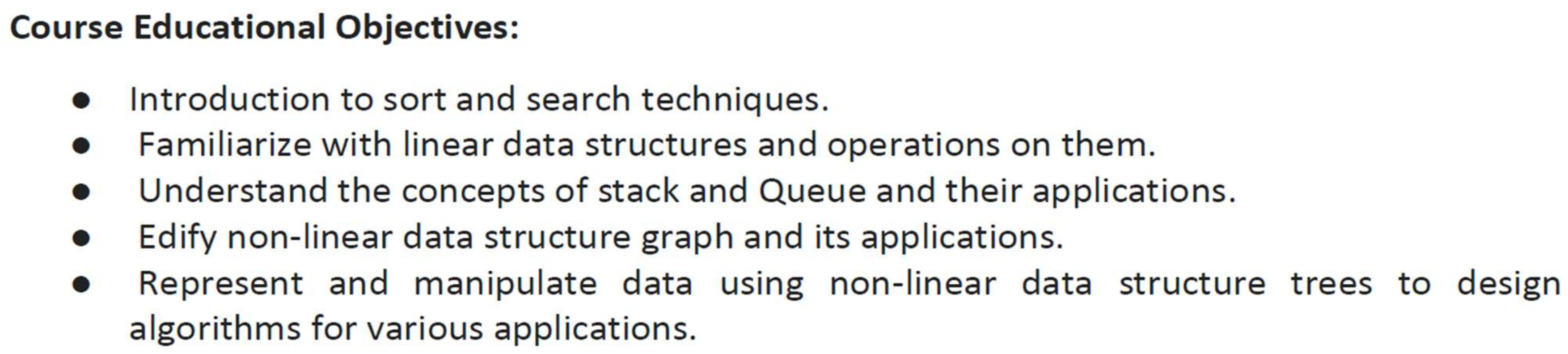
Data Structures

Dr. Bhargavi B.

Assistant Professor,

CSE Department, GSoT, GITAM-HYD.

Why this Course?



Course Educational Objectives:

- Introduction to sort and search techniques.
- Familiarize with linear data structures and operations on them.
- Understand the concepts of stack and Queue and their applications.
- Edify non-linear data structure graph and its applications.
- Represent and manipulate data using non-linear data structure trees to design algorithms for various applications.

Syllabus-Theory and Lab

Course Objectives

- To evaluate the quality of a program (Analysis of Algorithms: Running Time and memory space)
- To write fast programs
- To solve new problems
- To give non-trivial methods to solve problems.

[Your algorithm/program will be faster]

Unit-1

- Introduction to data structures
- Array/List based representations and operations
- Searching techniques
    - Linear Search
    - Binary Search

- Sorting Techniques
    - Insertion Sort, Selection Sort, Bubble Sort, Merge sort and Quick sort

Introduction to Data Structures

- Data Structure: A **data structure** is a way of arranging data on a computer so that it can be accessed and updated efficiently.

  - It must mainly include:
    1. Collection of data,
    2. Relationships among them and
    3. functions/operations that can be applied to the data like creation, insertion, deletion, search and traversal.

Examples of Data Structures

- Arrays
- Linked Lists
- Stacks
- Queues
- Trees
- Graphs

Application Areas of Data Structures

- Compiler Design (Syntax trees, symbol table)
- Operating System (Task scheduling Queues)
- Artificial Intelligence(Graphs for path finding)
- DBMS (Trees for indexing)
- Simulation (Queues for managing events)
- Graphics (Quad trees for storing polygons)
- Blockchain Technology (Linked lists for forming blockchain)

Algorithm Specification

-Pseudocode Conventions

C like statements

1. Comments begin with //
2. Blocks indicated with matching braces { and }.
3. Statements are delimited by ;
4. Simple Data types are not explicitly declared
5. Assignment <var>:=<expression>;
6. **true** and **false** Boolean values can be used
7. 2D array access as A[i,j]

Algorithm Specification

-Pseudocode Conventions

8. Conditional statement form:

```
if <condition> then <statement>
if <condition> then <statement 1> else        <statement 2>
```

9. Looping statements for, while, repeat-until can be used

10. Algorithm is the only procedure with heading and a body. Heading takes form:

Algorithm *Name (<parameter list>)*

Example 1: To find sum of two numbers

```
Algorithm Sum(a, b)
{
return a+b;
}
```

Example 2a: To find factorial of a given number

```
Algorithm Factorial(n)
{
  fact=1;
  for(i=1;i<=n;i++)
  fact=fact*i;
  return fact;
}
```

Example 2b: Recursive algorithm to find factorial of a given number

```
Algorithm RFactorial(n)
{
  if(n<=1) return 1;
  return n*RFactorial(n-1);
}
```

Example 3: Sum of n numbers

```
Algorithm Sumoflist( A[], n)
{
        sum=0;
        for(i=0;i<n;i++)
                sum=sum+A[i];
        return sum;
}
```

Week1 Tasks of DS Lab

1. Devise an algorithm and write program that inputs three integers and prints them in ascending order. Start the algorithm with **Algorithm Sort3Numbers(a,b,c)**
2. Devise an algorithm and write program to find the maximum value from an array of n numbers. Start the algorithm with **Algorithm MaxArray(A[], n)**
3. Devise an algorithm and write program to find the minimum value from an array of n numbers . Start the algorithm with **Algorithm MinArray(A[], n)**
4. Devise an algorithm and write program to find GCD of two numbers. Start the algorithm with **Algorithm GCD(a, b)**
5. Devise a recursive algorithm and write recursive program to find GCD of two numbers. Start the algorithm with **Algorithm RecursiveGCD(a, b)**
6. Devise an algorithm and write program to find the numbers of words in a string of length n. Start the algorithm with **Algorithm CountofWords(s[],n)**
7. Devise an algorithm and write program to find factorial of a number. Start the algorithm with **Algorithm Factorial(n)**
8. Devise a recursive algorithm and write recursive program to find factorial of a number. Start the algorithm with **Algorithm RecursiveFactorial(n)**

When do u say an algorithm is good?

- Efficient
  - Running Time
  - Space Used
- Running time depends on
  - Single vs Multi processor ✗
  - Read or Write speed to Memory ✗
  - 32 bit vs 64 bit ✗
  - Input -> rate of growth of time, Efficiency as a function of input (number of bits in an input number, number of data elements…) ✓

- **Time Complexity:** Amount of computation time (CPU time) where program needs to run

- **Space Complexity:** Amount of memory the program needs to run for completion

- It is necessary to **implement and test the algorithm** in order to determine its running time.
- In order to **compare** two or more algorithms, the same **hardware and software environments** should be used.
- To compare two algorithms, the ideal solution is to express the running time as function of input size *n*
  - Example: Searching

    -A1: Linear Search vs

    -A2: Binary Search

How to analyze time complexity?

<u>Step Count Method</u>

- 1 unit time for basic Arithmetic and Logical Operations
- 1 unit time for basic statements like assignment and return
- There is no count for { and }.
- If a basic statement is iterated, then multiply by the number of times the loop is run.
- Loop statement that is iterated *n* times, has a count of (n+1).
    - Loop runs *n* times for the true case and a check is performed for the loop exit, hence, n+1.

Example

```
Algorithm Sum(a, b)
{
 c=a+b;
return c;
}
```

- T_{sum} =1+1=2 units of time

  =c`=constant time

Example

```
Algorithm Sumoflist( A[], n)                 Step count
{                                            ----------------
        sum=0;                               1
        for(i=0;i<n;i++)                     n+1
                sum=sum+A[i];                n
        return sum;                          1
}
```

- $T_{sumoflist}$ =1+(n+1)+n+1=2*n+3 time units
  =cn+c’

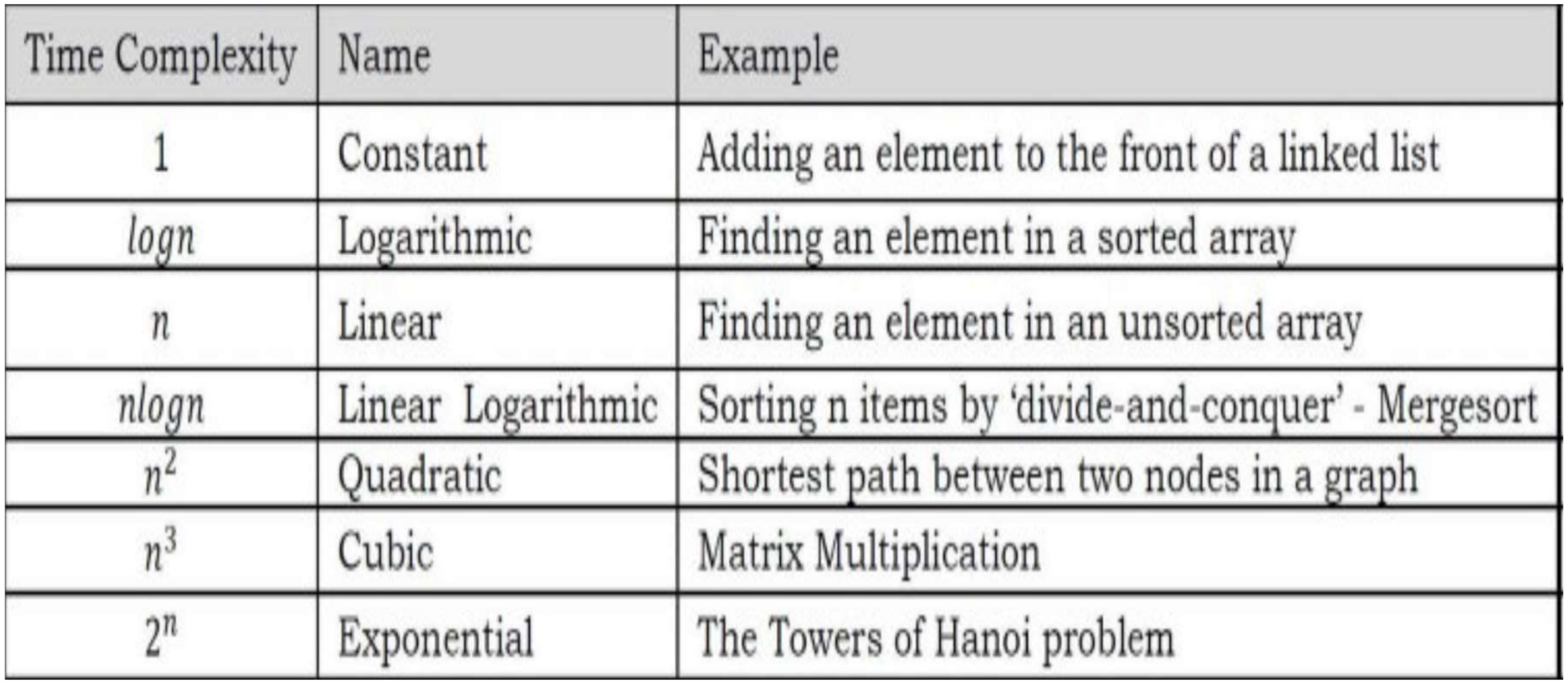
Using Step Count Method, find the time taken by below algorithms

1. **Algorithm Sort3Numbers(a, b, c)** that inputs three integers and prints them in ascending order.
2. **Algorithm MaxArray(A[], n)** to find the maximum value from an array of n numbers
3. **Algorithm MinArray(A[], n)** to find the minimum value from an array of n numbers .
4. **Algorithm Factorial(n)** to find factorial of a number.

