



# **Chapter 1**

## **Introduction**

**Lecturer:**  
**PRIYANKA**

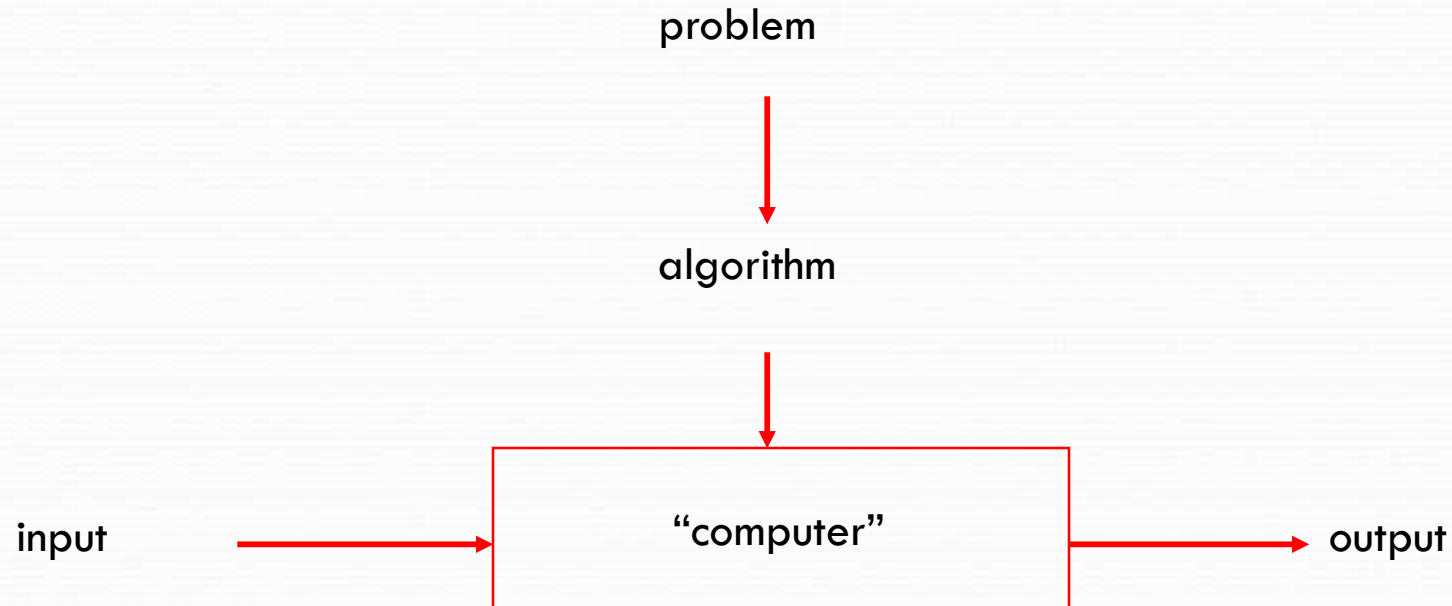
# Algorithm



- 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.



# Notion of algorithm



Algorithmic solution





# PSEUDOCODE

- Pseudocode (pronounced SOO-doh-kohd) is a detailed yet readable description of what a computer program or algorithm must do, expressed in a formally-styled natural language rather than in a programming language.
- It is sometimes used as a detailed step in the process of developing a program.
- It allows programmers to express the design in great detail and provides programmers a detailed template for the next step of writing code in a specific programming language.

# Formatting and Conventions in Pseudocoding



- INDENTATION in pseudocode should be identical to its implementation in a programming language. Try to indent at least four spaces.
- The pseudocode entries are to be cryptic, AND SHOULD NOT BE PROSE. NO SENTENCES.
- No flow boxes in pseudocode.
- Do not include data declarations in pseudocode.

# Some Keywords That Should be Used



- For looping and selection,
  - Do While...EndDo;
  - Do Until...Enddo;
  - Case...EndCase;
  - If...Endif;
  - Call ... with (parameters); Call; Return ....; Return; When; Always use scope terminators for loops and iteration.



## Some Keywords ...

- As verbs, use the words
  - generate, Compute, Process,
  - Set, reset,
  - increment,
  - calculate,
  - add, sum, multiply, ...
  - print, display,
  - input, output, edit, test , etc.



# Methods of finding GCD

## Competition

Computing Greatest Common Divisor:  $\text{gcd}(m,n)$



### Primary School

1.  $t := \min(m, n)$
2.  $m \bmod t = 0?$
3. Yes?  $n \bmod t = 0?$   
Return  $t$
4. No?  $t = t - 1$ ; goto 2

M - 1

### Secondary School

1. Find prime factors of  $m$
2. Find prime factors of  $n$
3. Identify common factors
4. Return product of these

M - 2

### University

1.  $n = 0?$
2. Yes? Return  $m$
3.  $r = m \bmod n$ ,  
 $m := n$   
 $n := r$
4. Go to 1

M - 3



Euclid

1-17

433-253 Algorithms and Data Structures



# Euclid's Algorithm

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

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

Euclid's algorithm is based on repeated application of equality

$$\gcd(m, n) = \gcd(n, m \bmod n)$$

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

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



## Two descriptions of Euclid's algorithm

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

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

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

while  $n \neq 0$  do

$r \leftarrow m \bmod n$

$m \leftarrow n$

$n \leftarrow r$

return  $m$

# $\gcd(m,n)$



Consecutive integer checking algorithm

Step 1 Assign the value of  $\min\{m,n\}$  to  $t$

Step 2 Divide  $m$  by  $t$ . If the remainder is 0, go to Step 3;  
otherwise, go to Step 4

Step 3 Divide  $n$  by  $t$ . If the remainder is 0, return  $t$  and  
stop;  
otherwise, go to Step 4

Step 4 Decrease  $t$  by 1 and go to Step 2



# Other methods for $\gcd(m,n)$ [cont.]



Middle-school procedure

Step 1 Find the prime factorization of  $m$

Step 2 Find the prime factorization of  $n$

Step 3 Find all the common prime factors

Step 4 Compute the product of all the common prime factors

and return it as  $\gcd(m,n)$

Is this an algorithm?



# Sieve of Eratosthenes



Input: Integer  $n \geq 2$

Output: List of primes less than or equal to  $n$

for  $p \leftarrow 2$  to  $n$  do  $A[p] \leftarrow p$

for  $p \leftarrow 2$  to  $\lfloor n \rfloor$  do

    if  $A[p] \neq 0$  //  $p$  hasn't been previously eliminated from the list

$j \leftarrow p * p$

        while  $j \leq n$  do

$A[j] \leftarrow 0$  //mark element as eliminated

$j \leftarrow j + p$

Example: 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20

# Termination of Euclid's Algorithm



- The second number of the pair gets smaller with each iteration and cannot become negative:
- Indeed, the new value of  $n$  is  $r = m \bmod n$ , which is always smaller than  $n$ .
- Eventually,  $r$  becomes zero, and the algorithm stops.



| Step | Q | Remainder          | Substitute  | Combine terms            |
|------|---|--------------------|---|--------------------------|
| 1    |   | 120                |   | $120 = 120 * 1 + 23 * 0$ |
| 2    |   | 23                 |   | $23 = 120 * 0 + 23 * 1$  |
| 3    | 5 | $5 = 120 - 23 * 5$ | $5 = (120 * 1 + 23 * 0) - (120 * 0 + 23 * 1) * 5$     | $5 = 120 * 1 + 23 * -5$  |
| 4    | 4 | $3 = 23 - 5 * 4$   | $3 = (120 * 0 + 23 * 1) - (120 * 1 + 23 * -5) * 4$    | $3 = 120 * -4 + 23 * 21$ |
| 5    | 1 | $2 = 5 - 3 * 1$    | $2 = (120 * 1 + 23 * -5) - (120 * -4 + 23 * 21) * 1$  | $2 = 120 * 5 + 23 * -26$ |
| 6    | 1 | $1 = 3 - 2 * 1$    | $1 = (120 * -4 + 23 * 21) - (120 * 5 + 23 * -26) * 1$ | $1 = 120 * -9 + 23 * 47$ |
| 7    | 2 | 0                  | <i>End of algorithm</i>                               |                          |

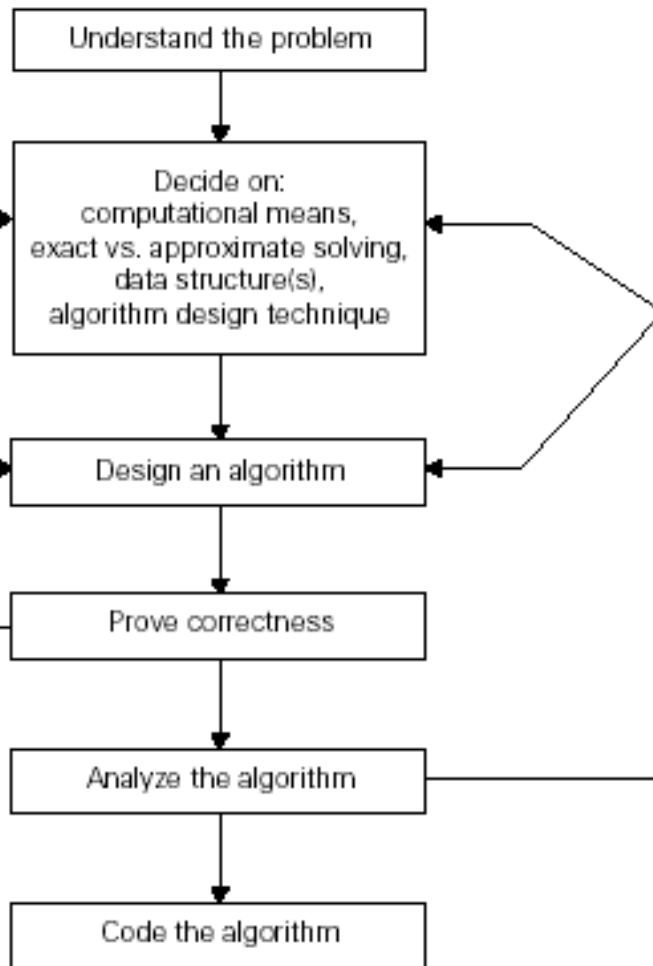


# FUNDAMENTALS OF ALGORITHMIC PROBLEM SOLVING

# Fundamentals of Algorithmic Problem Solving

- Algorithm = Procedural Solutions to Problem
- NOT an answer, BUT rather specific instructions of getting answers.
- Therefore, requires steps in designing and analyzing an algorithm

# Algorithm Design & Analysis Process



# Step 1: Understand the Problem



- ☐ Before designing an algorithm - understand completely the problem given.
- ☐ Read the problem's description carefully and ask questions if you have any doubts about the problem,
- ☐ Do a few small examples by hand, think about special cases, and ask questions again if needed.





# Step 1: Understand the Problem

- ❑ An input to an algorithm specifies an *instance* of the problem the algorithm solves.
- ❑ It is very important to specify exactly the range of instances the algorithm needs to handle.
- ❑ Failing which – the algorithm works correctly for some inputs , but crashes on some boundary values.
- ❑ Remember that a correct algorithm is not one that works most of the time but one that works correctly for *all* legitimate inputs.



# Capabilities of a computational device



- Algorithms designed to be executed on machines that executes instructions one after another are called *sequential algorithms*.
- Algorithms that take advantage of computers that can execute operations concurrently are called *parallel algorithms*.



# Step 3: Choosing between Exact & Approximate Problem Solving



- Solving the problem exactly - Exact algorithms
- Solving the problem approximately - Approximation algorithms
- Why approximation algorithms?
  1. Problems cannot be solved exactly.

Eg. Extracting square roots, solving non-linear equations
  2. Available exact algorithms are unacceptably slow because of problem's complexity

Eg. Traveling Salesman Problem
  3. Approx. Algs can be a part of algorithms that solve the problem exactly.



# Step 4: Deciding on Appropriate Data Structures



- In the new world of object-oriented programming, data structures remain important for both design and analysis of algorithms.
- However, we will assume a very basic data structure for now and concentrate on the algorithm side.



# Step 3: Algorithm Design Techniques



- An *algorithm design technique* (or “strategy” or “paradigm”) is a general approach to solving problems algorithmically that is applicable to a variety of problems from different areas of computing.
- Eg. Brute force, Divide-and-Conquer, Transform-and-Conquer
- Importance:
  1. Provide guidance for designing algorithms for new problems.
  2. To classify algorithms according to an underlying design idea.



# Step 6: Methods of Specifying an Algorithm



- *Pseudocode*, a mixture of a natural language and programming language-like constructs.
- *flowchart*, a method of expressing an algorithm by a collection of connected geometric shapes containing descriptions of the algorithm's steps.



# Step 7: Proving an Algorithm's Correctness



- Prove algorithm's correctness = prove that the algorithm yields a required result for every legitimate input in a finite amount of time.
- For an approximation algorithm, correctness means to be able to show that the error produced by the algorithm does not exceed a predefined limit.



# Algorithm



## 1. Efficiency

- *Time efficiency* indicates how fast the algorithm runs.
- *space efficiency* indicates how much extra memory the algorithm needs.

## 2. Simplicity

## 3. Generality

- Design an algorithm for a problem posed in more general terms.
- Design an algorithm that can handle a range of inputs that is natural for the problem at hand.



# Step 9: Coding the algorithm



- ☐ More than implementation
- ☐ Peril of incorrect & inefficient implementation
- ☐ Require testing & debugging
- ☐ Require code optimizing





# Important Problem Types



# Important Problem Types



- ☐ Sorting
- ☐ Searching
- ☐ String processing
- ☐ Graph problems
- ☐ Combinatorial problems
- ☐ Geometric problems
- ☐ Numerical problems



# Sorting



- The *sorting problem* asks us to rearrange the items of a given list in ascending order.
  
- we usually need to
  - sort lists of numbers,
  - characters from an alphabet,
  - character strings,
  - records similar to those maintained by schools about their students,
  - libraries about their holdings,
  - companies about their employees.

# Searching



- The *searching problem* deals with finding a given value, called a *search key*, in a given set (or a multiset, which permits several elements to have the same value).



# String Processing



- A *string* is a sequence of characters from an alphabet.
- String of particular interest:
  1. Text string – comprises letters, numbers, and special characters
  2. Bit string – comprises zeros and ones
  3. Gene sequence
- Mainly *string matching problem*: searching for a given word in a text



# Graph Problems



- A *graph* can be thought of as a collection of points called vertices, some of which are connected by line segments called edges.
- Used for modeling a wide variety of real-life applications.
- Basic graph algorithms include:
  1. **Graph traversal algorithms** - How can one visit all the points in a network?
  2. **Shortest-path algorithms** - What is the best Introduction route between two cities?
  3. **Topological sorting** for graphs with directed edges



# Combinatorial Problems



- *combinatorial problems*: problems that ask (explicitly or implicitly) to find a combinatorial object—such as a permutation, a combination, or a subset—that satisfies certain constraints and has some desired property (e.g., maximizes a value or minimizes a cost).
1. Combinatorial grows extremely fast with problem size
  2. No known algorithm solving most such problems exactly in an acceptable amount of time.



# Geometric Problems



- *Geometric algorithms* deal with geometric objects such as points, lines, and polygons.

2 class problems:

- The *closest pair problem*: given  $n$  points in the plane, find the closest pair among them.
- The *convex hull problem* asks to find the smallest convex polygon that would include all the points of a given set. If



# Numerical Problems



- *Numerical problems*, another large special area of applications, are problems that involve mathematical objects of continuous nature: solving equations and systems of equations, computing definite integrals, evaluating functions, and so on.





# Fundamentals of Analysis of algorithm efficiency



# Analysis of algorithms

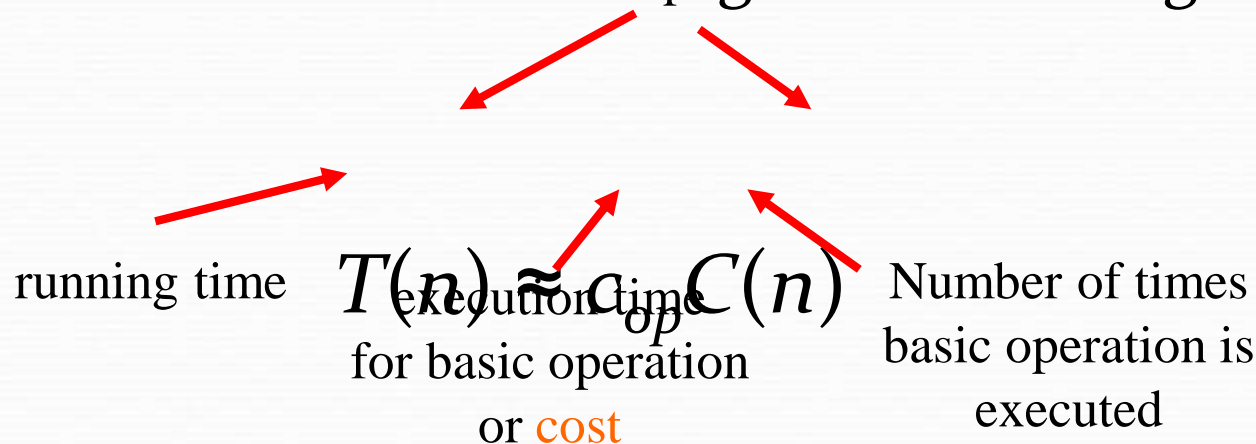
- Issues:
    - correctness
    - time efficiency
    - space efficiency
    - optimality
  - Approaches:
    - theoretical analysis
    - empirical analysis
- Design and Analysis of Algorithms – Unit I

# Theoretical analysis of time



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

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



**Note: Different basic operations may cost differently!**

# examples



| <i>Problem</i>                            | <i>Input size measure</i>                               | <i>Basic operation</i>                  |
|---|---|---|
| Searching for key in a list of $n$ items  | Number of list's items, i.e. $n$                        | Key comparison                          |
| Multiplication of two matrices            | Matrix dimensions or total number of elements           | Multiplication of two numbers           |
| Checking primality of a given integer $n$ | $n$ 'size = number of digits (in binary representation) | Division                                |
| Typical graph problem                     | #vertices and/or edges                                  | Visiting a vertex or traversing an edge |

# Empirical analysis of time efficiency



- Select a specific (typical) sample of inputs
- Use physical unit of time (e.g., milliseconds)  
or  
Count actual number of basic operation's executions
- Analyze the empirical data



# Efficiencies

- Worst Case Efficiency:
  - Is its efficiency for the worst case input of size  $n$ , which is an input of size  $n$  for which the algorithm runs the longest among all possible inputs of that size
  - $C_{\text{worst}}(n)$
- Best-case efficiency:
  - Is its efficiency for the worst case input of size  $n$ , which is an input of size  $n$  for which the algorithm runs the fastest among all possible inputs of that size
  - $C_{\text{best}}(n)$



# Amortized efficiency

- It applies not to a single run of an algorithm, but rather to a sequence of operations performed on the same data structure



# worst-case

For some algorithms, efficiency depends on form of input:

- Worst case:  $C_{\text{worst}}(n)$  – maximum over inputs of size  $n$
- Best case:  $C_{\text{best}}(n)$  – minimum over inputs of size  $n$
- Average case:  $C_{\text{avg}}(n)$  – “average” over inputs of size  $n$ 
  - Number of times the basic operation will be executed on typical input
  - NOT the average of worst and best case
  - Expected number of basic operations considered as a random variable under some assumption about the probability distribution of all possible inputs. So, avg = expected under uniform distribution.

