

# Lecture Title: Introduction & Preliminary Discussions on Algorithms

Course Code: CSC 2211

Course Title: Algorithms



**Dept. of Computer Science  
Faculty of Science and Technology**

<b>Lecture No:</b>	01	<b>Week No:</b>	01	<b>Semester:</b>	
<b>Lecturer:</b>	<i>Rifath Mahmud, rifath.mahmud@aiub.edu</i>				

# Lecture Outline



1. Vision, Mission, Goal, Objective
2. Definitions
3. Computational Problems, Analysis, Design, and importance of Algorithms
4. Space & Time Complexity as parameters of performance for Algorithms including Asymptotic notations and
5. Preliminary data structure review

# **Vision & Mission of AIUB**

## **VISION**

AMERICAN INTERNATIONAL UNIVERSITY-BANGLADESH (AIUB) envisions promoting professionals and excellent leadership catering to the technological progress and development needs of the country.

## **MISSION**

AMERICAN INTERNATIONAL UNIVERSITY-BANGLADESH (AIUB) is committed to provide quality and excellent computer-based academic programs responsive to the emerging challenges of the time. It is dedicated to nurture and produce competent world class professional imbued with strong sense of ethical values ready to face the competitive world of arts, business, science, social science and technology.

# Goals of AIUB

- Sustain development and progress of the university
- Continue to upgrade educational services and facilities responsive of the demands for change and needs of the society
- Inculcate professional culture among management, faculty and personnel in the attainment of the institution's vision, mission and goals
- Enhance research consciousness in discovering new dimensions for curriculum development and enrichment
- Implement meaningful and relevant community outreach programs reflective of the available resources and expertise of the university
- Establish strong networking of programs, sharing of resources and expertise with local and international educational institutions and organizations
- Accelerate the participation of alumni, students and professionals in the implementation of educational programs and development of projects designed to expand and improve global academic standards

# **Vision & Mission of Computer Science Department**

## **VISION**

Provides leadership in the pursuit of quality and excellent computer education and produce highly skilled and globally competitive IT professionals.

## **MISSION**

Committed to educate students to think analytically and communicate effectively; train them to acquire technological, industry and research-oriented accepted skills; keep them abreast of the new trends and progress in the world of information communication technology; and inculcate in them the value of professional ethics.

# **Goals of Computer Science Department**

- Enrich the computer education curriculum to suit the needs of the industry-wide standards for both domestic and international markets
- Equip the faculty and staff with professional, modern technological and research skills
- Upgrade continuously computer hardware's, facilities and instructional materials to cope with the challenges of the information technology age
- Initiate and conduct relevant research, software development and outreach services.
- Establish linkage with industry and other IT-based organizations/institutions for sharing of resources and expertise, and better job opportunities for students

# **Course Objectives**

- The objective of this course is to teach how we can compare algorithms, the complexity analysis of time and space, effective data structure for solving difficult problems and implementation of different algorithms with the practical examples.
- The purpose of the course is
  - a) to raise your level of sophistication in thinking about the design and analysis of algorithms;
  - b) learn some of the classic algorithms and recent improvements;
  - c) exercise your creativity in designing algorithms.

# Course Prerequisites

- Programming
  - Data types, operations
  - Conditional statements
  - Loops
  - Procedures and functions
  - C/ C++/ Java
- Discrete Mathematics
- Data Structures (array, structure, pointer, file, etc...)
- Computer lab (edit, compile, execute, debug)
- **If you lack of any of the above please refine yourselves.**

# Importance of the course

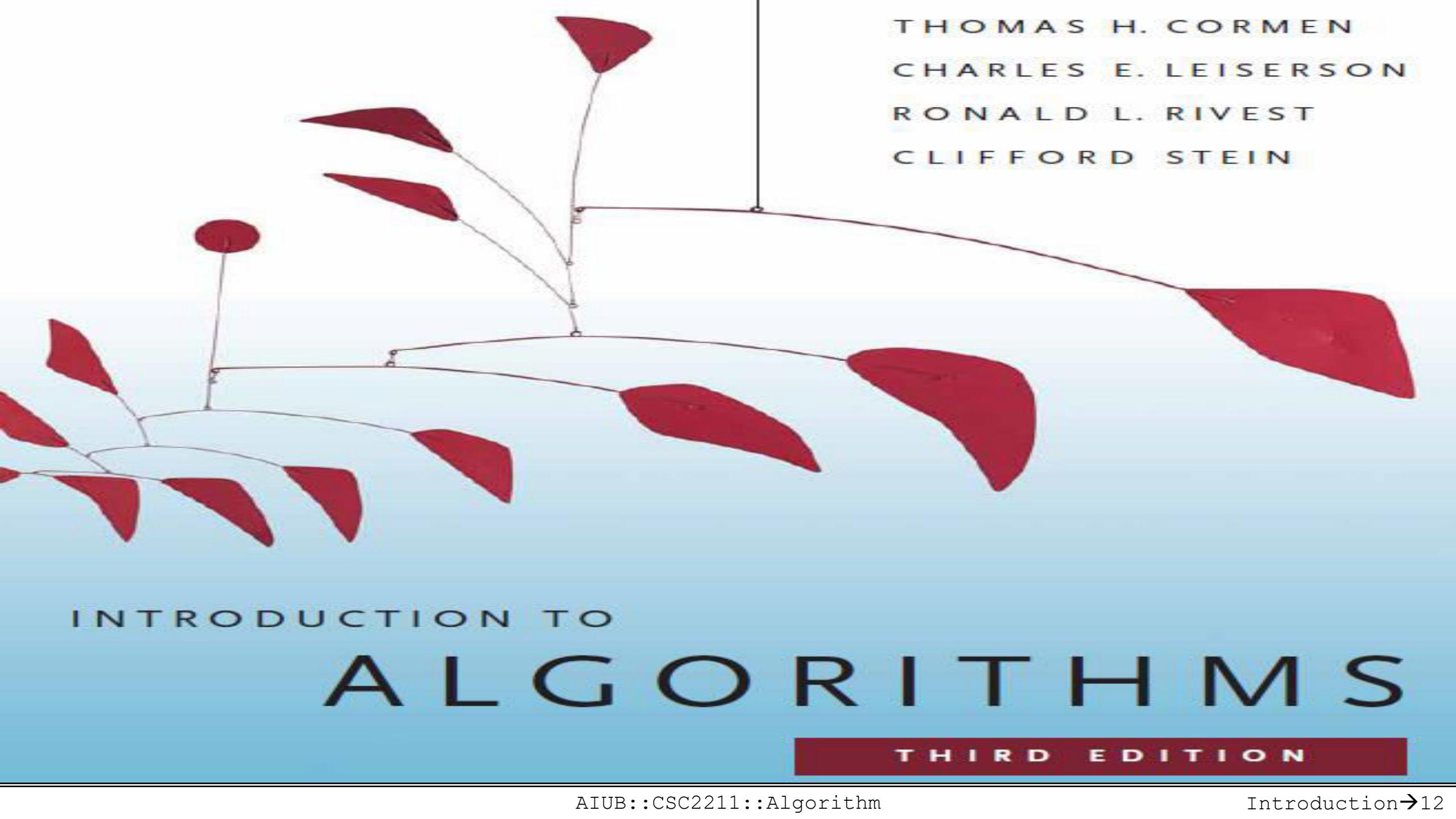
- This course is a continuation of the courses Programming Language 1 & 2, and Data Structure.
- Algorithm is required for all areas of computer science – especially for developing problem solving ability.
- This course will give the basic for the understanding of the courses – Theory of Computation, Artificial Intelligence, etc.
- This course will give the basic for the understanding of the concepts – Design and Analysis of Algorithm.

# Course Contents

- RAM model, Basic notation
- Recurrences & Master Method
- Dynamic Programming
- Greedy strategy
- Graphs Algorithms
- Greedy Graph Algorithm
- Shortest Path Algorithms
- Basic idea of NP – Completeness
- Basic idea of Elementary Geometric Methods & Review

# Resources & References

- *Introduction to Algorithms, Third Edition, Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein (CLRS).*
- *Fundamental of Computer Algorithms, Ellis Horowitz, Sartaj Sahni, Sanguthevar Rajasekaran (HSR)*
- *Lectures and Laboratory works will be provided online at the course website weekly.*



THOMAS H. CORMEN  
CHARLES E. LEISERSON  
RONALD L. RIVEST  
CLIFFORD STEIN

INTRODUCTION TO  
**ALGORITHMS**

THIRD EDITION

# Course Evaluation

<b>Midterm</b>	Quiz (Best One)	20	<b>40%</b>
	Laboratory Performance/Assignment/Exam	30	
	Class Attendance	10	
	Midterm Written Exam	40	
	<b>Midterm Total</b>		<b>100</b>
<b>Final term</b>	Quiz (Best One)	20	<b>60%</b>
	Laboratory Performance/Assignment/Exam	30	
	Class Attendance	10	
	Final term Written Exam	40	
	<b>Final Term Total</b>		<b>100</b>
<b>Grand Total</b>	<b>Final Grade of the Course</b>		<b>100</b>

# Classroom Policies

- **Must** be present inside the class in due time.
- **Class Break:** I would prefer to start the class in due time and leave the class in 10/15 minutes early for theory/Laboratory class respectively, instead of giving a break.
- Every class will start with a question-answer session about the last lecture. So students must be prepared with the contents and exercises from the last lecture.
- Students are suggested to ask questions during or after the lecture.
- **Additional/bonus marks** may be given to any good performances during the class.
- **Late in Class:**
  - Student coming after 10 minutes of due time is considered late.
  - 3 late attendances are considered as one absent.
  - Late during quiz/presentation are not given additional time.
  - Students who are regularly late might have additional deduction of marks.
  - A late student will be allowed to enter the class. Don't ask permission to enter the class, just get in slowly and silently. Same policy implies if a student wants to go out of the class for emergency reasons.

# Course Policies

- Attendance
- Laboratory Policies
- Makeup Evaluation (quiz, assignment, etc.)
- Grading Policies
- Dropping a Course

# Attendance

- At least 75% presence is required by the student. Absent classes must be defended by the student through application and proper documentation to the course teacher.
- Single absences or absences within 25% range will be judged by the course teacher.
- Long absences/irregular presence/absences out of 25% range must go through *application procedures* via department Head (+ probation office, if student is in *probation*) to attend the following classes.
- Acceptance of an application for absence only gives permission to attend the following classes. This might still result in deduction of marks (for attendance) which will be judged by the course teacher.

# Laboratory Policies

## ▪ **LABORATORY CLASSES:**

- ◆ First 0.5 – 1 hour will be spent explaining the problems/task/experiment to be performed.
- ◆ Next 1 – 1.5 hour(s) will be spent by the students to complete the experiment.
- ◆ Next 0.5 – 1 hour will be spent in checking, marking, and discussing the solution.
- ◆ Students are allowed to discuss with each other (unless instructed not to) in solving problems.
- ◆ But the checking (executing/viva) & marking will be with individual students only.

## ▪ **LABORATORY EXAM:**

- ◆ Laboratory exams are scheduled in the week before the major exams during the normal laboratory hours.
- ◆ Generally students are given one/more problems to be solved of which at least one part is solved using computers.
- ◆ One hour is given to the students to solve the problem. And half hour to submit and viva. Generally 20 students in the first 1.5 hours and the other 20 students in the rest 1.5 hours.
- ◆ Students may be given choices to select the problem. At most 3 selection can be given to a student with 0, 2, and 4 marks deduction as a penalty for each selection respectively.
- ◆ Only in case of unavoidable circumstances, the laboratory exams may be taken in the off days or week after the major exams.

# Makeup Evaluation

- There will be no makeup quiz as long as a student have appeared in 2 quizzes.
- Makeup for missing evaluations like quizzes/assignment submission date/presentation date/viva date/etc., must go through valid application procedure with supporting document within the deadline of the actual evaluation date.
- Makeup for missing Midterm/Final term must go through Set B form along with the supporting document within the <sup>1<sup>st</sup></sup> working day after exam week. The set B exam is generally scheduled from the <sup>2<sup>nd</sup></sup> working day after the exam week. Must get signature and exam date from the course teacher and get it approved by the department Head (monetary penalty might be imposed).
- Students unable to attend the set B exam may apply for set C exam within the same time limit as set B. Such applications must be supported by very strong reason and documentation, as they are generally rejected.
- The course teacher will be the judge of accepting/rejecting the request for makeup.

# Grading Policies

- All the evaluation categories & marks will be uploaded to the VUES within one week of the *evaluation process* except the attendance & performance, which will be uploaded along with the major (mid/final term) written exam marks.
- Letter grades ‘**A+**’ through ‘**F**’ is counted as grades. Other grades ‘**I**’ and ‘**UW**’ are considered as temporary grades which are counted/calculated as ‘F’ grade in the **CGPA**. These grades must/will be converted to the actual grades, i.e. ‘**A+**’ through ‘**F**’.
- ‘**I: INCOMPLETE**’ is given to students who have *missed* at most 30% of *evaluation categories* (quiz/assignment/etc.). Students must contact the course teacher for makeup, through valid application procedures immediately after grade release.
- ‘**UW: UNOFFICIAL WITHDRAW**’ is given when the *missing* *evaluation categories* are too high (more than 30%) to makeup. A student getting ‘**UW**’ has no option but to drop the course immediately after grade release

# Grading Policies...

- Once a student's gets 'I' or 'UW' and unable to fulfill the requirements with the course teacher for makeup, must drop the course within officially *mentioned time period* from the *registration department*.
- Students in probation or falls into the probation due to 'I'/'UW' grade are not allowed to drop the course.
- Unable to do so will result in the automatic conversion of the grades 'I'/'UW' to 'F' grade after the 4<sup>th</sup> week of the following semester.
- Any *problem with the mark/grade* must be consulted with the course teacher within *one week of the release of grades*.

# Dropping a Course

- Must fill up the drop form and get it signed by the course teacher, write an application to the vice chancellor and get it signed by the department Head, and finally submit the form & application to the registration department.
- The course teacher must write down the grades (if any) obtained in midterm, final, and grand total on the drop form.
- No drop is accepted during the following periods:
  - One week before midterm exam – grade release date of midterm exam.
  - One week before final term exam – grade release date of final grade.
- Student with 'F' grades in midterm, final term, or grand total cannot drop.
- Probation student are not allowed to drop any course.

# **Contacts**

- Contact information (email, office phone extension, office location, consulting hours, etc.) of the course teacher must be stored by the students.
- It is mandatory to contact/notify (preferably consulting hour/email) the course teacher for/of any problems/difficulties at the earliest possible. Late notification might not be considered.
- Update & correct your email address & phone number at VUES, as the teacher will contact/notify you of anything regarding the course through these information in VUES.

# Finally

- For any problems that could not be solved/understood during the lecture, students are advised to contact during the consultation hours and solve the problem.
- For any missing evaluation (quiz, assignment, etc.), classes, deadlines, etc. must contact/inform/notify the teacher immediately after missing in the consulting hour, via email, or in unavoidable circumstances – through the guardian or friend.
- Probation students must meet the course teacher once a week. So schedule your time with the teacher.
- Any kind of dishonesty, plagiarism, misbehavior, misconduct, etc. will not be tolerated. Might result in deduction of marks, 'F' grade, or reported to the AIUB Disciplinary Committee for drastic punishment.
- Always check/visit the AIUB home page for notices, rules & regulations of academic/university policies and important announcement for deadlines (Course drop, Exam permit, Exam Schedule, etc.).

Welcome to the  
course  
**Algorithms**

??????

**What will we do/learn in  
this course?**

# The Goals of this Course

- To *think algorithmically*
- To understand and learn the *idea* behind algorithm *design techniques*
- To get to know a *toolbox* of *classical* algorithms.
- To reason (in a precise and formal way) about the *efficiency* and the *correctness* of algorithms.

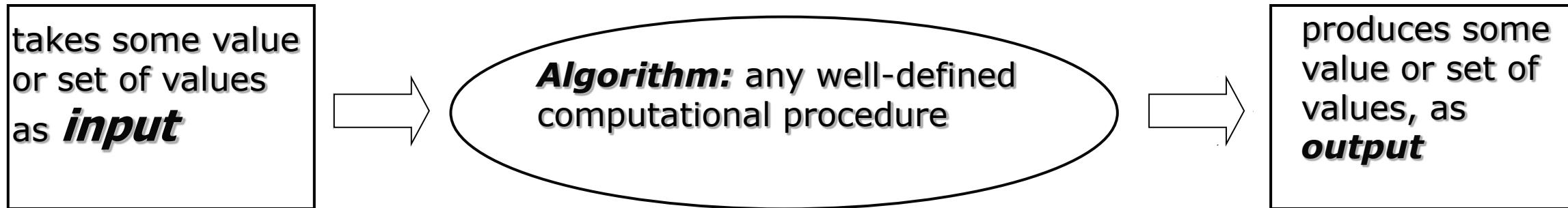
# I would request all of you to...

- Be *simple* and *precise* in understanding the problem
- Solve the problem first on the paper and then keyed in on the computer.
- During lectures:
  - Interaction is welcome; **ask questions.**
  - Additional explanations and examples if desired.
  - Speed up/slow down the progress.

??????

# What is Algorithm?

# Informally



- An algorithm is thus a sequence of ***computational steps*** that transform the input into the output.
- Solving a given problem:
  - **Data structure:** *Organization of data* to solve the problem at hand.
  - **Algorithm:** Outline, the essence of a computational procedure, step-by-step instructions.
  - **Program:** Implementation of an algorithm in some programming language.

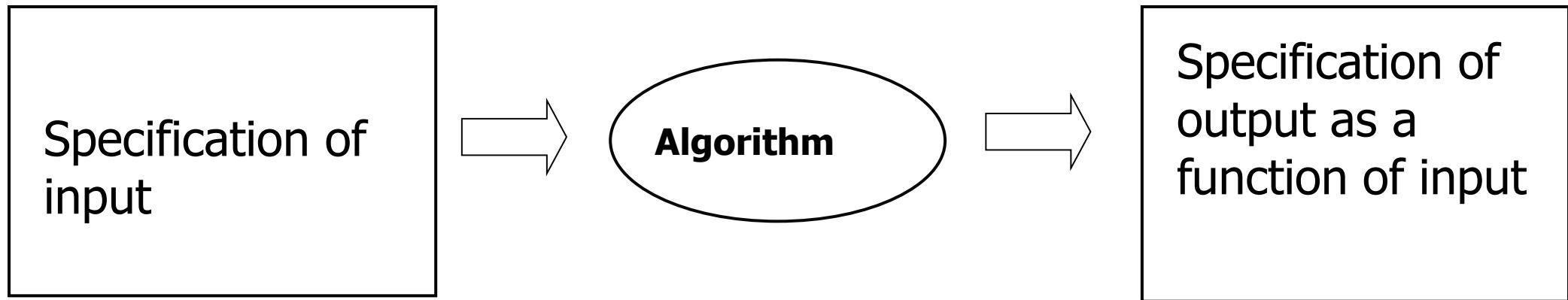
# **Kinds of Problem to be solved**

- **Sorting** and **Searching** are the basic and most common computational problem.
- Clever algorithms are employed for the Internet
  - to manage large volume of data transfer.
  - Finding good routes on which the data will travel.
  - Search engine to quickly find requested pages.
  - Etc...
- Numerical algorithms and number theory are employed in **electronic commerce** to keep and secure information such as credit card numbers, passwords, and bank statements.

# Kinds of Problem to be solved

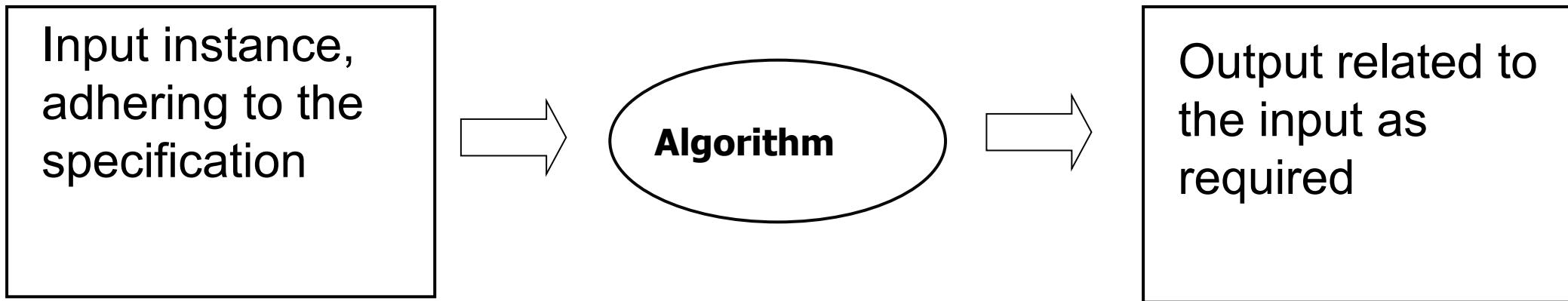
- Allocating scarce resources in the most beneficial way.
  - An oil company may wish to know where to place its wells in order to maximize its expected profit.
  - A candidate may want to determine where to spend money buying campaign advertising in order to maximize the chances of winning at election.
  - An airline may wish to assign crews to flight in the least expensive way possible, making sure that each flight is covered and that government policy regarding crew policies are met.
  - Etc...

# Algorithmic problem



- Finite number of input ***instances*** satisfying the specification.  
For example:
  - A sorted, non-decreasing sequence of natural numbers. The sequence is of non-zero, finite length:
    - ◆ 1, 20, 908, 909, 100000, 1000000000.
    - ◆ 3.

# Algorithmic Solution



- Algorithm describes actions on the input instance.
- There may be many correct algorithms for the same algorithmic problem.

# Definition of 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*.
- Properties:
  - Precision
  - Determinism
  - Finiteness
  - Efficiency
  - Correctness
  - Generality

# Overall Picture

Using a computer to help solve problems.

- Precisely specify the problem.
- Designing programs
  - Architecture → **data structure**
  - Technique → **algorithms**
- Writing programs
- Verifying (testing) programs

**Data Structure and Algorithm Design Goals**

**Correctness**



**Efficiency**



**Implementation Goals**

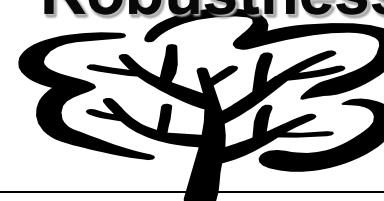
**Adaptability**



**Reusability**



**Robustness**



# How to Develop an Algorithm?

- **Precisely define** the problem.
  - Precisely specify the input and output.
  - Consider all cases.
- Come up with a **simple plan** to solve the problem at hand.
  - The plan is language independent.
  - The precise problem specification influences the plan.
- Turn the plan into an implementation
  - The problem representation (data structure) influences the implementation.

?????

***Suppose computers were infinitely fast  
and computer memory was free. Would  
you have any reason to study  
algorithms?***

# Algorithm Analysis

# Analysis of Algorithms

- Efficiency:
  - Running time
  - Space used
- Efficiency as a function of the *input size*:
  - Number of data elements (numbers, points).
  - The number of bits of an input number .

# The RAM Model

- RAM model represents a “generic” implementation of the algorithm
- Each “simple” operation (+, -, =, if, call) takes exactly 1 step.
- Loops and subroutine calls are not simple operations but depend upon the size of the data and the contents of a subroutine. We do not want “sort” to be a single step operation.
- Each memory access takes exactly 1 step.

# The RAM model (cntd..)

- It is important to choose the level of detail.
- The RAM model:
  - Instructions (each taking constant time), we usually choose one type of instruction as a **characteristic** operation that is counted:
    - ◆ Arithmetic (add, subtract, multiply, etc.)
    - ◆ Data movement (assign)
    - ◆ Control flow (branch, subroutine call, return)
    - ◆ Comparison (logical ops)
  - Data types – integers, characters, and floats

# Example

Algorithm *arrayMax*(*A*, *n*)

*currentMax*  $\leftarrow A[0]$

**for** (*i* = 1; *i* < *n*; *i*++)

**if** *A*[*i*] > *currentMax* **then**

*currentMax*  $\leftarrow A[i]$

**return** *currentMax*

# operations

2

$2n$  // (*i*=1 once, *i*<*n* *n* times, *i*++ (*n*-1) times)

$2(n - 1)$

$2(n - 1)$

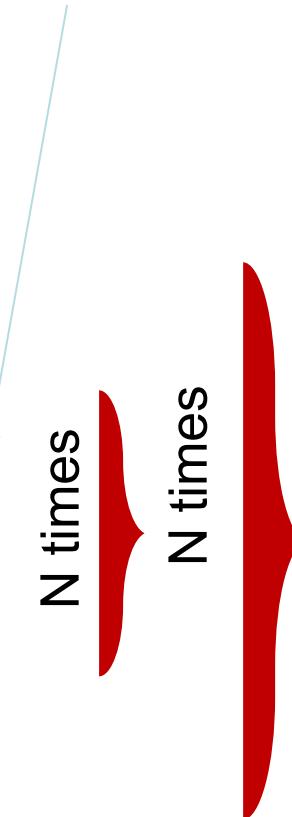
1

Total  $6n - 1$

## Example: N-by-N matrix, N-by-1 vector, multiply

(3 accesses, 1 multiply, 1 add)

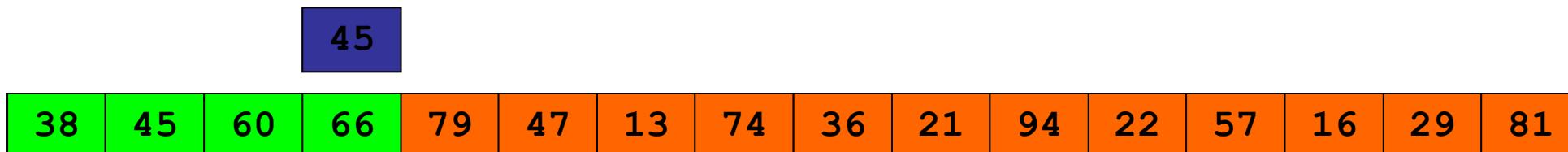
```
Y = zeros(N,1);      initialize space, c1N  
for i=1:N            initialize "for" loop, c2N  
    Y(i) = 0.0;        Scalar assignment, c3  
    for j=1:N          initialize "for" loop, c2N  
        Y(i) = Y(i) + A(i,j)*x(j);  
    end                End of loop, return/exit, c5  
end                  End of loop, return/exit, c5
```



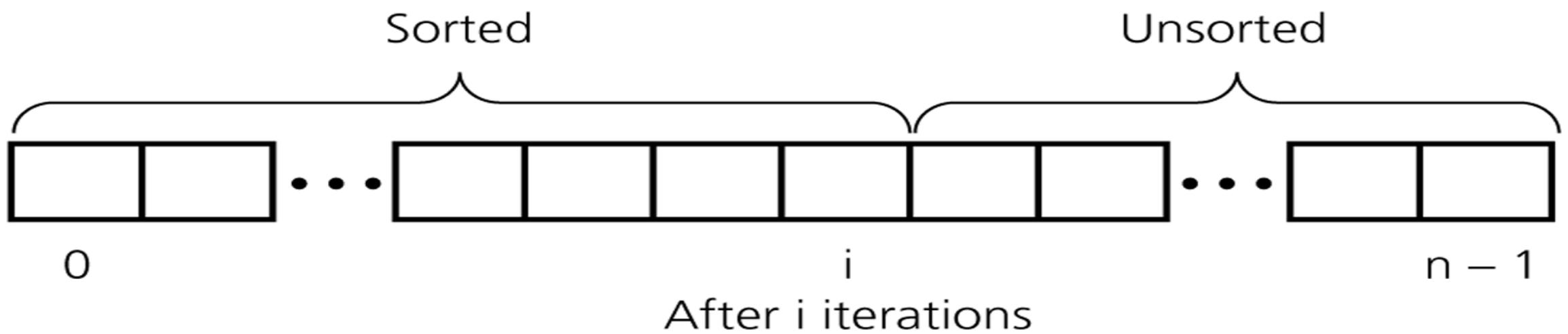
$$\begin{aligned}\text{Total} &= c_1N + c_2N + N(c_3 + c_2N + N(c_4 + c_5) + c_5) \\&= (c_2 + c_4 + c_5)N^2 + (c_1 + c_2 + c_3 + c_5)N \\&= c_6N^2 + c_7N\end{aligned}$$

# Insertion Sort

- while some elements unsorted:
  - Using linear search, find the location in the sorted portion where the 1<sup>st</sup> element of the unsorted portion should be inserted
  - Move all the elements after the insertion location up one position to make space for the new element

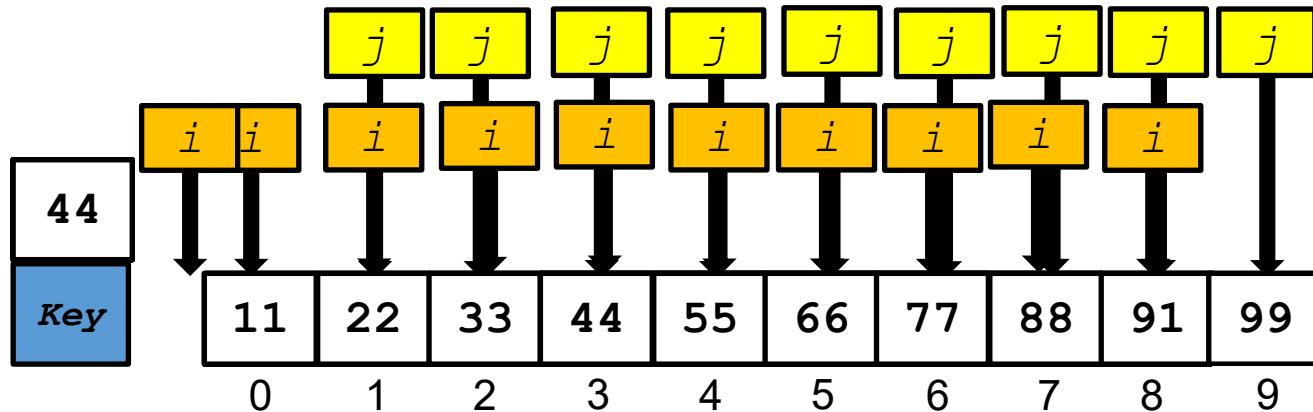


the fourth iteration of this loop is shown here



An insertion sort partitions the array into two regions

# Insertion Sort Simulation



INSERTION-SORT( $A$ )

```
1  for  $j = 2$  to  $A.length$ 
2      key =  $A[j]$ 
3      // Insert  $A[j]$  into the sorted sequence  $A[1..j - 1]$ 
4       $i = j - 1$ 
5      while  $i > 0$  and  $A[i] > key$ 
6           $A[i + 1] = A[i]$ 
7           $i = i - 1$ 
8       $A[i + 1] = key$ 
```

# Analysis of Insertion Sort

- Time to compute the **running time** as a function of the **input size** (exact analysis).

	cost	times
<b>for</b> $j := 2$ <b>to</b> $n$ <b>do</b>	$c_1$	$n$
key := $A[j]$	$c_2$	$n-1$
<b>// Insert <math>A[j]</math> into <math>A[1..j-1]</math></b>	0	$n-1$
$i := j-1$	$c_3$	$\sum_{j=2}^n t_j$
<b>while</b> $i > 0$ <b>and</b> $A[i] > \text{key}$ <b>do</b>	$c_4$	$\sum_{j=2}^n (t_j - 1)$
$A[i+1] := A[i]$	$c_5$	
$i --$	$c_6$	
$A[i+1] := \text{key}$	$c_7$	$n-1$

# ...Analysis of Insertion Sort

- The running time of an algorithm is the sum of the running times of each statement.
- A statement with cost  $c$  that is executed  $n$  times contributes  $c*n$  to the running time.
- The total running time  $T(n)$  of insertion sort is

$$T(n) = c_1 * n + c_2(n-1) + c_3(n-1) + c_4 \sum_{j=2}^n t_j + c_5 \sum_{j=2}^n (t_j - 1) + c_6 \sum_{j=2}^n (t_j - 1) \\ + c_7 (n-1)$$

# ...Analysis of Insertion Sort

- Often the performance depends on the details of the input
- This is modeled by  $t_j$ .
- In the case of insertion sort the time  $t_j$  depends on the original sorting of the input array.

# Performance Analysis

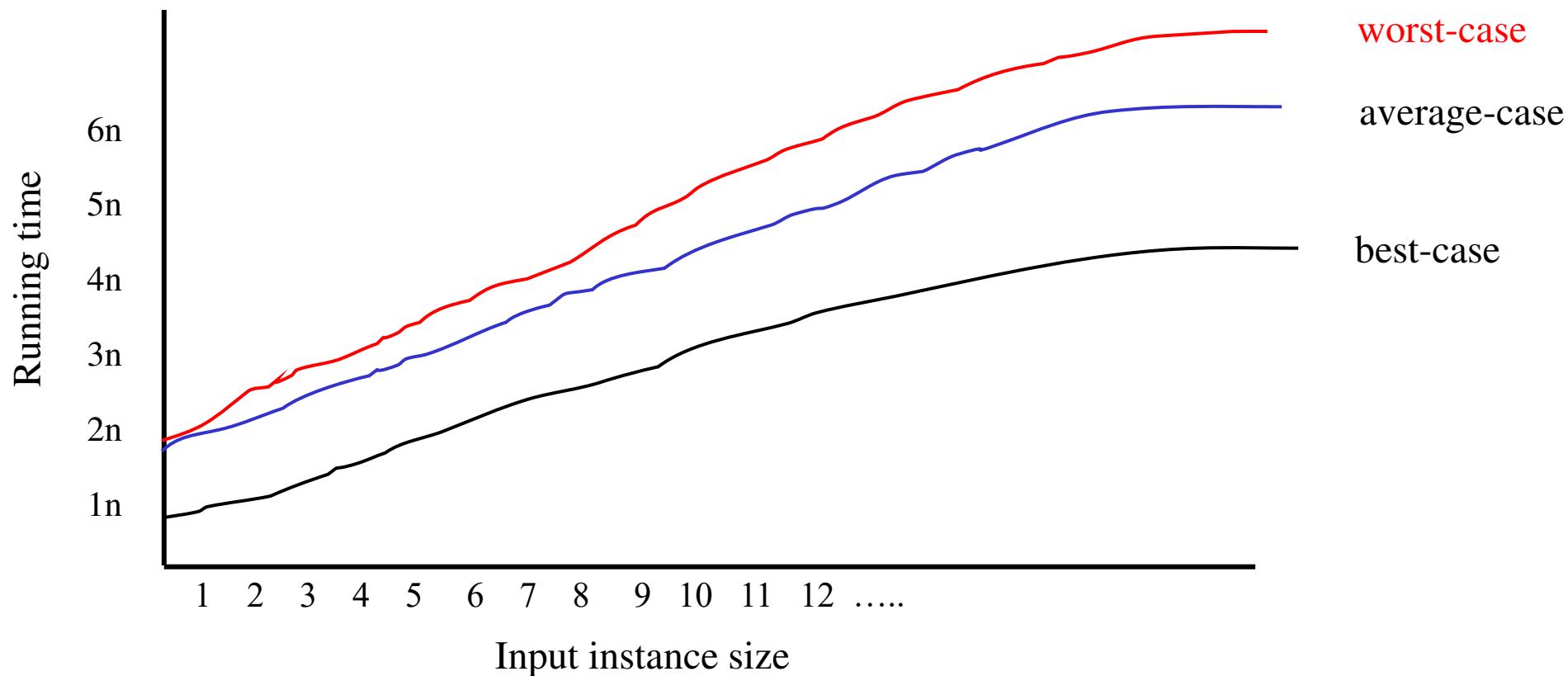
- Performance often draws the line between what is feasible and what is impossible.
- Often it is sufficient to count the number of iterations of the core (innermost) part.
  - No distinction between comparisons, assignments, etc (that means roughly the same cost for all of them).
  - Gives precise enough results.
- In some cases the cost of selected operations dominates all other costs.
  - Disk I/O versus RAM operations.
  - Database systems.

# Best/ Worst/ Average Case

- **Best case:** works fast on *some* input.
- **Worst case:** (usually) maximum time of algorithm on any input of size.
- **Average case:** (sometimes) expected time of algorithm over all inputs of size. Need assumption of statistical distribution of inputs.
- Analyzing insertion sort's
  - **Best case:** elements already sorted,  $t_j=1$ , running time  $\approx an + b$ , i.e., *linear* time.
  - **Worst case:** elements are sorted in inverse order,  $t_j=j$ , running time  $\approx an^2+bn+c$ , i.e., *quadratic* time.
  - **Average case:**  $t_j=j/2$ , running time  $\approx a'n^2+b'n+c'$ , i.e., *quadratic* time.

# ...Best/ Worst/ Average Case

- For inputs of all sizes:



# ...Best/ Worst/ Average Case

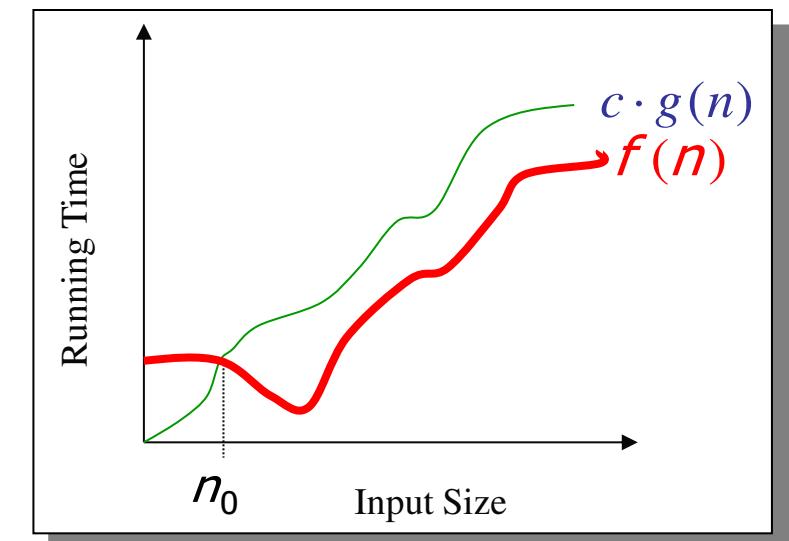
- **Worst case** is usually used:
  - It is an upper-bound.
  - In certain application domains (e.g., air traffic control, surgery) knowing the **worst-case** time complexity is of crucial importance.
  - For some algorithms **worst case** occurs fairly often
  - The **average case** is often as bad as the **worst case**.
  - Finding the **average case** can be very difficult.

# Asymptotic Notation

- **Asymptotic Notations** are languages that allow us to analyze an algorithm's running time by identifying its behavior as the input size for the algorithm increases.
- This is also known as an algorithm's growth rate.(Wiki)

# Asymptotic Notation

- The “big-Oh” O-Notation
  - asymptotic upper bound
  - $f(n) = O(g(n))$ , if there exists constants  $c > 0$  and  $n_0 > 0$ , s.t.  $f(n) \leq c g(n)$  for  $n \geq n_0$
  - $f(n)$  and  $g(n)$  are functions over non-negative integers
- Used for **worst-case** analysis



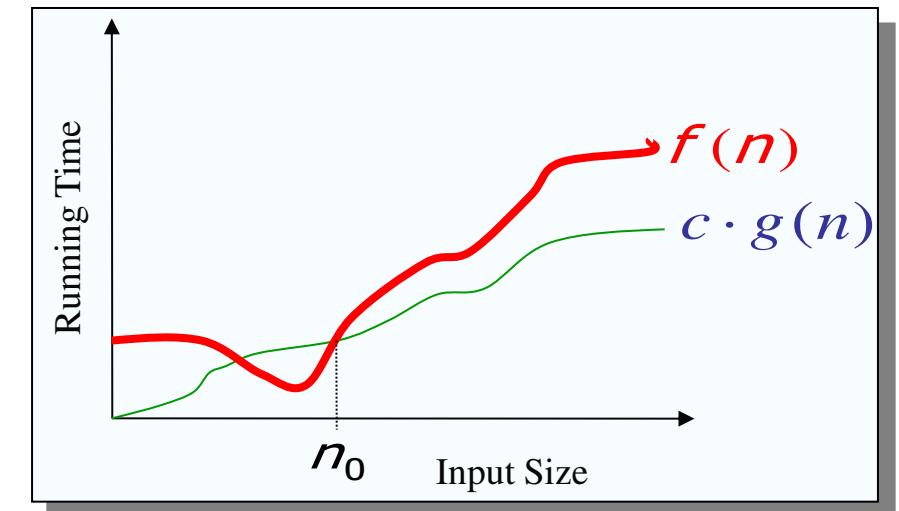
# Asymptotic Notation

notation	name
$O(1)$	constant
$O(\log(n))$	logarithmic
$O((\log(n))^c)$	polylogarithmic
$O(n)$	linear
$O(n^2)$	quadratic
$O(n^c)$	polynomial
$O(c^n)$	exponential

# ...Asymptotic Notation

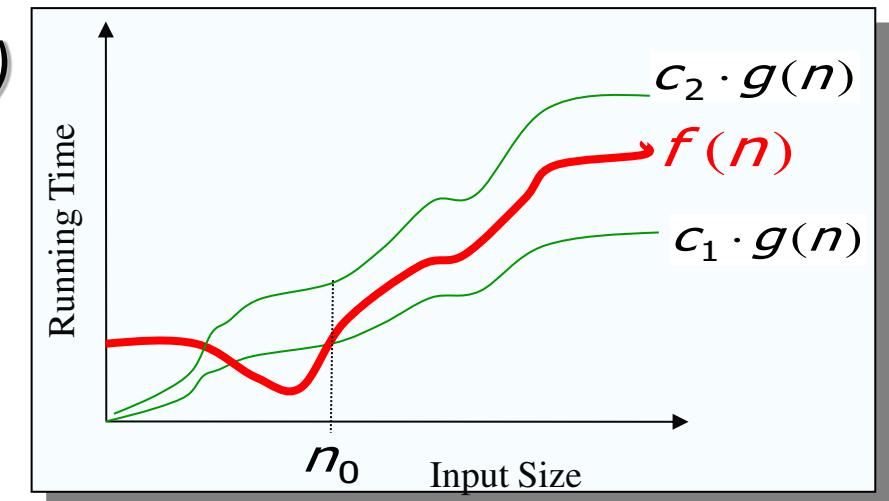
- The “big-Omega”  $\Omega$ -Notation
  - asymptotic lower bound
  - $f(n) = \Omega(g(n))$  if there exists constants  $c > 0$  and  $n_0 > 0$ , s.t.  $c g(n) \leq f(n)$  for  $n \geq n_0$
- Used to describe **best-case** running times or lower bounds of algorithmic problems.

E.g., lower-bound of searching in an unsorted array is  $\Omega(n)$ .



# ...Asymptotic Notation

- The “big-Theta”  $\Theta$ -Notation
  - asymptotically tight bound
  - $f(n) = \Theta(g(n))$  if there exists constants  $c_1 > 0$ ,  $c_2 > 0$ , and  $n_0 > 0$ , s.t. for  $n \geq n_0$   $c_1 g(n) \leq f(n) \leq c_2 g(n)$
- $f(n) = \Theta(g(n))$  if and only if  $f(n) = O(g(n))$ ,  $f(n) = \Omega(g(n))$



# Asymptotic Analysis

- **Goal:** to simplify the analysis of the running time by getting rid of details, which are affected by specific implementation and hardware
  - rounding of numbers:  $1,000,001 \approx 1,000,000$
  - rounding of functions:  $3n^2 \approx n^2$
- **Capturing the essence:** how the running time of an algorithm increases with the size of the input *in the limit.*
  - Asymptotically more efficient algorithms are best for all but small inputs

# ...Asymptotic Analysis

- **Simple Rule:** Drop lower order terms and constant factors.
  - $50 n \log n$  is  $O(n \log n)$
  - $7n - 3$  is  $O(n)$
  - $8n^2 \log n + 5n^2 + n$  is  $O(n^2 \log n)$

# Time complexity familiar tasks

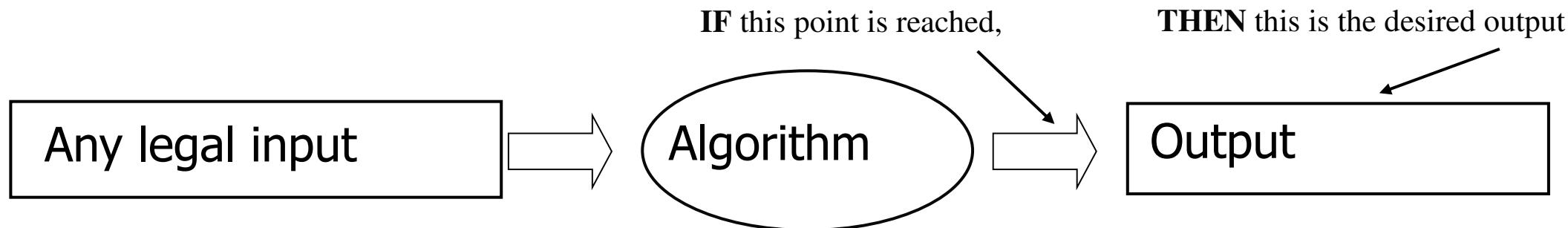
<u>Task</u>	Growth rate
Matrix/vector multiply	$O(N^2)$
Getting a specific element from a list	$O(1)$
Dividing a list in half, dividing one halve in half, etc	$O(\log_2 N)$
Binary Search	$O(\log_2 N)$
Scanning (brute force search) a list	$O(N)$
Nested <code>for</code> loops (k levels)	$O(N^k)$
MergeSort	$O(N \log_2 N)$
BubbleSort	$O(N^2)$
Generate all subsets of a set of data	$O(2^N)$
Generate all permutations of a set of data	$O(N!)$

# Correctness of Algorithms

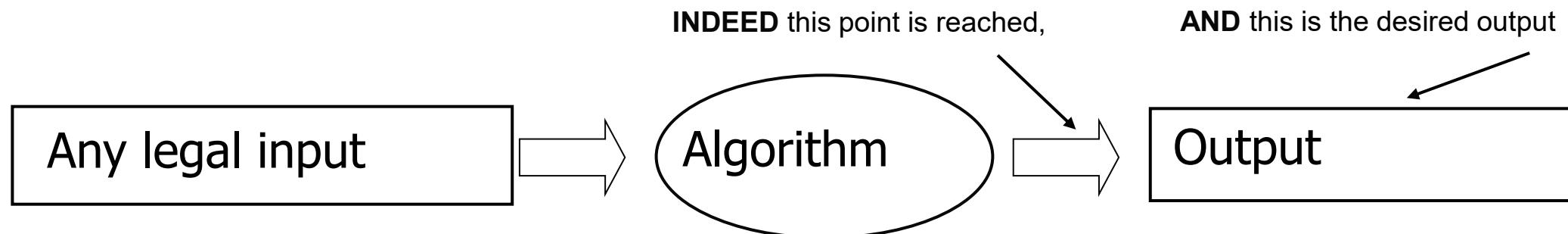
- An algorithm is ***correct*** if for any legal input it ***terminates*** and ***produces the desired output***.
- Automatic proof of correctness is not possible (so far).
- There are practical techniques and rigorous formalisms that help to reason about the correctness of (parts of) algorithms.

# Partial and Total Correctness

- Partial correctness



- Total correctness



# Assertions

- To prove partial correctness we associate a number of **assertions** (statements about the state of the execution) with specific checkpoints in the algorithm.
- **Preconditions** – assertions that must be valid *before* the execution of an algorithm or a subroutine (*INPUT*).
- **Postconditions** – assertions that must be valid *after* the execution of an algorithm or a subroutine (*OUTPUT*).

# Pre/post-conditions

- **Example:**
  - Write a pseudocode algorithm to find the two smallest numbers in a sequence of numbers (given as an array).
- **INPUT:** an array of integers  $A[1..n]$ ,  $n > 0$
- **OUTPUT:**  $(m_1, m_2)$  such that
  - $m_1 < m_2$  and for each  $i \in [1..n]$ :  $m_1 \leq A[i]$  and, if  $A[i] \neq m_1$ , then  $m_2 \leq A[i]$ .
  - $m_2 = m_1 = A[1]$  if  $\forall j, i \in [1..n]: A[i] = A[j]$

# Loop Invariants

- **Invariants:** assertions that are valid any time they are reached (many times during the execution of an algorithm, e.g., in loops)
- We must show three things about loop invariants:
  - **Initialization:** it is true prior to the first iteration.
  - **Maintenance:** *if* it is true before an iteration, *then* it is true after the iteration.
  - **Termination:** when a loop terminates the invariant gives a useful property to show the correctness of the algorithm

# Example: Binary Search

- We want to show that  $q$  is not in  $\mathbf{A}$  if **NIL** is returned.

- **Invariant:**

$$\forall i \in [1..l-1]: A[i] < q \quad (\text{Ia})$$

$$\forall i \in [r+1..n]: A[i] > q \quad (\text{Ib})$$

- **Initialization:**  $l = 1, r = n$  the invariant holds because there are no elements to the left of  $l$  or to the right of  $r$ .
- $l=1$  yields  $\forall j, i \in [1..0]: A[i] < q$   
this holds because  $[1..0]$  is empty
- $r=n$  yields  $\forall j, i \in [n+1..n]: A[i] > q$   
this holds because  $[n+1..n]$  is empty

```
l := 1
r := n
do
    m := ⌊(l+r)/2⌋
    if A[m] = q then return m
    else if A[m] > q then r := m-1
    else l := m+1
while l <= r
return NIL
```

# ...Example: Binary Search

## Invariant:

- $\forall i \in [1..l-1]: A[i] < q$  (Ia)
- $\forall i \in [r+1..n]: A[i] > q$  (Ib)

```
l := 1
r := n
do
    m := ⌊(l+r)/2⌋
    if A[m] = q then return m
    else if A[m] > q then r := m-1
    else l := m+1
while l <= r
return NIL
```

- Maintenance:**  $l, r, m = \lfloor(l+r)/2\rfloor$
- $A[m] \neq q$  &  $A[m] > q$ ,  $r=m-1$ , A sorted implies  
 $\forall k \in [r+1..n]: A[k] > q$  (Ib)
- $A[m] \neq q$  &  $A[m] < q$ ,  $l=m+1$ , A sorted implies  
 $\forall k \in [1..l-1]: A[k] < q$  (Ia)

# ...Example: Binary Search

- **Invariant:**

$$\forall i \in [1..l-1]: A[i] < q \quad (\text{Ia})$$

$$\forall i \in [r+1..n]: A[i] > q \quad (\text{Ib})$$

```
l := 1
r := n
do
    m := ⌊(l+r)/2⌋
    if A[m] = q then return m
    else if A[m] > q then r := m-1
    else l := m+1
while l <= r
return NIL
```

- **Termination:** l, r, l<=r
- Two cases:
  - l:=m+1 we get ⌊(l+r)/2⌋ +1 > l
  - r:=m-1 we get ⌊(l+r)/2⌋ -1 < r
- The range gets smaller during each iteration and the loop will terminate when l<=r no longer holds.

# Growth Rates and Dominance Relations

$n$	$f(n)$	$\lg n$	$n$	$n \lg n$	$n^2$	$2^n$	$n!$
10	0.003 $\mu$ s	0.01 $\mu$ s	0.033 $\mu$ s	0.1 $\mu$ s	1 $\mu$ s	3.63 ms	
20	0.004 $\mu$ s	0.02 $\mu$ s	0.086 $\mu$ s	0.4 $\mu$ s	1 ms	77.1 years	
30	0.005 $\mu$ s	0.03 $\mu$ s	0.147 $\mu$ s	0.9 $\mu$ s	1 sec	8.4 $\times 10^{15}$ yrs	
40	0.005 $\mu$ s	0.04 $\mu$ s	0.213 $\mu$ s	1.6 $\mu$ s	18.3 min		
50	0.006 $\mu$ s	0.05 $\mu$ s	0.282 $\mu$ s	2.5 $\mu$ s	13 days		
100	0.007 $\mu$ s	0.1 $\mu$ s	0.644 $\mu$ s	10 $\mu$ s	4 $\times 10^{13}$ yrs		
1,000	0.010 $\mu$ s	1.00 $\mu$ s	9.966 $\mu$ s	1 ms			
10,000	0.013 $\mu$ s	10 $\mu$ s	130 $\mu$ s	100 ms			
100,000	0.017 $\mu$ s	0.10 ms	1.67 ms	10 sec			
1,000,000	0.020 $\mu$ s	1 ms	19.93 ms	16.7 min			
10,000,000	0.023 $\mu$ s	0.01 sec	0.23 sec	1.16 days			
100,000,000	0.027 $\mu$ s	0.10 sec	2.66 sec	115.7 days			
1,000,000,000	0.030 $\mu$ s	1 sec	29.90 sec	31.7 years			

Growth rates of common functions measured in nanoseconds

$$n! \gg 2^n \gg n^3 \gg n^2 \gg n \log n \gg n \gg \log n \gg 1$$

# Proof by Induction

- We want to show that property  $P$  is true for all integers  $n \geq n_0$ .
- **Basis:** prove that  $P$  is true for  $n_0$ .
- **Inductive step:** prove that if  $P$  is true for all  $k$  such that  $n_0 \leq k \leq n - 1$  then  $P$  is also true for  $n$ .
- Example
- Basis

$$S(n) = \sum_{i=0}^n i = \frac{n(n+1)}{2} \text{ for } n \geq 1$$

$$S(1) = \sum_{i=0}^1 i = \frac{1(1+1)}{2}$$

# ...Proof by Induction

- Inductive Step

$$S(k) = \sum_{i=0}^k i = \frac{k(k+1)}{2} \text{ for } 1 \leq k \leq n-1$$

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

# Sorting

- Sorting is a classical and important algorithmic problem.
- We look at sorting arrays (in contrast to files, which restrict random access).
- A key constraint is the efficient management of the space
  - In-place sorting algorithms
- The efficiency comparison is based on the number of comparisons (C) and the number of movements (M).

# Sorting

- Simple sorting methods use roughly  $n * n$  comparisons
  - Insertion sort
  - Selection sort
  - Bubble sort
- Fast sorting methods use roughly  $n * \log n$  comparisons.
  - Merge sort
  - Heap sort
  - Quicksort

# References & Readings

- CLRS
  - Chapters: 1, 2 (2.1, 2.2), 3
  - Exercises: 1.2-2, 1.2-3, 2.1-3, 2.1-4, 2.2-1, 2.2-3, 3.1-1, 3.1-4, 3.1-6, 3.1
  - Problems: 1-1, 3-3
- HSR
  - Chapters: 1 (1.1-1.3)
  - Examples: 1.4-1.6, 1.11-1.13, 1.17-1.18
  - Exercises: 1.3 (1-4, 8, 9)
- Review for laboratory
  - HSR
    - ◆ Chapters: 2, 3.2 - 3.5
  - CLRS
    - ◆ Chapters: 6, 7, 10, 12

# Complexity of conventional Sorting Algorithms

Course Code: CSC2211

Course Title: Algorithms



**Dept. of Computer Science  
Faculty of Science and Technology**

<b>Lecturer No:</b>		<b>Week No:</b>	<b>02</b>	<b>Semester:</b>	
<b>Lecturer:</b>	<i>Name &amp; email</i>				



# Lecture Outline

## 1. Sorting Algorithms

- ✓ Insertion Sort
- ✓ Selection Sort
- ✓ Bubble Sort
- ✓ Merge Sort
- ✓ Quick Sort
- ✓ Counting Sort



## Sorting

- ↗ Simple sorting methods use roughly  $n * n$  comparisons
  - ↗ Insertion sort
  - ↗ Selection sort
  - ↗ Bubble sort
- ↗ Fast sorting methods use roughly  $n * \log n$  comparisons.
  - ↗ Merge sort
  - ↗ Quicksort

**Fastest sorting methods use roughly n  
COUNTING SORT ??**



# Sorting Algorithms with Polynomial time



## Insertion Sorting

Mark first element as sorted,  
Next for each unsorted element  
'extract' the element

```
for i = last Sorted Index to 0
    if current SortedElement > extracted Element
        move/shift sorted element to the right by 1
    else: insert extracted element
```



## Insertion Sorting

- To sort array  $A[0..n-1]$ , sort  $A[0..n-2]$  recursively and then insert  $A[n-1]$  in its proper place among the sorted  $A[0..n-2]$
- Usually implemented bottom up (non-recursively)

Example: 6, 4, 1, 8, 5

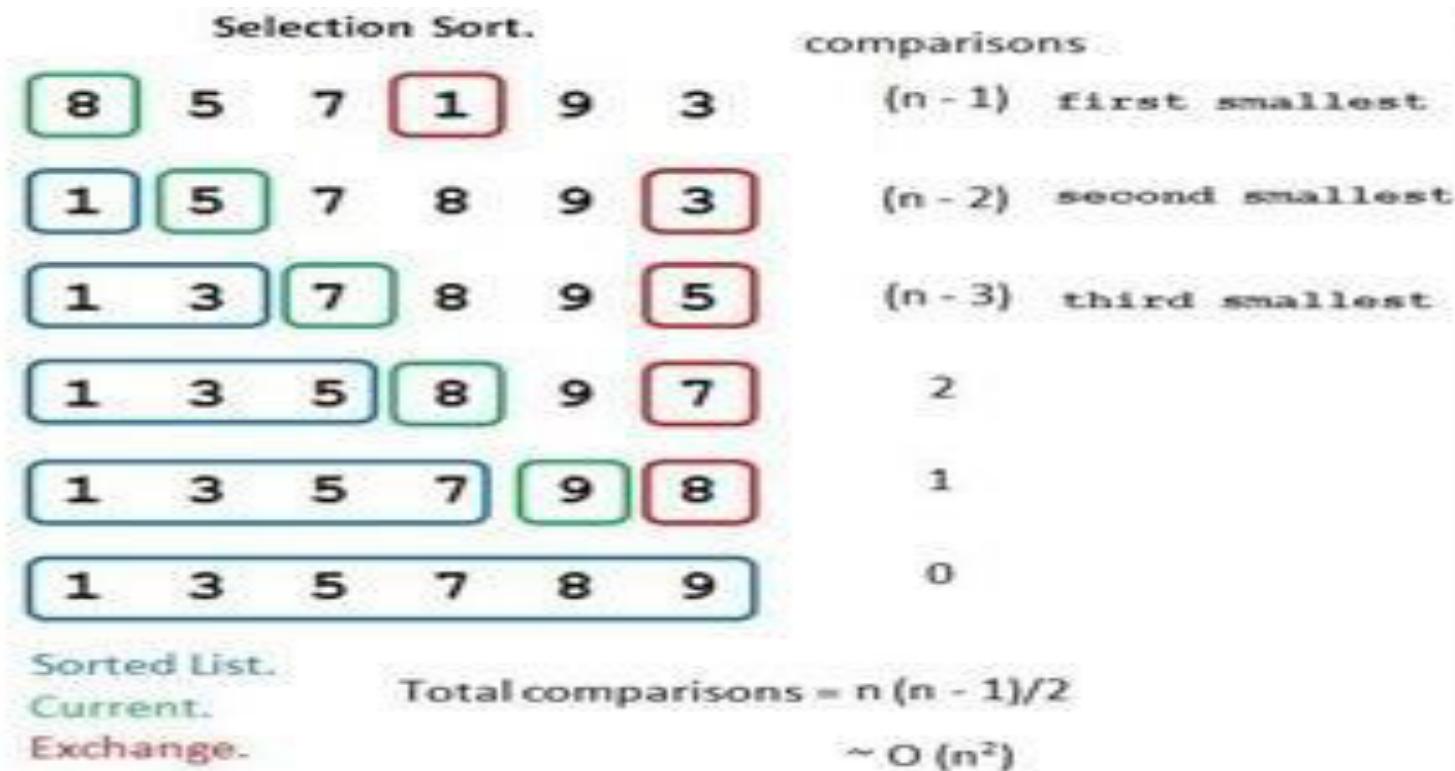
6		4	1	8	5
4	6		1	8	5
1	4	6		8	5
1	4	6	8		5
1	4	5	6	8	

# Selection Sorting

- Concept for sort in ascending order:

- Locate **smallest element** in array. **Exchange** it with element in **position 0**
- Locate **next smallest** element in array. Exchange it with element in **position 1**.
- Continue until all elements are arranged in order

Min value	Min Index
8	0
5	1
1	3
5	1
3	5
7	2
5	5
8	3
7	5
9	4





# Selection Sort Algorithm

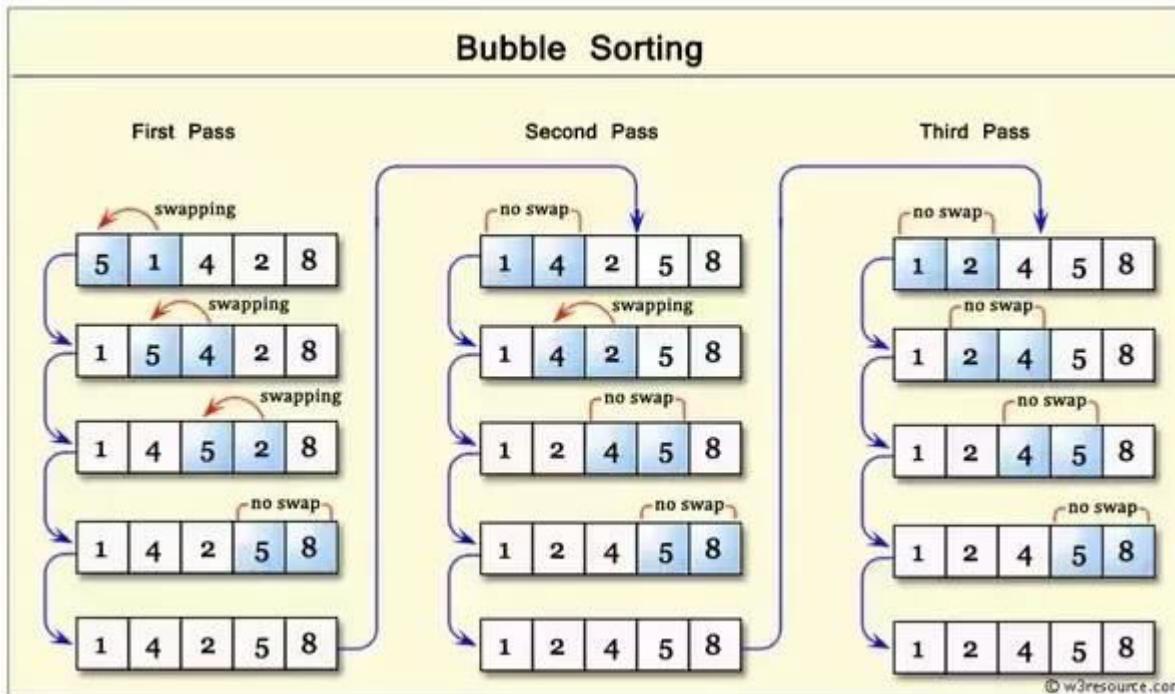
```
void selectionSort(int array[], int n)
{
    int select, minIndex, minValue;

    for (select = 0; select < (n - 1); select++)
    { //select the location and find the minimum value
        minIndex = select;
        minValue = array[select];
        for(int i = select + 1; i < n; i++)
        { //start from the next of selected one to find minimum
            if (array[i] < minValue)
            {
                minValue = array[i];
                minIndex = i;
            }
        }
        array[minIndex] = array[select];
        array[select] = minValue;
    }
}
```

# Bubble Sorting

## Concept:

- Compare 1<sup>st</sup> two elements
  - If out of order, **exchange** them to put in order
- Move down one element, compare 2<sup>nd</sup> and 3<sup>rd</sup> elements, **exchange if necessary**. Continue until end of array.
- Pass through array again, exchanging as necessary
- Repeat until pass made with **no exchanges**.





## Bubble Sort Algorithm

```
int i, j, temp;  
bool swapped;  
for(i=0;i<n-1;i++) {  
    swapped = false;  
    for(j=0;j<(n-i-1) ;j++) {  
        if( a[j]>a[j+1] ) {  
            temp = a[j];  
            a[j] = a[j+1];  
            a[j+1] = temp;  
            swapped = true; }  
    }  
    if ( !swapped) { break; }  
}
```



# Sorting Algorithms with $n(\log n)$ time

# Divide and Conquer

## ↗ Recursive in structure

- ↗ **Divide** the problem into independent sub-problems that are similar to the original but smaller in size
- ↗ **Conquer** the sub-problems by solving them **recursively**. If they are small enough, just solve them in a straightforward manner.

This can be done by reducing the problem until it reaches the base case, which is the solution.

- ↗ **Combine** the solutions of the sub-problems to create a solution to the original problem



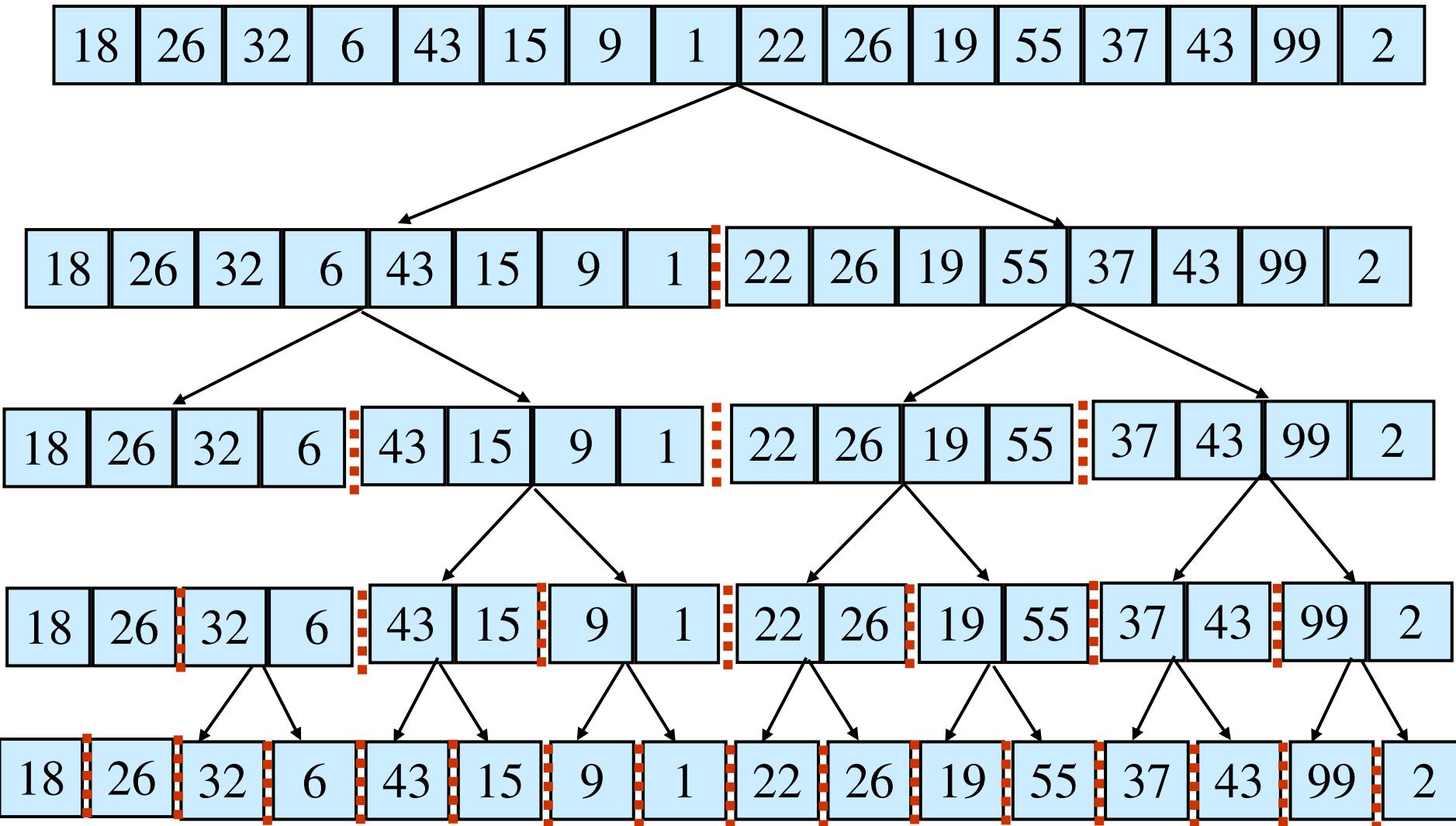
## Merge Sort

***Sorting Problem:*** Sort a sequence of  $n$  elements into non-decreasing order.

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



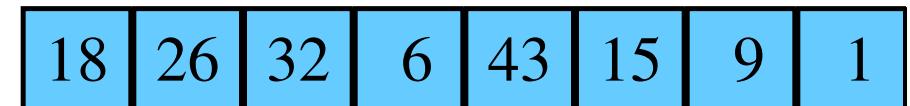
## Merge Sort Example



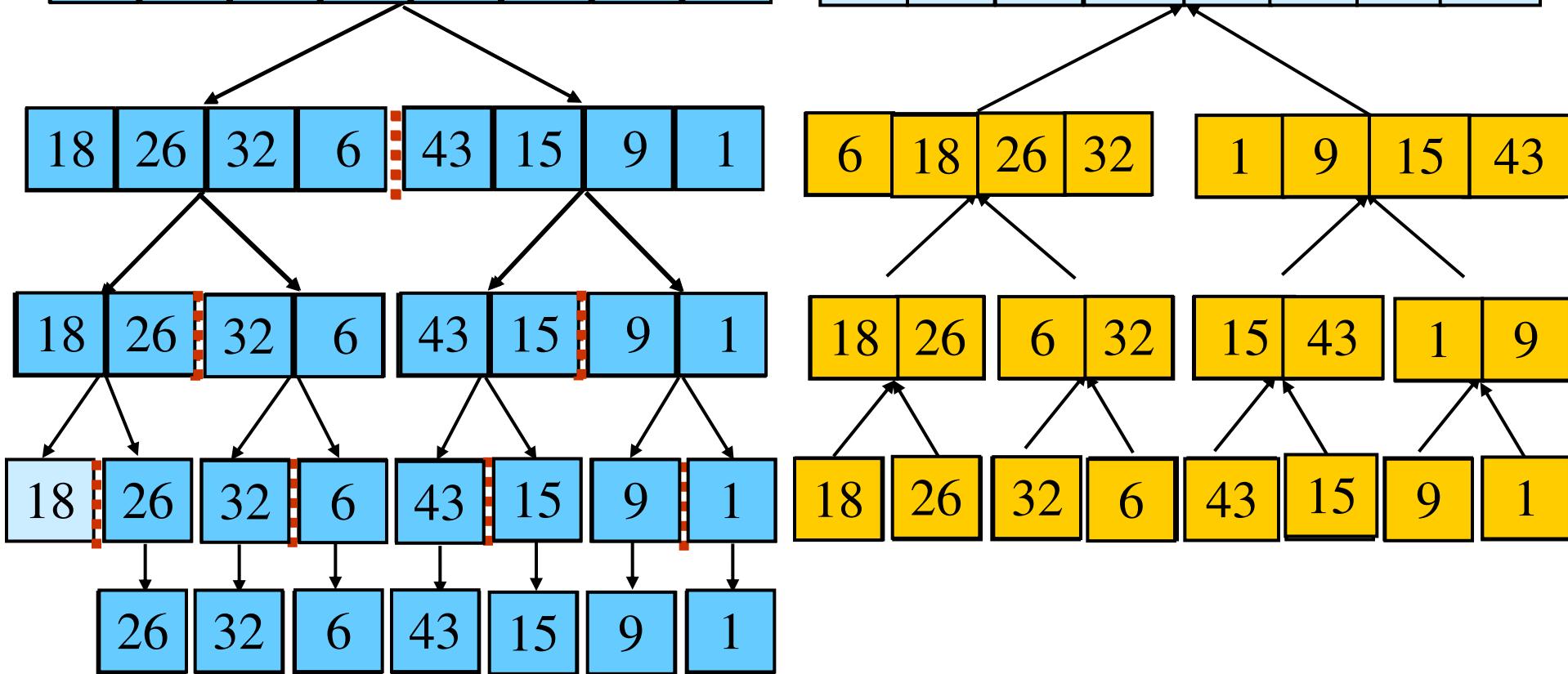
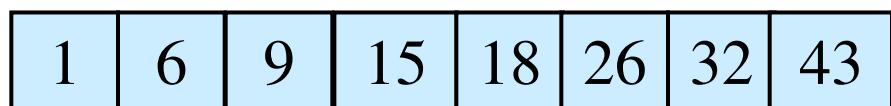


## Merge Sort Example

Original Sequence



Sorted Sequence





## Exercise

98	23	45	14	6	67	33	42
----	----	----	----	---	----	----	----



## Merge Sort Algorithm

**Step 1 – if it is only one element in the list it is already sorted, return.**

**Step 2 – divide the list recursively into two halves until it can no more be divided.**

**Step 3 – merge the smaller lists into new list in sorted order.**



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

```
1  if  $p < r$ 
2     $q = \lfloor (p + r)/2 \rfloor$ 
3    MERGE-SORT( $A, p, q$ )
4    MERGE-SORT( $A, q + 1, r$ )
5    MERGE( $A, p, q, r$ )
```

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

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



## Merge Sort Analysis

Statement

Cost

<i>MergeSort (A, p, r)</i>	$T(n)$ , to sort n elements
1 <i>if</i> $p < r$	$O(1)$
2 <i>then</i> $q \leftarrow \lfloor (p+r)/2 \rfloor$	$O(1)$
3 <i>MergeSort (A, p, q)</i>	$T(n/2)$ , to sort n/2 elements
4 <i>MergeSort (A, q+1, r)</i>	$T(n/2)$ , to sort n/2 elements
5 <i>Merge (A, p, q, r)</i>	$O(n)$

- ↗ So  $T(n) = O(1)$  when  $n = 1$ , and  
 $2T(n/2) + O(n)$  when  $n > 1$



## Merge Sort Analysis

1. The divide step takes constant time, regardless of the subarray size. After all, the divide step just computes the midpoint  $q$  of the indices  $p$  and  $r$ . Recall that in big- $O$  notation, we indicate constant time by  $O(1)$ .
2. The conquer step, where we recursively sort two subarrays of approximately  $n/2$  elements each, takes some amount of time, but we'll account for that time when we consider the subproblems.
3. The combine step merges a total of  $n$  elements, taking  $O(n)$  time.



# Merge Sort Analysis

Elements to sort / Subproblem size

$n$

Total merging time  
for all subproblems of  
this size

$Cn$



## Recursion-tree Method

### ↗ Recursion Trees

- Show successive expansions of recurrences using trees.
- Keep track of the time spent on the subproblems of a divide and conquer algorithm.
- Help organize the algebraic bookkeeping necessary to solve a recurrence.

