
# Design and Analysis of Algorithms

**Vandana M L**
Department of Computer Science & Engineering

# DESIGN AND ANALYSIS OF ALGORITHMS

## Introduction to Algorithms, Design Techniques and Analysis

Slides courtesy of **Anany Levitin**

**Vandana M L**
Department of Computer Science & Engineering

# Design and Analysis of Algorithms

## Syllabus

- **UNIT I (12 Hours)**
  - Introduction
  - Analysis of Algorithm Efficiency,
  - Algebric structures
- **UNIT II (12 Hours)**
  - Brute Force,
  - Divide-and-Conquer
- **UNIT III (10 Hours)**
  - Decrease-and-Conquer
  - Transform-and-Conquer
- **UNIT IV (10 Hours)**
  - Space and Time Tradeoffs
  - Greedy Technique
- **UNIT V (12 Hours)**
  - Limitations of Algorithm Power
  - Coping with the Limitations of Algorithm Power
  - Dynamic Programming

# 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<br>Anany Levitin | 2 | Pearson | 2012 |
| Reference Book | R1 | Introduction to Algorithms<br>Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein | 3 | Prentice-Hall India | 2009 |
| Reference Book | R2 | Fundamentals of Computer Algorithms<br>Horowitz, Sahni, Rajasekaran, | 2 | Universities Press | 2007 |
| Reference Book | R3 | Algorithm Design<br>Jon Kleinberg, Eva Tardos, | 1 | Pearson Education | 2006 |

# 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.

## Characteristics of Algorithm

Input: Zero or more quantities are externally supplied

Definiteness: Each instruction is clear and unambiguous

Finiteness: The algorithm terminates in a finite number of steps.

Effectiveness: Each instruction must be primitive and feasible

Output: At least one quantity is produced

# Algorithm

## 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

# Design and Analysis of Algorithms

## Basic Issues related to Algorithms

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

# Design and Analysis of Algorithms

## Basic Issues related to Algorithms

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

## 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

- Brute Force
- Divide and Conquer
- Decrease and Conquer
- Transform and Conquer
- Dynamic Programming
- Greedy Technique
- Branch and Bound
- Backtracking

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

# Design and Analysis of Algorithms

## Basic Issues related to Algorithms

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

## Methods of Specifying an Algorithm

- **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 ← for assignment
- **Flowchart**
  - Method of expressing algorithm by collection of connected geometric shapes

## Euclid's Algorithm

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

Examples: gcd(60,24) = 12, gcd(60,0) = 60, gcd(0,0) = ?

Euclid's algorithm is based on repeated application of equality

gcd(m,n) = gcd(n, m mod n)

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

Example: gcd(60,24) = gcd(24,12) = gcd(12,0) = 12

# Methods of Specifying an Algorithm

## Two descriptions of Euclid's algorithm

Euclid's algorithm for computing 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 gcd(m,n) by Euclid's method

//Input: Two nonnegative,not both zero integers

//Output:Greatest common divisor of m and n

while n ≠ 0 do

    r ← m mod n

    m← n

    n ← r

return m

## Algorithm for 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 ← 0
while i < n and A[i] ≠ K do
    i ← i + 1
if i < n return i
else return −1
```

# Design and Analysis of Algorithms

## Basic Issues related to Algorithms

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

# Design and Analysis of Algorithms

## Basic Issues related to Algorithms

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

## 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

## Complexity of an algorithm

- 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

PES UNIVERSITY ONLINE

## How to determine complexity of an algorithm?

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

## Limitations of Performance Measurement


- 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

## Performance Analysis


- 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


# THANK YOU

**Vandana M L**
Department of Computer Science & Engineering

**vandanamd@pes.edu**


# Design and Analysis of Algorithms
# Unit -4

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

# DESIGN AND ANALYSIS OF ALGORITHMS

## Unit 4: Space and Time Tradeoffs

### Space and Time Tradeoffs - Sorting by Counting

**Bharathi R**

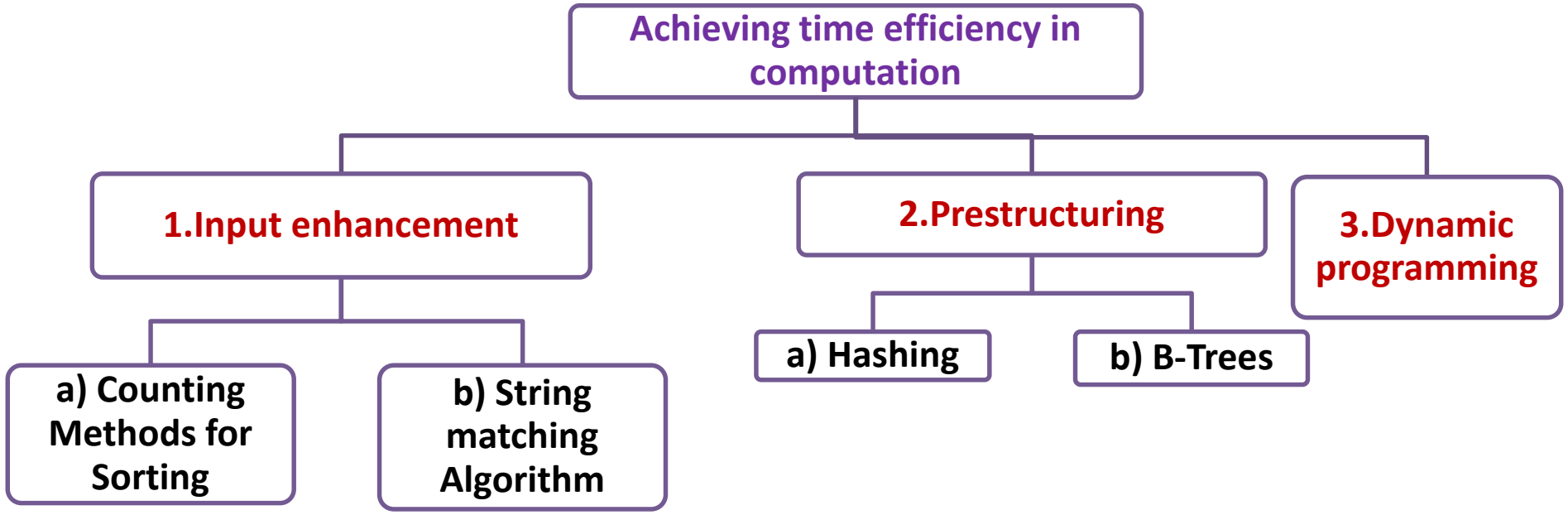
Department of Computer Science & Engineering

## Space and Time Tradeoffs- Introduction

- Space and time trade-offs in algorithm design are a well-known issue for both theoreticians and practitioners of computing.

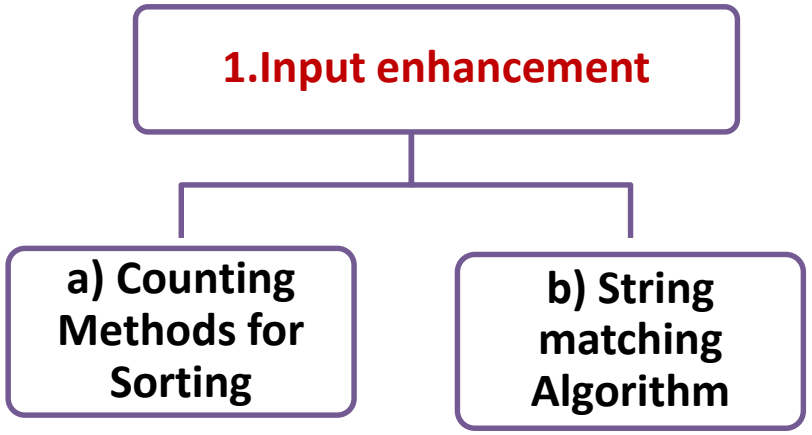
- As an algorithm design technique, trading space for time is much more prevalent than trading time for space.

## Principal Varieties for trading space for time in Algorithm design



## 1.Input enhancement

- **Input Enhancement**
  - Preprocess the problem's input, in whole or in part, and store the additional information obtained to accelerate solving the problem afterward.
  - Eg:
    1. Comparison counting sort
    2. Distribution Counting,
    3. Horspool's algorithm,
    4. Boyer-Moore's algorithm



# 1.Input enhancement

## a)Sorting by Counting

1. **Comparison Counting Sorting**
   I. For each element of the list, count the total number of elements smaller than this element.
   II. These numbers will indicate the positions of the elements in the sorted list.

2. **Distribution Counting Sorting**
   I. Suppose the elements of the list to be sorted belong to a finite set (aka domain).
   II. Count the frequency of each element of the set in the list to be sorted.
   III. Scan the set in order of sorting and print each element of the set according to its frequency, which will be the required sorted list.

# Design and Analysis of Algorithms

## 1.Input Enhancement

### → Sorting by Counting

### →→ Comparison Counting Sorting

1. Find the numbers that are less than a[0] i.e, 62, by scanning the array from the index 1 to 5

| | a[0] | a[1] | a[2] | a[3] | a[4] | a[5] |
|---|---|---|---|---|---|---|
| Array *a* | 62 | 31 | 84 | 96 | 19 | 47 |

2. Maintain another array called ***Count*** the elements that are **lesser than 62**

| | a[0] | a[1] | a[2] | a[3] | a[4] | a[5] |
|---|---|---|---|---|---|---|
| Array *a* | 62 | 31 | 84 | 96 | 19 | 47 |
| Array *Count* | **3** | | | | | |

## Example of sorting by comparison counting

| Array $A[0..5]$ | | 62 | 31 | 84 | 96 | 19 | 47 |
|---|---|---|---|---|---|---|---|
| Initially | *Count* [] | 0 | 0 | 0 | 0 | 0 | 0 |
| After pass $i = 0$ | *Count* [] | 3 | 0 | 1 | 1 | 0 | 0 |
| After pass $i = 1$ | *Count* [] | | 1 | 2 | 2 | 0 | 1 |
| After pass $i = 2$ | *Count* [] | | | 4 | 3 | 0 | 1 |
| After pass $i = 3$ | *Count* [] | | | | 5 | 0 | 1 |
| After pass $i = 4$ | *Count* [] | | | | | 0 | 2 |
| Final state | *Count* [] | 3 | 1 | 4 | 5 | 0 | 2 |

| Array $S[0..5]$ | 19 | 31 | 47 | 62 | 84 | 96 |
|---|---|---|---|---|---|---|

## Algorithm for Sorting by Counting

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

```
//Sorts an array by comparison counting
//Input: An array A[0..n − 1] of orderable elements
//Output: Array S[0..n − 1] of A's elements sorted
for i ← 0 to n − 1 do Count[i] ← 0
for i ← 0 to n − 2 do
    for j ← i + 1 to n − 1 do
        if A[i] < A[j]
            Count[j] ← Count[j] + 1
        else Count[i] ← Count[i] + 1
for i ← 0 to n − 1 do S[Count[i]] ← A[i]
return S
```

It should be quadratic because the algorithm considers all the different pairs of an *n*-element array

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

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

## Drawbacks

1. Thus, the algorithm makes the same number of key comparisons as selection sort,
2. In addition it uses a linear amount of extra space.

## References

"Introduction to the Design and Analysis of Algorithms", Anany Levitin, Pearson Education, Delhi (Indian Version), 3rd edition, 2012.
Chapter- 7


# THANK YOU

**Bharathi R**

Department of Computer Science & Engineering

**rbharathi@pes.edu**


# Design and Analysis of Algorithms

**Vandana M L**
Department of Computer Science & Engineering

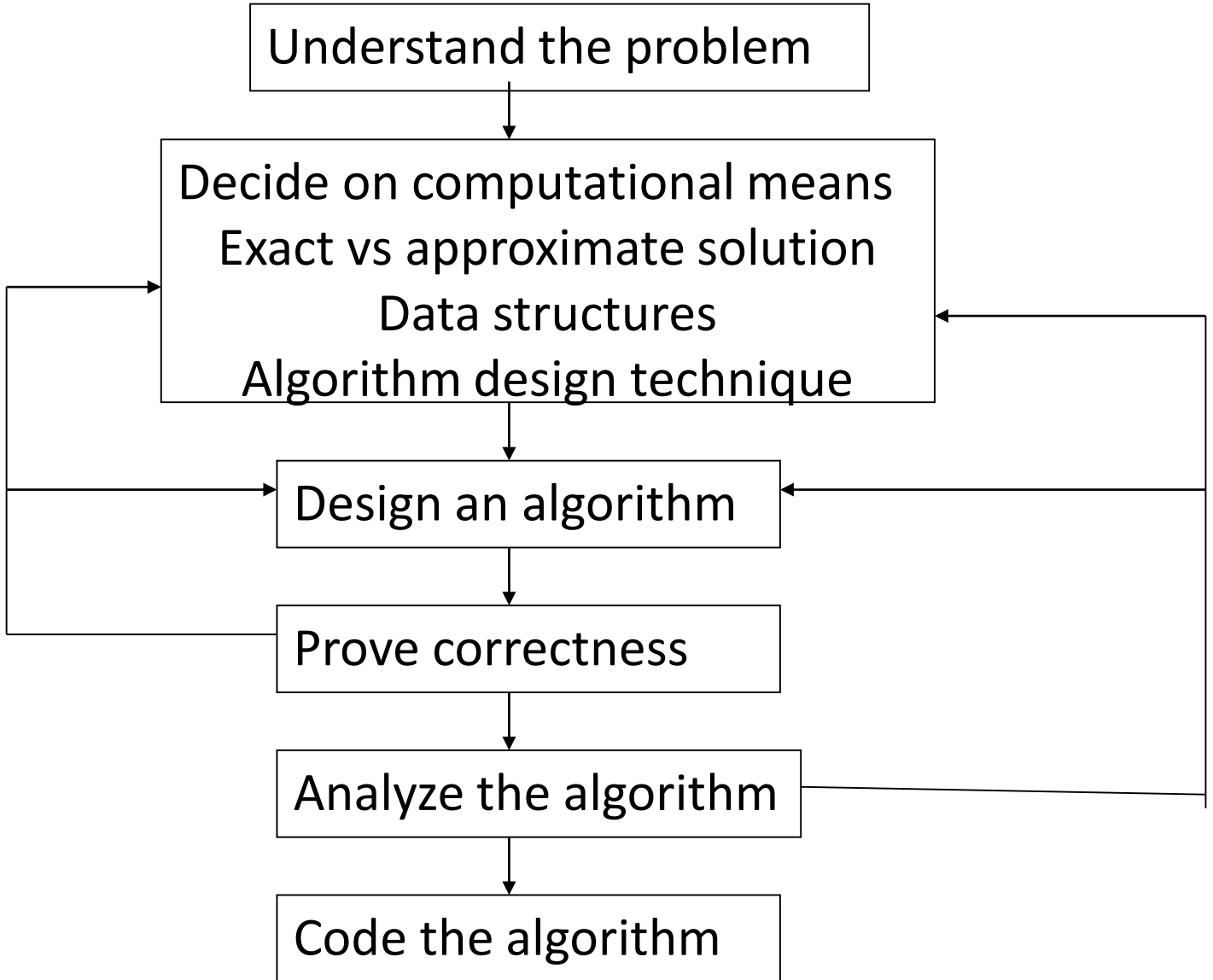
# DESIGN AND ANALYSIS OF ALGORITHMS

## Fundamentals of Algorithmic Problem Solving

Slides courtesy of **Anany Levitin**

**Vandana M L**
Department of Computer Science & Engineering

## Algorithm Design and Analysis Process



# Design and Analysis of Algorithms

## Computational Means


Computational Device the algorithm is intended for

RAM Sequential Algorithms

PRAM Parallel Algorithms

## Exact vs approximate solution


Travelling Salesman Problem

NP complete!!!

Approximate algorithm can be used to solve it

## Deciding on Data Structure

- 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

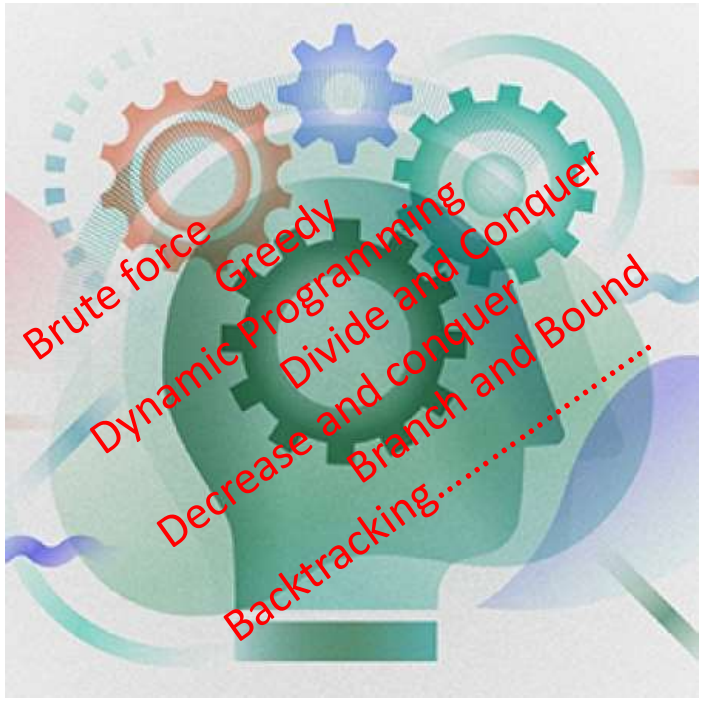
O(VlogV+E) with Fibonacci heap

## Algorithm Design Technique

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



# Design and Analysis of Algorithms

## Specifying an algorithm


- Natural Language
- Pseudo Code
- Flowchart

# Design and Analysis of Algorithms

## Specifying an algorithm


- Natural Language
- Pseudo Code
- Flowchart

# Proving Correctness


## 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

## 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

## Coding an algorithm


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


# THANK YOU

**Vandana M L**

Department of Computer Science & Engineering

**vandanamd@pes.edu**


# Design and Analysis of Algorithms

**Vandana M L**
Department of Computer Science & Engineering

# DESIGN AND ANALYSIS OF ALGORITHMS

## Important Problem Types

Slides courtesy of **Anany Levitin**

**Vandana M L**
Department of Computer Science & Engineering

## Important Problem Types


- sorting
- searching
- string processing
- graph problems
- combinatorial problems
- geometric problems
- numerical problems

## Important Problem Types: Sorting

- Rearrange the items of a given list in ascending order.
  - Input: A sequence of n numbers <a1, a2, …, an>
  - Output: A reordering <$a'_1$, $a'_2$, …, $a'_n$> of the input sequence such that $a'_1 \leq a'_2 \leq \ldots \leq a'_n$.
- 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.

## 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.

## Important Problem Types: Searching

Find a given value, called a search key, in a given set.

Examples of searching algorithms

- Sequential searching
- Binary searching…

## 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

## 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

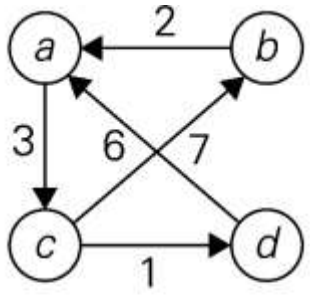
- Modeling WWW
- communication networks
- Project scheduling …

### Examples of graph algorithms

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

## Shortest paths in a graph

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




(a)

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

(b)

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

(c)

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

## Minimum cost spanning tree

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

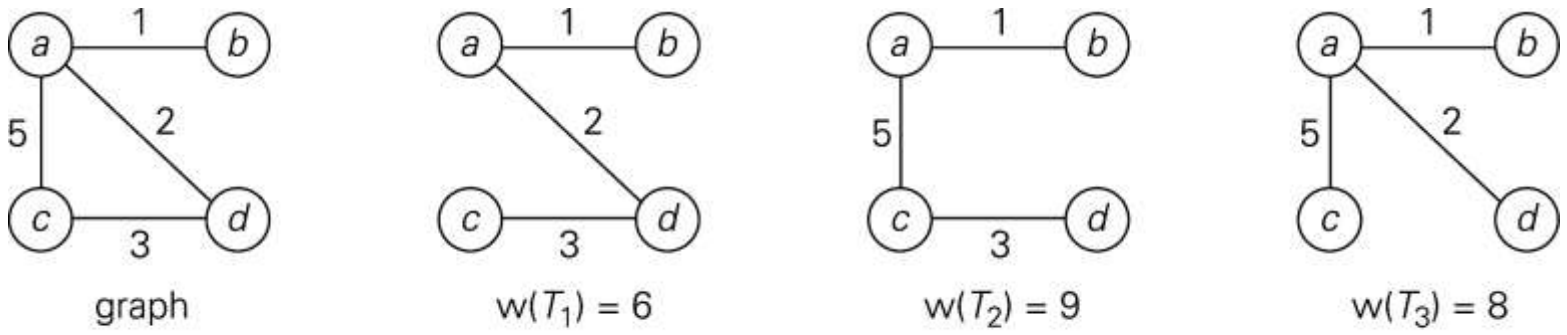
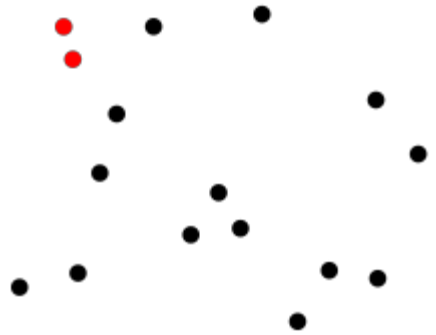



**FIGURE 9.1** Graph and its spanning trees; $T_1$ is the minimum spanning tree

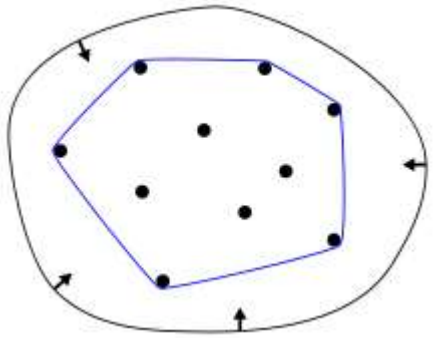
## Important Problem Types: Geometric Problems

PES UNIVERSITY ONLINE

### Closest Pair problem



### Convex Hull Problem



## Important Problem Types: Numerical Problems


- Solving Equations
- Computing definite integrals
- Evaluating functions


# THANK YOU

**Vandana M L**

Department of Computer Science & Engineering

**vandanamd@pes.edu**


# Design and Analysis of Algorithms

**Vandana M L**
Department of Computer Science & Engineering

# DESIGN AND ANALYSIS OF ALGORITHMS

## Analysis Framework

Slides courtesy of **Anany Levitin**

**Vandana M L**
Department of Computer Science & Engineering

## Analysis Framework

### 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

## Complexity of an 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

## Performance of Algorithm


important Criteria for performance:

- Space efficiency - the memory required, also called, space complexity
- Time efficiency - the time required, also called time complexity

## Space Complexity

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

## Space Complexity

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

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

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

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

## Time Complexity

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

- Compile time (C)
  independent of instance characteristics

- run (execution) time TP


## 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

## Limitations of Experimental study


- 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

## Theoretical Analysis


- 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

## Theoretical Analysis

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 executed.
2. *step counts* - determine the total number of steps, possibly lines of code, executed by the program.

## Analysis Framework

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

## Measuring an input's size

Efficiency is defined as a function of input size.

Input size depends on the problem.

Example 1, what is the input size of the problem of sorting n numbers?

Example 2, what is the input size of adding two n by n matrices?

## Units for Measuring Running Time

- 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.

# Analysis in the RAM Model

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

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

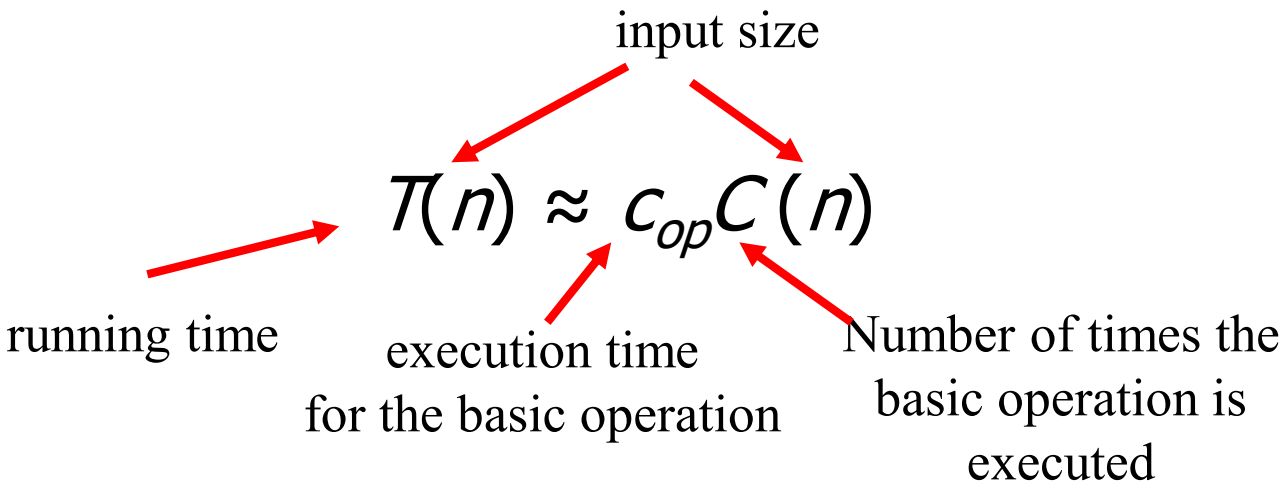
$$T(n) = nC_1 + C_2 \quad \Rightarrow T(n) \text{ is a } \textit{linear function} \text{ of } n$$

## Input Size and Basic Operation Examples

| ***Problem*** | ***Input size measure*** | ***Basic operation*** |
|---|---|---|
| Search for a key in a list of *n* items | Number of items in list, *n* | Key comparison |
| Add two *n* by *n* matrices | Dimensions of matrices, *n* | addition |
| multiply two matrices | Dimensions of matrices, *n* | multiplication |

## Theoretical Analysis of Time Efficiency: Basic operation count

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



## Order of Growth

C(n) Basic Operation Count

- The efficiency analysis framework ignores the multiplicative constants of C(n) and focuses on the orders of growth of the C(n).

- Simple characterization of the algorithm's efficiency by identifying relatively significant term in the C(n).

## Order of Growth

### Why do we care about the order of growth of an algorithm's efficiency function, i.e., the total number of basic operations?

- Because, for smaller inputs, it is difficult to distinguish inefficient algorithms vs. efficient ones.
- For example, if the number of basic operations of two algorithms to solve a particular problem are n and $n^2$ respectively, then
  - if $n = 2$, Basic operation will be executed 2 and 4 times respectively for algorithm1 and 2.

    Not much difference!!!
  - On the other hand, if $n = 10000$, then it does makes a difference whether the number of times the basic operation is executed is $n$ or $n^2$.

## 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

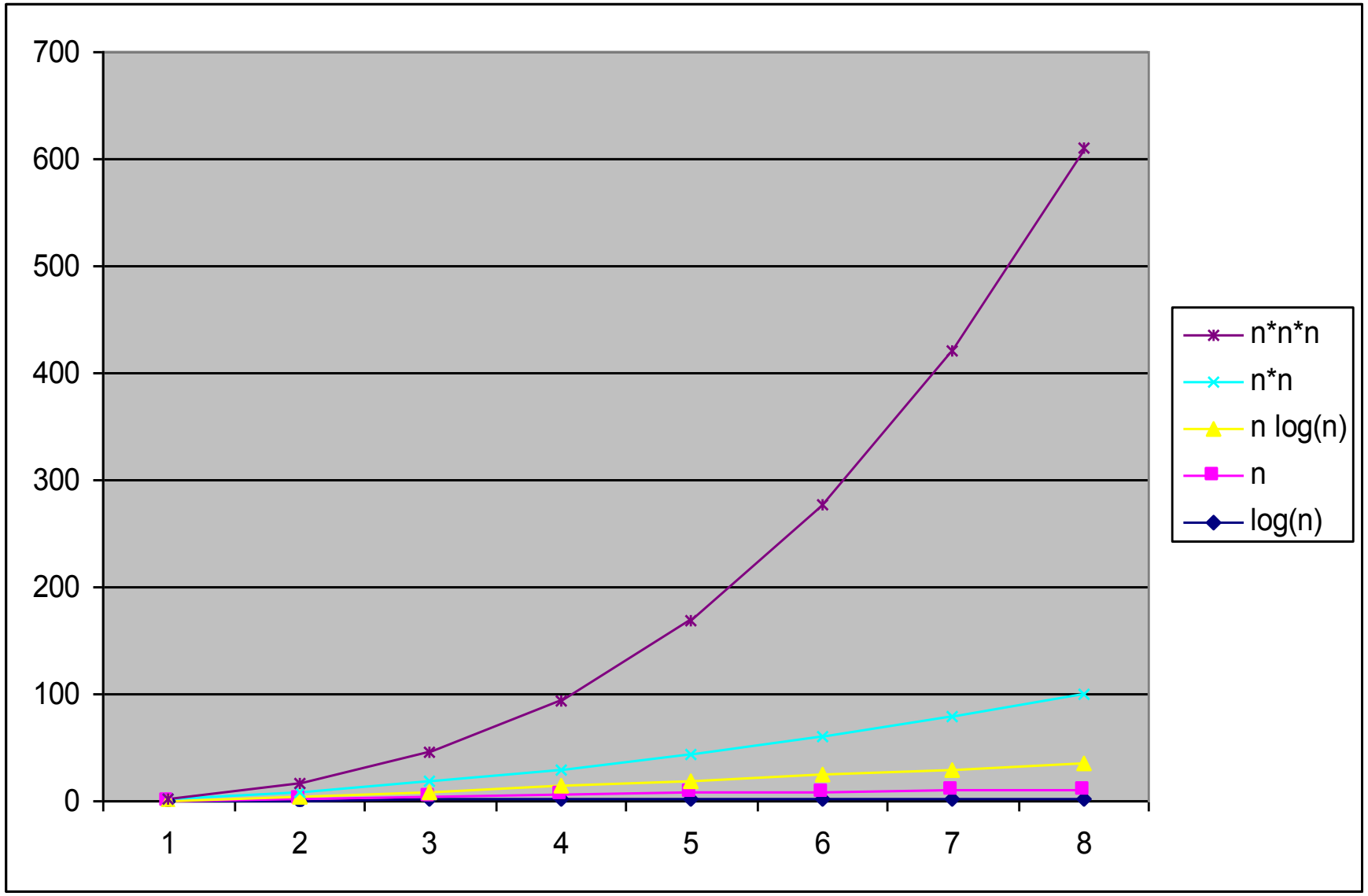
Exponential-growth functions

Orders of growth:

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

# Design and Analysis of Algorithms

## Order of Growth



## Basic Efficiency Classes

| | |
|---|---|
| 1 | constant |
| $\log n$ | logarithmic |
| $n$ | linear |
| $n \log n$ | $n$-log-$n$ |
| $n^2$ | quadratic |
| $n^3$ | cubic |
| $2^n$ | exponential |
| $n!$ | factorial |

# Design and Analysis of Algorithms

## Best, Worst and Average case Analysis

- Algorithm efficiency depends on the input size n
- For some algorithms efficiency depends on type of input.

  Example: Sequential Search

  Problem: Given a list of n elements and a search key K, find an element equal to K, if any.

  Algorithm: Scan the list and compare its successive elements with K until either a matching element is found (successful search) or the list is exhausted (unsuccessful search)

Given a sequential search problem of an input size of n,
what kind of input would make the running time the longest?
How many key comparisons?

# Design and Analysis of Algorithms

## Best, Worst and Average case Analysis

- Worst case Efficiency
  - Efficiency (# of times the basic operation will be executed) for the worst case input of size n.
  - The algorithm runs the longest among all possible inputs of size n.
- Best case
  - Efficiency (# of times the basic operation will be executed) for the best case input of size n.
  - The algorithm runs the fastest among all possible inputs of size n.
- Average case:
  - Efficiency (#of times the basic operation will be executed) for a typical/random input of size n.
  - NOT the average of worst and best case
- How to find the average case efficiency?

## Best, Worst and Average case Analysis