# Example: Sequential search



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

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

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

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

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

$i \leftarrow 0$

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

$i \leftarrow i + 1$

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

**else return**  $-1$

- Worst case  
n key comparisons
- Best case  
1 comparisons
- Average case  
 $(n+1)/2$ , assuming  $K$  is in  $A$



# Types of formulas for basic operation's count



- Exact formula

e.g.,  $C(n) = n(n-1)/2$

- Formula indicating order of growth with specific multiplicative constant

e.g.,  $C(n) \approx 0.5 n^2$

- Formula indicating order of growth with unknown multiplicative constant

e.g.,  $C(n) \approx cn^2$





# Order of growth

- Most important: Order of growth within a constant multiple as  $n \rightarrow \infty$

- Example:
  - How much faster will algorithm run on computer that is twice as fast?
  - How much longer does it take to solve problem of double input size?

# Values of some important functions as $n \rightarrow \infty$



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

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





# Asymptotic Notations

- $O$  (Big-Oh)-notation
- $\Omega$  (Big-Omega) -notation
- $\Theta$  (Big-Theta) -notation





# Asymptotic order of growth

A way of comparing functions that ignores constant factors and small input sizes (because?)

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

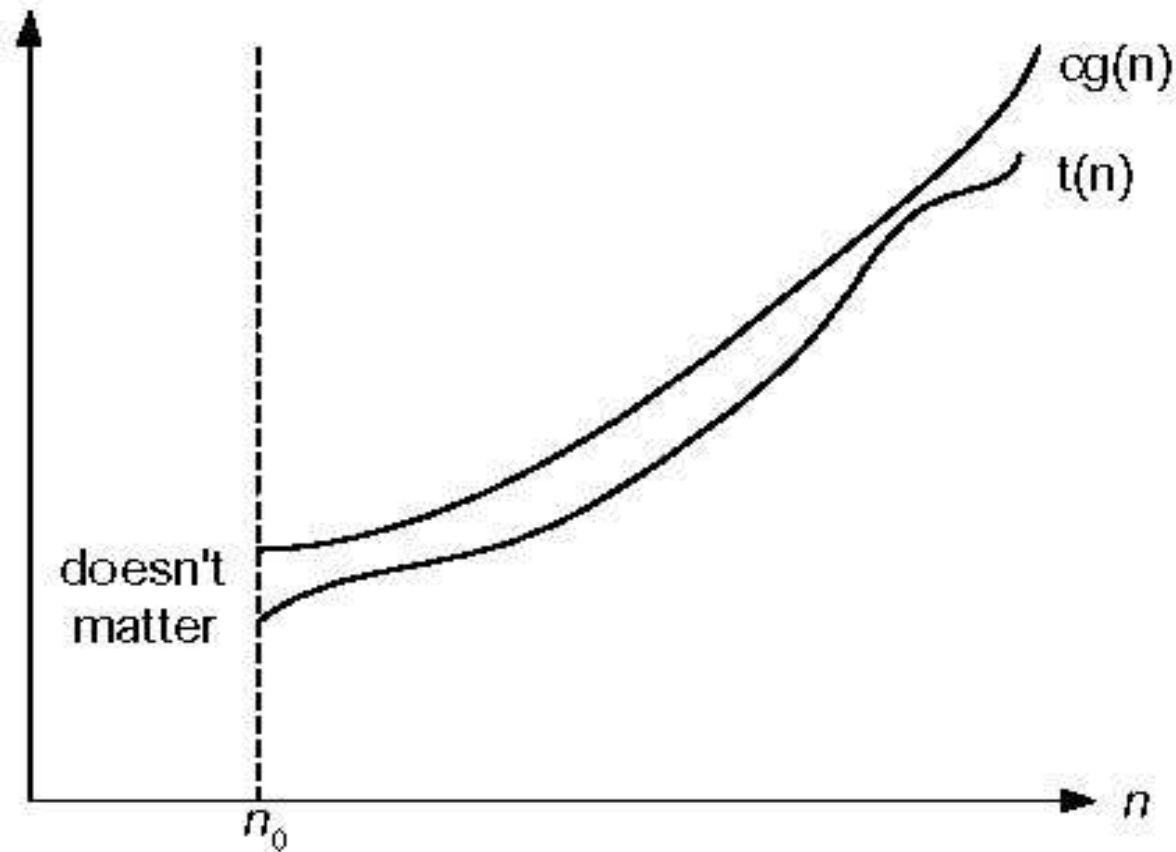


Definition: A function  $t(n)$  is said to be in  $O(g(n))$ , denoted  $t(n) \in O(g(n))$  is bounded above by some constant multiple of  $g(n)$  for all large  $n$ , i.e., there exist positive constant  $c$  and non-negative integer  $n_0$  such that

$$f(n) \leq c g(n) \text{ for every } n \geq n_0$$



# Big-



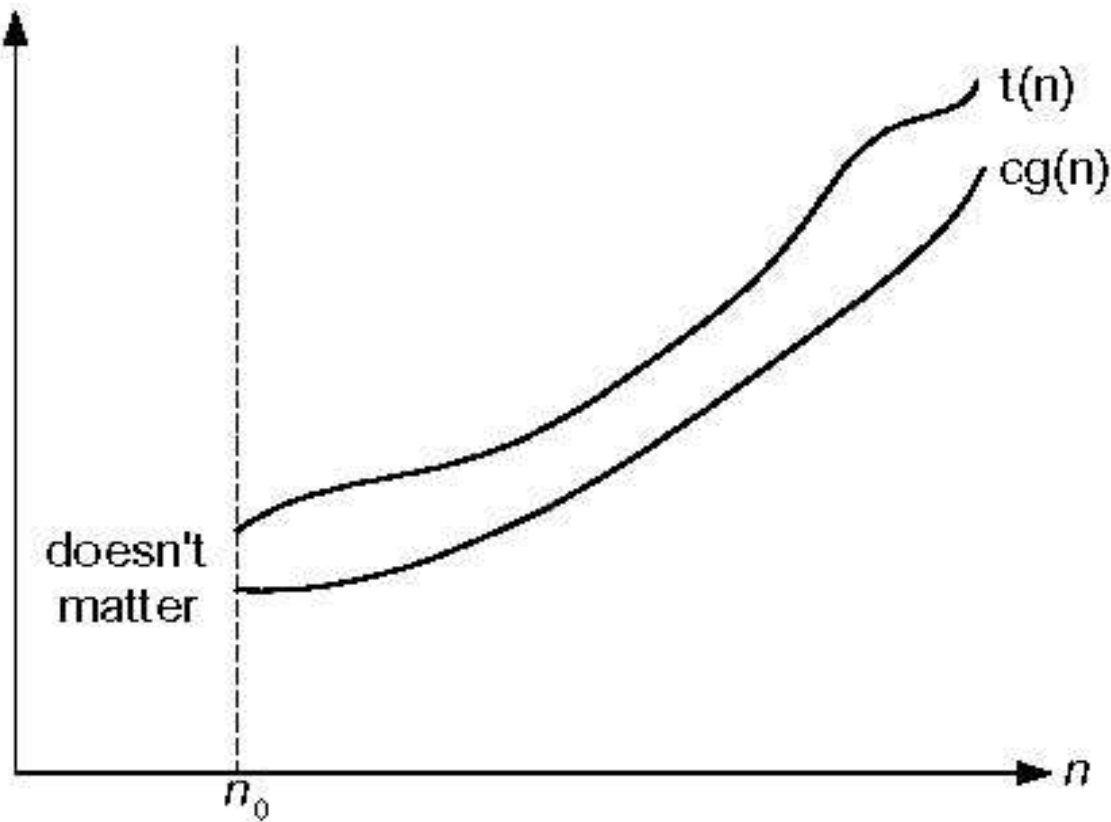
**Figure 2.1** Big-oh notation:  $t(n) \in O(g(n))$



# $\Omega$ -notation

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

# Big-



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

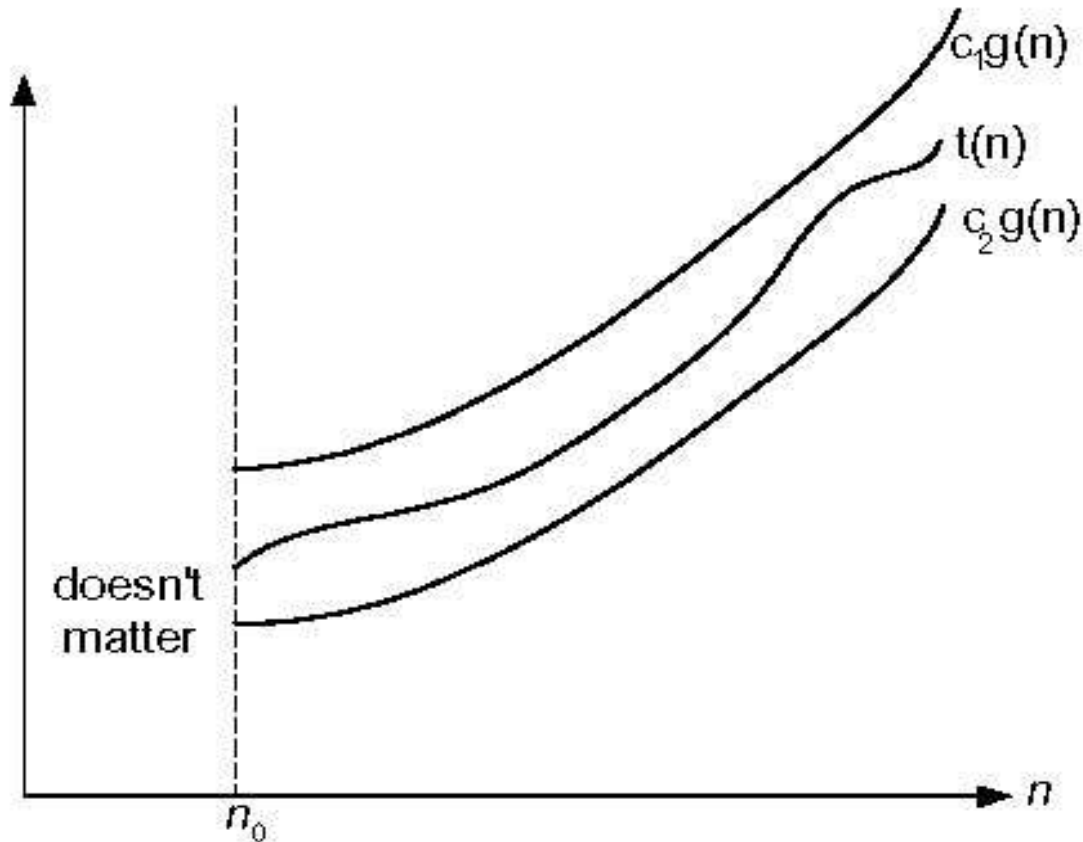


# Θ notation

## Formal definition

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

# Big-



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



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

- The analogous assertions are true for the  $\Omega$ -notation and  $\Theta$ -notation.

Proof. There exist constants  $c_1, c_2, n_1, n_2$  such that

$$t_1(n) \leq c_1 * g_1(n), \text{ for all } n \geq n_1$$

$$t_2(n) \leq c_2 * g_2(n), \text{ for all } n \geq n_2$$

**Define  $c_3 = c_1 + c_2$  and  $n_3 = \max\{n_1, n_2\}$ . Then**

$$t_1(n) + t_2(n) \leq c_3 * \max\{g_1(n), g_2(n)\}, \text{ for all } n \geq n_3$$

# Some properties of asymptotic order of growth



- $f(n) \in O(f(n))$
- $f(n) \in O(g(n))$  iff  $g(n) \in \Omega(f(n))$
- If  $f(n) \in O(g(n))$  and  $g(n) \in O(h(n))$ , then  $f(n) \in O(h(n))$

Note similarity with  $a \leq b$

- If  $f_1(n) \in O(g_1(n))$  and  $f_2(n) \in O(g_2(n))$ , then

$f_1(n) + f_2(n) \in O(\max\{g_1(n), g_2(n)\})$

# using limits



$$\lim_{n \rightarrow \infty} T(n)/g(n) = \begin{cases} 0 & \text{order of growth of } T(n) < \text{order of growth of } g(n) \\ c > 0 & \text{order of growth of } T(n) = \text{order of growth of } g(n) \\ \infty & \text{order of growth of } T(n) > \text{order of growth of } g(n) \end{cases}$$

# formula



L'Hôpital's rule: If  $\lim_{n \rightarrow \infty} f(n) = \lim_{n \rightarrow \infty} g(n) = \infty$  and the derivatives  $f'$ ,  $g'$  exist, then

$$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = \lim_{n \rightarrow \infty} \frac{f'(n)}{g'(n)}$$

Example:  $\log n$  vs.  $n$

Example:  $2^n$  vs.  $n!$

Stirling's formula:  $n! \approx (2\pi n)^{1/2} (n/e)^n$



# Orders of growth of some important functions



- All logarithmic functions  $\log_a n$  belong to the same class  $\Theta(\log n)$  no matter what the logarithm's base  $a > 1$  is

because  $\log_a n = \log_b n / \log_b a$

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

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

- Exponential functions  $a^n$  have different orders of growth for different  $a$ 's



# Basic asymptotic efficiency

classes

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

# Plan for analyzing nonrecursive algorithms



## General Plan for Analysis

- Decide on parameter  $n$  indicating input size
- Identify algorithm's basic operation
- Determine worst, average, and best cases for input of size  $n$
- Set up a sum for the number of times the basic operation is executed
- Simplify the sum using standard formulas and rules (see Appendix A)

# rules



$$\sum_{l \leq i \leq n} 1 = 1 + 1 + \dots + 1 = n - l + 1$$

In particular,  $\sum_{1 \leq i \leq n} 1 = n - 1 + 1 = n \in \Theta(n)$

$$\sum_{1 \leq i \leq n} i = 1 + 2 + \dots + n = n(n+1)/2 \approx n^2/2 \in \Theta(n^2)$$

$$\sum_{1 \leq i \leq n} i^2 = 1^2 + 2^2 + \dots + n^2 = n(n+1)(2n+1)/6 \approx n^3/3 \in \Theta(n^3)$$

$$\sum_{0 \leq i \leq n} a^i = 1 + a + \dots + a^n = (a^{n+1} - 1)/(a - 1) \text{ for any } a \neq 1$$

In particular,  $\sum_{0 \leq i \leq n} 2^i = 2^0 + 2^1 + \dots + 2^n = 2^{n+1} - 1 \in \Theta(2^n)$

$$\sum (a_i \pm b_i) = \sum a_i \pm \sum b_i \quad \sum c a_i = c \sum a_i \quad \sum_{l \leq i \leq u} a_i = \sum_{l \leq i \leq m} a_i + \sum_{m+1 \leq i \leq u} a_i$$

# element



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

//Determines the value of the largest element in a given array

//Input: An array  $A[0..n - 1]$  of real numbers

//Output: The value of the largest element in  $A$

*maxval*  $\leftarrow A[0]$

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

**if**  $A[i] > \textit{maxval}$

*maxval*  $\leftarrow A[i]$

**return** *maxval*

$T(n) = \sum_{1 \leq i \leq n-1} 1 = n-1 = \Theta(n)$  comparisons



# problem



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

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

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

//Output: Returns “true” if all the elements in  $A$  are distinct

//       and “false” otherwise

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

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

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

**return true**

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

$$= \sum_{0 \leq i \leq n-2} n-i-1 = (n-1+1)(n-1)/2$$

$$= \Theta(n^2) \text{ comparisons}$$



# Example 3: Matrix



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

//Multiplies two  $n$ -by- $n$  matrices by the definition-based algorithm

//Input: Two  $n$ -by- $n$  matrices  $A$  and  $B$

//Output: Matrix  $C = AB$

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

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

$C[i, j] \leftarrow 0.0$

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

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

**return**  $C$

$$\begin{aligned} T(n) &= \sum_{0 \leq i \leq n-1} \sum_{0 \leq j \leq n-1} n \\ &= \sum_{0 \leq i \leq n-1} \Theta(n^2) \\ &= \Theta(n^3) \text{ multiplications} \end{aligned}$$

# elimination



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

//Implements Gaussian elimination on an  $n$ -by- $(n+1)$  matrix  $A$

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

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

        for  $k \leftarrow i$  to  $n$  do

$A[j,k] \leftarrow A[j,k] - A[i,k] * A[j,i] / A[i,i]$

Find the efficiency class and a constant factor improvement.

