

# **Analysis of Algorithms**

## **Lecture-01**

# Introduction

- What is an Algorithm
  - Definition, characteristics, Writing an algorithm
  - Algorithm and Basic Concept
- Algorithm Complexity
  - Analysis of the algorithms
  - Computing run time of algorithms
  - Bounds
- Important Problem Type
  - Searching and Sorting- linear and binary search
  - String processing
- Fundamental Data Structure
  - Linear data Structure
  - Non Linear Data structure

# Algorithm: Definition

An algorithm is a **set of steps of operations to solve a problem performing calculation**, data processing, and automated reasoning tasks. An algorithm is an efficient method that can be expressed within finite amount of time and space.

- It can have **zero or more inputs**.
- It must have **at least one output**.
- It should be **efficient** both in terms of memory and time.
- It should be **finite**.
- Every statement should be **unambiguous**.

Algorithm is **independent** from any programming language.

Note: A **program** can run infinitely, for example a server program it runs 24X7X365. But an **algorithm** must be finite.

# Characteristics of an Algorithm

Not all procedures can be called an algorithm. An algorithm should have the following characteristics –

- **Unambiguous** – Algorithm should be clear and unambiguous. Each of its steps (or phases), and their inputs/outputs should be clear and must lead to only one meaning.
- **Input** – An algorithm should have 0 or more well-defined inputs.
- **Output** – An algorithm should have 1 or more well-defined outputs, and should match the desired output.
- **Finiteness** – Algorithms must terminate after a finite number of steps.
- **Feasibility** – Should be feasible with the available resources.
- **Independent** – An algorithm should have step-by-step directions, which should be independent of any programming code.
- Note: It must have a termination criteria

# Design and Analysis of Algorithm: Objectives

This subject focus on following points

- To **Construct** the algorithms for the problems to be solved or already solved by other methods, the problems may be from diverse fields and not restricted to Computer Science only.
- To **prove** that a proposed algorithm solve the problem correctly.
- To **analyze the time and space requirements** of an algorithm in standard, asymptotical notations, that is independent of particular machine.
- To prove that the proposed algorithm solve the problem **faster** than other solutions

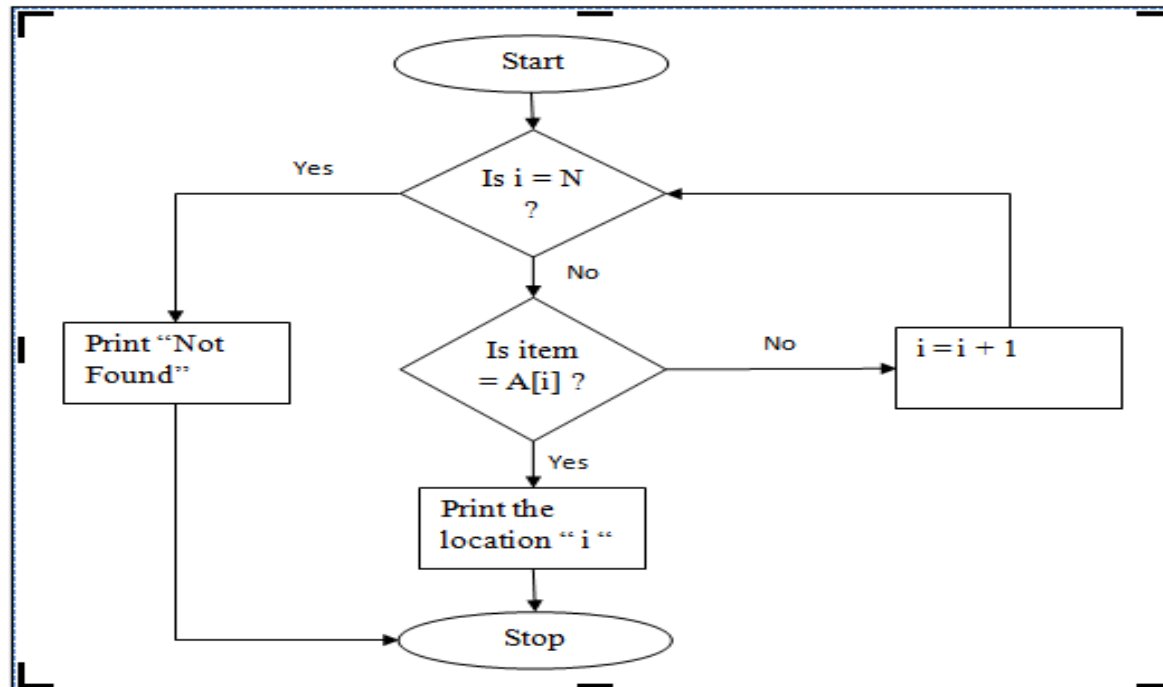
# Writing an Algorithm

An algorithm can be written in following ways:

- **Natural Language Representation:** Use of **Natural language in writing and algorithm** can be ambiguous and therefore algorithm may lack the characteristic of being definite.
- **Flowcharts:** Graphical representation of algorithmic steps, flowcharts are not suitable to write the solutions for complex problems.
- **Pseudocode:** has an advantage of being easily converted into any programming language. A Table of pseudocode conventions are given in textbook. Mostly used.

# Flowchart

- Flowcharts pictorially depict a process.
- They are easy to understand and are commonly used in case of simple problems.



# Pseudo Code

- The pseudo code has an advantage of being easily converted into any programming language.
- This way of writing algorithm is most acceptable and most widely used.
- In order to be able to write a pseudo code, one must be familiar with the conventions of writing it.

---

Algorithm LinearSearch(A, n, item)

```
{  
    for i := 1 to n step 1 do  
    {  
        if(A[i] = item) then  
        {  
            write i;  
            exit;  
        }  
    }  
    write "Not Found"  
}
```





- Brute force
  - Try all possible combinations
- Divide and conquer
  - breaking down a problem into two or more sub-problems of the same (or related) type, until these become simple enough to be solved directly.
- Decrease and conquer
  - Change an instance into one smaller instance of the problem.
  - Solve the smaller instance.
  - Convert the solution of the smaller instance into a solution for the larger instance.

# Algorithm design strategies

- Transform and conquer
  - a simpler instance of the same problem, or
  - a different representation of the same problem, or
  - an instance of a different problem
- Greedy approach
  - The problem could be solved in iterations
- Dynamic programming
  - An instance is solved using the solutions for smaller instances.
  - The solution for a smaller instance might be needed multiple times.
  - The solutions to smaller instances are stored in a table, so that each smaller instance is solved only once.
  - Additional space is used to save time.
- Backtracking and branch-and-bound

# Algorithm: Design Considerations

- The five most essential things are to be considered while writing an algorithm are as follows:
  - Time taken
  - Memory usage
  - Input
  - Process
  - Output.

# Algorithm Complexity

Suppose **X** is an algorithm and **n** is the size of input data, the time and space used by the algorithm X are the two main factors, which decide the efficiency of X.

- **Time Factor** – Time is measured by counting the number of key operations such as comparisons in the sorting algorithm.
- **Space Factor** – Space is measured by counting the maximum memory space required by the algorithm.

