

# Presentation Topic

## Design and Analysis of Algorithm

(AY 2022 – 23)

Department of Computer Engineering



**BRACT'S, Vishwakarma Institute of Information Technology, Pune-48**

(An Autonomous Institute affiliated to Savitribai Phule Pune University)  
Institute Accredited 'A' Grade by NAAC

# **CSUA32201: Design and Analysis of Algorithms**

TY-SEM-I

# Design and Analysis of Algorithms

- Design and Analysis of Algorithms subject is fundamental in computer science education as it provides students with problem-solving skills
- the ability to develop efficient software, and the necessary foundation for further studies and research

- **Algorithm Efficiency:** Efficient algorithms are at the core of creating high-performance software systems.
- Enables students to develop solutions that can handle large datasets, optimize resource utilization, and provide faster response times.
- **Problem-Solving Skills:** Learning algorithm design cultivates problem-solving skills
- **Algorithm Selection:** based on factors such as time complexity, space complexity, and input size.
- **Software Optimization:** Understanding algorithm analysis helps students identify and improve bottlenecks in software

- **Algorithm Paradigms:** The subject introduces students to various algorithmic paradigms, such as divide and conquer, greedy algorithms, dynamic programming, etc. Each paradigm provides a unique approach to problem-solving

- **Efficiency and Scalability:** AI and data science applications often deal with massive datasets and complex computations
- **Model Selection:** Algorithm analysis helps in comparing different models, understanding their strengths and weaknesses, and selecting the most appropriate one
- **Optimization:** Understanding the time complexity of optimization algorithms is essential for selecting efficient optimization techniques.

- Deep Learning: In the context of deep learning, where neural networks are trained on large datasets, algorithm design and optimization are essential for improving training efficiency
- Real-time and Interactive Systems: Some AI applications, such as real-time decision-making systems or chatbots, require algorithms that can process and respond to data quickly.
- Algorithm analysis helps ensure the responsiveness of such system

- Recommendation Systems: Understanding their efficiency is crucial for real-time recommendations



# **CSUA32201: Design and Analysis of Algorithms**

<b>Teaching Scheme</b>		<b>Examination Scheme</b>	
Credits: 4		Continuous Evaluation(CE): 20 Marks	
Lectures: 3 Hrs/week		In-Semester Examination(ISE): 30 Marks	
Practical : 2 Hrs/week		Skills & Competency Exam(SCE): 20 Marks	
		End Semester Examination(ESE): 30 Marks	
		PR/OR: 25 Marks	

<b>Prerequisites :</b>	
	Discrete Mathematics
	Data Structures
	Theory of Computation
<b>Course Objectives :</b>	
	To study the <b>analysis</b> of algorithms
	To study the <b>greedy and dynamic programming</b> algorithmic strategies
	To study the <b>backtracking and branch and bound algorithmic</b> strategies
	To study the concept of <b>hard problems</b> through understanding of <b>intractability and NP-Completeness</b>
	To study some <b>advance techniques to solve intractable problems</b>
	To study <b>multithreaded and distributed algorithms</b>

<b>Course Outcomes:</b>	
	After completion of the course, student will be able to
<b>1.</b>	Analyze algorithms for their time and space complexities in terms of asymptotic performance.
<b>2.</b>	Apply greedy and dynamic programming algorithmic strategies to solve a given problem
<b>3.</b>	Apply backtracking and branch and bound algorithmic strategies to solve a given problem
<b>4</b>	Identify intractable problems using concept of NP-Completeness
<b>5</b>	Use advance algorithms to solve intractable problems
<b>6</b>	Solve problems in parallel and distributed scenarios

## Unit I Introduction

Analysis of Algorithms, Best, Average and Worst case running times of algorithms, Mathematical notations for running times  $O$ ,  $\Omega$ ,  $\Theta$ , Master's Theorem

Problem solving principles: Classification of problem, problem solving strategies, classification of time complexities (linear, logarithmic etc.)

Divide and Conquer strategy: General strategy, Quick Sort and Merge Sort w.r.t. Complexity

---

## **Unit II**

## **Greedy Method & Dynamic Programming**

Greedy Method: General strategy,  
the principle of optimality,

Knapsack problem, Job Sequencing with Deadlines,  
Huffman coding.

Dynamic Programming: General Strategy, 0/1 Knapsack, OBST,  
multistage graphs

<b>Unit III</b>	<b>Backtracking, Branch and Bound</b>
<p>Backtracking: The General Method 8 Queen's problem, Graph Coloring Branch and Bound: 0/1 Knapsack, Traveling Salesperson Problem.</p>	

<b>Unit IV</b>	<b>Intractable Problems and NP-Completeness</b>
<p>Time-Space trade off, Tractable and Non-tractable Problems, Polynomial and non-polynomial problems, deterministic and non-deterministic algorithms P-class problems, NP-class of problems, Polynomial problem reduction, NP complete problems- Vertex cover and 3-SAT and NP hard problem - Hamiltonian cycle</p>	

## **Unit V   Approximation and Randomized Algorithms, Natural Algorithms**

Approximation algorithms, Solving TSP by approximation algorithm, approximating Max Clique Concept of randomized algorithms, randomized quicksort algorithms , Natural Algorithms–Evolutionary Computing and Evolutionary Algorithms, Introduction to Genetic Algorithm, Simulated Annealing



## **Unit VI Parallel and Concurrent Algorithms**

Parallel Algorithms: Sequential and parallel computing, RAM&PRAM models, Amdahl's Law, Brent's theorem, parallel algorithm analysis, multithreaded matrix multiplication, Concurrent Algorithms: Dining philosophers problem

<b>Text Books :</b>
Gilles Brassard, Paul Bratley, “Fundamentals of Algorithmics”, PHI, ISBN 978-81-203- 1131-2
Horowitz and Sahani, "Fundamentals of Computer Algorithms", University Press, ISBN: 978 81 7371 6126, 81 7371 61262
<b>Reference Books :</b>
Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein, “Introduction to Algorithms”, MIT Press; ISBN 978-0-262-03384-8
Parag Himanshu Dave, Himanshu Bhalchandra Dave, “Design And Analysis of Algorithms”, Pearson Education, ISBN 81-7758-595-9
Rajeev Motwani and Prabhakar Raghavan, “Randomized Algorithms”, Cambridge University Press, ISBN: 978-0-521-61390-3
Michael T. Goodrich, Roberto Tamassia , “Algorithm Design: Foundations, Analysis and Internet Examples”, Wiley, ISBN 978-81-265-0986-7
Dan Gusfield, “Algorithms on Strings, Trees and Sequences”, Cambridge University Press,ISBN:0-521- 7035-7

# Perform the following lab assignments using C++/Java/Python

1. Implement Quick Sort using divide and conquer strategy.
2. Implement 0/1 knapsack using Dynamic Programming.
3. Implement 8 queens problem using Backtracking
4. Implement Travelling Salesman problem using branch and bound technique.
5. Implement Travelling Salesman problem using Genetic Algorithm
6. Implement Concurrent Dining Philosopher Problem.
7. Implement multithreaded matrix multiplication.

# Objectives & Outcomes of Unit 1

- **Objective :**

Measure the performance of algorithms on the basis of time and space complexity.

- **Outcome :**

Analyze algorithms for their time and space complexities in terms of asymptotic performance

# Unit I: Introduction

# What is - Design & Analysis of Algorithm

- An **algorithm** is systematic method containing sequence of Instruction to solve a computational problem.
- It takes inputs, performs a well defined sequence steps and produces output.
- Once we **design** an algorithm we need to know how well it performs on any input.
- In particular we need to know whether there are any better algorithms for a problem.
- Hence we need to **analyze** an algorithm on the basis of **efficiency**.

# What is efficiency ?

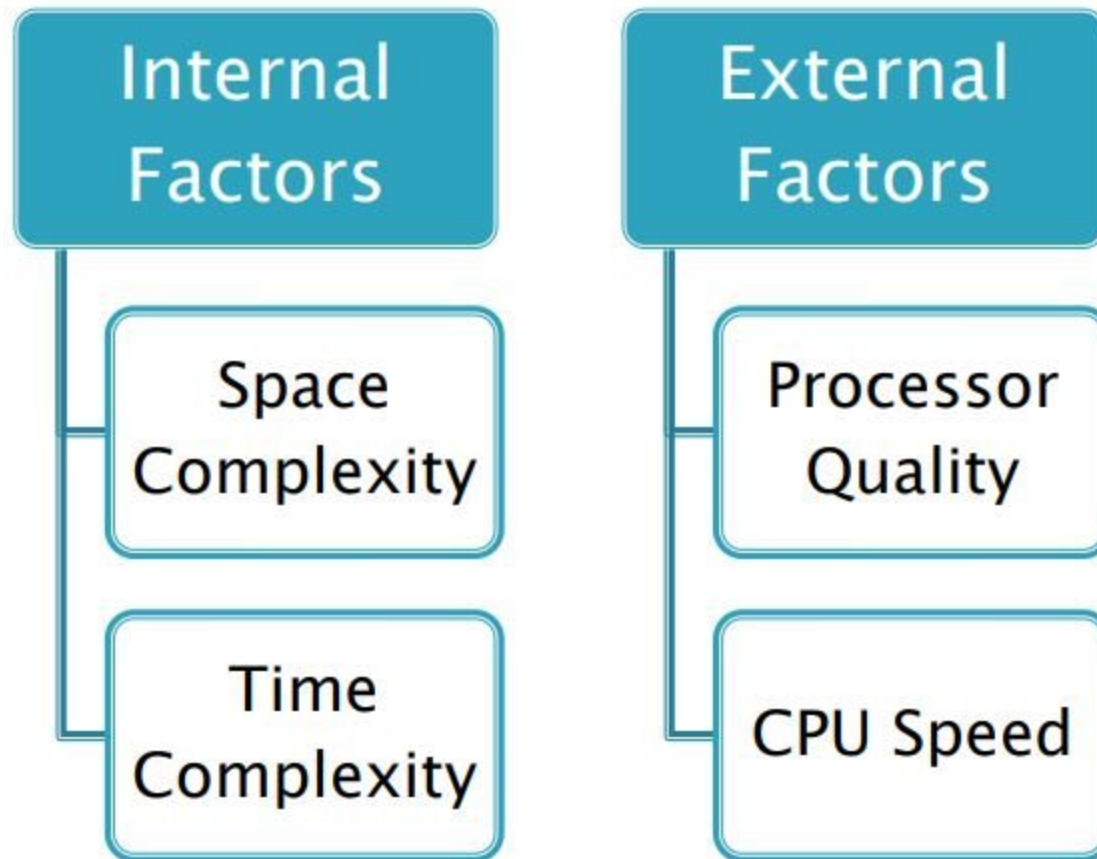
- The rate at which storage memory or time grows as a function of the input size is called efficiency.
- It is a measure to analyze the performance of a program code for a particular problem.

# Why do we need Efficiency?

- Several algorithms to solve the same problem
- Which algorithm is “best” ?
- How much **time** does the algorithm require ?
- How much **space** (memory) does the algorithm occupy ?



# Algorithm Complexity



# Types of Efficiency

- Space Complexity
- Time Complexity
- In general, both depend on the input (typically the input, size of the input).

# Space Complexity

- Space complexity is the amount of memory the program requires till its termination.
- It is a function of the size of the input.

# Why Space Complexity?

- To know in advance space / memory requirement
- To specify the amount of memory if the program is to run on multi-user system
- Extremely important if the program has to run with limited resources or has to handle input of large size

# Time Complexity

- It is the amount of time required to execute the program
- It indicates the relationship between the running time of the algorithm and size of the input

# Why Time Complexity?

- To estimate how long a program will run.
- To help focus on the parts of code that are executed the large number of times.
- To find an alternate solution.

# Why Time Complexity?

- Suppose Algorithm A has running time  $7n^2$ ;  
Algorithm B has running time  $2n^3$ .  
Which algorithm is more efficient?

# Example

- Suppose Algorithm A has running time  $7n^2$ ;  
Algorithm B has running time  $2n^3$ .  
Which algorithm is more efficient?
- Compare  $n^2$  and  $n^3$  for different values of  $n$ :

	$n=10$	$n=100$	$n=10000$	$n=100000$
$n^2$	$10^{-7}$ sec.	$10^{-5}$ sec.	0.1 sec.	10 sec.
$n^3$	$10^{-6}$ sec.	$10^{-3}$ sec.	17 min.	11.6 days

- For large  $n$ ,  *$n^2$  is much faster than  $n^3$* ,  
and the coefficients 7 and 2 doesn't make much difference.
- Thus, to evaluate the efficiency of an algorithm we need to identify the most important part in its running time. That important part is known as **algorithm order**.
- To define it formally, we need to introduce **Asymptotic Notation** .



# Analysis of Algorithms

# Analysis of Algorithm =

- Prediction of how ***fast*** an algorithm runs for a given **PROBLEM SIZE**
  - **Analysis** based on Time Computations = TIME COMPLEXITY
- Prediction of ***memory requirements*** (primary memory) based on the **PROBLEM SIZE**
  - **Analysis of algorithms** based on Memory Requirements = SPACE COMPLEXITY

# 2 Considerations in Analysis

- TIME and SPACE
  - How much TIME is required to Execute the algorithm?  
(*Time Complexity*)
  - How much SPACE is required to Execute the algorithm?  
(*Space Complexity*)

# How to Calculate Time (*fast*)?

Is **absolute time** to run the algorithm possible, because machines running the algorithm may be different, or the environment may be different?

Or something else?

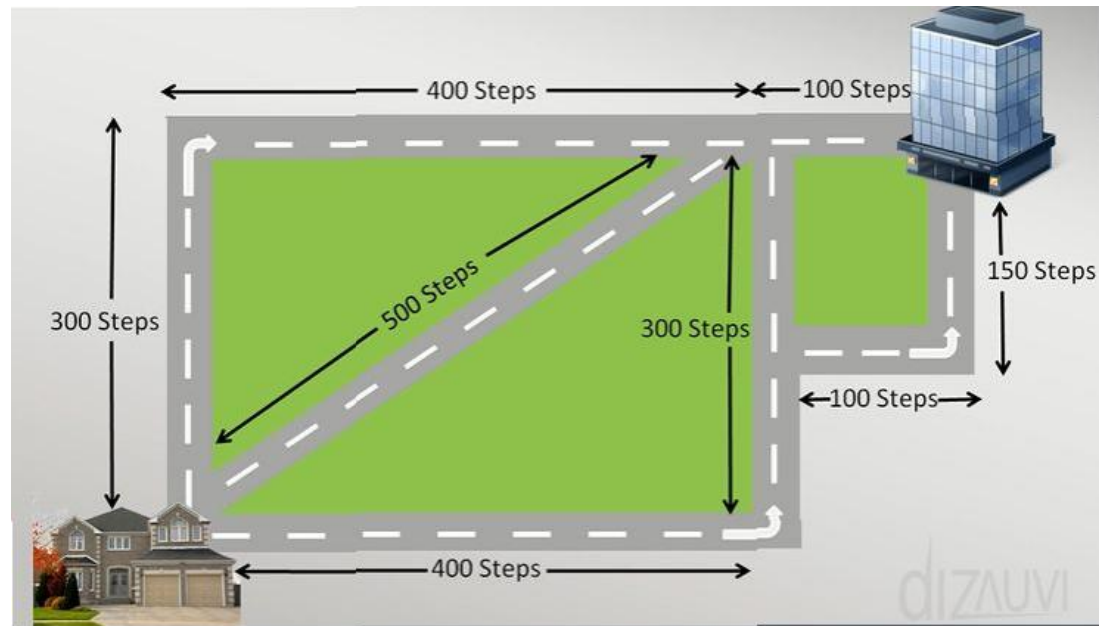


# How to calculate time for going from home to college/office?

Should speed of walking or vehicle type/traffic conditions be considered?

