



**PES**  
UNIVERSITY  
**ONLINE**

# **Design and Analysis of Algorithms**

---

**Bharathi R**

Department of Computer Science & Engineering

# DESIGN AND ANALYSIS OF ALGORITHMS-UE22CS241B

---

**Introduction to Algorithms,  
Design Techniques and Analysis**

Slides courtesy of **Anany Levitin**

**Bharathi R**

Department of Computer Science & Engineering

# Design and Analysis of Algorithms

## Syllabus

### ➤ **UNIT I (14 Hours)**

- Introduction
- Analysis of Algorithm Efficiency,
- Brute Force,

### ➤ **UNIT II (14 Hours)**

- Decrease-and-Conquer
- Divide-and-Conquer

### ➤ **UNIT III (16 Hours)**

- Transform-and-Conquer
- Space and Time Tradeoffs
- Greedy Technique

### ➤ **UNIT IV (12 Hours)**

- Limitations of Algorithm Power
- Coping with the Limitations of Algorithm Power
- Dynamic Programming

# **Design and Analysis of Algorithms**

## **Course Content: UE22CS241B : Design and Analysis of Algorithms**



### **Unit 1: Introduction and Brute Force**

Algorithms, Fundamentals of Algorithmic Problem Solving, Important Problem Types. Analysis of Algorithm Efficiency: Analysis Framework, Asymptotic Notations and Basic Efficiency Classes, Mathematical Analysis of Non Recursive and Recursive Algorithms. Brute Force: Selection Sort, Bubble Sort, Sequential Search, Brute Force String Matching, Exhaustive Search. **14 Hours**

### **Unit 2: Decrease – and – Conquer & Divide-and-Conquer**

Decrease-and-Conquer: Insertion Sort, Topological Sorting, Algorithms for Generating Combinatorial Objects, Decrease-by-a-Constant-Factor Algorithms. Divide-and-Conquer: Master Theorem, Merge Sort, Quick Sort, Binary Search, Binary Tree Traversals, Complexity analysis for finding the height of BST, Multiplication of Large Integers, Strassen's Matrix Multiplication. **14 Hours**

### **Unit 3: Transform-and-Conquer Space and Time Tradeoffs & Greedy Technique**

Transform and- Conquer: Pre-sorting, Heap Sort, Red-Black Trees, 2-3 Trees and Analysis of B Trees. Problems on - Decrease by a constant factor /constant number. Space and Time Tradeoffs: Sorting by Counting, Input Enhancement in String Matching - Horspool's and Boyer-Moore Algorithms. Greedy Technique: Prim's Algorithm, Kruskal's Algorithm and union-find algorithm, Dijkstra's Algorithm, Huffman Trees **16 Hours**

### **Unit 4: Limitations, Coping with the Limitations of Algorithm Power & Dynamic Programming,**

Limitations of Algorithm Power: Lower-Bound Arguments, Decision Trees, P, NP, and NP-Complete, NP-Hard Problems. Coping with the Limitations of Algorithm Power: Backtracking, Branch -and-Bound. Dynamic Programming: Computing a Binomial Coefficient, The Knapsack Problem and Memory Functions, Warshall's and Floyd's Algorithms. **12 Hours**

# Design and Analysis of Algorithms

## Text Books

Book Type	Code	Title & Author	Publication Information		
			Edition	Publisher	Year
Text Book	T1	Introduction to The Design and Analysis of Algorithms Anany Levitin	3	Pearson	2012
Reference Book	R1	Introduction to Algorithms Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein	3	Prentice-Hall India	2009
Reference Book	R2	Fundamentals of Computer Algorithms Horowitz, Sahni, Rajasekaran,	2	Universities Press	2007
Reference Book	R3	Algorithm Design Jon Kleinberg, Eva Tardos,	1	Pearson Education	2006

# Design and Analysis of Algorithms

## Evaluation Policy for the course



1	<b>Hands on Session</b>	<b>4 marks</b>
2	<b>Certification on Hacker Rank</b>	<b>4 marks</b>
3	Tutorial before ISA (one before each ISA)  Each Tutorial: 4 Questions to be solved in class  (2 subjective questions per unit )	<b>2 marks( all units)</b>
4	<b>ISA1 40 marks (Reduced to 20 marks)</b>	<b>20 marks</b>
5	<b>ISA2 40 marks (Reduced to 20 marks)</b>	<b>20 marks</b>
6	<b>ESA</b>	<b>50 marks</b>

**For Hackerrank**  
**Username: SRN**  
**Email id:**  
**Collegemailid**

## ALGORITHMS

"Algorithm" refers to a word which can be used by computer for the solution of problem.

The word algorithm comes from the name of Persian author Abu Ja'far Muhammad Ibh Musa Al-Khwarizmi, ninth century, who wrote a Text Book on Mathematics



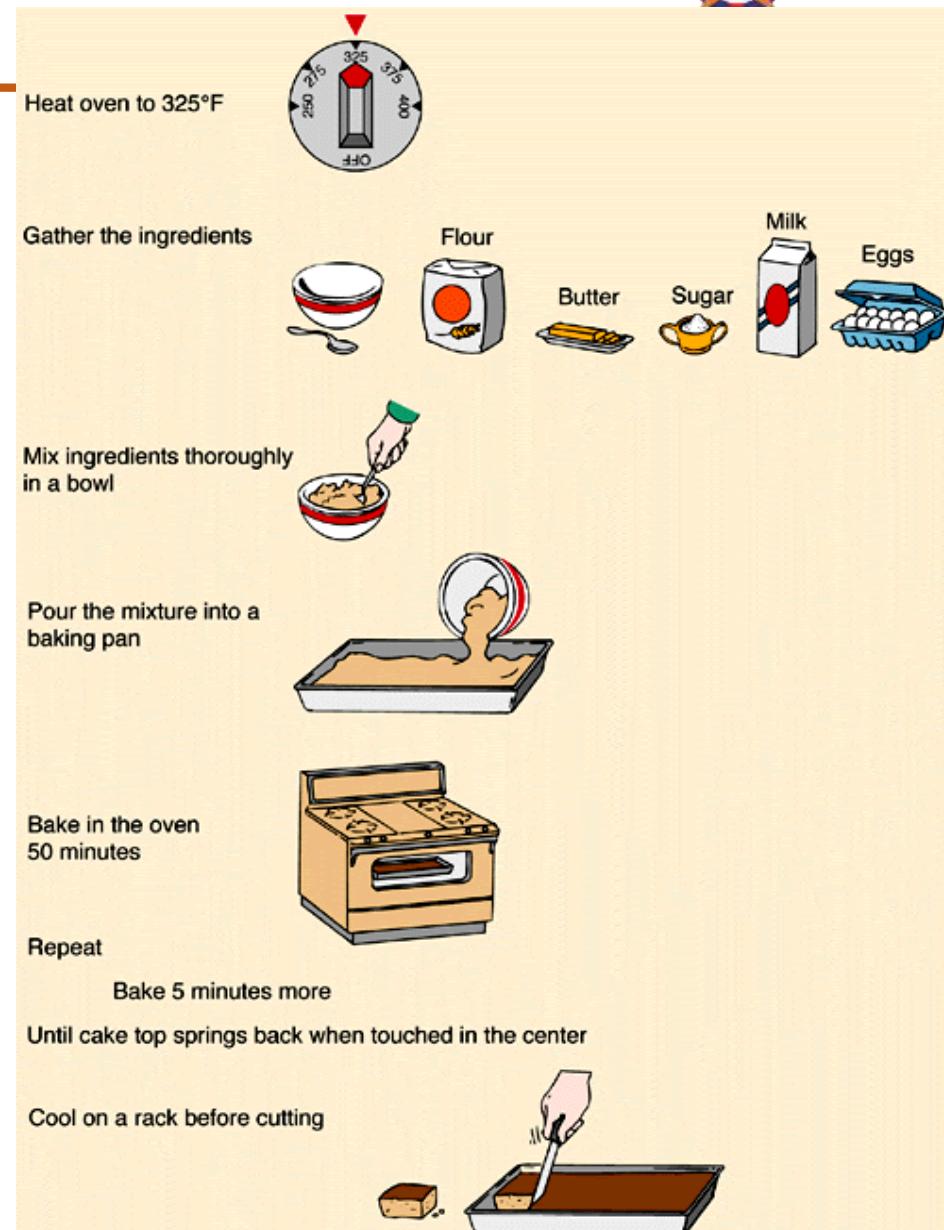
# Design and Analysis of Algorithms

## Algorithm



### What is an algorithm?

- An algorithm is a set of step-by-step procedures, or a set of rules to follow, for completing a specific task or solving a particular problem.
- Algorithms are all around us.
- The recipe for baking a cake, the method we use to solve a long division problem, and the process of doing laundry are all examples of an algorithm.



### What is an algorithm?

An algorithm is a sequence of **unambiguous instructions** for solving a problem, i.e., for obtaining a required output for any **legitimate input** in a **finite amount of time**

### Important Points about Algorithms

- The non-ambiguity requirement for each step of an algorithm cannot be compromised
- The range of inputs for which an algorithm works has to be specified carefully.
- The same algorithm can be implemented in several different ways
- There may exist several algorithms for solving the same problem.

Steve Jobs said,

*“Everyone should learn to program a computer, because it teaches you how to think.”*

Almost every part of your day-to-day life can be optimized and improved by developing a logical structure to save yourself time and get better results,

- 1) whether it's finding the best route to work,
- 2) getting the best parking space in the shortest amount of time,
- 3) doing your grocery shopping as quickly and cheaply as possible or
- A) getting the best deal on an airline ticket,

algorithms are powerful tools that teach you how to express your thoughts and solve seemingly unsolvable problems.



# Design and Analysis of Algorithms

## Algorithm and Program

Algorithm	Program
Design Stage	Implementation Stage
Domain Knowledge	Programmer
Written in any language	Written using Programming Langauge
H/W and Operating system independent	H/W and Operating system dependent
Analysis of Algorithm-(Time and Space Complexity)	Testing of Programs ( Time taken and Memory consumed in terms of bytes)

- 
1. **Input:** Zero or more quantities are externally supplied
  2. **Definiteness:** Each instruction is clear and unambiguous
  3. **Finiteness:** The algorithm terminates in a finite number of steps.
  4. **Effectiveness:** Each instruction must be primitive and feasible
  5. **Output:** At least one quantity is produced

### Why do we need Algorithms?

- It is a tool for solving well-specified Computational Problem.
- Problem statement specifies in general terms relation between input and output
- Algorithm describes computational procedure for achieving input/output relationship  
This Procedure is irrespective of implementation details

### Why do we need to study algorithms?

Exposure to different algorithms for solving various problems helps develop skills to design algorithms for the problems for which there are no published algorithms to solve it

- How to **design** algorithms
- How to **express** algorithms
- Proving **correctness** of designed algorithm
- Efficiency
  - Theoretical analysis
  - Empirical analysis

“efficient”

# Algorithms

“learn to be clever”

Input:

A number  $x$

$x^{21} = x^{16} * x^4 * x$  Store  $x$

Compute and store  $x^2 = x * x$

Compute and store  $x^4 = x^2 * x^2$

Compute and store  $x^8 = x^4 * x^4$

Compute and store  $x^{16} = x^8 * x^8$

Output:

The number  $x^{21}$

Hey, I know that!  
6 multiplications  
And I am done!

Compute and output

$x^{21} = x^{16} * x^4 * x$

$$\underline{x^{20}} = x^8 * x^8$$
$$x^{21} = x^8 * x^8 * x$$

### What do you mean by Algorithm Design Techniques?

General Approach to solving problems algorithmically .

Applicable to a variety of problems from different areas of computing

### Various Algorithm Design Techniques

- 1.➤ Brute Force
- 2.➤ Divide and Conquer
- 3.➤ Decrease and Conquer
- 4.➤ Transform and Conquer
- 5.➤ Dynamic Programming
- 6.➤ Greedy Technique
- 7.➤ Branch and Bound
- 8.➤ Backtracking

**Importance** Framework for designing and analyzing algorithms for new problems

- How to design Algorithm
- How to express algorithms
- Proving correctness of designed algroithm
- Efficiency
  - Theoretical Analysis
  - Empirical Analysis

- **Natural language**
  - Ambiguous
- **Pseudocode**
  - A mixture of a natural language and programming language-like structures
  - Precise and succinct.
  - Pseudocode in this course
    - omits declarations of variables
    - use indentation to show the scope of such statements as for, if, and while.
    - use  $\leftarrow$  for assignment
- **Flowchart**
  - Method of expressing algorithm by collection of connected geometric shapes

### ➤ Euclid's Algorithm

Problem: Find  $\text{gcd}(m,n)$ , the greatest common divisor of two nonnegative, not both zero integers  $m$  and  $n$

Examples:  $\text{gcd}(60,24) = 12$ ,  $\text{gcd}(60,0) = 60$ ,  $\text{gcd}(0,0) = ?$

Euclid's algorithm is based on repeated application of equality

$$\text{gcd}(m,n) = \text{gcd}(n, m \bmod n)$$

until the second number becomes 0, which makes the problem trivial.

Example:  $\text{gcd}(60,24) = \text{gcd}(24,12) = \text{gcd}(12,0) = 12$

### Two descriptions of Euclid's algorithm

Euclid's algorithm for computing  $\text{gcd}(m,n)$

Step 1 If  $n = 0$ , return  $m$  and stop; otherwise go to Step 2

Step 2 Divide  $m$  by  $n$  and assign the value of the remainder to  $r$

Step 3 Assign the value of  $n$  to  $m$  and the value of  $r$  to  $n$ . Go to step 1.

ALGORITHM Euclid( $m,n$ )

//computes  $\text{gcd}(m,n)$  by Euclid's method

//Input: Two nonnegative, not both zero integers

//Output:Greatest common divisor of  $m$  and  $n$

while  $n \neq 0$  do

$r \leftarrow m \bmod n$

$m \leftarrow n$

$n \leftarrow r$

return  $m$

- One of the most straightforward and elementary searches is the sequential search, also known as a linear search.
- As a real world example, pickup the nearest phonebook and open it to the first page of names. We're looking to find the first "Smith".
- Look at the first name. Is it "Smith"? Probably not (it's probably a name that begins with 'A').
- Now look at the next name. Is it "Smith"? Probably not.
- Keep looking at the next name until you find "Smith".

Instead of a phonebook, we have an array. Although the array can hold data elements of any type, for the simplicity of an example we'll just use an array of integers, like the following:

10	7	1	3	-4	2	20
----	---	---	---	----	---	----

Lets search for the number 3. We start at the beginning and check the first element in the array. Is it 3?

3?

10	7	1	3	-4	2	20
----	---	---	---	----	---	----

No, not it. Is it the next element?

3?

10	7	1	3	-4	2	20
----	---	---	---	----	---	----

Not there either. Next?

3?

We found it!!!

10	7	1	3	-4	2	20
----	---	---	---	----	---	----

### ALGORITHM *SequentialSearch*( $A[0..n - 1]$ , $K$ )

//Searches for a given value in a given array by sequential search

//Input: An array  $A[0..n - 1]$  and a search key  $K$

//Output: The index of the first element of  $A$  that matches  $K$

// or  $-1$  if there are no matching elements

$i \leftarrow 0$

**while**  $i < n$  **and**  $A[i] \neq K$  **do**

$i \leftarrow i + 1$

**if**  $i < n$  **return**  $i$

**else return**  $-1$

can we have just one condition in the loop?

can we return a value anything other than  $0 .. n-1$  if the element is not found?

## Need of Analysis

---

- To determine resource consumption
  - CPU time
  - Memory space
- Compare different methods for solving the same problem before actually implementing them and running the programs.
- To find an efficient algorithm for solving the problem

- A measure of the performance of an algorithm
- An algorithm's performance is characterized by
  - **Time complexity**  
How fast an algorithm maps input to output as a function of input
  - **Space complexity**  
amount of memory units required by the algorithm in addition to the memory needed for its input and output

### How to determine complexity of an algorithm?

- Experimental study(Performance Measurement)
- Theoretical Analysis (Performance Analysis)

- It is necessary to implement the algorithm, which may be difficult
- Results may not be indicative of the running time on other inputs not included in the experiment.
- In order to compare two algorithms, the same hardware and software environments must be used
- Experimental data though important is not sufficient

- Uses a high-level description of the algorithm instead of an implementation
- Characterizes running time as a function of the input size,  $n$ .
- Takes into account all possible inputs
- Allows us to evaluate the speed of an algorithm independent of the hardware/software environment

# Design and Analysis of Algorithms

## Exercise

---



- Design an algorithm for computing  $\text{gcd}(m,n)$  using Euclid's algorithm



**THANK YOU**

---

**Bharathi R**

Department of Computer Science & Engineering

**rbharathi@pes.edu**



# **Design and Analysis of Algorithms**

---

**Bharathi R**

Department of Computer Science & Engineering

# DESIGN AND ANALYSIS OF ALGORITHMS

---

## Fundamentals of Algorithmic Problem Solving

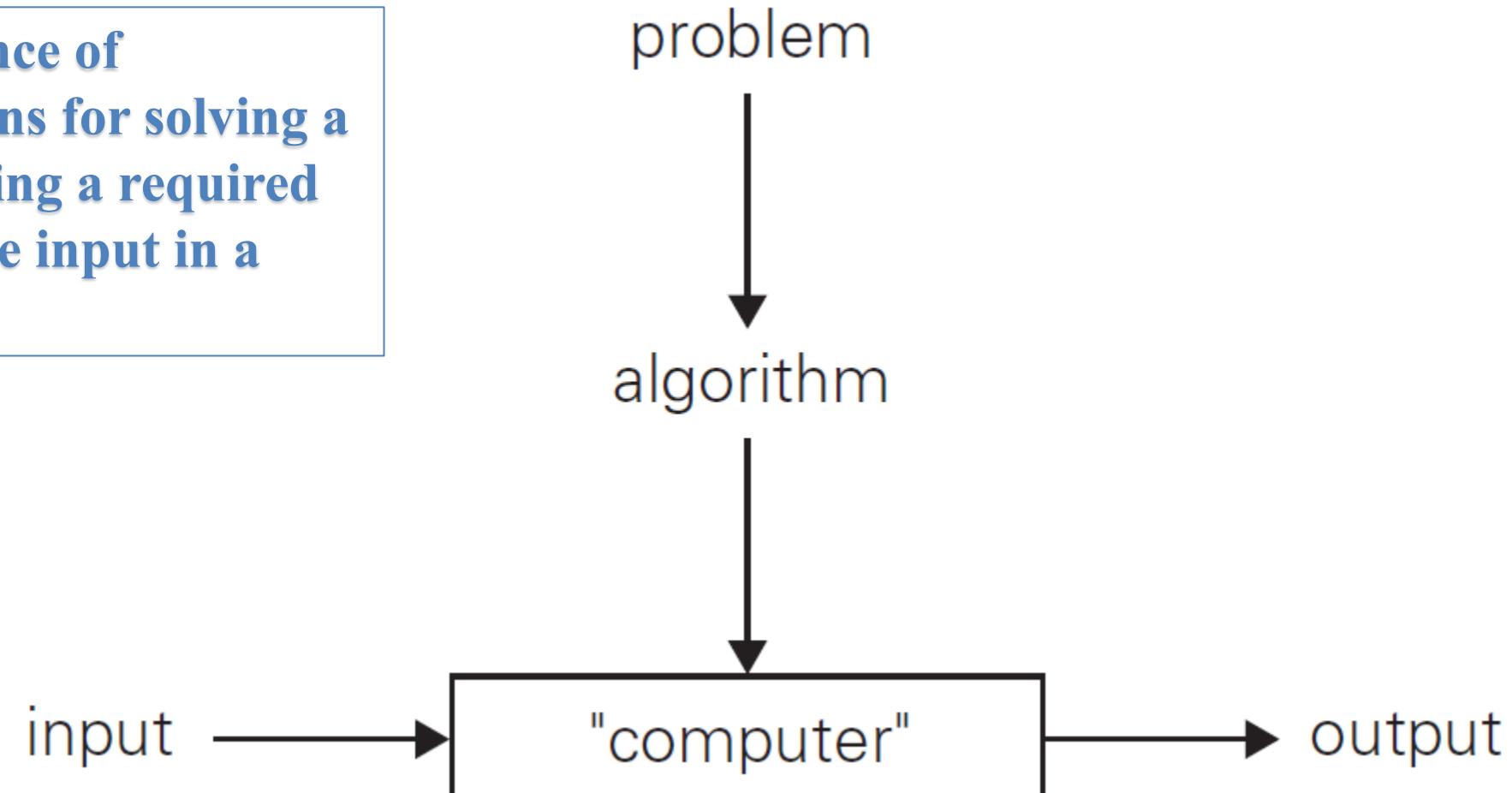
Slides courtesy of **Anany Levitin**

**Bharathi R**

Department of Computer Science & Engineering

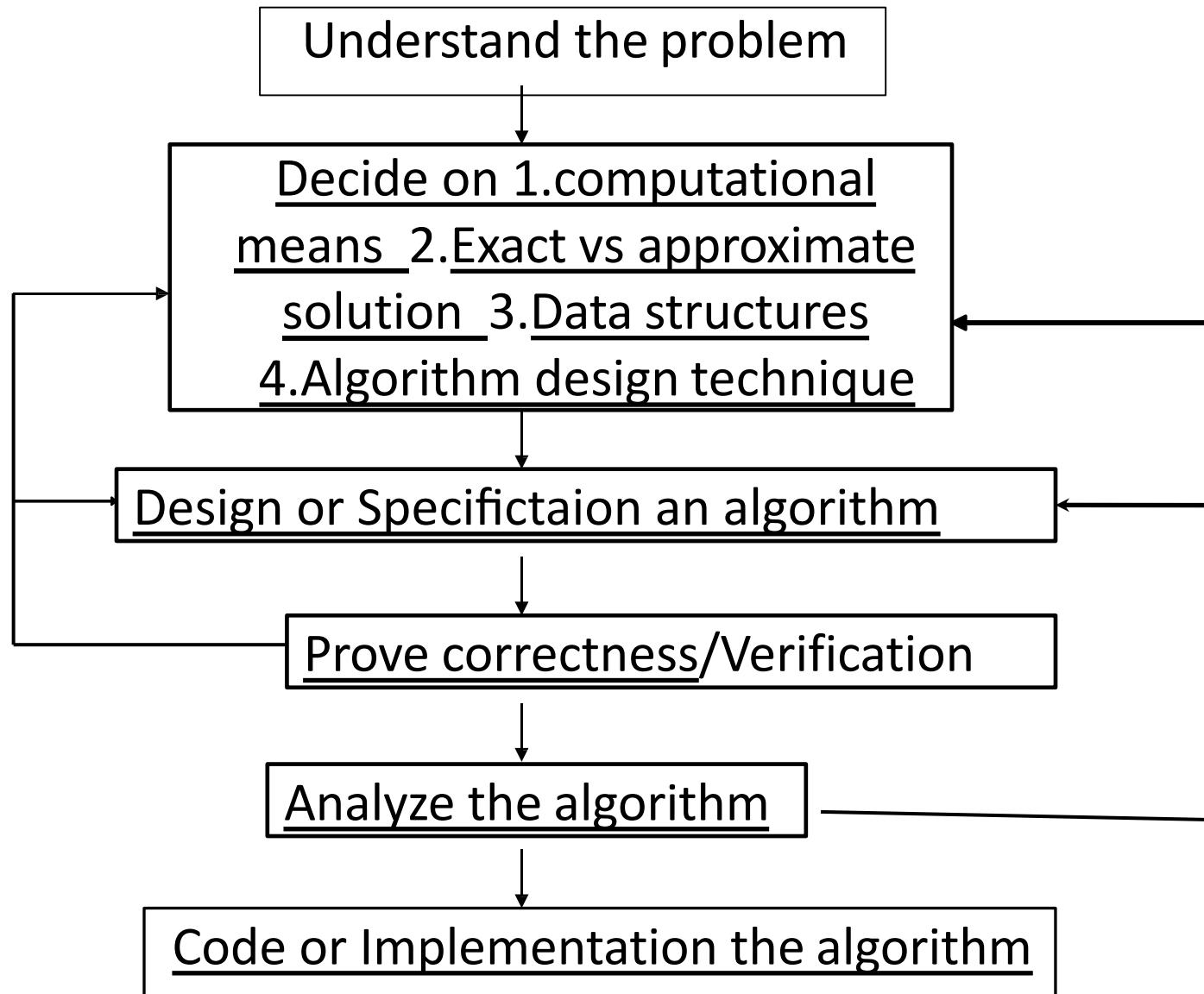
## What is an Algorithm formally then?

An *algorithm* is a sequence of unambiguous instructions for solving a problem, i.e., for obtaining a required output for any legitimate input in a finite amount of time.



# Design and Analysis of Algorithms

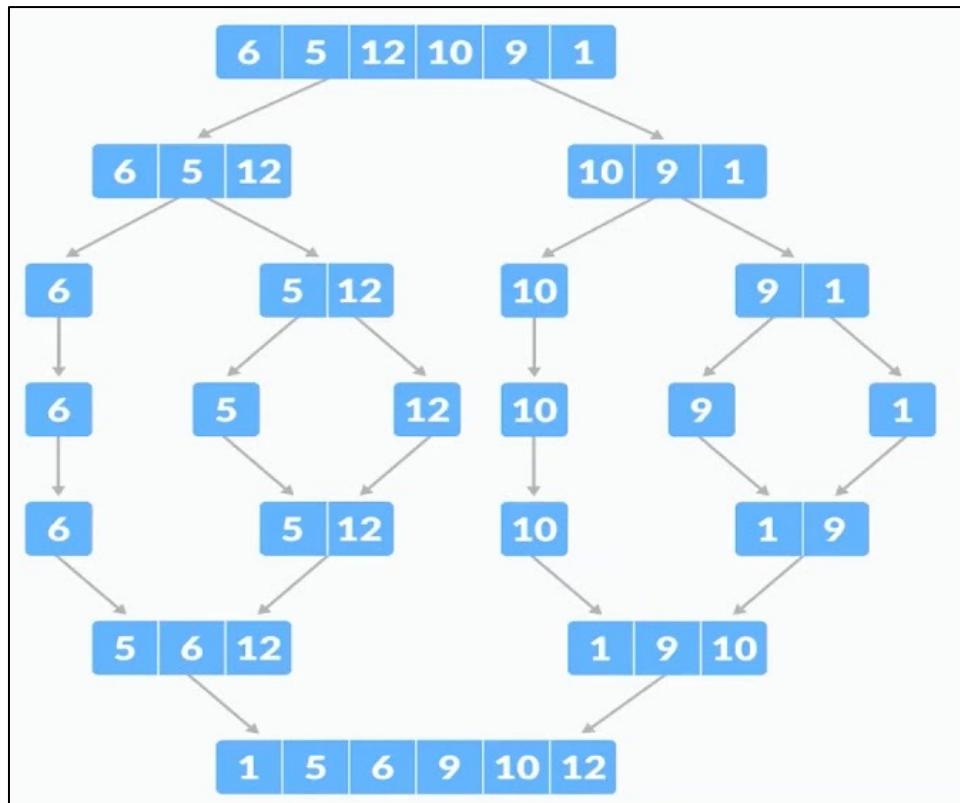
## Algorithm Design and Analysis Process



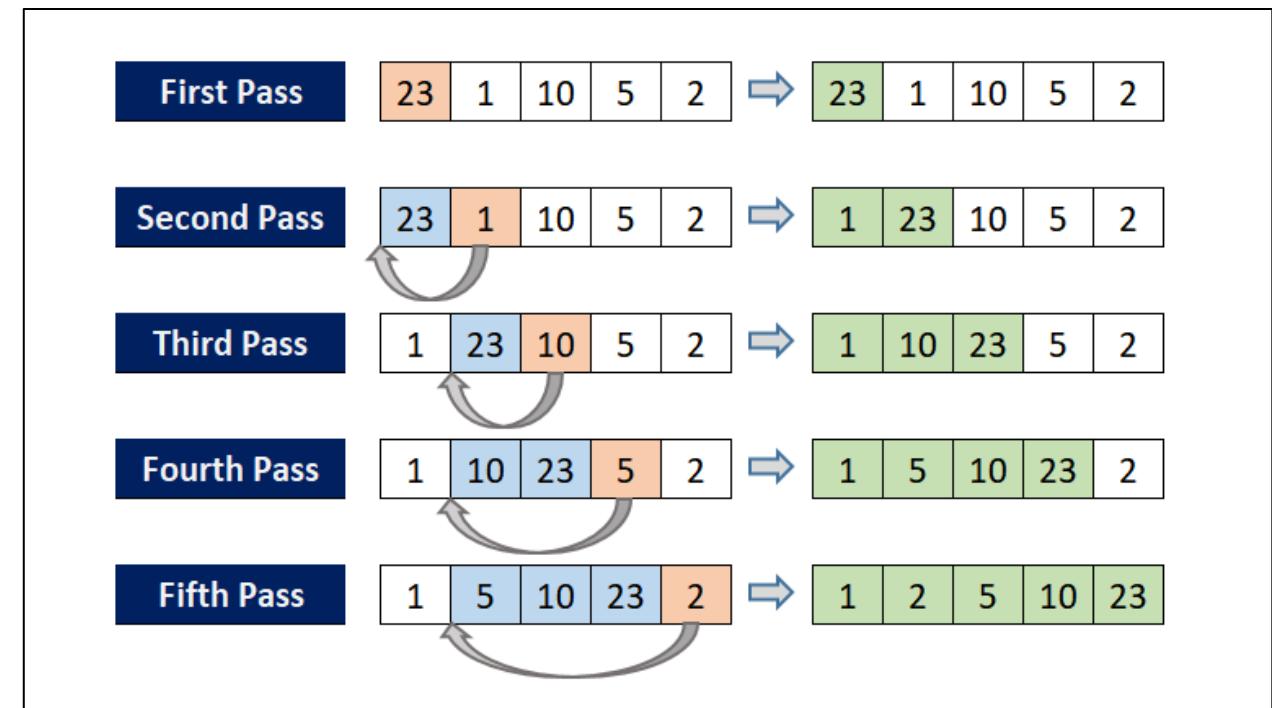
# Algorithmic Problem Solving

## Asserting the Capabilities of a computational Device

Merge Sort  $(C_2 n \lg n)$



Insertion Sort  $(C_1 n^2)$



# Algorithmic Problem Solving

## Asserting the Capabilities of a computational Device

n	n*n	nlogn
"1000"	1000000	9965.784285
"1,000,000"	1E+12	19931568.57
"1,000,000,000,000"	1E+24	3.98631E+13

Computational Time  
(in micro seconds)  
for input size

Insertion Sort $(C_1 n^2)$	Merge Sort $(C_2 n \lg n)$
Hardware: Computer A Executes 10 Billion Instructions/second	Hardware: Computer B Executes 10 million Instructions/second
Programmed by the world craftiest programmer	Programmed by average programmer
$C_1 = 2$	$C_2 = 50$
Input Size = 10,000,000	Input Size = 10,000,000
$\frac{2 \times (10^7)^2 \text{ instructions}}{10^{10} \text{ instructions/second}} = 20,000 \text{ seconds}$	$\frac{5 \times (10^7) \lg 10^7 \text{ instructions}}{10^7 \text{ instructions/second}} \approx 1163 \text{ seconds} < 20 \text{ s}$

Computational Device the algorithm is intended for

RAM Sequential Algorithms

PRAM Parallel Algorithms

## Exact vs approximate solution

---

The next principal decision is to choose between solving the problem **exactly** or solving it **approximately**.

In the former case, an algorithm is called an **exact algorithm**;

in the latter case, an algorithm is called an **approximation algorithm**

Example :Travelling Salesman Problem NP

complete!!!

Approximate algorithm can be used to solve it

- Linear

- Linear list, Stack, Queues

- Non Linear

- Trees,

- Graphs

Choice of Data structure for solving a problem using an algorithm may dramatically impact its time complexity

Dijkstra Algorithm

$O(V \log V + E)$  with Fibonacci heap

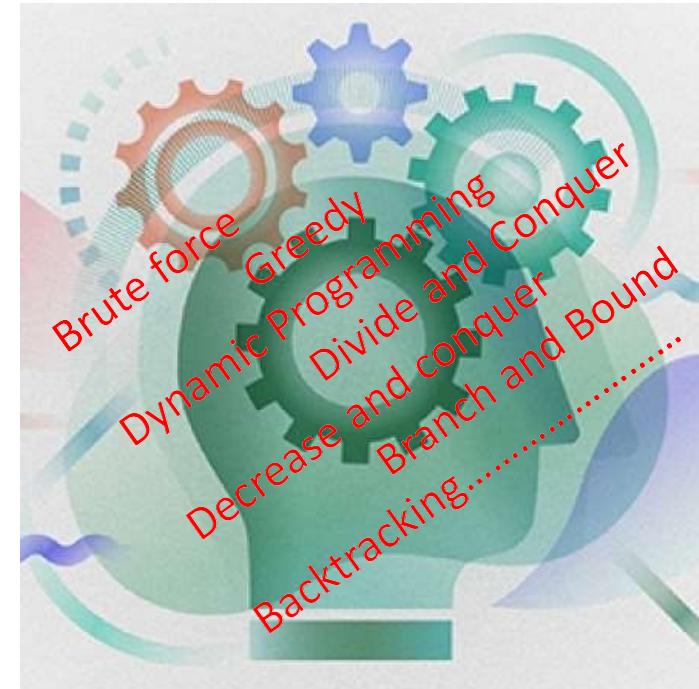
An algorithm design technique (or “strategy” or “paradigm”) is a general approach to solving problems algorithmically that is applicable to a variety of problems from different areas of computing.

General approach to solving problems algorithmically that is applicable to variety of problems from different areas of computing

ADT serves as heuristic for designing algorithms for new problems for which

no satisfactory algorithm exists!!!

Algorithm Designer’s Toolkit



## Specifying an algorithm

---

- Natural Language
- Pseudo Code
- Flowchart

Once an algorithm has been specified, you have to prove its correctness.

That is, you have to prove that the algorithm yields a required result for every legitimate input in a finite amount of time.

For example, the correctness of Euclid's algorithm : As follows

### Exact algorithms

Proving that algorithm yields a correct result for legitimate input in finite amount of time

### Approximation algorithms

Error produced by algorithm does not exceed a predefined limit

## Algorithm Correctness

- For example, the correctness of Euclid's algorithm : As follows
- Simple observation that the second integer gets smaller on every iteration of the algorithm, and the fact that the algorithm stops when the second integer becomes 0 in a  $\text{gcd}(m,n)$  algorithm can be a proof of algorithms correctness.
- Use mathematical induction because an algorithm's iterations provide a natural sequence of steps needed for such proofs
- Tracing the algorithm's performance for a few specific inputs can be a very worthwhile activity, it cannot prove the algorithm's correctness conclusively
- For an approximation algorithm, we usually would like to be able to show that the error produced by the algorithm does not exceed a predefined limit.

## Analyzing an algorithm

---

- Efficiency
  - Time efficiency
  - Space efficiency
- Simplicity
- Generality
  - Design an algorithm for the problem posed in more general terms
  - Design an algorithm that can handle a range of inputs that is natural for the problem at hand

# Algorithmic Problem Solving

## Algorithm Analysis

---

For each function  $f(n)$  and time  $t$  in the following table, determine the largest size  $n$  of a problem that can be solved in time  $t$ , assuming that the algorithm to solve the problem takes  $f(n)$  microseconds.

	Second	Minute	Hour	Day	Month	Year	Century
$\log(n)$							
$\sqrt{n}$							
$n$							
$n \log n$							
$n^2$							
$n^3$							
$2^n$							
$n!$							

# Algorithmic Problem Solving

## Algorithm Analysis

For each function  $f(n)$  and time  $t$  in the following table, determine the largest size  $n$  of a problem that can be solved in time  $t$ , assuming that the algorithm to solve the problem takes  $f(n)$  microseconds.

	Second	Minute	Hour	Day	Month	Year	Century
$\log(n)$							
$\sqrt{n}$							
$n$	1000000	60000000	3600000000	864000000000	2.592E+12	3.1536E+13	3.1536E+15
$n \log n$							
$n^2$							
$n^3$							
$2^n$							
$n!$							

- Efficient implementation
- Correctness of program
  - Mathematical Approach: Formal verification for small programs
  - Practical Methods: Testing and Debugging
- Code optimization

**Write an algorithm for finding the distance between the two closest elements in an array of numbers.**

Consider the following algorithm for finding the distance between the two closest elements in an array of numbers.

**ALGORITHM** *MinDistance(A[0..n - 1])*

//Input: Array A[0..n - 1] of numbers

//Output: Minimum distance between two of its elements

*dmin*  $\leftarrow \infty$

**for** *i*  $\leftarrow 0$  to *n* - 1 **do**

**for** *j*  $\leftarrow 0$  to *n*-1 **do**

**if** *i*  $\neq$  *j* and |A[i]-A[j]| < *dmin*

*dmin*  $\leftarrow$  |A[i]-A[j]|

**return** *dmin*

Make as many improvements as you can in this algorithmic solution to the problem. If you need to, you may change the algorithm altogether; if not, improve the implementation given.

You can:

decrease the number of times the innermost loop is executed, make that loop run faster (at least for some inputs), or, more significantly, design a faster algorithm from scratch.



**THANK YOU**

---

**Bharathi R**

Department of Computer Science & Engineering

[rbharathi@pes.edu](mailto:rbharathi@pes.edu)



# **Design and Analysis of Algorithms**

---

**Bharathi R**

Department of Computer Science & Engineering

# DESIGN AND ANALYSIS OF ALGORITHMS

---

## Important Problem Types

Slides courtesy of Anany Levitin

**Bharathi R**

Department of Computer Science & Engineering

1. Sorting
2. Searching
3. String processing
4. Graph problems
5. Combinatorial problems
6. Geometric problems
7. Numerical problems

## 1. Important Problem Types: Sorting

- Rearrange the items of a given list in ascending order.
  - Input: A sequence of n numbers  $\langle a_1, a_2, \dots, a_n \rangle$
  - Output: A reordering  $\langle a'_1, a'_2, \dots, a'_n \rangle$  of the input sequence such that  $a'_1 \leq a'_2 \leq \dots \leq a'_n$ .
- Why sorting?
  - Help searching
  - Algorithms often use sorting as a key subroutine.
- Sorting key

A specially chosen piece of information used to guide sorting.  
Example: sort student records by SRN.

### Sorting Algorithms



## 1. Important Problem Types:

### Sorting

---

- Rearrange the items of a given list in ascending order.
- Examples of sorting algorithms
  - Selection sort
  - Bubble sort
  - Insertion sort
  - Merge sort
  - Heap sort ...
- Evaluate sorting algorithm complexity: the number of key comparisons.
- Two properties
  - **Stability**: A sorting algorithm is called stable if it preserves the relative order of any two equal elements in its input.
  - **In place** : A sorting algorithm is in place if it does not require extra memory, except, possibly for a few memory units.

## 2. Important Problem Types: Searching

---

Find a given value, called a **search key**, in a given set. Examples of searching algorithms

- Sequential searching
- Binary searching...

[BACK](#)



## 3. Important Problem Types: String Processing

---

A string is a sequence of characters from an alphabet.

Text strings: letters, numbers, and special characters.

String matching: searching for a given word/pattern in a text.

**Text:** I am a **computer** science graduate

**Pattern:** computer

## 4. Important Problem Types: Graph Problems

### Definition

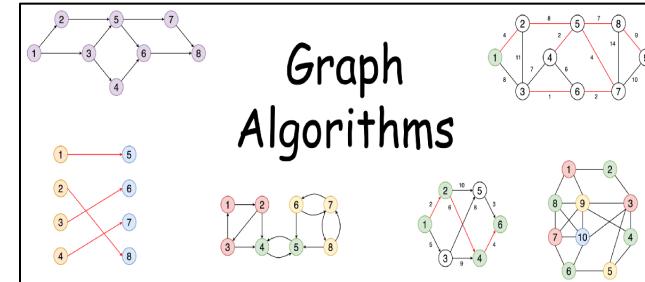
Graph G is represented as a pair  $G = (V, E)$ ,  
where V is a finite set of vertices and E is a finite set of edges

### Modeling real-life problems

- Modeling WWW
- communication networks
- Project scheduling ...

### Examples of graph algorithms

- Graph traversal algorithms
- Shortest-path algorithms
- Topological sorting



## 5. Important Problem Types: Combinatorial Problems

- The traveling salesman problem and the graph-coloring problem are examples of **combinatorial problems**.
- These are problems that ask, explicitly or implicitly, to find a combinatorial object—such as a permutation, a combination, or a subset—that satisfies certain constraints.
- Generally speaking, combinatorial problems are the most difficult problems in computing, from both a theoretical and practical standpoint.

Their difficulty stems from the following facts:

1. First, the number of combinatorial objects typically grows extremely fast with a problem's size, reaching unimaginable magnitudes even for moderate-sized instances.
2. Second, there are no known algorithms for solving most such problems exactly in an acceptable amount of time.

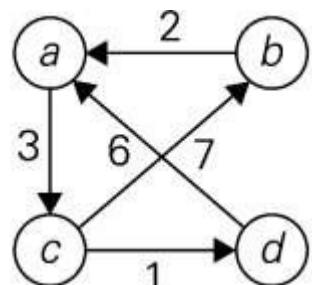


## 5. Important Problem Types: Combinatorial Problems

Some of the combinatorial problems can be solved:

Shortest paths in a graph

To find the distances from each vertex to all other vertices.



(a)

$$W = \begin{bmatrix} a & b & c & d \\ a & 0 & \infty & 3 & \infty \\ b & 2 & 0 & \infty & \infty \\ c & \infty & 7 & 0 & 1 \\ d & 6 & \infty & \infty & 0 \end{bmatrix}$$

(b)

$$D = \begin{bmatrix} a & b & c & d \\ a & 0 & 10 & 3 & 4 \\ b & 2 & 0 & 5 & 6 \\ c & 7 & 7 & 0 & 1 \\ d & 6 & 16 & 9 & 0 \end{bmatrix}$$

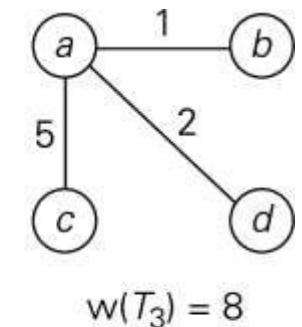
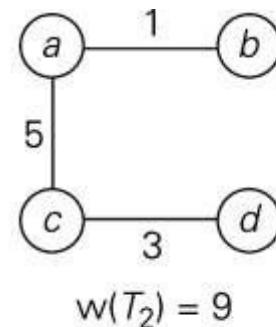
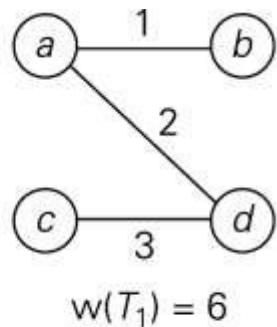
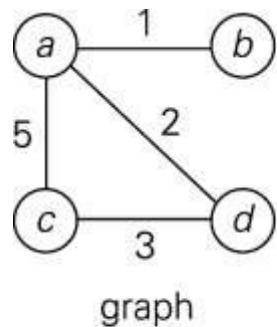
(c)

**FIGURE 8.5** (a) Digraph. (b) Its weight matrix. (c) Its distance matrix.

## 5. Important Problem Types: Combinatorial Problems

### Minimum cost spanning tree

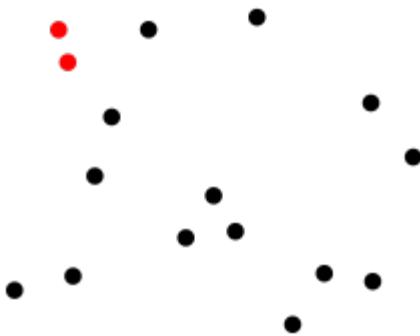
- A spanning tree of a connected graph is its connected acyclic sub graph (i.e. a tree).