The complexity of an algorithm **f(n)** gives the running time and/or the storage space required by the algorithm in terms of **n** as the size of input data.

# Analysis of algorithms

## ■ Issues:

- Correctness
- Space efficiency
- Time efficiency
- Optimality

## ■ Approaches:

- Theoretical analysis
- Empirical analysis

\*Apriori analysis vs. posterior analysis

# Types of the Algorithm Analysis



Efficiency of an algorithm can be analyzed at two different stages, before implementation and after implementation. They are the following

■ **A *Priori* Analysis** – This is a theoretical analysis of an algorithm. Efficiency of an algorithm is measured by assuming that all other factors, for example, processor speed, are constant and have no effect on the implementation.

■ **A *Posterior* Analysis** – This is an empirical analysis of an algorithm. The selected algorithm is implemented using programming language. This is then executed on target computer machine. In this analysis, actual statistics like running time and space required, are collected.

We shall learn about *a priori* algorithm analysis. Algorithm analysis deals with the execution or running time of various operations involved. The running time of an operation can be defined as the number of computer instructions executed per operation.

# Why to perform analysis of the Algorithms

The analysis of an algorithm is done because:

- The analysis of algorithm can be **more reliable than experiments**.
- Experiments can guarantee the behavior of algorithms on **certain test cases**, while theoretical analysis and establishment of run time bounds can guarantee the behavior of algorithm on whole domain.
- Analysis helps us to chose from many possible solutions.
- Performance of the program can be predicted before it is implemented.
- By analysis we can have idea of slow and faster parts of the algorithm, and accordingly we can plan our implementation strategies.

# Worst-Case/ Best-Case/ Average-Case Analysis

- **Worst-Case Analysis** –The maximum amount of time that an algorithm require to solve a problem of size  $n$ .
- **Best-Case Analysis** –The minimum amount of time that an algorithm require to solve a problem of size  $n$ . The best case behavior of an algorithm is NOT so useful.
- **Average-Case Analysis** –The average amount of time that an algorithm require to solve a problem of size  $n$ .
  - Worst-case analysis is more common than average-case analysis.



# Asymptotic Analysis

- Asymptotic analysis of an algorithm refers to defining the mathematical boundation/framing of its run-time performance. Using asymptotic analysis, we can very well conclude the best case, average case, and worst case scenario of an algorithm.
- Asymptotic analysis is input bound i.e., if there's no input to the algorithm, it is concluded to work in a constant time. Other than the "input" all other factors are considered constant.
- Asymptotic analysis refers to computing the running time of any operation in mathematical units of computation. For example, the running time of one operation is computed as  $f(n)$  and may be for another operation it is computed as  $g(n^2)$ . This means the first operation running time will increase linearly with the increase in  $n$  and the running time of the second operation will increase exponentially when  $n$  increases. Similarly, the running time of both operations will be nearly the same if  $n$  is significantly small.

# Asymptotic Analysis

## Common order-of-growth classifications

---

**Definition.** If  $f(N) \sim c g(N)$  for some constant  $c > 0$ , then the **order of growth** of  $f(N)$  is  $g(N)$ .

- Ignores leading coefficient.
- Ignores lower-order terms.

**Ex.** The order of growth of the **running time** of this code is  $\Theta(N^3)$ .

```
int count = 0;
for (int i = 0; i < N; i++)
    for (int j = i+1; j < N; j++)
        for (int k = j+1; k < N; k++)
            if (a[i] + a[j] + a[k] == 0)
                count++;
```

**Typical usage.** With running times.

# Asymptotic Notations

Following are the commonly used asymptotic notations to calculate the running time complexity of an algorithm.

- O Notation
- $\Omega$  Notation
- $\theta$  Notation

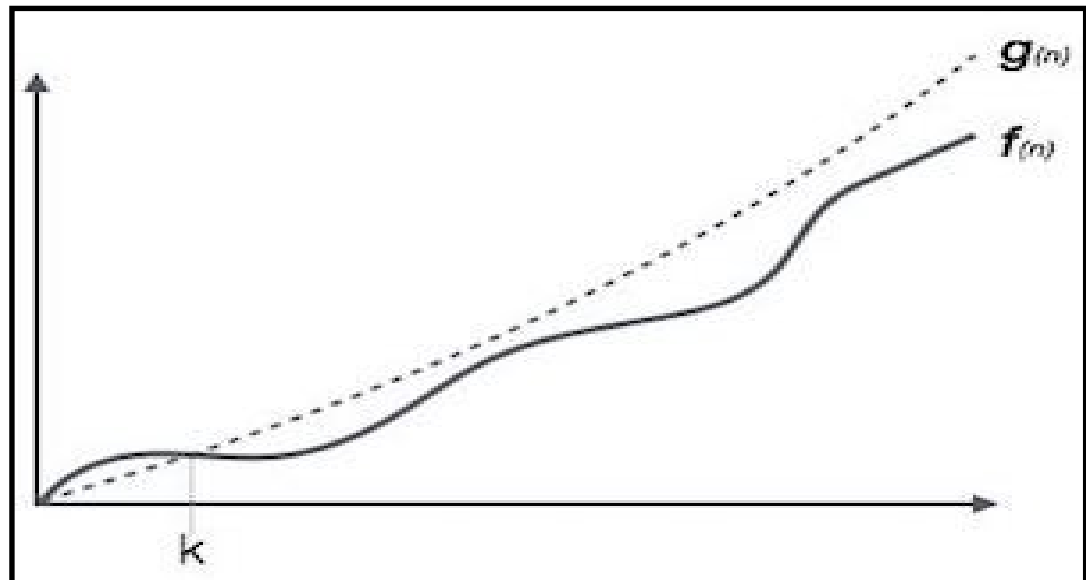
# Big-O Notation

**Definition:**  $f(n)$  is in  $O(g(n))$  if order of growth of  $f(n) \leq$  order of growth of  $g(n)$  (within constant multiple),

Examples:

