise \end{cases}$$

Solution: Let us try solving this function with substitution.

$T(n) = 3T(n-1)$

$T(n) = 3(3T(n-2)) = 3^2T(n-2)$

$T(n) = 3^2(3T(n-3))$

.

.

$T(n) = 3^nT(n-n) = 3^nT(0) = 3^n$

This clearly shows that the complexity of this function is $O(3^n)$.

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

Problem-22 Find the complexity of the below recurrence:

$$T(n) = \begin{cases} 2T(n-1) - 1, if\ n > 0, \\ 1, \qquad\qquad otherwise \end{cases}$$

Solution: Let us try solving this function with substitution.

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

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

$T(n) = 2^2(2T(n-3) - 2 - 1) - 1 = 2^3T(n-4) - 2^2 - 2^1 - 2^0$

$T(n) = 2^nT(n-n) - 2^{n-1} - 2^{n-2} - 2^{n-3} .... 2^2 - 2^1 - 2^0$

$T(n) = 2^n - 2^{n-1} - 2^{n-2} - 2^{n-3} .... 2^2 - 2^1 - 2^0$

$T(n) = 2^n - (2^n - 1)$ [*note:* $2^{n-1} + 2^{n-2} + \cdots + 2^0 = 2^n$]

$T(n) = 1$

∴ Time Complexity is O(1). Note that while the recurrence relation looks exponential, the solution to the recurrence relation here gives a different result.

Problem-23 What is the running time of the following function?

```
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 $O(n^2 logn)$.

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 $3^i \geq n$

$\leftrightarrow i \geq \log_3 n$, 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:

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

...

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

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

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

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

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

Problem-39 Write a recursion formula for the running time $T(n)$ of the function whose code is below.

```
function (int n) {
        if(n <= 1)  return;
        for(int i = 1; i < n; i + +)
                printf(" * ");
        function ( 0.8n ) ;
}
```

Solution: Consider the comments in the function below:

```
function (int n) {
   if(n <= 1)  return;   //constant time
   // this loop executes n times with constant time loop
   for(int i = 1; i < n; i + +)
      printf(" * ");
   //recursive call with 0.8n
   function ( 0.8n ) ;
}
```

The recurrence for this piece of code is $T(n) = T(.8n) + O(n) = T(4/5n) + O(n) = 4/5\ T(n) + O(n)$. Applying master theorem, we get $T(n) = O(n)$.

Problem-40 Find the complexity of the recurrence: $T(n) = 2T(\sqrt{n}) + logn$

Solution: The given recurrence is not in the master theorem format. Let us try to convert this to the master theorem format by assuming $n = 2^m$. Applying the logarithm on both sides gives, $logn = mlogl \Rightarrow m = logn$. Now, the given function becomes:

$$T(n) = T(2^m) = 2T\left(\sqrt{2^m}\right) + m = 2T\left(2^{\frac{m}{2}}\right) + m.$$

To make it simple we assume $S(m) = T(2^m) \Longrightarrow S(\frac{m}{2}) = T(2^{\frac{m}{2}}) \Longrightarrow S(m) = 2S\left(\frac{m}{2}\right) + m.$

Applying the master theorem format would result in $S(m) = O(mlogm)$.
If we substitute $m = logn$ back, $T(n) = S(logn) = O((logn)\ loglogn)$.

Problem-41 Find the complexity of the recurrence: $T(n) = T(\sqrt{n}) + 1$

Solution: Applying the logic of Problem-40 gives $S(m) = S\left(\frac{m}{2}\right) + 1$. Applying the master

theorem would result in $S(m) = O(logm)$. Substituting $m = logn$, gives $T(n) = S(logn) = O(loglogn)$.

Problem-42 Find the complexity of the recurrence: $T(n) = 2T(\sqrt{n}) + 1$

Solution: Applying the logic of Problem-40 gives: $S(m) = 2S\left(\frac{m}{2}\right) + 1$. Using the master theorem results $S(m) = O(m^{log_2^2})$. Substituting $m = logn$ gives $T(n) = O(logn)$.

Problem-43 Find the complexity of the below function.

```
int Function (int n) {
        if(n <= 2) return 1;
        else return (Function (floor(sqrt(n))) + 1);
}
```

Solution: Consider the comments in the function below:

```
int Function (int n) {
   if(n <= 2) return 1;          //constant time
   else                          // executes √n + 1 times
      return (Function (floor(sqrt(n))) + 1);
}
```

For the above code, the recurrence function can be given as: $T(n) = T(\sqrt{n}) + 1$. This is same as that of Problem-41.

Problem-44 Analyze the running time of the following recursive pseudo-code as a function of n.