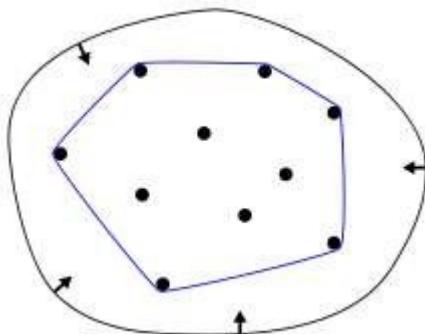
**FIGURE 9.1** Graph and its spanning trees;  $T_1$  is the minimum spanning tree

## 6. Important Problem Types: Geometric Problems

### Closest Pair problem



### Convex Hull Problem



- **Geometric algorithms** deal with geometric objects such as points, lines, and polygons.
- Today people are interested in geometric algorithms with quite different applications in mind, such as computer graphics, robotics.
- The **closest-pair problem** is self-explanatory: given n points in the plane, find the closest pair among them.
- The **convex-hull problem** asks to find the smallest convex polygon that would include all the points of a given set.

## 7. Important Problem Types: Numerical Problems

---

- Solving Equations
- Computing definite integrals
- Evaluating functions

# **Design and Analysis of Algorithms**

## **Fundamental Data Structures**

---



### **Linear Data Structures**

- 1. Arrays**
- 2. Strings**
- 3. Linked list**
- 4. Stack**
- 5. Queue**

### **Graphs**

### **Trees**

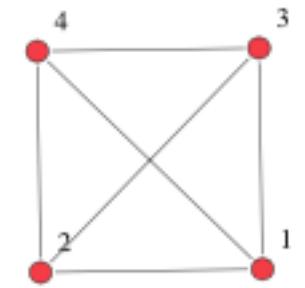
### **Sets and Dictionaries**

- a. Let  $A$  be the adjacency matrix of an undirected graph. Explain what property of the matrix indicates that
- i. the graph is complete.
  - ii. the graph has a loop, i.e., an edge connecting a vertex to itself.
  - iii. the graph has an isolated vertex, i.e., a vertex with no edges incident to it.
- b. Answer the same questions for the adjacency list representation.

- a. Let A be the adjacency matrix of an undirected graph. Explain what property of the matrix indicates that
- i. the graph is complete.

For the adjacency matrix representation:

- i. A graph is complete if and only if all the elements of its adjacency matrix except those on the main diagonal are equal to 1, i.e.,  $A[i, j] = 1$  for every  $1 \leq i, j \leq n, i \neq j$ .



$$\begin{pmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{pmatrix}$$

a. Let A be the adjacency matrix of an undirected graph. Explain what property of the matrix indicates that

ii. the graph has a loop, i.e., an edge connecting a vertex to itself.

A graph has a loop if and only if its adjacency matrix has an element equal to 1 on its main diagonal, i.e.,  $A[i,i] = 1$  for some  $1 \leq i \leq n$ .

- a. Let  $A$  be the adjacency matrix of an undirected graph. Explain what property of the matrix indicates that
- iii. the graph has an isolated vertex, i.e., a vertex with no edges incident to it.

An (undirected, without loops) graph has an isolated vertex if and only if its adjacency matrix has an all-zero row.

- a. Let  $A$  be the adjacency matrix of an undirected graph. Explain what property of the matrix indicates that
- i. the graph is complete.
  - ii. the graph has a loop, i.e., an edge connecting a vertex to itself.
  - iii. the graph has an isolated vertex, i.e., a vertex with no edges incident to it.
- b. Answer the same questions for the adjacency list representation.

- b. Answer the same questions for the adjacency list representation.
- i. A graph is complete if and only if each of its linked lists contains all the other vertices of the graph.
  - ii. A graph has a loop if and only if one of its adjacency lists contains the 24 vertex defining the list.
  - iii. An (undirected, without loops) graph has an isolated vertex if and only if one of its adjacency lists is empty.



THANK  
YOU

---

Bharathi R

Department of Computer Science & Engineering

[rbharathi@pes.edu](mailto:rbharathi@pes.edu)

Slides courtesy of Anany Levitin



**PES**  
UNIVERSITY  
**ONLINE**

## **Design and Analysis of Algorithms**

---

**Bharathi R**

Department of Computer Science & Engineering

# DESIGN AND ANALYSIS OF ALGORITHMS

---

## Analysis Framework

Slides courtesy of **Anany Levitin**

**Bharathi R**

Department of Computer Science & Engineering

### What do you mean by analysing an algorithm?

Investigation of Algorithm's efficiency with respect to two resources

- Time
- Space

### What is the need for Analysing an algorithm?

- To determine resource consumption
  - CPU time
  - Memory space
- Compare different methods for solving the same problem before actually implementing them and running the programs.
- To find an efficient algorithm

- A measure of the performance of an algorithm
- An algorithm's performance depends on
  - *internal factors*
    - Time required to run
    - Space (memory storage) required to run
  - *external factors*
    - Speed of the computer on which it is run
    - Quality of the compiler
    - Size of the input to the algorithm

## Algorithm's efficiency/ Performance Analysis

---

### 1. Space Complexity

Space complexity of an algorithm is the amount of memory it needs to run to completion.

### 2. Time Complexity

Time complexity of an algorithm is the amount of computation time it needs to run to completion.

$$S(P) = C + SP(I)$$

- Fixed Space Requirements (C)  
**Independent of the characteristics of the inputs and outputs**
  - instruction space
  - space for simple variables, fixed-size structured variable, constants
- Variable Space Requirements (SP(I))  
**dependent on the instance characteristic I**
  - number, size, values of inputs and outputs associated with I
  - recursive stack space, formal parameters, local variables, return address

## Example : 1 for Space Complexity

---

```
1 Algorithm abc(a,b,c)
2 {
3     return a + b + b * c + (a + b - c)/(a + b) + 4.0;
4 }
```

---

Algorithm 1.5 Computes  $a + b + b * c + (a + b - c)/(a + b) + 4.0$

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

$$c = ?$$

$$S_P = 0$$

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

```
float rsum(float list[ ], int n)
{
    if (n)
        return rsum(list, n-1) + list[n]
    return 0
}
```

$$S_{\text{sum}}(I)=S_{\text{sum}}(n)=6n$$

Type	Name	Number of bytes
parameter: float	list [ ]	2
parameter: integer	n	2
return address:(used internally)		2
TOTAL per recursive call		6

## 2. Time Complexity

---

$$T(n) = c \cdot C(n)$$

## Time Complexity

---

$$T(P) = C + T_P(I)$$

- Compile time ( $C$ )  
independent of instance characteristics
  
- run (execution) time  $T_P$

How to measure time complexity?

- Theoretical Analysis
- Experimental study

### Experimental study

- Write a program implementing the algorithm
- Run the program with inputs of varying size and composition
- Get an accurate measure of the actual running time
- Use a method like `System.currentTimeMillis()`
- Plot the results

- It is necessary to implement the algorithm, which may be difficult
- Results may not be indicative of the running time on other inputs not included in the experiment.
- In order to compare two algorithms, the same hardware and software environments must be used
- Experimental data though important is not sufficient

- Uses a high-level description of the algorithm instead of an implementation
- Characterizes running time as a function of the input size,  $n$ .
- Takes into account all possible inputs
- Allows us to evaluate the speed of an algorithm independent of the hardware/software environment

Two approaches:

### 1. Order of magnitude/asymptotic categorization –

This uses coarse categories and gives a general idea of performance.

If algorithms fall into the same category, if data size is small, or if performance is critical, use method 2

### 2. Estimation of running time -

1. *operation counts* - select operation(s) that are executed most frequently and determine how many times each is done.
2. *step counts* - determine the total number of steps, possibly lines of code, executed by the program.

- Measuring an input's size
- Measuring running time
- Orders of growth (of the algorithm's efficiency function)
- Worst-base, best-case and average efficiency

## Analysis framework

---

Analysis framework is a systematic approach of analyzing the performance of algorithm

It is based on certain factors:

1. Measuring Time complexity and Space complexity
2. Measuring Input size
3. Measuring Running time
4. Computing Order of growth
5. Finding best case, worst case and average case analysis.

➤ Measure the running time using standard unit of time measurements, such as seconds, minutes?

Depends on the speed of the computer.

➤ count the number of times each of an algorithm's operations is executed.

(step count method)

Difficult and unnecessary

➤ count the number of times an algorithm's basic operation is executed.

**Basic operation:** the most important operation of the algorithm, the operation contributing the most to the total running time.

For example, the basic operation is usually the most time-consuming operation in the algorithm's innermost loop.

## Measuring Running Time: Step Count Method

### Analysis in the RAM Model

SmartFibonacci( $n$ )	<i>cost</i>	<i>times</i> ( $n > 1$ )
1 <b>if</b> $n = 0$	$c_1$	1
2 <b>then return</b> 0	$c_2$	0
3 <b>elseif</b> $n = 1$	$c_3$	1
4 <b>then return</b> 1	$c_4$	0
5 <b>else</b> $p\text{prev} \leftarrow 0$	$c_5$	1
6 $\text{prev} \leftarrow 1$	$c_6$	1
7 <b>for</b> $i \leftarrow 2$ <b>to</b> $n$	$c_7$	$n$
8 <b>do</b> $f \leftarrow \text{prev} + p\text{prev}$	$c_8$	$n - 1$
9 $p\text{prev} \leftarrow \text{prev}$	$c_9$	$n - 1$
10 $\text{prev} \leftarrow f$	$c_{10}$	$n - 1$
11 <b>return</b> $f$	$c_{11}$	1

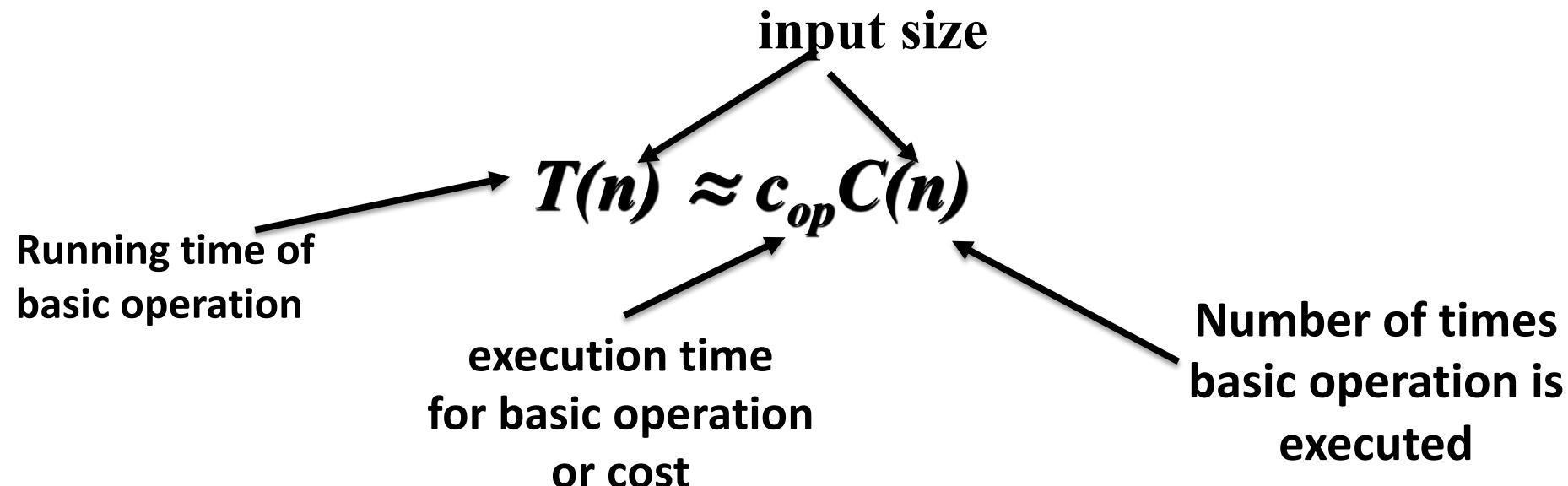
$$T(n) = c_1 + c_3 + c_5 + c_6 + c_{11} + nc_7 + (n - 1)(c_8 + c_9 + c_{10})$$

$T(n) = nC_1 + C_2 \Rightarrow T(n)$  is a *linear function* of  $n$

## Theoretical analysis of time efficiency

Time efficiency is analyzed by determining the number of repetitions of the basic operation as a function of input size

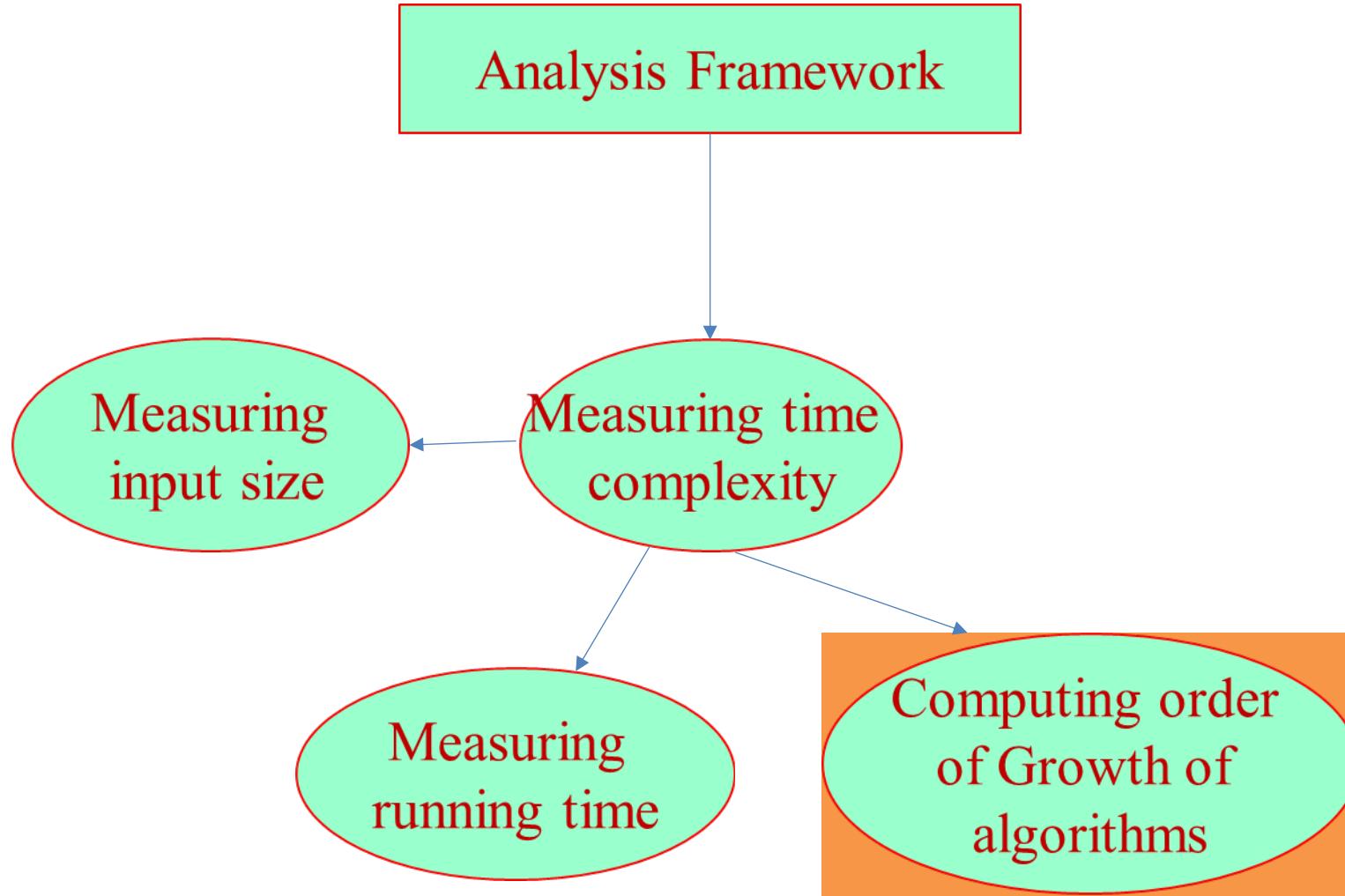
Basic operation: the operation that contributes the most towards the running time of the algorithm



Note: Different basic operations may cost differently!

## Input size and basic operation examples

Problem	Input size measure	Basic operation
Searching for key in a list of $n$ items	Number of list's items, i.e. $n$	Key comparison
Multiplication of two matrices	Matrix dimensions or total number of elements	Multiplication of two numbers
Computing GCD of two numbers	Two numbers	Division



Measuring the performance of an algorithm in relation with the input size “n” is called “Order of Growth”.

Exponential-growth functions

$n$	$\log_2 n$	$n$	$n \log_2 n$	$n^2$	$n^3$	$2^n$	$n!$
10	3.3	$10^1$	$3.3 \cdot 10^1$	$10^2$	$10^3$	$10^3$	$3.6 \cdot 10^6$
$10^2$	6.6	$10^2$	$6.6 \cdot 10^2$	$10^4$	$10^6$	$1.3 \cdot 10^{30}$	$9.3 \cdot 10^{157}$
$10^3$	10	$10^3$	$1.0 \cdot 10^4$	$10^6$	$10^9$		
$10^4$	13	$10^4$	$1.3 \cdot 10^5$	$10^8$	$10^{12}$		
$10^5$	17	$10^5$	$1.7 \cdot 10^6$	$10^{10}$	$10^{15}$		
$10^6$	20	$10^6$	$2.0 \cdot 10^7$	$10^{12}$	$10^{18}$		

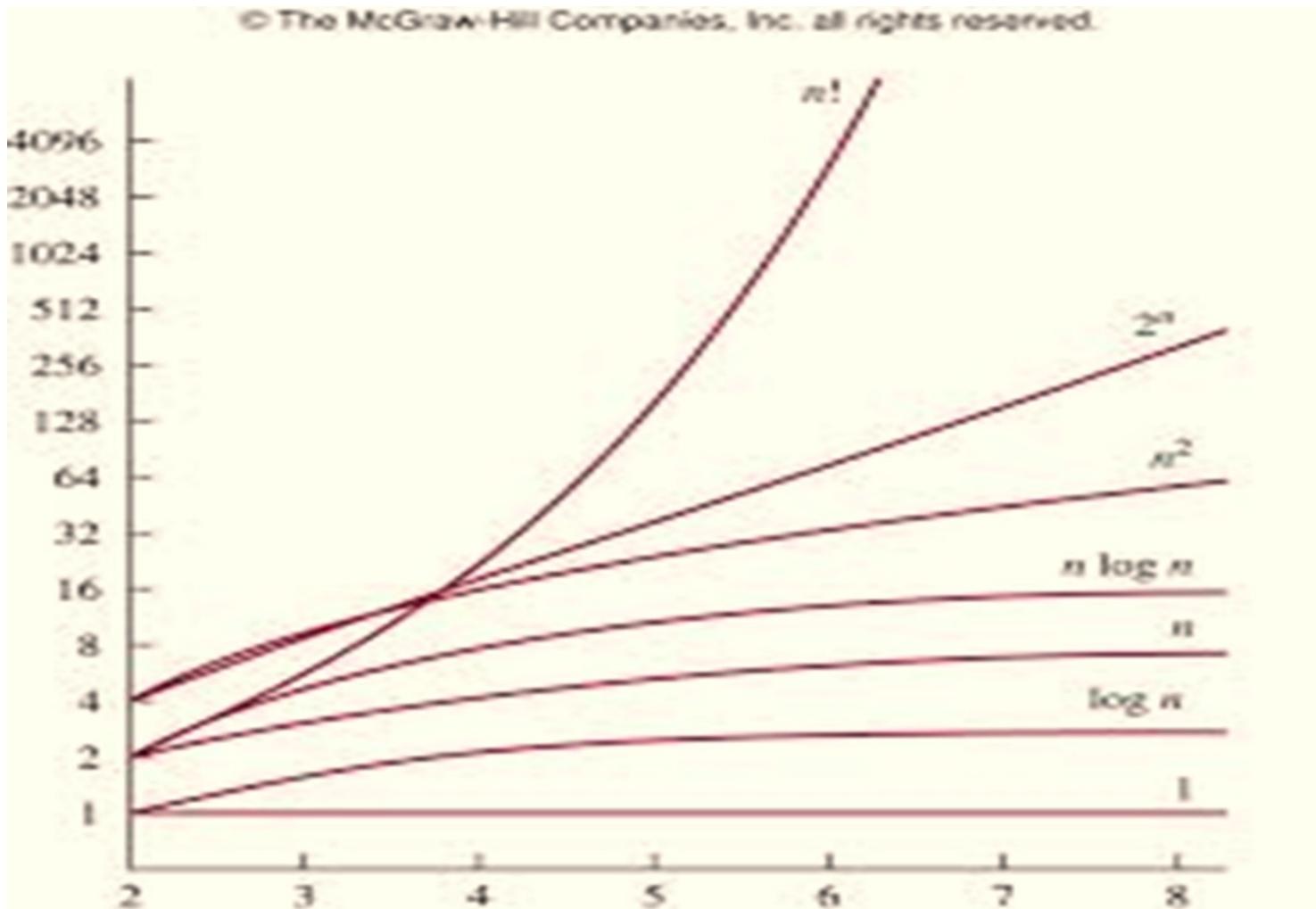
For example the order of growth for varying input size of n is given below

**Table 2.1** Values (some approximate) of several functions important for analysis of algorithms

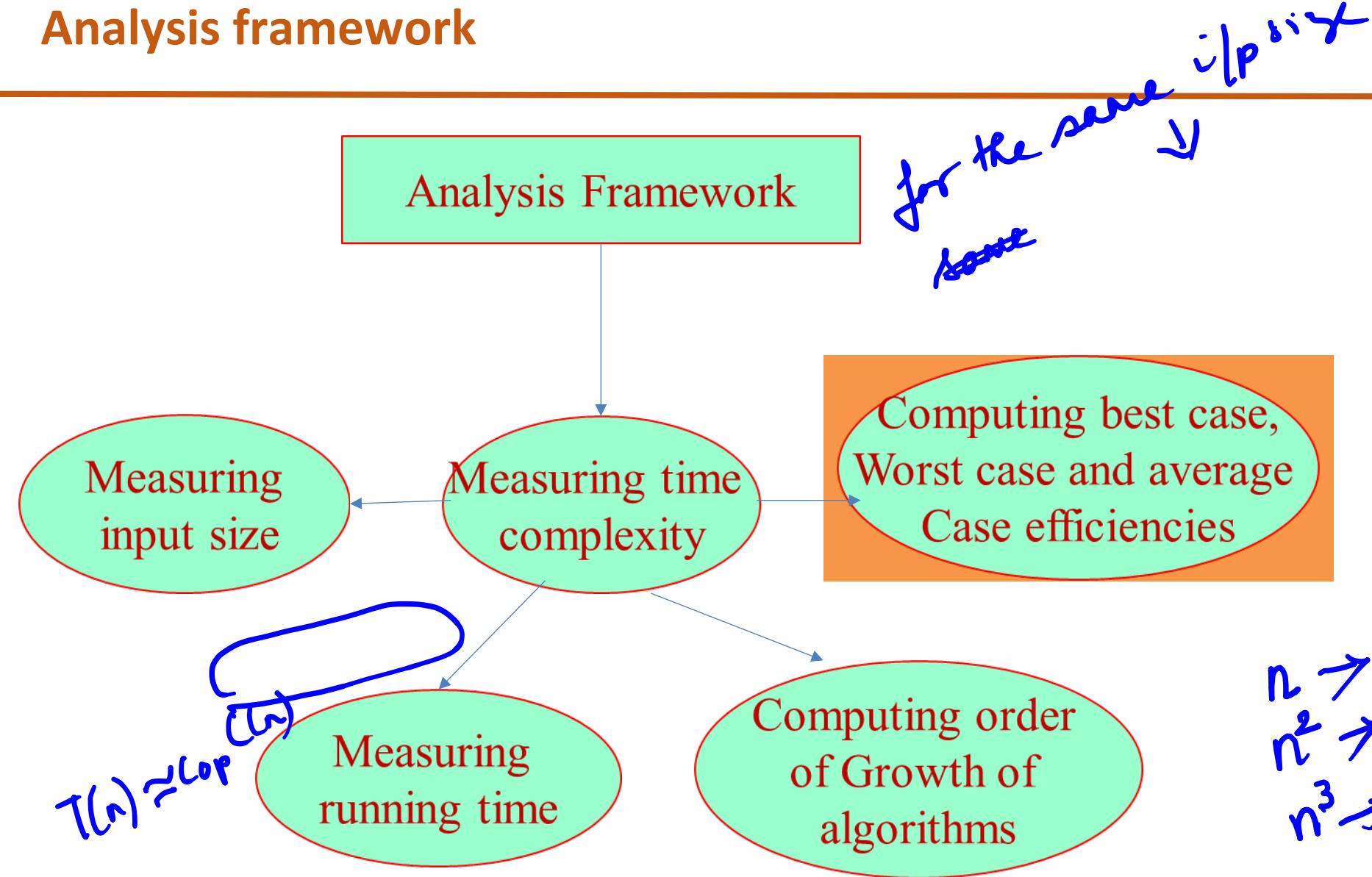
### Orders of growth:

- consider only the leading term of a formula
- ignore the constant coefficient.

## Order of Growth



## Analysis framework



### Best-case, average-case, worst-case

---

If an algorithm takes minimum amount of time to run to completion for a specific set of input it is called **best time complexity**

If an algorithm takes maximum amount of time to run to completion for a specific set of input it is called **worst time complexity**

The time complexity for certain set of inputs is as a average same. Then for the corresponding input such a time complexity is called **average time complexity**.

## Example: Sequential search

---

**ALGORITHM** *SequentialSearch( $A[0..n - 1]$ ,  $K$ )*

```
//Searches for a given value in a given array by sequential search
//Input: An array  $A[0..n - 1]$  and a search key  $K$ 
//Output: The index of the first element of  $A$  that matches  $K$ 
//          or  $-1$  if there are no matching elements
i  $\leftarrow 0$ 
while  $i < n$  and  $A[i] \neq K$  do
    i  $\leftarrow i + 1$ 
if  $i < n$  return i
else return  $-1$ 
```

**Worst case** -----  $n$  key comparisons

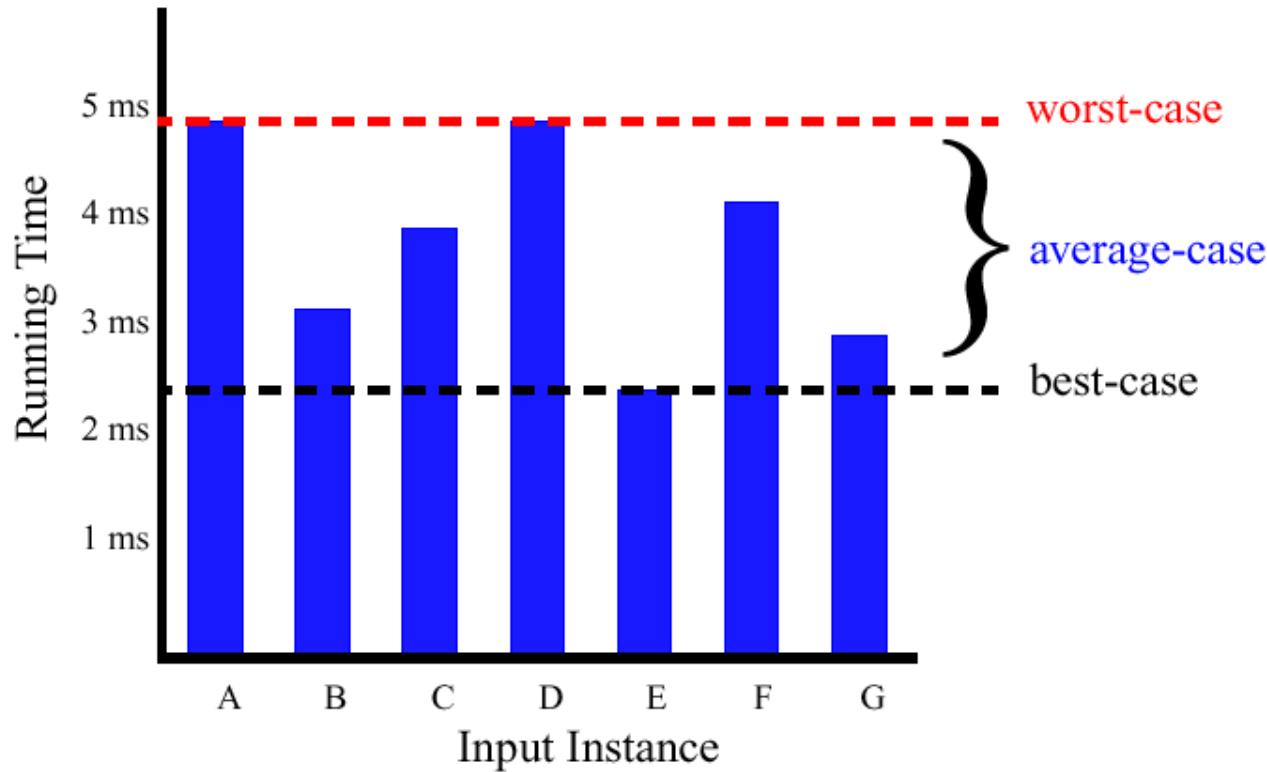
**Best case** ----- 1 comparisons

**Average case** -----  $(n+1)/2$ , assuming  $K$  is in  $A$

# Design and Analysis of Algorithms

## Example:

For a specific size of input  $n$ , investigate running times for different input instances:



# Importance of Analyzing Algorithm

---

- Need to recognize limitations of various algorithms for solving a problem
- Need to understand relationship between problem size and running time
  - When is a running program not good enough?
- Need to learn how to analyze an algorithm's running time without coding it
- Need to learn techniques for writing more efficient code
- Need to recognize bottlenecks in code as well as which parts of code are easiest to optimize



**THANK YOU**

---

**Bharathi R**

Department of Computer Science & Engineering

[rbharathi@pes.edu](mailto:rbharathi@pes.edu)

Slides courtesy of **Anany Levitin**



**PES**  
UNIVERSITY  
**ONLINE**

# **Design and Analysis of Algorithms**

---

**Bharathi R**

Department of Computer Science & Engineering

# DESIGN AND ANALYSIS OF ALGORITHMS

---

## Asymptotic Notations

Slides courtesy of **Anany Levitin**

**Bharathi R**

Department of Computer Science & Engineering

The analysis framework concentrates on the **order of growth** of an **algorithm's basic operation** count as the principal indicator of the algorithm's efficiency.

To compare and rank such orders of growth, computer scientists use three notations:

**O (big oh),**

**$\Omega$ (big omega),**

**$\Theta$ (big theta).**

**$o$ (small oh)**

**$\omega$ (small omega)**

an **asymptote** is a line that the function keeps getting close to but never actually touches(though we symbolically say it touches it at  $x = \infty$ )

For example in linear search and binary search, we really want to know is *how long* these algorithms take.

The running time of an algorithm depends on how long it takes a computer to run the lines of code of the algorithm—and that depends on the size of its input, speed of the computer, the programming language, and the compiler that translates the program from the programming language into code that runs directly on the computer,

First, we need to determine how long the algorithm takes, in terms of the size of its input.

This idea makes intuitive sense, doesn't it? We know that the maximum number of comparison in linear search and binary search increases as the length of the array increases.

For example think about a GPS. If it knew about only the interstate highway system, and not about every little road, it should be able to find routes more quickly, right? So we think about the running time of the algorithm as a *function of the size of its input*.

## Asymptotic Notations- Introduction

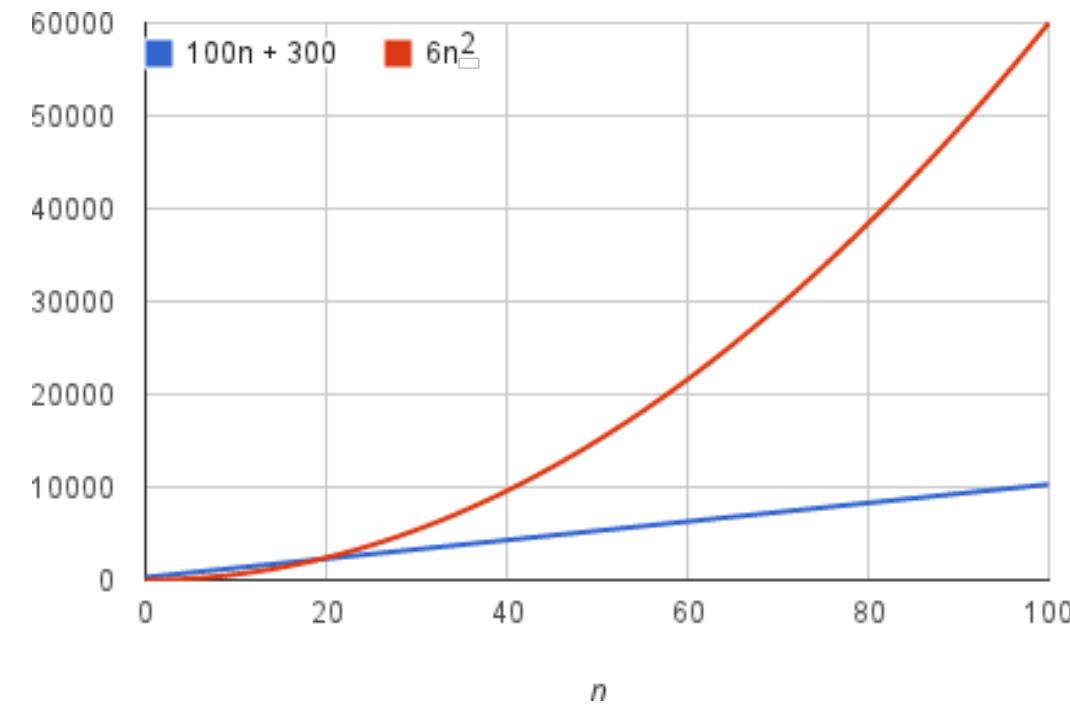


The second idea is that we must focus on how fast a function grows with the input size.

It is called as the **rate of growth** of the running time.

To keep things manageable, we need to simplify the function to distill the most important part and cast aside the less important parts.

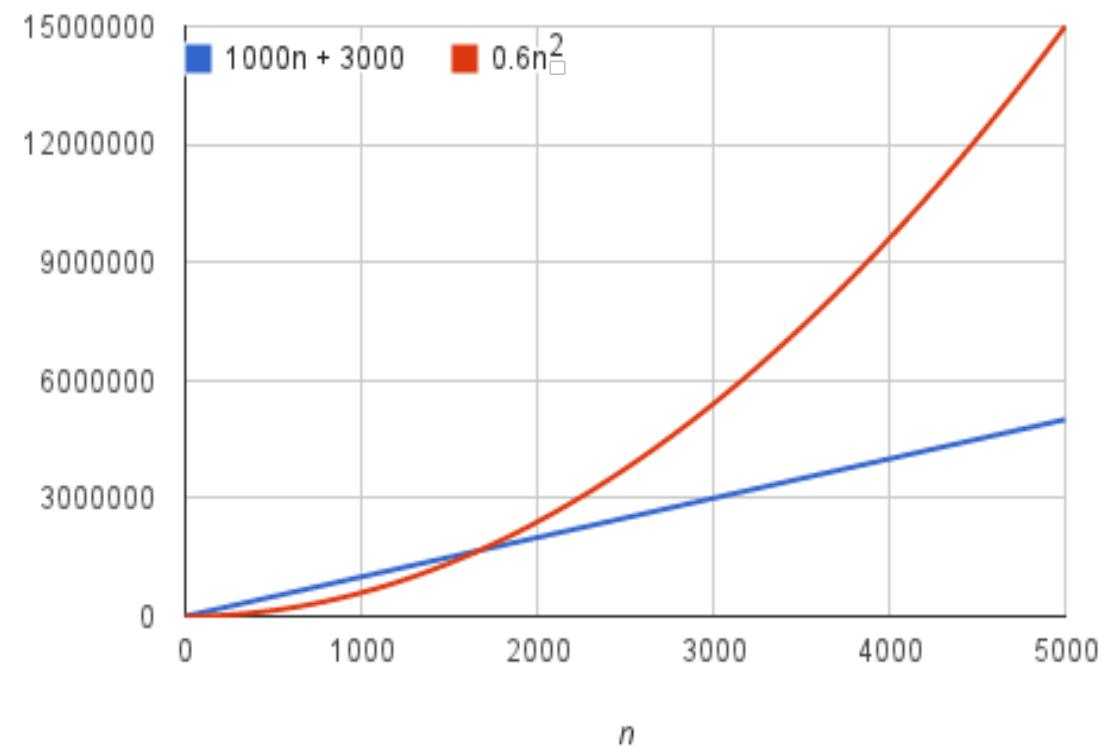
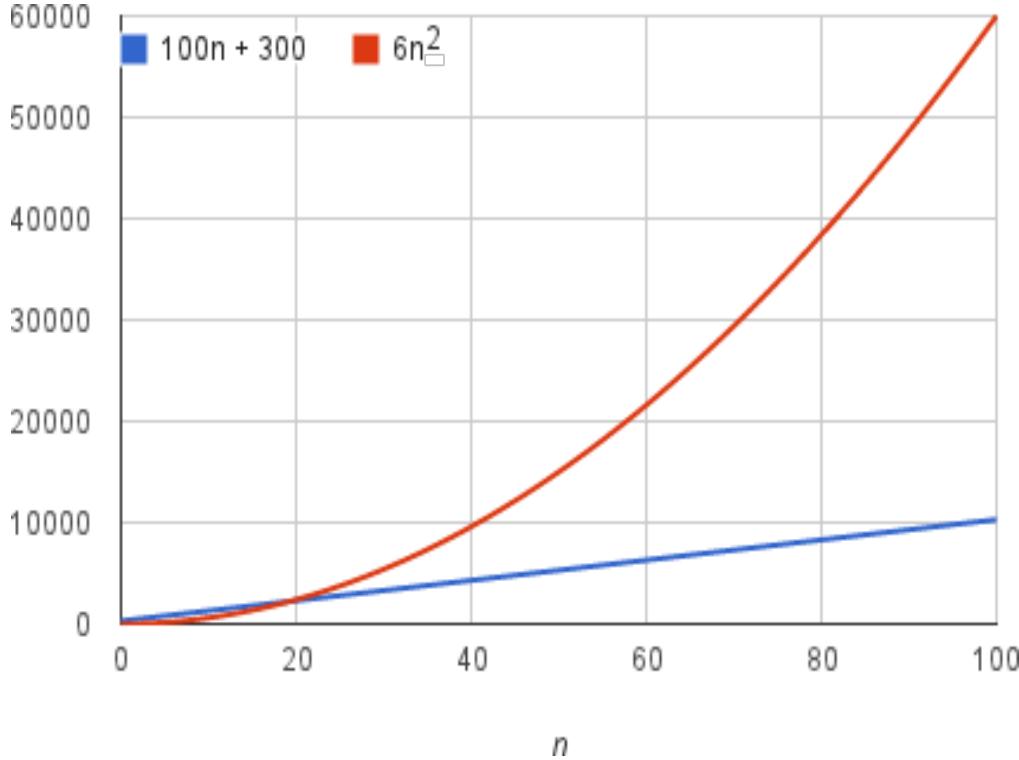
For example, suppose that an algorithm, running on an input of size  $n$ , takes  $6n^2+100n+300$  machine instructions. The  $6n^2$  term becomes larger than the remaining terms,  $100n+300$ , once  $n$  becomes large enough, 20 in this case.



Here's a chart showing values of  $6n^2$  and  $100n+300$  for values of  $n$  from 0 to 100:

# Design and Analysis of Algorithms

## Asymptotic Notations- Introduction



The value of  $n$  at which  $0.6n^2$  becomes greater than  $1000n+3000$  has increased, but there will always be such a crossover point, no matter what the constants.

By dropping the less significant terms and the constant coefficients, we can focus on the important part of an algorithm's running time—its rate of growth—

When we drop the constant coefficients and the less significant terms, we use **asymptotic notation**.

We'll see three forms of it:

**big- $\Theta$ \Theta notation,**

**big- $O$ \ big Oh notation, and**

**big- $\Omega$ \ big Omega notation.**

Orders of growth of an algorithm's basic operation count is important

How do we compare order of growth??