- $10n$  is  $O(n^2)$
- $5n+20$  is  $O(n)$

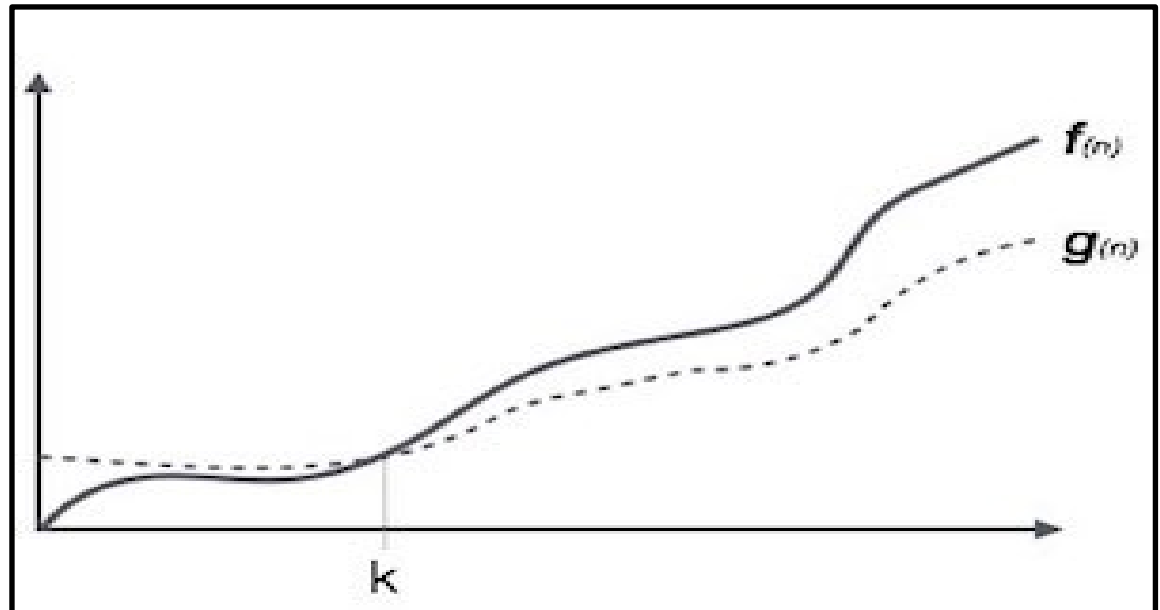
$$O(g(n)) = \left\{ f(n) : \text{there exist positive constants } c \text{ and } n_0 \text{ s.t.} \right. \\ \left. 0 \leq f(n) \leq cg(n) \text{ for all } n \geq n_0 \right\}$$



# Omega Notation, $\Omega$

The notation  $\Omega(n)$  is the formal way to express the Lower bound of an algorithm's running time.

$$\Omega(g(n)) = \left\{ \begin{array}{l} f(n) : \text{there exist positive constants } c \text{ and } n_0 \text{ s.t.} \\ 0 \leq cg(n) \leq f(n) \text{ for all } n \geq n_0 \end{array} \right\}$$

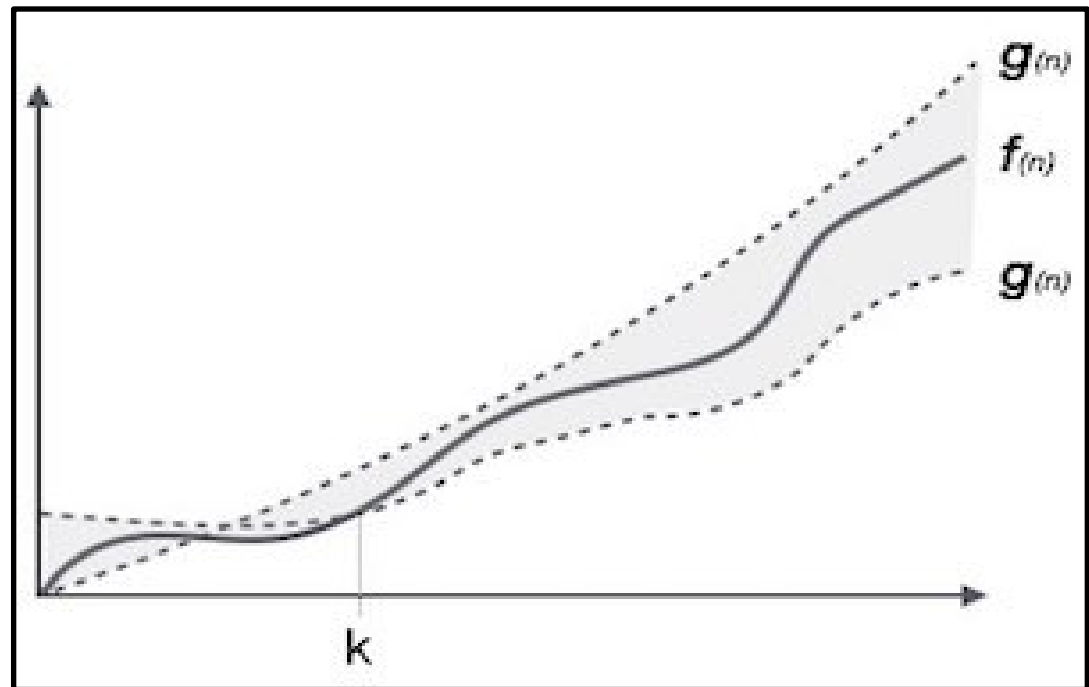


# Theta Notation, $\theta$



The notation  $\theta(n)$  is the formal way to express both the lower bound and the upper bound of an algorithm's running time. It is represented as follows –

$$\Theta(g(n)) = \left\{ \begin{array}{l} f(n) : \text{there exist positive constants } c_1, c_2, \text{ and } n_0 \text{ s.t.} \\ 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \text{ for all } n \geq n_0 \end{array} \right\}$$