```
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 ←0
  while i < n and A[i] ǂ K do
        i ← i + 1
  if i < n          //A[I] = K
        return i
  else
        return -1
```

## Best, Worst and Average case Analysis: Sequential Search

- Worst-Case: Cworst(n) = n
- Best-Case: Cbest(n) = 1
- Average-Case

  from **(n+1)/2** to **(n+1)**

## Average case Analysis: Sequential Search

Let **'p'** be the probability that key is found in the list

Assumption: All positions are equally probable

Case1: key is found in the list

$C_{avg,case1}(n) = p*(1 + 2 + \ldots + n) / n = p*(n + 1) / 2$

Case2: key is not found in the list

$C_{avg,\ case2}(n) = (1-p)*(n)$

$\mathbf{C_{avg}(n) = p}(n + 1) / 2 \mathbf{+ (1 - p)}(n)$


# THANK YOU

**Vandana M L**

Department of Computer Science & Engineering

**vandanamd@pes.edu**


# Design and Analysis of Algorithms

**Vandana M L**
Department of Computer Science & Engineering

# DESIGN AND ANALYSIS OF ALGORITHMS

## Asymptotic Notations

Slides courtesy of **Anany Levitin**

**Vandana M L**
Department of Computer Science & Engineering

## Asymptotic Notations

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

How do we compare order of growth??

Using Asymptotic Notations

A way of comparing functions that ignores constant factors and small input sizes

- $O(g(n))$: class of functions $f(n)$ that grow *no faster* than $g(n)$
- $\Omega(g(n))$: class of functions $f(n)$ that grow *at least as fast* as $g(n)$
- $\Theta(g(n))$: class of functions $f(n)$ that grow *at same rate* as $g(n)$
- $o(g(n))$: 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 $g(n)$

# Design and Analysis of Algorithms

## O-notation

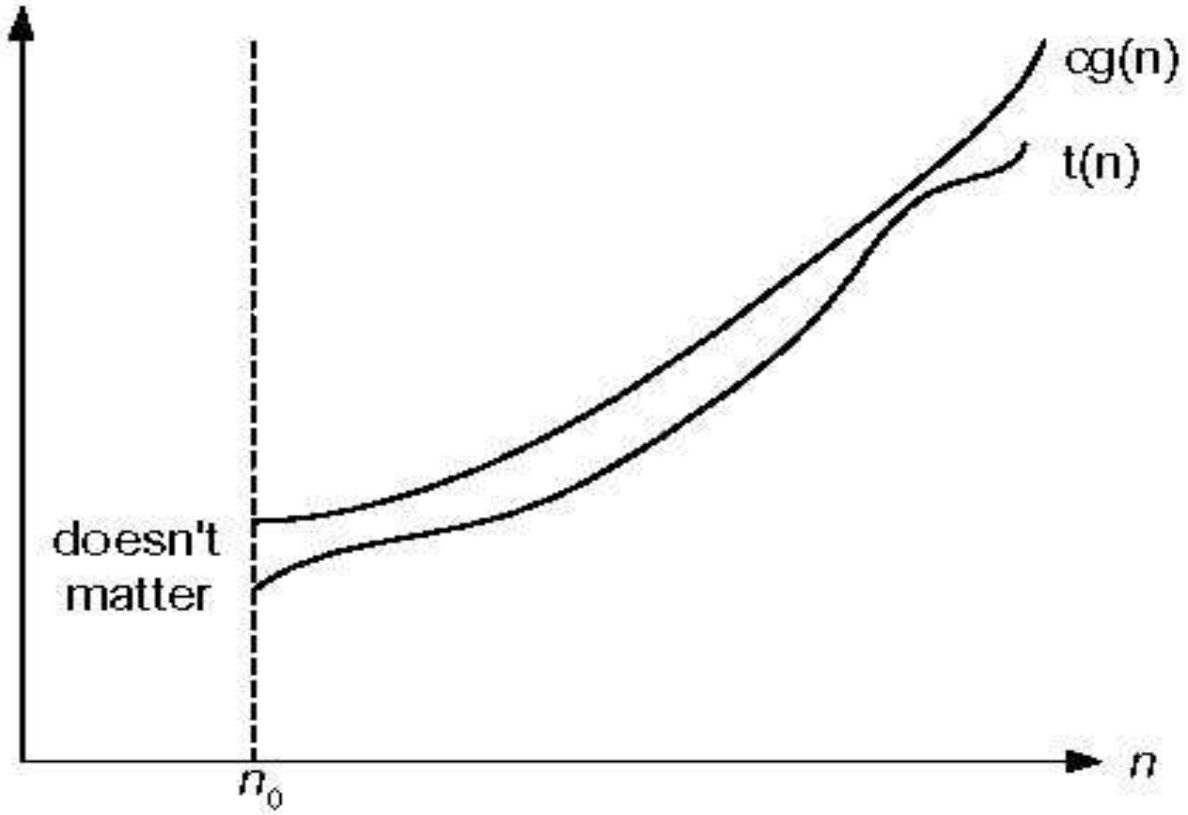



**Figure 2.1** Big-oh notation: $t(n) \in O(g(n))$

## O-notation

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

## Ω-notation

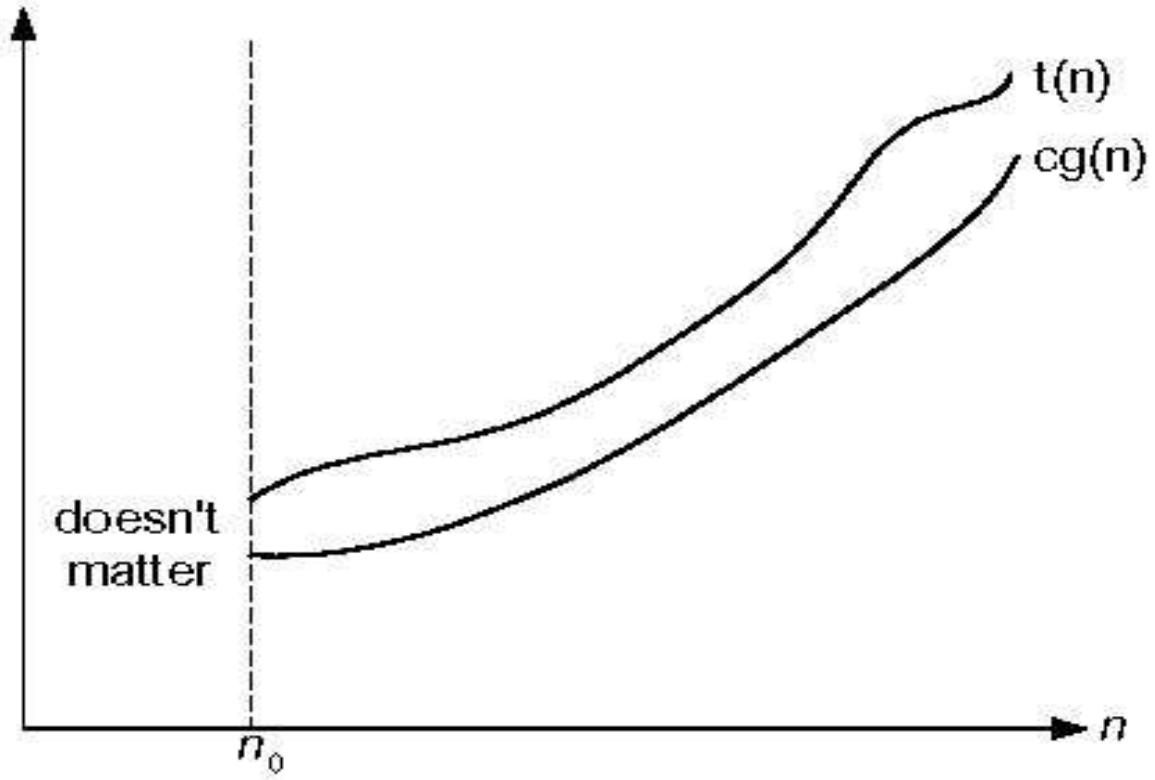



**Fig. 2.2** Big-omega notation: $t(n) \in \Omega(g(n))$

## Ω-notation

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

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

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

# Design and Analysis of Algorithms

## Θ-notation

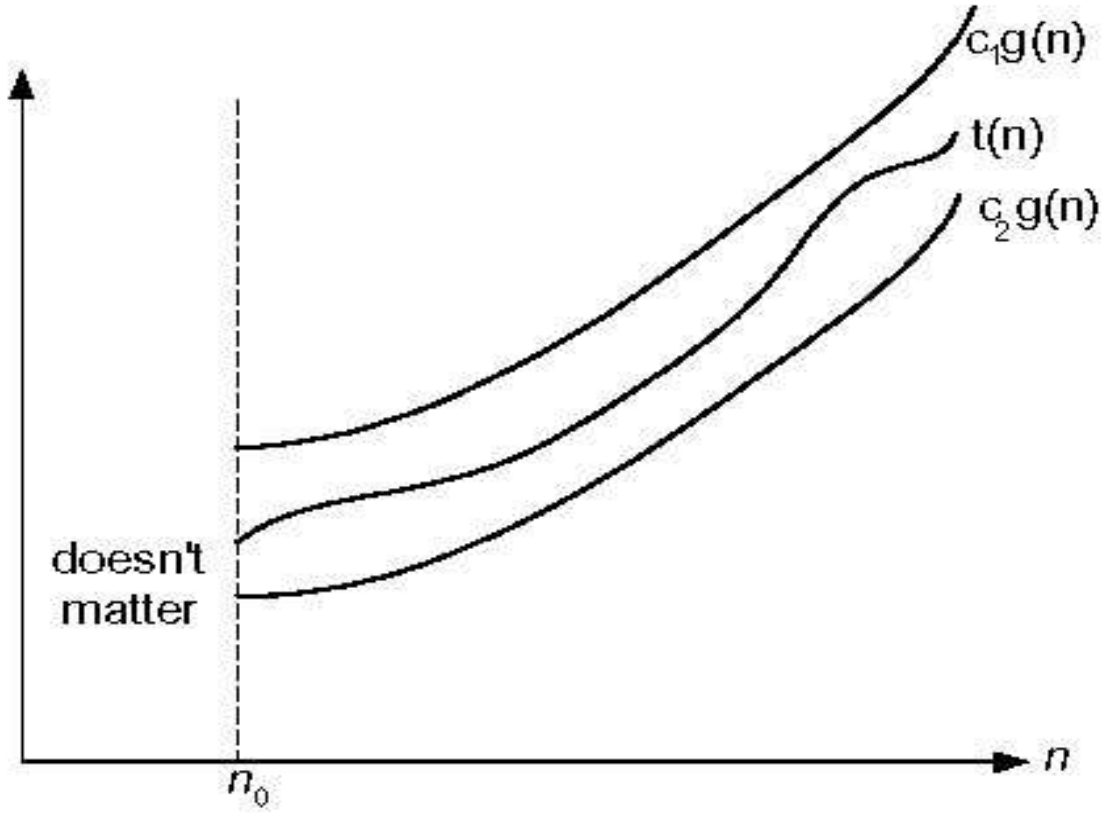



**Figure 2.3** Big-theta notation: $t(n) \in \Theta(g(n))$

## Θ-notation

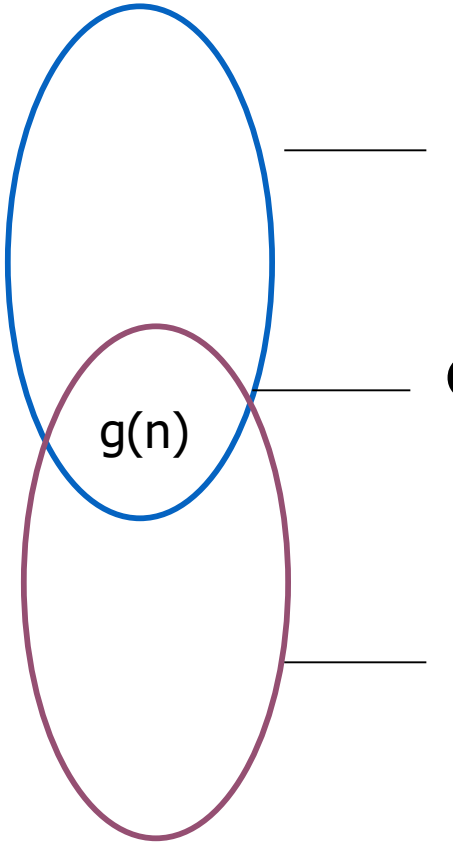
Formal definition

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 constants $c_1$ and $c_2$ and some nonnegative integer $n_0$ such that

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

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




>=

Ω(g(n)), functions that grow at least as fast as g(n)

=

Θ(g(n)), functions that grow at the same rate as g(n)

<=

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

## Little-o Notation

Formal Definition:

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

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

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

## 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 for any positive constant c and some nonnegative integer $n_0$

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

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

## Theorems

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

- If $t1(n) \in \Theta(g1(n))$ and $t2(n) \in \Theta(g2(n))$, then
  $t1(n) + t2(n) \in \Theta(\max\{g1(n), g2(n)\})$

- $t1(n) \in \Omega(g1(n))$ and $t2(n) \in \Omega(g2(n))$, then
  $t1(n) + t2(n) \in \Omega(\max\{g1(n), g2(n)\})$

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


# THANK YOU

**Vandana M L**
Department of Computer Science & Engineering

**vandanamd@pes.edu**


# Design and Analysis of Algorithms

**Vandana M L**
Department of Computer Science & Engineering

# DESIGN AND ANALYSIS OF ALGORITHMS

## Basic Efficiency Classes
## Problems based on Asymptotic notations

Slides courtesy of **Anany Levitin**

**Vandana M L**
Department of Computer Science & Engineering

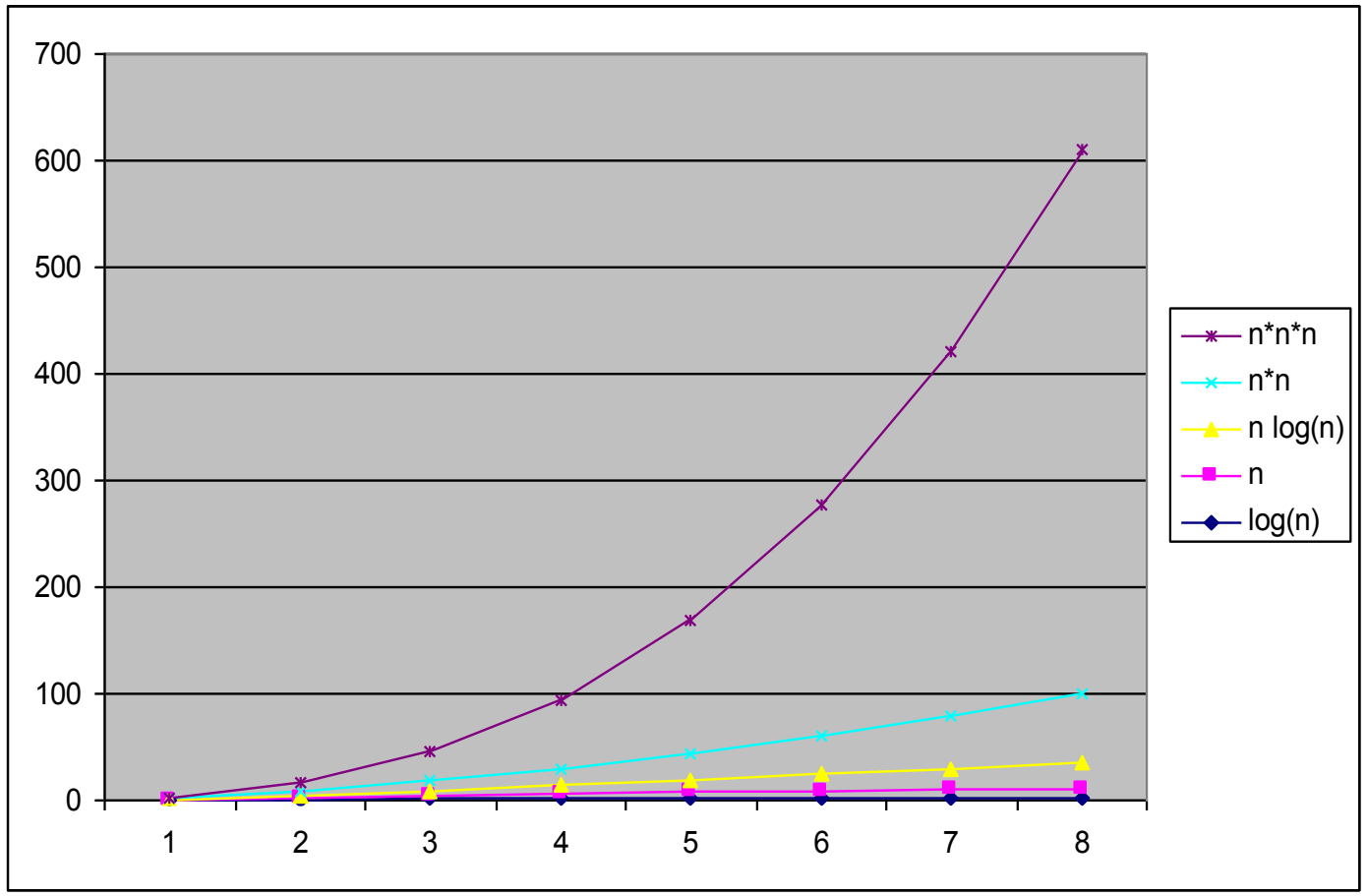
# Design and Analysis of Algorithms

## 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*-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



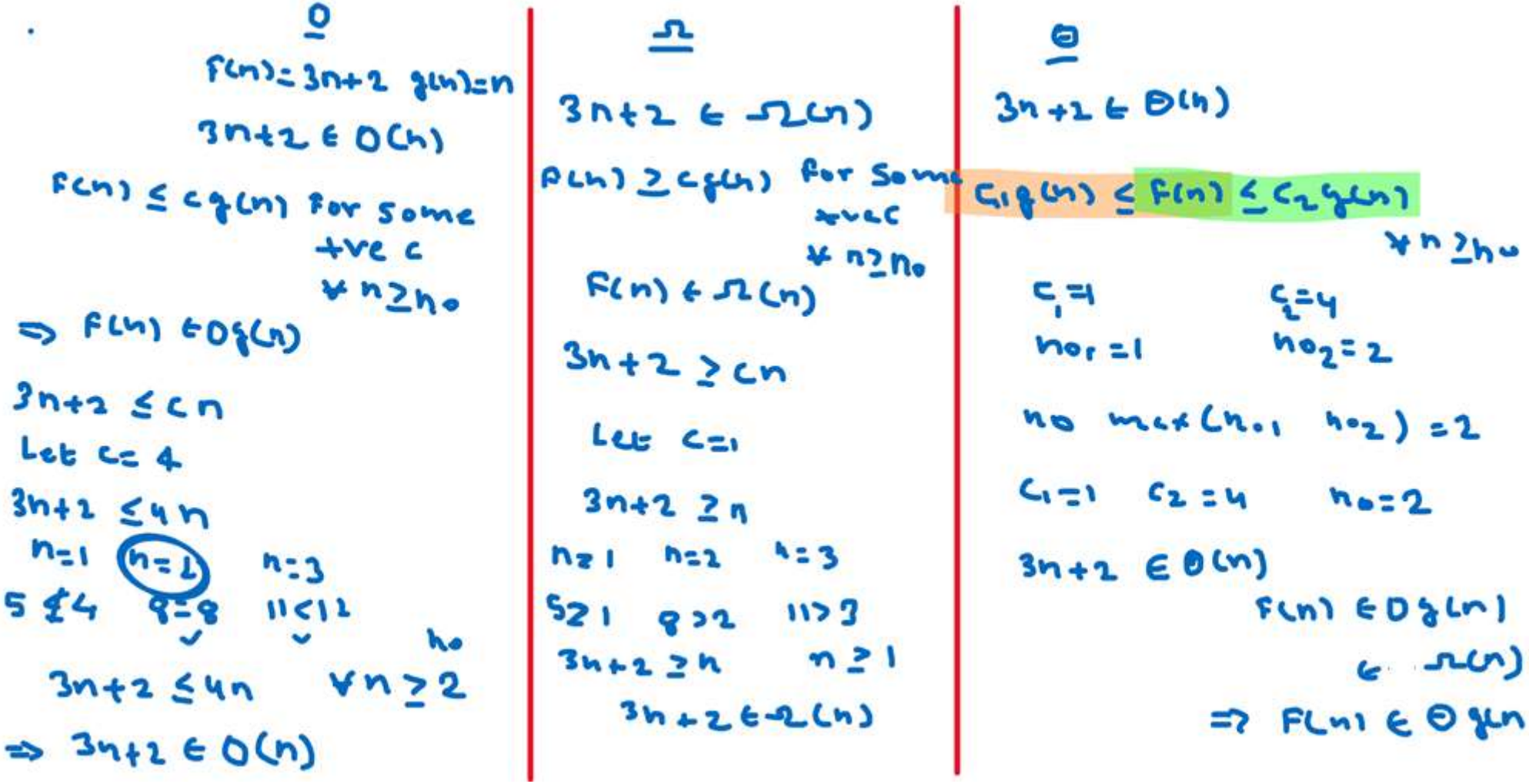
## Basic Efficiency Classes

| $n$ | $\log_2 n$ | $n$ | $n \log_2 n$ | $n^2$ | $n^3$ | $2^n$ | $n!$ |
|---|---|---|---|---|---|---|---|
| 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




# THANK YOU

**Vandana M L**

Department of Computer Science & Engineering

**vandanamd@pes.edu**


# Design and Analysis of Algorithms

**Vandana M L**
Department of Computer Science & Engineering

# DESIGN AND ANALYSIS OF ALGORITHMS

## Method of Limits for comparing order of Growth

Slides courtesy of **Anany Levitin**

**Vandana M L**
Department of Computer Science & Engineering

# Design and Analysis of Algorithms

## Using Limits to Compare Order of Growth

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

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

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

## Using Limits to Compare Order of Growth: Example 1

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

$t(n) = 5n3 + 6n + 2\,, \qquad g(n) = n4$

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

**As per case1**

**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) = Θ(g(n))

Leading term in square root n2

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

$$\lim_{n\to\infty} \frac{t(n)}{g(n)} = \lim_{n\to\infty} \frac{\sqrt{5n^2 + 4n + 2}}{\sqrt{n^2}}$$

$$= \lim_{n\to\infty} \sqrt{\frac{5n^2 + 4n + 2}{n^2}} = \lim_{n\to\infty} \sqrt{5 + \frac{4}{n} + \frac{2}{n^2}} = \sqrt{5}$$

**non-zero constant**

**Hence, t(n) = Θ(g(n)) = Θ(n)**

# Design and Analysis of Algorithms

## Using Limits to Compare Order of Growth

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

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

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

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

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

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

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

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

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

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

# Design and Analysis of Algorithms

## Orders of growth of some important functions

- 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:

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

- Exponential functions an have different orders of growth for different a's

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

- order log n < order $n^\alpha$ ($\alpha$>0) < order $a^n$ < order n! < order $n^n$

# How to Establish Orders of Growth of an Algorithm's Basic Operation Count


## 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

**Vandana M L**
Department of Computer Science & Engineering

**vandanamd@pes.edu**


# Design and Analysis of Algorithms

**Vandana M L**

Department of Computer Science & Engineering

# DESIGN AND ANALYSIS OF ALGORITHMS

## Mathematical Analysis of Non-recursive Algorithms

Slides courtesy of **Anany Levitin**

**Vandana M L**
Department of Computer Science & Engineering

# Time Efficiency of Non-recursive Algorithms

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

- Decide on parameter n indicating input size
- Identify algorithm's basic operation
- 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.
- Set up summation for C(n) reflecting the number of times the algorithm's basic operation is executed.
- Simplify summation using standard formulas

# Design and Analysis of Algorithms

## Useful Summation Formulas and Rules

$\Sigma_{l \le i \le u} 1 = 1+1+\ldots+1 = u - l + 1$

$\Sigma_{1 \le i \le n} i = 1+2+\ldots+n = n(n+1)/2$

$\Sigma_{1 \le i \le n} i^2 = 1^2+2^2+\ldots+n^2 = n(n+1)(2n+1)/6$

$\Sigma_{0 \le i \le n} a^i = 1 + a + \ldots + a^n = (a^{n+1} - 1)/(a - 1)$ for any $a \neq 1$

$\Sigma(a_i \pm b_i) = \Sigma a_i \pm \Sigma b_i \qquad \Sigma c a_i = c \Sigma a_i$

$\Sigma_{l \le i \le u} a_i = \Sigma_{l \le i \le m} a_i + \Sigma_{m+1 \le i \le u} a_i$

$$\sum_{i=l}^{u} 1 = (u - l + 1)$$

## 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 arrays of size n.
- Number of comparisons

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

## 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 ← 0 to n - 2 do
    for j ← 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

# Design and Analysis of Algorithms

## Example 2: Element Uniqueness Problem

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

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

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

Best-case: 1 comparison

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

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

## 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 ← 0 to n - 1 do
    for j ← 0 to n – 1 do
            C[i, j] ← 0.0
             for k ← 0 to n – 1 do
                    C[i, j] ← C[i, j] + A[i, k] * B[k, j]
return C
```