Using Asymptotic Notations

$O(g(n))$ : class of functions  $f(n)$  that grow no faster than  $g(n)$

$$t(n) \leq c g(n)$$

$\Omega(g(n))$ : class of functions  $f(n)$  that grow at least as fast as  $g(n)$

$$t(n) \geq c g(n)$$

$\Theta(g(n))$ : class of functions  $f(n)$  that grow at same rate as  $g(n)$

$$t(n) \approx c g(n)$$

$o(g(n))$ : class of functions  $f(n)$  that grow at slower rate than  $g(n)$

$\omega(g(n))$ : class of functions  $f(n)$  that grow at faster rate than  $g(n)$

### Formal definition

A function  $t(n)$  is said to be in  $O(g(n))$ , denoted  $t(n) \in O(g(n))$ , if  $t(n)$  is bounded above by some constant multiple of  $g(n)$  for all large  $n$ ,  
i.e., if there exist some positive constant c and some nonnegative integer  $n_0$  such that

$$t(n) \leq cg(n) \text{ for all } n \geq n_0$$

Example:  $100n + 5 \in O(n)$

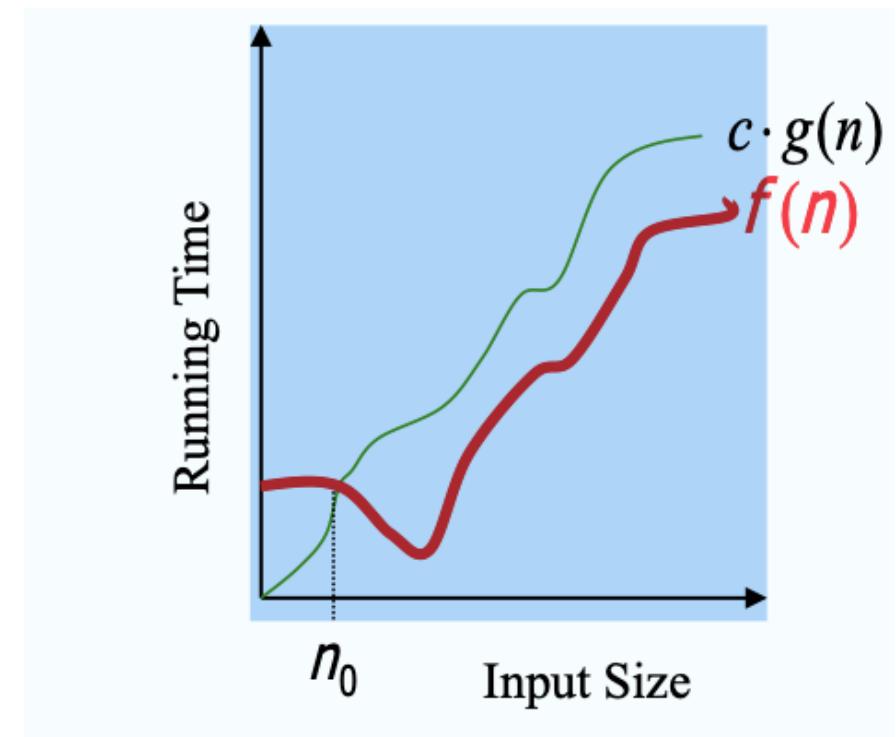
Exercises: prove the following using the above definition

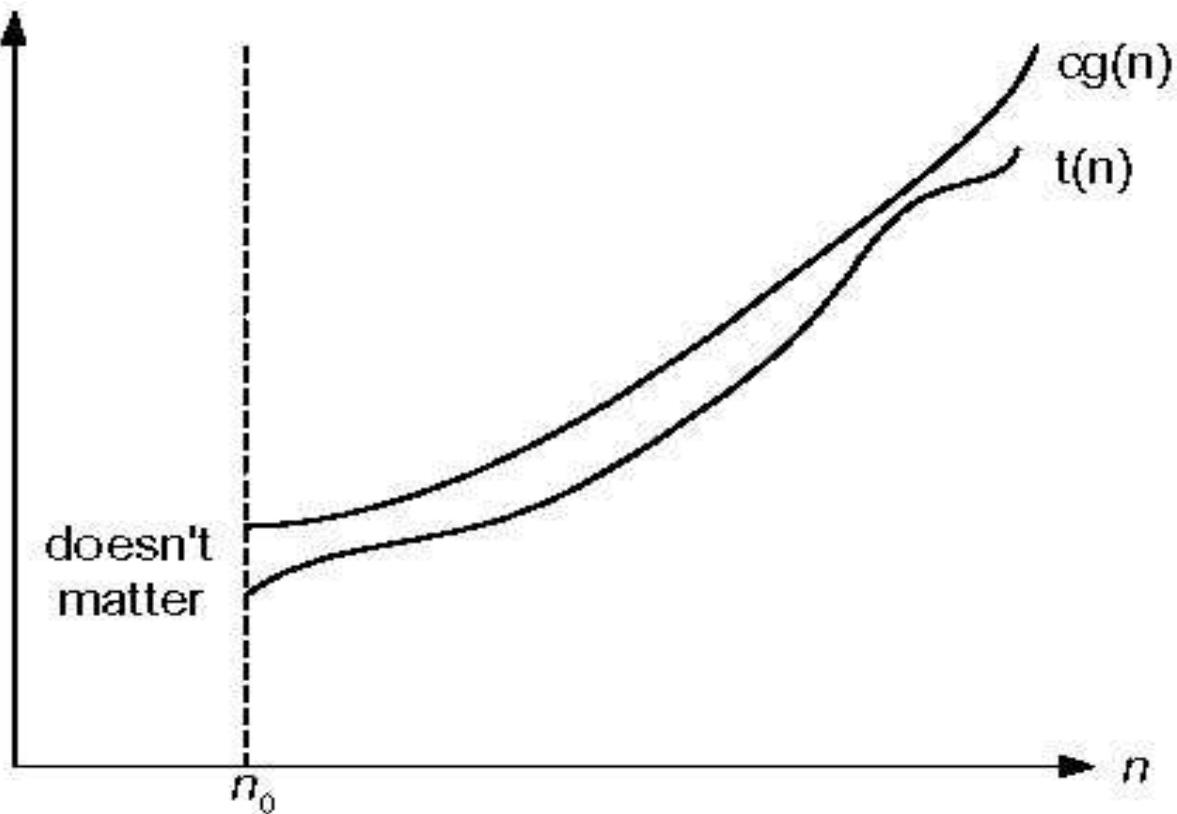
$$10n^2 \in O(n^2)$$

$$10n^2 + 2n \in O(n^2)$$

$$100n + 5 \in O(n^2)$$

$$5n+20 \in O(n)$$





**Figure 2.1** Big-oh notation:  $t(n) \in O(g(n))$

Eg:  $100n+5 \in O(n)$

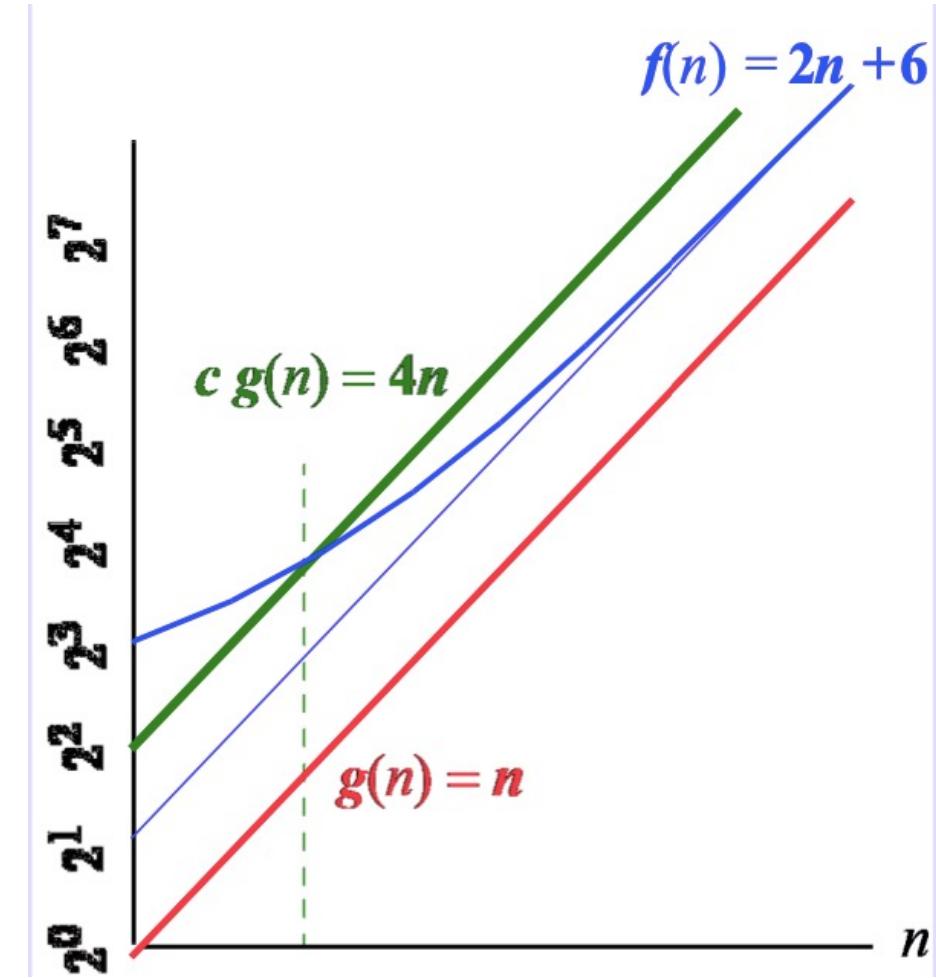
$$100n+5 \leq 100n+5n \ (\forall n \geq 1)$$

$$100n+5 \leq 105n \ \forall n \geq 1 \ (\therefore c=105, n_0=1)$$

$$100n+5 \leq 100n+n \ (\forall n \geq 5) = 101n \ \forall n \geq 5 \ (\therefore c=101, n_0=5)$$

Eg:  $100n+5 \in O(n^2)$

Eg:  $n(n-1)/2 \in O(n^2)$



### Formal definition

A function  $t(n)$  is said to be in  $\Omega(g(n))$ , denoted  $t(n) \in \Omega(g(n))$ , if  $t(n)$  is bounded below by some constant multiple of  $g(n)$  for all large  $n$ ,

i.e., if there exist some positive constant c and some nonnegative integer  $n_0$  such that

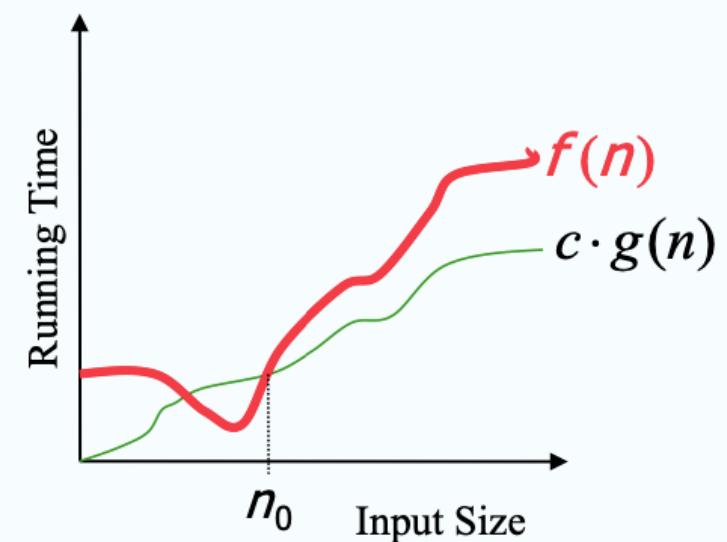
$$t(n) \geq cg(n) \text{ for all } n \geq n_0$$

Exercises: prove the following using the above definition

$$10n^2 \in \Omega(n^2)$$

$$10n^2 + 2n \in \Omega(n^2)$$

$$10n^3 \in \Omega(n^2)$$



# Design and Analysis of Algorithms

## $\Omega$ -notation



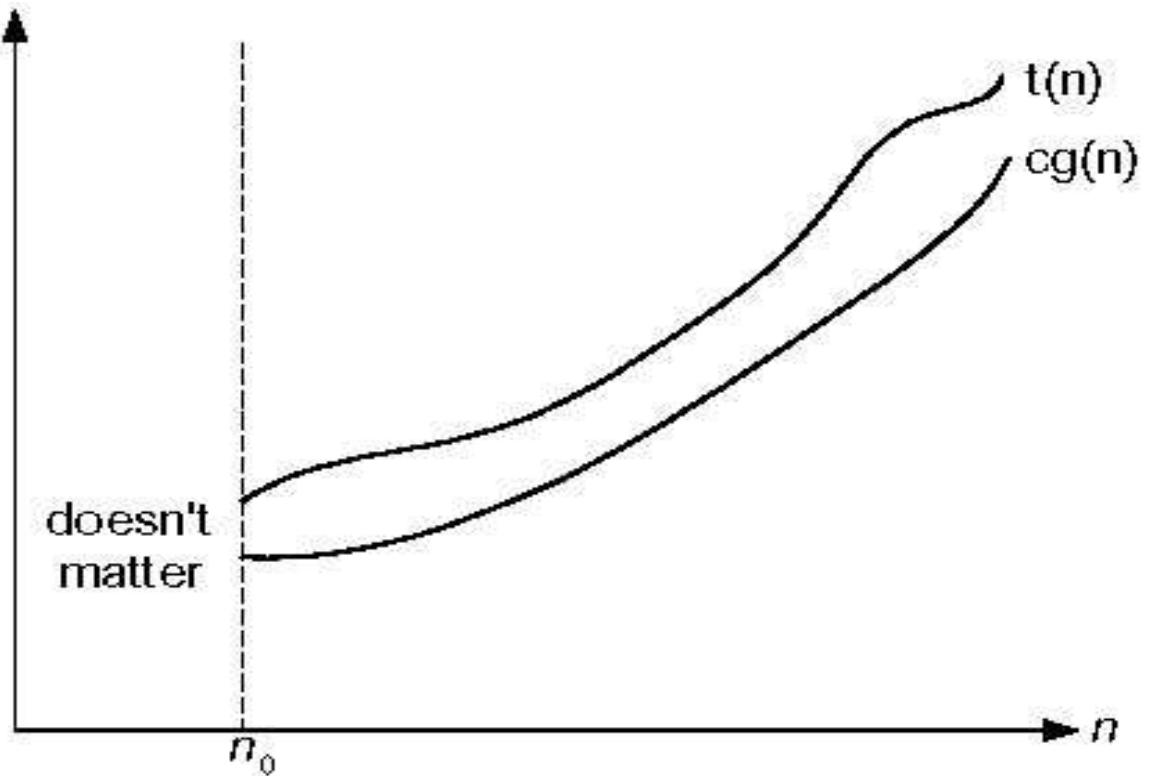
Eg:  $100n+5 \in \Omega(n)$

$$100n+5 \geq 100n \ (\forall n \geq 0)$$

$$100n+5 \geq 100n \ \forall n \geq 0 \ (\therefore c=100, n_0=0)$$

Eg:  $n(n-1)/2 \in \Omega(n^2)$

Eg:  $n(n-1)/2 \in \Omega(n)$



**Fig. 2.2** Big-omega notation:  $t(n) \in \Omega(g(n))$

## $\Theta$ -notation

Formal definition

A function  $t(n)$  is said to be in  $\Theta(g(n))$ , denoted  $t(n) \in \Theta(g(n))$ , if  $t(n)$  is bounded both above and below by some positive constant multiples of  $g(n)$  for all large  $n$ , i.e., if there exist some positive constant  $c_1$  and  $c_2$  and some nonnegative integer  $n_0$  such that

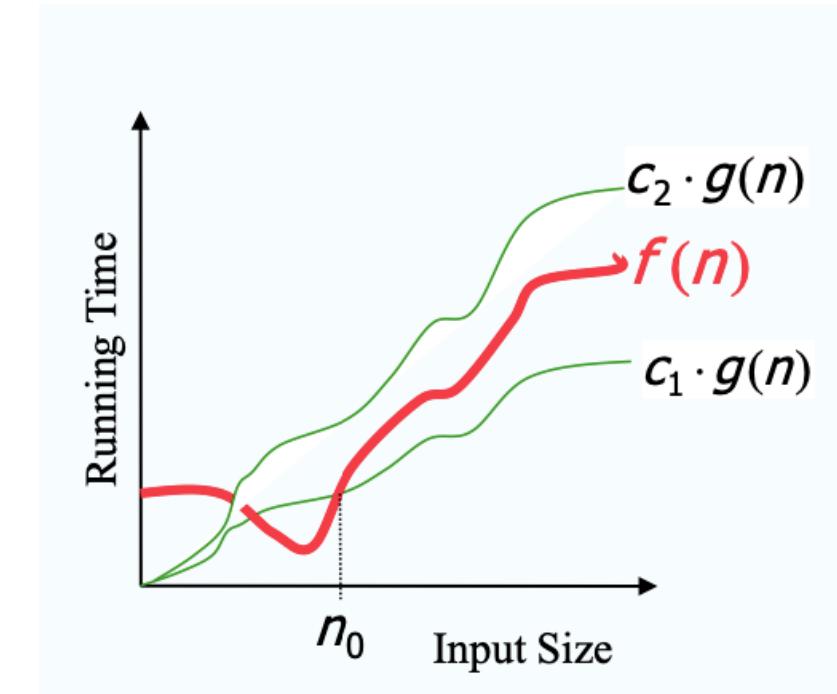
$$c_2 g(n) \leq t(n) \leq c_1 g(n) \text{ for all } n \geq n_0$$

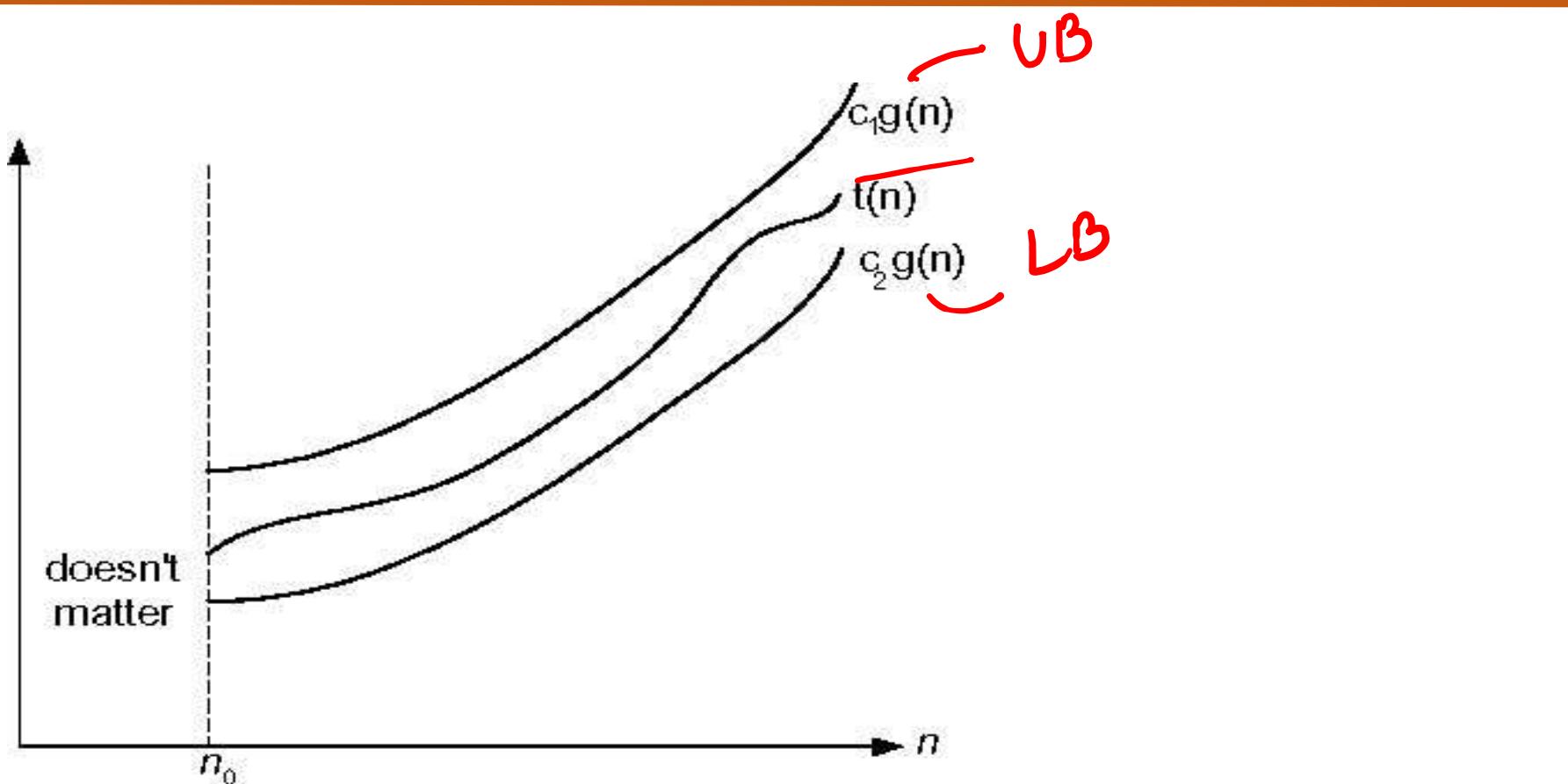
Exercises: prove the following using the above definition

$$10n^2 \in \Theta(n^2) \checkmark$$

$$10n^2 + 2n \in \Theta(n^2) \checkmark$$

$$(1/2)n(n-1) \in \Theta(n^2) \checkmark$$





**Figure 2.3** Big-theta notation:  $t(n) \in \Theta(g(n))$

## Θ-notation

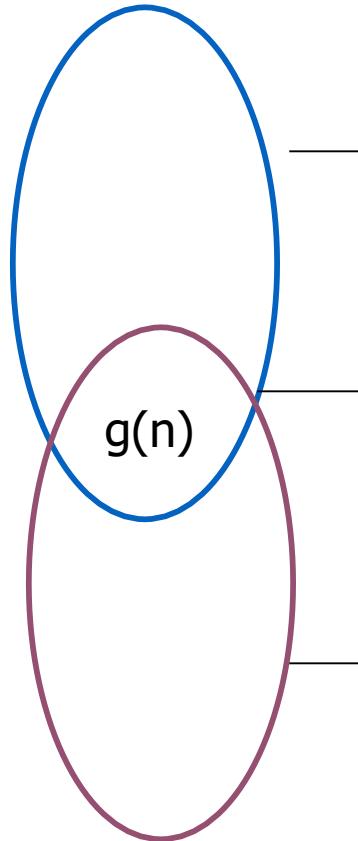
---

Eg:  $n(n-1)/2 \in \Theta(n^2)$

$$n(n-1)/2 = n^2/2 - n/2 \leq n^2/2 \quad \forall n \geq 0$$

$$n(n-1)/2 = n^2/2 - n/2 \geq n^2/2 - n^2/4 = n^2/4 \quad \forall n \geq 2$$

$$\frac{n^2}{4} \leq n(n-1)/2 \leq n^2/2 \quad \forall n \geq 2 \quad (\therefore c_1=1/2, c_2=1/4, n_0=2)$$



$\geq$

$\Omega(g(n))$ , functions that grow at least as fast as  $g(n)$

$=$

$\Theta(g(n))$ , functions that grow at the same rate as  $g(n)$

$\leq$

$O(g(n))$ , functions that grow no faster than  $g(n)$

Formal Definition:

A function  $t(n)$  is said to be in Little-o( $g(n)$ ), denoted  $t(n) \in o(g(n))$ ,  
if there exist some positive constant c and some nonnegative integer  $n_0$  such that

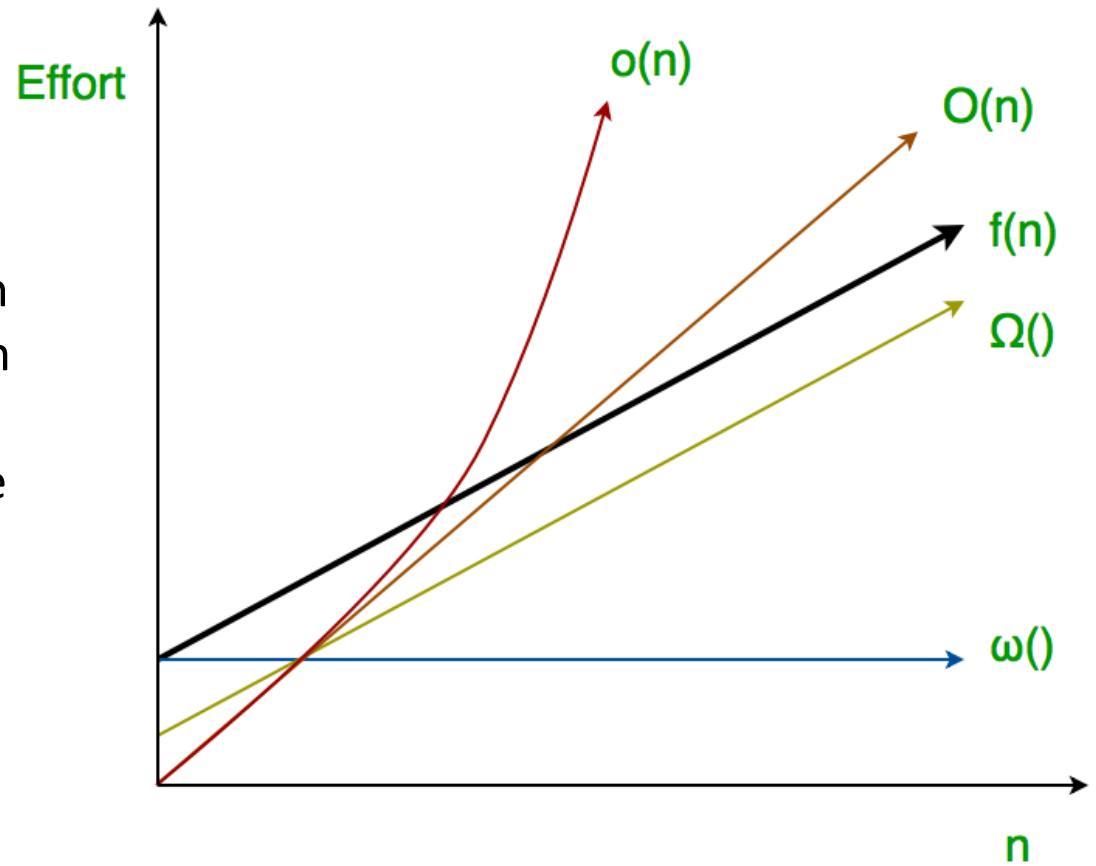
$$0 \leq t(n) < cg(n) \text{ for all } n \geq n_0$$

Example:  $n \in o(n^2)$

## Little o asymptotic notation

Thus, little  $o()$  means **loose upper-bound** of  $f(n)$ . Little  $o$  is a rough estimate of the maximum order of growth whereas Big-O may be the actual order of growth.

Big-O is used as a tight upper-bound on the growth of an algorithm's effort (this effort is described by the function  $f(n)$ ), even though, as written, it can also be a loose upper-bound. “Little-o” ( $o()$ ) notation is used to describe an upper-bound that cannot be tight.



## Little-o Notation

---

Example:  $f(n) = 2n^2$  and  $g(n) = n^2$  and  $c = 2$

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

$f(n) \neq o(g(n))$  - little-o

If  $f(n) = 2n$  &  $g(n) = n^2$ , and  $c = 3$ ,  
then  $f(n) = o(n^2)$

Note : For non-negative functions,  $f(n)$  and  $g(n)$ ,  
 $f(n)$  is little  $o$  of  $g(n)$ , if and only if,

$f(n) = O(g(n))$  and  $f(n) \neq \Theta(g(n))$   
[ strict upper bound, no lower bound]

## Little Omega Notation

---

Formal Definition:

A function  $t(n)$  is said to be in Little-  $\omega(g(n))$ , denoted  $t(n) \in \omega(g(n))$ ,  
if there exist some positive constant c and some nonnegative integer  $n_0$  such that

$$t(n) > cg(n) \geq 0 \text{ for all } n \geq n_0$$

Example:  $3n^2 + 2 \in \omega(n)$

## Little Omega Notation

---

Example : If  $f(n) = 3n + 2$ ,  $g(n) = n$  and  $c = 3$

$$f(n) = \omega(n)$$

Note : For non-negative functions,  $f(n)$  and  $g(n)$ ,  
 $f(n)$  is little  $\omega$  of  $g(n)$ , if and only if

$$f(n) = \Omega(g(n)) \text{ and } f(n) \neq \Theta(g(n))$$

[ strict lower bound, no upper bound]

Algo<sup>1</sup>      VB      Algo<sup>2</sup>      VB

---

**Theorem:** If  $t_1(n) \in O(g_1(n))$  and  $t_2(n) \in O(g_2(n))$ , then  
 $t_1(n) + t_2(n) \in O(\max\{g_1(n), g_2(n)\})$

That is, if

$t_1(n) \leq c_1 g_1(n) \quad \forall n \geq n_1$  and  $t_2(n) \leq c_2 g_2(n) \quad \forall n \geq n_2$

then

$t_1(n) + t_2(n) \leq c \max\{g_1(n), g_2(n)\} \quad \forall n \geq n_0$

**Proof:**

Hint:

W.r.t. to real numbers  $a_1, b_1, a_2, b_2$

if  $a_1 \leq b_1$  and  $a_2 \leq b_2$ , then  $a_1 + a_2 \leq 2 \max\{b_1, b_2\}$

- If  $t_1(n) \in O(g_1(n))$  and  $t_2(n) \in O(g_2(n))$ , then  
 $t_1(n) + t_2(n) \in O(\max\{g_1(n), g_2(n)\})$ .

For example,

$$5n^2 + 3n\log n \in O(n^2)$$

Similarly for Theta and Omega notations,

- If  $t_1(n) \in \Theta(g_1(n))$  and  $t_2(n) \in \Theta(g_2(n))$ , then  
 $t_1(n) + t_2(n) \in \Theta(\max\{g_1(n), g_2(n)\})$
- $t_1(n) \in \Omega(g_1(n))$  and  $t_2(n) \in \Omega(g_2(n))$ , then  
 $t_1(n) + t_2(n) \in \Omega(\max\{g_1(n), g_2(n)\})$

**Implication:** The algorithm's overall efficiency will be determined by the part with a larger order of growth, i.e., its least efficient part.

### Proof:

$$t_1(n) \leq c_1 g_1(n) \quad \forall n \geq n_1$$

$$t_2(n) \leq c_2 g_2(n) \quad \forall n \geq n_2$$

Let us denote  $c_3 = \max\{c_1, c_2\}$  and consider  $n \geq \max\{n_1, n_2\}$

✓

$$\begin{aligned} t_1(n) + t_2(n) &\leq c_1 g_1(n) + c_2 g_2(n) \quad \forall n \geq \max\{n_1, n_2\} \\ &\leq c_3 g_1(n) + c_3 g_2(n) \\ &\leq c_3 (g_1(n) + g_2(n)) \\ &\leq c_3 2 \max\{g_1(n), g_2(n)\} \end{aligned}$$

Therefore,  $t_1(n) + t_2(n) \in O(\max\{g_1(n), g_2(n)\})$

where  $c = 2 \max\{c_1, c_2\}$  and  $n_0 = \max\{n_1, n_2\}$



## Basic asymptotic efficiency classes

Class	Name
$O(1)$	constant ✓
$O(\log n)$	logarithmic ✓
$O(n)$	linear ✓
$n \log n$	linearithmic ✓

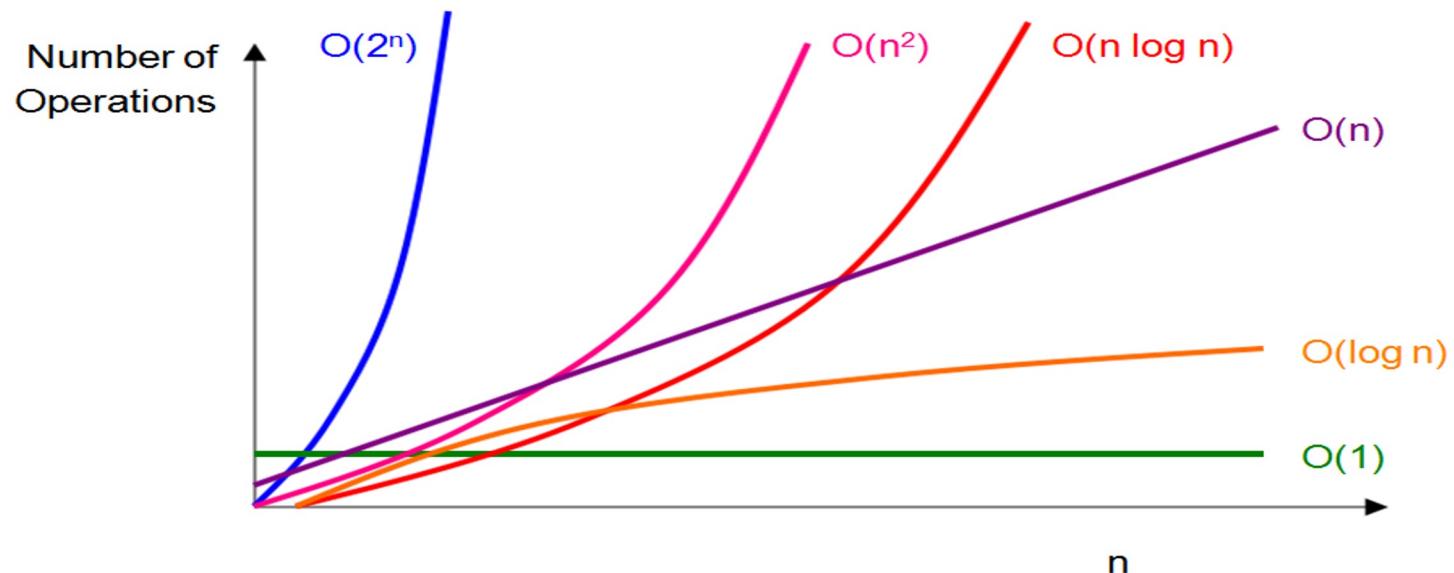
Class	Name
$n^2$	<i>quadratic</i>
$n^3$	<i>cubic</i>
$2^n$	<i>exponential</i>
$n!$	<i>factorial</i>

$n$	$O(1)$	$\Theta(\log n)$	$\Theta(n \log n)$	$\Theta(n^2)$	$\Theta(n^3)$
1	c	0	1	0	1
10	c	1	10	10	1000
100	c	2	100	10000	1000000

# Design and Analysis of Algorithms

## Asymptotic Notations- Summary

$1 < \log \log n < \log n < n^{0.001} < n^{0.5} < n < n \log n < n^2 < n^3 < n^{100} < 1.01^n < 2^n < 100^n < n! < n^n < \dots$



$t(n) \in O(g(n))$

Nonnegative functions defined on the set of natural numbers

- $t(n)$ : algorithm's running time by counting basic operation
- $g(n)$ : simple function to compare the count with.

$O(g(n))$  is the set of all functions with a **smaller or same** order of growth as  $g(n)$ .

E.g.1:  $100n + 5 \in O(n)$

$$\in O(n^2)$$

$$\notin O(\log n)$$

E.g.2  $n(n-1)/2 \in O(n^2)$

$$\in O(n^{10})$$

$$\notin O(n)$$

# Design and Analysis of Algorithms

## Asymptotic Notations- Summary

$\Omega(g(n))$  is the set of all functions with a larger or same order of growth as  $g(n)$ .

E.g.1:  $100n + 5 \in \Omega(n)$  ✓  
 $\notin \Omega(n^2)$  ✓  
 $\in \Omega(\log n)$  ✓

E.g.2  $n(n-1)/2 \in \Omega(n^2)$   
 $\notin \Omega(n^{10})$   
 $\in \Omega(n)$

$\Theta(g(n))$  is the set of all functions that have the same order of growth as  $g(n)$ .

E.g.:  $n(n-1)/2 \in \Theta(n^2)$  ✓  
 $\notin \Theta(n^3)$  ✓  
 $\notin \Theta(n \log n)$  ✓

lower bound.

"average bound"

$c_1 g(n) \leq t(n) \leq c_2 g(n)$

### Problems

---

Use the informal definitions of  $O$ ,  $\Theta$ , and  $\Omega$  to determine whether the following assertions are true or false.

- a.  $n(n + 1)/2 \in O(n^3)$
- b.  $n(n + 1)/2 \in O(n^2)$
- c.  $n(n + 1)/2 \in \Theta(n^3)$
- d.  $n(n + 1)/2 \in \Omega(n)$



## Problems

Use the informal definitions of  $O$ ,  $\Theta$ , and  $\Omega$  to determine whether the following assertions are true or false.

As per the definitions of Asymptotic notation, we can say

- a.  $n(n + 1)/2 \in O(n^3)$   $\Rightarrow T$
- c.  $n(n + 1)/2 \in \Theta(n^3)$   $\Rightarrow F$

- b.  $n(n + 1)/2 \in O(n^2) \Rightarrow T$
- d.  $n(n + 1)/2 \in \Omega(n) \Rightarrow T$

limit functions  
to prove mathematically

#### Problems Exercise 2.2. Problem 4a

---

4. Table 2.1 contains values of several functions that often arise in analysis of algorithms. These values certainly suggest that the functions

$\log n$ ,  $n$ ,  $n \log n$ ,  $n^2$ ,  $n^3$ ,  $2^n$ ,  $n!$

are listed in increasing order of their order of growth. Do these values prove this fact with mathematical certainty?

#### Problems Exercise 2.2. Problem 4a

---

The order of growth and the related notations  $O$ ,  $\Omega$ , and  $\Theta$  deal with the asymptotic behavior of functions as  $n$  goes to infinity. Therefore no specific values of functions within a finite range of  $n$ 's values, suggestive as they might be, can establish their orders of growth with mathematical certainty.



# THANK YOU

---

Bharathi R

Department of Computer Science & Engineering

[rbharathi@pes.edu](mailto:rbharathi@pes.edu)

Slides courtesy of **Anany Levitin**



# Design and Analysis of Algorithms

---

**Bharathi R**

Department of Computer Science & Engineering

# DESIGN AND ANALYSIS OF ALGORITHMS

---

**Basic Efficiency Classes**

**Problems based on Asymptotic notations**

Slides courtesy of **Anany Levitin**

**Bharathi R**

Department of Computer Science & Engineering

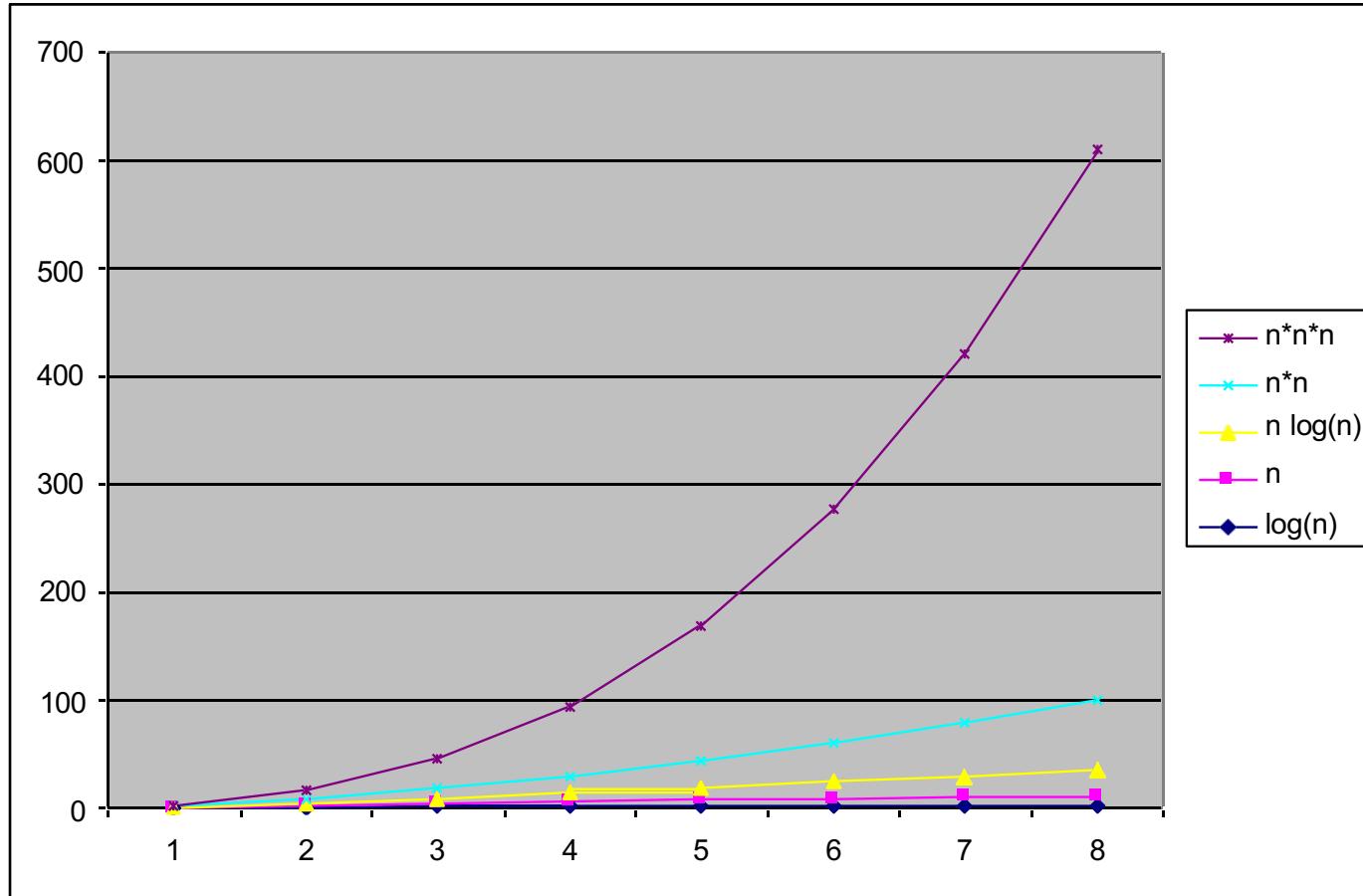
# Design and Analysis of Algorithms

## Basic Efficiency Classes

Class	Name	Example
$1$	constant	Best case for sequential search
$\log n$	logarithmic	Binary Search
$n$	linear	Worst case for sequential search
$n \log n$	$n\text{-log-}n$	Mergesort
$n^2$	quadratic	Bubble Sort
$n^3$	cubic	Matrix Multiplication
$2^n$	exponential	Subset generation
$n!$	factorial	TSP using exhaustive search

# Design and Analysis of Algorithms

## Basic Efficiency Classes



## Basic Efficiency Classes

Order of growth

$n$	$\log_2 n$	$n$	$n \log_2 n$	$n^2$	$n^3$	$2^n$	$n!$
10	3.3	$10^1$	$3.3 \cdot 10^1$	$10^2$	$10^3$	$10^3$	$3.6 \cdot 10^6$
$10^2$	6.6	$10^2$	$6.6 \cdot 10^2$	$10^4$	$10^6$	$1.3 \cdot 10^{30}$	$9.3 \cdot 10^{157}$
$10^3$	10	$10^3$	$1.0 \cdot 10^4$	$10^6$	$10^9$		
$10^4$	13	$10^4$	$1.3 \cdot 10^5$	$10^8$	$10^{12}$		
$10^5$	17	$10^5$	$1.7 \cdot 10^6$	$10^{10}$	$10^{15}$		
$10^6$	20	$10^6$	$2.0 \cdot 10^7$	$10^{12}$	$10^{18}$		

**Table 2.1** Values (some approximate) of several functions important for analysis of algorithms

# Design and Analysis of Algorithms

# Asymptotic notations



O

$f(n) = 3n + 2 \quad g(n) = n$

$3n + 2 \in O(n)$

$f(n) \leq cg(n)$  for some  
+ve c  
 $\forall n \geq n_0$

$\Rightarrow f(n) \in O(g(n))$

$3n + 2 \leq cn$

Let  $c = 4$

$3n + 2 \leq 4n$

$n=1$	$n=1$	$n=3$
$5 \not\leq 4$	$9 \leq 8$	$11 \leq 12$
	✓	✓
		$n_0$

$3n + 2 \leq 4n \quad \forall n \geq 2$

$\Rightarrow 3n + 2 \in O(n)$

$\Sigma$

$3n+2 \in \mathcal{L}(n)$

$P(n) \geq c_f(n)$  for some  $c_f > 0$

$F(n) \in \mathcal{L}(n)$   $\forall n \geq n_0$

$3n+2 \geq cn$

Let  $c = 1$

$3n+2 \geq n$

$n=1$	$n=2$	$n=3$
5	8	11

$5 > 1$     $8 > 2$     $11 > 3$

$3n+2 \geq n$     $n \geq 1$

$3n+2 \in \mathcal{L}(n)$

$$\begin{aligned}
 & \underline{\Theta} \\
 3n+2 & \in \Theta(n) \\
 c_1 g(n) & \leq f(n) \leq c_2 g(n) \quad n \geq n_0 \\
 c_1 = 1 & \quad c_2 = 4 \\
 n_0, r = 1 & \quad n_0 = 2 \\
 n_0 \max(n_{0,1}, n_{0,2}) & = 2 \\
 c_1 = 1 & \quad c_2 = 4 \quad n_0 = 2 \\
 3n+2 & \in \Theta(n) \\
 f(n) & \in Dg(n) \\
 & \leftarrow n(n) \\
 \Rightarrow f(n) & \in \Theta(n)
 \end{aligned}$$

# Design and Analysis of Algorithms

## Asymptotic notations

$$3n+2 \in \mathcal{O}(g(n))$$

$$3n+2 \geq C \cdot g(n)$$



PES  
UNIVERSITY  
ONLINE

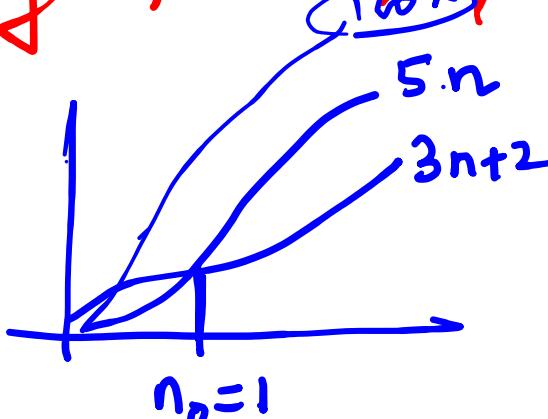
For the given function

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

1)  $f(n) = 3n+2$

Sol:-  $f(n) \leq C \cdot g(n)$

$$3n+2 \leq C \cdot g(n)$$



represent in  
 $O, \Theta, \Omega,$   
 $\downarrow$

$$f(n) \leq C \cdot g(n)$$

$$f(n) \in O(g(n))$$

Let us find the value of  $C$  &  $n_0$ .