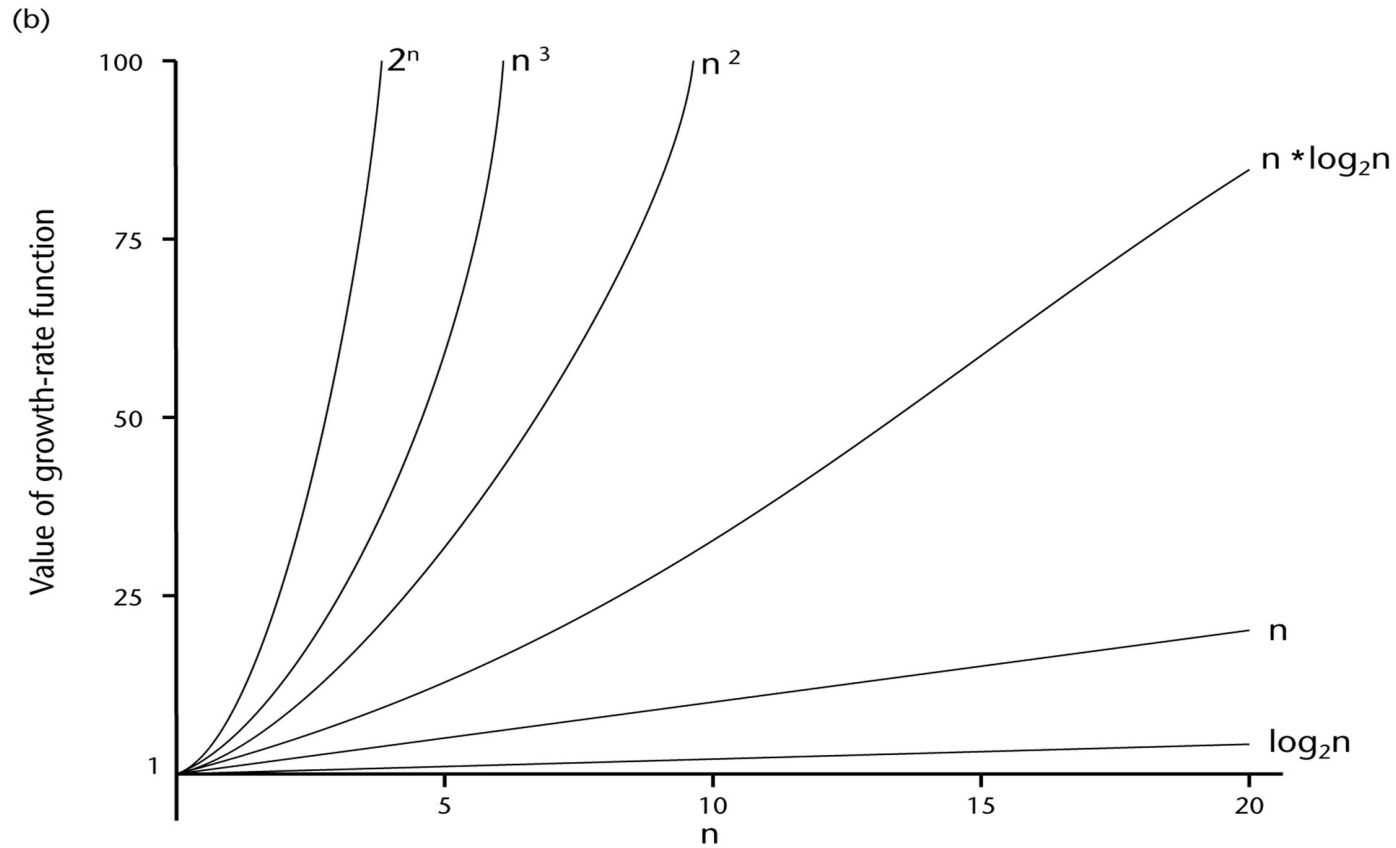


# Common Asymptotic Notations

## Common order-of-growth classifications

order of growth	name	typical code framework	description	example	$T(2N) / T(N)$
1	constant	<code>a = b + c;</code>	statement	add two numbers	1
$\log N$	logarithmic	<pre>while (N &gt; 1) {   N = N / 2; ... }</pre>	divide in half	binary search	$\sim 1$
$N$	linear	<pre>for (int i = 0; i &lt; N; i++) {   ... }</pre>	loop	find the maximum	2
$N \log N$	linearithmic	[see mergesort lecture]	divide and conquer	mergesort	$\sim 2$
$N^2$	quadratic	<pre>for (int i = 0; i &lt; N; i++)   for (int j = 0; j &lt; N; j++)   {     ...   }</pre>	double loop	check all pairs	4
$N^3$	cubic	<pre>for (int i = 0; i &lt; N; i++)   for (int j = 0; j &lt; N; j++)     for (int k = 0; k &lt; N; k++)     {       ...     }</pre>	triple loop	check all triples	8
$2^N$	exponential	[see combinatorial search lecture]	exhaustive search	check all subsets	$T(N)$

# A Comparison of Growth-Rate Functions



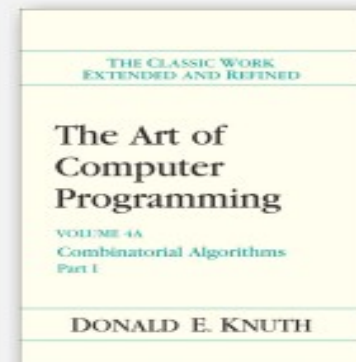
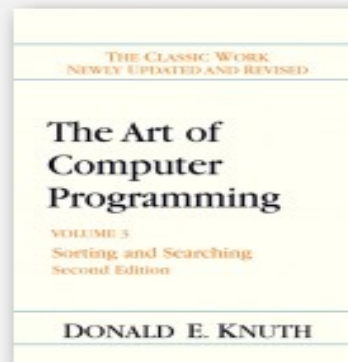
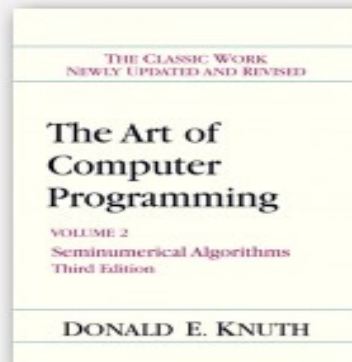
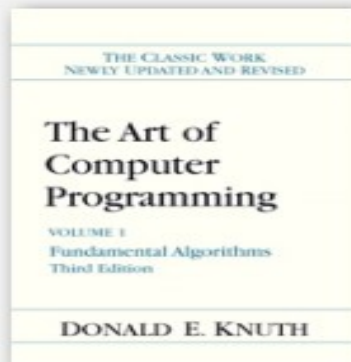


# How to Compute the run time of an Algorithm

## Mathematical models for running time

**Total running time:** sum of cost  $\times$  frequency for all operations.

- Need to analyze program to determine set of operations.
- Cost depends on machine, compiler.
- Frequency depends on algorithm, input data.



**In principle,** accurate mathematical models are available.

# How to Compute the run time of an Algorithm



## Mathematical models for running time

In principle, accurate mathematical models are available.

In practice,

- Formulas can be complicated.
- Advanced mathematics might be required.
- Exact models best left for experts.



costs (depend on machine, compiler)

$$T_N = c_1 A + c_2 B + c_3 C + c_4 D + c_5 E$$

$A$  = array access

$B$  = integer add

$C$  = integer compare

$D$  = increment

$E$  = variable assignment

frequencies  
(depend on algorithm, input)

Bottom line. We use **approximate** models in this course:  $T(N) \sim c N^3$ .

# Cost of basic operations



## Cost of basic operations

---

Challenge. How to estimate constants.

operation	example	nanoseconds †
integer add	$a + b$	2.1
integer multiply	$a * b$	2.4
integer divide	$a / b$	5.4
floating-point add	$a + b$	4.6
floating-point multiply	$a * b$	4.2
floating-point divide	$a / b$	13.5
sine	<code>Math.sin(theta)</code>	91.3
arctangent	<code>Math.atan2(y, x)</code>	129.0
...	...	...

† Running OS X on Macbook Pro 2.2GHz with 2GB RAM

Activate  
Go to Set

# Cost of basic operations



## Cost of basic operations

---

**Observation.** Most primitive operations take constant time.

operation	example	nanoseconds <sup>†</sup>
variable declaration	<code>int a</code>	$c_1$
assignment statement	<code>a = b</code>	$c_2$
integer compare	<code>a &lt; b</code>	$c_3$
array element access	<code>a[i]</code>	$c_4$
array length	<code>a.length</code>	$c_5$
1D array allocation	<code>new int[N]</code>	$c_6 N$
2D array allocation	<code>new int[N][N]</code>	$c_7 N^2$

**Caveat.** Non-primitive operations often take more than constant time.


# Cost of Instructions



## Example: 1-SUM

Q. How many instructions as a function of input size  $N$ ?

```
int count = 0;
for (int i = 0; i < N; i++)
    if (a[i] == 0)
        count++;
```

 N array accesses

operation	frequency
variable declaration	2
assignment statement	2
less than compare	$N + 1$
equal to compare	$N$
array access	$N$
increment	$N$ to $2N$