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

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

$B \leftarrow A[j,i] / A[i,i]$

        for  $k \leftarrow i$  to  $n$  do

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

# digits



## **ALGORITHM** *Binary*( $n$ )

//Input: A positive decimal integer  $n$

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

$count \leftarrow 1$

**while**  $n > 1$  **do**

$count \leftarrow count + 1$

$n \leftarrow \lfloor n/2 \rfloor$

**return**  $count$

# Algorithms



- Decide on a parameter indicating an input's size.
- Identify the algorithm's basic operation.
- Check whether the number of times the basic op. is executed may vary on different inputs of the same size. (If it may, the worst, average, and best cases must be investigated separately.)
- Set up a recurrence relation with an appropriate initial condition expressing the number of times the basic op. is executed.
- Solve the recurrence (or, at the very least, establish its solution's order of growth) by backward substitutions or another method.

# of $n!$



Definition:  $n! = 1 * 2 * \dots * (n-1) * n$  for  $n \geq 1$  and  $0! = 1$

Recursive definition of  $n!$ :  $F(n) = F(n-1) * n$  for  $n \geq 1$  and  $F(0) = 1$

**ALGORITHM**  $F(n)$

//Computes  $n!$  recursively

//Input: A nonnegative integer  $n$

//Output: The value of  $n!$

**if**  $n = 0$  **return** 1

**else return**  $F(n - 1) * n$

Size:

Basic operation:

Recurrence relation:

$n$

multiplication

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

$M(0) = 0$

# Solving the recurrence for $M(n)$



$$M(n) \equiv M(n-1) + 1, \quad M(0) = 0$$

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

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

...

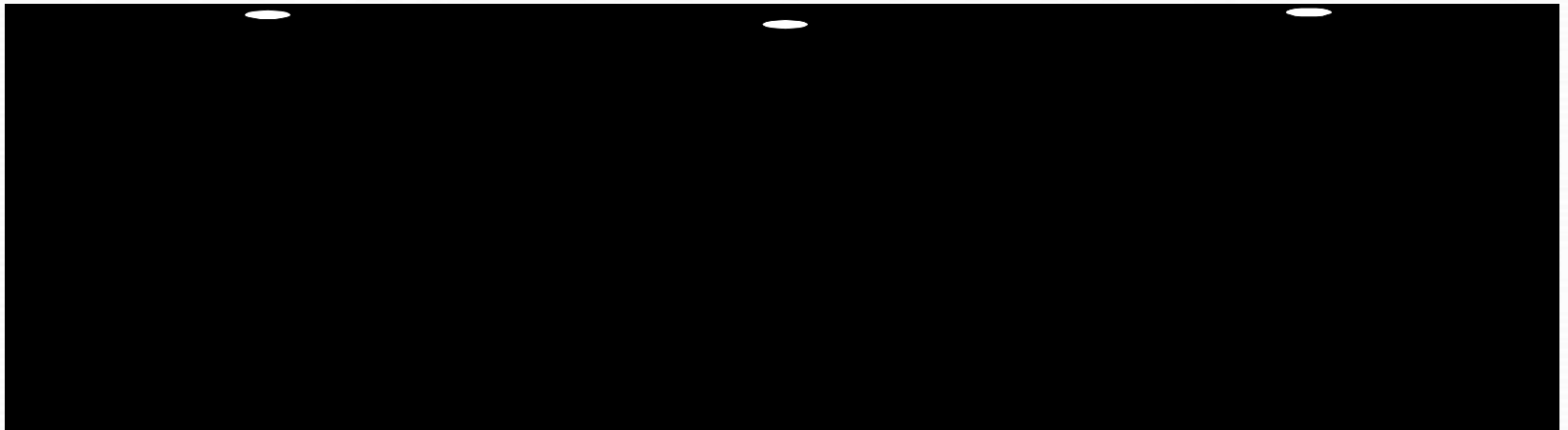
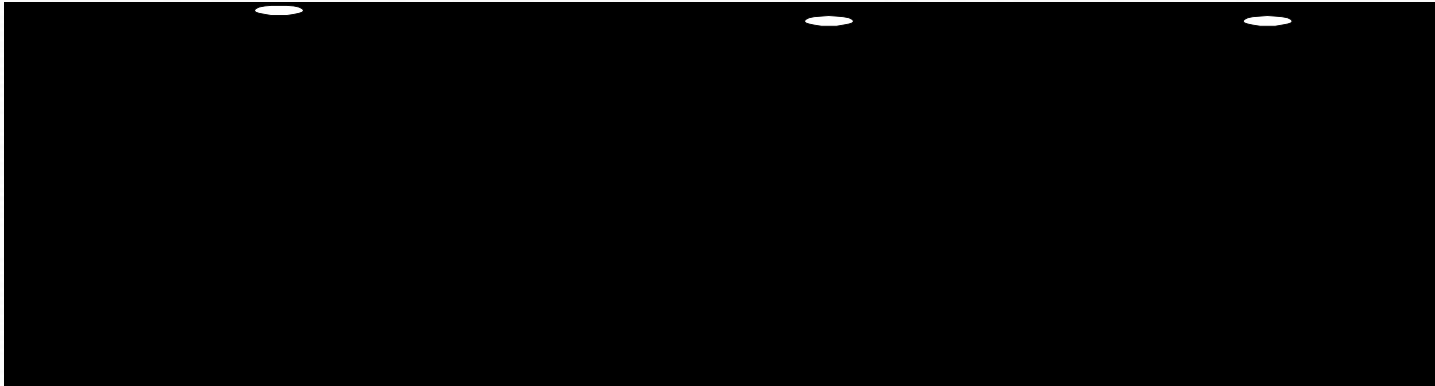
$$= M(n-i) + i$$

$$= M(0) + n$$

$$= n$$

The method is called **backward substitution**.

# Puzzle



Recurrence for number of moves:  $M(n) = 2M(n-1) + 1$



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

$$= 2(2M(n-2) + 1) + 1 = 2^2 * M(n-2) + 2^1 + 2^0$$

$$= 2^2 * (2M(n-3) + 1) + 2^1 + 2^0$$

$$= 2^3 * M(n-3) + 2^2 + 2^1 + 2^0$$

$$= \dots$$

$$= 2^{(n-1)} * M(1) + 2^{(n-2)} + \dots + 2^1 + 2^0$$

$$= 2^{(n-1)} + 2^{(n-2)} + \dots + 2^1 + 2^0$$

$$= 2^n - 1$$



## **[**ALGORITHM *BinRec*( $n$ )

//Input: A positive decimal integer  $n$

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

**if**  $n = 1$  **return** 1

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

$$A(n) = A(\lfloor n/2 \rfloor) + 1, \quad A(1) = 0$$

$$A(2^k) = A(2^{k-1}) + 1, \quad A(2^0) = 1 \quad (\text{using the Smoothness Rule})$$

$$= (A(2^{k-2}) + 1) + 1 = A(2^{k-2}) + 2$$

$$= A(2^{k-i}) + i$$

$$= A(2^{k-k}) + k = k + 0$$

$$= \log_2 n$$





# DIVIDE AND CONQUER



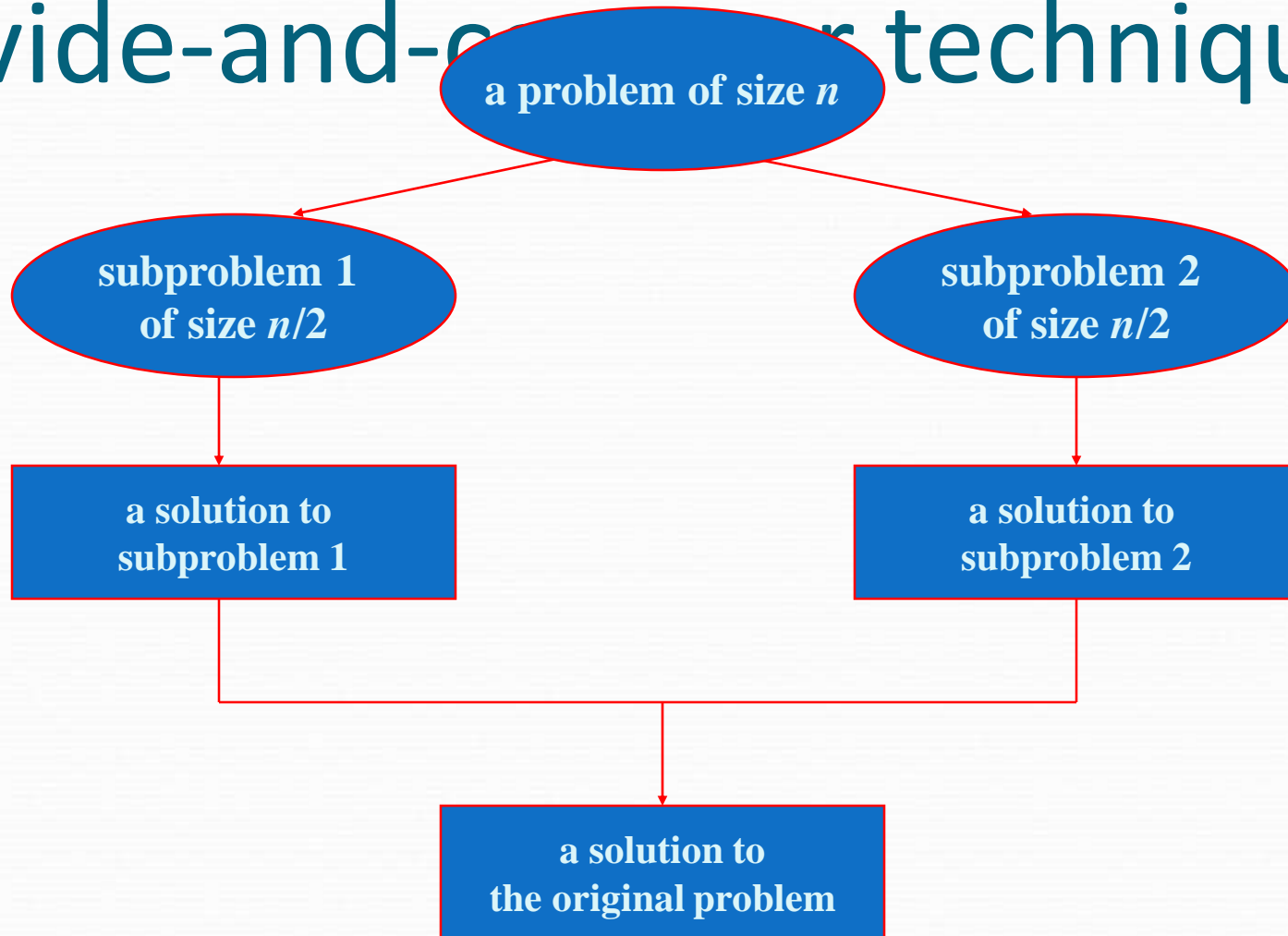
# Divide and Conquer

The most well known algorithm design strategy:

1. Divide instance of problem into two or more smaller instances
2. Solve smaller instances recursively
3. Obtain solution to original (larger) instance by combining these solutions



# Divide-and-conquer technique





# Divide and Conquer Examples

- Sorting: mergesort and quicksort
- Tree traversals
- Binary search
- Matrix multiplication-Strassen's algorithm

# General Divide and Conquer recurrence: Master Theorem



$$T(n) = aT(n/b) + f(n) \quad \text{where } f(n) \in \Theta(n^d)$$

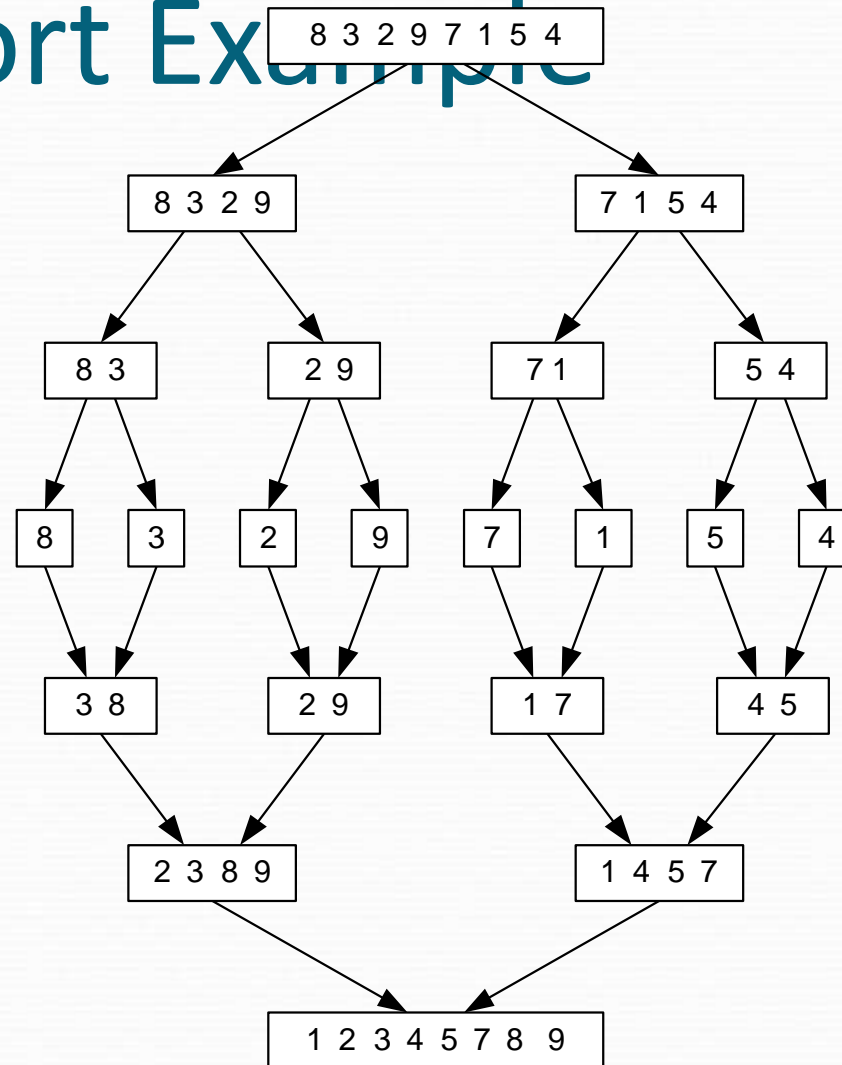
1.  $a < b^d$   $T(n) \in \Theta(n^d)$
2.  $a = b^d$   $T(n) \in \Theta(n^d \lg n)$
3.  $a > b^d$   $T(n) \in \Theta(n^{\log_b a})$
4. Note: the same results hold with  $O$  instead of  $\Theta$ .