↗ Running time of Merge Sort:  $T(n) = O(1)$  if  $n = 1$

$$T(n) = 2T(n/2) + O(n) \quad \text{if } n > 1$$

↗ Rewrite the recurrence as  $T(n) = c$  if  $n = 1$

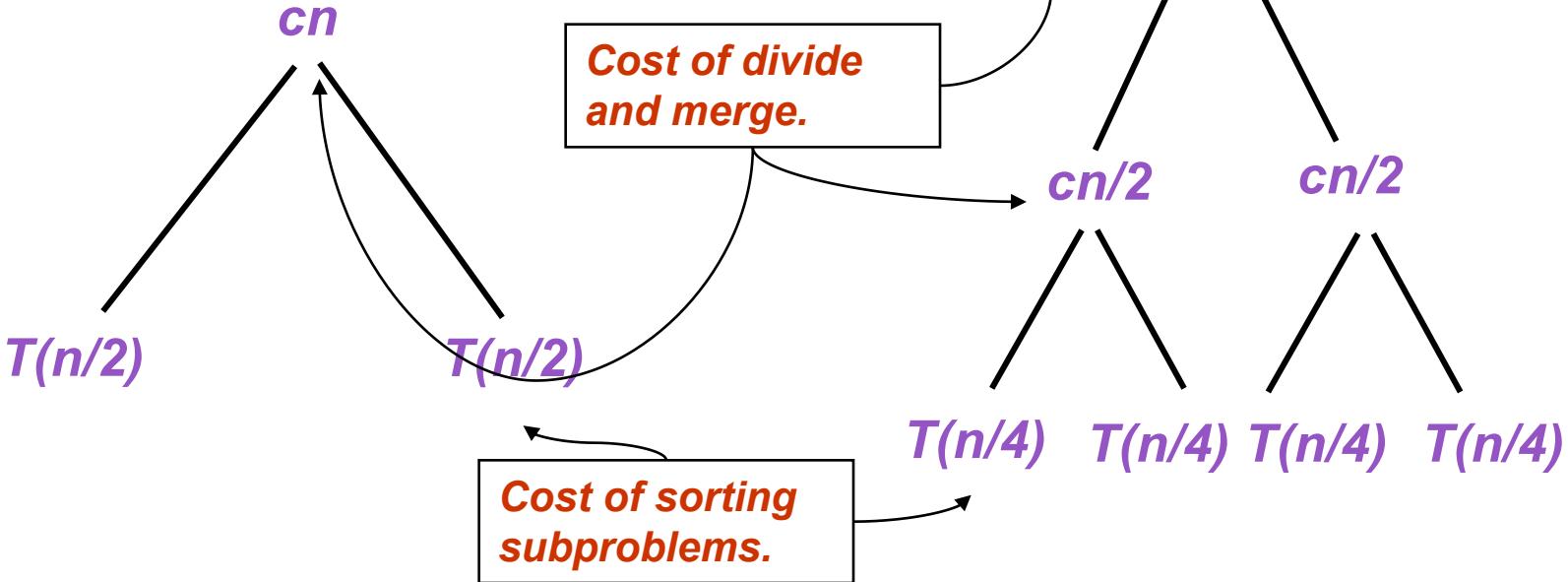
$$T(n) = 2T(n/2) + cn \quad \text{if } n > 1$$

$c > 0$ : Running time for the base case and

time per array element for the divide and combine steps.

# Recursion Tree for Merge Sort

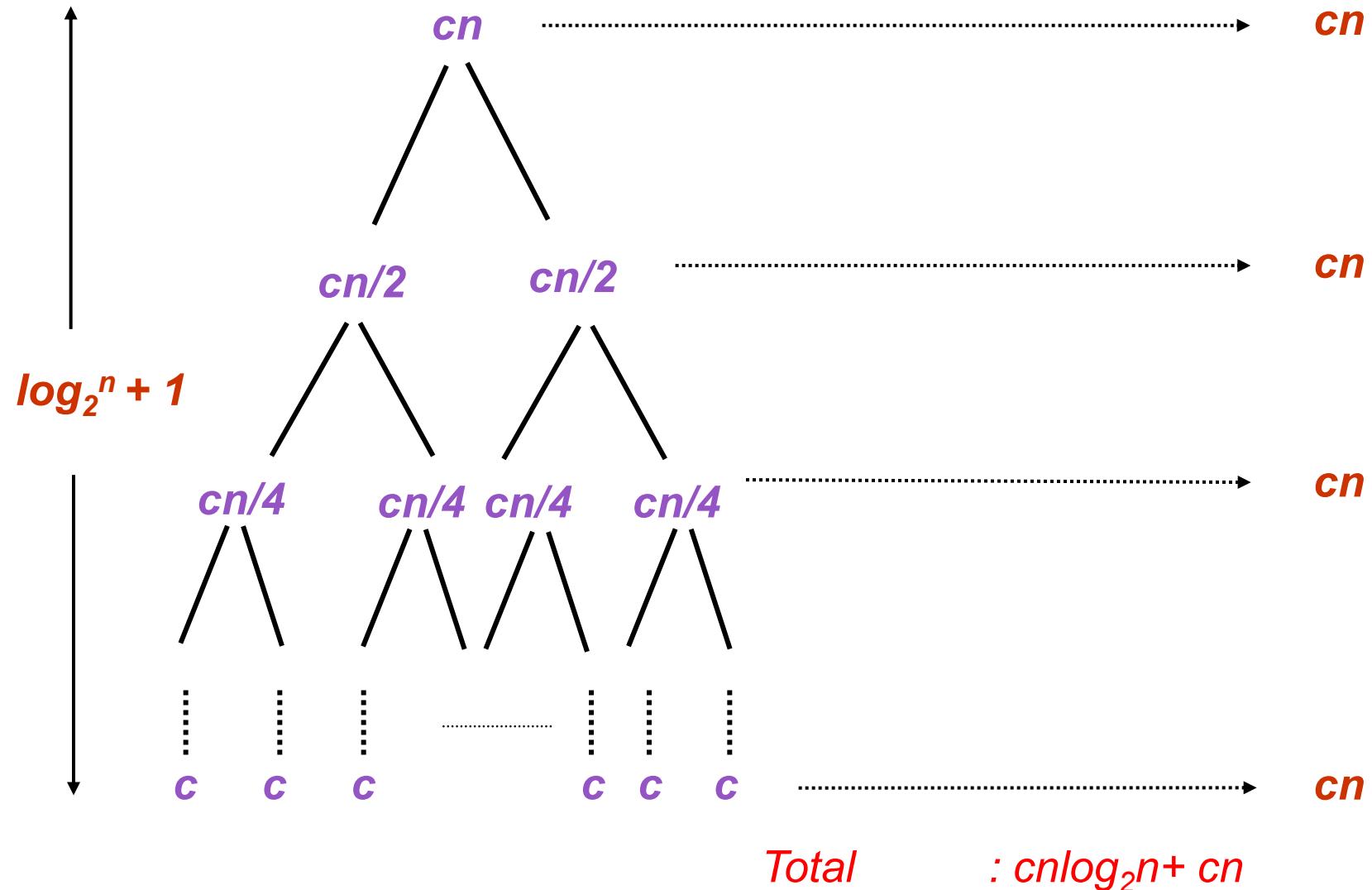
For the original problem, we have a cost of  $cn$ , plus two subproblems each of size  $(n/2)$  and running time  $T(n/2)$ .



Each of the size  $n/2$  problems has a cost of  $cn/2$  plus two subproblems, each costing  $T(n/4)$ .

## Recursion Tree for Merge Sort

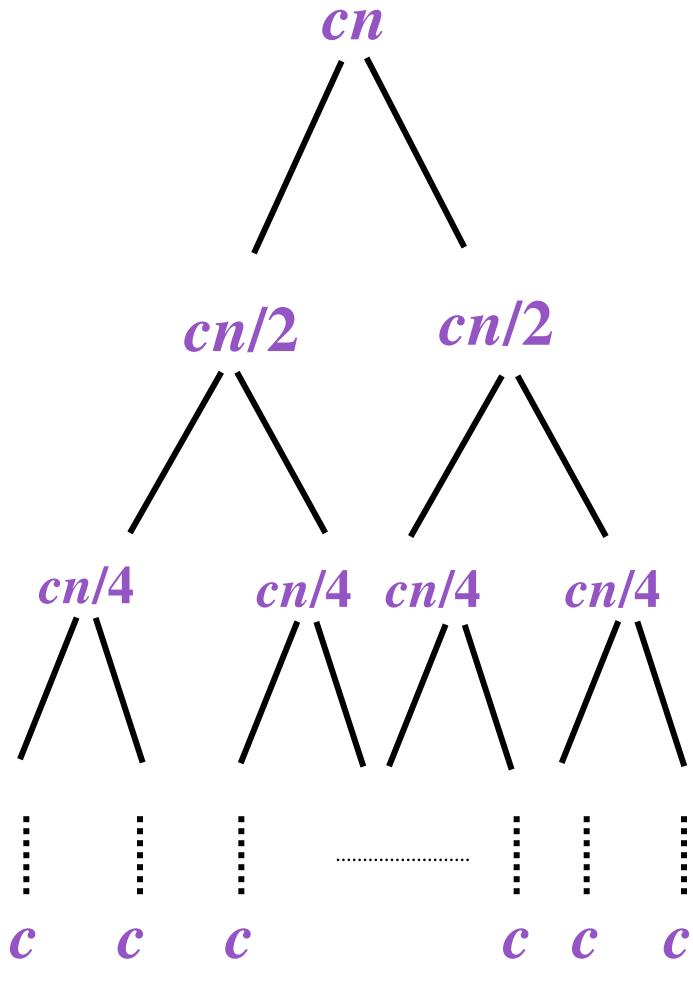
Continue expanding until the problem size reduces to 1.





## Recursion Tree for Merge Sort

Continue expanding until the problem size reduces to 1.



- Each level has total cost  $cn$ .
- Each time we go down one level, the number of subproblems doubles, but the cost per subproblem halves  
 $\Rightarrow$  *cost per level remains the same*.
- There are  $\lg n + 1$  levels, height is  $\lg n$ .
- Total cost = sum of costs at each level =  $(\lg n + 1)cn = cn \lg n + cn = O(n \lg n)$ .



## Analysis: solving recurrence

- All cases have same efficiency:  $O(n \log n)$

$$T(n) = 2T(n/2) + O(n), \text{ and } O(n)=cn$$

$$\begin{aligned} T(n) &= 2T\left(\frac{n}{2}\right) + cn \\ &= 2\left(2T\left(\frac{n}{4}\right) + \frac{cn}{2}\right) + cn \\ &= 4T\left(\frac{n}{4}\right) + 2cn \\ &= 4\left(2T\left(\frac{n}{8}\right) + \frac{cn}{4}\right) + 2cn \\ &= 8T\left(\frac{n}{8}\right) + 3cn \\ &= 2^k T\left(\frac{n}{2^k}\right) + kn \end{aligned}$$

Since  $n=2^k$ , we have  $k=\log_2 n$

$$\begin{aligned} T(n) &= 2^k T\left(\frac{n}{2^k}\right) + kn \\ &= n + n \log n \\ &= O(n \log n) \end{aligned}$$



## Quick Sort

- Quick sort is based on the divide-and-conquer approach.
- The idea is based on choosing one element as a pivot element and partitioning the array around it such that:
  - Left side of pivot contains all the elements that are less than the pivot element.
  - Right side contains all elements greater than the pivot.
- It **reduces the space complexity** and **removes the use of the auxiliary array** that is used in merge sort.
- Selecting a random pivot in an array results in an improved time complexity in most of the cases.

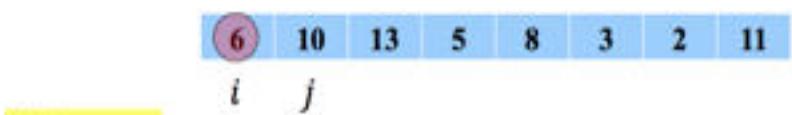


## Quick Sort Algorithm

```
Quick_sort ( A[ ] , start , end ) {  
    if( start < end ) {  
        piv_pos = Partition (A,start , end ) ;  
        Quick_sort (A, start , piv_pos -1);  
        Quick_sort ( A, piv_pos +1 , end) ; } }
```

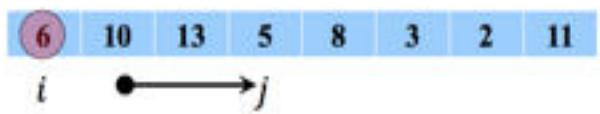
```
Partition ( A[], start , end) {  
    i = start + 1;  
    piv = A[start] ;  
    for( j =start + 1; j <= end ; j++ ) {  
        if ( A[ j ] < piv) {  
            swap (A[ i ],A [ j ]);  
            i += 1;  
        }  
    }  
    swap ( A[ start ] ,A[ i-1 ] );  
    return i-1;}
```

# Quick Sort Example

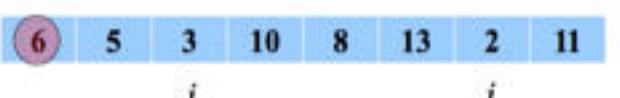
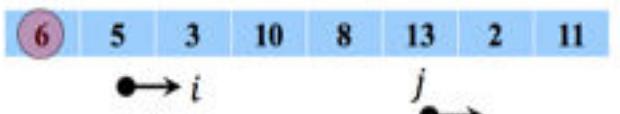


Start:  
pivot:  $x = 6$   
 $i = 1$   
 $j = 2$

Keep moving  $j$  until element is < pivot



Swap 10 & 5



Swap 10 & 2

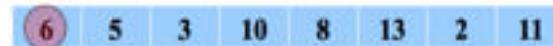


Increment  $i$

Keep moving  $j$  until element is < pivot



Swap 13 & 3



Loop terminates

Exchange pivot (first) with  $i^{\text{th}}$  location



Loop terminates, return position of pivot





# Sorting Algorithms with linear time



## Counting Sort

**Counting sort:** No comparisons between elements.

**Input:**  $A[1 \dots n]$ , where  $A[j] \in \{1, 2, \dots, k\}$ .

**Output:**  $B[1 \dots n]$ , sorted.

**Auxiliary storage:**  $C[1 \dots k]$ .

```
1  for i ← 1 to k
2      do C[i] ← 0
3  for j ← 1 to n
4      do C[A[j]] ← C[A[j]] + 1
5  for i ← 2 to k
6      do C[i] ← C[i] + C[i - 1]
7  for j ← n downto 1
8      do B[C[A[j]]] ← A[j]
9      C[A[j]] ← C[A[j]] - 1
```



## Counting sort example

1    2    3    4    5

A:	4	1	3	4	3
----	---	---	---	---	---

1    2    3    4

C:				
----	--	--	--	--

B:					
----	--	--	--	--	--



## Counting sort example Loop 1

1    2    3    4    5

A:

4	1	3	4	3
---	---	---	---	---

1    2    3    4

C:

0	0	0	0
---	---	---	---

B:

--	--	--	--	--

```
for i ← 1 to k  
    do C [i] ← 0
```



## Counting sort example Loop 2

	1	2	3	4	5
A:	4	1	3	4	3

	1	2	3	4
C:	0	0	0	1

B:					

```
for  $j \leftarrow 1$  to  $n$ 
    do  $C[A[j]] \leftarrow C[A[j]] + 1$ 
```



## Counting sort example Loop 2

	1	2	3	4	5
A:	4	1	3	4	3

	1	2	3	4
C:	1	0	0	1

B:					
----	--	--	--	--	--

```
for j ← 1 to n
    do C [A[j]] ← C [A[j]] + 1
```



## Counting sort example Loop 2

	1	2	3	4	5
A:	4	1	3	4	3

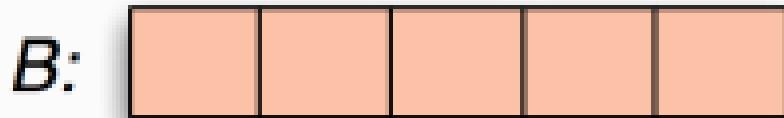
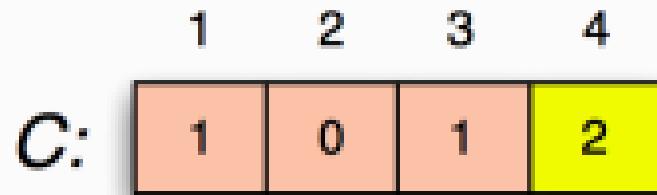
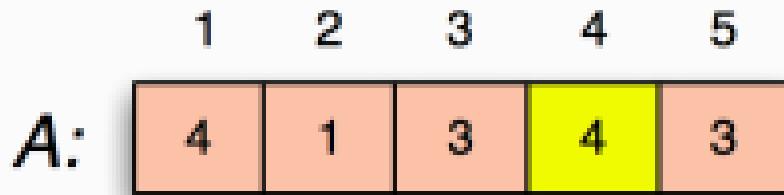
	1	2	3	4
C:	1	0	1	1

B:					
----	--	--	--	--	--

```
for j ← 1 to n
    do C [A[j]] ← C [A[j]] + 1
```



## Counting sort example Loop 2



```
for  $j \leftarrow 1$  to  $n$ 
    do  $C[A[j]] \leftarrow C[A[j]] + 1$ 
```



## Counting sort example Loop 2

1    2    3    4    5

A:	4	1	3	4	3
----	---	---	---	---	---

1    2    3    4

C:	1	0	2	2
----	---	---	---	---

B:					
----	--	--	--	--	--

```
for j ← 1 to n
    do C[A[j]] ← C[A[j]] + 1
```



## Counting sort example Loop 3

	1	2	3	4	5
A:	4	1	3	4	3

	1	2	3	4
C:	1	0	2	2

B:					
----	--	--	--	--	--

C':	1	1	2	2
-----	---	---	---	---

```
for i ← 2 to k
    do C [i] ← C [i] + C [i - 1]
```



## Counting sort example Loop 3

1	2	3	4	5
4	1	3	4	3

--	--	--	--	--

1	2	3	4
1	0	2	2

1	1	3	2
---	---	---	---

```
for i ← 2 to k  
    do C [i] ← C [i] + C [i -1]
```



## Counting sort example Loop 3

1    2    3    4    5

A:	4	1	3	4	3
----	---	---	---	---	---

1    2    3    4

C:	1	0	2	2
----	---	---	---	---

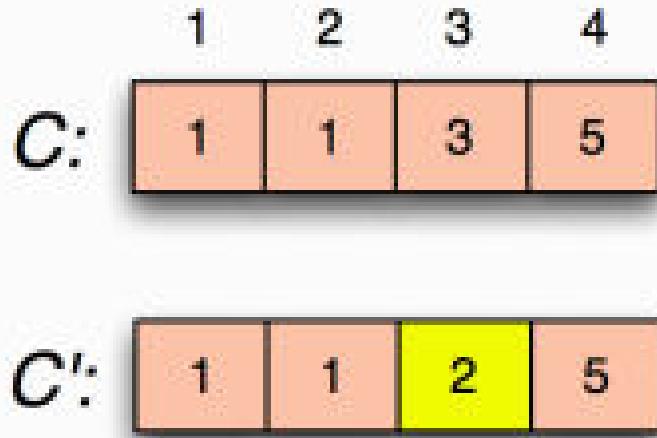
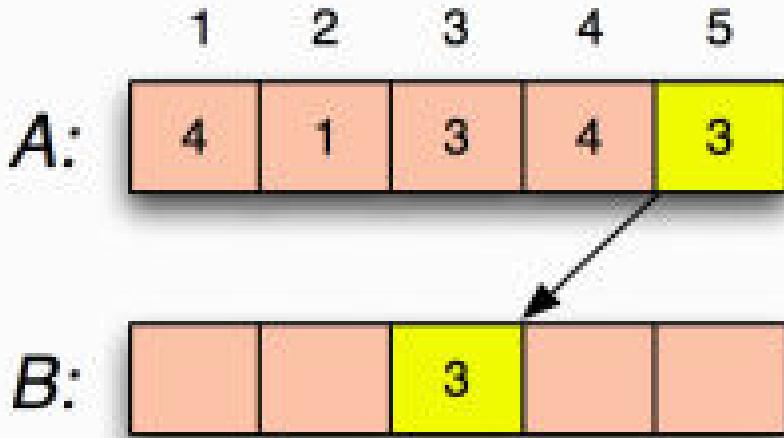
B:					
----	--	--	--	--	--

C':	1	1	3	5
-----	---	---	---	---

```
for i ← 2 to k  
    do C [i] ← C [i] + C [i -1]
```



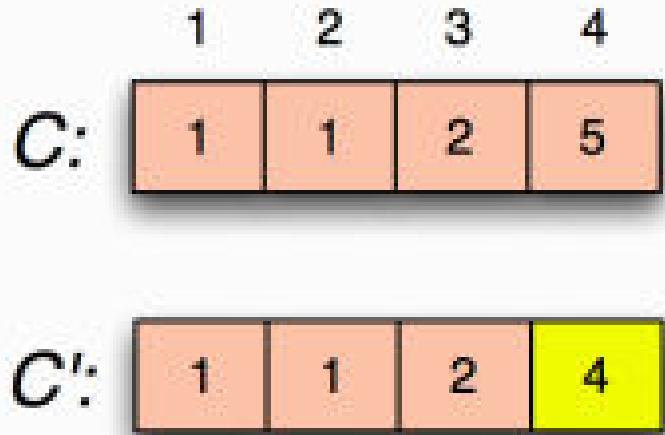
## Counting sort example loop 4



```
for  $j \leftarrow n$  downto 1  
    do  $B[C[A[j]]] \leftarrow A[j]$   
         $C[A[j]] \leftarrow C[A[j]] - 1$ 
```



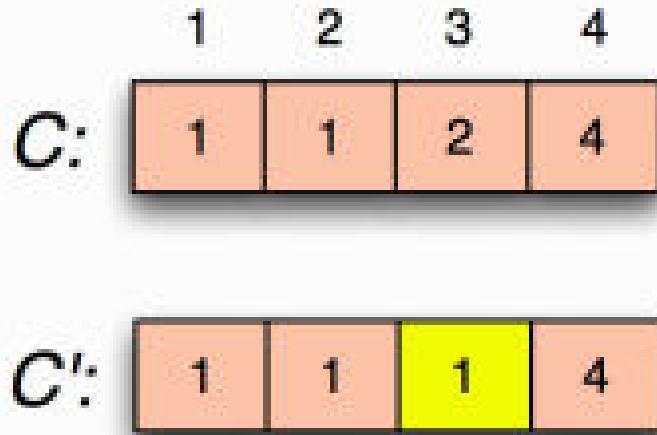
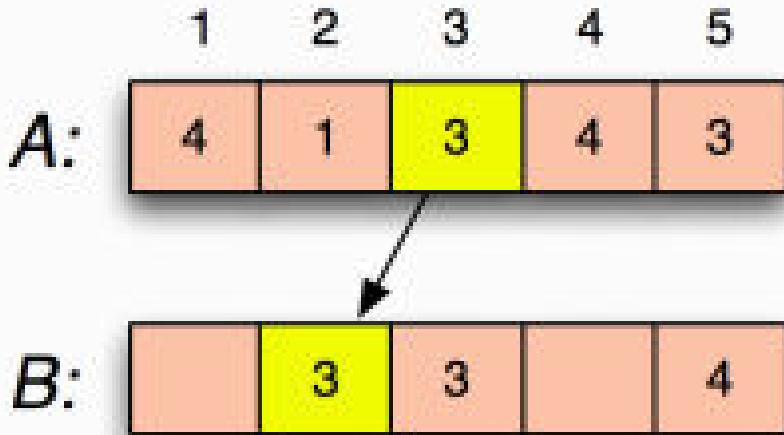
## Counting sort example Loop 4



```
for j ← n downto 1
    do B[C[A[j]]] ← A[j]
        C[A[j]] ← C[A[j]] - 1
```



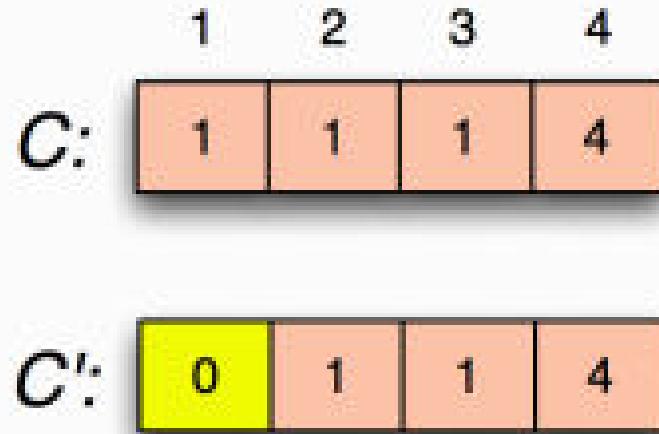
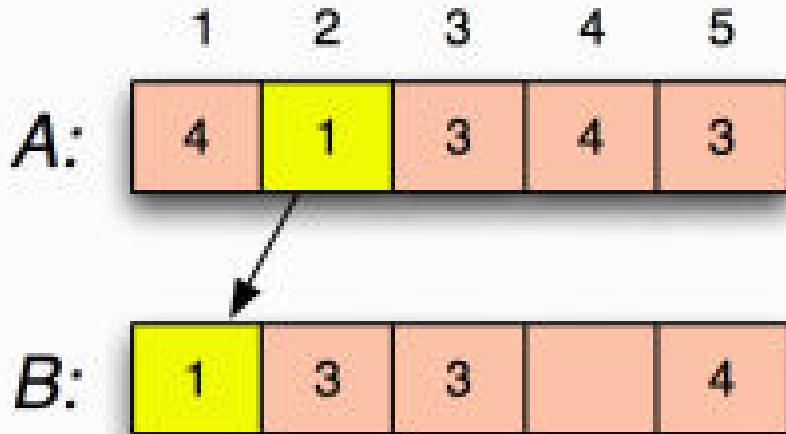
## Counting sort example Loop 4



```
for j ← n downto 1
    do B[C[A[j]]] ← A[j]
        C[A[j]] ← C[A[j]] - 1
```



## Counting sort example Loop 4



```
for  $j \leftarrow n$  downto 1
    do  $B[C[A[j]]] \leftarrow A[j]$ 
         $C[A[j]] \leftarrow C[A[j]] - 1$ 
```



## Counting sort example Loop 4

	1	2	3	4	5
A:	4	1	3	4	3

B:	1	3	3	4	4
----	---	---	---	---	---

	1	2	3	4
C:	0	1	1	4

C':	0	1	1	3
-----	---	---	---	---

```
for j ← n downto 1
    do B[C[A[j]]] ← A[j]
        C[A[j]] ← C[A[j]] - 1
```



## Counting sort Complexity

```
O(k) {   for i ← 1 to k  
           do C[i] ← 0  
  
O(n) {   for j ← 1 to n  
           do C[A[j]] ← C[A[j]] + 1  
  
O(k) {   for i ← 2 to k  
           do C[i] ← C[i] + C[i - 1]  
  
O(n) {   for j ← n downto 1  
           do B[C[A[j]]] ← A[j]  
           C[A[j]] ← C[A[j]] - 1
```

---

$$O(n + k)$$

The worst-case running time of Counting sort is  $O(n + k)$ .

If  $k = O(n)$ , then the worst case running time is  $O(n)$ .



# Books

***Introduction to Algorithms, Thomas H. Cormen, Charle E. Leiserson, Ronald L. Rivest, Clifford Stein (CLRS).***

***Fundamental of Computer Algorithms, Ellis Horowitz, Sartaj Sahni, Sanguthevar Rajasekaran (HSR)***



# References

[https://www.google.com/search?q=bubble+sort+step+by+step&sxsrf=ALeKk01uxzgfT3Oy6k1Q3WxVnSpiIN8\\_4g:1587999728942&tbm=isch&source=iu&ictx=1&fir=vRwFsGwVfJ6pJM%253A%252CSzhze6MPQr4cM%252C&vet=1&usg=AI4\\_kSrEEXqwRL-PkHhVUtn7jNfF9dB6g&sa=X&ved=2ahUKEwjeOPz974jpAhXRAnIKHWhMD2UQ\\_h0wAXoECAcQBg#imgrc=EN4Sdu7veOWVoM&imgdii=eOqvCu85p9-eBM](https://www.google.com/search?q=bubble+sort+step+by+step&sxsrf=ALeKk01uxzgfT3Oy6k1Q3WxVnSpiIN8_4g:1587999728942&tbm=isch&source=iu&ictx=1&fir=vRwFsGwVfJ6pJM%253A%252CSzhze6MPQr4cM%252C&vet=1&usg=AI4_kSrEEXqwRL-PkHhVUtn7jNfF9dB6g&sa=X&ved=2ahUKEwjeOPz974jpAhXRAnIKHWhMD2UQ_h0wAXoECAcQBg#imgrc=EN4Sdu7veOWVoM&imgdii=eOqvCu85p9-eBM)

<https://www.interviewcake.com/concept/java/counting-sort>

<https://www.geeksforgeeks.org/counting-sort/>

<https://www.hackerearth.com/practice/algorithms/sorting/quick-sort/tutorial/>

# Lecture Title: Recursion



**Dept. of Computer Science  
Faculty of Science and Technology**

<b>Lecture No:</b>	<b>03</b>	<b>Week No:</b>	<b>03</b>	<b>Semester:</b>	
<b>Lecturer:</b>	<i>Name &amp; email: Md. Manzurul Hasan, manzurul@aiub.edu</i>				



# Recurrences & Master Method

Md. MANZURUL hASAN

Slides are adopted from Mashior Rahman [Asso Dean, FsT]



# Lecture Outline

- 1. Divide and Conquer**
- 2. Recurrences in Divide and Conquer and methodologies for recurrence Solutions**
- 3. Repeated Backward Substitution Method**
- 4. Substitution Method**
- 5. Recursion Tree (Next Week)**
- 6. Master Method (Next Week)**

# The divide- and- conquer design paradigm

- Divide and conquer is just one of several powerful techniques for algorithm design.
- Divide- and- conquer algorithms can be analyzed using recurrences and the master method (so practice this math).
- Can lead to more efficient algorithms

# The divide- and- conquer design paradigm

1. ***Divide*** the problem (instance) into subproblems.
2. ***Conquer*** the subproblems by solving them recursively.
3. ***Combine*** subproblem solutions.

# The divide- and- conquer design paradigm

Example: merge sort

1. **Divide:** Trivial (array is halved).
2. **Conquer:** Recursively sort 2 subarrays.
3. **Combine:** Linear- time merge.

Recurrence for merge sort

Let  $T(n)$  = Time required for size  $n$

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

Sub-  
problem  
Size ( $n/2$ )

Number Of  
Subproblems

\* Time required for  
each Subproblem

+ Work Dividing  
and Combining

# The divide- and- conquer design paradigm

## Binary search

Find an element in a sorted array:

- 1. *Divide*:** Check middle element.
- 2. *Conquer*:** Recursively search 1 subarray.
- 3. *Combine*:** Trivial.

## Recurrence for binary search

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

**Number Of  
Subproblems**

**\* Time required for  
each Subproblem**

**+ Work Dividing  
and Combining**

# Recurrences

- Running times of algorithms with **recursive calls** can be described using recurrences.
- A **recurrence** is an equation or inequality that describes a function in terms of its value on smaller inputs.
- For **divide and conquer** algorithms:

$$T(n) = \begin{cases} \text{solving\_trivial\_problem} & \text{if } n = 1 \\ \text{num\_pieces } T(n/\text{subproblem\_size\_factor}) + \text{dividing} + \text{combining} & \text{if } n > 1 \end{cases}$$

- Example: Merge Sort

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

# Solving Recurrences

- ↗ **Repeated (backward) substitution method**
  - ↗ Expanding the recurrence by substitution and noticing a pattern (this is not a strictly formal proof).
- ↗ **Substitution method**
  - ↗ guessing the solutions
  - ↗ verifying the solution by the mathematical induction
- ↗ **Recursion-trees**
- ↗ **Master method**
  - ↗ templates for different classes of recurrences

# Repeated Substitution



↗ Let's find the running time of the merge sort

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

$$\begin{aligned} T(n) &= 2T(n/2) + n \quad \underline{\text{substitute}} \\ &= 2(2T(n/4) + n/2) + n \quad \underline{\text{expand}} \\ &= 2^2T(n/4) + 2n \quad \underline{\text{substitute}} \\ &= 2^2(2T(n/8) + n/4) + 2n \quad \underline{\text{expand}} \\ &= 2^3T(n/8) + 3n \quad \underline{\text{observe pattern}} \\ &= 2^3T(n/2^3) + 3n \quad \underline{\text{observe pattern}} \end{aligned}$$

assume  $2^{k-1} < n \leq 2^k$ .

So upper bound for  $k$  is

$$n = 2^k$$

$$K = \log_2 n = \lg n$$

$$\begin{aligned} T(n) &= 2^kT(n/2^k) + kn \\ &= nT(n/n) + n\lg n \\ &= n + n\lg n \end{aligned}$$

$$T(n) = O(n\lg n)$$

# Repeated Substitution Method

- ▶ The procedure is straightforward:
- ▶ **Substitute, Expand, Substitute, Expand, ...**
- ▶ Observe a ***pattern*** and determine the expression after the  $k$ -th substitution.
- ▶ Find out what the highest value of  $k$  (number of iterations, e.g.,  $\lg n$ ) should be to get to the base case of the recurrence (e.g.,  $T(1)$ ).
- ▶ Insert the value of  $T(1)$  and the expression of  $k$  into your expression.

## Another Example...

$$T(n) = \begin{cases} 2 & \text{if } n = 1 \\ 2T(n/2) + 2n + 3 & \text{if } n > 1 \end{cases}$$

$$T(n) = 2T(n/2) + 2n + 3 \quad \underline{\text{substitute}}$$

$$= 2(2T(n/4) + n + 3) + 2n + 3 \quad \underline{\text{expand}}$$

$$= 2^2 T(n/4) + 4n + 2 \times 3 + 3 \quad \underline{\text{substitute}}$$

$$= 2^2(2T(n/8) + n/2 + 3) + 4n + 2 \times 3 + 3 \quad \underline{\text{expand}}$$

$$= 2^3 T(n/2^3) + 2 \times 3n + 3 \times (2^2 + 2^1 + 2^0) \quad \underline{\text{observe pattern}}$$

$$T(n) = 2^k T(n/2^k) + 2kn + 3 \sum_{j=0}^{k-1} 2^j$$

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

$$= nT(n/n) + 2n \lg n + 3(n - 1)$$

$$= 2n + 2n \lg n + 3n - 3$$

$$= 5n + 2n \lg n - 3$$

assume  $2^{k-1} < n \leq 2^k$ .

So upper bound for  $k$  is

$$n = 2^k$$

$$K = \log_2 n = \lg n$$

$$T(n) = O(n \lg n)$$

# Substitution Method

- The substitution method to solve recurrences entails two steps:
  - Guess the solution.
  - Use **induction** to prove the solution.
- Example:
  - $T(n) = 4T(n/2) + n$

## ...Substitution Method

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

- 1) Guess  $T(n) = O(n^3)$ , i.e.,  $T(n)$  is of the form  $cn^3$
- 2) Prove  $T(n) \leq cn^3$  by induction

$$\begin{aligned} T(n) &= 4T(n/2) + n && \text{recurrence} \\ &\leq 4c(n/2)^3 + n && \text{induction hypothesis} \\ &= 0.5cn^3 + n && \text{simplify} \\ &= cn^3 - (0.5cn^3 - n) && \text{rearrange} \\ &\leq cn^3 \text{ if } c \geq 2 \text{ and } n \geq 1 \end{aligned}$$

Thus  $T(n) = O(n^3)$

## ...Substitution Method

↗ Tighter bound for  $T(n) = 4T(n/2) + n$ :

Try to show  $T(n) = O(n^2)$

Prove  $T(n) \leq cn^2$  by induction

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

$$\leq 4c(n/2)^2 + n$$

$$= cn^2 + n$$

$$\leq cn^2$$

## ...Substitution Method

- What is the problem? Rewriting

$$T(n) = O(n^2) = cn^2 - (\text{something positive})$$

- As  $T(n) \leq cn^2$  does not work with the inductive proof.
- Solution: **Strengthen the hypothesis** for the inductive proof:
  - $T(n) \leq (\text{answer you want}) - (\text{something} > 0)$

## ...Substitution Method

- ↗ Fixed proof: strengthen the inductive hypothesis by subtracting lower-order terms:

Prove  $T(n) \leq cn^2 - dn$  by induction

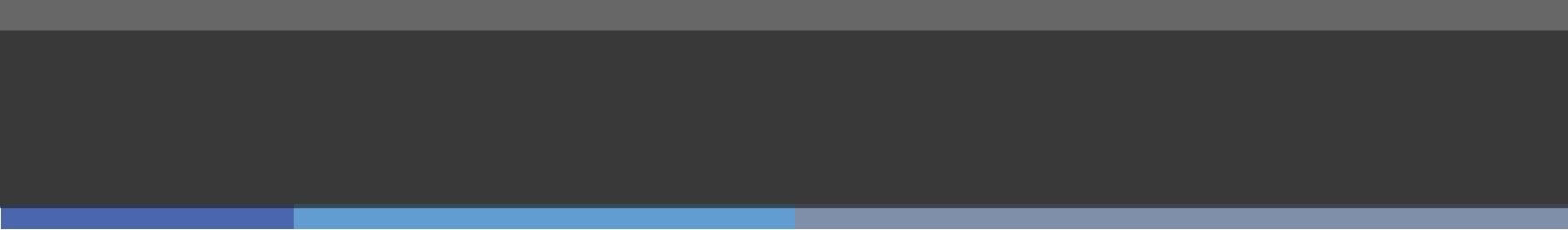
$$T(n) = 4T(n/2) + n$$

$$\leq 4(c(n/2)^2 - d(n/2)) + n$$

$$= cn^2 - 2dn + n$$

$$= (cn^2 - dn) - (dn - n)$$

$$\leq cn^2 - dn \text{ if } d \geq 1$$



↗ Don't miss the next class !!!!!!!!!!

# Lecture Title: Recursion (Cont.)

## Recurrences & Master Method



**Dept. of Computer Science  
Faculty of Science and Technology**

<b>Lecture No:</b>	04	<b>Week No:</b>	04	<b>Semester:</b>	
<b>Lecturer:</b>	<i>Name &amp; email: Md. Manzurul Hasan, manzurul@aiub.edu</i>				

# Lecture Outline

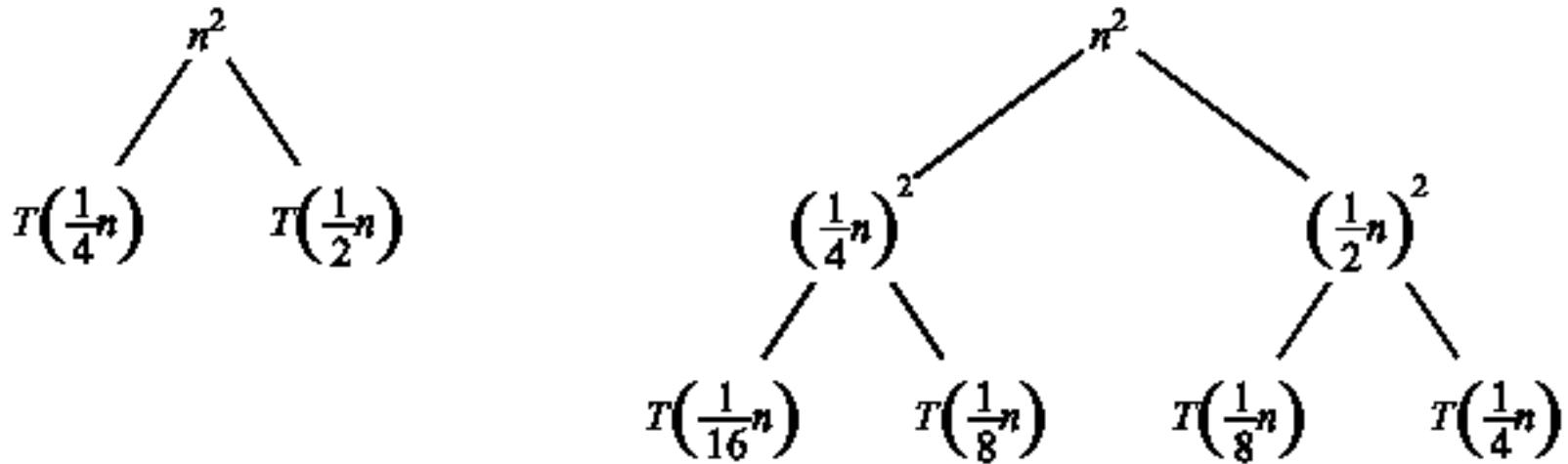


1. Divide and Conquer (Previous Week)
2. Recurrences in Divide and Conquer and methodologies for recurrence Solutions (Previous Week)
3. Repeated Backward Substitution Method (Previous Week)
4. Substitution Method (Previous Week)
- 5. Recursion Tree**
- 6. Master Method**

# Recursion Tree

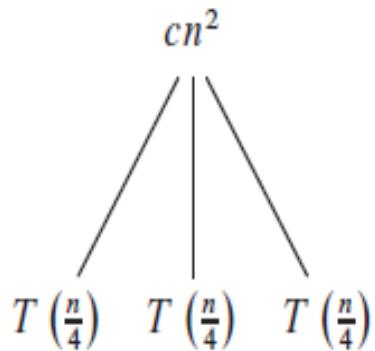
- A recursion tree is a convenient way to visualize what happens when a recurrence is iterated.
- Good for "guessing" asymptotic solutions to recurrences

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



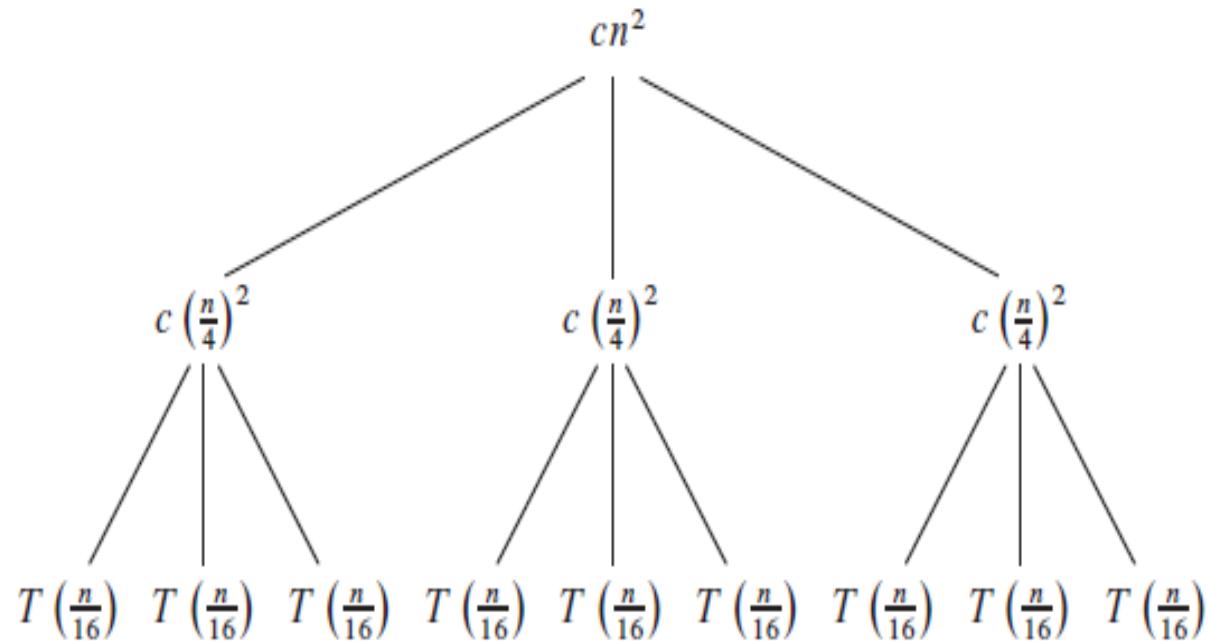
$$T(n) = 3T(\lfloor n/4 \rfloor) + \Theta(n^2) \quad T(n) = 3T(n/4) + cn^2$$

$T(n)$



(a)

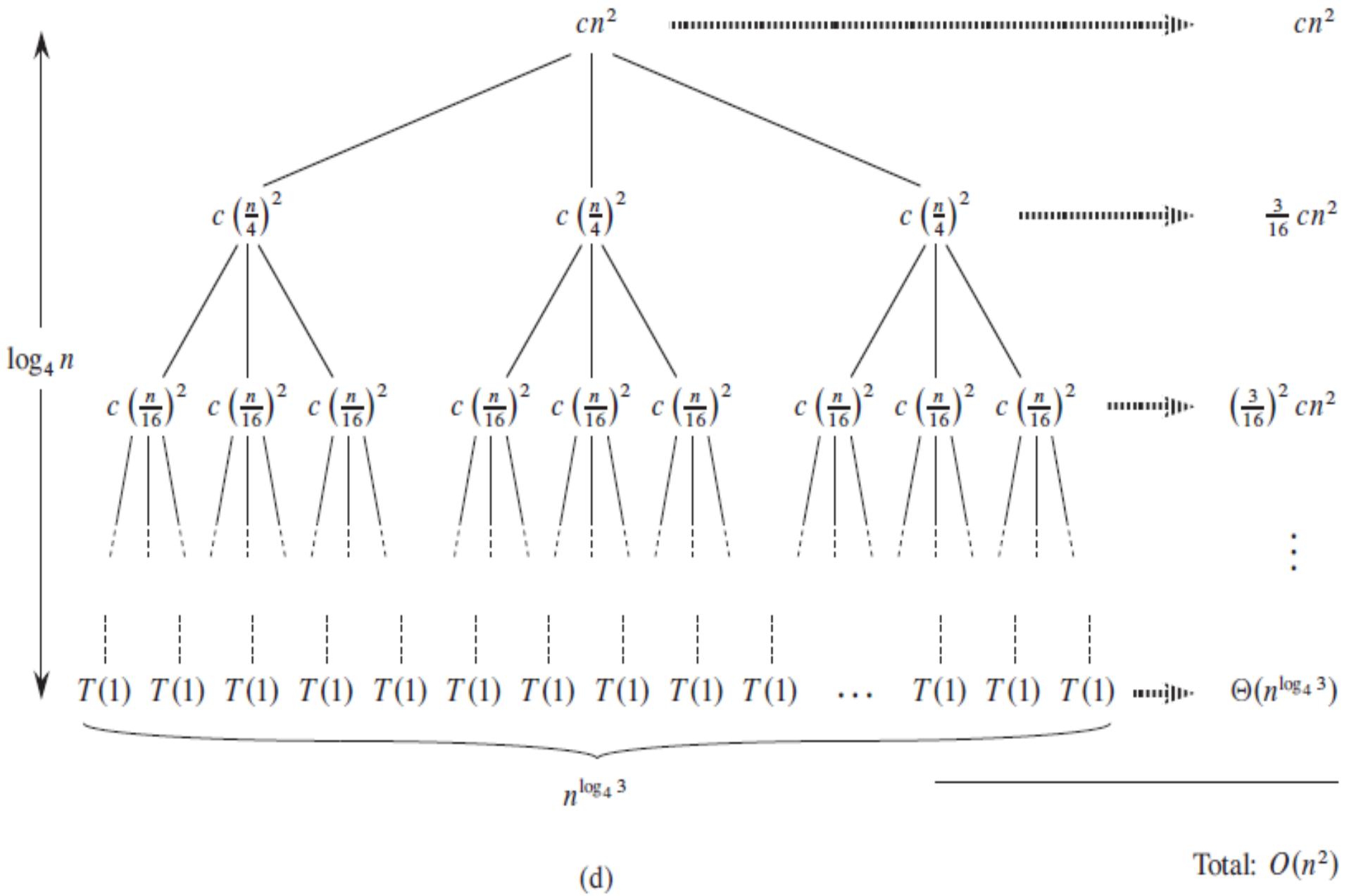
$cn^2$



(b)

$cn^2$

(c)



$$\begin{aligned}
 T(n) &= cn^2 + \frac{3}{16}cn^2 + \left(\frac{3}{16}\right)^2 cn^2 + \cdots + \left(\frac{3}{16}\right)^{\log_4 n - 1} cn^2 + \Theta(n^{\log_4 3}) \\
 &= \sum_{i=0}^{\log_4 n - 1} \left(\frac{3}{16}\right)^i cn^2 + \Theta(n^{\log_4 3}) \\
 &= \frac{(3/16)^{\log_4 n} - 1}{(3/16) - 1} cn^2 + \Theta(n^{\log_4 3}) \quad (\text{by equation (A.5)}).
 \end{aligned}$$

## Geometric series

For real  $x \neq 1$ , the summation

$$\sum_{k=0}^n x^k = 1 + x + x^2 + \cdots + x^n$$

is a *geometric* or *exponential series* and has the value

$$\sum_{k=0}^n x^k = \frac{x^{n+1} - 1}{x - 1}. \tag{A.5}$$

When the summation is infinite and  $|x| < 1$ , we have the infinite decreasing geometric series

$$\sum_{k=0}^{\infty} x^k = \frac{1}{1-x}. \tag{A.6}$$

$$\begin{aligned}
 T(n) &= \sum_{i=0}^{\log_4 n - 1} \left(\frac{3}{16}\right)^i cn^2 + \Theta(n^{\log_4 3}) \\
 &< \sum_{i=0}^{\infty} \left(\frac{3}{16}\right)^i cn^2 + \Theta(n^{\log_4 3}) \\
 &= \frac{1}{1 - (3/16)} cn^2 + \Theta(n^{\log_4 3}) \\
 &= \frac{16}{13} cn^2 + \Theta(n^{\log_4 3}) \\
 &= O(n^2).
 \end{aligned}$$

## Verify our guess [substitution]

$$\begin{aligned} T(n) &\leq 3T(\lfloor n/4 \rfloor) + cn^2 \\ &\leq 3d \lfloor n/4 \rfloor^2 + cn^2 \\ &\leq 3d(n/4)^2 + cn^2 \\ &= \frac{3}{16}dn^2 + cn^2 \\ &\leq dn^2, \end{aligned}$$

where the last step holds as long as  $d \geq (16/13)c$ .



# Master Method

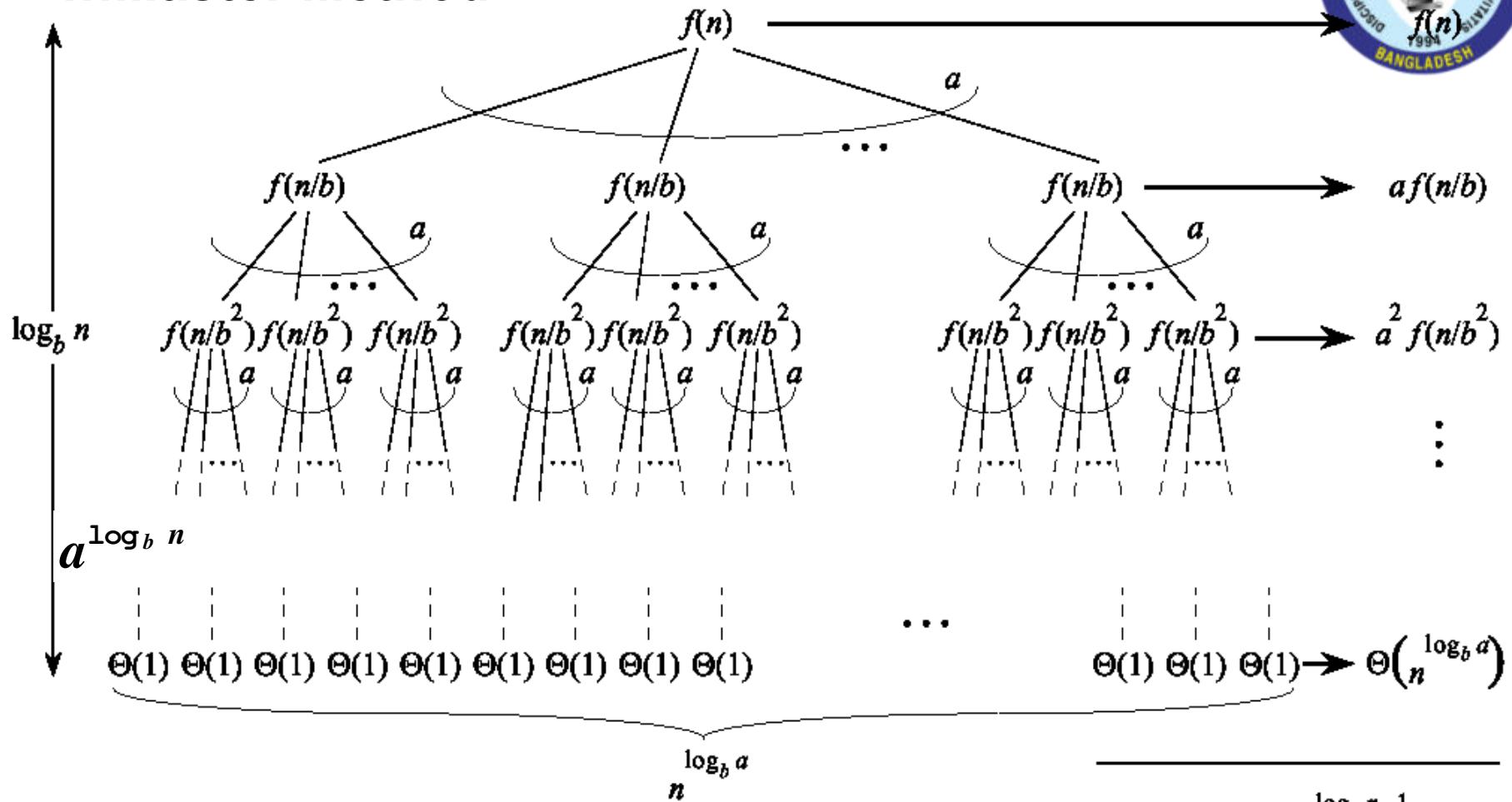
- ▶ The idea is to solve a class of recurrences that have the form

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

- ▶ **Assumptions:**  $a \geq 1$  and  $b > 1$ , and  $f(n)$  is asymptotically positive.
- ▶ Abstractly speaking,  $T(n)$  is the runtime for an algorithm, and we know that
  - ▶  $a$  number of subproblems of size  $n/b$  are solved recursively, each in time  $T(n/b)$ .
  - ▶  $f(n)$  is the cost of dividing the problem and combining the results. e.g., In merge-sort .

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

# ...Master Method



Split problem into  $a$  parts. There are  $\log_b n$  levels. There are  $a^{\log_b n} = n^{\log_b a}$  leaves.

$$\text{Total: } \Theta\left(n^{\log_b a}\right) + \sum_{j=0}^{\log_b n-1} a^j f(n/b^j)$$



## ...Master Method

- ▶ Iterating the recurrence (expanding the tree) yields

$$\begin{aligned}T(n) &= f(n) + aT(n/b) \\&= f(n) + af(n/b) + a^2T(n/b^2) \\&= f(n) + af(n/b) + a^2f(n/b^2) + \dots \\&\quad a^{\log_b n - 1}f(n/b^{\log_b n - 1}) + a^{\log_b n}T(1)\end{aligned}$$

$$T(n) = \sum_{j=0}^{\log_b n - 1} a^j f(n/b^j) + \Theta(n^{\log_b a})$$

- ▶ The first term is a division/recombination cost (totaled across all levels of the tree).
- ▶ The second term is the cost of doing all subproblems of size 1 (total of all work pushed to leaves).

# Master Method, Intuition

- ▶ **Three common cases:**
  1. Running time dominated by cost at leaves.
  2. Running time evenly distributed throughout the tree.
  3. Running time dominated by cost at the root.
- ▶ To solve the recurrence, we need to identify the dominant term.
- ▶ In each case compare  $f(n)$  with  $O(n^{\log_b a})$

# Master Method, Case 1

- ↗ If  $f(n) = O(n^{\log_b a - \varepsilon})$  for some constant  $\varepsilon > 0$  then
  - ↗  $f(n)$  grows polynomially slower than  $n^{\log_b a}$  (by factor  $n^\varepsilon$ ).
- ↗ **The work at the leaf level dominates**

Cost of all the leaves  $\Theta(n^{\log_b a})$

## Master Method, Case 2

- ↗ If  $f(n) = \Theta(n^{\log_b a})$  then
  - ↗  $f(n)$  and  $n^{\log_b a}$  are asymptotically the same
- ↗ The work is distributed equally throughout the tree

(level cost)(number of levels)

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

## Master Method, Case 3

↗ If  $f(n) = \Omega(n^{\log_b a + \varepsilon})$  for some constant  $\varepsilon > 0$  then

↗ Inverse of Case 1

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

↗ Also need a “regularity” condition

$\exists c < 1$  and  $n_0 > 0$  such that  $af(n/b) \leq cf(n) \quad \forall n > n_0$

↗ The work at the root dominates

division/recombination cost

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

# Master Theorem, Summarized

Given: recurrence of the form  $T(n) = aT(n/b) + f(n)$

$$1. \quad f(n) = O\left(n^{\log_b a - \varepsilon}\right)$$

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

$$2. \quad f(n) = \Theta\left(n^{\log_b a}\right)$$

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

$$3. \quad f(n) = \Omega\left(n^{\log_b a + \varepsilon}\right) \text{ and } af(n/b) \leq cf(n), \text{ for some } c < 1, n > n_0$$

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

# Strategy

1. Extract  $a$ ,  $b$ , and  $f(n)$  from a given recurrence
2. Determine  $n^{\log_b a}$
3. Compare  $f(n)$  and  $n^{\log_b a}$  asymptotically
4. Determine appropriate MT case and apply it

Merge sort:  $T(n) = 2T(n/2) + Q(n)$

1.  $a=2$ ,  $b=2$ ,  $f(n) = Q(n)$

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

3.  $Q(n) = Q(n)$

→ Case 2:  $T(n) = Q(n^{\log_2 2} \log n) = Q(n \log n)$

# Examples of Master Method

```
BinarySearch (A, l, r, q) :  
    m := (l+r)/2  
    if A[m]=q then return m  
    else if A[m]>q then  
        BinarySearch (A, l, m-1, q)  
    else BinarySearch (A, m+1, r, q)
```

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

1.  $a=1, b=2, f(n) = 1$

2.  $n \log_2 2 = n$

3.  $1 = O(1)$

→ Case 2:  $T(n) = O(\log n)$

## ...Examples of Master Method

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

1.  $a=9, b=3, f(n) = n$
  2.  $n \log_3 9 = n^2$
  3.  $n = O(n \log_3 9 - e)$  with  $e = 1$
- Case 1:  $T(n) = O(n^2)$

## ...Examples of Master Method

$$T(n) = 3T(n/4) + n \log n$$

1.  $a=3, b=4, f(n) = n \log n$

2.  $n \log_4 3 = n^{0.792}$

3.  $n \log n = W(n \log_4 3 + e)$  with  $e = 0.208$

→ Case 3:

regularity condition:  $af(n/b) \leq cf(n)$

$$af(n/b) = 3(n/4)\log(n/4) \leq (3/4)n \log n = cf(n) \quad ; \text{with } c=3/4$$

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

# References & Readings

## ↗ CLRS

- ↗ **Chapter: 4 (4.1-4.3)**
- ↗ **4.4 for bedtime reading**
- ↗ **Exercises**
  - ◆ **4.1, 4.2, 4.3**
- ↗ **Problems**
  - ◆ **4-1, 4-4**

## ↗ HSR

- ↗ **Chapter: 3 (3.1-3.6)**
- ↗ **Examples: 3.1-3.5**
- ↗ **Exercises: 3.1 (1, 2), 3.2 (1, 3-6),**



# Greedy Algorithm

Course Code: CSC2211

Course Title: Algorithms

**Dept. of Computer Science  
Faculty of Science and Technology**

<b>Lecturer No:</b>	06	<b>Week No:</b>	06	<b>Semester:</b>	
<b>Lecturer:</b>	<i>Name &amp; email</i>				



# Lecture Outline

1. Optimization Problem
2. Greedy Algorithm.
3. Coin Changing Problem.
4. Fractional knapsack Problem.
5. Huffman Encoding.
6. Activity Selection Problem.



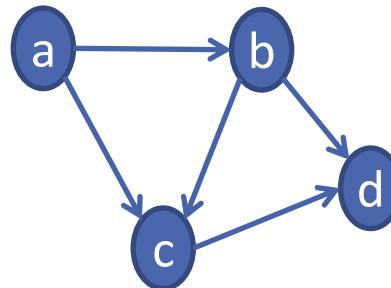
# Optimization Problems

- An optimization problem is one in which you want to find, not just *a* solution, but the *best* solution

## Example

- Graph coloring optimization problem: given an undirected graph,  $G = (V, E)$ , what is the minimum number of colors required to assign “colors” to each vertex in such a way that no two adjacent vertices have the same color

- Path optimization problem: Find the shortest path(s) from  $u$  to  $v$  in a given graph  $G$ .



Find all the  
**shortest** paths  
from a to d:  
 $a \rightarrow b \rightarrow d$   
 $a \rightarrow c \rightarrow d$

# Greedy Algorithm

- Solves an optimization problem
- For many optimization problems, greedy algorithm can be used. (not always)
- Greedy algorithm for optimization problems typically go through a sequence of steps, with a set of choices at each step. **Current choice does not depend on evaluating potential future choices or pre-solving repeatedly occurring subproblems** (a.k.a., *overlapping subproblems*). With each step, the original problem is reduced to a smaller problem.
- Greedy algorithm always makes the choice that looks best at the moment.
- It makes a locally optimal choice in the hope that this choice will lead to a globally optimal solution.



# Optimal Solution

## **What is an Optimal Solution?**

- Given a problem, more than one solution exist
- One of the solution is the best based on some given constraints, that solution is called the optimal solution

## **What is Global Optimal Solution?**

- Optimal Solution to the main problem

## **What is local Optimal Solution?**

- Optimal Solution to the sub-problems



# Example of Greedy Algorithms

- Activity Selection Problem
- Coin Changing Problem
- Job Scheduling Problem
- Fractional Knapsac Problem
- Dijkstra's Shortest Path Problem
- Minimum Spanning Tree Problem



## Coin changing problem

Definition

Given coin denominations in  $\{C\}$ , make change for a given amount  $A$  with the minimum number of coins.

**Example:**

Coin denominations,  $C = \{25, 10, 5, 1\}$  Amount to change,  $A = 73$

Choose 2 25 coins, so remaining is  $73 - 2 * 25 = 23$

Choose 2 10 coins, so remaining is  $23 - 2 * 10 = 3$

Choose 0 5 coins, so remaining is 3

Choose 3 1 coins, so remaining is  $3 - 1 * 3 = 0$

Solution (and it's optimal):  $2 \times 25 + 2 \times 10 + 3 \times 1 = 7$  coins

**Key Question**

*Does a greedy approach always produce the optimal solution?*



## Coin changing problem (continued)

Coin denominations,  $C = \{12, 5, 1\}$       Amount to change,  $A = 15$

### Example (using greedy strategy)

Choose 1 12 coins, so remaining is  $15 - 1 * 12 = 3$

Choose 3 1 coins, so remaining is  $3 - 3 * 1 = 0$

Solution: 4 coins.

### Example (using optimal strategy)

Choose 0 12 coins, so remaining is 15

Choose 3 5 coins, so remaining is  $15 - 3 * 5 = 0$

Solution: 3 coins.

**Correctness depends on the choice of coins,  
so greedy strategy does not provide a general solution to this problem!**



## Fractional knapsack problem

### Definition (fractional knapsack problem)

Given a set  $S$  of  $n$  items, such that each item  $i$  has a positive benefit  $b_i$  and a positive weight  $w_i$ , the goal is to find the maximum-benefit subset that does not exceed a given weight  $W$ , allowing for fractional items.

### Key question

- What strategy to use to select the next item (and the amount of it)?



## Fractional knapsack problem

# Capacity $W=20$

Item	Benefit	Weight
A	25	18 kg
B	15	10 kg
C	24	15 kg

**First Strategy:** Greedy method using **Profit** as it's measure. At each step it will choose an object that increases the profit the **most**.



## Fractional knapsack problem

Item	Benefit	Weight
A	25	18 kg
B	15	10 kg
C	24	15 kg

- Note here, If we select  $C$  as first element and  $B$  as second element then,
- Total benefit =  $24 + 15(5/10) = 31.5$
- So, previous solution is not the optimal one.
- Thus, First Strategy fails.



## Fractional knapsack problem

**Second Strategy:** Greedy method using **Capacity** as it's measure will, at each step choose an object that increases the capacity the least.

- B has the least weight 10. So select this. Remaining Capacity:  
 $20 - 10 = 10$



## Fractional knapsack problem

**Third Strategy:** Strives to achieve a balance between the rate at which profit increases and the rate at which capacity is used.

- At each step, include the object which has the maximum profit per unit of capacity used.
- That means, objects are considered in order of the ratio  $b_i/w_i$ .



## Fractional knapsack in action

A

B

C

D

Tk. 140

2 kg

Tk. 200

1 kg

Tk. 150

5 kg

Tk. 240

3 kg

Licensed under

CSE 221: Algorithms

14 /

8

Item	Benefit	Weight
A	140	2 kg
B	200	1 kg
C	150	5 kg
D	240	3 kg

Calculate benefit/kg – the **value index**.



## Fractional knapsack in action

A	B	C	D
Tk. 140	Tk. 200	Tk. 150	Tk. 240
2 kg	1 kg	5 kg	3 kg
Item	Benefit	Weight	Value index
A	140	2 kg	70
B	200	1 kg	200
C	150	5 kg	30
D	240	3 kg	80

Sort by **non-increasing** value index.



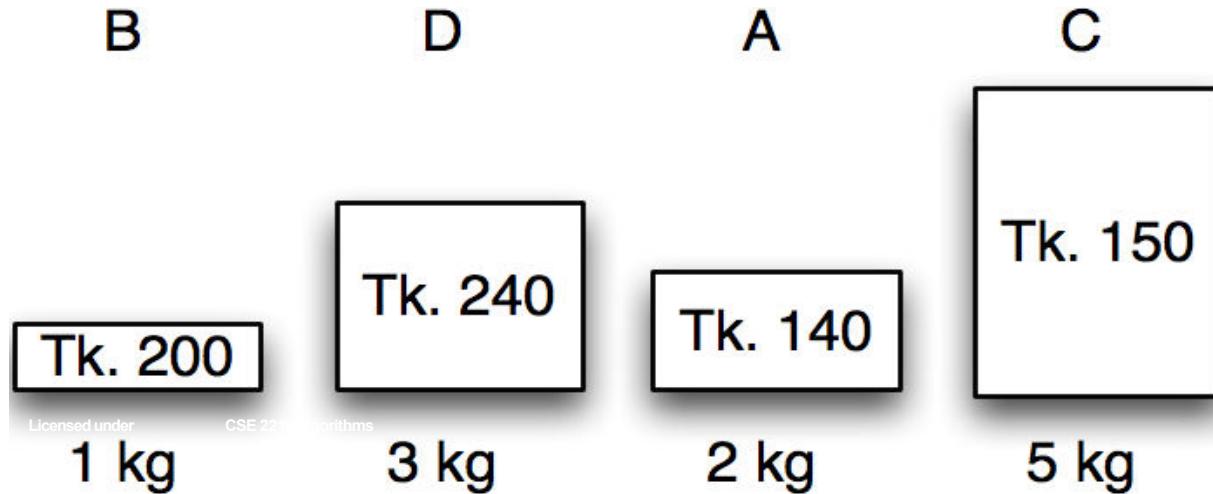
## Fractional knapsack in action

	B	D	A	C
	Tk. 200	Tk. 240	Tk. 140	Tk. 150
Licensed under	CSE 221: Algorithms			
	1 kg	3 kg	2 kg	5 kg
Item	Benefit	Weight	Value index	
B	200	1 kg	200	
D	240	3 kg	80	
A	140	2 kg	70	
C	150	5 kg	30	

Maximum weight: **5 kg**



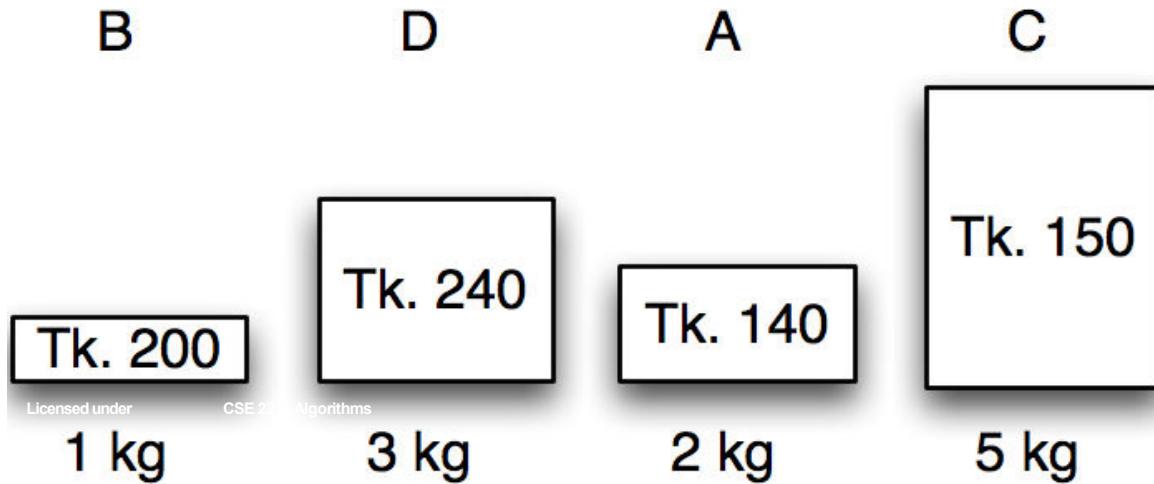
## Fractional knapsack in action



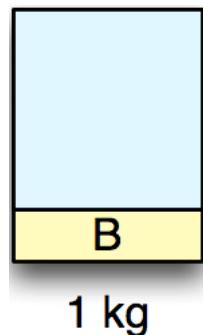
Maximum weight:	5 kg	Remaining:	5 kg	Benefit:	0 kg
				0 kg	



# Fractional knapsack in action



Item	Benefit	Weight	Value index	Chosen
B	200	1 kg	200	1 kg
D	240	3 kg	80	0 kg
A	140	2 kg	70	0 kg
C	150	5 kg	30	0 kg

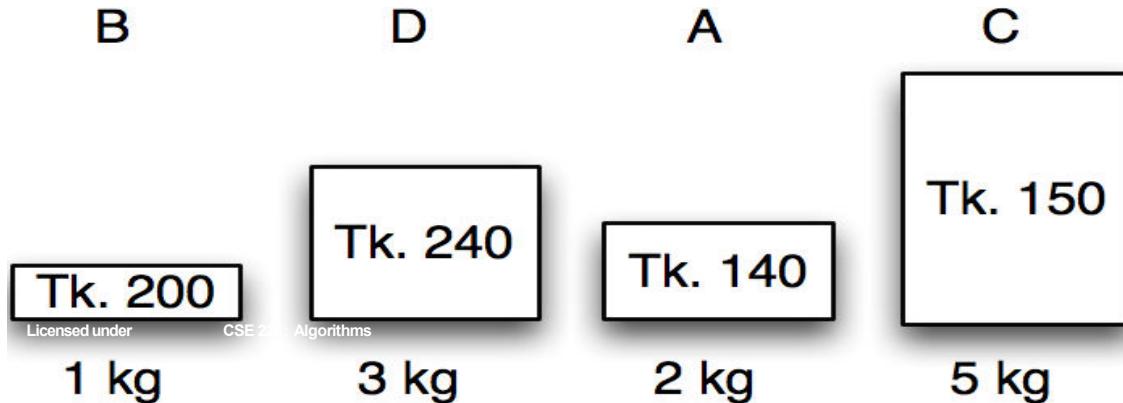


Maximum weight: 5 kg

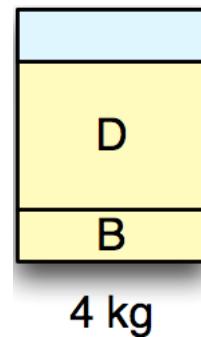
Remaining: 4 kg

Benefit: 200 kg

# Fractional knapsack in action



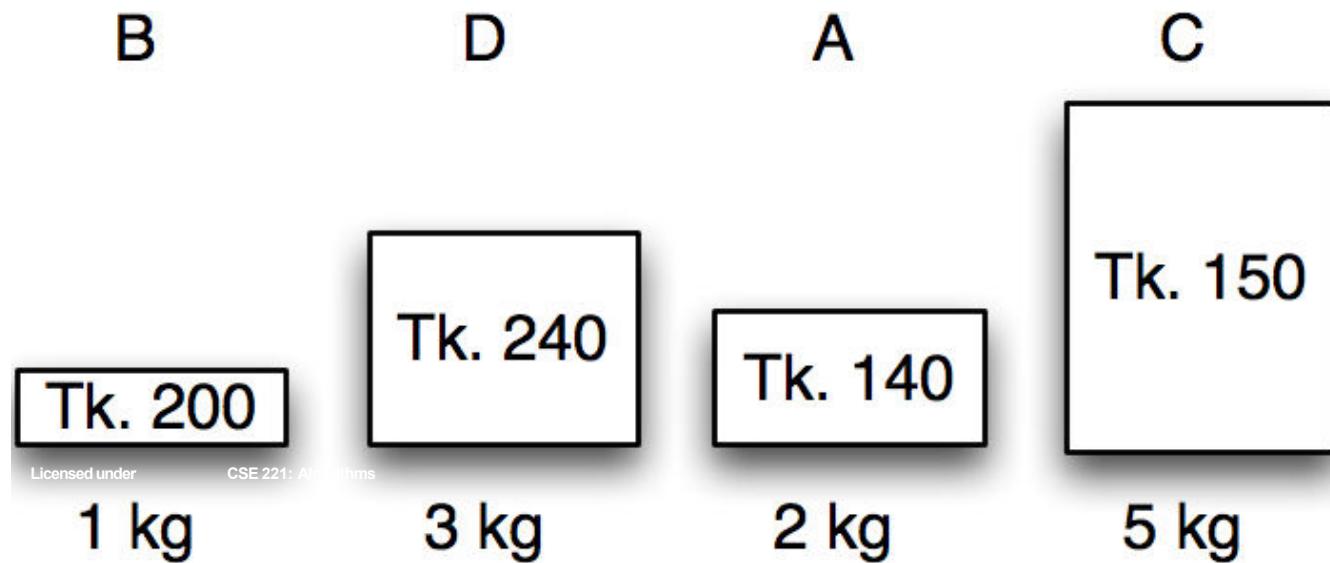
Item	Benefit	Weight	Value index	Chosen
B	200	1 kg	200	1 kg
D	240	3 kg	80	3 kg
A	140	2 kg	70	0 kg
C	150	5 kg	30	0 kg



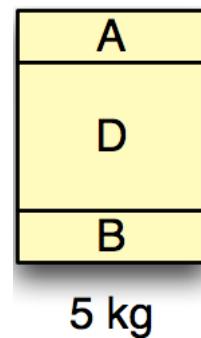
Maximum weight: 5 kg      Remaining: 1 kg      Benefit: 440 kg



## Fractional knapsack in action



Item	Benefit	Weight	Value index	Chosen
B	200	1 kg	200	1 kg
D	240	3 kg	80	3 kg
A	140	2 kg	70	1 kg
C	150	5 kg	30	0 kg



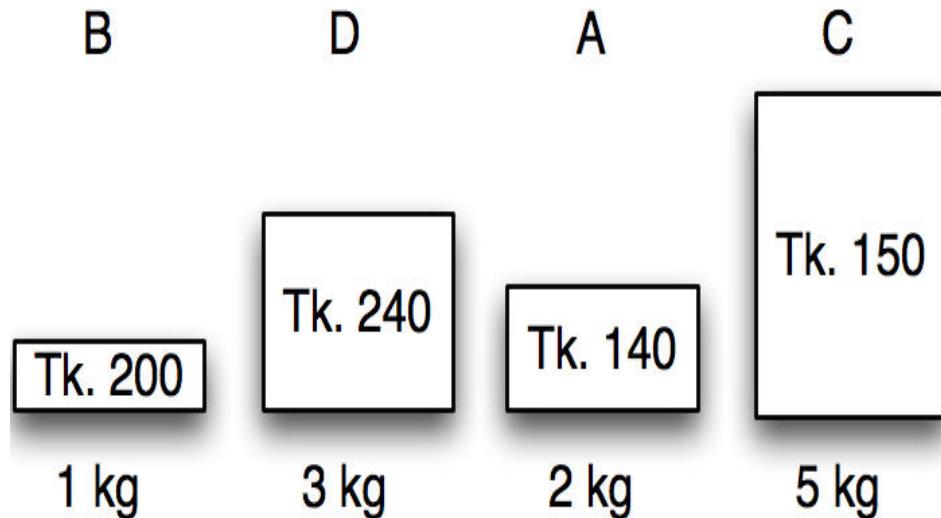
Maximum weight: 5 kg

Remaining: 0 kg

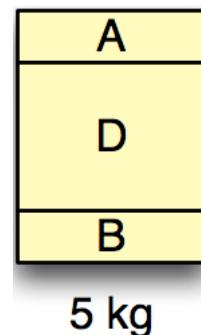
Benefit: 510 kg



# Fractional knapsack in action



Item	Benefit	Weight	Value index	Chosen
B	200	1 kg	200	1 kg
D	240	3 kg	80	3 kg
A	140	2 kg	70	1 kg
C	150	5 kg	30	0 kg



Maximum weight: 5 kg

Remaining: 0 kg

Benefit: 510 kg



## Fractional knapsack greedy algorithm

Algorithm *fractionalKnapsack(S, W)*

**Input:** set  $S$  of items w/ benefit  $b_i$  and weight  $w_i$ ; max. weight  $W$

**Output:** amount  $x_i$  of each item  $i$  to maximize benefit w/ weight at most  $W$

**for** *each item i in S*

$x_i \leftarrow 0$

$v_i \leftarrow b_i / w_i$  {value}

$w \leftarrow 0$  {total weight}

**while**  $w < W$

*remove item i with highest  $v_i$*

$x_i \leftarrow \min\{w_i, W - w\}$

$w \leftarrow w + x_i$



## Fractional knapsack algorithm

### Running time:

Given a collection  $S$  of  $n$  items, such that each item  $i$  has a benefit  $b_i$  and weight  $w_i$ , we can construct a maximum-benefit subset of  $S$ , allowing for fractional amounts, that has a total weight  $W$  in  $O(n \log n)$  time. (how?)