Or something else?

**Ans: Number of steps** ? With assumption that step size is uniform.



# PROBLEM SIZE

Types of problem size:-

- Number of **Inputs / Outputs**
  - E.g. For Sorting Algorithm :
    - Inputs = Total number of elements to be sorted
    - Output = Total number of sorted elements
- Number of **Operations Involved** in the algorithm
  - E.g. For Searching Algorithm :
    - Operations Involved = total number of Comparisons with search element

# Summary

- Analysis of Algorithm aims at determination of Time Complexity and Space Complexity


# Questions

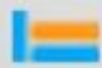
- Define Analysis of Algorithm?
- What is meant by PROBLEM SIZE?




Best, Average and Worst case  
running times of algorithms

# Types of Time Complexity

 Worst Case

 Best Case

 Average Case

## Types of Time Complexity

 Worst Case 

 Best Case

 Average Case 

## Sample – Best, Worst, Average Case - Search for number $x$ in an array

- Best case: First number in array equals  $x$   
Time effort: let's say 1 unit
- Worst case: the last number ( $n$  th entry) in array equals  $x$  or is not found  
Time effort:  $n$  units
- Average case: ?  $(1+2+3+4+---+n)/n$  units =  $n(n+1)/n$  units

# Assumptions to Calculate Time Complexity

## RAM Model of Computation

- ▮ We have infinite memory.
- ▮ Each operation(+, -, \*, /, =) takes unit time.
- ▮ For each memory access, unit time is consumed.
- ▮ Data may be accessed from RAM or disk, it is assumed that the data is in the RAM.

# Step Count for Time Complexity

# Step Count for Time Complexity

- *The step count method is one of the methods to analyze the Time complexity of an algorithm.*
- *This method counts the number of times each instruction is executed.*
- *Based on that Time Complexity will be calculated*

# Step Count for Time Complexity

	Frequency	Total
Algorithm sum()		
{		
Sum=0		
for i:= 1 to n		
Sum= Sum + i		
}		
	TOTAL    T(n)=	

The time complexity of Algorithm **Add\_arr**, with array size 'n' =  
 $T(n) =$

# Step Count for Time Complexity

	Frequency	Total
Algorithm <b>Add_arr</b> (A,B,C,n)		
{		
for i:= 1 to n		
for j:=1 to n		
C[i,j]=B[i,j]+A[i,j];		
}		
	TOTAL    T(n)=	

The time complexity of Algorithm **Add\_arr**, with array size 'n' =  
 $T(n) =$



# Step Count for Time Complexity

	Frequency	Total
Algorithm <b>Add_arr</b> (A,B,C,n)	0	-
{	0	-
for i:= 1 to n	n+1	n+1
for j:=1 to n	n(n+1)	n <sup>2</sup> +n
C[i,j]=B[i,j]+A[i,j];	n(n)	n <sup>2</sup>
}		
	TOTAL    T(n)=	2n <sup>2</sup> +2n+1

The time complexity of Algorithm **Add\_arr**, with array size 'n' =

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

# Space Complexity

	Variables	space
Algorithm <b>Add_arr</b> (A,B,C,n)		
{		
for i:= 1 to n		
for j:=1 to n		
C[i,j]=B[i,j]+A[i,j];		
}		

The space complexity of Algorithm **Add\_arr**, with array size 'n' =  
 $T(n) =$

# Space Complexity

	Variables	space
Algorithm <b>Add_arr</b> (A,B,C,n)	i	1
{	j	1
for i:= 1 to n	Array a of size n	$n^2$
for j:=1 to n	Array b of size n	$n^2$
C[i,j]=B[i,j]+A[i,j];	Array c of size n	$n^2$
}	n	1
	TOTAL (n)=	$3n^2+3$

The space complexity of Algorithm **Add\_arr**, with array size 'n' =

$$T(n) = 3n^2 + 3 = \mathbf{O(n^2)}$$

# Step Count for Time Complexity

	Frequency	Total
Algorithm <b>Add_mul</b> (A,B,C,n)	0	-
{	0	-
for i:= 1 to n		
for j:=1 to n		
C[i,j]+=0		
for k:=1 to n		
C[i,j]+=A[i,k]*B[k,j];		
}	TOTAL T(n)=	

The time complexity of Algorithm **Mul\_arr**, with array size 'n' =

# Step Count for Time Complexity

	Frequency	Total
Algorithm <b>Add_mul</b> (A,B,C,n)	0	-
{	0	-
for i:= 1 to n	n+1	n+1
for j:=1 to n	n(n+1)	n <sup>2</sup> +n
C[i,j]+=0	n*n	n <sup>2</sup>
for k:=1 to n	n*n(n+1)	n <sup>2</sup> (n+1)
C[i,j]+=A[i,k]+B[k,j];	n*n*n	n <sup>3</sup>
}	TOTAL T(n)=	2n <sup>3</sup> +3n <sup>2</sup> +2n+1

The time complexity of Algorithm **Mul\_arr**, with array size 'n' =

$$T(n) = 2n^3 + 3n^2 + 2n + 1 = \mathbf{O(n^3)}$$

# Space Complexity

		Total
Algorithm <b>Add_mul</b> (A,B,C,n)		
{		
for i:= 1 to n		
for j:=1 to n		
C[i,j]+=0		
for k:=1 to n		
C[i,j]+=A[i,k]+B[k,j];		
}		

The space complexity of Algorithm **Mul\_arr**, with array size 'n' =  
=

# Space Complexity

		Total
Algorithm <b>Add_mul</b> (A,B,C,n)	0	-
{	0	-
for i:= 1 to n	Array a	$n^2$
for j:=1 to n	Array b	$n^2$
C[i,j]+=0	Array c	$n^2$
for k:=1 to n	i,j,k,n	4
C[i,j]+=A[i,k]+B[k,j];		
}	TOTAL	$3n^2+4$

The space complexity of Algorithm **Mul\_arr**, with array size 'n' =  
 $=3n^2+4= \mathbf{O(n^2)}$

**Time complexities** are categorized based on the rate at which the running time of an algorithm increases with respect to the size of the input 'n'.

**1. Constant Time Complexity ( $O(1)$ ):** the running time remains the same regardless of the size of the input.

- Regardless of how large the input becomes the algorithm takes a constant amount of time to execute.
- Example: To print first element of an array
- Give some other example



- To find first element of the array.
- Accessing a particular value within an array
- Push()
- Pop()

**2. Linear Time Complexity ( $O(n)$ ):** If an algorithm's time complexity is linear, it means that the runtime of the algorithm grows almost linearly with the input size.

Algorithms with linear time complexity have a running time that is directly proportional to the size of the input 'n'. As the input grows, the algorithm's running time grows linearly.

**Example:**

Searching Algorithm-searches for a specific element  $x$  in the array `arr`.

The running time increases linearly with the size of the array, as the function may need to iterate through all elements to find the target element.

## **Quadratic Time Complexity ( $O(n^2)$ ):**

Algorithms with quadratic time complexity have a running time that grows proportionally to the square of the input size 'n'

There are other complexities as well, such as logarithmic time ( $O(\log n)$ ), cubic time ( $O(n^3)$ ), and so on, which represent different growth rates with respect to the input size.

- Sort the following in increasing time complexities :

Linear time –  $O(n)$ , Logarithmic time –  $O(\log n)$ , Constant time –  $O(1)$ , Cubic time –  $O(n^3)$ , Quadratic time –  $O(n^2)$

- Sort the following in increasing time complexities :  
Linear time –  $O(n)$ , Logarithmic time –  $O(\log n)$ , Constant time –  $O(1)$ , Cubic time –  $O(n^3)$ , Quadratic time –  $O(n^2)$

- Ans:

1. Constant time –  $O(1)$
2. Logarithmic time –  $O(\log n)$
3. Linear time –  $O(n)$
4. Quadratic time –  $O(n^2)$
5. Cubic time –  $O(n^3)$

**Q. Which of the following best describes the useful criterion for comparing the efficiency of algorithms?**

1. Time
2. Memory
3. Both of the above
4. None of the above

## **Q. How is time complexity measured?**

1. By counting the number of algorithms in an algorithm.
2. By counting the number of primitive operations performed by the algorithm on a given input size.
3. By counting the size of data input to the algorithm.
4. None of the above

# The Asymptotic Notations – $O, \Theta, \Omega$

Mathematics of Time Complexity



# Big-oh 'O' Notation

- *It specifies the **upper bound of a function**.*
- *It specifies the maximum time required by an algorithm or the worst-case time complexity*
- *Big-O notation represents the upper bound of the running time of an algorithm. hence, it gives the **worst-case complexity of an algorithm**.*

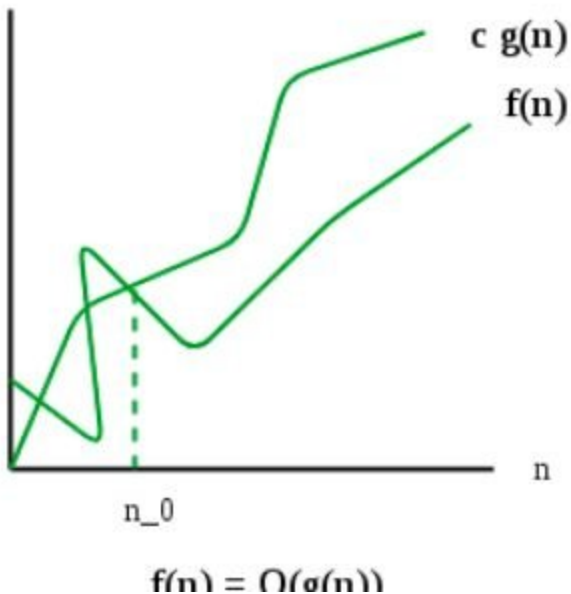
# Big-oh 'O' Notation

## Definition

Let  $f(n)$ , and  $g(n)$  be 2 non negative functions then function  $f(n) = O(g(n))$

if there exist a positive constant  $C$  and  $n_0$  such that,  $0 \leq f(n) \leq Cg(n)$  for all  $n \geq n_0$

**$f(n)$**  defines running time of an algorithm



- $f(n)=3n+2, g(n)=n$
- $f(n) \leq c \cdot g(n)$
- Assume  $C=4$
- For  $n_0=1$
- $3 \cdot 1 + 2 \leq 4 \cdot 1$  false
- $n_0=2$
- $3 \cdot 2 + 2 \leq 4 \cdot 2$  True
- $n_0=3$
- $3 \cdot 3 + 2 \leq 4 \cdot 3$  True
- $f(n) \leq cg(n)$  for all  $n \geq 2$

$$\log n \leq \sqrt{n} \leq n \leq n \log n \leq n^2 \leq n^3 \leq 2^n \leq n!$$

# Big 'O' Notation

- Solve Example
- $f(n)=3n+3 = O(n)$
- 

$$\log n \leq \sqrt{n} \leq n \leq n \log n \leq n^2 \leq n^3 \leq 2^n \leq n!$$

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

$$f(n)=O(n^2)$$

$$f(n)=O(n^3)$$

$$f(n)=O(2^n)$$

$$f(n)=O(\log n)$$

# Asymptotic Notations

- Asymptotic notations are used to describe time and space complexity i.e. to **measure efficiency with respect to problem size**
  - Indicates behavior of a function with respect to the size of the input ( $n$ ).
- Different asymptotic notations used are:-
  1. Big Oh Notation ( $O$ )
  2. Omega Notation ( $\Omega$ )
  3. Theta Notation ( $\Theta$ )

# Big 'Omega' ( $\Omega$ ) Notation

- *It specifies the lower bound of a function.*
- *It specifies the minimum time required by an algorithm or the best-case time complexity*
- *Big-Oh notation represents the lower bound of the running time of an algorithm. hence, it gives the best-case complexity of an algorithm.*

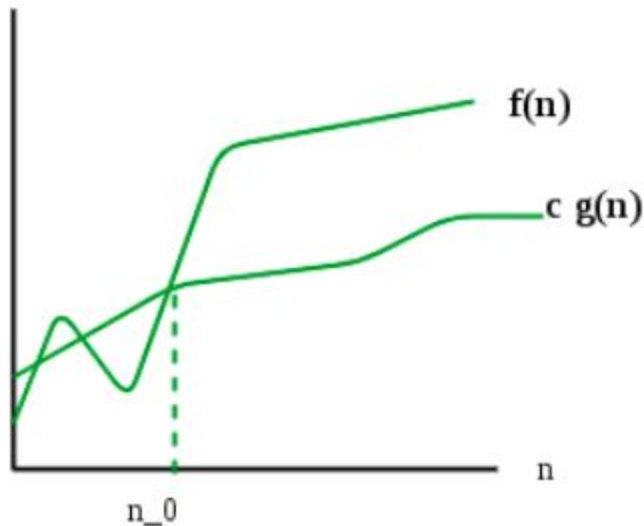
# Big 'Omega' ( $\Omega$ ) Notation

Definition:

Let  $f(n)$ , and  $g(n)$  be 2 non negative functions then  $f(n) = \Omega(g(n))$

if there exist a positive constant  $C$  and  $n_0$  such that,  $f(n) \geq Cg(n)$  for all  $n \geq n_0$

$$\log n \leq \sqrt{n} \leq n \leq n \log n \leq n^2 \leq n^3 \leq 2^n \leq n!$$



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

- $f(n)=3n+2, g(n)=n$
- $f(n) \geq c * g(n)$
- $F(n) \geq c * n$
- Assume  $C=1$
- For  $n_0=1$
- $3*1+2 \geq 1*1$  true
- $n_0=2$
- $3*2+2 \geq 1*2$  True
- $n_0=3$
- $3*3+2 \geq 1*3$  True
- $f(n) \geq c g(n)$  for all  $n \geq 1$

$$\log n \leq \sqrt{n} \leq n \leq n \log n \leq n^2 \leq n^3 \leq 2^n \leq n!$$

$$f(n) = \Omega(\text{square root}(n))$$

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

$$f(n) = \Omega(n^2)$$

$$f(n) = \Omega(2^3)$$



# theta ' $\Theta$ ' Notation

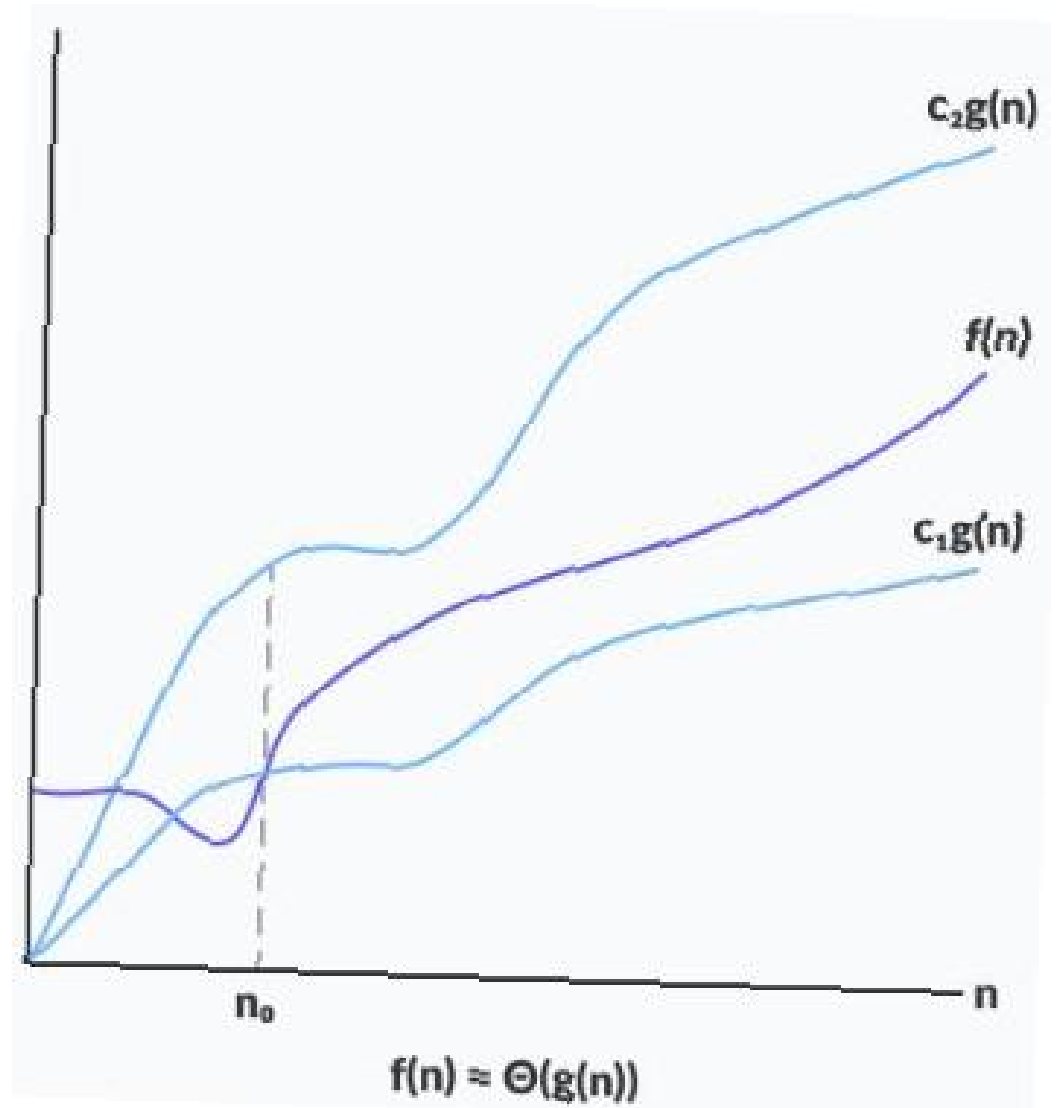
- *It specifies the avg bound of a function.*
- *It specifies the avg time required by an algorithm or the best-case time complexity*
- ***Big-  $\Theta$**  notation represents the avg bound of the running time of an algorithm. hence, it gives the avg-case complexity of an algorithm.*

# Big 'Θ' Notation

- Definition:

- Let  $f(n)$ , and  $g(n)$  be 2 non negative functions then  $f(n) = \Theta(g(n))$

- if there exist 3 positive constant  $C_1$ ,  $C_2$ , and  $n_0$  such that,  $c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$  for all  $n > n_0$



- $c_1 * g(n) \leq f(n) \leq c_2 * g(n)$  for all  $n > n_0$
- $c_1 * n \leq 3n + 2 \leq c_2 * n$
- $C_1 = 1, C_2 = 4$
- For  $n_0 = 1$
- $1 * 1 \leq 3 * 1 + 2 \leq 4 * 1$  false
- $n_0 = 2$
- $1 * 2 \leq 3 * 2 + 2 \leq 4 * 2$  True
- $n_0 = 3$
- $1 * 3 \leq 3 * 3 + 2 \leq 4 * 3$  True
- $c_1 * g(n) \leq f(n) \leq c_2 * g(n)$  for all  $n > n_0$
- for all  $n \geq 2$ , when  $C_1 = 1$ , and  $C_2 = 4$

# Asymptotic Analysis for Algorithms

- The asymptotic behavior of a function  $f(n)$  refers to the growth of  $f(n)$  as  $n$  gets large.
- We typically ignore small values of  $n$ , since we are usually interested in estimating how slow the program will be on large inputs.

# Big 'Oh' notation ( $O$ ) for TC

- **Big 'Oh'** *notation represents the upper bound of the running time of an algorithm, and it is used for analyzing the **best-case** complexity of an algorithm.*

# Big Omega notation ( $\Omega$ ) for TC

- **Omega notation** represents the lower bound of the running time of an algorithm, and it is used for analyzing the **best-case** complexity of an algorithm.
  - For any value of  $n$ , the minimum time required by the algorithm is given by Omega  $\Omega(g(n))$

# Theta notation ( $\Theta$ ) for TC

- *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 gives a tight bound of the time complexity of an algorithm*
- *It is used for analyzing the **average-case** complexity of an algorithm.*

# Increasing Time Complexity Functions

$\log n \leq \sqrt{n} \leq n \leq n \log n \leq n^2 \leq n^3 \leq 2^n \leq n!$

Name	Time Complexity
Constant Time	$O(1)$
Logarithmic Time	$O(\log n)$
Linear Time	$O(n)$
Quasilinear Time	$O(n \log n)$
Quadratic Time	$O(n^2)$
Exponential Time	$O(2^n)$
Factorial Time	$O(n!)$



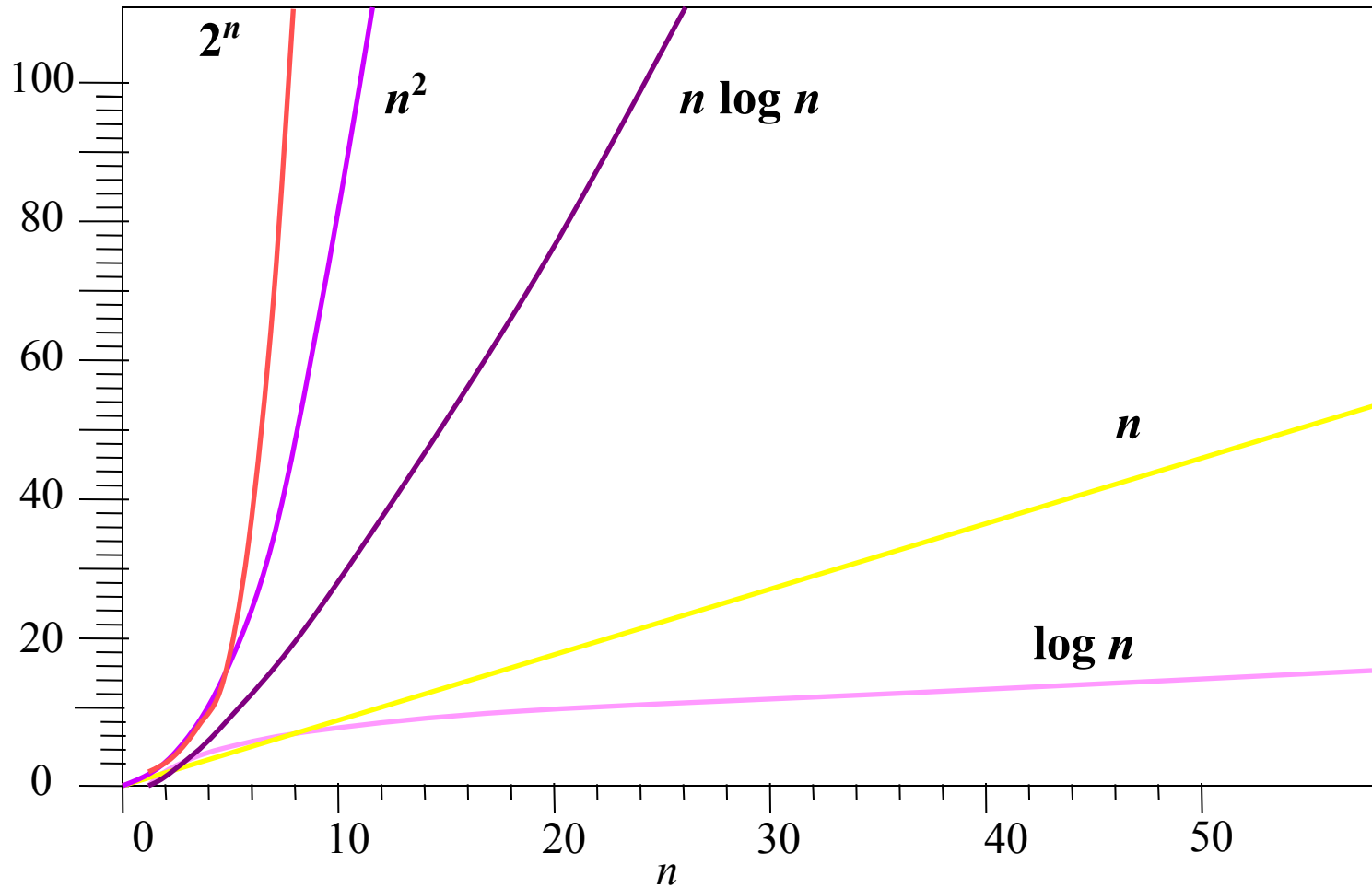
# Time comparisons of the common algorithm orders

<b>f(n)</b>	<b>n=10</b>	<b>n=1000</b>	<b>n=10<sup>5</sup></b>	<b>n=10<sup>7</sup></b>
<b>log<sub>2</sub>n</b>	3.3 × 10 <sup>-9</sup> sec.	10 <sup>-8</sup> sec.	1.7 × 10 <sup>-8</sup> sec.	2.3 × 10 <sup>-8</sup> sec.
<b>n</b>	10 <sup>-8</sup> sec.	10 <sup>-6</sup> sec.	10 <sup>-4</sup> sec.	0.01 sec.
<b>n · log<sub>2</sub>n</b>	3.3 × 10 <sup>-8</sup> sec.	10 <sup>-5</sup> sec.	0.0017 sec.	0.23 sec.
<b>n<sup>2</sup></b>	10 <sup>-7</sup> sec.	10 <sup>-3</sup> sec.	10 sec.	27.8 min.
<b>n<sup>3</sup></b>	10 <sup>-6</sup> sec.	1 sec.	11.6 min.	317 cent.
<b>2<sup>n</sup></b>	10 <sup>-6</sup> sec.	3.4 × 10 <sup>284</sup> years	3.2 × 10 <sup>30095</sup> years	3.1 × 10 <sup>3001022</sup> years

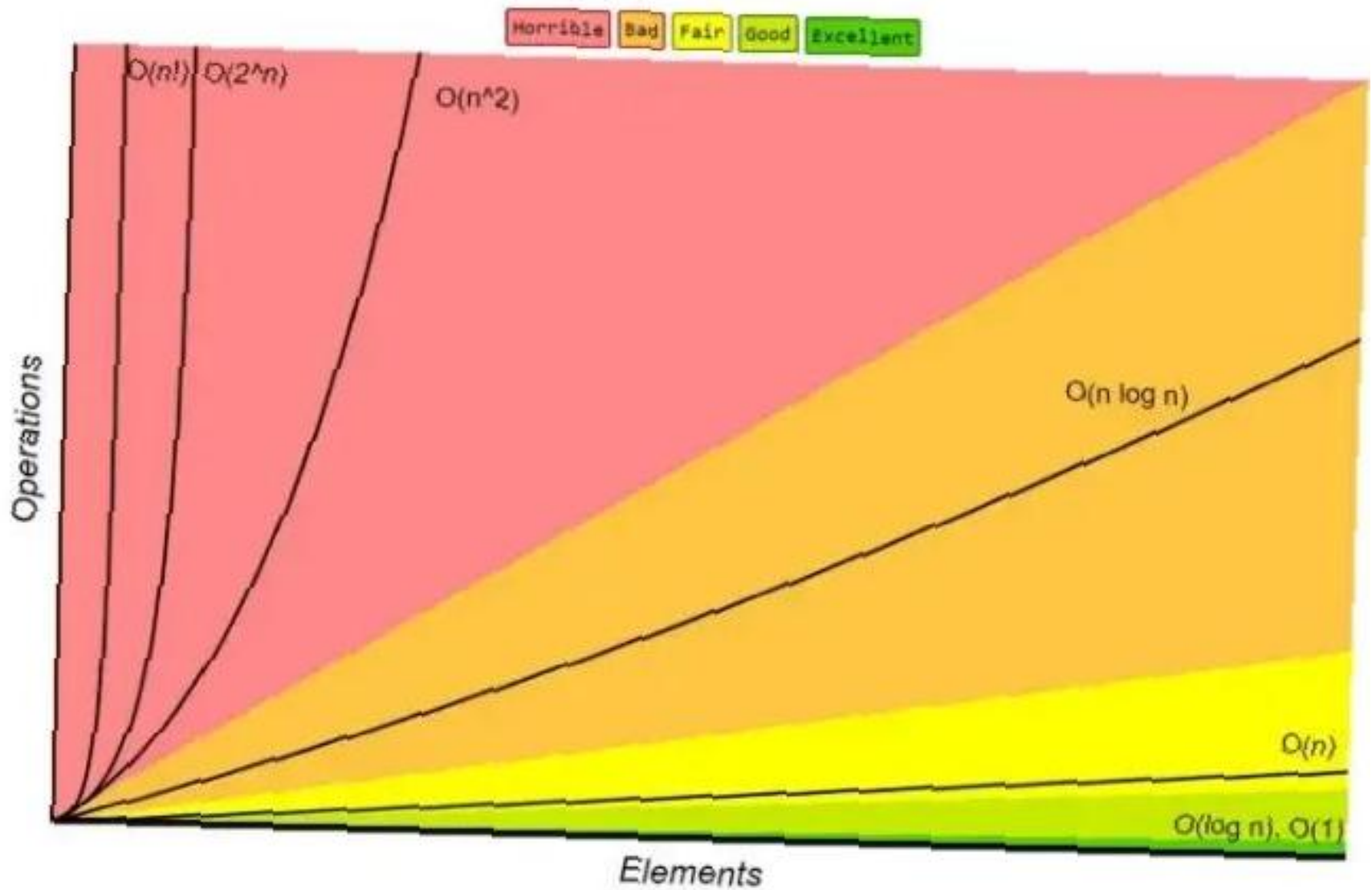
# Classification of Time Complexity

Notation	Complexity	Description	Example
$O(1)$	Constant	Simple statement	Addition
$O(\log(n))$	Logarithmic	Divide in half	Binary search
$O(n)$	Linear	loop	Linear search
$O(n \cdot \log(n))$	Linearithmic	Divide & Conquer	Merge sort
$O(n^2)$	Quadratic	Double loop	Check all pairs
$O(n^3)$	Cubic	Triple loop	Check all triples
$O(2^n)$	Exponential	Exhaustive search	Check all subsets
$O(n!)$	Factorial	Recursive function	Factorial