Replace 2 as  $2n$  in  $g(n)$

$$3n+2 \leq 3n+2n$$

$$3n+2 \leq 5n$$

$$\begin{array}{ll} n=1 & n=2 \\ 5 \leq 5 & 8 \leq 10 \end{array}$$

# Design and Analysis of Algorithms

## Asymptotic notations



PES  
UNIVERSITY  
ONLINE

"same order".

$$f(n) \in \sim g(n)$$

$$3n+2 \in \sim g(n)$$

$$3n+2 \geq c \cdot g(n)$$

$$\text{let } c = 1$$

$$\boxed{3n+2 \geq 1 \cdot n} \text{ for } \forall n \geq 1$$

$$\text{what is } n_0, n_0 \geq 1$$

$$3$$

$$f(n) \in \Theta(g(n)).$$



**THANK YOU**

---

**Bharathi R**

Department of Computer Science & Engineering



**PES**  
UNIVERSITY  
**ONLINE**

## **Design and Analysis of Algorithms**

---

**Dr. Bharathi R**

Department of Computer Science & Engineering

# DESIGN AND ANALYSIS OF ALGORITHMS

---

## Method of Limits for comparing order of Growth

Slides courtesy of **Anany Levitin**

**Dr. Bharathi R**

Department of Computer Science & Engineering

# Design and Analysis of Algorithms

## Using Limits to Compare Order of Growth

---

$$\lim_{n \rightarrow \infty} \frac{t(n)}{g(n)} = \begin{cases} 0 & \text{implies that } t(n) \text{ has a smaller order of growth than } g(n), \\ c & \text{implies that } t(n) \text{ has the same order of growth as } g(n), \\ \infty & \text{implies that } t(n) \text{ has a larger order of growth than } g(n). \end{cases}$$

Case1:  $t(n) \in O(g(n))$

Case2:  $t(n) \in \Theta(g(n))$

Case3:  $g(n) \in O(t(n))$

$t'(n)$  and  $g'(n)$  are first-order derivatives of  $t(n)$  and  $g(n)$

L'Hopital's Rule

$$\lim_{n \rightarrow \infty} \frac{t(n)}{g(n)} = \lim_{n \rightarrow \infty} \frac{t'(n)}{g'(n)}$$

Stirling's Formula

$$n! \approx \sqrt{2\pi n} \left(\frac{n}{e}\right)^n \text{ for large values of } n$$

Compare the orders of growth of

$$\frac{1}{2}n(n - 1) \text{ and } n^2.$$

---



## Example 1

Compare the orders of growth of  $\frac{1}{2}n(n-1)$  &  $n^2$  using limit based approach.

### Solution

$$\lim_{n \rightarrow \infty} \frac{\frac{1}{2}n(n-1)}{n^2} = \frac{1}{2}$$

Since the limit is equal to a positive constant, the functions have the same order of growth.  
Symbolically

$$\boxed{\frac{1}{2}n(n-1) \in \Theta(n^2)}$$

Compare the orders of growth of  $\log_2 n$  and  $\sqrt{n}$ .

2. Compare the orders of growth of  $\log_2 n$  &  $\sqrt{n}$ .

Sol:-

$$\lim_{n \rightarrow \infty} \frac{\log_2 n}{\sqrt{n}} \xrightarrow{H.R.} g(n)$$

Apply L'Hopital's rule,

$$= \frac{(\log_2 n)'}{(\sqrt{n})'} = \frac{2 \log_2}{\log 2} \lim_{n \rightarrow \infty} \frac{1}{\sqrt{n}} = 0$$

$\therefore \log_2 n$  &  ~~$\sqrt{n}$~~  has a smaller order of growth than  $\sqrt{n}$ .

$$\boxed{\log_2 n \in O(\sqrt{n})}$$

Compare the orders of growth of  $n!$  and  $2^n$ .

3. Compare the orders of growth  $n!$  &  $2^n$

$\downarrow t(n)$        $\downarrow g(n)$

Sol:

$\lim_{n \rightarrow \infty} \frac{n!}{2^n}$ , Apply Stirling's formula,

$$= \lim_{n \rightarrow \infty} \frac{\sqrt{2\pi n} \left(\frac{n}{e}\right)^n}{2^n} = \lim_{n \rightarrow \infty} \sqrt{2\pi n} \left(\frac{n}{2e}\right)^n = \infty.$$

$n! \in \Omega(2^n)$ .

$2^n$  grows very fast &  $n!$  grows still faster  
 $n!$  grows faster than  $2^n$ .

Compare  $t(n)$  &  $g(n)$

---

$$t(n) = \sqrt{5n^3 + 4n + 2} \quad \text{and } g(n) = n^4.$$

$$\lim_{n \rightarrow \infty} \frac{t(n)}{g(n)} = 0$$

Compare the order of growth of  $f(n)$  and  $g(n)$  using method of limits

$$f(n) = 5n^3 + 6n + 2 , \quad g(n) = n^4$$

$$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = \lim_{n \rightarrow \infty} \frac{5n^3 + 6n + 2}{n^4} = \lim_{n \rightarrow \infty} \left( \frac{5}{n} + \frac{6}{n^3} + \frac{2}{n^4} \right) = 0$$

As per case 1

t(n) = O(g(n))

$$5n^3 + 6n + 2 = O(n^4)$$

## Using Limits to Compare Order of Growth: Example 2

$$t(n) = \sqrt{5n^2 + 4n + 2}$$

using the Limits approach determine  $g(n)$  such that  $f(n) = \Theta(g(n))$

Leading term in square root  $n^2$

$$g(n) = \sqrt{n^2} = n$$

$$\begin{aligned}\lim_{n \rightarrow \infty} \frac{t(n)}{g(n)} &= \lim_{n \rightarrow \infty} \frac{\sqrt{5n^2 + 4n + 2}}{\sqrt{n^2}} \\ &= \lim_{n \rightarrow \infty} \sqrt{\frac{5n^2 + 4n + 2}{n^2}} = \lim_{n \rightarrow \infty} \sqrt{5 + \frac{4}{n} + \frac{2}{n^2}} = \sqrt{5}\end{aligned}$$

**non-zero constant**

Hence,  $t(n) = \Theta(g(n)) = \Theta(n)$

## Ex 2.2

3. Indicate the class  $\Theta(g(n))$  the function belongs to,  
 for  
 (Hint: use possible  $g(n)$ )



PES  
UNIVERSITY  
ONLINE

$$t(n) = (n^2 + 1)^{10}$$

Assume:

$$\frac{(n^2 + 1)^{10} \approx (n^2)^{10} = n^{20} \in \Theta(n^{20})}{t(n) \in \Theta(n)} \Rightarrow (n^2 + 1)^{10} \in \Theta(n^{20})$$

How to prove? Using limits,

$$\lim_{n \rightarrow \infty} \frac{(n^2 + 1)^{10}}{n^{20}} = \text{constant} \quad | \quad \text{Hence constant.}$$

Hence  $(n^2 + 1)^{10} \in \Theta(n^{20})$

$\Theta$ ?  
 $t(n) \in \Theta(g(n))$   
 when  $t(n) & g(n)$   
 has same order  
 of growth.

5. Arrange the following functions according to their order of decay (from the highest to the lowest):

$$\frac{(n+1)!}{\cancel{2}^{3n}}, \frac{2n^4 + 2n^3 + 4}{\cancel{n}}, \frac{n \log n}{\cancel{\log n}}, \frac{\log n}{\cancel{\log n}}, \frac{bn}{\cancel{b}}, \frac{8n^2}{\cancel{1}}$$

there are 6 functions,



- 
- a.  $\frac{T(2n)}{T(n)} \approx \frac{c_M \frac{1}{3} (2n)^3}{c_M \frac{1}{3} n^3} = 8$ , where  $c_M$  is the time of one multiplication.
- b. We can estimate the running time for solving systems of order  $n$  on the old computer and that of order  $N$  on the new computer as  $T_{old}(n) \approx c_M \frac{1}{3} n^3$  and  $T_{new}(N) \approx 10^{-3} c_M \frac{1}{3} N^3$ , respectively, where  $c_M$  is the time of one multiplication on the old computer. Replacing  $T_{old}(n)$  and  $T_{new}(N)$  by these estimates in the equation  $T_{old}(n) = T_{new}(N)$  yields  $c_M \frac{1}{3} n^3 \approx 10^{-3} c_M \frac{1}{3} N^3$  or  $\frac{N}{n} \approx 10$ .

$$\lim_{n \rightarrow \infty} t(n)/g(n) \neq 0, \infty \Rightarrow t(n) \in \Theta(g(n))$$

$$\lim_{n \rightarrow \infty} t(n)/g(n) \neq \infty \Rightarrow t(n) \in O(g(n))$$

$$\lim_{n \rightarrow \infty} t(n)/g(n) \neq 0 \Rightarrow t(n) \in \Omega(g(n))$$

$$\lim_{n \rightarrow \infty} t(n)/g(n) = 0 \Rightarrow t(n) \in o(g(n))$$

$$\lim_{n \rightarrow \infty} t(n)/g(n) = \infty \Rightarrow t(n) \in \omega(g(n))$$

## Using Limits to Compare Order of Growth: Example 3

Compare the order of growth of  $t(n)$  and  $g(n)$  using method of limits

$$t(n) = \log_2 n, g(n) = \sqrt{n}$$

$$\lim_{n \rightarrow \infty} \frac{\log_2 n}{\sqrt{n}} = \lim_{n \rightarrow \infty} \frac{(\log_2 n)'}{(\sqrt{n})'} = \lim_{n \rightarrow \infty} \frac{(\log_2 e) \frac{1}{n}}{\frac{1}{2\sqrt{n}}} = 2 \log_2 e \lim_{n \rightarrow \infty} \frac{1}{\sqrt{n}} = 0$$

$$\log_2 n \in o(\sqrt{n})$$

➤ All logarithmic functions  $\log_a n$  belong to the same class

$\Theta(\log n)$  no matter what the logarithm's base  $a > 1$  is

$$\log_{10} n \in \Theta(\log_2 n)$$

➤ All polynomials of the same degree  $k$  belong to the same class:

$$a_k n^k + a_{k-1} n^{k-1} + \dots + a_0 \in \Theta(n^k)$$

➤ Exponential functions can have different orders of growth for different  $a$ 's

$$3^n \notin \Theta(2^n)$$

➤ order  $\log n < \text{order } n^\alpha$  ( $\alpha > 0$ )  $< \text{order } a^n < \text{order } n! < \text{order } n^n$

### Summary

- Method 1: Using limits.
  - L' Hôpital's rule
- Method 2: Using the theorem.
- Method 3: Using the definitions of O-,  $\Omega$ -, and  $\Theta$ -notation.



**THANK YOU**

---

**Bharathi R**

Department of Computer Science & Engineering



**PES**  
**UNIVERSITY**  
**ONLINE**

## Design and Analysis of Algorithms

**Dr.Bharathi R**

Department of Computer Science & Engineering

# DESIGN AND ANALYSIS OF ALGORITHMS

---

## Mathematical Analysis of Non-recursive Algorithms

Slides courtesy of **Anany Levitin**

**Dr.Bharathi R**

Department of Computer Science & Engineering

## Example 1: Finding Max Element in a list

**Algorithm MaxElement ( $A[0..n-1]$ )**

```
//Determines the value of the largest element  
in a given array  
//Input: An array A[0..n-1] of real numbers  
//Output: The value of the largest element in A  
maxval ← A[0]  
for i ← 1 to n-1 do  
    if A[i] > maxval  
        maxval ← A[i]  
return maxval
```

- The basic operation- comparison
- Number of comparisons is the same for all  $i$  of size  $n$ .
- Number of comparisons

Input Size:  $n$

Basic Operation : (  $A[i] > \text{maxval}$  )

$$C_{\text{worst}}(n) = C_{\text{best}}(n) = n-1 \in \Theta(n)$$

$$C(n) = \sum_{i=1}^{n-1} 1 = n - 1 \in \Theta(n)$$

### Steps in mathematical analysis of non-recursive algorithms:

1. Decide on parameter  $n$  indicating input size
2. Identify algorithm's basic operation
3. Check whether the number of times the basic operation is executed depends only on the input size  $n$ . If it also depends on the type of input, investigate worst, average, and best case efficiency separately.
4. Set up summation for  $C(n)$  reflecting the number of times the algorithm's basic operation is executed.
5. Simplify summation using standard formulas

Two basic rules for sum manipulation,

$$1. \sum_{i=l}^u c a_i = c \sum_{i=l}^u a_i$$

$$2. \sum_{i=l}^u a_i \pm b_i = \sum_{i=l}^u a_i \pm \sum_{i=l}^u b_i$$

$$3. \sum_{i=l}^u i = u - l + 1 \quad \text{where } l \leq u \quad \text{are some upper integer limits}$$

$$4. \sum_{i=0}^n i = \sum_{i=1}^n i = 1 + 2 + \dots + n - 1 + n = \frac{n(n+1)}{2} \approx \frac{n^2}{2} + \frac{n}{2} \approx \frac{n^2}{2} \in O(n^2)$$

## Useful Summation Formulas and Rules

## Important Summation Formulas

 $n-l+1 = n$  times.

$$1. \sum_{i=l}^u 1 = \underbrace{1 + 1 + \dots + 1}_{n-l+1 \text{ times}} = u - l + 1 \quad (l, u \text{ are integer limits}, l \leq u); \quad \sum_{i=1}^n 1 = n$$

$$2. \sum_{i=1}^n i = 1 + 2 + \dots + n = \frac{n(n+1)}{2} \approx \frac{1}{2}n^2$$

$$3. \sum_{i=1}^n i^2 = 1^2 + 2^2 + \dots + n^2 = \frac{n(n+1)(2n+1)}{6} \approx \frac{1}{3}n^3$$

$$4. \sum_{i=1}^n i^k = 1^k + 2^k + \dots + n^k \approx \frac{1}{k+1}n^{k+1}$$

$$5. \sum_{i=0}^n a^i = 1 + a + \dots + a^n = \frac{a^{n+1} - 1}{a - 1} \quad (a \neq 1); \quad \sum_{i=0}^n 2^i = 2^{n+1} - 1$$

$$6. \sum_{i=1}^n i2^i = 1 \cdot 2 + 2 \cdot 2^2 + \dots + n2^n = (n-1)2^{n+1} + 2$$

$$7. \sum_{i=1}^n \frac{1}{i} = 1 + \frac{1}{2} + \dots + \frac{1}{n} \approx \ln n + \gamma, \text{ where } \gamma \approx 0.5772 \dots \text{ (Euler's constant)}$$

$$8. \sum_{i=1}^n \lg i \approx n \lg n$$

## Example 2: Element Uniqueness Problem

**Algorithm** UniqueElements ( $A[0..n-1]$ )

//Checks whether all the elements in a given array are distinct

//Input: An array A[0..n-1]

//Output: Returns true if all the elements in A are distinct and false otherwise

for  $i \leftarrow 0$  to  $n - 2$  do

    for  $j \leftarrow i + 1$  to  $n - 1$  do

        if  $A[i] = A[j]$  return false

return true

### Best-case:

If the two first elements of the array are the same No of comparisons in Best case = 1 comparison

### Worst-case:

- Arrays with no equal elements
- Arrays in which only the last two elements are the pair of equal elements

## Example 2: Element Uniqueness Problem

**Algorithm:** UniqueElements ( $A[0..n-1]$ )

Input Size:  $n$

Basic Operation : ( $A[i] = A[j]$ )

$$C_{\text{worst}}(n) = n * (n - 1) / 2 \in \Theta(n^2)$$

$$C_{\text{best}}(n) = 1 \in \Theta(1)$$

$$\begin{aligned} C_{\text{worst}}(n) &= \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1 = \sum_{i=0}^{n-2} [(n-1) - (i+1) + 1] = \sum_{i=0}^{n-2} (n-1-i) \\ &= \sum_{i=0}^{n-2} (n-1) - \sum_{i=0}^{n-2} i = (n-1) \sum_{i=0}^{n-2} 1 - \frac{(n-2)(n-1)}{2} \\ &= (n-1)^2 - \frac{(n-2)(n-1)}{2} = \frac{(n-1)n}{2} \approx \frac{1}{2}n^2 \in \Theta(n^2). \end{aligned}$$

## Example 2: Element Uniqueness Problem

---

$$\begin{aligned}C_{worst}(n) &= \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1 = \sum_{i=0}^{n-2} [(n-1) - (i+1) + 1] = \sum_{i=0}^{n-2} (n-1-i) \\&= \sum_{i=0}^{n-2} (n-1) - \sum_{i=0}^{n-2} i = (n-1) \sum_{i=0}^{n-2} 1 - \frac{(n-2)(n-1)}{2} \\&= (n-1)^2 - \frac{(n-2)(n-1)}{2} = \frac{(n-1)n}{2} \approx \frac{1}{2}n^2\end{aligned}$$

Best-case: 1 comparison

Worst-case:  $n^2/2$  comparisons

$$T(n)_{\text{worst case}} = O(n^2)$$

## Example 3:Matrix Multiplication

---

**EXAMPLE 3** Given two  $n \times n$  matrices  $A$  and  $B$ , find the time efficiency of the definition-based algorithm for computing their product  $C = AB$ . By definition,  $C$  is an  $n \times n$  matrix whose elements are computed as the scalar (dot) products of the rows of matrix  $A$  and the columns of matrix  $B$ :

$$\text{row } i \begin{bmatrix} & & & & \\ \boxed{\quad} & \boxed{\quad} & \boxed{\quad} & \boxed{\quad} & \boxed{\quad} \end{bmatrix} * \begin{bmatrix} & & \\ & \boxed{\quad} & \\ & \boxed{\quad} & \\ & \boxed{\quad} & \\ & \boxed{\quad} & \end{bmatrix} = \begin{bmatrix} & \\ C[i,j] & \end{bmatrix}$$

A                            B                            C

col.  $j$

where  $C[i, j] = A[i, 0]B[0, j] + \dots + A[i, k]B[k, j] + \dots + A[i, n - 1]B[n - 1, j]$   
for every pair of indices  $0 \leq i, j \leq n - 1$ .

## Example 3:Matrix Multiplication

---

**Algorithm** MatrixMultiplication( $A[0..n-1, 0..n-1]$ ,  $B[0..n-1, 0..n-1]$ )

//Multiplies two square matrices of order n by the definition-based algorithm

//Input: two n-by-n matrices A and B

//Output: Matrix C = AB

for i  $\leftarrow$  0 to n - 1 do

    for j  $\leftarrow$  0 to n - 1 do

        C[i, j]  $\leftarrow$  0.0

        for k  $\leftarrow$  0 to n - 1 do

            C[i, j]  $\leftarrow$  C[i, j] + A[i, k] \* B[k, j]

return C

Input Size:  $n$

Basic Operation : Multiplication

( $A[i, k] * B[k, j]$ )

$C_{worst}(n) = C_{best}(n) = n^3 \in \Theta(n^3)$

$M(n) \in \Theta(n^3)$

## Example 3:Matrix Multiplication : Time Complexity

The total number of multiplications  $M(n)$  is expressed by the following triple sum:

$$\begin{aligned} M(n) &= \text{Outermost loop X Inner loop X Innermost loop (1 execution)} \\ &= [\text{For loop using } i] \times [\text{For loop using } j] \times [\text{For loop using } k] \text{ (1 execution)} \end{aligned}$$

$$M(n) = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} \sum_{k=0}^{n-1} 1.$$

$$M(n) \in \Theta(n^3)$$

$$M(n) = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} \sum_{k=0}^{n-1} 1 = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} n = \sum_{i=0}^{n-1} n^2 = n^3.$$

## Example 3:Matrix Multiplication : Time Complexity

---

If we now want to estimate the running time of the algorithm on a particular machine, we can do it by the product

$$T(n) \approx c_m M(n) = c_m n^3,$$

where  $c_m$  is the time of one multiplication on the machine in question. We would get a more accurate estimate if we took into account the time spent on the additions, too:

$$T(n) \approx c_m M(n) + c_a A(n) = c_m n^3 + c_a n^3 = (c_m + c_a)n^3,$$

Consider the following algorithm

- a. What does this algorithm compute?
- b. What is its basic operation?
- c. How many times is the basic operation executed?
- d. What is the efficiency class of this algorithm?
- e. Suggest an improvement, or a better algorithm altogether, and indicate its efficiency class. If you cannot do it, try to prove that, in fact, it cannot be done.

### ALGORITHM *Mystery(n)*

```
//Input: A nonnegative integer n
S ← 0
for i ← 1 to n do
    S ← S + i
return S
```

1. Compute the following sums.

a.  $1 + 3 + 5 + 7 + \dots + 999$

b.  $2 + 4 + 8 + 16 + \dots + 1024$

c.  $\sum_{i=3}^{n+1} 1$

d.  $\sum_{i=3}^{n+1} i$

e.  $\sum_{i=0}^{n-1} i(i + 1)$

f.  $\sum_{j=1}^n 3^{j+1}$

g.  $\sum_{i=1}^n \sum_{j=1}^n ij$

h.  $\sum_{i=0}^{n-1} 1/i(i + 1)$



**THANK YOU**

---

**Dr.Bharathi R**

Department of Computer Science & Engineering

**rbharathi@pes.edu**



## Design and Analysis of Algorithms

**Dr Bharathi R**

Department of Computer Science & Engineering

# DESIGN AND ANALYSIS OF ALGORITHMS

---

## Mathematical Analysis of Recursive Algorithms

Slides courtesy of **Anany Levitin**

**Dr Bharathi R**

Department of Computer Science & Engineering

1. Decide on parameter  $n$  indicating input size
2. Identify algorithm's basic operation
3. If the number of times the basic operation is executed varies with different inputs of same sizes , investigate worst, average, and best case efficiency separately
4. Set up a recurrence relation and initial condition(s) for  $C(n)$ -the number of times the basic operation will be executed for an input of size  $n$
5. Solve the recurrence or estimate the order of magnitude of the solution

### ➤ Decrease-by-one recurrences

A decrease-by-one algorithm solves a problem by exploiting a relationship between a given instance of size  $n$  and a smaller size  $n - 1$ .

**Example:**  $n!$

The recurrence equation has the form

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

### ➤ Decrease-by-a-constant-factor recurrences

A decrease-by-a-constant algorithm solves a problem by dividing its given instance of size  $n$  into several smaller instances of size  $n/b$ , solving each of them recursively, and then, if necessary, combining the solutions to the smaller instances into a solution to the given instance.

**Example:** binary search.

The recurrence has the form

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

- Substitution Method
  - Mathematical Induction
  - Backward substitution
- Recursion Tree Method
- Master Method (Decrease by constant factor recurrences)

**EXAMPLE 1** Compute the factorial function  $F(n) = n!$  for an arbitrary nonnegative integer  $n$ . Since

$$n! = 1 \cdot \dots \cdot (n - 1) \cdot n = (n - 1)! \cdot n \quad \text{for } n \geq 1$$

and  $0! = 1$  by definition, we can compute  $F(n) = F(n - 1) \cdot n$  with the following recursive algorithm.

## Example1: Recursive Algorithms

---

**EXAMPLE 1** Compute the factorial function  $F(n) = n!$  for an arbitrary nonnegative integer  $n$ . Since

$$n! = 1 \cdot \dots \cdot (n-1) \cdot n = (n-1)! \cdot n \quad \text{for } n \geq 1$$

and  $0! = 1$  by definition, we can compute  $F(n) = F(n-1) \cdot n$  with the following recursive algorithm.

### ALGORITHM $F(n)$

```
//Computes  $n!$  recursively
//Input: A nonnegative integer  $n$ 
//Output: The value of  $n!$ 
if  $n = 0$  return 1
else return  $F(n - 1) * n$ 
```

## Example1: Recursive Algorithms

---

### ALGORITHM $F(n)$

```
//Computes  $n!$  recursively
//Input: A nonnegative integer  $n$ 
//Output: The value of  $n!$ 
if  $n = 0$  return 1
else return  $F(n - 1) * n$ 
```

The **basic operation of the algorithm is multiplication**, whose number of executions we denote  $M(n)$ . Since the function  $F(n)$  is computed according to the formula

$$F(n) = F(n - 1) \cdot n \text{ for } n > 0,$$

the number of multiplications  $M(n)$  needed to compute it must satisfy the equality

$$M(n) = M(n - 1) + \frac{1}{\text{to multiply } F(n-1) \text{ by } n} \quad \text{for } n > 0.$$

# Design and Analysis of Algorithms

## Example1: Recursive Algorithms

ALGORITHM  $F(n) \rightarrow M(n)$

//Computes  $n!$  recursively  
//Input: A nonnegative integer  $n$   
//Output: The value of  $n!$

if  $n = 0$  return 1 → initial  $M(0) = 0$   
else return  $F(n - 1) * n \rightarrow M(n-1) + 1$

$$M(n) = M(n-1) + 1 \text{ for } n > 0$$
$$M(0) = 0 \quad \rightarrow \text{initial condition}$$

Solution  $\rightarrow M(n) = n$

1. Input size =  $n$
2. Basic op = multiplication
3. No best, worst case
4. Recurrence relation & initial condition
5. Solve the recurrence relation

$$\begin{aligned} M(n) &= M(n-1) + 1 \\ &= M(n-2) + 1 + 1 \\ &= M(n-2) + 2 \\ &= M(n-3) + 3 \\ &\dots \\ &= M(n-k) + k \\ &\dots \\ &= M(n-n) + n \\ &= M(0) + n \\ &= 0 + n \end{aligned}$$



PES  
UNIVERSITY  
ONLINE

## Example:2

---

Let us investigate a recursive version of the algorithm discussed at the end of Section 2.3.

**Counting number of binary digits in binary representation of a number**

## Counting number of binary digits in binary representation of a number

**ALGORITHM** *BinRec( $n$ )*

//Input: A positive decimal integer  $n$

# input size?

//Output: The number of binary digits in  $n$ 's binary representation  
basic operation?

# **basic operation?**

if  $n = 1$  return 1

## Best/Worst/Average Case?

**else return**  $\text{BinRec}(\lfloor n/2 \rfloor) + 1$

## Counting number of binary digits in binary representation of a number

**ALGORITHM** *BinRec( $n$ )*

```
//Input: A positive decimal integer  $n$ 
//Output: The number of binary digits in  $n$ 's binary representation
if  $n = 1$  return 1
else return BinRec( $\lfloor n/2 \rfloor$ ) + 1
```

input size?

basic operation?

Best/Worst/Average Case?

$$A(2^k) = A(2^{k-1}) + 1 \quad \text{for } k > 0,$$

$$A(2^0) = 0.$$

$$\begin{aligned} A(2^k) &= A(2^{k-1}) + 1 && \text{substitute } A(2^{k-1}) = A(2^{k-2}) + 1 \\ &= [A(2^{k-2}) + 1] + 1 = A(2^{k-2}) + 2 && \text{substitute } A(2^{k-2}) = A(2^{k-3}) + 1 \\ &= [A(2^{k-3}) + 1] + 2 = A(2^{k-3}) + 3 && \dots \\ &\quad \dots && \\ &= A(2^{k-i}) + i && \\ &\quad \dots && \\ &= A(2^{k-k}) + k. && \end{aligned}$$

$$A(n) = \log_2 n \in \Theta(\log n).$$

## Exercise

---

Consider the following recursive algorithm for computing the sum of the first  $n$  cubes:  $S(n) = 1^3 + 2^3 + \dots + n^3$ .

### ALGORITHM $S(n)$

```
//Input: A positive integer n
//Output: The sum of the first n cubes
if n = 1 return 1
else return S(n - 1) + n * n * n
```

- Set up and solve a recurrence relation for the number of times the algorithm's basic operation is executed.
- How does this algorithm compare with the straightforward nonrecursive algorithm for computing this sum?

---

We can solve it by backward substitutions:

$$\begin{aligned}M(n) &= M(n - 1) + 2 \\&= [M(n - 2) + 2] + 2 = M(n - 2) + 2 + 2 \\&= [M(n - 3) + 2] + 2 + 2 = M(n - 3) + 2 + 2 + 2 \\&= \dots \\&= M(n - i) + 2i \\&= \dots \\&= M(1) + 2(n - 1) = 2(n - 1).\end{aligned}$$

## Exercise

---

4. Consider the following recursive algorithm.

**ALGORITHM**  $Q(n)$

```
//Input: A positive integer n
if  $n = 1$  return 1
else return  $Q(n - 1) + 2 * n - 1$ 
```

- Set up a recurrence relation for this function's values and solve it to determine what this algorithm computes.
- Set up a recurrence relation for the number of multiplications made by this algorithm and solve it.
- Set up a recurrence relation for the number of additions/subtractions made by this algorithm and solve it.

## Solution

---

a.  $Q(n) = Q(n - 1) + 2n - 1 \quad \text{for } n > 1, \quad Q(1) = 1.$

Computing the first few terms of the sequence yields the following:

$$Q(2) = Q(1) + 2 \cdot 2 - 1 = 1 + 2 \cdot 2 - 1 = 4;$$

$$Q(3) = Q(2) + 2 \cdot 3 - 1 = 4 + 2 \cdot 3 - 1 = 9;$$

$$Q(4) = Q(3) + 2 \cdot 4 - 1 = 9 + 2 \cdot 4 - 1 = 16.$$

Thus, it appears that  $Q(n) = n^2$ . We'll check this hypothesis by substituting this formula into the recurrence equation and the initial condition. The left hand side yields  $Q(n) = n^2$ . The right hand side yields

$$Q(n - 1) + 2n - 1 = (n - 1)^2 + 2n - 1 = n^2.$$

The initial condition is verified immediately:  $Q(1) = 1^2 = 1$ .

## Tower of Hanoi

---

**Algorithm TowerOfHanoi( $n$ , Src, Aux, Dst)**

```
if ( $n = 0$ )
    return
    TowerOfHanoi( $n-1$ , Src, Dst, Aux)
    Move disk  $n$  from Src to Dst
    TowerOfHanoi( $n-1$ , Aux, Src, Dst)
```

Input Size:  $n$

Basic Operation : **Move disk  $n$  from Src to Dst**

$C(n) = 2C(n-1) + 1$  for  $n > 0$  and  $C(0)=0$   
 $= 2^n - 1 \in \Theta(2^n)$

Tower of Hanoi

Algorithm TowerOfHanoi( $n$ , Src, Aux, Dst)  $\rightarrow M(n) = ?$

{ if ( $n = 0$ ) } // initial condition  
return

TowerOfHanoi( $n-1$ , Src, Dst, Aux)  $\rightarrow M(n-1)$

Move disk  $n$  from Src to Dst  $\rightarrow$

TowerOfHanoi( $n-1$ , Aux, Src, Dst)  $\rightarrow M(n-1)$

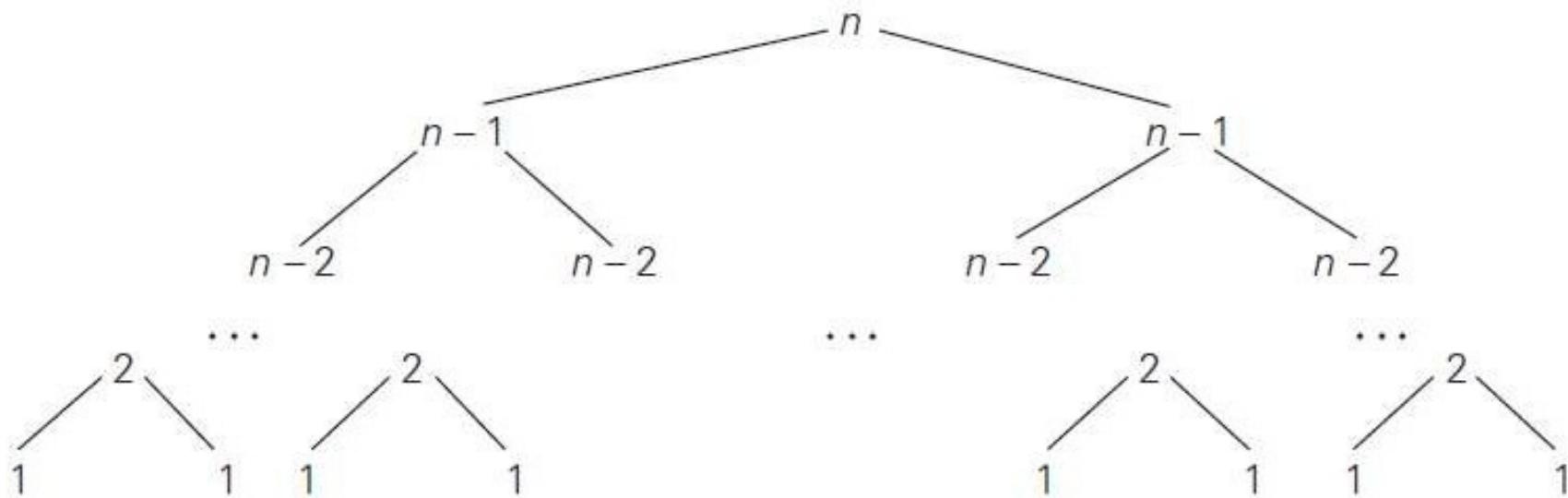
$$\{ M(n) = 2M(n-1) + 1 \text{ for } n > 1 \\ M(1) = 1 \quad n = 1 .$$

Input Size:  $n$

Basic Operation : Move disk  $n$  from Src to Dst

$$C(n) = 2C(n-1) + 1 \text{ for } n > 0 \text{ and } C(0)=0 \\ = 2^n - 1 \in \Theta(2^n)$$

## Tower of Hanoi : Tree of Recursive calls



$$C(n) = \sum_{l=0}^{n-1} 2^l = 2^n - 1$$

Solve the recurrence relation-

$$M(n) = 2M(n-1) + 1 \quad ; \quad M(1) = 1$$



PES  
UNIVERSITY  
ONLINE

By backward substitution

$$\begin{aligned} M(n) &= 2M(n-1) + 1 \\ &= 2[2M(n-2) + 1] + 1 \\ &= 2^2 M(n-2) + 2 + 1 \\ &= 2^3 M(n-3) + 2^2 + 2 + 1 \end{aligned}$$

After  $i$  subs:

$$\begin{aligned} &= 2^i M(n-i) + 2^{i-1} + 2^{i-2} + \dots + 2 + 1 \\ &= 2^i M(n-i) + 2^i - 1 \end{aligned}$$

We need to proceed till we get the initial condition.

$$\text{Let us } i = n-1$$

Sub:  $a = 2$

$$\begin{aligned} 2^{n+1}-1 &= 2^n + 2^{n-1} + \dots + 2 + 1 \\ \sum_{i=0}^n 2^i &= 1 + 2 + \dots + 2^n \\ &= \frac{2^{n+1}-1}{2-1} \end{aligned}$$

If  $a = 2$

$$\frac{2^i-1}{2-1}$$

Subs  $i = n-1$

$$\begin{aligned}M(n) &= 2^{n-1} M(n-n+1) + 2^{n-1} - 1 \\&= 2^{n-1} M(1) + 2^{n-1} - 1 \quad \because M(1)=1 \\&= 2^{n-1} + 2^{n-1} - 1\end{aligned}$$

$$\boxed{M(n) = 2^n - 1}$$

$$\boxed{M(n) \in \Theta(2^n)}$$



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

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

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

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

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

$$T(n) = 2T(\sqrt{n}) + n$$

$$T(n) = 2T(\sqrt{n}) + \log n$$

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

$$\approx 3T(n/3) + 1$$

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

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

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

$$\frac{n}{2} = n$$

1. Solve the following recurrence relations.

a.  $x(n) = x(n - 1) + 5 \quad \text{for } n > 1, \quad x(1) = 0$

b.  $x(n) = 3x(n - 1) \quad \text{for } n > 1, \quad x(1) = 4$

c.  $x(n) = x(n - 1) + n \quad \text{for } n > 0, \quad x(0) = 0$

d.  $x(n) = x(n/2) + n \quad \text{for } n > 1, \quad x(1) = 1 \quad (\text{solve for } n = 2^k)$

e.  $x(n) = x(n/3) + 1 \quad \text{for } n > 1, \quad x(1) = 1 \quad (\text{solve for } n = 3^k)$



**THANK YOU**

---

**Bharathi R**

Department of Computer Science & Engineering



**PES**  
UNIVERSITY  
**ONLINE**

## Design and Analysis of Algorithms

**Bharathi R**

Department of Computer Science & Engineering

# DESIGN AND ANALYSIS OF ALGORITHMS

---

## Solving Recurrences

Slides courtesy of Anany Levitin

Bharathi R

Department of Computer Science & Engineering

# Design and Analysis of Algorithms

## Solving Recurrences: Example1

---

$$T(n) = T(n-1) + 1 \quad n > 0 \quad T(0) = 1$$



$$T(n) = T(n-1) + 1 \quad n > 0 \quad T(0) = 1$$

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

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

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

...

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

...

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

# Design and Analysis of Algorithms

## Solving Recurrences: Example2

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

$$T(0)=0$$



$$\begin{aligned} T(n) &= T(n-1) + 2n - 1 & T(0) = 0 \\ &= [T(n-2) + 2(n-1) - 1] + 2n - 1 \\ &= T(n-2) + 2(n-1) + 2n - 2 \\ &= [T(n-3) + 2(n-2) - 1] + 2(n-1) + 2n - 2 \\ &= T(n-3) + 2(n-2) + 2(n-1) + 2n - 3 \\ &\quad \dots \\ &= T(n-i) + 2(n-i+1) + \dots + 2n - i \\ &\quad \dots \\ &= T(n-n) + 2(n-n+1) + \dots + 2n - n \\ &= 0 + 2 + 4 + \dots + 2n - n \\ &= 2 + 4 + \dots + 2n - n \\ &= 2 * n * (n+1) / 2 - n \\ // \text{arithmetic progression formula } 1 + \dots + n = n(n+1)/2 // \\ &= O(n^2) \end{aligned}$$

# Design and Analysis of Algorithms

## Solving Recurrences: Example3

---

$$T(n) = T(n/2) + 1 \quad n > 1$$

$$T(1) = 1$$



# Design and Analysis of Algorithms

## Solving Recurrences: Example3

---



$$T(n) = T(n/2) + 1 \quad n > 1$$

$$T(1) = 1$$

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

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

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

.....

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

.....

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

$$= 1 + \log n$$

$$= O(\log n)$$

# Design and Analysis of Algorithms

## Solving Recurrences: Example4

---



$$T(n) = 2T(n/2) + cn \quad n > 1 \quad T(1) = c$$

$$\begin{aligned} T(n) &= 2T(n/2) + cn \\ &= 2(2T(n/2^2) + c(n/2)) + cn = 2^2 T(n/2^2) + cn + cn \\ &= 2^2 (2T(n/2^3) + c(n/2^2)) + cn + cn = 2^3 T(n/2^3) + 3cn \\ &\dots\dots \\ &= 2^i T(n/2^i) + icn \\ &\dots\dots \\ &= 2^k T(n/2^k) + kcn \quad (k = \log n) \\ &= nT(1) + cn \log n = cn + cn \log n \\ &= O(n \log n) \end{aligned}$$



**THANK YOU**

---

**Bharathi R**

Department of Computer Science & Engineering

[rbharathi@pes.edu](mailto:rbharathi@pes.edu)



**PES**  
UNIVERSITY  
**ONLINE**

# **Design and Analysis of Algorithms**

---

**Vandana M L**

Department of Computer Science & Engineering

# DESIGN AND ANALYSIS OF ALGORITHMS

---

## Performance Analysis Vs Performance Measurement

Slides courtesy of **Anany Levitin**

**Vandana M L**

Department of Computer Science & Engineering

➤ Performance Analysis → Algorithm

- Machine Independent
- Prior Evaluation

➤ Performance Measurement → program → C, C++, Java, Python etc.

- Machine Dependent , compiler
- Posterior Evaluation → testing .

## Performance Analysis of Sequential search :Worst Case

---

ALGORITHM SequentialSearch(A[0..n-1], K)

//Searches for a given value in a given array by sequential search

//Input: An array A[0..n-1] and a search key K

//Output: Returns the index of the first element of A that matches K or -1 if there are no matching elements

i  $\leftarrow$  0

while i < n and A[i]  $\neq$  K do

    i  $\leftarrow$  i + 1

if i < n        //A[i] = K

    return i

else

    return -1

Basic operation:      A[i]  $\neq$  K

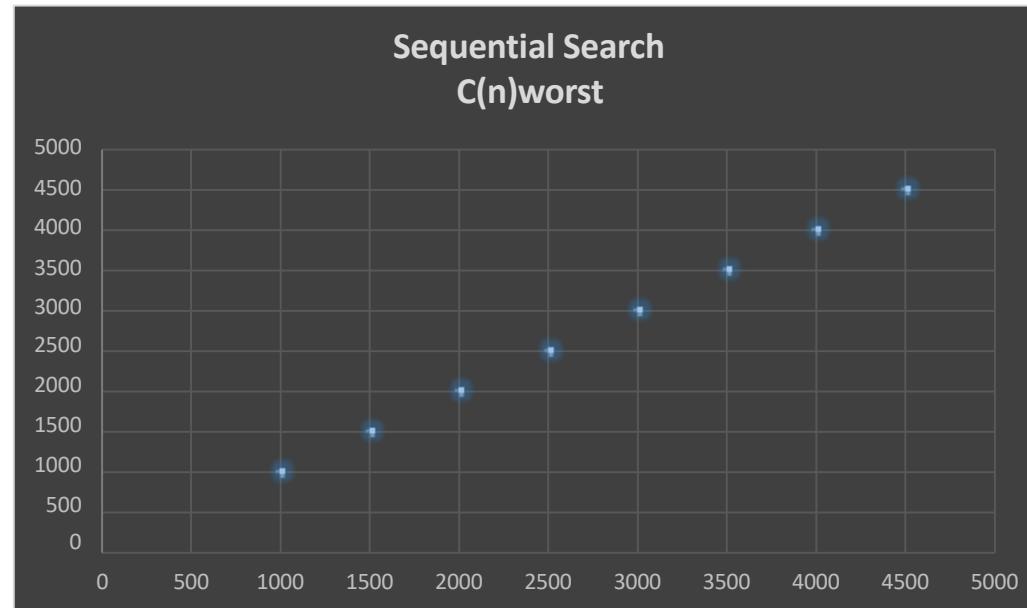
Basic operation count: n

Time Complexity:      T(n)  $\in$  O(n)

# Design and Analysis of Algorithms

## Performance Analysis of Sequential Search

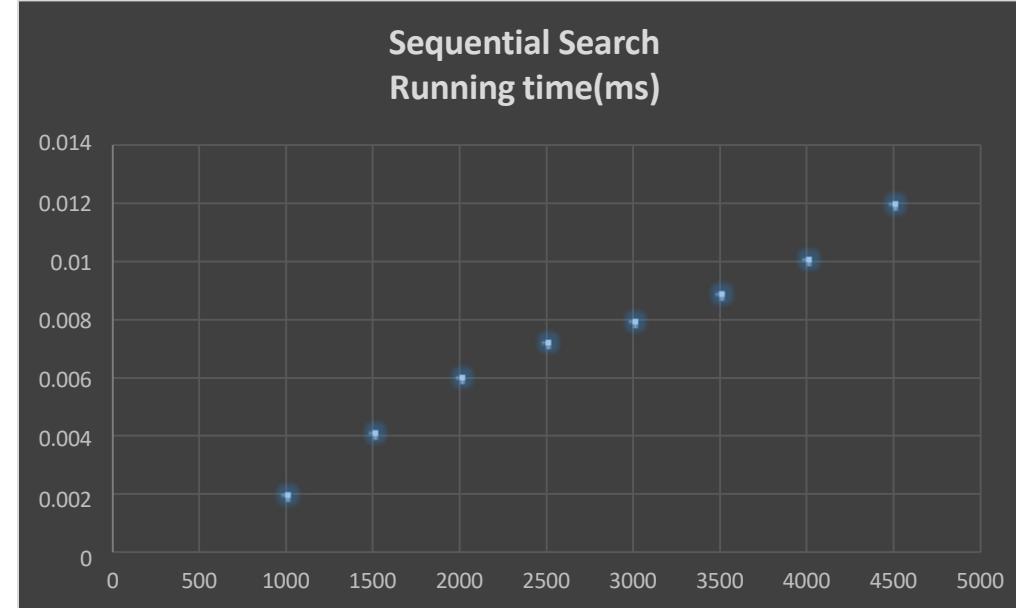
Input Size	Sequential Search $C(n)$ worst
1000	1000
1500	1500
2000	2000
2500	2500
3000	3000
3500	3500
4000	4000
4500	4500



# Design and Analysis of Algorithms

## Performance Measurement of Sequential Search

Input Size	Sequential Search Actual Running Time(ms)
1000	0.001907
1500	0.004053
2000	0.00596
2500	0.007153
3000	0.007868
3500	0.008821
4000	0.010014
4500	0.011921





**THANK YOU**

---

**Vandana M L**

Department of Computer Science & Engineering

[vandanamd@pes.edu](mailto:vandanamd@pes.edu)



# **DESIGN AND ANALYSIS OF ALGORITHMS**

## **UE22CS241B**

---

**Bharathi R**

Department of Computer Science  
& Engineering

# DESIGN AND ANALYSIS OF ALGORITHMS

---

## Brute Force: Selection Sort

Major Slides Content: Anany Levitin

Department of Computer Science & Engineering

# DESIGN AND ANALYSIS OF ALGORITHMS

## Brute Force

---



When in doubt, use  
brute force.