Use heap-based priority queue to store  $S$

Removing the item with the highest value takes  $O(\log n)$  time

In the worst case, need to remove all items



## Exercise

Assume that we have a knapsack with max weight capacity,  $W = 16$ . our objective is to fill the knapsack with items such that the benefit (value or profit) is maximum.

Consider the following items and their associated weight and value

ITEM	WEIGHT	VALUE
i1	6	6
i2	10	2
i3	3	1
i4	5	8
i5	1	3
i6	3	5



## Huffman Codes

Widely used technique for data compression

Assume the data to be a sequence of characters

Looking for an effective way of storing the data

***Binary character code***

Uniquely represents a character by a binary string



## Fixed-Length Codes

*E.g.:* Data file containing 100,000 characters

	a	b	c	d	e	f
Frequency (thousands)	45	13	12	16	9	5

3 bits needed

$a = 000, b = 001, c = 010, d = 011, e = 100, f = 101$

Requires:  $100,000 \cdot 3 = 300,000$  bits

***Idea: Use the frequencies of occurrence of characters to build a optimal way of representing each character***

	a	b	c	d	e	f
Frequency (thousands)	45	13	12	16	9	5



## Variable-Length Codes

*E.g.:* Data file containing 100,000 characters

	a	b	c	d	e	f
Frequency (thousands)	45	13	12	16	9	5

Assign short codewords to frequent characters and long codewords to infrequent characters

$$\begin{aligned}a &= 0, b = 101, c = 100, d = 111, e = 1101, f = 1100 \\(45 \cdot 1 + 13 \cdot 3 + 12 \cdot 3 + 16 \cdot 3 + 9 \cdot 4 + 5 \cdot 4) \cdot 1,000 \\&= 224,000 \text{ bits}\end{aligned}$$



## Prefix Codes

Prefix codes:

Codes for which no codeword is also a prefix of some other codeword

Better name would be “prefix-free codes”

We can achieve optimal data compression using prefix codes

We will restrict our attention to prefix codes



## Encoding with Binary Character Codes

### Encoding

Concatenate the codewords representing each character in the file

*E.g.:*

$a = 0, b = 101, c = 100, d = 111, e = 1101, f = 1100$

$abc = 0 \cdot 101 \cdot 100 = 0101100$



## Decoding with Binary Character Codes

Prefix codes simplify decoding

No codeword is a prefix of another  $\Rightarrow$  the codeword that begins an encoded file is unambiguous

Approach

Identify the initial codeword

Translate it back to the original character

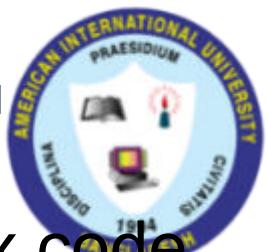
Repeat the process on the remainder of the file

*E.g.:*

$a = 0, b = 101, c = 100, d = 111, e = 1101, f = 1100$

$001011101 =$

$0 \cdot 0 \cdot 101 \cdot 1101 = aabe$



## Constructing a Huffman Code

A greedy algorithm that constructs an optimal prefix code called a **Huffman code**

Assume that:

$C$  is a set of  $n$  characters

Each character has a frequency  $f(c)$

The tree  $T$  is built in a bottom up manner

Left means '0', right means '1'

More frequent characters will be higher in the tree

Idea:

f: 5	e: 9	c: 12	b: 13	d: 16	a: 45
------	------	-------	-------	-------	-------

Start with a set of  $|C|$  leaves

At each step, merge the two least frequent objects: the frequency of the new node = sum of two frequencies

Use a min-priority queue  $Q$ , keyed on  $f$  to identify the two least frequent objects



## Constructing a Huffman tree

**Alg.:** HUFFMAN( $C$ )

1.  $n \leftarrow |C|$

Running time:  $O(n \lg n)$

2.  $Q \leftarrow C$

3. **for**  $i \leftarrow 1$  **to**  $n - 1$   $\leftarrow O(n)$

4.     **do** allocate a new node  $z$

5.          $\text{left}[z] \leftarrow x \leftarrow \text{EXTRACT-MIN}(Q)$

6.          $\text{right}[z] \leftarrow y \leftarrow \text{EXTRACT-MIN}(Q)$

7.          $f[z] \leftarrow f[x] + f[y]$

8.          $\text{INSERT}(Q, z)$

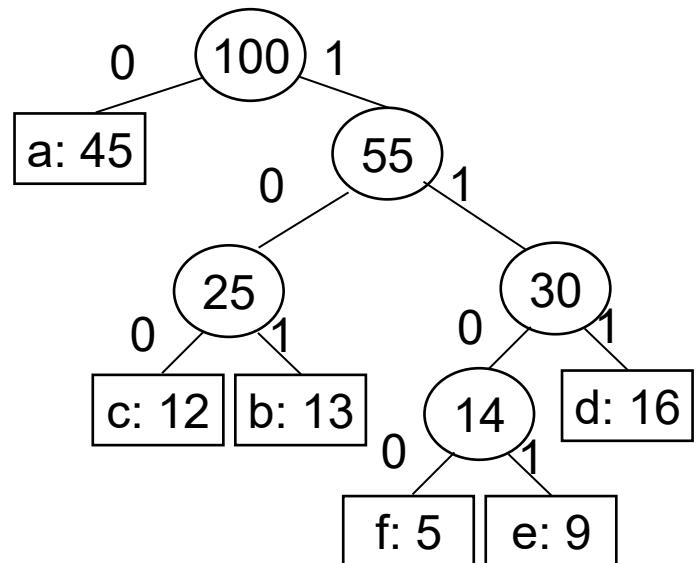
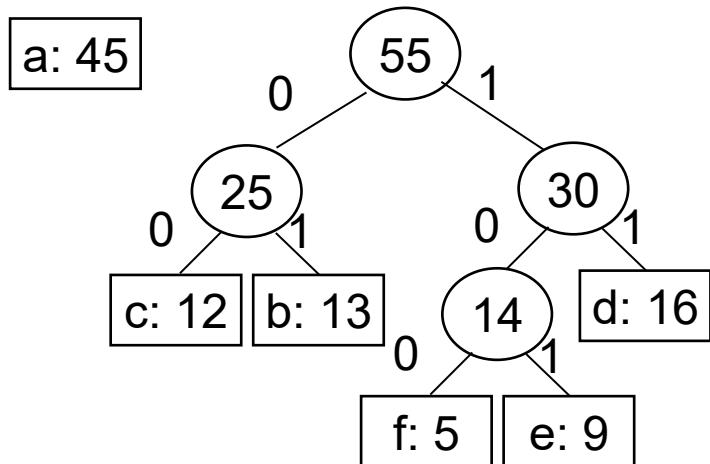
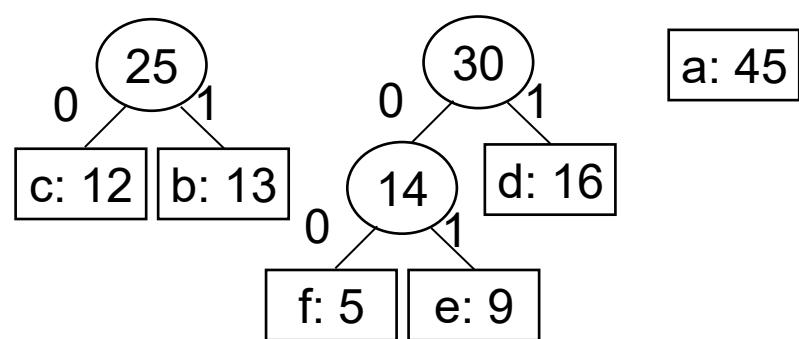
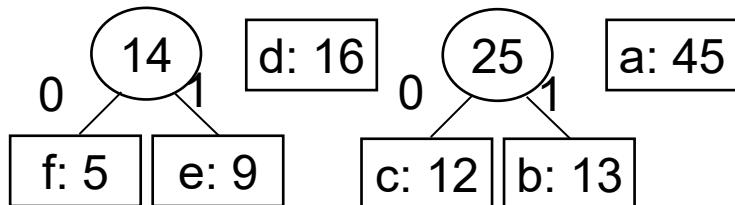
9. **return**  $\text{EXTRACT-MIN}(Q)$

$\} O(\lg n)$

## Example

f: 5 | e: 9 | c: 12 | b: 13 | d: 16 | a: 45

c: 12 | b: 13 | 0 | 1 | 14 | d: 16 | a: 45





## Huffman encoding/decoding using Huffman tree

### **Encoding:**

- Construct Huffman tree using the previous algorithm
- Traverse the Huffman tree to assign a code to each leaf (representing an input character )
- Use these codes to encode the file

### **Decoding:**

- Construct Huffman tree using the previous algorithm
- Traverse the Huffman tree according to the bits you encounter until you reach a leaf node at which point you output the character represented by that leaf node.
- Continue in this fashion until all the bits in the file are read.

# Activity-Selection Problem

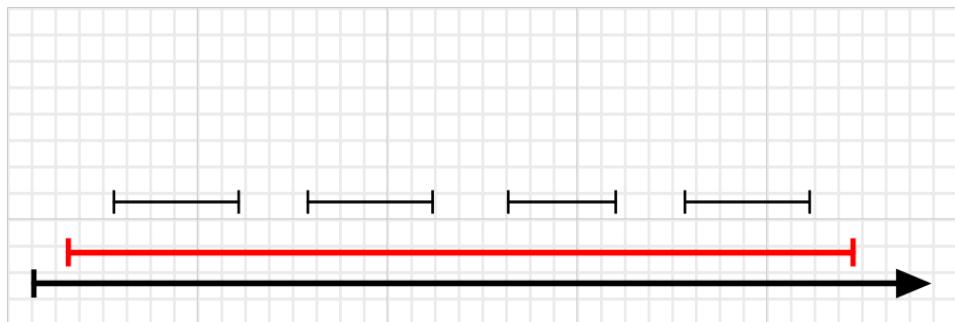
- ↗ Problem: get your money's worth out of a festival
  - ↗ Buy a wristband that lets you onto any ride
  - ↗ Lots of rides, each starting and ending at different times
  - ↗ Your goal: ride as many rides as possible
    - ↗ Another, alternative goal that we don't solve here: maximize time spent on rides
- ↗ Welcome to the *activity selection problem*



## Strategy 1:

**The idea is to start using the resource as early as possible.**

1. Sort the activities by starting time, breaking ties arbitrarily.
2. Pick the first one, removing it from the list along with all the activities that conflict with it.
3. Repeat Step 2, until the list is empty.



Number Of Activity = 1

***This strategy does not lead to an optimal solution.***

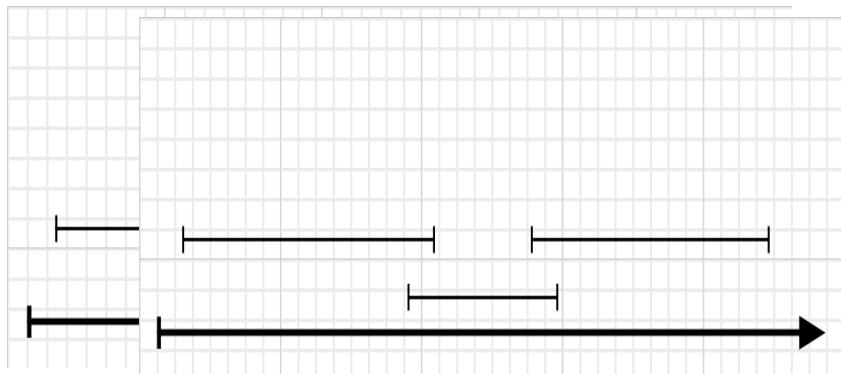


## Strategy 2:

The idea is to start shortest activity.

1. Sort the activities by length, breaking ties arbitrarily.
2. Pick the first one, removing it from the list along with all the activities that conflict with it.

Repeat Step 2, until the list is empty.



Number Of Activity = ?  
Number Of Activity = 1

*This strategy does not lead to an optimal solution.*

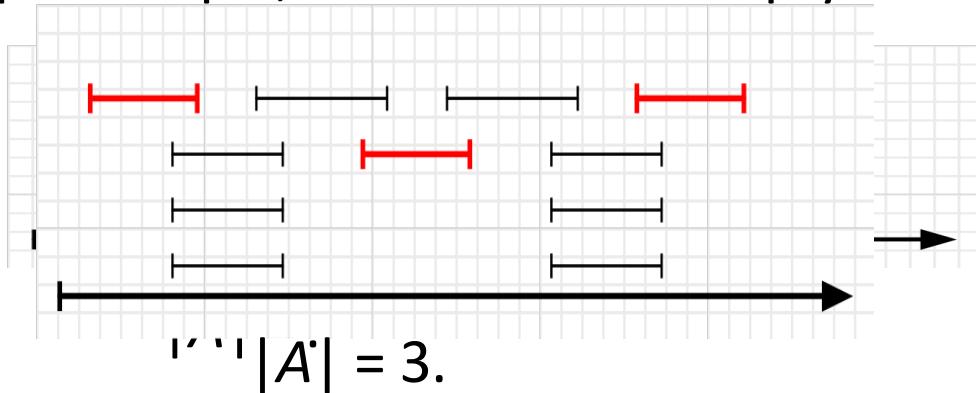


## Strategy 3:

**The idea is to start with least-conflict activity.**

The *Shortest First* strategy failed perhaps because the shorter ones had more conflicts, and ruled out too many activities in the process.

1. Sort the activities by the number of other activities which conflict with it.
2. Pick the first one, removing it from the list along with all the activities that conflict with it.
3. Repeat Step 2, until the list is empty.

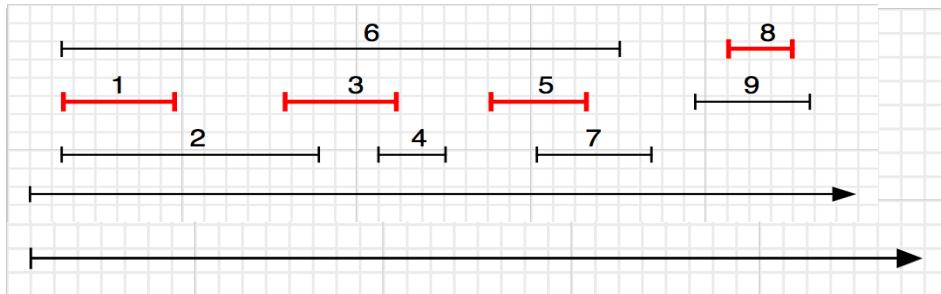


***This strategy does not lead to an optimal solution.***

## Strategy 4:

**The idea is to start with finish-first activity.**

1. Sort the activities by finishing time, breaking ties arbitrarily.
2. Pick the first one, removing it from the list along with all the activities that conflict with it.
3. Repeat Step 2, until the list is empty.



$$|A| = 4.$$

**This strategy is the one that works.**



## Example

Here are a set of start and finish times

i	1	2	3	4	5	6	7	8	9	10	11
$S_j$	1	5	0	3	6	5	3	12	8	2	8
$f_j$	4	7	6	5	10	9	9	16	12	14	11

Sorted according to finishing time

i	1	2	3	4	5	6	7	8	9	10	11
$S_j$	1	3	0	5	3	5	6	8	8	2	12
$f_j$	4	5	6	7	9	9	10	11	12	14	16

- $\{1, 4, 8, 11\}$  which is a larger set (*an optimal solution*)
- Solution is not unique, consider  $\{2, 4, 9, 11\}$  (another optimal solution)



## Activity selection Algorithm

GREEDY-ACTIVITY-SELECTOR( $s, f$ )

```
1    $n \leftarrow \text{length}[s]$ 
2    $A \leftarrow \{a_1\}$ 
3    $i \leftarrow 1$ 
4   for  $m \leftarrow 2$  to  $n$ 
5       do if  $s_m \geq f_i$ 
6           then  $A \leftarrow A \cup \{a_m\}$ 
7            $i \leftarrow m$ 
8   return  $A$ 
```



## Exercise

Input: A list of different activities with starting and ending times.

$\{(5,9), (1,2), (3,4), (0,6), (5,7), (8,9)\}$

Use greedy algorithm to do maximum number of activities.



## Problem types solved by greedy algorithms

- There is no general of knowing whether a problem can be solved by a greedy algorithm.



## Conclusion

- Greedy algorithms often lead to polynomial time-solution for an exponential-time problem.



# Books

***Introduction to Algorithms, Thomas H. Cormen, Charle E. Leiserson, Ronald L. Rivest, Clifford Stein (CLRS).***

***Fundamental of Computer Algorithms, Ellis Horowitz, Sartaj Sahni, Sanguthevar Rajasekaran (HSR)***



## References

[https://www.tutorialspoint.com/data\\_structures\\_algorithms/greedy\\_algorithms.htm](https://www.tutorialspoint.com/data_structures_algorithms/greedy_algorithms.htm)

<https://www.geeksforgeeks.org/fractional-knapsack-problem/>

<https://www.geeksforgeeks.org/activity-selection-problem-greedy-algo-1/>

<https://www.tutorialspoint.com/Huffman-Coding-Algorithm>

<https://www.geeksforgeeks.org/huffman-coding-greedy-algo-3/>

# Dynamic Programming

Course Code: CSC 2211

Course Title: Algorithms



**Dept. of Computer Science  
Faculty of Science and Technology**

<b>Lecturer No:</b>		<b>Week No:</b>	<b>08</b>	<b>Semester :</b>	<b>Summer 2019-202</b>
<b>Lecturer:</b>	<i>Name &amp; email</i>				



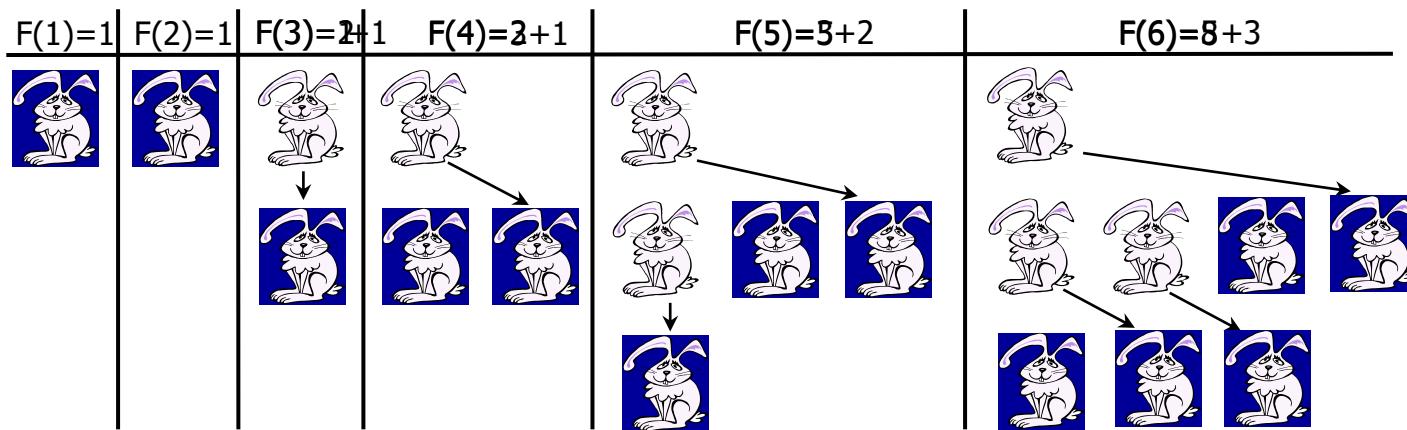
# Lecture Outline

1. Introduction to Dynamic Programming
2. Elements of Dynamic Programming
3. Designing a Dynamic Programming Algorithm
4. 0/1 Knapsack Problem

# Fibonacci Numbers

↗ Leonardo Fibonacci (1202):

- ↗ A rabbit starts producing offspring during the second year after its birth and produces one child each generation
- ↗ How many rabbits will there be after  $n$  generations?





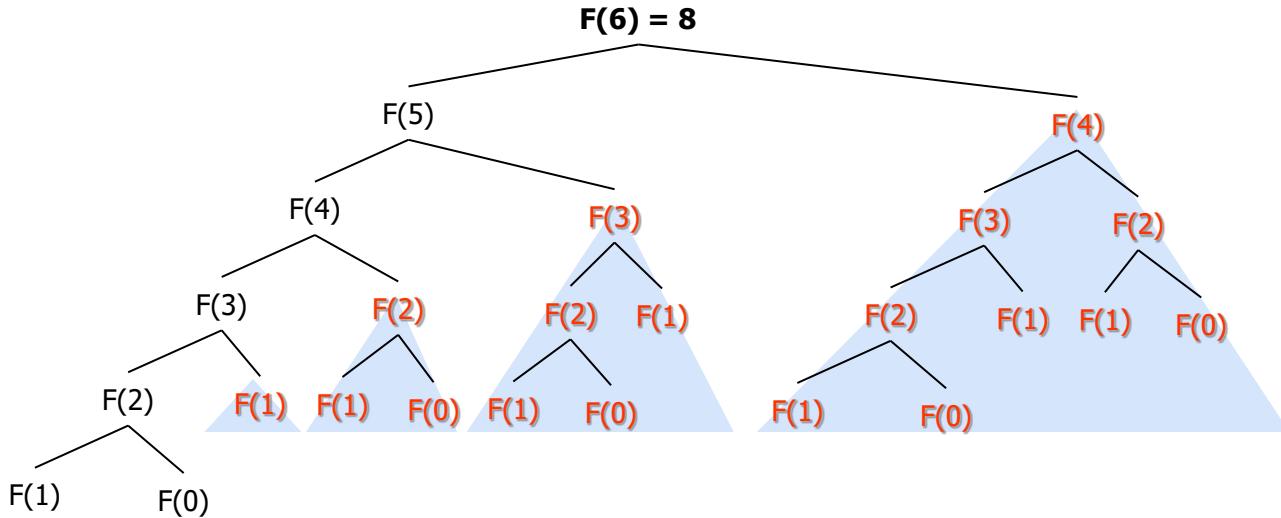
# ...Fibonacci Numbers

- ↗  $F(n) = F(n-1) + F(n-2)$
- ↗  $F(0) = 0, F(1) = 1$
- ↗ 0, 1, 1, 2, 3, 5, 8, 13, 21, 34 ...

```
FibonacciR(n)
01 if n ≤ 1 then return n
02 else return FibonacciR(n-1) + FibonacciR(n-2)
```

- ↗ Straightforward recursive procedure is slow!

# ...Fibonacci Numbers



- ↗ We keep calculating the same value over and over!
- ↗ Subproblems are overlapping – they share sub-subproblems



## ...Fibonacci Numbers

- ↗ How many summations are there in  $F(n)$ ?
  - ↗  $F(n) = F(n - 1) + F(n - 2) + 1$
  - ↗  $F(n) \geq 2F(n - 2) + 1$  and  $F(1) = F(0) = 0$
  - ↗ Solving the recurrence we get  
$$F(n) \geq 2^{n/2} - 1 \approx 1.4^n$$
- ↗ Running time is *exponential!*



## ...Fibonacci Numbers

- ↗ We can calculate  $F(n)$  in *linear* time by remembering solutions to the solved sub-problems (= *dynamic programming*).
- ↗ Compute solution in a bottom-up fashion
- ↗ Trade space for time!

**Fibonacci (n)**

```
01 F[0]←0
02 F[1]←1
03 for i ← 2 to n do
04     F[i] ← F[i-1] + F[i-2]
05 return F[n]
```



## ...Fibonacci Numbers

- In fact, only two values need to be remembered at any time!

```
FibonacciImproved(n)
```

```
01 if n ≤ 1 then return n
02 Fim2 ← 0
03 Fim1 ← 1
04 for i ← 2 to n do
05   Fi ← Fim1 + Fim2
06   Fim2 ← Fim1
07   Fim1 ← Fi
05 return Fi
```



# History

## ➤ Dynamic programming

- Invented in the 1957 by *Richard Bellman* as a general method for optimizing multistage decision processes
- The term “programming” refers to a *tabular method*.
- Often used for *optimization* problems.

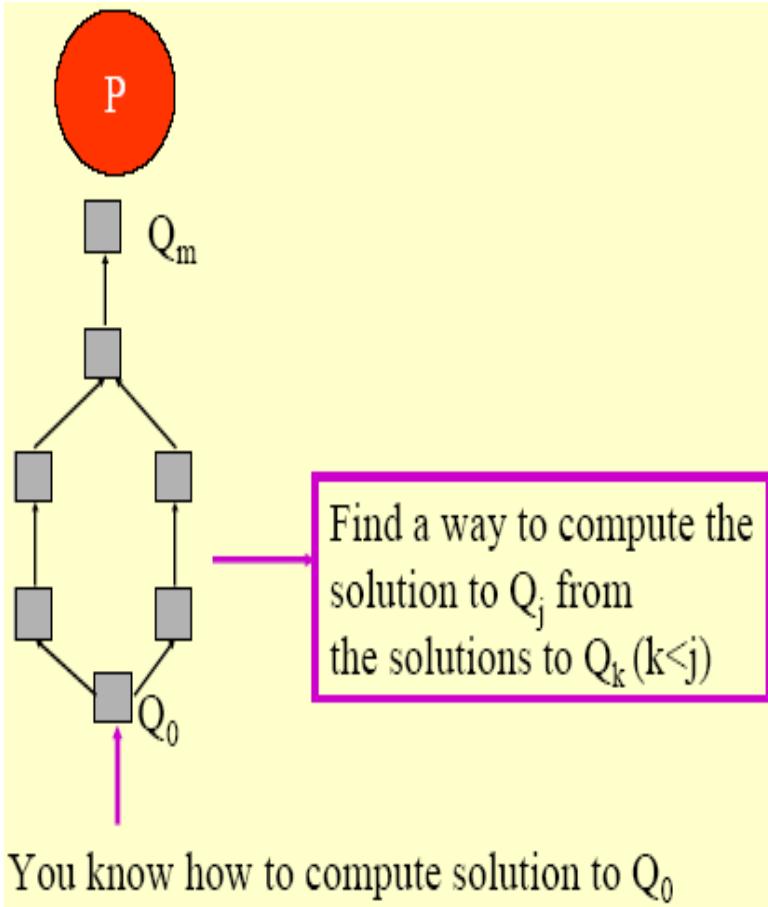


# Dynamic Programming

- ↗ Solves problems by **combining** the solutions to sub problems that contain common sub-sub-problems.
- ↗ Difference between DP and Divide-and-Conquer
  - ↗ Using ***Divide and Conquer*** to solve these problems is **inefficient** as the same common sub-sub-problems have to be solved **many times**.
  - ↗ DP will solve each of them **once** and their **answers are stored in a table** for future reference.

# Intuitive Explanation

- Given a problem  $P$ , obtain a sequence of problems  $Q_0, Q_1, \dots, Q_m$ , where:
  - You have a solution to  $Q_0$
  - The solution to a problem  $Q_j$ ,  $j > 0$ , can be obtained from solutions to problems  $Q_0 \dots Q_k$ ,  $k < j$ , that appear earlier in the “sequence”.

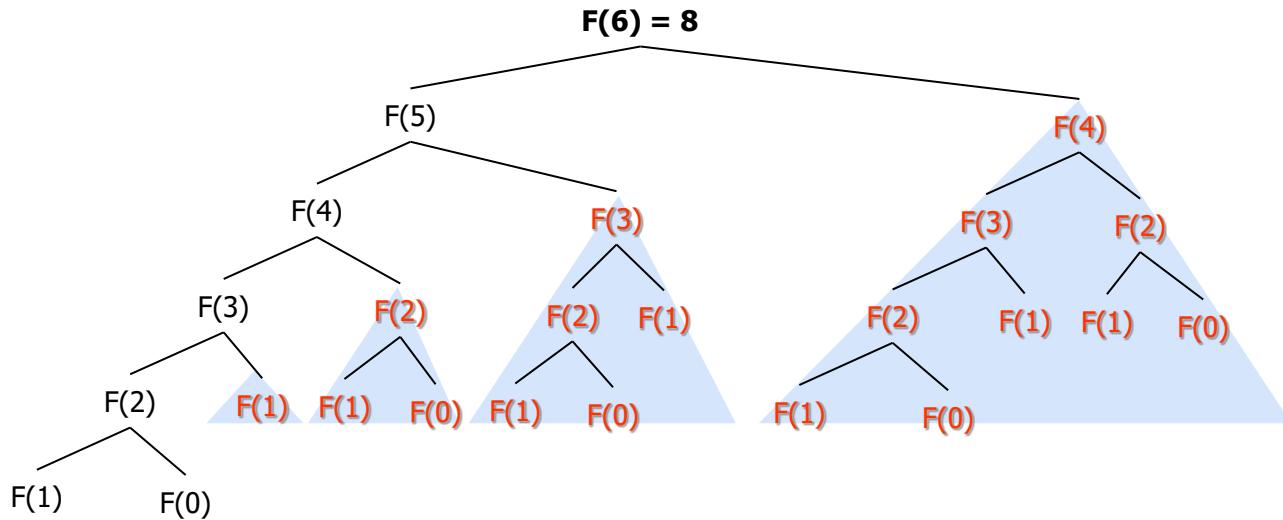




# Elements of Dynamic Programming

- ↗ DP is used to solve problems with the following characteristics:
  - ↗ **Optimal sub-structure** (Principle of Optimality)
    - ↗ an optimal solution to the problem contains within it optimal solutions to sub-problems.
  - ↗ **Overlapping sub problems**
    - ↗ there exist some places where we solve the same sub problem more than once

# Example: Fibonacci Numbers



- ↗ We keep calculating the same value over and over!
- ↗ Subproblems are overlapping – they share sub-subproblems



# Steps to Designing a Dynamic Programming Algorithm

1. Characterize *optimal sub-structure*
2. *Recursively* define the value of an optimal solution
3. Compute the value *bottom up*
4. (if needed) *Construct* an optimal solution



# 1. Optimal Sub-Structure

- An optimal solution to the problem contains optimal solutions to sub-problems
- Solution to a problem:
  - Making a **choice** out of a number of **possibilities** (look what possible choices there can be)
  - **Solving** one or more **sub-problems** that are the **result of a choice** (characterize the space of sub-problems)
- Show that solutions to sub-problems must themselves be optimal for the whole solution to be optimal.



## 2. Recursive Solution

- ↗ Write a recursive solution for the value of an optimal solution.  
$$M_{opt} = \underset{\text{over all } k \text{ choices}}{\text{Optimal}} \left[ \begin{array}{l} \text{Combination of } M_{opt} \text{ for all subproblems resulting from choice } k \\ + \text{The cost associated with making the choice } k \end{array} \right]$$
- ↗ Show that the number of different instances of sub-problems is bounded by a polynomial.



### 3. Bottom Up

- ↗ **Compute** the value of an optimal solution in a bottom-up fashion, so that you always have the necessary **sub-results pre-computed** (or use memoization)
- ↗ Check if it is possible to **reduce** the **space** requirements, by **“forgetting”** solutions to sub-problems that will not be used any more



## 4. Construct Optimal Solution

- Construct an optimal solution from **computed information** (which records a sequence of choices made that lead to an optimal solution)



# Knapsack Problem



# The Knapsack Problem

- ↗ The famous *knapsack problem*:
  - ↗ A thief breaks into a museum. Fabulous paintings, sculptures, and jewels are everywhere. The thief has a good eye for the value of these objects, and knows that each will fetch hundreds or thousands of dollars on the clandestine art collector's market. But, the thief has only brought a single knapsack to the scene of the robbery, and can take away only what he can carry. What items should the thief take to maximize the haul?



# 0-1 Knapsack problem

- ↗ Given a knapsack with maximum capacity  $W$ , and a set  $S$  consisting of  $n$  items
- ↗ Each item  $i$  has some weight  $w_i$  and benefit value  $b_i$  (all  $w_i$ ,  $b_i$  and  $W$  are integer values)
- ↗ Problem: How to pack the knapsack to achieve maximum total value of packed items?



Item	A	B	C	D
Profit	24	18	18	10
Weight	24	10	10	7

Knapsack Size = 25

Item	A	B	C
Price	100	280	120
Weight	10	40	20
Ratio	10	7	6

Knapsack size = 60



# 0-1 Knapsack problem: a picture

This is a knapsack  
Max weight:  $W = 20$

$W = 20$

Items	Weight $w_i$	Benefit value $b_i$
	2	3
	3	4
	4	5
	5	8
	9	10



# The Knapsack Problem

- ↗ More formally, the *0-1 knapsack problem*:
  - ↗ The thief must choose among  $n$  items, where the  $i$ th item worth  $v_i$  dollars and weighs  $w_i$  pounds
  - ↗ Carrying at most  $W$  pounds, maximize value
    - ↗ Note: assume  $v_i$ ,  $w_i$ , and  $W$  are all integers
    - ↗ “0-1” b/c each item must be taken or left in entirety
- ↗ A variation, the *fractional knapsack problem*:
  - ↗ Thief can take fractions of items
  - ↗ Think of items in 0-1 problem as gold ingots, in fractional problem as buckets of gold dust



# 0-1 Knapsack problem

↗ Problem, in other words, is to find

$$\max \sum_{i \in T} b_i \text{ subject to } \sum_{i \in T} w_i \leq W$$

- The problem is called a “0-1” problem, because each item must be entirely accepted or rejected.
- Just another version of this problem is the “*Fractional Knapsack Problem*”, where we can take fractions of items.



## 0-1 Knapsack problem: brute-force approach

Let's first solve this problem with a straightforward algorithm

- ↗ Since there are  $n$  items, there are  $2^n$  possible combinations of items.
- ↗ We go through all combinations and find the one with the most total value and with total weight less or equal to  $W$
- ↗ Running time will be  $O(2^n)$



## 0-1 Knapsack problem: brute-force approach

- ↗ Can we do better?
- ↗ Yes, with an algorithm based on dynamic programming
- ↗ We need to carefully identify the subproblems

Let's try this:

If items are labeled  $1..n$ , then a subproblem would be to find an optimal solution for

$$S_k = \{ \text{items labeled } 1, 2, \dots, k \}$$



# Defining a Subproblem

If items are labeled  $1..n$ , then a subproblem would be to find an optimal solution for  $S_k = \{items\ labeled\ 1,\ 2,\ ..\ k\}$

- ↗ This is a valid subproblem definition.
- ↗ The question is: can we describe the final solution ( $S_n$ ) in terms of subproblems ( $S_k$ )?
- ↗ Unfortunately, we can't do that. Explanation follows....



# Defining a Subproblem

$w_1 = 2$	$w_2 = 4$	$w_3 = 5$	$w_4 = 3$	
$b_1 = 3$	$b_2 = 5$	$b_3 = 8$	$b_4 = 4$	?

Max weight:  $W = 20$

For  $S_4$ :

Total weight: 14;  
total benefit: 20

$w_1 = 2$	$w_2 = 4$	$w_3 = 5$	$w_4 = 9$
$b_1 = 3$	$b_2 = 5$	$b_3 = 8$	$b_4 = 10$

For  $S_5$ :

Total weight: 20  
total benefit: 26

Item #	Weight $W_i$	Benefit $b_i$
1	2	3
2	3	4
3	4	5
4	5	8
5	9	10

$S_5$

$S_4$

**Solution for  $S_4$  is  
not part of the  
solution for  $S_5$ !!!**



# Defining a Subproblem (continued)

- ↗ As we have seen, the solution for  $S_4$  is not part of the solution for  $S_5$
- ↗ So our definition of a subproblem is flawed and we need another one!
- ↗ Let's add another parameter:  $w$ , which will represent the exact weight for each subset of items
- ↗ The subproblem then will be to compute  $B[k, w]$



# Recursive Formula for subproblems

## ■ Recursive formula for subproblems:

$$B[k, w] = \begin{cases} B[k - 1, w] & \text{if } w_k > w \\ \max \{B[k - 1, w], B[k - 1, w - w_k] + b_k\} & \text{else} \end{cases}$$

- ↗ It means, that the best subset of  $S_k$  that has total weight  $w$  is one of the two:
  - 1) the best subset of  $S_{k-1}$  that has total weight  $w$ , or
  - 2) the best subset of  $S_{k-1}$  that has total weight  $w-w_k$  plus the item  $k$



# Recursive Formula

$$B[k, w] = \begin{cases} B[k - 1, w] & \text{if } w_k > w \\ \max\{ B[k - 1, w], B[k - 1, w - w_k] + b_k \} & \text{else} \end{cases}$$

- ↗ The best subset of  $S_k$  that has the total weight  $w$ , either contains item  $k$  or not.
- ↗ First case:  $w_k > w$ . Item  $k$  can't be part of the solution, since if it was, the total weight would be  $> w$ , which is unacceptable
- ↗ Second case:  $w_k \leq w$ . Then the item  $k$  can be in the solution, and we choose the case with greater value



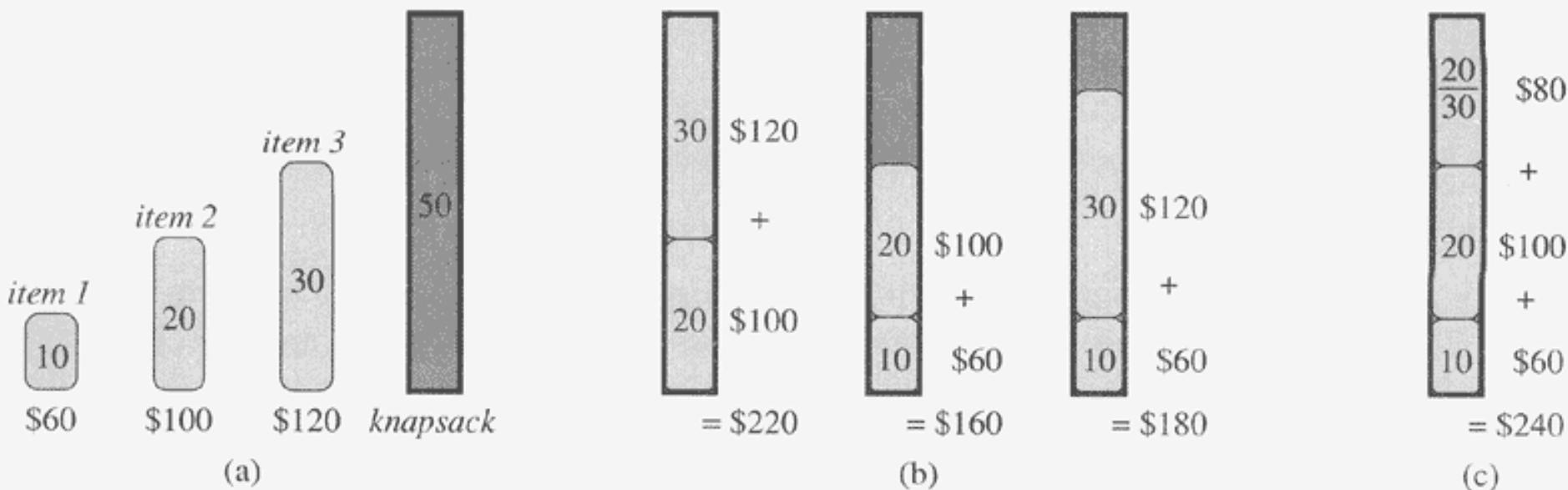
# The Knapsack Problem And Optimal Substructure

- ↗ Both variations exhibit optimal substructure
- ↗ To show this for the 0-1 problem, consider the most valuable load weighing at most  $W$  pounds
  - ↗ *If we remove item  $j$  from the load, what do we know about the remaining load?*
  - ↗ A: remainder must be the most valuable load weighing at most  $W - w_j$  that thief could take from museum, excluding item j



# Solving The Knapsack Problem

- ↗ The optimal solution to the fractional knapsack problem can be found with a greedy algorithm
  - ↗ *How?*
- ↗ The optimal solution to the 0-1 problem cannot be found with the same greedy strategy
  - ↗ Greedy strategy: take in order of dollars/pound
  - ↗ Example: 3 items weighing 10, 20, and 30 pounds, knapsack can hold 50 pounds
    - ↗ *Suppose item 2 is worth \$100. Assign values to the other items so that the greedy strategy will fail*



**Figure 16.2** The greedy strategy does not work for the 0-1 knapsack problem. (a) The thief must select a subset of the three items shown whose weight must not exceed 50 pounds. (b) The optimal subset includes items 2 and 3. Any solution with item 1 is suboptimal, even though item 1 has the greatest value per pound. (c) For the fractional knapsack problem, taking the items in order of greatest value per pound yields an optimal solution.



# The Knapsack Problem: Greedy Vs. Dynamic

- ↗ The fractional problem can be solved greedily
- ↗ The 0-1 problem cannot be solved with a greedy approach
  - ↗ As you have seen, however, it can be solved with dynamic programming



# Fractional-knapsack

- ↗ Greedy-fractional-knapsack ( $w, v, W$ )
- ↗ FOR  $i = 1$  to  $n$ 
  - do  $x[i] = 0$
  - weight = 0
  - while weight <  $W$ 
    - do  $i = \text{best remaining item}$
    - IF weight +  $w[i] \leq W$ 
      - then  $x[i] = 1$
      - weight = weight +  $w[i]$
    - else
      - $x[i] = (w - \text{weight}) / w[i]$
      - weight =  $W$
  - return  $x$



# 0-1 Knapsack Algorithm

for  $w = 0$  to  $W$

$B[0,w] = 0$

for  $i = 0$  to  $n$

$B[i,0] = 0$

    for  $w = 0$  to  $W$

        if  $w_i \leq w$  // item  $i$  can be part of the solution

            if  $b_i + B[i-1, w-w_i] > B[i-1, w]$

$B[i,w] = b_i + B[i-1, w-w_i]$

            else

$B[i,w] = B[i-1,w]$

        else  $B[i,w] = B[i-1,w]$  //  $w_i > w$



# Running time

for  $w = 0$  to  $W$

$O(W)$

$B[0,w] = 0$

for  $i = 0$  to  $n$

Repeat  $n$  times

$B[i,0] = 0$

for  $w = 0$  to  $W$

$O(W)$

< the rest of the code >

What is the running time of this algorithm?

$O(n*W)$

Remember that the brute-force algorithm takes  $O(2^n)$



# Example

Let's run our algorithm on the following data:

$n = 4$  (# of elements)

$W = 5$  (max weight)

Elements (weight, benefit):

(2,3), (3,4), (4,5), (5,6)

## Example (2)

w	i	0	1	2	3	4
0	0					
1	0					
2	0					
3	0					
4	0					
5	0					

for  $w = 0$  to  $W$   
 $B[0,w] = 0$

## Example (3)

w	i	0	1	2	3	4
0	0	0	0	0	0	0
1	0					
2	0					
3	0					
4	0					
5	0					

for  $i = 0$  to  $n$   
 $B[i,0] = 0$

## Example (4)

w	0	1	2	3	4
i	0	0	0	0	0
	0	0			
1	0	0			
2	0				
3	0				
4	0				
5	0				

Items:

1: (2,3)

2: (3,4)

3: (4,5)

4: (5,6)

$i=1$

$b_i=3$

$w_i=2$

$w=1$

$w - w_i = -1$

if  $w_i \leq w$  // item i can be part of the solution

if  $b_i + B[i-1, w-w_i] > B[i-1, w]$

$B[i, w] = b_i + B[i-1, w-w_i]$

else

$B[i, w] = B[i-1, w]$

else  $B[i, w] = B[i-1, w]$  //  $w_i > w$

## Example (5)

i	0	1	2	3	4
w	0	0	0	0	0
1	0	0			
2	0		3		
3	0				
4	0				
5	0				

## Items:

1: (2,3)

2: (3,4)

3: (4,5)

4: (5,6)

i=1

$$b_i=3$$

w<sub>i</sub>=2

**w=2**

$$w - w_i = 0$$

if  $w_i \leq w$  // item i can be part of the solution

if  $b_i + B[i-1, w-w_i] > B[i-1, w]$

$$B[i, w] = b_i + B[i-1, w - w_i]$$

else

$$B[i,w] = B[i-1,w]$$

else  $B[i,w] = B[i-1,w]$  //  $w_i > w$

## Example (6)

w	0	1	2	3	4
i	0	0	0	0	0
	0	0			
	0	3			
	0	<b>3</b>			
	0				
	0				

Items:

1: (2,3)

2: (3,4)

3: (4,5)

4: (5,6)

$i=1$

$b_i=3$

$w_i=2$

$w=3$

$w-w_i=1$

```

if  $w_i \leq w$  // item i can be part of the solution
  if  $b_i + B[i-1, w-w_i] > B[i-1, w]$ 
     $B[i, w] = b_i + B[i-1, w-w_i]$ 
  else
     $B[i, w] = B[i-1, w]$ 
  else  $B[i, w] = B[i-1, w]$  //  $w_i > w$ 

```

## Example (7)

w	0	1	2	3	4
i	0	0	0	0	0
	0	0			
	0	3			
	0	3			
	0	3			
	0				
	0				

Items:

1: (2,3)

2: (3,4)

3: (4,5)

4: (5,6)

$i=1$

$b_i=3$

$w_i=2$

$w=4$

$w - w_i = 2$

if  $w_i \leq w$  // item i can be part of the solution

if  $b_i + B[i-1, w-w_i] > B[i-1, w]$

$B[i, w] = b_i + B[i-1, w-w_i]$

else

$B[i, w] = B[i-1, w]$

else  $B[i, w] = B[i-1, w]$  //  $w_i > w$

## Example (8)

w	0	1	2	3	4
i	0	0	0	0	0
	0	0			
2	0	3			
3	0	3			
4	0	3			
5	0	3			

Items:

1: (2,3)
----------

2: (3,4)

3: (4,5)

4: (5,6)

$i=1$

$b_i=3$

$w_i=2$

$w=5$

$w - w_i = 2$

if  $w_i \leq w$  // item i can be part of the solution

if  $b_i + B[i-1, w-w_i] > B[i-1, w]$

$B[i, w] = b_i + B[i-1, w-w_i]$

else

$B[i, w] = B[i-1, w]$

else  $B[i, w] = B[i-1, w]$  //  $w_i > w$

## Example (9)

w	0	1	2	3	4
i	0	0	0	0	0
	0	0	0	0	0
1	0	0	0	0	0
2	0	3	0	0	0
3	0	3	0	0	0
4	0	3	0	0	0
5	0	3	0	0	0

Items:

1: (2,3)

2: (3,4)

3: (4,5)

4: (5,6)

$i=2$

$b_i=4$

$w_i=3$

$w=1$

$w - w_i = -2$

if  $w_i \leq w$  // item i can be part of the solution

if  $b_i + B[i-1, w-w_i] > B[i-1, w]$

$B[i, w] = b_i + B[i-1, w-w_i]$

else

$B[i, w] = B[i-1, w]$

else  $B[i, w] = B[i-1, w]$  //  $w_i > w$

# Example (10)

w	0	1	2	3	4
i	0	0	0	0	0
	0	0	0		
2	0	3	→ 3		
3	0	3			
4	0	3			
5	0	3			

Items:

1: (2,3)

2: (3,4)

3: (4,5)

4: (5,6)

$i=2$

$b_i=4$

$w_i=3$

$w=2$

$w - w_i = -1$

if  $w_i \leq w$  // item i can be part of the solution

if  $b_i + B[i-1, w-w_i] > B[i-1, w]$

$B[i, w] = b_i + B[i-1, w-w_i]$

else

$B[i, w] = B[i-1, w]$

else  $B[i, w] = B[i-1, w]$  //  $w_i > w$

# Example (11)

w	0	1	2	3	4
i	0	0	0	0	0
	0	0	0		
2	0	3	3		
3	0	3	4		
4	0	3			
5	0	3			

Items:

1: (2,3)

2: (3,4)

3: (4,5)

4: (5,6)

$$i=2$$

$$b_i=4$$

$$w_i=3$$

$$w=3$$

$$w-w_i=0$$

if  $w_i \leq w$  // item i can be part of the solution

if  $b_i + B[i-1, w-w_i] > B[i-1, w]$

$B[i, w] = b_i + B[i-1, w-w_i]$

else

$B[i, w] = B[i-1, w]$

else  $B[i, w] = B[i-1, w]$  //  $w_i > w$

## Example (12)

w	0	1	2	3	4
i	0	0	0	0	0
	0	0	0		
2	0	3	3		
3	0	3	4		
4	0	3	4		
5	0	3			

Items:

1: (2,3)

2: (3,4)

3: (4,5)

4: (5,6)

$i=2$

$b_i=4$

$w_i=3$

$w=4$

$w - w_i = 1$

if  $w_i \leq w$  // item i can be part of the solution

if  $b_i + B[i-1, w-w_i] > B[i-1, w]$

$B[i, w] = b_i + B[i-1, w-w_i]$

else

$B[i, w] = B[i-1, w]$

else  $B[i, w] = B[i-1, w]$  //  $w_i > w$

# Example (13)

w	i	0	1	2	3	4
0		0	0	0	0	0
1		0	0	0		
2		0	3	3		
3		0	3	4		
4		0	3	4		
5		0	3	7		

Items:

1: (2,3)

2: (3,4)

3: (4,5)

4: (5,6)

$i=2$

$b_i=4$

$w_i=3$

$w=5$

$w - w_i = 2$

if  $w_i \leq w$  // item i can be part of the solution

if  $b_i + B[i-1, w-w_i] > B[i-1, w]$

$B[i, w] = b_i + B[i-1, w-w_i]$

else

$B[i, w] = B[i-1, w]$

else  $B[i, w] = B[i-1, w]$  //  $w_i > w$

# Example (14)

w	0	1	2	3	4
i	0	0	0	0	0
0	0	0	0	0	0
1	0	0	0	0	0
2	0	3	3	3	0
3	0	3	4	4	0
4	0	3	4	0	0
5	0	3	7	0	0

Items:

1: (2,3)

2: (3,4)

3: (4,5)

4: (5,6)

$i=3$

$b_i=5$

$w_i=4$

$w=1..3$

if  $w_i \leq w$  // item i can be part of the solution

if  $b_i + B[i-1, w-w_i] > B[i-1, w]$

$B[i, w] = b_i + B[i-1, w-w_i]$

else

$B[i, w] = B[i-1, w]$

else  $B[i, w] = B[i-1, w]$  //  $w_i > w$

# Example (15)

w	0	1	2	3	4
i	0	0	0	0	0
	0	0	0	0	
	0	3	3	3	
	0	3	4	4	
	0	3	4	5	
	0	3	7		

Items:

1: (2,3)

2: (3,4)

3: (4,5)

4: (5,6)

$i=3$

$b_i=5$

$w_i=4$

$w=4$

$w - w_i = 0$

if  $w_i \leq w$  // item i can be part of the solution

if  $b_i + B[i-1, w-w_i] > B[i-1, w]$

$B[i, w] = b_i + B[i-1, w-w_i]$

else

$B[i, w] = B[i-1, w]$

else  $B[i, w] = B[i-1, w]$  //  $w_i > w$

## Example (15)

w	i	0	1	2	3	4
0		0	0	0	0	0
1		0	0	0	0	
2		0	3	3	3	
3		0	3	4	4	
4		0	3	4	5	
5		0	3	7	7	

Items:

1: (2,3)

2: (3,4)

3: (4,5)

4: (5,6)

$i=3$

$b_i=5$

$w_i=4$

$w=5$

$w - w_i = 1$

if  $w_i \leq w$  // item i can be part of the solution

if  $b_i + B[i-1, w-w_i] > B[i-1, w]$

$B[i, w] = b_i + B[i-1, w-w_i]$

else

$B[i, w] = B[i-1, w]$

else  $B[i, w] = B[i-1, w]$  //  $w_i > w$

# Example (16)

w	0	1	2	3	4
i	0	0	0	0	0
0	0	0	0	0	0
1	0	0	0	0	0
2	0	3	3	3	3
3	0	3	4	4	4
4	0	3	4	5	5
5	0	3	7	7	

Items:

1: (2,3)

2: (3,4)

3: (4,5)

4: (5,6)

$i=3$

$b_i=5$

$w_i=4$

$w=1..4$

if  $w_i \leq w$  // item i can be part of the solution

if  $b_i + B[i-1, w-w_i] > B[i-1, w]$

$B[i, w] = b_i + B[i-1, w-w_i]$

else

$B[i, w] = B[i-1, w]$

else  $B[i, w] = B[i-1, w]$  //  $w_i > w$

# Example (17)

w	i	0	1	2	3	4
0		0	0	0	0	0
1		0	0	0	0	0
2		0	3	3	3	3
3		0	3	4	4	4
4		0	3	4	5	5
5		0	3	7	7	7

Items:

- 1: (2,3)
- 2: (3,4)
- 3: (4,5)
- 4: (5,6)

$$i=3$$

$$b_i=5$$

$$w_i=4$$

$$w=5$$

if  $w_i \leq w$  // item i can be part of the solution

if  $b_i + B[i-1, w-w_i] > B[i-1, w]$

$B[i, w] = b_i + B[i-1, w-w_i]$

else

**$B[i, w] = B[i-1, w]$**

else  $B[i, w] = B[i-1, w]$  //  $w_i > w$



# Comments

- ↗ This algorithm only finds the max possible value that can be carried in the knapsack
- ↗ To know the items that make this maximum value, an addition to this algorithm is necessary
- ↗ Please see LCS algorithm lecture for the example how to extract this data from the table we built



# Books

1. *Introduction to Algorithms, Third Edition, Thomas H. Cormen, Charle E. Leiserson, Ronald L. Rivest, Clifford Stein (CLRS)*.
2. *Fundamental of Computer Algorithms, Ellis Horowitz, Sartaj Sahni, Sanguthevar Rajasekaran (HSR)*



# References

- ↗ [https://algorithmist.com/wiki/Dynamic\\_programming](https://algorithmist.com/wiki/Dynamic_programming)
- ↗ <https://www.topcoder.com/community/competitive-programming/tutorials/dynamic-programming-from-novice-to-advanced/>
- ↗ CLRS: 15.3
- ↗ HSR: 5.1, 5.5

# Dynamic Programming

Course Code: CSC 2211

Course Title: Algorithms



**Dept. of Computer Science  
Faculty of Science and Technology**

<b>Lecturer No:</b>		<b>Week No:</b>	<b>09</b>	<b>Semester:</b>	<b>Summer 2019-202</b>
<b>Lecturer:</b>	<i>Name &amp; email</i>				



# Lecture Outline

1. Matrix Chain Multiplication (MCM)
2. Longest Common Subsequence (LCS)



# Review: Matrix Multiplication

Matrix-Multiply(A, B) :

```
1  if columns[A] != rows[B] then
2      error "incompatible dimensions"
3  else{
4      for i = 1 to rows[A] do
5          for j = 1 to columns[B] do{
6              C[i,j] = 0
7              for k = 1 to columns[A] do
8                  C[i,j] = C[i,j]+A[i,k]*B[k,j]
9
10         }
11     }
12 return C
```

A: p x q

B: q x r

# Time complexity = O (pqr) ,

▪ where | A | = p × q and | B | = q × r



# Multiplying Matrices

- Two matrices,  $A$  –  $n \times m$  matrix and  $B$  –  $m \times k$  matrix, can be multiplied to get  $C$  with dimensions  $n \times k$ , using  $n \times m \times k$  scalar multiplications

$$\begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \\ a_{31} & a_{32} \end{pmatrix} \begin{pmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \end{pmatrix} = \begin{pmatrix} \dots & \dots & \dots \\ \dots & c_{22} & \dots \\ \dots & \dots & \dots \end{pmatrix} \quad c_{i,j} = \sum_{l=1}^m a_{i,l} \cdot b_{l,j}$$

- **Problem:** Compute a product of many matrices **efficiently**



# Matrix Chain Multiplication [MCM]: The Problem

- ↗ **Input:** Matrices  $T_1, T_2, \dots, T_n$ , each  $T_i$  of size  $d_{i-1} \times d_i$
- ↗ **Output:** Fully *parenthesized* product  $T_1 T_2 \dots T_n$  that minimizes the number of scalar multiplications.
- ↗ A product of matrices is fully parenthesized if it is either
  - a) a single matrix, or
  - b) the product of 2 fully parenthesized matrix products surrounded by parentheses.
- ↗ Example:  $T_1 \ T_2 \ T_3 \ T_4$  can be fully parenthesized as:
  - 1.  $(T_1 \ (T_2 \ (T_3 \ T_4 \ )))$
  - 2.  $(T_1 \ ((T_2 \ T_3 \ ) T_4 \ ))$
  - 3.  $((T_1 \ T_2 \ ) (T_3 \ T_4 \ ))$
  - 4.  $((T_1 \ (T_2 \ T_3 \ )) T_4 \ )$
  - 5.  $((((T_1 \ T_2 \ ) T_3 \ ) T_4 \ ))$

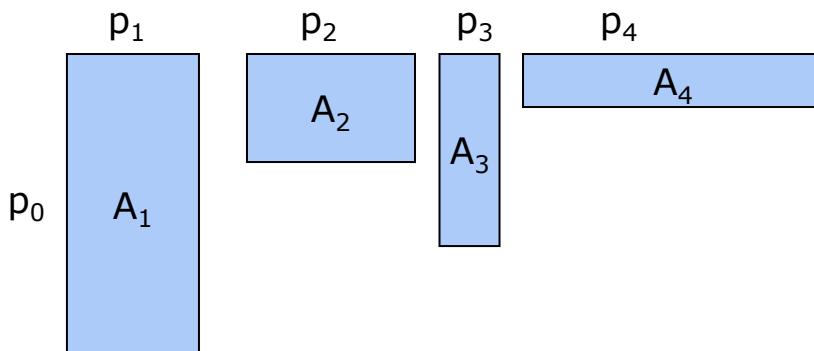


# Matrix Chain Multiplication [MCM]: The Problem

Suppose size of  $A_i$  is  $p_{i-1}$  by  $p_i$ .

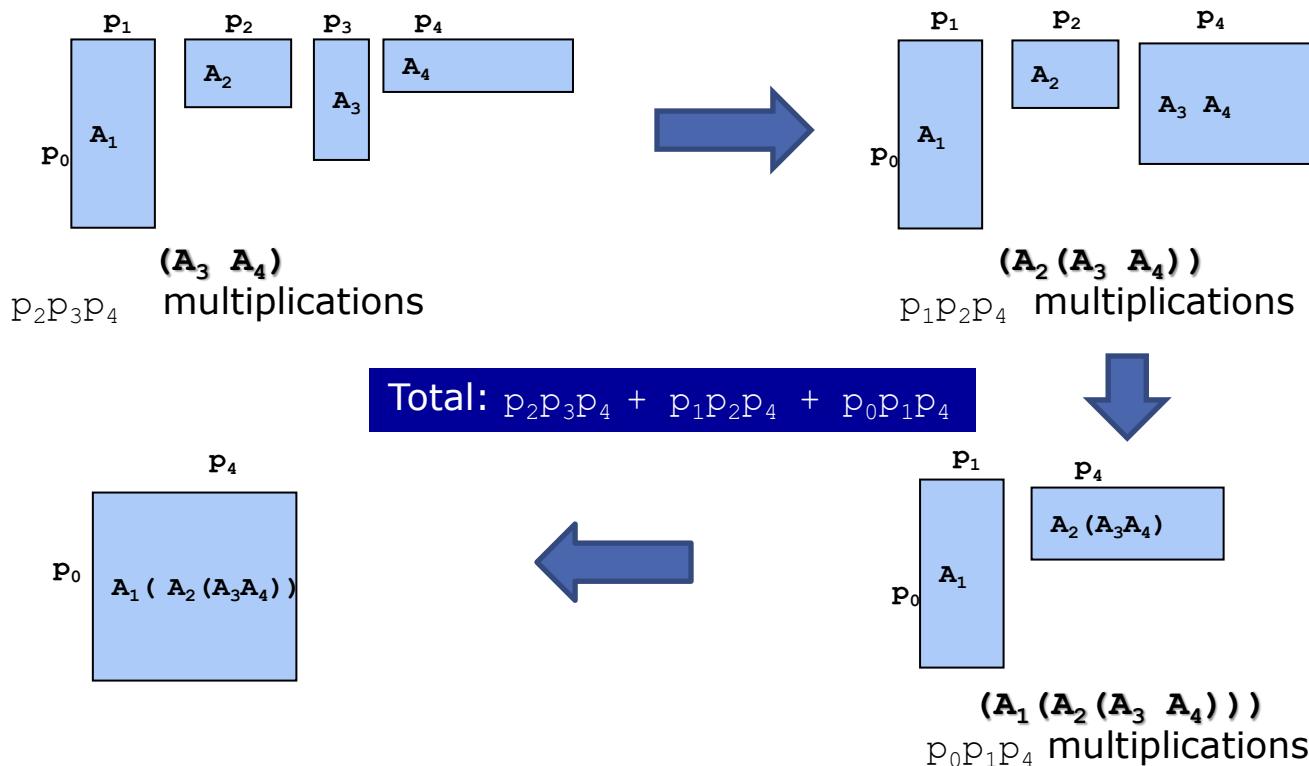
$$(A_1 \ ( \ A_2 \ ( \ A_3 \ A_4 \ ) \ ) \ ) : p_2 \ p_3 \ p_4 + p_1 p_2 p_4 + p_0 p_1 p_4$$

$$(( (A_1 \ A_2) \ A_3) \ A_4) : p_0 \ p_1 \ p_2 + p_0 p_2 p_3 + p_0 p_3 p_4$$

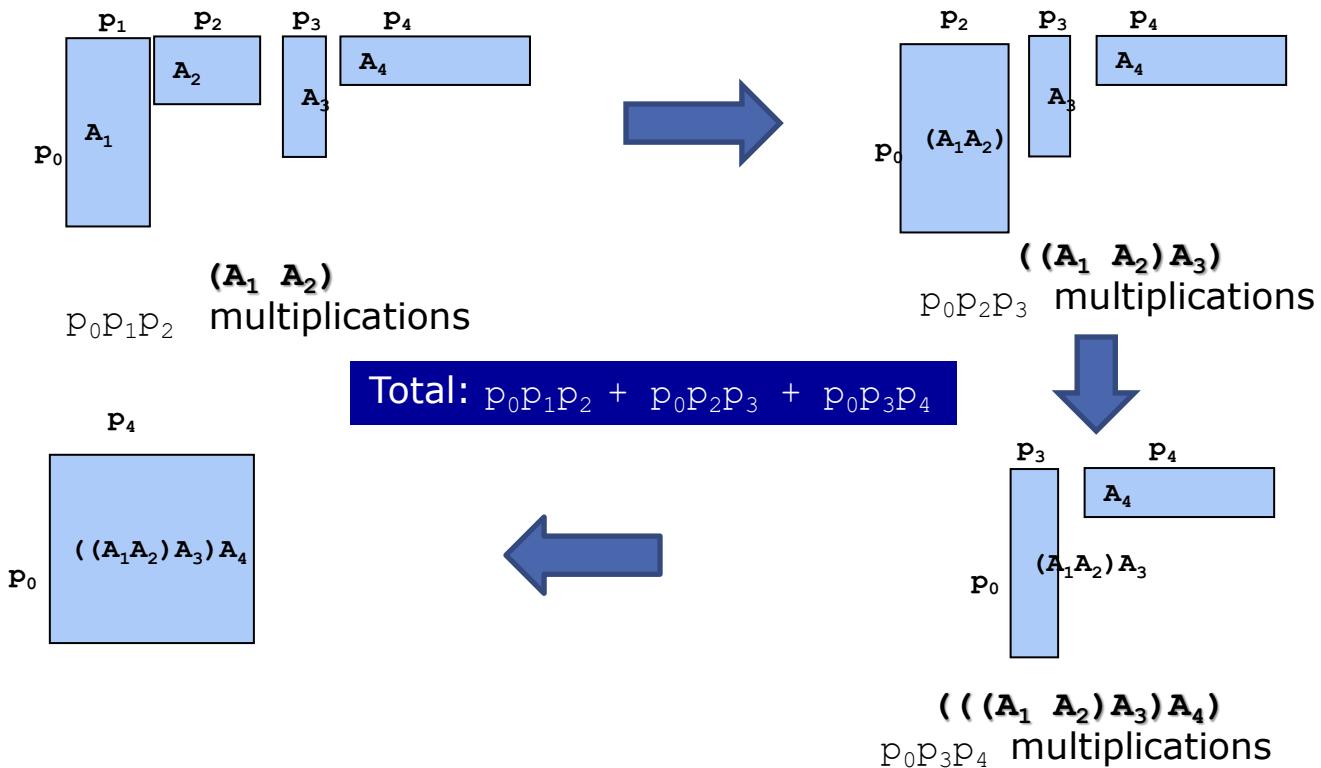




# Matrix Chain Multiplication [MCM]: $(A_1(A_2(A_3 A_4)))$

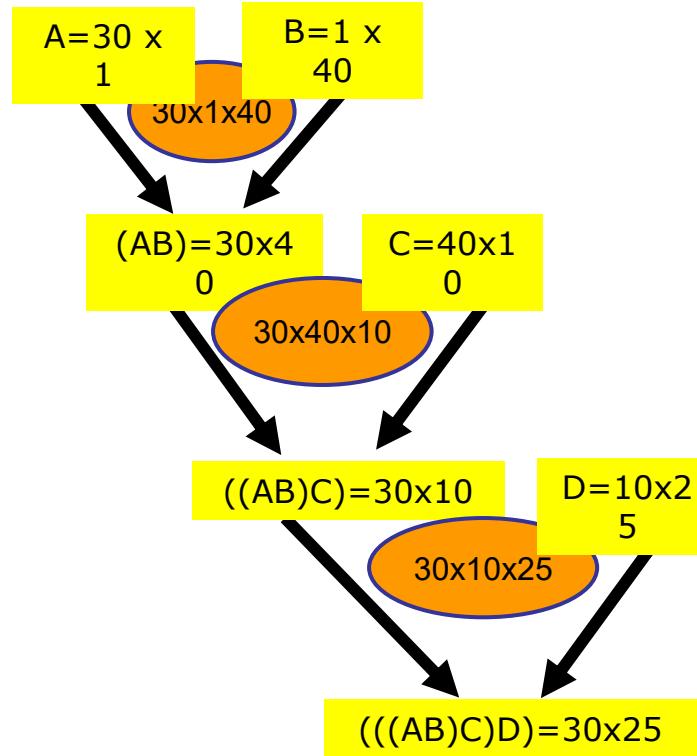


# Matrix Chain Multiplication [MCM]: (((A<sub>1</sub> A<sub>2</sub>) A<sub>3</sub>) A<sub>4</sub>)



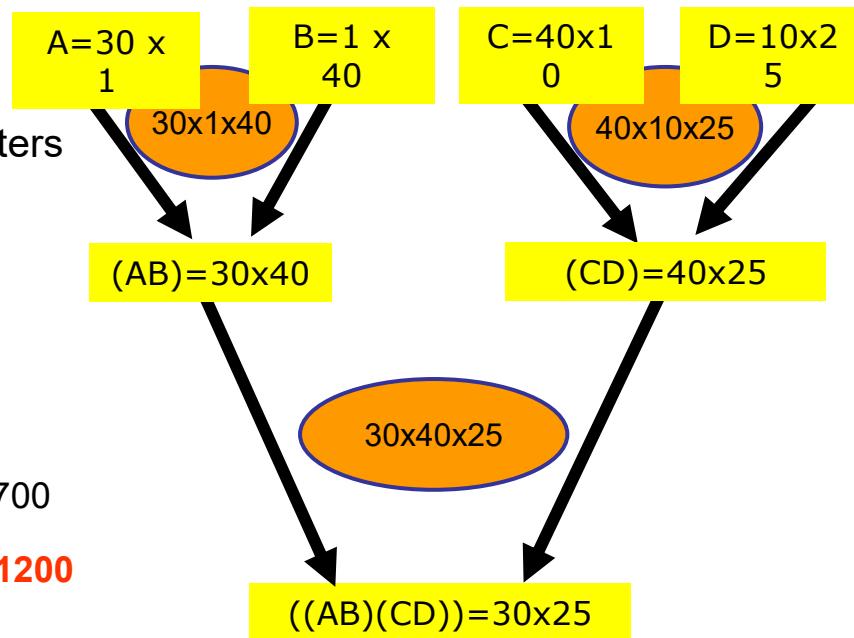
# MCM: Parenthesization

- # Some Preliminaries
- # Matrix multiplication is ***associative***
  - #  $(AB)C = A(BC)$
- # The **parenthesization** matters
- # Consider  $A \times B \times C \times D$ , where
  - #  $A$  is  $30 \times 1$ ,  $B$  is  $1 \times 40$ ,
  - #  $C$  is  $40 \times 10$ ,  $D$  is  $10 \times 25$
- # Costs:
  - #  $((AB)C)D$   
 $\begin{matrix} -120 & +1200 & +7500 = 20700 \\ 0 & 0 \end{matrix}$



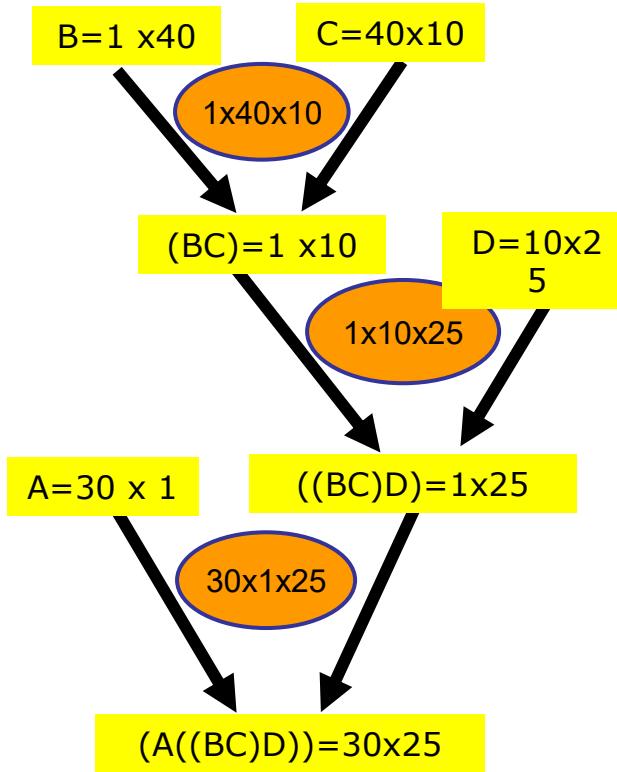
# MCM: Parenthesization

- # Some Preliminaries
- # Matrix multiplication is **associative**
  - #  $(AB)C = A(BC)$
- # The **parenthesization** matters
- # Consider  $A \times B \times C \times D$ , where
  - #  $A$  is  $30 \times 1$ ,  $B$  is  $1 \times 40$ ,
  - #  $C$  is  $40 \times 10$ ,  $D$  is  $10 \times 25$
- # Costs:
  - #  $((AB)C)D$   
 $= 1200 + 12000 + 7500 = 20700$
  - #  $((AB)(CD))$   
 $= 120 + 1000 + 3000 = 41200$   
0            0            0



# MCM: Parenthesization

- # Some Preliminaries
- # Matrix multiplication is **associative**
  - #  $(AB)C = A(BC)$
- # The **parenthesization** matters
- # Consider  $A \times B \times C \times D$ , where
  - #  $A$  is  $30 \times 1$ ,  $B$  is  $1 \times 40$ ,
  - #  $C$  is  $40 \times 10$ ,  $D$  is  $10 \times 25$
- # Costs:
  - #  $((AB)C)D)$   
 $= 1200 + 12000 + 7500 = 20700$
  - #  $((AB)(CD))$   
 $= 1200 + 10000 + 30000 = 41200$
  - #  $(A((BC)D))$   
**=400 + 250 + 750 = 1400**





# MCM: Parenthesization

- ↗ Some Preliminaries
- ↗ Matrix multiplication is *associative*
  - ↗  $(AB)C = A(BC)$
- ↗ The **parenthesization** matters
- ↗ Consider  $A \times B \times C \times D$ , where
  - ↗ **A** is  $30 \times 1$ , **B** is  $1 \times 40$ ,
  - ↗ **C** is  $40 \times 10$ , **D** is  $10 \times 25$
- ↗ Costs:
  - ↗  $((AB)C)D)$   
 $= 1200 + 12000 + 7500 = 20700$
  - ↗  $((AB)(CD))$   
 $= 1200 + 10000 + 30000 = 41200$
  - ↗  $(A((BC)D))$   
 $= 400 + 250 + 750 = 1400$

- We need to optimally parenthesize  $T_1 \times T_2 \times \dots \times T_n$ , where  $T_i$  is a  $d_{i-1} \times d_i$  matrix.
- According to the given example
  - $\mathbf{d} = \{30, 1, 40, 10, 25\}$
  - $T_1 \times T_2 \times T_3 \times T_4$  where
    - $T_1 = d_{1-1} \times d_1 = d_0 \times d_1 = 30 \times 1$
    - $T_2 = d_{2-1} \times d_2 = d_1 \times d_2 = 1 \times 40$
    - $T_3 = d_{3-1} \times d_3 = d_2 \times d_3 = 40 \times 10$
    - $T_4 = d_{4-1} \times d_4 = d_3 \times d_4 = 10 \times 25$
- Costs:
  - $((T_1 T_2) T_3) T_4 = 20700$
  - $(T_1 T_2) (T_3 T_4) = 41200$
  - $T_1 ((T_2 T_3) T_4) = 1400$

# MCM: Parenthesization

# Let the number of different parenthesizations,  $P(n)$ .

# Then,  $P(n) = \begin{cases} 1 & \text{if } n=1 \\ \sum_{k=1}^{n-1} P(k)P(n-k) & \text{if } n \geq 2 \end{cases}$

# Using ***Generating Function*** we have,

⊕  $P(n) = C(n-1)$ , the  $(n-1)^{\text{th}}$  ***Catalan Number*** where

$$C(n) = 1 / (n+1) C_n^{2n} = \Omega(4^n / n^{3/2})$$

# Exhaustively checking all possible parenthesizations take exponential time!



# MCM :: Step 1: Characterize Optimal Sub-structure

- ↗ Let  $M(i, j)$  be the *minimum* number of multiplications necessary to compute  $T_{i..j}$   
 $= T_i \times \dots \times T_j$
- ↗ Key observations
  - ↗ The outermost parenthesis partitions the chain of matrices  $(i, j)$  at some  $k$ , ( $i \leq k < j$ ) :  $(T_{i..k}) (T_{k+1..j})$
  - ↗ The optimal parenthesization of matrices  $(i, j)$  is also optimal on either side of  $k$ ; i.e., for matrices  $(i, k)$  and  $(k+1, j)$
  - ↗ Within the optimal parenthesization of  $T_{i..j}$ ,
    - (a) the parenthesization of  $T_{i..k}$  must be optimal
    - (b) the parenthesization of  $T_{k+1..j}$  must be optimal
  - ↗ Why?