# Graphical Comparison

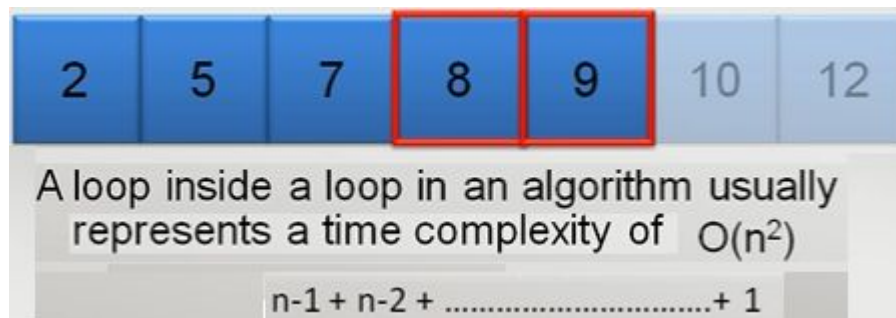
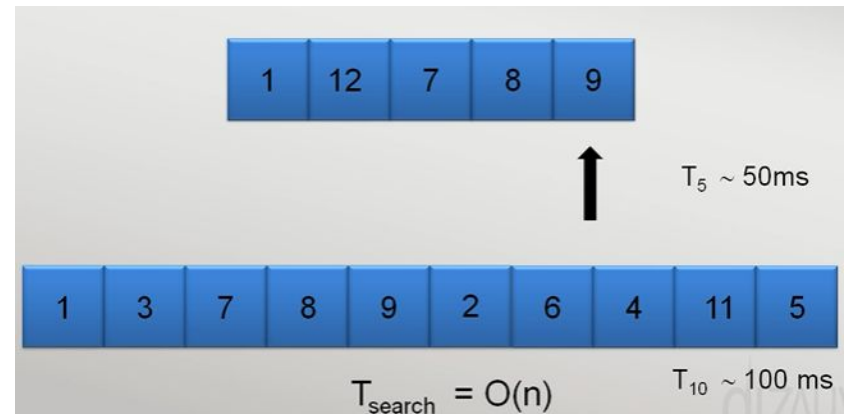
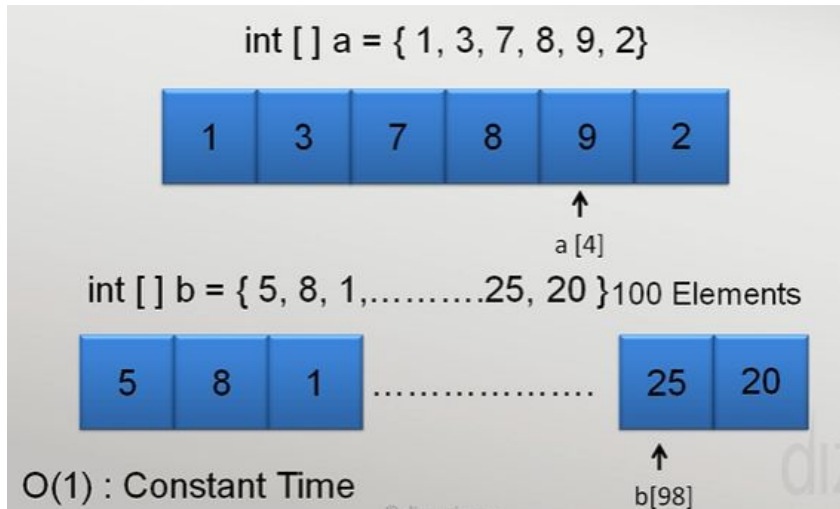


# Big 'O' Complexity



Big-O Complexity Chart: <http://bigocheatsheet.com/>

# Meaning of $O(1)$ , $O(n)$ , $O(n^2)$



# Recap Questions

- What are the 3 kinds of time complexities?
- Define Big 'Oh'
- Sort the following in increasing time complexities :

Linear time –  $O(n)$ , Logarithmic time –  $O(\log n)$ , Constant time –  $O(1)$ , Cubic time –  $O(n^3)$ , Quadratic time –  $O(n^2)$

- What are the 3 kinds of time complexities?
- Best case, Average case, Worst case.

## Q) Select the correct statement

- A) Asymptotic notations give correct, meaningful and exact running times of an algorithm
- B) Asymptotic notations give correct, meaningful but in-exact running times of an algorithm



Q)

Normally we are not interested in the best case running times of algorithms, because this might happen only rarely and generally is too optimistic for a fair characterization of the algorithm's running time

T/F?

Q)

Normally we are not interested in the best case running times of algorithms, because this might happen only rarely and generally is too optimistic for a fair characterization of the algorithm's running time

**Ans: True**

Q)

- What is the need for worst case time complexity?

- The reason we prioritize the worst-case scenario is to ensure that the **algorithm performs efficiently even in the most challenging and demanding situations.**
- It gives us a more accurate and conservative estimate of how the algorithm will behave **when dealing with real-world data and inputs.**
- **Best-case running times can be misleading because they might only occur for very specific** and uncommon input patterns, which do not represent the average or typical case scenarios.
- **Optimizing an algorithm solely based on its best-case performance might lead to poor overall performance** in practical situations.
- Therefore, by analyzing and understanding the worst-case time complexity of an algorithm, we can make better decisions about its suitability for specific tasks and choose the most appropriate algorithm for a given problem.

- the worst-case time complexity is essential for making informed decisions, ensuring algorithm **performance guarantees, resource planning, and providing safety and reliability** in software systems and **critical applications**.
- It enables us to understand how an algorithm behaves under the most adverse conditions, leading to more robust and efficient solutions.

This is especially important for **real-time applications**, such as for the computers that monitor an air traffic control system.

Here, it would not be acceptable to use an algorithm that can handle airplanes quickly enough *most of the time*, but which fails to perform quickly enough when all airplanes are coming from the same direction

Q)

The advantage of - Analyzing the worst case running times of an algorithm - is that you know **for certain** that the algorithm must perform at least that well.

- $\log$
- $F(n)=3n+3=O(n)$
- $F(n)=3n+3=\Omega(n)$

- $\log$
- $3n^2+2 = O(n)$
- $\log$
- $10n+2 = O(1)$

- $\Theta$
- $10n+2 = \Omega(n^2)$
- $\Theta$
- $3n^2+2 = \Omega(n^4)$



# Unit-1

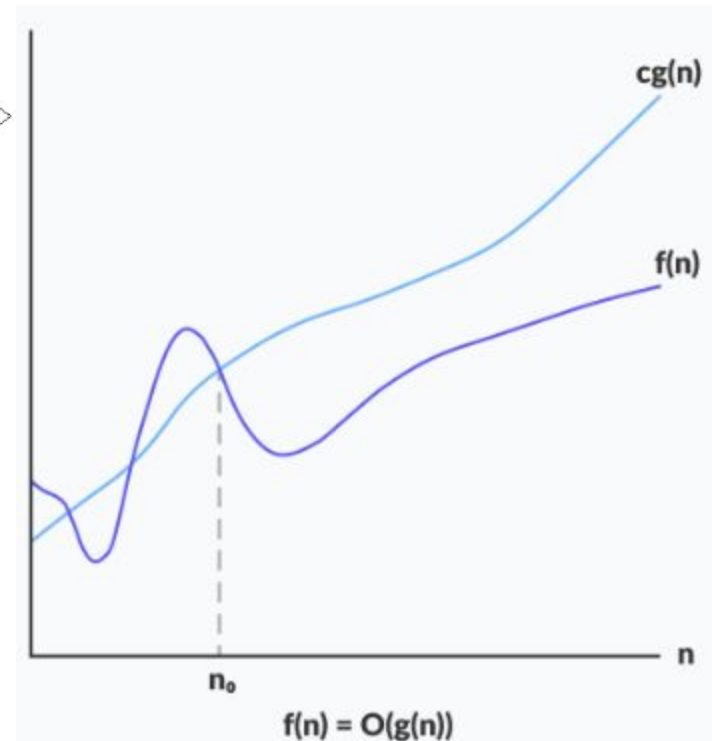
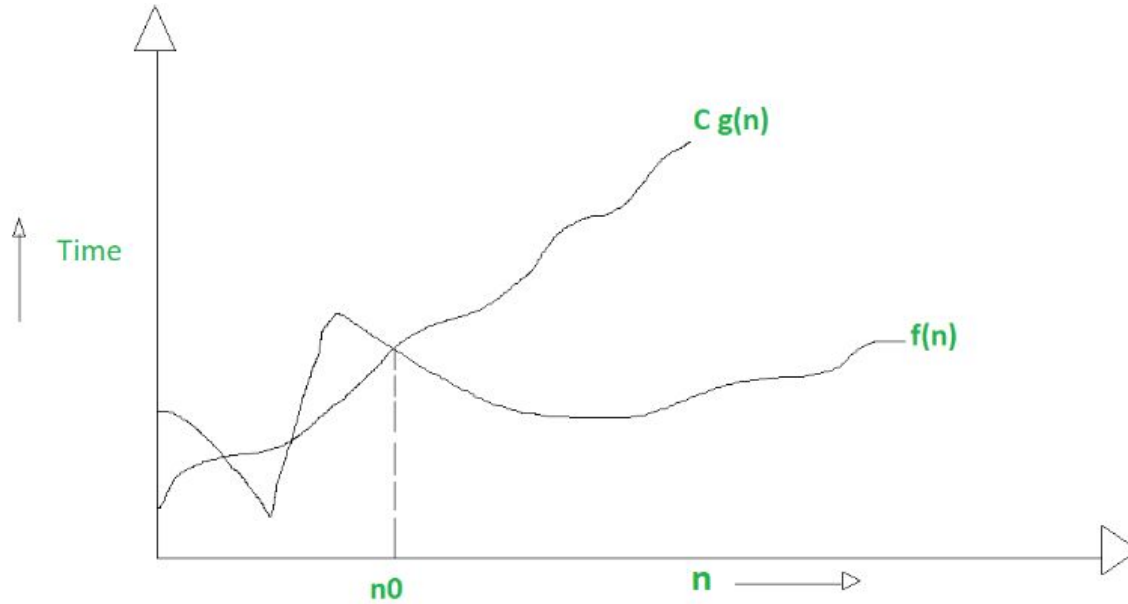
- Analysis of Algorithms
  - Best, Average and Worst case running times of algorithms,
  - **Mathematical notations for running times  $O$ ,  $\Omega$ ,  $\Theta$ ,**
  - Master's Theorem
- 
- Problem solving principles:
  - Classification of problem,
  - problem solving strategies,
  - classification of time complexities (linear, logarithmic etc.)
- 
- Divide and Conquer strategy:
  - General strategy,
  - Quick Sort and Merge Sort w.r.t. Complexity

## Sample Examples

$3n+2 = O(n)$	Because $3n+2 \leq 4n$ , for $n \geq 2$
$3n+3 = O(n)$	Because $3n+3 \leq 4n$ , for $n \geq 3$
$100n+6 = O(n)$	Because $100n+6 \leq 101n$ , for $n \geq 6$
$10n^2+4n+2 = O(n^2)$	Because $10n^2+4n+2 \leq 11n^2$ , for $n \geq 5$
$1000n^2+100n-6 = O(n^2)$	Because $1000n^2+100n-6 \leq 1001n^2$ , for $n \geq 100$
$6 \cdot 2^n + n^2 = O(2^n)$	$6 \cdot 2^n + n^2 \leq 7(2^n)$ , for $n \geq 4$
$3n+2 = O(n^2)$	Because $3n+2 \leq 3n^2$ , for $n \geq 2$
$10n^2+4n+2 = O(n^4)$	Because $10n^2+4n+2 \leq 10n^4$ , for $n \geq 2$
$3n^2+2 \neq O(n)$	Because $3n^2+2$ is <b>NOT</b> less than or equal to <b>any</b> $cn$ for all $n \geq n_0$
$10n+2 \neq O(1)$	Because $10n+2$ is <b>NOT</b> less than or equal to <b>any</b> $c$ for all $n \geq n_0$

$3n+2 = \Omega(n)$	Because $3n+2 \geq 3n$ , for $n \geq 1$
$3n+3 = \Omega(n)$	Because $3n+3 \geq 3n$ , for $n \geq 1$
$100n+6 = \Omega(n)$	Because $100n+6 \geq 100n$ , for $n \geq 1$
$10n^2+4n+2 = \Omega(n^2)$	Because $10n^2+4n+2 \geq 10n^2$ , for $n \geq 1$
$1000n^2+100n-6 = \Omega(n^2)$	Because $1000n^2+100n-6 \geq 1000n^2$ , for $n \geq 1$
$6 \cdot 2^n + n^2 = \Omega(2^n)$	$6 \cdot 2^n + n^2 \geq 6(2^n)$ , for $n \geq 1$
$3n+2 = \Omega(1)$	Because $3n+2 \geq 3$ , for $n \geq 1$
$10n^2+4n+2 = \Omega(n)$	Because $10n^2+4n+2 \geq 10n$ , for $n \geq 1$
$3n^2+2 \neq \Omega(n^4)$	Because $3n^2+2$ is <b>NOT</b> greater than or equal to <b>any</b> $cn^4$ for all $n \geq n_0$
$10n+2 \neq \Omega(n^2)$	Because $10n+2$ is <b>NOT</b> greater than or equal to <b>any</b> $cn^2$ for all $n \geq n_0$

# Graphical example for Big Oh ( $O$ )



# Sample Examples

$3n+2 = O(n)$	Because $3n+2 \leq 4n$ , for $n \geq 2$
$3n+3 = O(n)$	Because $3n+3 \leq 4n$ , for $n \geq 3$
$100n+6 = O(n)$	Because $100n+6 \leq 101n$ , for $n \geq 6$
$10n^2+4n+2 = O(n^2)$	Because $10n^2+4n+2 \leq 11n^2$ , for $n \geq 5$
$1000n^2+100n-6 = O(n^2)$	Because $1000n^2+100n-6 \leq 1001n^2$ , for $n \geq 100$
$6 \cdot 2^n + n^2 = O(2^n)$	$6 \cdot 2^n + n^2 \leq 7(2^n)$ , for $n \geq 4$
$3n+2 = O(n^2)$	Because $3n+2 \leq 3n^2$ , for $n \geq 2$
$10n^2+4n+2 = O(n^4)$	Because $10n^2+4n+2 \leq 10n^4$ , for $n \geq 2$
$3n^2+2 \neq O(n)$	Because $3n^2+2$ is <b>NOT</b> less than or equal to <b>any cn</b> for all $n \geq n_0$
$10n+2 \neq O(1)$	Because $10n+2$ is <b>NOT</b> less than or equal to <b>any c</b> for all $n \geq n_0$