What is rate of growth?

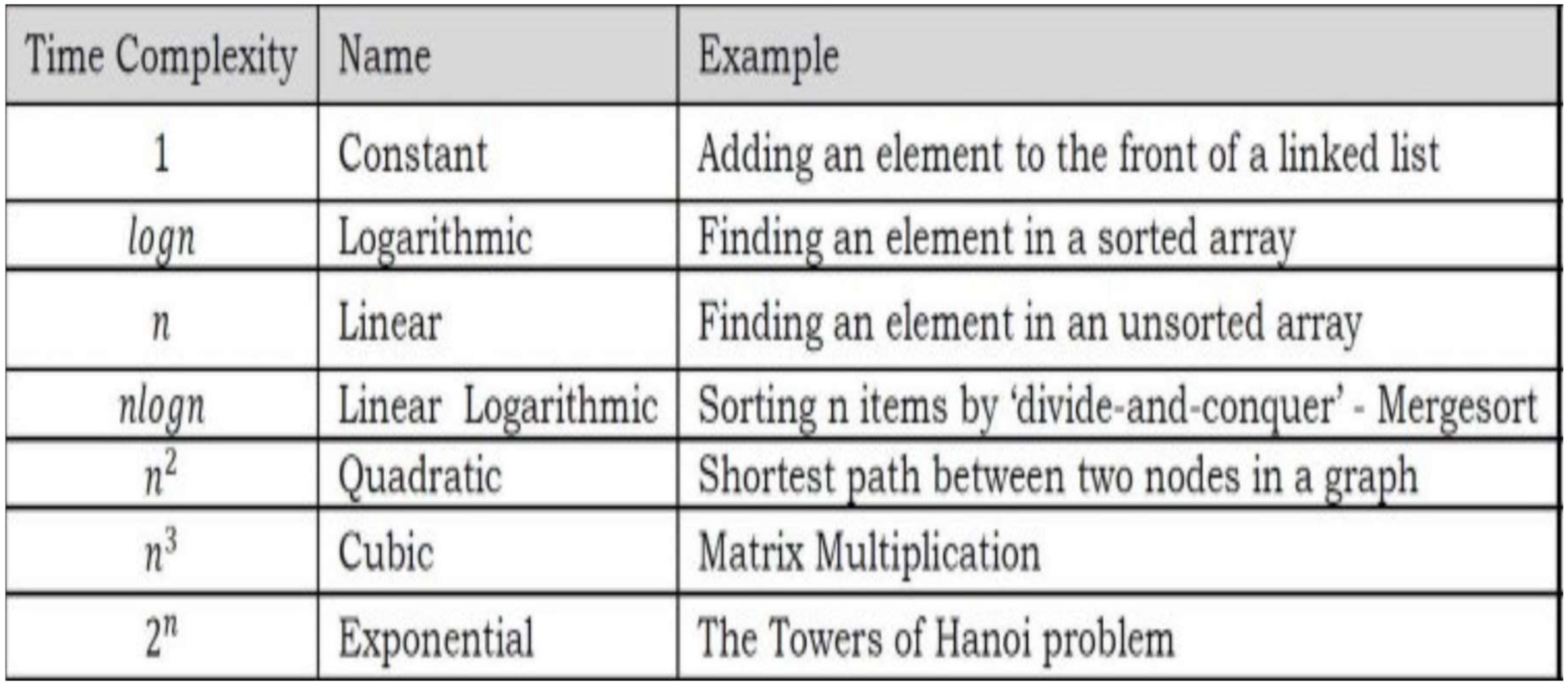
- The rate at which the running time increases as a function of input size.
- Example: 1, logn, n, n^2, 2^n.

Commonly used rates of growth



Time Complexity	Name	Example
1	Constant	Adding an element to the front of a linked list
$logn$	Logarithmic	Finding an element in a sorted array
n	Linear	Finding an element in an unsorted array
$nlogn$	Linear Logarithmic	Sorting n items by 'divide-and-conquer' - Mergesort
n^2	Quadratic	Shortest path between two nodes in a graph
n^3	Cubic	Matrix Multiplication
2^n	Exponential	The Towers of Hanoi problem

Plot graphs for n={1, 2, 4, 8, 16, 32} for following time complexities



Time Complexity	Name	Example
1	Constant	Adding an element to the front of a linked list
$logn$	Logarithmic	Finding an element in a sorted array
n	Linear	Finding an element in an unsorted array
$nlogn$	Linear Logarithmic	Sorting n items by 'divide-and-conquer' - Mergesort
n^2	Quadratic	Shortest path between two nodes in a graph
n^3	Cubic	Matrix Multiplication
2^n	Exponential	The Towers of Hanoi problem

Types of Time Complexity Analysis

- Worst Case
- Average Case
- Best Case

Types of Analysis

- **Worst case**
  - Defines the input for which the algorithm takes a long time (slowest time to complete).
  - Input is the one for which the algorithm runs the slowest.
- **Best case**
  - Defines the input for which the algorithm takes the least time (fastest time to complete).
  - Input is the one for which the algorithm runs the fastest.
- **Average case**
  - Provides a prediction about the running time of the algorithm.
  - Run the algorithm many times, using many different inputs that come from some distribution that generates these inputs, compute the total running time (by adding the individual times), and divide by the number of trials.
  - Assumes that the input is random.

Lower Bound <= Average Time <= Upper Bound

*Asymptotic Notations- O, Ω, θ

- They are the notations that describe the different rate-of-growth relations between the defining function and the defined set of functions.
- They are defined for functions over natural numbers
  - Example :f(n)=O(n)
- They define a *set* of functions; in practice used to compare two function sizes.
- Notations:
  - Big Oh Notation(O)—Upper bound on time complexity
  - Omega Notation(Ω)—Lower bound on time complexity
  - Theta Notation(θ)—Tight bound on time complexity

Big-Oh Notation

- **Big Oh notation** (with a capital letter O, not a zero), also called **Landau's symbol**, is a symbolism used in Computational Complexity Theory, Computer Science, and Mathematics to describe the asymptotic behavior of functions.
- Basically, it tells you how fast a function grows or declines.

- Landau's symbol comes from the name of the German number theoretician **Edmund Landau** who invented the notation.
- The letter O is used because the rate of growth of a function is also called its order.

*Big-Oh Notation

- Given functions $f(n)$ and $g(n)$, we say that $f(n)$ is $O(g(n))$ if there are positive constants c and n_0 such that

  $f(n) \le cg(n)$ for $n \ge n_0$

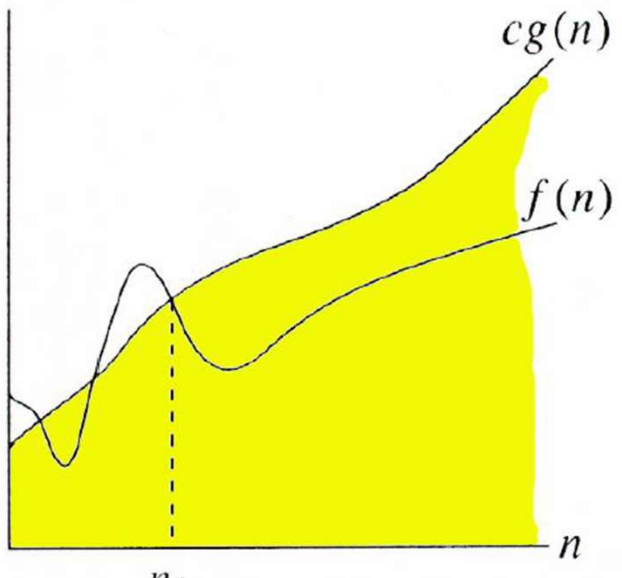
- Example: $2n + 10$ is $O(n)$

  $2n + 10 \le cn$

  $(c - 2)\, n \ge 10$

  $n \ge 10/(c - 2)$

  Pick $c = 3$ and $n_0 = 10$



$f(n) = O(g(n))$

Big-Oh Notation: Example

Given f(n)=2n+10
Show that f(n)=O(n) =>
Find c and n_0 such that f(n)<=cg(n) for some c, n>=n_0