## MCM :: Step 2: Recursive (Recurrence) Formulation

- ↗ Need to find  $T_{1..n}$
- ↗ Let  $M(i, j)$  = minimum # of scalar multiplications needed to compute  $T_{i..j}$
- ↗ Since  $T_{i..j}$  can be obtained by breaking it into  $T_{i..k}$  &  $T_{k+1..j}$ , we have

$$M(i, j) = \begin{cases} 0 & \text{if } i = j \\ \min_{i \leq k < j} \{ M(i, k) + M(k+1, j) + d_{i-1} d_k d_j \} & \text{if } i < j \end{cases}$$

- ↗ Note: The sizes of  $T_{i..k}$  is  $d_{i-1} \times d_k$  and  $T_{k+1..j}$  is  $d_k \times d_j$  ,  
and  $T_{i..k} T_{k+1..j}$  is  $d_{i-1} \times d_j$  after  $d_{i-1} d_k d_j$  scalar multiplications.
- ↗ Let  $s(i, j)$  be the value  $k$  where the optimal split occurs
- ↗ A direct recursive implementation is exponential – a lot of duplicated work.
- ↗ But there are only few different subproblems  $(i, j)$ : one solution for each choice of  $i$  and  $j$  ( $i < j$ ).



## MCM::Step 2: Recursive (Recurrence) Formulation

Recursive-Matrix-Chain( $d, i, j$ )

```
1  if  $i = j$  then
2      return 0;
3   $M[i, j] = \infty;$ 
4  for  $k = i$  to  $j-1$  do
5       $q = \text{Recursive-Matrix-Chain}(d, i, k) +$ 
           $\text{Recursive-Matrix-Chain}(d, k+1, j) +$ 
           $d_{i-1}d_kd_j$ 
6      if  $q < M[i, j]$  then
7           $M[i, j] = q ;$ 
           $S[i, j] = k ;$ 
8  return  $M[i, j] ;$ 
```



## MCM :: Step 2: Recursive (Recurrence) Formulation

↗ Overlapping Subproblems

↗ Let  $T(n)$  be the time complexity of

**Recursive-Matrix-Chain (d, 1, n)**

↗ For  $n > 1$ , we have

$$T(n) = 1 + \sum_{k=1}^{n-1} (T(k) + T(n-k) + 1)$$

a) 1 is used to cover the cost of lines 1-3, and 8

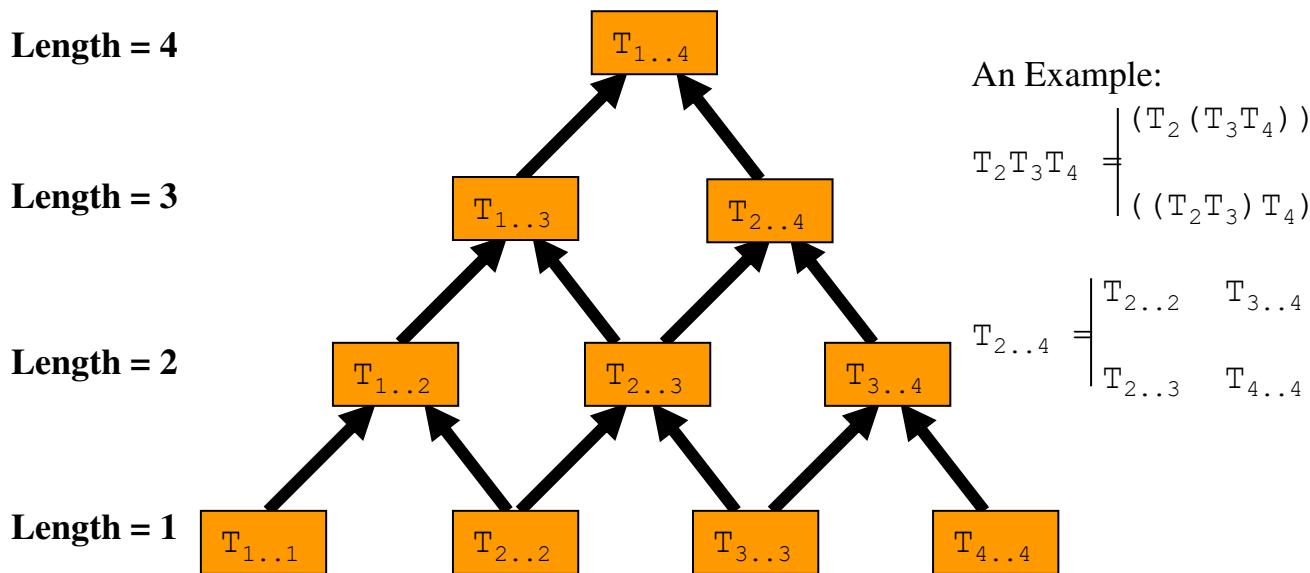
b) 1 is used to cover the cost of lines 6-7

↗ Using substitution, we can show that  $T(n) \geq 2^{n-1}$

↗ Hence  $T(n) = \Omega(2^n)$

# MCM :: Step 3: Computing Optimal Costs

- ↗ To compute  $T_{i..j}$  we need only values for subproblems of length  $< i-j$ .
- ↗ Solve subproblems in the ***increasing length*** of subproblems: first subproblems of length 2, then of length 3 and so on.





## MCM :: Step 3: Computing the Optimal Costs

- Idea: store the optimal cost  $M(i, j)$  for each subproblem in a 2d array  $M[1..n, 1..n]$ 
  - Trivially  $M(i, i) = 0$ ,  $1 \leq i \leq n$
  - To compute  $M(i, j)$ , where  $i - j = L$ , we need only values of  $M$  for subproblems of length  $< L$ .
  - Thus we have to solve subproblems in the **increasing length** of subproblems: first subproblems of length 2, then of length 3 and so on.
- To reconstruct an optimal parenthesization for each pair  $(i, j)$  we record in  $s[i, j] = k$  the optimal split into two subproblems  $(i, k)$  and  $(k+1, j)$



## MCM :: Step 3: Computing the Optimal Costs

**Matrix-Chain-Order( *d* )**

```
01 n = length[d]-1                                // d is the array of matrix sizes
02 for i = 1 to n do
03   M[i,i] = 0                                // no multiplication for 1 matrix
04 for len = 2 to n do                            // len is length of sub-chain
05   for i = 1 to n-len+1 do                      // i: start of sub-chain
06     j = i+len-1                                // j: end of sub-chain
07   M[i,j] = ∞
08   for k = i to j-1 do
09     q = M[i,k]+M[k+1,j]+di-1dkdj
10     if q < M[i,j] then
11       M[i,j] = q
12       s[i,j] = k
13 return M and s
```

Time complexity = O(n<sup>3</sup>)



## MCM :: Step 3: Computing the Optimal Costs

- After the execution:  $M[1, n]$  contains the value of an optimal solution and  $s$  contains optimal subdivisions (choices of  $k$ ) of any subproblem into two sub-subproblems
- Let us run the algorithm on the six matrices:

Matrix	Dimension
T1	30 X 35
T2	35 X 15
T3	15 X 5
T4	5 X 10
T5	10 X 20
T6	20 X 25

$$d = \{ 30, 35, 15, 5, 10, 20, 25 \}$$

- See CLRS Figure 15.3

# Simulation-MCM

**Matrix-Chain-Order( d )**

```

01 n = length[d]-1
02 for i = 1 to n do
03   M[i,i] = 0
04 for len = 2 to n do
05   for i = 1 to n-len+1 do
06     j = i+len-1
07     M[i,j] = ∞
08     for k = i to j-1 do
09       q = M[i,k]+M[k+1,j]+di-1dkdj
10      if q < M[i,j] then
11        M[i,j] = q
12        s[i,j] = k
13 return M and s
  
```

d	0	1	2	3	4	5	6
	30	35	15	5	10	20	25

M	1	2	3	4	5	6
1						
2						
3						
4						
5						
6						

S	1	2	3	4	5	6
1						
2						
3						
4						
5						
6						

# Simulation-MCM

**Matrix-Chain-Order( d )**

```

01 n = length[d]-1
02 for i = 1 to n do
03   M[i,i] = 0
04 for len = 2 to n do
05   for i = 1 to n-len+1 do
06     j = i+len-1
07     M[i,j] = ∞
08     for k = i to j-1 do
09       q = M[i,k]+M[k+1,j]+di-1dkdj
10      if q < M[i,j] then
11        M[i,j] = q
12        s[i,j] = k
13 return M and s

```

d	0	1	2	3	4	5	6
	30	35	15	5	10	20	25

M	1	2	3	4	5	6
1	0					
2		0				
3			0			
4				0		
5					0	
6						0

s	1	2	3	4	5	6
1						
2						
3						
4						
5						
6						

# Simulation-MCM

**Matrix-Chain-Order( d )**

```

01 n = length[d]-1
02 for i = 1 to n do
03   M[i,i] = 0
04 for len = 2 to n do
05   for i = 1 to n-len+1 do
06     j = i+len-1
07     M[i,j] = ∞
08     for k = i to j-1 do
09       q = M[i,k]+M[k+1,j]+d_{i-1}d_kd_j
10      if q < M[i,j] then
11        M[i,j] = q
12        s[i,j] = k
13 return M and s

```

d	0	1	2	3	4	5	6
	30	35	15	5	10	20	25

M	1	2	3	4	5	6
1	0	1575 0				
2		0	2625			
3			0	750		
4				0	1000	
5					0	5000
6						0

len=2, [i,j]=[1,2] to [5,6], k = 1 to 5

$$\begin{aligned}
 M[1,2] &= M[1,1]+M[2,2]+d_0 \cdot d_1 \cdot d_2 = 0+0+30 \cdot 35 \cdot 15 = 15750; \\
 M[2,3] &= M[2,2]+M[3,3]+d_1 \cdot d_2 \cdot d_3 = 0+0+35 \cdot 15 \cdot 5 = 2625; \\
 M[3,4] &= M[3,3]+M[4,4]+d_2 \cdot d_3 \cdot d_4 = 0+0+15 \cdot 5 \cdot 10 = 750; \\
 M[4,5] &= M[4,4]+M[5,5]+d_3 \cdot d_4 \cdot d_5 = 0+0+5 \cdot 10 \cdot 20 = 1000; \\
 M[5,6] &= M[5,5]+M[6,6]+d_4 \cdot d_5 \cdot d_6 = 0+0+10 \cdot 20 \cdot 25 = 5000;
 \end{aligned}$$

s	1	2	3	4	5	6
1		1				
2			2			
3				3		
4					4	
5						5
6						

# Simulation-MCM

**Matrix-Chain-Order( d )**

```

01 n = length[d]-1
02 for i = 1 to n do
03   M[i,i] = 0
04 for len = 2 to n do
05   for i = 1 to n-len+1 do
06     j = i+len-1
07     M[i,j] = ∞
08     for k = i to j-1 do
09       q = M[i,k]+M[k+1,j]+d_{i-1}d_kd_j
10      if q < M[i,j] then
11        M[i,j] = q
12        s[i,j] = k
13 return M and s

```

len=3, [i,j]=[1,3] to [4,6], k = [1,2] to [4,5]

$$\begin{aligned}
 M[1,3] &= M[1,1]+M[2,3]+d_0 \cdot d_1 \cdot d_3 = 0 + 2625 + 30 \cdot 35 \cdot 5 = 7875; \\
 &= M[1,2]+M[3,3]+d_0 \cdot d_2 \cdot d_3 = 15750 + 0 + 30 \cdot 15 \cdot 5 = 18000; \\
 M[2,4] &= M[2,2]+M[3,4]+d_1 \cdot d_2 \cdot d_4 = 0 + 750 + 35 \cdot 15 \cdot 10 = 6000; \\
 &= M[2,3]+M[4,4]+d_1 \cdot d_3 \cdot d_4 = 2625 + 0 + 35 \cdot 5 \cdot 10 = 4375; \\
 M[3,5] &= M[3,3]+M[4,5]+d_2 \cdot d_3 \cdot d_5 = 0 + 1000 + 15 \cdot 5 \cdot 20 = 2500; \\
 &= M[3,4]+M[5,5]+d_2 \cdot d_4 \cdot d_5 = 750 + 0 + 15 \cdot 10 \cdot 20 = 3750; \\
 M[4,6] &= M[4,4]+M[5,6]+d_3 \cdot d_4 \cdot d_6 = 0 + 5000 + 5 \cdot 10 \cdot 25 = 6250; \\
 &= M[4,5]+M[6,6]+d_3 \cdot d_5 \cdot d_6 = 1000 + 0 + 5 \cdot 20 \cdot 25 = 3500;
 \end{aligned}$$

d	0	1	2	3	4	5	6
	30	35	15	5	10	20	25

M	1	2	3	4	5	6
1	0	1575 0	7875			
2		0	2625	4375		
3			0	750	2500	
4				0	1000	3500
5					0	5000
6						0

s	1	2	3	4	5	6
1		1	1			
2			2	3		
3				3	3	
4					4	5
5						5
6						

# Simulation-MCM

**Matrix-Chain-Order( d )**

```

01 n = length[d]-1
02 for i = 1 to n do
03   M[i,i] = 0
04 for len = 2 to n do
05   for i = 1 to n-len+1 do
06     j = i+len-1
07     M[i,j] = ∞
08     for k = i to j-1 do
09       q = M[i,k]+M[k+1,j]+di-1dkdj
10      if q < M[i,j] then
11        M[i,j] = q
12        s[i,j] = k
13 return M and s

```

len=4, [i,j]=[1,4] to [3,6], k = [1,2,3] to [3,4,5]

$$\begin{aligned}
 M[1,4] &= M[1,1]+M[2,4]+d_0 \cdot d_1 \cdot d_4 = 0 + 4375 + 30 \cdot 35 \cdot 10 = 14875; \\
 &= M[1,2]+M[3,4]+d_0 \cdot d_2 \cdot d_4 = 15750 + 750 + 30 \cdot 15 \cdot 10 = 21000; \\
 &= M[1,3]+M[4,4]+d_0 \cdot d_3 \cdot d_4 = 7875 + 0 + 30 \cdot 5 \cdot 10 = 9375; \\
 M[2,5] &= M[2,2]+M[3,5]+d_1 \cdot d_2 \cdot d_5 = 0 + 2500 + 35 \cdot 15 \cdot 20 = 13000; \\
 &= M[2,3]+M[4,5]+d_1 \cdot d_3 \cdot d_5 = 2625 + 1000 + 35 \cdot 5 \cdot 20 = 7125; \\
 &= M[2,4]+M[4,5]+d_1 \cdot d_4 \cdot d_5 = 4375 + 1000 + 35 \cdot 10 \cdot 20 = 12375; \\
 M[3,6] &= M[3,3]+M[4,6]+d_2 \cdot d_3 \cdot d_6 = 0 + 3500 + 15 \cdot 5 \cdot 25 = 5375; \\
 &= M[3,4]+M[5,6]+d_2 \cdot d_4 \cdot d_6 = 750 + 5000 + 15 \cdot 10 \cdot 25 = 9500; \\
 &= M[3,5]+M[6,6]+d_2 \cdot d_5 \cdot d_6 = 2500 + 0 + 15 \cdot 20 \cdot 25 = 10000;
 \end{aligned}$$

d	0	1	2	3	4	5	6
	30	35	15	5	10	20	25

M	1	2	3	4	5	6
1	0	1575 0	7875	9375		
2		0	2625	4375	7125	
3			0	750	2500	5375
4				0	1000	3500
5					0	5000
6						0

s	1	2	3	4	5	6
1		1	1	3		
2			2	3	3	
3				3	3	3
4					4	5
5						5
6						

# Simulation-MCM

**Matrix-Chain-Order( d )**

```

01 n = length[d]-1
02 for i = 1 to n do
03   M[i,i] = 0
04 for len = 2 to n do
05   for i = 1 to n-len+1 do
06     j = i+len-1
07     M[i,j] = ∞
08     for k = i to j-1 do
09       q = M[i,k]+M[k+1,j]+d_{i-1}d_kd_j
10      if q < M[i,j] then
11        M[i,j] = q
12        s[i,j] = k
13 return M and s

```

d	0	1	2	3	4	5	6
	30	35	15	5	10	20	25

M	1	2	3	4	5	6
1	0	1575 0	7875	9375	1187 5	
2		0	2625	4375	7125	10500
3			0	750	2500	5375
4				0	1000	3500
5					0	5000
6						0

len=5, [i,j]=[1,5] to [2,6], k = [1,2,3,4] to [2,3,4,5]

$$\begin{aligned}
 M[1,5] &= M[1,1]+M[2,5]+d_0 \cdot d_1 \cdot d_5 = 0 + 7125 + 30 \cdot 35 \cdot 20 = 28125; \\
 &= M[1,2]+M[3,5]+d_0 \cdot d_2 \cdot d_5 = 15750 + 2500 + 30 \cdot 15 \cdot 20 = 27250; \\
 &= M[1,3]+M[4,5]+d_0 \cdot d_3 \cdot d_5 = 7875 + 1000 + 30 \cdot 5 \cdot 20 = 11875; \\
 &= M[1,4]+M[5,5]+d_0 \cdot d_4 \cdot d_5 = 9375 + 0 + 30 \cdot 10 \cdot 20 = 15375;
 \end{aligned}$$

$$\begin{aligned}
 M[2,6] &= M[2,2]+M[3,6]+d_1 \cdot d_2 \cdot d_6 = 0 + 5375 + 35 \cdot 15 \cdot 25 = 18500; \\
 &= M[2,3]+M[4,6]+d_1 \cdot d_3 \cdot d_6 = 2625 + 3500 + 35 \cdot 5 \cdot 25 = 10500; \\
 &= M[2,4]+M[5,6]+d_1 \cdot d_4 \cdot d_6 = 4375 + 5000 + 35 \cdot 10 \cdot 25 = 18125; \\
 &= M[2,5]+M[6,6]+d_1 \cdot d_5 \cdot d_6 = 0 + 5375 + 35 \cdot 20 \cdot 25 = 22875;
 \end{aligned}$$

s	1	2	3	4	5	6
1		1	1	3	3	
2			2	3	3	3
3				3	3	3
4					4	5
5						5
6						0



# Simulation-MCM

d	0	1	2	3	4	5	6
	30	35	15	5	10	20	25

Matrix-Chain-Order( d )

```

01 n = length[d]-1
02 for i = 1 to n do
03   M[i,i] = 0
04 for len = 2 to n do
05   for i = 1 to n-len+1 do
06     j = i+len-1
07     M[i,j] = ∞
08     for k = i to j-1 do
09       q = M[i,k]+M[k+1,j]+d_{i-1}d_kd_j
10      if q < M[i,j] then
11        M[i,j] = q
12        s[i,j] = k
13 return M and s

```

M	1	2	3	4	5	6
1	0	1575 0	7875	9375	1187 5	15125
2		0	2625	4375	7125	10500
3			0	750	2500	5375
4				0	1000	3500
5					0	5000
6						0

len=6, [i,j]=[1,6], k = [1,2,3,4,5]

$$\begin{aligned}
 M[1,6] &= M[1,1]+M[2,6]+d_0*d_1*d_6 = 0+ 10500+30*35*25 = 36750; \\
 &= M[1,2]+M[3,6]+d_0*d_2*d_6 = 15750+ 5375+30*15*25 = 32375; \\
 &= M[1,3]+M[4,6]+d_0*d_3*d_6 = 7875+ 3500+30*5*25 = 15125; \\
 &= M[1,4]+M[5,6]+d_0*d_4*d_6 = 9375+ 5000+30*10*25 = 21875; \\
 &= M[1,5]+M[6,6]+d_0*d_5*d_6 = 11875+ 0+30*20*25 = 26875;
 \end{aligned}$$

S	1	2	3	4	5	6
1		1	1	3	3	3
2			2	3	3	3
3				3	3	3
4					4	5
5						5
6						



## MCM :: Step 4: Constructing Optimal Solution

- To get the optimal solution  $T_{1..6}$ ,  $s[]$  is used as follows:

$$T_{1..6}$$

$$= (T_{1..3} \ T_{4..6}) \quad ; \text{since } s[1,6] = 3$$

$$= ((T_{1..1} \ T_{2..3}) \ (T_{4..5} \ T_{6..6})) \quad ; \text{since } s[1,3] = 1 \text{ and } s[4,6] = 5$$

$$= ((T_1 \ (T_2 \ T_3)) \ ((T_4 \ T_5) \ T_6))$$

MCM can be solved in  $\mathcal{O}(n^3)$  time



# Matrix Chain Multiplication Problem

- ↗ Running time
  - ↗ It is easy to see that it is  $O(n^3)$  (three nested loops)
  - ↗ It turns out it is also  $\Omega(n^3)$
- ↗ Thus, a reduction from exponential time to polynomial time.



# Memoization

- ↗ ***Memoization is one way to deal with overlapping subproblems***
  - ↗ After computing the solution to a subproblem, store it in a table
  - ↗ Subsequent calls just do a table lookup
- ↗ **Can modify recursive algorithm to use memoization**
- ↗ **If we prefer recursion we can structure our algorithm as a recursive algorithm:**

```
MemoMCM(i,j)
1.  if i = j then return 0
2.  else if M[i,j] < ∞ then return M[i,j]
3.  else for k := i to j-1 do
4.      q := MemoMCM(i,k) +
              MemoMCM(k+1,j)
      + di-1dkdj
5.      if q < M[i,j] then
6.          M[i,j] := q
7.      return M[i,j]
```

- ↗ Initialize all elements to  $\infty$  and call **MemoMCM(i,j)**



# Memoization

- ↗ Memoization:
  - ↗ Solve the problem in a ***top-down*** fashion, but record the solutions to subproblems in a table.
- ↗ Pros and cons:
  - ↗ 😞 Recursion is usually slower than loops and uses stack space
  - ↗ 😊 Easier to understand
  - ↗ 😊 If not all subproblems need to be solved, you are sure that only the necessary ones are solved



# Longest Common Subsequence The Problem

- ↗ Given two sequences
  - ↗  $x = < x_1, x_2, \dots, x_m >$
  - ↗  $z = < z_1, z_2, \dots, z_k >$
- ↗  $z$  is a ***subsequence*** of  $x$  if
  - ↗  $z_j = x_{i(j)}$ , for all  $j = 1, \dots, k$
  - ↗ where  $<i(1), i(2), \dots, i(k)>$  is ***strictly increasing*** (but not required to be consecutive)
- ↗ Examples:
  - ↗ Let  $x = < A, B, C, B, D, A, B >$
  - ↗  $< A >$ ,  $< B >$ ,  $< C >$ , and  $< D >$  are subsequences.
  - ↗  $< C, A >$ ,  $< C, B >$ ,  $< C, B, A, B >$  are subsequences.
  - ↗ How many possible subsequences for a  $n$ -element sequence?



# Longest Common Subsequence Problem

- ↗ **Z** is a *common* subsequence of sequences **X** and **Y** if **Z** is a subsequence of both **X** and **Y**.
  
- ↗ Example:
  - ↗ Let **X** = < **A**, **B**, **C**, **B**, **D**, **A**, **B** > and **Y** = < **B**, **D**, **C**, **A**, **B**, **A** >
  - ↗ < **A** >, <**B**>, <**C**>, and <**D**> are common subsequences.
  - ↗ < **C**, **A** > is, but < **A**, **C** > is not.
  - ↗ < **B**, **C**, **A** > is a common subsequence
  - ↗ < **B**, **C**, **B**, **A** > is the longest common subsequence.
  
- ↗ Example:
  - ↗ **x** = "sariempioolcewe"
  - ↗ **y** = "westigmupsalrte"



# Longest Common Subsequence Problem

- ↗ **LCS:**
- ↗ Input: two sequences  $x[1..m]$  and  $y[1..n]$
- ↗ Output: longest common subsequence of  $x$  and  $y$  (denoted  $\text{LCS}(x, y)$ )
- ↗ Brute-force algorithm:
  - ↗ For every subsequence of  $x$ , check if it is a subsequence of  $y$
  - ↗  $2^m$  subsequences of  $x$  to check against  $n$  elements of  $y$  :  $O(n \cdot 2^m)$



# LCS: Optimal Sub-structure

- ↗ The  $i^{th}$  prefix of  $\mathbf{x} = \langle \mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_m \rangle$  is denoted  $\mathbf{x}_i = \langle \mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_i \rangle$ 
  - ↗  $\mathbf{x}_0$  is the **empty** sequence
  - ↗  $\mathbf{x}_m$  is the **whole** sequence  $\mathbf{x}$
- ↗ Theorem 15.1 (Optimal Sub-structure of LCS)
- ↗ Let  $\mathbf{X}=\langle \mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_m \rangle$ ,  $\mathbf{Y}=\langle \mathbf{y}_1, \mathbf{y}_2, \dots, \mathbf{y}_n \rangle$ , and  $\mathbf{Z}=\langle \mathbf{z}_1, \mathbf{z}_2, \dots, \mathbf{z}_k \rangle$  be **LCS**( $\mathbf{X}, \mathbf{Y}$ ).
  - (1) If  $\mathbf{x}_m = \mathbf{y}_n$ , then  $\mathbf{z}_k = \mathbf{x}_m = \mathbf{y}_n$  and  $\mathbf{Z}_{k-1}$  is **LCS**( $\mathbf{X}_{m-1}, \mathbf{Y}_{n-1}$ )
  - (2) if  $\mathbf{x}_m \neq \mathbf{y}_n$ , then  $\mathbf{z}_k \neq \mathbf{x}_m$  implies  $\mathbf{Z}$  is **LCS**( $\mathbf{X}_{m-1}, \mathbf{Y}$ )
  - (3) if  $\mathbf{x}_m \neq \mathbf{y}_n$ , then  $\mathbf{z}_k \neq \mathbf{y}_n$  implies  $\mathbf{Z}$  is **LCS**( $\mathbf{X}, \mathbf{Y}_{n-1}$ )



# LCS: Optimal Substructure

- ↗ We make  $Z$  to be empty and proceed from the ends of  $X_m = "x_1 x_2 \dots x_m"$  and  $Y_n = "y_1 y_2 \dots y_n"$
- ↗ If  $x_m = y_n$ , append this symbol to the beginning of  $Z$ , and find optimally **LCS ( $X_{m-1}, Y_{n-1}$ )**
- ↗ If  $x_m \neq y_n$ 
  - ↗ Skip either a letter from **X**
  - ↗ or a letter from **Y**
  - ↗ Decide which decision to do by comparing **LCS ( $X_m, Y_{n-1}$ )** and **LCS ( $X_{m-1}, Y_n$ )**
- ↗ Starting from beginning is equivalent.



# LCS: Optimal Substructure

- ↗ **LCS** has an optimal sub-structure defined by *prefixes* of **X** and **Y**
- ↗ The sub-problems of finding **LCS** ( $X_{m-1}, Y_n$ ) and **LCS** ( $X_m, Y_{n-1}$ ) share a common sub-sub-problem of finding **LCS** ( $X_{m-1}, Y_{n-1}$ ). They are overlapping.
- ↗ **Simplify:** just worry about **LCS** length for now
  - ↗ Define  $c[i, j] = \text{length of } \text{LCS}(X_i, Y_j)$
  - ↗ So  $c[m, n] = \text{length of } \text{LCS}(X, Y)$



# LCS: Recurrence

- Define  $c[i, j]$  = length of LCS of  $x[1..i], y[1..j]$

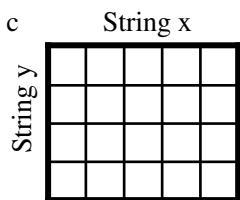
$$c[i, j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ c[i-1, j-1] + 1 & \text{if } i, j > 0 \text{ and } x_i = y_j \\ \max\{c[i, j-1], c[i-1, j]\} & \text{if } i, j > 0 \text{ and } x_i \neq y_j \end{cases}$$

- Note that the conditions in the problem restrict sub-problems  
(if  $x_i = y_i$  we consider  $x_{i-1}$  and  $y_{i-1}$ , etc)
- Use  $b[i, j]$  to remember where to extract an element of an LCS.



# LCS: Recurrence

```
int lcsMemo(int i, int j) {  
    if (c[i][j] != -1) return c[i][j]  
    else if (x[i] == y[j]) {  
        c[i][j] = lcsMemo(i-1, j-1) + 1  
        return c[i][j]  
    }  
    else {  
        c[i][j]=max(lcsMemo(i-1, j), lcsMemo(i, j-1))  
        return c[i][j]  
    }  
}
```



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

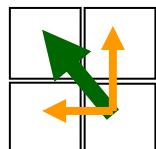


# LCS: Computing Length

```
LCS-Length(X, Y, m, n)
1  for i←1 to m do
2      c[i,0] ← 0
3  for j←0 to n do
4      c[0,j] ← 0
5  for i←1 to m do
6      for j←1 to n do
7          if xi = yj then
8              c[i,j] ← c[i-1,j-1]+1
9              b[i,j] ← "copy"
10         else if c[i-1,j] ≥ c[i,j-1]
then
11             c[i,j] ← c[i-1,j]
12             b[i,j] ← "skipX"
13         else
14             c[i,j] ← c[i,j-1]
15             b[i,j] ← "skipY"
16 return c, b
```

# LCS: Example

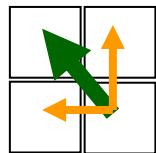
$$c[i,j] = \begin{cases} 0, & \text{if } i=0, j=0 \\ c[i-1, j-1] + 1 & \text{if } i, j > 0 \text{ and } x_i = y_j \\ \max ( c[i, j-1], c[i-1, j] ) & \text{if } i, j > 0 \text{ and } x_i \neq y_j \end{cases}$$



	X	a	b	b	c	a	a	c
Y	0	0	0	0	0	0	0	0
a	0							
c	0							
c	0							
b	0							
c	0							
c	0							
a	0							

# LCS: Example

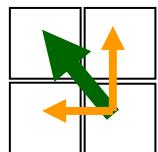
$$c[i,j] = \begin{cases} 0, & \text{if } i=0, j=0 \\ c[i-1, j-1] + 1 & \text{if } i, j > 0 \text{ and } x_i = y_j \\ \max ( c[i, j-1], c[i-1, j] ) & \text{if } i, j > 0 \text{ and } x_i \neq y_j \end{cases}$$



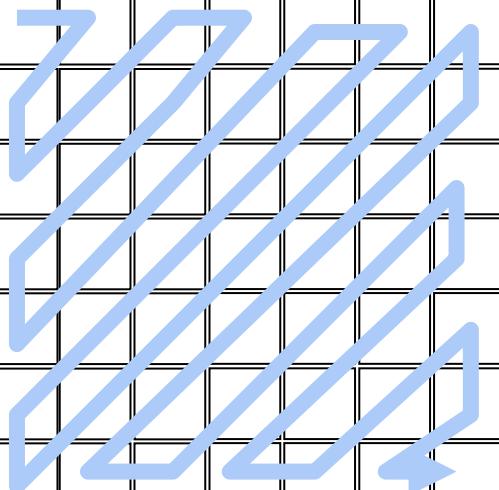
	a	b	b	c	a	a	c
0	0	0	0	0	0	0	0
a	0						
c	0						
c	0						
b	0						
c	0						
c	0						
a	0						

# LCS: Example

$$c[i,j] = \begin{cases} 0, & \text{if } i=0, j=0 \\ c[i-1, j-1] + 1 & \text{if } i, j > 0 \text{ and } x_i = y_j \\ \max ( c[i, j-1], c[i-1, j] ) & \text{if } i, j > 0 \text{ and } x_i \neq y_j \end{cases}$$

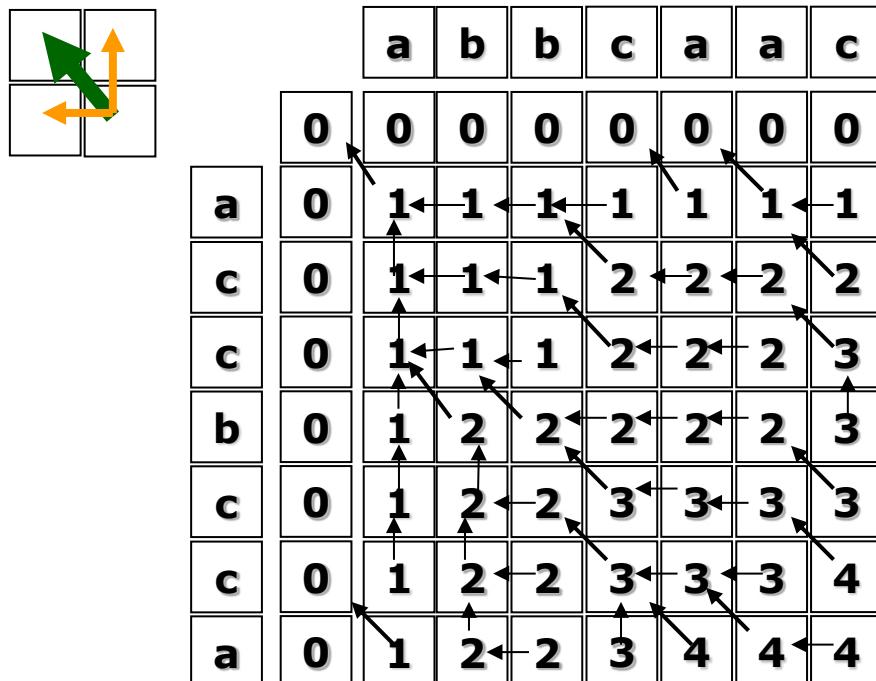


	a	b	b	c	a	a	c
a	0	0	0	0	0	0	0
c	0						
c	0						
b	0						
c	0						
c	0						
a	0						



# LCS: Example

$c[i, j] =$	$0, \quad \text{if } i=0, j=0$
	$c[i-1, j-1]+1 \quad \text{if } i, j > 0 \text{ and } x_i = y_j$
	$\max(c[i, j-1], c[i-1, j]) \quad \text{if } i, j > 0 \text{ and } x_i \neq y_j$





# Constructing an LCS

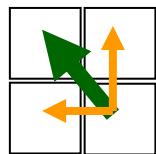
```
Print-LCS(b, X, i, j)
```

```
1 if i = 0 or j = 0 then
2     return
3 if b[i,j] = "copy" then
4     Print-LCS(b,X,i-1,j-1)
5         print x[i]
6 elseif b[i,j] = "skipX" then
7     Print-LCS(b,X,i-1,j)
8 else
9     Print-LCS(b,X,i,j-1)
```

length[X] = m, length[Y] = n,  
Call Print-LCS(b, X, n, m) to  
construct LCS  
Time complexity: O (m+n).

# LCS: Example

$c[i, j] = \begin{cases} 0, & \text{if } i=0, j=0 \\ c[i-1, j-1]+1 & \text{if } i, j > 0 \text{ and } x_i = y_j \\ \max(c[i, j-1], c[i-1, j]) & \text{if } i, j > 0 \text{ and } x_i \neq y_j \end{cases}$	$\text{if } i=0, j=0$ $\text{if } i, j > 0 \text{ and } x_i = y_j$ $\text{if } i, j > 0 \text{ and } x_i \neq y_j$
--	--



	a	b	b	c	a	a	c
a	0	0	0	0	0	0	0
c	0	1	1	1	1	1	1
c	0	1	1	1	2	2	2
b	0	1	2	2	2	2	3
c	0	1	2	2	3	3	3
c	0	1	2	2	3	3	4
a	0	1	2	2	3	4	4



# Longest Common Subsequence

- ↗ There is a need to quantify how similar they are:
  - ↗ Comparing DNA sequences in studies of evolution of different species
  - ↗ Spell checkers
  - ↗ Editing



# Books

1. *Introduction to Algorithms, Third Edition, Thomas H. Cormen, Charle E. Leiserson, Ronald L. Rivest, Clifford Stein (CLRS)*.
2. *Fundamental of Computer Algorithms, Ellis Horowitz, Sartaj Sahni, Sanguthevar Rajasekaran (HSR)*



## References

- ↗ <https://www.radford.edu/~nokie/classes/360/dp-matrix-parens.html>
- ↗ <http://www.personal.kent.edu/~rmuhamma/Algorithms/MyAlgorithms/Dynamic/chainMatrixMult.htm>
- ↗ <https://www.ics.uci.edu/~eppstein/161/960229.html>
- ↗ CLRS: 15.2, 15.4

# Graphs and Trees

Course Code: CSC 2211

Course Title: Algorithms



**Dept. of Computer Science  
Faculty of Science and Technology**

<b>Lecturer No:</b>		<b>Week No:</b>	<b>10</b>	<b>Semester:</b>	<b>Spring 2019-2020</b>
<b>Lecturer:</b>	<i>Name &amp; email</i>				



# Lecture Outline

- Graph Basics
- Graph Searching
  - Depth First Search
  - Breadth First Search
- Topological Sort

# Graphs



## ● Graph

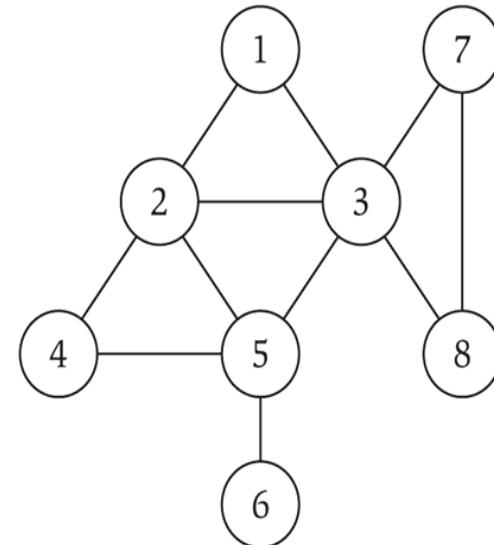
- **V = nodes** (vertices, points).
- **E = edges** (links, arcs) between pairs of nodes.
- Denoted by  $G = (V, E)$ .
- Captures pair wise relationship between objects.
- **Graph size** parameters:  $n = |V|$ ,  $m = |E|$ .

$V = \{ 1, 2, 3, 4, 5, 6, 7, 8 \}$

$E = \{ \{1,2\}, \{1,3\}, \{2,3\}, \{2,4\}, \{2,5\}, \{3,5\}, \{3,7\}, \{3,8\}, \{4,5\}, \{5,6\} \}$

$n = 8$

$m = 11$

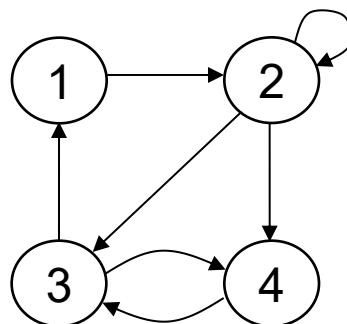


# Graphs

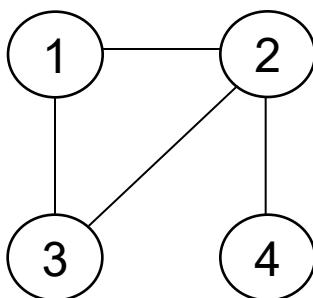
**Undirected Graph:** A graph whose edges are unordered pairs of vertices. That is, each edge connects two vertices where edge  $(u, v) = \text{edge } (v, u)$ .

**Directed Graph:** A graph whose edges are ordered pairs of vertices. That is, each edge can be followed from one vertex to another vertex where edge  $(u, v)$  goes from vertex  $u$  to vertex  $v$ .

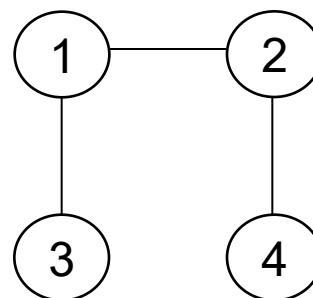
**Acyclic Graph:** A graph with no path that starts and ends at the same vertex.



Directed graph



Undirected graph



Acyclic graph

## Graph Example

A **directed** graph  $G = (V, E)$ , where

$V = \{1, 2, 3, 4, 5, 6\}$  and

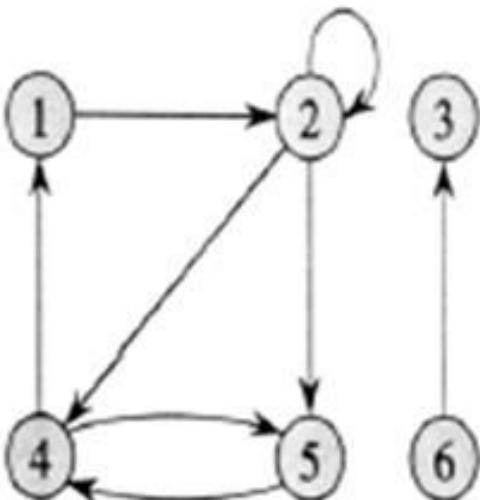
$E = \{(1,2), (2,2), (2,4), (2,5), (4,1), (4,5), (5,4), (6,3)\}$ .

The edge  $(2,2)$  is **self loop**.

Vertex 5 has **in-degree 2** and **out-degree 1**.

Vertex 4 is **adjacent** to vertex 5; and vice versa;

6 is not **adjacent** to any other vertex except 3.



(a)

## Graph Example

An **undirected** graph  $G = (V, E)$ , where

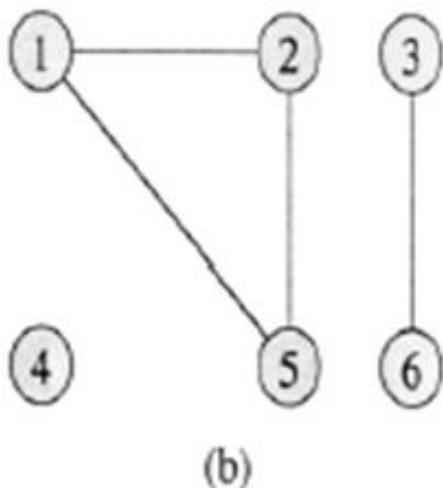
$V = \{1, 2, 3, 4, 5, 6\}$  and

$E = \{\{1,2\}, \{1,5\}, \{2,5\}, \{3,6\}\}$ .

The vertex 4 is **isolated**.

Vertex 1, 2, 5 has **degree 2**; vertex 3, 6 has **degree 1**; vertex 4 has **degree 0**.

Vertex 3 is **adjacent** to vertex 6 and vice versa; {1, 5} is **adjacent** to 2; 4 is not **adjacent** to any other vertex.

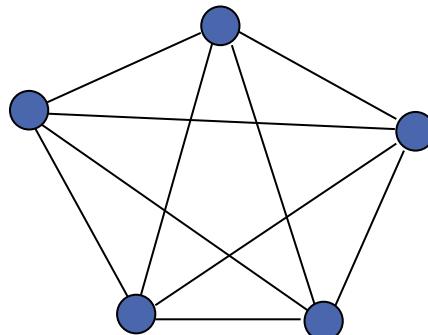




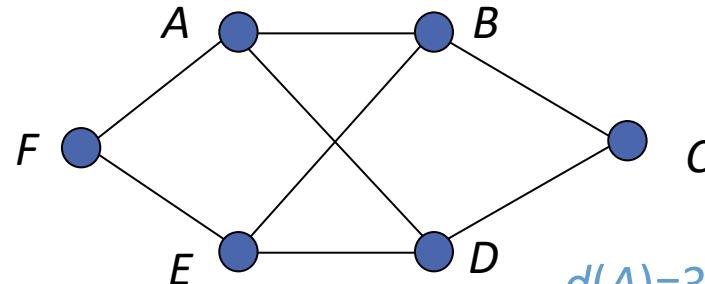
# Graph Introduction

**Complete graph:** When every vertex is strictly connected to each other. (The number of edges in the graph is maximum).

**Degree of a vertex v:** The degree of vertex v in a graph G, written  $d(v)$ , is the number of edges incident to v, except that each loop at v counts twice (*in-degree* and *out-degree* for directed graphs)



Complete Graph



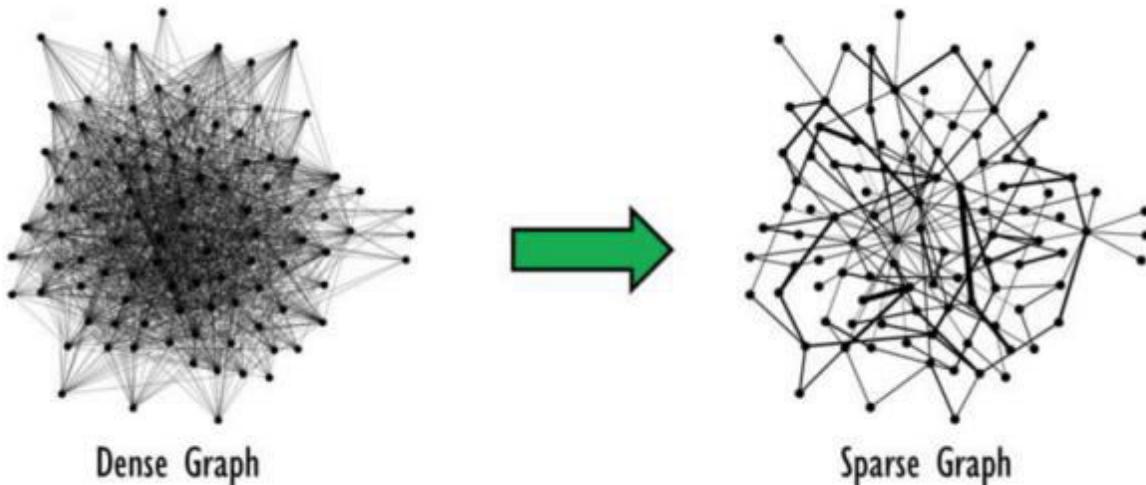
$$d(A)=3, d(B)=3, d(C)=2$$

$$d(D)=3, d(E)=3, d(F)=2$$

# Graph Introduction

**Dense graph:** When the number of edges in the graph is close to maximum. (*adjacency matrix* is used to store info for this)

**Sparse graph:** When number of edges in the graph is very few. (*adjacency list* is used to store info for this)





# Graph Introduction

**Weighted graph:** associates weights with either the edges or the vertices

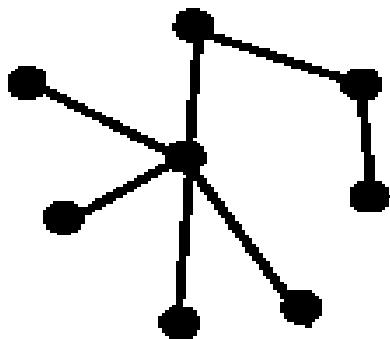
**DAG:** Directed acyclic graphs (every 2 vertices are not reachable from each other)

**Connected:** if every vertex of a graph can *reach* every other vertex, i.e., every pair of vertices is connected by a path

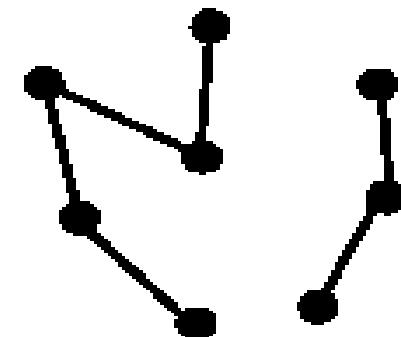
**Strongly connected:** every 2 vertices are reachable from each other (in a digraph)

**Connected Component:** equivalence classes of vertices under “is reachable from” relation. Simply put, it is a subgraph in which any two vertices are **connected** to each other by paths, and which is **connected** to no additional vertices in the supergraph.

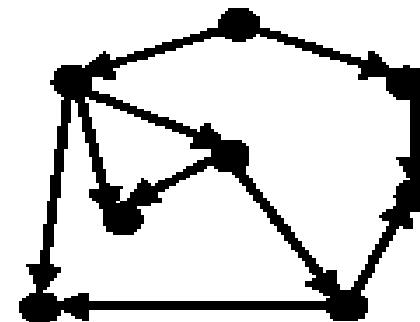
# Forests, DAG, Components



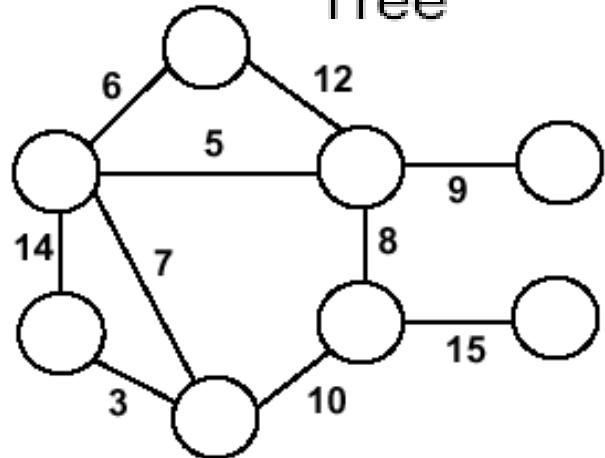
Tree



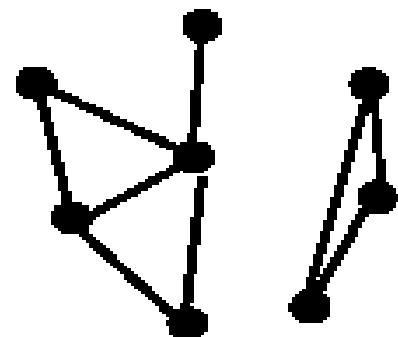
Forest



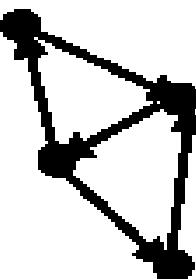
DAG



Weighted Graph



2 Components



Strongly-connected Component

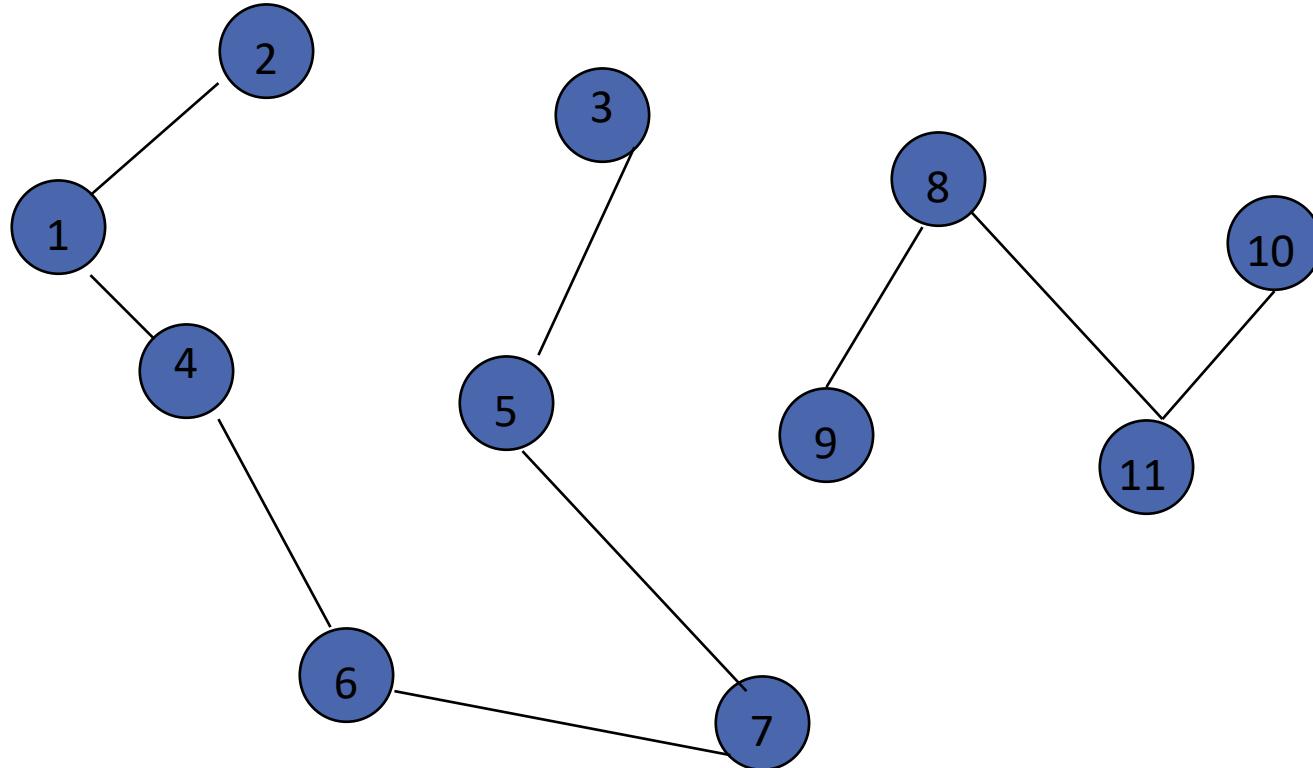


# Graph Applications

- ❖ State-space search in Artificial Intelligence
- ❖ Geographical information systems, electronic street directory
- ❖ Logistics and supply chain management
- ❖ Telecommunications network design
- ❖ Many more industry applications
- ❖ The graphic representation of world wide web (www)
- ❖ Resource allocation graph for processes that are active in the system.
- ❖ The graphic representation of a map
- ❖ **Scene graphs:** The contents of a visual scene are also managed by using graph data structure.



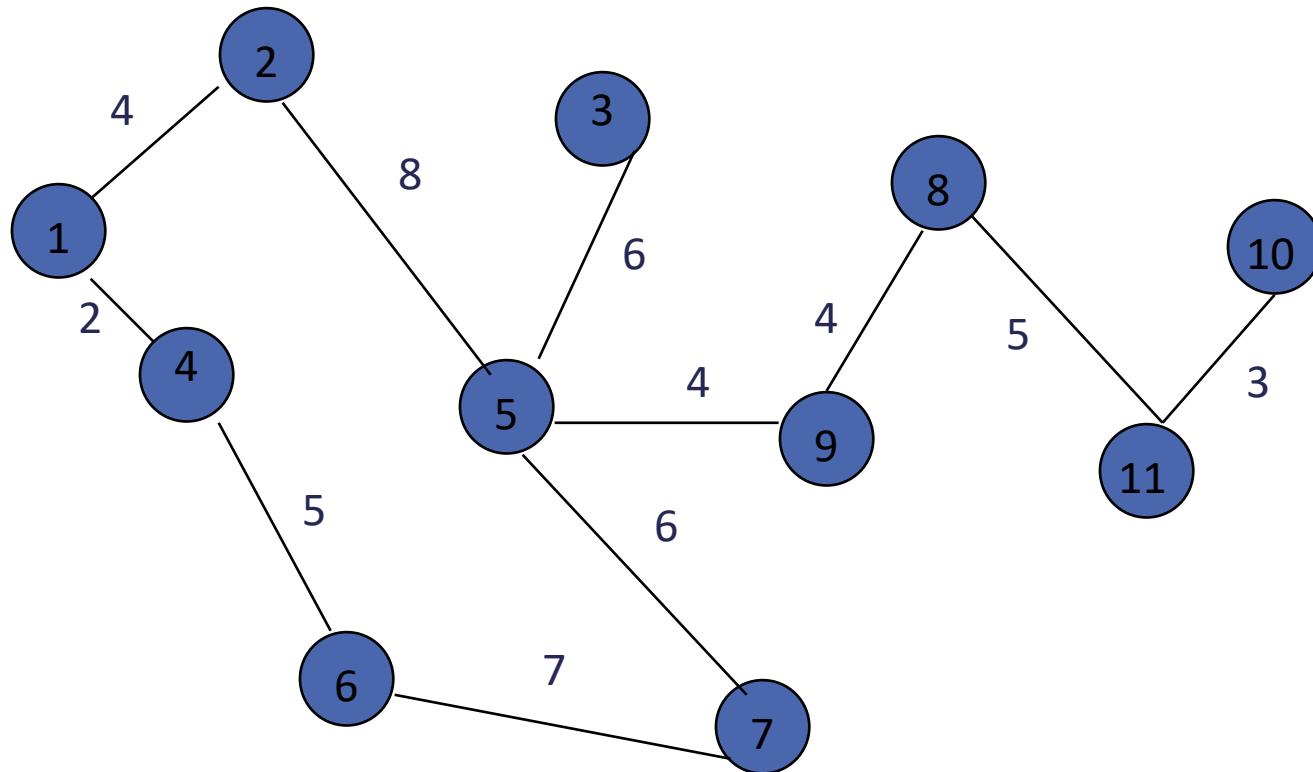
# Applications—Communication Network



Vertex = city, edge = communication link.



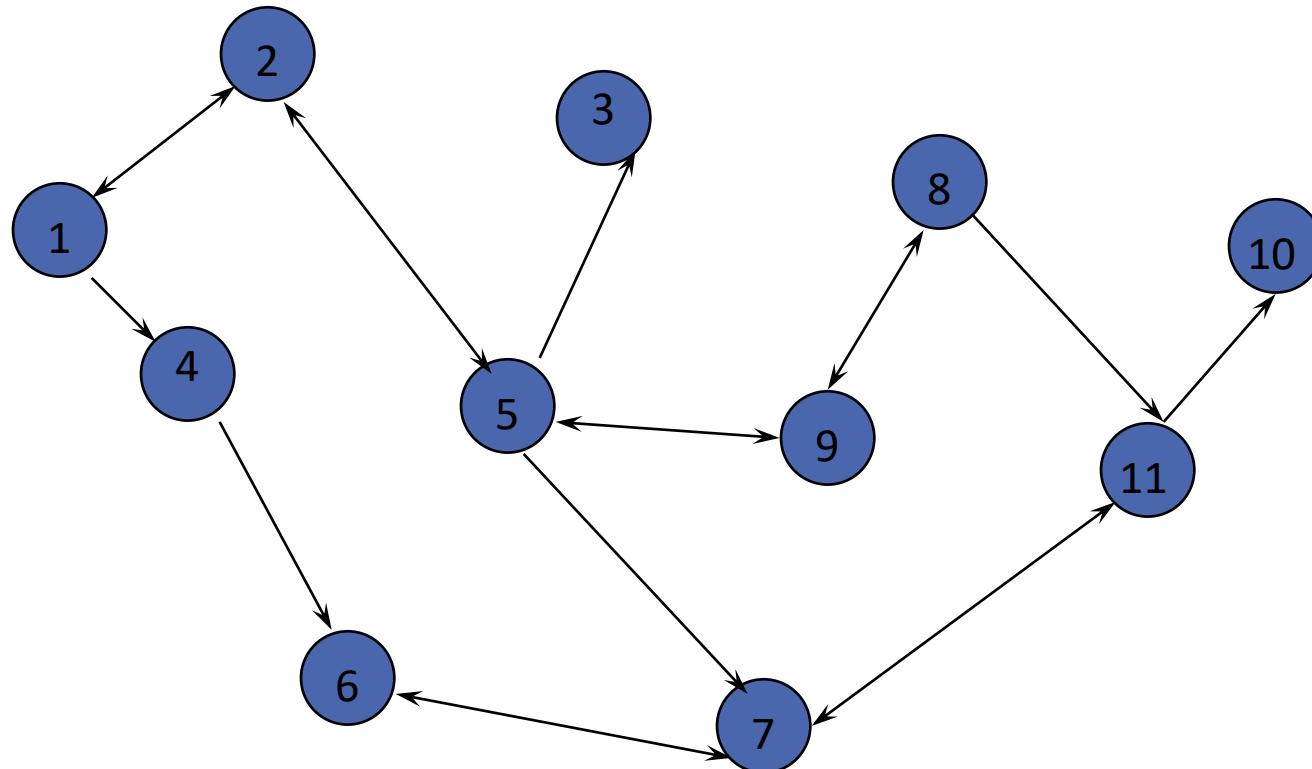
## Applications—Driving Distance/Time Map



Vertex = city, edge weight = driving distance/time.



## Applications—Street Map



Some streets are one way.



# Graph Representation

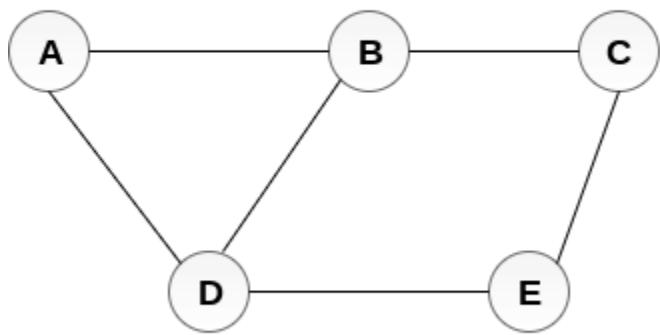
**Adjacency matrix:** represents a graph as  $n \times n$  matrix  $A$  (here,  $n$  is the number of nodes/ vertices):

$A[i, j] = 1$  if edge  $(i, j) \in E$  (or weight of edge)  
 $= 0$  if edge  $(i, j) \notin E$

**Storage requirements:**  $O(n^2)$

- ❖ Using adjacency matrix is more efficient to represent **dense** graphs
- ❖ Especially if store just one bit/edge
- ❖ Undirected graph: only need half of matrix

# Graph Representation



**Undirected Graph**

(a)

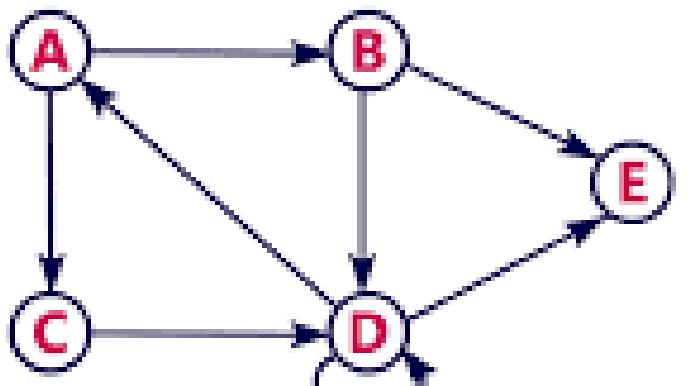
	A	B	C	D	E
A	0	1	0	1	0
B	1	0	1	1	0
C	0	1	0	0	1
D	1	1	0	0	1
E	0	0	1	1	0

**Adjacency Matrix**

(b)

- a. An **undirected graph**  $G$  having five vertices and six edges.
- b. The **adjacency-matrix** representation of  $G$ .

# Graph Representation



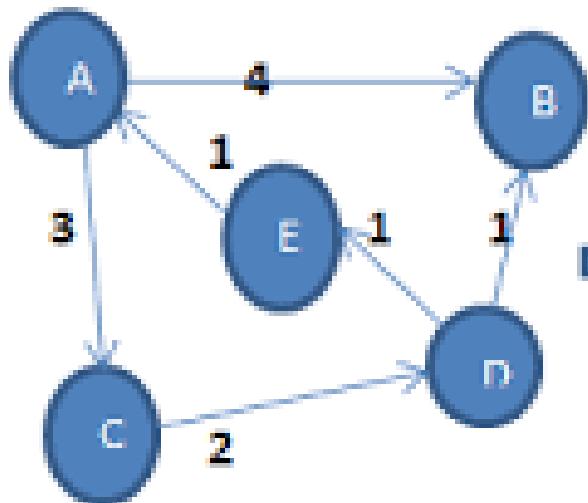
(a)

	A	B	C	D	E
A	0	1	1	0	0
B	0	0	0	1	1
C	0	0	0	1	0
D	1	0	0	1	1
E	0	0	0	0	0

(b)

- a. A **directed graph**  $G$  having five vertices and eight edges.
- b. The **adjacency-matrix** representation of  $G$ .

# Graph Representation



**Weighted Graph**

	A	B	C	D	E
A	0	4	3	0	0
B	0	0	0	0	0
C	0	0	0	2	0
D	0	1	0	0	1
E	1	0	0	0	0

**Adjacency matrix**

(a)

(b)

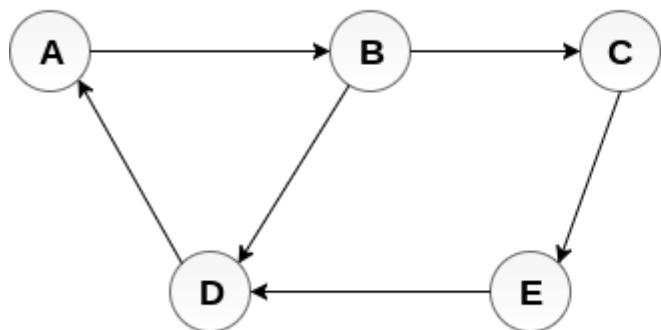
- a. A **directed weighted graph**  $G$  having five vertices and six edges.
- b. The **adjacency-matrix** representation of  $G$ .

# Graph Representation

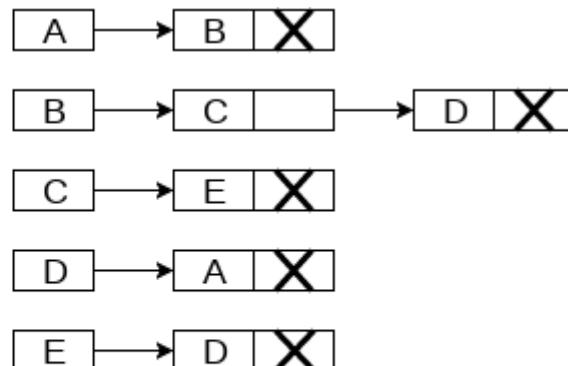
**Adjacency list:** list of adjacent vertices. For each vertex  $v \in V$ , store a list of vertices adjacent to  $v$

**Storage requirements:**  $O(n+e)$

Using adjacency list is more efficient to represent **sparse** graphs



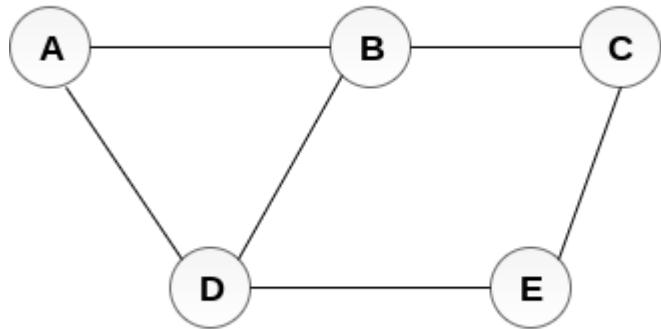
Directed Graph  
(a)



Adjacency List  
(b)

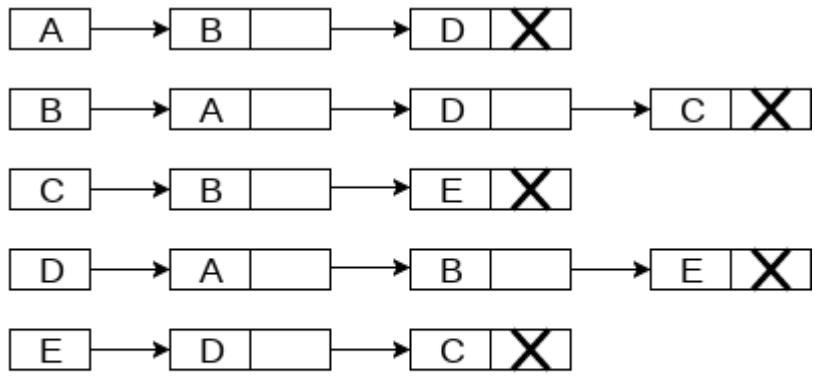
- A **directed graph**  $G$  having five vertices and six edges.
- The **adjacency-list** representation of  $G$  by using linked list.

# Graph Representation



Undirected Graph

(a)



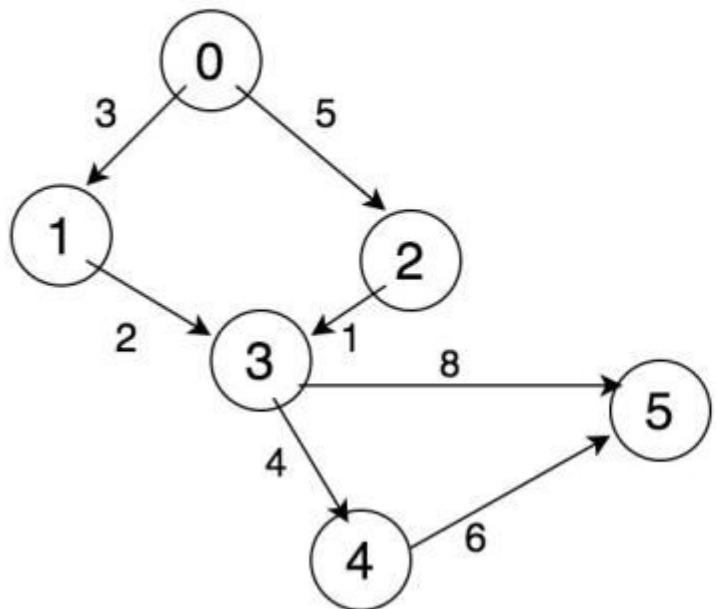
Adjacency List

(b)

- An **undirected graph**  $G$  having five vertices and six edges.
- The **adjacency-list** representation of  $G$ .

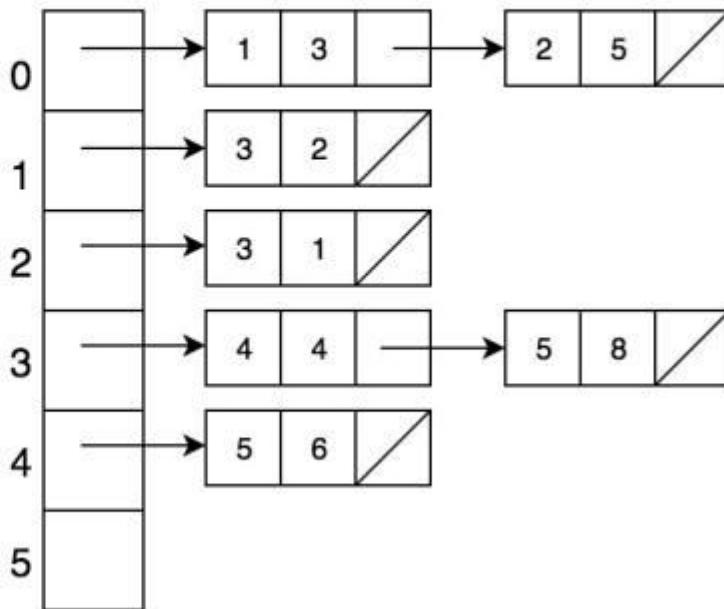
# Graph Representation

## Directed Graph



(a)

## Adjacency List Representation



(b)

- a. A **directed weighted graph**  $G$  having six vertices and seven edges.
- b. The **adjacency-list** representation of  $G$ .



# Graph Searching

- ↗ Given: a graph  $\mathbf{G} = (\mathbf{V}, \mathbf{E})$ , directed or undirected
- ↗ Goal: methodically explore every vertex and edge
- ↗ Ultimately: build a *tree* on the graph
  - ↗ Pick a vertex as the *root*
  - ↗ Choose certain *edges* to produce a tree
  - ↗ Note: might also build a *forest* if graph is not connected
- ↗ Breadth-first search
- ↗ Depth-first search
- ↗ Other variants: best-first, iterated deepening search, etc.