# Big Omega notation ( $\Omega$ )

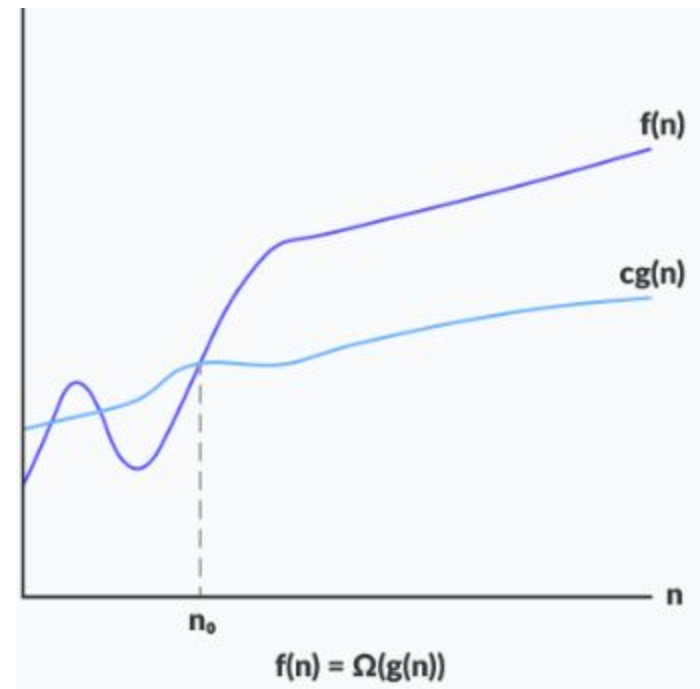
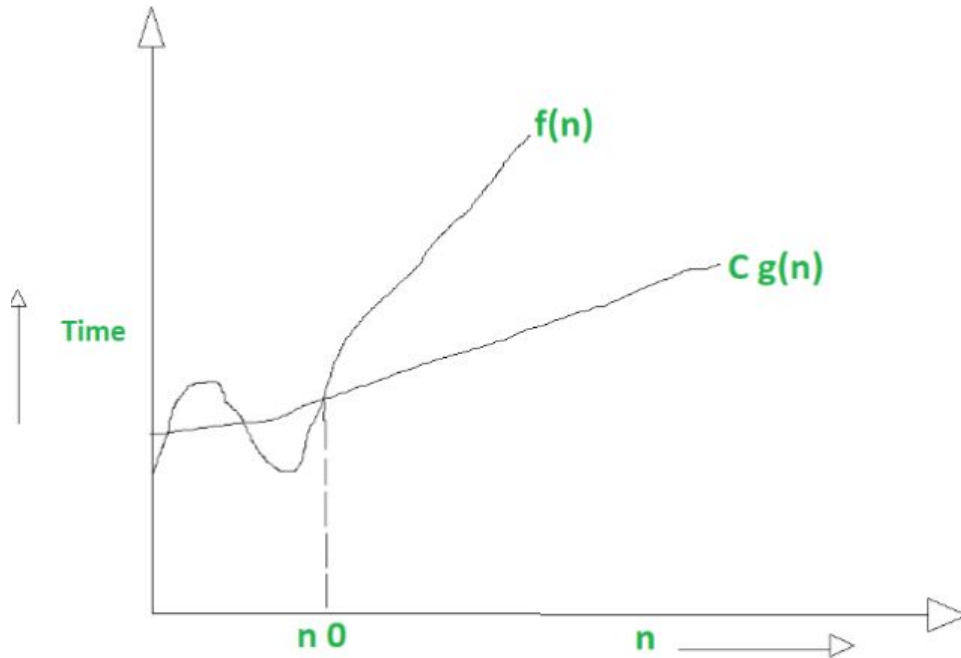
$f(n)$  defines running time of an algorithm

$f(n)$  is said to be  $\Omega(g(n))$  if there exists positive constant  $C$  and  $(n_0)$  such that

$$0 \leq Cg(n) \leq f(n) \text{ for all } n \geq n_0$$

If a function is  $\Omega(n^2)$  it is automatically  $\Omega(n)$  ,  $\Omega(1)$  as well

# Graphical example for **Big Omega ( $\Omega$ )**



# Sample Examples

$3n+2 = \Omega(n)$	Because $3n+2 \geq 3n$ , for $n \geq 1$
$3n+3 = \Omega(n)$	Because $3n+3 \geq 3n$ , for $n \geq 1$
$100n+6 = \Omega(n)$	Because $100n+6 \geq 100n$ , for $n \geq 1$
$10n^2+4n+2 = \Omega(n^2)$	Because $10n^2+4n+2 \geq 10n^2$ , for $n \geq 1$
$1000n^2+100n-6 = \Omega(n^2)$	Because $1000n^2+100n-6 \geq 1000n^2$ , for $n \geq 1$
$6 \cdot 2^n + n^2 = \Omega(2^n)$	$6 \cdot 2^n + n^2 \geq 6(2^n)$ , for $n \geq 1$
$3n+2 = \Omega(1)$	Because $3n+2 \geq 3$ , for $n \geq 1$
$10n^2+4n+2 = \Omega(n)$	Because $10n^2+4n+2 \geq 10n$ , for $n \geq 1$
$3n^2+2 \neq \Omega(n^4)$	Because $3n^2+2$ is <b>NOT</b> greater than or equal to <b>any</b> $cn^4$ for <b>all</b> $n \geq n_0$
$10n+2 \neq \Omega(n^2)$	Because $10n+2$ is <b>NOT</b> greater than or equal to <b>any</b> $cn^2$ <b>for all</b> $n \geq n_0$

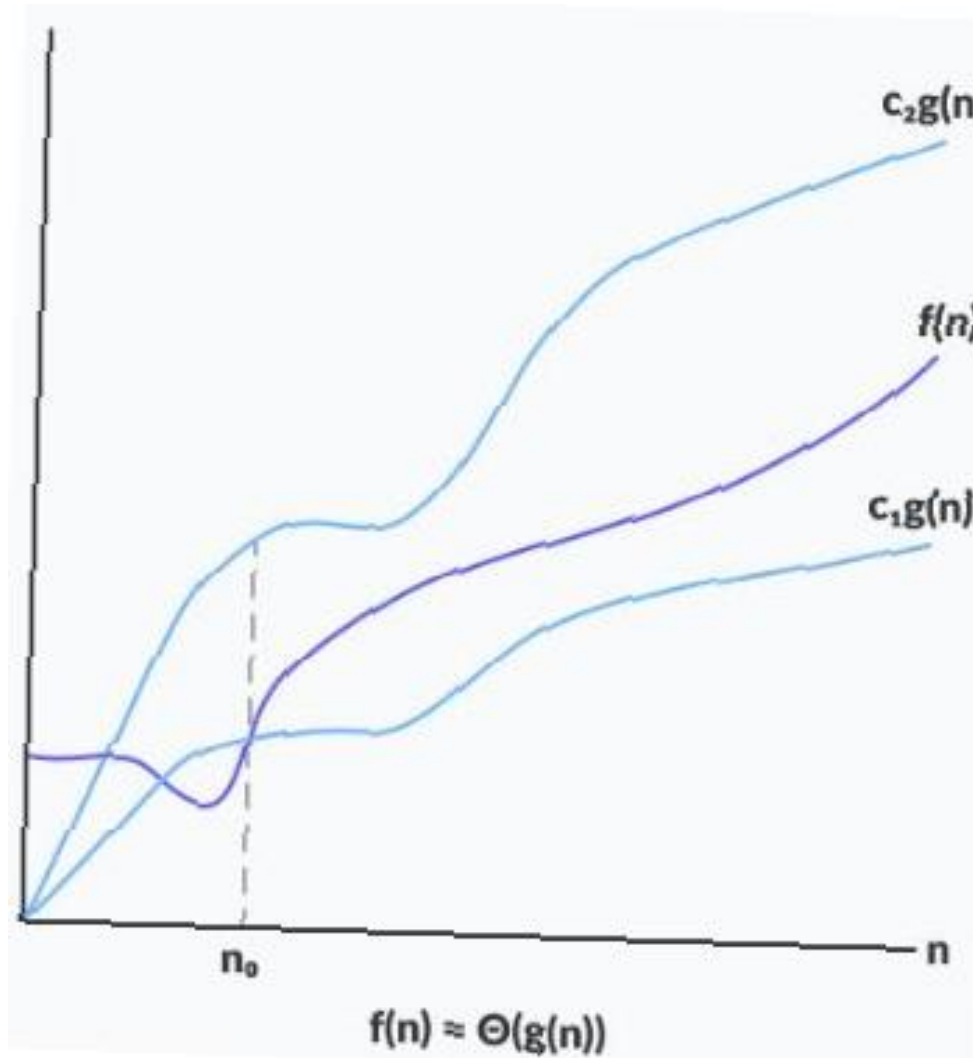
# Big Theta notation ( $\Theta$ )

It is define as tightest bound

- and tightest bound is the best of all the worst case times that the algorithm can take.
- Let  **$f(n)$**  define running time of an algorithm.  
 **$f(n)$**  is said to be  $\Theta$   
 **$(g(n))$**  if  **$f(n)$**  is  **$O(g(n))$**  and  **$f(n)$**  is  **$\Omega(g(n))$** .



# Graphical representation



# Example

- Consider  $f(n)=3n+2$

$$f(n) \geq 1n \quad \Rightarrow c_1=1, n_0 \geq 1 \quad \text{i.e. } f(n) = \Omega(n)$$

$$f(n) \leq 5n \quad \Rightarrow c_2=5, n_0 \geq 1 \quad \text{i.e. } f(n) = O(n)$$

$$1n \leq f(n) \leq 5n$$

$$c_1 * n \leq f(n) \leq c_2 * n \text{ for } n \geq n_0$$

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

# Sample Examples

$3n+2 = \theta(n)$	Because $3n \leq 3n+2 \leq 4n$ , for $n \geq 2$
$3n+3 = \theta(n)$	Because $3n \leq 3n+3 \leq 4n$ , for $n \geq 3$
$100n+6 = \theta(n)$	Because $100n \leq 100n+6 \leq 101n$ , for $n \geq 6$
$10n^2+4n+2 = \theta(n^2)$	Because $10n^2 \leq 10n^2+4n+2 \leq 11n^2$ , for $n \geq 5$
$1000n^2+100n-6 = \theta(n^2)$	Because $1000n^2 \leq 1000n^2+100n-6 \leq 1001n^2$ , for $n \geq 100$
$6 \cdot 2^n + n^2 = \theta(2^n)$	Because $6 \cdot 2^n \leq 6 \cdot 2^n + n^2 \leq 7(2^n)$ , for $n \geq 4$

# Mathematics of Time Complexity

- with Asymptotic Notations  $O$ ,  $\Theta$ ,  
 $\Omega$

Use of Asymptotic Notations

# Asymptotic Analysis for Algorithms

- The asymptotic behavior of a function  $f(n)$  refers to the growth of  $f(n)$  as  $n$  gets large.
- We typically ignore small values of  $n$ , since we are usually interested in estimating how slow the program will be on large inputs.

# Big 'Oh' notation (O) for TC

- **Big 'Oh'** *notation represents the upper bound of the running time of an algorithm, and it is used for analyzing the **best-case** complexity of an algorithm.*

# Big Omega notation ( $\Omega$ ) for TC

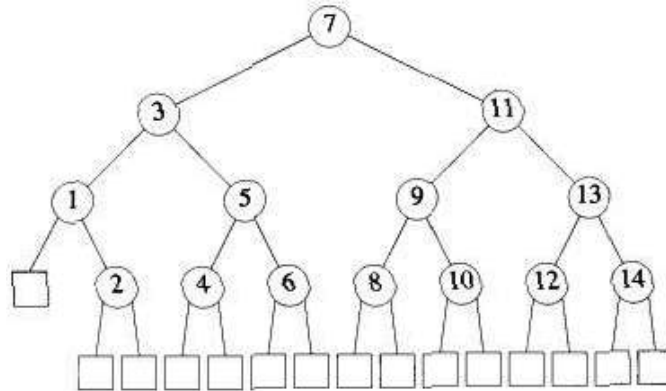
- **Omega notation** represents the lower bound of the running time of an algorithm, and it is used for analyzing the **best-case** complexity of an algorithm.
  - For any value of  $n$ , the minimum time required by the algorithm is given by Omega  $\Omega(g(n))$

# Theta notation ( $\Theta$ ) for TC

- *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 gives a tight bound of the time complexity of an algorithm*
- *It is used for analyzing the **average-case** complexity of an algorithm.*

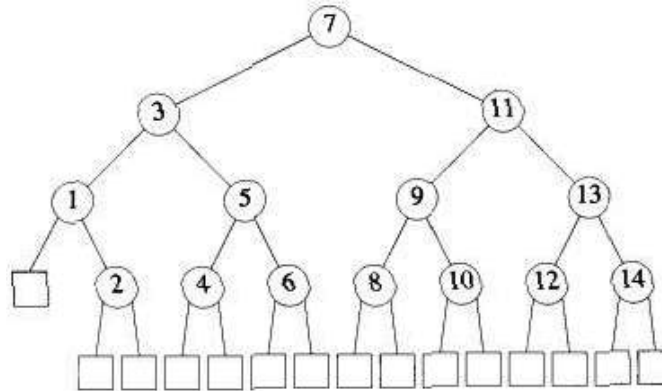


# Revisiting Best, Worst, Average Case T.C. - Binary Search (Which is Correct?)



Successful Searches			Unsuccessful Searches		
$O(1)$ , Best	$O(\log n)$ , Average	$O(\log n)$ Worst	$O(\log n)$ , Best	$O(\log n)$ , Average	$O(\log n)$ Worst
Successful Searches			Unsuccessful Searches		
$\Omega(1)$ , Best	$\Omega(\log n)$ , Average	$\Omega(\log n)$ Worst	$\Omega(\log n)$ , Best	$\Omega(\log n)$ , Average	$\Omega(\log n)$ Worst
Successful Searches			Unsuccessful Searches		
$\theta(1)$ , Best	$\theta(\log n)$ , Average	$\theta(\log n)$ Worst	$\theta(\log n)$ , Best	$\theta(\log n)$ , Average	$\theta(\log n)$ Worst

# Revisiting Best, Worst, Average Case T.C. - Binary Search (Which is Correct? [Ans: All])



Successful Searches			Unsuccessful Searches		
$O(1)$ , Best	$O(\log n)$ , Average	$O(\log n)$ Worst	$O(\log n)$ , Best	$O(\log n)$ , Average	$O(\log n)$ Worst

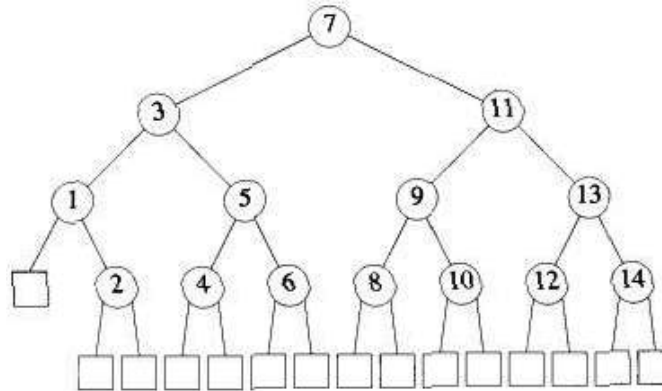
  

Successful Searches			Unsuccessful Searches		
$\Omega(1)$ , Best	$\Omega(\log n)$ , Average	$\Omega(\log n)$ Worst	$\Omega(\log n)$ , Best	$\Omega(\log n)$ , Average	$\Omega(\log n)$ Worst

Successful Searches			Unsuccessful Searches		
$\theta(1)$ , Best	$\theta(\log n)$ , Average	$\theta(\log n)$ Worst	$\theta(\log n)$ , Best	$\theta(\log n)$ , Average	$\theta(\log n)$ Worst

# Revisiting Best, Worst, Average Case T.C. - Binary Search (Which is Correct? – Ans: All are correct)



Successful Searches			Unsuccessful Searches		
$O(1)$ , Best	$O(\log n)$ , Average	$O(\log n)$ Worst	$O(\log n)$ , Best	$O(\log n)$ , Average	$O(\log n)$ Worst

Successful Searches			Unsuccessful Searches		
$\Omega(1)$ , Best	$\Omega(\log n)$ , Average	$\Omega(\log n)$ Worst	$\Omega(\log n)$ , Best	$\Omega(\log n)$ , Average	$\Omega(\log n)$ Worst

Successful Searches			Unsuccessful Searches		
$\theta(1)$ , Best	$\theta(\log n)$ , Average	$\theta(\log n)$ Worst	$\theta(\log n)$ , Best	$\theta(\log n)$ , Average	$\theta(\log n)$ Worst

# Obtaining Time Complexities of Recursive Functions

---

```
1  Algorithm DAndC( $P$ )
2  {
3      if Small( $P$ ) then return S( $P$ );
4      else
5      {
6          divide  $P$  into smaller instances  $P_1, P_2, \dots, P_k$ ,  $k \geq 1$ ;
7          Apply DAndC to each of these subproblems;
8          return Combine(DAndC( $P_1$ ), DAndC( $P_2$ ), ..., DAndC( $P_k$ ));
9      }
10 }
```

---

Control abstraction for divide-and-conquer

# Recurrence Relation for Time Complexity (TC) of the General Algorithm DAndC

$$T(n) = \begin{cases} g(n) & \text{for SMALL } n \\ T(n_1) + T(n_2) + \dots + T(n_k) + f(n) & \text{Otherwise} \end{cases}$$

Complexity of **Many divide and conquer algorithms** is

$$T(n) = \begin{cases} T(1) & \text{for } n = 1 \\ aT(n/b) + f(n) & \text{for } n > 1 \end{cases}$$

Where  $a, b$  are known constants.

