Module 1

1 Analyse the complexity of the following function (3)

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

Answer: Time Complexity of the above function $O(n \log^2 n)$

```
void function(int n)
{ int count = 0;
   for (int i=n/2; i<=n; i++)                          ;Executes  O(n/2) times

       for (int j=1; j<=n; j = 2 * j)                  ;Executes  O(log n) times

              for (int k=1; k<=n; k = k * 2)           ;Executes  O(log n) times

                    count++;                        ;Executes  O(n/2* logn * log n) times
}
```

2 Solve using Iteration method $T(n)=2T(n/2)+n, T(1)=1$ (3)

Answer:

$T(n)=2T(n/2)+n$

$=2[2T(n/4)+(n/2)]+n$

$=2^2 T(n/4)+n+n=2^2 T(n/4)+2n$

$=2^2 [2T(n/8)+n/4]+2n$

$=2^3 T(n/8)+n+2n=2^3 T(n/8)+3n$

....After k times

$=2^k T(n/2^k)+kn$---eq1

Sub problem size is 1 after $n/2^k =1$

ie $2^k=n$,

$k=\log n$

then eq 1becomes

$T(n)=nT(1)+n \log n$

$=n*1+n\log n$

$=O(n\log n)$.

3 Analyse the complexity of the following functions (4)

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

Answer: Time Complexity of the above function O(n). The inner loop is bounded by n, but due to break statement it is executing only once.

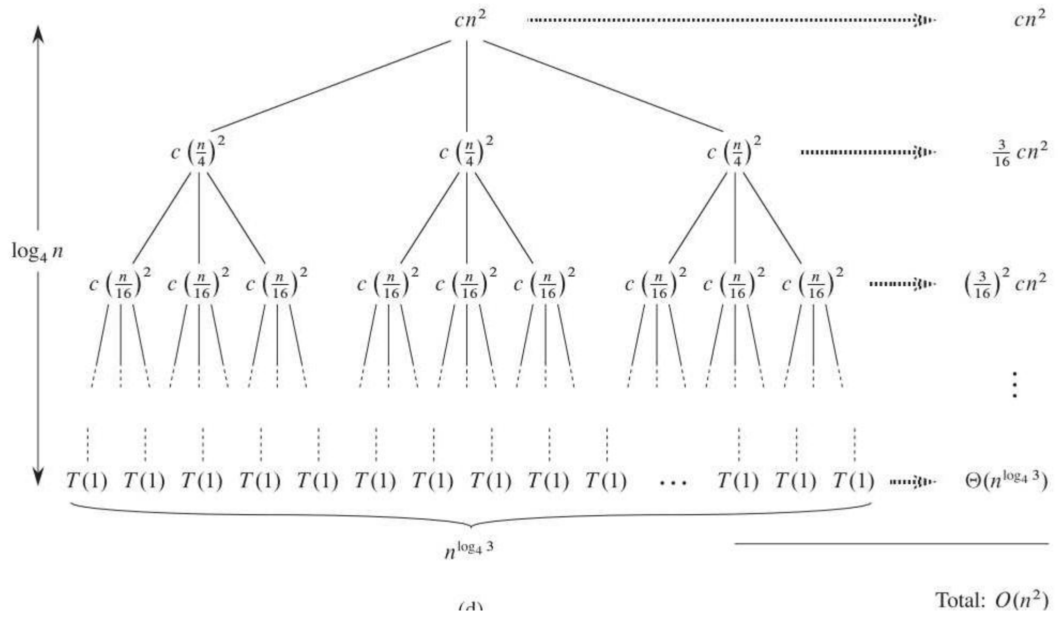
ii) void function(int n)

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

Answer: Time Complexity of the above function O(√n).
We can define the terms ‘s’ according to relation $s_i = s_{i-1} + i$. The value of ‘i’ increases by one for each iteration. The value contained in ‘s’ at the i^{th} iteration is the sum of the first ‘i’ positive integers. If k is total number of iterations taken by the program, then while loop terminates if: 1 + 2 + 3 ….+ k = [k(k+1)/2] > n So k = O(√n).