# Depth-First Search (DFS)

- ↗ Explore “deeper” in the graph whenever possible
  - ↗ Edges are *explored* out of the *most recently discovered* vertex  $v$  that still has unexplored edges (**LIFO**)
  - ↗ When all of  $v$ ’s edges have been explored, **backtrack** to the vertex from which  $v$  was discovered
  - ↗ computes 2 timestamps:  $d[ ]$  (**discovered**) and  $f[ ]$  (**finished**)
  - ↗ builds one or more **depth-first tree(s)** (**depth-first forest**)
- ↗ Algorithm colors each vertex
  - ↗ **WHITE**: undiscovered
  - ↗ **GRAY**: discovered, in process
  - ↗ **BLACK**: finished, all adjacent vertices have been discovered



# Depth-First Search: The Code

DFS(G)

{

```
    for each vertex u $\in$ V  
        color[u] = WHITE;  
    time = 0;  
    for each vertex u $\in$ V  
        if (color[u] == WHITE)  
            DFS_Visit(u);
```

}

**DFS\_Visit(u)**

{

```
    color[u] = GREY;  
    time = time+1;  
    d[u] = time; // compute d[]  
    for each v adjacent to u  
        if (color[v] == WHITE)  
            p[v]= u // build tree  
            DFS_Visit(v);  
    color[u] = BLACK;  
    time = time+1;  
    f[u] = time; // compute f[]
```

}



# DFS Classification of Edges

- ↗ DFS can be used to classify edges of  $G$ :
  1. **Tree edges**: edges in the depth-first forest.
  2. **Back edges**: edges  $(u, v)$  connecting a vertex  $u$  to an ancestor  $v$  in a depth-first tree.
  3. **Forward edges**: non-tree edges  $(u, v)$  connecting a vertex  $u$  to a **descendant**  $v$  in a depth-first tree.
  4. **Cross edges**: all other edges.
- ↗ DFS yields valuable information about the **structure** of a graph.

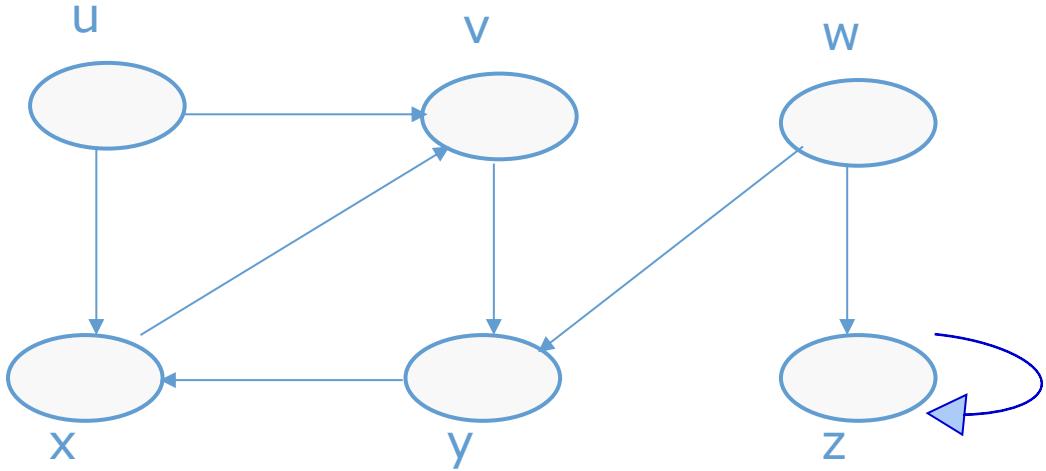
# Operations of DFS

Undiscovered

Discovered,  
On Process

Finished, all  
adjacent vertices  
have been  
discovered

Discover time/ Finish time



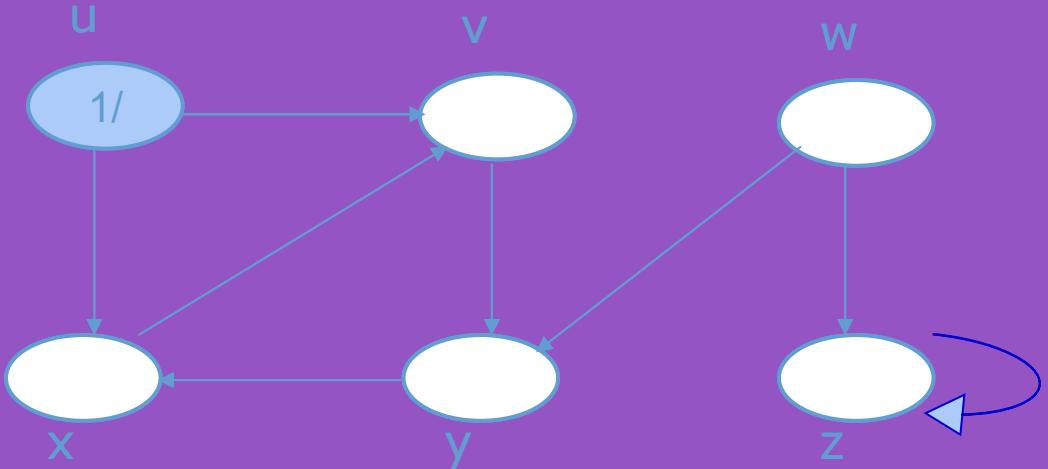
# Operations of DFS

Undiscovered

Discovered,  
On Process

Finished, all  
adjacent vertices  
have been  
discovered

Discover time/ Finish time



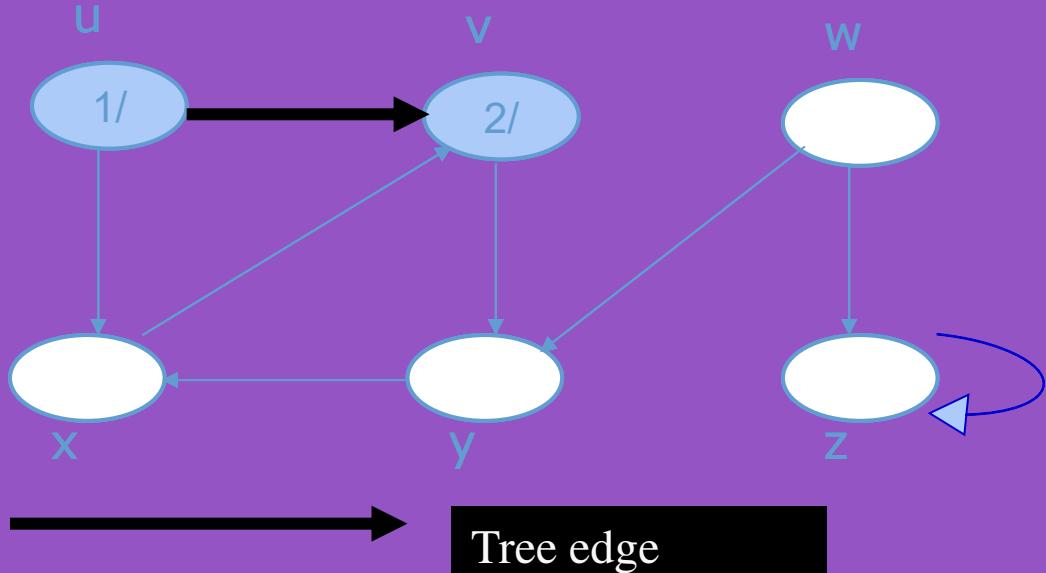
# Operations of DFS

Undiscovered

Discovered,  
On Process

Finished, all  
adjacent vertices  
have been  
discovered

Discover time/ Finish time



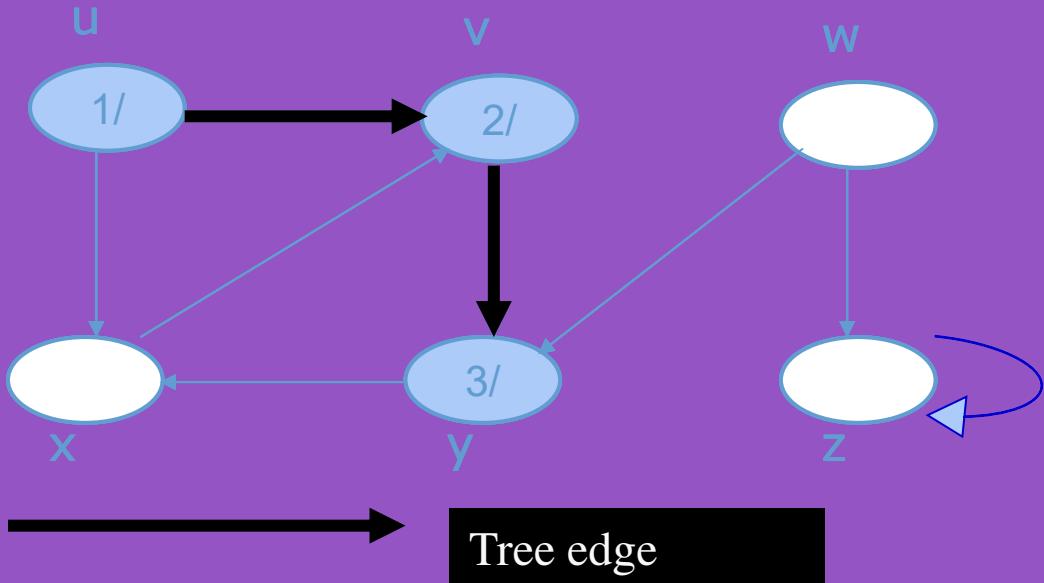
# Operations of DFS

Undiscovered

Discovered,  
On Process

Finished, all  
adjacent vertices  
have been  
discovered

Discover time/ Finish time

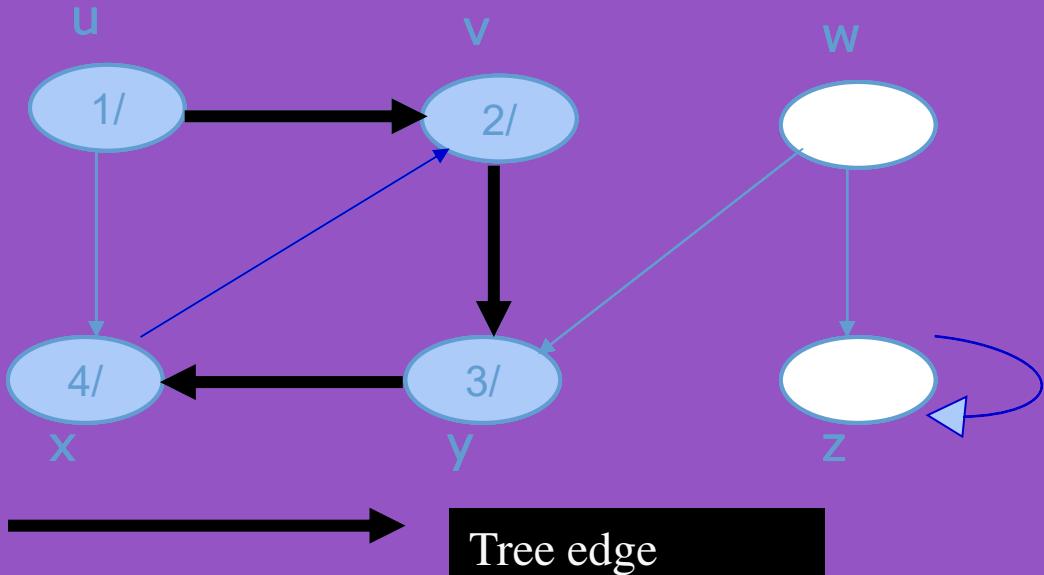


# Operations of DFS

Undiscovered

Discovered,  
On Process

Finished, all  
adjacent vertices  
have been  
discovered



Discover time/ Finish time

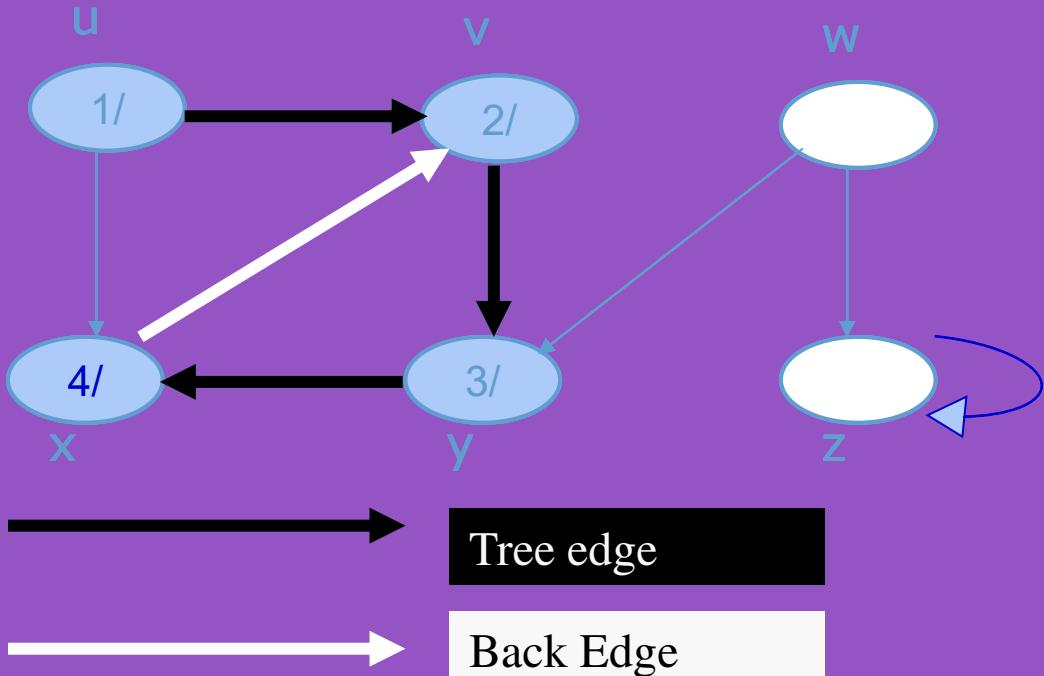
# Operations of DFS

Undiscovered

Discovered,  
On Process

Finished, all  
adjacent vertices  
have been  
discovered

Discover time/ Finish time



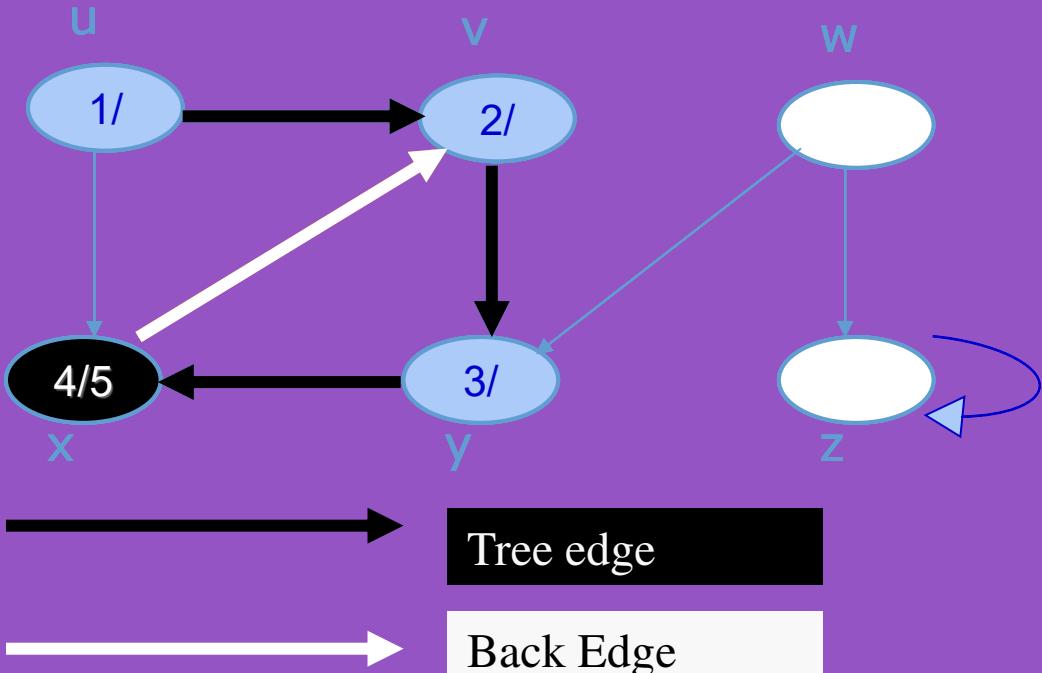
# Operations of DFS

Undiscovered

Discovered,  
On Process

Finished, all  
adjacent vertices  
have been  
discovered

Discover time/ Finish time



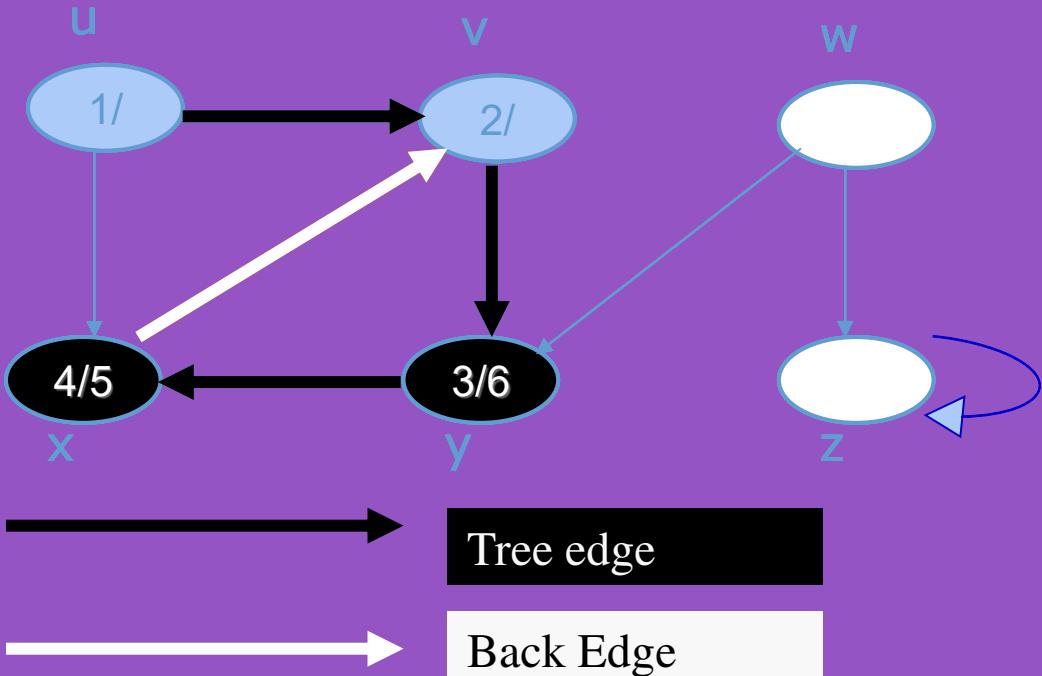
# Operations of DFS

Undiscovered

Discovered,  
On Process

Finished, all  
adjacent vertices  
have been  
discovered

Discover time/ Finish time



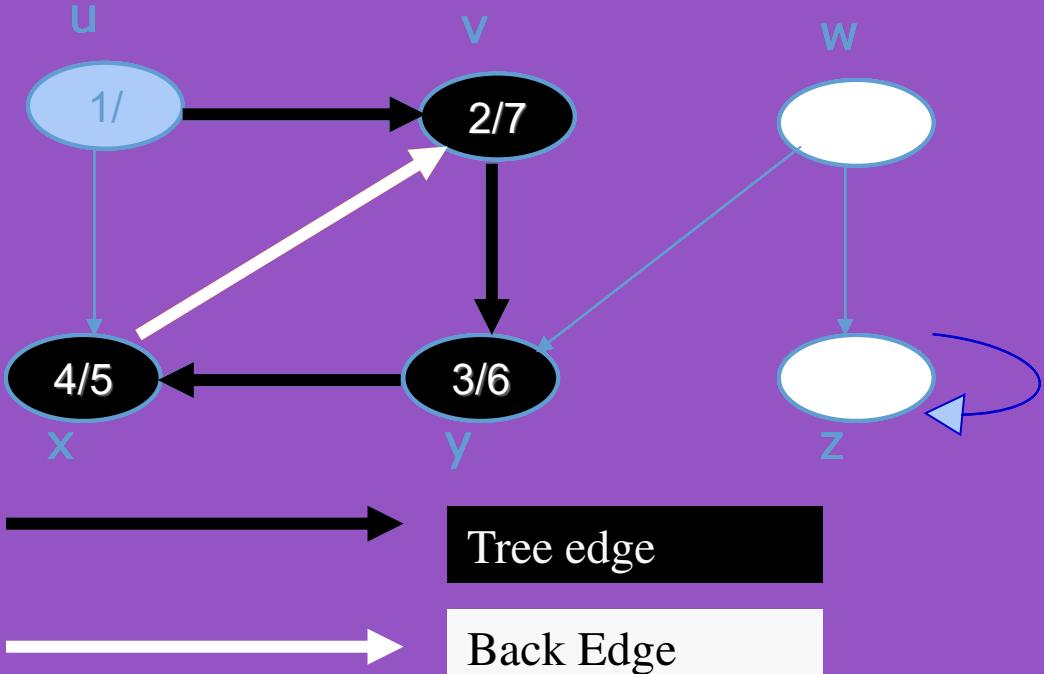
# Operations of DFS

Undiscovered

Discovered, On Process

Finished, all adjacent vertices have been discovered

Discover time/ Finish time



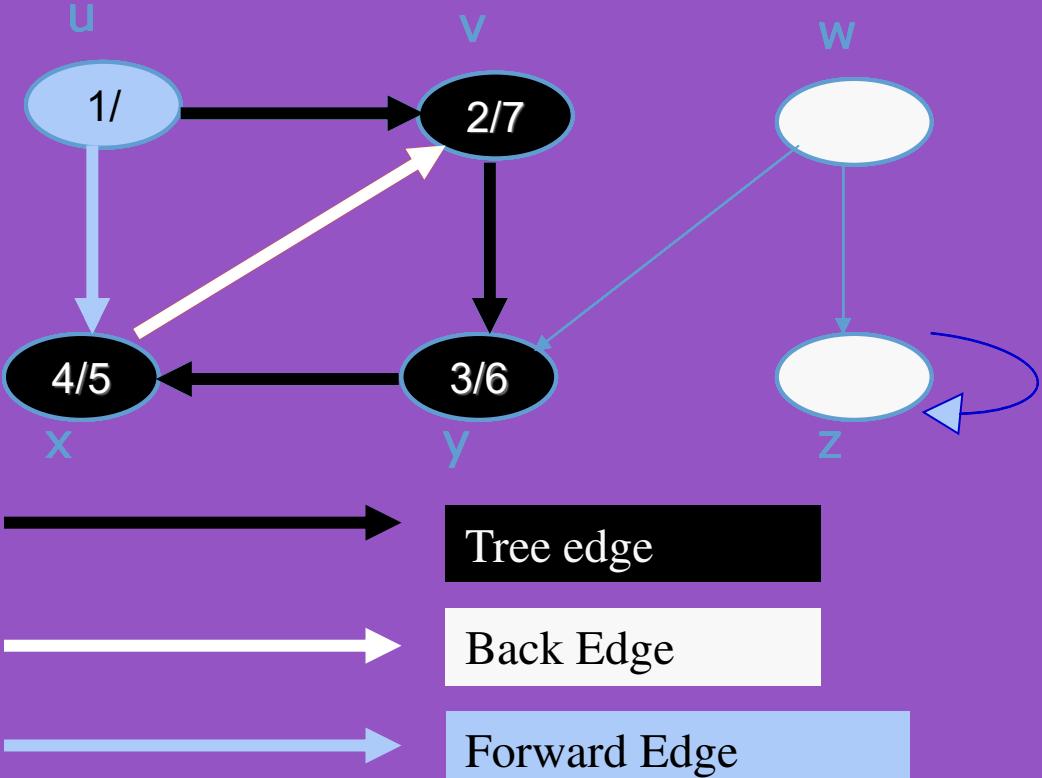
# Operations of DFS

Undiscovered

Discovered,  
On Process

Finished, all  
adjacent vertices  
have been  
discovered

Discover time/ Finish time



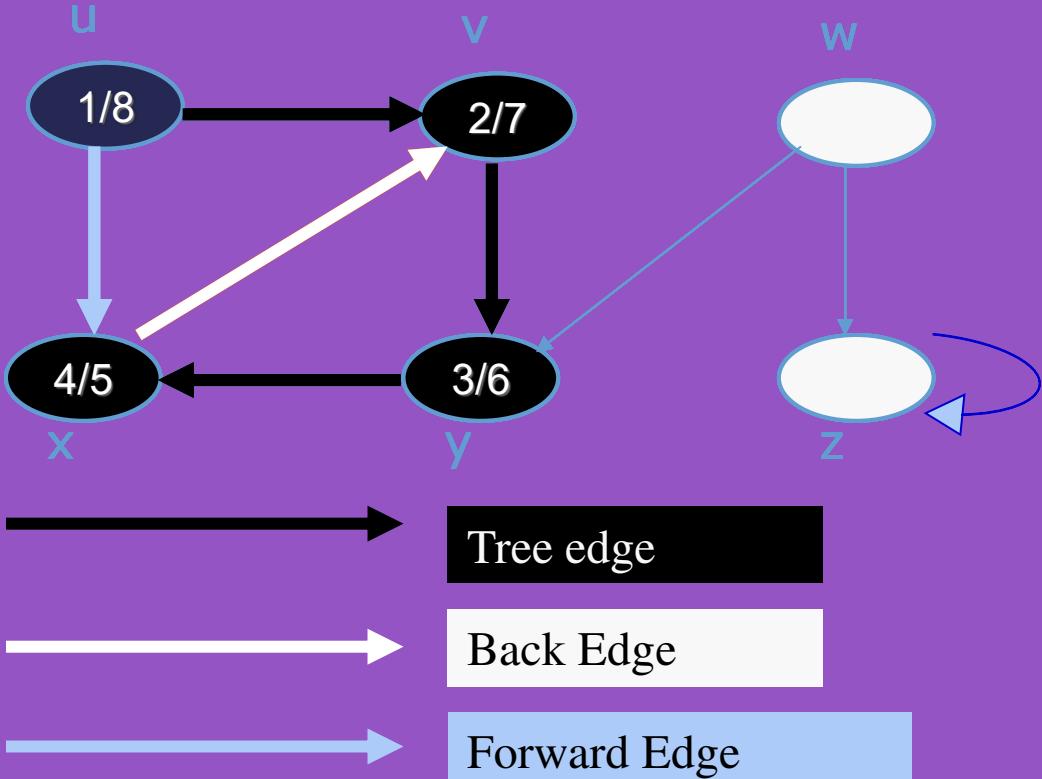
# Operations of DFS

Undiscovered

Discovered,  
On Process

Finished, all  
adjacent vertices  
have been  
discovered

Discover time/ Finish time



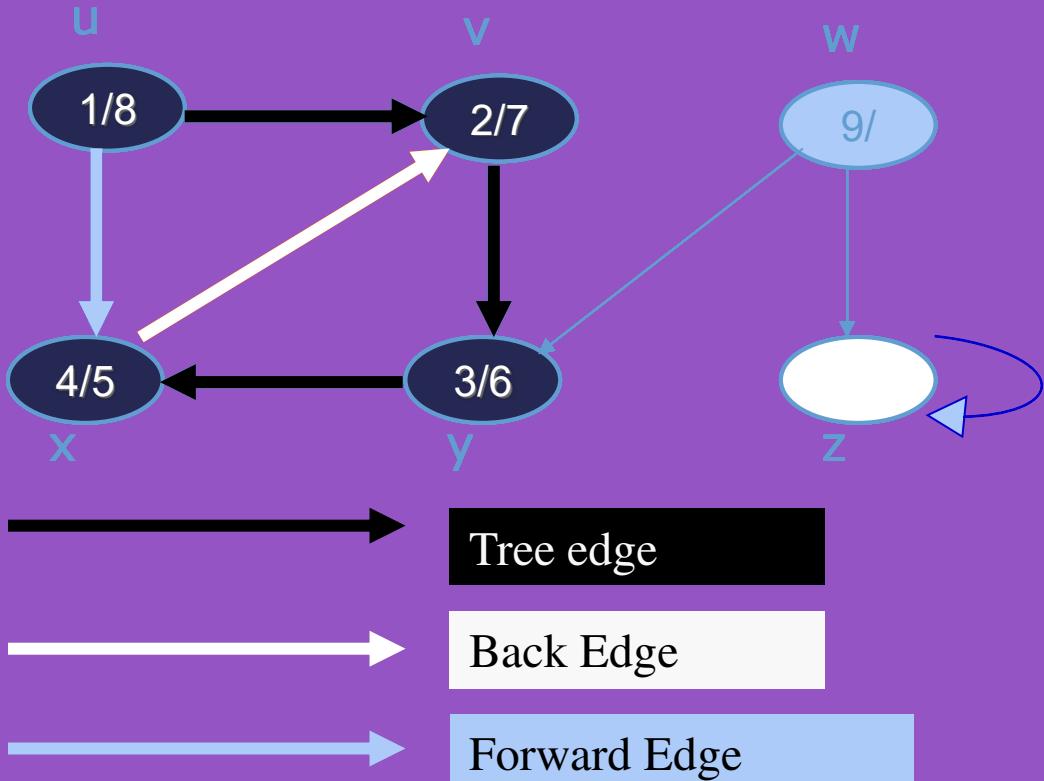
# Operations of DFS

Undiscovered

Discovered,  
On Process

Finished, all  
adjacent vertices  
have been  
discovered

Discover time/ Finish time



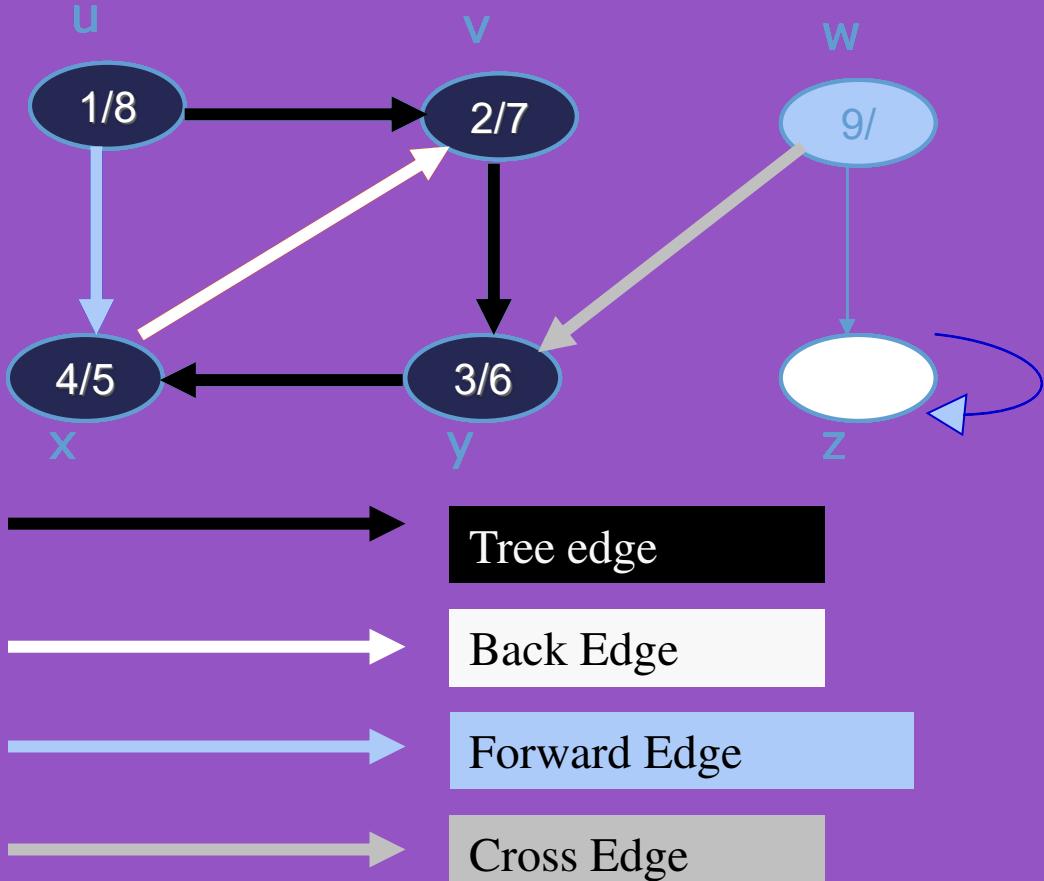
# Operations of DFS

Undiscovered

Discovered,  
On Process

Finished, all  
adjacent vertices  
have been  
discovered

Discover time/ Finish time



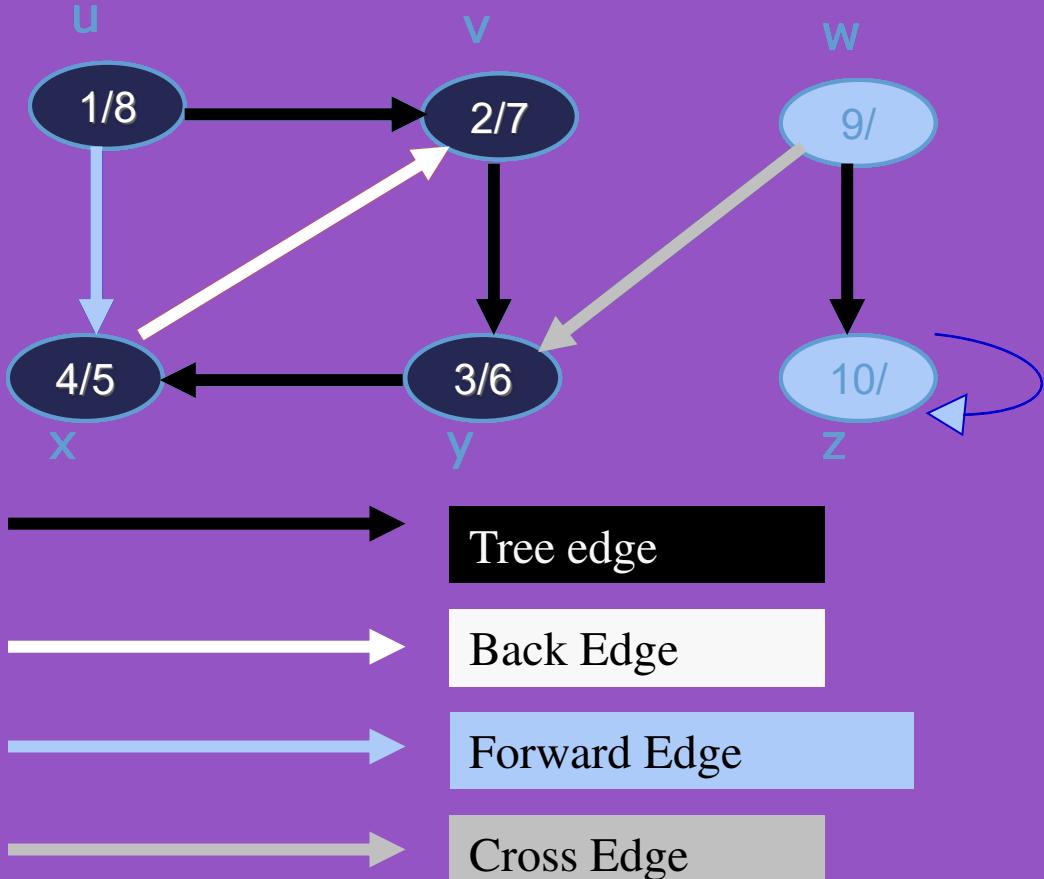
# Operations of DFS

Undiscovered

Discovered,  
On Process

Finished, all  
adjacent vertices  
have been  
discovered

Discover time/ Finish time



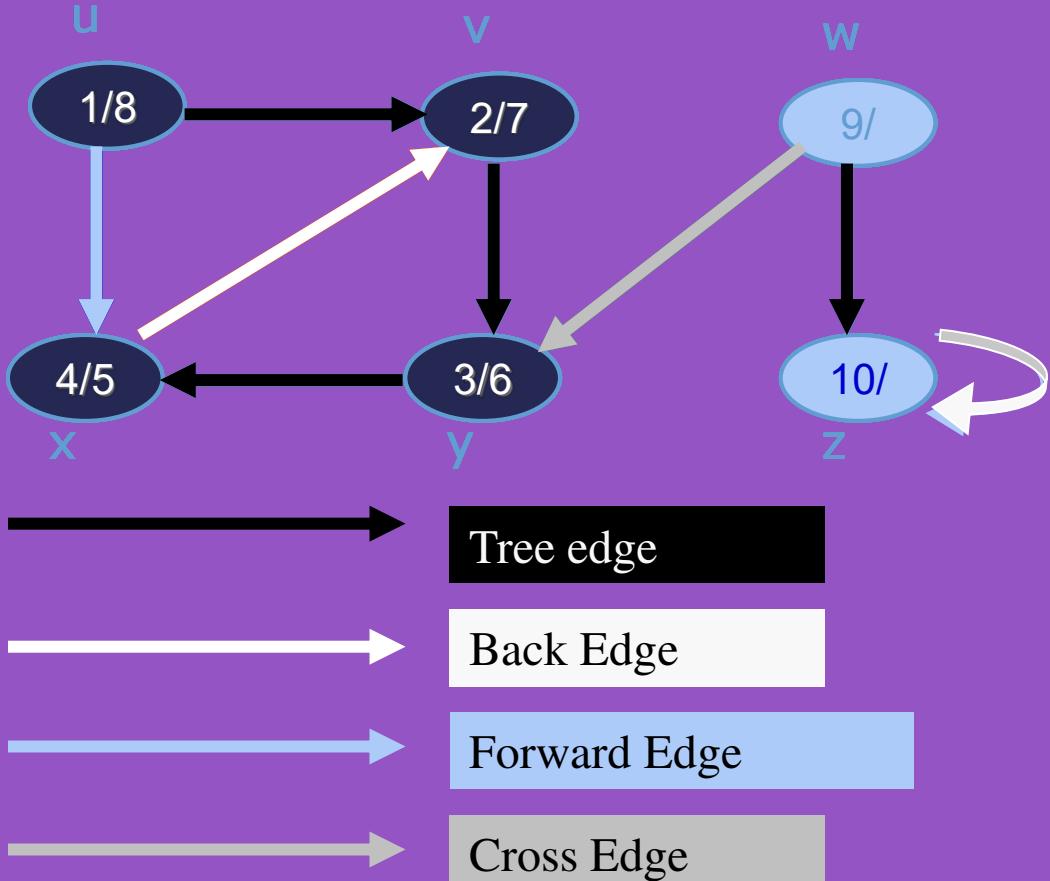
# Operations of DFS

Undiscovered

Discovered,  
On Process

Finished, all  
adjacent vertices  
have been  
discovered

Discover time/ Finish time



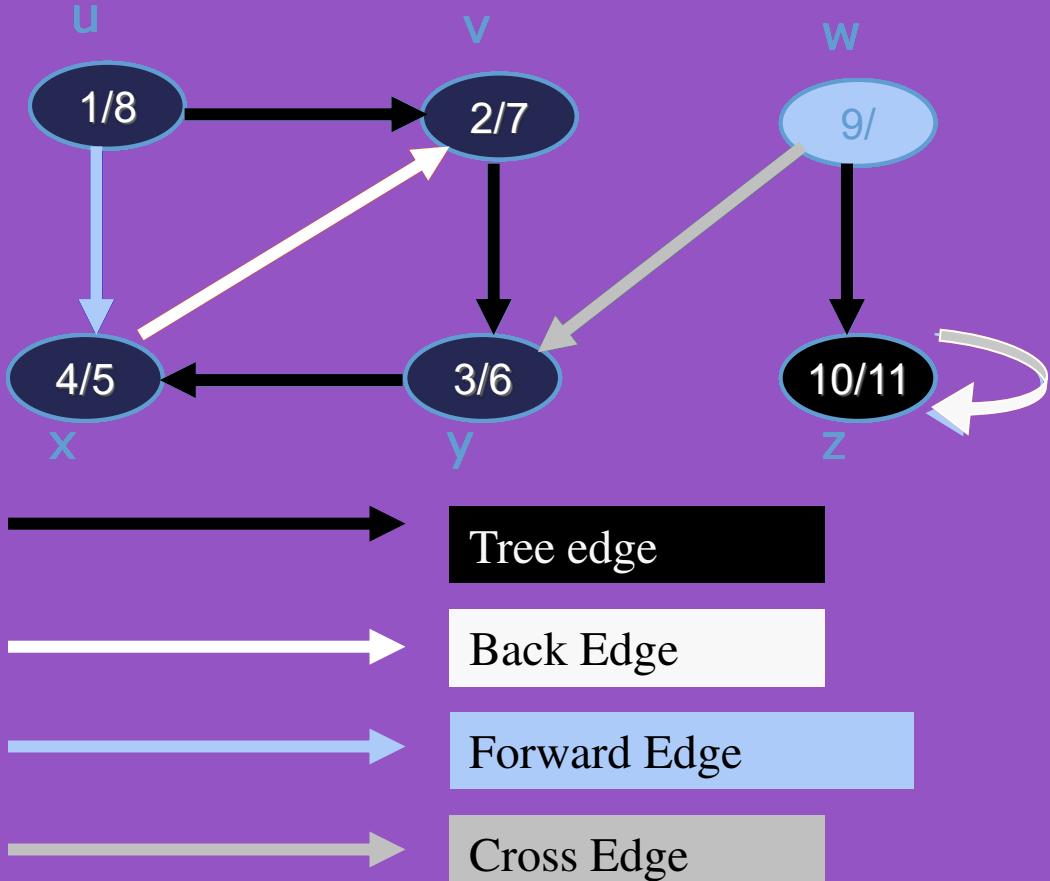
# Operations of DFS

Undiscovered

Discovered,  
On Process

Finished, all  
adjacent vertices  
have been  
discovered

Discover time/ Finish time



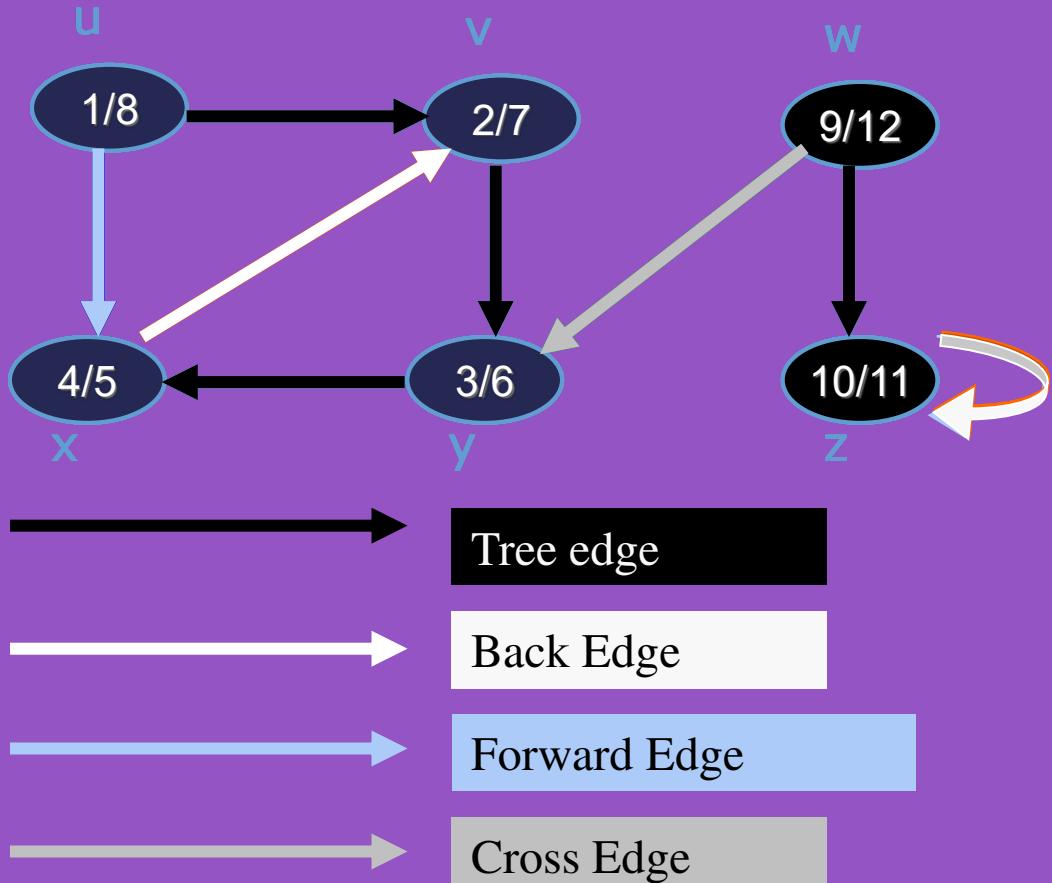
# Operations of DFS

Undiscovered

Discovered,  
On Process

Finished, all  
adjacent vertices  
have been  
discovered

Discover time/ Finish time





# DFS Analysis

- ↗ Running time of DFS =  $O(n+e)$
- ↗ DFS (excluding DFS\_Visit) takes  $O(n)$  time
- ↗ DFS\_Visit:
  - ↗  $\text{DFS\_Visit}(v)$  is called exactly once for each vertex  $v$
  - ↗ During  $\text{DFS\_Visit}(v)$ , adjacency list of  $v$  is scanned once
  - ↗ sum of lengths of adjacency lists =  $O(e)$
- ↗ This type of aggregate analysis is an informal example of *amortized analysis*



# Cycle Detection

- ↗ Theorem: An undirected graph is *acyclic* iff a DFS yields no **back edges**
- ↗ Acyclic: If a graph contains no cycles.
- ↗ Proof
  - ↗ If acyclic, no back edges by definition (because a back edge implies a cycle)
  - ↗ If no back edges, acyclic
    - ↗ No back edges implies only tree edges (*Why?*)
    - ↗ Only tree edges implies we have a tree or a forest
    - ↗ Which by definition is acyclic
- ↗ Thus, can run DFS to find whether a graph has a cycle.
- ↗ ***How would you modify the code to detect cycles?***



# Breadth-First Search (BFS)

- ↗ Given source vertex  $s$ ,
  - ↗ systematically explore the ***breadth*** of the frontier to
  - ↗ discover every vertex reachable from  $s$
  - ↗ computes the distance  $d[ ]$  from  $s$  to all reachable vertices
  - ↗ builds a **breadth-first tree** rooted at  $s$
- ↗ Algorithm
  - ↗ colors each vertex:
    - ↗ **WHITE** : undiscovered
    - ↗ **GRAY**: discovered, in process
    - ↗ **BLACK**: finished, all **adjacent** vertices have been discovered



# Breadth-First Search (BFS)

## Characteristic of BFS:

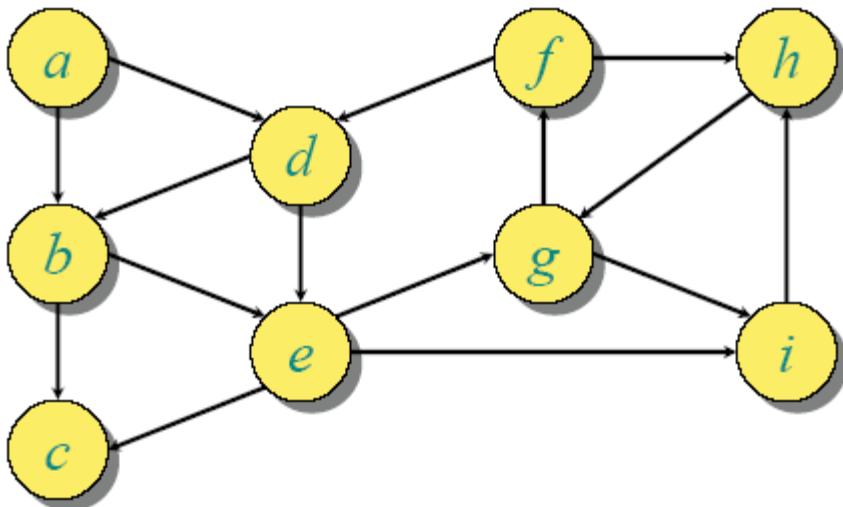
- ↗ Follow **FIFO** mechanism.
- ↗ We get the **shortest path** using BFS.
- ↗ BFS only used for **unweighted graphs**.



# BFS: The Code

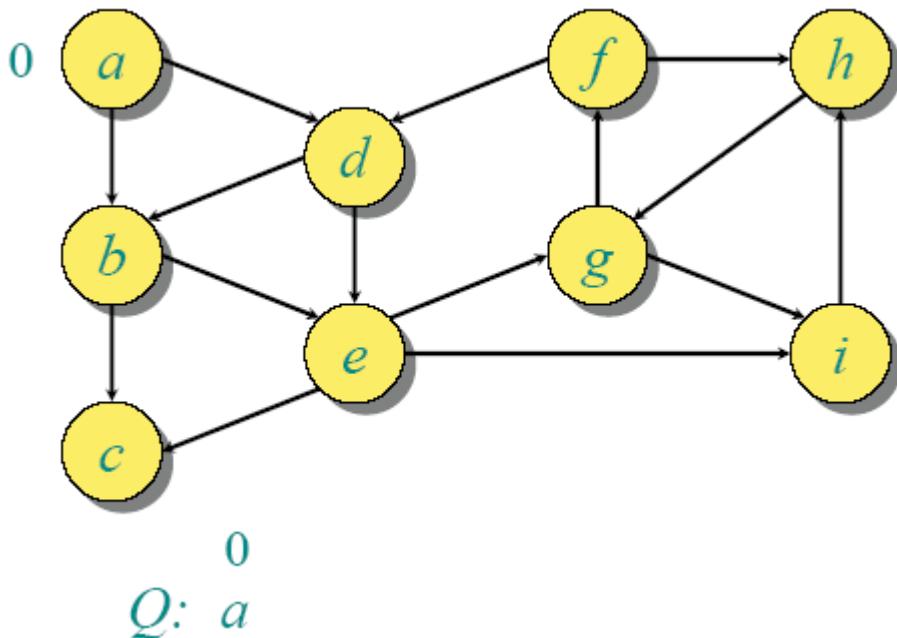
```
BFS(G, s) {  
    initialize vertices;  
    Q = {s};  
    while (Q not empty) {  
        u = Dequeue(Q);  
        for each v adjacent to u do {  
            if (color[v] == WHITE) {  
                color[v] = GRAY;  
                d[v] = d[u] + 1; // compute d[]  
                p[v] = u; // build BFS tree  
                Enqueue(Q, v);  
            }  
        }  
        color[u] = BLACK;  
    }  
}
```

# BFS Example

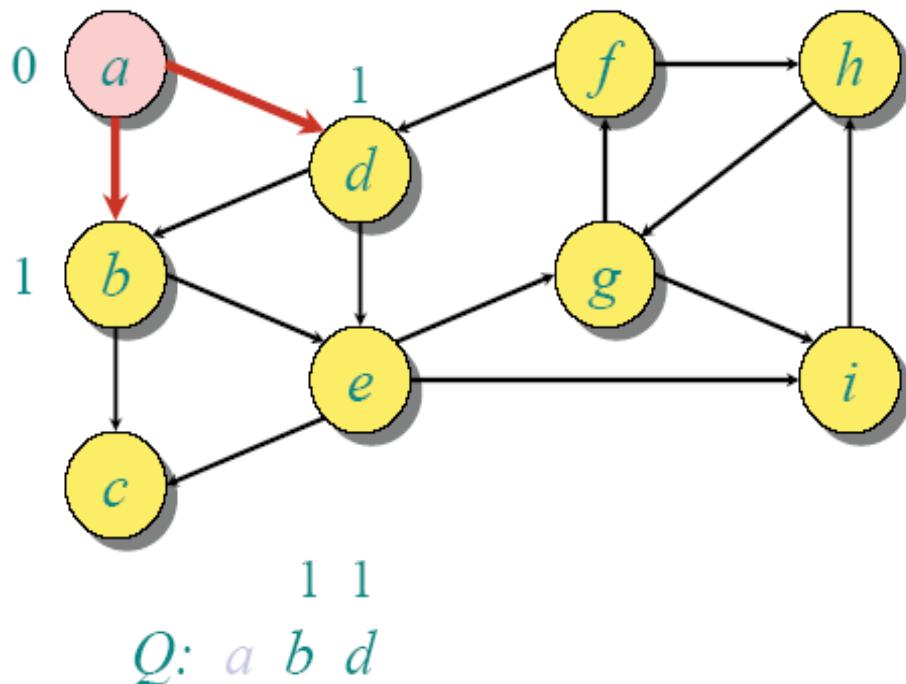


*Q:*

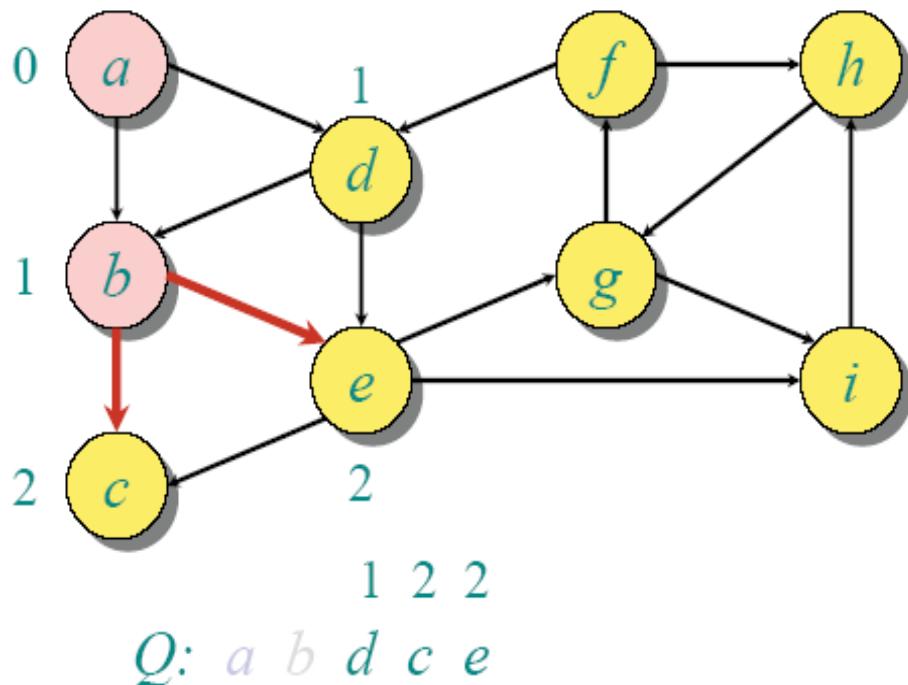
# BFS Example



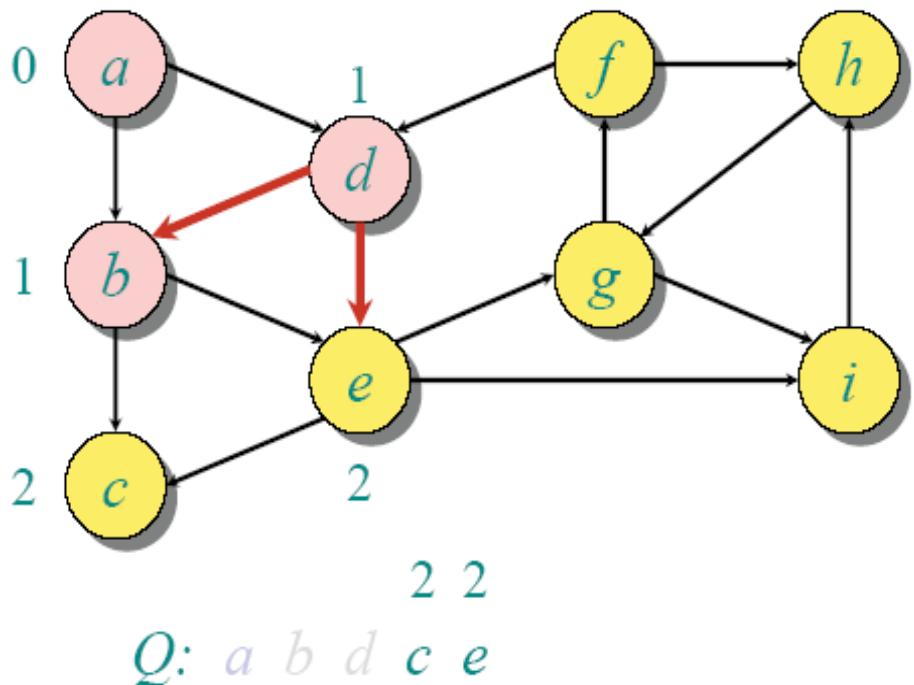
# BFS Example



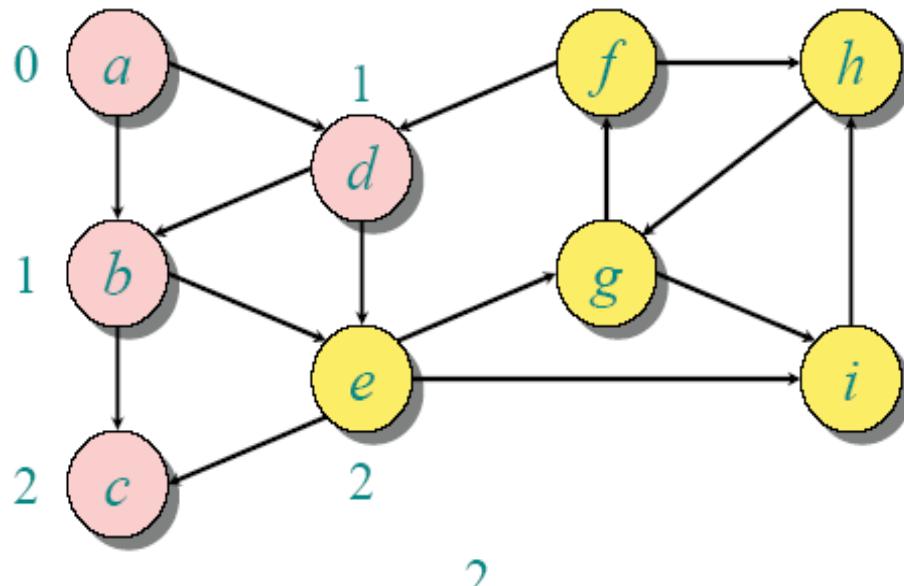
# BFS Example



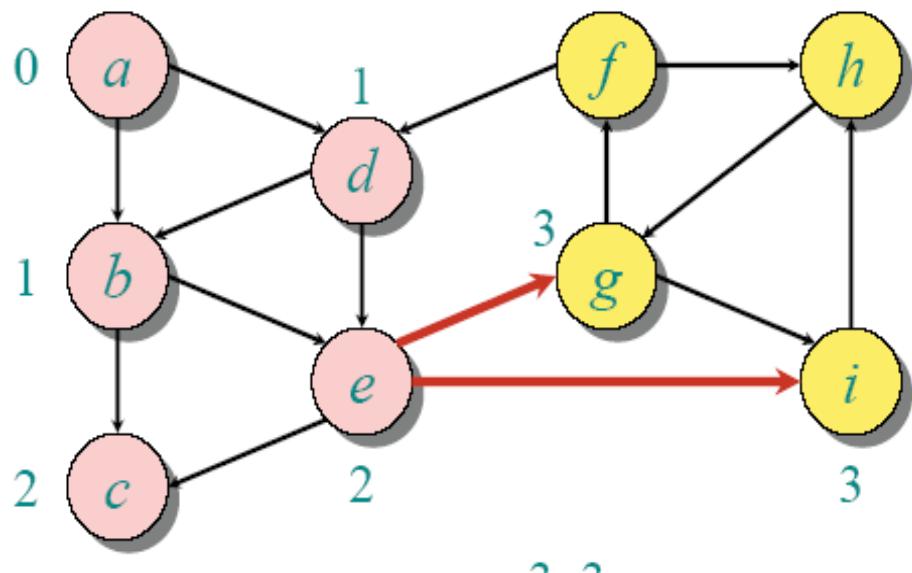
# BFS Example



# BFS Example

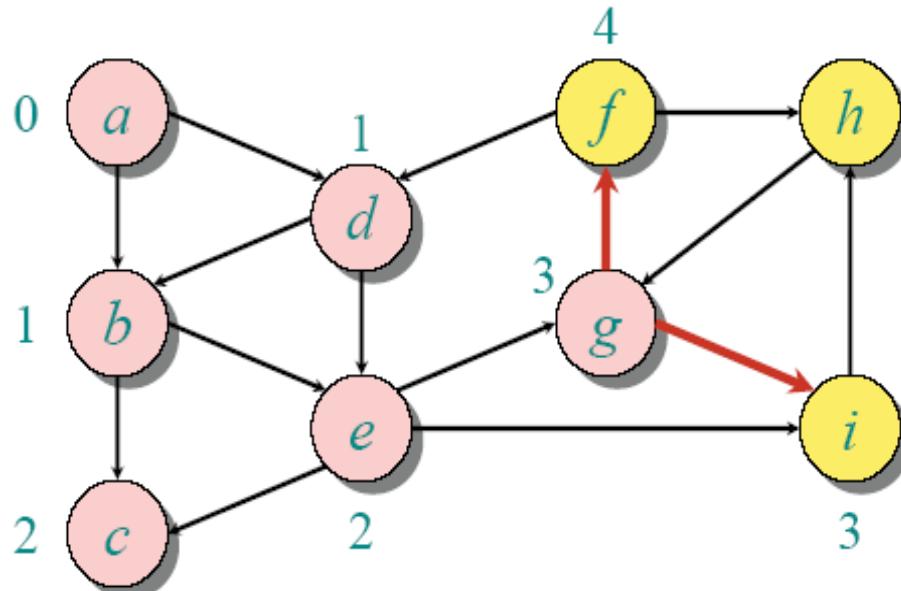


# BFS Example



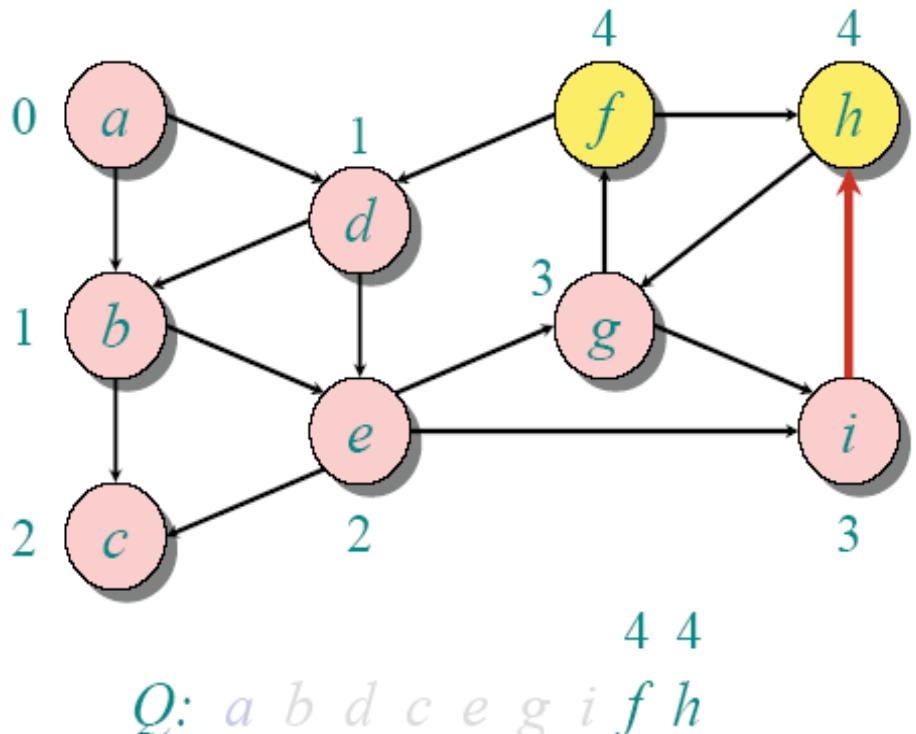
$Q: a \ b \ d \ c \ e \ g \ i$

# BFS Example

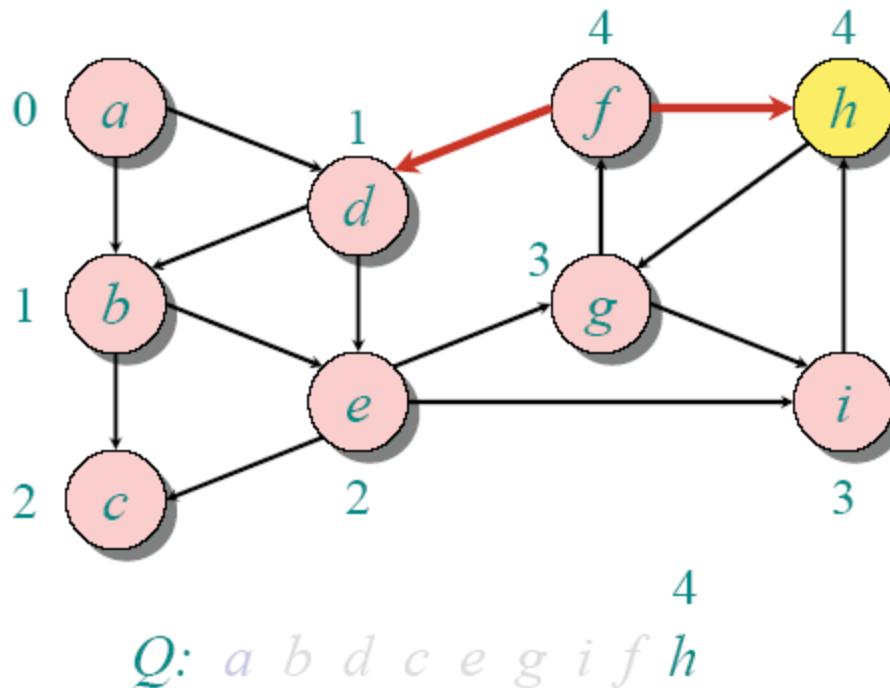


$Q: a \ b \ d \ c \ e \ g \ i \ f$

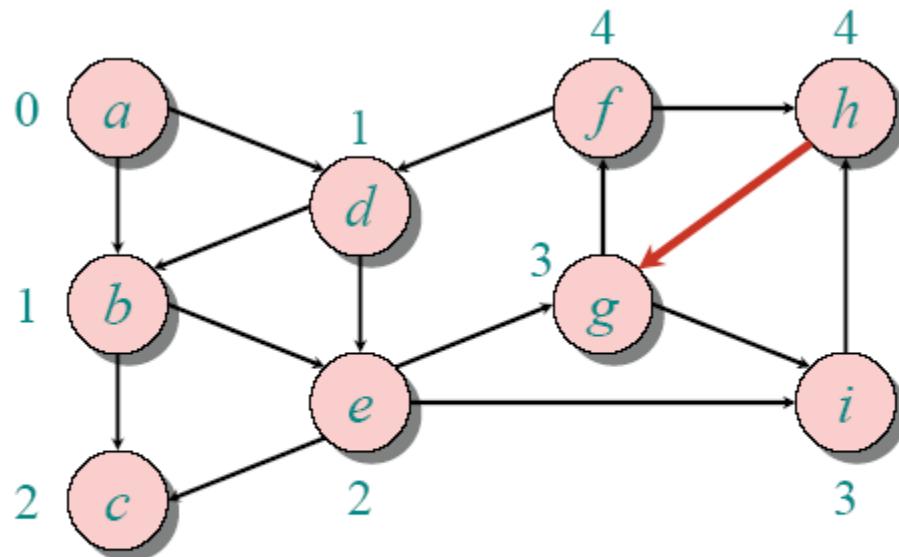
# BFS Example



# BFS Example

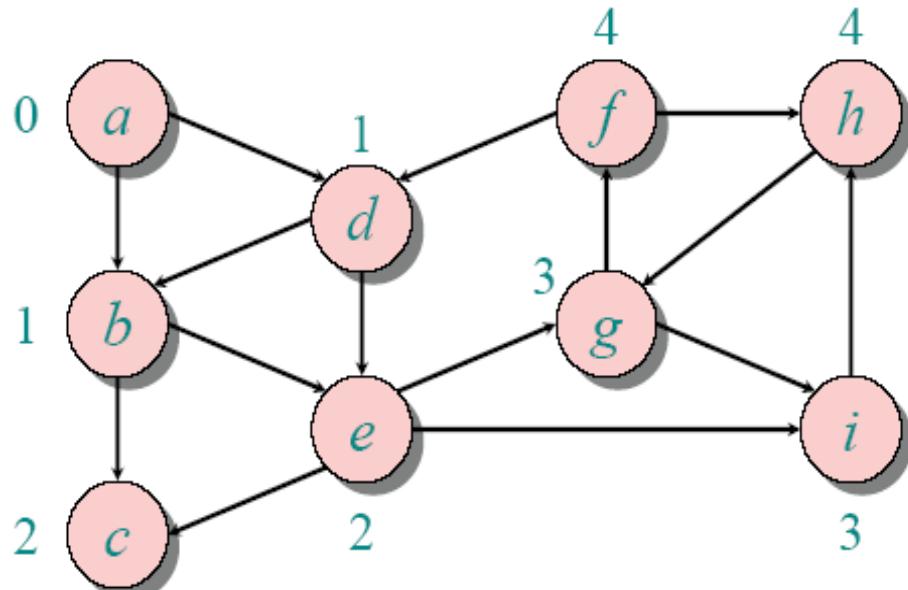


# BFS Example



$Q: a \ b \ d \ c \ e \ g \ i \ f \ h$

# BFS Example



$Q: a \ b \ d \ c \ e \ g \ i \ f \ h$



# BFS Analysis

- ↗ ***initialize : O(n)***
- ↗ *Loop: Queue operations and Adjacency checks*
- ↗ *Queue operations*
  - ↗ each vertex is enqueue/dequeued at most once. Why?
  - ↗ each operation takes **O(1)** time, hence **O(n)**
- ↗ *Adjacency checks*
  - ↗ adjacency list of each vertex is scanned at most once
  - ↗ **sum of lengths of adjacency lists = O(e)**
- ↗ **Total run time of BFS = O(n+e)**



# Breadth-First Search: Properties

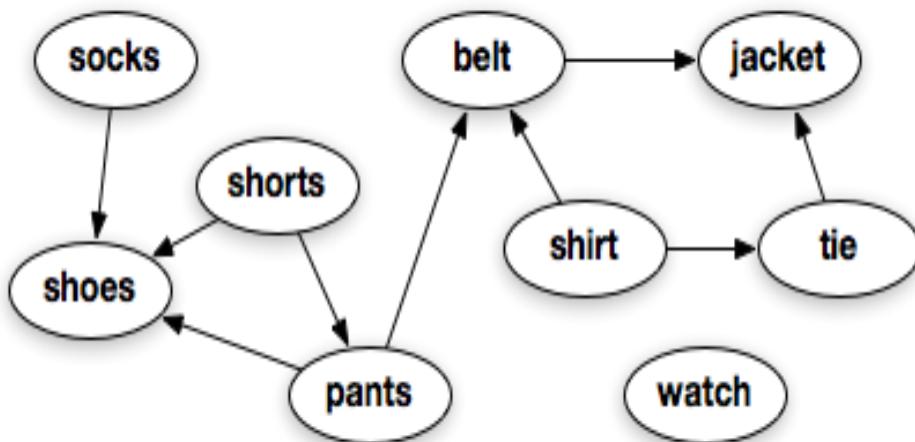
- ↗ What do we get the end of BFS?
- 1.  $d[v] = \text{shortest-path distance}$  from  $s$  to  $v$ , i.e. minimum number of edges from  $s$  to  $v$ , or  $\infty$  if  $v$  not reachable from  $s$ 
  - ↗ Proof : refer CLRS
- 2. a ***breadth-first tree***, in which path from root  $s$  to any vertex  $v$  represent a shortest path
  - ↗ Thus can use BFS to calculate shortest path from one vertex to another in  $O(n+e)$  time, for unweighted graphs.



# Topological Sort

- ↗ Find a linear ordering of all vertices of the DAG such that if G contains an edge  $(u, v)$ , u appears before v in the ordering.
- ↗ In general, there may be many legal topological orders for a given DAG.
- ↗ *Idea:*
  1. Call **DFS(G)** to compute finishing time  $f[ ]$
  2. Insert vertices onto a linked list according to decreasing order of  $f[ ]$
- ↗ *How to modify DFS to perform Topological Sort in  $O(n+e)$  time?*

Things	Things to Wear Earlier
Socks	.....
Shoes	Socks, Shorts, Pants
Sorts	.....
Pants	Shorts
Belts	Pants, Shirt
Shirt	.....
Jacket	Belt, Tie
Tie	Shirt
Watch	.....



Watch → Shirt → Tie → Shorts → Pants → Belt → Jacket → Socks → Shoes

Shirt → Tie → Watch → Socks → Shorts → Pants → Belt → Jacket → Shoes



# Books

1. *Introduction to Algorithms, Third Edition, Thomas H. Cormen, Charle E. Leiserson, Ronald L. Rivest, Clifford Stein (CLRS)*.
2. *Fundamental of Computer Algorithms, Ellis Horowitz, Sartaj Sahni, Sanguthevar Rajasekaran (HSR)*



# References

- <http://www.mathcs.emory.edu/~cheung/Courses/171/Syllabus/11-Graph/dfs.html>
- CLRS: 22.1, 22.2, 22.3, 22.4, 22.5
- HSR: 2.2, 2.5

# Minimum Spanning Tree

Course Code: CSC2211

Course Title: Algorithms



**Dept. of Computer Science  
Faculty of Science and Technology**

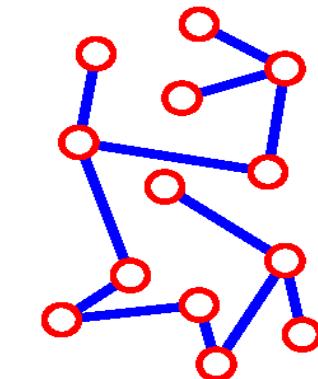
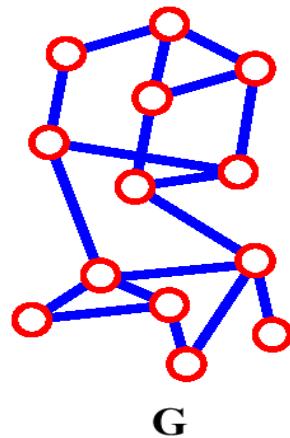
<b>Lecturer No:</b>		<b>Week No:</b>	<b>11</b>	<b>Semester:</b>	
<b>Lecturer:</b>	<i>Name &amp; email</i>				



# Lecture Outline

1. Recalling Spanning Tree.
2. What is Minimum Spanning Tree (MST).
3. Applications of MST.
4. Algorithmic technique to find MST.
5. Kruskal's Algorithm
6. Prim-Jarnik's Algorithm.
7. Running time of Kruskal's and Prim-Jarnik's Algorithm.