We assume, for TC calculations,  $T(1)$  is known and  $n$  is a power of  $b$   
i.e.  $n = b^k$

Solve the recurrence relation for  $a, b=2$   
and  $f(n)=n$  and  $T(1)=7$

- $T(n) = \begin{cases} T(1) & \text{for } n = 1 \\ aT(n/b) + f(n) & \text{for } n > 1 \end{cases}$

Using SUBSTITUTION METHOD

# Unit-1

- Analysis of Algorithms
  - Best, Average and Worst case running times of algorithms,
  - Mathematical notations for running times  $O$ ,  $\Omega$ ,  $\Theta$ ,
  - **Master's Theorem**
- 
- Problem solving principles: Classification of problem, problem solving strategies, classification of time complexities (linear, logarithmic etc.)
  - Divide and Conquer strategy: General strategy, Quick Sort and Merge Sort w.r.t. Complexity



# What is Master's Theorem?

- The master theorem always yields **asymptotically tight bounds** to recurrences from divide and conquer algorithms that partition an input into smaller subproblems of equal sizes, solve the subproblems recursively, and then combine the subproblem solutions to give a solution to the original problem.

## ..contd.. What is Master's Theorem?

- The time for such an algorithm can be expressed by **adding the work** that they perform at the top level of their recursion (to **divide** the problems into subproblems and then **combine** the subproblem solutions) together with the time made in the recursive calls of the algorithm
- This can be expressed by a [recurrence relation](#) that takes the form:

# Master Method

- The master method provides a “cookbook” method for solving recurrences of the 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

# Master Theorem

$T(n) = aT(n/b) + f(n)$ , where  $a \geq 1$ ,  $b > 1$  and  $f(n)$  is a given positive function;

$$f(n) = \theta(n^k \log^p n)$$

case 1: if  $\log_b a > k$  ,  $\theta(n^{\log_b a})$

case 2: if  $\log_b a = k$  ,

$$\text{if } p > -1, \quad \theta(n^k \log^{p+1} n)$$

$$\text{if } p = -1, \quad \theta(n^k \log \log n)$$

$$\text{if } p < -1, \quad \theta(n^k)$$

case 3: if  $\log_b a < k$  ,

$$\text{if } p \geq 0, \quad \theta(n^k \log^p n)$$

$$\text{if } p < 0, \quad O(n^k)$$

# Master Theorem Example Case 1

$T(n) = aT(n/b) + f(n)$ , where  $a \geq 1, b > 1$ ,  $f(n) = \theta(n^k \log^p n)$

**case 1:** if  $\log_b a > k$ ,  $\theta(n^{\log_b a})$

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

$$a=2, b=2$$

$$f(n) = 1;$$

$$k=0$$

$$p=0$$

$$\log_b a = \log_2 2 = 1$$

$$\log_b a > 0 \text{ Hence Case 1}$$

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

$$T(n) = \theta(n^1)$$

# Master Theorem Example Case 1

$T(n) = aT(n/b) + f(n)$ , where  $a \geq 1, b > 1$ ,  $f(n) = \theta(n^k \log^p n)$

**case 1:** if  $\log_b a > k$ ,  $\theta(n^{\log_b a})$

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

$$a=4, b=2$$

$$f(n) = n;$$

$$k=1$$

$$p=0$$

$$\log_b a = \log_2 4 = 2$$

$\log_b a > 1$  Hence Case 1

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

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

# Master Theorem Example Case 1

$T(n) = aT(n/b) + f(n)$ , where  $a \geq 1, b > 1$ ,  $f(n) = \theta(n^k \log^p n)$

**case 1:** if  $\log_b a > k$ ,  $\theta(n^{\log_b a})$

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

$$a=8, b=2$$

$$f(n) = n;$$

$$k=1$$

$$p=0$$

$$\log_b a = \log_2 8 = 3$$

$\log_b a > k$  Hence Case 1

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

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

# Master Theorem Case 2

$T(n) = aT(n/b) + f(n)$ , where  $a \geq 1, b > 1$ ,  $f(n) = \theta(n^k \log^p n)$

case 2: if  $\log_b a = k$ ,

$$\text{if } p > -1, \quad \theta(n^k \log^{p+1} n)$$

$$\text{if } p = -1, \quad \theta(n^k \log \log n)$$

$$\text{if } p < -1, \quad \theta(n^k)$$

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

$$a=2, b=2$$

$$f(n) = n;$$

$$k=1$$

$$p=0$$

$$\log_b a = \log_2 2 = 1$$

$$\log_b a = k \text{ Hence Case 2 } p > -1$$

$$T(n) = \theta(n^k \log^{p+1} n)$$

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



# Master Theorem Case 2

$T(n) = aT(n/b) + f(n)$ , where  $a \geq 1, b > 1$ ,  $f(n) = \theta(n^k \log^p n)$

case 2: if  $\log_b a = k$ ,

if  $p > -1$ ,  $\theta(n^k \log^{p+1} n)$

if  $p = -1$ ,  $\theta(n^k \log \log n)$

if  $p < -1$ ,  $\theta(n^k)$

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

$$a=2, b=2$$

$$f(n) = n/\log n;$$

$$k=1$$

$$p=-1$$

$$\log_b a = \log_2 2 = 1$$

$$\log_b a = k \text{ Hence Case 2 } p = -1$$

$$T(n) = \theta(n^k \log \log n)$$

$$\text{Therefore } T(n) = \theta(n \log \log n)$$

# Master Theorem Case 2

$T(n) = aT(n/b) + f(n)$ , where  $a \geq 1, b > 1$ ,  $f(n) = \theta(n^k \log^p n)$

case 2: if  $\log_b a = k$ ,

if  $p > -1$ ,  $\theta(n^k \log^{p+1} n)$

if  $p = -1$ ,  $\theta(n^k \log \log n)$

if  $p < -1$ ,  $\theta(n^k)$

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

$$a=2, b=2$$

$$f(n) = n/\log^2 n;$$

$$k=1$$

$$p=-2$$

$$\log_b a = \log_2 2 = 1$$

$$\log_b a = k \text{ Hence Case 2 } p < -1$$

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

$$\text{Therefore } T(n) = \theta(n)$$

# Master Theorem Case 3

•  $T(n) = aT(n/b) + f(n)$ , where  $a \geq 1, b > 1$ ,  $f(n) = \theta(n^k \log^p n)$

Case 3: if  $\log_b a < k$ ,  
if  $p \geq 0$ ,  $\theta(n^k \log^p n)$   
if  $p < 0$ ,  $O(n^k)$

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

$$a=2, b=2$$

$$f(n) = n^2 \log n;$$

$$k=2$$

$$p=1$$

$$\log_b a = \log_2 2 = 1$$

$$\log_b a < k \text{ Hence Case 3 } p > 0$$

$$T(n) = \theta(n^k \log^p n)$$

$$\text{Therefore } T(n) = \theta(n^2 \log n)$$

# Master Theorem Case 3

•  $T(n) = aT(n/b) + f(n)$ , where  $a \geq 1, b > 1$ ,  $f(n) = \theta(n^k \log^p n)$

Case 3: if  $\log_b a < k$ ,  
if  $p \geq 0$ ,  $\theta(n^k \log^p n)$   
if  $p < 0$ ,  $O(n^k)$

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

$$a=1, b=2$$

$$f(n) = n^2;$$

$$k=2$$

$$p=0$$

$$\log_b a = \log_2 1 = 0$$

$$\log_b a < k \text{ Hence Case 3 } p=0$$

$$T(n) = \theta(n^k \log^p n)$$

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

# Solve using Master Theorem

1.  $T(n) = 9 T(n/3) + 1$
2.  $T(n) = 8 T(n/2) + n$
3.  $T(n) = 8 T(n/2) + n \log n$
4.  $T(n) = 4 T(n/2) + n^2$
5.  $T(n) = 4 T(n/2) + n^2 \log n$
6.  $T(n) = 8 T(n/2) + n^3$
7.  $T(n) = 4 T(n/2) + n^3$

# Solution to Given problems

1.  $T(n) = 9 T(n/3) + 1$

$a=9$   $b=3$   $\log_b a = \log_3 9 = 2 > k=0$ , thus answer =  $\theta(n^2)$

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

$a=8$   $b=2$   $\log_b a = \log_2 8 = 3 > k=1$ , thus answer =  $\theta(n^3)$

3.  $T(n) = 8 T(n/2) + n \log n$

$a=8$   $b=2$   $\log_b a = \log_2 8 = 3 > k=1$ , thus answer =  $\theta(n^3)$

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

$a=4$   $b=2$   $\log_b a = \log_2 4 = 2 = k=2$ ,  $P = 0 > -1$ , thus answer =  $\theta(n^2 \log n)$

5.  $T(n) = 4 T(n/2) + n^2 \log n$

$a=4$   $b=2$   $\log_b a = \log_2 4 = 2 = k=2$ ,  $P = 1 > -1$ , thus answer =  $\theta(n^2 \log^2 n)$

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

$a=8$   $b=2$   $\log_b a = \log_2 8 = 3 = k=3$ ,  $P = 0 > -1$ , thus answer =  $\theta(n^3 \log n)$

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

$a=4$   $b=2$   $\log_b a = \log_2 4 = 2 < k=3$ ,  $P = 0 > -1$ , thus answer =  $\theta(n^3)$

# Unit-1

- Analysis of Algorithms
  - Best, Average and Worst case running times of algorithms,
  - Mathematical notations for running times  $O$ ,  $\Omega$ ,  $\Theta$ ,
  - Master's Theorem
- 
- **Problem solving principles: Classification of problem,**
  - problem solving strategies,
  - classification of time complexities (linear, logarithmic etc.)
- 
- Divide and Conquer strategy: General strategy, Quick Sort and Merge Sort w.r.t. Complexity

# Problem solving

- Application of ideas, skills, or factual information to achieve the solution to a problem or to reach a desired outcome. Let's talk about different types of problems and different types of solutions.



# Kinds of problems encountered

- A **well-defined problem** is one that has a clear goal or solution, and problem solving strategies are easily developed.
  - In contrast, **poorly-defined** problem is the opposite. It is unclear, abstract, or confusing, and that does not have a clear problem solving strategy.
- **Routine problem** is one that is typical and has a simple solution
  - In contrast, **non-routine** problem is more abstract or subjective and requires a strategy to solve

# Unit-1

- Analysis of Algorithms
  - Best, Average and Worst case running times of algorithms,
  - Mathematical notations for running times  $O$ ,  $\Omega$ ,  $\Theta$ ,
  - Master's Theorem
- 
- Problem solving principles: Classification of problem,
  - **problem solving strategies,**
  - classification of time complexities (linear, logarithmic etc.)
- 
- Divide and Conquer strategy: General strategy, Quick Sort and Merge Sort w.r.t. Complexity

# Strategies

- To solve a routine problem algorithm can be used. Algorithms are step-by-step strategies or processes for how to solve a problem or achieve a goal.
- Another solution that many people use to solve problems is called heuristics.
  - Heuristics are general strategies used to make quick, short-cut solutions to problems that sometimes lead to solutions but sometimes lead to errors.
  - Heuristics are based on past experiences.

# Principle of Problem Solving

- Identify a problem
- Understand the problem
- Identify alternative ways to solve a problem
- Select best way to solve a problem from the list of alternative solutions
- Evaluate the solution

# Problem Solving Strategies in Course

1. Divide & Conquer
2. Greedy Method
3. Dynamic Programming
4. Backtracking
5. Branch and Bound
  
6. Randomized algorithms
7. Approximation algorithms
8. Natural Algorithms
9. Parallel and concurrent

# Unit-1

- Analysis of Algorithms
  - Best, Average and Worst case running times of algorithms,
  - Mathematical notations for running times  $O$ ,  $\Omega$ ,  $\Theta$ ,
  - Master's Theorem
- 
- Problem solving principles: Classification of problem,
  - problem solving strategies,
  - **classification of time complexities (linear, logarithmic etc.)**
- 
- Divide and Conquer strategy: General strategy,
  - Quick Sort and Merge Sort w.r.t. Complexity

# Increasing Time Complexity Functions

$\log n \leq \sqrt{n} \leq n \leq n \log n \leq n^2 \leq n^3 \leq 2^n \leq n!$

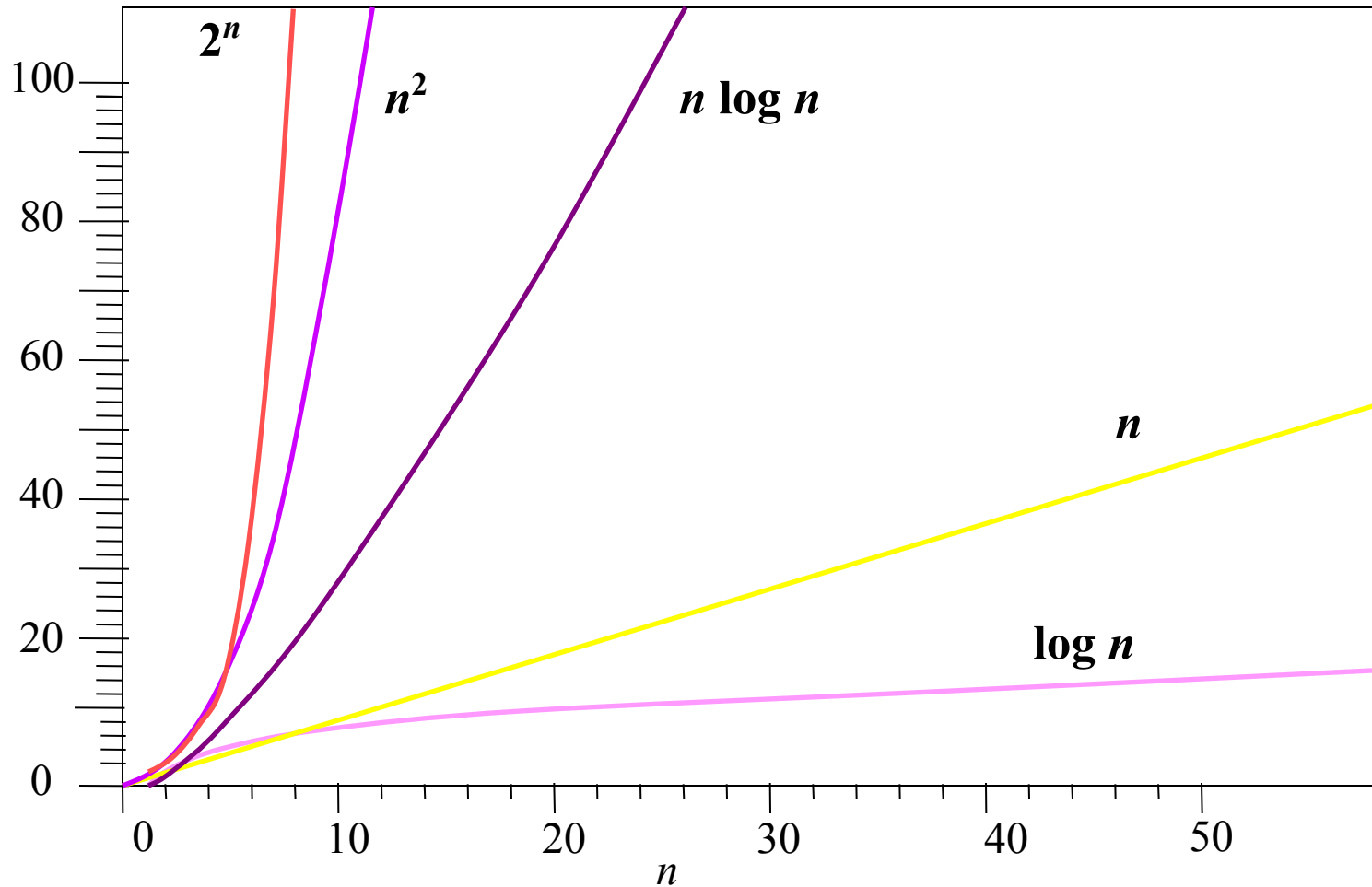
Name	Time Complexity
Constant Time	$O(1)$
Logarithmic Time	$O(\log n)$
Linear Time	$O(n)$
Quasilinear Time	$O(n \log n)$
Quadratic Time	$O(n^2)$
Exponential Time	$O(2^n)$
Factorial Time	$O(n!)$