Ken Thompson

 quotefancy

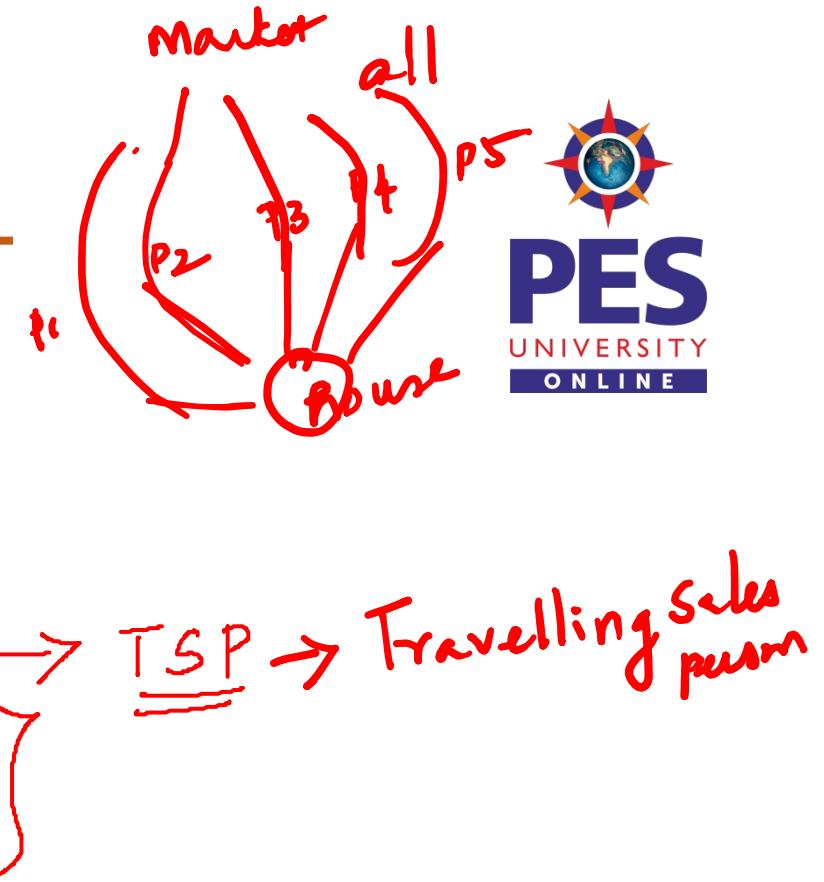
"Just do it".  
One of the easiest method apply.

- Brute Force is a straightforward approach to solving a problem, usually directly based on the problem statement and definitions of the concepts involved

# DESIGN AND ANALYSIS OF ALGORITHMS

## Brute Force

- In computer science, brute-force search or exhaustive search, also known as generate and test, is a very general problem-solving technique and algorithmic paradigm that consists of:
  - systematically enumerating all possible candidates for the solution
  - checking whether each candidate satisfies the problem's statement



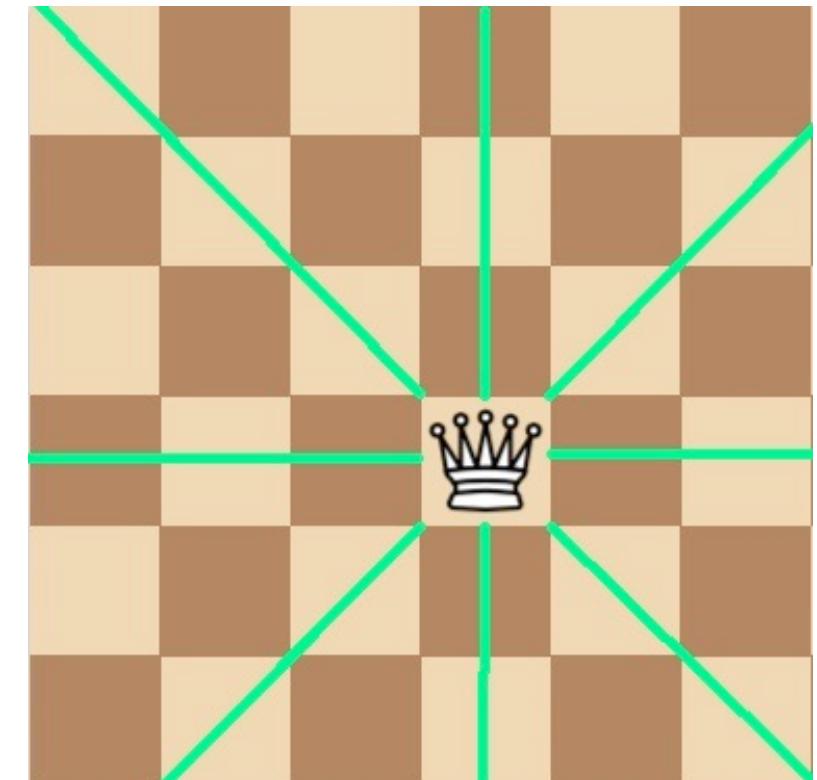
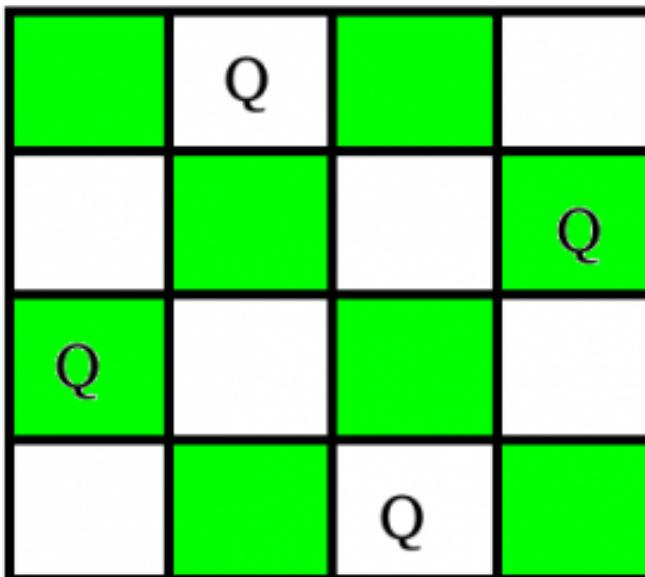
## Brute Force

---

- A brute-force algorithm to find the divisors of a natural number  $n$  would
  - enumerate all integers from 1 to  $n$
  - check whether each of them divides  $n$  without remainder
- A brute-force approach for the eight queens puzzle would
  - examine all possible arrangements of 8 pieces on the 64-square chessboard
  - check whether each (queen) piece can attack any other, for each arrangement
- The brute-force method for finding an item in a table (linear search)
  - checks all entries of the table, sequentially, with the item

## Brute Force

- A brute-force approach for the eight queens puzzle would
  - examine all possible arrangements of 8 pieces on the 64-square chessboard
  - check whether each (queen) piece can attack any other, for each arrangement



## Brute Force

---

- A brute-force search is simple to implement, and will always find a solution if it exists
- But, its cost is proportional to the number of candidate solutions – which in many practical problems tends to grow very quickly as the size of the problem increases (Combinatorial explosion)
- Brute-force search is typically used
  - when the problem size is limited
  - when there are problem-specific heuristics that can be used to reduce the set of candidate solutions to a manageable size
  - when the simplicity of implementation is more important than speed

# DESIGN AND ANALYSIS OF ALGORITHMS

## Brute Force Sorting Algorithms

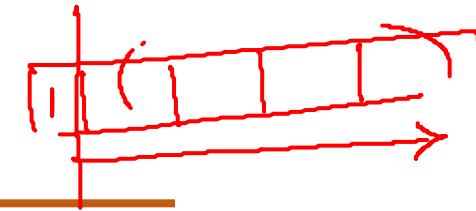
---



- Selection Sort
- Bubble Sort

# DESIGN AND ANALYSIS OF ALGORITHMS

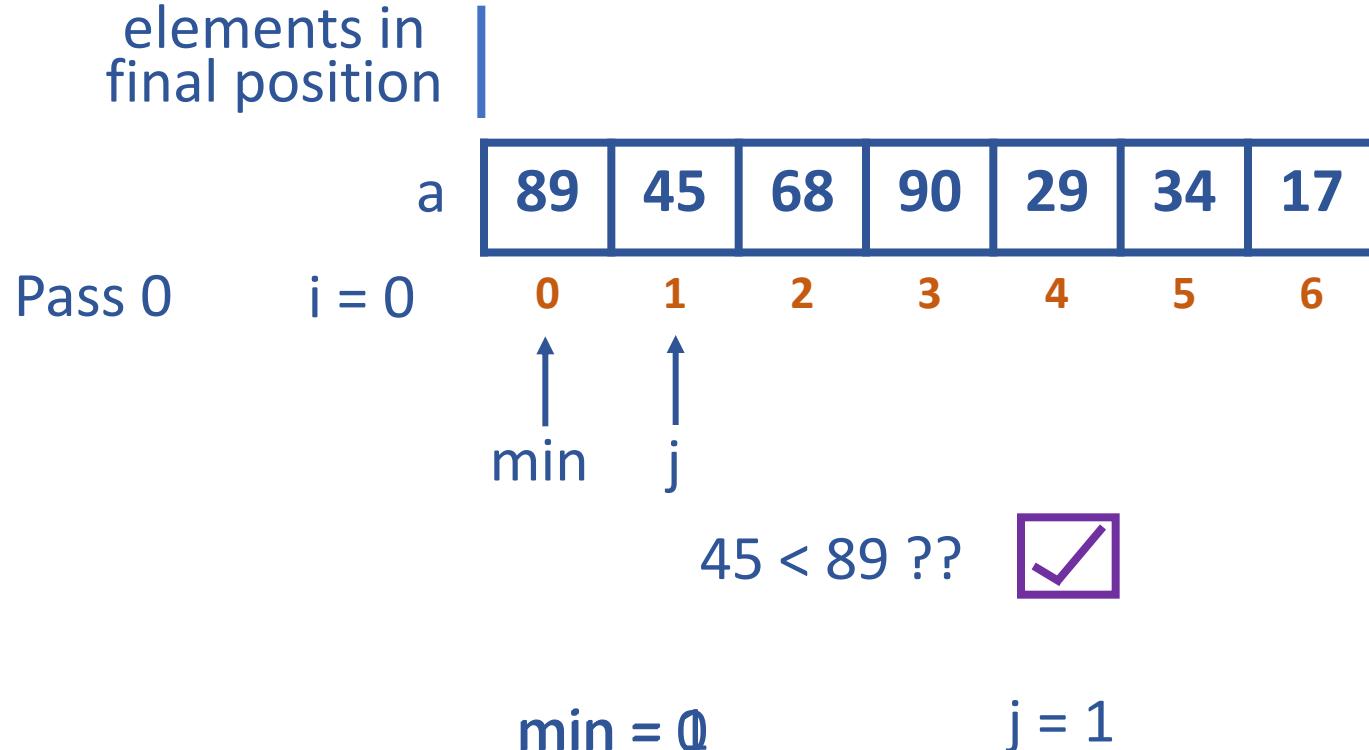
# Selection Sort



- Scan the array to find its smallest element and swap it with the first element, putting the smallest element in its final position in the sorted list.
  - Then, starting with the second element, scan the elements to the right of it to find the smallest among them and swap it with the second element, putting the second smallest element in its final position
  - Generally, on pass  $i$  ( $0 \leq i \leq n-2$ ), find the smallest element in  $A[i..n-1]$  and swap it with  $A[i]$

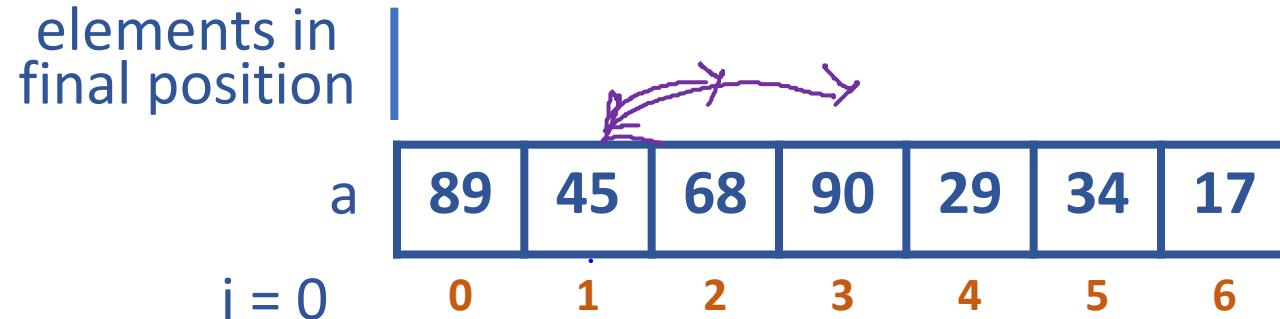


*n=7 elements  
0 to b → array size*



# DESIGN AND ANALYSIS OF ALGORITHMS

## Selection Sort



i }  $A[j] < A[min]$   
 $A[2] < A[1]$

$A[min] = 45$      $min = 1$

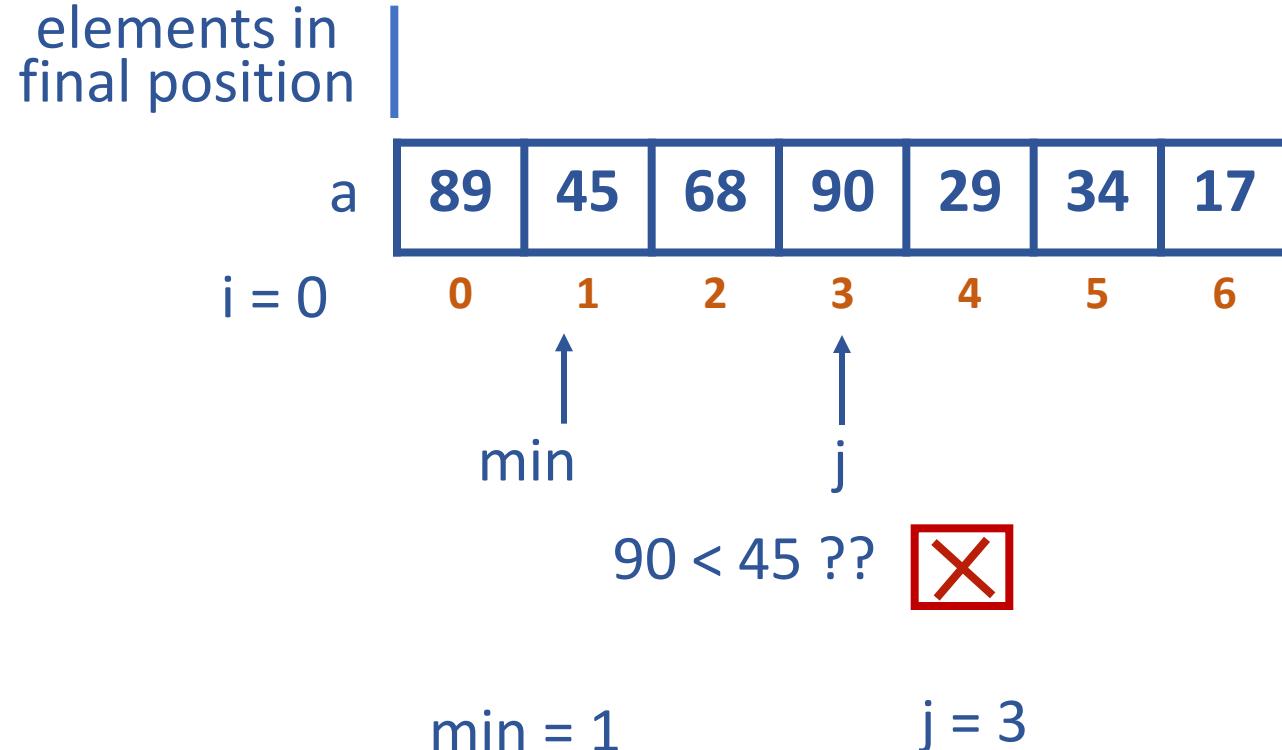
$68 < 45 ??$  

$min = 1$

$j = 2$

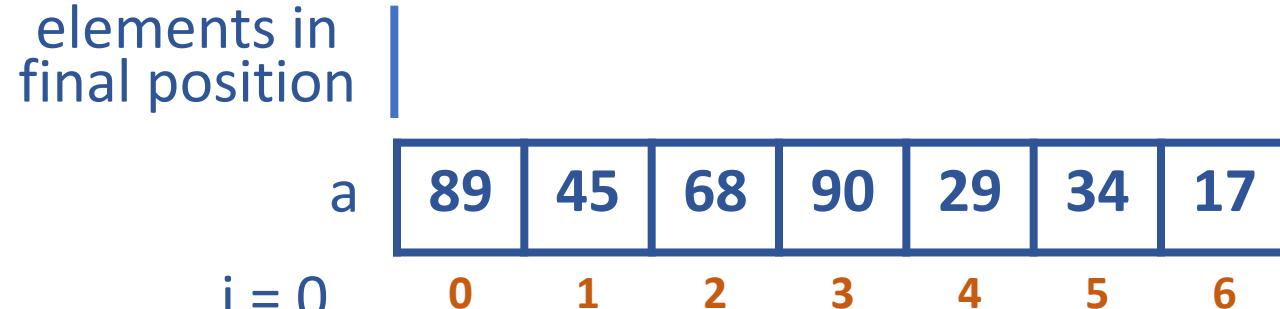
# DESIGN AND ANALYSIS OF ALGORITHMS

## Selection Sort



# DESIGN AND ANALYSIS OF ALGORITHMS

## Selection Sort

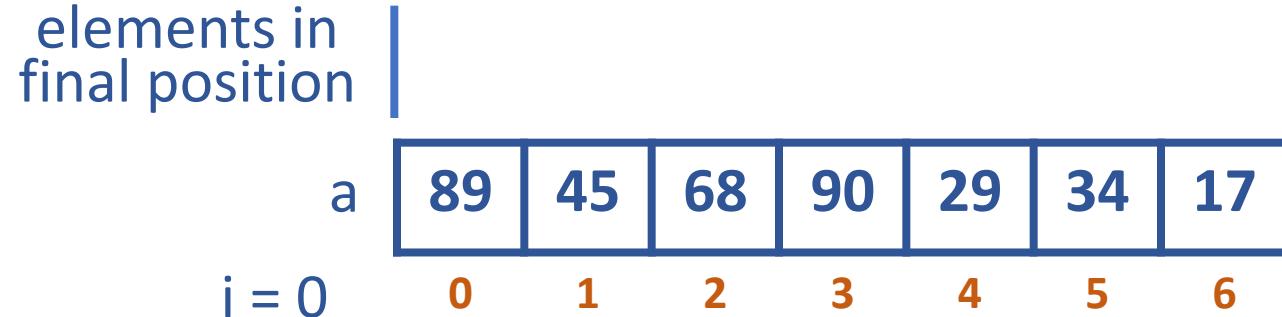


min = 4

j = 4

# DESIGN AND ANALYSIS OF ALGORITHMS

## Selection Sort



min ↑  
j ↑

$34 < 29 ??$  

$\text{min} = 4$

$j = 5$

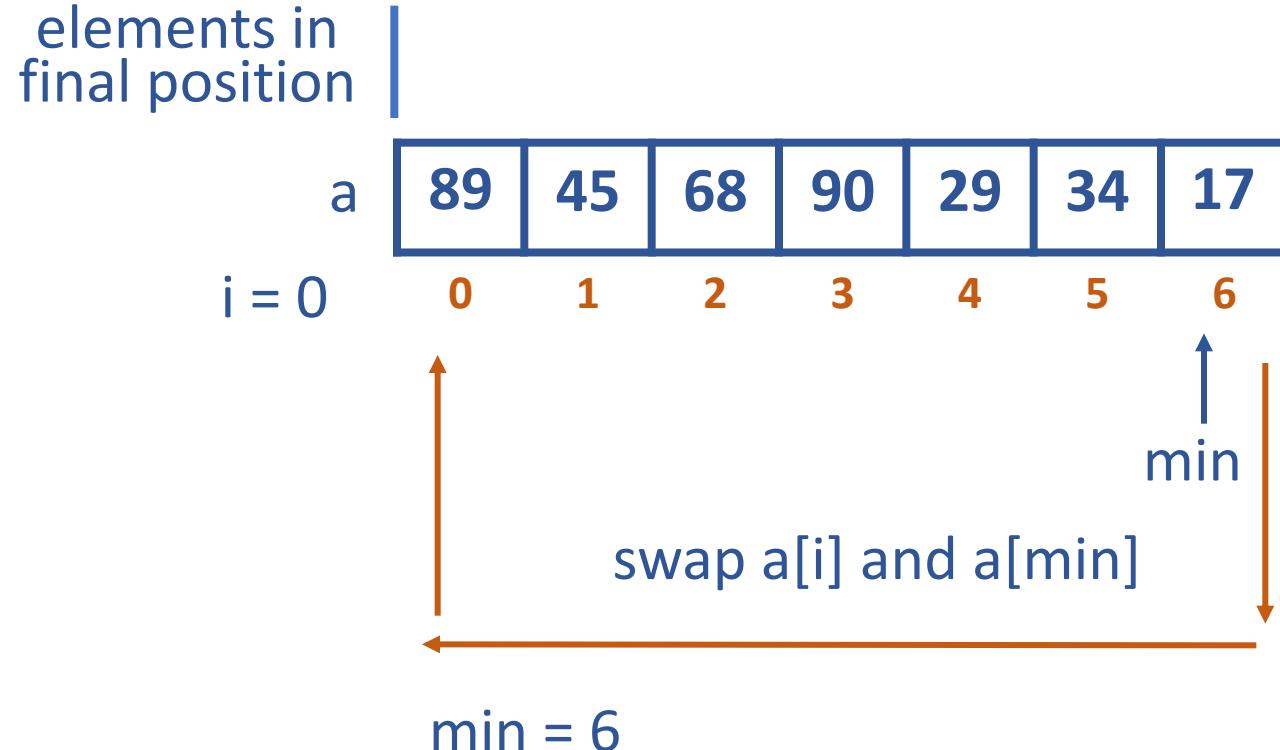
# DESIGN AND ANALYSIS OF ALGORITHMS

# Selection Sort



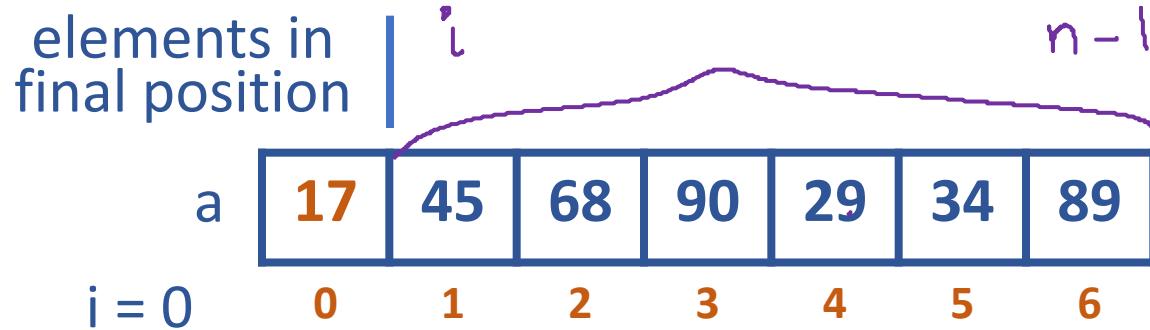
# DESIGN AND ANALYSIS OF ALGORITHMS

## Selection Sort



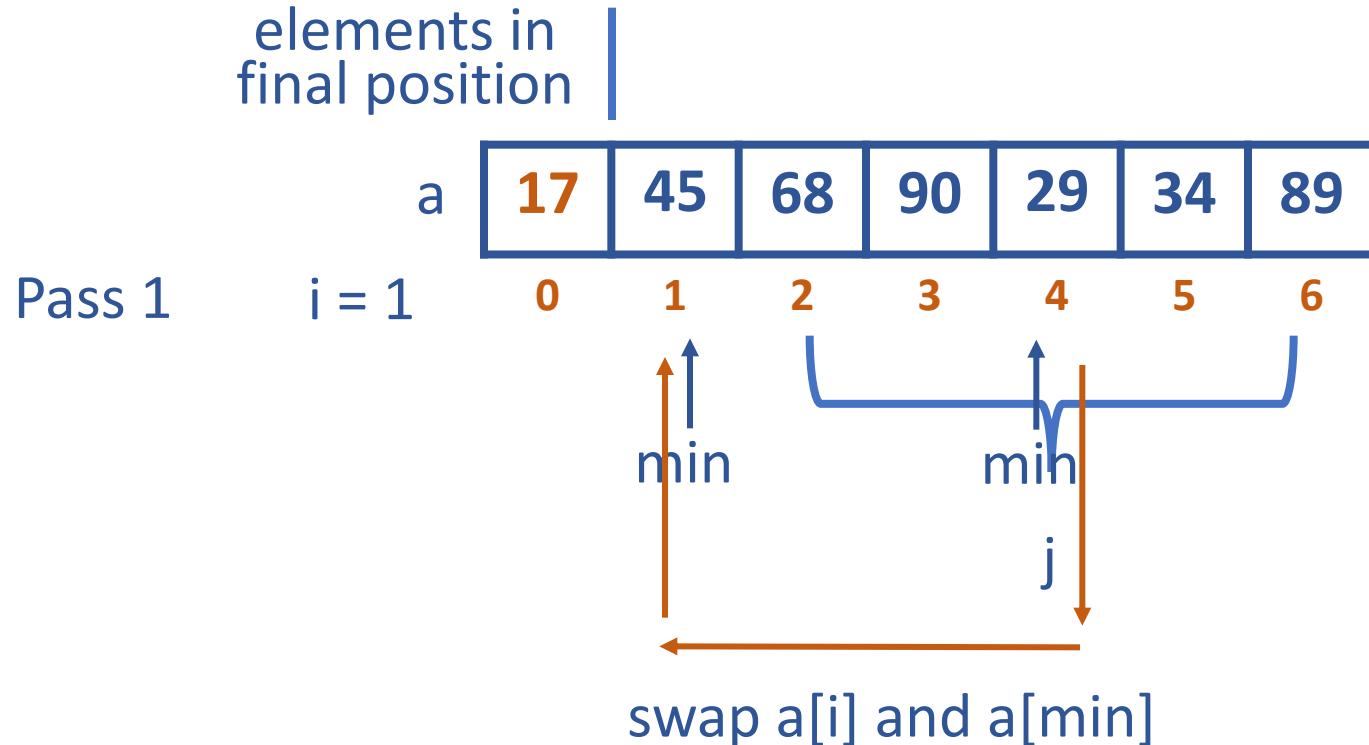
# DESIGN AND ANALYSIS OF ALGORITHMS

## Selection Sort



# DESIGN AND ANALYSIS OF ALGORITHMS

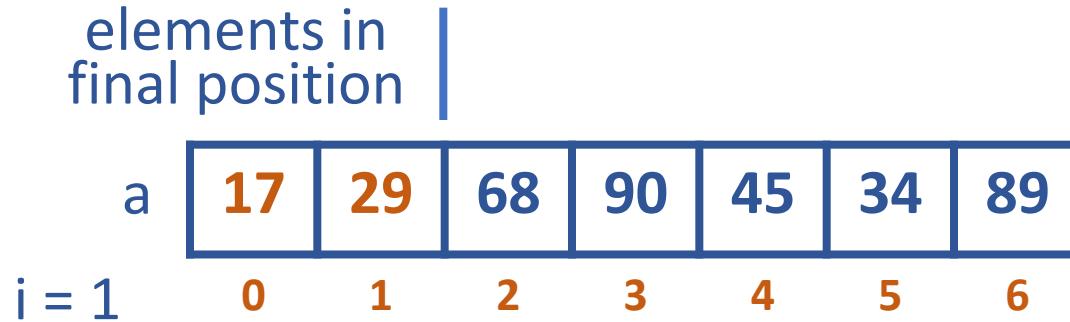
## Selection Sort



$$\min = 4$$

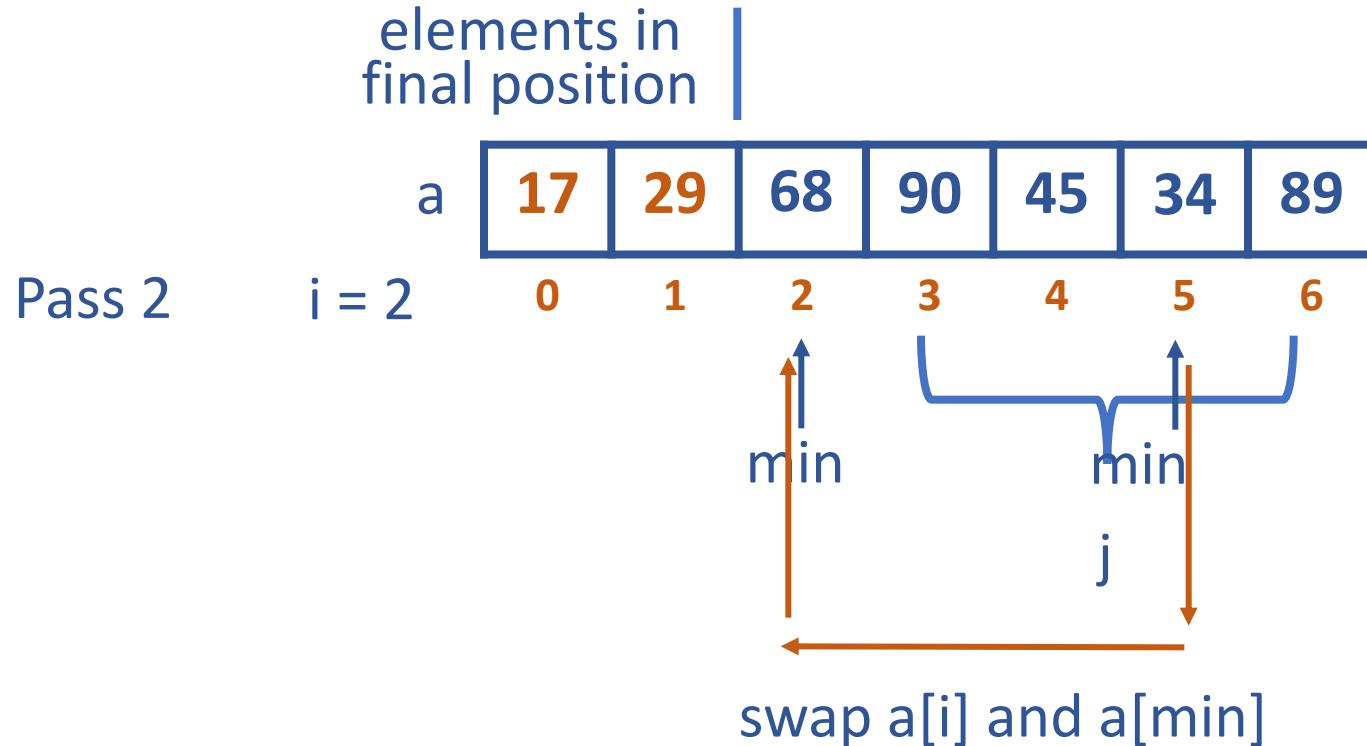
# DESIGN AND ANALYSIS OF ALGORITHMS

## Selection Sort



# DESIGN AND ANALYSIS OF ALGORITHMS

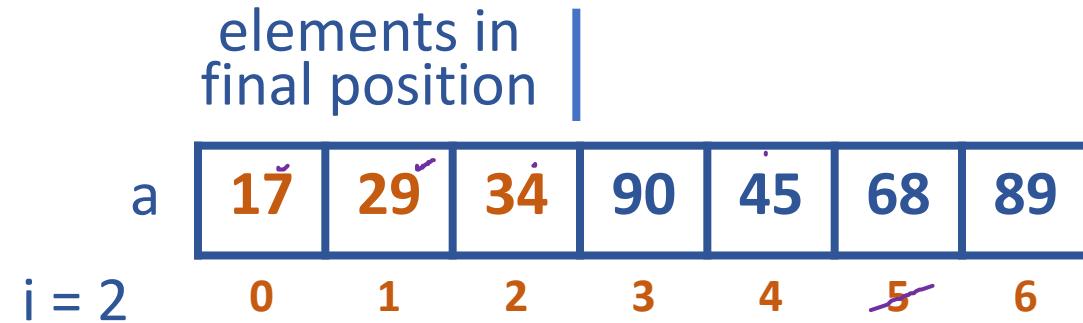
## Selection Sort



$min = 5$

# DESIGN AND ANALYSIS OF ALGORITHMS

## Selection Sort

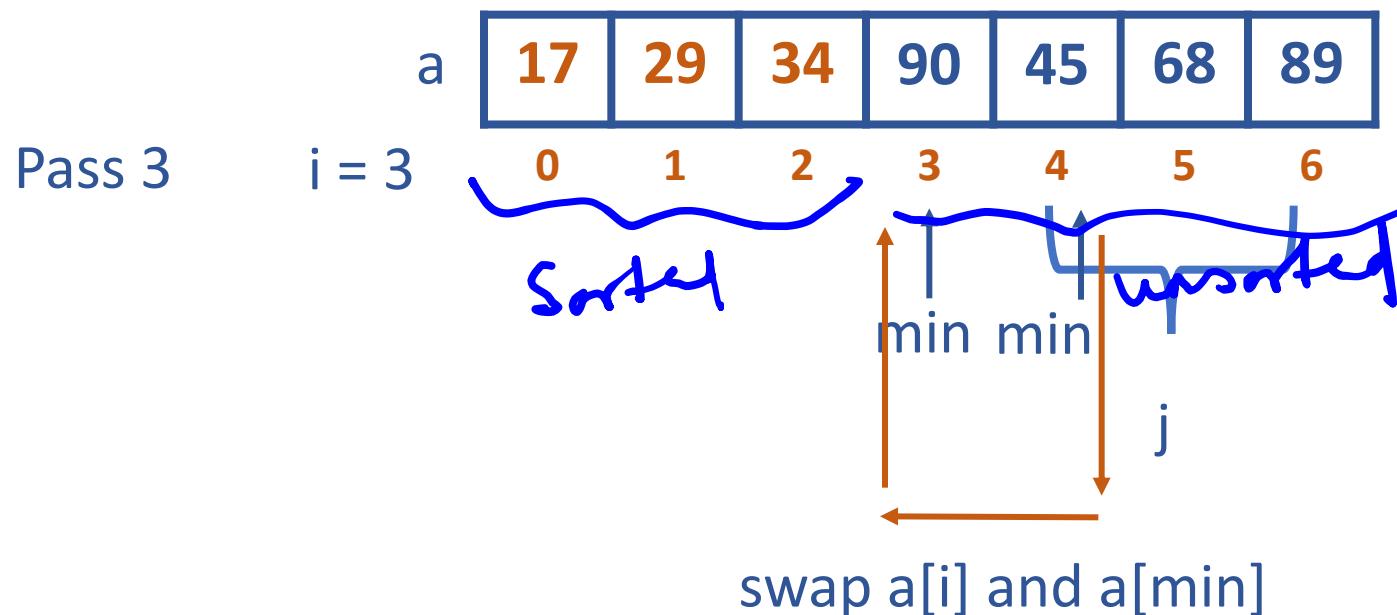


# DESIGN AND ANALYSIS OF ALGORITHMS

## Selection Sort

$A_0 \leq A_1 \leq \dots \leq A_{i-1}$  |  $A_i \dots A_{\min} \dots A_{n-1}$

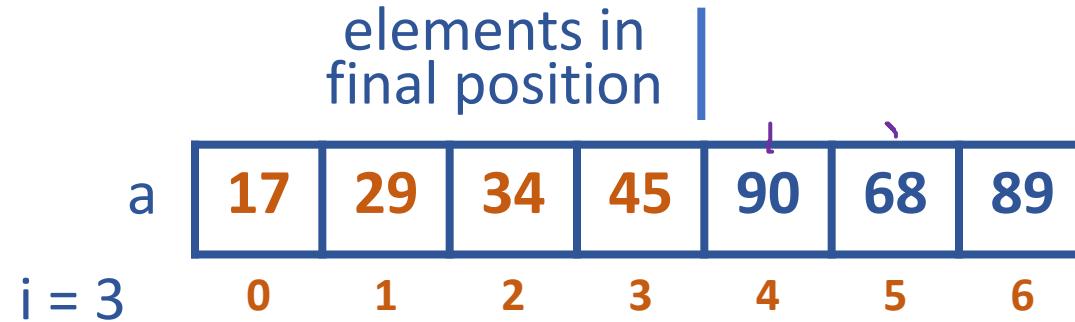
elements in final position

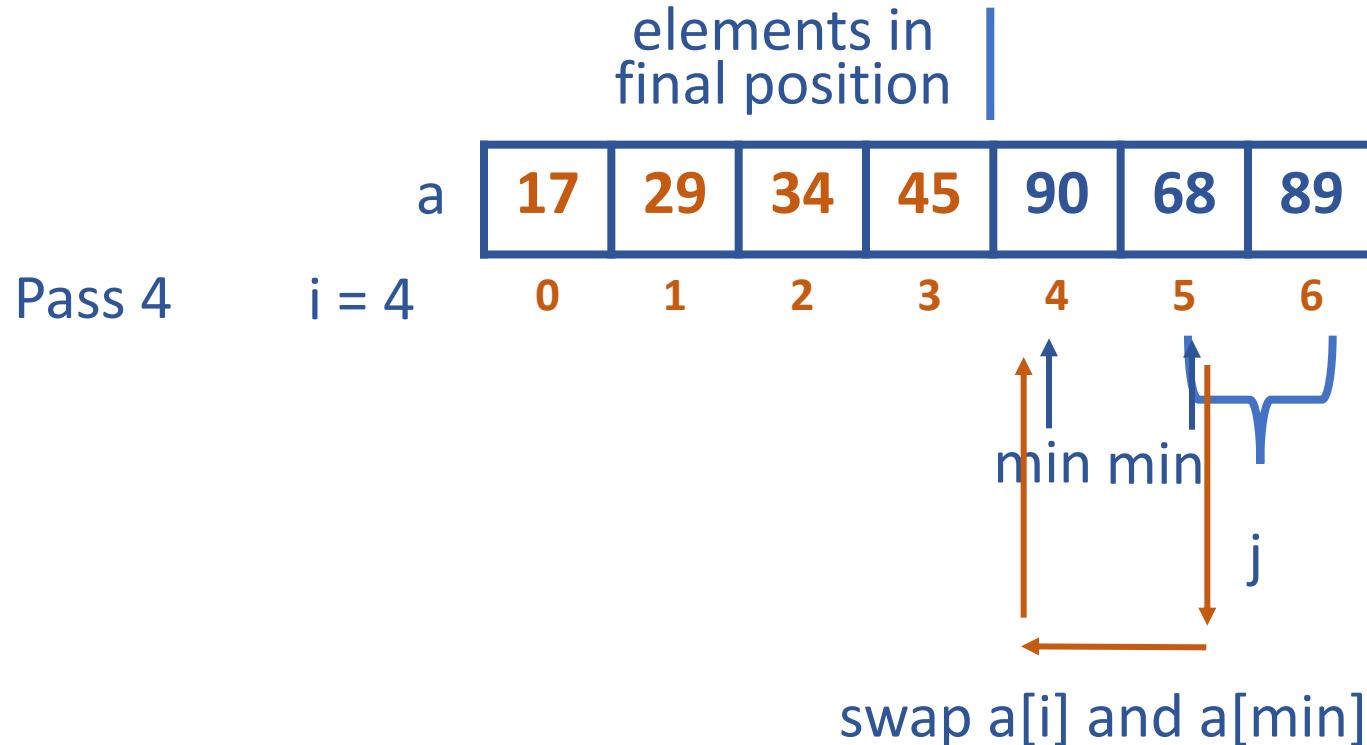


$$\min = 4$$

# DESIGN AND ANALYSIS OF ALGORITHMS

## Selection Sort

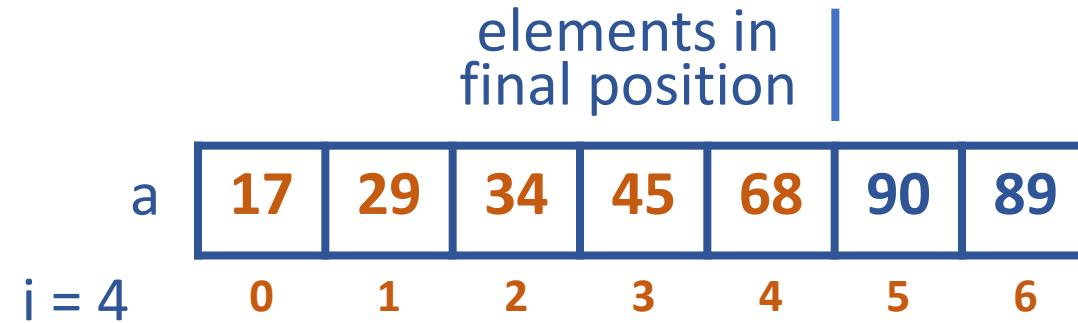


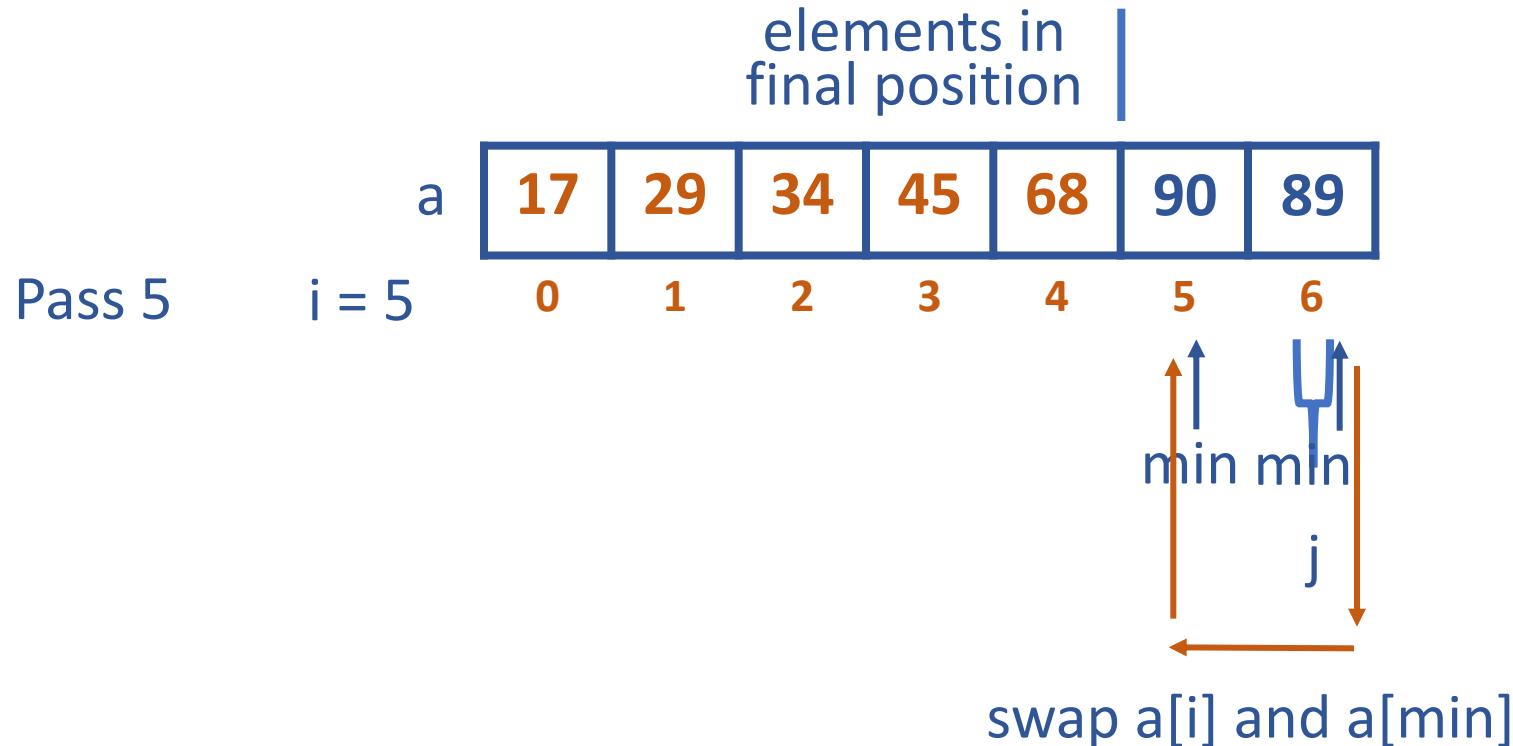


$\min = 5$

# DESIGN AND ANALYSIS OF ALGORITHMS

## Selection Sort

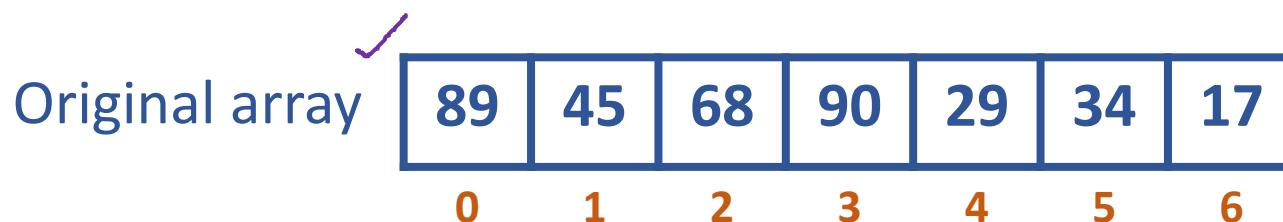
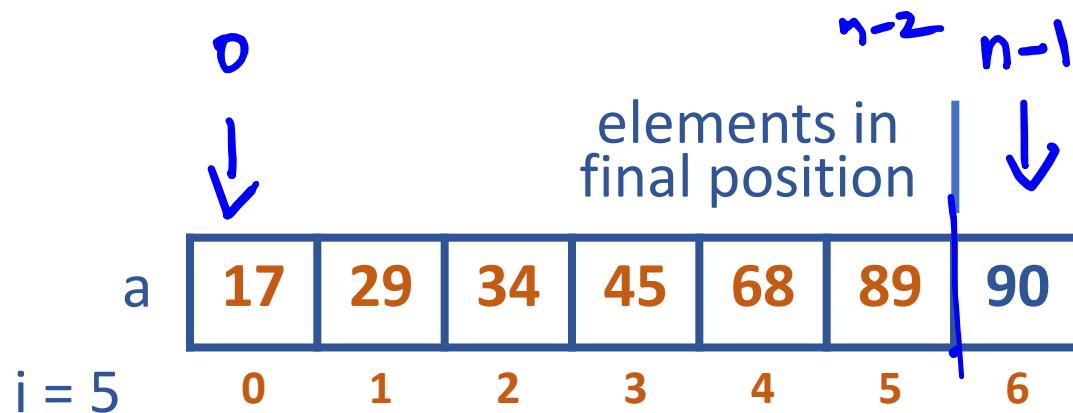