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


# THANK YOU

**Vandana M L**
Department of Computer Science & Engineering

**vandanamd@pes.edu**


# Design and Analysis of Algorithms

**Vandana M L**
Department of Computer Science & Engineering

# DESIGN AND ANALYSIS OF ALGORITHMS

## Mathematical Analysis of Recursive Algorithms

Slides courtesy of **Anany Levitin**

**Vandana M L**
Department of Computer Science & Engineering

## Steps in Mathematical Analysis of Recursive Algorithms

- Decide on parameter n indicating input size
- Identify algorithm's basic operation
- 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
- 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
- Solve the recurrence or estimate the order of magnitude of the solution

## Important Recurrence Types

- 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

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

# Design and Analysis of Algorithms
## Decrease-by-one Recurrences

- One (constant) operation reduces problem size by one.

  $T(n) = T(n-1) + c$ $\quad\quad$ $T(1) = d$

  Solution: $T(n) = (n-1)c + d$ $\quad\quad$ ***linear***

- A pass through input reduces problem size by one.

  $T(n) = T(n-1) + cn$ $\quad\quad$ $T(1) = d$

  Solution: $T(n) = [n(n+1)/2 - 1]\, c + d$ $\quad\quad$ ***quadratic***

## Methods to solve recurrences


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

# Design and Analysis of Algorithms

## Recursive Evaluation of n!

$n! = 1 * 2 * \ldots * (n-1) * n$ for $n \geq 1$ and $0! = 1$

Recursive definition of $n!$:

$F(n) = F(n-1) * n$ for $n \geq 1$

$F(0) = 1$

**input size?**

**basic operation?**

**Best/Worst/Average Case?**

**ALGORITHM** $F(n)$

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

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

**M(n-1) = M(n-2) + 1;** **M(n-2) = M(n-3)+1**

**M(n) = n**

**Overall time Complexity: Θ(n)**

# Design and Analysis of Algorithms

## 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(⌊n/2⌋) + 1
```

input size?

basic operation?

Best/Worst/Average Case?

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

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

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

## 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)
```

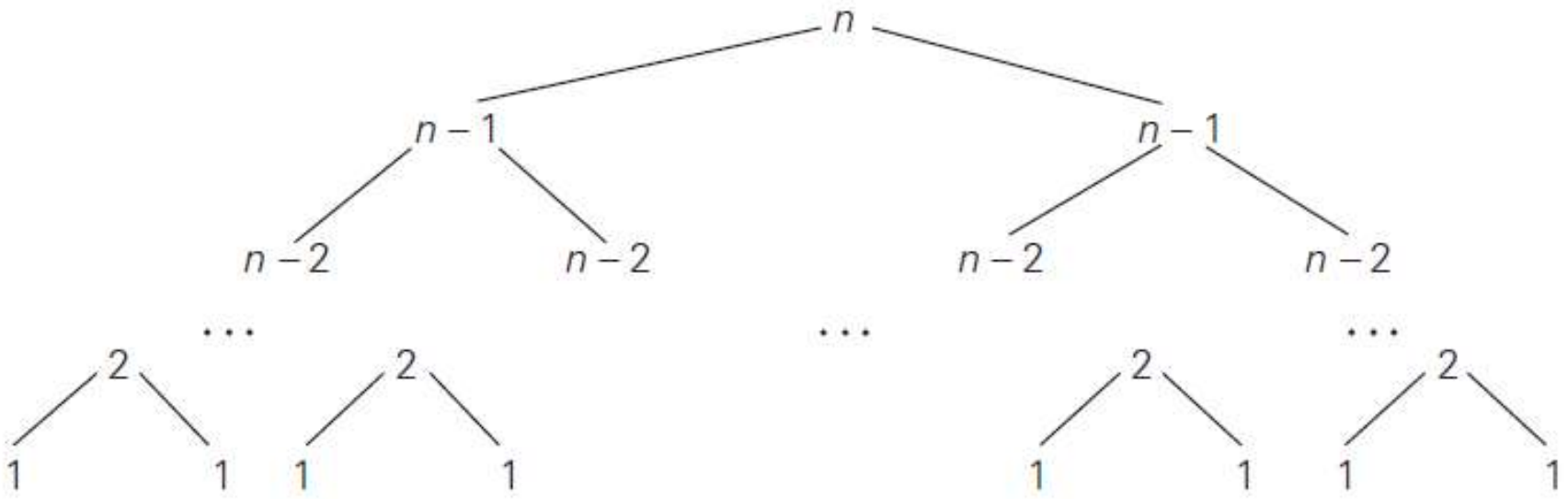
Input Size: **n**

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 : Tree of Recursive calls




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


# THANK YOU

**Vandana M L**
Department of Computer Science & Engineering

**vandanamd@pes.edu**


# Design and Analysis of Algorithms

**Vandana M L**
Department of Computer Science & Engineering

# DESIGN AND ANALYSIS OF ALGORITHMS

## Solving Recurrences

Slides courtesy of **Anany Levitin**

**Vandana M L**
Department of Computer Science & Engineering

## Solving Recurrences: Example1

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

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

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

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

$\ldots$

$= T(n-i) + i$

$\ldots$

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

## Solving Recurrences: Example2

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

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

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

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

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

...

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

...

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

$= 0 + 2 + 4 + \ldots + 2n - n$

$= 2 + 4 + \ldots + 2n - n$

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

// arithmetic progression formula $1+\ldots+n = n(n+1)/2$ //

$= O(n^2)$

# 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

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

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

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

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

.....

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

.....

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

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

$$= \mathrm{O}(n\log\ n)$$


# THANK YOU

**Vandana M L**
Department of Computer Science & Engineering

**vandanamd@pes.edu**


# 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

# Design and Analysis of Algorithms

## Performance Evaluation of Algorithm

- Performance Analysis
  - Machine Independent
  - Prior Evaluation
- Performance Measurement
  - Machine Dependent
  - Posterior Evaluation

