Data Structures and Algorithms

Module 1

Let me know, if any typos or any other type of mistakes are in shared PPT.

Module No. 1		
Notion of an Algorithm, Fundamentals of Algorithmic Problem Solving, Important Problem Types, Fundamentals of the Analysis of Algorithm Efficiency, Analysis Framework, Asymptotic Notations and its properties, Mathematical analysis for Recursive and Non-recursive algorithms.		

Algorithm

- An algorithm is a step by step procedure for solving the given problem.
- An algorithm is independent of any programming language and machine.
- An Algorithm is a finite sequence of effective steps to solve a particular problem where each step is unambiguous(definite) and which terminates for all possible inputs in a finite amount of time.
- An Algorithm accepts zero or more inputs and produces one or more outputs.

The criteria for any set of instruction for an algorithm is as follows:

1. **Input** : Zero of more quantities that are externally applied
2. **Output** : At least one quantity is produced
3. **Definiteness** : Each instruction should be clear and unambiguous
4. **Finiteness** : Algorithm terminates after finite number of steps for all test cases.
5. **Effectiveness** : Each instruction is basic enough for a person to carried out using a pen and paper. That means ensure not only definite but also check whether feasible or not.

Definiteness & Effectiveness

- Each instruction should be clear and unambiguous.
- It must be perfectly clear what should be done.
- Example Directions which are not permitted

  "add 6 or 7 to x"

  compute x/0

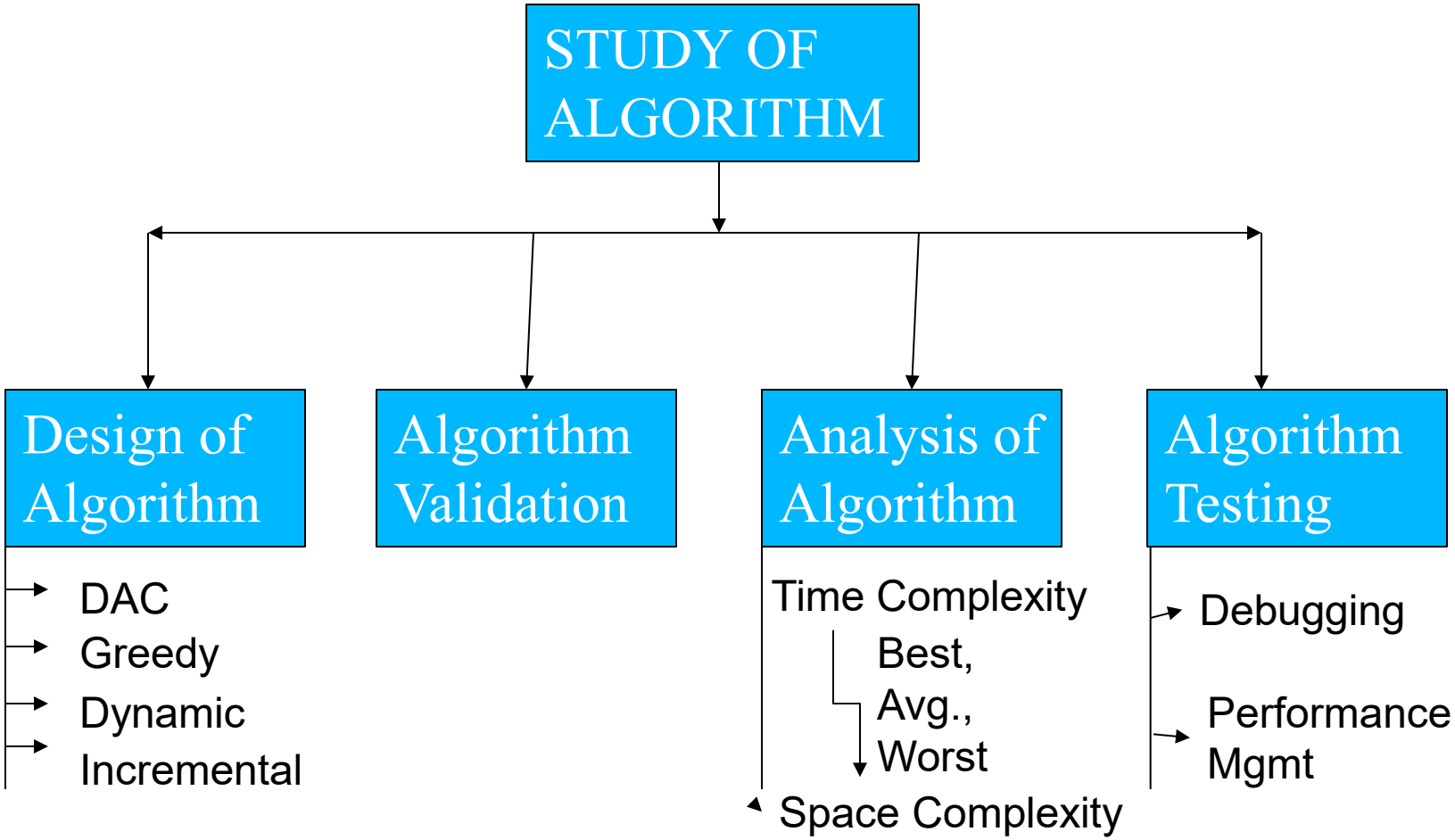
  remainder of m/n when m and n are –ve numbers
- it is not clear which of the 2 possibilities should be done.
- Achievement using programming language for algorithms designed such that each legitimate sentence has a unique meaning

Effectiveness

- Each step must be such that it can at least in principle be done by a person using paper and pencil in a finite amount of time.
- Example -Effective
- Performing arithmetic on integers
- Example not effective
- Some arithmetic with real numbers.

Finiteness

- The algorithm should terminate after a finite number of steps in all the cases.
- The time for termination should be reasonably short.



- ***Design:*** Various design techniques are available which yield good algorithm. **Divide and conquer**, incremental approach, **greedy**, **dynamic programming** etc.
- ***Divide and Conquer:*** Most of the algorithms are recursive in nature. That means, for solving a particular problem they call themselves repeatedly one or more times. All these algorithm follows DAC approach.
- In this approach, whole problem is divided into several sub problems. These sub problems are similar to the original problem but smaller in size. All these sub problems are solved recursively. The solution obtained from these sub problems are then combined to create a solution to the original problem.

- ***Step 1: Divide-*** In this step whole problem is divided into several sub problems.
- ***Step 2: Conquer-*** The sub problems are conquered by solving them recursively only if they are small enough to be solved.
- ***Step 3: Combine-*** Finally the solution obtained by the sub problems are combined to create solution to the original problems.

- ***Analysis:*** Analysis of algorithm depends upon various factors such as memory, communication bandwidth, or computer hardware.
- But the most often used is the Computation time that an algorithm requires for completing the given task.
- The analysis of algo. Focuses on time and space complexity.
- For an algo. the time complexity depends upon the size of the input, thus it is a function of i/p size "n". The amount of time needed by an algo. to run to completion.
- It may be noted that different time can arise for the same algorithm. Usually we deal with Best, Avg, and Worst case

- ***Best Case:*** The minimum amount of time that an algorithm requires for an input size n is referred to as Best case time complexity.

- ***Avg. Case:*** It is the execution of an algorithm having typically input data of size n.

- ***Worst Case:*** The maximum amount of time needed by an algorithm for an input size n is referred to as worst case time complexity.
- The amount of memory needed by program to run to completion is referred to as space complexity.

Algorithms Validation and Testing

- Algorithm validation checks the algo results for all legal set of input. After designing the algo it is necessary to check the algorithm whether it computes the correct and desired result or not, for all possible legal set of input.
- After that we check the program output for all possible set of input and called as program verification.

- ***Testing:*** It consist of 2 phases i.e. debugging and performance measurement. Debugging is the process of finding and correcting the cause as variance with the desired and observed behavior.
- Debugging can only point to the presence of errors.
- Performance measurement or profiling precisely describes the correct program execution for all possible data sets.

Good algorithm

- One which works correctly
  - should give the desired output
- Readable
  - Steps are clear and understandable
- Efficient in terms of time and memory utilization .
  - give the desired output faster
  - utilize less resources.
- Given multiple solutions, finding out the better one.

Mathematical Formulae Contd..

$\log_a xy = \log_a x + \log_a y$

$\log_a (x/y) = \log_a x - \log_a y$

$\log_a x^n = n \log_a x$

$\log_a 1 = 0$

$\log_x x = 1$

$\log_a x = 1/\log_x a$

$\log_a(x) = \log_a(b) * \log_b(x)$

$\log_a x = \log_b x / \log_b a$

$a^{\log_a x} = x$

if $2^k = n$ then $k = \log_2 n$

$x^{\log_b y} = y^{\log_b x}$

Performance Analysis

- To improve existing algorithms
- To choose among several available algorithms
- Two types
  - Apriori Analysis
  - Aposteriori Analysis

Space Time Tradeoff Example

Store Employee information

- Solutions
  - Array
  - Linked list
- Which locates a particular employee faster?
- Which provides better utilization of memory?
- Which is easy to code?
- Which is easier to test?

Space Complexity

- Amount of memory the algorithm needs to run to completion
- Requirements
  - Fixed part
  - Variable part
- S(P)=c+Sp

Example Algorithm

```
Algorithm sum(a,n)
{s = 0.0;
for i:=1 to n do s:=s+a[i];
return s;  }
```

one word for each integer n,i,s,a[]

s(n) >= n+3

Example Algorithm


```
algorithm Rsum(a,n)
{ if (n<=0) then
        return 0.0
  else
        return Rsum(a,n-1)+a[n];
}
```

- Instance characterized by n
- Recursive stack space required at each level
  - (n,return address,*a[]) 3 words
- Depth of recursion considering first invocation - (n+1)
- S(n)>=3*(n+1)

Time Complexity

- Amount of computer time needed for the algorithm to run to completion
- T(P) = C(P) + R(P)
- C(P) is independent of problem instance and is always not required (recompilation)
- So usually only R(P) is considered

Apriori Measures

- Instruction count
  - Basic operations
  - Basic steps
- Size of input.