# 1. Spanning Tree.



A **spanning tree of **G**** is a subgraph which is a tree that contains **all vertices** of **G** and **no cycle**.

If a graph **G** has **v** vertices and **e** number of edges, then a spanning tree of that graph will have **n vertices and (n-1) number of edges**.



# Minimum Spanning Tree (MST)

Definition

A **minimum spanning tree** (MST) or **minimum cost spanning tree** is a subset of the edges of a connected, weighted undirected graph that connects all the vertices together, without any cycles and with the **minimum** possible total edge weight.



## MST....

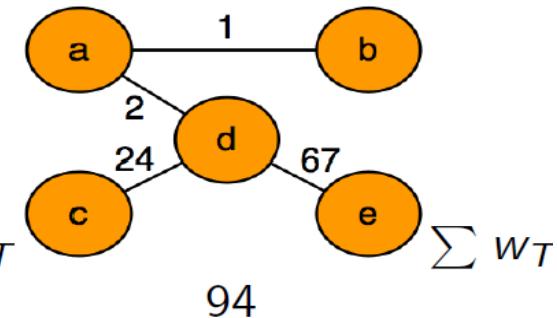
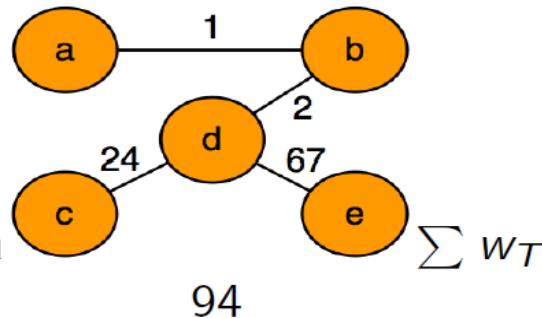
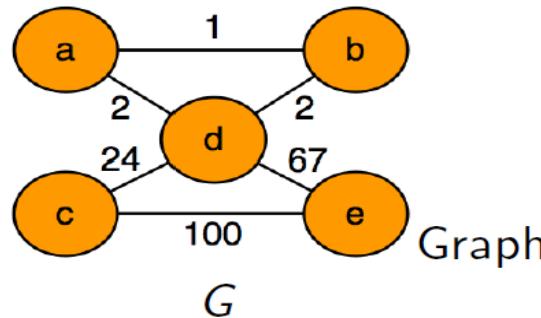
- Let  $G = (V, E)$  be a connected, undirected graph.
- For each edge  $(u, v)$  in  $E$ , we have a weight  $w(u, v)$  specifying the cost (length of edge) to connect  $u$  and  $v$ .
- We wish to find a (acyclic) subset  $T$  of  $E$  that connects all of the vertices in  $V$  and whose total weight is minimized.
- Since the total weight is minimized, the subset  $T$  must be acyclic (no circuit).
- Thus,  $T$  is a tree. We call it a **spanning tree**.
- The problem of determining the tree  $T$  is called the **minimum-spanning-tree problem**.



# Minimum Spanning Tree (MST)

## Uniqueness of MST

The minimum-cost spanning tree may not be unique!





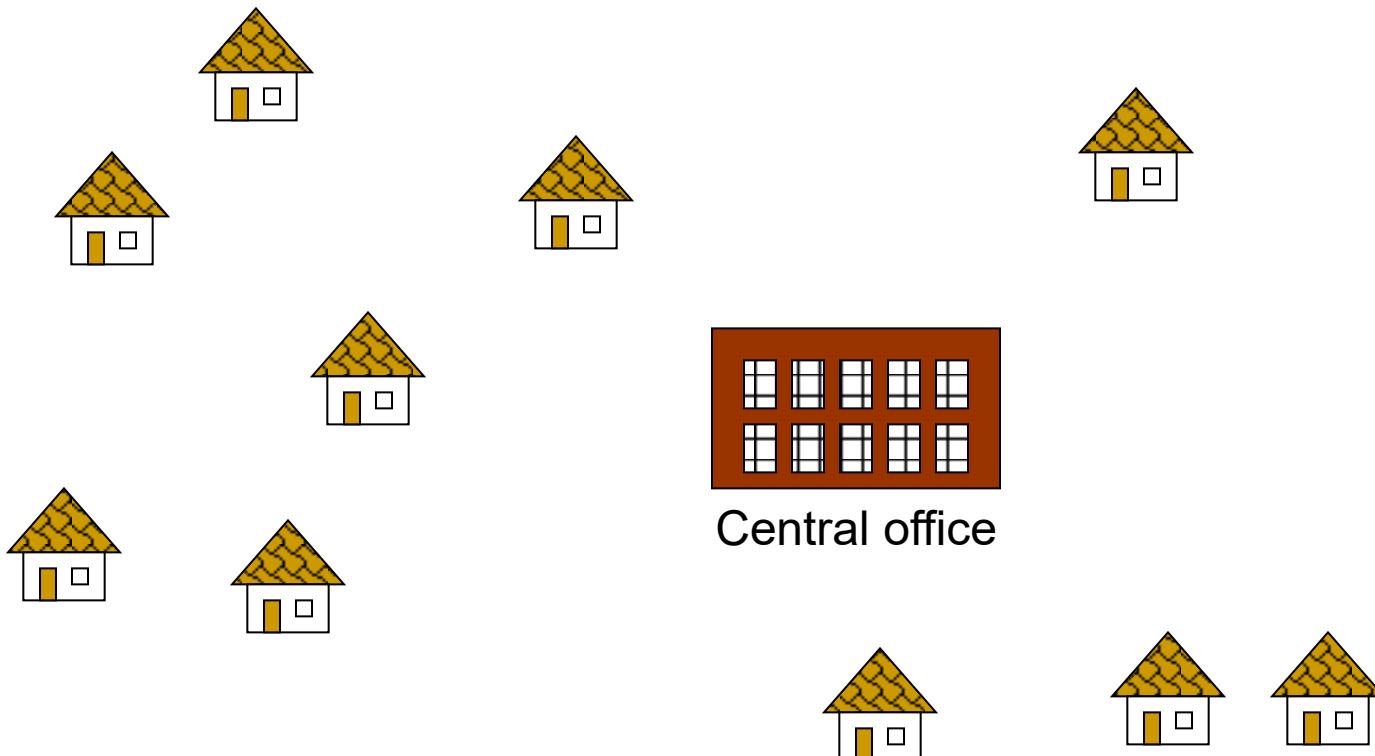
## Application.....

In the design of electronic circuitry, it is often necessary to make a set of pins electrically equivalent by wiring them together.

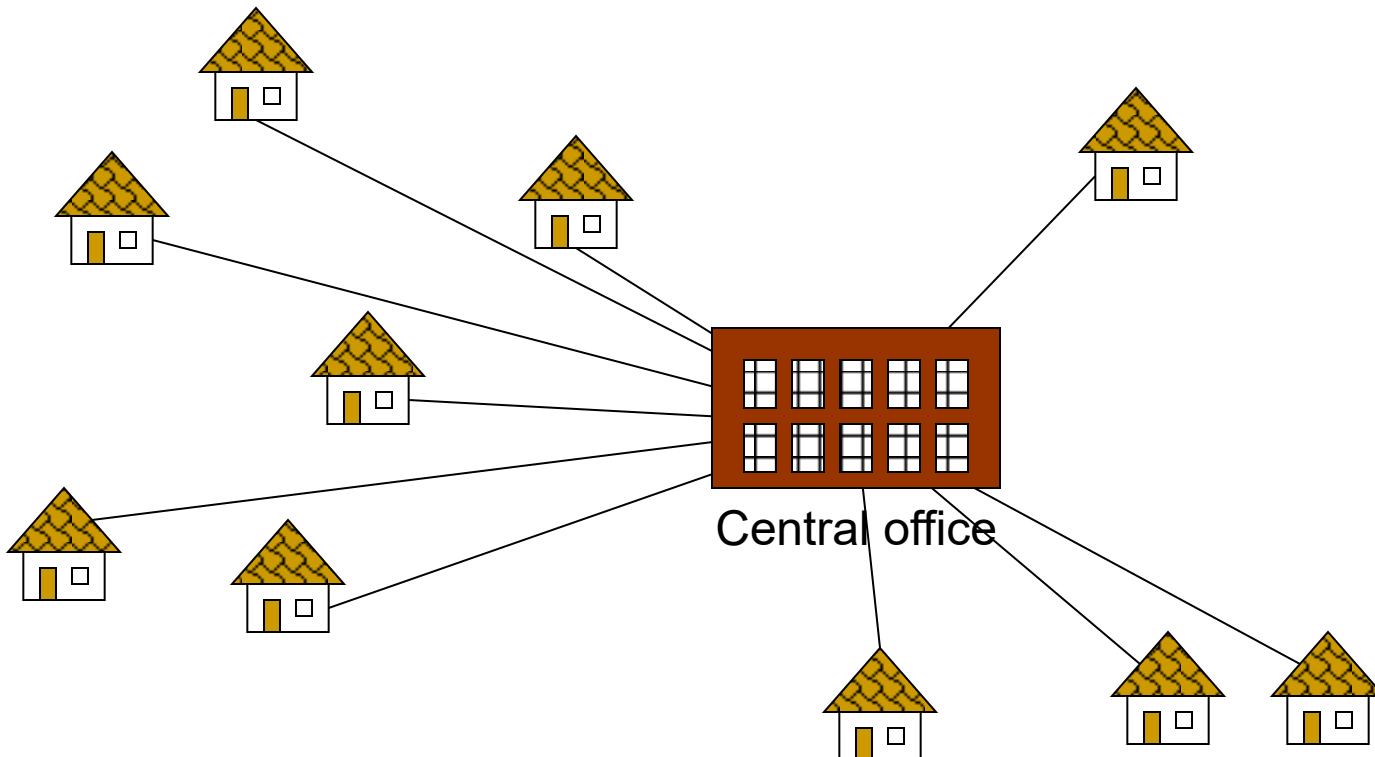
To interconnect  $n$  pins, we can use  $n-1$  wires, each connecting two pins.  
We want to minimize the total length of the wires.

Minimum Spanning Trees can be used to model this problem.

## ↗ Problem: Laying Telephone Wire

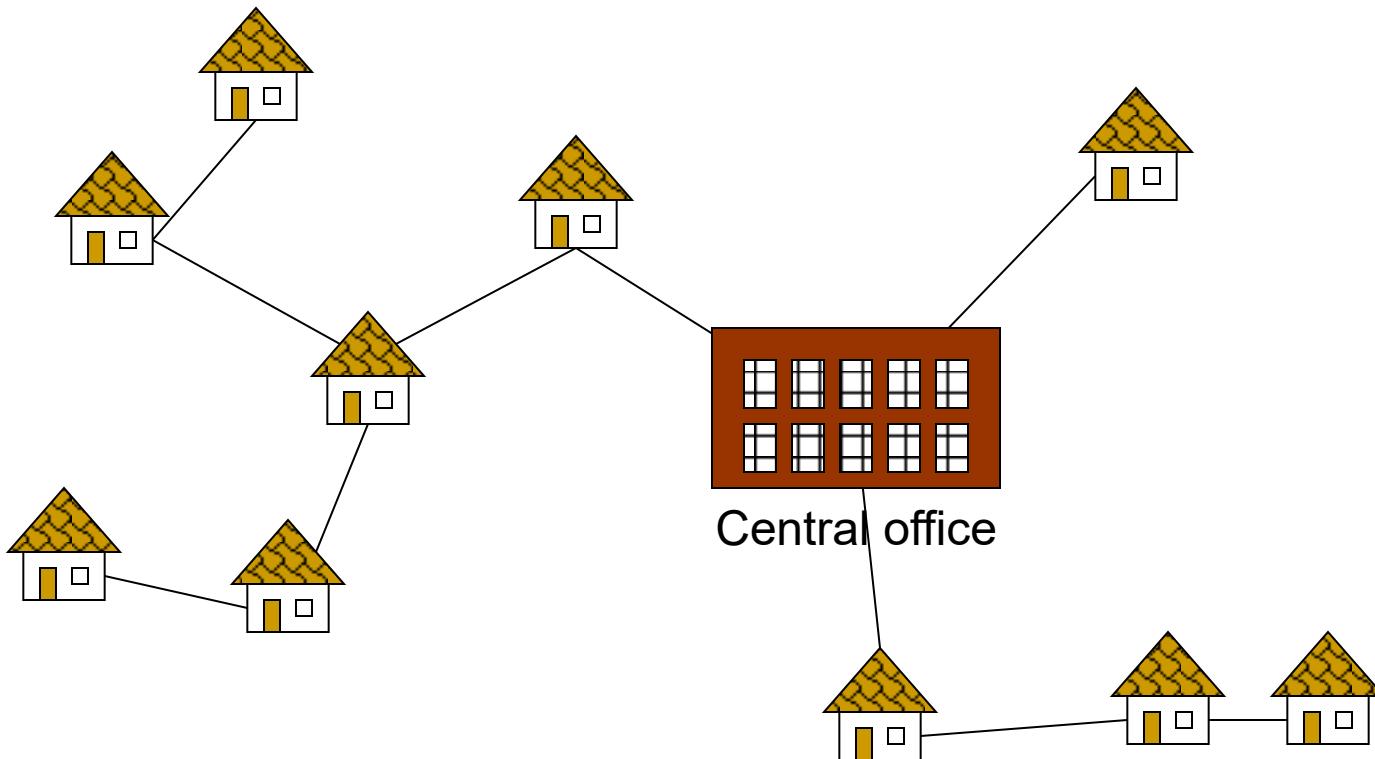


## ➤ Wiring: Naïve Approach



**Expensive!**

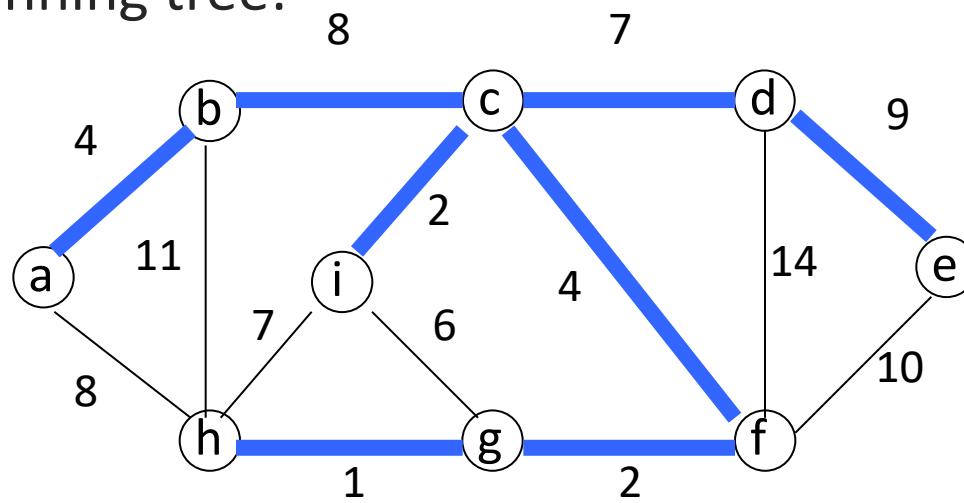
## → Wiring: Better Approach



Minimize the total length of wire connecting the customers



Here is an example of a connected graph and its minimum spanning tree:



Notice that the tree is not unique:  
replacing (b,c) with (a,h) yields another spanning tree with the same minimum weight.

# Growing a MST(Generic Algorithm)

- ↗ Set  $A$  is always a subset of some minimum spanning tree. This property is called the **invariant Property**.
- ↗ An edge  $(u, v)$  is a **safe edge** for  $A$  if adding the edge to  $A$  does not destroy the invariant.
- ↗ A **safe edge** is just the CORRECT edge to choose to add to  $T$ .

```
GENERIC_MST(G,w)
```

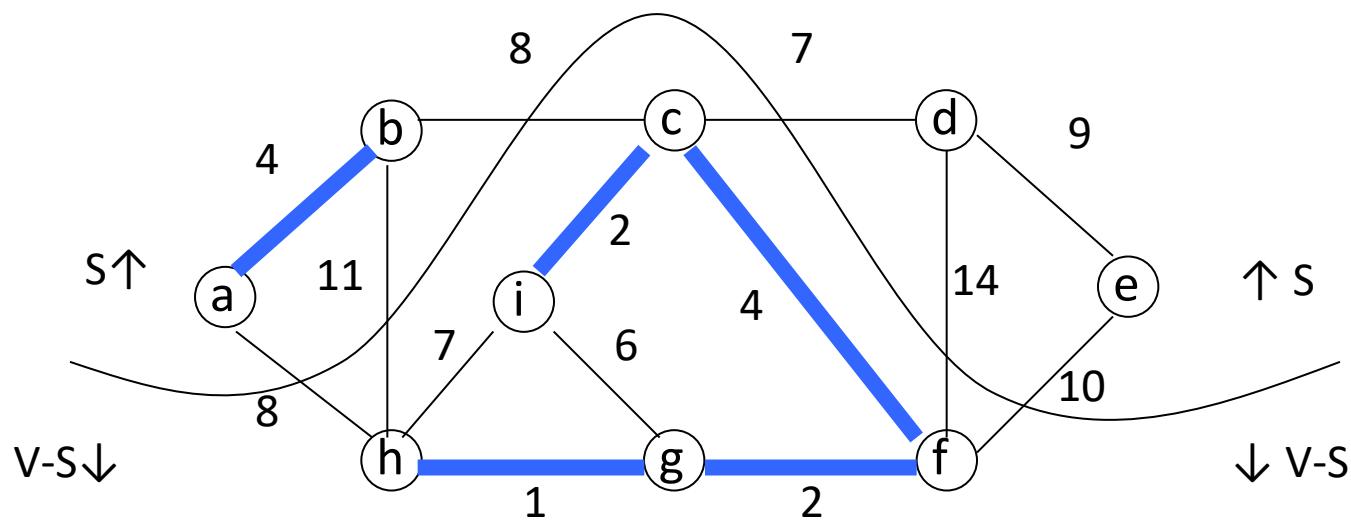
```
1      A =  $\emptyset$ 
2      while A does not form a spanning tree
3          find an edge  $(u,v)$  that is safe for A
4          A = A  $\cup \{(u,v)\}$ 
5      return A
```

# How to Find a Safe Edge

We need some definitions and a theorem.

- A **cut**  $(S, V - S)$  of an undirected graph  $G = (V, E)$  is a partition of  $V$ .
- An edge **crosses** the cut  $(S, V - S)$  if one of its endpoints is in  $S$  and the other is in  $V - S$ .
- An edge is a **light edge** crossing a cut if its weight is the minimum of any edge crossing the cut.

# How to Find a Safe Edge



- This figure shows a cut  $(S, V-S)$  of the graph.
- The edge  $(d, c)$  is the unique light edge crossing the cut.

# The Algorithms of Kruskal and Prim

- The two algorithms are elaborations of the generic algorithm.
- They each use a specific rule to determine a safe edge in line 3 of GENERIC\_MST.
- In Kruskal's algorithm,
  - The set  $A$  is a forest.
  - The safe edge added to  $A$  is always a least-weight edge in the graph that connects two distinct components.
- In Prim's algorithm,
  - The set  $A$  forms a single tree.
  - The safe edge added to  $A$  is always a least-weight edge connecting the tree to a vertex not in the tree.

# Related Topics

## Disjoint-Set

- Keep a collection of disjoint sets:  $S_1, S_2, \dots, S_k$ ,
  - Each  $S_i$  is a set, e.g.,  $S_1 = \{v_1, v_2, v_8\}$ .
- Three operations
  - **Make-Set(x)** - creates a new set whose only member is  $x$ .
  - **Union(x, y)** – unites the sets that contain  $x$  and  $y$ , say,  $S_x$  and  $S_y$ , into a new set that is the union of the two sets.
  - **Find-Set(x)** - returns a pointer to the representative of the set containing  $x$ .

# Kruskal's Algorithm

MST\_KRUSKAL( $G, w$ )

- 1         $A = \emptyset$
- 2        **for** each vertex  $v \in G.V$
- 3            MAKE\_SET( $v$ )
- 4        sort the edges of  $G.E$  into nondecreasing order by weight  $w$
- 5        **for** each edge  $(u, v) \in G.E$ , taken in nondecreasing order  
          by weight
- 6        if  $\text{FIND\_SET}(u) \neq \text{FIND\_SET}(v)$
- 7                     $A = A \cup \{(u, v)\}$
- 8                    UNION( $u, v$ )
- 9        return  $A$

# Kruskal's algorithm

- ↗ Basic idea:
  - ↗ Grow many small trees
  - ↗ Find two trees that are closest (i.e., connected with the lightest edge), join them with the lightest edge
  - ↗ Terminate when a single tree forms

# Time Complexity of Kruskal's Algorithm

MST\_KRUSKAL(G,w)

1  $A = \emptyset$

2 for each vertex  $v \in G.V$

3     MAKE\_SET( $v$ )

4 sort the edges of  $G.E$  into  
nondecreasing order by  
weight  $w$

5 for each edge  $(u, v) \in G.E$ , taken  
in nondecreasing order by weight

6 if  $\text{FIND\_SET}(u) \neq \text{FIND\_SET}(v)$

7      $A = A \cup \{(u, v)\}$

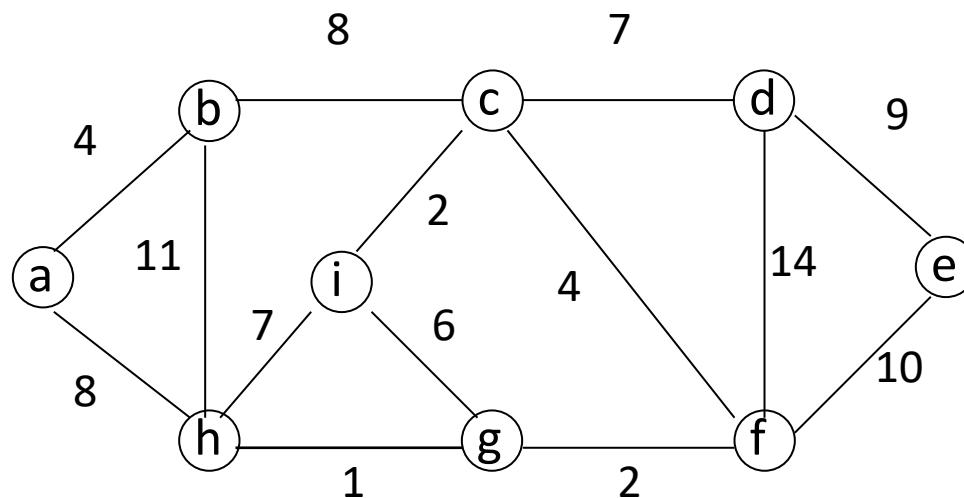
8     UNION( $u, v$ )

9 return  $A$

- Line 4 sorts  $E$  edges => it takes  $O(E \lg E)$  time, if we use MergeSort
- But  $\lg E = O(\lg V)$ , since  $E < V^2 \Rightarrow \lg E < 2 \lg V \Rightarrow \lg E$  is  $O(\lg V)$   
Hence  $O(E \lg E) = O(E \lg V)$
- If we make total  $m$  calls of Make\_set, Union, and Find\_set in any algorithm, then these calls will take total  $O(m \lg n)$  time, where  $n$  is the number of Make-Set calls.  
  
The 1<sup>st</sup> for loop makes  $O(V)$  MAKE\_SET() calls and the 2<sup>nd</sup> for loop makes  $O(E)$  Find\_SET and UNION calls
  - Total  $O(V+E)$  calls of MAKE\_SET, FIND\_SET, and UNION, out of which  $O(V)$  calls were MAKE\_SET calls =>  $O((V+E) \lg V)$  time
  - $E \geq V-1$ , since the graph is connected
    - ↗ =>  $V$  is  $O(E) \Rightarrow O((V+E) \lg V) = O(E \lg V)$
- Hence total time =  $O(E \lg V)$

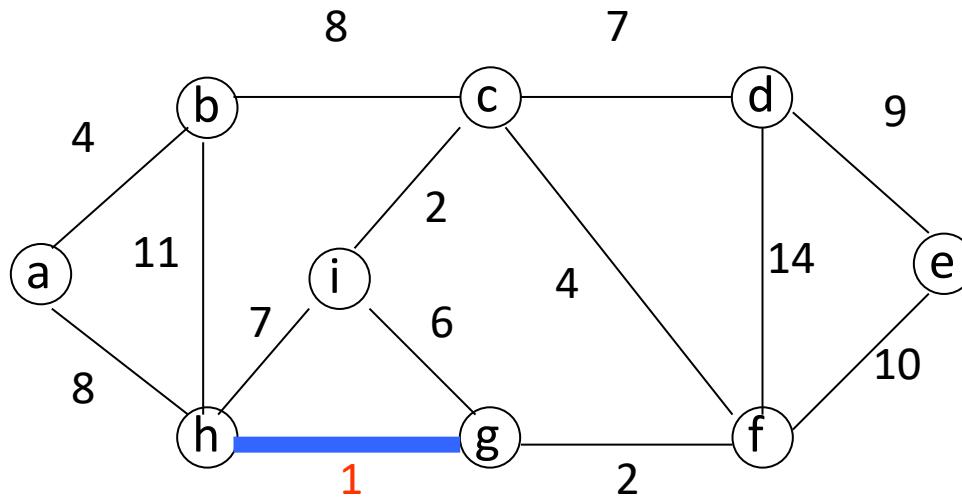
# Kruskal's Algorithm: Example

- The edges considered by the algorithm are sorted by weight.
- The edge under consideration at each step is shown with a red weight number.



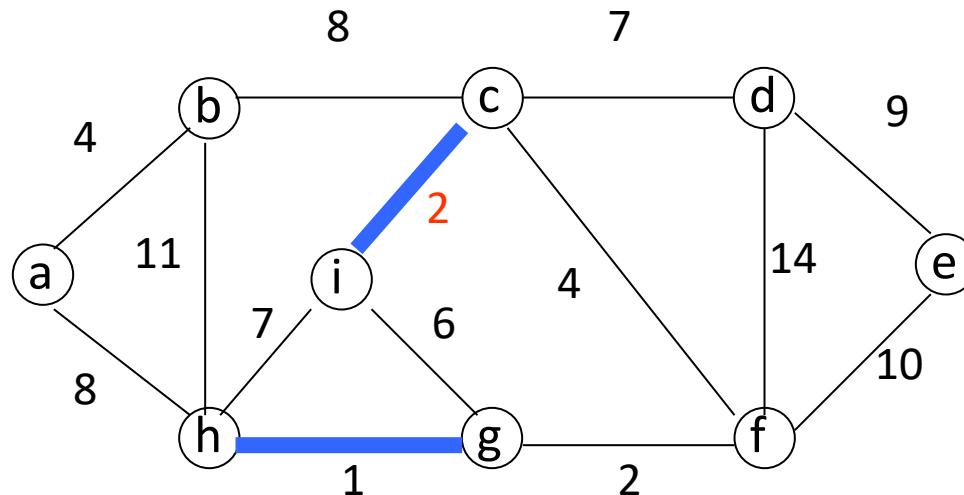
# Kruskal's Algorithm: Example

- The edges are considered by the algorithm in sorted order by weight.
- The edge under consideration at each step is shown with a red weight number.



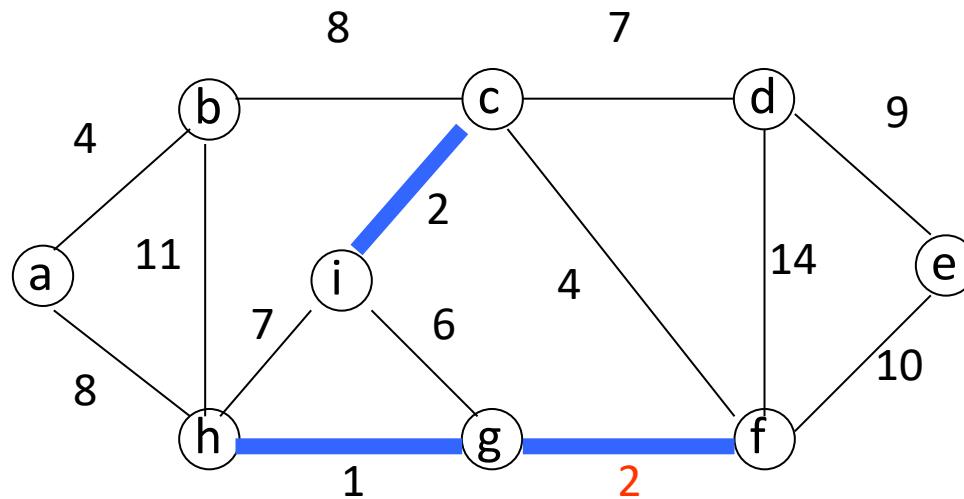
# Kruskal's Algorithm: Example

- The edges are considered by the algorithm in sorted order by weight.
- The edge under consideration at each step is shown with a red weight number.



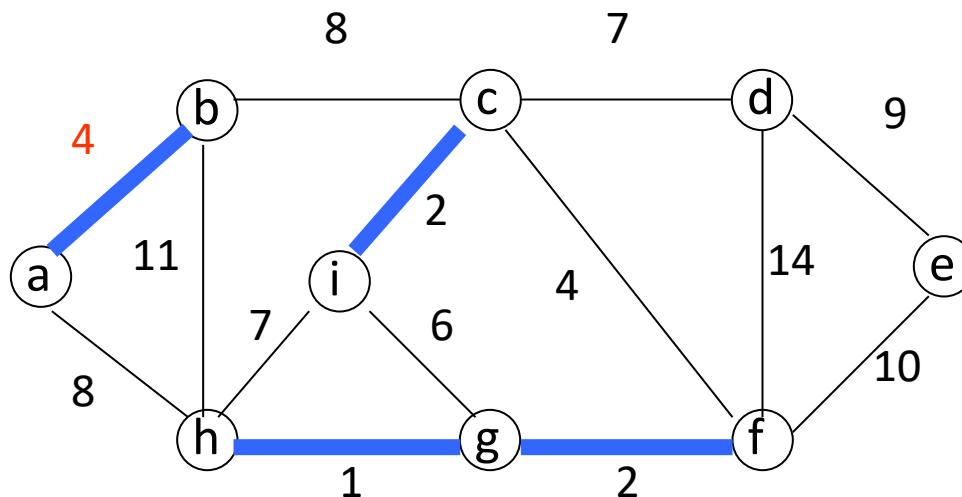
# Kruskal's Algorithm: Example

- The edges are considered by the algorithm in sorted order by weight.
- The edge under consideration at each step is shown with a red weight number.



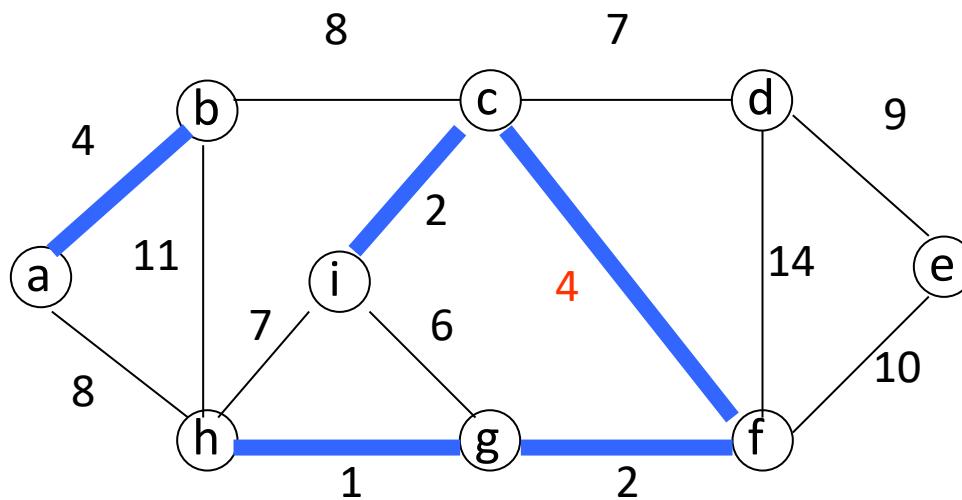
# Kruskal's Algorithm: Example

- The edges are considered by the algorithm in sorted order by weight.
- The edge under consideration at each step is shown with a red weight number.



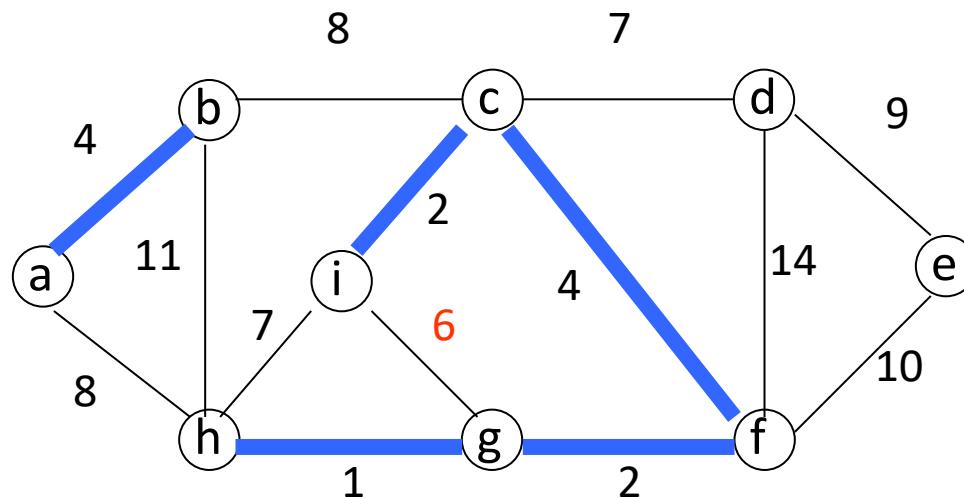
# Kruskal's Algorithm: Example

- The edges are considered by the algorithm in sorted order by weight.
- The edge under consideration at each step is shown with a red weight number.



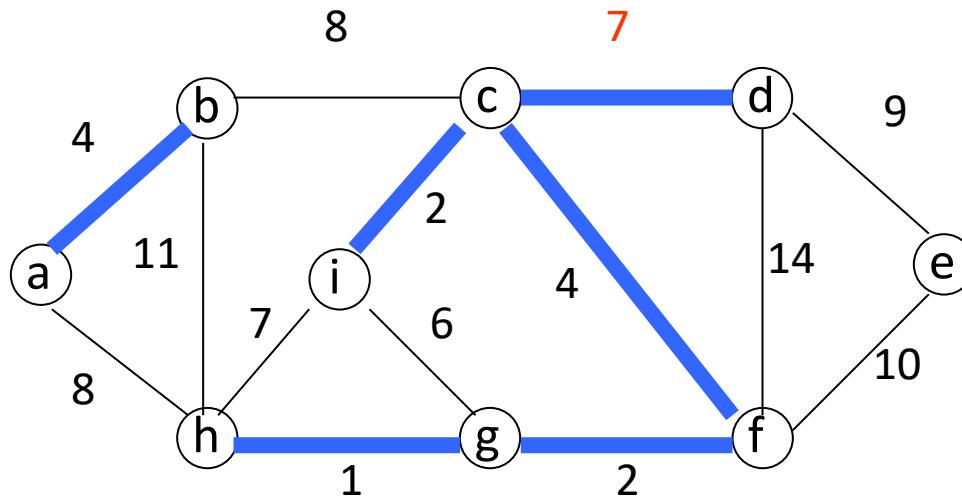
# Kruskal's Algorithm: Example

- The edges are considered by the algorithm in sorted order by weight.
- The edge under consideration at each step is shown with a red weight number.



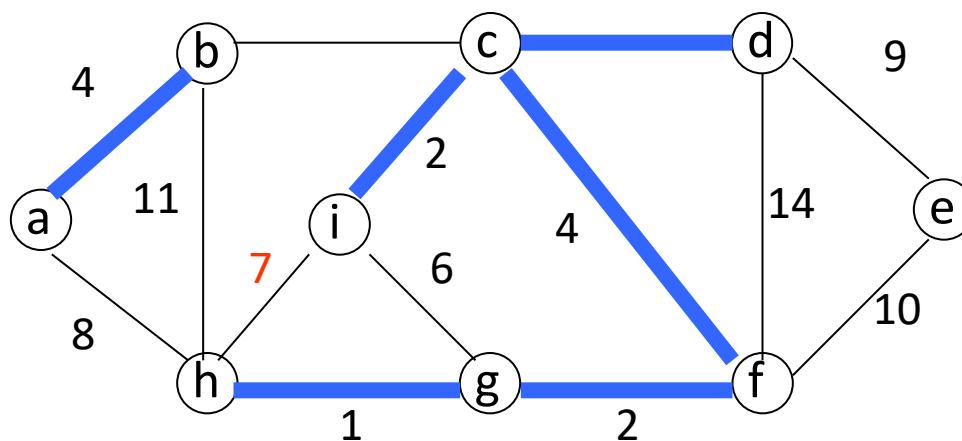
# Kruskal's Algorithm: Example

- The edges are considered by the algorithm in sorted order by weight.
- The edge under consideration at each step is shown with a red weight number.



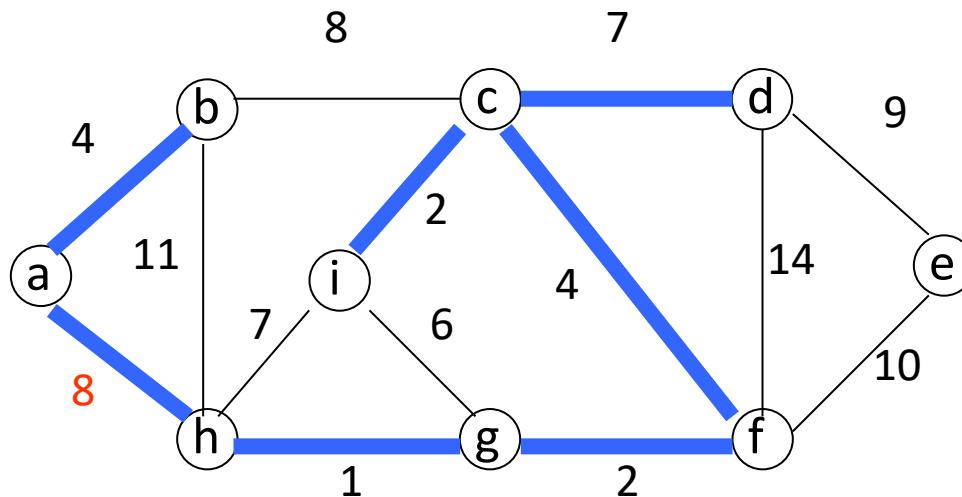
# Kruskal's Algorithm: Example

- The edges are considered by the algorithm in sorted order by weight.
- The edge under consideration at each step is shown with a red weight number.



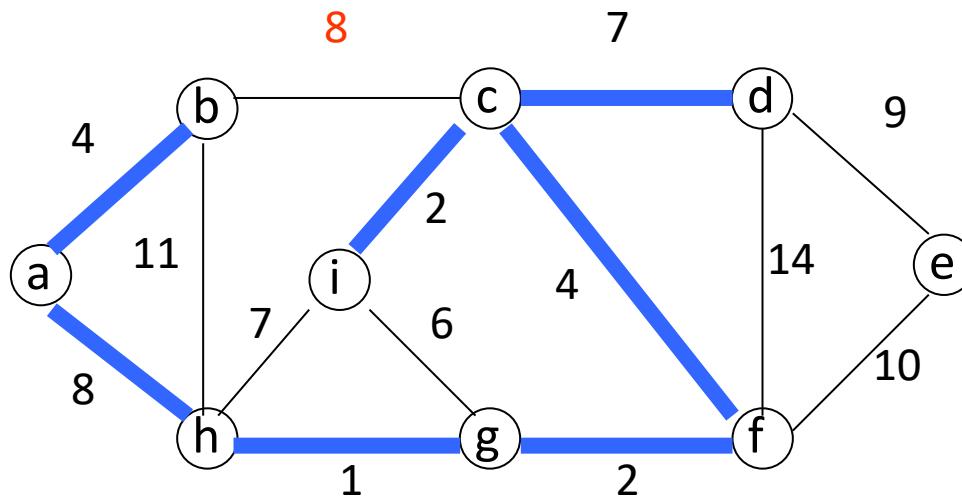
# Kruskal's Algorithm: Example

- The edges are considered by the algorithm in sorted order by weight.
- The edge under consideration at each step is shown with a red weight number.



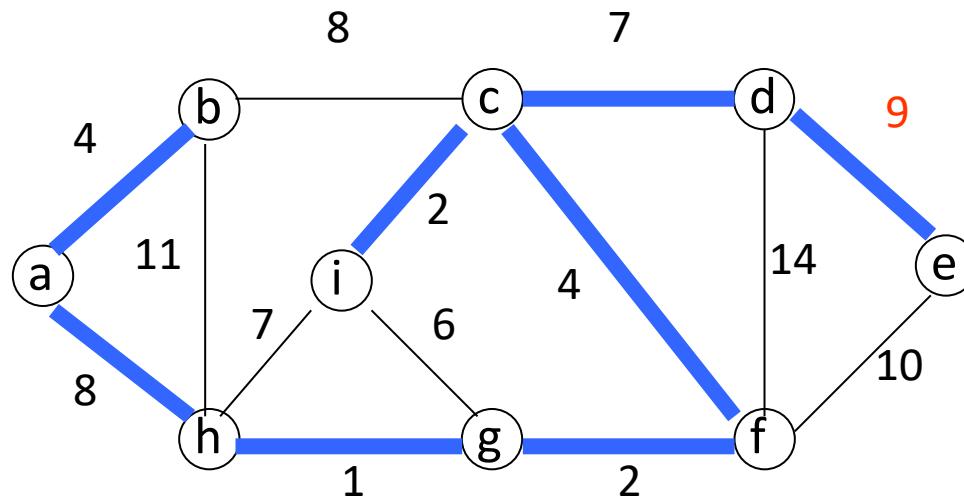
# Kruskal's Algorithm: Example

- The edges are considered by the algorithm in sorted order by weight.
- The edge under consideration at each step is shown with a red weight number.



# Kruskal's Algorithm: Example

- The edges are considered by the algorithm in sorted order by weight.
- The edge under consideration at each step is shown with a red weight number.



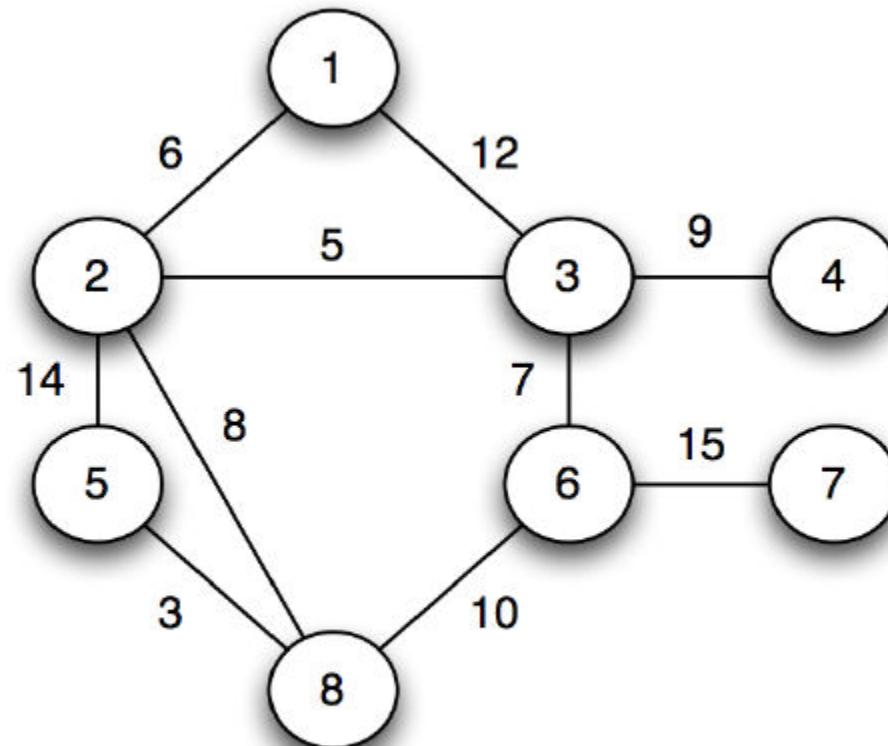
# Kruskal's algorithm in action

$w(u, v)$	$(u, v)$
3	(5,8)
5	(2,3)
6	(1,2)
7	(3,6)
8	(2,8)
9	(3,4)
10	(6,8)
12	(1,3)
14	(2,5)
15	(6,7)

$$|V| = 8$$

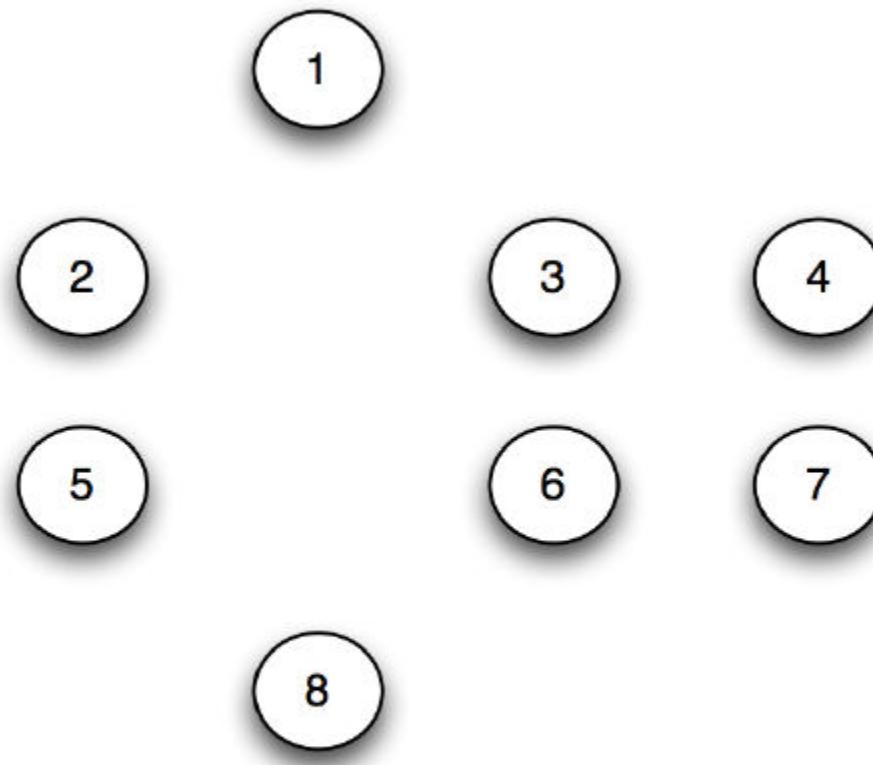
$$|E| = 10$$

$$|T| = 0$$



# Kruskal's algorithm in action

$w(u, v)$	$(u, v)$
3	(5,8)
5	(2,3)
6	(1,2)
7	(3,6)
8	(2,8)
9	(3,4)
10	(6,8)
12	(1,3)
14	(2,5)
15	(6,7)



$$|V| = 8$$

$$|E| = 10$$

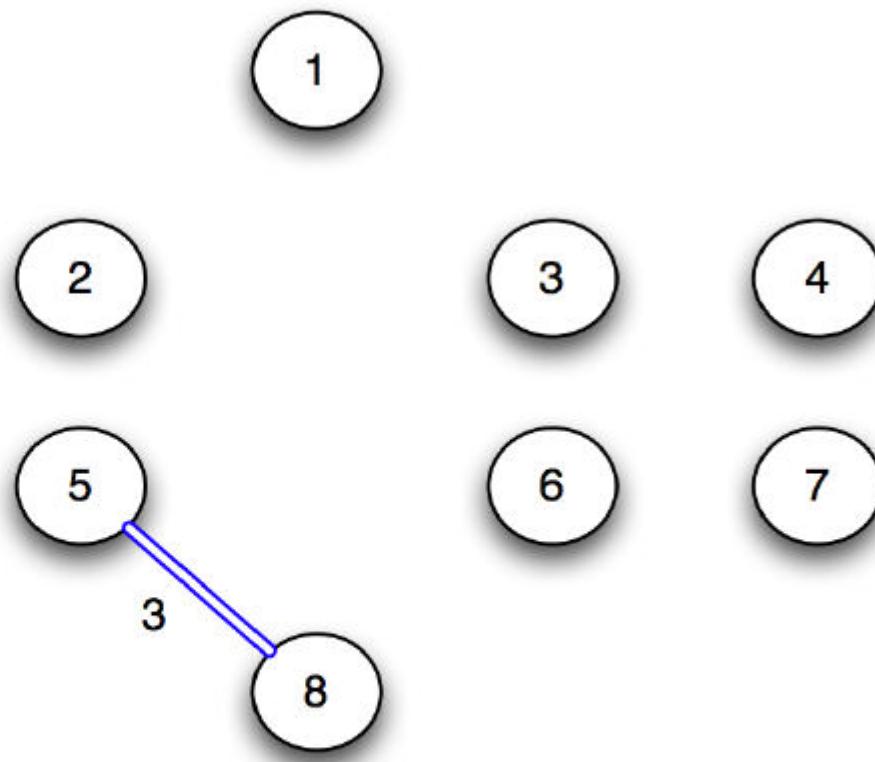
$$|T| = 0$$

**Vertex sets:**

{1}, {2}, {3}, {4}, {5}, {6}, {7}, {8}

# Kruskal's algorithm in action

$w(u, v)$	$(u, v)$
yellow 3	(5,8)
5	(2,3)
6	(1,2)
7	(3,6)
8	(2,8)
9	(3,4)
10	(6,8)
12	(1,3)
14	(2,5)
15	(6,7)



$$|V| = 8$$

$$|E| = 10$$

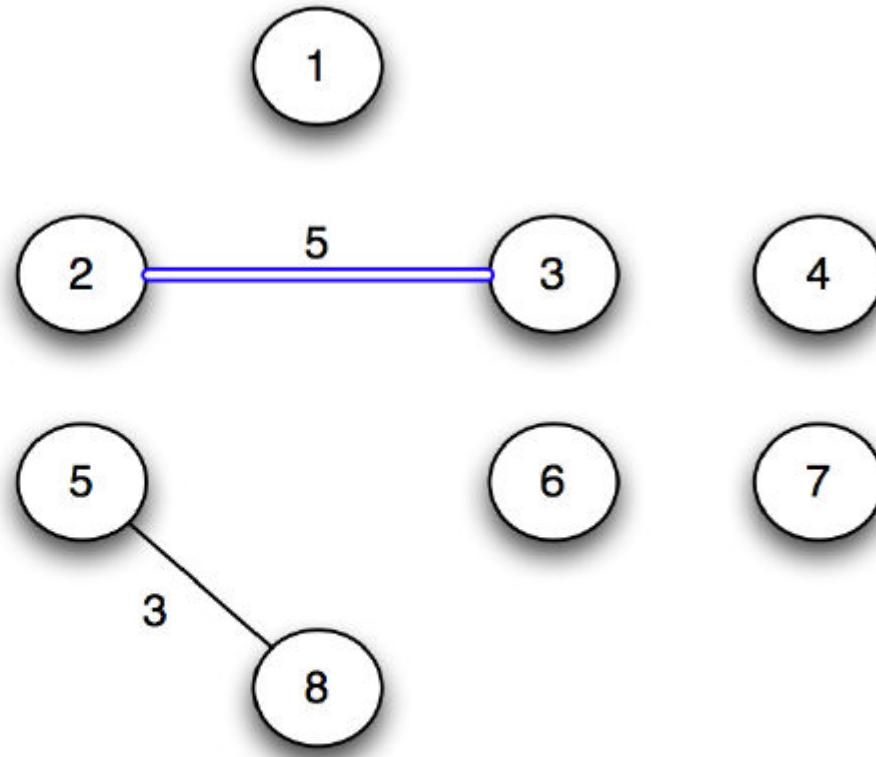
$$|T| = 1$$

**Vertex sets:**

$$\{1\}, \{2\}, \{3\}, \{4\}, \{5\}, \{6\}, \{7\}, \{8\} \implies \{1\}, \{2\}, \{3\}, \{4\}, \boxed{\{5, 8\}}, \{6\}, \{7\}$$

# Kruskal's algorithm in action

$w(u, v)$	$(u, v)$
✓ 3	(5,8)
yellow 5	(2,3)
6	(1,2)
7	(3,6)
8	(2,8)
9	(3,4)
10	(6,8)
12	(1,3)
14	(2,5)
15	(6,7)



$$|V| = 8$$

$$|E| = 10$$

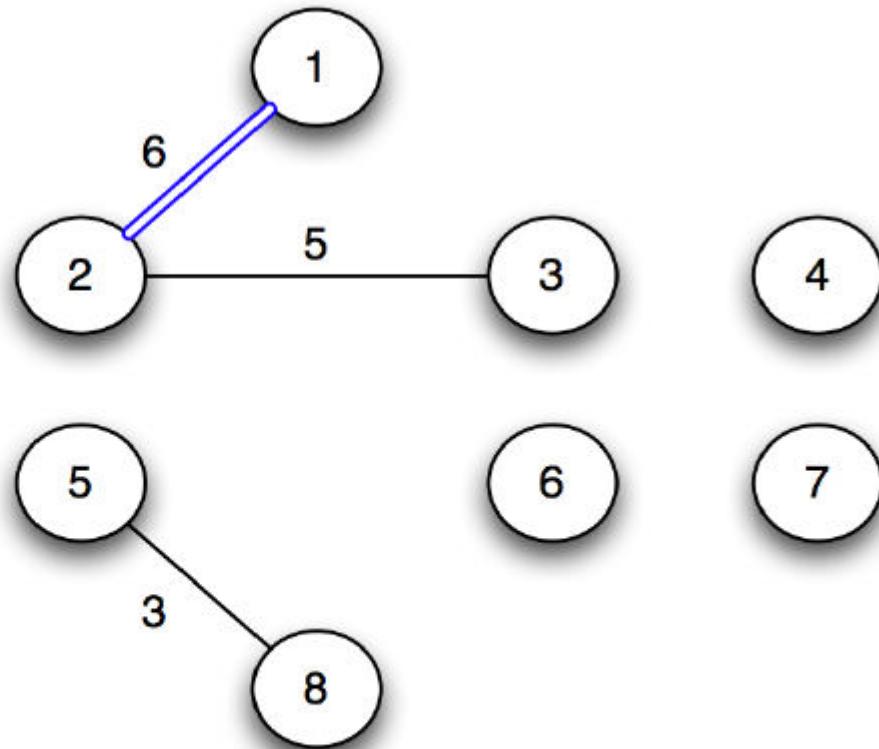
$$|T| = 2$$

Vertex sets:

$$\{1\}, \{2\}, \{3\}, \{4\}, \{5, 8\}, \{6\}, \{7\} \Rightarrow \{1\}, \boxed{\{2, 3\}}, \{4\}, \{5, 8\}, \{6\}, \{7\}$$

# Kruskal's algorithm in action

$w(u, v)$	$(u, v)$
✓ 3	(5,8)
✓ 5	(2,3)
yellow 6	(1,2)
7	(3,6)
8	(2,8)
9	(3,4)
10	(6,8)
12	(1,3)
14	(2,5)
15	(6,7)



$$|V| = 8$$

$$|E| = 10$$

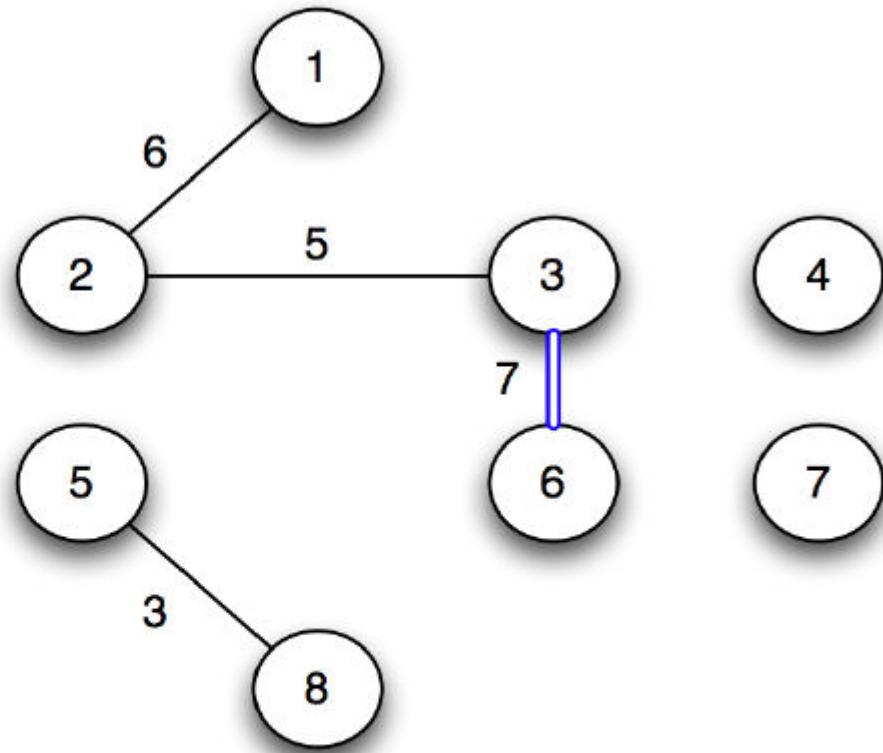
$$|T| = 3$$

Vertex sets:

$$\{1\}, \{2, 3\}, \{4\}, \{5, 8\}, \{6\}, \{7\} \Rightarrow \boxed{\{1, 2, 3\}}, \{4\}, \{5, 8\}, \{6\}, \{7\}$$

# Kruskal's algorithm in action

$w(u, v)$	$(u, v)$
✓ 3	(5,8)
✓ 5	(2,3)
✓ 6	(1,2)
yellow 7	(3,6)
8	(2,8)
9	(3,4)
10	(6,8)
12	(1,3)
14	(2,5)
15	(6,7)



$$|V| = 8$$

$$|E| = 10$$

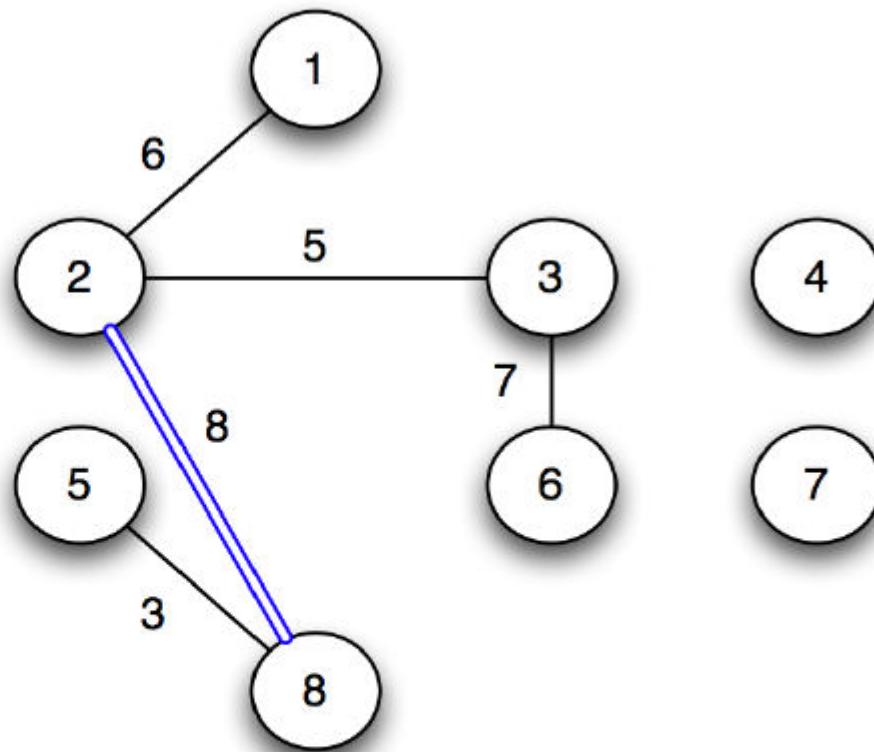
$$|T| = 4$$

Vertex sets:

$$\{1, 2, 3\}, \{4\}, \{5, 8\}, \{6\}, \{7\} \Rightarrow \boxed{\{1, 2, 3, 6\}}, \{4\}, \{5, 8\}, \{7\}$$

# Kruskal's algorithm in action

$w(u, v)$	$(u, v)$
✓ 3	(5,8)
✓ 5	(2,3)
✓ 6	(1,2)
✓ 7	(3,6)
yellow 8	(2,8)
9	(3,4)
10	(6,8)
12	(1,3)
14	(2,5)
15	(6,7)



$$|V| = 8$$

$$|E| = 10$$

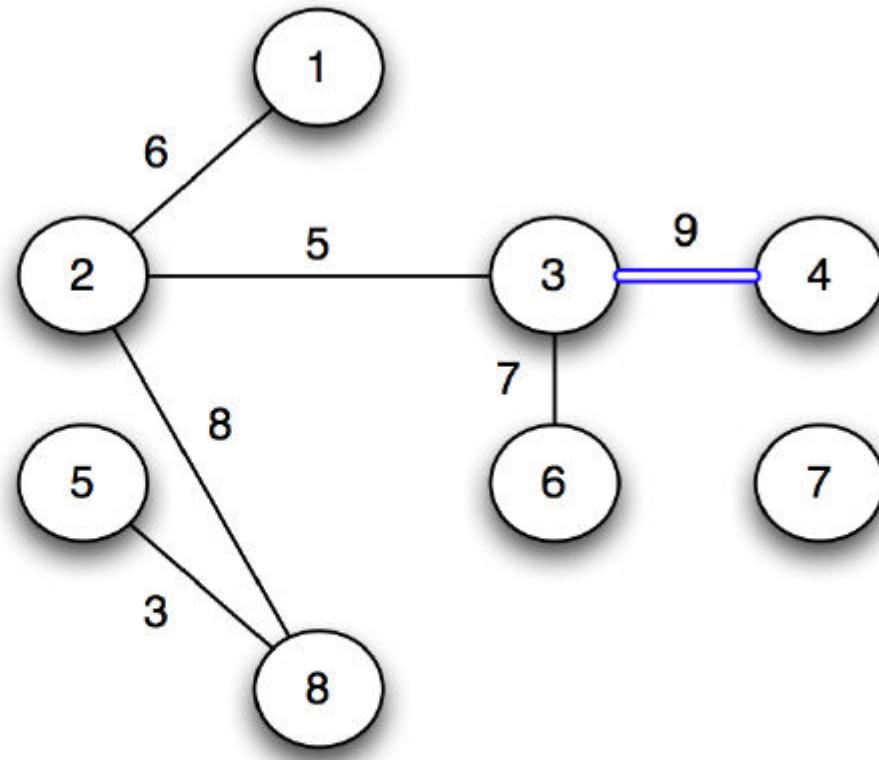
$$|T| = 5$$

Vertex sets:

$$\{1, 2, 3, 6\}, \{4\}, \{5, 8\}, \{7\} \Rightarrow \boxed{\{1, 2, 3, 5, 6, 8\}}, \{4\}, \{7\}$$

# Kruskal's algorithm in action

$w(u, v)$	$(u, v)$
✓ 3	(5,8)
✓ 5	(2,3)
✓ 6	(1,2)
✓ 7	(3,6)
✓ 8	(2,8)
yellow 9	(3,4)
10	(6,8)
12	(1,3)
14	(2,5)
15	(6,7)



$$|V| = 8$$

$$|E| = 10$$

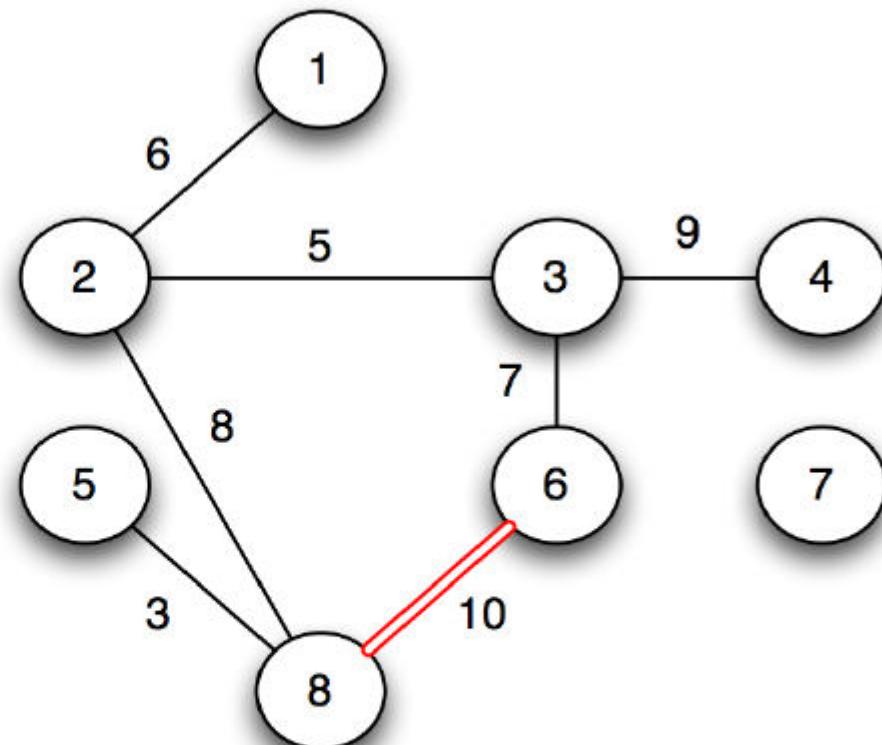
$$|T| = 6$$

Vertex sets:

$$\{1, 2, 3, 5, 6, 8\}, \{4\}, \{7\} \implies \boxed{\{1, 2, 3, 4, 5, 6, 8\}, \{7\}}$$

# Kruskal's algorithm in action

$w(u, v)$	$(u, v)$
✓ 3	(5,8)
✓ 5	(2,3)
✓ 6	(1,2)
✓ 7	(3,6)
✓ 8	(2,8)
✓ 9	(3,4)
yellow 10	(6,8)
12	(1,3)
14	(2,5)
15	(6,7)



$$|V| = 8$$

$$|E| = 10$$

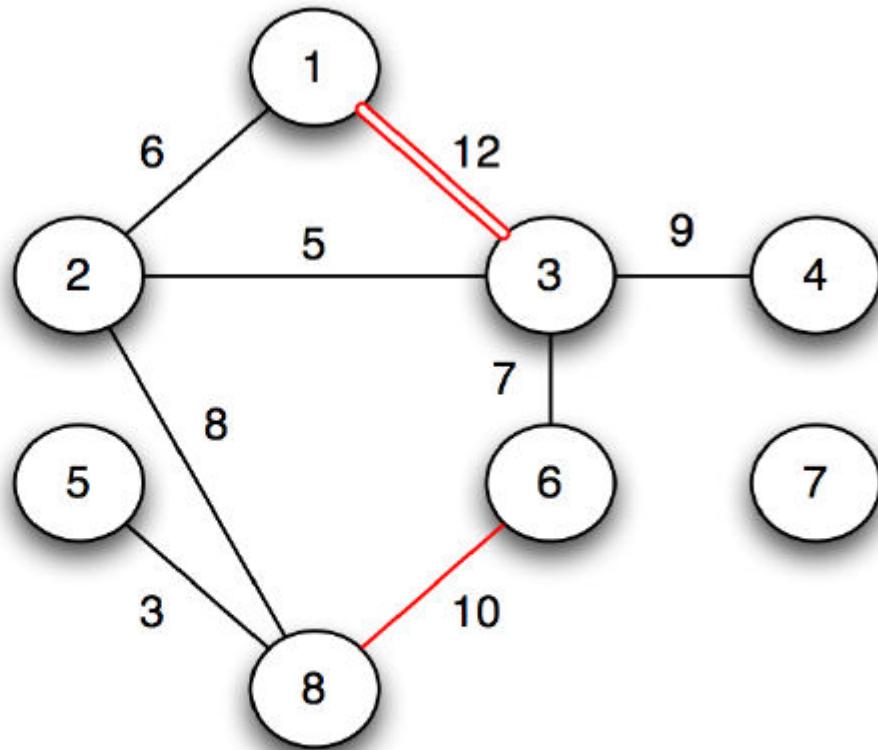
$$|T| = 6$$

Vertex sets:

$$\{1, 2, 3, 4, 5, 6, 8\}, \{7\} \implies \{1, 2, 3, 4, 5, 6, 8\}, \{7\}$$

# Kruskal's algorithm in action

$w(u, v)$	$(u, v)$
✓ 3	(5,8)
✓ 5	(2,3)
✓ 6	(1,2)
✓ 7	(3,6)
✓ 8	(2,8)
✓ 9	(3,4)
✗ 10	(6,8)
yellow 12	(1,3)
14	(2,5)
15	(6,7)



$$|V| = 8$$

$$|E| = 10$$

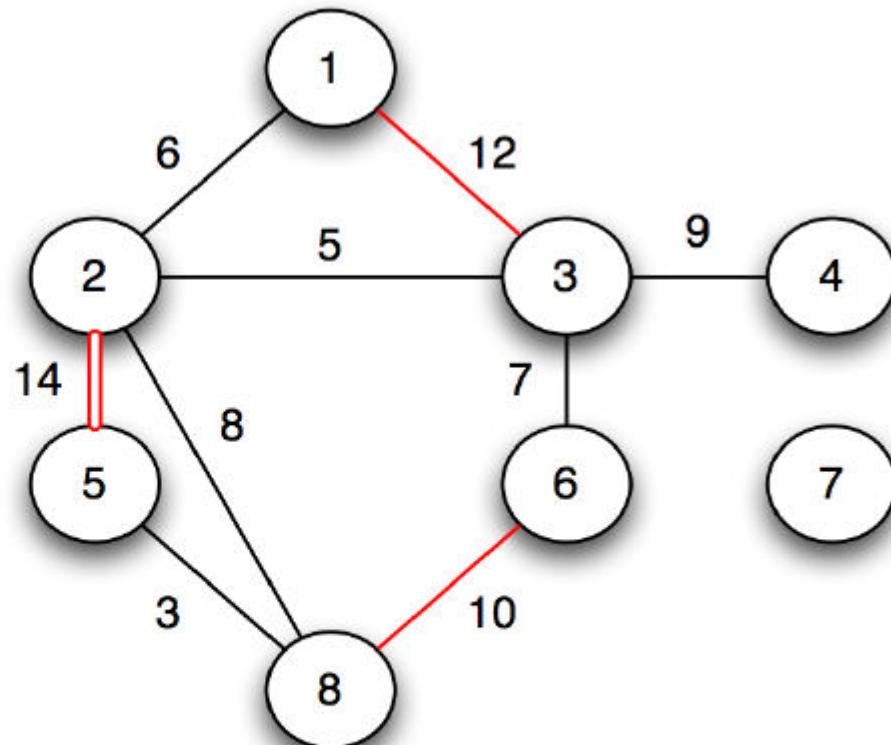
$$|T| = 6$$

Vertex sets:

$$\{1, 2, 3, 4, 5, 6, 8\}, \{7\} \Rightarrow \{1, 2, 3, 4, 5, 6, 8\}, \{7\}$$

# Kruskal's algorithm in action

$w(u, v)$	$(u, v)$
✓ 3	(5,8)
✓ 5	(2,3)
✓ 6	(1,2)
✓ 7	(3,6)
✓ 8	(2,8)
✓ 9	(3,4)
✗ 10	(6,8)
✗ 12	(1,3)
yellow 14	(2,5)
15	(6,7)



$$|V| = 8$$

$$|E| = 10$$

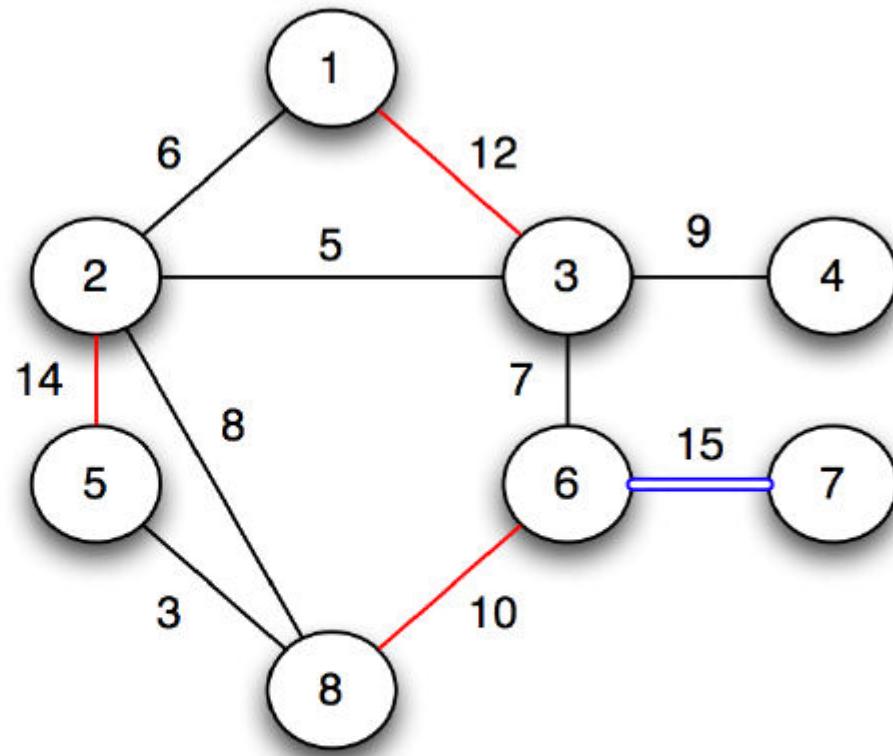
$$|T| = 6$$

Vertex sets:

$$\{1, 2, 3, 4, 5, 6, 8\}, \{7\} \Rightarrow \{1, 2, 3, 4, 5, 6, 8\}, \{7\}$$

# Kruskal's algorithm in action

$w(u, v)$	$(u, v)$
✓ 3	(5,8)
✓ 5	(2,3)
✓ 6	(1,2)
✓ 7	(3,6)
✓ 8	(2,8)
✓ 9	(3,4)
✗ 10	(6,8)
✗ 12	(1,3)
✗ 14	(2,5)
yellow 15	(6,7)



$$|V| = 8$$

$$|E| = 10$$

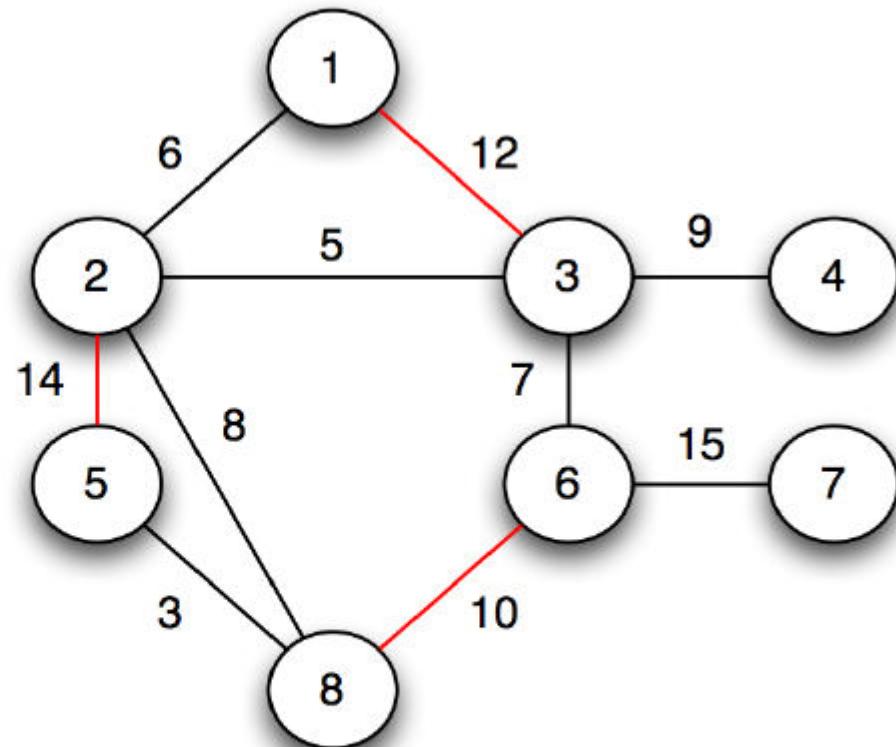
$$|T| = 7$$

Vertex sets:

$$\{1, 2, 3, 4, 5, 6, 8\}, \{7\} \implies \boxed{\{1, 2, 3, 4, 5, 6, 7, 8\}}$$

# Kruskal's algorithm in action

$w(u, v)$	$(u, v)$
✓ 3	(5,8)
✓ 5	(2,3)
✓ 6	(1,2)
✓ 7	(3,6)
✓ 8	(2,8)
✓ 9	(3,4)
✗ 10	(6,8)
✗ 12	(1,3)
✗ 14	(2,5)
✓ 15	(6,7)



$$|V| = 8$$

$$|E| = 10$$

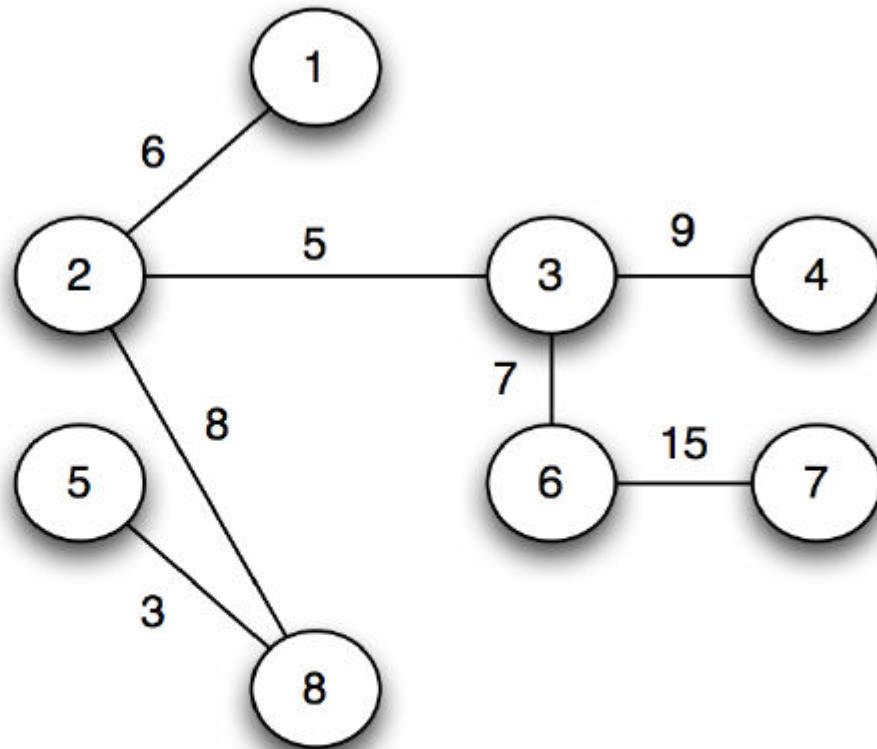
$$|T| = 7$$

Vertex sets:

{1, 2, 3, 4, 5, 6, 7, 8}

# Kruskal's algorithm in action

$w(u, v)$	$(u, v)$
✓ 3	(5,8)
✓ 5	(2,3)
✓ 6	(1,2)
✓ 7	(3,6)
✓ 8	(2,8)
✓ 9	(3,4)
✗ 10	(6,8)
✗ 12	(1,3)
✗ 14	(2,5)
✓ 15	(6,7)



$$|V| = 8$$

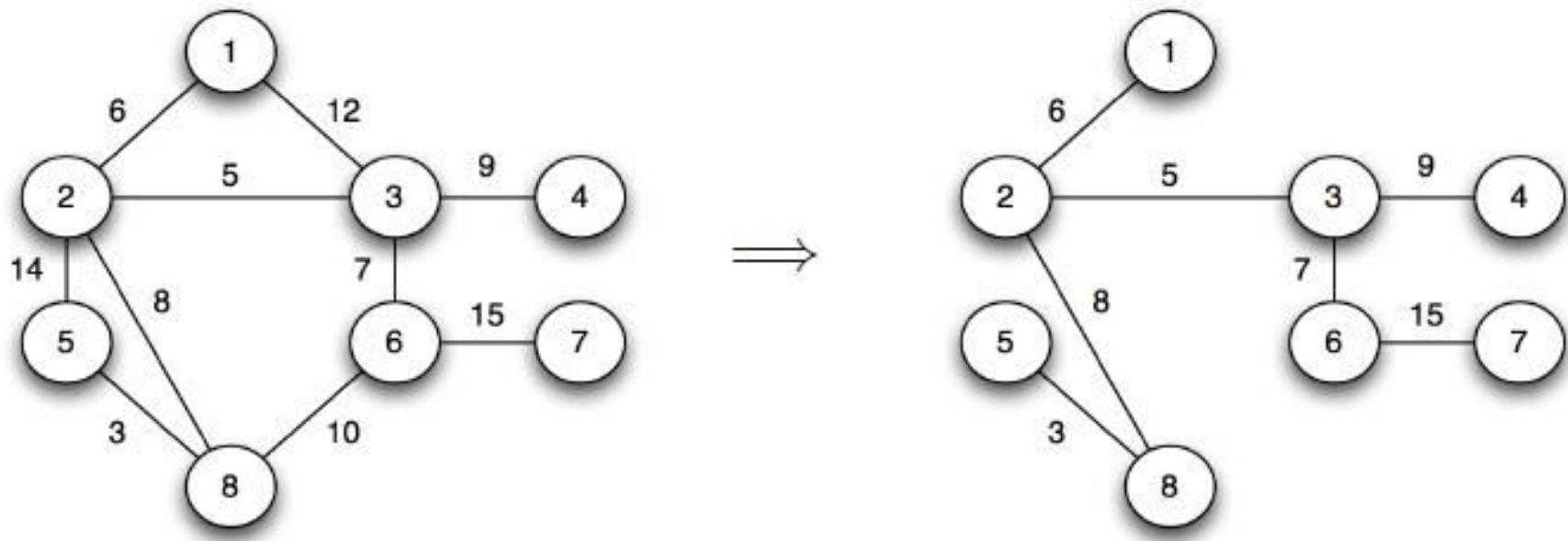
$$|E| = 10$$

$$|T| = 7$$

Vertex sets:

{1, 2, 3, 4, 5, 6, 7, 8}

# Kruskal's algorithm in action



# Prim's Algorithm

**MST\_PRIM(G,w,r)**

```
1   for each  $u$  in  $G.V$ 
2      $u.key = \infty$ 
3      $u.\pi = \text{NIL}$ 
4      $r.key = 0$ 
5      $Q = G.V$  //Q is a min-priority queue
6     while  $Q \neq \emptyset$ 
7        $u = \text{EXTRACT\_MIN}(Q)$ 
8       for each  $v \in G.Adj[u]$ 
9         if  $v \in Q$  and  $w(u, v) < v.key$ 
10            $v.\pi = u$ 
11            $v.key = w(u, v)$ 
```

# Prim's Algorithm

- ↗ Grow the minimum spanning tree from the root vertex  $r$ .
- ↗  $Q$  is a priority queue, holding all vertices that are not in the tree now.
- ↗  $\text{key}[v]$  is the minimum weight of any edge connecting  $v$  to a vertex in the tree.
- ↗  $\text{parent}[v]$  names the parent of  $v$  in the tree.
- ↗ When the algorithm terminates,  $Q$  is empty; the minimum spanning tree  $A$  for  $G$  is thus  $A=\{(v,\text{parent}[v]): v \in V - \{r\}\}$ .
- ↗ Running time:  $O(E \lg V)$ .