$$2n+10<=c*n$$

Big-Oh Notation: Example

Given $f(n)=2n+10$

Show that $f(n)=O(n)$ =>

Find c and n_0 such that $f(n)<=cg(n)$ for some c, $n>=n_0$

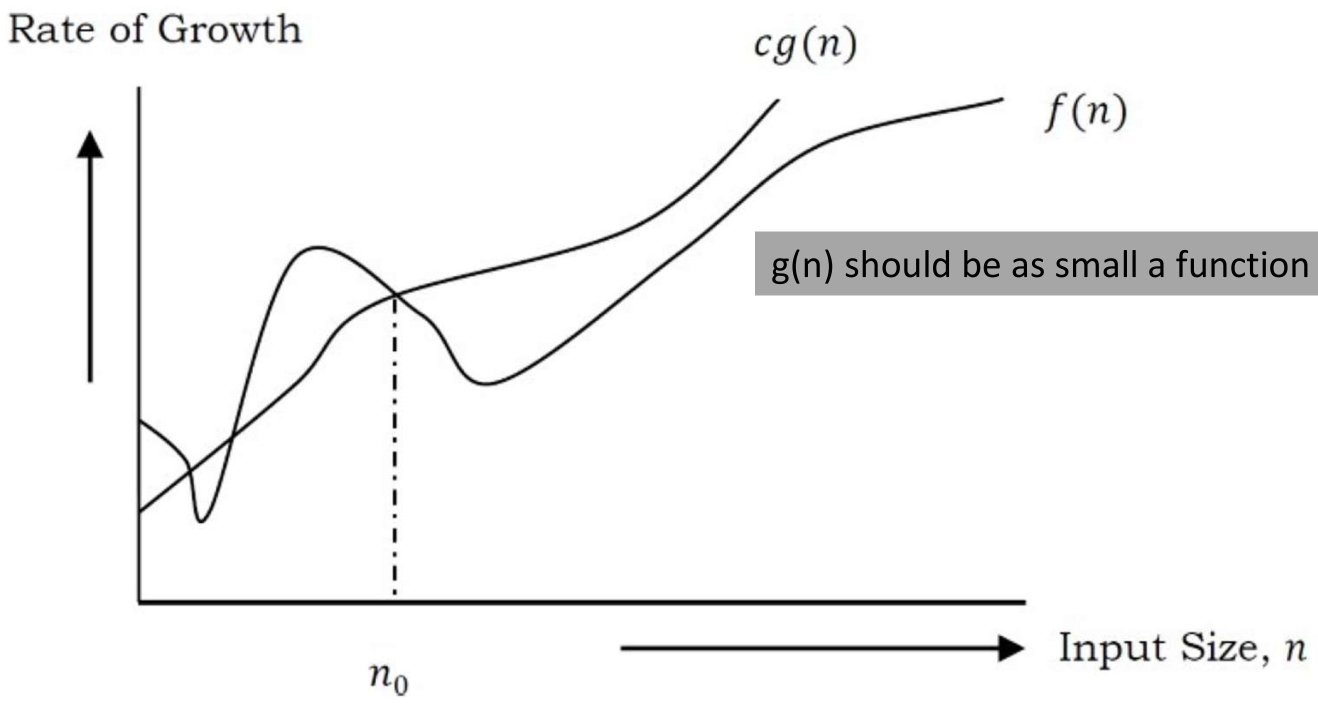
$2n+10<=c*n$

Let $2n+10<=12n$

- ✓ for n=1, 12<=12
- ✓ n=2, 14<=28
- ✓ n=3, 18<=36

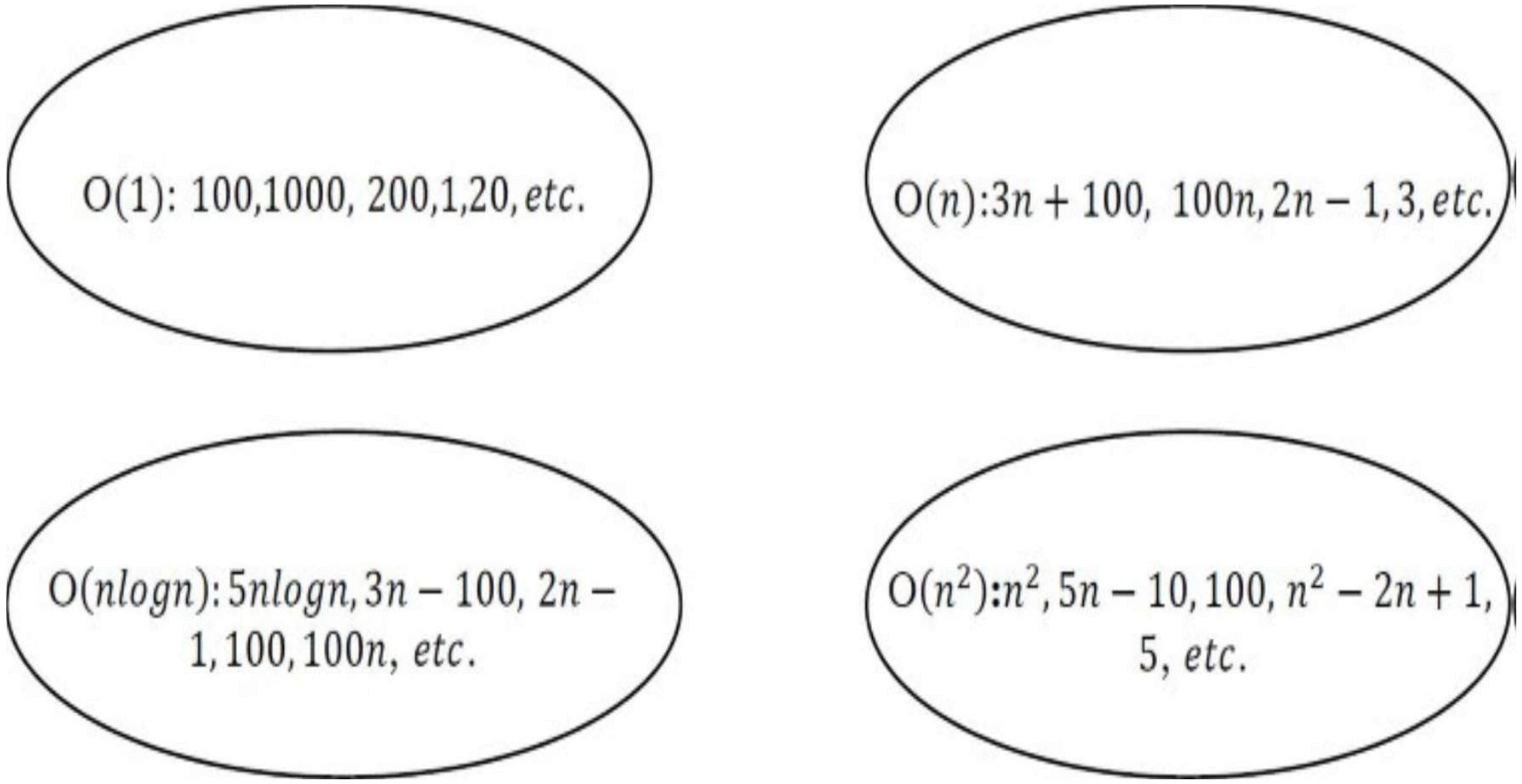
….=> c=12 and **n_0=1**

Big-Oh Notation



g(n) should be as small a function of n for which f(n)=O(g(n))

Big-Oh Visualization


O(1): 100,1000, 200,1,20, $etc.$
O(n):$3n + 100$, $100n$, $2n - 1$, 3, $etc.$
O($nlogn$): $5nlogn$, $3n - 100$, $2n - 1$, 100, $100n$, $etc.$
O(n^2):n^2, $5n - 10$, 100, $n^2 - 2n + 1$, 5, $etc.$

Find upper bound (in Big O) for the following functions with appropriate c and n_0

1. $f(n)=3n+8$
2. $f(n)=410$
3. $f(n)=n^2 + 1$

Big Oh notation

- Example: the function n^2 is not $O(n)$

  $n^2 \leq cn$

  $n \leq c$

  The above inequality cannot be satisfied since c must be a constant

  n^2 is $O(n^2)$.

Big Oh notation

- 7n-2

  7n-2 is O(n)

  need $c > 0$ and $n_0 \geq 1$ such that $7n-2 \leq c \bullet n$ for $n \geq n_0$

  this is true for $c = 7$ and $n_0 = 1$

- $3n^3 + 20n^2 + 5$

  $3n^3 + 20n^2 + 5$ is $O(n^3)$

  need $c > 0$ and $n_0 \geq 1$ such that $3n^3 + 20n^2 + 5 \leq c \bullet n^3$ for $n \geq n_0$

  this is true for $c = 4$ and $n_0 = 21$

- 3 log n + 5

  3 log n + 5 is O(log n)

  need $c > 0$ and $n_0 \geq 1$ such that $3 \log n + 5 \leq c \bullet \log n$ for $n \geq n_0$

  this is true for $c = 8$ and $n_0 = 2$

Big Oh notation

Theorem: If $f(n)=a1n^{m} + a2n^{m-1} + ...a'n+a0$, then $f(n)=O(n^{m})$

Big Oh notation: Quiz

1. Can 7n-2 be O(1)?

Big Oh notation: Quiz

1. Can 7n-2 be O(1)?

Answer: No

Big Oh notation: Quiz

2. Can 7n-2 be O(n^2)?

Big Oh notation: Quiz

2. Can 7n-2 be O(n^2)?

Answer: Yes

Time Complexity

- We analyze time complexity for
  a) A very large input size
  b) Worst case scenario
- Rules
  a) Drop lower order terms
  b) Drop Constant multipliers

Big Oh notation: Quiz

- Find the time complexity of the following code snippet:

```
int n=32;
int steps=0;
for(int i=1; i<=n; i=i*2)
            steps=steps+1;
printf(“%d”, steps);
```

Big Oh notation: Quiz

- Find the time complexity of the following code snippet:

```
int n=32;
int steps=0;
for(int i=1;i<=n;i=i*2)
            steps=steps+1;
printf("%d", steps);
```

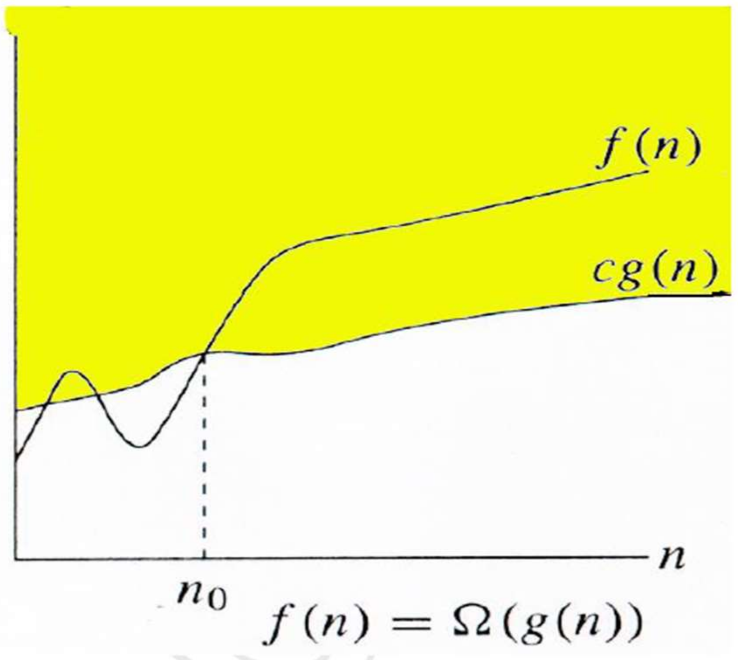
Answer:O(logn)

*Ω- notation

For function $g(n)$, we define $\Omega(g(n))$, big-Omega of n, as the set:

$\Omega(g(n)) = \{f(n)$:
∃ positive constants c and n_0, such that $\forall n \geq n_0$,
we have $0 \leq cg(n) \leq f(n)\}$

Intuitively: Set of all functions whose *rate of growth* is the same as or higher than that of $g(n)$.