Computing Instruction Count

- Initialization instructions
- Loops.
  - Number of passes of the loops
- Count the basic operations/steps

Counting Steps

- Time complexity can also be expressed in terms of the basic steps performed.
- Expressed as a function of instance characteristics
  - input size
  - output size etc .

Counting Steps Implementation

- Assignment – count 1
- For, while, repeat loops
  - Using a variable count
  - Using steps per execution of instructions

Use of a variable count

- Use a variable count to store step count and increment it wherever necessary
- Display count

Computing Step Count from s/e

- Each statement will have an (s/e) steps/execution depending on the type of statement
- Frequency of each statement is multiplied with the corresponding s/e of that statement to get the step count for that statement.
- Statement Step counts are added to give the total count

$$\sum f \times (s/e)$$

Example 1

```
Algorithm sum(a,n)
{ s=0.0;
for i: =1 to n do
{ s:=s+a[i];  }
return s;       }
```

s/e	freq	total
1	1	1
1	n+1	n+1
1	n	n
1	1	1
		2n+3

Example 2

```
algorithm add(a,b,b,m,n)
{
for i:=1 to m do
        for j:=1 to n do
    c[i,j]:=a[i,j]+b[i,j];
 }
```

S/e	Freq	Total
1	m+1	m+1
1	m(n+1)	m(n+1)
1	mn	mn

		2mn+2m+1

End

Design and Analysis of Algorithm

Module 1.1

Dr. K Hemant Reddy
SCOPE

Example 1

Algorithm sum(a,n)	s/e	freq	= total
{ s=0.0;	1	1	= 1
for i: =1 to n do	1	n+1	= n+1
{ s:=s+a[i]; }	1	n	= n
return s; }	1	1	= 1

			= 2n+3

Example 2

algorithm add(a,b,b,m,n)	S/e freq	= Total
{for i:=1 to m do ⟶	1 m+1	= m+1
for j:=1 to n do ⟶	1 m(n+1)	= m(n+1)
c[i,j]:=a[i,j]+b[i,j]; ⟶	1 mn	= mn
}		---
		2mn+2m+1

Example 3

Algorithm sum(M1,M2,n)	s/e	freq	= total
{ M3[][];	1	1	= 1
for i: =1 to n do	1	n+1	= n+1
{ for j: =1 to n do	1	n*n	= n^2+n
{ for k: =1 to n do	1	n*n*n	= n^3+n^2
M3[i][j]+=M1[i][k]*M2[k][j];	1	n*n*n	=n^3
}			
}			= $2n^3+2n^2+2n+2$
}			

- The efficiency of an algorithm depends on the amount of time, storage and other resources required to execute the algorithm. The efficiency is measured with the help of **asymptotic notations**.

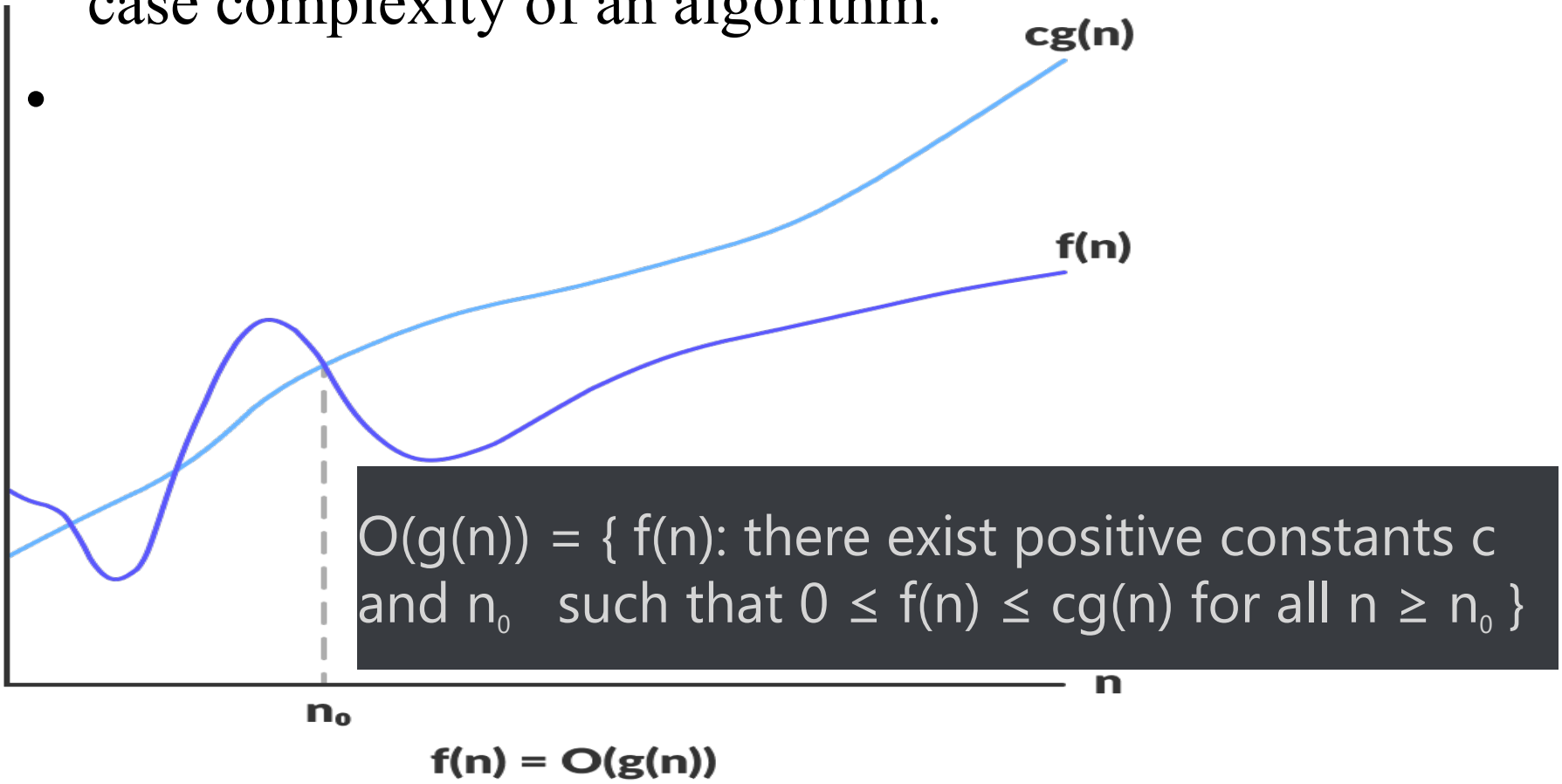
- The study of **change in performance** of the algorithm with the **change in the order of the input size** is defined as asymptotic analysis

The following 3 asymptotic notations are mostly used to represent the time complexity of algorithms.

- **Big-O notation**
- **Omega notation**
- **Theta notation**

Big-O Notation (O-notation)

- **Big-O notation** represents the upper bound of the running time of an algorithm. Thus, it gives the worst-case complexity of an algorithm.
-



$O(g(n)) = \{ f(n)$: there exist positive constants c and n_0 such that $0 \leq f(n) \leq cg(n)$ for all $n \geq n_0 \}$

The above expression can be described as a function **f(n)** belongs to the set **O(g(n))** if there exists a positive constant **c** such that it lies between **0 and cg(n),** for sufficiently large **n**.

For any value of **n**, the running time of an algorithm does not cross the time provided by **O(g(n)).**

Since it gives the **worst-case running time** of an algorithm, it is widely used to analyze an algorithm as we are always interested in the worst-case scenario.

Properties

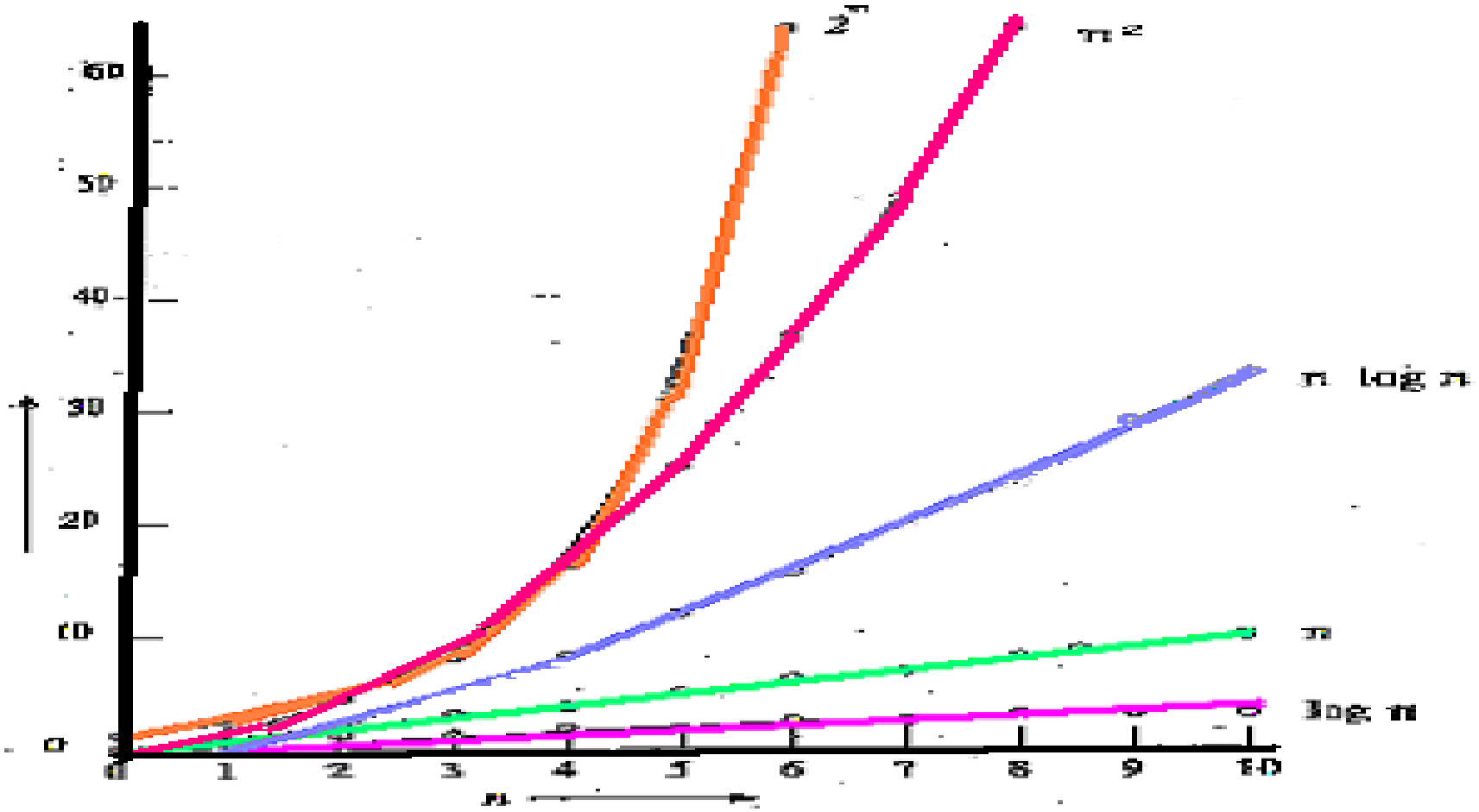
- Ignore all constant values
- Always measure n → infinity
- In case of upper bound, we need to take least upper bound
- In case of lower bound, we need to take greatest lower bound