4 Solve using Recursion Tree method 5

$T(n)=3T(n/4)+n^2$



The fully expanded tree has height $\log_4 n$ (it has $\log_4 n + 1$ levels).
Sub problem size at depth i $=n/4^i$
Sub problem size is 1 when $n/4^i=1 \Rightarrow i=\log_4 n$
So, no. of levels $=1+ \log_4 n$
Cost of each level = (no. of nodes) x (cost of each node)

No. Of nodes at depth $i=3^i$

Cost of each node at depth $i=c\,(n/4^i)^2$

Cost of each level at depth $i=3^i c\,(n/4^i)^2 = (3/16)^i cn^2$

$T(n)= \sum_{i=0}^{\log_4 n} cn^2(3/16)^i$

$T(n)= \sum_{i=0}^{\log_4 n-1} cn^2(3/16)^i$ + cost of last level

Cost of nodes in last level $=3^i T(1)$

⇨ $c3^{\log_4 n}$ (at last level $i=\log_4 n$)

⇨ $cn^{\log_4 3}$

$$T(n)=\sum_{i=0}^{\log_4 n-1} cn^2(3/16)^i + c\,n^{\log_4 3}$$

$$<= cn^2\sum_{i=0}^{\infty}(3/16)^i + cn^{\log_4 3}$$

⇨ $<= cn^2*(16/13)+ cn^{\log_4 3} \Rightarrow T(n)=O(n^2)$

5 **Define the terms Best case, Worst case and Average case time complexities.**

There are 3 cases, in general, to find the complexity function f(n):
1. Best case: The minimum value for any possible input.
2. Worst case: The maximum value for any possible input.
3. Average case: The value of which is in between maximum and minimum for any possible input.

Best Case – 1 Mark, Worst Case – 1 Mark, Average Case – 1 Mark

6 **What is the smallest value of n such that an algorithm whose running times is $100n^2$ runs faster than an algorithm whose running time is 2^n on the same machine?**

A= $100n^2$ B= 2^n

Let's start checking from n=1 and go up for values of n which are power of 2.

$n=1\Rightarrow100\times1^2=100>2^1$; $n=2\Rightarrow100\times2^2=400>2^2$; $n=4\Rightarrow100\times4^2=1600>2^4$

$n=8\Rightarrow100\times8^2=6400>2^8$; $n=16\Rightarrow100\times16^2=25600<2^{16}$

Somewhere between 8 and 16, A starts to run faster than B. Let's do what we were doing but now we are going to try middle value of the range, repeatedly (binary search).

$n=(8+16)/2=12\Rightarrow100\times12^2=14400>2^{12}$

$n=(12+16)/2=14\Rightarrow100\times14^2=19600>2^{14}$

$n=(14+16)/2=15\Rightarrow100\times15^2=22500<2^{15}$

So, at n=15, A starts to run faster than B.

Minimum value of n=15 – 1 Mark. Steps – 2 Marks

7 **a) Determine the time complexities of the following two functions fun1() and fun2():**

```
int fun1(int n)
{
        if (n <= 1) return n;
        return 2*fun1(n-1);
}
int fun2(int n)
{
        if (n <= 1) return n;
        return fun2(n-1) + fun2(n-1);
}
```

Answer

Fun1 – O(n) - 1 Mark

Fun2 – $O(2^n)$ – 1 Mark

Time complexity of fun1() can be written as
T(n) = T(n-1) + C which is O(n)

Time complexity of fun2() can be written as
T(n) = 2T(n-1) + C which is O(2^n)

b) Find the solution to the recurrence equation using iteration method:

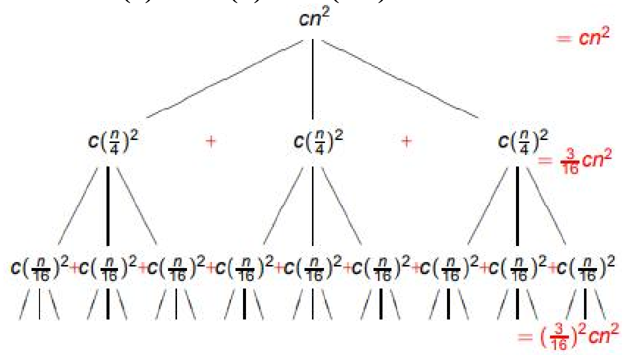
$T(2^k) = 3\ T(2^{k-1}) + 1$, T (1) = 1

$T(2^k) = 3\ T(2^{k-1}) + 1$
$= 3^2\ T(2^{k-2}) + 1 + 3$
$= 3^3\ T(2^{k-3}) + 1 + 3 + 9$
. . . (k steps of recursion (recursion depth))
$= 3^k\ T(2^{k-k}) + (1 + 3 + 9 + 27 + \ldots + 3^{k-1})$
$= 3^k + ((3^k - 1) / 2)$
$= ((2 * 3^k) + 3^k - 1)/2$
$= ((3 * 3^k) - 1) / 2$
$= O((3^{k+1} - 1) / 2)$