# Mergesort

Algorithm:

- Split array  $A[1..n]$  in two and make copies of each half in arrays  $B[1.. n/2 ]$  and  $C[1.. n/2 ]$
- Sort arrays B and C
- Merge sorted arrays B and C into array A as follows:
  - Repeat the following until no elements remain in one of the arrays:
    - compare the first elements in the remaining unprocessed portions of the arrays
    - copy the smaller of the two into A, while incrementing the index indicating the unprocessed portion of that array
  - Once all elements in one of the arrays are processed,

# Mergesort Example



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

```

1  //Sorts array  $A[0..n - 1]$  by recursive mergesort
  //Input: An array  $A[0..n - 1]$  of orderable elements
  //Output: Array  $A[0..n - 1]$  sorted in nondecreasing order
  if  $n > 1$ 
    copy  $A[0..\lfloor n/2 \rfloor - 1]$  to  $B[0..\lfloor n/2 \rfloor - 1]$ 
    copy  $A[\lfloor n/2 \rfloor..n - 1]$  to  $C[0..\lfloor n/2 \rfloor - 1]$ 
    Mergesort( $B[0..\lfloor n/2 \rfloor - 1]$ )
    Mergesort( $C[0..\lfloor n/2 \rfloor - 1]$ )
    Merge( $B, C, A$ )
  ]
  
```

*Mergesort*( $C[0..\lfloor n/2 \rfloor - 1]$ )

*Merge*( $B, C, A$ )

**ALGORITHM** *Merge*( $B[0..p-1]$ ,  $C[0..q-1]$ ,  $A[0..p+q-1]$ )

//Merges two sorted arrays into one sorted array

//Input: Arrays  $B[0..p-1]$  and  $C[0..q-1]$  both sorted

//Output: Sorted array  $A[0..p+q-1]$  of the elements of  $B$  and  $C$

$i \leftarrow 0$ ;  $j \leftarrow 0$ ;  $k \leftarrow 0$

**while**  $i < p$  **and**  $j < q$  **do**

**if**  $B[i] \leq C[j]$

$A[k] \leftarrow B[i]$ ;  $i \leftarrow i + 1$

**else**  $A[k] \leftarrow C[j]$ ;  $j \leftarrow j + 1$

$k \leftarrow k + 1$

**if**  $i = p$

    copy  $C[j..q-1]$  to  $A[k..p+q-1]$

**else** copy  $B[i..p-1]$  to  $A[k..p+q-1]$

# Recurrence Relation for Mergesort



- Let  $T(n)$  be worst case time on a sequence of  $n$  keys
- If  $n = 1$ , then  $T(n) = \Theta(1)$  (constant)
- If  $n > 1$ , then  $T(n) = 2 T(n/2) + \Theta(n)$ 
  - two subproblems of size  $n/2$  each that are solved recursively
  - $\Theta(n)$  time to do the merge

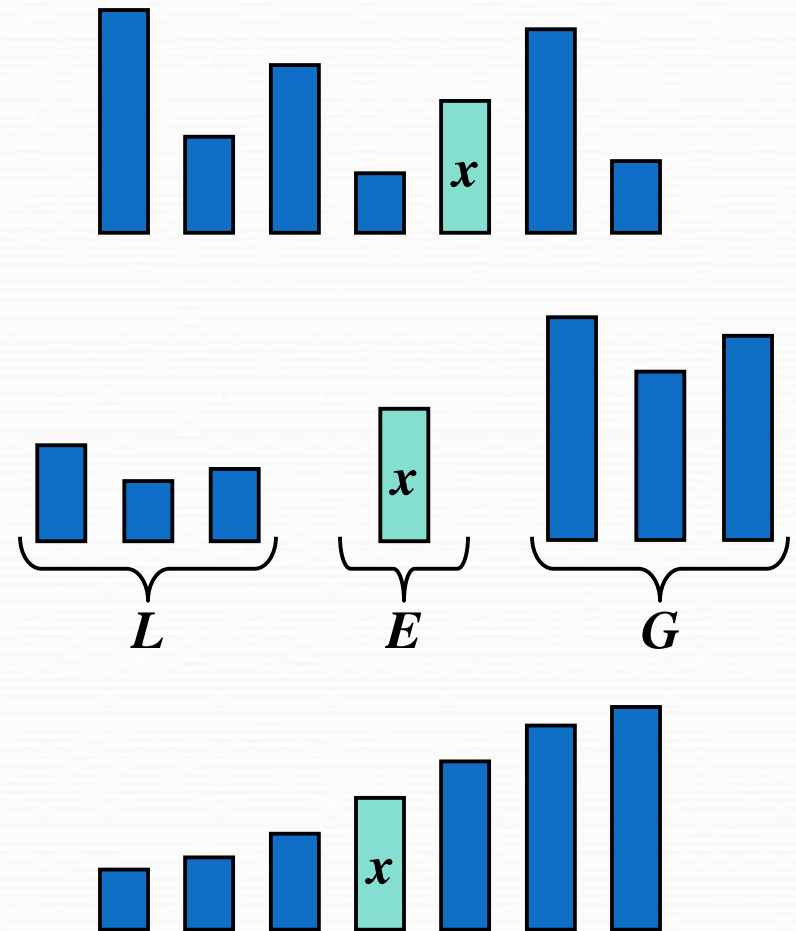


# Efficiency of mergesort

- All cases have same efficiency:  $\Theta( n \log n )$
- Number of comparisons is close to theoretical minimum for comparison-based sorting:
  - $\log n ! \approx n \lg n - 1.44 n$
- Space requirement:  $\Theta( n )$  (NOT in-place)
- Can be implemented without recursion (bottom-up)

# Quick-Sort

- Quick-sort is a randomized sorting algorithm based on the divide-and-conquer paradigm:
- Divide:** pick a random element  $x$  (called **pivot**) and partition  $S$  into
  - $L$  elements less than  $x$
  - $E$  elements equal  $x$
  - $G$  elements greater than  $x$
- Recur:** sort  $L$  and  $G$
- Conquer:** join  $L$ ,  $E$  and  $G$



# Quicksort

- Select a *pivot* (partitioning element)
- Rearrange the list so that all the elements in the positions before the pivot are smaller than or equal to the pivot and those after the pivot are larger than the pivot
- Exchange the pivot with the last element in the first (i.e., the last element in the first subarray)
- Sort the two sublists



$$A[i] \leq p$$

$$A[i] > p$$

# The partition algorithm

```

Algorithm Partition( $A[l..r]$ )
//Partitions a subarray by using its first element as a pivot
//Input: A subarray  $A[l..r]$  of  $A[0..n - 1]$ , defined by its left and right
//      indices  $l$  and  $r$  ( $l < r$ )
//Output: A partition of  $A[l..r]$ , with the split position returned as
//      this function's value
 $p \leftarrow A[l]$ 
 $i \leftarrow l; \quad j \leftarrow r + 1$ 
repeat
    repeat  $i \leftarrow i + 1$  until  $A[i] \geq p$ 
    repeat  $j \leftarrow j - 1$  until  $A[j] < p$ 
    swap( $A[i], A[j]$ )
until  $i \geq j$ 
swap( $A[i], A[j]$ ) //undo last swap when  $i \geq j$ 
swap( $A[l], A[j]$ )
return  $j$ 
    
```

# Efficiency of quicksort

- Best case: split in the middle —  $\Theta( n \log n )$
- Worst case: sorted array! —  $\Theta( n^2 )$
- Average case: random arrays —  $\Theta( n \log n )$
- Improvements:
  - better pivot selection: median of three partitioning avoids worst case in sorted files
  - switch to insertion sort on small subfiles

Considered the method of choice for internal sorting  
 Design and Analysis of Algorithms – Unit I  
 for large files ( $n \geq 10000$ )  
 Design and Analysis of Algorithms - Unit II

# Algorithm



Very efficient algorithm for searching in sorted array:

$K$  vs  $A[0] \dots A[m] \dots A[n-1]$

If  $K = A[m]$ , stop (successful search);

otherwise, continue searching by the same method  
in  $A[0..m-1]$  if  $K < A[m]$

and in  $A[m+1..n-1]$  if  $K > A[m]$



# Pseudocode for Binary Search

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

$l \leftarrow 0$ ;  $r \leftarrow n-1$

while  $l \leq r$  do

//  $l$  and  $r$  crosses over  $\rightarrow$  can't

find  $K$

$m \leftarrow \lfloor (l+r)/2 \rfloor$

if  $K = A[m]$  return  $m$

//the key is found

else if  $K < A[m]$   $r \leftarrow m-1$   
of

//the key is on the left half

the array

else  $l \leftarrow m+1$   
half of

// the key is on the right  
the array

# Binary Search – a Recursive



## Algorithm

ALGORITHM BinarySearchRecur( $A[o..n-1]$ ,  $l$ ,  $r$ ,  $K$ )

if  $l > r$

    return  $-1$

else

$m \leftarrow \lfloor (l + r) / 2 \rfloor$

    if  $K = A[m]$

        return  $m$

    else if  $K < A[m]$

        return BinarySearchRecur( $A[o..n-1]$ ,  $l$ ,  $m-1$ ,  $K$ )

else

    return BinarySearchRecur( $A[o..n-1]$ ,  $m+1$ ,  $r$ ,  $K$ )

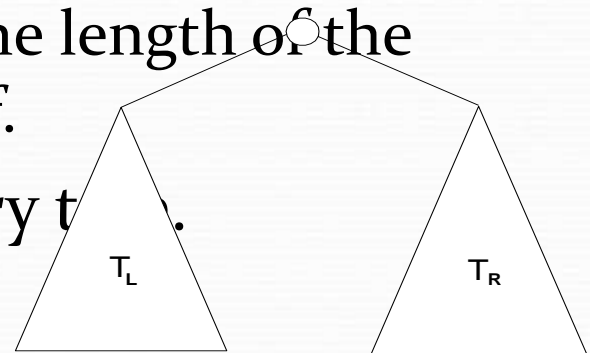
# Analysis of Binary Search



- Worst-case (successful or fail) :
  - $C_w(n) = 1 + C_w(\lfloor n/2 \rfloor),$
  - $C_w(1) = 1$   
solution:  $C_w(n) = \lfloor \log_2 n \rfloor + 1 = \lceil \log_2(n+1) \rceil$
- This is VERY fast: e.g.,  $C_w(10^6) = 20$
- Best-case:      successful  $C_b(n) = 1,$   
fail  $C_b(n) = \lfloor \log_2 n \rfloor + 1$
- Average-case: successful  $C_{avg}(n) = \log_2 n - 1$   
fail  $C_{avg}(n) = \log_2(n+1)$

# Binary Tree Traversals

- Definitions
  - A binary tree  $T$  is defined as a finite set of nodes that is either empty or consists of a root and two disjoint binary trees  $T_L$  and  $T_R$  called, respectively, the left and right subtree of the root.
  - The height of a tree is defined as the length of the longest path from the root to a leaf.
- Problem: find the height of a binary tree.



# Binary Tree



ALGORITHM Height( $T$ )

//Computes recursively the height of a binary tree

//Input: A binary tree  $T$

//Output: The height of  $T$

if  $T = \emptyset$

    return -1

else

    return  $\max\{\text{Height}(T_L), \text{Height}(T_R)\} + 1$



# Analysis:



Number of comparisons of a tree  $T$  with  $\emptyset$ :  $2n + 1$

Number of comparisons made to compute height is the same as number of additions:

$$A(n(T)) = A(n(T_L)) + A(n(T_R)) + 1 \text{ for } n > 0,$$

$$A(0) = 0$$

The solution is  $A(n) = n$

# Binary Tree Traversals– preorder, inorder, and postorder traversal



- Binary tree traversal: visit all nodes of a binary tree recursively.

Algorithm Preorder(T)

//Implement the preorder traversal of a binary tree

//Input: Binary tree T (with labeled vertices)

//Output: Node labels listed in preorder

if  $T \neq \emptyset$

    write label of T's root

    Preorder( $T_L$ )

    Preorder( $T_R$ )

# Integers



Consider the problem of multiplying two (large)  $n$ -digit integers represented by arrays of their digits such as:

$A = 12345678901357986429$     $B = 87654321284820912836$

The grade-school algorithm:

$$\begin{array}{cccc} a_1 & a_2 & \dots & a_n \\ b_1 & b_2 & \dots & b_n \end{array}$$

---

$$\begin{array}{ccccccc} & & (d_{10}) & d_{11} & d_{12} & \dots & d_{1n} \\ (d_{20}) & d_{21} & d_{22} & \dots & d_{2n} & & \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ (d_{n0}) & d_{n1} & d_{n2} & \dots & d_{nn} & & \end{array}$$

---

Efficiency:  $n^2$  one-digit multiplications



# First Divide-and-Conquer Algorithm



A small example:  $A * B$  where  $A = 2135$  and  $B = 4014$

$$A = (21 \cdot 10^2 + 35), \quad B = (40 \cdot 10^2 + 14)$$

$$\begin{aligned} \text{So, } A * B &= (21 \cdot 10^2 + 35) * (40 \cdot 10^2 + 14) \\ &= 21 * 40 \cdot 10^4 + (21 * 14 + 35 * 40) \cdot 10^2 + 35 * 14 \end{aligned}$$

In general, if  $A = A_1A_2$  and  $B = B_1B_2$  (where  $A$  and  $B$  are  $n$ -digit,  $A_1, A_2, B_1, B_2$  are  $n/2$ -digit numbers),

$$A * B = A_1 * B_1 \cdot 10^n + (A_1 * B_2 + A_2 * B_1) \cdot 10^{n/2} + A_2 * B_2$$

Recurrence for the number of one-digit multiplications  $M(n)$ :

$$M(n) = 4M(n/2), \quad M(1) = 1$$

Solution:  $M(n) = n^2$

# Second Divide-and-Conquer Algorithm



$$A * B = A_1 * B_1 \cdot 10^n + (A_1 * B_2 + A_2 * B_1) \cdot 10^{n/2} + A_2 * B_2$$

The idea is to decrease the number of multiplications from 4 to 3:

$$(A_1 + A_2) * (B_1 + B_2) = A_1 * B_1 + (A_1 * B_2 + A_2 * B_1) + A_2 * B_2,$$

$$\text{I.e., } (A_1 * B_2 + A_2 * B_1) = (A_1 + A_2) * (B_1 + B_2) - A_1 * B_1 - A_2 * B_2,$$

which requires only 3 multiplications at the expense of (4-1) extra add/sub.

Recurrence for the number of multiplications  $M(n)$ :

$$M(n) = 3M(n/2), \quad M(1) = 1$$

$$\text{Solution: } M(n) = 3^{\log_2 n} = n^{\log_2 3} \approx n^{1.585}$$

# Strassen's matrix multiplication

- Strassen observed [1969] that the product of two matrices can be computed as follows:

$$\begin{array}{ccc}
 \left( \begin{array}{cc} C_{00} & C_{01} \\ C_{10} & C_{11} \end{array} \right) & = & \left( \begin{array}{cc} A_{00} & A_{01} \\ A_{10} & A_{11} \end{array} \right) * \left( \begin{array}{cc} B_{00} & B_{01} \\ B_{10} & B_{11} \end{array} \right) \\
 & & = \left( \begin{array}{cc} M_1 + M_4 - M_5 + M_7 & M_3 + M_5 \\ M_2 + M_4 & M_1 + M_3 - M_2 + M_6 \end{array} \right)
 \end{array}$$

# Submatrices:

- $M_1 = (A_{oo} + A_{11}) * (B_{oo} + B_{11})$
- $M_2 = (A_{1o} + A_{11}) * B_{oo}$
- $M_3 = A_{oo} * (B_{o1} - B_{11})$
- $M_4 = A_{11} * (B_{1o} - B_{oo})$
- $M_5 = (A_{oo} + A_{o1}) * B_{11}$
- $M_6 = (A_{1o} - A_{oo}) * (B_{oo} + B_{o1})$

# Efficiency of Strassen's algorithm



- If  $n$  is not a power of 2, matrices can be padded with zeros
- Number of multiplications: 7
- Number of additions: 18





# Time Analysis

$$T(1) = 1 \quad (\text{assume } N = 2^k)$$

$$T(N) = 7T(N/2)$$

$$T(N) = 7^k T(N/2^k) = 7^k$$

$$T(N) = 7^{\log N} = N^{\log 7} = N^{2.81}$$

# Standard vs Strassen



|                 | N      | Multiplications       | Additions            |
|-----------------|--------|-----------------------|----------------------|
| Standard alg.   | 100    | 1,000,000             | 990,000              |
| Strassen's alg. | 100    | 411,822               | 2,470,334            |
| Standard alg.   | 1000   | 1,000,000,000         | 999,000,000          |
| Strassen's alg. | 1000   | 264,280,285           | 1,579,681,709        |
| Standard alg.   | 10,000 | $10^{12}$             | $9.99 \cdot 10^{11}$ |
| Strassen's alg. | 10,000 | $0.169 \cdot 10^{12}$ | $10^{12}$            |

# UNIT-V

## BRANCH AND BOUND

LECTURER:  
DHANANJAY

# Feasible Solution vs. Optimal Solution

- ⦿ DFS, BFS, hill climbing and best-first search can be used to solve some searching problem **for searching a feasible solution.**
- ⦿ However, they cannot be used to solve the optimization problems **for searching an (the) optimal solution.**

# The branch-and-bound strategy

- ◎ This strategy can be used to solve optimization problems without an exhaustive search in the average case.