BETTER

WORSE

- $O(1)$ constant time
- $O(\log n)$ log time
- $O(n)$ linear time
- $O(n \log n)$ log linear time
- $O(n^2)$ quadratic time
- $O(n^2 \log n)$ log quadratic time
- $O(n^3)$ cubic time
- $O(2^n)$ exponential time
- $O(n^n)$ exponential time

Running Times- Comparison



Example 1

- f(n)=3n+6, prove that f(n) <= O C*g(n)
- **You have find C and n_0 value such way that it should satisfy the condition.**
- **f(n)=3n+6 and if we can take g(n)=3n+6n then**
- 3n+6 <= 3n+6n
- 3n+6<=9n, where C=9
- Whereas n>=n0, and here for all n >=n_0 , n_0>=1
- **Can I take g(n)as 3n+3, if yes find C and n0**

Example 2

- If f(n)=2n+4 then prove that f(n)<=C*g(n)
- **You have find C and n_0 value such way that it should satisfy the condition.**
- Lets take g(n)= 2n+n
- Where for all n>=n0
- ➔2n+4<= 2n+n
- ➔2n+4 <= 3n and C=3
- **The above holds true where C=3**
- **and n0>=4**

n	f(n)	g(n)
1	6	3
2	8	6
3	10	9
4	12	12
5	14	15
6	16	18
7	18	21

Example 3

- If f(n)=2n+4 and let me take g(n)= 2n+2n
- f(n)<=C*g(n)
- **You have find C and n_0 value suc t should satisfy the condition.**

n	f(n)	g(n)
1	6	4
2	8	8
3	10	12
4	12	16
5	14	20
6	16	24
7	18	28

- ➔2n+4<= 4n
- ➔2n+4 <= 4n and C=4
- Where for all n>=n_0, and n_0 >=2

Example 4


```
public class Example{
public static void main(String[] args) {
    int i,n=100;
     for(i=1; i<=n; i=i*2){
         System.out.println(i);
         }
}
}
```

If n=1
1 time

If n=2
2 times

If n=4
3 times

If n=10
4

If n=20
5 times

‘i‘ value ➔	1	2	4	8	16	32	64	128
	2^0	2^1	2^2	2^3	2^4	2^5	2^6	2^7
Here	k=0, 1, 2, 3, 4, 5, 6, 7							
Prints ➔	1	1	1	1	1	1	1	1

$2^k <= n$, apply both side log, then $k = \log_2 n$

$F(n)=O(\log_2 n)$

Example 5

- for(i=1;i<=n;i=i*2)
  - print("xxxx");

Value of "i" ➔ 1, 2, 4, 8, 16 , 32, 64,….

➔ $2^0, 2^1, 2^2, 2^3\ 2^4, 2^5\ , 2^6, \ldots..2^k$

➔ $2^k = n$

➔ Apply log both sides, $\log_2(2^k) = \log_2(n)$

➔ $k \log_2(2) = \log_2(n)$

➔ $k = \log_2(n)$

➔ $T(n) = \log_2(n)$

Example 6

- for(i=1; i<=n; i=i*5)
  - print("xxxx");

Value of "i" ➔ 1, 5, 25, 125,....

➔ 5^0, 5^1, 5^2, 5^3 5^4, 5^5 , 5^6, ...5^k

➔ $5^k = n$

➔ Apply log both sides, $\log_5(5^k) = \log_5(n)$

➔ $k \log_5(5) = \log_5(n)$

➔ $k = \log_5(n)$

➔ $T(n) = \log_5(n)$

Example 7

- for(i=n;i>=1; i=i/2)
  - print("xxxx");

Value of "i" ➔ n, n/2, n/4, n/8, n/16 , n/32, ….

➔ $n/2^0$, $n/2^1$, $n/2^2$, $n/2^3$, $n/2^4$, …..$n/2^k$

➔ $n/2^k = 1$

➔ $n = 2^k$

➔ Apply log both sides, $\log_2(2^k) = \log_2(n)$

➔ $k \log_2(2) = \log_2(n)$

➔ $k = \log_2(n)$

➔ $T(n) = \log_2(n)$

Example 8

- for(i=2;i<=n; i=i*i)
  - print("xxxx");

Value of "i" ➔ 2, 4, 16 , 16*16,….

➔ $2^{2^0}, 2^{2^1}, 2^{2^2}, 2^{2^3}, 2^{2^4}, \ldots 2^{2^k}$

➔ $2^{2^k} = n$

➔ Apply log both sides, $\log_2(2^{2^k}) = \log_2(n)$

➔ $2^k \log_2(2) = \log_2(n)$

➔ $2^k = \log_2(n)$, again apply log

➔ $\text{Log}_2(2^k) = \log_2(\log_2(n))$

➔ k **$\text{Log}_2(2)$** $= \log_2(\log_2(n))$ ➔ **$k = \log_2(\log_2(n))$**

Example 9

- for(i=n; i>=2; i=sqrt(i))
  - print("xxxx");

Value of "i" ➔ n, $n^{1/2}$, $n^{1/4}$, $n^{1/8}$, ….. ➔ $n^{1/(2^0)}$, $n^{1/(2^1)}$, $n^{1/(2^2)}$, $n^{1/(2^3)}$…. $n^{1/(2^k)}$

➔ $\mathbf{n^{1/2^k} >= 2}$

➔ Apply log both sides, $\log_2(n^{1/2^k}) = \log_2(2)$

➔ $^{1/2^k}\log_2(n) = 1$

➔ $\log_2(n) = 2^k$

➔ Apply log both sides again

➔ $\log_2(2^k) = \log_2(\log_2(n))$

➔ $k \log_2(2) = \log_2(\log_2(n))$

➔ $\mathbf{k = \log_2(\log_2(n))}$

Example 10


```
for(i = n / 2; i <= n; i++)
    for(j = 2; j <= n; j = j * 2)
        {              k = k + n / 2;       }
```

- For i value➔ n/2, (n/2)+1, (n/2)+2, (n/2)+3,….n
- **It iterates n/2 times which is same as n, when n➔ infinity**
- For j value➔2, 4,8,16,32, 64, 128…..
  - $2^1, 2^2, 2^3\ 2^4, 2^5, 2^6, \ldots..2^k$
  - $\log_2(n)$
- $T(n)=n\log_2(n)$

Example 10

- for(i=1; i<=2^n; i=i*2)
  - print(“xxxx”);

Example 9

- for(i=1; i<=n^2; i=i+10)
  - print(“xxxx”);

Example 10

- for(i=n/2; i<=n; i=i*i)
  - print(“xxxx”);

Example 11

- for(i=1; i<=n; i++)
  - for(i=1; i<=n; i++)
    - for(i=1; i<=n; i++)
      - Print("XXX");

Example 12

- for(i=1; i<=n^2; i=i*2)
  - for(j=n; j>=n/2; j=j/2)
    - for(k=2; k<=2^n; k=k^2)
      - Print("XXX");