```
void function(int n) {
        if( n < 2 ) return;
        else counter = 0;
        for i = 1 to 8 do
                function (n/2);
        for i =1 to n^3 do
                counter = counter + 1;
}
```

Solution: Consider the comments in below pseudo-code and call running time of function(n) as $T(n)$.

```
void function(int n) {
    if( n < 2 ) return;   //constant time
    else      counter = 0;
    // this loop executes 8 times with n value half in every call
    for i = 1 to 8 do
        function(n/2);
    // this loop executes n^3 times with constant time loop
    for i =1 to n^3 do
            counter = counter + 1;
}
```

$T(n)$ can be defined as follows:

$$T(n) = 1 \; if \; n < 2,$$
$$= 8T(\frac{n}{2}) + n^3 + 1 \; otherwise.$$

Using the master theorem gives: $T(n) = \Theta(n^{log_2^8} logn) = \Theta(n^3 logn)$.

Problem-45 Find the complexity of the below pseudocode:

```
temp = 1
repeat
        for i = 1 to n
                temp = temp + 1;
        n = n/2;
until n <= 1
```

Solution: Consider the comments in the pseudocode below:

```
temp = 1  //const time
repeat          // this loops executes n times
  for i = 1 to n
    temp = temp + 1;
    //recursive call with n/2 value
    n = n/2;
until n <= 1
```

The recurrence for this function is $T(n) = T(n/2) + n$. Using master theorem, we get $T(n) = O(n)$.

Problem-46 Running time of the following program?

```
function(int n) {
        for(int i = 1 ; i <= n ; i + + )
                for(int j = 1 ; j <= n ; j * = 2 )
                        printf( " * " );
}
```

Solution: Consider the comments in the below function:

```
function(int n) {
    for(int i = 1 ; i <= n ; i + + ) // this loops executes n times
        // this loops executes logn times from our logarithms guideline
        for(int j = 1 ; j <= n ; j * = 2 )
            printf( " * " );
}
```

Complexity of above program is: O(*nlogn*).

Problem-47 Running time of the following program?

```
function(int n) {
        for(int i = 1 ; i <= n/3 ; i + + )
                for(int j = 1 ; j <= n ; j += 4 )
                        printf( " * " );
}
```

Solution: Consider the comments in the below function:

```
function(int n) {       // this loops executes n/3 times
    for(int i = 1 ; i <= n/3 ; i + + )
        // this loops executes n/4 times
        for(int j = 1 ; j <= n ; j += 4)
            printf( " * " );
}
```

The time complexity of this program is: O(n^2).

Problem-48 Find the complexity of the below function:

```
void function(int n) {
    if(n <=  1) return;
    if(n > 1) {
            printf (" * ");
            function( n/2 );
            function( n/2 );
        }
}
```

Solution: Consider the comments in the below function:

```
void function(int n) {
    if(n <= 1) return; //constant time
    if(n > 1) {
            //constant time
            printf (" * ");
            //recursion with n/2 value
            function( n/2 );
            //recursion with n/2 value
            function( n/2 );
    }
}
```

The recurrence for this function is: $T(n) = 2T\left(\frac{n}{2}\right) + 1$. Using master theorem, we get $T(n) = O(n)$.

Problem-49 Find the complexity of the below function:

```
function(int n) {
        int i=1;
        while (i < n) {
                int j=n;
                while(j > 0)
                    j = j/2;
                i=2*i;
        } // i
}
```

Solution:

```
function(int n) {
        int i=1;
        while (i < n) {
                int j=n;
                while(j > 0)
                   j = j/2;   //logn code
                i=2*i; //logn times
        } // i
}
```

Time Complexity: O(*logn* * *logn*) = O($log^2 n$).

Problem-50 $\sum_{i \le k \le n} O(n)$, where O($n$) stands for order n is:

(A) O(n)
(B) O(n^2)
(C) O(n^3)
(D) O($3n^2$)
(E) O($1.5n^2$)

Solution: (B). $\sum_{i \le k \le n} O(n) = O(n) \sum_{i \le k \le n} 1 = O(n^2)$.

Problem-51 Which of the following three claims are correct?

I $(n + k)^m = \Theta(n^m)$, where k and m are constants
II $2^{n+1} = O(2^n)$
III $2^{2n+1} = O(2^n)$

(A) I and II
(B) I and III
(C) II and III
(D) I, II and III

Solution: (A). (I) $(n + k)^m = n^h + c1*n^{k-1} + \ldots k^m = \Theta(n^h)$ and (II) $2^{n+1} = 2*2^n = O(2^n)$

Problem-52 Consider the following functions:

f(n) = 2^n
g(n) = $n!$
h(n) = n^{logn}

Which of the following statements about the asymptotic behavior of f(n), g(n), and h(n) is true?

(A) f(n) = O(g(n)); g(n) = O(h(n))
(B) f(n) = Ω (g(n)); g(n) = O(h(n))

(C) $g(n) = O(f(n)); h(n) = O(f(n))$
(D) $h(n) = O(f(n)); g(n) = \Omega(f(n))$

Solution: (D). According to the rate of growth: $h(n) < f(n) < g(n)$ ($g(n)$ is asymptotically greater than $f(n)$, and $f(n)$ is asymptotically greater than $h(n)$). We can easily see the above order by taking logarithms of the given 3 functions: $lognlogn < n < log(n!)$. Note that, $log(n!) = O(nlogn)$.

Problem-53 Consider the following segment of C-code:

```
int j=1, n;
while (j <=n)
        j = j*2;
```

The number of comparisons made in the execution of the loop for any $n > 0$ is:
(A) $\text{ceil}(log_2^n) + 1$
(B) n
(C) $\text{ceil}(log_2^n)$
(D) $\text{floor}(log_2^n) + 1$

Solution: (a). Let us assume that the loop executes k times. After k^{th} step the value *of* j is 2^k. Taking logarithms on both sides gives $k = log_2^n$. Since we are doing one more comparison for exiting from the loop, the answer is $\text{ceil}(log_2^n) + 1$.

Problem-54 Consider the following C code segment. Let $T(n)$ denote the number of times the for loop is executed by the program on input n. Which of the following is true?