$min = 6$

# DESIGN AND ANALYSIS OF ALGORITHMS

## Selection Sort



ALGORITHM SelectionSort(A[0 .. n -1])

//Sorts a given array by selection sort

//Input: An array A[0 .. n - 1] of orderable elements

//Output: Array A[0 .. n - 1] sorted in ascending order

for i <- 0 to n - 2 do

    min <- i

        for j <- i+1 to n-1 do

            if A[j] < A[min] min <- j

            swap A[i] and A[min]

# DESIGN AND ANALYSIS OF ALGORITHMS

## Selection Sort

$$\frac{n^2}{2} - \frac{n}{2}$$

ALGORITHM SelectionSort(A[0 .. n - 1])

//Sorts a given array by selection sort

//Input: An array A[0 .. n - 1] of orderable elements

//Output: Array A[0 .. n - 1] sorted in ascending order

for i <- 0 to n - 2 do

    min <- i

    for j <- i+1 to n-1 do

        if A[j] < A[min] min <- j

    swap A[i] and A[min]

number of swaps =  $\Theta(n)$   
exactly  $(n-1) + \text{no}$

## Analysis

Basic op: Comparison  
i/p size: n

$$\begin{aligned}C(n) &= \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1 \\&= \sum_{i=0}^{n-2} [(n-1)-(i+1)+1] \\&= \sum_{i=0}^{n-2} (n-1-i) = \frac{(n-1)n}{2}\end{aligned}$$



### Selection Sort Analysis

$$C(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1 = \sum_{i=0}^{n-2} (n - 1 - i) = \frac{(n - 1)n}{2}$$

No best & worst case

Selection Sort is a  $\Theta(n^2)$  algorithm  $\Rightarrow$  w.r.t. Comparison

## Selection Sort

### Selection Sort:

Example: 8 4 6 9 2 3 1



# DESIGN AND ANALYSIS OF ALGORITHMS

Selection Sort

Is selection sort stable?



Selection Sort:

Example: 8 4 6 9 2 3 1

Example 2  
there 2 equal  
elements

The two equal elements  
 $b^2$  &  $b^3$  are in the position  
2 & 3 where  $2 < 3$ .

then after sorting is the  
order is maintained

1 2 3 4  $b^2$   $b^3$  8  $\rightarrow$  Stable

or

1 2 3 4  $b^3$   $b^2$  8  $\rightarrow$  unstable

### Stable Vs Unstable Sorts



Unsorted Array



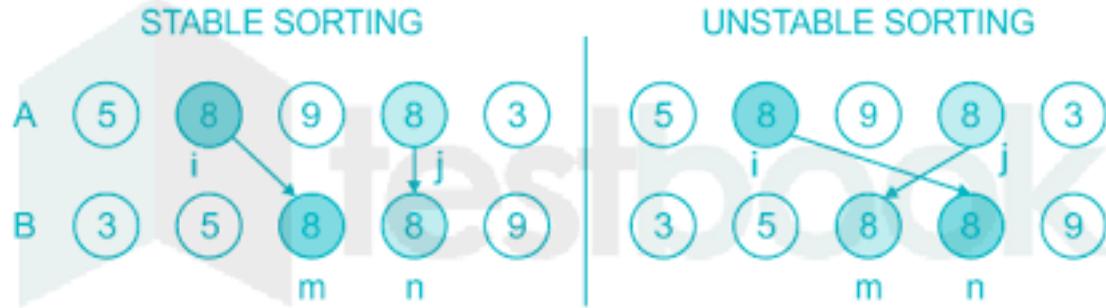
Stable Sort



Unstable Sort

# DESIGN AND ANALYSIS OF ALGORITHMS

## Examples



**Additional Slides**

# DESIGN AND ANALYSIS OF ALGORITHMS

Examples Time complexity analysis



PES  
UNIVERSITY  
ONLINE

1. A loop incrementing / decrementing by a constant

- i) `for(i=0 ; i<n ; i=i+c)` // incrementing loop  $T(n) \in \Theta(n)$   
  { // do something  
  }  
  
ii) `for(i=n ; i>1 ; i=i-c)` // decrementing loop  $\Rightarrow T(n) = \Theta(n)$   
  { // do something;  
  }  
  
iii) `i=0`  
     `while(i < n)`  
      { // do something  
      }  
         `i++` // incrementing loop  $\Rightarrow T(n) = \Theta(n)$

# DESIGN AND ANALYSIS OF ALGORITHMS

## Examples

2. A loop incrementing /decrementing by a constant factor

i) `for ( i=1 ; i<n ; i = i * 2 )`  
{     // statement  
}

// incrementing    $T(n) \in \Theta(\log_2 n)$   
 $\log_2 n$

ii) `for ( i=n ; i>0 ; i = i / 2 )`  
{     // statement  
}

// decrementing    $T(n) \in \Theta(\log_2 n)$

iii)    $i=1$   
`while ( i < n )`  
{     // st  
    *(\*)*    $i = i * 2$

// incrementing    $T(n) \in \Theta(\log_2 n)$

# DESIGN AND ANALYSIS OF ALGORITHMS

## Examples



PES  
UNIVERSITY  
ONLINE

3. A loop running constant multiple of n times

```
for( int i=1 ; i<=2n ;  
     i<=3*n ; i = i+1 )  
{ // do something  
}
```

$n=10 \rightarrow 30$  times

It belongs to class of linear algorithms.

$$T(n) \leq \Theta(g(n))$$

$$T(n) \in \Theta(n)$$

$$T(n) \in \Theta(2n) \approx \Theta(n)$$

$$\in \Theta(3n) \approx \Theta(n)$$

Order of growth is  
still linear

# DESIGN AND ANALYSIS OF ALGORITHMS

## Examples

### 4. Consecutive Single loops

for( $i=0$  ;  $i < m$  ;  $i++$ )  $\rightarrow$  loop 1

{ // do something

}

for( $i=0$  ;  $i < n$  ;  $i++$ )  $\rightarrow$  loop 2

{ // do something

}

$T(n) = T(n)$  of loop 1 +  $T(n)$  of loop 2

$$= \Theta(m) + \Theta(n) = \boxed{2\Theta(m)} + \Theta(m+n) \approx \Theta(n)$$

## Examples

5. Loop incrementing by constant power

```
for( int i=2 ; i<=n ; i = power(i, c) )
{
    // do something ;
}
```

Incrementing factor.

I	iteration	$i = 2$	$\Rightarrow i = 2$
II	"	$i = 2^1 c$	$\Rightarrow i = 2^2 = 4$
III	"	$i = 2^1 c^1 \times 2^2 c^2 \Rightarrow i = (2^2)^2 = 8$	
IV	"	$i = 2^1 c^3$	$= 2^2 \times 2^8 =$
i	iteration	$i = 2^1 c^i$	

# DESIGN AND ANALYSIS OF ALGORITHMS

## Examples

the loop will terminate when  $i > n$

$$2^c i > n$$

Take log on both sides (with base 2)

$$\log_2 2^c i = \log_2 n$$

$$c i \log_2 2 = \log_2 n$$

$$c i = \log_2 n$$

$$c^i = \log_2 n$$

# DESIGN AND ANALYSIS OF ALGORITHMS

## Examples

$c^i = \log_2 n$   
Again take  $\log_2$  on both sides

$$i = \log_2 c (\log_2 n)$$

$$i \in \log(c \log n)$$

# DESIGN AND ANALYSIS OF ALGORITHMS

## Examples



PES  
UNIVERSITY  
ONLINE

b.  $i \leftarrow 1$

while  $i \leq n$  do .

<do something>

$i \leftarrow i * c$  // c is some constant

- a) what is the least value of c ?  $c \geq 2$
- b) what is the number of operations in terms of  $n + C$  ?
- c) Would the asymptotic complexity depend on the particular legal value of c ?

$\log_2^n, \log_{10}^n, \log_3^n, \dots$

$T(n)$

# DESIGN AND ANALYSIS OF ALGORITHMS

## Examples

7. function()

```
i=1, s=1;
```

```
while(s <= n)
```

```
{ i++; }
```

```
s = s + i;
```

```
printf("Welcome");
```

```
}
```

## Iteration

i	s
1	$1 = 1$

2	$3 = 1+2$
---	-----------

3	$6 = 1+2+3$
---	-------------

4	$10 = 1+2+3+4 = \frac{4(5)}{2} = \frac{k(k+1)}{2}$
---	--

the loop will terminate when

$$s > n$$

$$\frac{k(k+1)}{2} > n$$

$$\frac{k^2+k}{2} > n$$

$$\therefore k \approx \Theta(n)$$

How many times "Welcome" will be printed.  $\rightarrow$   
 Analyze the time complexity of the algorithm.

# DESIGN AND ANALYSIS OF ALGORITHMS

## Examples

8) function ()

```
{ for(i=1; i<=n; i++) // independent loop.
```

```
    for(j=1; j<=n; j=j+i) // dependent loop
```

```
        printf("Welcome") ;
```

```
}
```

i = 1      j = <sup>Welcome</sup>  
              |  
              | To n times

i = 2      n/2 times

i = 3      n/3 times

i = 4      n/4 times

i = n      n/n times = 1 time.

$$1 + \frac{1}{2} + \frac{1}{3} + \dots + \frac{1}{n} = \log n .$$

n = 10, 20, ~~2000~~, 1998.

$$T(n) = 2 \left( n + \frac{n}{2} + \frac{n}{3} + \dots + \frac{n}{n} \right) \text{ times}$$

$$= n \left( 1 + \frac{1}{2} + \frac{1}{3} + \dots + \frac{1}{n} \right)$$

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

$$= 1.39 \log_{\frac{1}{2}} 10 = 33.4$$

# DESIGN AND ANALYSIS OF ALGORITHMS

## Examples

---



**PES**  
UNIVERSITY  
ONLINE

1. Execute C program to find fibonacci series . Count the number of recursive calls.
2. Execute C <sup>recursive</sup> program for Tower of Hanoi without return statement.



**THANK YOU**

---

**Bharathi R**

Department of Computer Science  
& Engineering



# **DESIGN AND ANALYSIS OF ALGORITHMS**

## **UE22CS241B**

---

**Bharathi R**

Department of Computer Science  
& Engineering

Brute Force → Straight forward approach.

## DESIGN AND ANALYSIS OF ALGORITHMS

---

### Bubble Sort

Major Slides Content: Anany Levitin



Shylaja S S

Department of Computer Science & Engineering

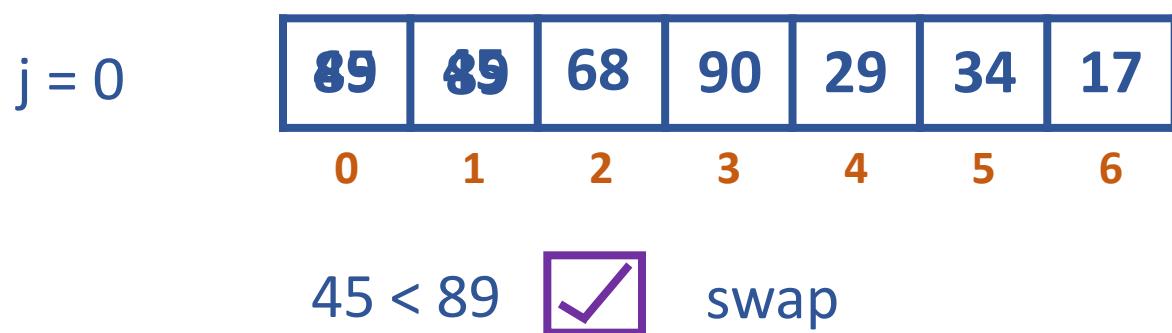
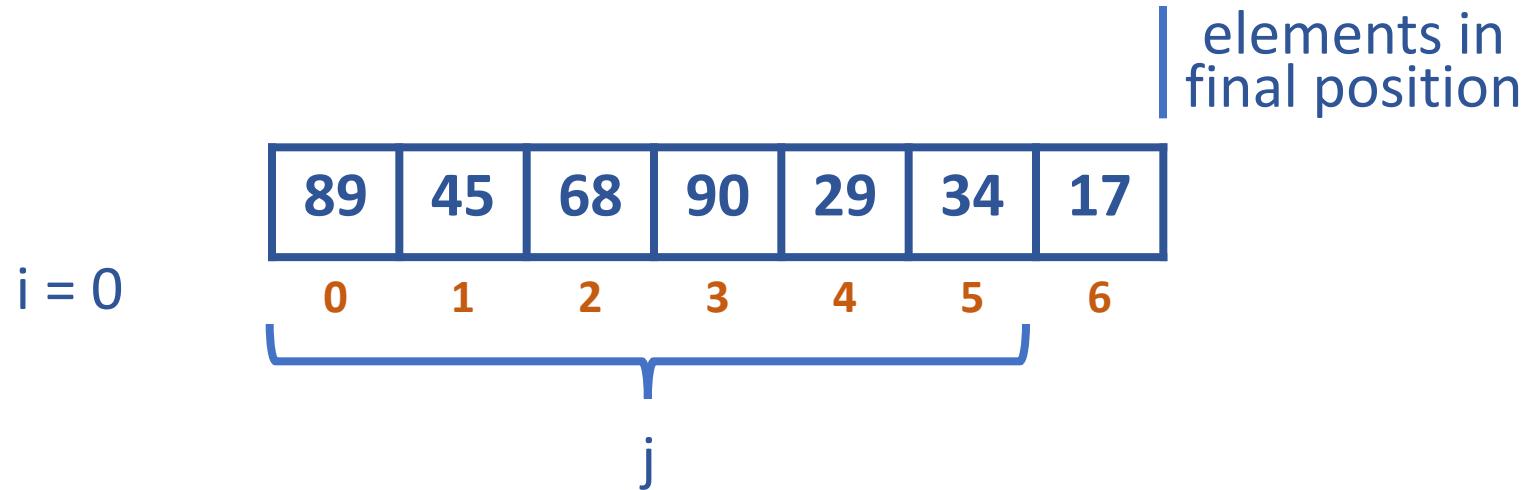
- Compare adjacent elements of the list and exchange them if they are out of order
- By doing it repeatedly, we end up bubbling the largest element to the last position on the list
- The next pass bubbles up the second largest element and so on and after  $n - 1$  passes, the list is sorted
- Pass  $i$  ( $0 \leq i \leq n - 2$ ) can be represented as follows:

$A[0], A[1], A[2], \dots, A[j] \leftrightarrow ? A[j+1], \dots, A[n-i-1] \mid A[n-i] \leq \dots \leq A[n-1]$

in their final positions

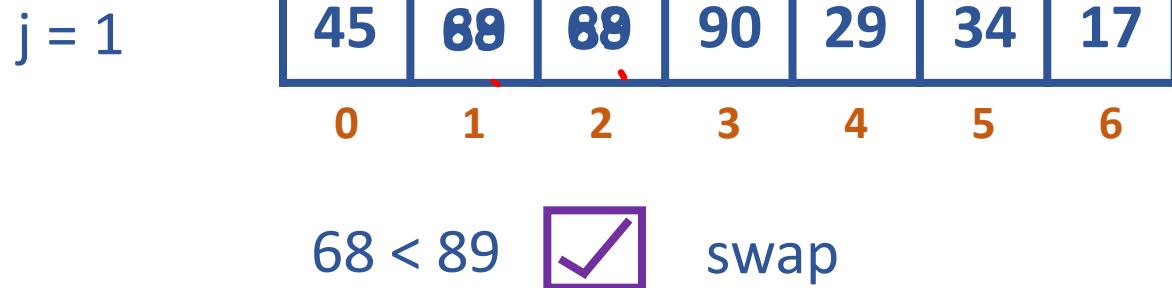
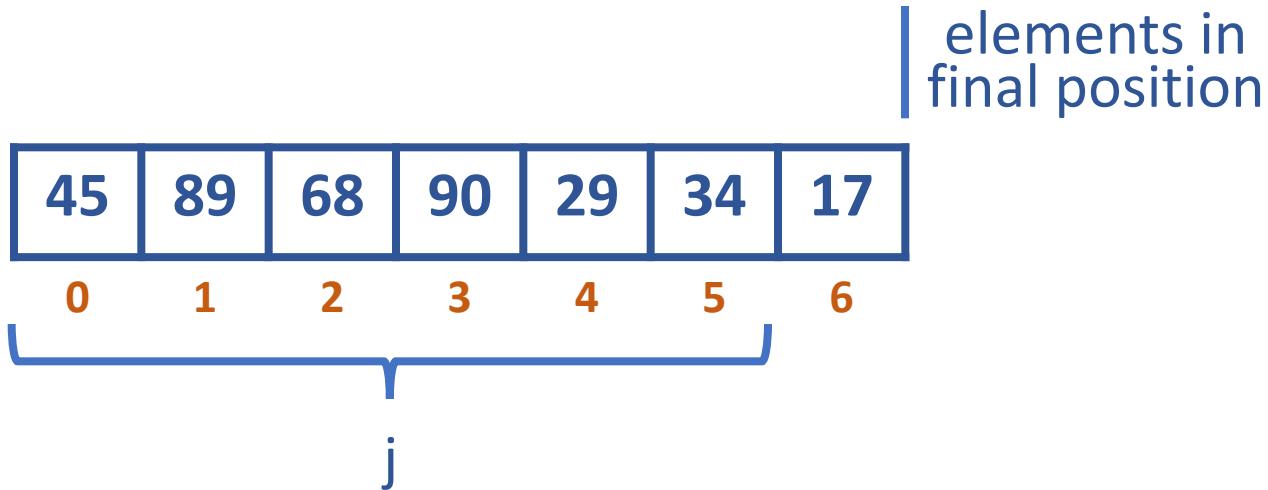
# DESIGN AND ANALYSIS OF ALGORITHMS

## Bubble Sort



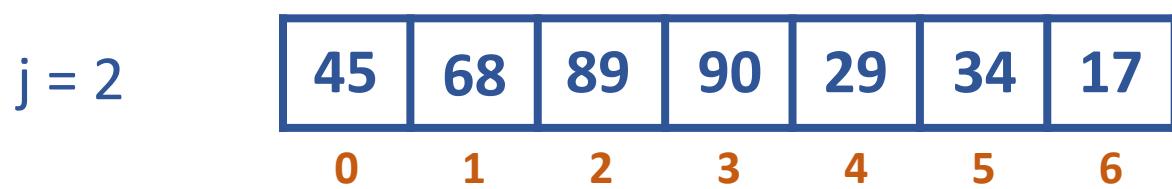
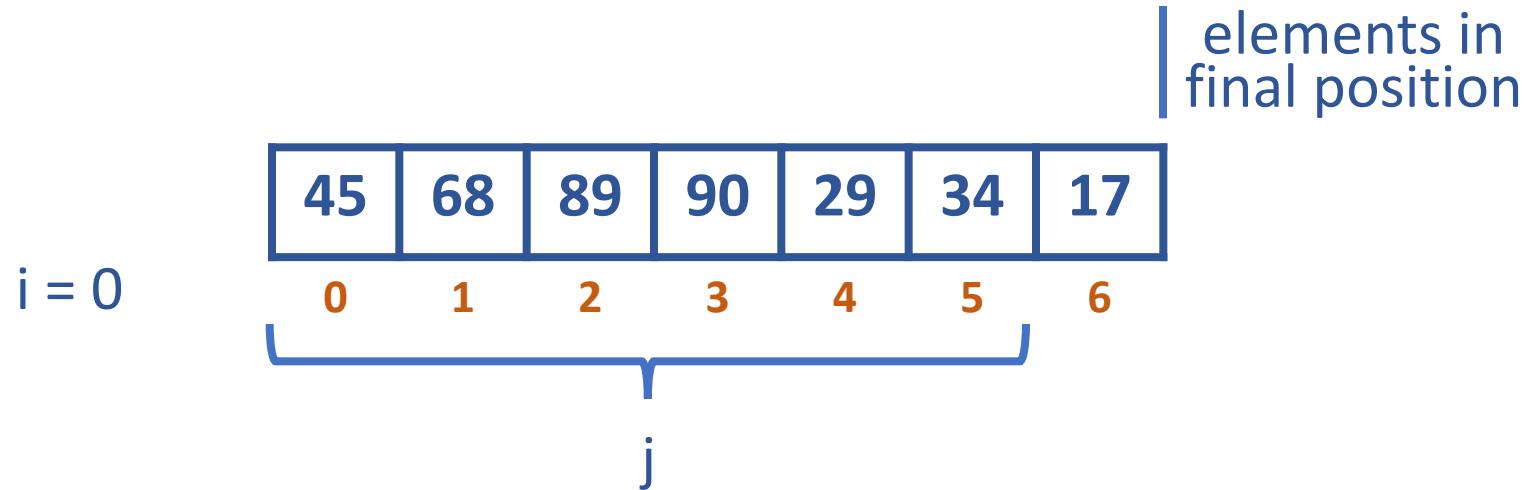
# DESIGN AND ANALYSIS OF ALGORITHMS

## Bubble Sort



# DESIGN AND ANALYSIS OF ALGORITHMS

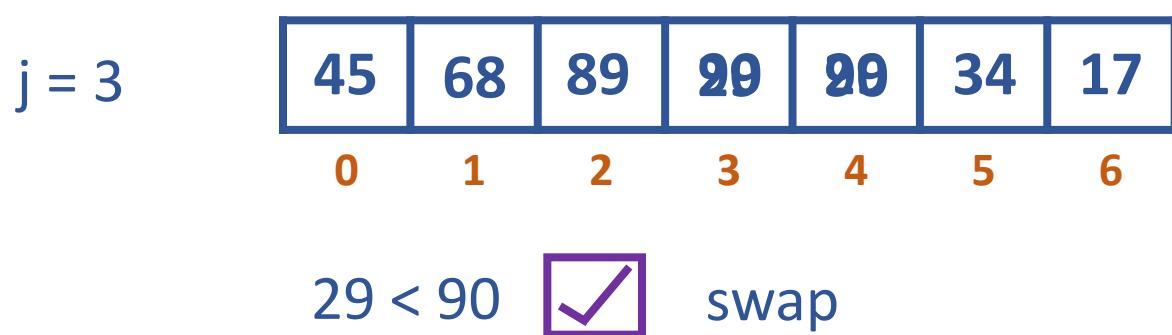
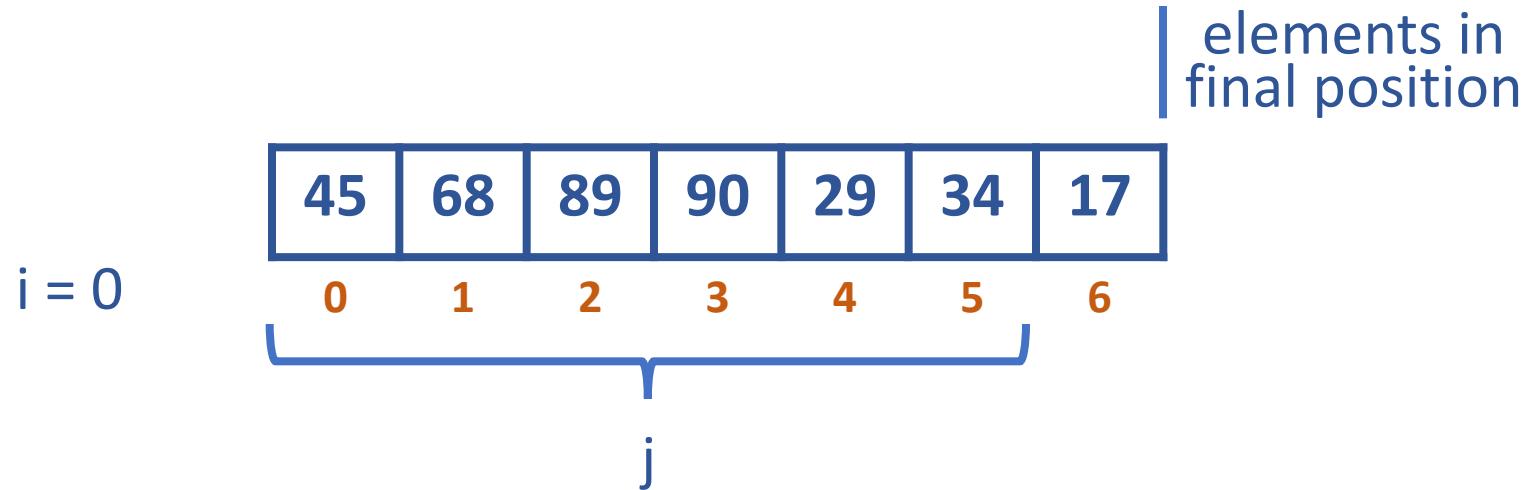
## Bubble Sort



90 < 89  no swap

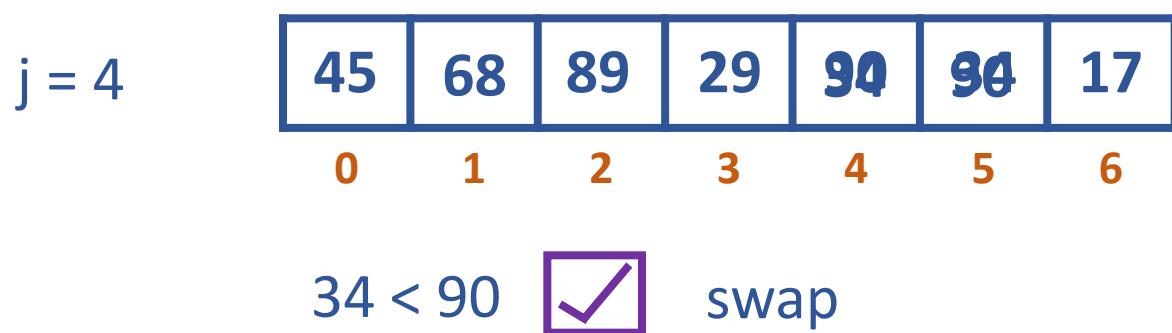
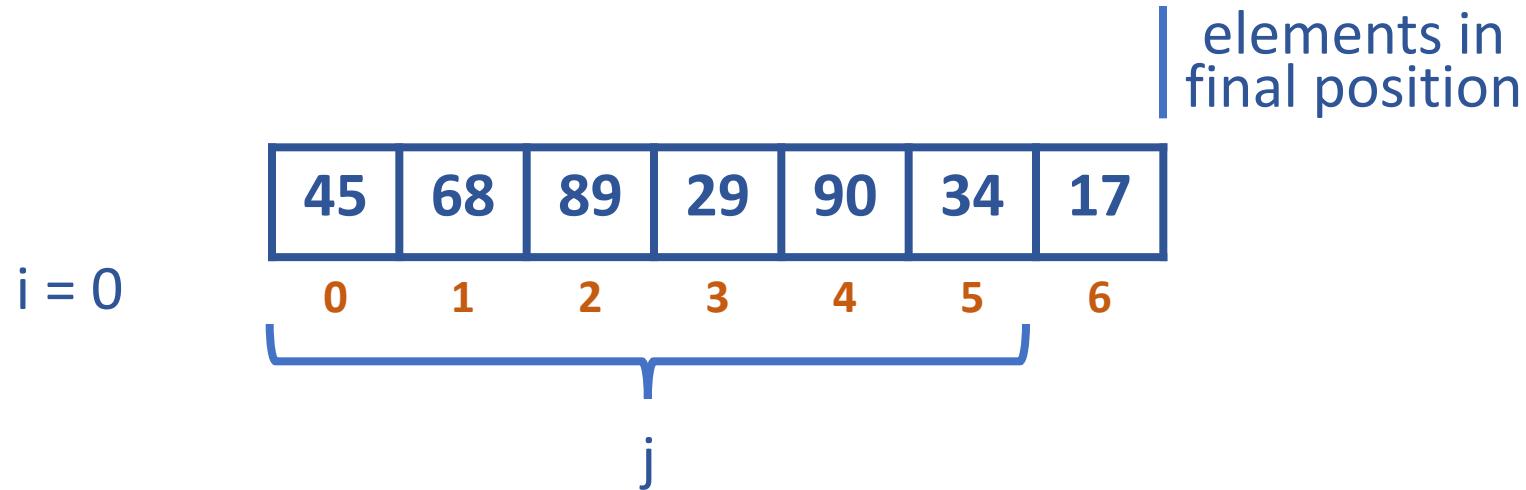
# DESIGN AND ANALYSIS OF ALGORITHMS

## Bubble Sort



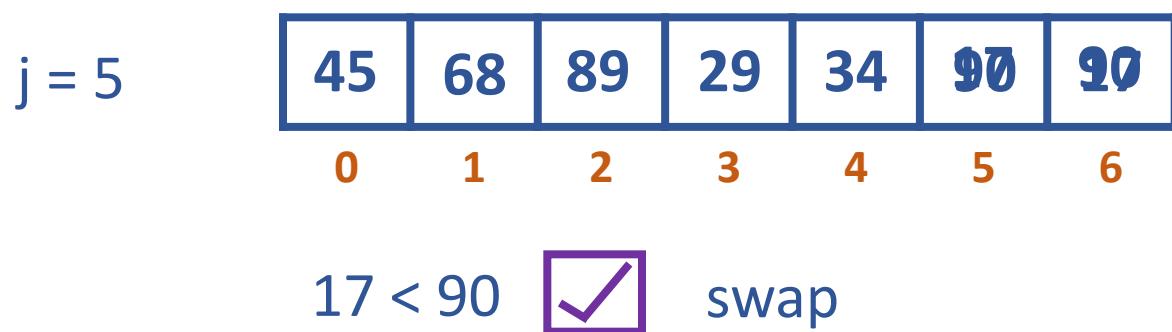
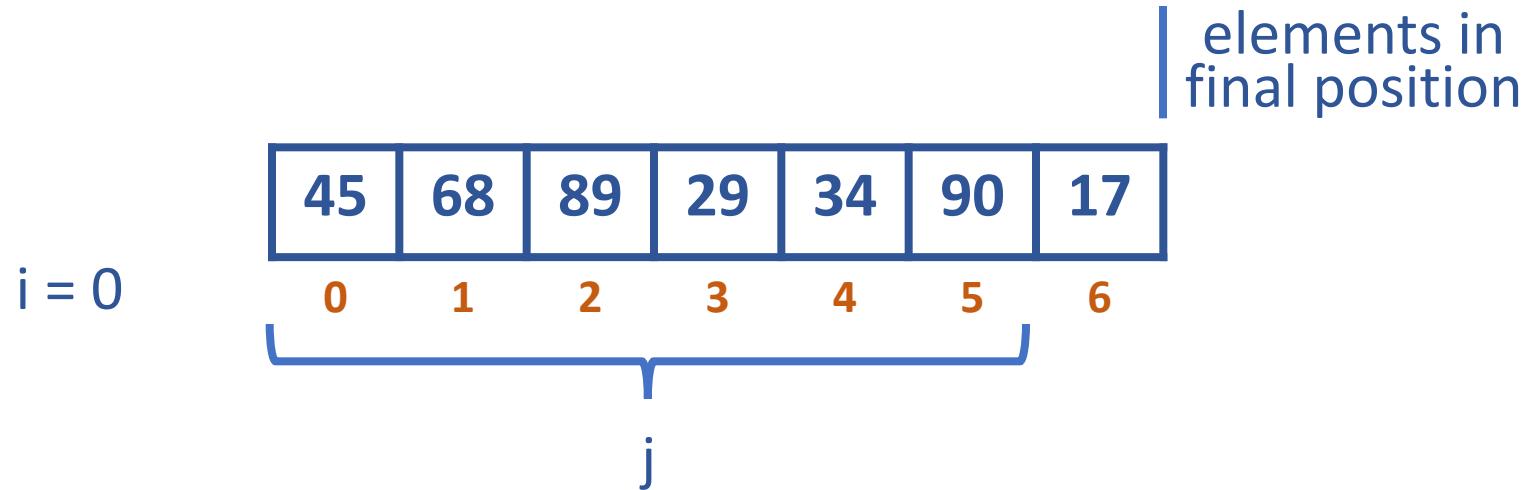
# DESIGN AND ANALYSIS OF ALGORITHMS

## Bubble Sort



# DESIGN AND ANALYSIS OF ALGORITHMS

## Bubble Sort



# DESIGN AND ANALYSIS OF ALGORITHMS

## Bubble Sort

After  
first  
iteration

$i = 1$

elements in  
final position

45	68	89	29	34	17	90
----	----	----	----	----	----	----

0 1 2 3 4 5 6



j

$j = 0$

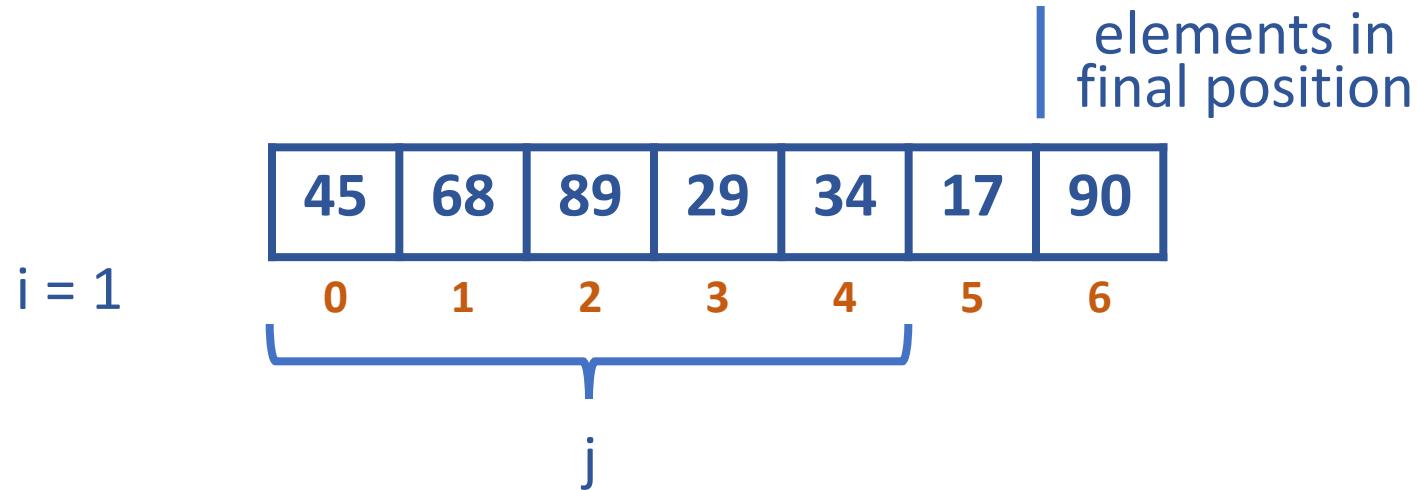
45	68	89	29	34	17	90
----	----	----	----	----	----	----

0 1 2 3 4 5 6

$68 < 45$   no swap

# DESIGN AND ANALYSIS OF ALGORITHMS

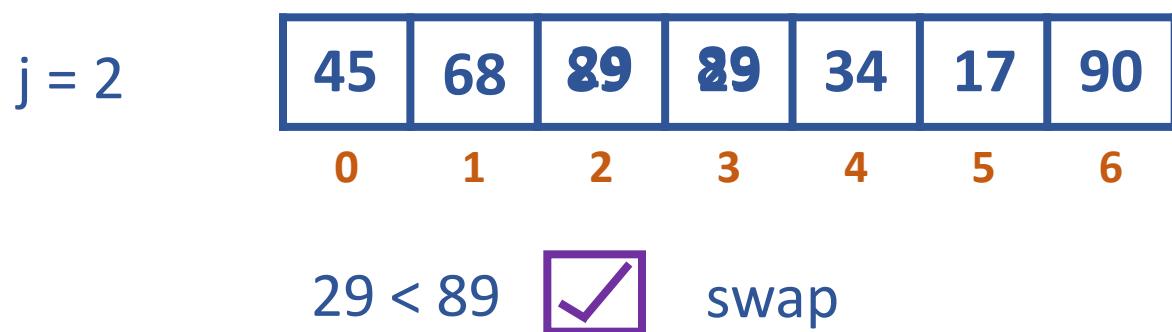
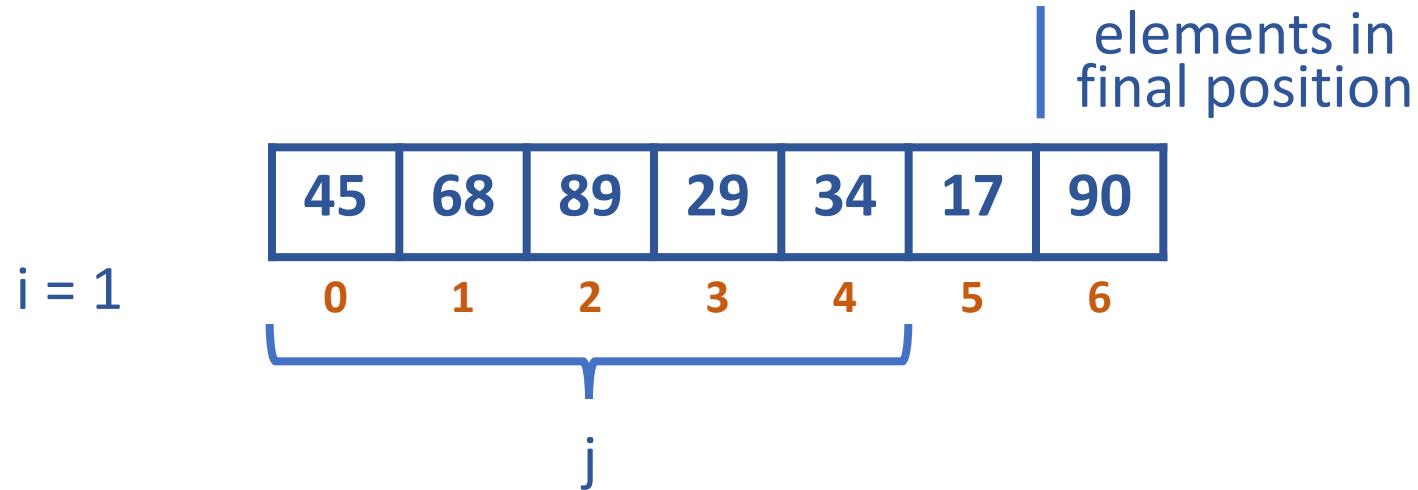
## Bubble Sort