Example 13

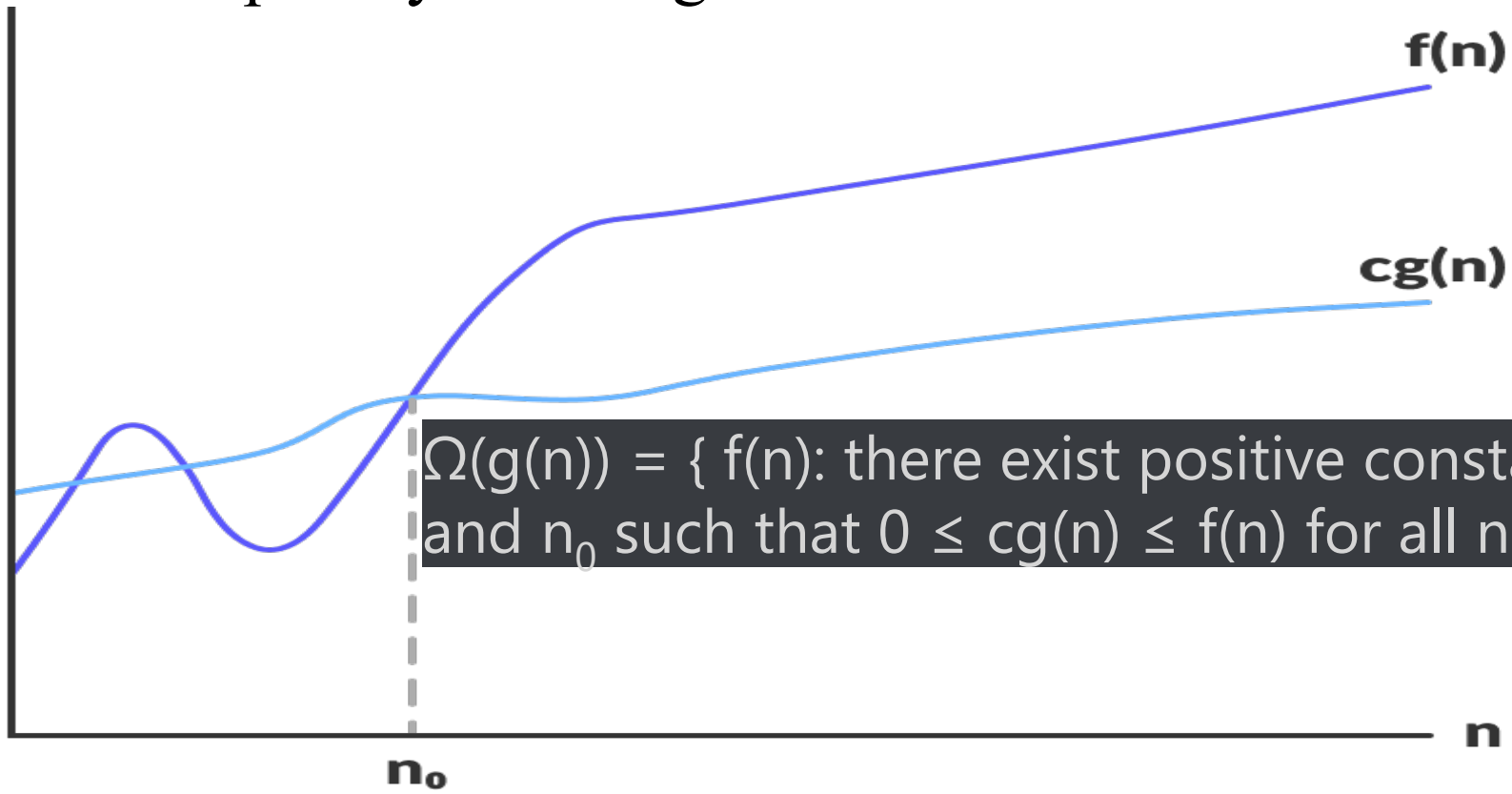
- for(i= n^2; i>=1; i=i/2)
  - for(j=1; j<=n; j=j*10)
    - for(k=1; k<=2^n; k=2*k)
      - Print("XXX");

Prove the followings

- $3n+2=O(n)$
- $3n+2<=4n$ for $n>=2$
- $3n+3=O(n)$
- $3n+3<=4n$ for $n>=3$
- $100n+6=O(n)$
- $100n+6<=101n$ for $n>=10$
- $10n2+4n+2=O(n2)$
- $10n2+4n+2<=11n2$ for $n>=5$
- $6*2^n+n^2=O(2n)$
- $6*2^n+n^2$ $7*2n$ for $n>=4$

Omega Notation (Ω-notation)

Omega notation represents the lower bound of the running time of an algorithm. Thus, it provides the best case complexity of an algorithm.



Ω(g(n)) = { f(n): there exist positive constants c and n_0 such that $0 \leq cg(n) \leq f(n)$ for all $n \geq n_0$ }

Continue

- The above expression can be described as a function f(n) belongs to the set Ω(g(n)) if there exists a positive constant **c** such that it lies above **cg(n)**, for sufficiently large n.

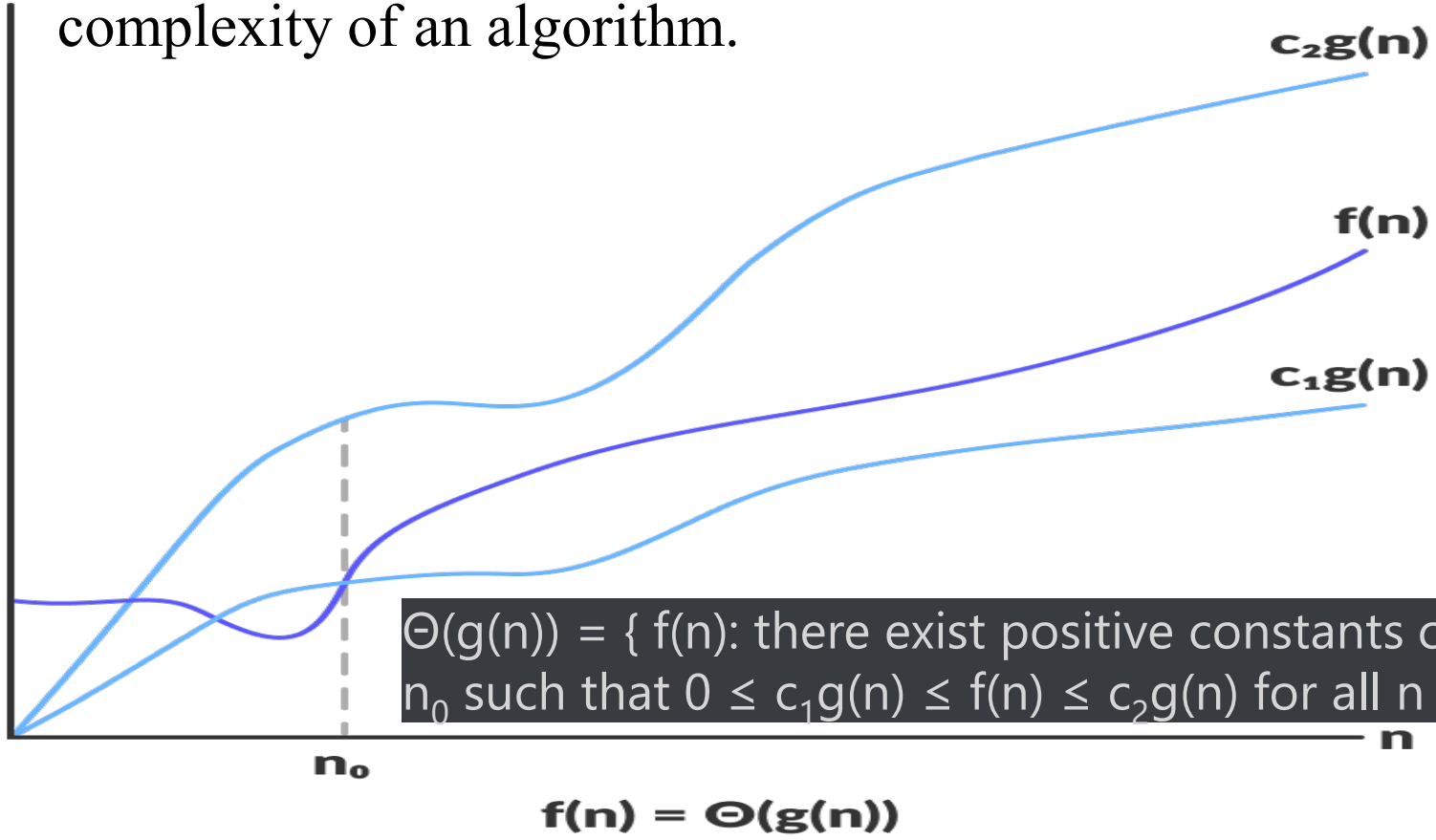
- For any value of **n**, the minimum time required by the algorithm is given by **Omega Ω(g(n)).**
-

Example

- Show that the following equalities are correct.
- $10n^2+4n+2= \Omega(n^2)$
- $F(n)= 10n^2+4n+2$, Let $C*g(n)=10n^2$
- $C*g(n)<=f(n)$ or $f(n)>= C*g(n)$
- $10n^2+4n+2 >=10n^2$
- $C=10$ and $n_0>=1$

Theta Notation (Θ-notation)

Theta notation encloses the function from above and below. Since it represents the upper and the lower bound of the running time of an algorithm, it is used for analyzing the average-case complexity of an algorithm.



$\Theta(g(n))$ = { f(n): there exist positive constants c_1, c_2 and n_0 such that $0 \le c_1 g(n) \le f(n) \le c_2 g(n)$ for all $n \ge n_0$ }

- The above expression can be described as a function f(n) belongs to the set $\Theta(g(n))$ if there exist positive constants c_1 and c_2 such that it can be sandwiched between $c_1g(n)$ and $c_2g(n)$, for sufficiently large n.
- If a function f(n) lies anywhere in between $c_1g(n)$ and $c_2g(n)$ for all $n \geq n0$, then f(n) is said to be asymptotically tight bound.

Example

- Show that the following equalities are correct.
- $5n^2-6n= \Theta(n^2)$
- Let F(n)= $5n^2$-6n, and we need to find C1g(n) and C2g(n) in such a way that: **C1*g(n)<=f (n) <=C2*g(n)**
- **C1*g(n)= $5n^2-n^2$ and C2*g(n)=$5n^2$**
- **$5n^2-n^2$ <=** $5n^2$-6n **<= $5n^2$**

N	C1g(n)	F(n)	C2g(n)
1	4	-1	5
2	16	8	20
3	36	27	45
4	64	40	80
5	100	50	125
6	144	144	180
7	196	203	245

Prove the followings

- $3n+2 = \Theta(n)$
- For $n>=2$, $c1=3$ and $c2=4$
- $10n2 +4n+2 >= \Theta(n2)$
- For $n=?$, $c1=10$ and $c2=11$
- $10n2 +4n+2 >= 10n2$
- $10n2 +4n+2 <=11n2$

Space Complexity Analysis

The **space complexity** of an algorithm or a computer program is the amount of memory space required to solve an instance of the computational problem as a function of characteristics of the input. It is the memory required by an algorithm until it executes completely.

Similar to time complexity, space complexity is often expressed asymptotically in big O notation, such as O(n), O(nlog n), $O(2^n)$ etc., where *n* is a characteristic of the input influencing space complexity.