$g(n)$ is an *asymptotic lower bound* for $f(n)$.

Ω- notation

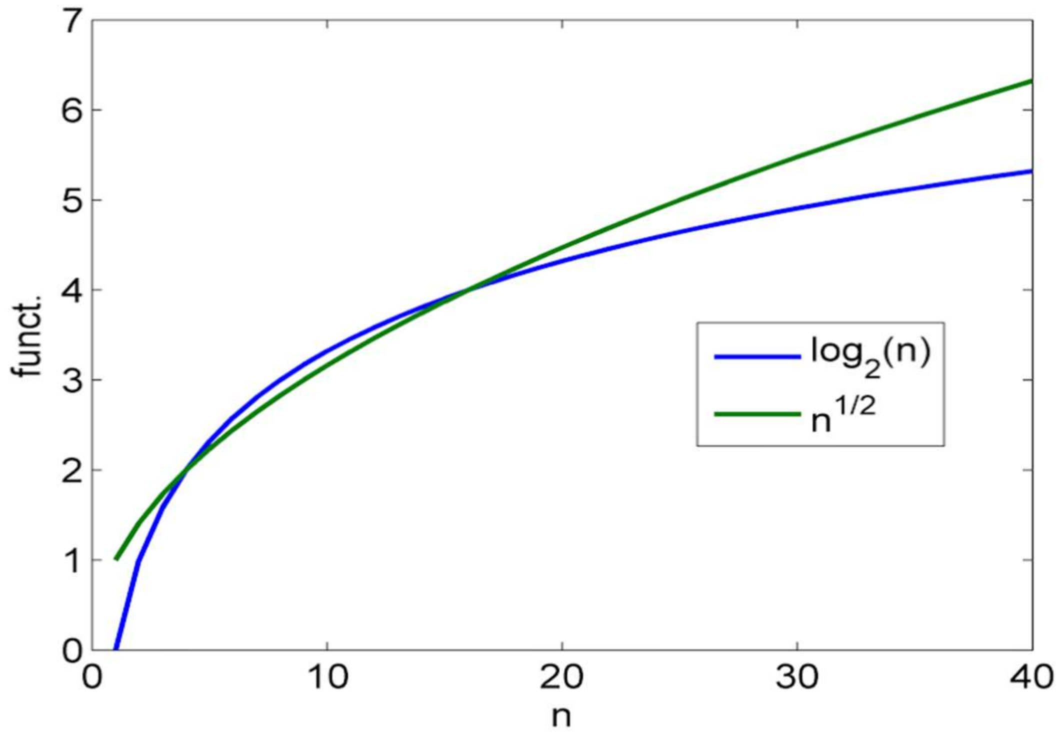
$\Omega(g(n)) = \{f(n) :\exists$ positive constants c and n_0, such that $\forall n \geq n_0$, we have $0 \leq cg(n) \leq f(n)\}$

- $\sqrt{n} = \Omega(\log n)$. Choose c and n_0.

  for c=1 and n_0 =16,

$$c * \log n \leq \sqrt{n} \quad , \forall n \geq 16$$

Ω- notation



Ω- notation

- Omega gives us a **LOWER BOUND** on a function.

❖ Big-Oh says, "Your algorithm is at least this good."
❖ Omega says, "Your algorithm is at least this bad."

*ϴ- notation

For function $g(n)$, we define $\Theta(g(n))$, big-Theta of n, as the set:

$\Theta(g(n)) = \{f(n)$:

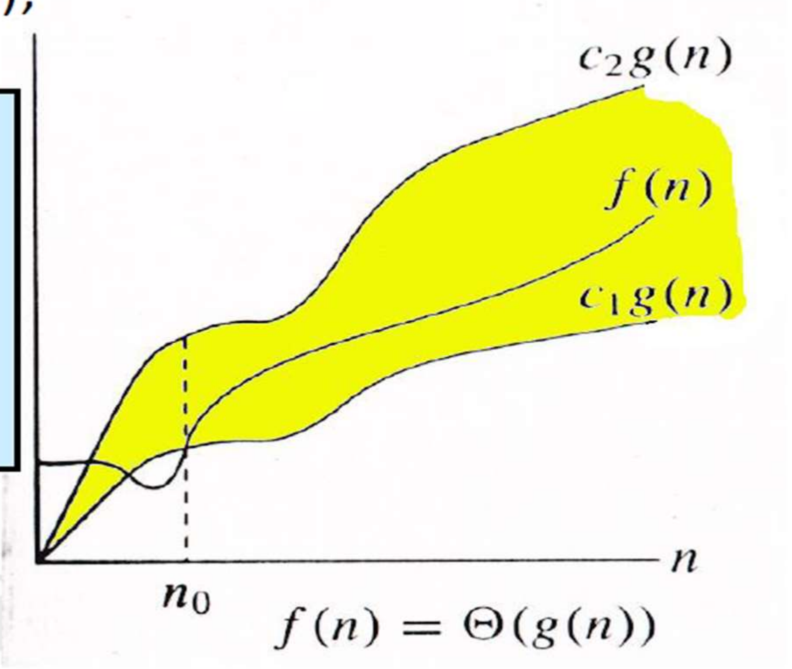
$\exists$ **positive constants c_1, c_2, and n_0,**

such that $\forall n \geq n_0$,

we have $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$

$\}$

Intuitively: Set of all functions that have the same *rate of growth* as $g(n)$.



***g*(*n*) is an *asymptotically tight bound* for *f*(*n*).**

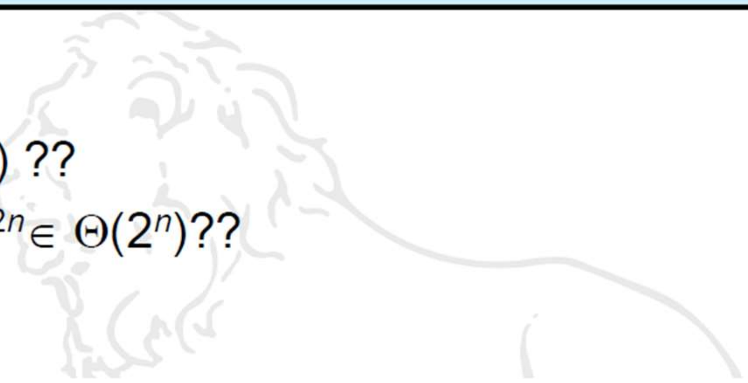
ϴ- notation

$\Theta(g(n)) = \{f(n) :$ ∃ positive constants c_1, c_2, and n_0, such that $\forall n \geq n_0$, $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)\}$

- $10n^2 - 3n = \Theta(n^2)$
- What constants for n_0, c_1, and c_2 will work?
- Make c_1 a little smaller than the leading coefficient, and c_2 a little bigger.
- ***To compare orders of growth, look at the leading term.***

Θ- notation

$\Theta(g(n)) = \{f(n) : \exists$ positive constants c_1, c_2, and n_0, such that $\forall n \geq n_0$, $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)\}$



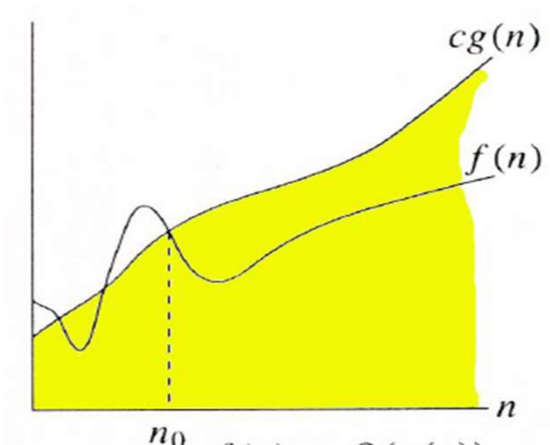
- Is $3n^3 \in \Theta(n^4)$??
- How about $2^{2n} \in \Theta(2^n)$??

Example

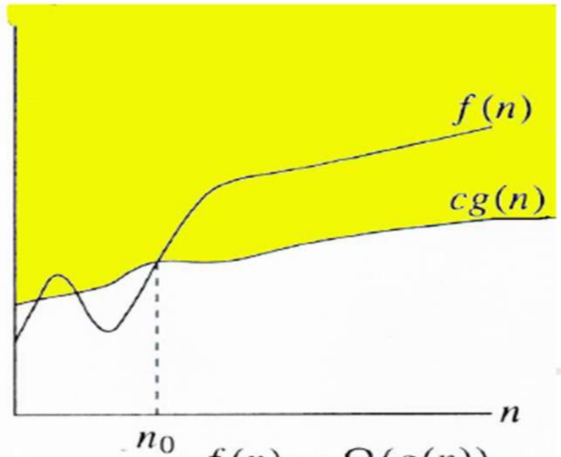
$3n^2 + 17$

- ***$\Omega(1)$, $\Omega(n)$, $\Omega(n^2)$*** ➔ lower bounds
- *$O(n^2)$, $O(n^3)$, ...* ➔ upper bounds
- *$\Theta(n^2)$* ➔ exact bound

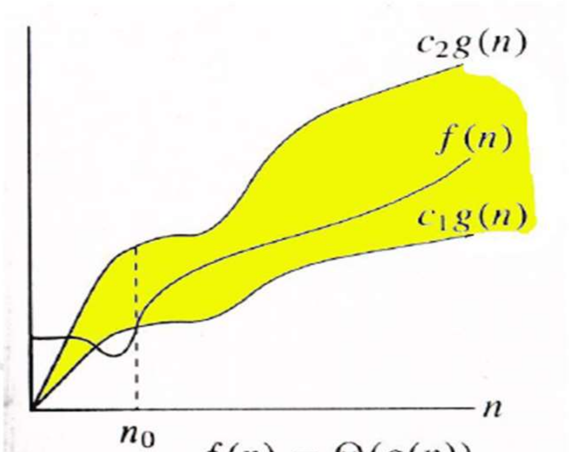
Relations between O, Ω and Ɵ



$f(n) = O(g(n))$



$f(n) = \Omega(g(n))$



$f(n) = \Theta(g(n))$

*Comparison of O, Ω and Θ

$$f \leftrightarrow g \;\approx\; a \leftrightarrow b$$

$$f(n) = O(g(n)) \;\approx\; a \le b$$

$$f(n) = \Omega(g(n)) \;\approx\; a \ge b$$

$$f(n) = \Theta(g(n)) \;\approx\; a = b$$

Limits of O, Ω and Θ

- $\lim_{n\to\infty} [f(n) / g(n)] < \infty \Rightarrow f(n) \in O(g(n))$
- $0 < \lim_{n\to\infty} [f(n) / g(n)] < \infty \Rightarrow f(n) \in \Theta(g(n))$
- $0 < \lim_{n\to\infty} [f(n) / g(n)] \Rightarrow f(n) \in \Omega(g(n))$
- $\lim_{n\to\infty} [f(n) / g(n)]$ undefined $\Rightarrow$ can't say