## Growth-Rate Functions – Example1



	<u>Cost</u>	<u>Times</u>
i = 1;	c1	1
sum = 0;	c2	1
while (i <= n) {	c3	n+1
i = i + 1;	c4	n
sum = sum + i;	c5	n
}		

$$\begin{aligned}T(n) &= c1 + c2 + (n+1)*c3 + n*c4 + n*c5 \\&= (c3+c4+c5)*n + (c1+c2+c3) \\&= a*n + b\end{aligned}$$

🏗️ So, the growth-rate function for this algorithm is **O(n)**

## Growth-Rate Functions – Example2



	<u>Cost</u>	<u>Times</u>
i=1;	c1	1
sum = 0;	c2	1
while (i <= n) {	c3	n+1
j=1;	c4	n
while (j <= n) {	c5	n*(n+1)
sum = sum + i;	c6	n*n
j = j + 1;	c7	n*n
}		
i = i + 1;	c8	n
}		

$$\begin{aligned}
 T(n) &= c1 + c2 + (n+1)*c3 + n*c4 + \\
 &\quad n*(n+1)*c5 + n*n*c6 + n*n*c7 + n*c8 \\
 &= (c5+c6+c7)*n^2 + (c3+c4+c5+c8)*n + (c1+c2+c3) \\
 &= a*n^2 + b*n + c
 \end{aligned}$$

⚙️ So, the growth-rate function for this algorithm is  **$O(n^2)$**

# Growth-Rate Functions – Example3



	<u>Cost</u>	<u>Times</u>
for (i=1; i<=n; i++)	c1	n+1
for (j=1; j<=i; j++)	c2	$\sum_{j=1}^n (j+1)$
for (no=1; no<=j; no++)	c3	$\sum_{j=1}^n \sum_{k=1}^j (k+1)$
x=x+1;	c4	$\sum_{j=1}^n \sum_{k=1}^j k$

$$T(n) = c1*(n+1) + c2*\left(\sum_{j=1}^n (j+1)\right) + c3*\left(\sum_{j=1}^n \sum_{k=1}^j (k+1)\right) + c4*\left(\sum_{j=1}^n \sum_{k=1}^j k\right)$$

$$= a*n^3 + b*n^2 + c*n + d$$

So, the growth-rate function for this algorithm is **O(n<sup>3</sup>)**



# Some Well-known Computational Problems

- Searching and Sorting
- Combinatorial problems
- Geometrical Problems
- Traveling salesman problem
- Knapsack problem
- Chess
- Towers of Hanoi
- Graph Problems

---

# Algorithms Examples

---

# Search Problems

## ■ Statement of problem:

- *Input:* A sequence of  $n$  numbers  $\langle a_1, a_2, \dots, a_n \rangle$
- Key= item to be Searched
- *Output:* Index of the item to be searched

■ **Instance:** The sequence  $\langle 5, 3, 2, 8, 3 \rangle$

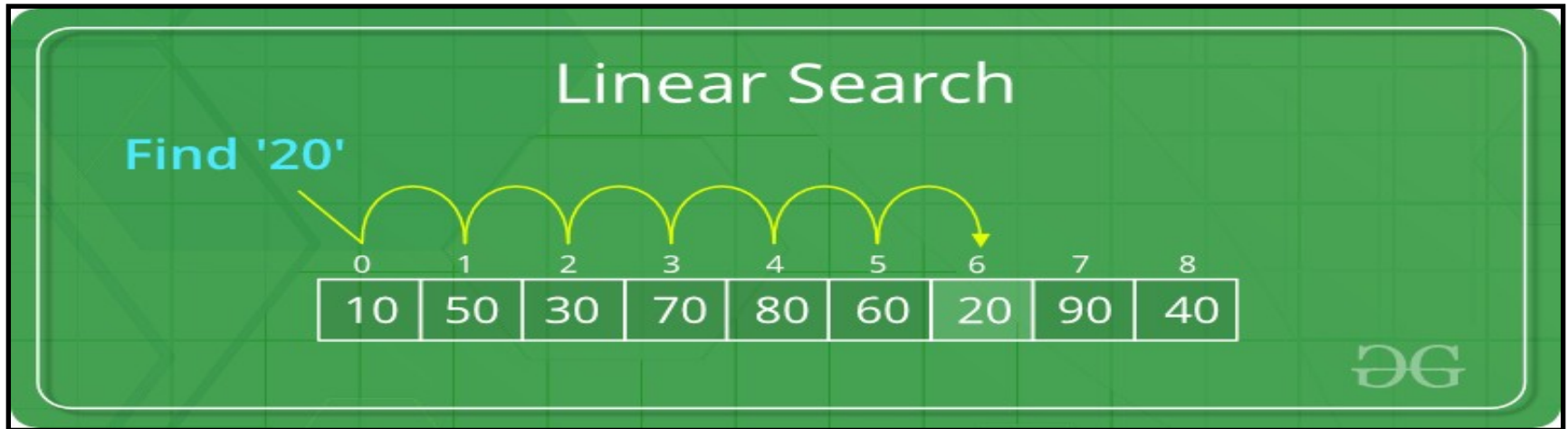
## ■ Algorithms:

- Sequential Search
- Binary Search(for sorted arrays)

# Linear Search

Worst and average case time complexity is order of  $N$ . So we need a better algorithm.

Key=20



# Binary Search

**Goal.** Given a sorted array and a key, find index of the key in the array?

**Binary search.** Compare key against middle entry.

- Too small, go left.
- Too big, go right.
- Equal, found.



**successful search for 33**

6	13	14	25	33	43	51	53	64	72	84	93	95	96	97
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
↑														↑
lo														hi

# Binary Search

**Goal.** Given a sorted array and a key, find index of the key in the array?

**Binary search.** Compare key against middle entry.

- Too small, go left.
- Too big, go right.
- Equal, found.



**successful search for 33**

6	13	14	25	33	43	51	53	64	72	84	93	95	96	97
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
↑														↑
lo														hi

# Binary Search Implementation

## Binary search: Java implementation

---

### Trivial to implement?

- First binary search published in 1946.
- First bug-free one in 1962.
- Bug in Java's `Arrays.binarySearch()` discovered in 2006.

```
public static int binarySearch(int[] a, int key)
{
    int lo = 0, hi = a.length-1;
    while (lo <= hi)
    {
        int mid = lo + (hi - lo) / 2;
        if      (key < a[mid]) hi = mid - 1;
        else if (key > a[mid]) lo = mid + 1;
        else return mid;
    }
    return -1;
}
```

← one "3-way compare"

**Invariant.** If key appears in the array `a[]`, then  $a[lo] \leq key \leq a[hi]$ .


# Analysis of Binary Search

## Binary search: mathematical analysis

**Proposition.** Binary search uses at most  $1 + \lg N$  key compares to search in a sorted array of size  $N$ .


**Def.**  $T(N)$  = # key compares to binary search a sorted subarray of size  $\leq N$ .

**Binary search recurrence.**  $T(N) \leq T(N/2) + 1$  for  $N > 1$ , with  $T(1) = 1$ .

$\uparrow$                        $\uparrow$   1. check for a condition whether  $N=1$

left or right half      possible to implement with one  
(floored division)      2-way compare (instead of 3-way)