# Branch-and-bound strategy

## ⦿ 2 mechanisms:

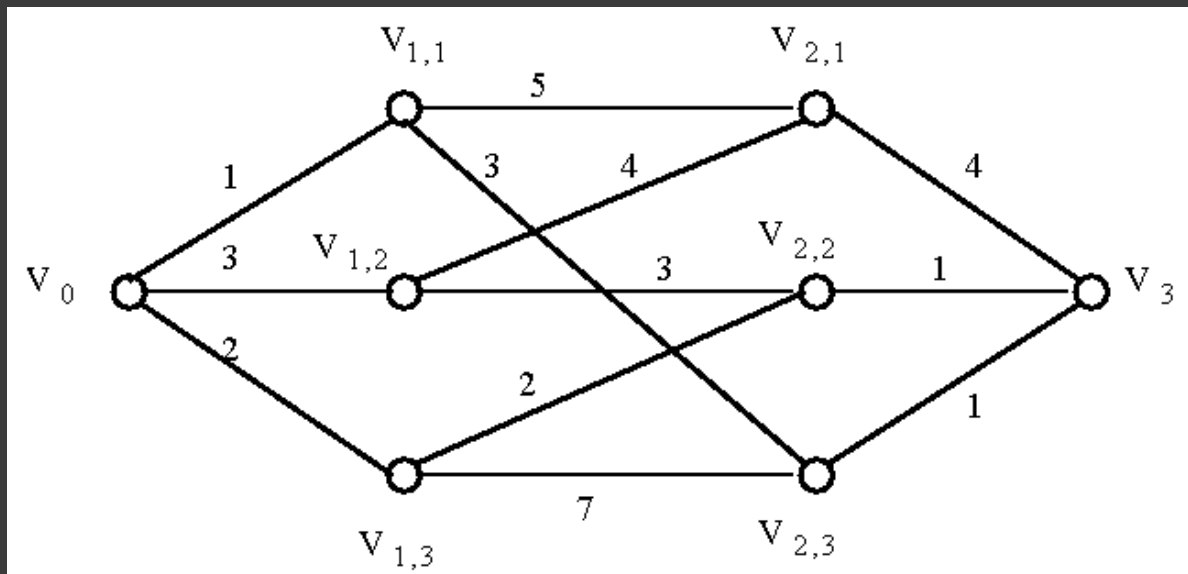
- A mechanism to generate branches when searching the solution space
- A mechanism to generate a bound so that many branches can be terminated

# Branch-and-bound strategy

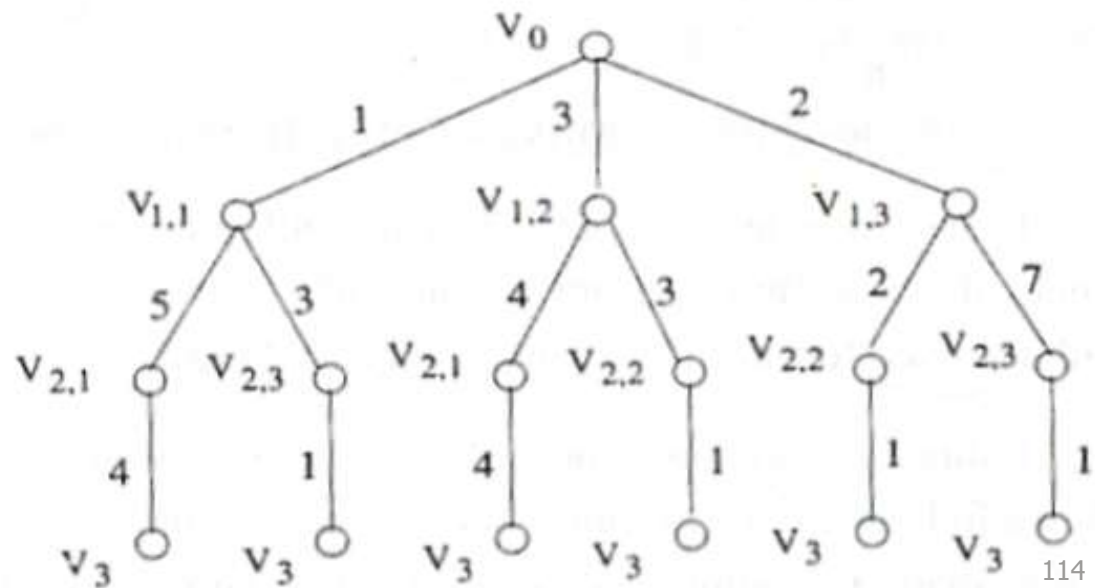
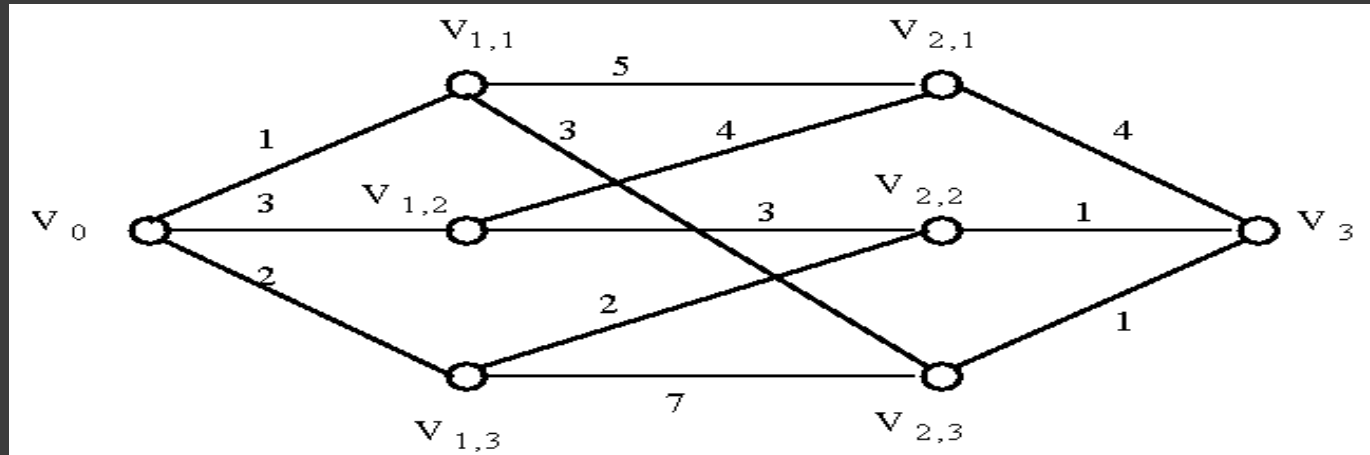
- ⦿ It is efficient **in the average case** because many branches can be terminated very early.
- ⦿ Although it is usually very efficient, a very large tree may be generated in the worst case.
- ⦿ Many NP-hard problem can be solved by B&B efficiently in the average case; however, **the worst case time complexity is still exponential.**

# A Multi-Stage Graph Searching Problem.

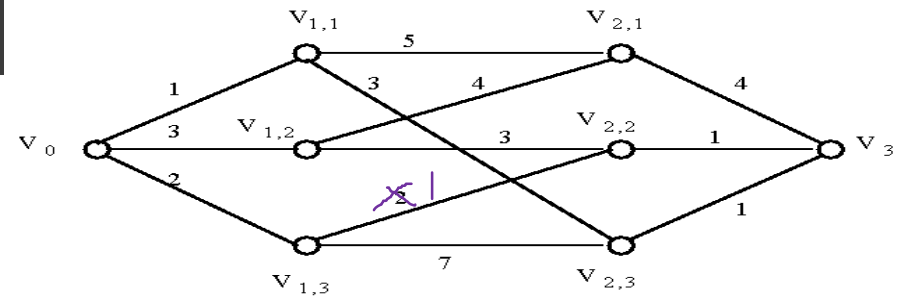
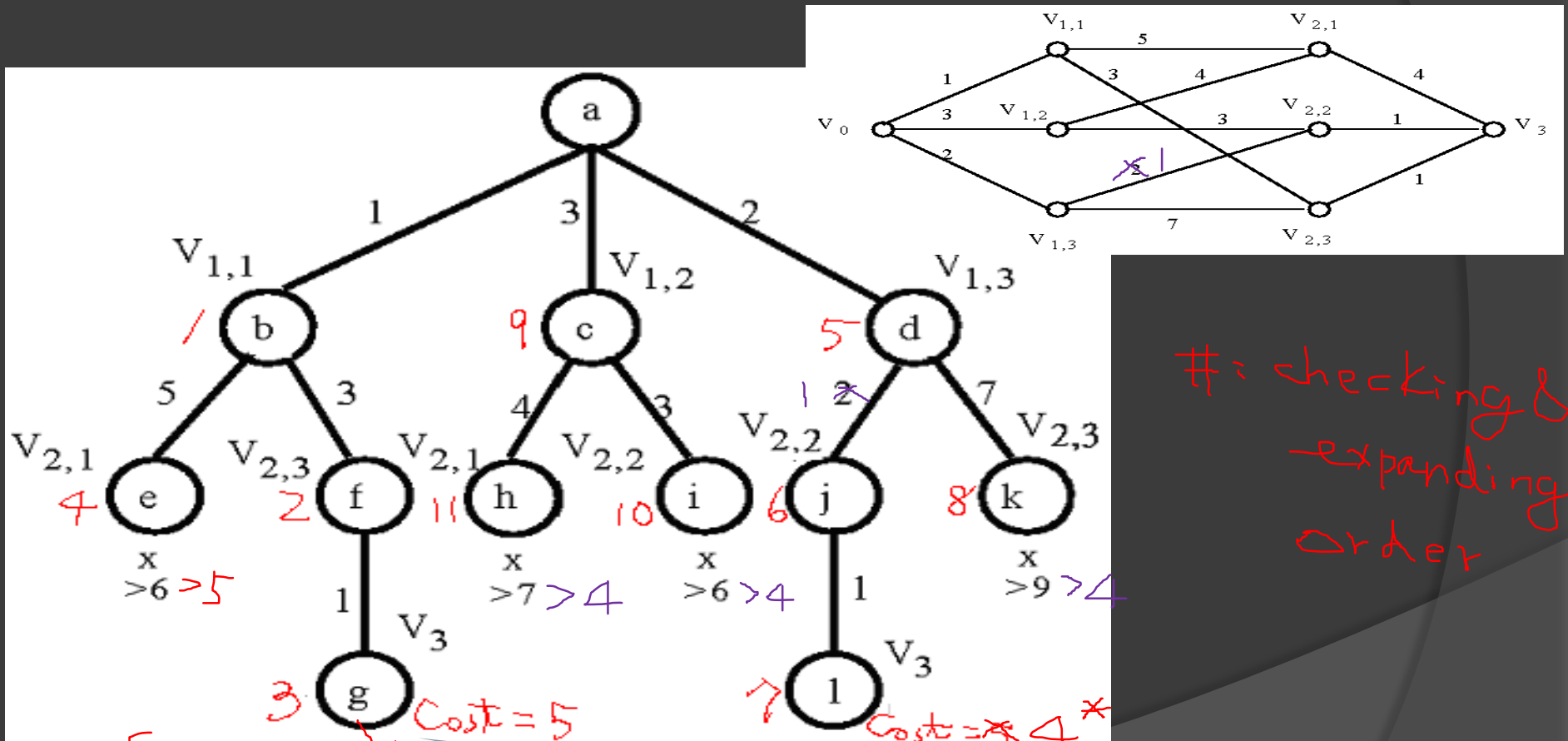
Find the shortest path from  $V_0$  to  $V_3$



## E.G.:A Multi-Stage Graph Searching Problem



# Solved by branch-and-bound (hill-climbing with bounds)



# = checking & expanding order

Feasible solution  
(upper bound)

A feasible solution is found whose cost is equal to 5.  
An **upper bound** of the optimal solution is first found here.

# For Minimization Problems

$\infty$

Upper Bound  
(for feasible solutions)

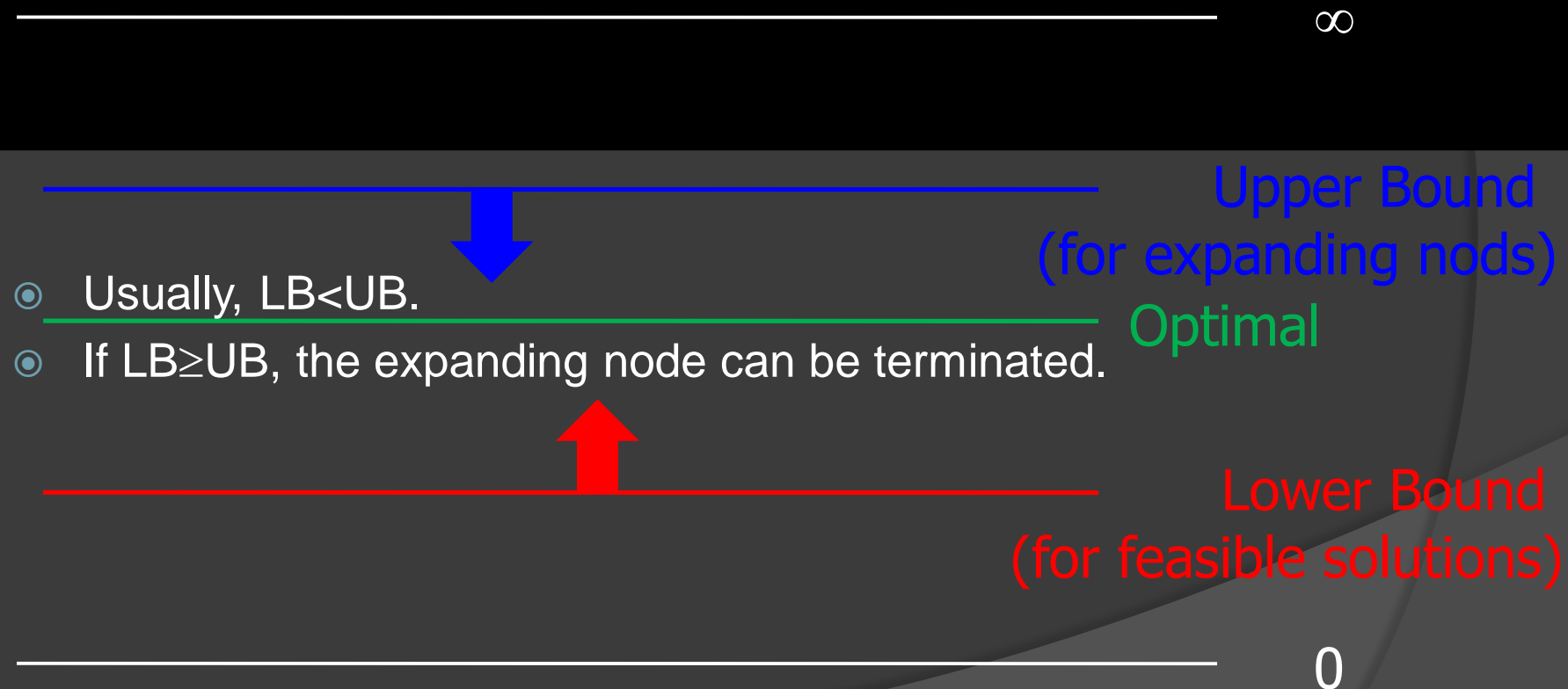
- Usually,  $LB < UB$ .
- If  $LB \geq UB$ , the expanding node can be terminated.

Optimal

Lower Bound  
(for expanding nodes)

0

# For Maximization Problems



# The traveling salesperson optimization problem

- Given a graph, the TSP Optimization problem is to find a tour, starting from any vertex, visiting every other vertex and returning to the starting vertex, with **minimal** cost.
- It is NP-hard.
- We try to avoid  $n!$  exhaustive search by the branch-and-bound technique on the average case. (Recall that  $O(n!) > O(2^n)$ .)

# The traveling salesperson optimization problem

- ◉ E.g. A Cost Matrix for a Traveling Salesperson Problem.

| $i \backslash j$ | 1        | 2        | 3        | 4        | 5        | 6        | 7        |
|------------------|----------|----------|----------|----------|----------|----------|----------|
| 1                | $\infty$ | 3        | 93       | 13       | 33       | 9        | 57       |
| 2                | 4        | $\infty$ | 77       | 42       | 21       | 16       | 34       |
| 3                | 45       | 17       | $\infty$ | 36       | 16       | 28       | 25       |
| 4                | 39       | 90       | 80       | $\infty$ | 56       | 7        | 91       |
| 5                | 28       | 46       | 88       | 33       | $\infty$ | 25       | 57       |
| 6                | 3        | 88       | 18       | 46       | 92       | $\infty$ | 7        |
| 7                | 44       | 26       | 33       | 27       | 84       | 39       | $\infty$ |

# The basic idea

- ⦿ There is a way to split the solution space (branch)
- ⦿ There is a way to predict a lower bound for a class of solutions. There is also a way to find an upper bound of an optimal solution. If the lower bound of a solution exceeds the upper bound, this solution cannot be optimal and thus we should terminate the branching associated with this solution.

# Splitting

- ⦿ We split a solution into two groups:
  - One group including a particular arc
  - The other excluding the arc
- ⦿ Each splitting incurs a lower bound and we shall traverse the searching tree with the “lower” lower bound.

# The traveling salesperson optimization problem

- The Cost Matrix for a Traveling Salesperson Problem.

Step 1 to reduce: Search each row for the smallest value

|        | i \ j | 1        | 2        | 3        | 4        | 5        | 6        | 7        | to j |
|--------|-------|----------|----------|----------|----------|----------|----------|----------|------|
|        |       |          |          |          |          |          |          |          |      |
| from i | 1     | $\infty$ | 3        | 93       | 13       | 33       | 9        | 57       |      |
|        | 2     | 4        | $\infty$ | 77       | 42       | 21       | 16       | 34       |      |
|        | 3     | 45       | 17       | $\infty$ | 36       | 16       | 28       | 25       |      |
|        | 4     | 39       | 90       | 80       | $\infty$ | 56       | 7        | 91       |      |
|        | 5     | 28       | 46       | 88       | 33       | $\infty$ | 25       | 57       |      |
|        | 6     | 3        | 88       | 18       | 46       | 92       | $\infty$ | 7        |      |
|        | 7     | 44       | 26       | 33       | 27       | 84       | 39       | $\infty$ |      |

# The traveling salesperson optimization problem

Step 2 to reduce: Search each column for the smallest value

## Reduced cost matrix:

| $i \backslash j$ | 1        | 2        | 3        | 4        | 5        | 6        | 7        |       |
|------------------|----------|----------|----------|----------|----------|----------|----------|-------|
| 1                | $\infty$ | 0        | 90       | 10       | 30       | 6        | 54       | (-3)  |
| 2                | 0        | $\infty$ | 73       | 38       | 17       | 12       | 30       | (-4)  |
| 3                | 29       | 1        | $\infty$ | 20       | 0        | 12       | 9        | (-16) |
| 4                | 32       | 83       | 73       | $\infty$ | 49       | 0        | 84       | (-7)  |
| 5                | 3        | 21       | 63       | 8        | $\infty$ | 0        | 32       | (-25) |
| 6                | 0        | 85       | 15       | 43       | 89       | $\infty$ | 4        | (-3)  |
| 7                | 18       | 0        | 7        | 1        | 58       | 13       | $\infty$ | (-26) |

reduced:84

A Reduced Cost Matrix.

# The traveling salesperson optimization problem

| j | 1        | 2        | 3        | 4        | 5        | 6        | 7        |
|---|----------|----------|----------|----------|----------|----------|----------|
| i |          |          |          |          |          |          |          |
| 1 | $\infty$ | 0        | 83       | 9        | 30       | 6        | 50       |
| 2 | 0        | $\infty$ | 66       | 37       | 17       | 12       | 26       |
| 3 | 29       | 1        | $\infty$ | 19       | 0        | 12       | 5        |
| 4 | 32       | 83       | 66       | $\infty$ | 49       | 0        | 80       |
| 5 | 3        | 21       | 56       | 7        | $\infty$ | 0        | 28       |
| 6 | 0        | 85       | 8        | 42       | 89       | $\infty$ | 0        |
| 7 | 18       | 0        | 0        | 0        | 58       | 13       | $\infty$ |

Table 6-5 Another Reduced Cost Matrix.