Represent following functions using O, Ω and Θ

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

$$T(n) = 16n + \log n$$

Guidelines for Asymptotic Analysis

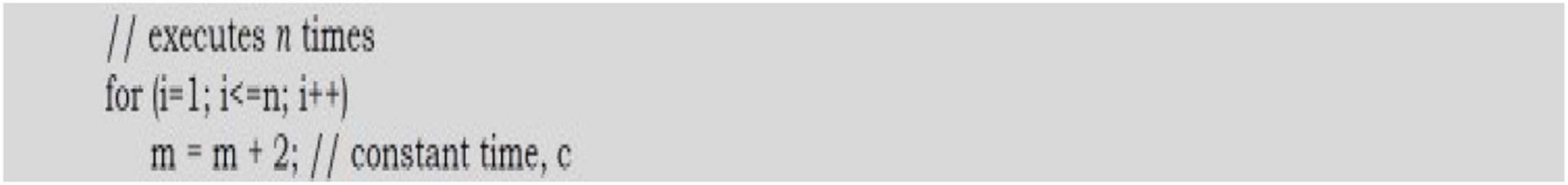
- General rules to determine running time of an algorithm:

    1. Loops
    2. Nested Loops
    3. Consecutive statements
    4. If-then-else statements
    5. Logarithmic statements

Guidelines for Asymptotic Analysis

- General rules to determine running time of an algorithm:

  1. Loops: Running time depends on no. of statements inside the loop multiplied by the number of iterations

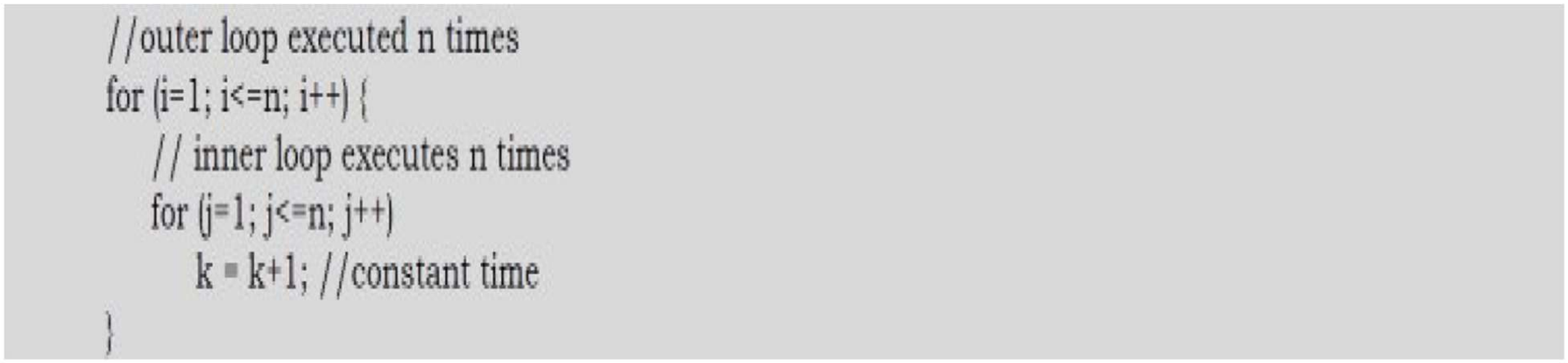


```
// executes n times
for (i=1; i<=n; i++)
    m = m + 2; // constant time, c
```

Total time = a constant $c \times n = c\,n = O(n)$.

Guidelines for Asymptotic Analysis

- General rules to determine running time of an algorithm:
  - 2. Nested Loops: Analyze from the inside out. Total running time is the product of the sizes of all the loops.

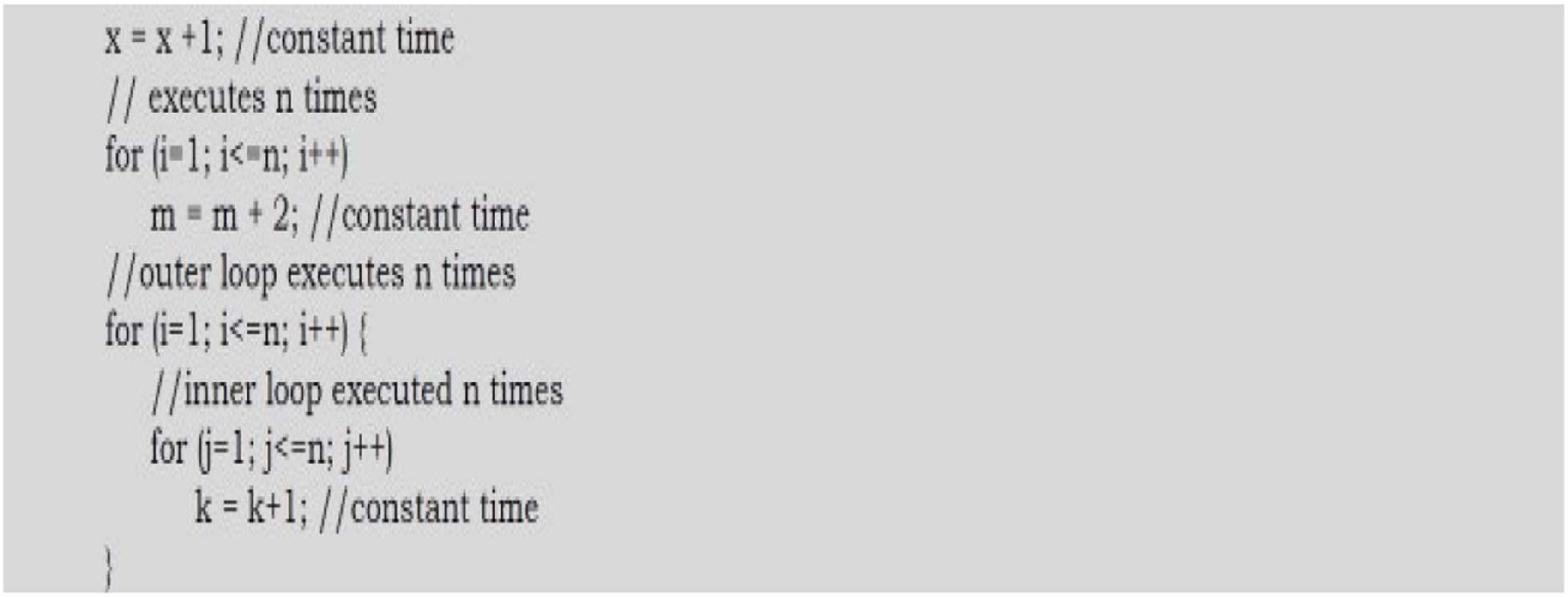


```
//outer loop executed n times
for (i=1; i<=n; i++) {
    // inner loop executes n times
    for (j=1; j<=n; j++)
        k = k+1; //constant time
}
```

Total time = $c \times n \times n = cn^2 = O(n^2)$.

Guidelines for Asymptotic Analysis

3. Consecutive statements: Add the time complexities of each statement.



```
x = x +1; //constant time
// executes n times
for (i=1; i<=n; i++)
    m = m + 2; //constant time
//outer loop executes n times
for (i=1; i<=n; i++) {
    //inner loop executed n times
    for (j=1; j<=n; j++)
        k = k+1; //constant time
}
```

Total time = $c_0 + c_1 n + c_2 n^2 = O(n^2)$.

Guidelines for Asymptotic Analysis

4. If-then-else statements: Consider the test condition and either then part or else part(whichever is larger)

```
if(length==0)
            return 0;
else
            {
                        for(int i=0;i<n;i++)
                                    //simple statements
            }
```

Total time=c1+c2*n=O(n).

Guidelines for Asymptotic Analysis

5. Logarithmic statements: An algorithm is O(logn) if it takes a constant time to cut the problem size by a fraction (usually by ½).

```
for (i=1; i<=n;)
    i = i*2;
```

If we observe carefully, the value of i is doubling every time. Initially $i = 1$, in next step $i = 2$, and in subsequent steps $i = 4,8$ and so on. Let us assume that the loop is executing some k times. At k^{th} step $2^k = n$, and at $(k + 1)^{th}$ step we come out of the *loop*. Taking logarithm on both sides, gives

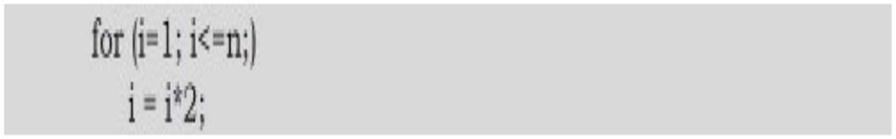
$$log(2^k) = logn$$
$$klog2 = logn$$
$$k = logn \qquad //\text{if we assume base-2}$$

Total time = O(*logn*).

Guidelines for Asymptotic Analysis

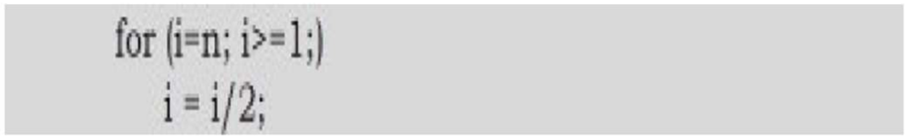
5. Logarithmic statements: An algorithm is O(logn) if it takes a constant time to cut the problem size by a fraction (usually by ½).