**Pf sketch.** [assume  $N$  is a power of 2]

$$\begin{aligned} T(N) &\leq T(N/2) + 1 && \text{[ given ]} \\ &\leq T(N/4) + 1 + 1 && \text{[ apply recurrence to first term ]} \\ &\leq T(N/8) + 1 + 1 + 1 && \text{[ apply recurrence to first term ]} \\ &\vdots \\ &\leq T(N/N) + 1 + 1 + \dots + 1 && \text{[ stop applying, } T(1) = 1 \text{ ]} \\ &= 1 + \lg N \end{aligned}$$




# Sorting Problems

## ■ Statement of problem:

- *Input:* A sequence of  $n$  numbers  $\langle a_1, a_2, \dots, a_n \rangle$
- *Output:* A reordering of the input sequence  $\langle a'_1, a'_2, \dots, a'_n \rangle$  so that  $a'_i \leq a'_j$  whenever  $i < j$  ✓

■ **Instance:** The sequence  $\langle 5, 3, 2, 8, 3 \rangle$

## ■ Algorithms:

- Selection sort
- Insertion sort
- **Merge sort**
- **Quick Sort**
- (many others)

# 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.

## Examples:

- (i) searching for a word or phrase on WWW or in a Word document
- (ii) searching for a short read in the reference genomic sequence

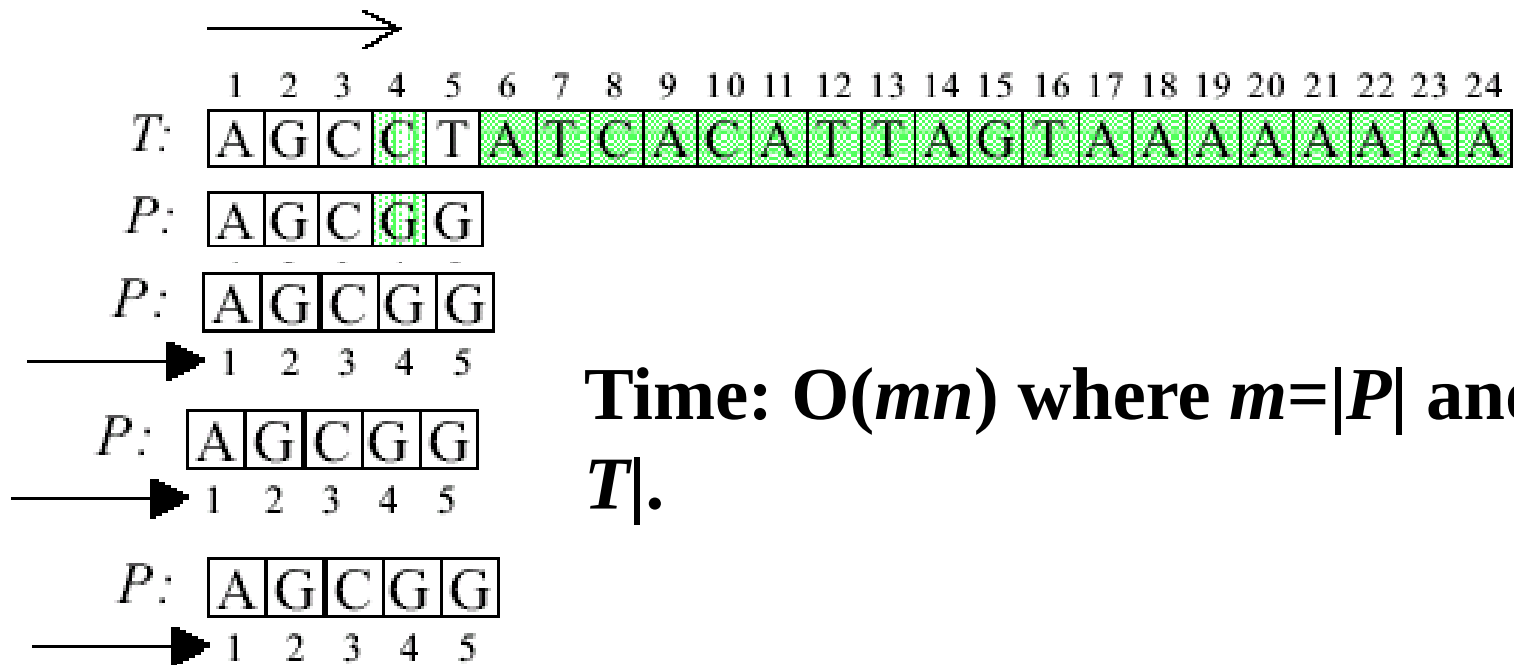
# String Matching

- Given a text string  $T$  of length  $n$  and a pattern string  $P$  of length  $m$ , the exact string matching problem is to find all occurrences of  $P$  in  $T$ .
- Example:  $T = \text{"AGCTTGA"}$        $P = \text{"GCT"}$
- **Applications:**
  - Searching keywords in a file
  - Searching engines (like ~~Google~~ and Openfind)
  - Database searching (GenBank)
- More string matching algorithms (with source codes):  
<http://www-igm.univ-mlv.fr/~lecroq/string/>

# String Matching

## Brute Force algorithm

- The brute force algorithm consists in checking, at all positions in the text between 0 and  $n-m$ , whether an occurrence of the pattern starts there or not. Then, after each attempt, it shifts the pattern by exactly one position to the right. ✓



# Graph Problems

## Informal definition

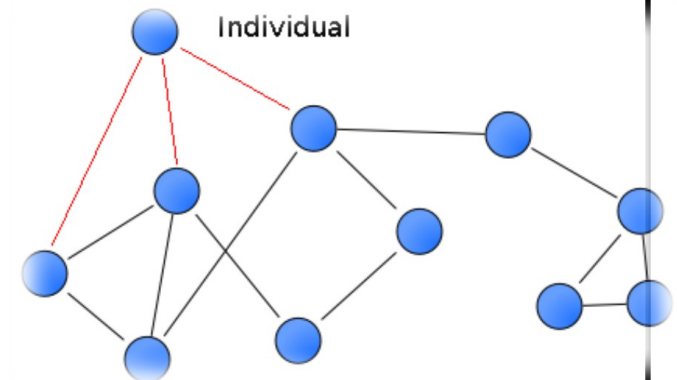
- A graph is a collection of points called vertices, some of which are connected by line segments called edges. ✓

## Modeling Real-life Problems

- Modeling WWW
- Communication networks
- Project scheduling ...