# Classification of Time Complexity

Notation	Complexity	Description	Example
<b><math>O(1)</math></b>	Constant	Simple statement	<b>Addition</b>
<b><math>O(\log(n))</math></b>	Logarithmic	Divide in half	<b>Binary search</b>
<b><math>O(n)</math></b>	Linear	loop	<b>Linear search</b>
<b><math>O(n \cdot \log(n))</math></b>	Linearithmic	Divide & Conquer	<b>Merge sort</b>
<b><math>O(n^2)</math></b>	Quadratic	Double loop	<b>Check all pairs</b>
<b><math>O(n^3)</math></b>	Cubic	Triple loop	<b>Check all triples</b>
<b><math>O(2^n)</math></b>	Exponential	Exhaustive search	<b>Check all subsets</b>
<b><math>O(n!)</math></b>	Factorial	Recursive function	<b>Factorial</b>



# Graphical Comparison



# Unit-1

- Analysis of Algorithms
  - Best, Average and Worst case running times of algorithms,
  - Mathematical notations for running times  $O$ ,  $\Omega$ ,  $\Theta$ ,
  - Master's Theorem
- 
- Problem solving principles: Classification of problem,
  - problem solving strategies,
  - classification of time complexities (linear, logarithmic etc.)
- 
- **Divide and Conquer strategy: General strategy.**
  - Quick Sort and Merge Sort w.r.t. Complexity

# Divide & Conquer

- Given a function to compute on  $n$  inputs the divide and conquer strategy suggests splitting the inputs into  $k$  distinct substs,  $1 < k \leq n$ , yielding  $k$  sub problems.
- These sub problems must be solved and then a method must be found to combine sub solutions into a solution of the whole.
- Sub problems resulting from a divide and conquer strategy design are of the same type as the original problem.
- Hence reapplication of the divide and conquer principle is naturally expressed as recursive algorithm.

# Control abstraction for DA&C

Algorithm DA&C( $P$ )

```
{  
  if Small( $P$ ) then return  $S(P)$ ;  
  else  
  {  
    divide  $P$  into smaller instances of  $P_1, P_2, \dots, P_k$ ,  $k \geq 1$ ;  
    Apply DA&C to each of these sub problems;  
    return Combine(DA&C( $P_1$ ), DA&C( $P_2$ ), ..., DA&C( $P_k$ ));  
  }  
}
```

# Time Complexity of DA&C

$$\begin{aligned} T(n) &= T(1) & n=1 \\ &a T(n/b) + f(n) & n>1 \end{aligned}$$

a and b are known constants .

We assume  $T(1)$

is known and n is a power of b(i. e  $n=b^k$  )

# Unit-1

- Analysis of Algorithms
  - Best, Average and Worst case running times of algorithms,
  - Mathematical notations for running times  $O$ ,  $\Omega$ ,  $\Theta$ ,
  - Master's Theorem
- 
- Problem solving principles: Classification of problem,
  - problem solving strategies,
  - classification of time complexities (linear, logarithmic etc.)
- 
- Divide and Conquer strategy: General strategy,
  - **Quick Sort and Merge Sort w.r.t. Complexity**

# Mergesort

# Merge Sort

The merge sort algorithm closely follows the divide-and-conquer paradigm. Intuitively, it operates as follows.

- **Divide:** Divide the  $n$ -element sequence to be sorted into two subsequences of  $n/2$  elements each.
- **Conquer:** Sort the two subsequences recursively using merge sort.
- **Combine:** Merge the two sorted subsequences to produce the sorted answer.



# Merge sort

MERGE-SORT( $A, p, r$ )

1 **if**  $p < r$

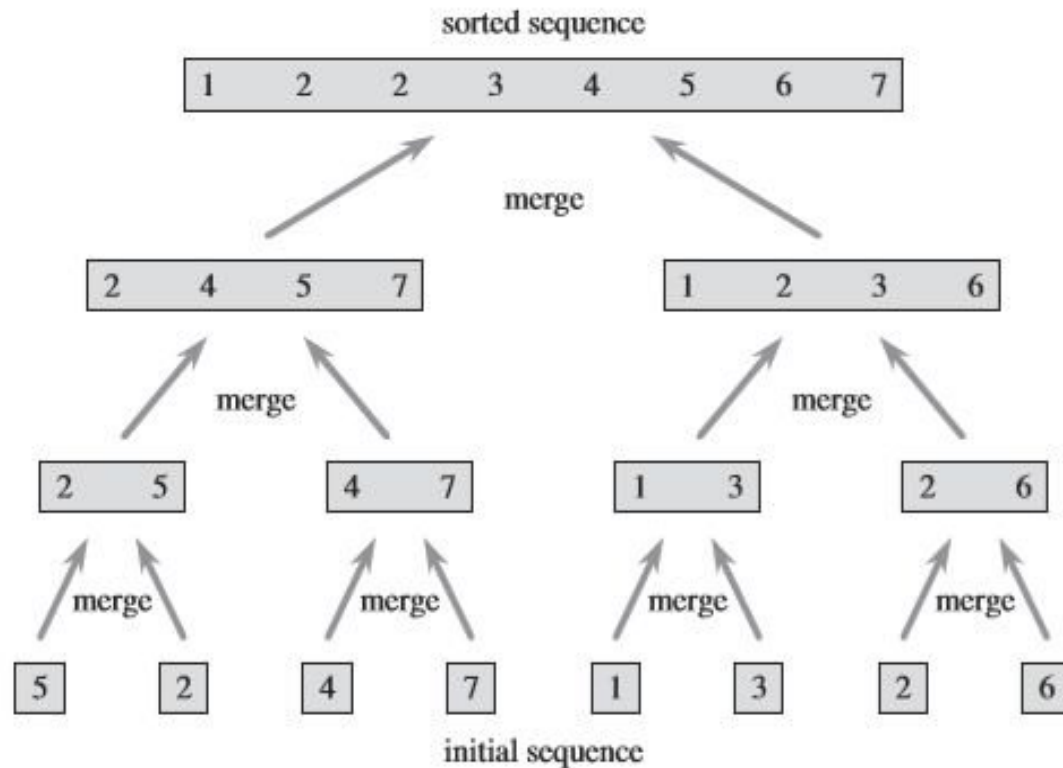
2      $q = \lfloor (p + r) / 2 \rfloor$

3     MERGE-SORT( $A, p, q$ )

4     MERGE-SORT( $A, q + 1, r$ )

5     MERGE( $A, p, q, r$ )

# Merge Sort Example



# Merge sort

MERGE( $A, p, q, r$ )

```
1   $n_1 = q - p + 1$ 
2   $n_2 = r - q$ 
3  let  $L[1 \dots n_1 + 1]$  and  $R[1 \dots n_2 + 1]$  be new arrays
4  for  $i = 1$  to  $n_1$ 
5       $L[i] = A[p + i - 1]$ 
6  for  $j = 1$  to  $n_2$ 
7       $R[j] = A[q + j]$ 
8   $L[n_1 + 1] = \infty$ 
9   $R[n_2 + 1] = \infty$ 
10  $i = 1$ 
11  $j = 1$ 
12 for  $k = p$  to  $r$ 
13     if  $L[i] \leq R[j]$ 
14          $A[k] = L[i]$ 
15          $i = i + 1$ 
16     else  $A[k] = R[j]$ 
17          $j = j + 1$ 
```

# Recurrence Relation for Merge Sort

$$T(n) = \begin{cases} \Theta(1) & \text{if } n \leq c, \\ aT(n/b) + D(n) + C(n) & \text{otherwise.} \end{cases}$$

**Divide:** The divide step just computes the middle of the subarray, which takes constant time. Thus,  $D(n) = \Theta(1)$ .

**Conquer:** We recursively solve two subproblems, each of size  $n/2$ , which contributes  $2T(n/2)$  to the running time.

**Combine:** We have already noted that the MERGE procedure on an  $n$ -element subarray takes time  $\Theta(n)$ , and so  $C(n) = \Theta(n)$ .

When we add the functions  $D(n)$  and  $C(n)$  for the merge sort analysis, we are adding a function that is  $\Theta(n)$  and a function that is  $\Theta(1)$ . This sum is a linear function of  $n$ , that is,  $\Theta(n)$ . Adding it to the  $2T(n/2)$  term from the “conquer” step gives the recurrence for the worst-case running time  $T(n)$  of merge sort:

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1, \\ 2T(n/2) + \Theta(n) & \text{if } n > 1. \end{cases} \quad (2.1)$$

Let us rewrite recurrence (2.1) as

$$T(n) = \begin{cases} c & \text{if } n = 1, \\ 2T(n/2) + cn & \text{if } n > 1, \end{cases} \quad (2.2)$$

# Quicksort

# Quicksort

QUICKSORT( $A, p, r$ )

```
1  if  $p < r$ 
2       $q = \text{PARTITION}(A, p, r)$ 
3      QUICKSORT( $A, p, q - 1$ )
4      QUICKSORT( $A, q + 1, r$ )
```

To sort an entire array  $A$ , the initial call is QUICKSORT( $A, 1, A.length$ ).

## Partitioning the array

The key to the algorithm is the PARTITION procedure, which rearranges the subarray  $A[p \dots r]$  in place.

PARTITION( $A, p, r$ )

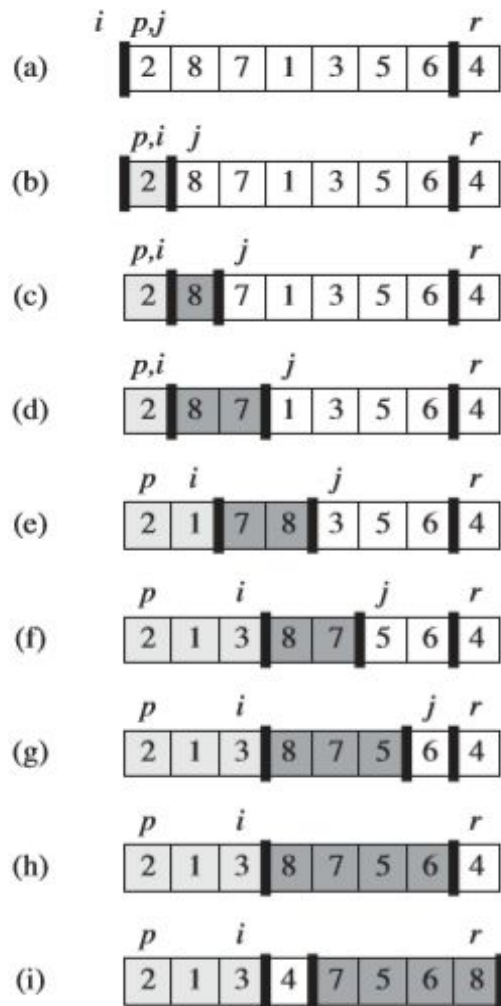
```
1   $x = A[r]$ 
2   $i = p - 1$ 
3  for  $j = p$  to  $r - 1$ 
4      if  $A[j] \leq x$ 
5           $i = i + 1$ 
6          exchange  $A[i]$  with  $A[j]$ 
7  exchange  $A[i + 1]$  with  $A[r]$ 
8  return  $i + 1$ 
```

# Quicksort Algorithm

Given an array of  $n$  elements (e.g., integers):

- If array only contains one element, return
- Else
  - pick one element to use as *pivot*.
  - Partition elements into two sub-arrays:
    - Elements less than or equal to pivot
    - Elements greater than pivot
  - Quicksort two sub-arrays
  - Return results

# Quicksort



**Figure 7.1** The operation of PARTITION on a sample array. Array entry  $A[r]$  becomes the pivot element  $x$ . Lightly shaded array elements are all in the first partition with values no greater than  $x$ . Heavily shaded elements are in the second partition with values greater than  $x$ . The unshaded elements have not yet been put in one of the first two partitions, and the final white element is the pivot  $x$ . (a) The initial array and variable settings. None of the elements have been placed in either of the first two partitions. (b) The value 2 is “swapped with itself” and put in the partition of smaller values. (c)–(d) The values 8 and 7 are added to the partition of larger values. (e) The values 1 and 8 are swapped, and the smaller partition grows. (f) The values 3 and 7 are swapped, and the smaller partition grows. (g)–(h) The larger partition grows to include 5 and 6, and the loop terminates. (i) In lines 7–8, the pivot element is swapped so that it lies between the two partitions.



- QuickSort Example
- 2 4 8 10 16 18 17

## Suggestion for improving Quick Sort

- Select middle element as pivot
- Select Random element as pivot
- Space Complexity:
- Quick sort is a recursive algorithm so it uses stack. Size of stack in worst case is  $n$  and for best case is  $\log n$  so size requirement varies from
- $\log n$  to  $n$

# Example

We are given array of n integers to sort:

40	20	10	80	60	50	7	30	100
----	----	----	----	----	----	---	----	-----

# Quicksort Analysis

- Assume that keys are random, uniformly distributed.
- What is best case running time?

# Quicksort Analysis

- Assume that keys are random, uniformly distributed.
- What is best case running time?
  - Recursion:
    1. Partition splits array in two sub-arrays of size  $n/2$
    2. Quicksort each sub-array

# Quicksort Analysis

- Assume that keys are random, uniformly distributed.
- What is best case running time?
  - Recursion:
    1. Partition splits array in two sub-arrays of size  $n/2$
    2. Quicksort each sub-array
  - Depth of recursion tree?