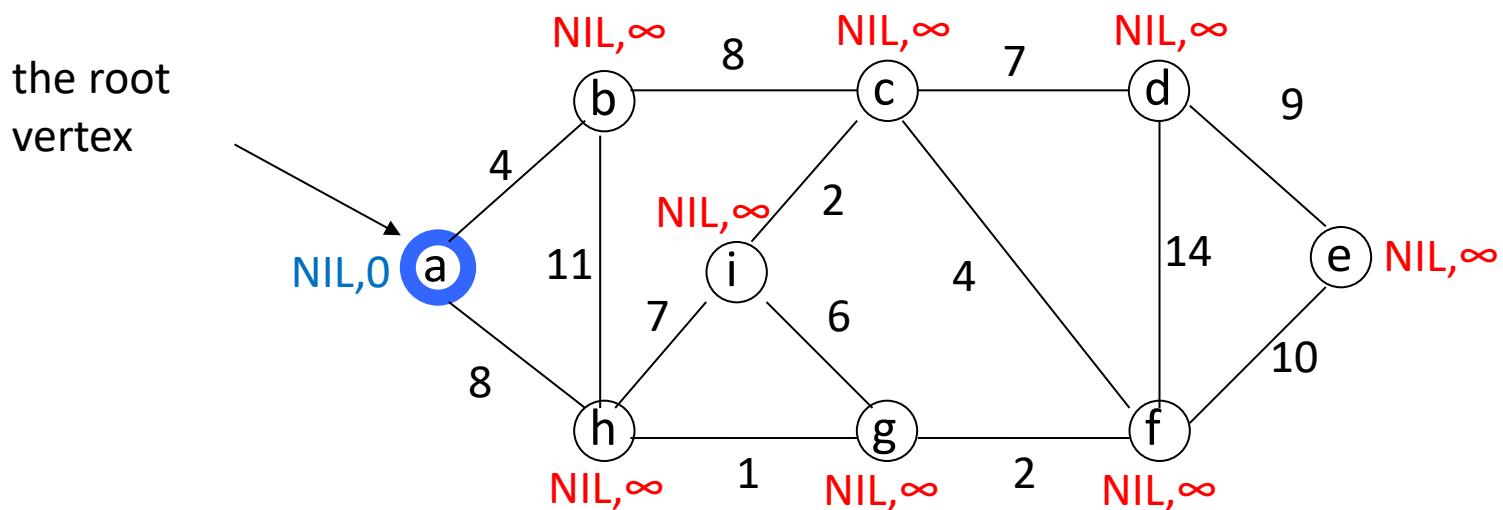
# Running time of Prim's Algorithm

MST\_PRIM( $G, w, r$ )

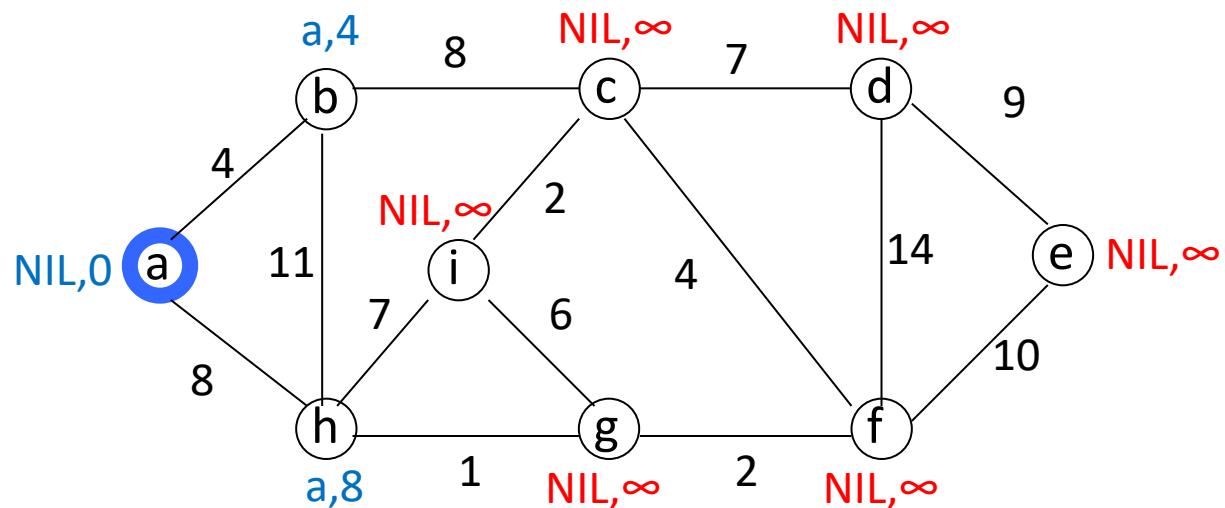
```
1   for each  $u$  in  $G.V$ 
2      $u.key = \infty$            //  $O(V)$ 
3      $u.\pi = \text{NIL}$ 
4      $r.key = 0$ 
5      $Q = G.V$     //  $Q$  is a min-priority queue      //  $\Theta(V)$  time to build the heap
6     while  $Q \neq \emptyset$ 
7        $u = \text{EXTRACT\_MIN}(Q)$       //  $V$  Heap-Extract-Min operations:  $O(V \lg V)$ 
8       for each  $v \in G.Adj[u]$ 
9         if  $v \in Q$  and  $w(u, v) < v.key$ 
10           $v.\pi = u$ 
11           $v.key = w(u, v)$  //  $E$  Heap-Decrease-Key operations:  $O(E \lg V)$ 
```

Total time =  $O((V+E) \lg V) = O(E \lg V)$  since  $E \geq V-1 \Rightarrow V$  is  $O(E)$

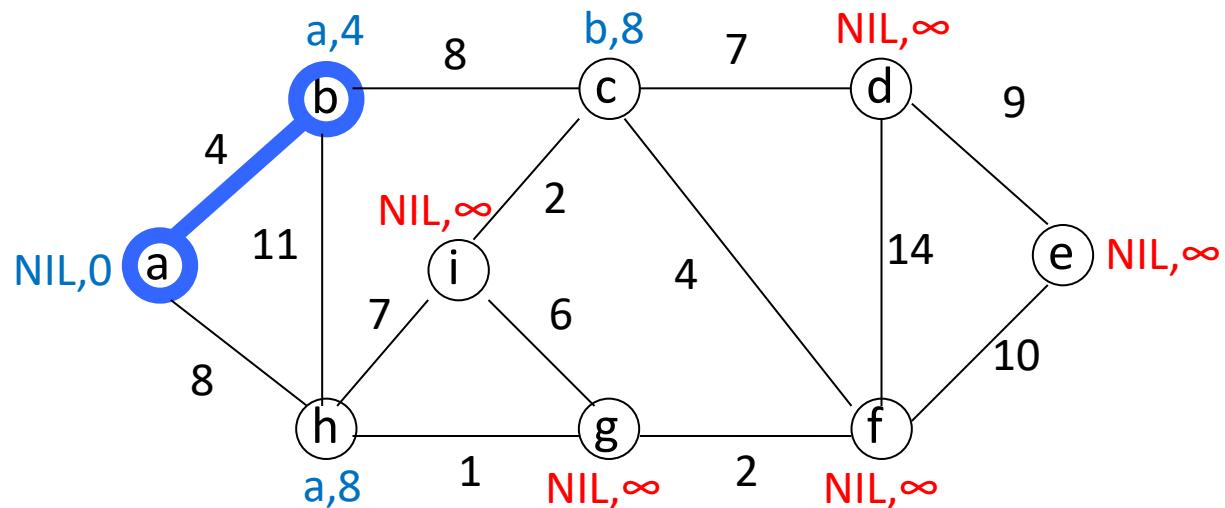
# Prim's Algorithm: Example



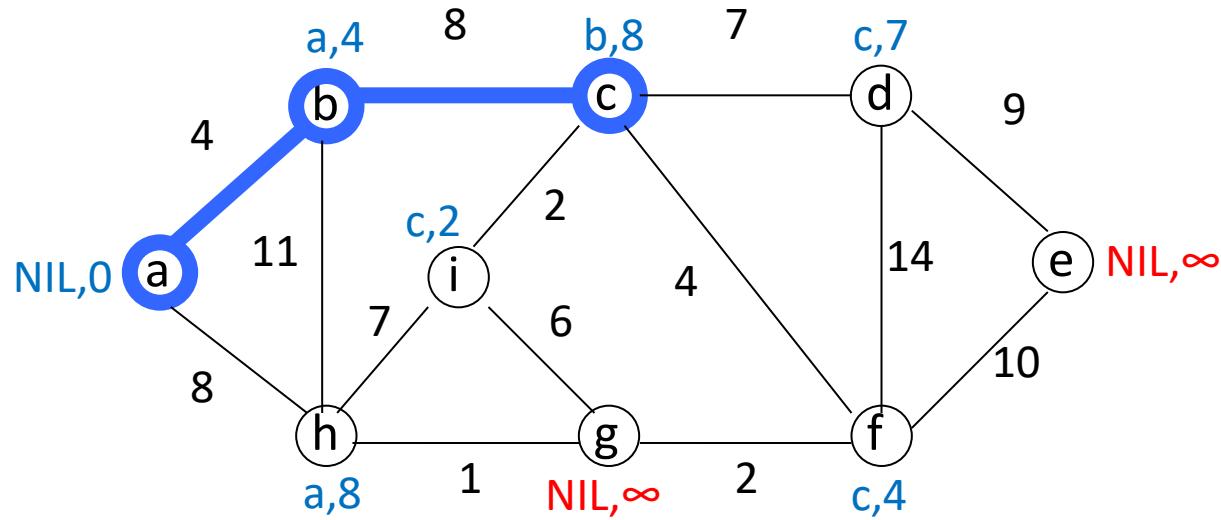
# Prim's Algorithm: Example



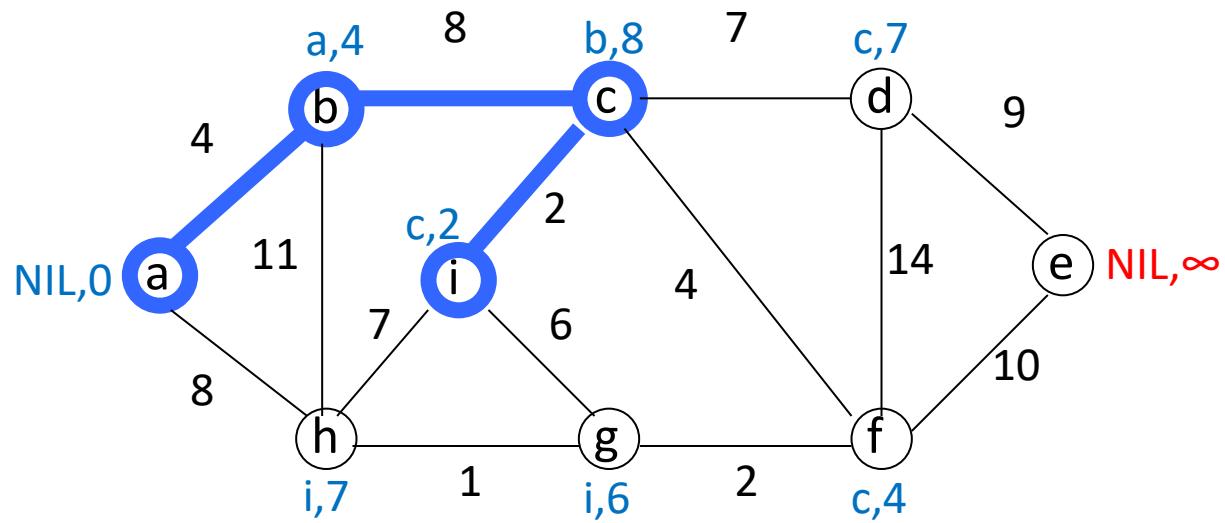
# Prim's Algorithm: Example



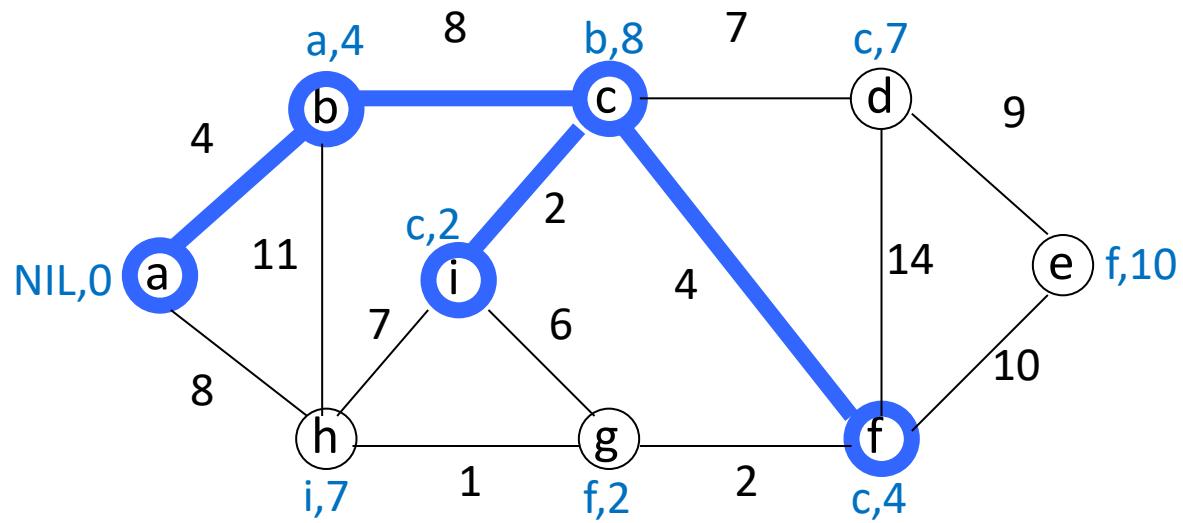
# Prim's Algorithm: Example



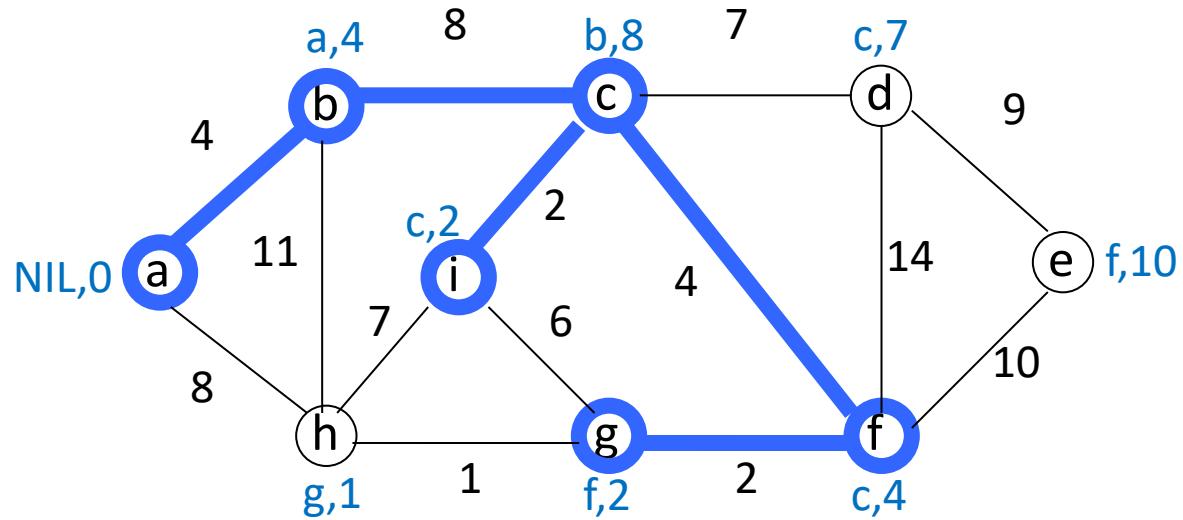
# Prim's Algorithm: Example



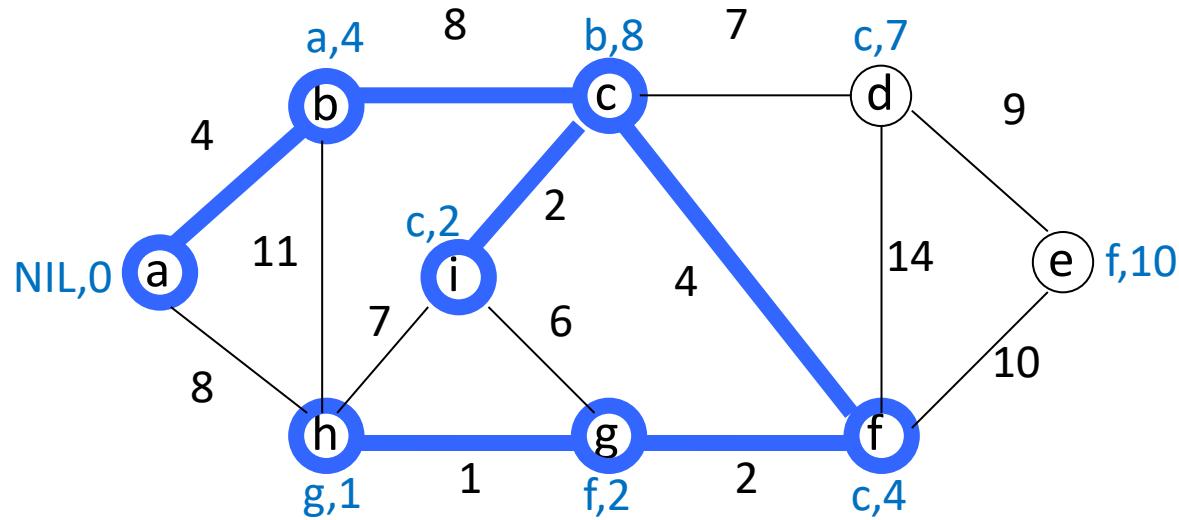
# Prim's Algorithm: Example



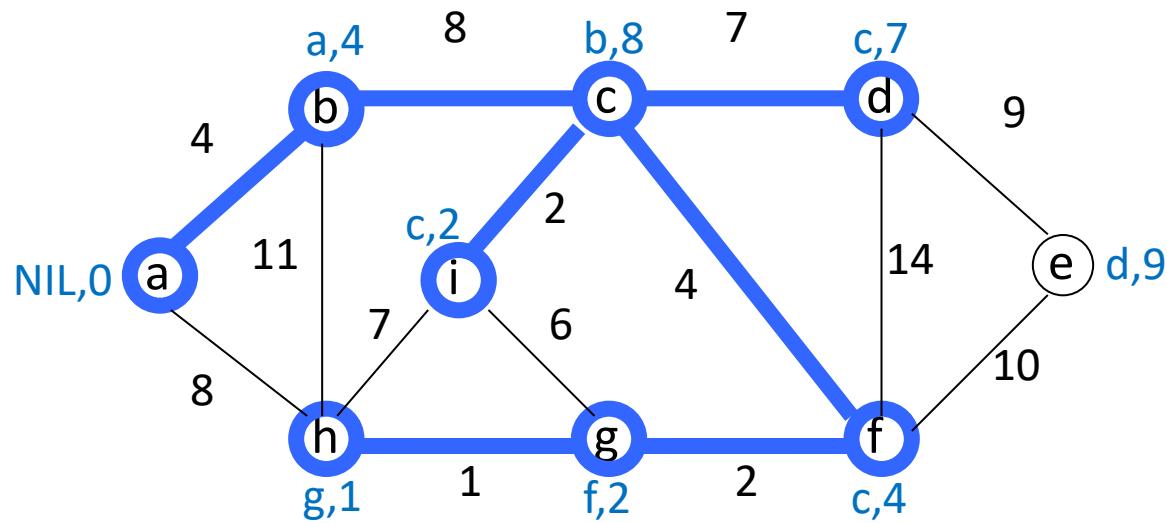
# Prim's Algorithm: Example



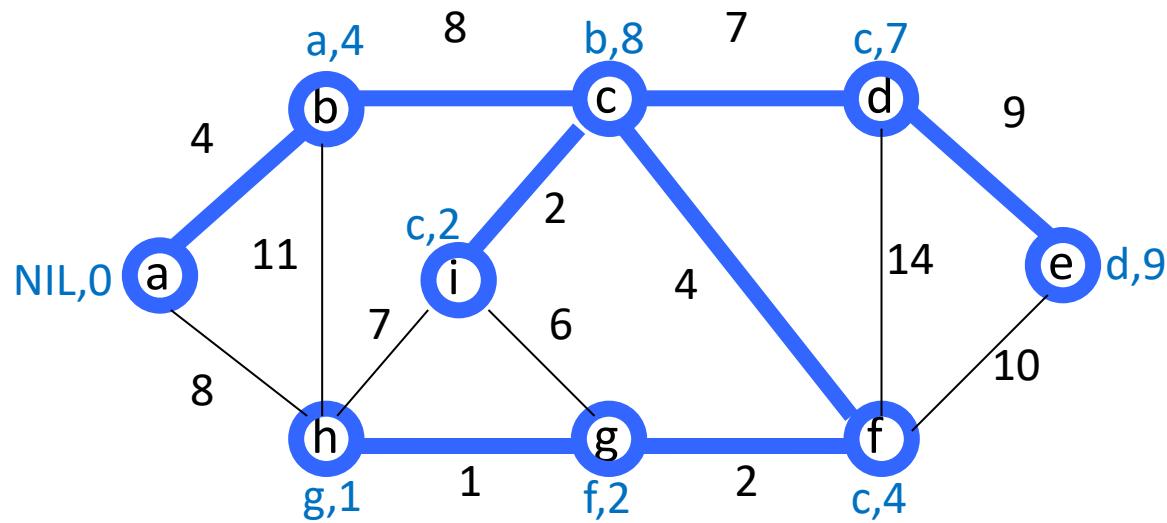
# Prim's Algorithm: Example



# Prim's Algorithm: Example



# Prim's Algorithm: Example



# Prim Algorithm:Variables

r:Grow the minimum spanning tree from the **root vertex “r”**.

Q: is a priority queue, holding all vertices that are **not in the tree** now.

key[v]: is the **minimum weight** of any edge connecting v to a vertex in the tree.

$\pi [v]$ : names the **parent of v** in the tree.

T[v] – Vertex v is **already included** in MST if T[v]==1, otherwise, it is not included yet.

Removing v from set Q adds it to set Q-V of vertices in tree, thus adding (v, p[ v]) to A.

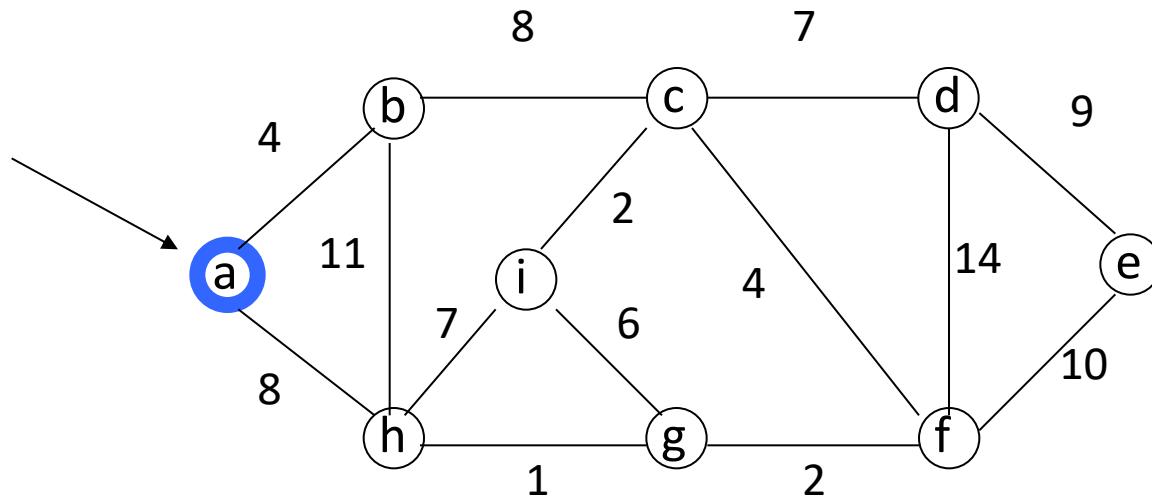
# Prim Algorithm (2)

**MST-Prim**(G, r)

```
01 Q ← V[G]    // Q - vertices out of T
02 for each u ∈ Q
03   key[u] ← ∞
04 key[r] ← 0           // r is the first tree node,
  let r=1
05 π[r] ← NIL
06 while Q ≠ ∅ do
07   u ← ExtractMin(Q)  // making u part of T
08   for each v ∈ Adj[u] do
09     if v ∈ Q and w(u,v) < key[v] then
10       π[v] ← u
11       key[v] ← w(u,v)
```

# The execution of Prim's algorithm

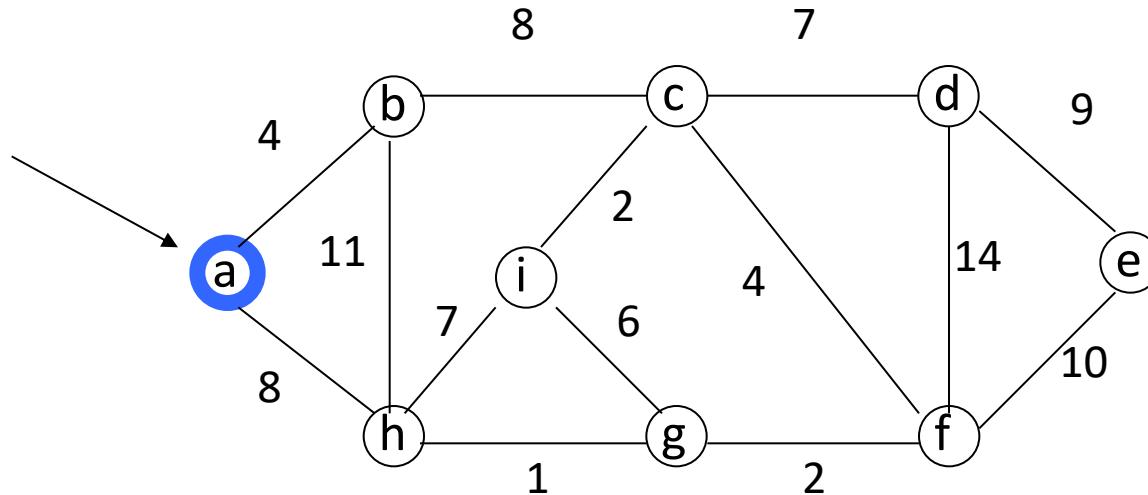
the root vertex



V	a	b	c	d	e	f	g	h	i
T	1	0	0	0	0	0	0	0	0
Key	0	-	-	-	-	-	-	-	-
$\pi$	-1	-	-	-	-	-	-	-	-

# The execution of Prim's algorithm

the root vertex

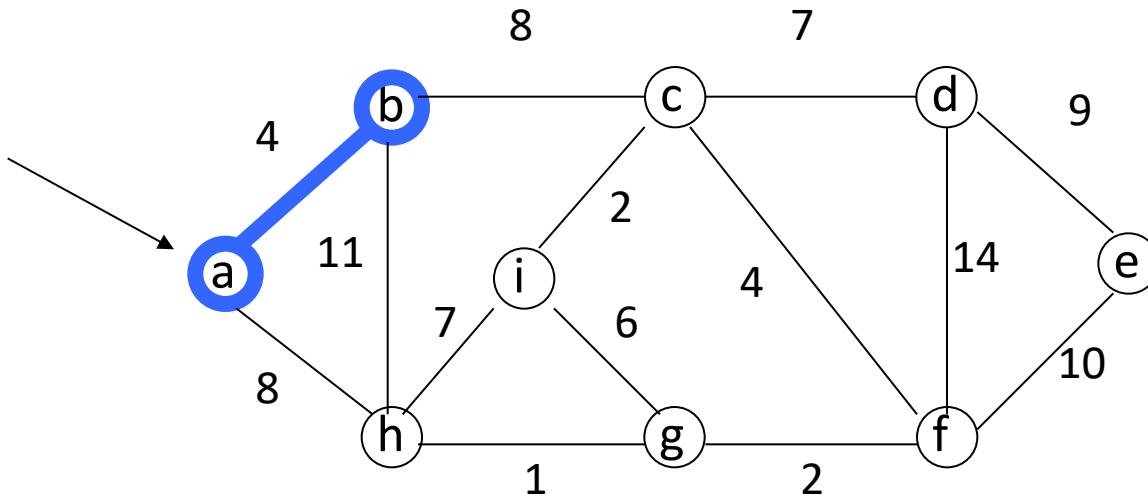


V	a	b	c	d	e	f	g	h	i
T	1	0	0	0	0	0	0	0	0
Key	0	4	-	-	-	-	-	8	-
$\pi$	-1	a	-	-	-	-	-	a	-



# The execution of Prim's algorithm

the root vertex



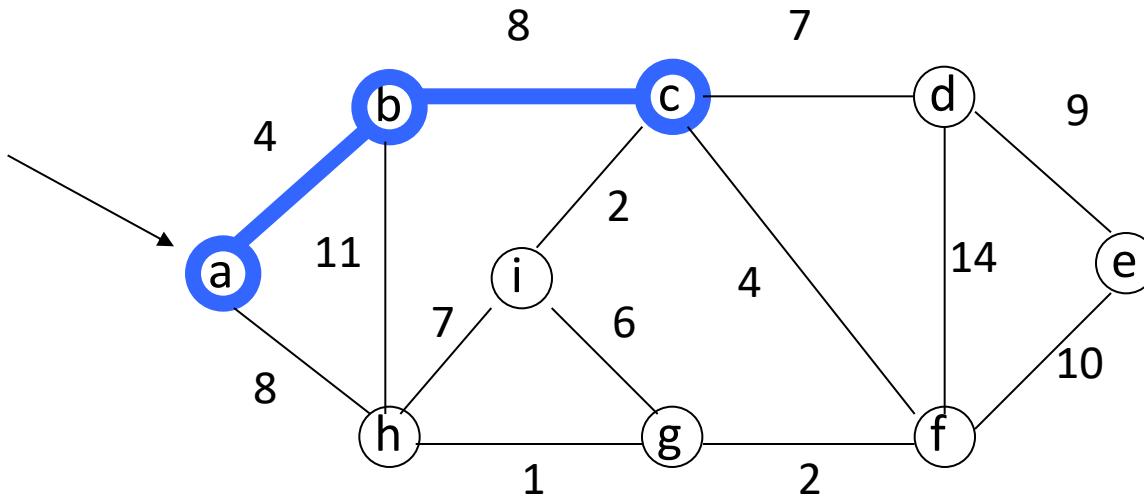
**Important:** Update Key[v] only if  $T[v]==0$

V	a	b	c	d	e	f	g	h	i
T	1	1	0	0	0	0	0	0	0
Key	0	4	8	-	-	-	-	8	-
$\pi$	-1	a	b	-	-	-	-	a	-



# The execution of Prim's algorithm

the root vertex

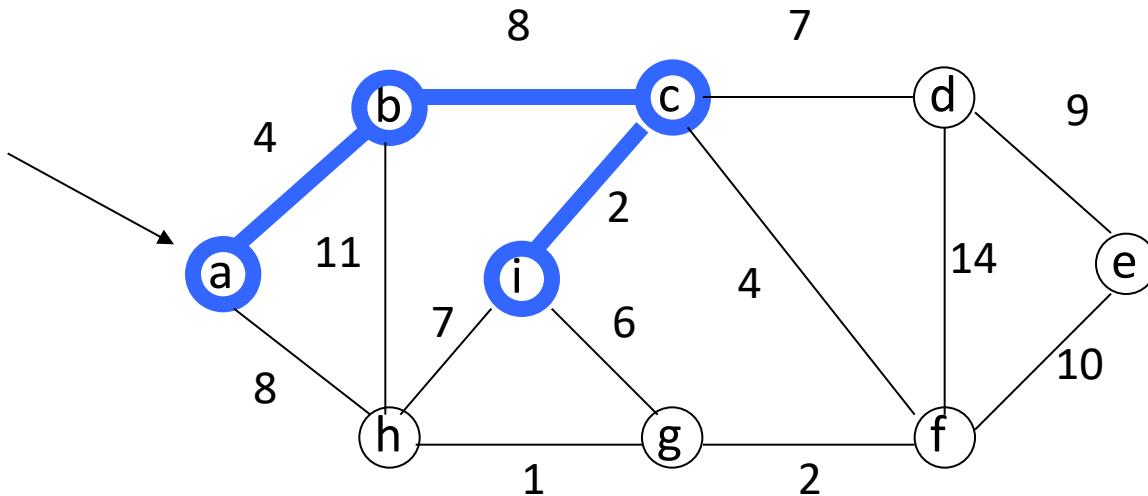


V	a	b	c	d	e	f	g	h	i
T	1	1	1	0	0	0	0	0	0
Key	0	4	8	7	-	4	-	8	2
$\pi$	-1	a	b	c	-	c	-	a	c



# The execution of Prim's algorithm

the root vertex

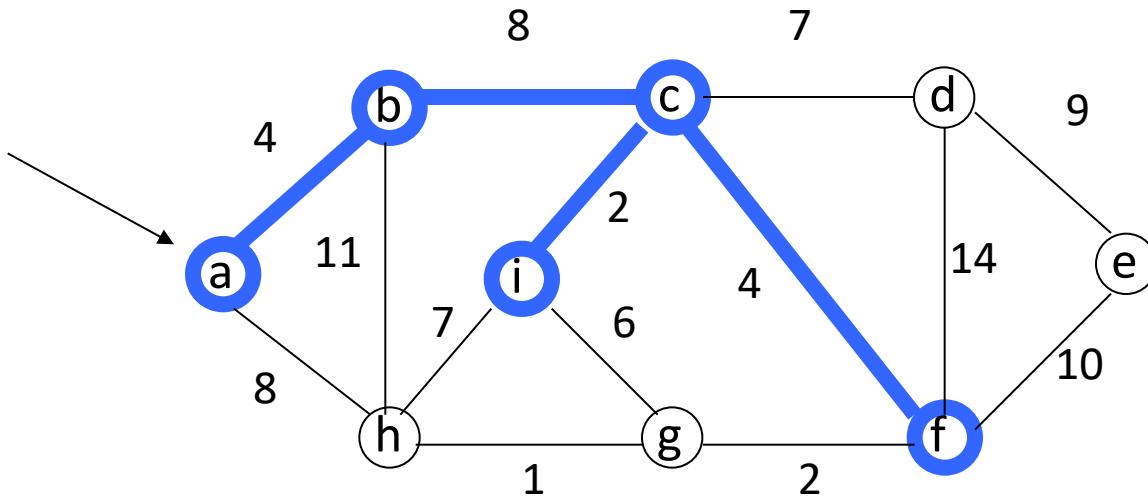


V	a	b	c	d	e	f	g	h	i
T	1	1	1	0	0	0	0	0	1
Key	0	4	8	7	-	4	6	7	2
$\pi$	-1	a	b	c	-	c	i	i	c



# The execution of Prim's algorithm

the root vertex

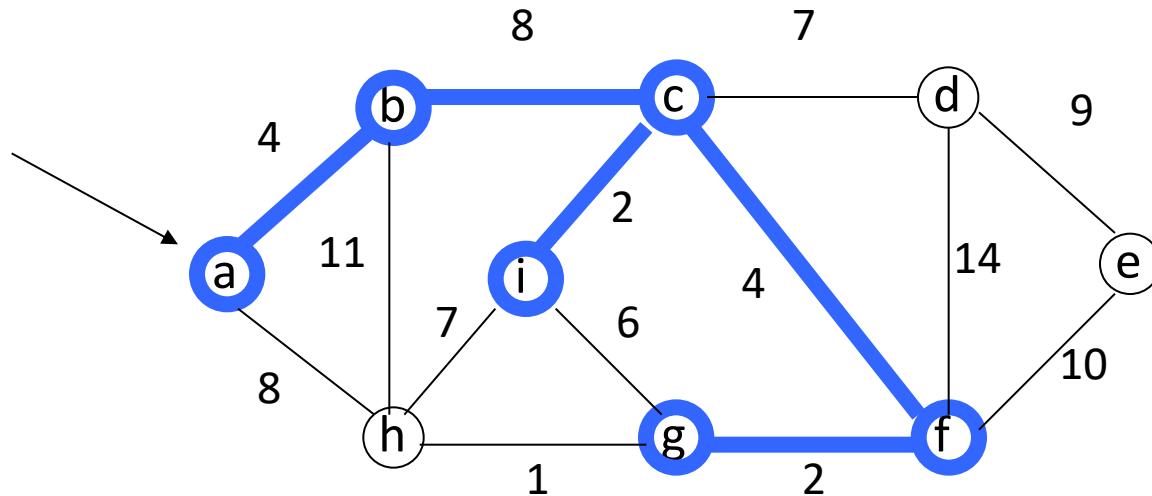


V	a	b	c	d	e	f	g	h	i
T	1	1	1	0	0	1	0	0	1
Key	0	4	8	7	10	4	2	7	2
$\pi$	-1	a	b	c	f	c	f	i	c



# The execution of Prim's algorithm

the root vertex

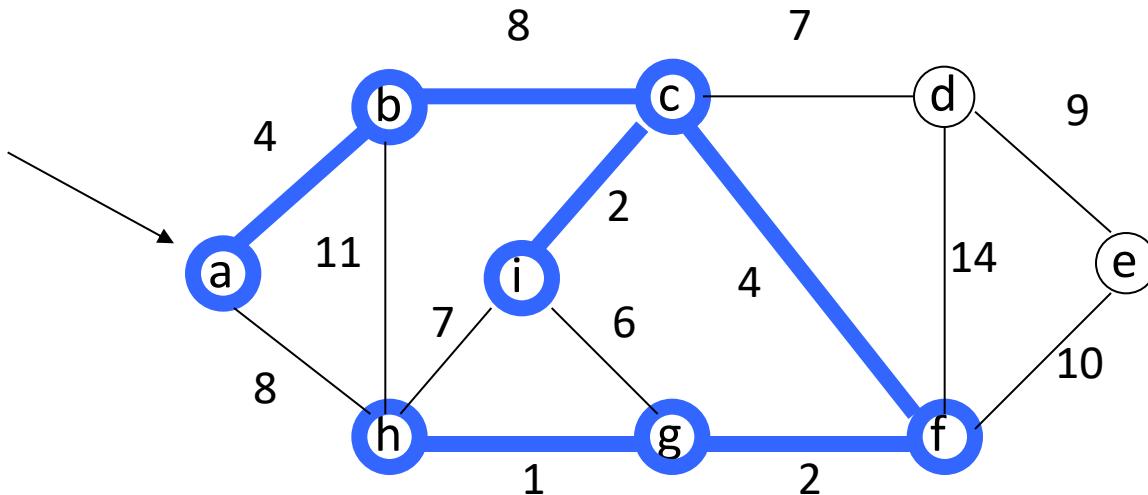


V	a	b	c	d	e	f	g	h	i
T	1	1	1	0	0	1	1	0	1
Key	0	4	8	7	10	4	2	1	2
$\pi$	-1	a	b	c	f	c	f	g	c



# The execution of Prim's algorithm

the root vertex

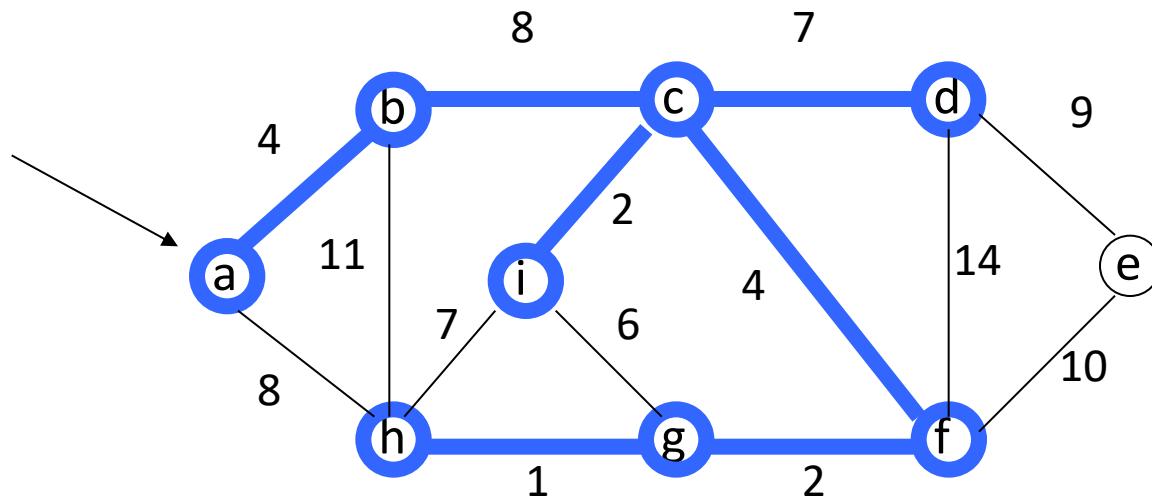


V	a	b	c	d	e	f	g	h	i
T	1	1	1	0	0	1	1	1	1
Key	0	4	8	7	10	4	2	1	2
$\pi$	-1	a	b	c	f	c	f	g	c



# The execution of Prim's algorithm

the root vertex

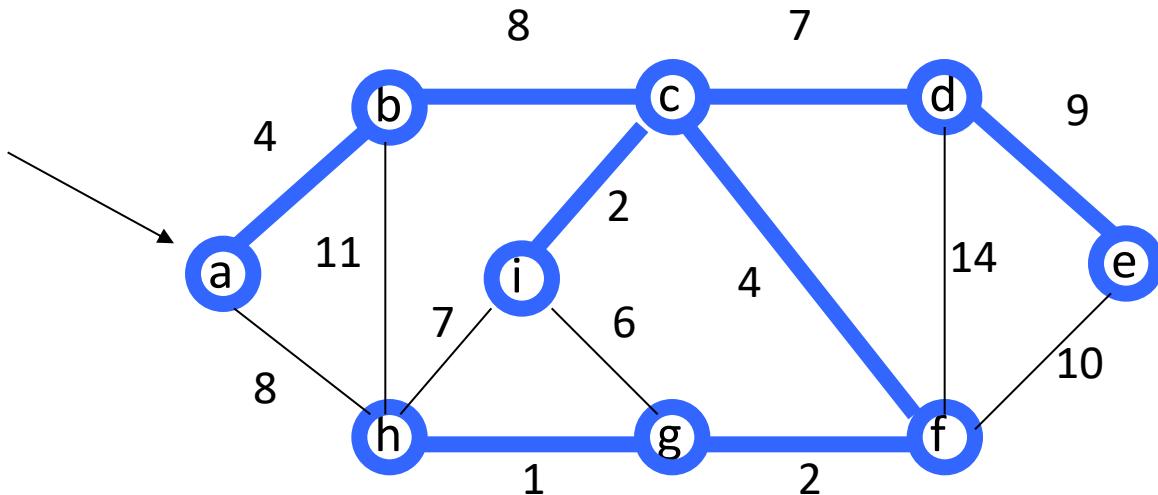


V	a	b	c	d	e	f	g	h	i
T	1	1	1	1	0	1	1	1	1
Key	0	4	8	7	9	4	2	1	2
$\pi$	-1	a	b	c	d	c	f	g	c



# The execution of Prim's algorithm

the root vertex



V	a	b	c	d	e	f	g	h	i
T	1	1	1	1	1	1	1	1	1
Key	0	4	8	7	9	4	2	1	2
$\pi$	-1	a	b	c	d	c	f	g	c

# Complexity: Prim Algorithm

**MST-Prim**( $G, r$ )

```
01  $Q \leftarrow V[G]$  //  $Q$  - vertices out of  $T$ 
02 for each  $u \in Q$   $O(V)$ 
03    $\text{key}[u] \leftarrow \infty$ 
04  $\text{key}[r] \leftarrow 0$ 
05  $\pi[r] \leftarrow \text{NIL}$ 
06 while  $Q \neq \emptyset$  do  $O(V)$ 
07    $u \leftarrow \text{ExtractMin}(Q)$  // making  $u$  part of  $T$  Heap:  $O(\lg V)$ 
08   for each  $v \in \text{Adj}[u]$  do Overall:  $O(E)$ 
09     if  $v \in Q$  and  $w(u, v) < \text{key}[v]$  then
10        $\pi[v] \leftarrow u$ 
11        $\text{key}[v] \leftarrow w(u, v)$  Decrease Key:  $O(\lg V)$ 
```

Overall complexity:  $O(V) + O(V \lg V + E \lg V) = O(E \lg V)$

# Summary

## Kruskal's algorithm

1. Select the shortest edge in a network
2. Select the next shortest edge which does not create a cycle
3. Repeat step 2 until all vertices have been connected

## Prim's algorithm

1. Select any vertex
2. Select the shortest edge connected to that vertex
3. Select the shortest edge connected to any vertex already connected
4. Repeat step 3 until all vertices have been connected

# Summary

- ↗ Both are greedy algorithms
- ↗ Both have the same output → MST
- ↗ Kruskal's begins with forest and merge into a tree
- ↗ Prim's always stays as a tree



# Books

- *Introduction to Algorithms, Thomas H. Cormen, Charle E. Leiserson, Ronald L. Rivest, Clifford Stein (CLRS).*
- *Fundamental of Computer Algorithms, Ellis Horowitz, Sartaj Sahni, Sanguthevar Rajasekaran (HSR)*



# References

<https://www.geeksforgeeks.org/kruskals-minimum-spanning-tree-algorithm-greedy-algo-2/>

<https://www.geeksforgeeks.org/applications-of-minimum-spanning-tree/>

[https://www.tutorialspoint.com/data\\_structures\\_algorithms/prims\\_spanning\\_tree\\_algorithm.htm](https://www.tutorialspoint.com/data_structures_algorithms/prims_spanning_tree_algorithm.htm)

[https://www.tutorialspoint.com/data\\_structures\\_algorithms/kruskals\\_spanning\\_tree\\_algorithm.htm](https://www.tutorialspoint.com/data_structures_algorithms/kruskals_spanning_tree_algorithm.htm)

<https://www.cs.usfca.edu/~galles/visualization/Prim.html>

<https://visualgo.net/en/mst?slide=1>

# Lecture Title: Shortest Path Algorithms



Dept. of Computer Science  
Faculty of Science and Technology

Lecture No:	12	Week No:	12	Semester:	
Lecturer:	Name & email: Md. Manzurul Hasan, <a href="mailto:manzurul@aiub.edu">manzurul@aiub.edu</a>				

# **CSC2211: Algorithms**

## **Shortest Path**

**Md. Manzurul Hasan**

**[manzurul@aiub.edu](mailto:manzurul@aiub.edu)**

**American International University Bangladesh**

# Lecture Outline



1. Shortest Path
2. Dijkstra's Algorithm
3. Bellman-Ford' Algorithms (Optional)
4. Floyd-Warshall's Algorithm

# Shortest Path

- # Generalize distance to weighted setting
- # Digraph  $G = (V, E)$  with weight function  $w : E \rightarrow R$  (assigning real values to edges)
- # Weight of path  $p = v_1 \rightarrow v_2 \rightarrow \dots \rightarrow v_k$  is
$$w(p) = \sum_{i=1}^{k-1} w(v_i, v_{i+1})$$
- # **Shortest path** = a path of **minimum weight** (cost)
- # Applications
  - # static/dynamic network routing
  - # robot motion planning
  - # map/route generation in traffic

# Shortest-Path Problems

## # Shortest-Path problems

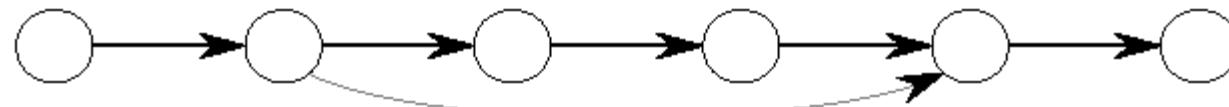
- **Single-source (single-destination).** Find a shortest path from a given source (vertex  $s$ ) to each of the vertices.
- **Single-pair.** Given two vertices, find a shortest path between them. Solution to single-source problem solves this problem efficiently, too.
- **All-pairs.** Find shortest-paths for every pair of vertices. Dynamic programming algorithm.
- Unweighted shortest-paths – **BFS**.

# Optimal Substructure

# *Theorem:* subpaths of shortest paths are shortest paths

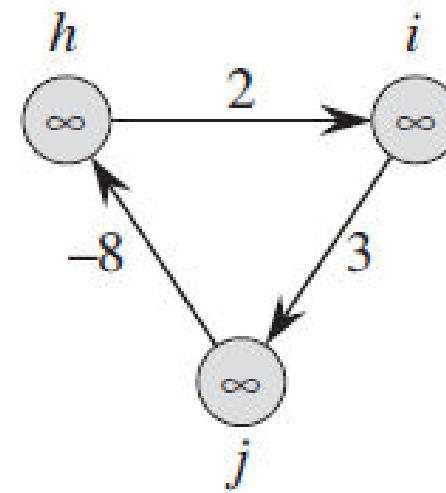
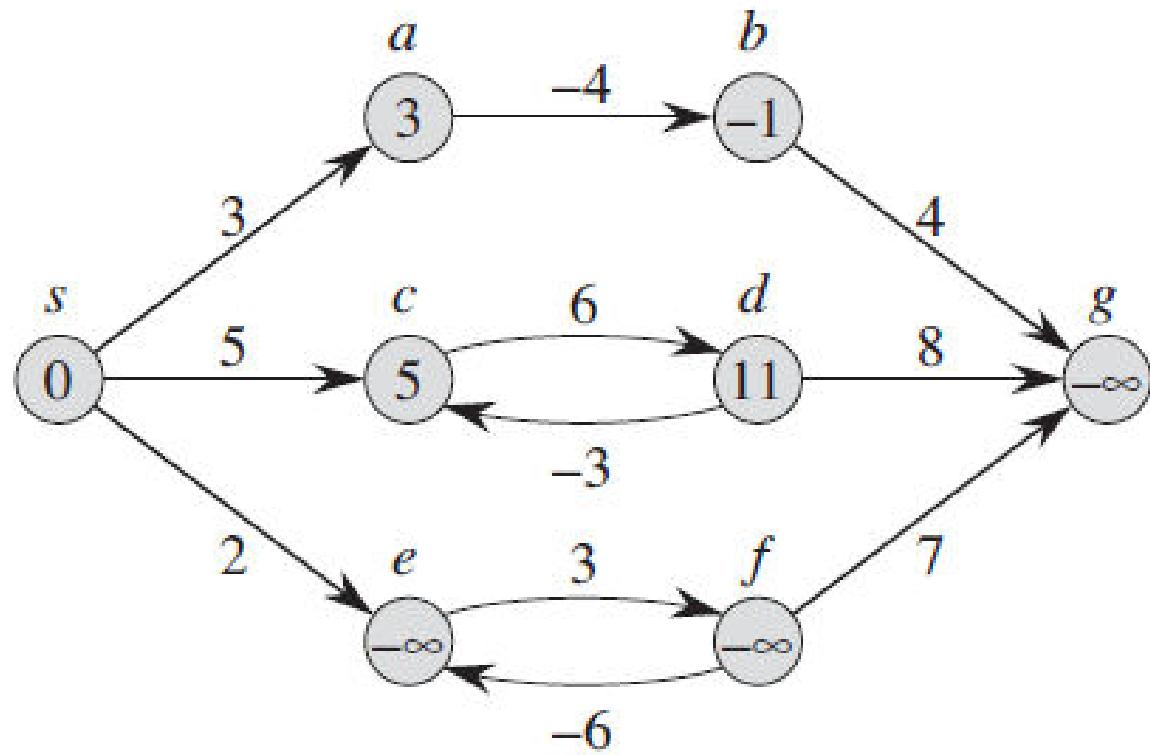
# Proof:

- if some subpath were not the shortest path, one could substitute the shorter subpath and create a shorter total path



# Negative Weights and Cycles

- # Negative edges are OK, as long as there are no *negative weight cycles* (otherwise paths with arbitrary small “lengths” would be possible).
- # Shortest-paths can have no cycles (otherwise we could improve them by removing cycles).
  - # Any shortest-path in graph  $G$  can be no longer than  $n-1$  edges, where  $n$  is the number of vertices

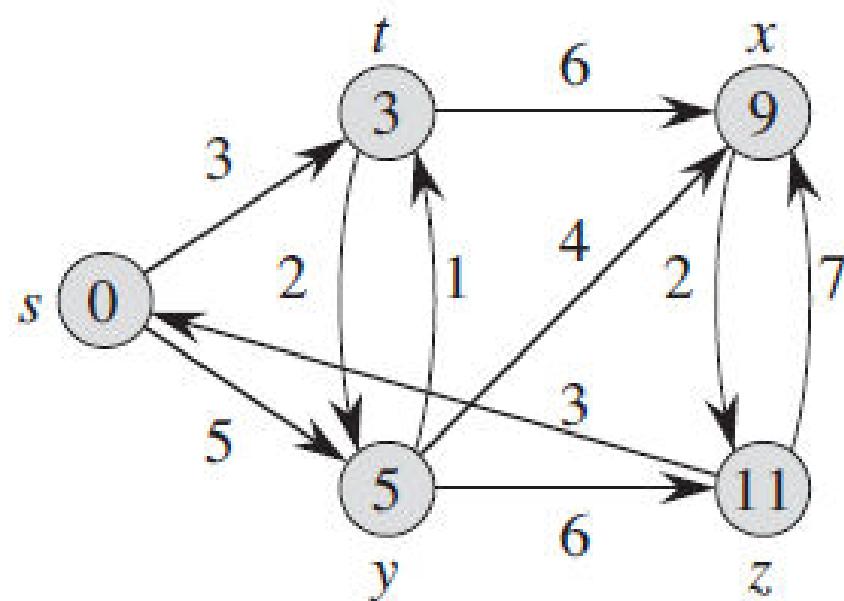


Shortest Path → 8

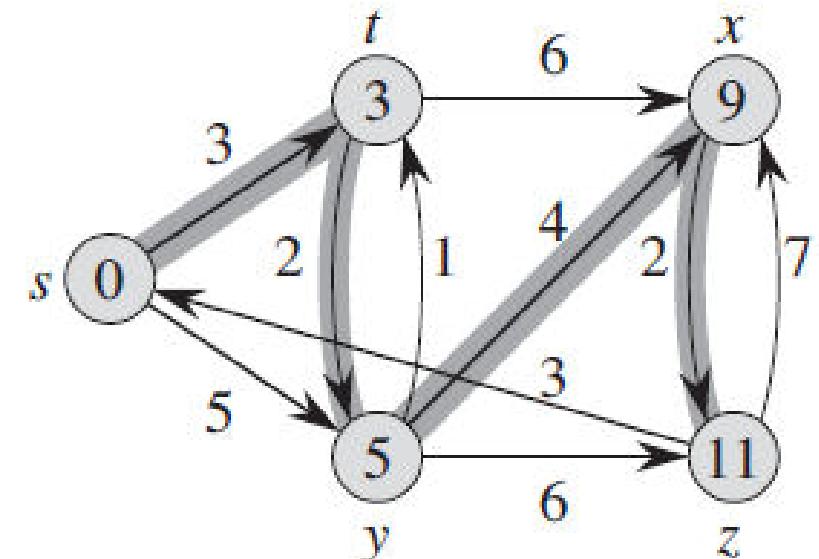
# Shortest Path Tree

- # The result of the algorithms is a *shortest path tree*. For each vertex  $v$ , it
  - records a shortest path from the start vertex  $s$  to  $v$ .
  - $v.\text{pred}$  is the predecessor of  $v$  in this shortest path
  - $v.\text{dist}$  is the shortest path length from  $s$  to  $v$

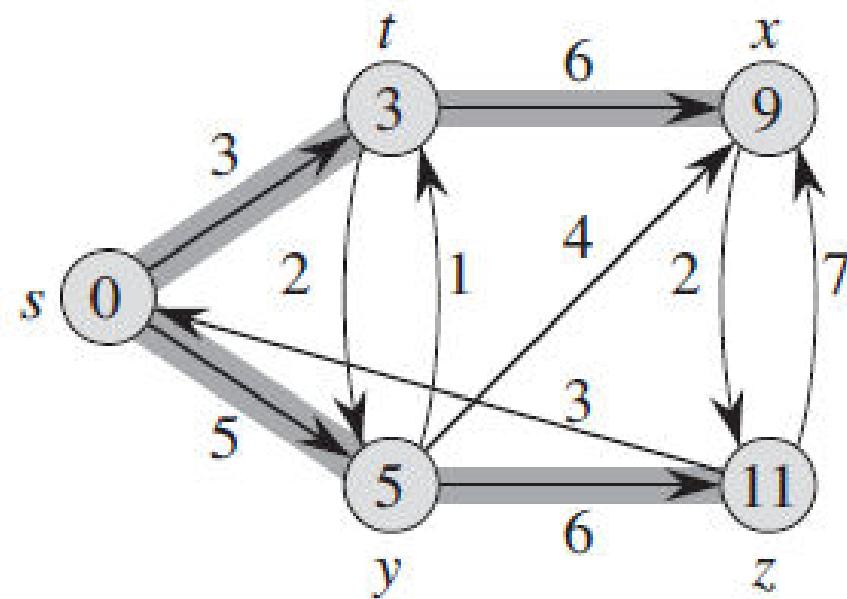
For each vertex  $v$  in the graph, we maintain  $v.\text{dist}$ , the estimate of the shortest path from  $s$ . It is initialized to  $\infty$  at the start.



(a)



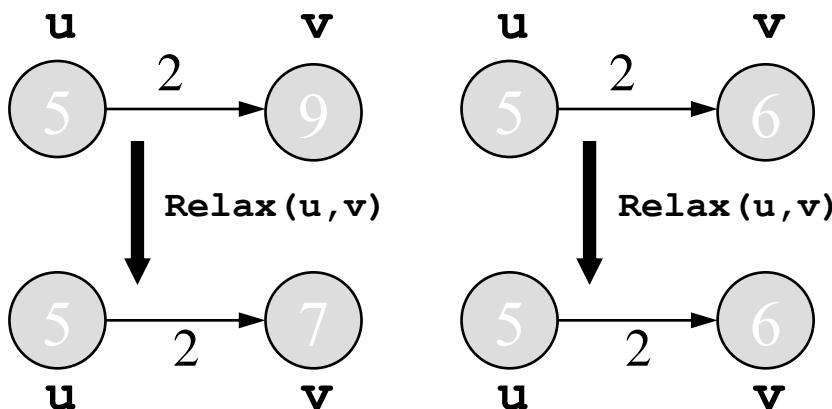
(c)



(b)

# Relaxation

- # Relaxing an edge  $(u, v)$  means testing whether we can improve the shortest path to  $v$  found so far by going through  $u$ .



```
Relax (u, v, G)
if v.dist > u.dist + w(u, v) then
    v.dist := u.dist + w(u, v)
    v.pred := u
```

# Dijkstra's Algorithm

- # Non-negative edge weights
- # Greedy, similar to Prim's algorithm for MST
- # Use  $Q$ , a priority queue with keys  $v.\text{dist}$ , which is re-organized whenever some  $\text{dist}$  decreases
- # Basic idea
  - ▣ maintain a set  $s$  of solved vertices
  - ▣ at each step select "closest" vertex  $u$ , add it to  $s$ , and relax all edges from  $u$

# Dijkstra's Pseudo Code

# Input: Graph  $G$ , start vertex  $s$

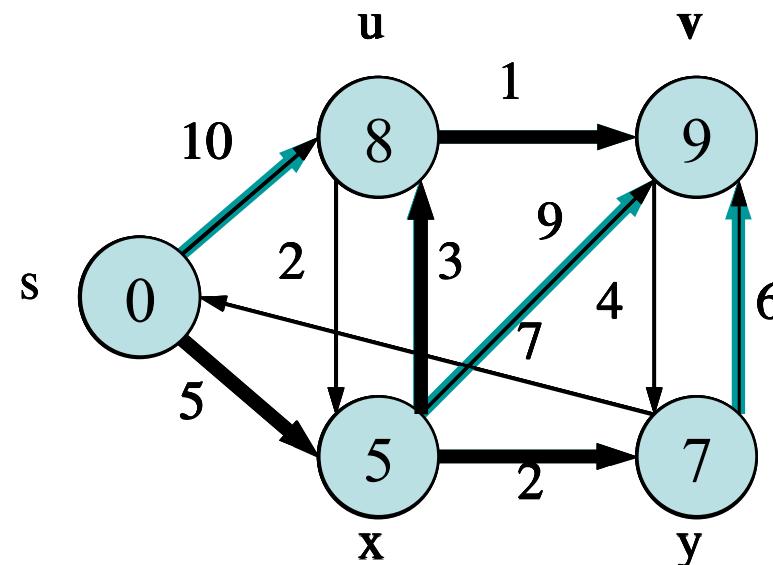
**Dijkstra**( $G, s$ )

```
01 for each vertex  $u \in G.V$ 
02      $u.dist := \infty$ 
03      $u.pred := \text{NIL}$ 
04  $s.dist := 0$ 
05  $S := \emptyset$           //  $S$  is used to explain the algorithm
06 init( $Q, G.V$ )    //  $Q$  is a priority queue
07 while not isEmpty( $Q$ )
08      $u := \text{extractMin}(Q)$ 
09      $S := S \cup \{u\}$ 
10     for each  $v \in u.adj$  do
11         Relax( $u, v, G$ )
12         modifyKey( $Q, v$ )
```

relaxing  
edges

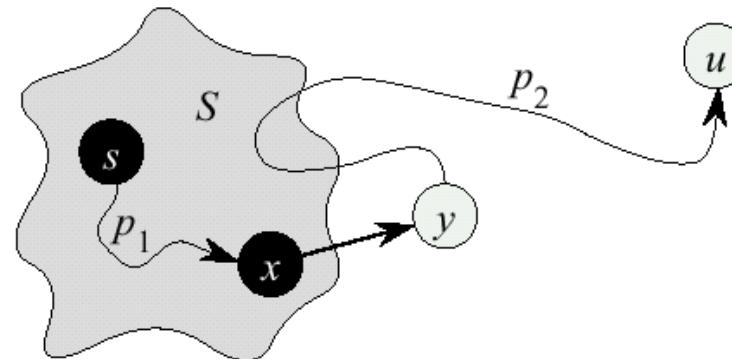
# Dijkstra's Example

```
Dijkstra(G, s)
01 for each vertex u ∈ G.V
02     u.dist := ∞
03     u.pred := NIL
04 s.dist := 0
05 S := ∅
06 init(Q, G.V)
07 while not isEmpty(Q)
08     u := extractMin(Q)
09     S := S ∪ {u}
10    for each v ∈ u.adj do
11        Relax(u, v, G)
12        modifyKey(Q, v)
```



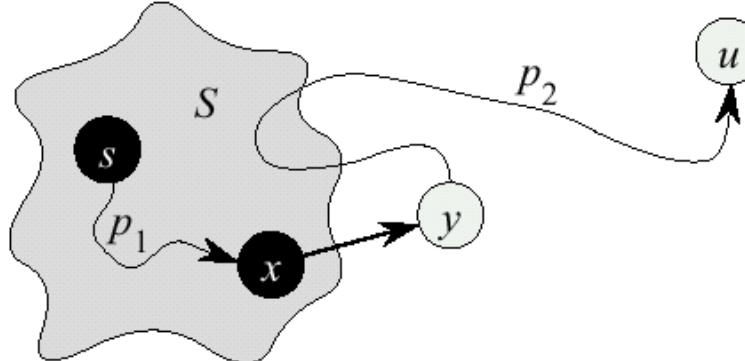
# Dijkstra's Correctness

- # We prove that whenever  $u$  is added to  $S$ ,  $u.\text{dist} = \delta(s, u)$ , i.e.,  $\text{dist}$  is minimum, and that equality is maintained thereafter.
- # Proof (by contradiction)
  - ⊕ Initially  $\forall v: v.\text{dist} \geq \delta(s, v)$
  - ⊕ Let  $u$  be the **first** vertex such that there is a shorter path than  $u.\text{dist}$ , i.e.,  $u.\text{dist} > \delta(s, u)$
  - ⊕ We will show that this assumption leads to a contradiction



# ...Dijkstra Correctness

- # Let  $y$  be the first vertex  $\in V-S$  on the actual shortest path from  $s$  to  $u$ , then it must be that  $y.dist = \delta(s, y)$  because
  - ▣  $x.dist$  is set correctly for  $y$ 's predecessor  $x \in S$  on the shortest path (by choice of  $u$  as the first vertex for which  $dist$  is set incorrectly)
  - ▣ when the algorithm inserted  $x$  into  $S$ , it relaxed the edge  $(x, y)$  , setting  $y.dist$  to the correct value



# ...Dijkstra Correctness

$$u.\text{dist} > \delta(s, u)$$

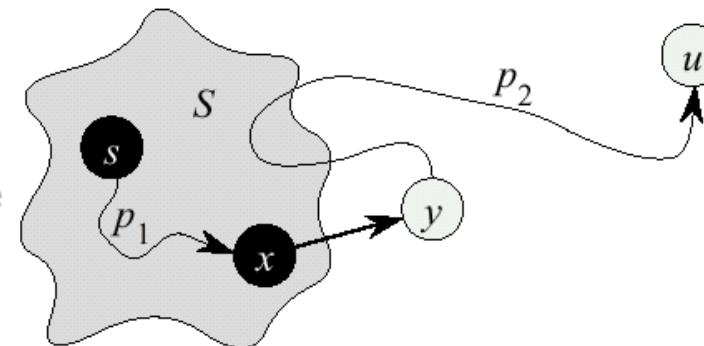
initial assumption

$$= \delta(s, y) + \delta(y, u) \text{ optimal substructure}$$

$$= y.\text{dist} + \delta(y, u) \text{ correctness of } y.\text{dist}$$

$$\geq y.\text{dist}$$

no negative weights



- # But  $u.\text{dist} > y.\text{dist} \Rightarrow$  algorithm would have chosen  $y$  (from the PQ) to process next, not  $u$   
 $\Rightarrow$  contradiction
- # Thus,  $u.\text{dist} = \delta(s, u)$  at time of insertion of  $u$  into  $S$ , and Dijkstra's algorithm is correct

# Dijkstra's Running Time

- # Extract-Min executed  $|V|$  time
- # Decrease-Key executed  $|E|$  time
- # Time =  $|V| T_{\text{Extract-Min}} + |E| T_{\text{Decrease-Key}}$
- #  $T$  depends on different  $Q$  implementations

Q	T(Extract-Min)	T(Decrease-Key)	Total
array	$\mathcal{O}(V)$	$\mathcal{O}(1)$	$\mathcal{O}(V^2)$
binary heap	$\mathcal{O}(\lg V)$	$\mathcal{O}(\lg V)$	$\mathcal{O}(E \lg V)$

# Bellman-Ford Algorithm

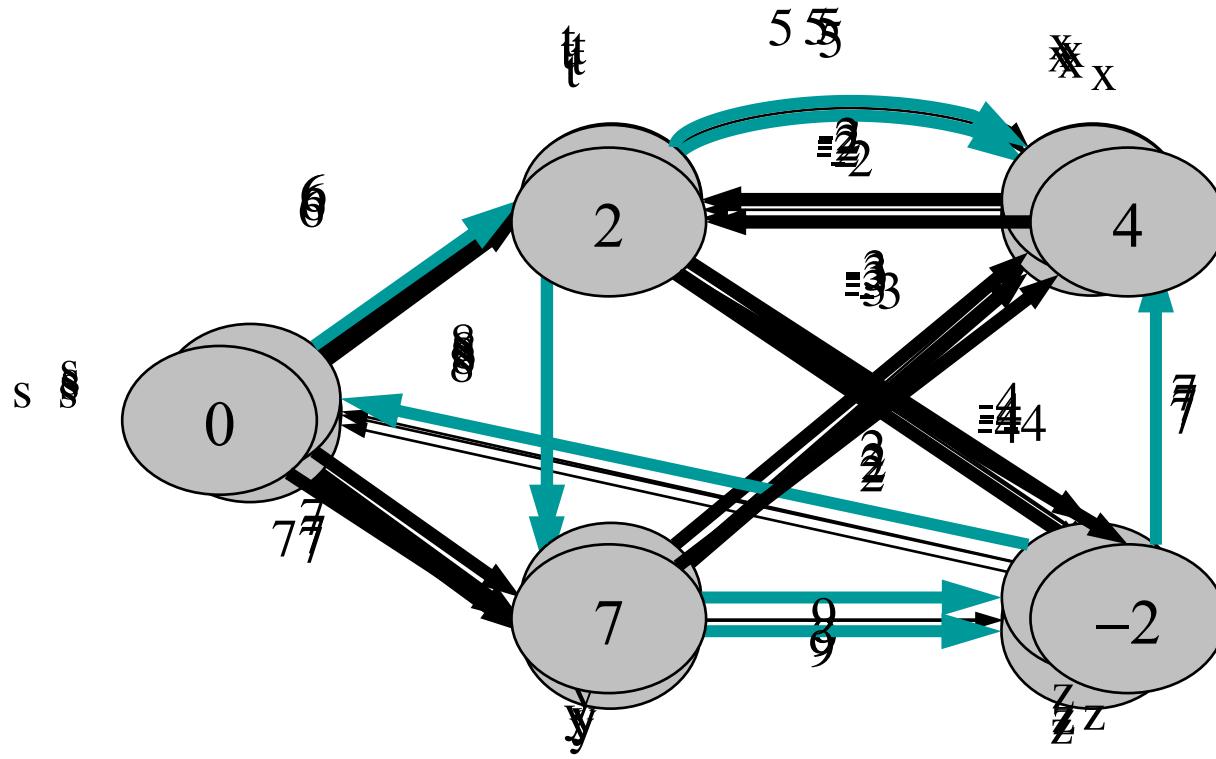
- # Dijkstra's doesn't work when there are ***negative edges***:
  - ▣ Intuition – we cannot be greedy anymore on the assumption that the lengths of paths will only increase in the future
- # Bellman-Ford algorithm detects negative cycles (returns *false*) or returns the shortest path-tree

# Bellman-Ford Algorithm

**Bellman-Ford**(G, s)

```
01 for each vertex u ∈ G.V
02     u.dist := ∞
03     u.pred := NIL
04 s.dist := 0
05 for i := 1 to |G.V()|-1 do
06     for each edge (u,v) ∈ G.E do
07         Relax (u,v,G)
08 for each edge (u,v) ∈ G.E do
09     if v.dist > u.dist + w(u,v) then
10         return false
11 return true
```

# Bellman-Ford Example



# Bellman-Ford running time:

$$\blacksquare (|V|-1)|E| + |E| = Q(VE)$$

# Correctness of Bellman-Ford

- # Let  $\delta_i(s, u)$  denote the length of path from  $s$  to  $u$ , that is shortest among all paths, that contain at most  $i$  edges
- # Prove by induction that  $u.\text{dist} = \delta_i(s, u)$  after the  $i^{th}$  iteration of Bellman-Ford
  - # Base case ( $i=0$ ) trivial
  - # Inductive step (say  $u.\text{dist} = \delta_{i-1}(s, u)$ ):
    - ◆ Either  $\delta_i(s, u) = \delta_{i-1}(s, u)$
    - ◆ Or  $\delta_i(s, u) = \delta_{i-1}(s, z) + w(z, u)$
    - ◆ In an iteration we try to relax each edge  $(z, u)$  also, so we handle both cases, thus  $u.\text{dist} = \delta_i(s, u)$

# Correctness of Bellman-Ford

- # After  $n-1$  iterations,  $u.\text{dist} = \delta_{n-1}(s, u)$ , for each vertex  $u$ .
- # If there is still some edge to relax in the graph, then there is a vertex  $u$ , such that  $\delta_n(s, u) < \delta_{n-1}(s, u)$ . But there are only  $n$  vertices in  $G$  – we have a cycle, and it is negative.
- # Otherwise,  $u.\text{dist} = \delta_{n-1}(s, u) = \delta(s, u)$ , for all  $u$ , since any shortest path will have at most  $n-1$  edges

# Shortest-Path in DAG's

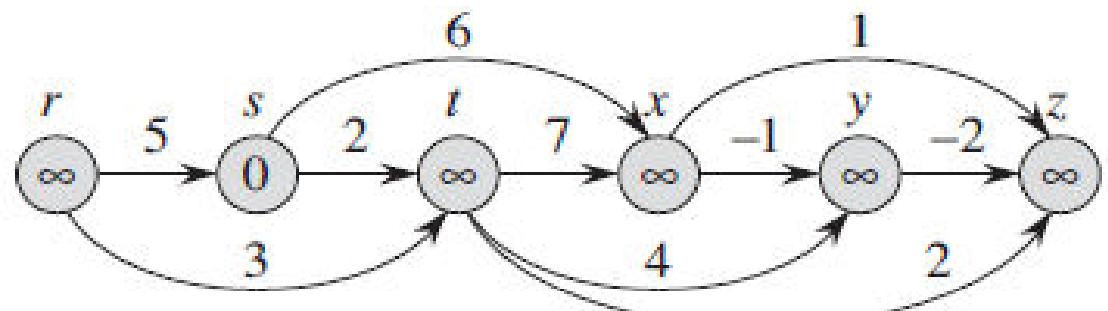
- # Finding shortest paths in DAG's is much easier, because it is easy to find an order in which to do relaxations – Topological sorting!