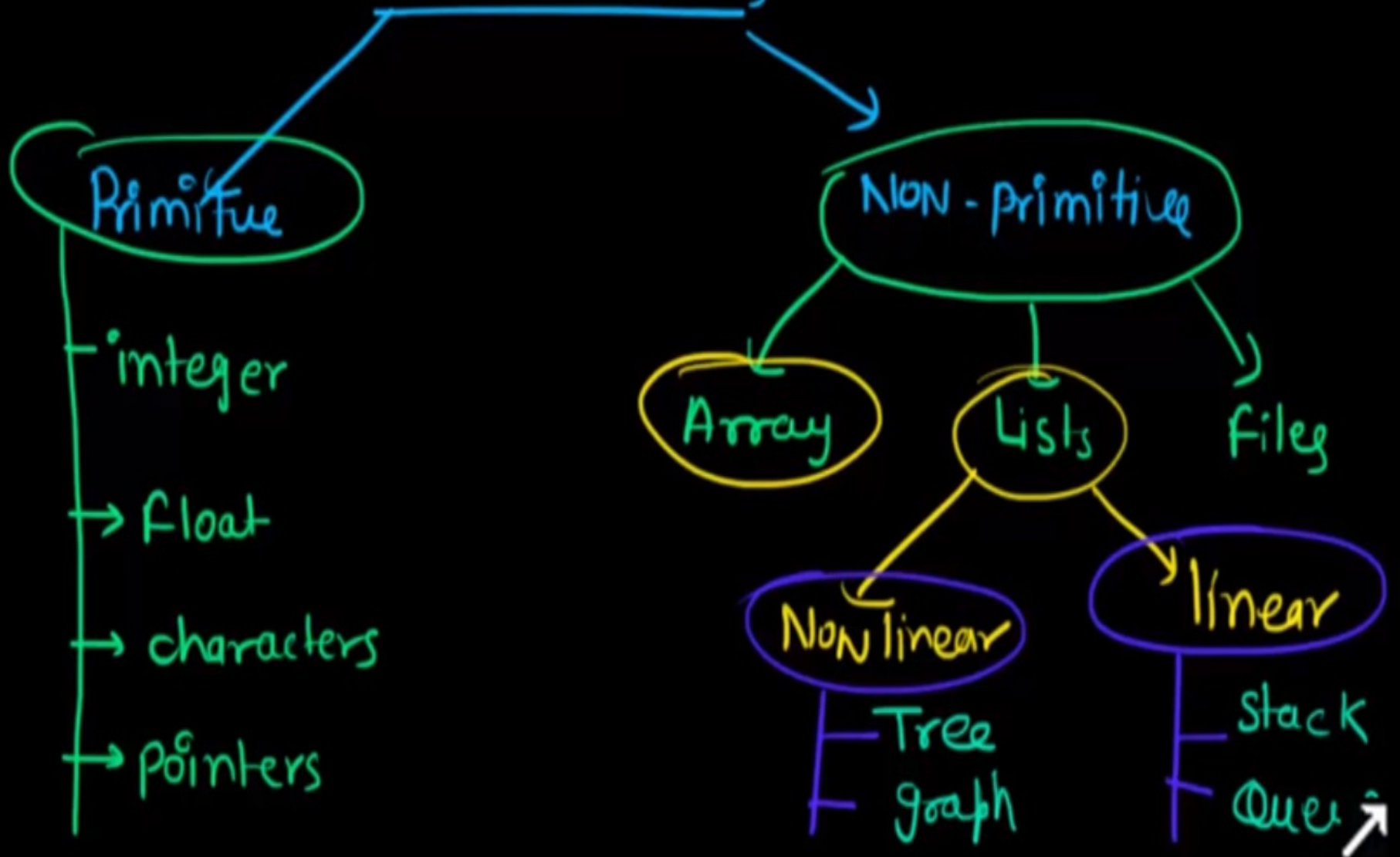
## Examples of Graph Algorithms

- Graph traversal algorithms
- Shortest-path algorithms
- .....



# Data Structures Review

# Data structures



# Fundamental data structures

## ■ list

- array
- linked list
- string

## ■ stack

## ■ queue

## ■ priority queue

□ graph

□ tree and binary tree



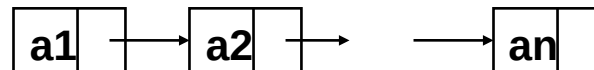
# Linear Data Structures

## Arrays

- A sequence of  $n$  items of the same data type that are stored contiguously in computer memory and made accessible by specifying a value of the array's index.

## Linked List

- A sequence of zero or more nodes each containing two kinds of information: some data and one or more links called pointers to other nodes of the linked list.
- Singly linked list (next pointer)
- Doubly linked list (next + previous pointers)



## Arrays

- **fixed length (need preliminary reservation of memory)**
- **contiguous memory locations**
- **direct access**
- **Insert/delete**

## Linked Lists

- **dynamic length**
- **arbitrary memory locations**
- **access by following links**
- **Insert/delete**

# Stacks and Queues

## Stacks

- A stack of plates
  - insertion/deletion can be done only at the top.
  - LIFO
- Two operations (push and pop)

## Queues

- A queue of customers waiting for services
  - Insertion/enqueue from the rear and deletion/dequeue from the front.
  - FIFO
- **Two operations** (enqueue and dequeue)

# Priority Queue and Heap

- Priority queues (implemented using heaps)
  - A data structure for maintaining a set of elements, each associated with a key/priority, with the following operations:
    - Finding the element with the highest priority
    - Deleting the element with the highest priority
    - Inserting a new element
  - Scheduling jobs on a shared computer

# Graphs

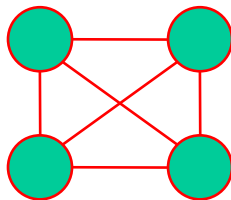
## Formal definition

- A graph  $G = \langle V, E \rangle$  is defined by a pair of two sets: a finite set  $V$  of items called **vertices** and a set  $E$  of vertex pairs called **edges**.

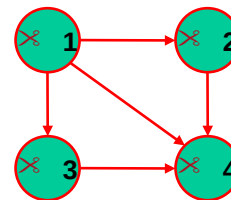
**Undirected** and **directed** graphs (**digraphs**).

## Complete, dense, and sparse graphs

- A graph with every pair of its vertices connected by an edge is called **complete**,  $K_{|V|}$
- **In Dense graph**, the number of edges is close to the maximal number of edges.



Complete



Dense

# Graph Representation

## Adjacency matrix

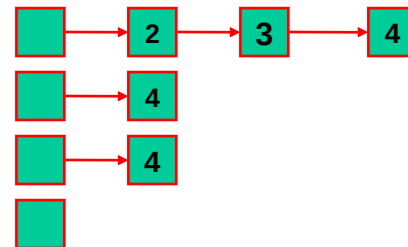
- $n \times n$  boolean matrix if  $|V|$  is  $n$ .
- The element on the  $i$ th row and  $j$ th column is 1 if there's an edge from  $i$ th vertex to the  $j$ th vertex; otherwise 0.
- The adjacency matrix of an undirected graph is symmetric.

## Adjacency linked lists

- A collection of linked lists, one for each vertex, that contain all the vertices adjacent to the list's vertex.