## 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 ←0
  while i < n and A[i] ǂ K do
        i ← i + 1
  if i < n          //A[i] = K
        return i
  else
        return -1
```

Basic operation: A[i] ǂ K

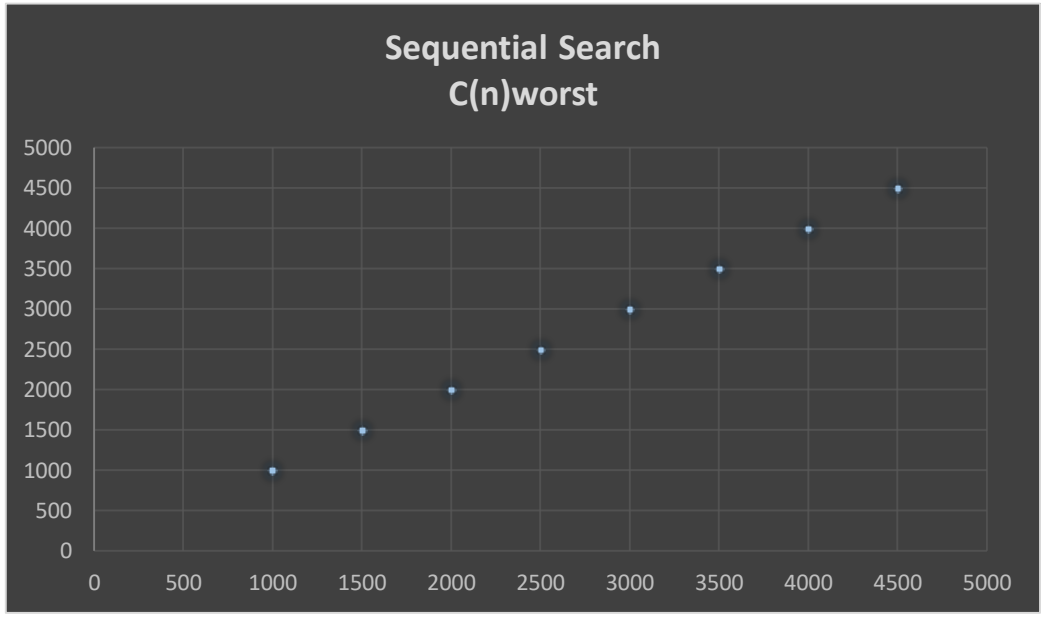
Basic operation count: n

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

# Design and Analysis of Algorithms

## Performance Analysis of Sequential Search

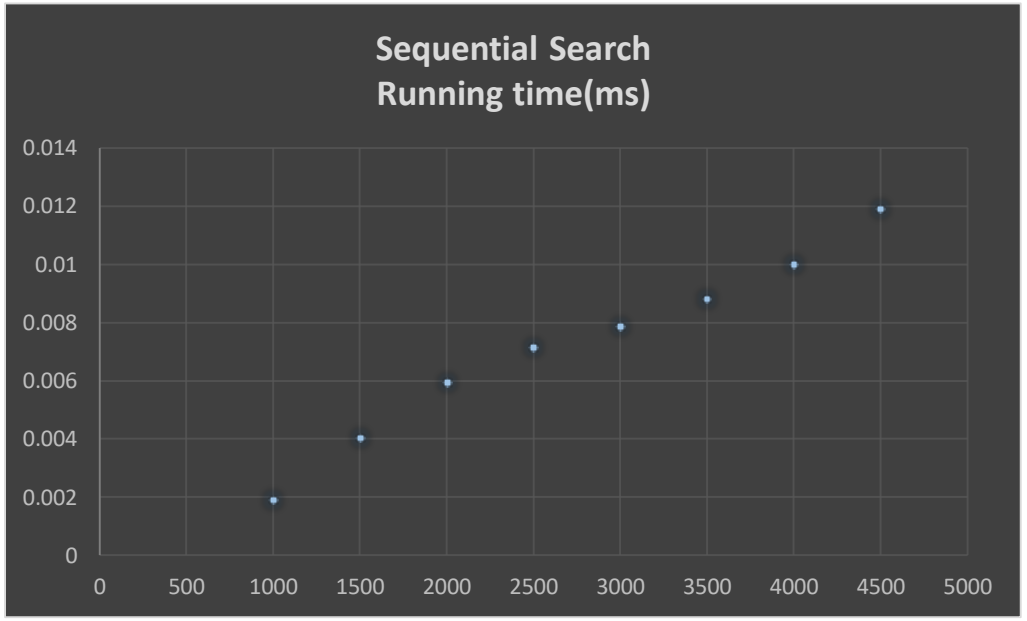
| Input Size | Sequential Search C(n)worst |
|---|---|
| 1000 | 1000 |
| 1500 | 1500 |
| 2000 | 2000 |
| 2500 | 2500 |
| 3000 | 3000 |
| 3500 | 3500 |
| 4000 | 4000 |
| 4500 | 4500 |



# 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**


# DESIGN AND ANALYSIS OF ALGORITHMS
# UE19CS251

**Shylaja S S**
Department of Computer Science
& Engineering

# DESIGN AND ANALYSIS OF ALGORITHMS

## Brute Force: Selection Sort

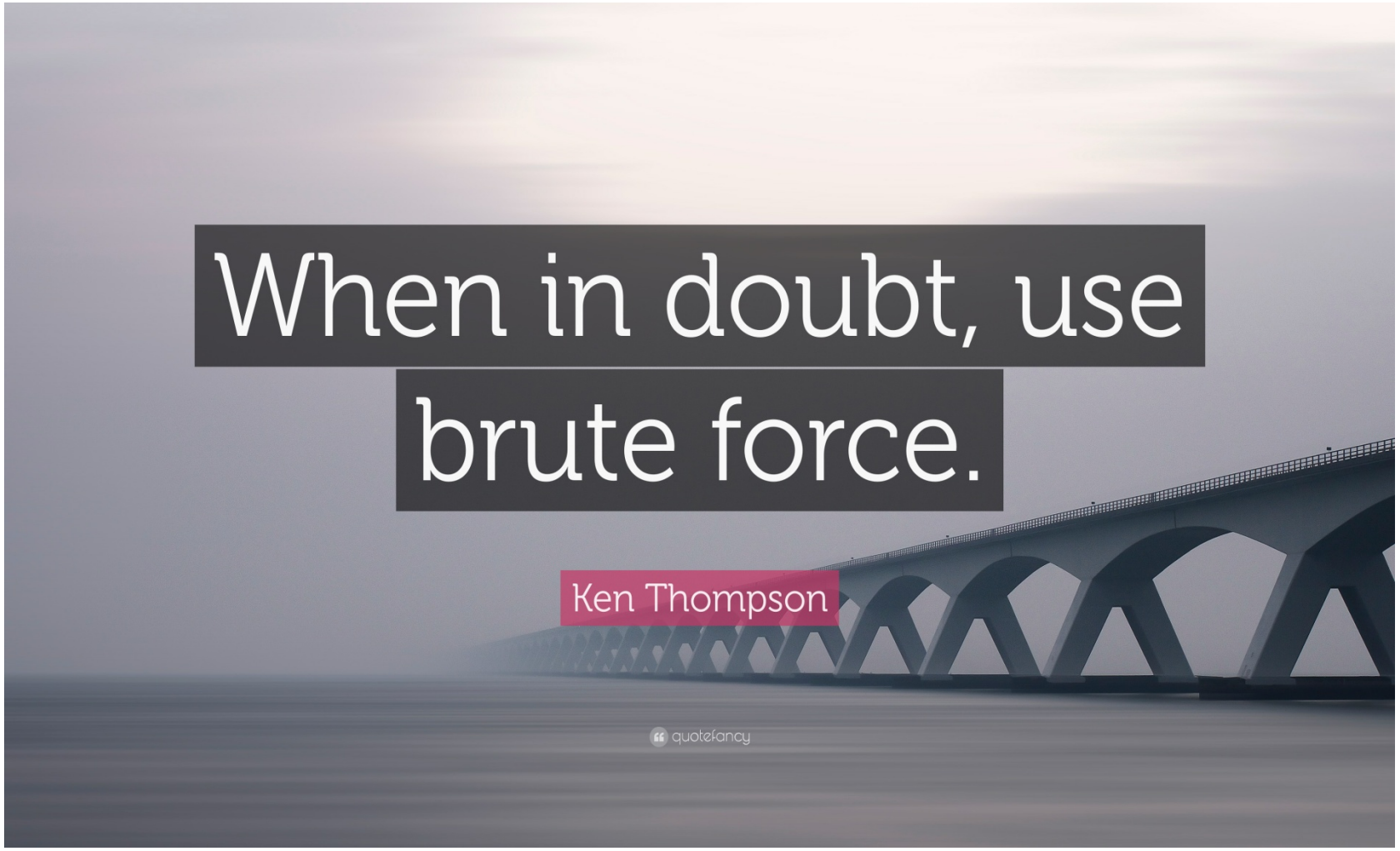
Major Slides Content: Anany Levitin

**Shylaja S S**

Department of Computer Science & Engineering

# DESIGN AND ANALYSIS OF ALGORITHMS

## Brute Force



## Brute Force

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

## 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

https://en.wikipedia.org/wiki/Brute-force_search

## 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 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

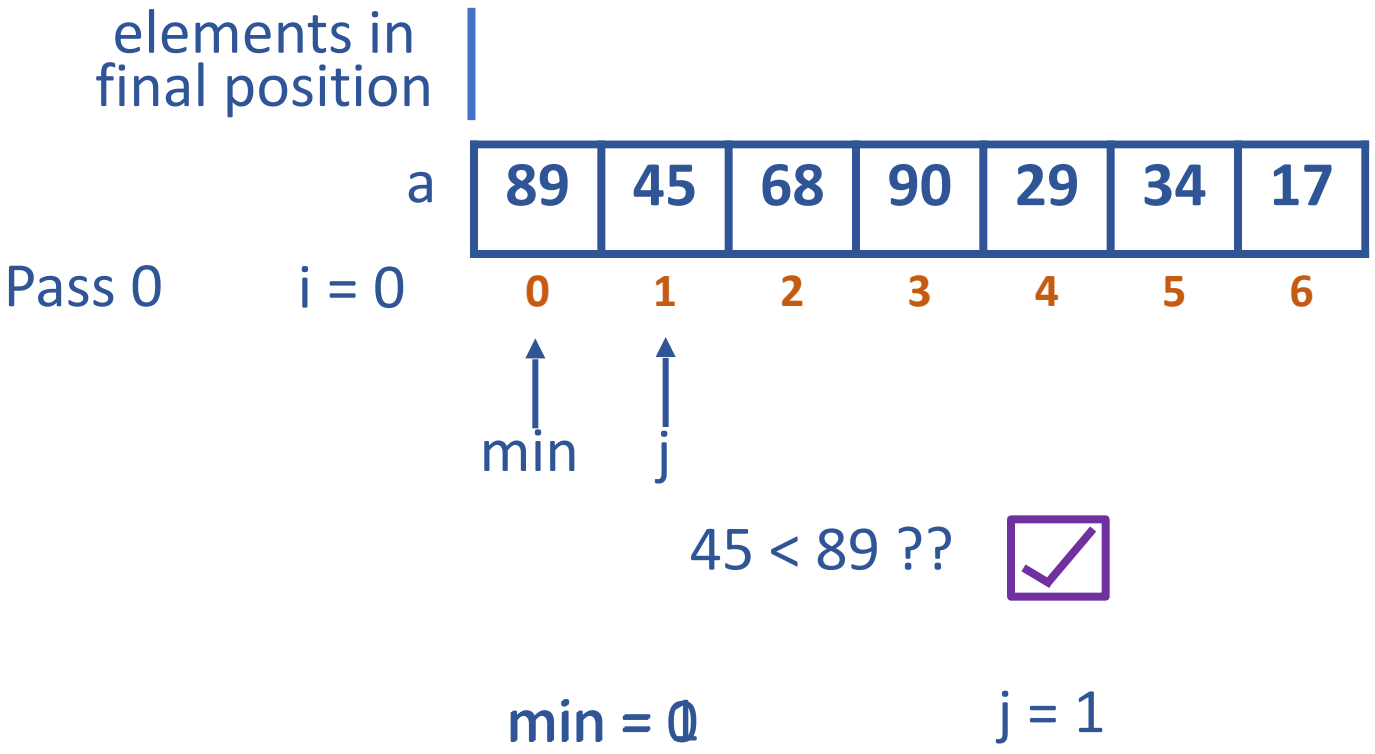

- Selection Sort
- Bubble Sort

## Selection Sort

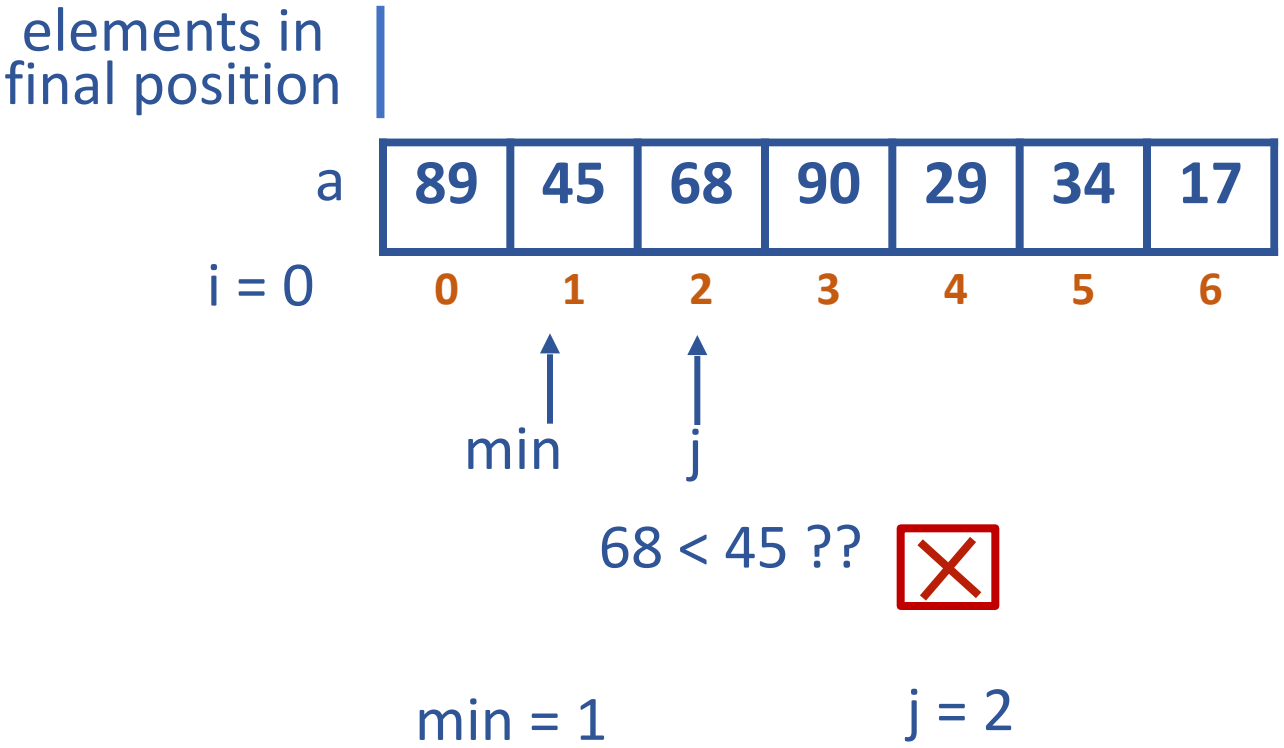
- Scan the array to find its smallest element and swap it with the first element, putting the smallest element in its final position in the sorted list