(-7)

(-1)

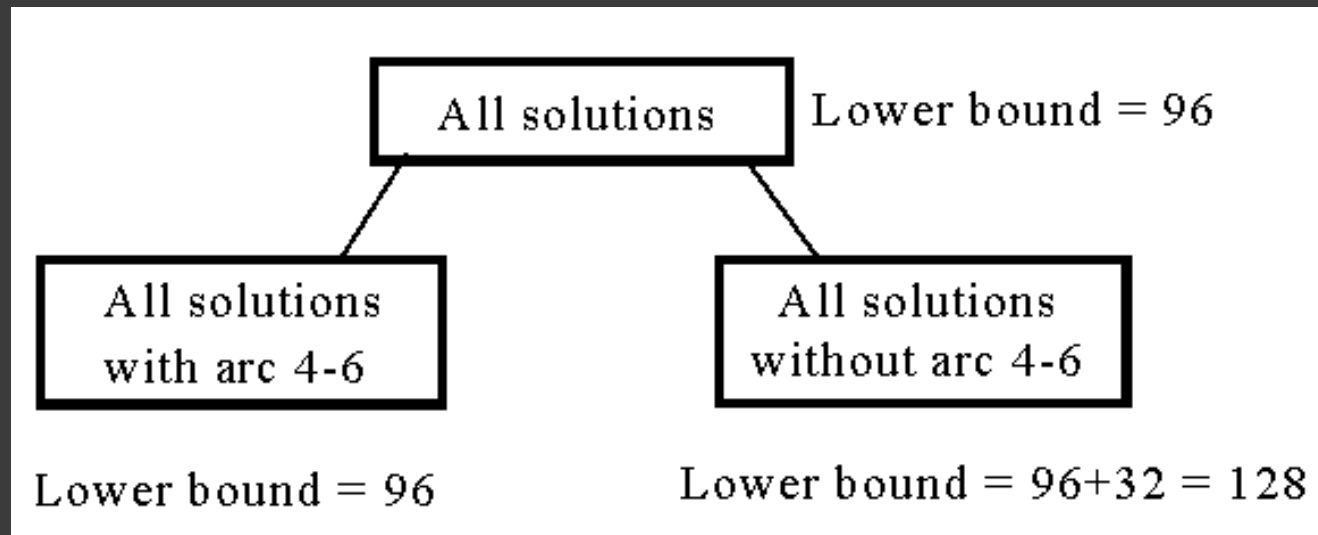
(-4)

# Lower bound

- The total cost of  $84+12=96$  is subtracted. Thus, we know the lower bound of feasible solutions to this TSP problem is 96.

# The traveling salesperson optimization problem

- Total cost reduced:  $84+7+1+4 = 96$  (lower bound)  
decision tree:



The Highest Level of a Decision Tree.

- If we use arc 3-5 to split, the difference on the lower bounds is  $17+1 = 18$ .

## Heuristic to select an arc to split the solution space

- If an arc of cost  $0(x)$  is selected, then the lower bound is added by  $0(x)$  when the arc is included.
- If an arc  $\langle i, j \rangle$  is not included, then the cost of the second smallest value ( $y$ ) in row  $i$  and the second smallest value ( $z$ ) in column  $j$  is added to the lower bound.
- Select the arc with the largest  $(y+z)-x$

We only have to set c4-6 to be  $\infty$ .  
For the right subtree  
(Arc 4-6 is excluded)

| j | 1        | 2        | 3        | 4        | 5        | 6        | 7        |
|---|----------|----------|----------|----------|----------|----------|----------|
| i |          |          |          |          |          |          |          |
| 1 | $\infty$ | 0        | 83       | 9        | 30       | 6        | 50       |
| 2 | 0        | $\infty$ | 66       | 37       | 17       | 12       | 26       |
| 3 | 29       | 1        | $\infty$ | 19       | 0        | 12       | 5        |
| 4 | 32       | 83       | 66       | $\infty$ | 49       | $\infty$ | 80       |
| 5 | 3        | 21       | 56       | 7        | $\infty$ | 0        | 28       |
| 6 | 0        | 85       | 8        | 42       | 89       | $\infty$ | 0        |
| 7 | 18       | 0        | 0        | 0        | 58       | 13       | $\infty$ |

# For the left subtree (Arc 4-6 is included)

| j<br>i | 1        | 2        | 3        | 4        | 5        | 7        |
|--------|----------|----------|----------|----------|----------|----------|
| 1      | $\infty$ | 0        | 83       | 9        | 30       | 50       |
| 2      | 0        | $\infty$ | 66       | 37       | 17       | 26       |
| 3      | 29       | 1        | $\infty$ | 19       | 0        | 5        |
| 5      | 3        | 21       | 56       | 7        | $\infty$ | 28       |
| 6      | 0        | 85       | 8        | $\infty$ | 89       | 0        |
| 7      | 18       | 0        | 0        | 0        | 58       | $\infty$ |

A Reduced Cost Matrix if Arc 4-6 is included.

1. 4<sup>th</sup> row is deleted.
2. 6<sup>th</sup> column is deleted.
3. We must set  $c_{6-4}$  to be  $\infty$ . (The reason will be clear later.)

# For the left subtree

- The cost matrix for all solution with arc 4-6:

| j<br>i | 1        | 2        | 3        | 4        | 5        | 7        |
|--------|----------|----------|----------|----------|----------|----------|
| 1      | $\infty$ | 0        | 83       | 9        | 30       | 50       |
| 2      | 0        | $\infty$ | 66       | 37       | 17       | 26       |
| 3      | 29       | 1        | $\infty$ | 19       | 0        | 5        |
| 5      | 0        | 18       | 53       | 4        | $\infty$ | 25 (-3)  |
| 6      | 0        | 85       | 8        | $\infty$ | 89       | 0        |
| 7      | 18       | 0        | 0        | 0        | 58       | $\infty$ |

A Reduced Cost Matrix for that in Table 6-6.

- Total cost reduced:  $96+3 = 99$  (new lower bound)

# Upper bound

- ⦿ We follow the best-first search scheme and can obtain a feasible solution with cost  $C$ .
- ⦿  $C$  serves as an upper bound of the optimal solution and many branches may be terminated if their lower bounds are equal to or larger than  $C$ .

⊕: expanding order (selecting order)  
 #: adding order

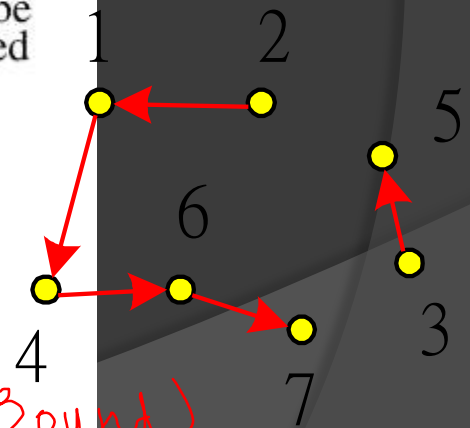
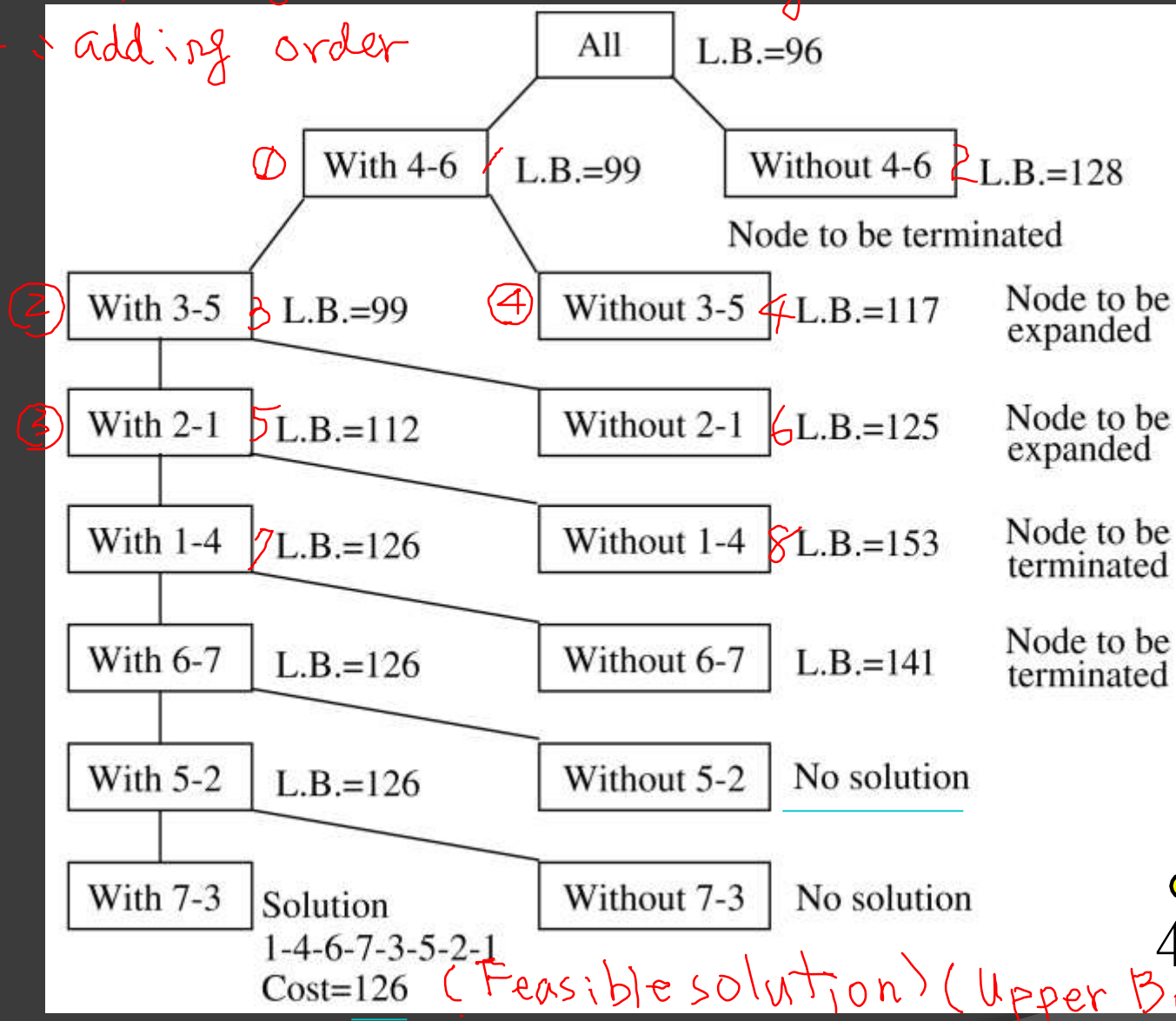


Fig 6-26 A Branch-and-Bound Solution of a Traveling Salesperson Problem.

# Preventing an arc

- ⦿ In general, if paths  $i_1-i_2-\dots-i_m$  and  $j_1-j_2-\dots-j_n$  have already been included and a path from  $i_m$  to  $j_1$  is to be added, then path from  $j_n$  to  $i_1$  must be prevented (by assigning the cost of  $j_n$  to  $i_1$  to be  $\infty$ )
- ⦿ For example, if 4-6, 2-1 are included and 1-4 is to be added, we must prevent 6-2 from being used by setting  $c_{6-2}=\infty$ . If 6-2 is used, there will be a loop which is forbidden.

# The 0/1 knapsack problem

- Positive integer  $P_1, P_2, \dots, P_n$  (profit)  
 $W_1, W_2, \dots, W_n$  (weight)  
 $M$  (capacity)

$$\begin{aligned} &\text{maximize} \quad \sum_{i=1}^n P_i X_i \\ &\text{subject to} \quad \sum_{i=1}^n W_i X_i \leq M \quad X_i = 0 \text{ or } 1, i = 1, \dots, n. \end{aligned}$$

The problem is modified:

$$\text{minimize} \quad - \sum_{i=1}^n P_i X_i$$

# The 0/1 knapsack problem

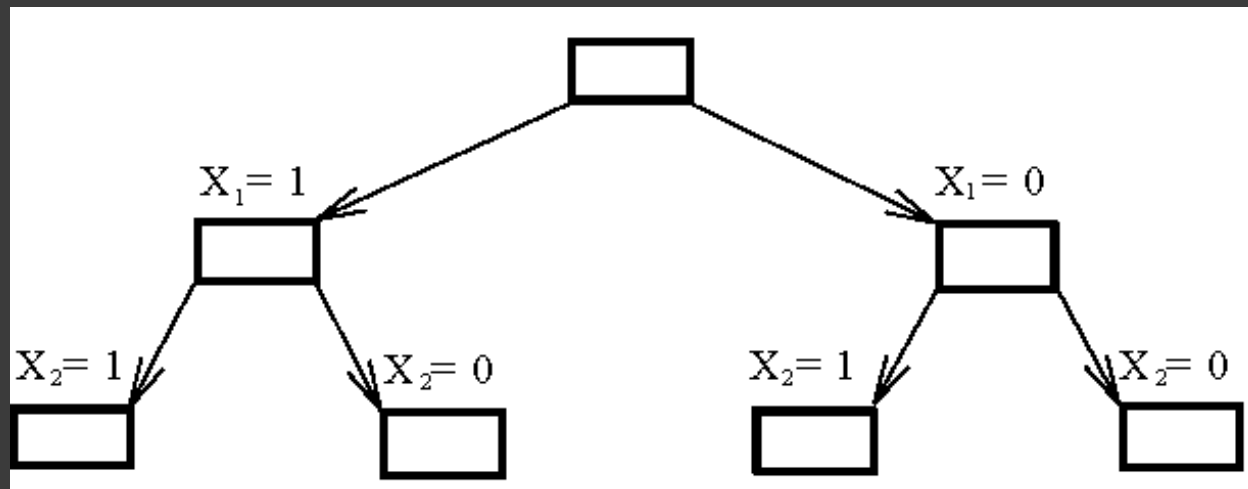


Fig. 6-27 The Branching Mechanism in the Branch-and-Bound Strategy to Solve 0/1 Knapsack Problem.

# How to find the upper bound?

- Ans: by quickly finding a feasible solution in a **greedy manner**: starting from the smallest available  $i$ , scanning towards the largest  $i$ 's until  $M$  is exceeded. The upper bound can be calculated.

# The 0/1 knapsack problem

- E.g.  $n = 6$ ,  $M = 34$

|       |    |    |   |    |    |   |
|-------|----|----|---|----|----|---|
| $i$   | 1  | 2  | 3 | 4  | 5  | 6 |
| $P_i$ | 6  | 10 | 4 | 5  | 6  | 4 |
| $W_i$ | 10 | 19 | 8 | 10 | 12 | 8 |

$$(P_i/W_i \geq P_{i+1}/W_{i+1})$$

- A feasible solution:  $X_1 = 1$ ,  $X_2 = 1$ ,  $X_3 = 0$ ,  $X_4 = 0$ ,  
 $X_5 = 0$ ,  $X_6 = 0$   
 $-(P_1 + P_2) = -16$  (upper bound)  
Any solution higher than -16 can not be an optimal solution.