Example


```
void sumFun( )
{    int a, b, c, sum;
     sum = a + b + c;
     return(sum);
}
```

S(n)=4 ➔ S(n)=C

```
void sumFun(int a, int b)
{    int sum;
     sum = a + b;
     return(sum);
}
```

S(n)=1 ➔ S(n)=C

Example


```
int sum(int arr[n], int n){
int B[n], k, i = 0;
for(k = 0; k < n; k++)
{       i  = i + arr[k];
        B[i]=arr[i]
}
return(i);
}
```

n+1+1 memory spaces

==========

Total=O(n)

Space Complexity

```
int factorial(int n) {
    if (n == 0)
        return 1;
    return n * factorial(n - 1);
}
```

F(5)→F(4)

F(4)→F(3)

F(3)→F(2)

F(2)→F(1)

F(1)→F(0)

Space Complexity=O(n)

In iteration, there is a repeated execution of the set of instructions. In Iteration, loops are used to execute the set of instructions repetitively until the condition is false.

Recursion is the process of calling a function itself within its own code.

Recurrence Relations

- Recurrence is an equation or inequality that describes a function in terms of its value on smaller inputs, and one or more base cases
  - Useful for analyzing recurrent algorithms
  - Make it easier to compare the complexity of algorithms
  - Methods for solving recurrences
    - **Substitution method**
    - **Recursion tree method**
    - **Master method**
    - **Iteration method**

3.1 Methods for Resolving Recurrence Relations

Recurrence relations can be resolved with any of the following four methods:

1. Substitution Method (Guess-and-Verify)
2. Iteration Method.
3. Recursion Tree Method.
4. Master Method.

Substitution Method

- Use mathematical induction to derive an answer
- Derive a function of n (or other variables used to express the size of the problem) that is not a recurrence so we can establish an upper and/or lower bound on the recurrence. May get an exact solution or may just get upper or lower bounds on the solution

Steps

- **Guess the form of the solution**
- **Use the mathematical induction to find the boundary condition and shows that the guess is correct.**

Recurrence Relation


```
int factorial(int n) {                          s/e  freq   = total
    if (n == 1)                                 1      1    =  1
      return 1;                                 1      1    =  1
    return n * factorial(n - 1);                1   1+f(n-1)= T(n-1)+1
}                                               ----------------------
                                                            = T(n-1)+3
```

T(n)={if n==1, 1
else 1+T(n-1)

Many algorithms are recursive. When we analyze them, we get a recurrence relation for time complexity.

Recurrence Relation


```
int Test(int n) {
    if (n == 1)
      S.o.p(n);
    else
    { for(i=0;i<n; i++)
        S.o.p(n);
      Test(n - 1);
    }
}
```

	s/e	freq	= total
if (n == 1)	1	1	= 1
S.o.p(n);	1	1	= 1
for(i=0;i<n; i++)	1	n+1	=n+1
S.o.p(n);	**1**	**n**	**=n**
Test(n - 1);	1	T(n-1)	=1+T(n-1)

T(n)={if n==1, 1
else 2n+1 +T(n-1)

Recurrence Relation


```
void test(int n)
{    if(n>1)
          sop(n)
          test(n/2);
     else
          s.o.p(n);
}
```

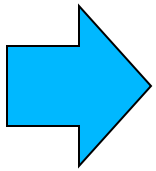
$T(n)=T(n/2)+1$

$T(n)=T(n/4)+1+1$

$=T(n/8)+3$

$=T(n/16)+4$

$=T(n/32)+5$



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

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

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

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

$=T(n/2^5)+5$

Let $2^k=n$

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

$=T(1)+k$

If $2^k=n$, then k=?

Apply log both side

$k=\log_2 n$

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

Substitution Method

Example1

```
A(n)
{
if (n>1)
return (A(n-1))
}
```

T(n) = 1 + T(n-1) ... Eqn. (1)

Step1: Substitute n-1 at the place of n in Eqn. (1)

T(n-1) = 1 + T(n-2) .. .Eqn. (2)

Step2: Substitute n-2 at the place of n in Eqn. (1)

T(n-2) = 1 + T(n-3) ... Eqn. (3)

T(n)= 1 + 1+ T(n-2) = 2 + T(n-2) ... Eqn. (4)

Recursion Example 2

T(n)=T(n-1)+3,for all value of n>0

T(0)=1

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

T(n-1)=T(n-2)+3

➔**T(n)=T(n-2)+3+3 => T(n-2)+6**

T(n-2)=T(n-3)+3

➔ **T(n)=T(n-3)+3 +6 =>T(n-3)+9**

➔T(n)=T(n-3)+3*3

➔T(n)=T(n-4)+4*3

➔T(n)=T(n-5)+5*3

➔T(n)=T(n-k)+3*k

Let n=k, then n-k=0

T(n)=T(n-k)+3k

T(n)=T(0)+3k

If n-k=0, then n=k

=T(0)+3n

T(0)=1

T(n)=1+3n, because k=n

T(n)=O(n)

Recursion Example 3

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

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

$=n+(n-1)+(n-2)+T(n-3)$

$= n+(n-1)+(n-2)+(n-3)+T(n-4)$

$= n+(n-1)+(n-2)+(n-3)+(n-4)+T(n-5)$

$=n+(n-1)+(n-2)+\ )+(n-3)+(n-4)+(n-5)+\ . . ..(n-(k-1))+T(n-k)$

……..let k=n

$= n+(n-1)+(n-2)+\ )+(n-3)+(n-4)+(n-5)+\ . . .\ .+(1)+T(0)$

$=\{n*(n+1)\}/2$

$=O(n^2)$

Recursion Example 5

- $T(n)=2T(n/2)+n$ and $T(1)=1$
- Replace n with n/2,
  - $T(n/2)=2T(n/4)+n/2$,
- $T(n)= 2[2T(n/4)+n/2]+n$
- ➔$4T(n/4)+n+n$ we can write➔ $2^2T(n/2^2)+2n$
- Replace n with n/4,
  - $T(n/4)=2T(n/8)+n/4$,
- $T(n)= 4[2T(n/8)+n/4]+2n$
- ➔$8T(n/8)+n+2n$
- we can write ➔ $2^3T(n/2^3)+3n$

Recursion Example 5

- we can write ➔ $2^3T(n/2^3)+3n$
- So, we guess the General equation,
- **$T(n)= 2^k * T(n/2^k)+kn$**
- **Let $n=2^k$**
- **$T(n)= 2^k T(1)+kn$**
- **then, $n= 2^k$ Apply log both sides, $k=\log_2(n)$**
- **Now $T(n)= n+n \log_2(n)$**
- **➔$n+n \log_2(n)$ ➔$O(n \log_2(n))$**

Recursion tree Method

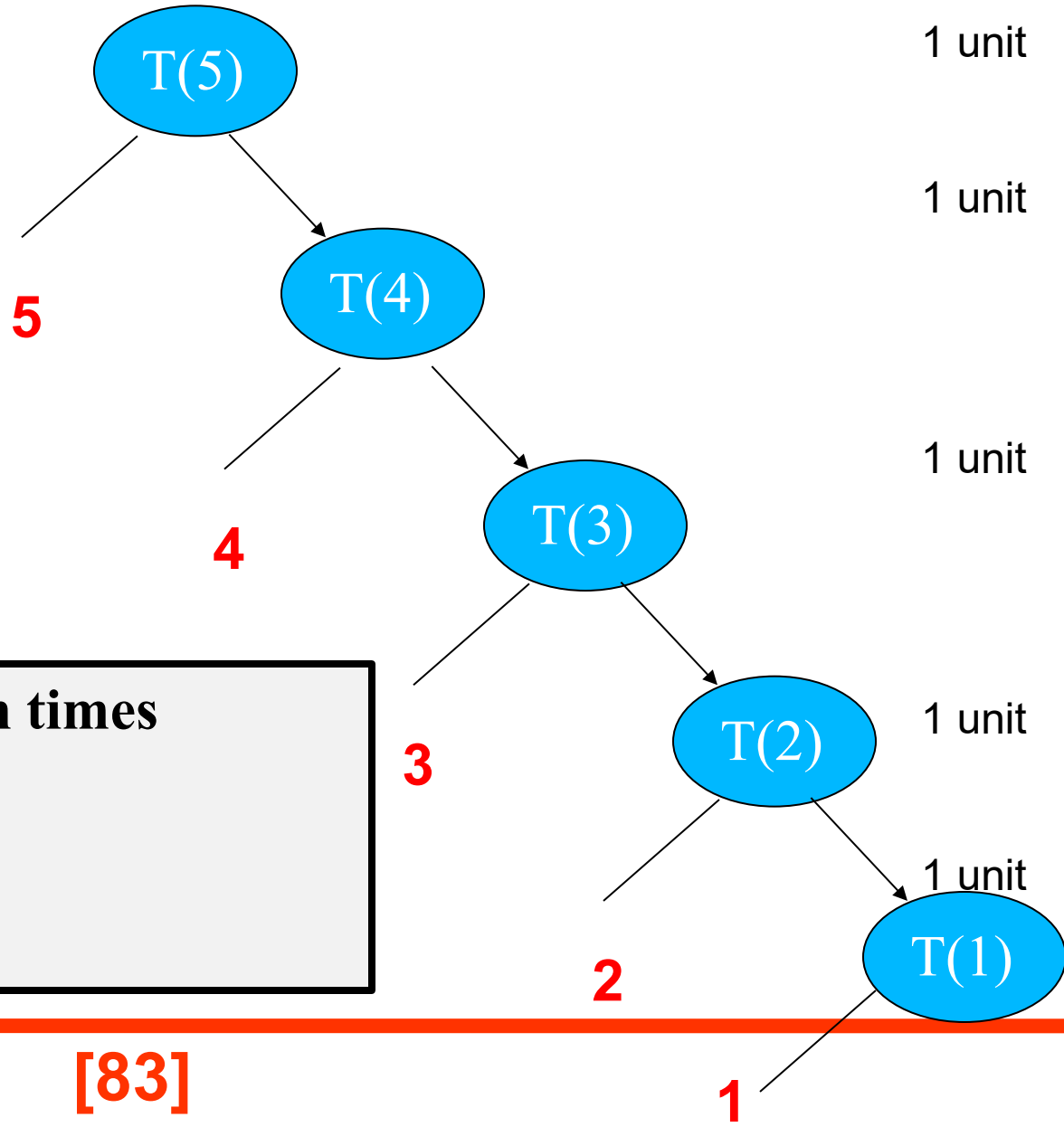
- Main disadvantage of Substitution method is that it is always difficult to come up with a good guess.
- Recursion tree method allows you make a good guess for the substitution method.
- Allows to visualize the process of iterating the recurrence

Steps

- **Convert the recurrence into a tree.**
- Each node represents the cost of a single sub problem somewhere in the set of recursive function invocations
- Sum the costs within each level of the tree to obtain a set of per-level costs
- Sum all the per-level costs to determine the total cost of all levels of the recursion.