- Then, starting with the second element, scan the elements to the right of it to find the smallest among them and swap it with the second element, putting 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 $A[i]$

- $A[0] \leq A[1] \leq A[2] \ldots \leq A[i-1]$ | A[i], ....., A[min], ...., A[n-1]
- in their final positions the last n – i elements

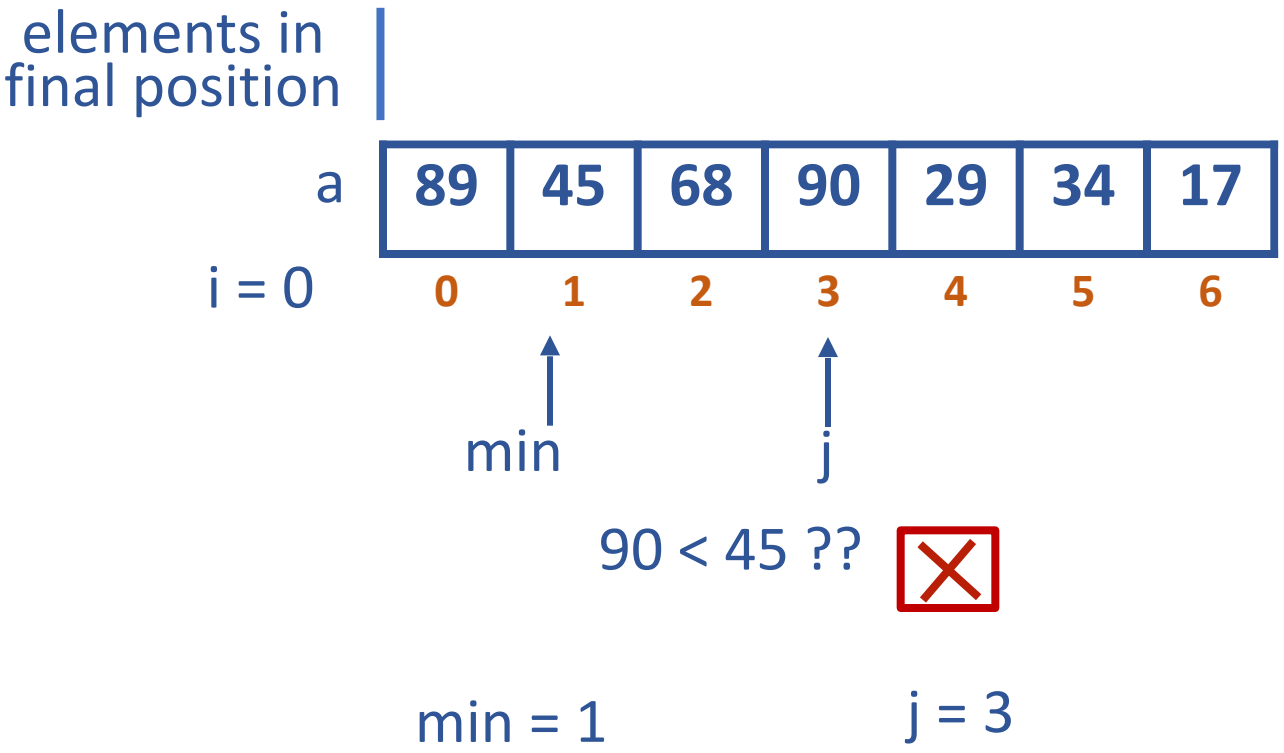
## Selection Sort



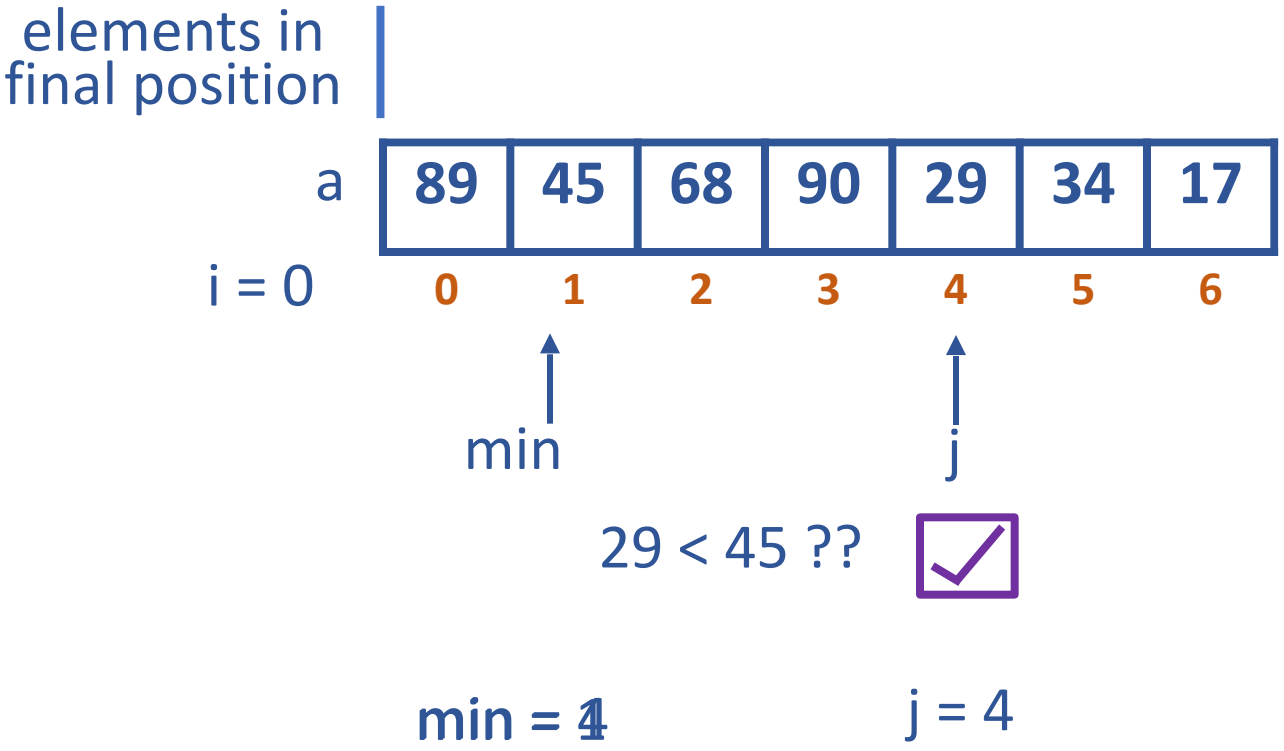
## Selection Sort



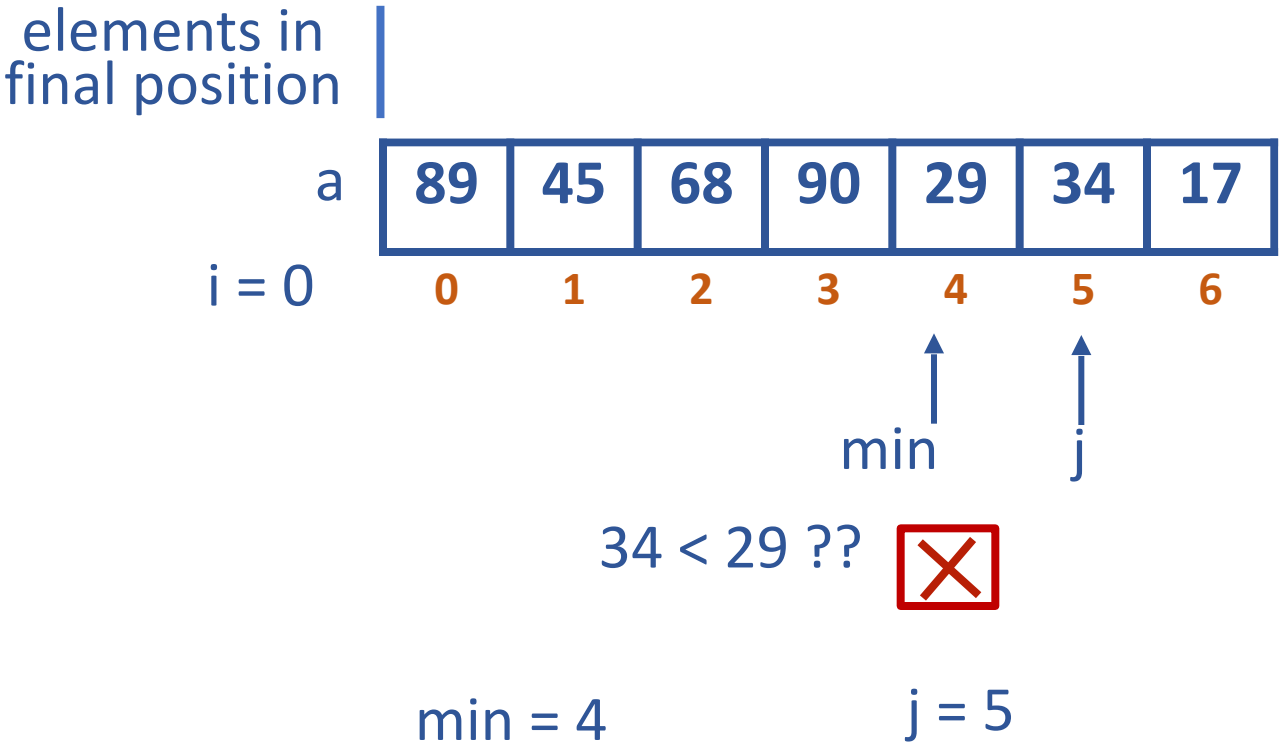
## Selection Sort



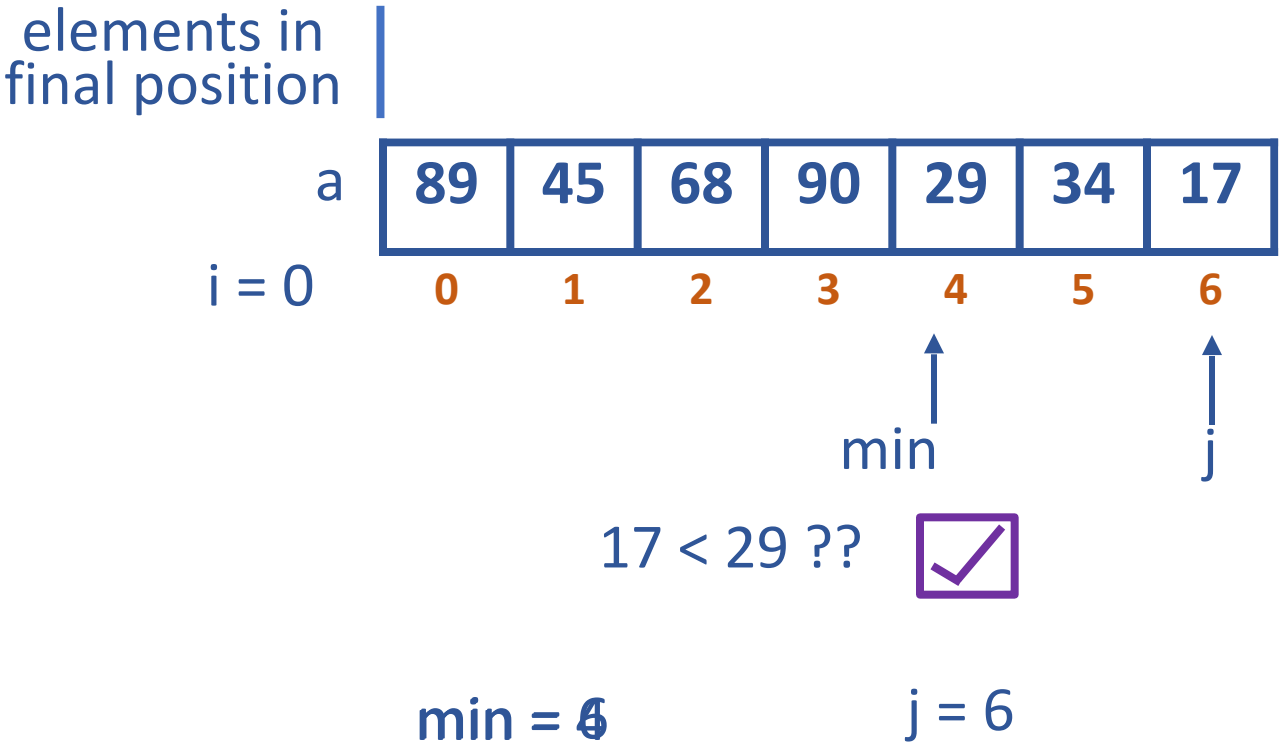
## Selection Sort



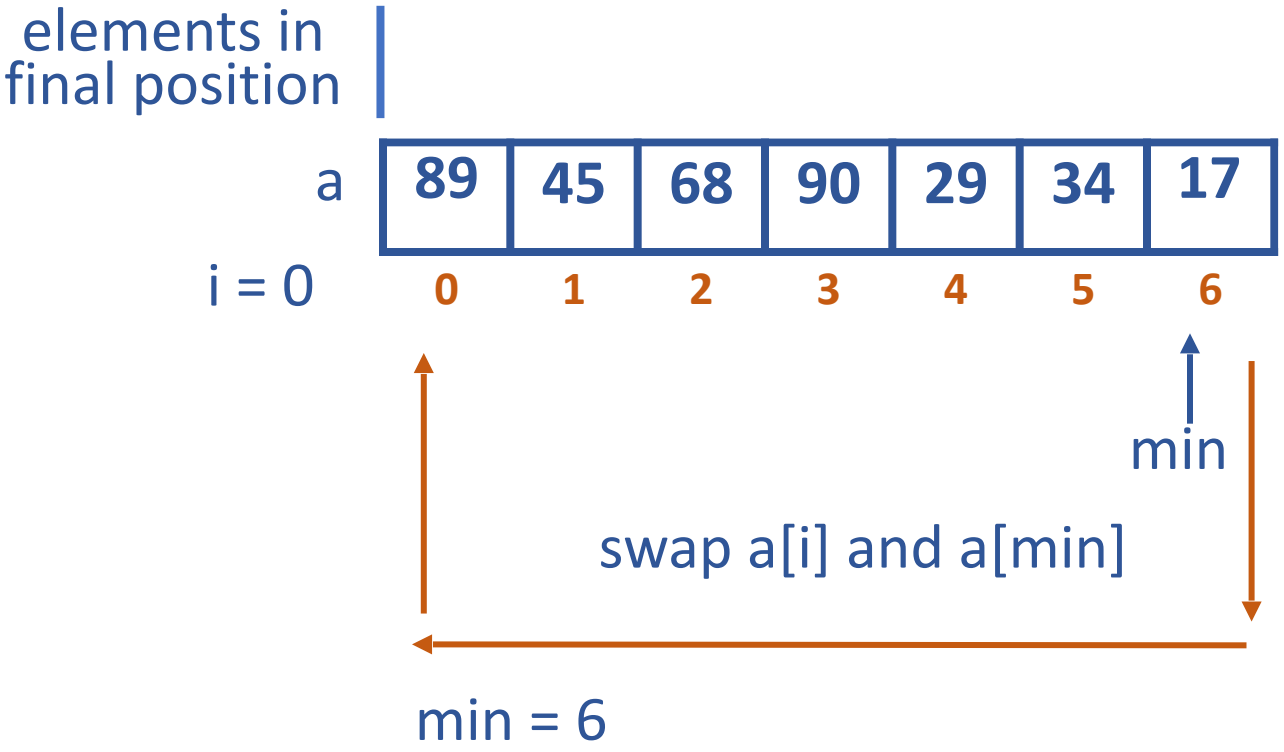
## Selection Sort



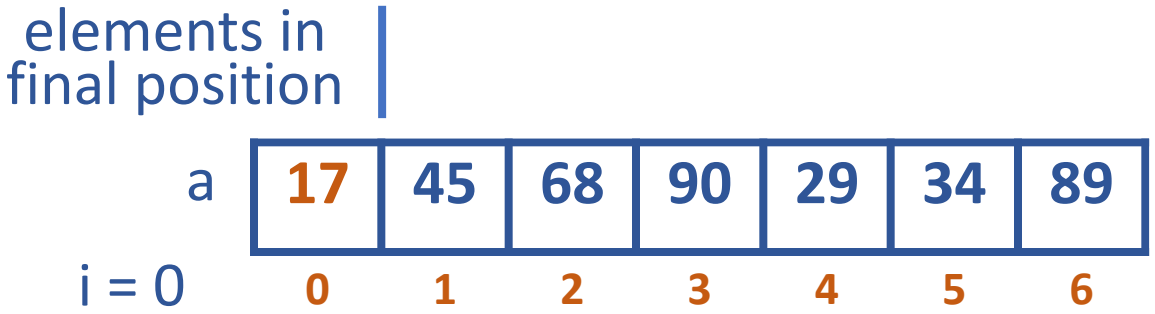
## Selection Sort



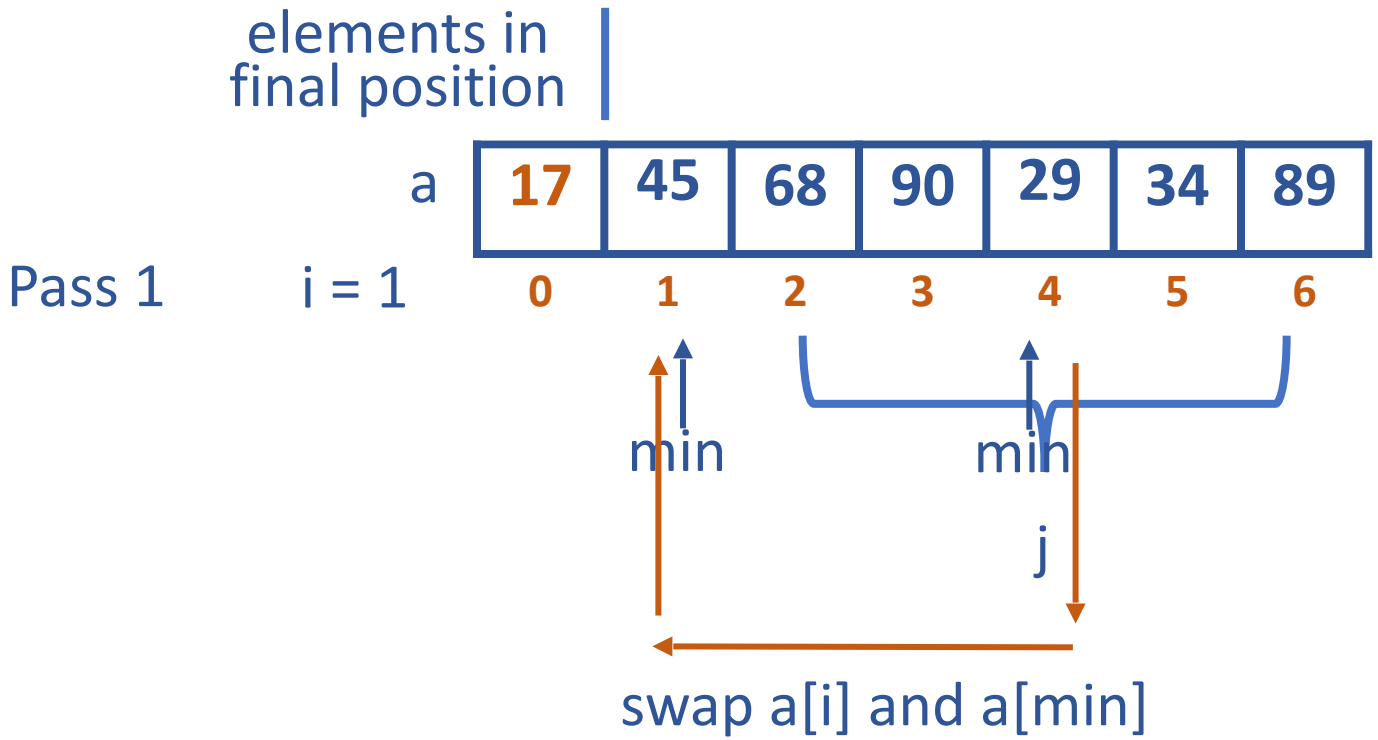
## Selection Sort



## Selection Sort

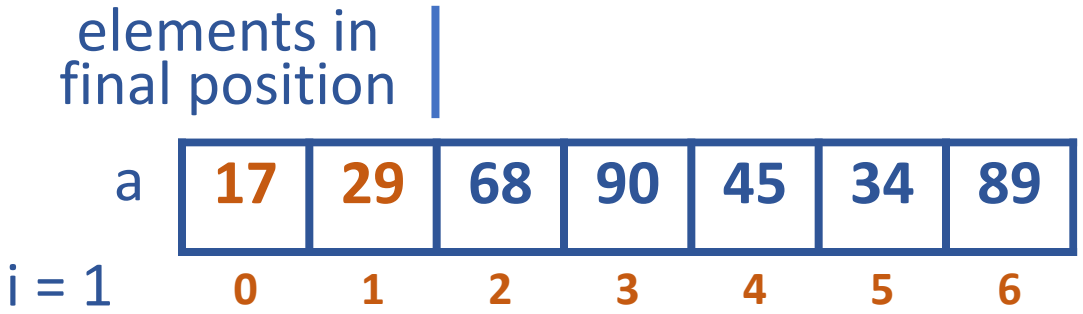


## Selection Sort

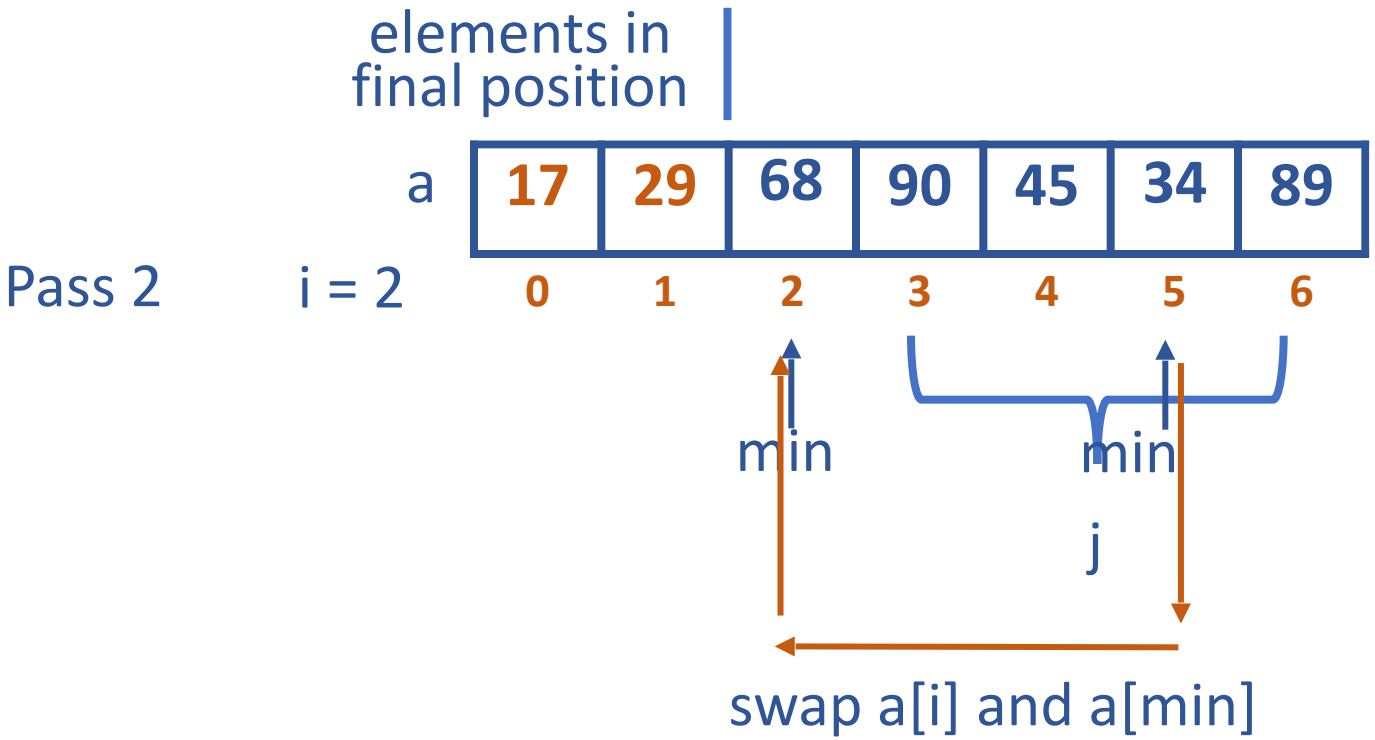



min = 4

## Selection Sort

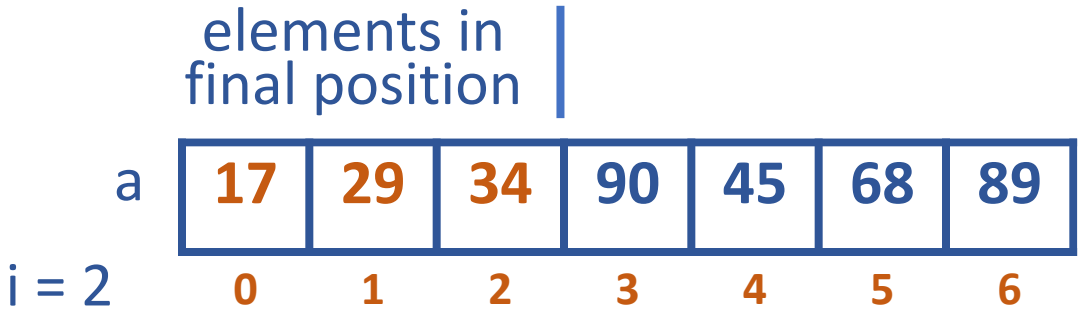


## Selection Sort

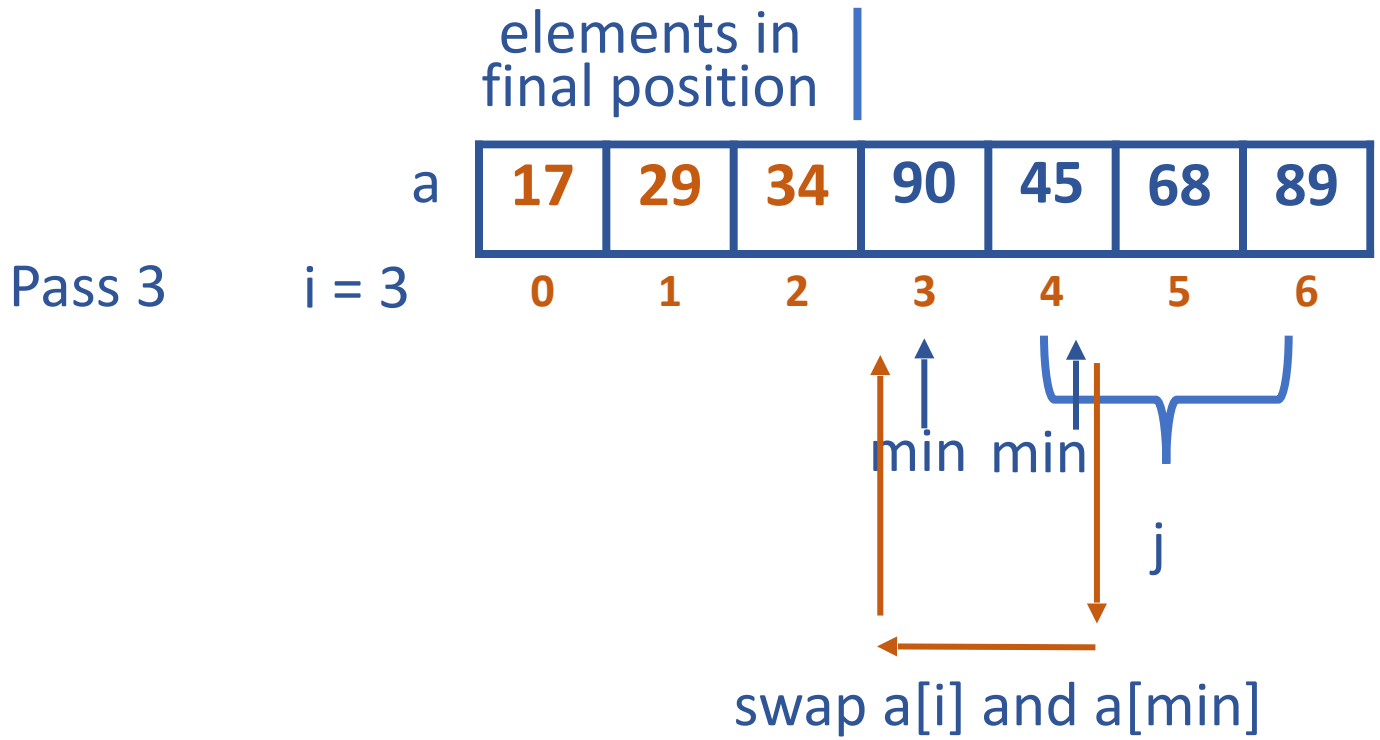



min = 5

## Selection Sort

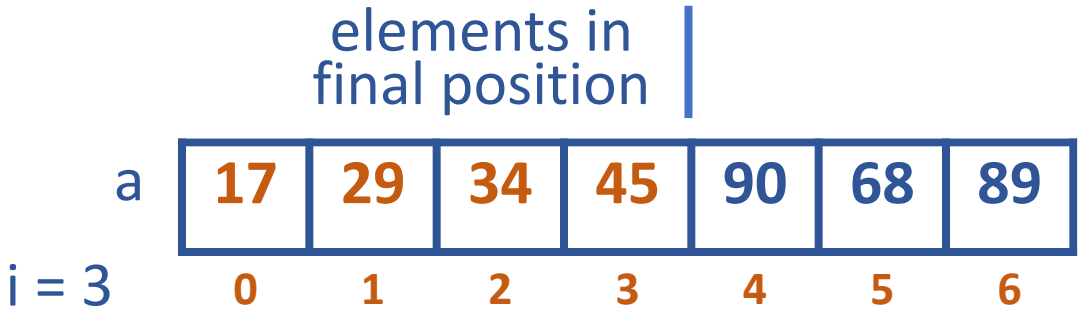


## Selection Sort

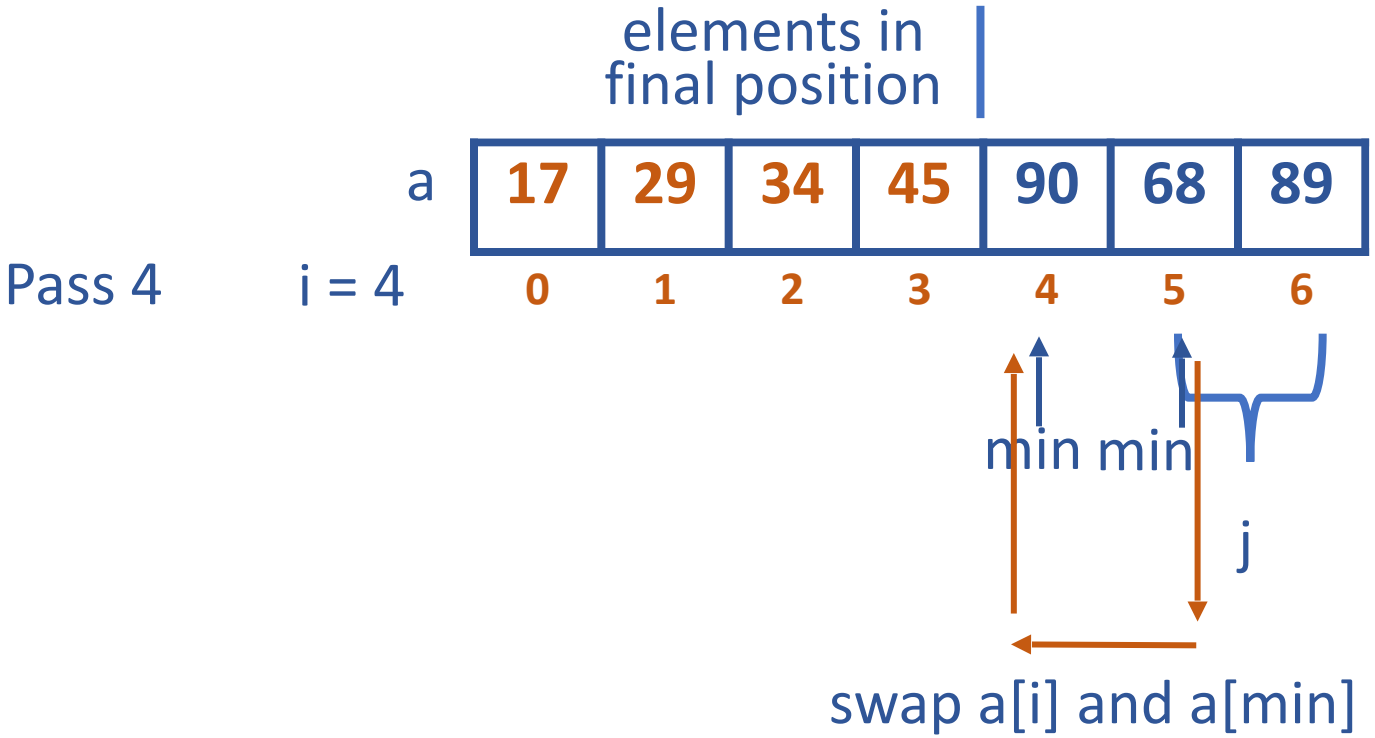



min = 4

## Selection Sort

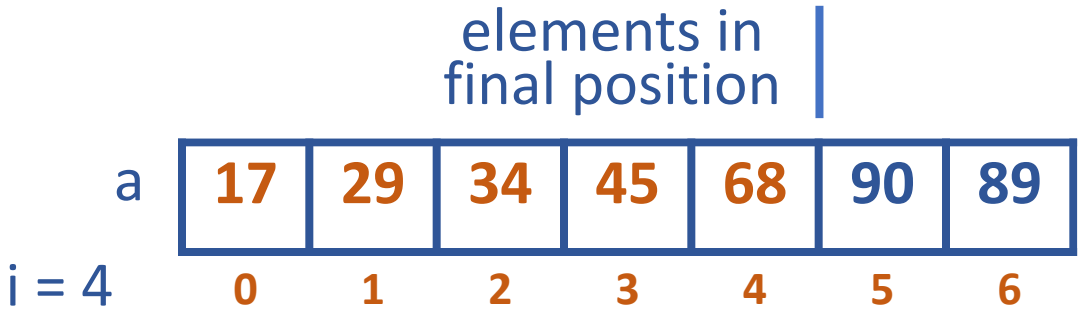


## Selection Sort

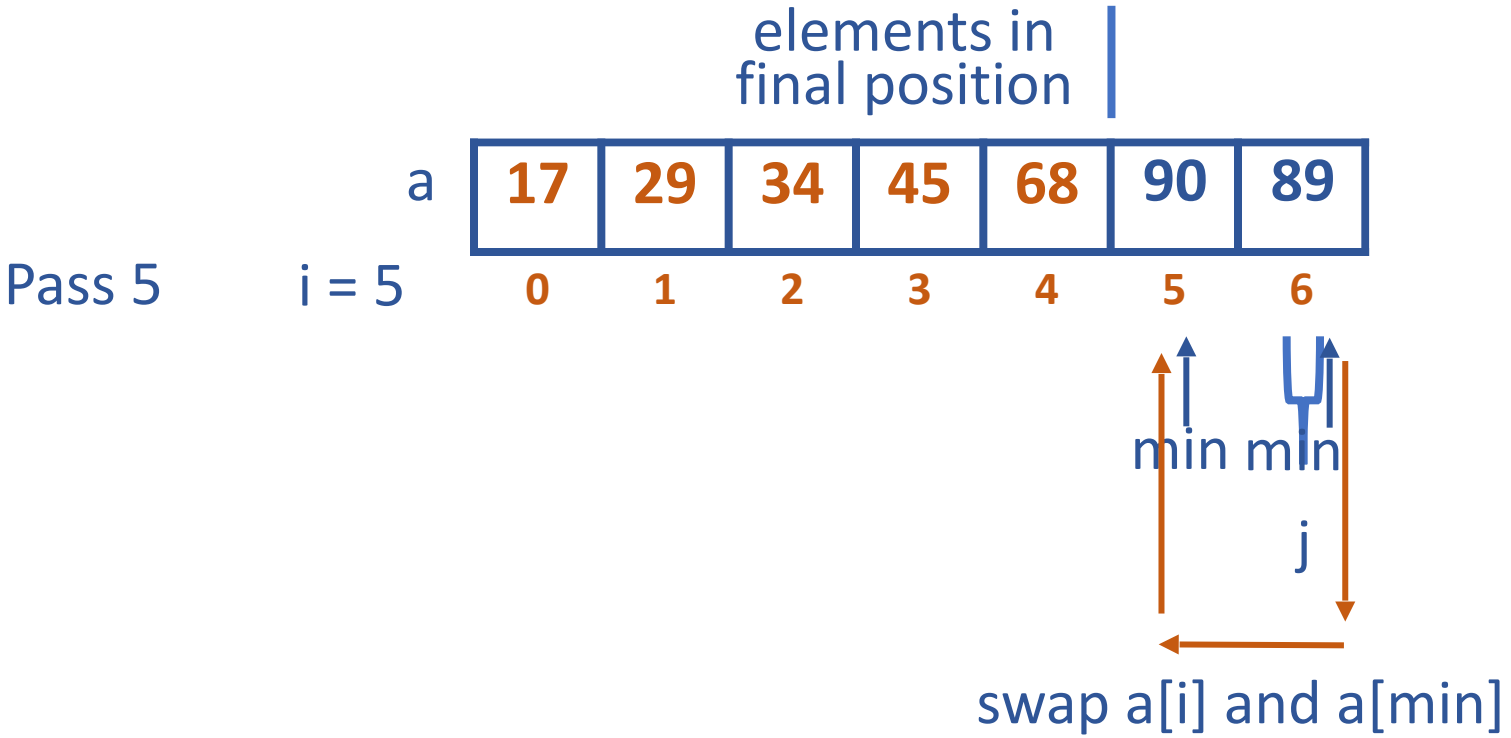



min = 5

## Selection Sort

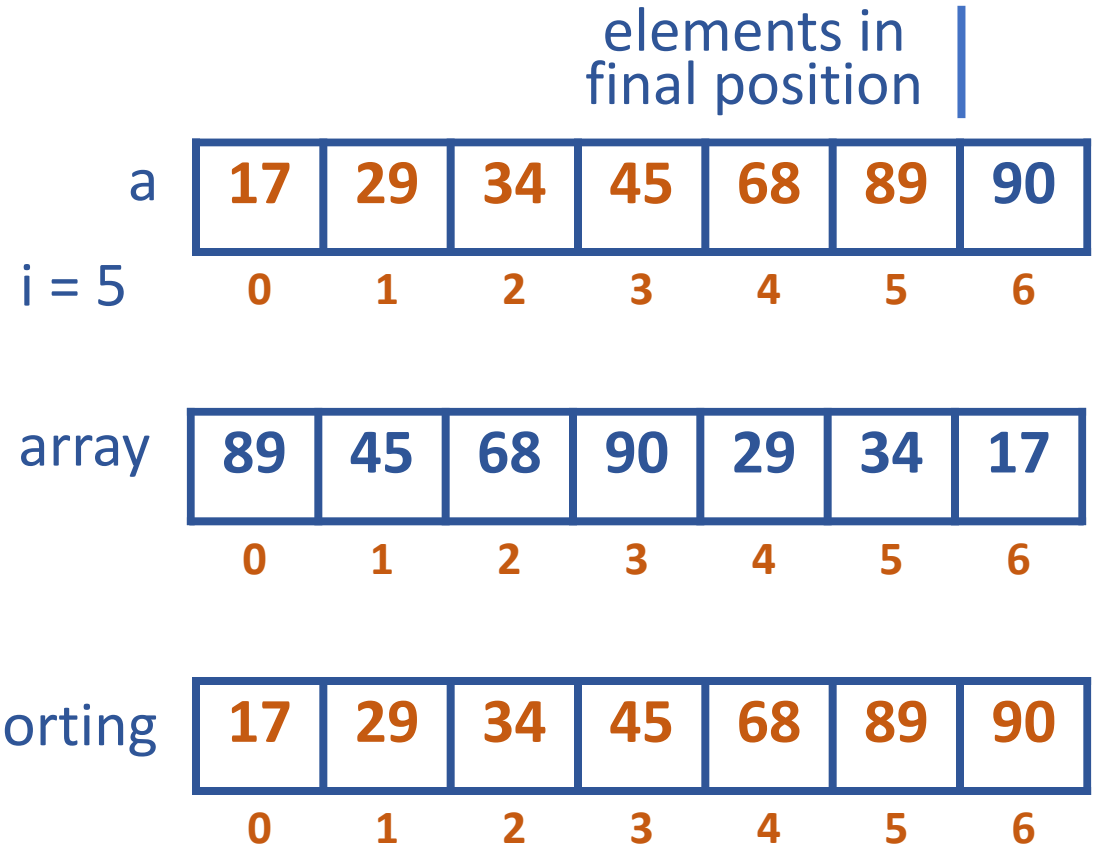


## Selection Sort




min = 6

## Selection Sort



## 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+l to n-1 do
                if A[j] <  A[min]  min  <- j
        swap  A[i] and A[min]
```