$89 < 68$   no swap

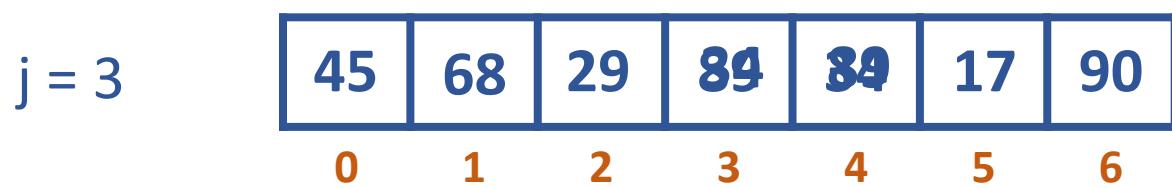
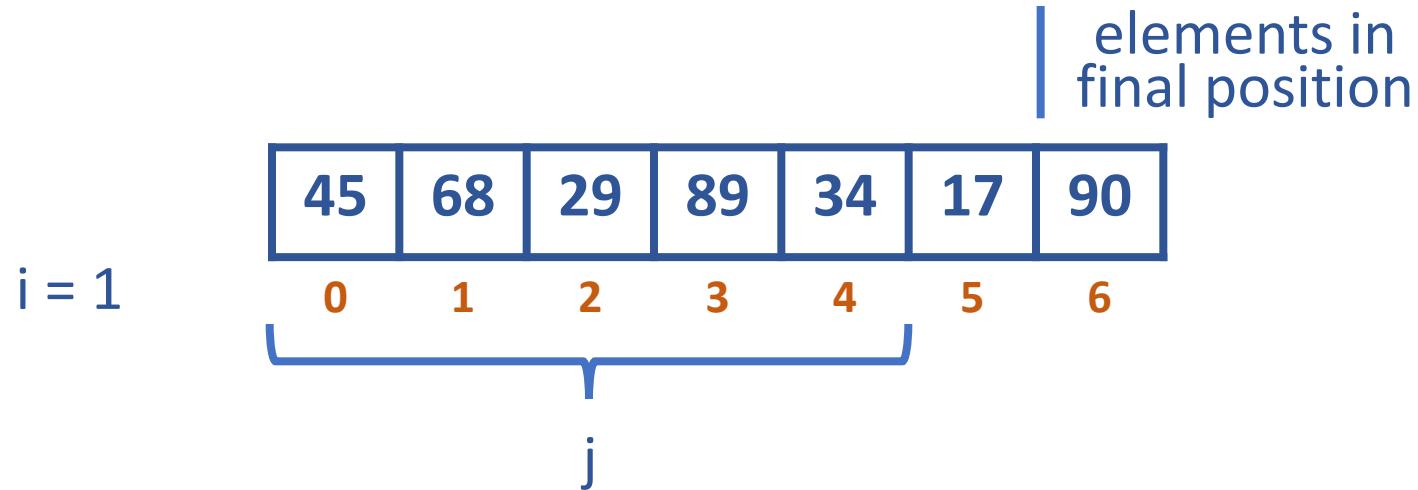
# DESIGN AND ANALYSIS OF ALGORITHMS

## Bubble Sort



# DESIGN AND ANALYSIS OF ALGORITHMS

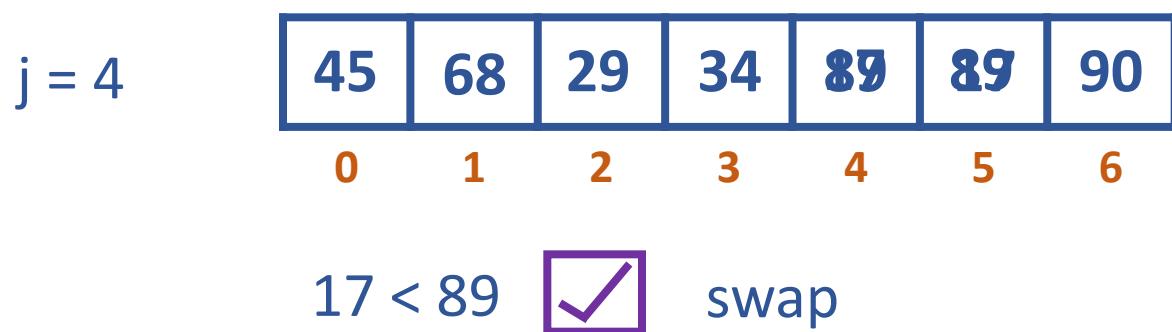
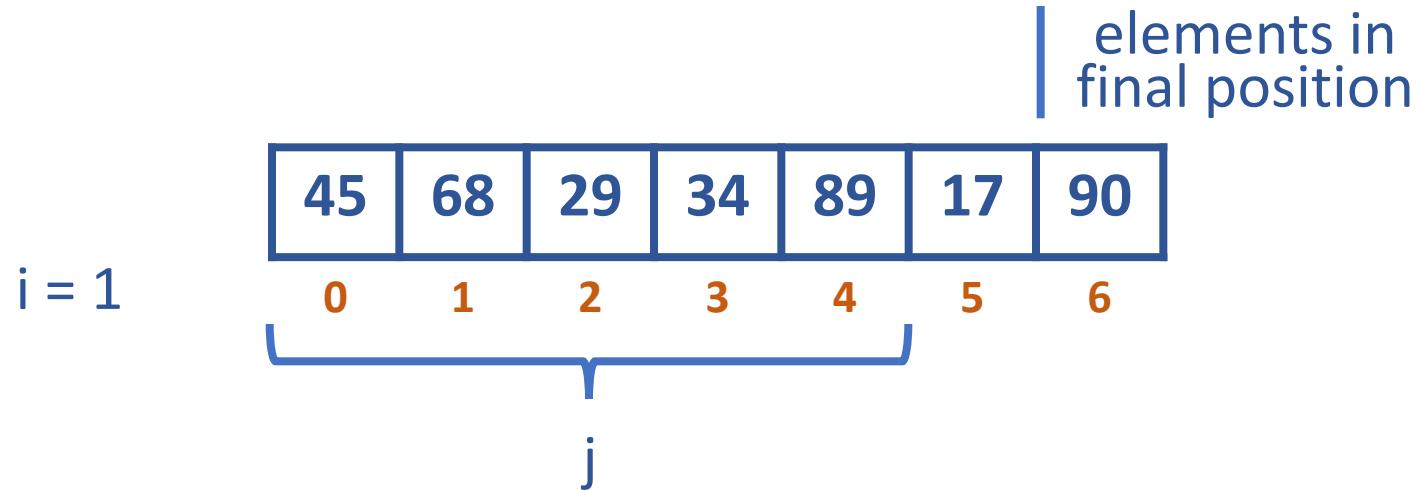
## Bubble Sort



$34 < 89$   swap

# DESIGN AND ANALYSIS OF ALGORITHMS

## Bubble Sort

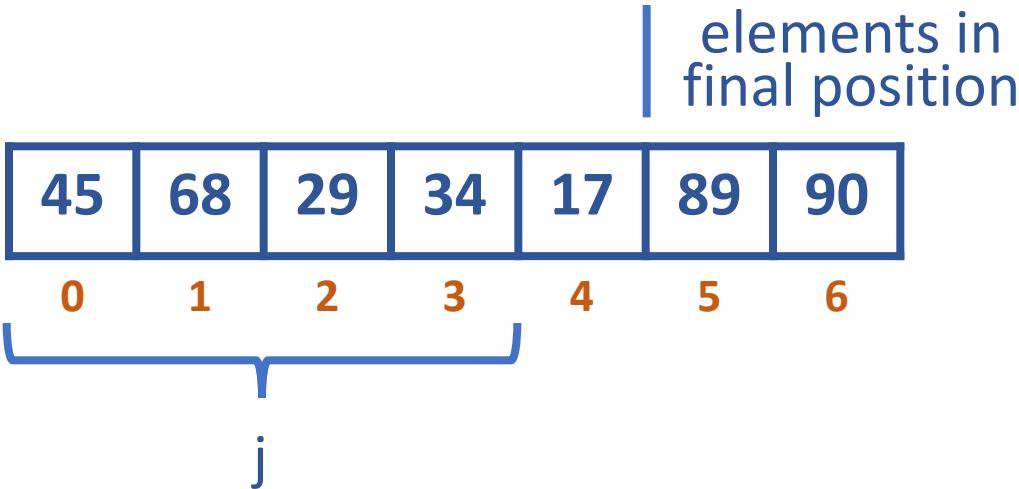


# DESIGN AND ANALYSIS OF ALGORITHMS

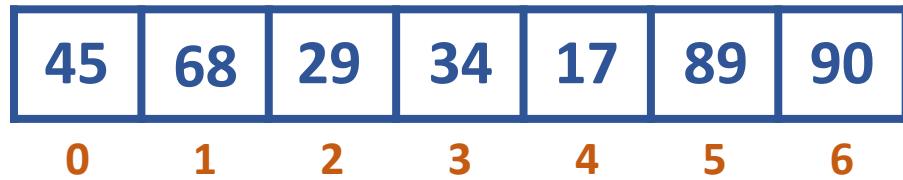
## Bubble Sort

After  
second  
iteration

$i = 2$



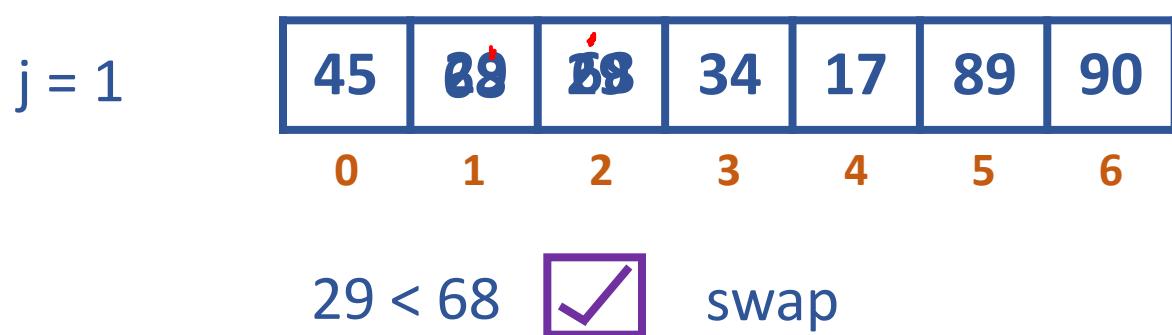
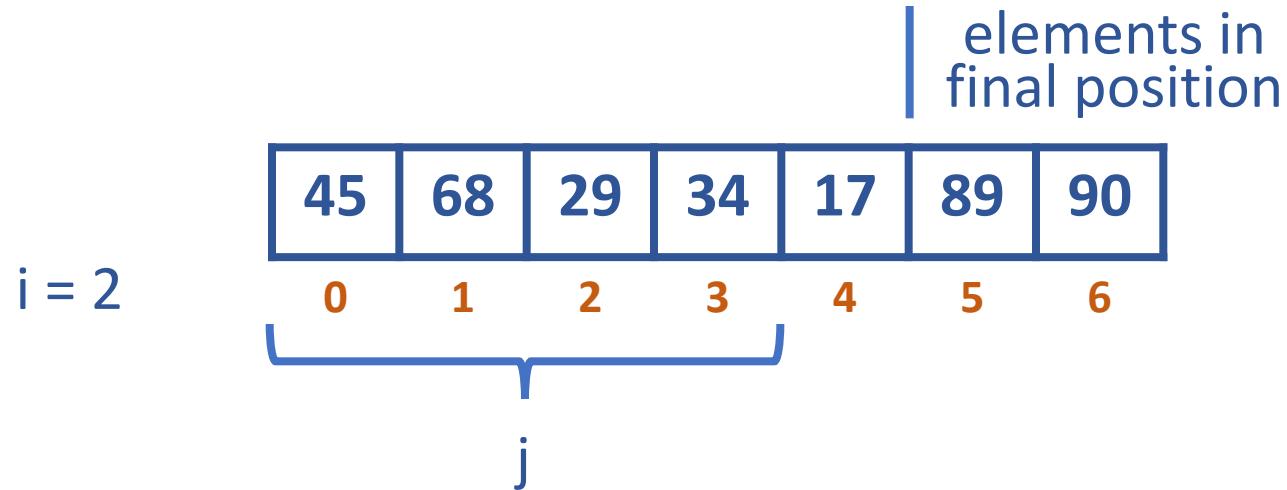
$j = 0$



$68 < 45$   no swap

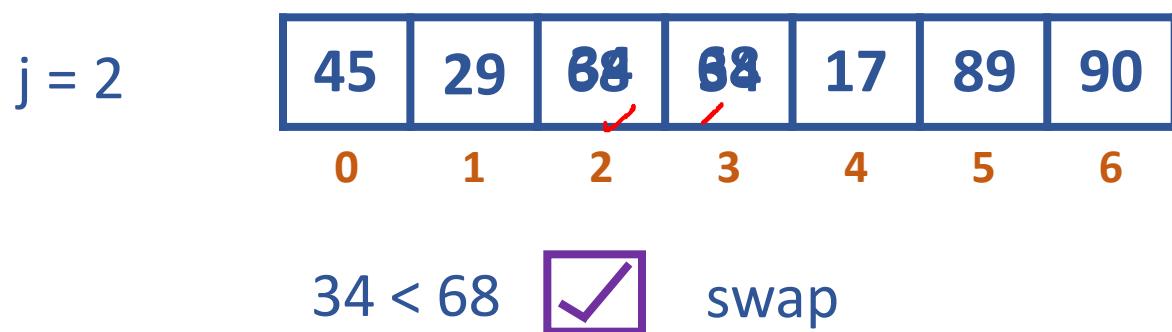
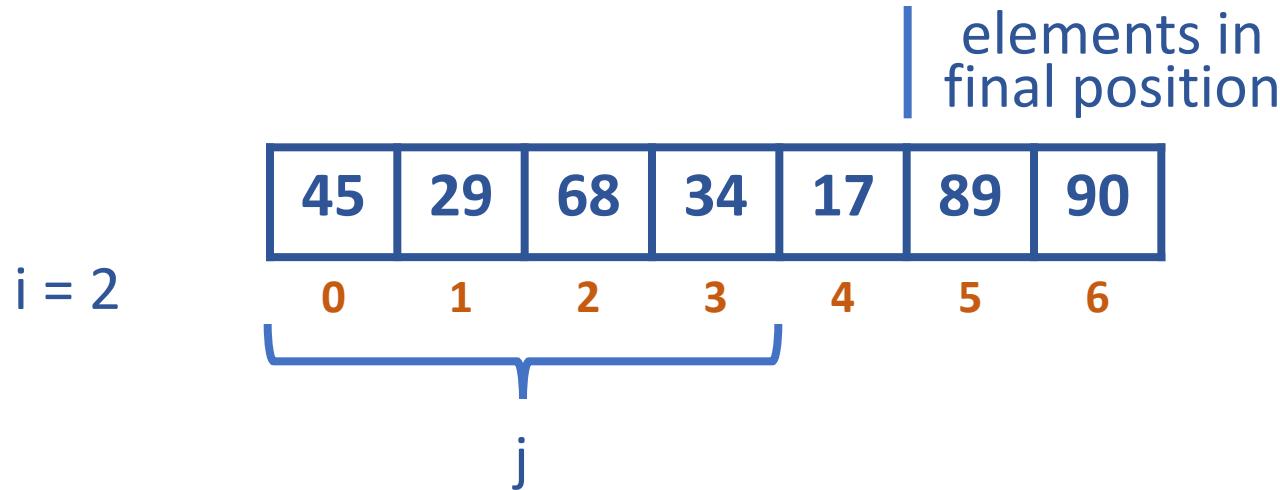
# DESIGN AND ANALYSIS OF ALGORITHMS

## Bubble Sort



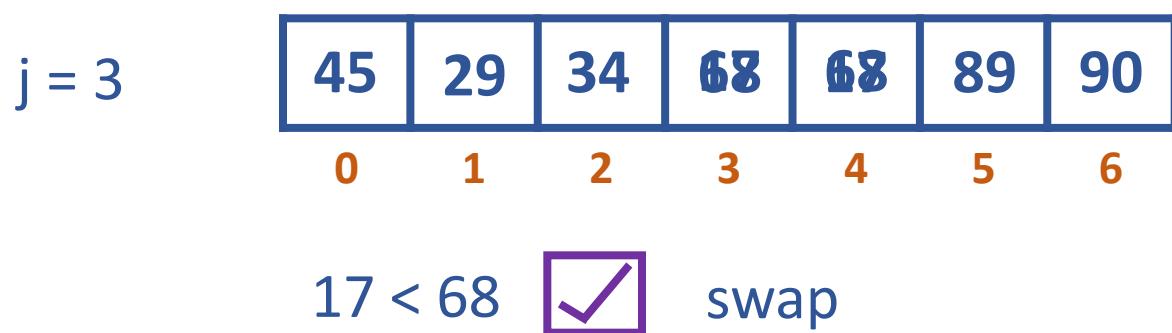
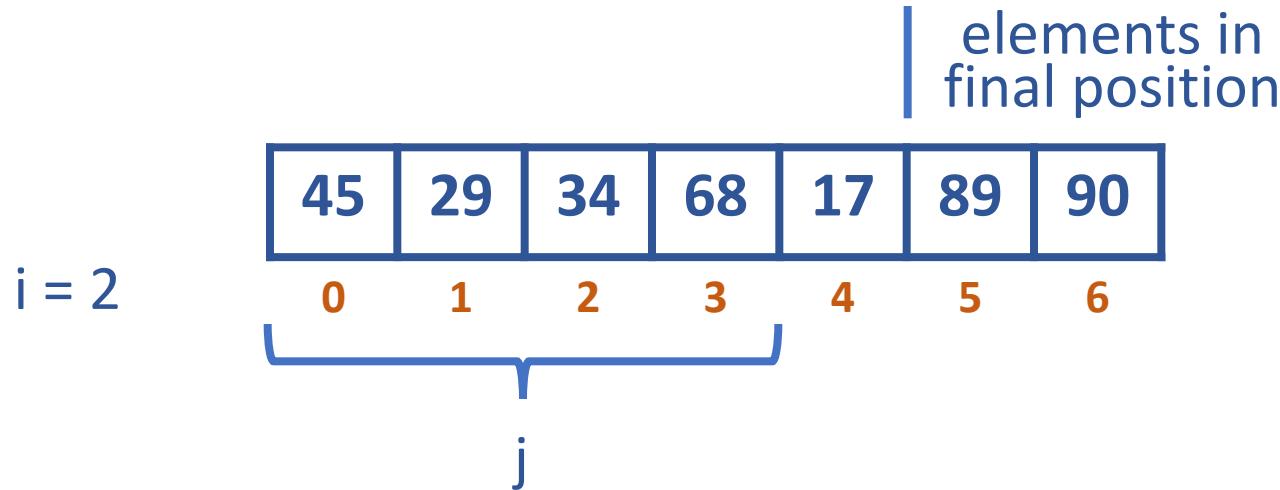
# DESIGN AND ANALYSIS OF ALGORITHMS

## Bubble Sort



# DESIGN AND ANALYSIS OF ALGORITHMS

## Bubble Sort

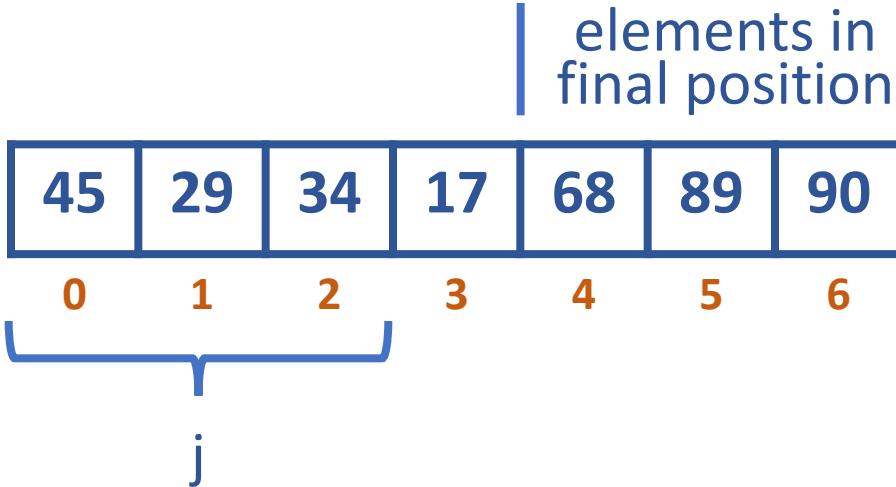


# DESIGN AND ANALYSIS OF ALGORITHMS

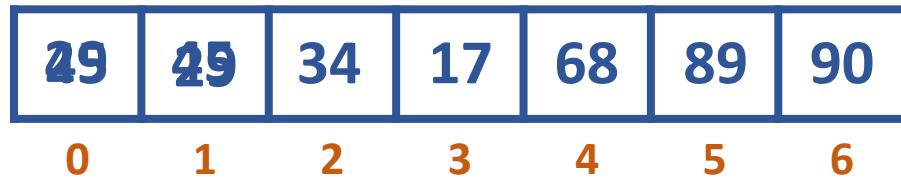
## Bubble Sort

After  
third  
iteration

$i = 3$



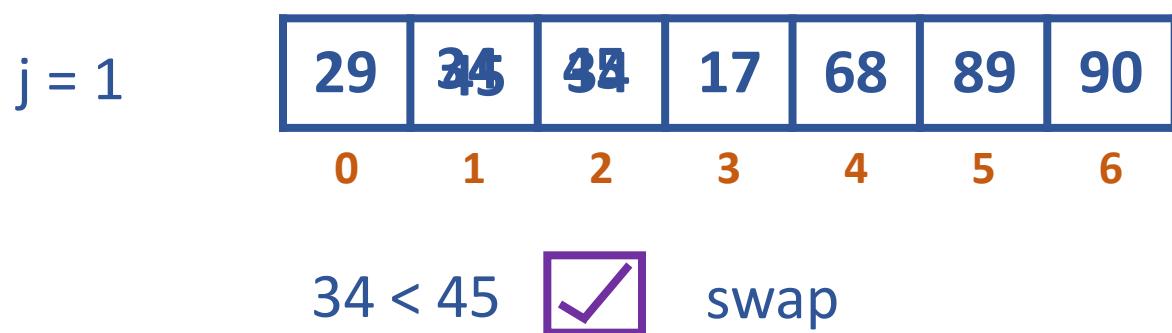
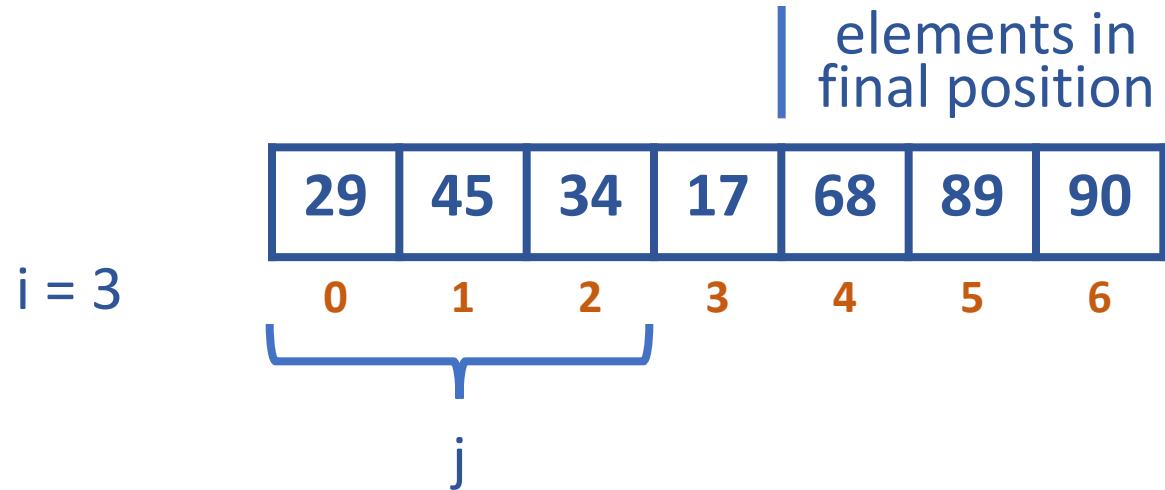
$j = 0$



$29 < 45$   swap

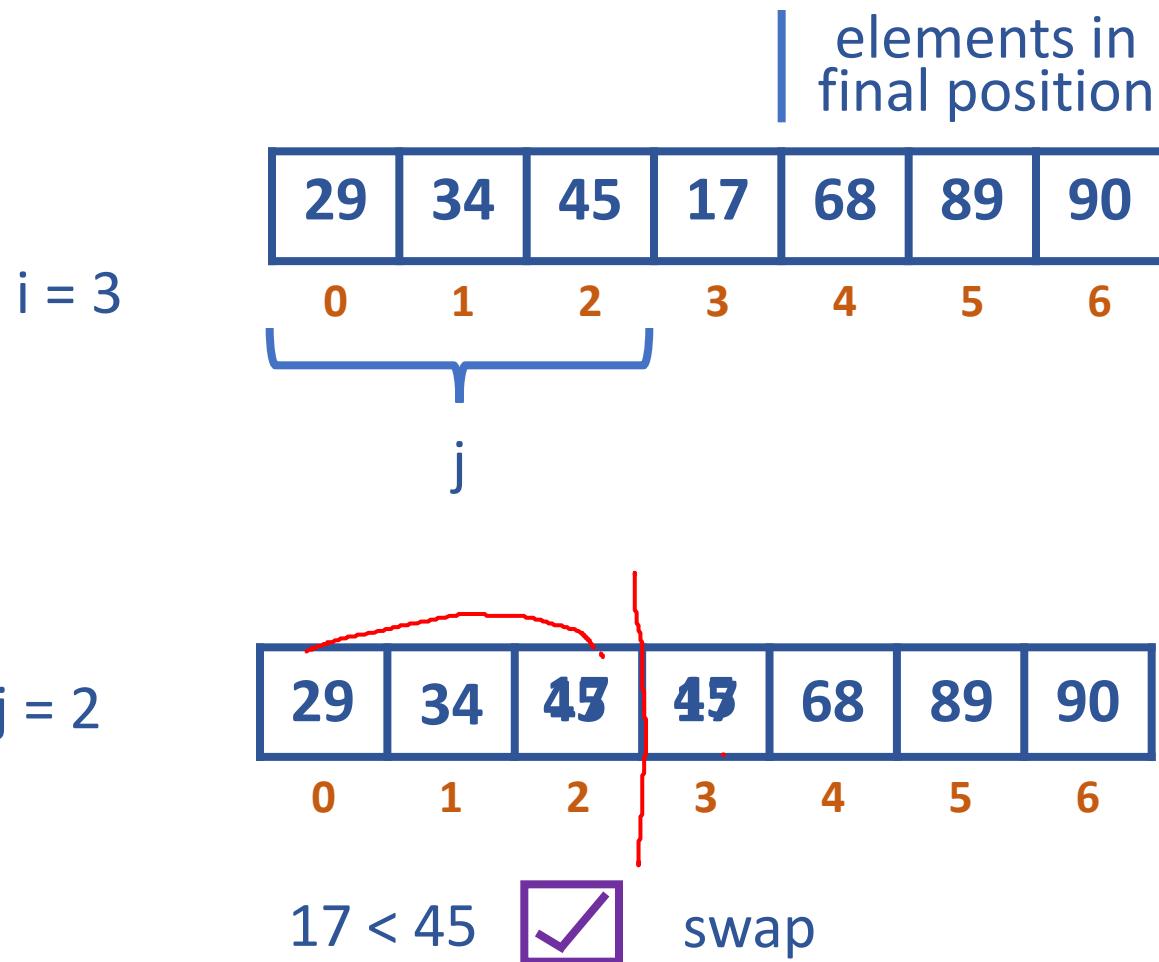
# DESIGN AND ANALYSIS OF ALGORITHMS

## Bubble Sort



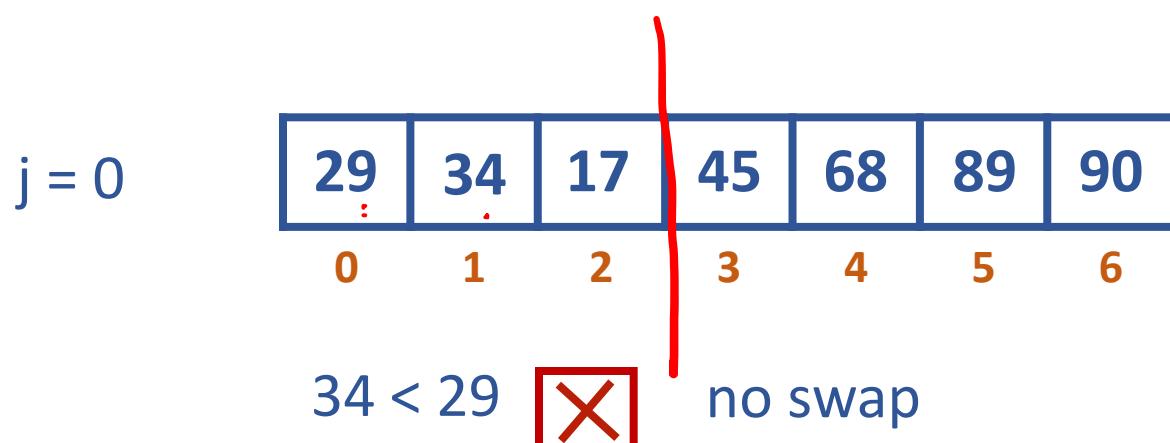
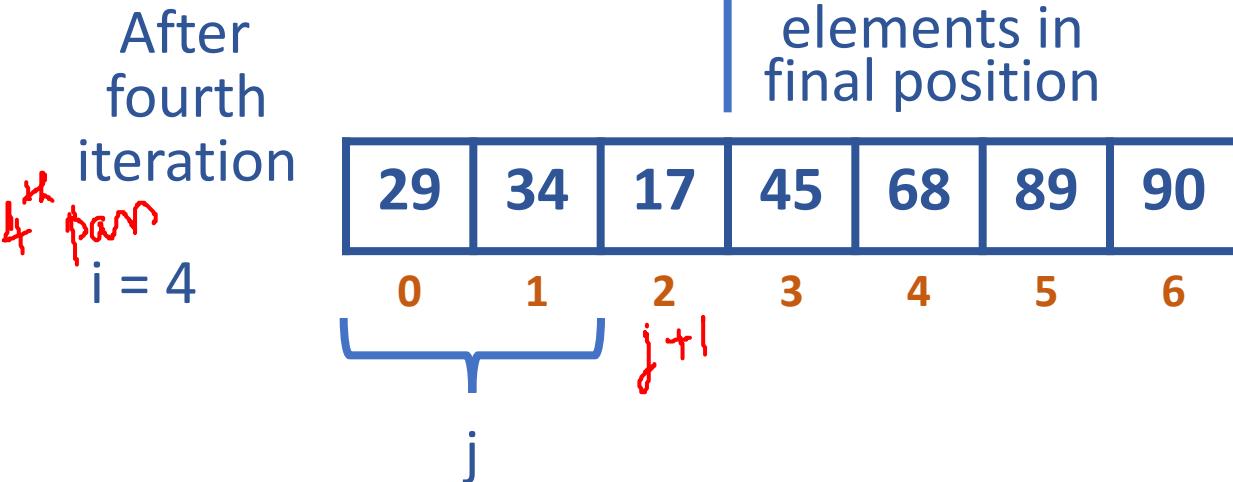
# DESIGN AND ANALYSIS OF ALGORITHMS

## Bubble Sort



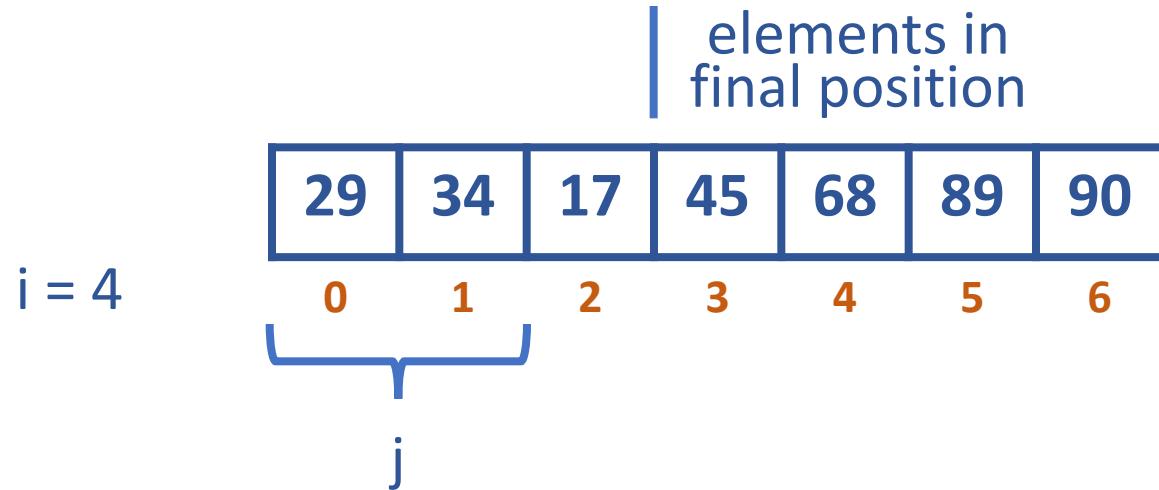
# DESIGN AND ANALYSIS OF ALGORITHMS

## Bubble Sort



# DESIGN AND ANALYSIS OF ALGORITHMS

## Bubble Sort

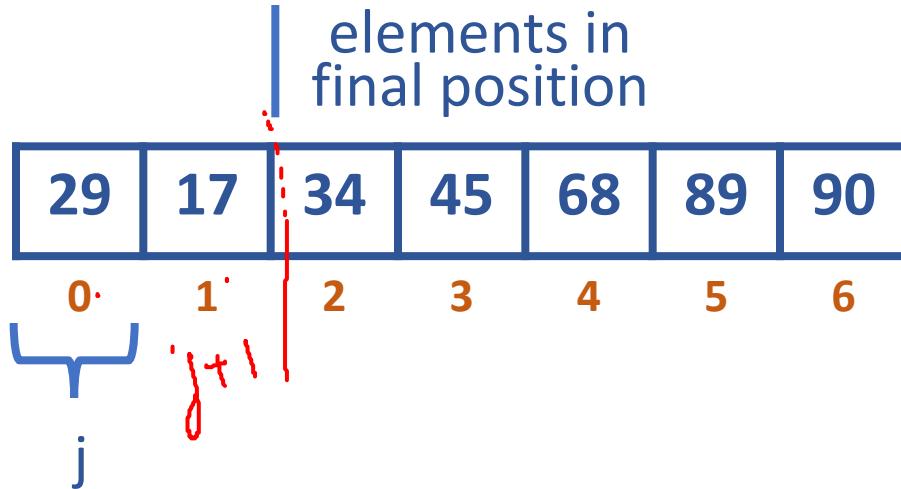


# DESIGN AND ANALYSIS OF ALGORITHMS

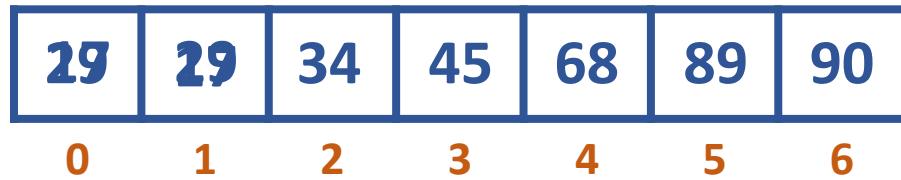
## Bubble Sort

After  
fifth  
iteration

$i = 5$



$j = 0$



$17 < 29$   swap

# DESIGN AND ANALYSIS OF ALGORITHMS

## Bubble Sort

After sixth iteration	elements in final position	17	29	34	45	68	89	90
		0	1	2	3	4	5	6

unsorted list

90, 89, 68, 45, 34, 29, 17

Sort is ascending order

How many swaps ? → worst swaps

# DESIGN AND ANALYSIS OF ALGORITHMS

## Bubble Sort

---



ALGORITHM BubbleSort( $A[0 .. n - 1]$ )

//Sorts a given array by bubble sort in their final positions

//Input: An array  $A[0 .. n - 1]$  of orderable elements

//Output: Array  $A[0 .. n - 1]$  sorted in ascending order

for  $i \leftarrow 0$  to  $n - 2$  do

    for  $j \leftarrow 0$  to  $n - 2 - i$  do

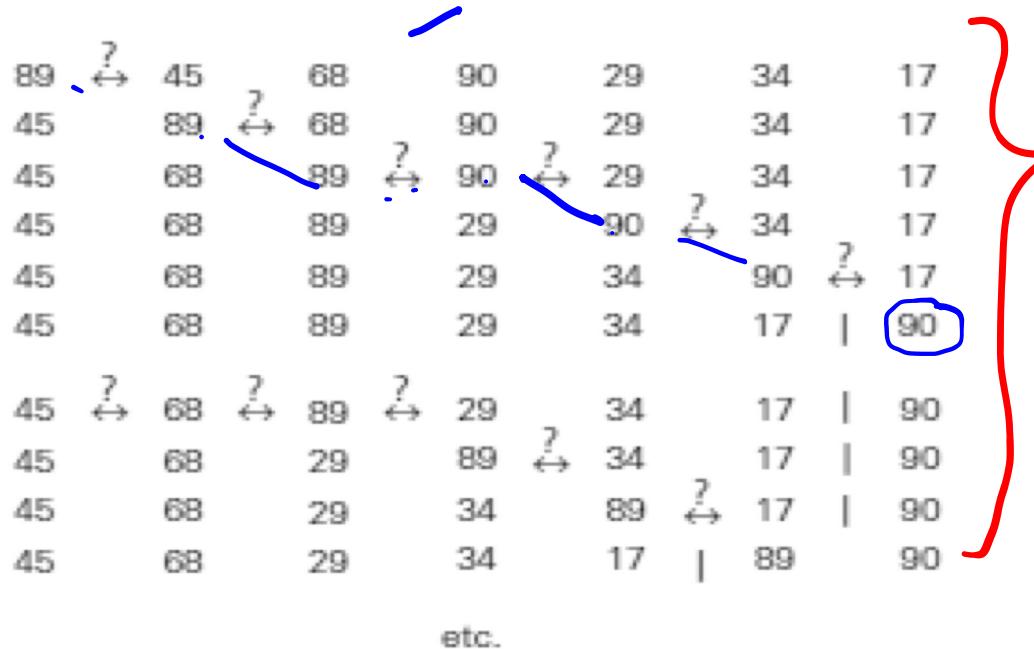
        if  $A[j + 1] < A[j]$  swap  $A[j]$  and  $A[j + 1]$

# DESIGN AND ANALYSIS OF ALGORITHMS

## Bubble Sort

→ Text

Tracing



**FIGURE 3.2** First two passes of bubble sort on the list 89, 45, 68, 90, 29, 34, 17. A new line is shown after a swap of two elements is done. The elements to the right of the vertical bar are in their final positions and are not considered in subsequent iterations of the algorithm.

# DESIGN AND ANALYSIS OF ALGORITHMS

## Bubble Sort

$E^1, X, A, M, P, L, E^2$

tracing



$n=7$	$i=0$	$i=1$	$i=2$	$i=3$	$i=4$	$i=5 (n-2)$	→ Outer
Original Array	I pass	II pass	III pass	IV pass	V pass	VI pass	
$A[0]=89$	45	45	45	29	29	17	
$A[1]=45$	68	68	29	34	17	29	
$A[2]=68$	89	29	34	17	34	34	
$A[3]=90$	29	34	17	45	45	45	
$A[4]=29$	34	17	68	68	68	68	
$A[5]=34$	17	89	89	89	89	89	
$A[6]=17$	90	90	90	90	90	90	

### Bubble Sort Analysis

$$C(n) = \sum_{i=0}^{n-2} \sum_{j=0}^{n-2-i} 1 = \sum_{i=0}^{n-2} [(n - 2 - i) - 0 + 1]$$

$$= \sum_{i=0}^{n-2} (n - 1 - i) = \frac{(n - 1)n}{2} \in \Theta(n^2)$$

Bubble Sort is a  $\Theta(n^2)$  algorithm

# DESIGN AND ANALYSIS OF ALGORITHMS

## Bubble Sort - Exercise

---



1. Sort the list E<sup>1</sup>, X, A, M, P, L, E<sup>2</sup> in alphabetical order by bubble sort.
2. Is Bubble sort stable ?

# DESIGN AND ANALYSIS OF ALGORITHMS

## Bubble Sort - Exercise

---



1. Sort the list E, X, A, M, P, L, E in alphabetical order by bubble sort.
2. Is Bubble sort stable ?

Stable?

"In place"

→ An algorithm is said to be in place if it does not require extra memory, except possibly for a few memory units.



**PES**  
UNIVERSITY  
ONLINE

**THANK YOU**

---

Bharathi R

Department of Computer Science  
& Engineering

[rbharathi@pes.edu](mailto:rbharathi@pes.edu)



# DESIGN AND ANALYSIS OF ALGORITHMS

## UE21CS241B

---

**Bharathi R**

Department of Computer Science  
& Engineering

2 different Sequential search algorithms



# DESIGN AND ANALYSIS OF ALGORITHMS

---

## Sequential Search

Major Slides Content: Anany Levitin

**Bharathi R**

Department of Computer Science & Engineering

### ALGORITHM *SequentialSearch*( $A[0..n - 1]$ , $K$ )

//Searches for a given value in a given array by sequential search

//Input: An array  $A[0..n - 1]$  and a search key  $K$

//Output: The index of the first element in  $A$  that matches  $K$

// or  $-1$  if there are no matching elements

$i \leftarrow 0$

**while**  $i < n$  **and**  $A[i] \neq K$  **do**

$i \leftarrow i + 1$

**if**  $i < n$  **return**  $i$

**else return**  $-1$

## Sequential Search

---

- Compares successive elements of a given list with a given search key until:
  - A match is encountered (Successful Search)
  - List is exhausted without finding a match (Unsuccessful Search)
- An improvisation to the algorithm is to append the key to the end of the list
- This means the search has to be successful always and we can eliminate the end of list check

## Sequential Search

Best case }  $\Theta(1)$   
 Average case }  
 Worst case }  $\Theta(n)$

- Sequential / Linear Search

returns index value

10	14	19	26	27	31	33	35	42	44
0	1	2	3	4	5	6	7	8	9

- For key = 33, 6 is returned ✓ → Successful search
- For key = 50, -1 is returned ✗ → Unsuccessful search

Single array  
 ↓  
 for loop (Google)  
 Comparison  
 $A[i] == \text{key}$

## Sequential Search

---

ALGORITHM SequentialSearch2(A[0 .. n ], K)

//Implements sequential search with a search key as a sentinel

//Input: An array A of n elements and a search key K

//Output: The index of the first element in A[0 .. n -1] whose value is

// equal to K or -1 if no such element is found

A[n]<---K // Sentinel

i<---0

while A[i] ≠ K do

    i<--- i + 1

if i < n return i // Successful search

else return -1 // Unsuccessful

### Sequential Search Analysis

- Sequential Search is a  $\Theta(n)$  algorithm

Input Size:  $n$

Basic Operation :  $(A[i] \neq K)$

$C_{worst}(n)$  = Count of the basic operation at the max  
 $= n + 1 \in \Theta(n)$

$C_{best}(n) = 1 \in \Theta(1)$

$C_{avg}(n)$  = from  $(n+1)/2$  to  $(n+1)$  depending on the probability of search key being present in the input array.