Recursive Example 1


```
void Test(int n)
{
    if (n = = 1)
        S.O.P(1);
    else
    { S.O.P(n);
        Test(n - 1);
    }
}
```



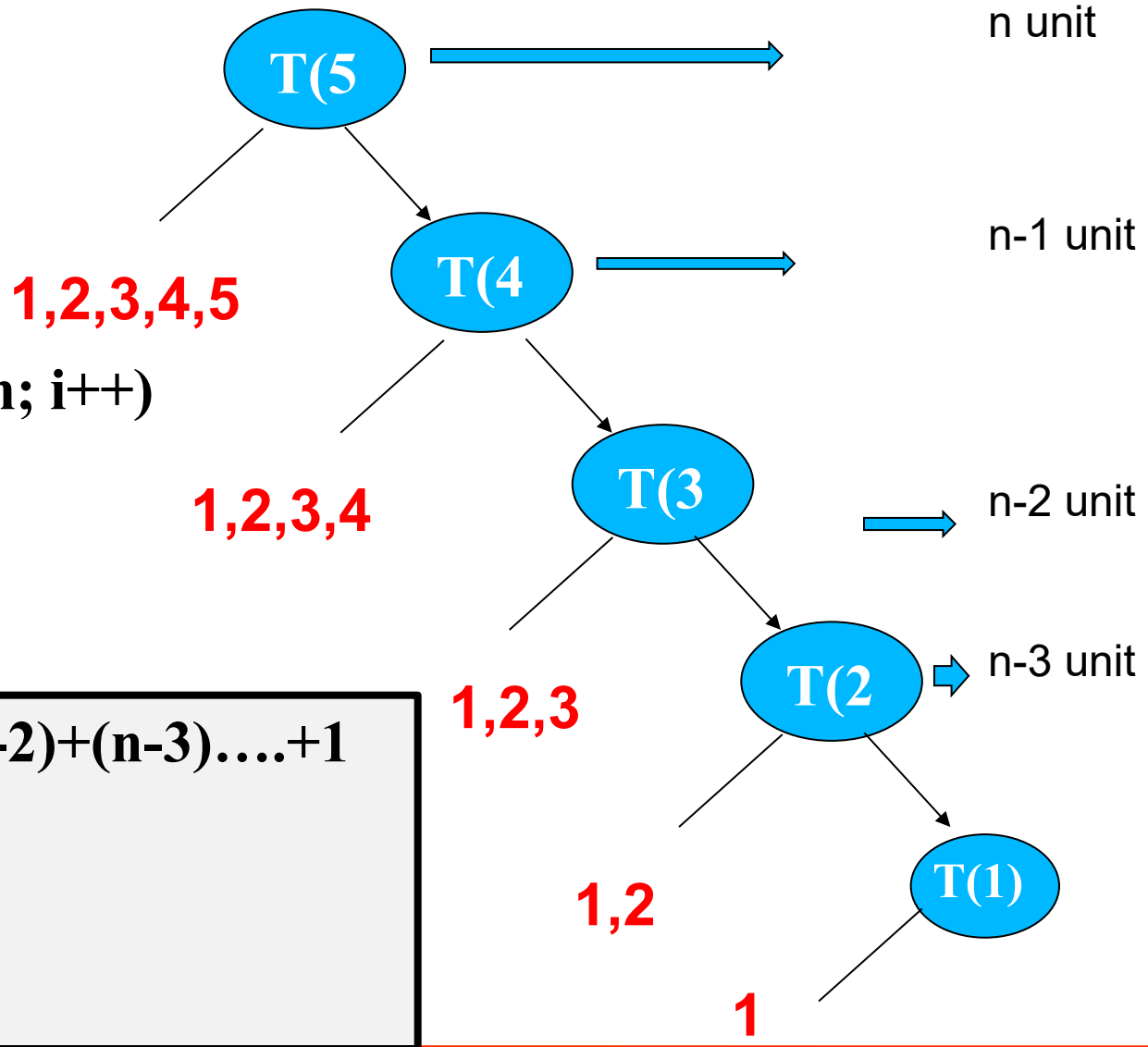
T(n)=1+1+1+1+……n times

=nTimes

=O(n)

Recursive: Using Recurrence Graph


```
void Test(int n) {
   if (n == 1)
       S.O.P(1);
   else
   { for(int i=1;i<=n; i++)
         S.O.P(n);
    Test(n - 1);
   }
}
```



$T(n)=n+(n-1)+(n-2)+(n-3)\ldots.+1$

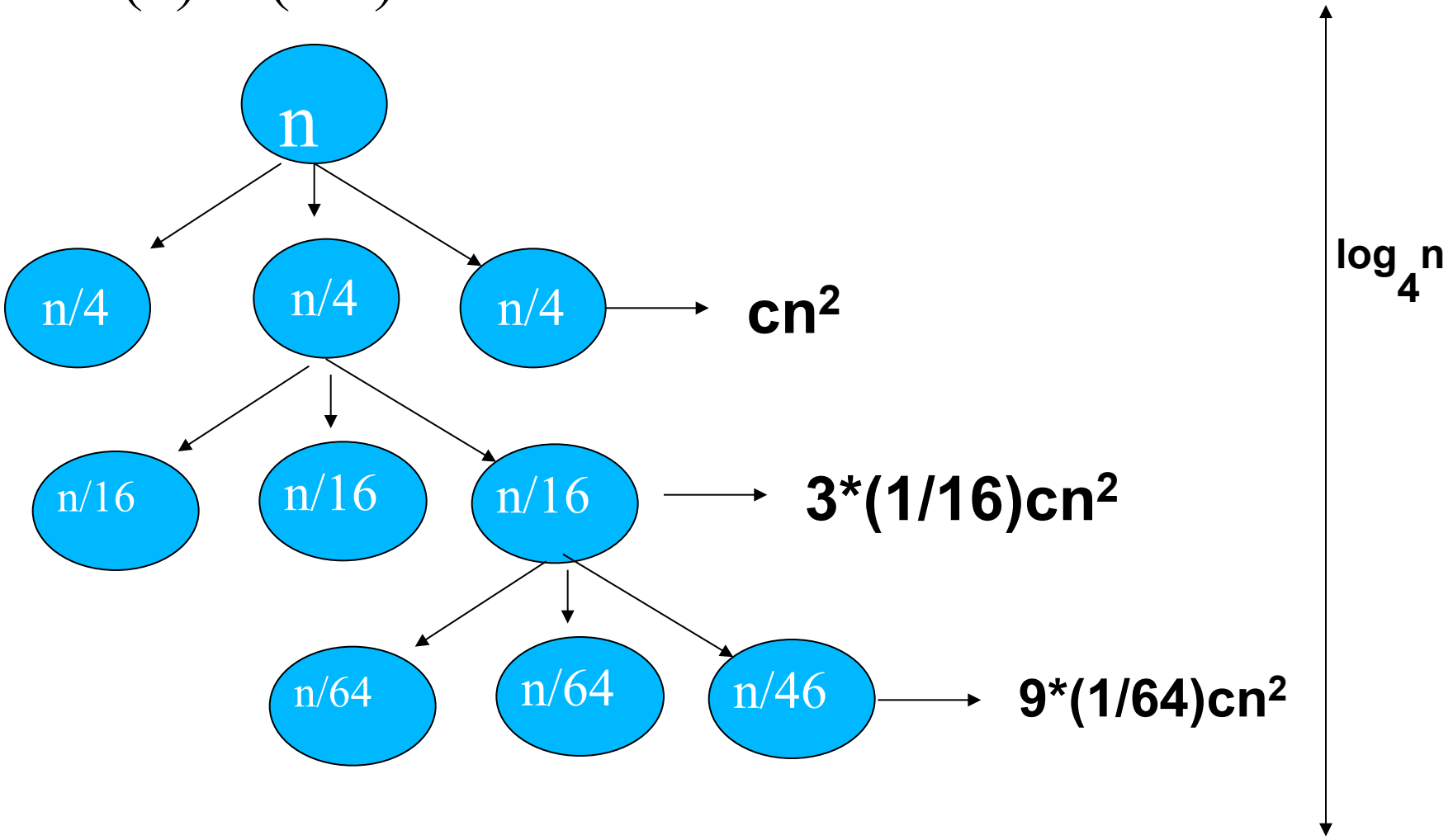
$T(n)=(n*(n+1))/2$

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

$=O(n^2)$

Example

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



$T(n)=cn^2+ (3/16)^2\, cn^2 + (3/16)^3\, cn^2 + (3/16)^4\, cn^2$ ……
$a= cn^2$, $r=(3/16)$, $m=\log n$

As per geometric progressive

The sum of first n terms of the Geometric progression is

$$S_n = a + ar + ar^2 + ar^3 + \ldots + ar^{n-2} + ar^{n-1} \text{ ----------> (1)}$$

The sum of the first m terms of a geometric progression is:

$$S_m = a\frac{r^m - 1}{r - 1}.$$

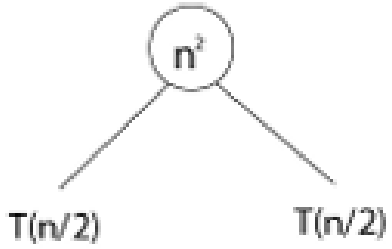
$$T(n) = Cn^2 \cdot \frac{\left(\frac{3}{16}\right)^{\log n} - 1}{\frac{3}{16} - 1}.$$