## Selection Sort

### Selection Sort Analysis

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

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


# THANK YOU

**Shylaja S S**

Department of Computer Science & Engineering

**shylaja.sharath@pes.edu**


# DESIGN AND ANALYSIS OF ALGORITHMS
# UE19CS251

**Shylaja S S**
Department of Computer Science
& Engineering

# DESIGN AND ANALYSIS OF ALGORITHMS

## Bubble Sort

Major Slides Content: Anany Levitin

**Shylaja S S**

Department of Computer Science & Engineering

## Bubble Sort

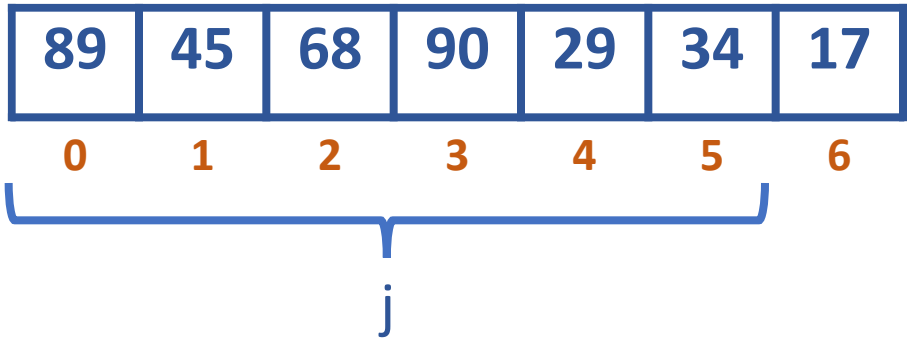
- Compare adjacent elements of the list and exchange them if they are out of order
- By doing it repeatedly, we end up bubbling the largest element to the last position on the list
- The next pass bubbles up the second largest element and so on and after n – 1 passes, the list is sorted
- Pass i ($0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

$A[0], A[1], A[2], \ldots, A[j] \overset{?}{\longleftrightarrow} A[j+1], \ldots, A[n-i-1] \mid A[n-i] \leq \ldots. \leq A[n-1]$ in their final positions