# Quicksort Analysis

- Assume that keys are random, uniformly distributed.
- What is best case running time?
  - Recursion:
    1. Partition splits array in two sub-arrays of size  $n/2$
    2. Quicksort each sub-array
  - Depth of recursion tree?  $O(\log_2 n)$

# Quicksort Analysis

- Assume that keys are random, uniformly distributed.
- What is best case running time?
  - Recursion:
    1. Partition splits array in two sub-arrays of size  $n/2$
    2. Quicksort each sub-array
  - Depth of recursion tree?  $O(\log_2 n)$
  - Number of accesses in partition?



# Quicksort Analysis

- Assume that keys are random, uniformly distributed.
- What is best case running time?
  - Recursion:
    1. Partition splits array in two sub-arrays of size  $n/2$
    2. Quicksort each sub-array
  - Depth of recursion tree?  $O(\log_2 n)$
  - Number of accesses in partition?  $O(n)$

# Quicksort Analysis

- Assume that keys are random, uniformly distributed.
- Best case running time:  $O(n \log_2 n)$

# Quicksort Analysis

- Assume that keys are random, uniformly distributed.
- Best case running time:  $O(n \log_2 n)$
- Worst case running time?

# Quicksort Analysis

- Assume that keys are random, uniformly distributed.
- Best case running time:  $O(n \log_2 n)$
- Worst case running time?
  - Recursion:
    1. Partition splits array in two sub-arrays:
      - one sub-array of size 0
      - the other sub-array of size  $n-1$
    2. Quicksort each sub-array
  - Depth of recursion tree?

# Quicksort Analysis

- Assume that keys are random, uniformly distributed.
- Best case running time:  $O(n \log_2 n)$
- Worst case running time?
  - Recursion:
    1. Partition splits array in two sub-arrays:
      - one sub-array of size 0
      - the other sub-array of size  $n-1$
    2. Quicksort each sub-array
  - Depth of recursion tree?  $O(n)$

# Quicksort Analysis

- Assume that keys are random, uniformly distributed.
- Best case running time:  $O(n \log_2 n)$
- Worst case running time?
  - Recursion:
    1. Partition splits array in two sub-arrays:
      - one sub-array of size 0
      - the other sub-array of size  $n-1$
    2. Quicksort each sub-array
  - Depth of recursion tree?  $O(n)$
  - Number of accesses per partition?

# Quicksort Analysis

- Assume that keys are random, uniformly distributed.
- Best case running time:  $O(n \log_2 n)$
- Worst case running time?
  - Recursion:
    1. Partition splits array in two sub-arrays:
      - one sub-array of size 0
      - the other sub-array of size  $n-1$
    2. Quicksort each sub-array
  - Depth of recursion tree?  $O(n)$
  - Number of accesses per partition?  $O(n)$

# Quicksort Analysis

- Assume that keys are random, uniformly distributed.
- Best case running time:  $O(n \log_2 n)$
- Worst case running time:  $O(n^2)$ !!!



# Quicksort Analysis

- Assume that keys are random, uniformly distributed.
- Best case running time:  $O(n \log_2 n)$
- Worst case running time:  $O(n^2)$ !!!
- What can we do to avoid worst case?

# Improved Pivot Selection

Pick median value of three elements from data array:  
data[0], data[n/2], and data[n-1].

Use this median value as pivot.

# Improving Performance of Quicksort

- Improved selection of pivot.
- For sub-arrays of size 3 or less, apply brute force search:
  - Sub-array of size 1: trivial
  - Sub-array of size 2:
    - if(`data[first] > data[second]`) swap them
  - Sub-array of size 3: left as an exercise.

# Quicksort

The three-step divide-and-conquer process for sorting a typical subarray  $A[p..r]$

**Divide:** Partition (rearrange) the array  $A[p..r]$  into two (possibly empty) subarrays  $A[p..q-1]$  and  $A[q+1..r]$  such that each element of  $A[p..q-1]$  is less than or equal to  $A[q]$ , which is, in turn, less than or equal to each element of  $A[q+1..r]$ . Compute the index  $q$  as part of this partitioning procedure.

**Conquer:** Sort the two subarrays  $A[p..q-1]$  and  $A[q+1..r]$  by recursive calls to quicksort.

**Combine:** Because the subarrays are already sorted, no work is needed to combine them: the entire array  $A[p..r]$  is now sorted.

# Quicksort

QUICKSORT( $A, p, r$ )

```
1  if  $p < r$ 
2       $q = \text{PARTITION}(A, p, r)$ 
3      QUICKSORT( $A, p, q - 1$ )
4      QUICKSORT( $A, q + 1, r$ )
```

To sort an entire array  $A$ , the initial call is QUICKSORT( $A, 1, A.length$ ).

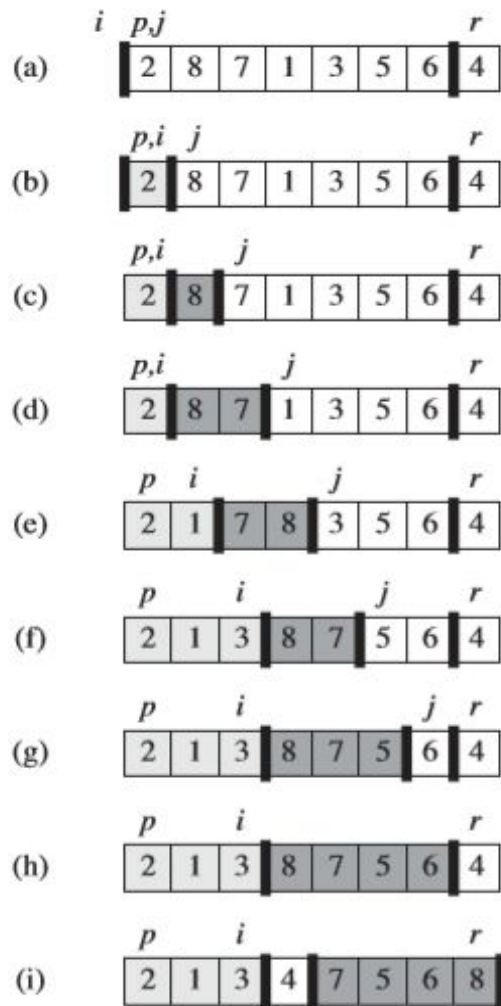
## Partitioning the array

The key to the algorithm is the PARTITION procedure, which rearranges the subarray  $A[p..r]$  in place.

PARTITION( $A, p, r$ )

```
1   $x = A[r]$ 
2   $i = p - 1$ 
3  for  $j = p$  to  $r - 1$ 
4      if  $A[j] \leq x$ 
5           $i = i + 1$ 
6          exchange  $A[i]$  with  $A[j]$ 
7  exchange  $A[i + 1]$  with  $A[r]$ 
8  return  $i + 1$ 
```

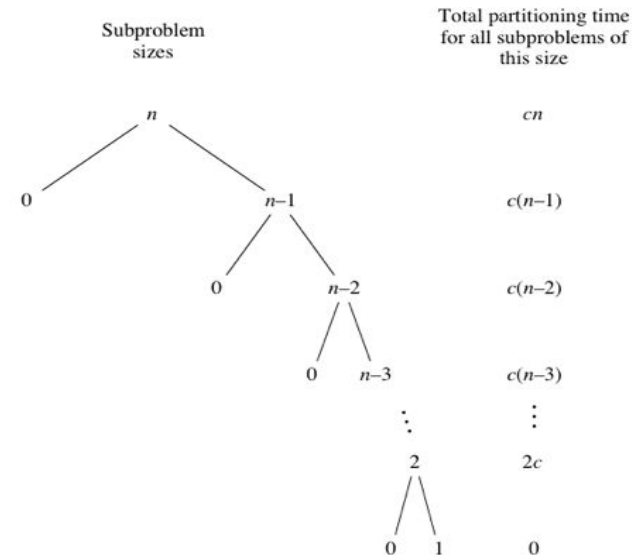
# Quicksort



**Figure 7.1** The operation of PARTITION on a sample array. Array entry  $A[r]$  becomes the pivot element  $x$ . Lightly shaded array elements are all in the first partition with values no greater than  $x$ . Heavily shaded elements are in the second partition with values greater than  $x$ . The unshaded elements have not yet been put in one of the first two partitions, and the final white element is the pivot  $x$ . (a) The initial array and variable settings. None of the elements have been placed in either of the first two partitions. (b) The value 2 is “swapped with itself” and put in the partition of smaller values. (c)–(d) The values 8 and 7 are added to the partition of larger values. (e) The values 1 and 8 are swapped, and the smaller partition grows. (f) The values 3 and 7 are swapped, and the smaller partition grows. (g)–(h) The larger partition grows to include 5 and 6, and the loop terminates. (i) In lines 7–8, the pivot element is swapped so that it lies between the two partitions.

# Quick Sort Analysis (Worst Case)

When quicksort always has the most unbalanced partitions possible, then the original call takes  $cn$  time for some constant  $c$ , the recursive call on  $n - 1$  elements takes  $c(n - 1)$  time, the recursive call on  $n - 2$  elements takes  $c(n - 2)$  time, and so on. Here's a tree of the subproblem sizes with their partitioning times:



When we sum all the partition times at each level :

$$cn + c(n-1) + c(n-2) + \dots + 2c = c(n + (n-1) + (n-2) + \dots + 2) = c((n+1)(n/2) - 1)$$

Therefore timecomplexity is  $\Theta(n^2)$

Recurrence relation for quicksort in worst case

$$T(n) = 1 \quad \text{if } n=0$$

$$T(n-1) + n \quad \text{if } n > 0$$



# Quick Sort Analysis (Worst Case) by substitution

Recurrence relation for quicksort in worst case

$$T(n) = \begin{cases} 1 & \text{if } n=0 \\ T(n-1) + n & \text{if } n > 0 \end{cases}$$

$$T(n) = T(n-1) + n \quad \text{----- (eq 1)}$$

Find value of  $T(n-1)$  by replacing  $n-1$  in place of  $n$  in eq 1

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

Put value of  $T(n-1)$  in equation 1 we have

$$T(n) = T(n-2) + (n-1) + n \quad \text{----- (eq 2)}$$

Find value of  $T(n-2)$  by replacing  $n-2$  in place of  $n$  in eq 1

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

Put value of  $T(n-2)$  in equation 2 we have

$$T(n) = T(n-3) + (n-2) + (n-1) + n \quad \text{----- (eq 3)}$$

|

$$T(n-k) + (n-(k-1)) + (n-(k-2)) + \dots + (n-1) + n$$

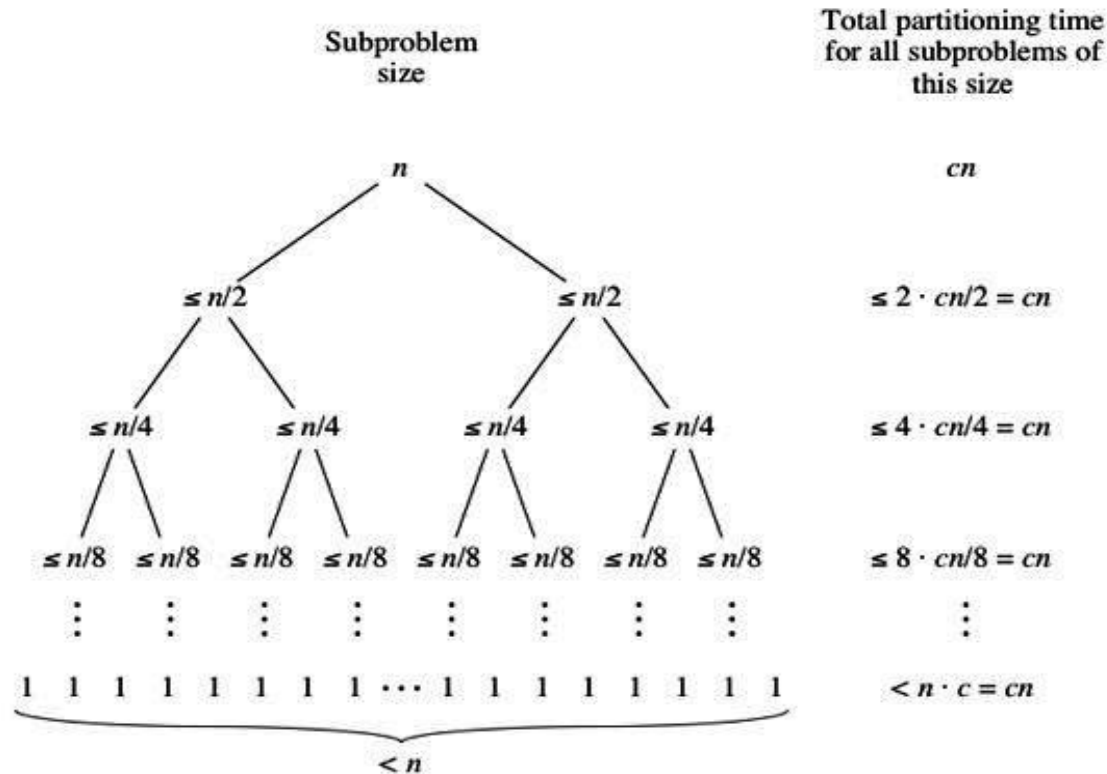
Assume  $n - k = 0$  therefore  $k = n$

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

$$T(n) = T(0) + 1 + 2 + 3 + \dots + (n-1) + n$$

$$T(n) = 1 + n(n+1)/2 = \Theta(n^2)$$

# Quick Sort Analysis (Best Case)



Best case occurs when partitioning the element array divides it into equal size sub arrays. The time required for comparison at each level is  $n$  and there are  $\log n$  levels (height of tree). Therefore time complexity is  $\Theta(n \log_2 n)$ .

# Quick Sort Analysis (Best Case)

Recurrence relation for quicksort in worst case

$$T(n) = \begin{cases} 1 & \text{if } n = 1 \\ 2T(n/2) + n & \text{if } n > 1 \end{cases}$$

$$T(n) = 2 T(n/2) + n \text{ ----- ( eq 1 )}$$

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

Put  $T(n/2)$  in eq 1 we have

$$\begin{aligned} T(n) &= 2 [2 T(n/2^2) + n/2] + n \\ &= 2^2 T(n/2^2) + n + n \\ &= 2^2 T(n/2^2) + 2n \text{ ----- (eq 2)} \end{aligned}$$

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

Put  $2 T(n/2^2)$  in eq 2 we have

$$\begin{aligned} T(n) &= 2^2 [2 T(n/2^3) + n/2^2] + 2n \\ &= 2^3 T(n/2^3) + n + 2n \\ &= 2^3 T(n/2^3) + 3n \text{ ----- (eq 3)} \end{aligned}$$

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

Solve  $T(n/2^k)$  till  $T(1)$

# Quick Sort Analysis (Best Case)

Recurrence relation for quicksort in worst case

$$T(n) = \begin{cases} 1 & \text{if } n = 1 \\ 2T(n/2) + n & \text{if } n > 1 \end{cases}$$

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

Solve  $T(n/2^k)$  till it becomes  $T(1)$

Therefore  $T(n/2^k) = T(1)$

$n/2^k = 1$ , therefore  $n = 2^k$ , **thus  $k = \log n$**

$$T(n) = 2^k T(1) + kn$$

$$= n * 1 + n \log n$$

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

# Practice Assignment

- Derive the time complexity of Merge sort using :
  1. Master theorem
  2. Using Substitution method

# References

1. Fundamentals Of Computer Algorithms

- E. Horowitz , S. Sahni , S. Rajasekaran

2. Fundamentals of Algorithm

- Gilles Brassard, Paul Bratley

3. Introduction to Algorithms

- T. H. Cormen , C. E. Leiserson, R. L. Rivest,  
Clifford Stein