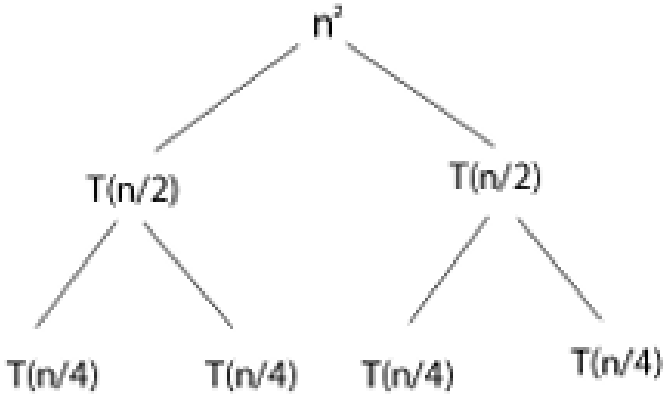
Recursion Tree Method

Consider $T(n) = 2T\left(\frac{n}{2}\right) + n^2$

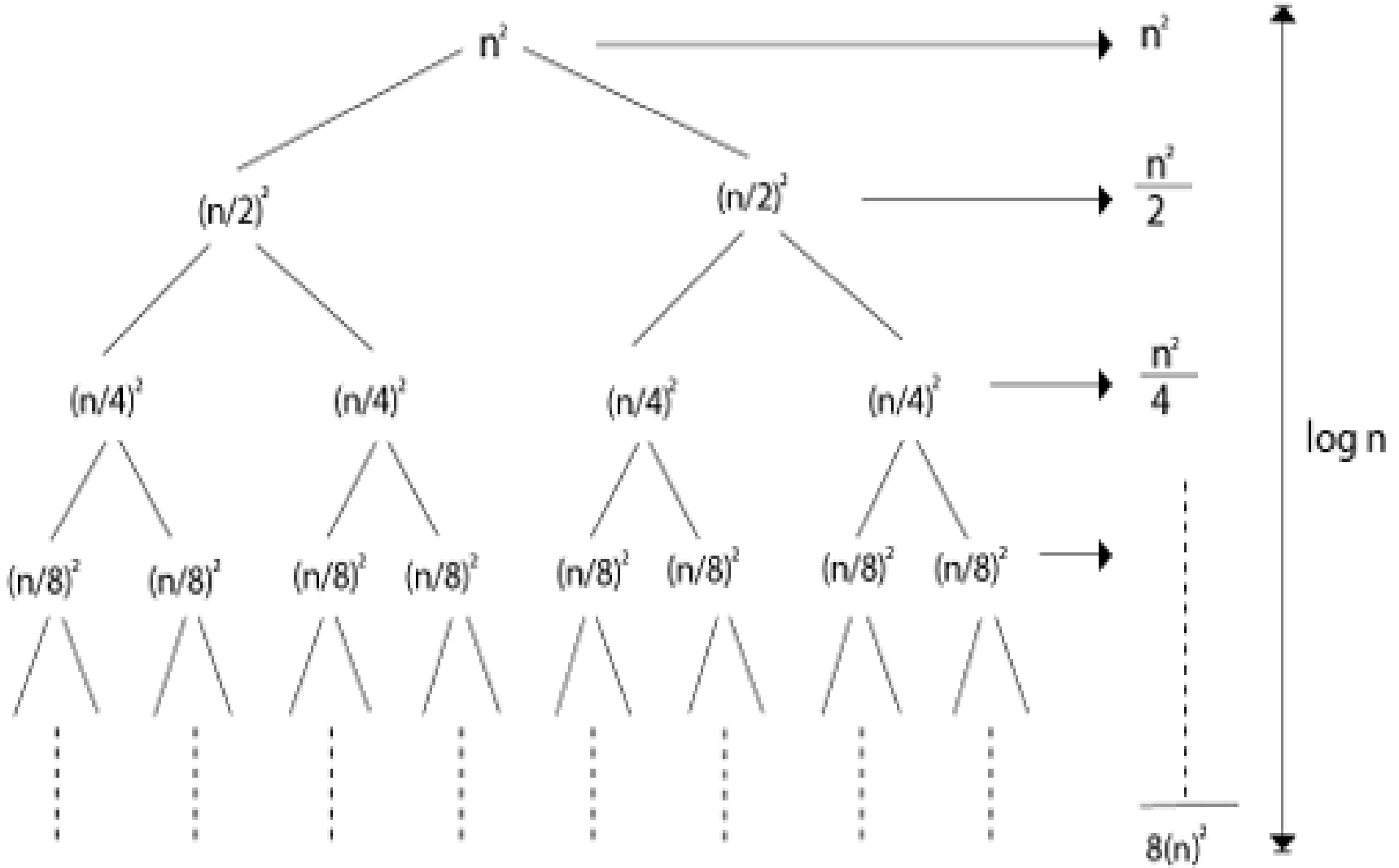
We have to obtain the asymptotic bound using recursion tree method.

Solution: The Recursion tree for the above recurrence is





Continue….


n²
(n/2)²
(n/2)²
(n/4)²
(n/4)²
(n/4)²
(n/4)²
(n/8)²
(n/8)²
(n/8)²
(n/8)²
(n/8)²
(n/8)²
(n/8)²
(n/8)²
n²
n²/2
n²/4
8(n)³
log n

- $T(n)=n^2+(n^2/2)+(n^2/4)+(n^2/8)+\ldots..$ Log n times
- $T(n)=n^2(1+(1/2)+(1/4)+(1/8)+\ldots..$ Log n times
- $T(n)=n^2(1/2^0+(1/2^1)+(1/2^2)+(1/2^3)+\ldots..$ Log n times

$$\leq n^2 \sum_{i=0}^{\infty} \left(\frac{1}{2^i}\right)$$

$$\leq n^2 \left(\frac{1}{1-\frac{1}{2}}\right) \leq 2n^2$$

$$\mathbf{T\ (n) = \theta n^2}$$

- The number of terms is $\log n$ (base 2, assumed unless otherwise stated).

The sum of the first m terms of a geometric progression is:

$$S_m = a\frac{r^m - 1}{r - 1}.$$

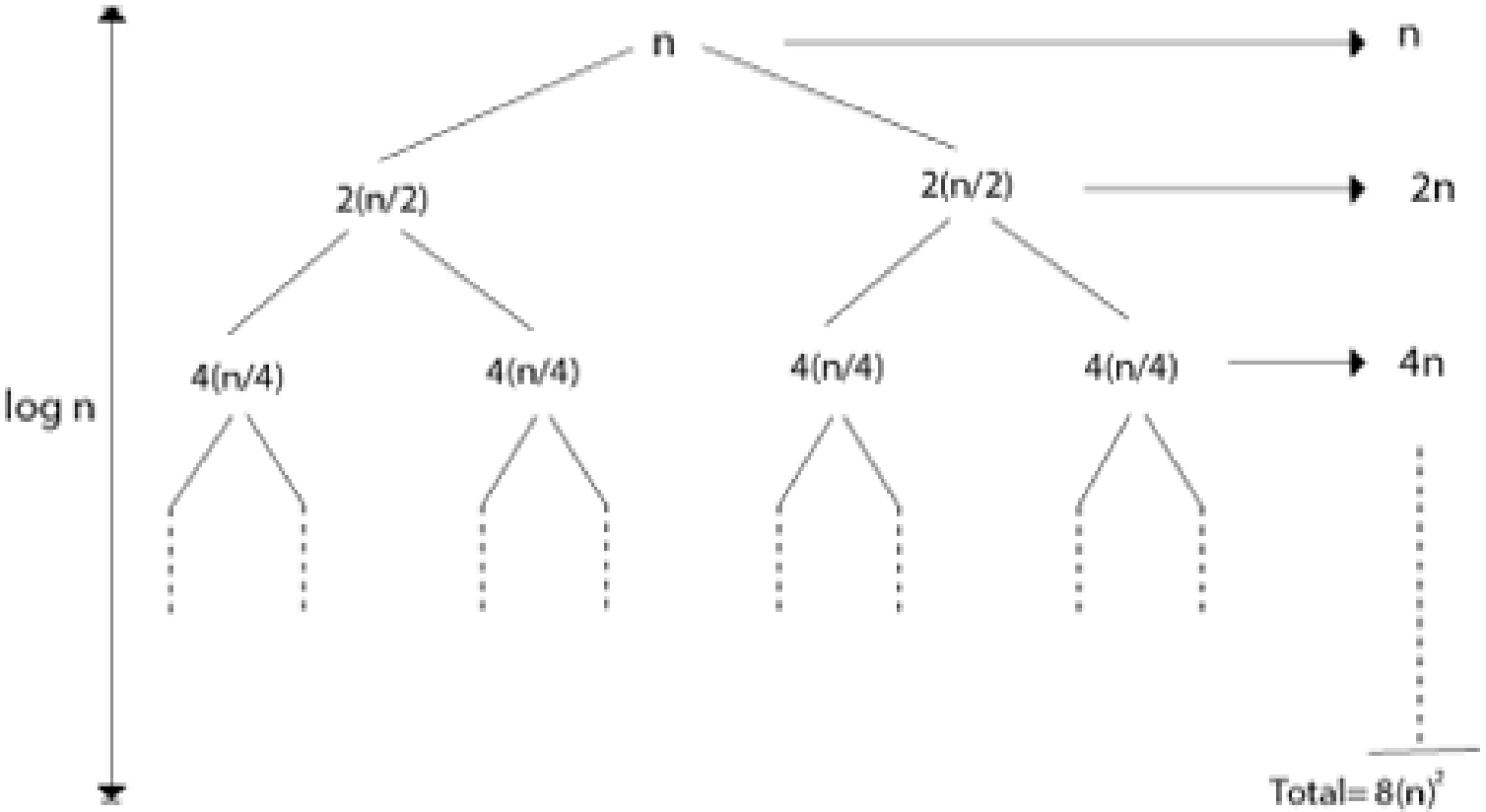
Example

Example 2: Consider the following recurrence

$$T(n) = 4T\left(\frac{n}{2}\right) + n$$

Obtain the asymptotic bound using recursion tree method.