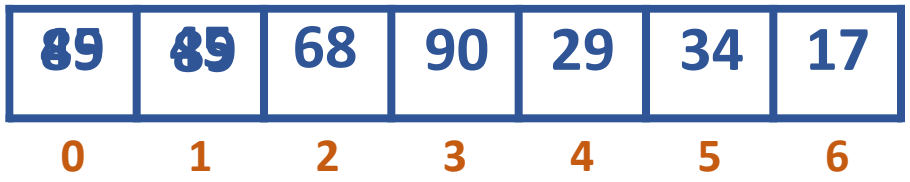
## Bubble Sort

elements in final position

i = 0




j = 0

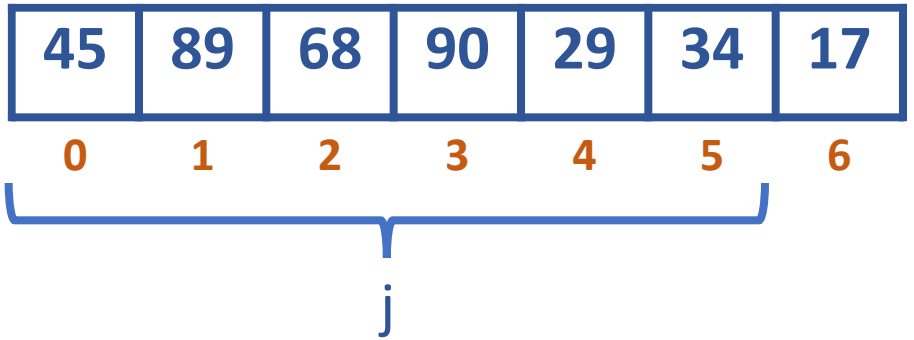



45 < 89 ☑ swap

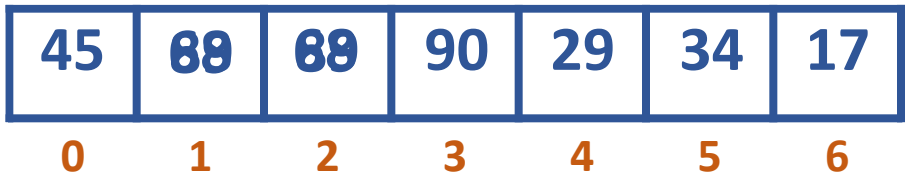
## Bubble Sort

elements in final position

i = 0




j = 1

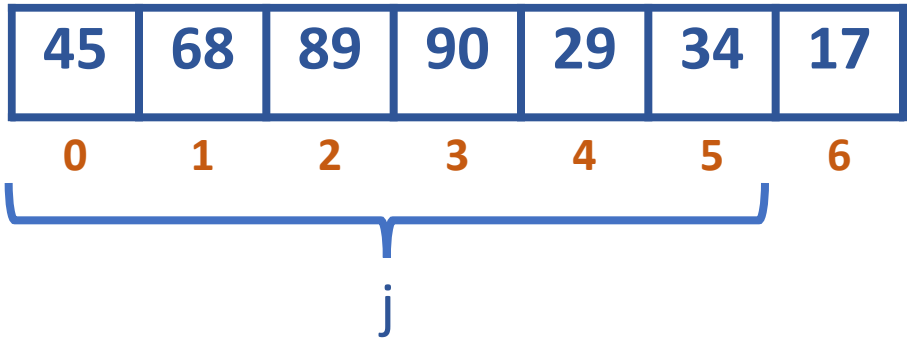



68 < 89 ☑ swap

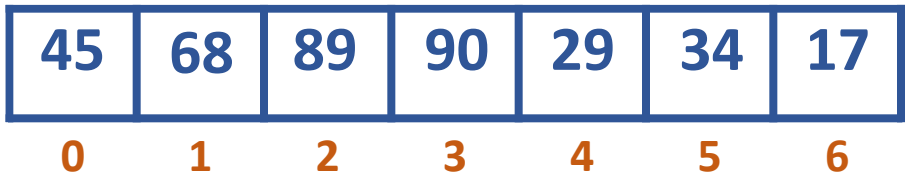
## Bubble Sort

elements in final position

i = 0




j = 2

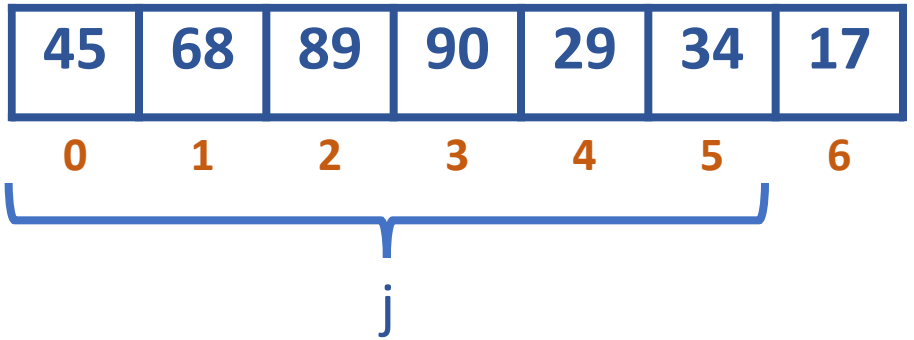



90 < 89 ☒ no swap

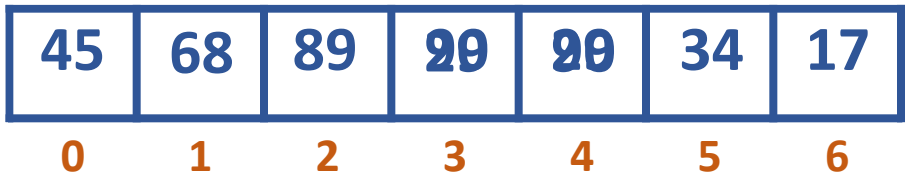
## Bubble Sort

elements in final position

i = 0




j = 3

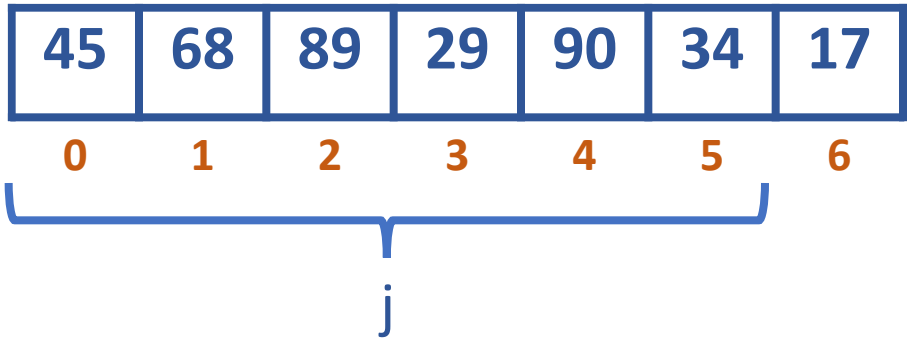



29 < 90 ☑ swap

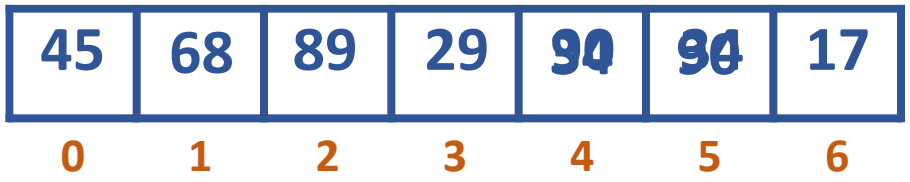
## Bubble Sort

elements in final position

i = 0




j = 4

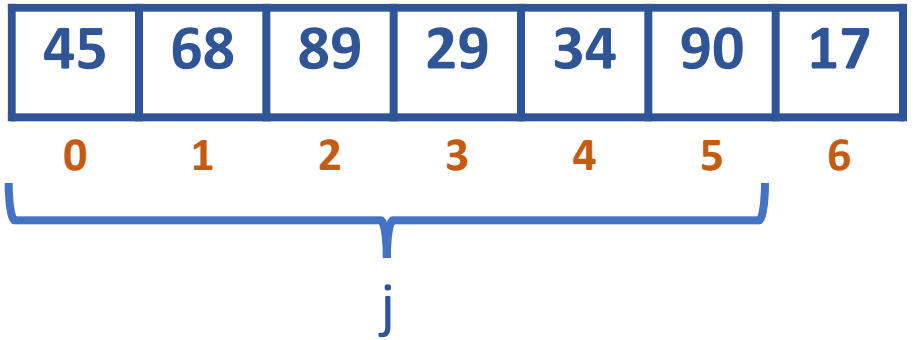



34 < 90 ☑ swap

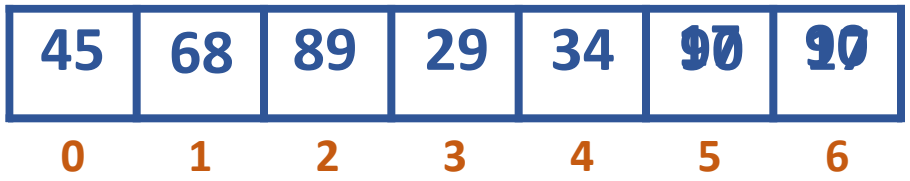
## Bubble Sort

elements in final position

i = 0




j = 5



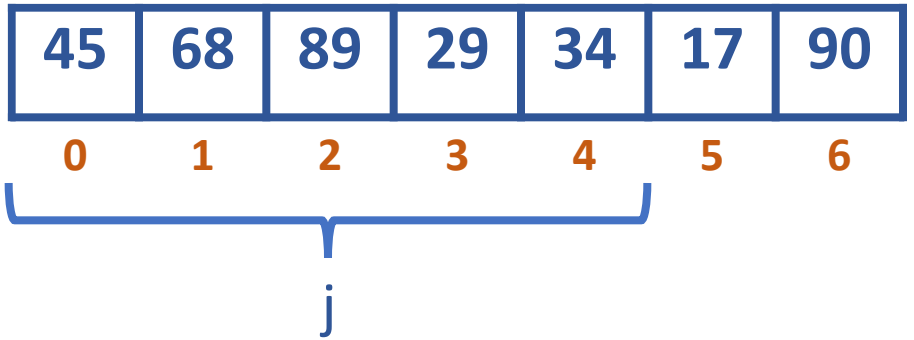

17 < 90 ☑ swap

## Bubble Sort

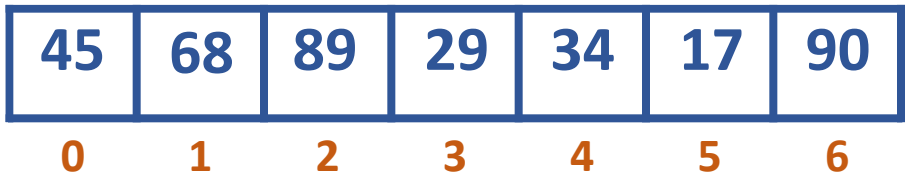
After first iteration

i = 1

elements in final position




j = 0

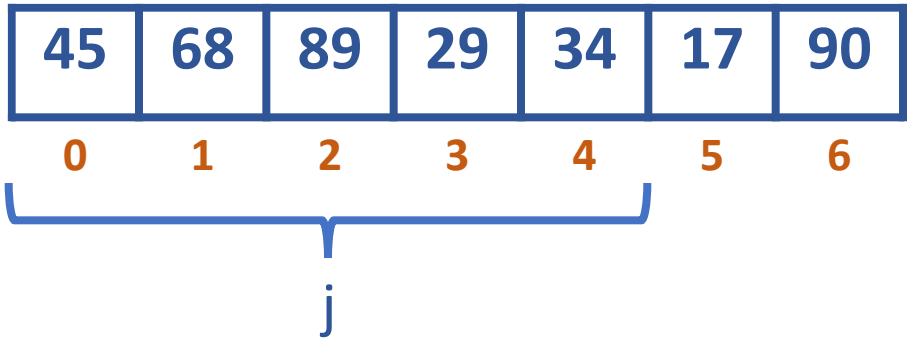



68 < 45 ☒ no swap

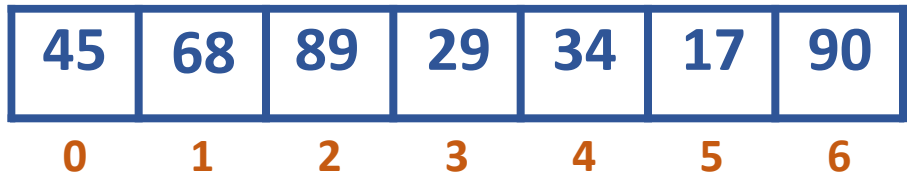
## Bubble Sort

elements in final position

i = 1




j = 1

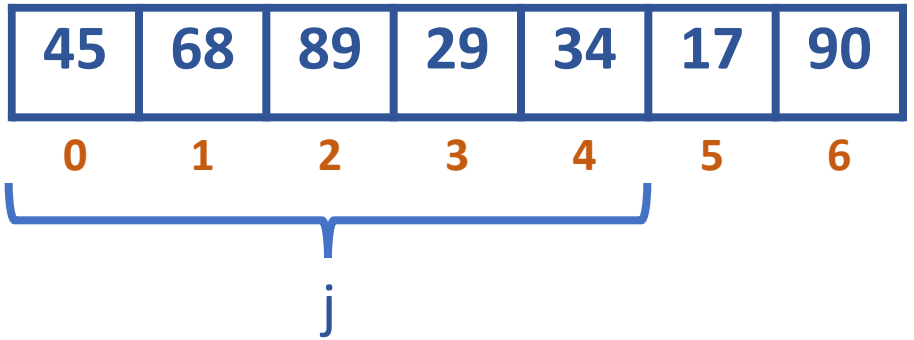



89 < 68 ☒ no swap

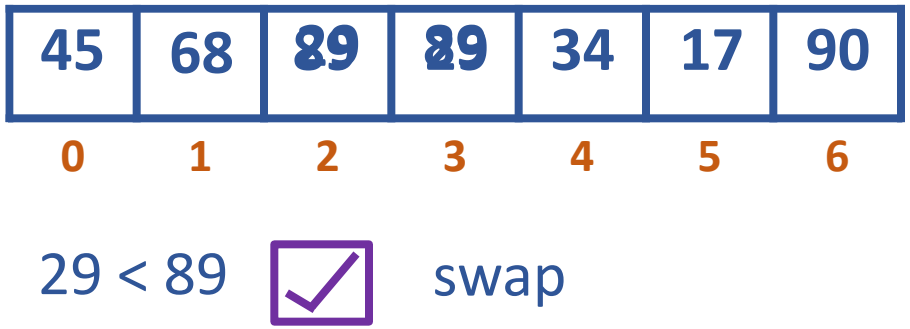
## Bubble Sort

elements in final position

i = 1




j = 2

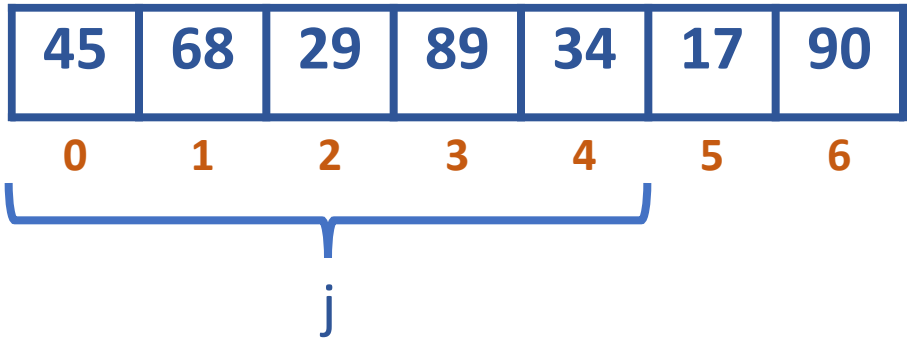



29 < 89 ☑ swap

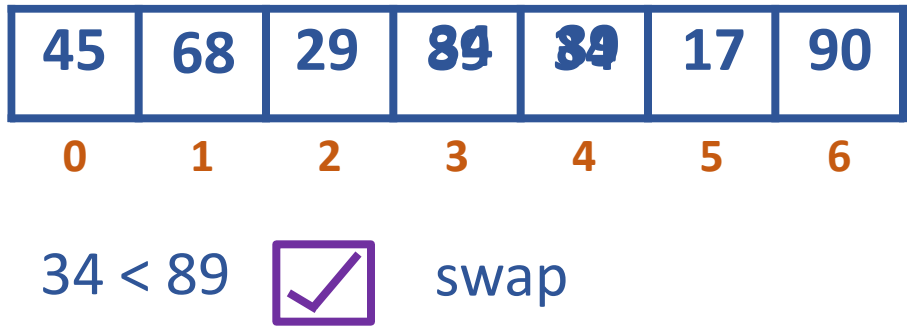
## Bubble Sort

elements in final position

i = 1




j = 3

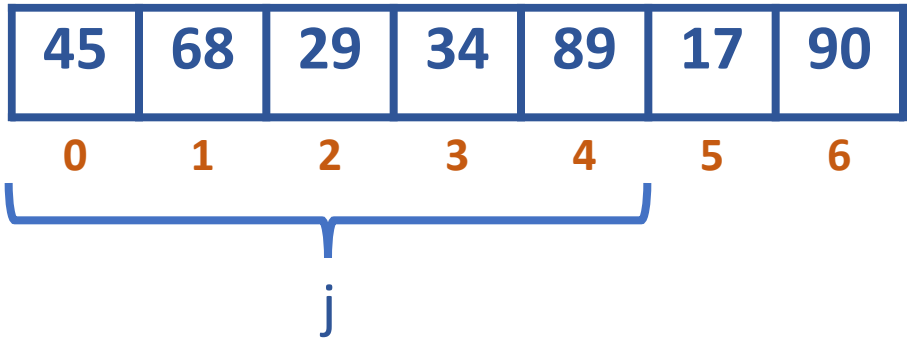



34 < 89 ☑ swap

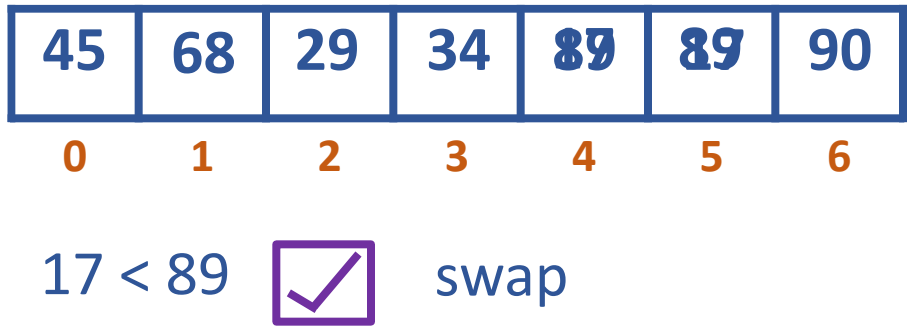
## Bubble Sort

elements in final position

i = 1




j = 4



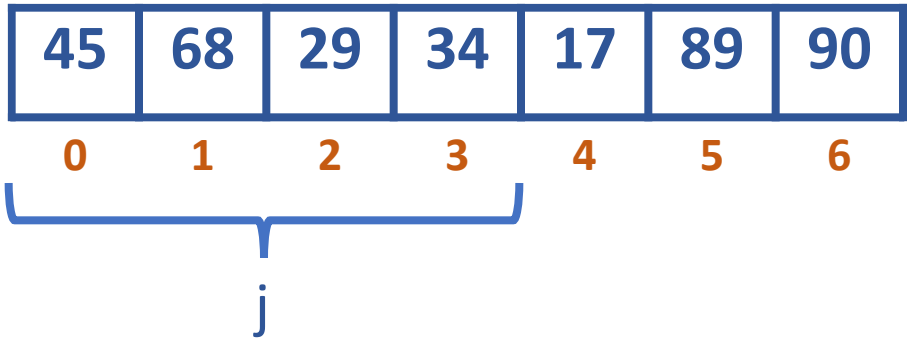

17 < 89 ☑ swap

## Bubble Sort

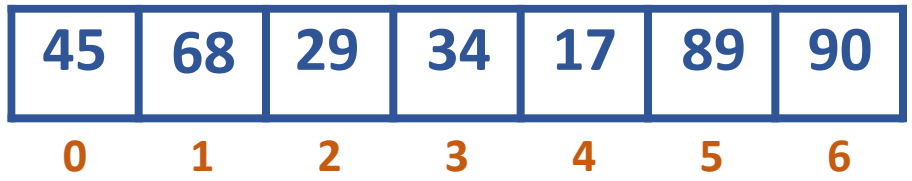
After second iteration

i = 2

elements in final position




j = 0

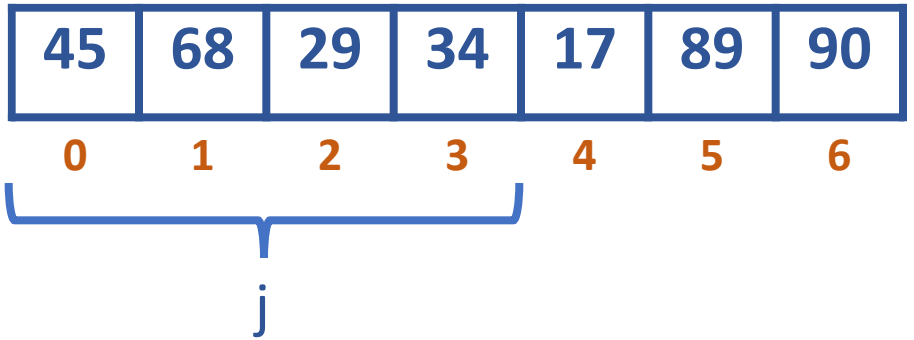



68 < 45 ☒ no swap

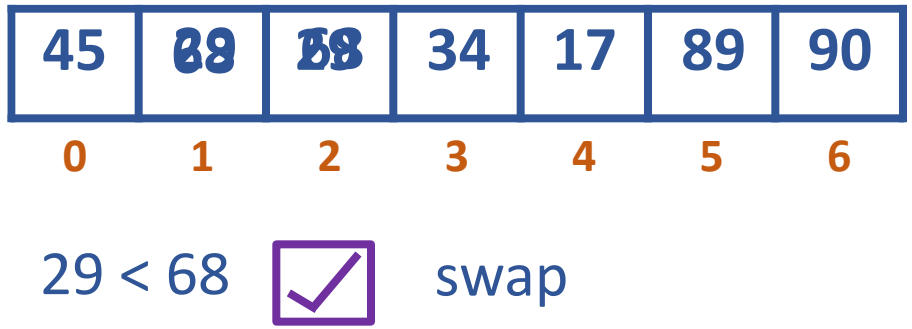
## Bubble Sort

elements in final position

i = 2




j = 1

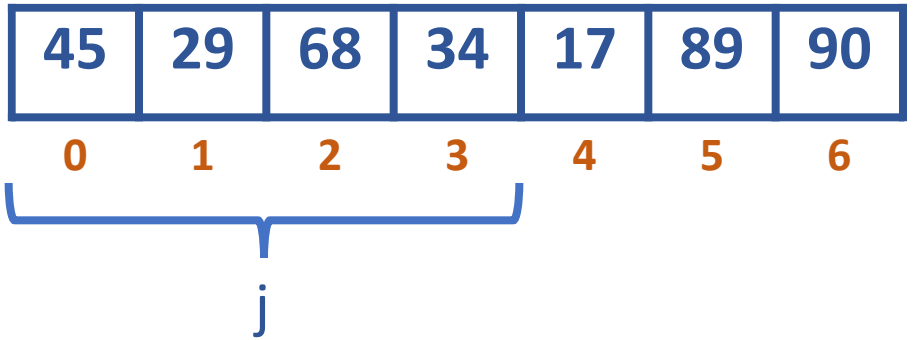



29 < 68 ☑ swap

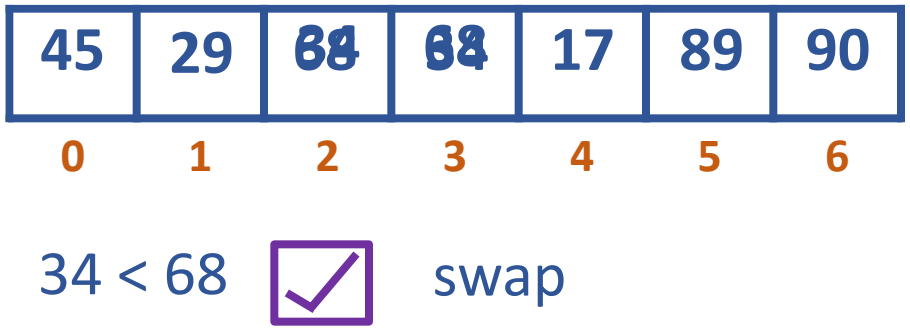
## Bubble Sort

elements in final position

i = 2




j = 2

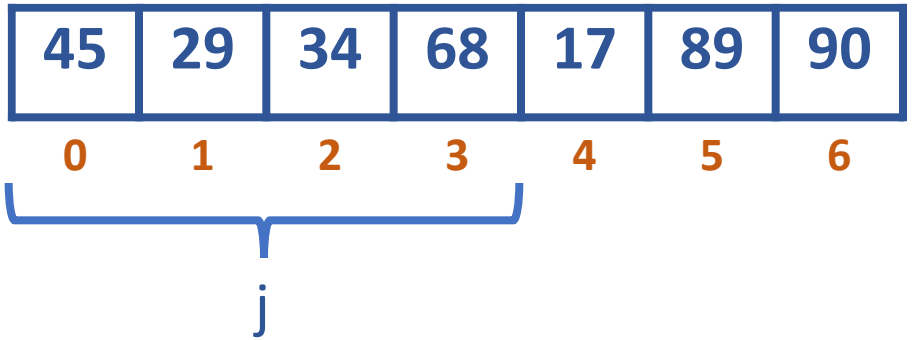



34 < 68 ☑ swap

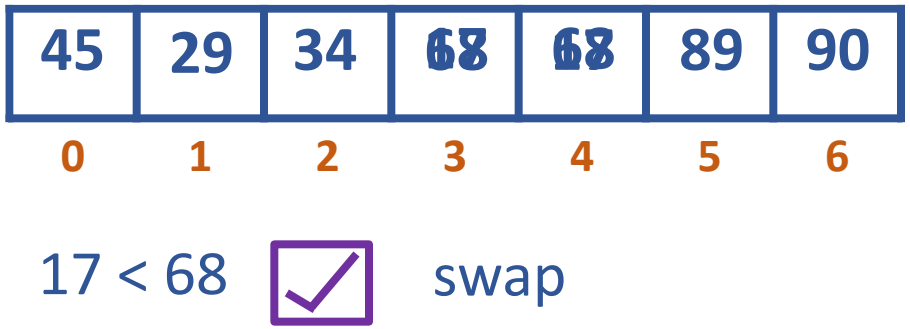
## Bubble Sort

elements in final position

i = 2




j = 3



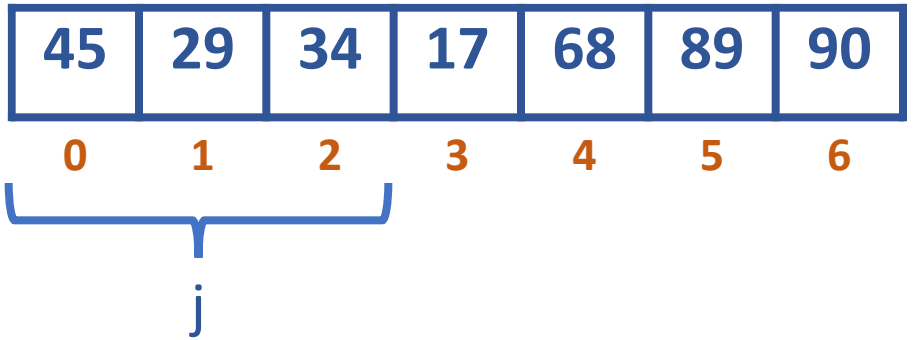

17 < 68 ☑ swap

## Bubble Sort

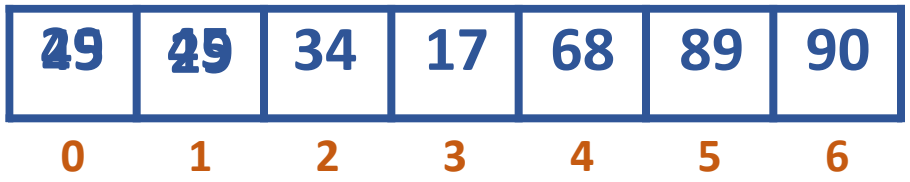
After third iteration

i = 3

elements in final position




j = 0

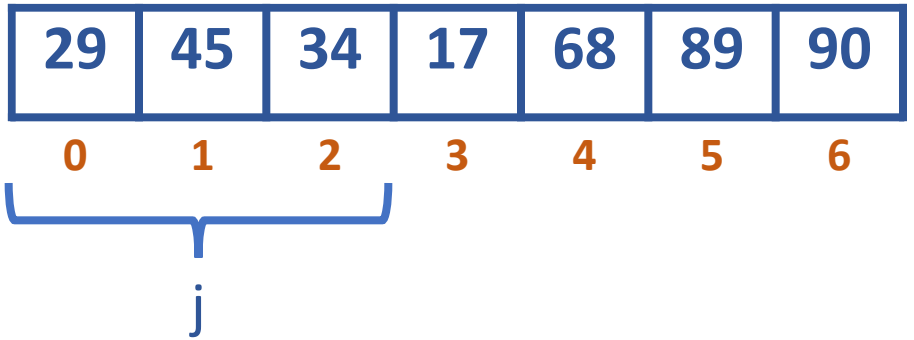



29 < 45 ☑ swap

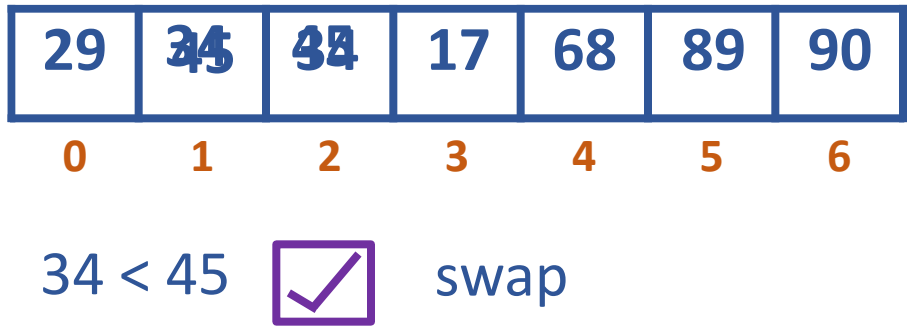
## Bubble Sort

elements in final position

i = 3

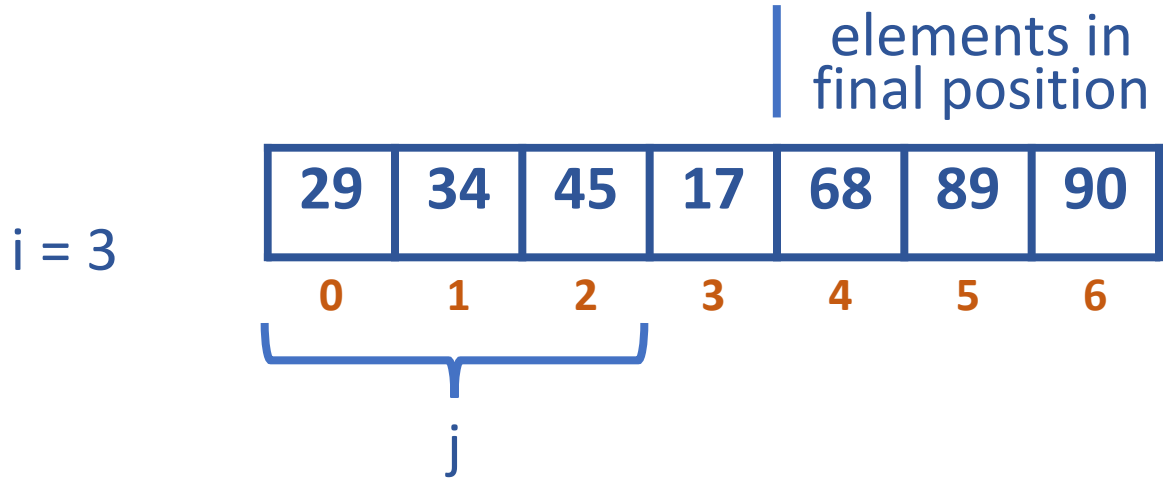


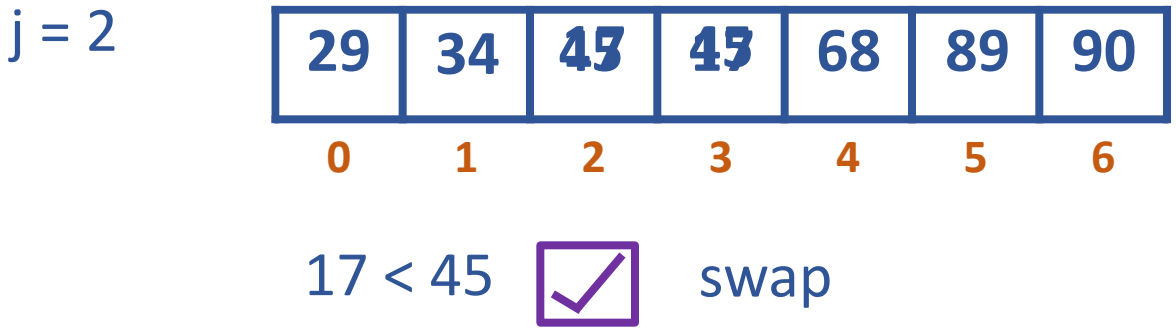

j = 1




34 < 45 ☑ swap

## Bubble Sort


elements in final position
i = 3
29
34
45
17
68
89
90
0
1
2
3
4
5
6
j

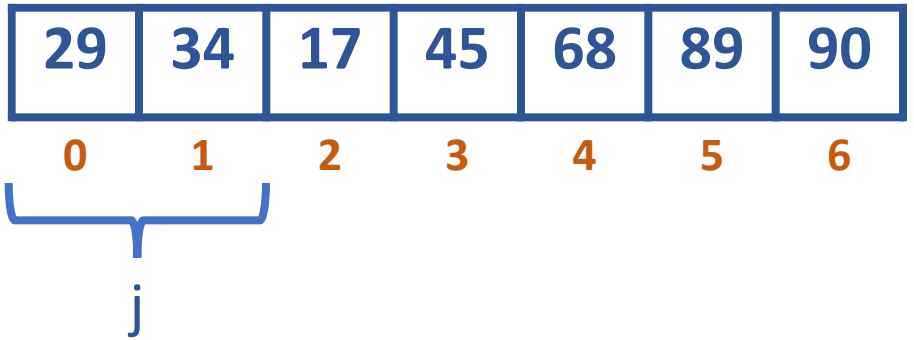

j = 2
29
34
45
17
68
89
90
0
1
2
3
4
5
6
17 < 45
swap

## Bubble Sort

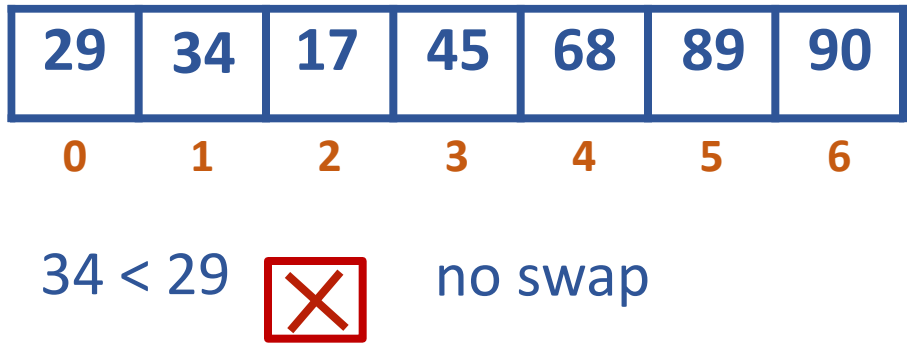
After fourth iteration

i = 4

elements in final position

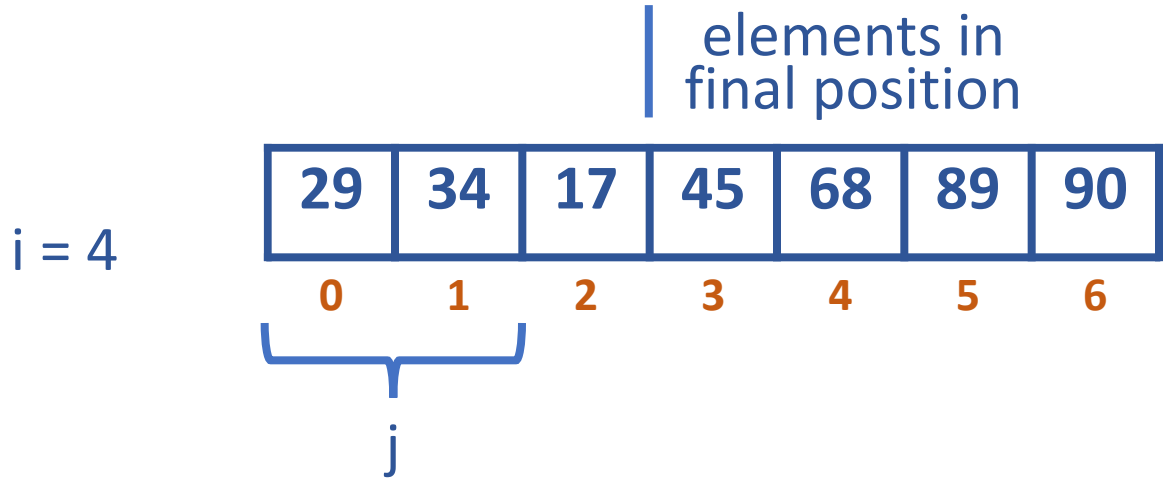


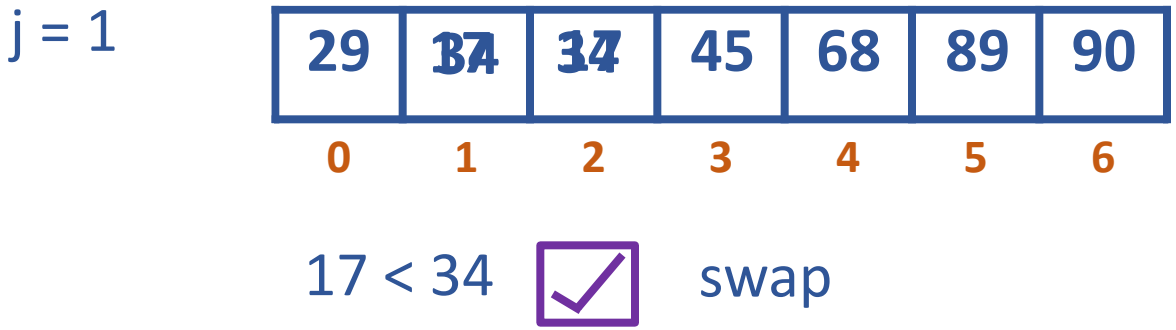

j = 0




34 < 29 ☒ no swap

## Bubble Sort


elements in
final position
i = 4
29
34
17
45
68
89
90
0
1
2
3
4
5
6
j

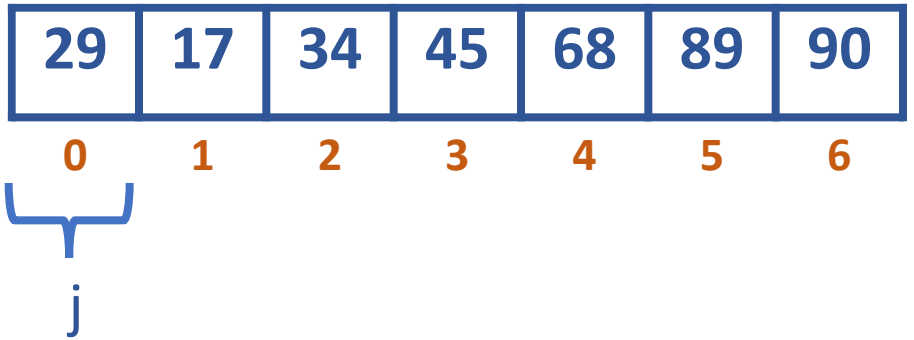

j = 1
29
34
17
45
68
89
90
0
1
2
3
4
5
6
17 < 34
swap

## Bubble Sort

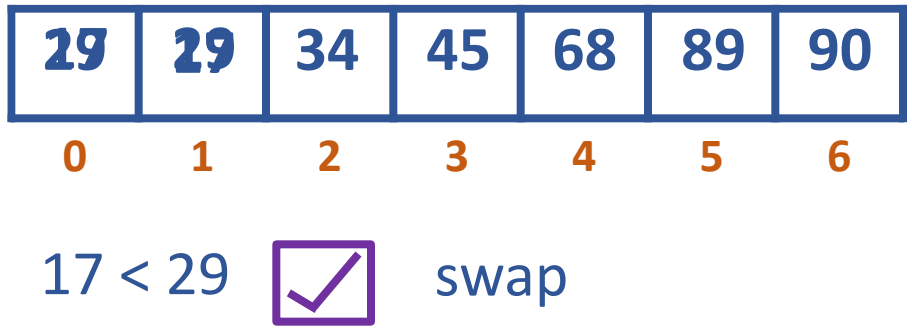
After fifth iteration

i = 5

elements in final position




j = 0

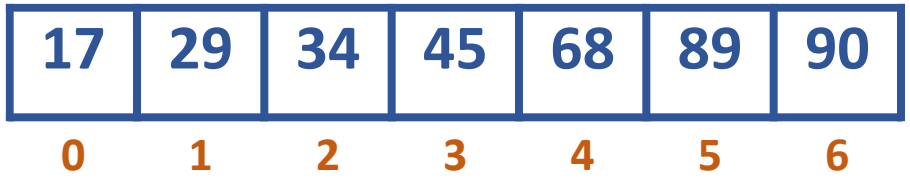



17 < 29 ☑ swap

## Bubble Sort

After sixth iteration

elements in final position



## 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  <-- 0 to n - 2 do
   for  j  <-- 0 to n - 2 - i  do
         if A[ j + 1 ]  <  A[ j ]  swap A[ j ] and A[ j + 1 ]
```

## Bubble Sort

### 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


# THANK YOU

**Shylaja S S**

Department of Computer Science & Engineering

**shylaja.sharath@pes.edu**


# DESIGN AND ANALYSIS OF ALGORITHMS
# UE19CS251

**Shylaja S S**
Department of Computer Science
& Engineering