Example 1



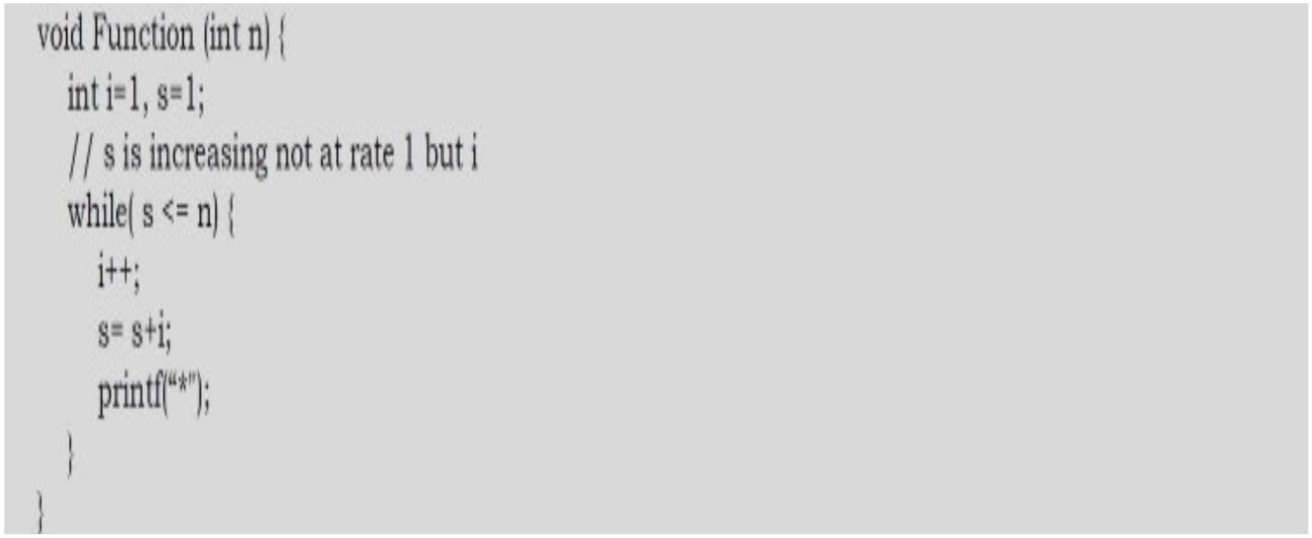
```
for (i=1; i<=n;)
   i = i*2;
```

Example 2



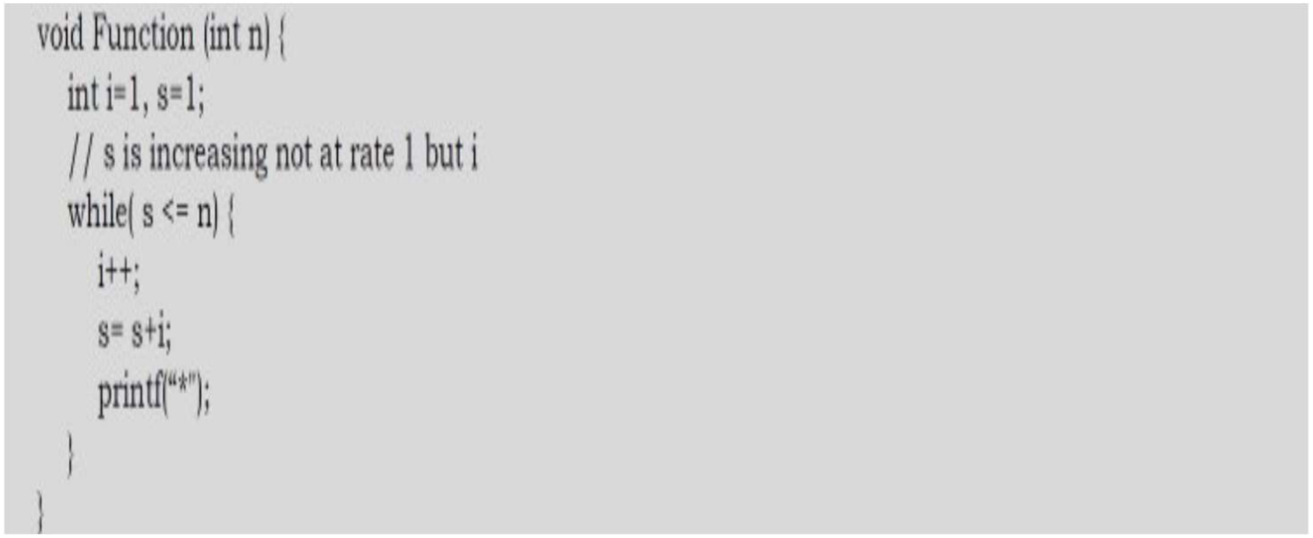
```
for (i=n; i>=1;)
   i = i/2;
```

What is time complexity of following 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("*");
   }
}
```

What is time complexity of following function?

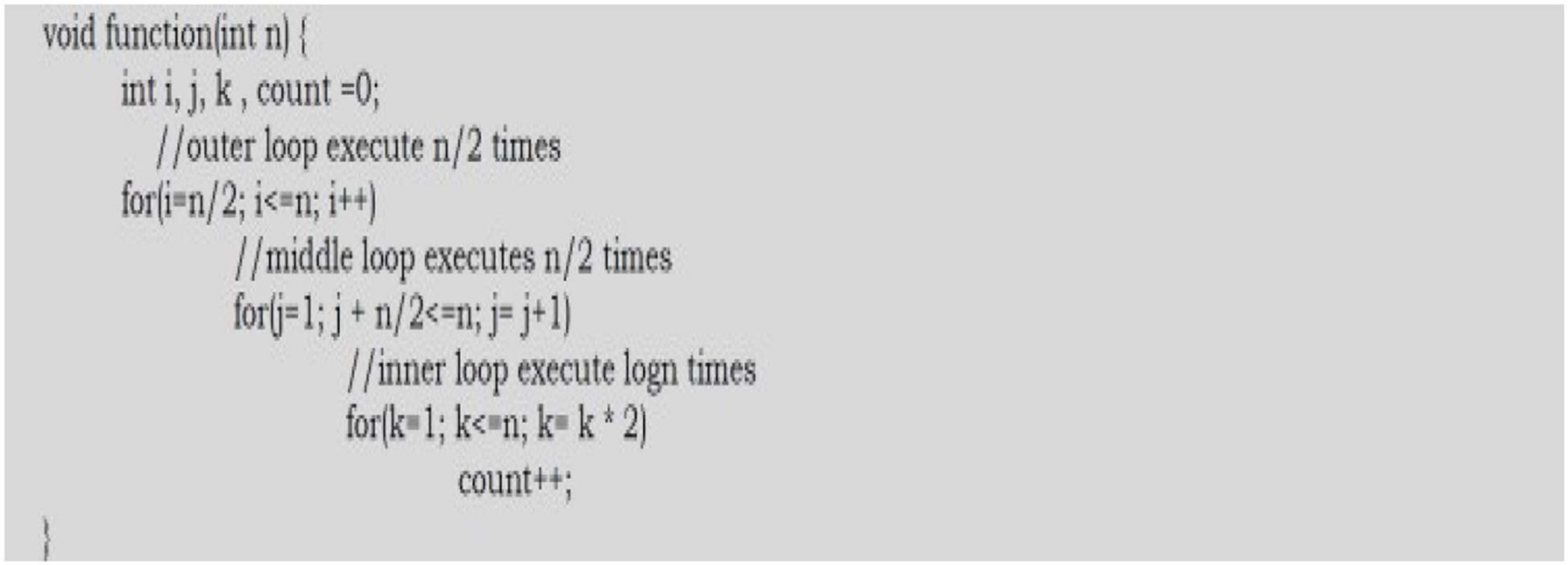


```
void Function (int n) {
  int i=1, s=1;
  // s is increasing not at rate 1 but i
  while( s <= n) {
    i++;
    s= s+i;
    printf("*");
  }
}
```

- We can define the *'s'* terms according to the relation $s_i = s_{i-1} + i$. The value of t' increases by 1 for each iteration.
- The value contained in *'s'* at the *ith* iteration is the sum of the first k positive integers.
- If *k* is the total number of iterations taken by the program, then the *while* loop terminates if:

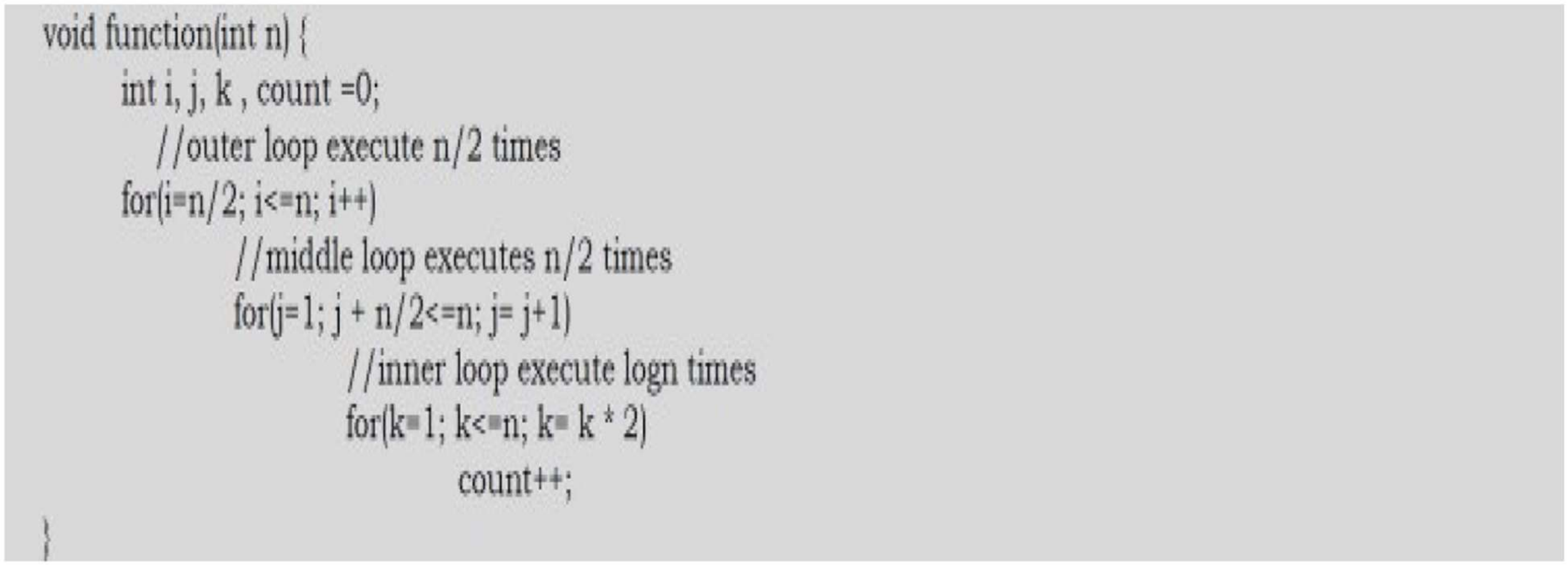
$$1 + 2 + \ldots + k = \frac{k(k+1)}{2} > n \Rightarrow k = O(\sqrt{n}).$$

What is time complexity of 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++;
}
```