$C_{avg}(n) \in \Theta(n)$



**THANK YOU**

---

**Bharathi R**

Department of Computer Science  
& Engineering



# DESIGN AND ANALYSIS OF ALGORITHMS

## UE21CS241B

---

**Bharathi R**

Department of Computer Science  
& Engineering

# DESIGN AND ANALYSIS OF ALGORITHMS

---

## String Matching

Major Slides Content: Anany Levitin

**Bharathi R**

Department of Computer Science & Engineering

### String Matching - Terms

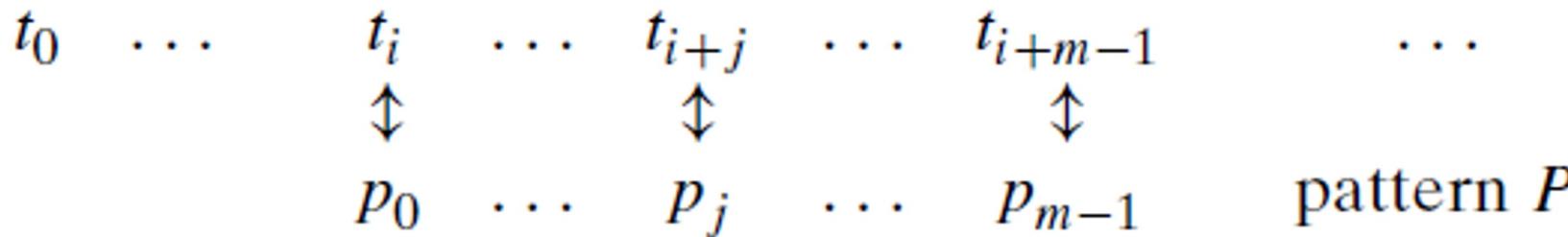
- pattern:  
a string of **m** characters to search for
- text:  
a (longer) string of **n** characters to search in
- problem:  
find a substring in the text that matches the pattern

## Brute – Force String Matching

In an  $n$ -characters **text**, search for the first occurrence of  $m$ -character **pattern**.

There are  $n-m+1$  substrings of length  $m$  in the text of length  $n$ . Find the first such substring which matches with the pattern.

Find  $i$ , the index of the leftmost character of the first matching substring in the text such that



$$t_i = p_0, \dots, t_{i+j} = p_j, \dots, t_{i+m-1} = p_{m-1}$$

$n=12$    text:  $\begin{smallmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 \\ \text{good\_morning} \end{smallmatrix}$   
 pat: morn    $m=4$   
 no. of substrings of length 4  
 $t_{n-1}^{n-m+1} = 12-4+1 = 9$  substr.

## String Matching Idea

---

Step 1: Align pattern at beginning of text

Step 2: Moving from left to right, compare each character of pattern to the corresponding character in text until:

- all characters are found to match (successful search); or
- a mismatch is detected

Step 3: While pattern is not found and the text is not yet exhausted, realign pattern one position to the right and repeat Step 2

```
ALGORITHM BruteForceStringMatch(T[0 .. n - 1], P[0 .. m - 1])
//Implements brute-force string matching
//Input: An array T[0 .. n - 1] of n characters representing a text
// and an array P[0 .. m - 1] of m characters representing a pattern
//Output: The index of the first character in the text that starts a
//matching substring or -1 if the search is unsuccessful
for i ← 0 to n-m do
    j ← 0
    while j < m and P[j] = T[i + j] do
        j ← j+1
    if j = m return i
return -1
```

# DESIGN AND ANALYSIS OF ALGORITHMS

## String Matching

```
ALGORITHM BruteForceStringMatch(T[0 .. n - 1], P[0 .. m - 1])  
//Implements brute-force string matching  
//Input: An array T[0 .. n - 1] of n characters representing a text  
// and an array P[0 .. m - 1] of m characters representing a pattern  
//Output: The index of the first character in the text that starts a  
//matching substring or -1 if the search is unsuccessful  
for i ← 0 to n-m do // n-m+1 → (n-m+1) times  
    j ← 0 // for pattern  
    while j < m and P[j] = T[i + j] do // m is length of the pattern → m times  
        j ← j+1  
    if j = m return i → // Successful search  
return -1 // unsuccessful search
```

## String Matching Example

---

N O B O D Y \_ N O T I C E D \_ H I M  
N O T

## String Matching Example

---

N O B O D Y \_ N O T I C E D \_ H I M  
N O T  
N O T  
N O T  
N O T  
N O T  
N O T  
N O T  
N O T  
N O T

# DESIGN AND ANALYSIS OF ALGORITHMS

Pattern = Simple

# String Matching Example

## String Matching Analysis

Worst Case:

- The algorithm might have to make all the 'm' comparisons for each of the  $(n-m+1)$  tries
- Therefore, the algorithm makes  $m(n-m+1)$  comparisons
- Brute Force String Matching is a  $O(nm)$  algorithm

$$\text{Complexity} = (n-m+1)m$$

$\frac{(n-m+1)m}{(n) m}$

- Repeated  $n m$  overlapping strings: text is found in this text book,  $\frac{n}{m}$ .
- ~ overlapping strings: pattern: text      patt: ana      banana



**THANK YOU**

---

**Bharathi R**

Department of Computer Science  
& Engineering



# **DESIGN AND ANALYSIS OF ALGORITHMS**

## **UE22CS241B**

---

**Bharathi R**

Department of Computer Science  
& Engineering

# DESIGN AND ANALYSIS OF ALGORITHMS

---

## Travelling Salesman Problem

Major Slides Content: Anany Levitin

**Bharathi R**

Department of Computer Science & Engineering

## Exhaustive Search:

A brute force solution to a problem involving search for an element with a special property, usually among **combinatorial** objects such as **permutations, combinations, or subsets of a set.**

## Method:

- Generate a list of **all potential solutions** to the problem in a systematic manner.
- **Evaluate** potential solutions one by one, **disqualifying** infeasible ones and, for an optimization problem, keeping track of the **best one** found so far.
- When search ends, **announce the solution(s)** found.

## Travelling Salesman Problem:

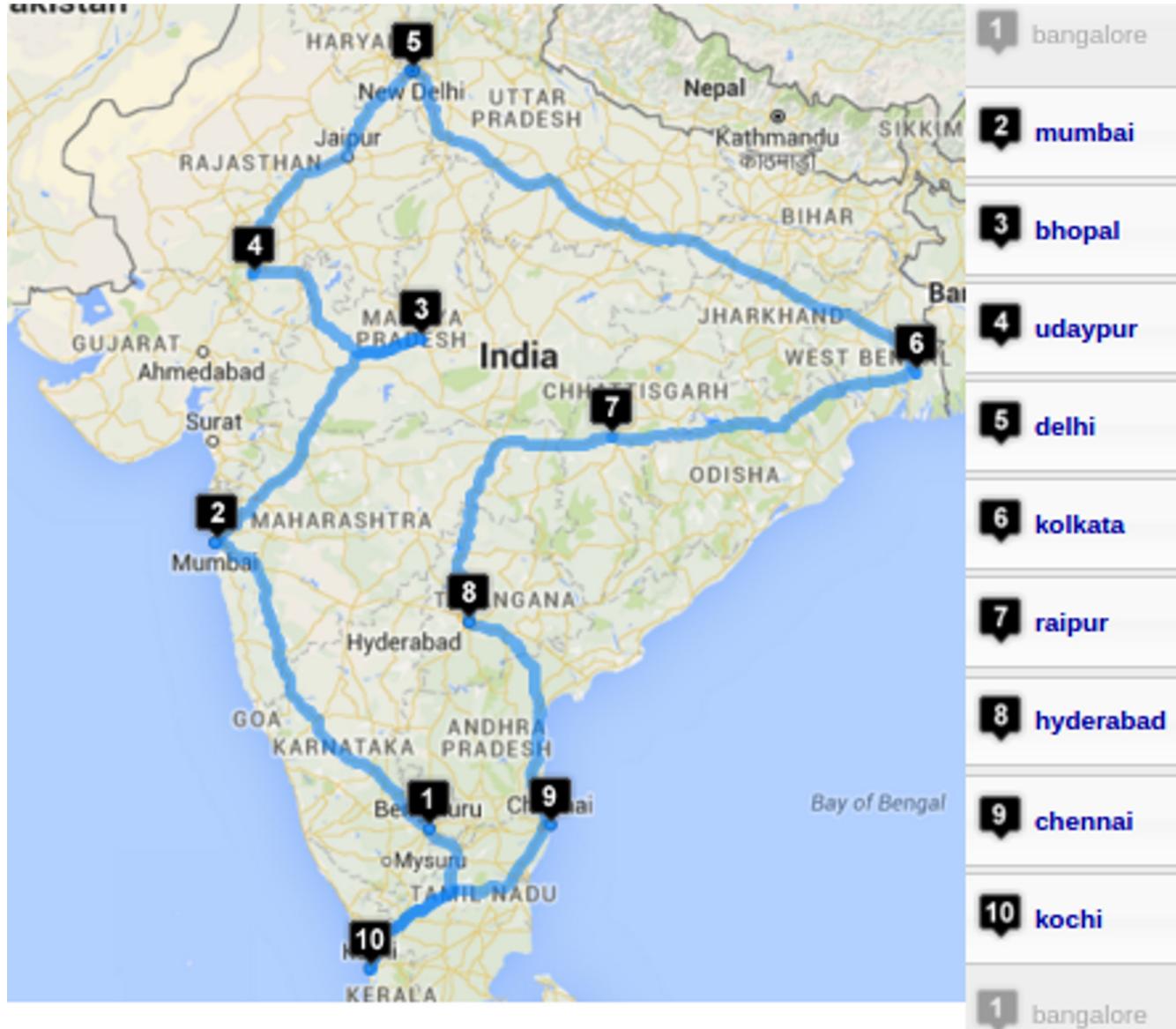
1. Given  $n$  cities and distances between each pair of cities, find the shortest tour that passes through all other cities and returns to the origin city.
2. In case of a weighted complete graph, it's about finding the shortest *Hamiltonian circuit*.

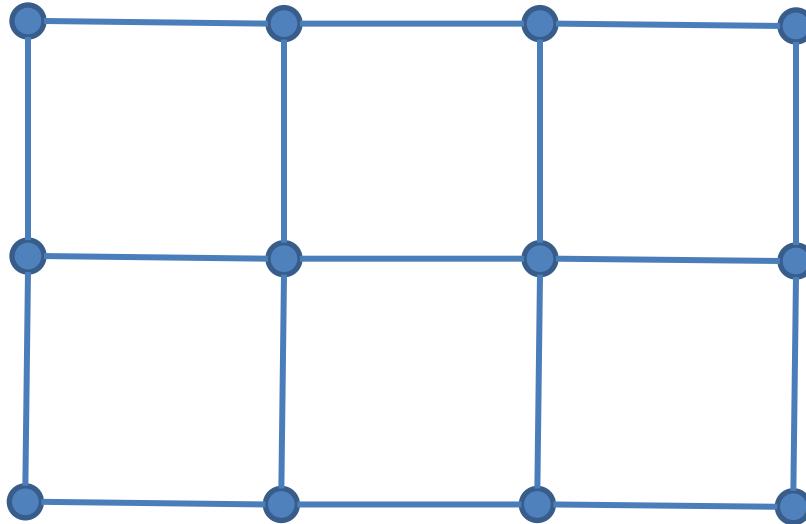
Eg: Driving time between some 10 cities of India (Cost Matrix).

000000	110189	050573	020948	109480	034435	028433	074836	091767	068406
109006	000000	079663	118195	079397	143304	083593	045792	037923	068146
051516	080265	000000	070149	121881	083636	044745	043763	042416	067450
021557	119539	069838	000000	095820	042397	037471	084186	111032	077756
110053	081231	121373	095977	000000	134475	085826	087690	100264	054016
034488	144238	082769	041728	134042	000000	062482	108885	123963	102455
028473	084770	045153	037117	085732	062772	000000	049417	078006	042987
075056	046162	044536	084245	086579	109354	049641	000000	031151	038399
092933	037994	042414	111566	099497	125053	078960	031010	000000	068113
068718	068844	068336	077907	055357	103016	043305	038648	068634	000000

## Travelling Salesman Problem



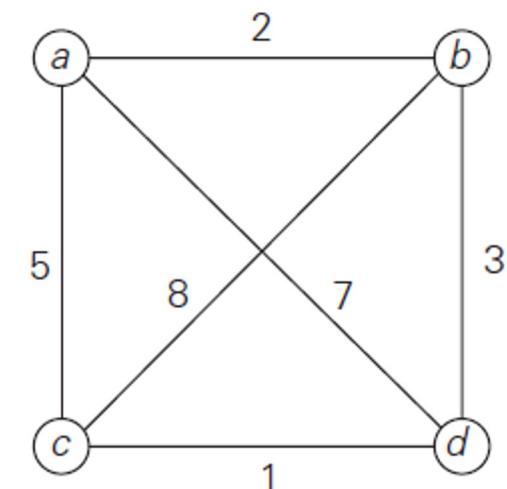




- The TSP problem can be stated as the problem of finding the shortest Hamiltonian circuit of the graph.
- (A Hamiltonian circuit is defined as a cycle that passes through all the vertices of the graph exactly once. It is named after the Irish mathematician Sir William Rowan Hamilton (1805–1865),

## Travelling Salesman Problem:

<u>Tour</u>	<u>Length</u>	
$a \rightarrow b \rightarrow c \rightarrow d \rightarrow a$	$l = 2 + 8 + 1 + 7 = 18$	
$a \rightarrow b \rightarrow d \rightarrow c \rightarrow a$	$l = 2 + 3 + 1 + 5 = 11$	optimal
$a \rightarrow c \rightarrow b \rightarrow d \rightarrow a$	$l = 5 + 8 + 3 + 7 = 23$	
$a \rightarrow c \rightarrow d \rightarrow b \rightarrow a$	$l = 5 + 1 + 3 + 2 = 11$	optimal
$a \rightarrow d \rightarrow b \rightarrow c \rightarrow a$	$l = 7 + 3 + 8 + 5 = 23$	
$a \rightarrow d \rightarrow c \rightarrow b \rightarrow a$	$l = 7 + 1 + 8 + 2 = 18$	



## Exhaustive Search

---

- Exhaustive Search is a brute – force problem solving technique
- It suggests generating each and every element of the problem domain, selecting those of them that satisfy all the constraints and then finding a desired element
- The desired element might be one which minimizes or maximizes a certain characteristic
- Typically the problem domain involves combinatorial objects such as permutations, combinations and subsets of a given set

## Exhaustive Search - Method

---

- Generate a list of all potential solutions to the problem in a systematic manner
- Evaluate potential solutions one by one, disqualifying infeasible ones and, for an optimization problem, keeping track of the best one found so far
- When search ends, announce the solution(s) found

## Travelling Salesman Problem

---

- Given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city and returns to the origin city?

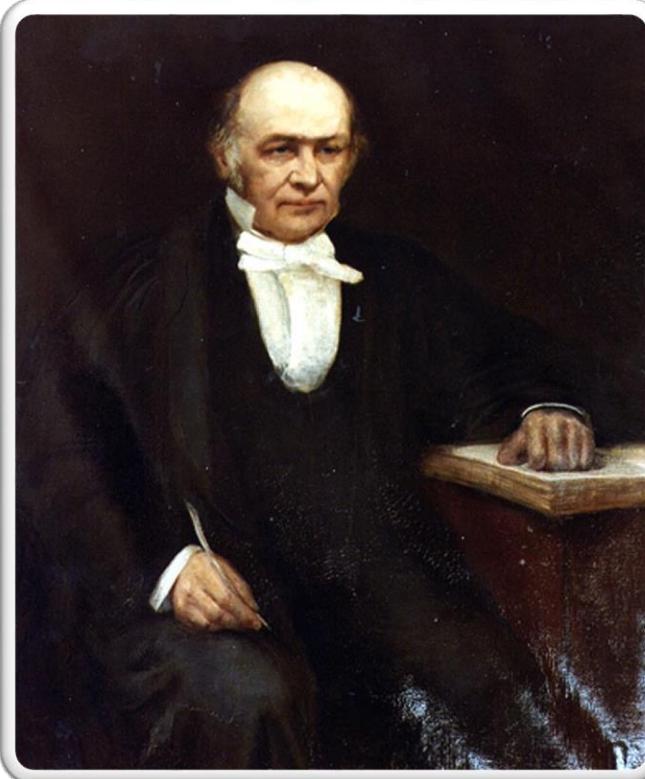
Alternative way to state the problem:

- Find the shortest Hamiltonian Circuit in a weighted connected graph

## TSP: History and Relevance

---

- The Travelling Salesman Problem was mathematically formulated by Irish Mathematician Sir William Rowan Hamilton
- It is one of the most intensively studied problems in optimization
- It has applications in logistics and planning



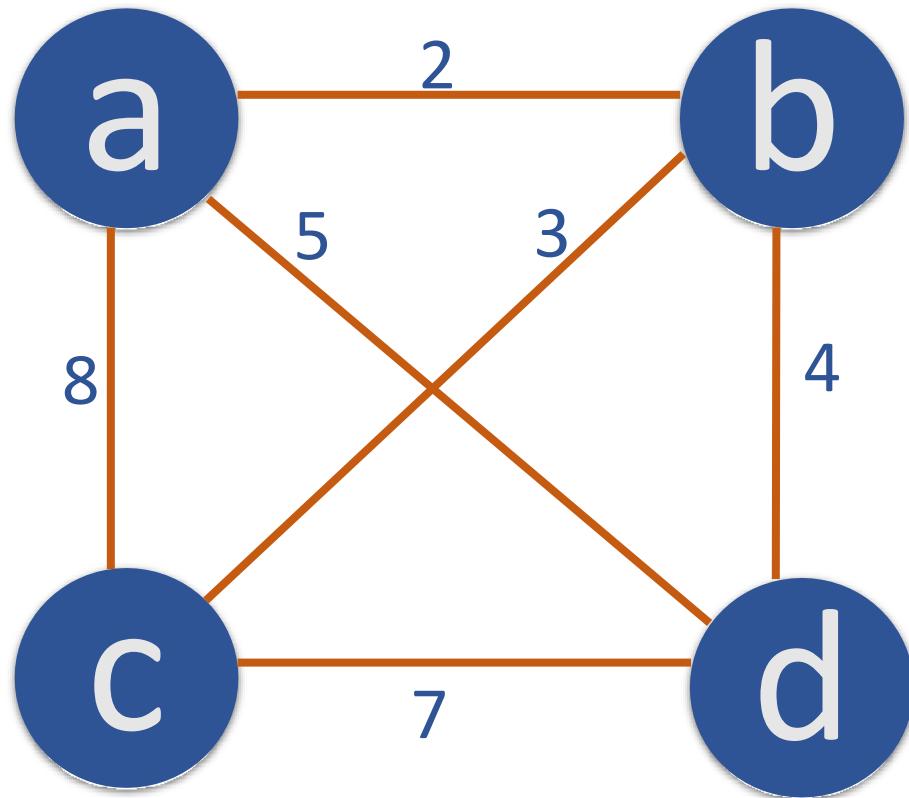
# DESIGN AND ANALYSIS OF ALGORITHMS

## Travelling Salesman Problem



PES  
UNIVERSITY  
ONLINE

Example



Tour	Length
a → b → c → d → a	2 + 3 + 7 + 5 = 17
a → b → d → c → a	2 + 4 + 7 + 8 = 21
a → c → b → d → a	8 + 3 + 4 + 5 = 20
a → c → d → b → a	8 + 7 + 4 + 2 = 21
a → d → b → c → a	5 + 4 + 3 + 8 = 20
a → d → c → b → a	5 + 7 + 3 + 2 = 17

Starting vertex b.

b - c - d - a - b

b - c - a - d - b

= 6

z

optimal (minimal)

optimal (minimal)

### Efficiency

- The Exhaustive Search solution to the Travelling Salesman problem can be obtained by keeping the origin city constant and generating permutations of all the other  $n - 1$  cities
- Thus, the total number of permutations needed will be  $(n - 1)!$



**THANK YOU**

---

**Bharathi R**

Department of Computer Science  
& Engineering

**rbharathi@pes.edu**

Exhaustive search

- 1) TSP  $\rightarrow (n-1)!$
- 2) knapsack problem
- 3) Assignment problem.



# DESIGN AND ANALYSIS OF ALGORITHMS

## UE22CS241B

---

**Bharathi R**  
Department of Computer Science  
& Engineering

# DESIGN AND ANALYSIS OF ALGORITHMS

---

## Knapsack Problem

Major Slides Content: Anany Levitin

**Bharathi R**

Department of Computer Science & Engineering

## Knapsack Problem

### Knapsack Problem:

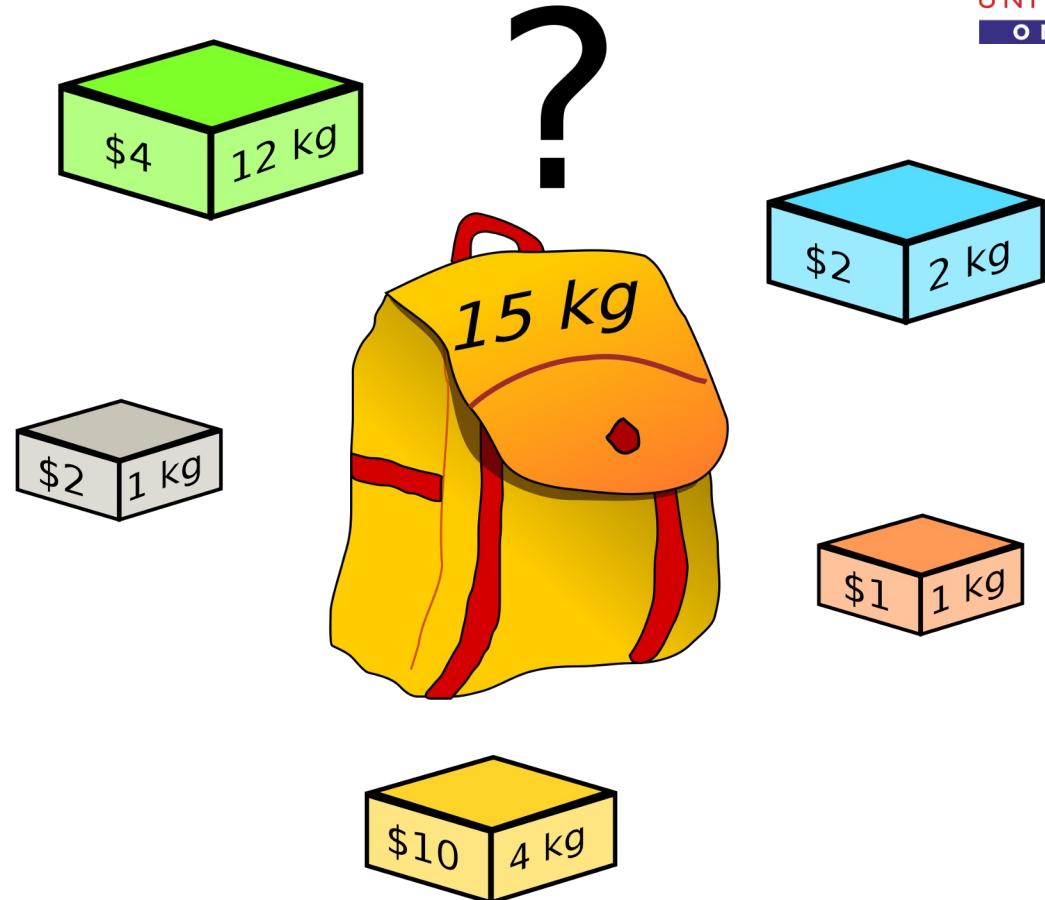
Given  $n$  items:

weights:  $w_1 \ w_2 \dots \ w_n$

values:  $v_1 \ v_2 \dots \ v_n$

a knapsack of capacity  $W$

Find most valuable subset of the items that fit into the knapsack.



## Knapsack Problem

### Knapsack Problem:

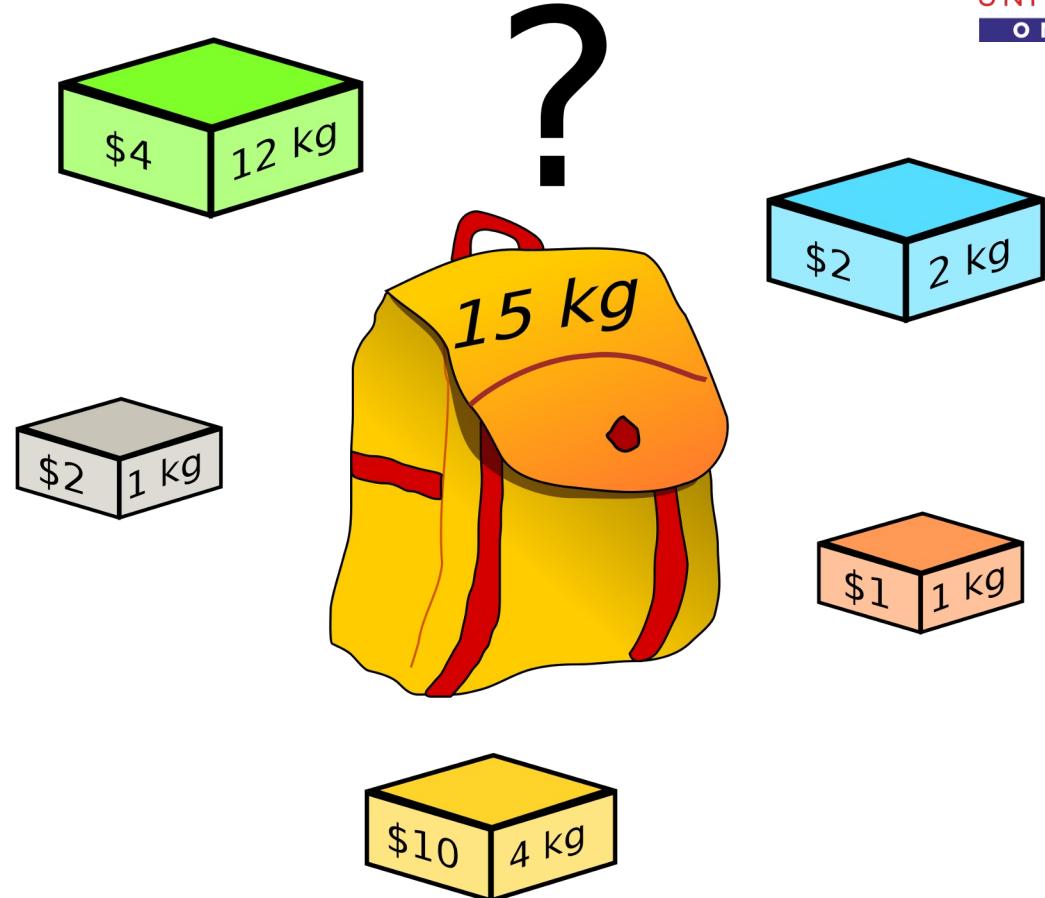
Given  $n$  items:

weights:  $w_1 \ w_2 \dots \ w_n$

values:  $v_1 \ v_2 \dots \ v_n$

a knapsack of capacity  $W$

Find most valuable subset of the items that fit into the knapsack.



## Knapsack Problem

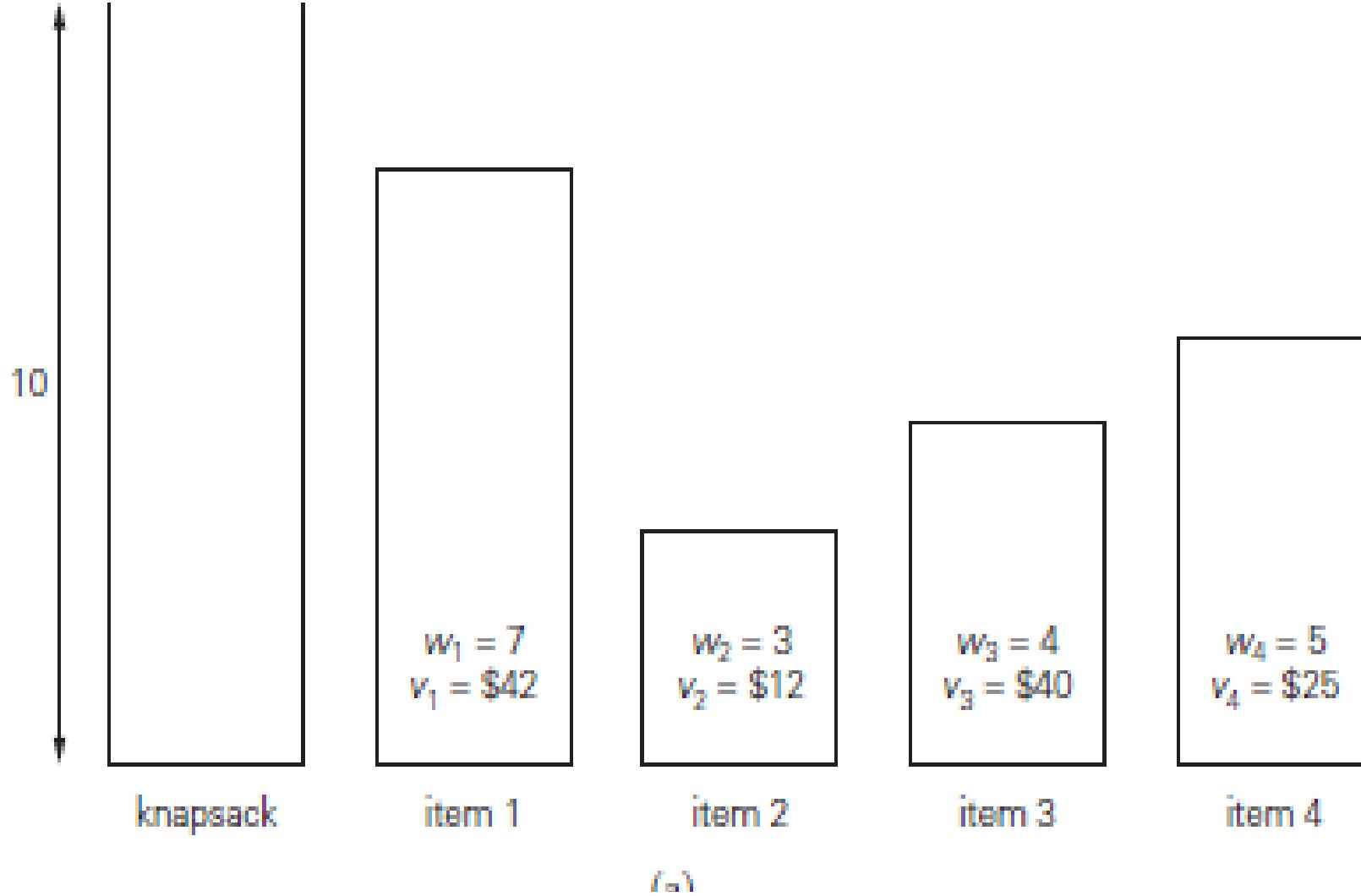
---

Given n items:

- weights:  $w_1 \ w_2 \dots \ w_n$
- values:  $v_1 \ v_2 \dots \ v_n$
- a knapsack of capacity W

Find the most valuable subset of items that fit into the knapsack

## Knapsack Problem





4 items

$$S = \{1, 2, 3, 4\}$$

$P(S) = 2^4$  Subsets  $\therefore 16$  subsets

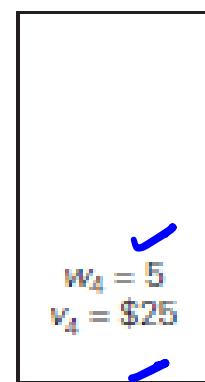
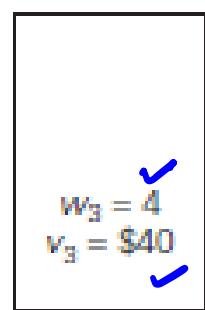
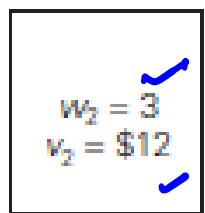
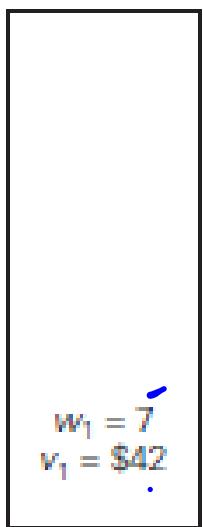
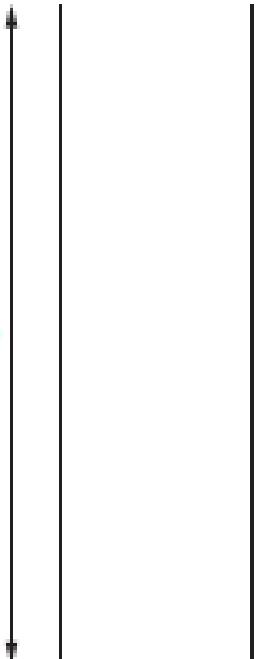
$$= \emptyset, \{1\} \dots \{4\}$$

$$\{1, 2\} \dots$$

$$\{1, 2, 3\}$$

$$\{1, 2, 3, 4\}$$

$$W = 10$$



knapsack

item 1

item 2

item 3

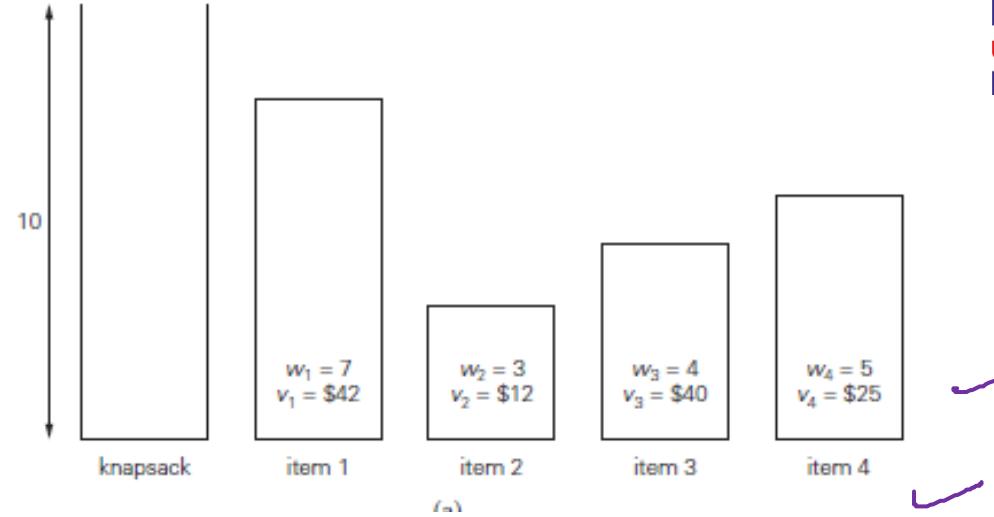
item 4

$i=1$

$n \rightarrow 2^n \rightarrow \underline{\text{exponential}}$   
 $\underline{\text{problem.}}$

# DESIGN AND ANALYSIS OF ALGORITHMS

## Knapsack Problem

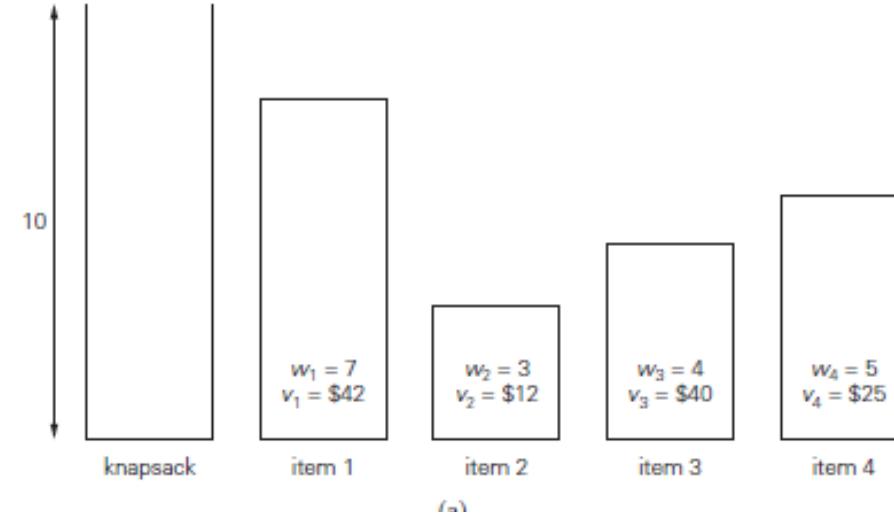


Item	Weight	Value
1	7	\$42
2	3	\$12
3	4	\$40
4	5	\$25

# DESIGN AND ANALYSIS OF ALGORITHMS

## Knapsack Problem

Subset	Total weight	Total value
$\emptyset$	0	\$ 0
{1}	7	\$42
{2}	3	\$12
{3}	4	\$40
{4}	5	\$25
{1, 2}	10	\$54
{1, 3}	11	not feasible
{1, 4}	12	not feasible
{2, 3}	7	\$52
{2, 4}	8	\$37
{3, 4}	9	\$65
{1, 2, 3}	14	not feasible
{1, 2, 4}	15	not feasible
{1, 3, 4}	16	not feasible
{2, 3, 4}	12	not feasible
{1, 2, 3, 4}	19	not feasible



Item	Weight	Value
1	7	\$42
2	3	\$12
3	4	\$40
4	5	\$25

# DESIGN AND ANALYSIS OF ALGORITHMS

## Knapsack Problem



PES  
UNIVERSITY  
ONLINE

Example

Find the optimal value for the given knapsack problem.

Knapsack Capacity  $W = 16$

Item	Weight	Value
1	2	20
2	5	30
3	10	50
4	5	10

## Knapsack Problem

Subset	Total Weight	Total Value
{1}	2	20
{2}	5	30
{3}	10	50
{4}	5	10
{1, 2}	7	50
{1, 3}	12	70
{1, 4}	7	30
{2, 3}	15	80
{2, 4}	10	40
{3, 4}	15	60
{1, 2, 3}	17	Not Feasible
{1, 2, 4}	12	60
{1, 3, 4}	17	Not Feasible
{2, 3, 4}	20	Not Feasible
{1, 2, 3, 4}	22	Not Feasible

Knapsack Problem by  
Exhaustive Search

## Knapsack Problem

---

- The Exhaustive Search solution to the Knapsack Problem is obtained by **generating all subsets** of the set of n items given and computing the total weight of each subset in order to identify the feasible subsets
- The number of subsets for a set of n elements is  **$2^n$**
- The Exhaustive Search solution to the Knapsack Problem belongs to  **$\Omega(2^n)$**



**THANK YOU**

---

**Bharathi R**

Department of Computer Science  
& Engineering

**rbharathi@pes.edu**



# **DESIGN AND ANALYSIS OF ALGORITHMS**

## **UE22CS241B**

---

**Bharathi R**

Department of Computer Science  
& Engineering

# DESIGN AND ANALYSIS OF ALGORITHMS

---

## Assignment Problem

Major Slides Content: Anany Levitin

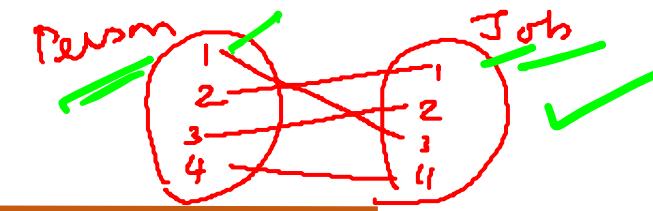
**Bharathi R**

Department of Computer Science & Engineering

# DESIGN AND ANALYSIS OF ALGORITHMS

## The Assignment Problem

"Permutation"



**PES**  
UNIVERSITY  
ONLINE

- There are n people who need to be assigned to n jobs, one person per job. The cost of assigning person i to job j is  $C[i, j]$ .  
Find an assignment that minimizes the total cost

# DESIGN AND ANALYSIS OF ALGORITHMS

## The Assignment Problem

Example

	Job 1	Job 2	Job 3	Job 4
Person 1	9	2	7	8
Person 2	6	4	3	7
Person 3	5	8	1	8
Person 4	7	6	9	4

# DESIGN AND ANALYSIS OF ALGORITHMS

## The Assignment Problem

Optimization . Minimization  
 Only one person is assigned to execute  
 one job.  
 total cost = minimum



PES  
UNIVERSITY  
ONLINE

Example

	Job 1	Job 2	Job 3	Job 4
Person 1	9	2	7	8
Person 2	6	4	3	7
Person 3	5	8	1	8
Person 4	7	6	9	4

$$\begin{array}{l}
 \begin{array}{cccc}
 P_1 & P_2 & P_3 & P_4 \\
 \text{Job 1} & 1 & 2 & 3 & 4 \\
 \text{Job 2} & 3 & 1 & 4 & 2 \\
 \text{Job 3} & 4 & 2 & 1 & 3 \\
 \text{Job 4} & 2 & 4 & 3 & 1
 \end{array} \\
 = 9 + 4 + 1 + 4 = 18 \\
 \text{Cost} = 2 + 3 + 5 + 4 = 14
 \end{array}$$

## The Assignment Problem

---

Algorithmic Plan ✓

1. Generate all legitimate assignments → "permutations"
2. Compute their costs ✓
3. Select the cheapest one ✓ "minimization problem"

## The Assignment Problem



### The Assignment Problem by Exhaustive Search

$$C = \begin{bmatrix} 9 & 2 & 7 & 8 \\ 6 & 4 & 3 & 7 \\ 5 & 8 & 1 & 8 \\ 7 & 6 & 9 & 4 \end{bmatrix}$$

*4x4*

$\langle 1, 2, 3, 4 \rangle$	cost = $9 + 4 + 1 + 4 = 18$	$4! = 24$ etc. . . .
$\langle 1, 2, 4, 3 \rangle$	cost = $9 + 4 + 8 + 9 = 30$	
$\langle 1, 3, 2, 4 \rangle$	cost = $9 + 3 + 8 + 4 = 24$	
$\langle 1, 3, 4, 2 \rangle$	cost = $9 + 3 + 8 + 6 = 26$	
$\langle 1, 4, 2, 3 \rangle$	cost = $9 + 7 + 8 + 9 = 33$	
$\langle 1, 4, 3, 2 \rangle$	cost = $9 + 7 + 1 + 6 = 23$	

**FIGURE 3.9** First few iterations of solving a small instance of the assignment problem by exhaustive search.

*1=3*  
*3 permutations*  
 1 2 3  
 1 3 2  
 2 1 3 =  $6 = 3!$   
 2 3 1  
 3 1 2  
 etc., 3 2 1

	Job 1	Job 2	Job 3	Job 4
Person 1	9	2	7	8
Person 2	6	4	3	7
Person 3	5	8	1	8
Person 4	7	6	9	4

# DESIGN AND ANALYSIS OF ALGORITHMS

## The Assignment Problem

→ can be solved efficiently

using a method  
"Hungarian method".

### Efficiency

- The Assignment Problem is solved by generating all permutations of n
- The number of permutations for a given number n is  $n!$
- Therefore, the exhaustive search is impractical for all but very small instances of the problem

- time
- |               |          |
|---------------|----------|
| 1. <u>TSP</u> | $(n-1)!$ |
| 2. knapsack   | $2^n$    |
| 3. Assignment | $n!$     |
- } NP Hard problems

DAA

J<sub>1</sub>

Find the  
optimal solution  
for the  
assignment  
problem given



**PES**  
UNIVERSITY  
ONLINE

Problem 4  
page 113.

THANK YOU

Person	J <sub>1</sub>	J <sub>2</sub>	J <sub>3</sub>	J <sub>4</sub>
1	4	3	8	6
2	5	7	2	4
3	16	9	3	1
4	2	5	3	7

Total Cost is minimum.

Bharathi R

Department of Computer Science  
& Engineering