# DESIGN AND ANALYSIS OF ALGORITHMS

## Sequential Search

Major Slides Content: Anany Levitin

**Shylaja S S**

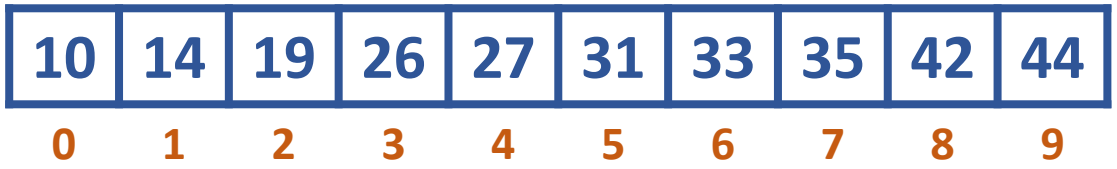
Department of Computer Science & Engineering

## 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

- Sequential / Linear Search




- For key = 33, 6 is returned
- For key = 50, -1 is returned

## 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
i<---0
while A[i] ≠K  do
   i<--- i + 1
if i  <  n return i
else return -1
```

## Sequential Search

### Sequential Search Analysis

- Sequential Search is a Θ(n) algorithm


# THANK YOU

**Shylaja S S**

Department of Computer Science & Engineering

**shylaja.sharath@pes.edu**


# DESIGN AND ANALYSIS OF ALGORITHMS
# UE19CS251

**Shylaja S S**
Department of Computer Science
& Engineering

# DESIGN AND ANALYSIS OF ALGORITHMS

## String Matching

Major Slides Content: Anany Levitin

**Shylaja S S**

Department of Computer Science & Engineering

## Brute – Force String Matching

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

## 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

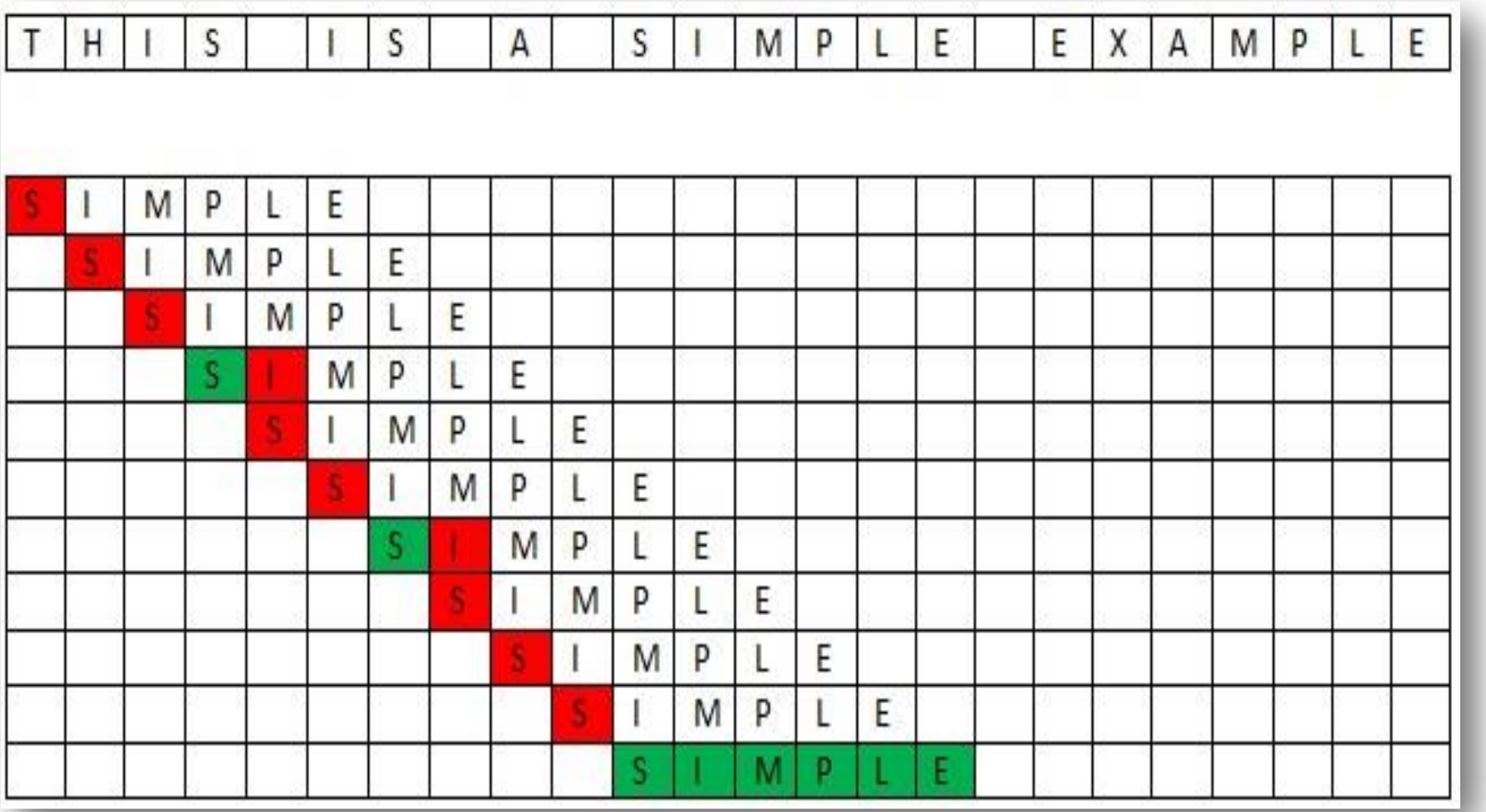
Step 3: While pattern is not found and the text is not yet exhausted, realign pattern one position to the right and repeat Step 2

## 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
   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 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


# THANK YOU

**Shylaja S S**

Department of Computer Science & Engineering

**shylaja.sharath@pes.edu**


# DESIGN AND ANALYSIS OF ALGORITHMS
# UE19CS251

**Shylaja S S**
Department of Computer Science
& Engineering

# DESIGN AND ANALYSIS OF ALGORITHMS

## Travelling Salesman Problem

Major Slides Content: Anany Levitin

**Shylaja S S**

Department of Computer Science & Engineering

## 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

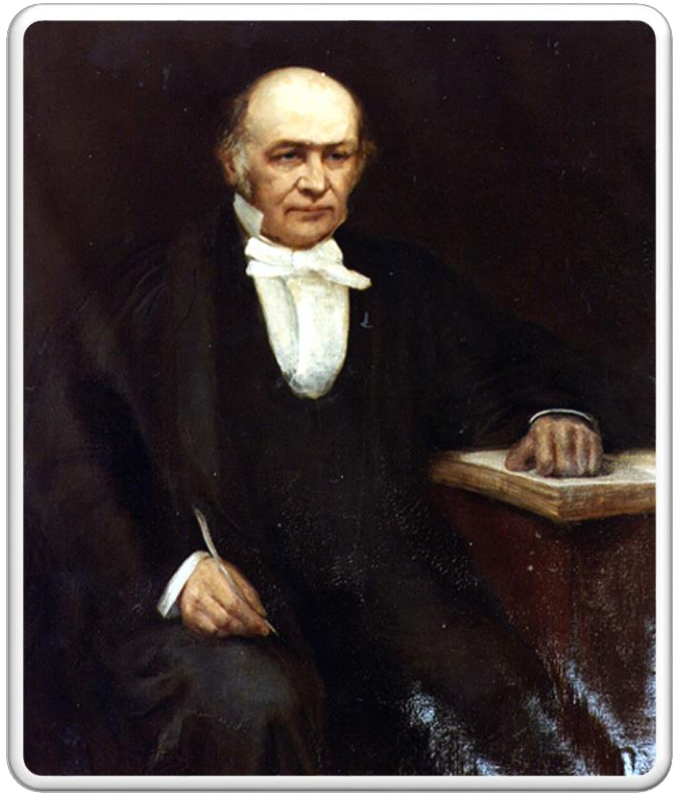
- Given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city and returns to the origin city?

Alternative way to state the problem:

- Find the shortest Hamiltonian Circuit in a weighted connected graph

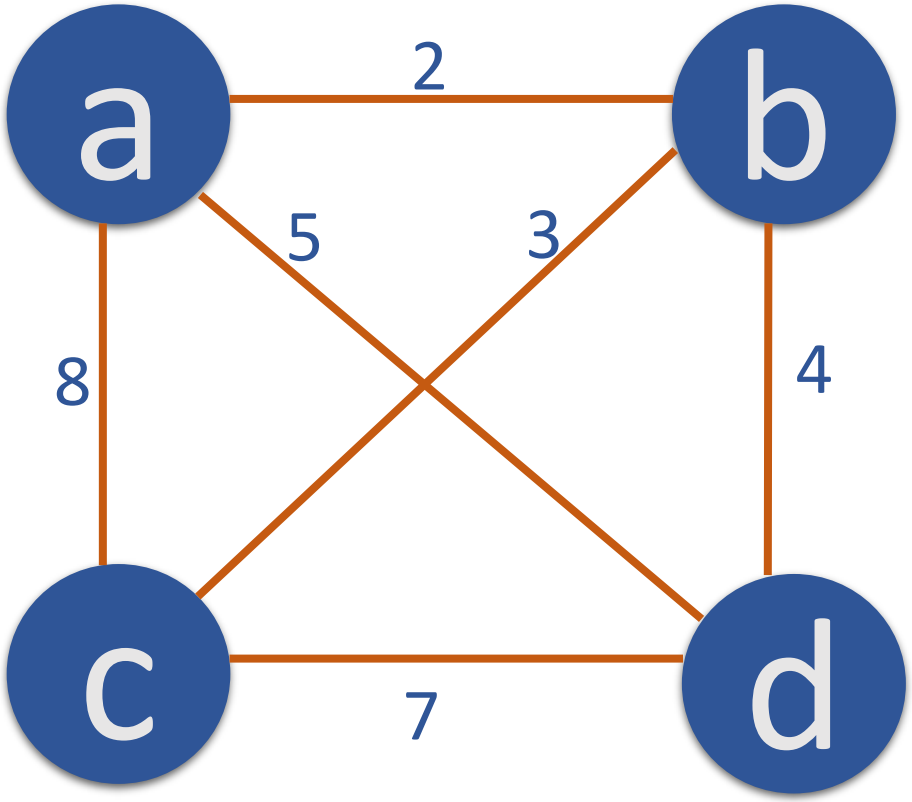
## TSP: History and Relevance

- 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



## Travelling Salesman Problem

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** |

## Travelling Salesman Problem

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

**Shylaja S S**

Department of Computer Science & Engineering

**shylaja.sharath@pes.edu**


# DESIGN AND ANALYSIS OF ALGORITHMS
# UE19CS251

**Shylaja S S**
Department of Computer Science
& Engineering

# DESIGN AND ANALYSIS OF ALGORITHMS

## Knapsack Problem

Major Slides Content: Anany Levitin

**Shylaja S S**

Department of Computer Science & Engineering

## Knapsack Problem

Given n items:

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

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

## Knapsack Problem

Example

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

**Shylaja S S**

Department of Computer Science & Engineering

**shylaja.sharath@pes.edu**


# DESIGN AND ANALYSIS OF ALGORITHMS
# UE19CS251

**Shylaja S S**
Department of Computer Science
& Engineering

# DESIGN AND ANALYSIS OF ALGORITHMS

## Assignment Problem

Major Slides Content: Anany Levitin

**Shylaja S S**

Department of Computer Science & Engineering

## The Assignment Problem

- 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

## 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 |

## The Assignment Problem

Algorithmic Plan

1. Generate all legitimate assignments
2. Compute their costs
3. Select the cheapest one

## The Assignment Problem

### The Assignment Problem by Exhaustive Search

| Assignment | Cost |
|---|---|
| 1, 2, 3, 4 | 9 + 4 + 1 + 4 = 18 |
| 1, 3, 4, 2 | 9 + 8 + 9 + 7 = 33 |
| 1, 4, 3, 2 | 9 + 6 + 1 + 7 = 23 |
| 1, 4, 2, 3 | 9 + 6 + 3 + 8 = 26 |
| 1, 3, 2, 4 | 9 + 8 + 3 + 4 = 24 |
| 1, 2, 4, 3 | 9 + 4 + 9 + 8 = 30 |

etc.,

## The Assignment Problem

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


# THANK YOU

**Shylaja S S**

Department of Computer Science & Engineering

**shylaja.sharath@pes.edu**