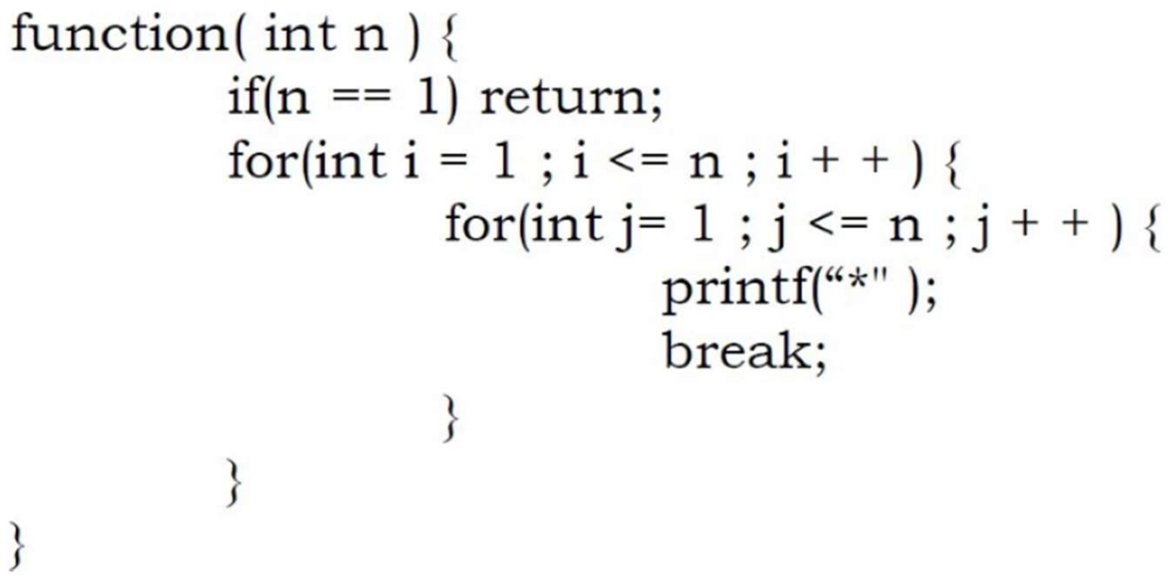
What is time complexity of following function?



```
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++;
}
```

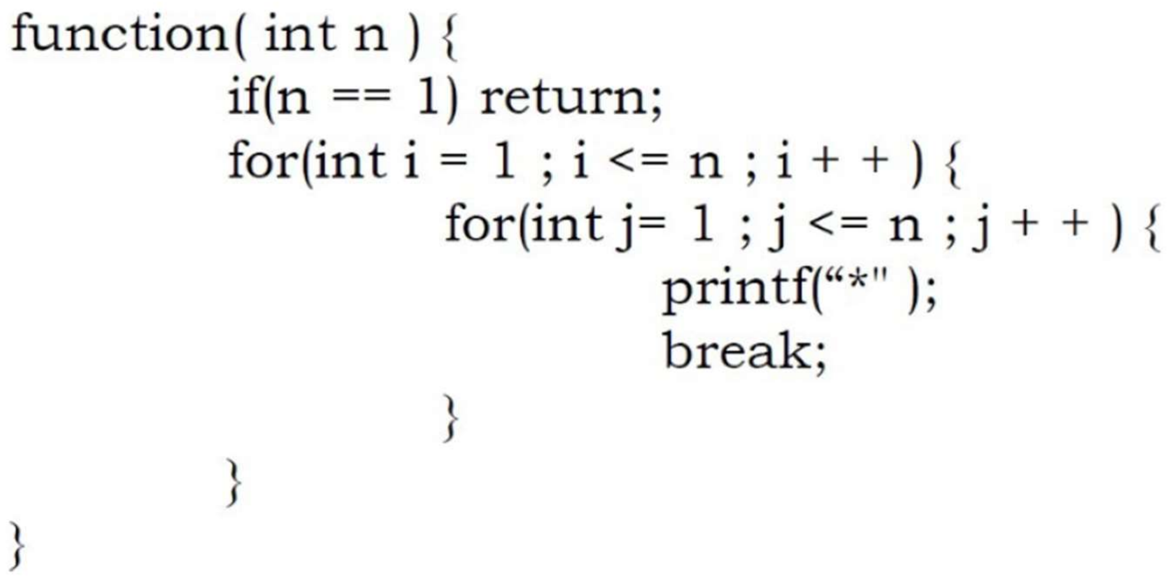
Answer: $O(n^2 \log n)$

What is time complexity of following function?



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

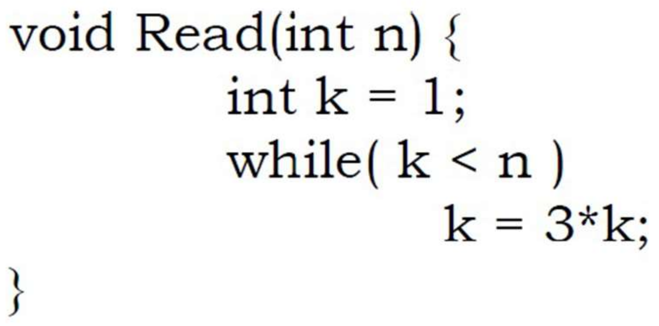
What is time complexity of following function?



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

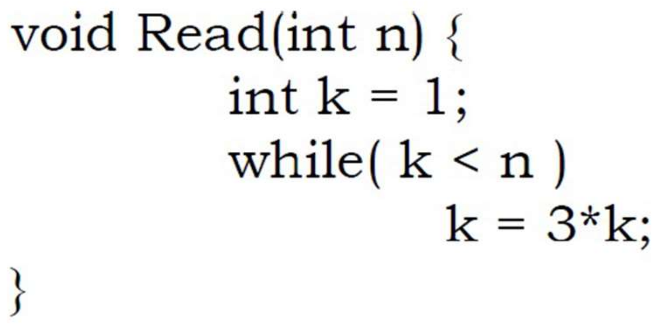
Answer: O(n)

What is time complexity of following function?



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

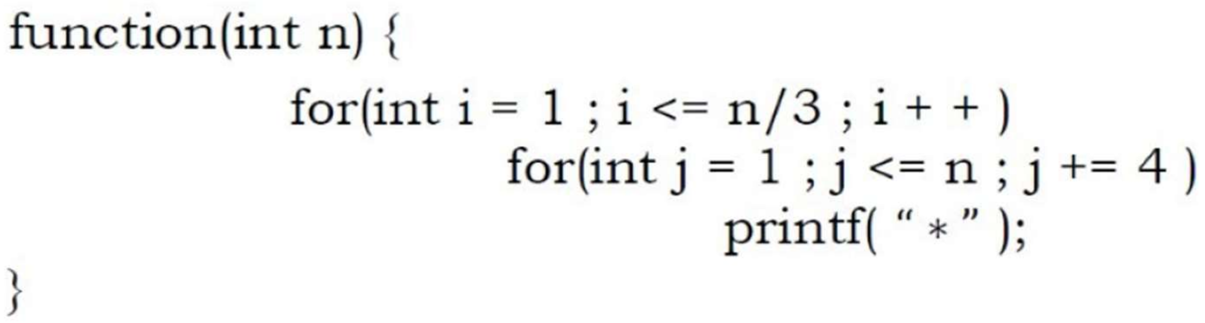
What is time complexity of following function?



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

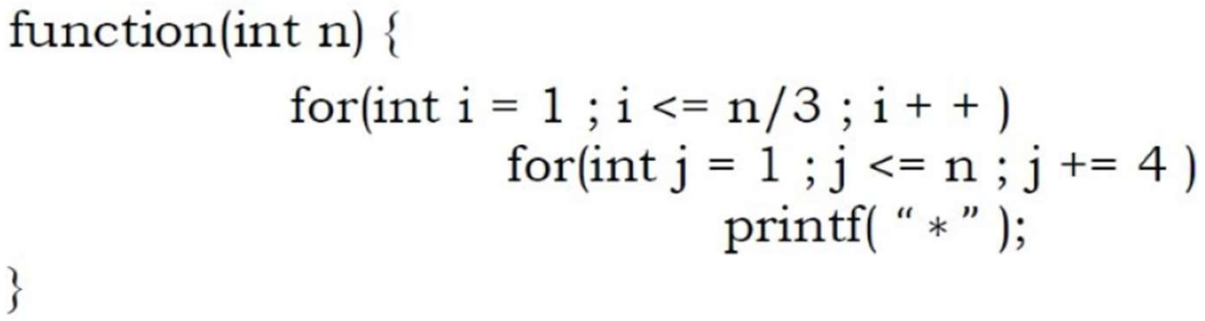
Answer:O(logn)

What is time complexity of following function?



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

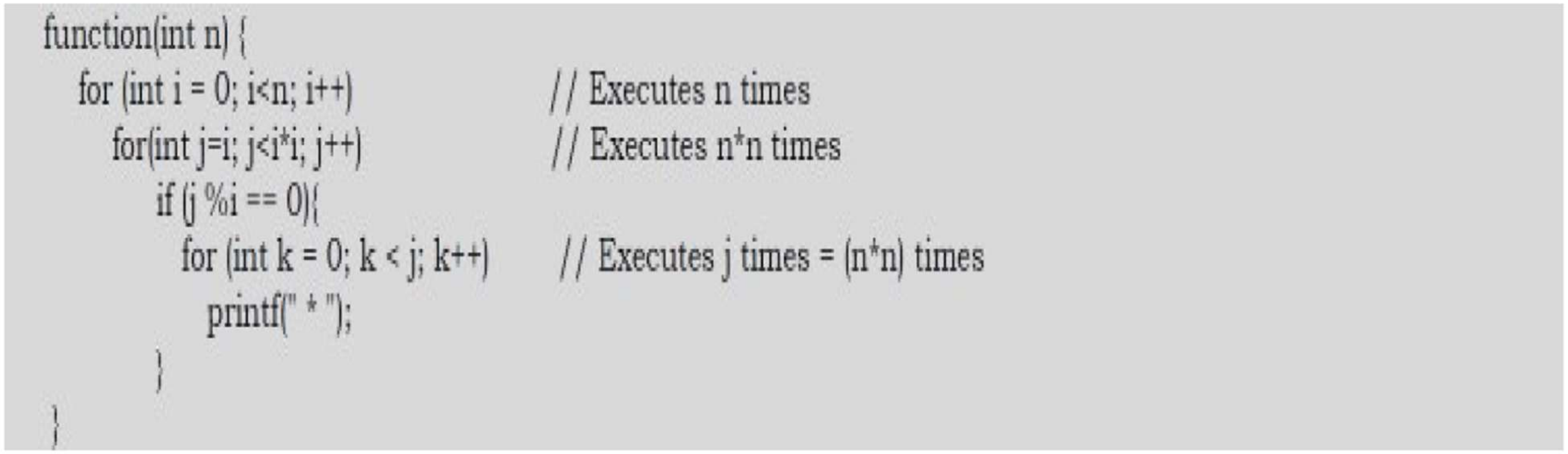
What is time complexity of following function?