Solution: The recursion trees for the above recurrence



$A^{\log_c b} = b^{\log_c A}$

$2^{\log_2 n} = n^{\log_2 2}$

We have $n + 2n + 4n + \ldots\ldots \log_2 n$ times

$= n\,(1 + 2 + 4 + \ldots\ldots \log_2 n \text{ times})$

$$= n\frac{(2^{\log_2 n} - 1)}{(2-1)} = \frac{n(n-1)}{1} = n^2 - n = \theta(n^2)$$

$$T(n) = \theta(n^2)$$

- The first term $a = 1$,
- The common ratio $r = 2$,
- The number of terms is $\log n$ (base 2, assumed unless otherwise stated).

The sum of the first m terms of a geometric progression is:

$$S_m = a\frac{r^m - 1}{r - 1}.$$

For this series, $a = 1$, $r = 2$, and $m = \log n$. Substituting these values:

Tower of Hanoi

- In this puzzle, we have n disks of different sizes that can slide onto any of three pegs.
- Initially, all the disks are on the first peg in order of size, the largest on the bottom and the smallest on top.
- The goal is to move all the disks to the third peg, using the second one as an auxiliary, if necessary.
- We can move only one disk at a time, and it is forbidden to place a larger disk on top of a smaller one

Tower of Hanoi

B A E

```
void ToH(int n, char B, char E,char A){
  if (n == 0) { return; }
  ToH(n-1, B, E, A);
  S.O.P("Move disk " +n+”From”+B+”To”+E);
  ToH(n-1,A, B, E);
}
```

T(n)=T(n-1)+T(n-1)+c for n>1 and T(1)=1

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

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

$T(n-1) \rightarrow 2T(n-2)+1$, then substitute

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

$=4T(n-2)+2+1$

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

$T(n-2) \rightarrow 2T(n-3)+1$, then substitute

$T(n)=2^2[2T(n-3)+1]+2+1$

$=2^3T(n-3)+2^2+2+1$

after i substitutions, we get

$T(n)=2^iT(n-i)+2^{i-1}+2^{i-2}+2^{i-3}+2^{i-4}+\ldots\ldots+2+1$

- $T(n)=2^iT(n-i)+2^{i-1}+2^{i-2}+2^{i-3}+2^{i-4}+\ldots\ldots+2+1$
- $= 2^iT(n-i)+2^i-1$
- after k-1, where k=n substitutions, we get
- $T(n)=2^{k-1}T(n-(k-1)+2^{k-1}-1$
- $T(n)= 2^{n-1}T(n-(n-1)+2^{n-1}-1$
- $T(n)= 2^{n-1}T(1)+2^{n-1}-1$
- $T(n)= 2^n-1$

Example: $2^3+2^2+2^1+2^0 = 2^4-1$

- The **Master Method** is a commonly used approach for analyzing the time complexity of divide-and-conquer algorithms.
- It applies to recurrence relations of the form:

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

Where:

a ≥1: The number of subproblems.

b >1: The factor by which the problem size is divided in each recursive step.

f(n) : The cost of the work done outside the recursive calls (e.g., merging, dividing).

Master method

- Master's theorem can only be applied on decreasing and dividing recurring functions. If the relation is not decreasing or dividing, master's theorem must not be applied.

Master's Theorem for Dividing Functions

- **T(n) = aT(n/b) + f(n)**

where, **a >= 1** and **b >= 1**,

n – size of the problem

a – number of sub-problems in the recursion

n/b – size of the sub problems based on the assumption that all sub-problems are of the same size.

f(n) – represents the cost of work done outside the recursion

If the recurrence relation is in the above given form, then there are three cases in the master theorem to determine the asymptotic notations –

Case 1

- $T(n)=aT(n/b)+f(n)$

- The Master Theorem provides three cases to determine the asymptotic bounds of (T(n)):

CASE 1: if $f(n)=O(n^{\log_b a-E})$ for some constant where $E>0$, then $\mathbf{T(n)= \Theta (n^{\log_b a})}$

If $f(n)$ grows slower than $n^{\log_b a}$, the total time complexity is determined by the recursive part:

$$T(n) = \Theta\left(n^{\log_b a}\right)$$

Case 1: Example 1: Binary Sear

$$T(n) = T\left(\frac{n}{2}\right) + O(1)$$

Here:

- $a = 1, b = 2$, and $f(n) = O(1)$.
- $\log_b a = \log_2 1 = 0$.

$f(n) = O(n^{\log_2 1 - \epsilon})$ for $\epsilon > 0$, this is **Case 1**, and the complexity is:

T(n)= Θ ($n^{\log_b a}$) $T(n) = \Theta(\log n)$

Case 1: Example 2

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

As given $T(n)=aT(n/b)+f(n)$

Here we get a=4, b=2 and f(n)=n

Check for CASE1

$f(n)=O(n^{\log_b a - E})$

➔ $O(n^{\log_2 4})$

➔ $O(n^{\log_2 2^2})$

➔ $O(n^{2\log_2 2})$

➔ $O(n^2)$

$f(n)=n$, so

$n=O(n^{2-E})$, Here $E>0$, so it satisfy case 1

$T(n)= \Theta (n^{\log_b a}) = \Theta (n^{\log_2 4}) = O(n^{2\log_2 2})$

$T(n)= \Theta (n^2)$

Case 1: Example 3

$$T(n) = 7T\left(\frac{n}{2}\right) + O(n^2)$$

Here:

- $a = 7, b = 2$, and $f(n) = O(n^2)$.
- $\log_b a = \log_2 7 \approx 2.81$.

$f(n) = O(n^{\log_2 7 - \epsilon})$ for $\epsilon > 0$, this is **Case 1**, and the complexity is:

$$T(n) = \Theta(n^{\log_2 7})$$

- $T(n)=aT(n/b)+f(n)$

- **CASE 2:** if $f(n)= \Theta (n^{\log_b a})$ then
  - **$T(n)= \Theta (n^{\log_b a} * \log_b n)$**

If $f(n)$ grows at the same rate as $n^{\log_b a}$, the total time complexity includes the contribution from $f(n)$:

$$T(n) = \Theta\left(n^{\log_b a} \log n\right)$$

Case 2: Example 1: Merge Sort

$$T(n) = 2T\left(\frac{n}{2}\right) + O(n)$$

Here:

- $a = 2, b = 2$, and $f(n) = O(n)$.
- $\log_b a = \log_2 2 = 1$.

$f(n) = O(n^{\log_2 2})$, this is **Case 2**, and the complexity is:

$$T(n) = \Theta(n \log n)$$

Case 2: Example 2

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

As given $T(n)=aT(n/b)+f(n)$

Here, **$a=4$, $b=2$ and $f(n)=n^2$**

Check for CASE1

$f(n)=O(n^{\log_b a-E})$

➔$O(n^{\log_2 4})$

➔ $O(n^{\log_2 2^2})$

➔ $O(n^{2\log_2 2})$

➔$O(n^2)$

$n^2= O(n^{2-E})$, so it is not satisfy case 1

Check for CASE 2:

$n^2= O(n^2)$, so it is satisfy case 2

$T(n)= \Theta (n^{\log_b a} * \log n)$

$T(n)= \Theta (n^2 * \log n)$

- T(n)=aT(n/b)+f(n)

CASE 3: if $f(n) = \Omega(n^{\log_b a+E})$ then for some constant and if $f(n/b) <= Cf(n)$ for some constant $C<1$ and all sufficiently large n, then

- $T(n) = \Theta(f(n))$

If $f(n)$ grows faster than $n^{\log_b a}$

Case 3: Example 1

$$T(n) = 2T\left(\frac{n}{2}\right) + n^3$$

Here:

- $a = 2, b = 2$, and $f(n) = n^3$.
- $n^{\log_b a} = n^{\log_2 2} = n^1$.

$$f(n) = n^3 \in \Omega(n^{1+\epsilon}) \text{ for } \epsilon = 2.$$

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

Example

- $T(n)=7T(n/2)+n^2$
- As given $T(n)=aT(n/b)+f(n)$
- Here $a=7$, $b=2$ and $f(n)=n^2$
- Check for CASE1
  - $f(n)=O(n^{\log_b a-E})$
  - ➔$O(n^{\log_2 7})$➔Here $\log_2 7=2.81$
  - ➔$O(n^{2.81})$
  - $n^2= O(n^{2.81-E})$, so it is satisfy case 1

  $T(n)= \mathbf{\Theta\ (n^{\log_b a}\ )}$
  - **$T(n)= \Theta\ (n^{2.81}\ )$**

Example

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

As given $T(n)=aT(n/b)+f(n)$

Here a=8, b=2 and $f(n)=n^2$

Check for CASE1

$f(n)=O(n^{\log_b a-E})$

➔$O(n^{\log_2 8})$

➔ $O(n^{\log_2 2^3})$

➔ $O(n^{3\log_2 2})$

➔$O(n^3)$

$n^2= O(n^{3-E})$, so it is satisfy case 1

$T(n)= \Theta (n^{\log_b a})$

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

Examples

$T(n)=aT(n/b)+f(n)$	$f(n)$	$n^{\log_b a}$
$T(n)=2T(n/2)+sqrt(n)$ $T(n) = \mathbf{\Theta(n)}$	$Sqrt(n)$	$n^{\log_2 2} = n$
$T(n)=T(n/2)+sqrt(n)$ $T(n) = \mathbf{\Theta(sqrt(n))}$	$Sqrt(n)$	$n^{\log_2 1} = n^0$
$T(n)=8T(n/4)+n^4$ $T(n)=\mathbf{\Theta(n^4)}$	n^4	$n^{\log_4 8} = n^3$
$T(n)=8T(n/2)+n^2$ $T(n)=\mathbf{\Theta(n^3)}$	n^2	$n^{\log_2 8} = n^3$

End