```
int IsPrime(int n){
    for(int i=2;i<=sqrt(n);i++)
    if(n%i == 0){
        printf("Not Prime\n");
        return 0;
    }
    return 1;
}
```

(A) $T(n) = O(\sqrt{n})$ and $T(n) = \Omega(\sqrt{n})$
(B) $T(n) = O(\sqrt{n})$ and $T(n) = \Omega(1)$
(C) $T(n) = O(n)$ and $T(n) = \Omega(\sqrt{n})$
(D) None of the above

Solution: (B). Big O notation describes the tight upper bound and Big Omega notation describes the tight lower bound for an algorithm. The *for* loop in the question is run maximum $\sqrt{n}$ times and

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) Ω(n)
(C) $\Theta(log_2 log_2^n)$
(D) Θ(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 $T(n) = T(\frac{n}{2}) + T(\frac{n}{4}) + T(\frac{n}{8}) + n$.

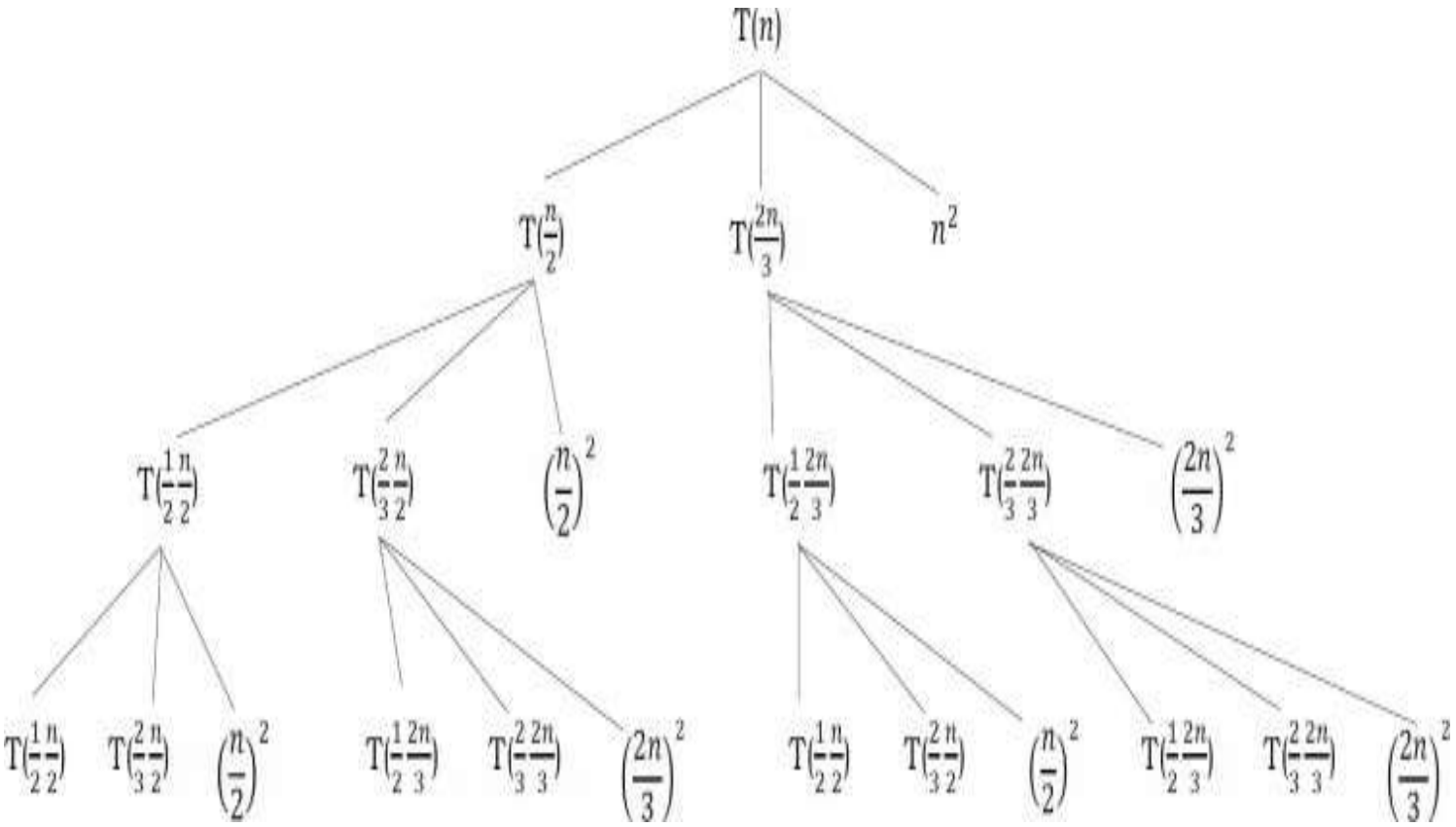
Solution: Let us solve this problem by method of guessing. The total size on each level of the recurrance tree is less than n, so we guess that $f(n) = n$ will dominate. Assume for all $i < n$ that $c_1 n \leq T(i) < c_2 n$. Then,

$$
\begin{aligned}
c_1\frac{n}{2} + c_1\frac{n}{4} + c_1\frac{n}{8} + kn \quad &\le T(n) \le \quad c_2\frac{n}{2} + c_2\frac{n}{4} + c_2\frac{n}{8} + kn \\
c_1 n\left(\frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \frac{k}{c_1}\right) \quad &\le T(n) \le \quad c_2 n\left(\frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \frac{k}{c_2}\right) \\
c_1 n\left(\frac{7}{8} + \frac{k}{c_1}\right) \quad &\le T(n) \le \quad c_2 n\left(\frac{7}{8} + \frac{k}{c_2}\right)
\end{aligned}
$$

If $c_1 \geq 8k$ and $c_2 \leq 8k$, 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:

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

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:

Function	Rate of Growth
$(n+1)!$	$O(n!)$
$n!$	$O(n!)$
4^n	$O(4^n)$
$n \times 3^n$	$O(n3^n)$
$3^n + n^2 + 20n$	$O(3^n)$
$(\frac{3}{2})^n$	$O((\frac{3}{2})^n)$
$4n^2$	$O(n^2)$
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	$O(1)$

Decreasing rate of growths

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.

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

Time Complexity: $O(n^2)$.

Problem-64 Can we say $3^{n^{0.75}} = O(3^n)$??

Solution: Yes: because $3^{n^{0.75}} < 3^{n^1}$

Problem-65 Can we say $2^{3n} = O(2^n)$?

Solution: No: because $2^{3n} = (2^3)^n = 8^n$ not less than 2^n.