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

Answer: $O(n^2)$

What is time complexity of following function?



```
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(" * ");
         }
}
```

Answer: $O(n^5)$

Time complexity of Recursive function

Solve the following recurrence relations and represent in Big O notation

1. T(n)= c + T(n-1) if n>1

   =d,   if n<=1

2. T(n)=0,   if N=0

   =2T(n-1)+1, if N>0

Time complexity of Recursive functions

Solve the following recurrence relations and represent in Big O notation

1. $T(n) = c + T(n-1)$ if $n>1$

   $= d$, if $n<=1$

2. $T(n)=0$, if $N=0$

   $=2T(n-1)+1$, if $N>0$

Answer:

1. $O(n)$
2. $O(2^n)$

Space Complexity

- Components:
  - ❑Fixed space requirements **(c)**: doesn't depend on number and size of the program's inputs and outputs.
    - ❑Instruction space
    - ❑Space for simple variables, constants
  - ❑Variable space requirements $S_P(I)$: depends on particular instance of the problem
    - ❑Data Space: n or input array size
    - ❑Environmental stack space
- Thus, space needed by a program

$$S(P)=c+S_P(I)$$

Space Complexity

```
float abc(float a, float b,float c)
{
 return a+b+b*c+(a+b-c)/(a+b)+4.00;
}
```

$S(P)=c+S_P(I)$

$=3(4)+0=12$

Space complexity: O(1)

Space Complexity

```
float sum(float list[], int n)
{
  float tempsum=0;
   int i;
   for(i=0;i<n;i++)
       tempsum+=list[i];
  return tempsum;
}
```

$S(P)=c+S_P(I)$
$=3(4)+n(4)$
Space complexity: $O(n)$

Example 1: To find sum of two numbers

```
Algorithm Sum(a, b)
{
return a+b;
}
```

$S(P)=c+S_P(I)$
$=2(4)+0$
Space complexity: $O(1)$

Example 2a: To find factorial of a given number

```
Algorithm Factorial(n)
{
 fact=1;
 for(i=1;i<=n;i++)//n+1
         fact=fact*n;//n
 return fact;
}
```

$S(P)=c+S_P(I)$
$=3(4)+0$
Space complexity: O(1)
Time Complexity: O(n)

Example 2b: Recursive algorithm to find factorial of a given number

```
Algorithm RFactorial(n)
{
  if(n<=1) return 1;// 4bytes
  return n*Rfactorial(n-1);//return address has 4 bytes
}
```

$S(P)=c+S_P(I)$

$=0+8*n$

Space complexity: O(n)

Time Complexity:O(n)

n	No. of times Rfactorial is called
2	1[Rfactorial(2)]+1[Rfactorial(1)]
3	3
4	4
k	k

Example 3: Sum of n numbers

```
Algorithm Sumoflist( A[], n)
{
    sum=0;//sum occupies 4 bytes
    for(i=0;i<n; i++)// i, n -4 bytes each
        sum= sum + A[i];
    return sum;
}
```

S(P)=c+S(I)
c=4(3)
S[I]=n(4)
S(P)=12+4n
Space complexity: O(n)

Data Abstraction
-Abstract Data Type(ADT)

- By default, all primitive data types (int, float, etc.) support basic operations such as addition and subtraction.
  - The system provides the implementations for the primitive data types.
  - E.g.
    - Data type *int* consists of objects {0,+1,-1,+2,-2,…INT_MAX, INT_MIN}
    - INT_MAX and INT_MIN are the largest and smallest integers that are defined in limits.h
    - Operators on integers include infix operators like +, -, *, / and % , prefix operators like atoi(), sqrt() etc.
- In general, user defined data types are defined along with their operations.
- We combine the data structures with their operations and we call this *Abstract Data Types* (ADTs).

Data Abstraction
-Abstract Data Type(ADT)

- An ADT consists of 2 parts:
    1. Declaration/Specification of data
    2. Declaration/Specification of operations on the data: specification of operation or function includes function name, types of its arguments and the type of result.

- ADT focuses on specification only and is implementation independent
- ADT is used to specify the functions that can be applied on the data by program designer
- Commonly used ADTs *include:* Linked Lists, Stacks, Queues, Priority Queues, Binary Trees, Dictionaries, Disjoint Sets (Union and Find), Hash Tables, Graphs, and many others.

Data Abstraction -Abstract Data Type(ADT)

- The ADT definition begins with the name of ADT
- ADT has two main sections
    1. Data/Objects
    2. Functions/Operations
- Example:

ADT Array is
Data: Sequence of elements of same type
Functions:
Create(a, n): to create an array a of n values
Store(a, i, e): to store an element e at index i in array a
Retrieve(a, i) : to retrieve an element stored at index i

SEARCHING TECHNIQUES

Linear Search

Binary Search

LINEAR SEARCH

- It is also called as Sequential Search
- Searches for a particular value (*KEY)* in an array sequentially
- Compares *KEY* with every element in the array in a sequence until a match is found
- It is mostly used to search an unordered list of elements.

LINEAR SEARCH

Example: A[]={1,22,33,55,44},

- key=55
- Comparisons:
    - 1==55 : F
    - 22==55:F
    - 33==55:F
    - 55==55: True =>Key found

- Key=7
- No. of comparisons:5
    - 1==7,22==7,33==7,55==7,44==7:False =>Key not found

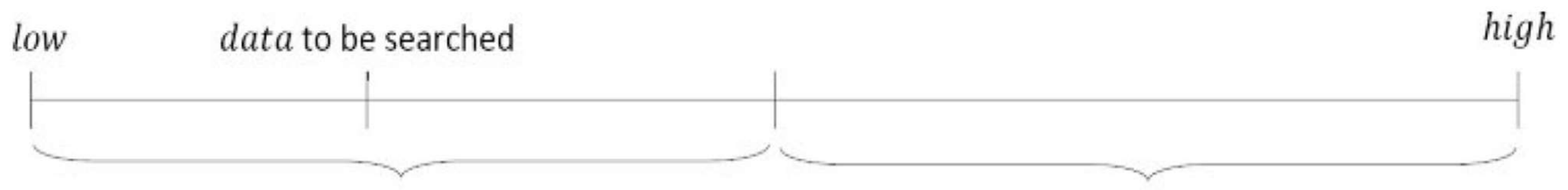
LINEAR SEARCH ALGORITHM

```
Algorithm LinearSearch(A[],key,n)
{
for(i=0;i<n;i++){
if (A[i]==key){
 print("key found");break;}
}
if(i==n)
print("Key Not Found")
}
```

Time complexity: O(n)

BINARY SEARCH

- Searches for a particular value (*KEY)* in a sorted array with run-time complexity of O(logn)
- It works on the principle of Divide and Conquer
- Compares *KEY* with middle element(*mid)* in the array



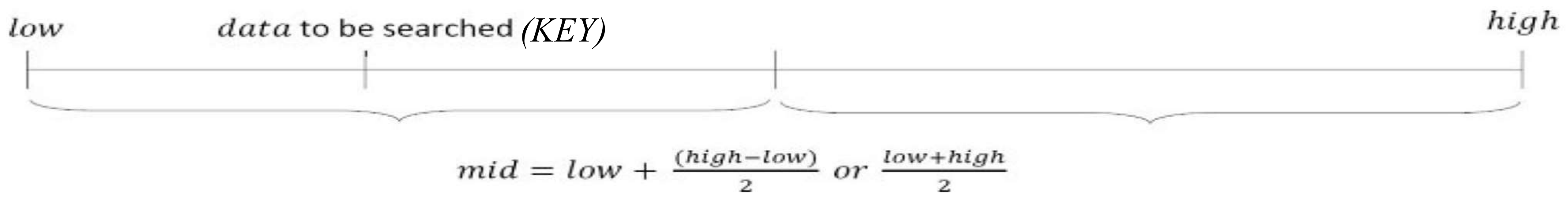
$$mid = low + \frac{(high-low)}{2} \ or \ \frac{low+high}{2}$$

BINARY SEARCH

In binary search,

1. If *KEY==A[mid],* then search successful.
2. If *KEY* < A*[mid],* then KEY will be present in the left segment of the array. So, the value of *high* will be changed as *high* = *mid*– 1.
3. If *KEY* > A*[mid],* then *KEY* will be present in the right segment of the array. So, the value of *low* will be changed as *low*= *mid* + 1.

- Finally, if *KEY* is not present in the array, then eventually, *high* will be less than *low*. When this happens, the algorithm will terminate and the search will be unsuccessful.


low
data to be searched (KEY)
high
$mid = low + \frac{(high-low)}{2}$ or $\frac{low+high}{2}$

BINARY SEARCH

Example: A[]={1,22,33,44,55},

- key=55
- Comparisons: low=0, mid=2, high=4
    - 55==33(A[2]):F
    - 55>33:True => low=3, mid=3, high=4
    - 55==44 (A[3]): :F ,55>44=>low=4,mid=4,high=4
    - 55==55((A[4]): : True =>Key found

BINARY SEARCH

Example: A[]={1,22,33,44,55},

- key=7
- Comparisons: low=0, mid=2, high=4
  - 7==33(A[2]):F
  - 7<33:True => low=0, mid=0, high=1
  - 7==1(A[0]): :F ,7>1=>low=1,mid=1,high=1
  - 7==22((A[1]): :F,7<22=>low=1,high=0
  - low>high, True =>Key not found
- More Examples: https://www.cs.usfca.edu/~galles/visualization/Search.html

BINARY SEARCH ALGORITHM

```
Algorithm BinarySearch(A[],key,n)
{
low=0;high=n-1;
mid=(low+high)/2
while(low<=high){
     if (key==A[mid]){ print("key found");break;}
     else if (key<A[mid]){ high=mid-1;}
     else          low=mid+1;
}
if(low>high)  print("Key Not Found")
}
}
```

Time complexity: O(logn)