# How to find the lower bound?

- Ans: by relaxing our restriction from  $X_i = 0$  or  $1$  to  $0 \leq X_i \leq 1$  (knapsack problem)

Let  $-\sum_{i=1}^n P_i X_i$  be an optimal solution for 0/1

knapsack problem and  $-\sum_{i=1}^n P_i X'_i$  be an optimal

solution for **fractional knapsack problem**. Let

$$Y = -\sum_{i=1}^n P_i X_i, \quad Y' = -\sum_{i=1}^n P_i X'_i.$$

$$\Rightarrow Y' \leq Y$$

# The knapsack problem

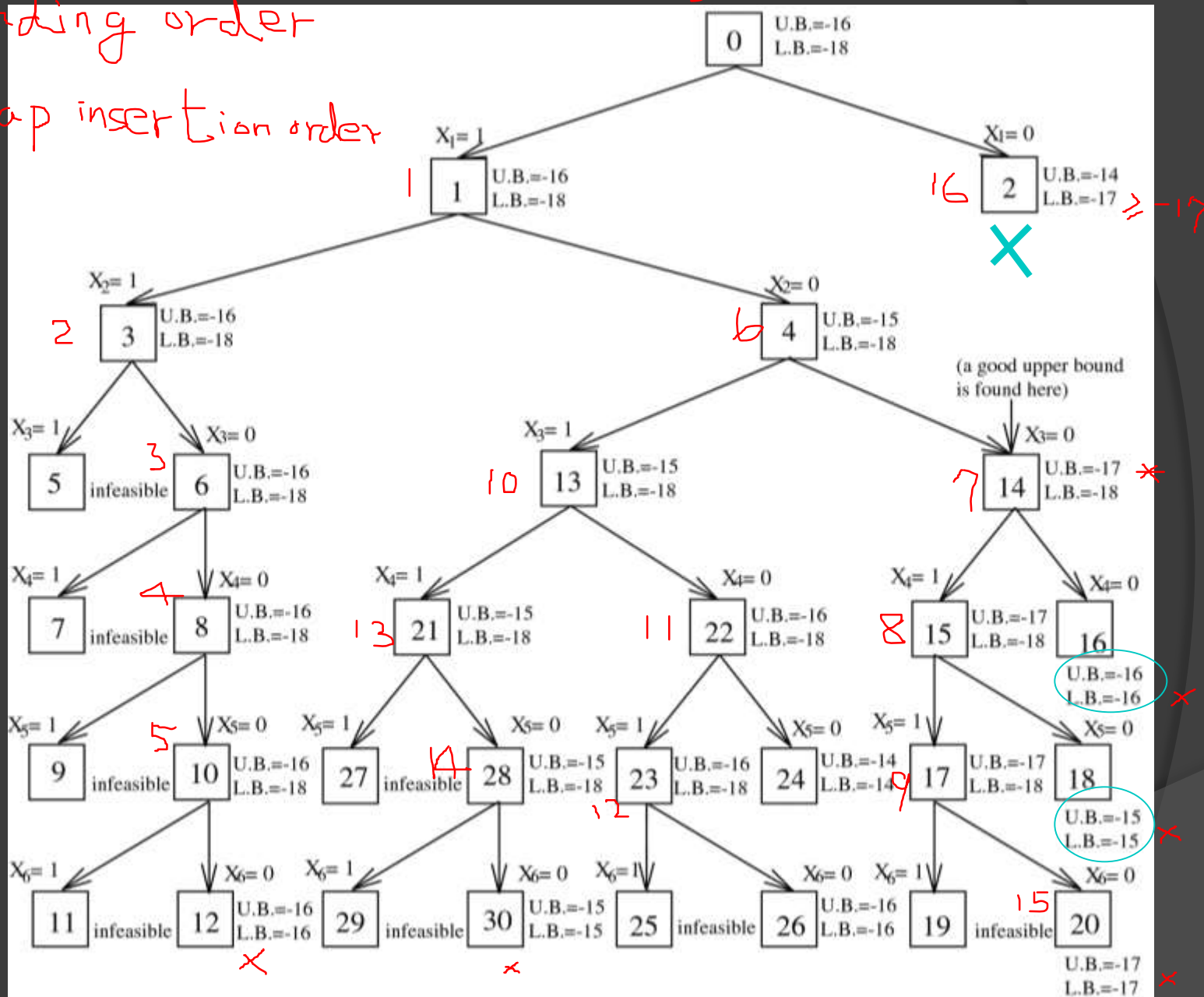
- We can use the greedy method to find an optimal solution for knapsack problem.
- For example, for the state of  $X_1=1$  and  $X_2=1$ , we have  
 $X_1 = 1, X_2 = 1, X_3 = (34-6-10)/8=5/8, X_4 = 0, X_5 = 0, X_6 = 0$   
 $-(P_1+P_2+5/8P_3) = -18.5$  (lower bound)  
-18 is our lower bound. (We only consider integers, since the benefits of a 0/1 knapsack problem will be integers.)

# How to expand the tree?

- ⦿ By the best-first search scheme
- ⦿ That is, by expanding the node with the best lower bound. If two nodes have the same lower bounds, expand the node with the lower upper bound.

#: expanding order

# = heap insertion order



0/1 Knapsack Problem Solved by Branch-and-Bound Strategy

- ⦿ Node 2 is terminated because its lower bound is equal to the upper bound of node 14.
- ⦿ Nodes 16, 18 and others are terminated because the local lower bound is equal to the local upper bound.  
(lower bound  $\leq$  optimal solution  $\leq$  upper bound)

# The A\* algorithm

- Used to solve optimization problems.
- Using the **best-first strategy**.
- If a feasible solution (goal node) **is selected to expand**, then **it is optimal** and we can stop.
- Estimated cost function of a node  $n$  :  $f(n)$

$$f(n) = g(n) + h(n)$$

$g(n)$ : cost from root to node  $n$ .

$h(n)$ : estimated cost from node  $n$  to a goal node.

$h^*(n)$ : “real” cost from node  $n$  to a goal node.

$f^*(n)$ : “real” cost of node  $n$

$$h(n) \leq h^*(n)$$

$$\Rightarrow f(n) = g(n) + h(n) \leq g(n) + h^*(n)$$

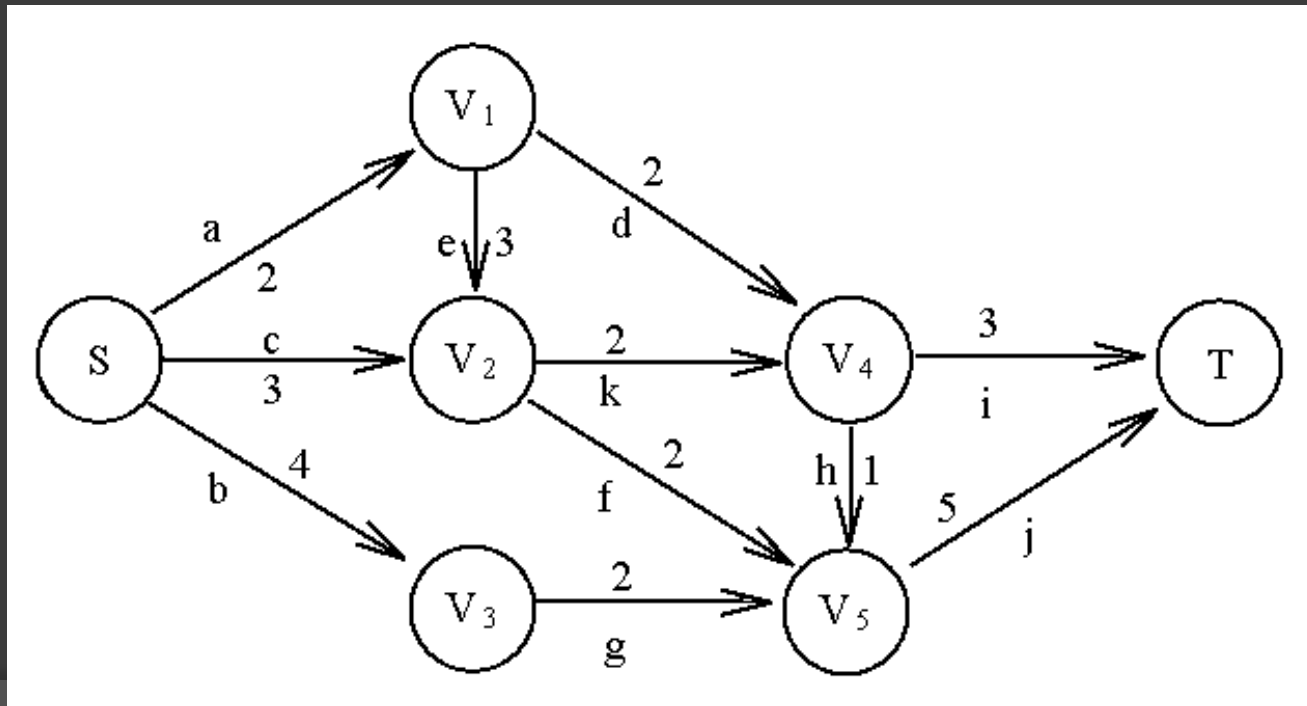
**Estimated further cost should never exceed the real further cost.**

# Reasoning

- Let  $t$  be the selected goal node. We have  $f^*(t) = f(t) + h(t) = f(t) + 0 = f(t) \dots (2)$
- Assume that  $t$  is not the optimal node.** There must exist one node, say  $s$ , that has been generated but not selected and that will lead to the optimal node.
- Since we take the best first search strategy, we have  $f(t) \leq f(s) \dots (3)$ .
- We have  $f^*(t) = f(t) \leq f(s) \leq f^*(s)$  by Eqs. (1), (2) and (3), which means that  $s$  is not the node leading to the optimal node. **Contradiction occurs.**
- Therefore,  $t$  is the optimal node.**

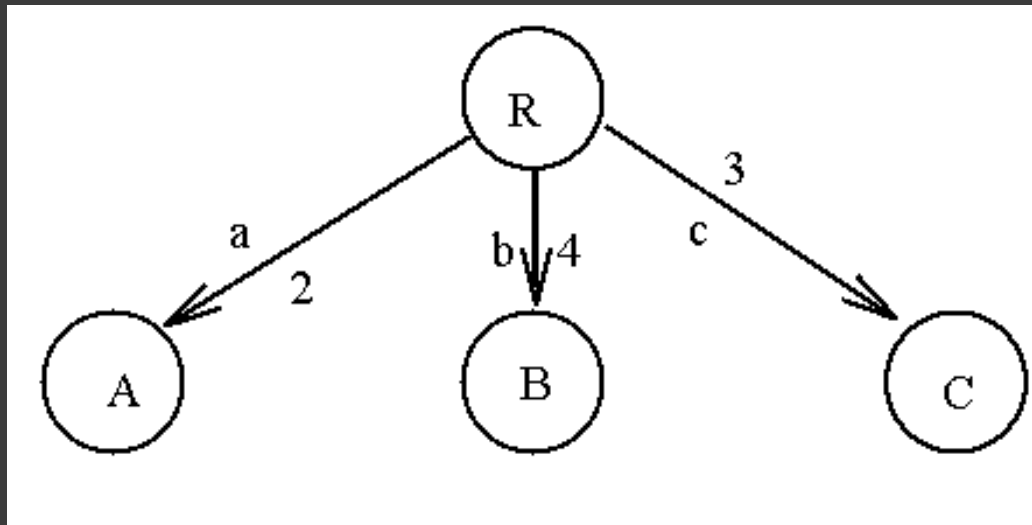
# The A\* algorithm

- Stop when the selected node is also a goal node. It is optimal iff  $h(n) \leq h^*(n)$
- E.g.: To find a shortest path from node s to node t