Solution – 1 Mark Steps – 2 marks.

c) Solve the recurrence using recursion tree method:

T(1) = 1 $T(n) = 3T(n/4) + cn^2$

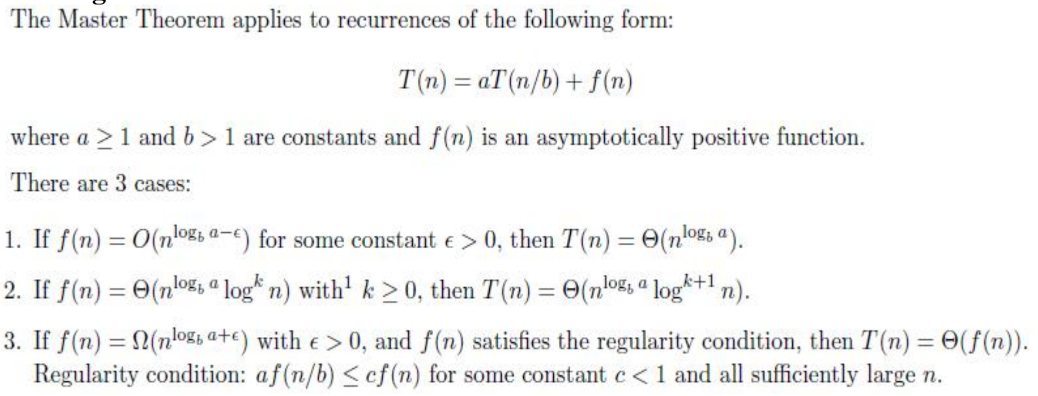


cn^2
$= cn^2$
$c(\frac{n}{4})^2$ + $c(\frac{n}{4})^2$ + $c(\frac{n}{4})^2$
$= \frac{3}{16}cn^2$
$c(\frac{n}{16})^2 + c(\frac{n}{16})^2 + c(\frac{n}{16})^2 + c(\frac{n}{16})^2 + c(\frac{n}{16})^2 + c(\frac{n}{16})^2 + c(\frac{n}{16})^2 + c(\frac{n}{16})^2 + c(\frac{n}{16})^2$
$= (\frac{3}{16})^2 cn^2$

So summing: $cn^2 + 3/16\ cn^2 + (3/16)^2\ cn^2 + \ldots = O(n^2)$

Solution - 1 Mark. Recursion Tree with Minimum 3 levels – 3 Marks

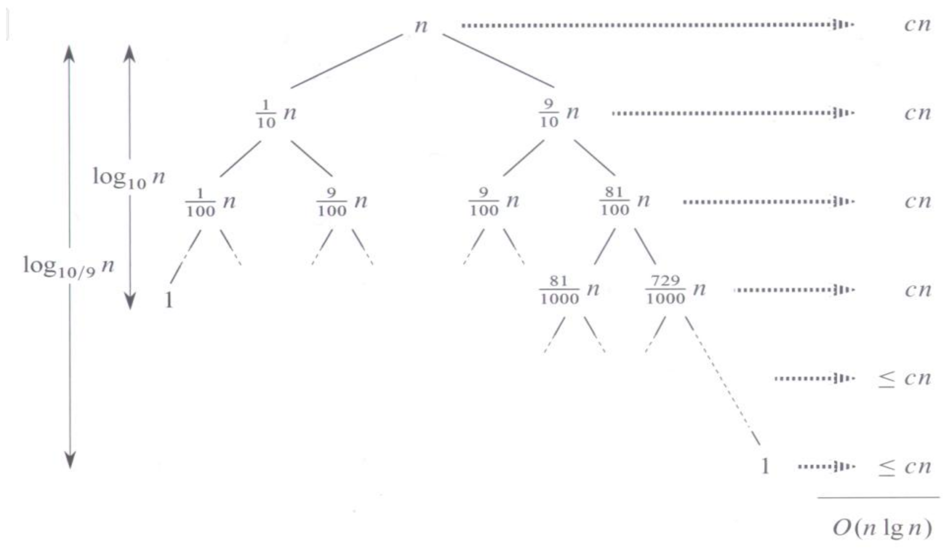
8	**a) Determine the best case and worst-case time complexity of the following function:**
	void fun(int n, int arr[])
	{
	int i = 0, j = 0;
	for(; i < n; ++i)
	while(j < n && arr[i] < arr[j])
	j++;
	}
	Answer:
	Both O(n)
	Best case Expression – 1.5 Marks, Worst case expression – 1.5 Marks
	In the first look, the time complexity seems to be O(n^2) due to two loops. But, the variable **j** is not initialized for each value of variable **i**. The while loop will execute only once for all value of i. So 0the inner loop runs at most **n** times.	