**DAG-Shortest-Paths** ( $G, w, s$ )

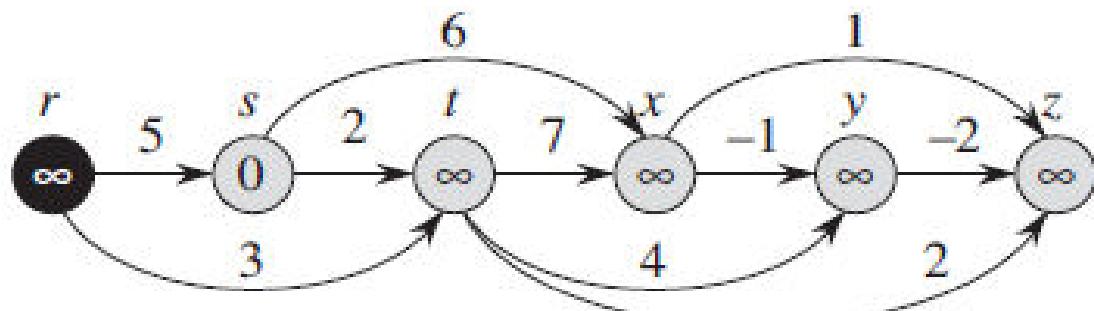
```
01 01 for each vertex  $u \in G.V$ 
02      $u.dist := \infty$ 
03      $u.pred := NIL$ 
04  $s.dist := 0$ 
05 topologically sort  $G$ 
06 for each vertex  $u$  taken in topological order do
07     for each  $v \in u.adj$  do
08         Relax( $u, v, G$ )
```

- # Running time:

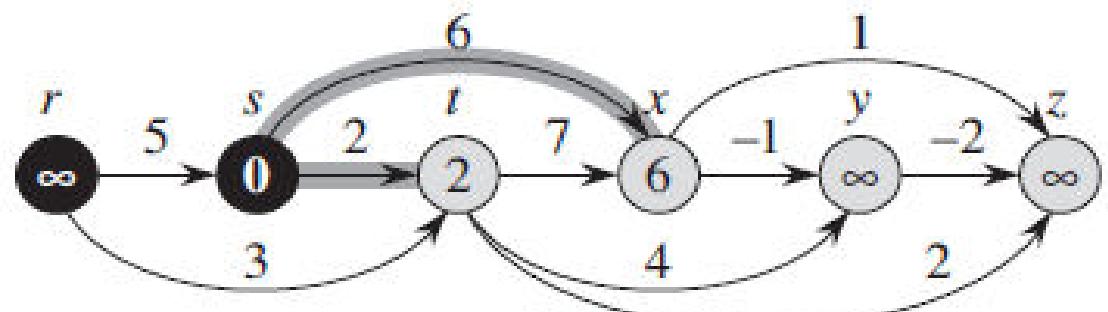
- ▣  $\Theta(V+E)$  – only one relaxation for each edge,  $V$  times faster than Bellman-Ford



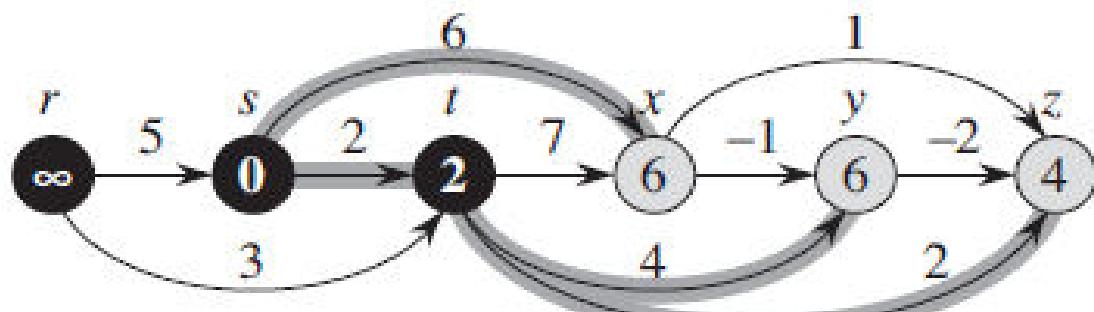
(a)



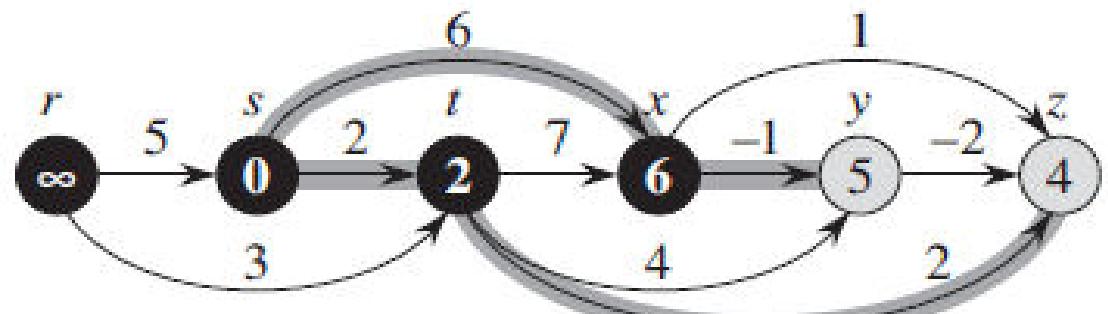
(b)



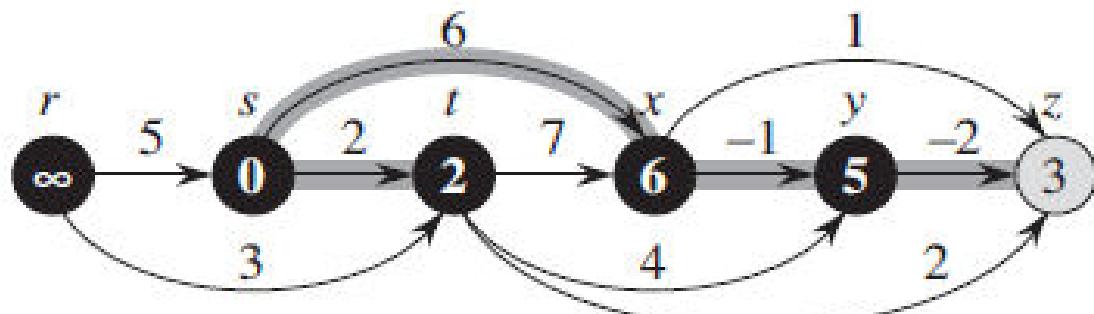
(c)



(d)



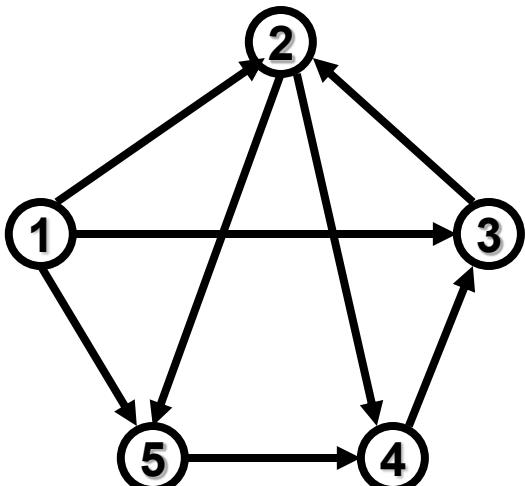
(e)



(f)

# All Pair Shortest Path

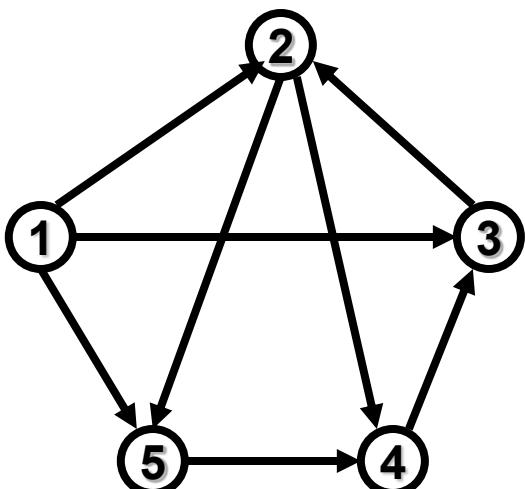
- #  $G(V, E)$  is a directed graph in its **adjacency matrix** representation.
- # The adjacency matrix representation is of size  $|V| \times |V|$



$$A = \begin{matrix} 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \end{matrix}$$
$$a_{ij} = \begin{cases} 1 & \text{if } (i, j) \in E \\ 0 & \text{otherwise} \end{cases}$$

# All Pair Shortest Path

# The *Transitive Closure* of  $G(V, E)$  is



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

$$A^* = \begin{matrix} 0 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 & 1 \end{matrix}$$

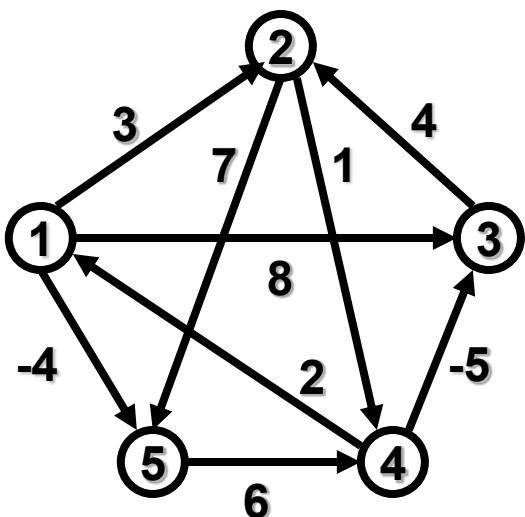
$$\begin{matrix} 0 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 & 1 \end{matrix}$$

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

$$a_{ij}^* = \begin{cases} 1 & \text{if there is a path from } i \text{ to } j \\ 0 & \text{otherwise} \end{cases}$$

# All Pair Shortest Path

#  $G(V, E)$  is a directed graph in its adjacency matrix representation (*initially*)

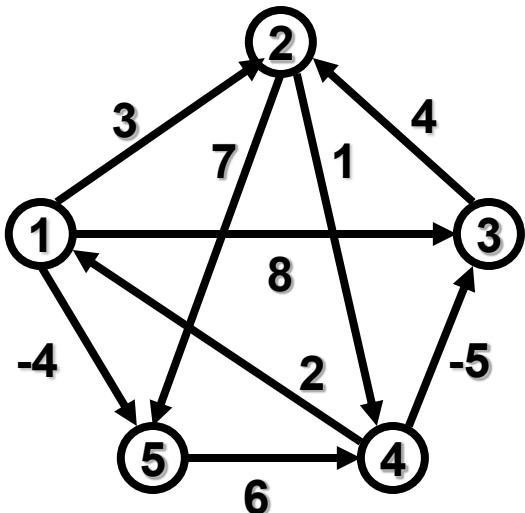


$$W = \begin{matrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{matrix}$$

$$w_{ij} = \begin{cases} \text{Cost}(i, j) & \text{if } (i, j) \in E \\ +\infty & \text{otherwise} \end{cases}$$

# All Pair Shortest Path

- # Assumption: the graph has **NO NEGATIVE CYCLES**, otherwise the shortest distance is not well defined.
- # The Paths Distance Matrix of  $G(V, E)$  is



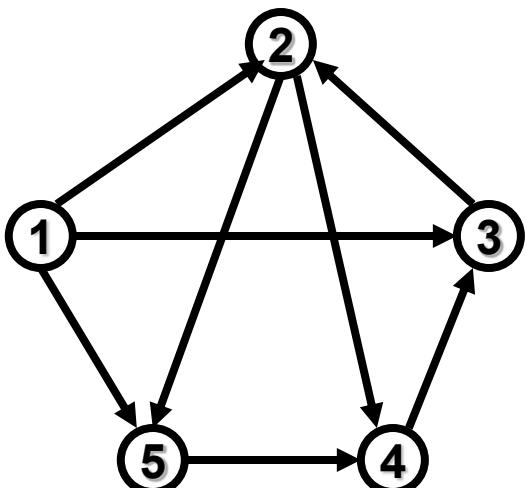
$$D^* = \begin{matrix} 0 & 1 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{matrix}$$

$d^*_{ij}$  = Cost of Shortest path from  $i$  to  $j$

# All Pair Shortest Path

# Towards a *transitive closure* algorithm:

# Insert 1's on the diagonal of **A**



$$A = \begin{matrix} 1 & 1 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 1 \\ 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 \end{matrix}$$

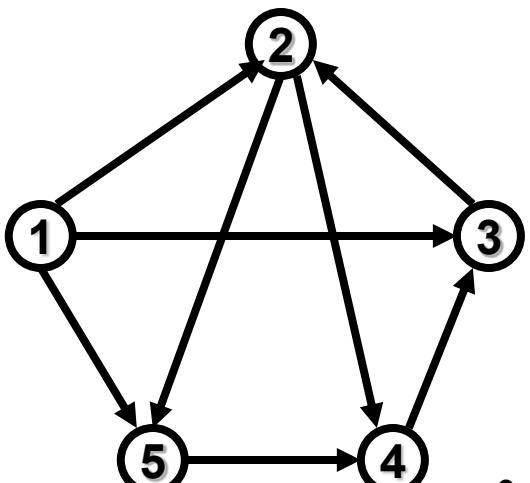
$$a_{ij} = \begin{cases} 1 & \text{if } (i, j) \in E \\ 1 & \text{if } i = j \\ 0 & \text{otherwise} \end{cases}$$

# All Pair Shortest Path

# Transitive closure and matrix multiplication:

# What is  $A^2 = A \times A$  ?

$$a_{ij}^2 = V_{x=1}^n a_{ix} a_{xj}$$



$$A = \begin{matrix} 1 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 & 1 \end{matrix}$$

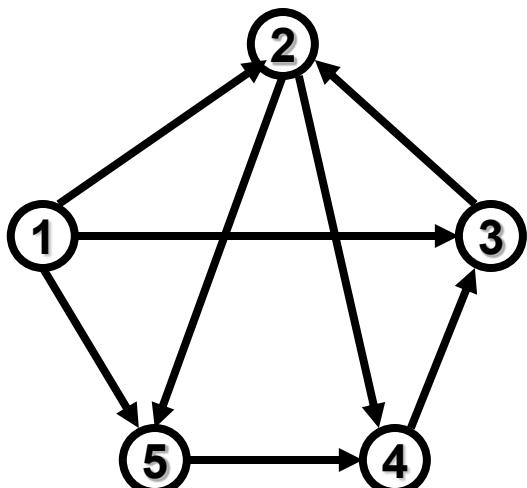
$$a_{ij}^2 = \begin{cases} 1 & \text{if } (i, j) \in E \\ 1 & \text{if } \exists x \{ (i, x) \in E \text{ and } (x, j) \in E \} \\ 0 & \text{otherwise} \end{cases}$$

# All Pair Shortest Path

# Transitive closure and matrix multiplication:

# What is  $A^3 = A^2 \times A$  ?

$$a_{ij}^3 = V_{x=1}^n a_{ix}^2 a_{xj}$$



$$A = \begin{matrix} 1 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 & 1 \end{matrix}$$

$$a_{ij}^3 = \begin{cases} 1 & \text{if there is path of length} \\ & \text{at most 3 from } i \text{ to } j \\ 0 & \text{otherwise} \end{cases}$$

# All Pair Shortest Path

# Transitive closure and matrix multiplication:

# What is  $A^k = A^{k-1} \times A$  ?

$$a_{ij}^k = V_{x=1}^n a_{ix}^{k-1} a_{xj}$$

$$a_{ij}^k = \begin{cases} 1 & \text{if there is path of length} \\ & \text{at most } k \text{ from } i \text{ to } j \\ 0 & \text{otherwise} \end{cases}$$

# All Pair Shortest Path

# Transitive closure and matrix multiplication:

# What is  $A^n = A \times A \times \dots \times A$  ?

$$a_{ij}^n = V_{x=1}^n a_{ix}^{n-1} a_{xj}$$

$$a_{ij}^n = \begin{cases} 1 & \text{if there is path of length} \\ & \text{at most } n \text{ from } i \text{ to } j \\ 0 & \text{otherwise} \end{cases}$$

$$A^n = A \times A \times \dots \times A = A^*$$

# All Pair Shortest Path

# What is the complexity of multiplying two  $n \times n$  matrices?

→  $O(n^3)$

# How many  $n \times n$  matrix multiplications ?

→  $O(n)$

# Total →  $O(n^4)$

Can We do Better ?

# All Pair Shortest Path

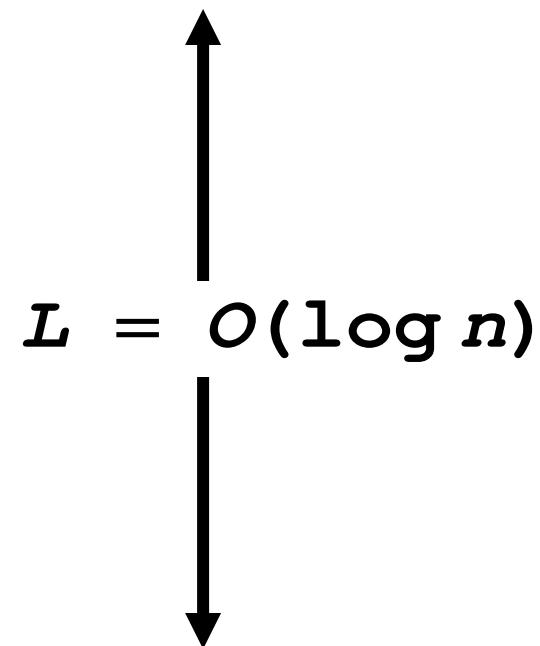
$$A$$

$$A^2 = A \times A$$

$$A^4 = A^2 \times A^2$$

• • • • • • • • •

$$A^n = A^{n/2} \times A^{n/2}$$



Total :  $O(n^3 \log n)$

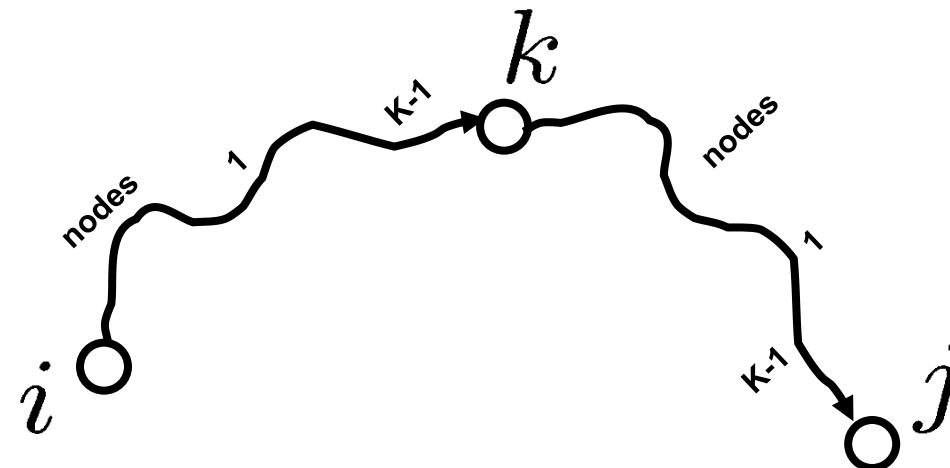
Can We do Better ?

# All Pair Shortest Path

## ANOTHER DEFINITION

$$a_{ij}^k = \begin{cases} 1 & \text{if there is a path from } i \text{ to } j \\ & \text{with intermediate nodes } 1, \dots, k \\ 0 & \text{otherwise} \end{cases}$$

$$a_{ij}^k = a_{ik}^{k-1} a_{kj}^{k-1}$$

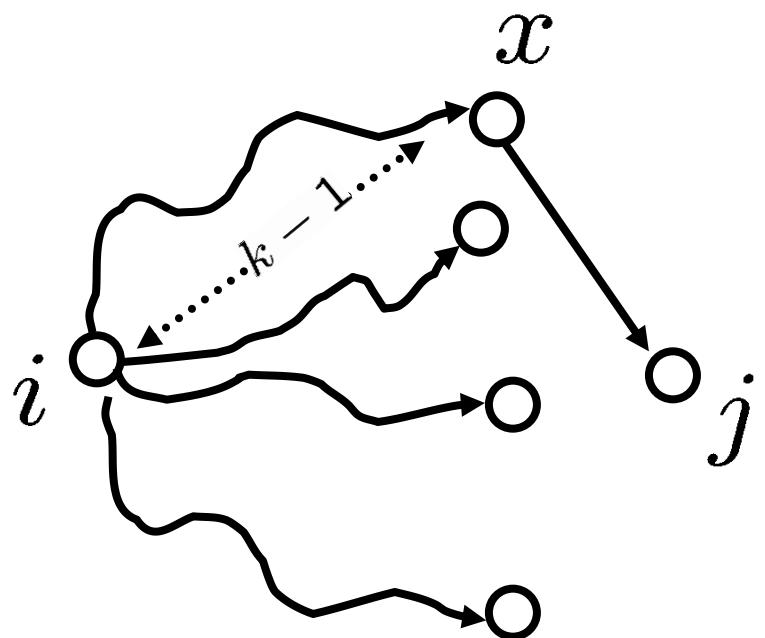


# All Pair Shortest Path

## FIRST DEFINITION

$$a_{ij}^k = \begin{cases} 1 & \text{if there is a path of length} \\ & \text{at most } k \text{ from } i \text{ to } j \\ 0 & \text{otherwise} \end{cases}$$

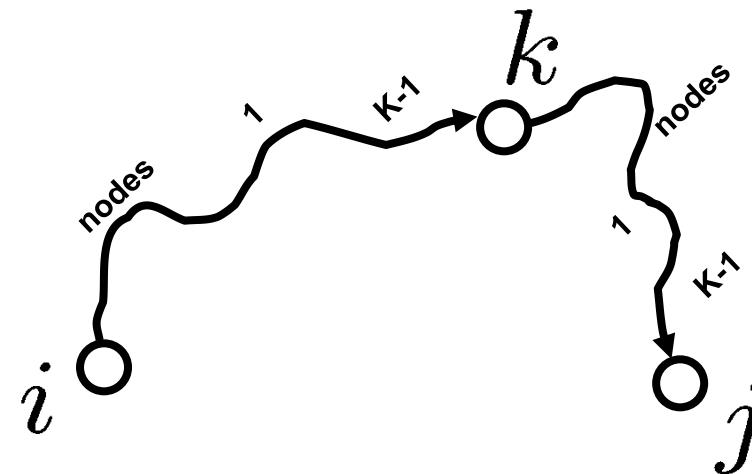
$$a_{ij}^k = V_{x=1}^n a_{ix}^{k-1} a_{xj}$$



## SECOND DEFINITION

$$a_{ij}^k = \begin{cases} 1 & \text{if there is a path from } i \text{ to } j \\ & \text{with intermediate nodes } 1, \dots, k \\ 0 & \text{otherwise} \end{cases}$$

$$a_{ij}^k = a_{ik}^{k-1} a_{kj}^{k-1}$$



# All Pair Shortest Path

## FIRST DEFINITION

initialize  $a_{ij}^1$  to  $A$

for  $k = 2$  to  $n$

    for  $i = 1$  to  $n$

        for  $j = 1$  to  $n$

            initialize  $a_{ij}^k = a_{ij}^{k-1}$

            for  $x = 1$  to  $n$

$a_{ij}^k = a_{ij}^k \vee a_{ix}^{k-1} a_{xj}^1$

$O(n^4)$

## SECOND DEFINITION

initialize  $a_{ij}^0$  to  $A$

for  $k = 1$  to  $n$

    for  $i = 1$  to  $n$

        for  $j = 1$  to  $n$

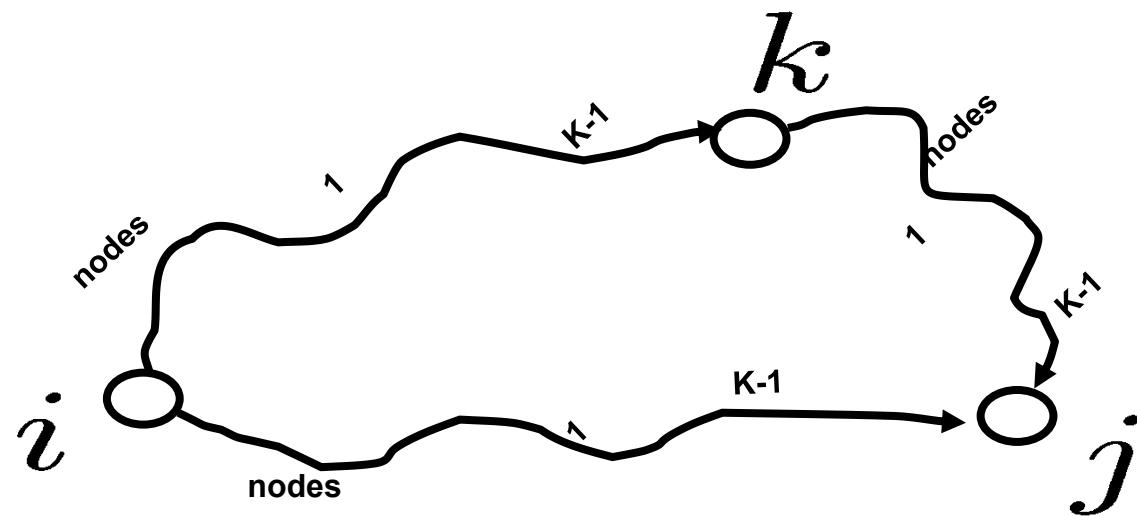
$a_{ij}^k = a_{ij}^{k-1} \vee a_{ik}^{k-1} a_{kj}^{k-1}$

$O(n^3)$

# All Pair Shortest Path

$$d_{ij}^k = \begin{cases} x & \text{if there is a path from } i \text{ to } j \\ & \text{of cost at most } x \\ & \text{with intermediate nodes } 1, \dots, k \\ +\infty & \text{otherwise} \end{cases}$$

$$d_{ij}^k = \min \{ d_{ij}^{k-1}, d_{ik}^{k-1} + d_{kj}^{k-1} \}$$



## All Pair Shortest Path, Floyd-Warshall Algorithm

initialize  $d_{ij}^0 = \begin{cases} 0 & \text{if } i = j \\ w_{ij} & \text{if } (i, j) \in E \text{ and } i \neq j \end{cases}$

for  $k = 1$  to  $n$

    for  $i = 1$  to  $n$

        for  $j = 1$  to  $n$

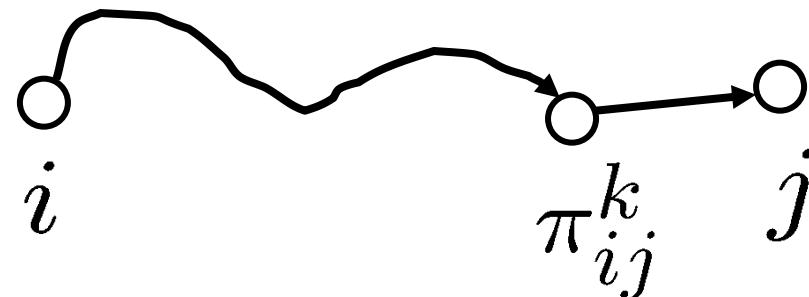
$$d_{ij}^k = \min \{d_{ij}^{k-1}, d_{ik}^{k-1} + d_{kj}^{k-1}\}$$

$O(n^3)$

# All Pair Shortest Path, Floyd-Warshall Algorithm

## PREDECESSOR MATRIX

$\pi_{ij}^k$  is the following:



the vertex before  $j$   
in a shortest path from  $i$  to  $j$   
that uses as intermediate nodes  $1, \dots, k$

# All Pair Shortest Path, Floyd-Warshall Algorithm

initialize  $d_{ij}^0 = \begin{cases} 0 & \text{if } i = j \\ w_{ij} & \text{if } (i, j) \in E \text{ and } i \neq j \end{cases}$

initialize  $\pi_{ij}^0 = \begin{cases} \text{NIL} & \text{if } i = j \\ \text{NIL} & \text{if } (i, j) \notin E \text{ and } i \neq j \\ i & \text{if } (i, j) \in E \text{ and } i \neq j \end{cases}$

for  $k = 1$  to  $n$

    for  $i = 1$  to  $n$

        for  $j = 1$  to  $n$

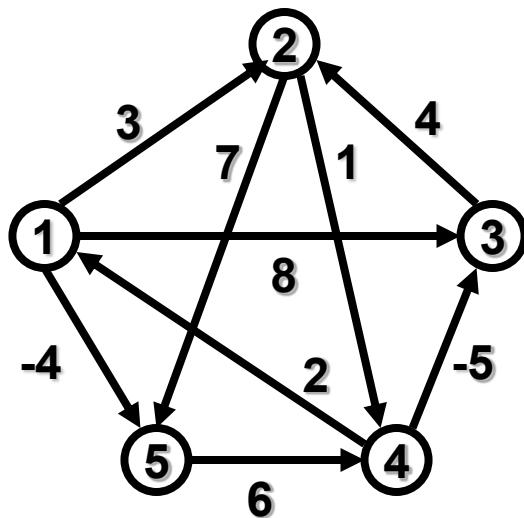
            if  $d_{ij}^{k-1} \leq d_{ik}^{k-1} + d_{kj}^{k-1}$  then  $d_{ij}^k = d_{ij}^{k-1}$

$\pi_{ij}^k = \pi_{ij}^{k-1}$

            if  $d_{ij}^{k-1} > d_{ik}^{k-1} + d_{kj}^{k-1}$  then  $d_{ij}^k = d_{ik}^{k-1} + d_{kj}^{k-1}$

$\pi_{ij}^k = \pi_{jk}^{k-1}$

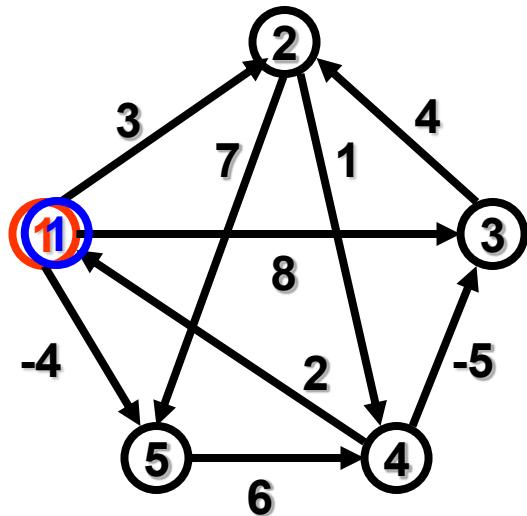
# All Pair Shortest Path, Floyd-Warshall Algorithm



$$D^{(0)} = \begin{matrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{matrix}$$

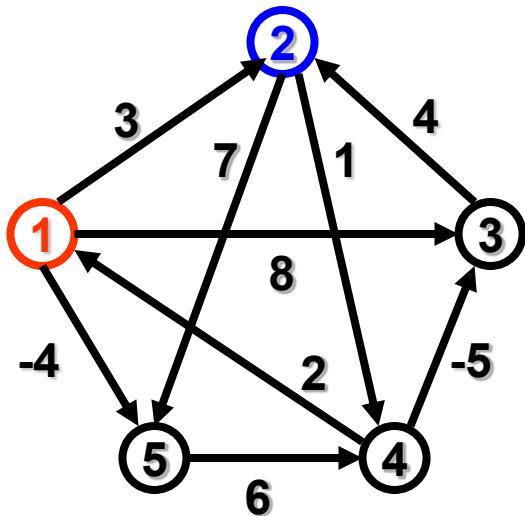
$$\Pi^{(0)} = \begin{matrix} \text{NIL} & 1 & 1 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & \text{NIL} & \text{NIL} \\ 4 & \text{NIL} & 4 & \text{NIL} & \text{NIL} \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{matrix}$$

# All Pair Shortest Path, Floyd-Warshall Algorithm



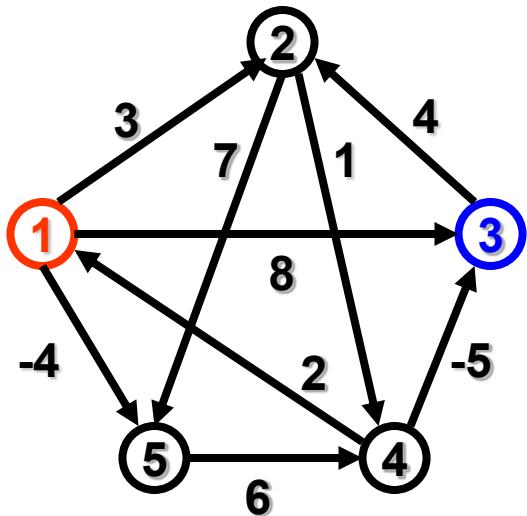
	0	3	8	$\infty$	-	4	$\leftarrow 1$
	$\infty$	0	$\infty$	1		7	
$D^{(1)}$	=	$\infty$	4	0	$\infty$	$\infty$	
	2	$\infty$	-5	0		$\infty$	
	$\infty$	$\infty$	$\infty$	6		0	
	NIL	1	1	NIL	1		
	NIL	NIL	NIL	2	2		
$\Pi^{(1)}$	=	NIL	3	NIL	NIL	NIL	
	4	NIL	4	NIL	NIL		
	NIL	NIL	NIL	5	NIL		

# All Pair Shortest Path, Floyd-Warshall Algorithm



$D^{(1)}$	0	3	8	$\infty$	-	4
	$\infty$	0	$\infty$	1	7	← 2
	$\infty$	4	0	$\infty$	$\infty$	
	2	$\infty$	-5	0	$\infty$	
	$\infty$	$\infty$	$\infty$	6	0	
$\Pi^{(1)}$	NIL	1	1	NIL	1	
	NIL	NIL	NIL	2	2	
	NIL	3	NIL	NIL	NIL	
	4	NIL	4	NIL	NIL	
	NIL	NIL	NIL	5	NIL	

# All Pair Shortest Path, Floyd-Warshall Algorithm



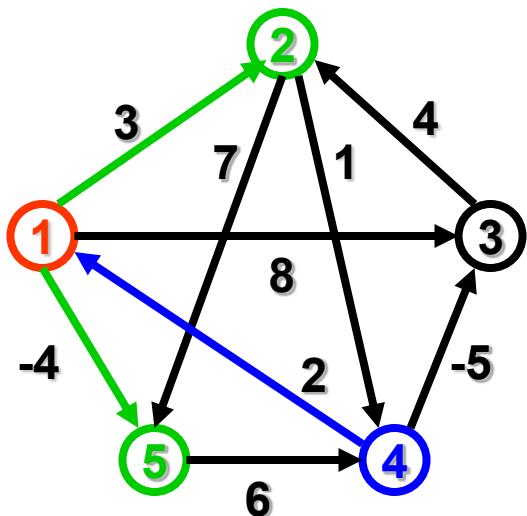
$$D^{(1)} = \begin{matrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{matrix}$$

← 3

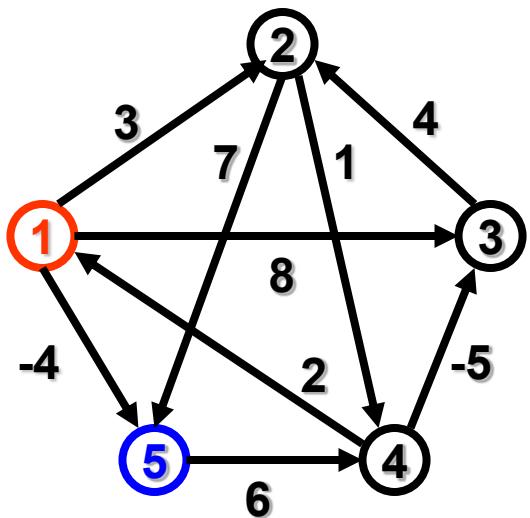
$$\Pi^{(1)} = \begin{matrix} \text{NIL} & 1 & 1 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & \text{NIL} & \text{NIL} \\ 4 & \text{NIL} & 4 & \text{NIL} & \text{NIL} \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{matrix}$$

# All Pair Shortest Path, Floyd-Warshall Algorithm



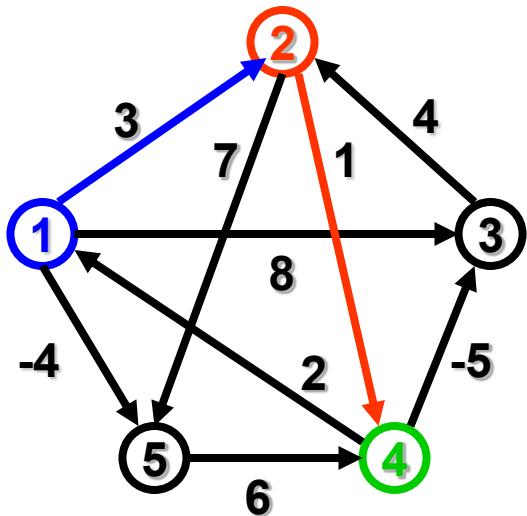
$D^{(1)}$	0	3	8	$\infty$	-4
	$\infty$	0	$\infty$	1	7
	$\infty$	4	0	$\infty$	$\infty$
	2	[5]	-5	0	[-2] ← 4
	$\infty$	$\infty$	$\infty$	6	0
$\Pi^{(1)}$	NIL	1	1	NIL	1
	NIL	NIL	NIL	2	2
	NIL	3	NIL	NIL	NIL
	4	[1]	4	NIL	[1]
	NIL	NIL	NIL	5	NIL

# All Pair Shortest Path, Floyd-Warshall Algorithm



$D^{(1)}$	0	3	8	$\infty$	-	4
	$\infty$	0	$\infty$	1	7	
	$\infty$	4	0	$\infty$	$\infty$	
	2	[5]	-5	0	-2	
	$\infty$	$\infty$	$\infty$	6	0	← 5
$\Pi^{(1)}$	NIL	1	1	NIL	1	
	NIL	NIL	NIL	2	2	
	NIL	3	NIL	NIL	NIL	
	4	[1]	4	NIL	1	
	NIL	NIL	NIL	5	NIL	

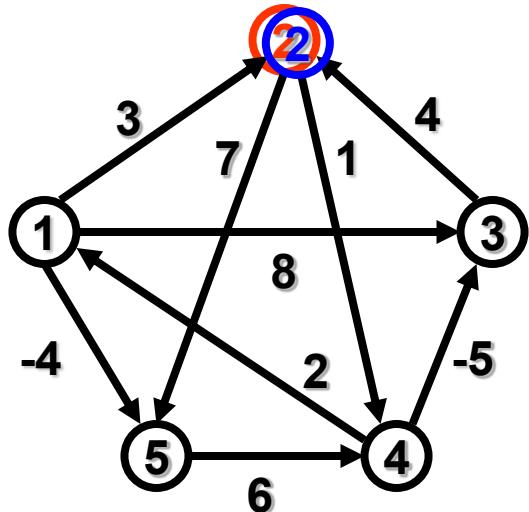
# All Pair Shortest Path, Floyd-Warshall Algorithm



$$D^{(2)} = \begin{matrix} 0 & 3 & 8 & [4] & -4 & \leftarrow 1 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{matrix}$$

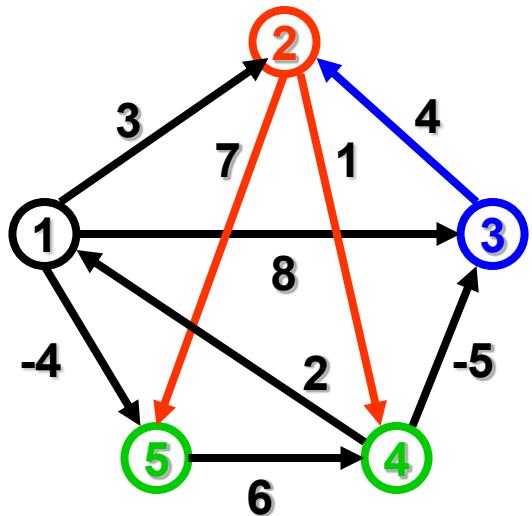
$$\Pi^{(2)} = \begin{matrix} \text{NIL} & 1 & 1 & [2] & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & \text{NIL} & \text{NIL} \\ 4 & [1] & 4 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{matrix}$$

# All Pair Shortest Path, Floyd-Warshall Algorithm



	0	3	8	[4]	-	4
	∞	0	∞	1	7	← 2
$D^{(2)}$ =	∞	4	0	∞	∞	
	2	5	-5	0	-2	
	∞	∞	∞	6	0	
$\Pi^{(2)} =$	NIL	1	1	[2]	1	
	NIL	NIL	NIL	2	2	
	NIL	3	NIL	NIL	NIL	
	4	[1]	4	NIL	1	
	NIL	NIL	NIL	5	NIL	

# All Pair Shortest Path, Floyd-Warshall Algorithm

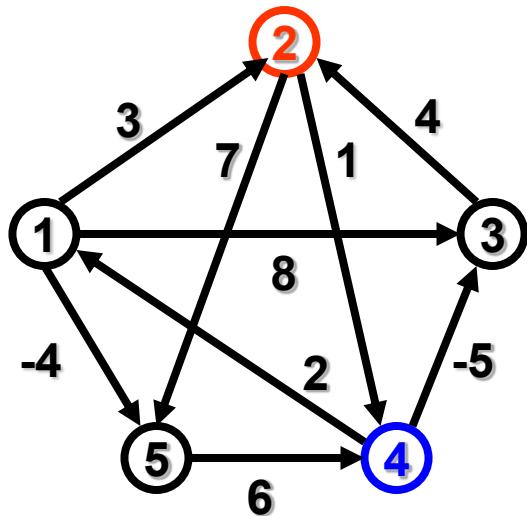


$$D^{(2)} = \begin{matrix} 0 & 3 & 8 & [4] & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & [5] & [11] \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{matrix}$$

← 3

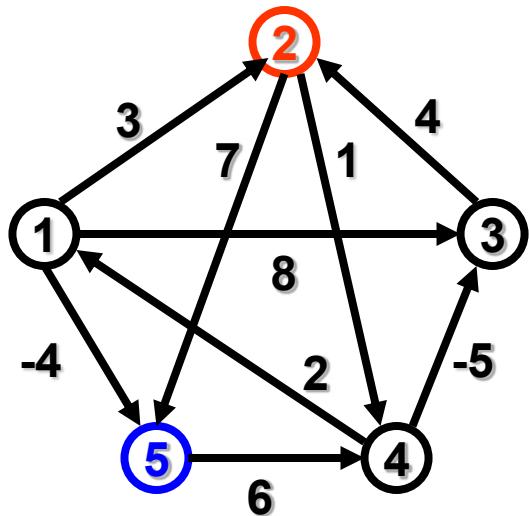
$$\Pi^{(2)} = \begin{matrix} \text{NIL} & 1 & 1 & [2] & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & [2] & [2] \\ 4 & [1] & 4 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{matrix}$$

# All Pair Shortest Path, Floyd-Warshall Algorithm



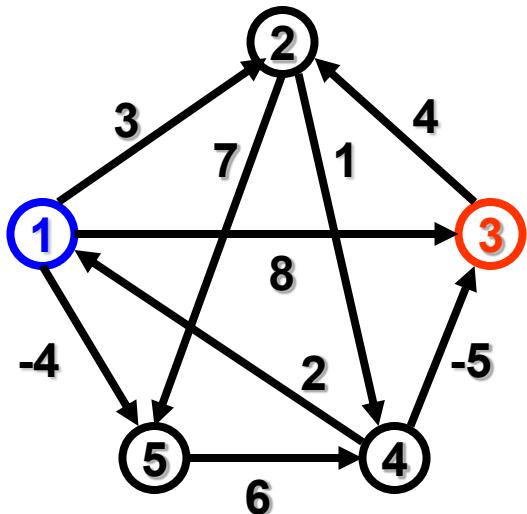
$D^{(2)}$	0	3	8	[4]	-	4
	$\infty$	0	$\infty$	1	7	
	$\infty$	4	0	[5]	[11]	
	2	5	-5	0	-2	← 4
	$\infty$	$\infty$	$\infty$	6	0	
$\Pi^{(2)}$	NIL	1	1	[2]	1	
	NIL	NIL	NIL	2	2	
	NIL	3	NIL	[2]	[2]	
	4	[1]	4	NIL	1	
	NIL	NIL	NIL	5	NIL	

# All Pair Shortest Path, Floyd-Warshall Algorithm



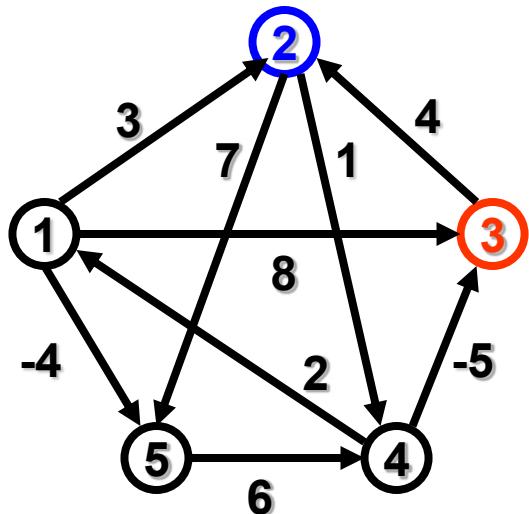
$D^{(2)}$	0	3	8	[4]	-	4
	∞	0	∞	1	7	
	∞	4	0	[5]	[11]	
	2	5	-5	0	-2	
	∞	∞	∞	6	0	← 5
$\Pi^{(2)}$	NIL	1	1	[2]	1	
	NIL	NIL	NIL	2	2	
	NIL	3	NIL	[2]	[2]	
	4	[1]	4	NIL	1	
	NIL	NIL	NIL	5	NIL	

# All Pair Shortest Path, Floyd-Warshall Algorithm



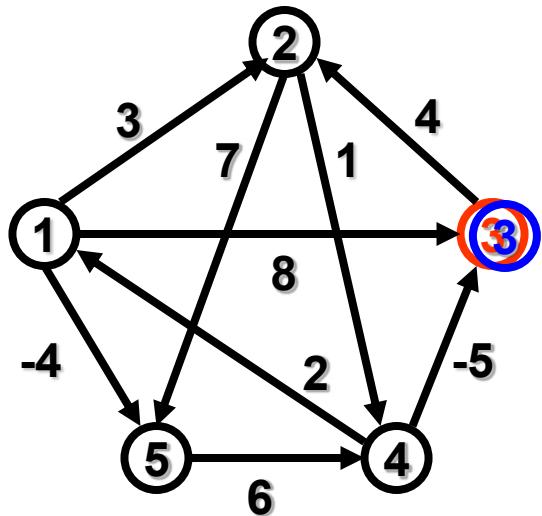
$D^{(3)}$	=	0	3	8	4	-4	1
		$\infty$	0	$\infty$	1	7	
		$\infty$	4	0	5	11	
		2	5	-5	0	-2	
		$\infty$	$\infty$	$\infty$	6	0	
$\Pi^{(3)}$	=	NIL	1	1	2	1	
		NIL	NIL	NIL	2	2	
		NIL	3	NIL	2	2	
		4	1	4	NIL	1	
		NIL	NIL	NIL	5	NIL	

# All Pair Shortest Path, Floyd-Warshall Algorithm



$D^{(3)}$	0	3	8	4	-4
	$\infty$	0	$\infty$	1	7 <span style="color: blue;">← 2</span>
	$\infty$	4	0	5	11
	2	5	-5	0	-2
	$\infty$	$\infty$	$\infty$	6	0
$\Pi^{(3)}$	NIL	1	1	2	1
	NIL	NIL	NIL	2	2
	NIL	3	NIL	2	2
	4	1	4	NIL	1
	NIL	NIL	NIL	5	NIL

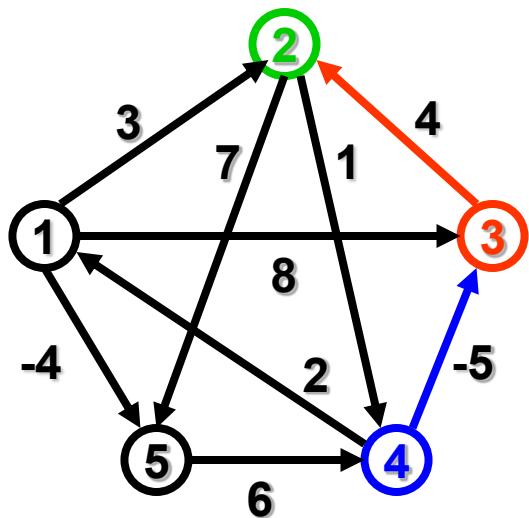
# All Pair Shortest Path, Floyd-Warshall Algorithm



$$D^{(3)} = \begin{matrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \xleftarrow{3} \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{matrix}$$

$$\Pi^{(3)} = \begin{matrix} \text{NIL} & 1 & 1 & 2 & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & 2 & 2 \\ 4 & 1 & 4 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{matrix}$$

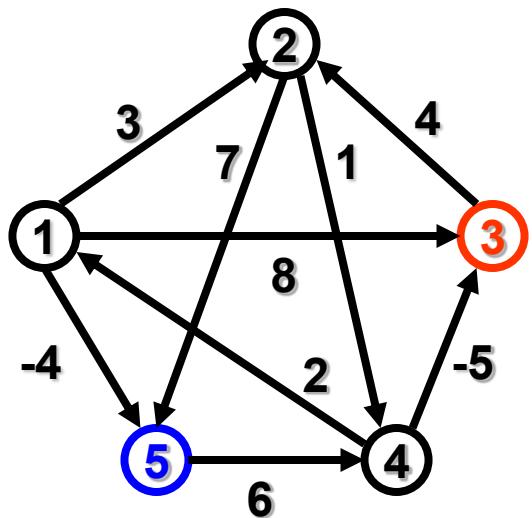
# All Pair Shortest Path, Floyd-Warshall Algorithm



$$D^{(3)} = \begin{matrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & [-1] & -5 & 0 & -2 \xleftarrow{4} \\ \infty & \infty & \infty & 6 & 0 \end{matrix}$$
  

$$\Pi^{(3)} = \begin{matrix} \text{NIL} & 1 & 1 & 2 & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & 2 & 2 \\ 4 & [3] & 4 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{matrix}$$

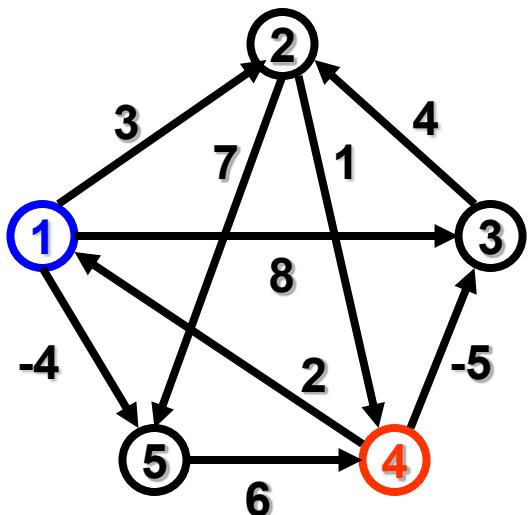
# All Pair Shortest Path, Floyd-Warshall Algorithm



$$D^{(3)} = \begin{matrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & [-1] & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \leftarrow 5 \end{matrix}$$
  

$$\Pi^{(3)} = \begin{matrix} \text{NIL} & 1 & 1 & 2 & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & 2 & 2 \\ 4 & [3] & 4 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{matrix}$$

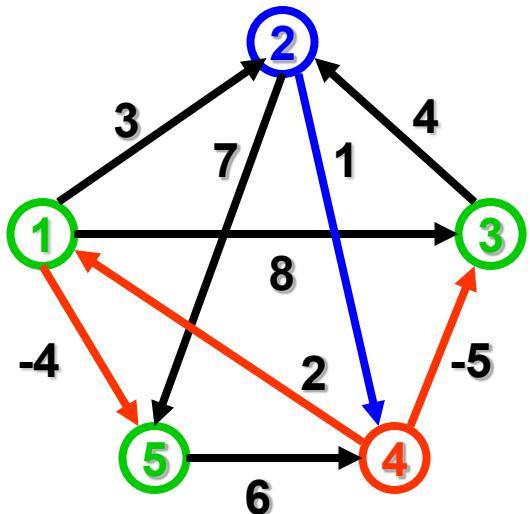
# All Pair Shortest Path, Floyd-Warshall Algorithm



$$D^{(4)} = \begin{matrix} 0 & 3 & 8 & 4 & -4 & \textcircled{1} \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & -1 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{matrix}$$
  

$$\Pi^{(4)} = \begin{matrix} \text{NIL} & 1 & 1 & 2 & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & 2 & 2 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{matrix}$$

# All Pair Shortest Path, Floyd-Warshall Algorithm



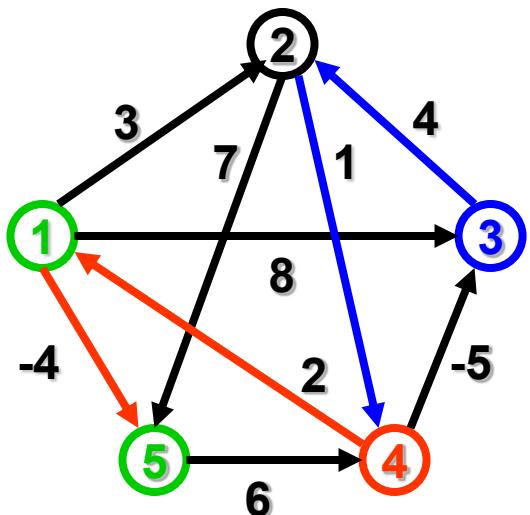
$$D^{(4)} = \begin{bmatrix} 0 & 3 & 8 & 4 & -4 \\ [3] & 0 & [-4] & 1 & [-1] \end{bmatrix} \xleftarrow{\textcircled{2}}$$

$$\begin{matrix} \infty & 4 & 0 & 5 & 11 \\ 2 & -1 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{matrix}$$
  

$$\Pi^{(4)} = \begin{bmatrix} \text{NIL} & 1 & 1 & 2 & 1 \\ [4] & \text{NIL} & [4] & 2 & [1] \end{bmatrix}$$

$$\begin{matrix} \text{NIL} & 3 & \text{NIL} & 2 & 2 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{matrix}$$

# All Pair Shortest Path, Floyd-Warshall Algorithm

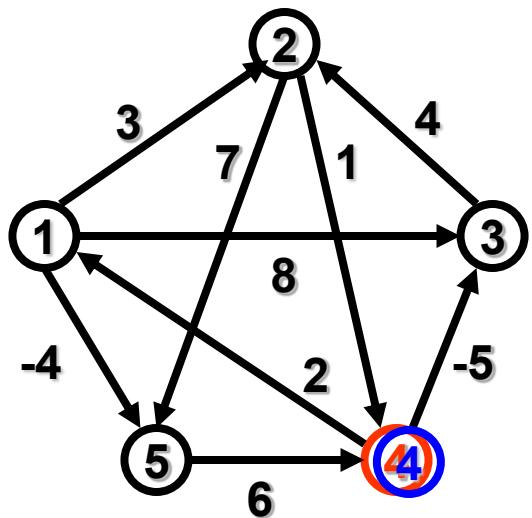


$$D^{(4)} = \begin{bmatrix} 0 & 3 & 8 & 4 & -4 \\ [3] & 0 & [-4] & 1 & [-1] \\ [7] & 4 & 0 & 5 & [3] \\ 2 & -1 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{bmatrix}$$

$D^{(4)}$  is highlighted with a blue circle around the value 3 at position [3][4].

$$\Pi^{(4)} = \begin{bmatrix} \text{NIL} & 1 & 1 & 2 & 1 \\ [4] & \text{NIL} & [4] & 2 & [1] \\ [4] & 3 & \text{NIL} & 2 & [1] \\ 4 & 3 & 4 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{bmatrix}$$

# All Pair Shortest Path, Floyd-Warshall Algorithm

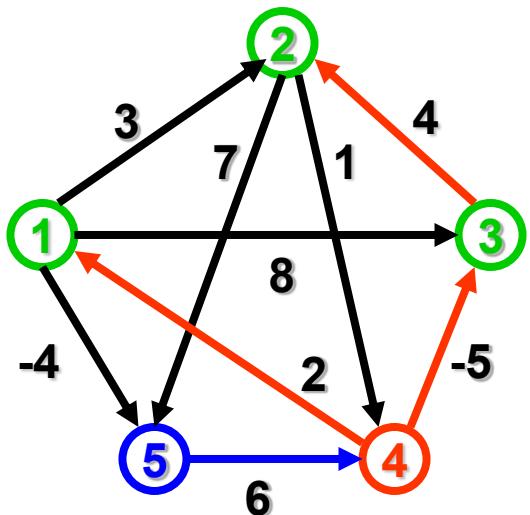


$$D^{(4)} = \begin{bmatrix} 0 & 3 & 8 & 4 & -4 \\ [3] & 0 & [-4] & 1 & [-1] \\ [7] & 4 & 0 & 5 & [3] \\ 2 & -1 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{bmatrix}$$

← 4

$$\Pi^{(4)} = \begin{bmatrix} \text{NIL} & 1 & 1 & 2 & 1 \\ [4] & \text{NIL} & [4] & 2 & [1] \\ [4] & 3 & \text{NIL} & 2 & [1] \\ 4 & 3 & 4 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{bmatrix}$$

# All Pair Shortest Path, Floyd-Warshall Algorithm

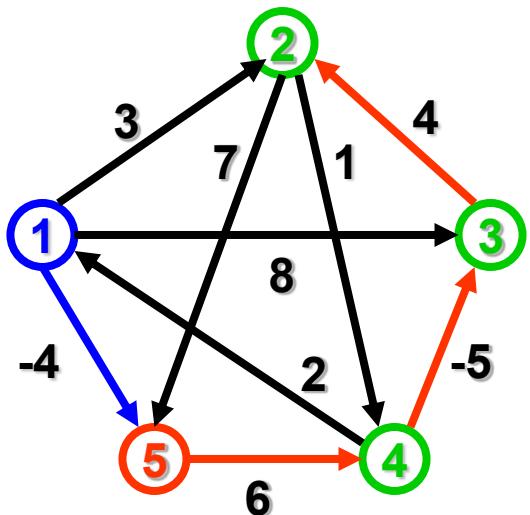


$$D^{(4)} = \begin{bmatrix} 0 & 3 & [-1] & 4 & -4 \\ [3] & 0 & [-4] & 1 & [-1] \\ [7] & 4 & 0 & 5 & [3] \\ 2 & -1 & -5 & 0 & -2 \\ [8] & [5] & [1] & 6 & 0 \end{bmatrix}$$

← 5

$$\Pi^{(4)} = \begin{bmatrix} \text{NIL} & 1 & [4] & 2 & 1 \\ [4] & \text{NIL} & [4] & 2 & [1] \\ [4] & 3 & \text{NIL} & 2 & [1] \\ 4 & 3 & 4 & \text{NIL} & 1 \\ [4] & [3] & [4] & 5 & \text{NIL} \end{bmatrix}$$

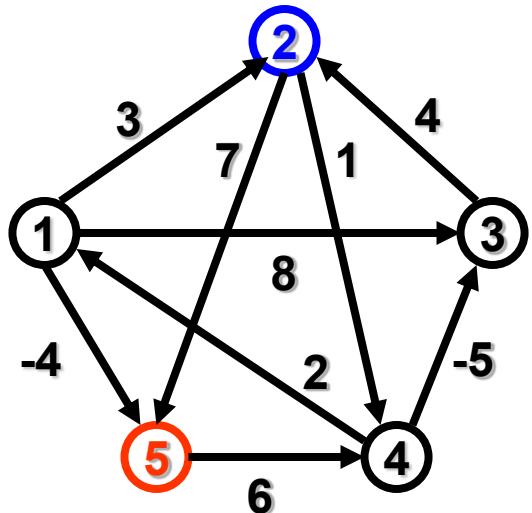
# All Pair Shortest Path, Floyd-Warshall Algorithm



$$D^{(5)} = \begin{bmatrix} 0 & [1] & [-3] & [2] & -4 & \leftarrow 1 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{bmatrix}$$

$$\Pi^{(5)} = \begin{bmatrix} \text{NIL} & [3] & [4] & [5] & 1 \\ 4 & \text{NIL} & 4 & 2 & 1 \\ 4 & 3 & \text{NIL} & 2 & 1 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ 4 & 3 & 4 & 5 & \text{NIL} \end{bmatrix}$$

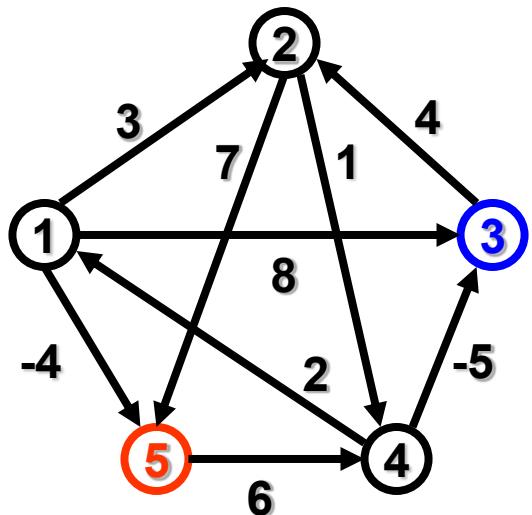
# All Pair Shortest Path, Floyd-Warshall Algorithm



$$D^{(5)} = \begin{bmatrix} 0 & [1] & [-3] & [2] & -4 \\ 3 & 0 & -4 & 1 & -1 \end{bmatrix} \xrightarrow{\text{2}} \begin{bmatrix} 0 & [1] & [-3] & [2] & -4 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{bmatrix}$$

$$\Pi^{(5)} = \begin{bmatrix} \text{NIL} & [3] & [4] & [5] & 1 \\ 4 & \text{NIL} & 4 & 2 & 1 \\ 4 & 3 & \text{NIL} & 2 & 1 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ 4 & 3 & 4 & 5 & \text{NIL} \end{bmatrix}$$

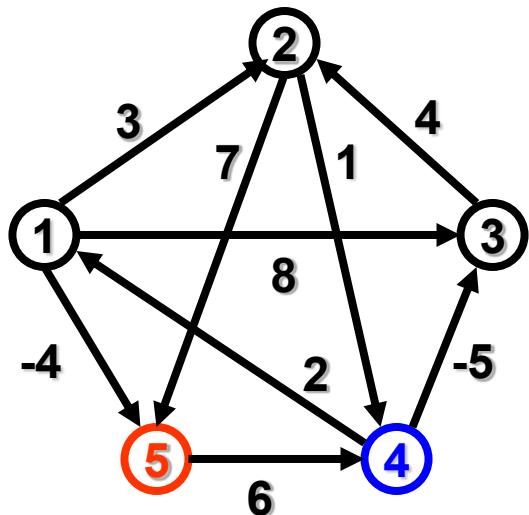
# All Pair Shortest Path, Floyd-Warshall Algorithm



$$D^{(5)} = \begin{bmatrix} 0 & [1] & [-3] & [2] & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{bmatrix}$$

$$\Pi^{(5)} = \begin{bmatrix} \text{NIL} & [3] & [4] & [5] & 1 \\ 4 & \text{NIL} & 4 & 2 & 1 \\ 4 & 3 & \text{NIL} & 2 & 1 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ 4 & 3 & 4 & 5 & \text{NIL} \end{bmatrix}$$

# All Pair Shortest Path, Floyd-Warshall Algorithm

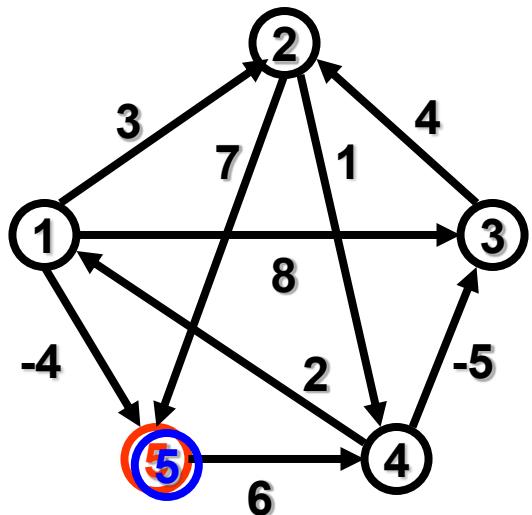


$$D^{(5)} = \begin{bmatrix} 0 & [1] & [-3] & [2] & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{bmatrix}$$

← 4

$$\Pi^{(5)} = \begin{bmatrix} \text{NIL} & [3] & [4] & [5] & 1 \\ 4 & \text{NIL} & 4 & 2 & 1 \\ 4 & 3 & \text{NIL} & 2 & 1 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ 4 & 3 & 4 & 5 & \text{NIL} \end{bmatrix}$$

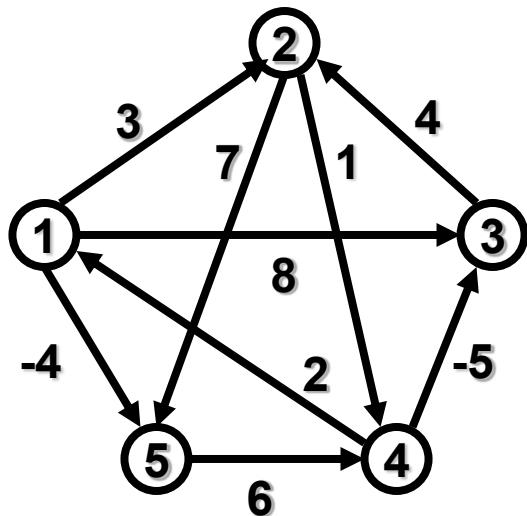
# All Pair Shortest Path, Floyd-Warshall Algorithm



$$D^{(5)} = \begin{bmatrix} 0 & [1] & [-3] & [2] & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{bmatrix} \xleftarrow{\textcircled{5}}$$

$$\Pi^{(5)} = \begin{bmatrix} \text{NIL} & [3] & [4] & [5] & 1 \\ 4 & \text{NIL} & 4 & 2 & 1 \\ 4 & 3 & \text{NIL} & 2 & 1 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ 4 & 3 & 4 & 5 & \text{NIL} \end{bmatrix}$$

# All Pair Shortest Path, Floyd-Warshall Algorithm



$$D^* = \begin{matrix} 0 & 1 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{matrix}$$

$$\Pi^* = \begin{matrix} \text{NIL} & 3 & 4 & 5 & 1 \\ 4 & \text{NIL} & 4 & 2 & 1 \\ 4 & 3 & \text{NIL} & 2 & 1 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ 4 & 3 & 4 & 5 & \text{NIL} \end{matrix}$$

# Reading Materials

## # CLRS

### ❖ Exercises:

- ◆ 24.1-1, 24.1-2, 24.1-3, 24.2-2, 25.2-6

### ❖ Problems:

- ◆ 24-3

## # HSR

### ❖ Examples:

- ◆ 4.11, 4.12, 5.14