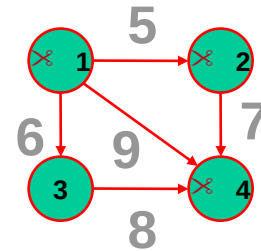
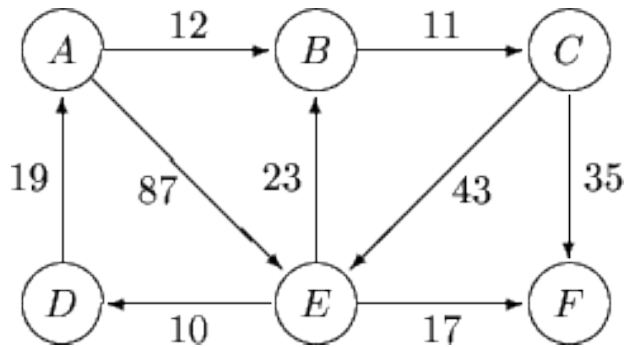
**Which data structure would you use if the graph is a 100-node star shape?**

0	1	1	1
0	0	0	1
0	0	0	1
0	0	0	1



# Weighted Graphs

- Graphs or digraphs with numbers assigned to the edges.



# Graph Properties -- Paths and Connectivity

## Paths

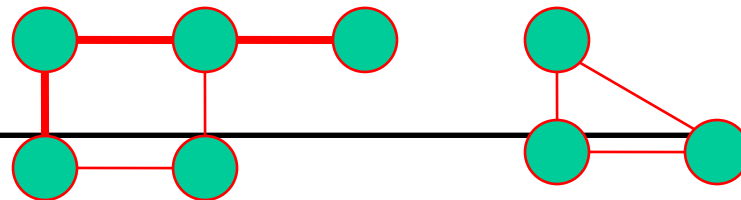
- A **path** from vertex  $u$  to  $v$  of a graph  $G$  is defined as a sequence of adjacent (connected by an edge) vertices that starts with  $u$  and ends with  $v$ .
- **Simple paths**: a path in a graph which does not have repeating vertices..
- **Path lengths**: the number of edges, or the number of vertices  $- 1$ .

## Connected graphs

- A graph is said to be **connected** if for every pair of its vertices  $u$  and  $v$  there is a path from  $u$  to  $v$ .

## Connected component

- The maximum connected subgraph of a given graph.



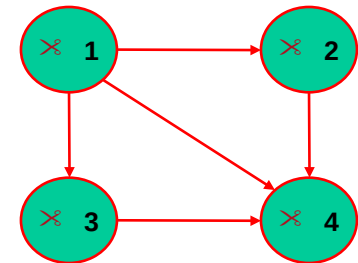
# Graph Properties – A cyclicity

## ■ Cycle

- A simple path of a positive length that starts and ends at the same vertex.

## ■ Acyclic graph

- A graph without cycles
- DAG (Directed Acyclic Graph)





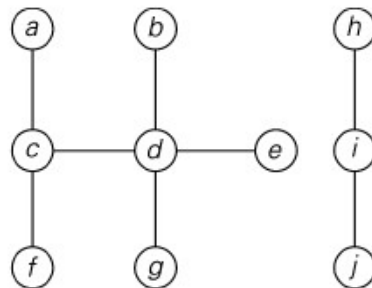
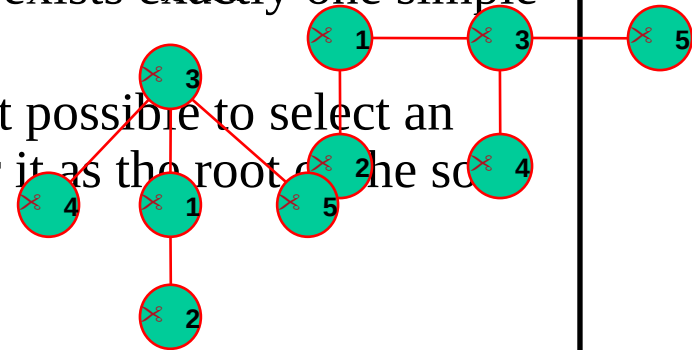
# Trees

## Trees

- A **tree** (or **free tree**) is a connected acyclic graph.
- **Forest**: a graph that has no cycles but is not necessarily connected.

## Properties of trees

- For every two vertices in a tree there always exists ~~exactly~~ one simple path from one of these vertices to the other.
  - **Rooted trees**: The above property makes it possible to select an arbitrary vertex in a free tree and consider it as the root of the so called rooted tree.
  - Levels in a rooted tree.



Forest

# Rooted Trees (I)

## Ancestors/predecessors

- For any vertex  $v$  in a tree  $T$ , all the vertices on the simple path from the root to that vertex are called ancestors.

## Descendants/children

- All the vertices for which a vertex  $v$  is an ancestor are said to be descendants of  $v$ .

## Parent, child and siblings

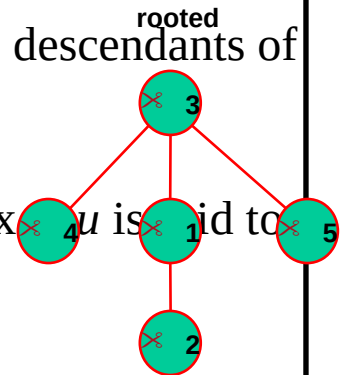
- If  $(u, v)$  is the last edge of the simple path from the root to vertex  $v$ ,  $u$  is called the parent of  $v$  and  $v$  is called a child of  $u$ .
- Vertices that have the same parent are called **siblings**.

## Leaves

- A vertex without children is called a leaf.

## Subtree

- A vertex  $v$  with all its descendants is called the subtree of  $T$  rooted at  $v$ .



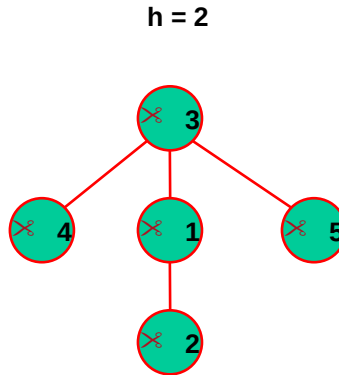
# Rooted Trees (II)

## Depth of a vertex

- The length of the simple path from the root to the vertex.

## Height of a tree

- The length of the longest simple path from the root to a leaf.



# Ordered Trees

## Ordered trees

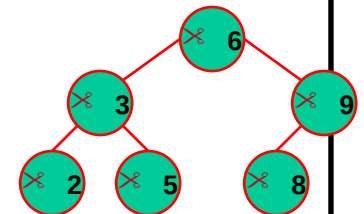
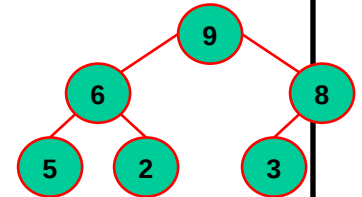
- An ordered tree is a rooted tree in which all the children of each vertex are ordered.

## Binary trees

- A binary tree is an ordered tree in which every vertex has no more than two children and each children is designated either a left child or a right child of its parent.

## Binary search trees

- Each vertex is assigned a number.
- A number assigned to each parental vertex is larger than all the numbers in its left subtree and smaller than all the numbers in its right subtree.



## Topics To be covered in next class

- Divide and Conquer
- General method,
- Applications-Binary search, Quick sort, Merge sort
- Selection Sort
- Finding maximum and minimum.
- Solving recurrence relations using Masters Theorem,
- Substitution method