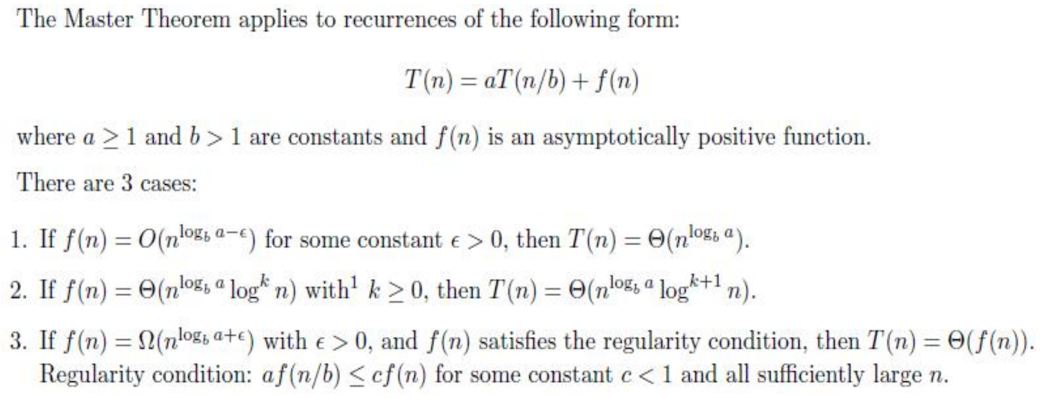
9	Is $2^{n+1} = O(2^n)$? Is $2^{2n} = O(2^n)$? Justify your answer. Answer: Yes ,justification -giving the constants, ***c*** and ***n0***, specified in the definition. For eg. ***c=2 n0=1*** is one solution.i.e.$2^{n+1} = O(2^n)$ can be written as $2^{n+1} <= c.2^n$,if c=2, $2^{n+1} = 2.2^n => 2^{n+1} = 2^{n+1}$ for all n>=n_0,where n_0 >=1 . (1.5 Marks) No ,justification -showing no constant could be derived for every ***n.*** For eg. Suppose $\boldsymbol{2^{2n} = O(2^n)}$. Then there is a constant ***c > 0*** such that ***c > 2***. Since $\boldsymbol{2^n}$ is unbounded, no such ***c*** can exist.(1.5 Marks)	(3)
10	**State Master's Theorem. Find the solution to the following recurrence equation using Master's theorem.** The Master Theorem applies to recurrences of the following form: $T(n) = aT(n/b) + f(n)$ where $a \geq 1$ and $b > 1$ are constants and $f(n)$ is an asymptotically positive function. There are 3 cases: 1. If $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$. 2. If $f(n) = \Theta(n^{\log_b a} \log^k n)$ with[1] $k \geq 0$, then $T(n) = \Theta(n^{\log_b a} \log^{k+1} n)$. 3. If $f(n) = \Omega(n^{\log_b a+\epsilon})$ with $\epsilon > 0$, and $f(n)$ satisfies the regularity condition, then $T(n) = \Theta(f(n))$. Regularity condition: $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n. **(1 mark)**	(3)

	a) T (n) = 2T (n/2)+ n log n **Solution-** We compare the given recurrence relation with $T(n) = aT(n/b) + \theta (n^k \log^p n)$. Then, we have-a = 2,b = 2,k = 1,p = 1 Now, a = 2 and $b^k = 2^1 = 2$. Clearly, $a = b^k$. So, we follow case-02. Since p = 1, so we have-$T(n) = \theta (n^{\log_b a}.\log^{p+1} n)$ $T(n) = \theta (n^{\log_2 2}.\log^{1+1} n)$ Thus, $\mathbf{T(n) = \theta (n\log^2 n)}$	1
11	a) $T (n) = 2^n T (n/2) + n^n$ Master method does not apply, (a is not constant).	1

12	Analyse the complexity of the following program
	
	main ( )
	{
	for (inti=1; i<=n;i=i*2)
	sum =sum+i+func(i)
	}
	void func(m)
	{
	for (int j=1; j<=m; j++)
	Statement with O(1) complexity
	
	Answer:
	O(n).
	For loop in ***main( )*** function executes ***$\log_2 n$*** times.
	Hence loop in function executes $2^0,2^1,2^2,\ldots,2^k$ times where ***$k= \log_2 n$***
	Total time = $\mathbf{2^0+2^1+2^2+\ldots+2^k = 2^{\log_2 n} * 2 = 2*n = O(n).}$
	(Correct order (either big O or big Theta) + any valid explanation
		3

13	a) Using iteration solve the following recurrence equation T(n)= 2 if n=1 else T(n)= 2T(n/2)+2n+3 ***Θ(n logn)*** Correct derivaton using iteration method (5 Marks) Answer: T(n)=2T(n/2)+n =2[2T(n/4)+(n/2)]+n $=2^2$ T(n/4)+n+n$=2^2$ T(n/4)+2n $=2^2$ [2T(n/8)+n/4]+2n $=2^3$ T(n/8)+n+2n$=2^3$ T(n/8)+3n ....After k times $=2^k T(n/2^k)+kn$---eq1 Sub problem size is 1 after $n/2^k =1$	

	ie 2^k=n, k=log n then eq 1becomes T(n)=nT(1)+n logn =n*1+nlogn =O(nlogn).	
	b)Using Recursion Tree method, solve. Assume constant time for small values of n. T(n)= 2T(n/10)+ T(9n/10)+n Answer ***Θ(n logn)*** Correct derivaton using recursion tree method (4 Marks)	
14	Using Recursion Tree method, solve T(n)= T(n/10)+ T(9n/10)+n  n cn $\frac{1}{10}n$ $\frac{9}{10}n$ cn $\frac{1}{100}n$ $\frac{9}{100}n$ $\frac{9}{100}n$ $\frac{81}{100}n$ cn $\frac{81}{1000}n$ $\frac{729}{1000}n$ cn $\le cn$ 1 $\le cn$ $\log_{10} n$ $\log_{10/9} n$ 1 $O(n \lg n)$ **Figure 7.4** A recursion tree for QUICKSORT in which PARTITION always produces a 9-to-1 split, yielding a running time of $O(n \lg n)$. Nodes show subproblem sizes, with per-level costs on the right. The per-level costs include the constant c implicit in the $\Theta(n)$ term. **$\log_{10/9}(n)$ = (1 / ln(10/9)) * ln(n) = (1 / 0.10536) * ln(n) = 9.49122 * ln(n)** The cost at the top level is to do **1** partition, at the next level down it is to do **2** partitions, then **4** partitions, then **8** and so forth. In spite of the increasing number of partitions as we go down the levels, the cost is about the same, since all the partitions at a given level work on the same number of array elements. At the level given by **$\log_{10}$(n)**, the leftmost downward line gets to a leaf node and terminates. Other lines give out as the depth increases, and this is shown in the diagram with "<= cn" as the cost, since parts of the array are sorted and no longer worked on. The deepest line downward is at the far right, with depth **$\log_{10/9}$(n)**. In any case, the algorithm is **Θ(n*log(n)).**	3

1	State Master's Theorem. Find the solution to the following recurrence equation using Master's theorem.  The Master Theorem applies to recurrences of the following form: $$T(n) = aT(n/b) + f(n)$$ where $a \geq 1$ and $b > 1$ are constants and $f(n)$ is an asymptotically positive function. There are 3 cases: 1. If $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$. 2. If $f(n) = \Theta(n^{\log_b a} \log^k n)$ with[1] $k \geq 0$, then $T(n) = \Theta(n^{\log_b a} \log^{k+1} n)$. 3. If $f(n) = \Omega(n^{\log_b a + \epsilon})$ with $\epsilon > 0$, and $f(n)$ satisfies the regularity condition, then $T(n) = \Theta(f(n))$. Regularity condition: $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n. (1 mark)	(3)
	b) T (n) = 2T (n/2)+ n log n **<u>Solution-</u>** We compare the given recurrence relation with T(n) = aT(n/b) + θ ($n^k \log^p n$). Then, we have-a = 2,b = 2,k = 1,p = 1 Now, a = 2 and $b^k = 2^1 = 2$. Clearly, $a = b^k$. So, we follow case-02. Since p = 1, so we have-T(n) = θ ($n^{\log_b a}.\log^{p+1} n$) T(n) = θ ($n^{\log_2 2}.\log^{1+1} n$) Thus, **T(n) = θ ($n\log^2 n$)**	1
	c) T (n) = 2^nT (n/2) + n^n Master method does not apply, (a is not constant).	1
9	Explain Asymptotic notations in algorithm analysis The main idea of asymptotic analysis is to have a measure of efficiency of algorithms that doesn't depend on machine specific constants, and doesn't require algorithms to be implemented and time taken by programs to be compared. Asymptotic notations are mathematical tools to represent time complexity of algorithms for asymptotic analysis. The following 3 asymptotic notations are mostly used to represent time complexity of algorithms. **1) Θ Notation:** The theta notation bounds functions from above and below, so it defines	

	exact asymptotic behaviour. A simple way to get Theta notation of an expression is to drop low order terms and ignore leading constants. For example, consider the following expression. $3n^3 + 6n^2 + 6000 = \Theta(n^3)$ Dropping lower order terms is always fine because there will always be a n_0 after which $\Theta(n^3)$ has higher values than $\Theta(n^2)$ irrespective of the constants involved. For a given function g(n), we denote Θ(g(n)) is following set of functions. $\Theta(g(n)) = \{f(n)$: there exist positive constants c1, c2 and n_0 such that $0 \leq c1*g(n) \leq f(n) \leq c2*g(n)$ for all $n \geq n_0\}$ The above definition means, if f(n) is theta of g(n), then the value f(n) is always between c1*g(n) and c2*g(n) for large values of n ($n \geq n_0$). The definition of theta also requires that f(n) must be non-negative for values of n greater than n_0. **2) Big O Notation:** The Big O notation defines an upper bound of an algorithm, it bounds a function only from above. For example, consider the case of Insertion Sort. It takes linear time in best case and quadratic time in worst case. We can safely say that the time complexity of Insertion sort is O(n^2). Note that O(n^2) also covers linear time. If we use Θ notation to represent time complexity of Insertion sort, we have to use two statements for best and worst cases. 1. The worst case time complexity of Insertion Sort is Θ(n^2). 2. The best case time complexity of Insertion Sort is Θ(n). The Big O notation is useful when we only have upper bound on time complexity of an algorithm. Many times we easily find an upper bound by simply looking at the algorithm. O(g(n)) = { f(n): there exist positive constants c and n0 such that $0 \leq f(n) \leq c*g(n)$ for all $n \geq n0\}$ **3) Ω Notation:** Just as Big O notation provides an asymptotic upper bound on a function, Ω notation provides an asymptotic lower bound. Ω Notation can be useful when we have lower bound on time complexity of an algorithm. As discussed in the previous post, the best case performance of an algorithm is generally not useful, the Omega notation is the least used notation among all three. For a given function g(n), we denote by Ω(g(n)) the set of functions. Ω (g(n)) = {f(n): there exist positive constants c and n0 such that 0 <= c*g(n) <= f(n) for all n >= n0}.	
11	Solve using Masters theorem i) $T(n)=2T(n/4)+\sqrt{n}$ ii) $T(n)=7T(n/2)+ n^2$ Solve using Masters theorem i) $O(\sqrt{n}\lg n)$ **Compare with master theorem ,a=2,b=4 and $f(n)=n^{112}$** $T(n)=n^{\log_4 2} =n^{1/2}$ i.e f(n) is polynomially same as t(n), therefore it is second case	

therefore $T(n) = \Theta(n^{1/2} \lg n)$ (case 2).

ii) O(n^lg7)

Compare with master theorem ,a=7,b=2 and $f(n)=n^2$

$T(n)=n^{\log_2 7} = n^{2.81}$ i.e f(n) is polynomially smaller than t(n), therefore it is first case

therefore, ans is $\ominus(n^{\log 7})$ or $\ominus(n^{2.81})$