# The A\* algorithm

## Step 1.



$$g(A)=2$$

$$g(B)=4$$

$$g(C)=3$$

$$h(A)=\min\{2,3\}=2$$

$$h(B)=\min\{2\}=2$$

$$h(C)=\min\{2,2\}=2$$

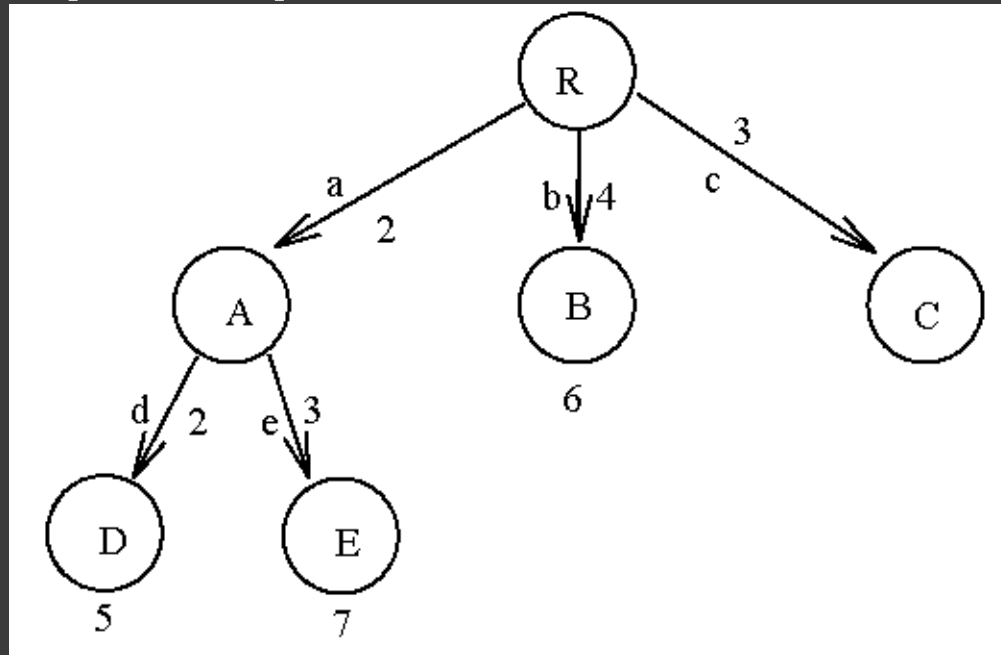
$$f(A)=2+2=4$$

$$f(B)=4+2=6$$

$$f(C)=3+2=5$$

# The A\* algorithm

## Step 2. Expand A



$$g(D)=2+2=4$$

$$h(D)=\min\{3,1\}=1$$

$$f(D)=4+1=5$$

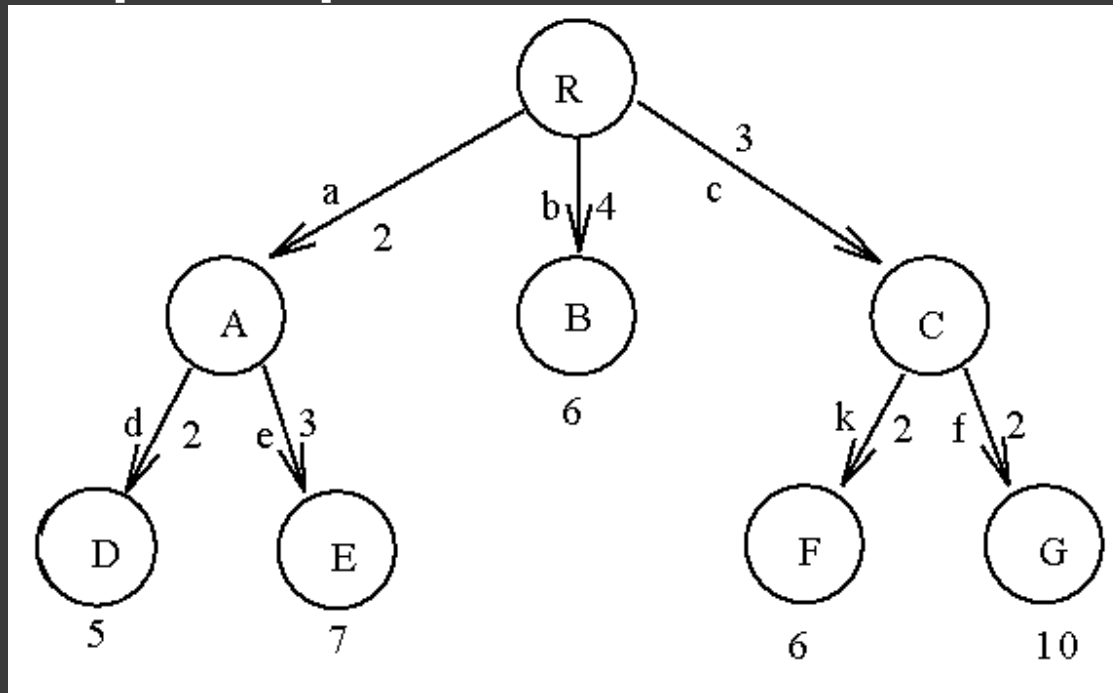
$$g(E)=2+3=5$$

$$h(E)=\min\{2,2\}=2$$

$$f(E)=5+2=7$$

# The A\* algorithm

## Step 3. Expand C



$$g(F)=3+2=5$$

$$h(F)=\min\{3,1\}=1$$

$$f(F)=5+1=6$$

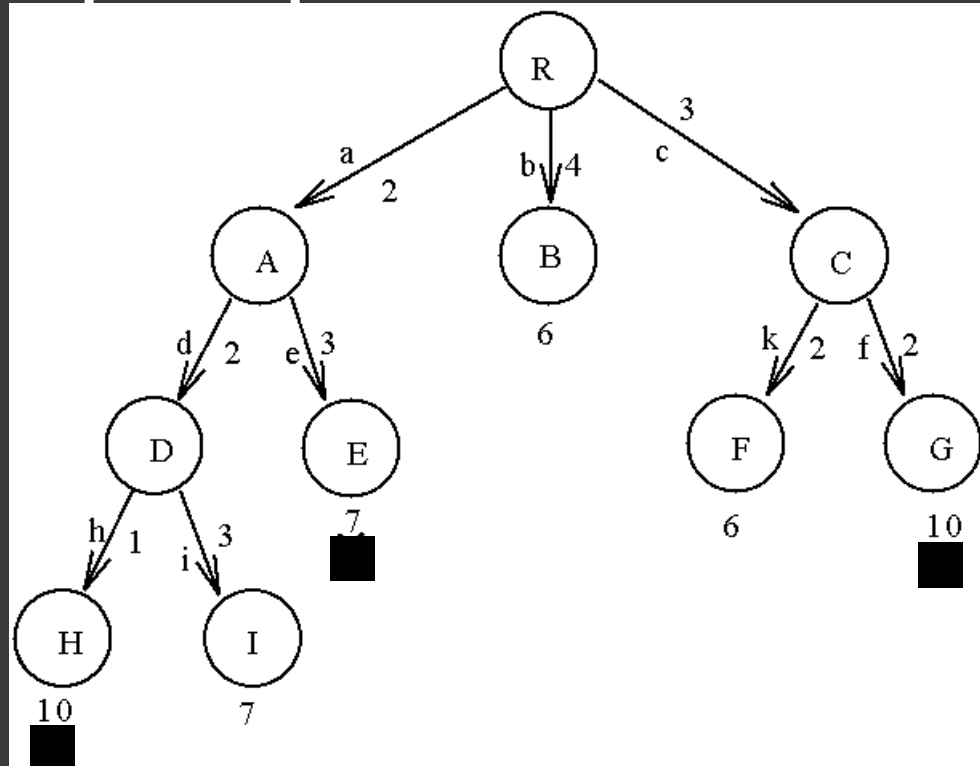
$$g(G)=3+2=5$$

$$h(G)=\min\{5\}=5$$

$$f(G)=5+5=10$$

# The A\* algorithm

## Step 4. Expand D



$$g(H) = 2 + 2 + 1 = 5$$

$$g(I) = 2 + 2 + 3 = 7$$

$$h(H) = \min\{5\} = 5$$

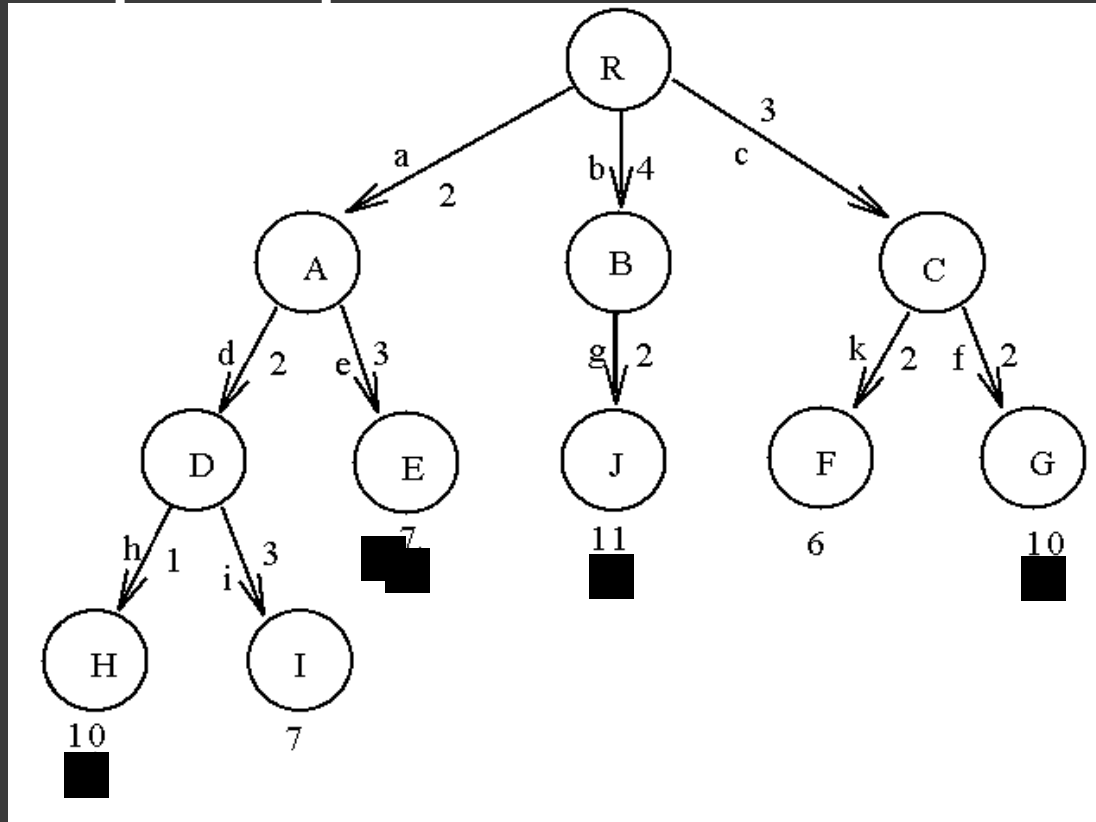
$$h(I) = 0$$

$$f(H) = 5 + 5 = 10$$

$$f(I) = 7 + 0 = 7$$

# The A\* algorithm

## Step 5. Expand B



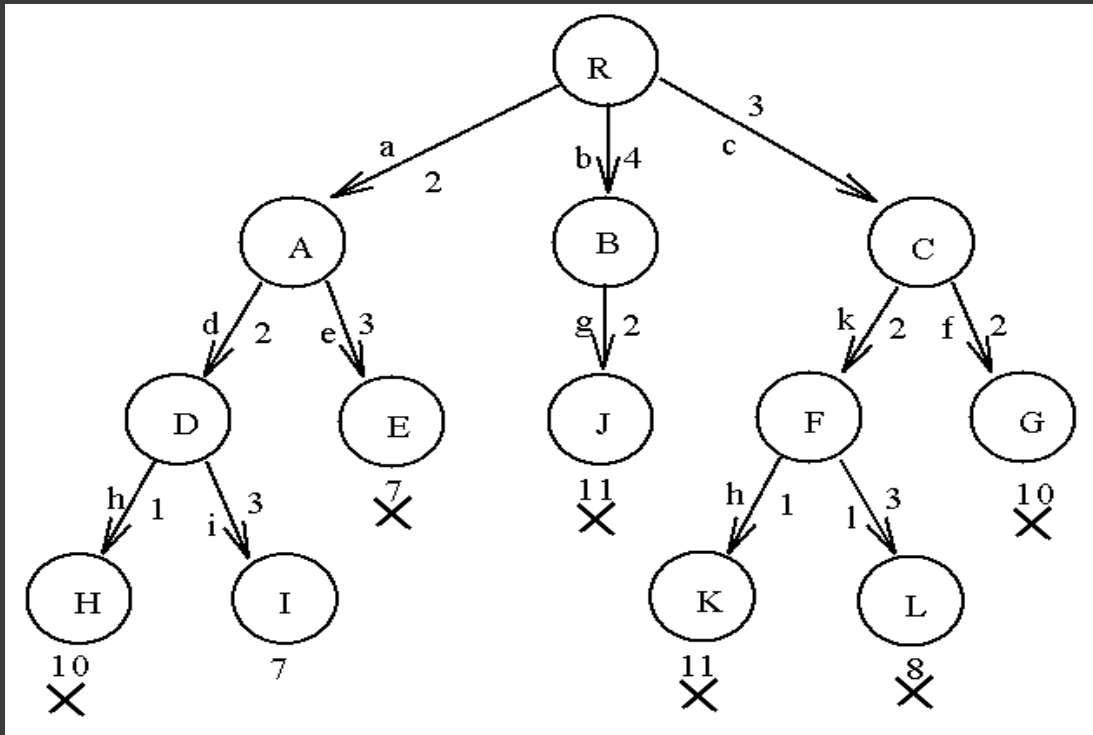
$$g(J) = 4 + 2 = 6$$

$$h(J) = \min\{5\} = 5$$

$$f(J) = 6 + 5 = 11$$

# The A\* algorithm

## Step 6. Expand F



I is selected to expand.  
The A\* algorithm stops,  
since I is a goal node.

$$g(K) = 3 + 2 + 1 = 6$$

$$h(K) = \min\{5\} = 5$$

$$f(K) = 6 + 5 = 11$$

$$g(L) = 3 + 2 + 3 = 8$$

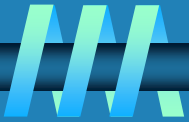
$$h(L) = 0$$

$$f(L) = 8 + 0 = 8$$

# The A\* Algorithm

- ⦿ Can be considered as a special type of branch-and-bound algorithm.
- ⦿ When the first feasible solution is found, all nodes in the heap (priority queue) are terminated.
- ⦿ \* stands for "real"
- ⦿ "A\* algorithm" stands for "real good algorithm"

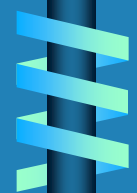
# UNIT – III



## DYNAMIC PROGRAMMING

Lecturer:

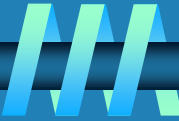
**Dhananjay**



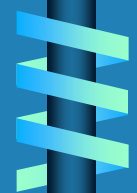
# Introduction

- ❧ **Dynamic programming is a technique for solving problems with overlapping sub-problems.**
- ❧ **Typically, these sub-problems arise from a recurrence relating a solution to a given problem with solutions to its smaller sub-problems of the same type.**

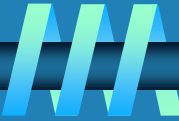
# Introduction



- ⌘ Rather than solving overlapping sub-problems again and again,
- ⌘ dynamic programming suggests solving each of the smaller sub-problems only once
- ⌘ and recording the results in a table from which we can then obtain a solution to the original problem.

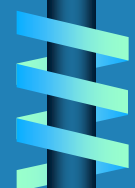


# Dynamic Programming

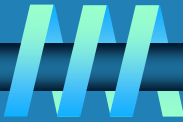


*Dynamic Programming* is a general algorithm design technique for solving problems defined by or formulated as recurrences with overlapping subinstances

- Invented by American mathematician Richard Bellman in the 1950s to solve optimization problems
- “Programming” here means “planning”
- Main idea:
  - set up a recurrence relating a solution to a larger instance to solutions of some smaller instances
  - solve smaller instances once
  - record solutions in a table
  - extract solution to the initial instance from that table



# Example: Fibonacci numbers



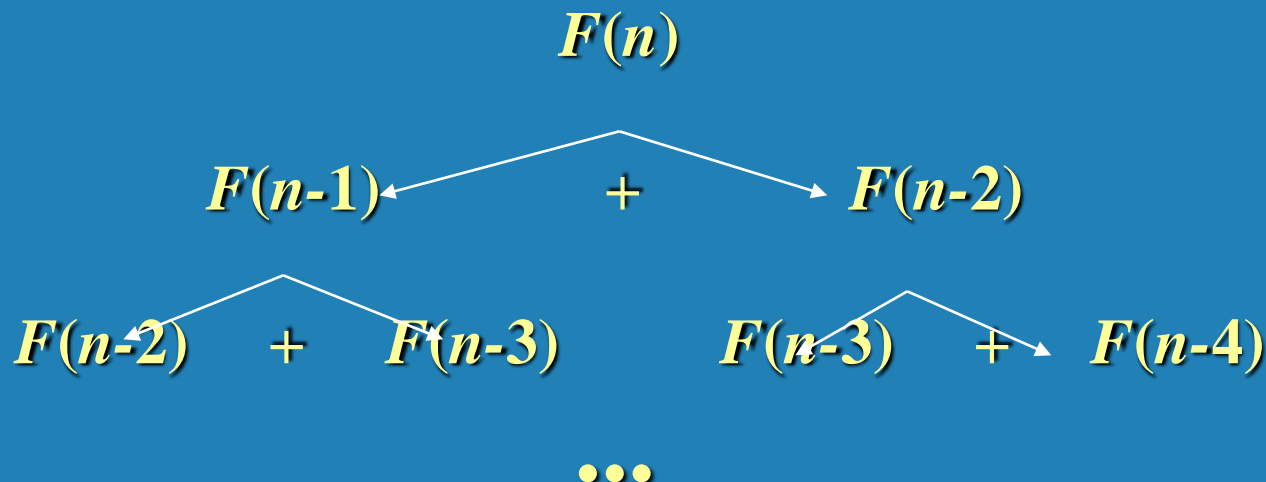
- Recall definition of Fibonacci numbers:

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

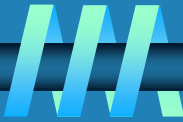
$$F(0) = 0$$

$$F(1) = 1$$

- Computing the  $n^{\text{th}}$  Fibonacci number recursively (top-down):



# Example: Fibonacci numbers (cont.)



Computing the  $n^{\text{th}}$  Fibonacci number using bottom-up iteration and recording results:

$$F(0) = 0$$

$$F(1) = 1$$

$$F(2) = 1 + 0 = 1$$

...

$$F(n-2) =$$

$$F(n-1) =$$

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

|   |   |   |       |          |          |        |
|---|---|---|-------|----------|----------|--------|
| 0 | 1 | 1 | . . . | $F(n-2)$ | $F(n-1)$ | $F(n)$ |
|---|---|---|-------|----------|----------|--------|

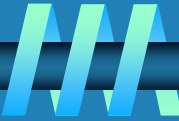
Efficiency:

- time  $n$
- space  $n$

What if we solve  
it recursively?



# Introduction



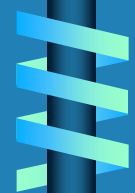
- ❧ The Fibonacci numbers are the elements of the sequence  
 $0, 1, 1, 2, 3, 5, 8, 13, 21, 34, \dots$ ,

Algorithm fib(n)

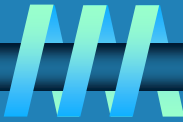
if  $n = 0$  or  $n = 1$  return 1

return fib( $n - 1$ ) + fib( $n - 2$ )

- ❧ The original problem  $F(n)$  is defined by  $F(n-1)$  and  $F(n-2)$

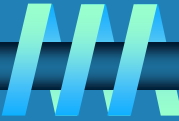


# Introduction



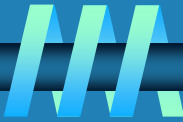
- Notice that if we call, say,  $\text{fib}(5)$ , we produce a call tree that calls the function on the same value many different times:
- $\text{fib}(5)$
- $\text{fib}(4) + \text{fib}(3)$
- $(\text{fib}(3) + \text{fib}(2)) + (\text{fib}(2) + \text{fib}(1))$
- $((\text{fib}(2) + \text{fib}(1)) + (\text{fib}(1) + \text{fib}(0))) + ((\text{fib}(1) + \text{fib}(0)) + \text{fib}(1))$
- $((((\text{fib}(1) + \text{fib}(0)) + \text{fib}(1)) + (\text{fib}(1) + \text{fib}(0)))) + ((\text{fib}(1) + \text{fib}(0)) + \text{fib}(1))$
- If we try to use recurrence directly to compute the  $n^{\text{th}}$  Fibonacci number  $F(n)$ , we would have to recompute the same values of this function many times

# Introduction



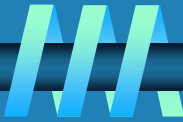
- ❧ Certain algorithms compute the  $n^{\text{th}}$  Fibonacci number without computing all the preceding elements of this sequence.
- ❧ It is typical of an algorithm based on the classic bottom-up dynamic programming approach,
- ❧ A top-down variation of it exploits so-called memory functions
- ❧ The crucial step in designing such an algorithm remains the same => Deriving a recurrence relating a solution to the problem's instance with solutions of its smaller (and overlapping) subinstances.

# Introduction



- ❧ **Dynamic programming usually takes one of two approaches:**
- ❧ **Bottom-up approach:** All subproblems that might be needed are solved in advance and then used to build up solutions to larger problems. This approach is slightly better in stack space and number of function calls, but it is sometimes not intuitive to figure out all the subproblems needed for solving the given problem.
- ❧ **Top-down approach:** The problem is broken into subproblems, and these subproblems are solved and the solutions remembered, in case they need to be solved again. This is recursion and Memory Function combined together.

# Bottom Up



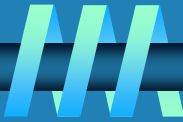
- ⌚ In the bottom-up approach we calculate the smaller values of Fibo first, then build larger values from them. This method also uses linear ( $O(n)$ ) time since it contains a loop that repeats  $n - 1$  times.

Algorithm Fibo(n)

```
a = 0, b = 1
repeat n - 1 times
    c = a + b
    a = b
    b = c
return b
```

- ⌚ In both these examples, we only calculate fib(2) one time, and then use it to calculate both fib(4) and fib(3), instead of computing it every time either of them is evaluated.

# Top-Down



- ⌚ suppose we have a simple map object,  $m$ , which maps each value of Fibo that has already been calculated to its result, and we modify our function to use it and update it. The resulting function requires only  $O(n)$  time instead of exponential time:

```
m [0] = 0
```

```
m [1] = 1
```

```
Algorithm Fibo(n)
```

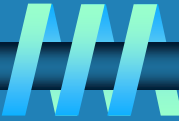
```
  if map  $m$  does not contain key  $n$ 
```

```
     $m[n] := \text{Fibo}(n - 1) + \text{Fibo}(n - 2)$ 
```

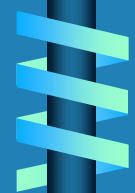
```
  return  $m[n]$ 
```

- ⌚ This technique of saving values that have already been calculated is called Memory Function; this is the top-down approach, since we first break the problem into subproblems and then calculate and store values

# Examples of DP algorithms



- Computing a binomial coefficient
- Longest common subsequence
- Warshall's algorithm for transitive closure
- Floyd's algorithm for all-pairs shortest paths
- Constructing an optimal binary search tree
- Some instances of difficult discrete optimization problems:
  - traveling salesman
  - knapsack



# Computing a binomial coefficient by DP

Binomial coefficients are coefficients of the binomial formula:

$$(a + b)^n = C(n,0)a^n b^0 + \dots + C(n,k)a^{n-k}b^k + \dots + C(n,n)a^0 b^n$$

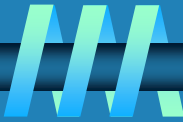
Recurrence:  $C(n,k) = C(n-1,k) + C(n-1,k-1)$  for  $n > k > 0$

$$C(n,0) = 1, \quad C(n,n) = 1 \quad \text{for } n \geq 0$$

Value of  $C(n,k)$  can be computed by filling a table:

|       | 0 | 1 | 2 | ... | $k-1$        | $k$        |
|-------|---|---|---|-----|--------------|------------|
| 0     | 1 |   |   |     |              |            |
| 1     | 1 | 1 |   |     |              |            |
| .     |   |   |   |     |              |            |
| .     |   |   |   |     |              |            |
| .     |   |   |   |     |              |            |
| $n-1$ |   |   |   |     | $C(n-1,k-1)$ | $C(n-1,k)$ |
| $n$   |   |   |   |     |              | $C(n,k)$   |

# Computing $C(n, k)$ : pseudocode and analysis



**ALGORITHM** *Binomial*( $n, k$ )

//Computes  $C(n, k)$  by the dynamic programming algorithm

//Input: A pair of nonnegative integers  $n \geq k \geq 0$

//Output: The value of  $C(n, k)$

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

**for**  $j \leftarrow 0$  **to**  $\min(i, k)$  **do**

**if**  $j = 0$  **or**  $j = i$

$C[i, j] \leftarrow 1$

**else**  $C[i, j] \leftarrow C[i - 1, j - 1] + C[i - 1, j]$

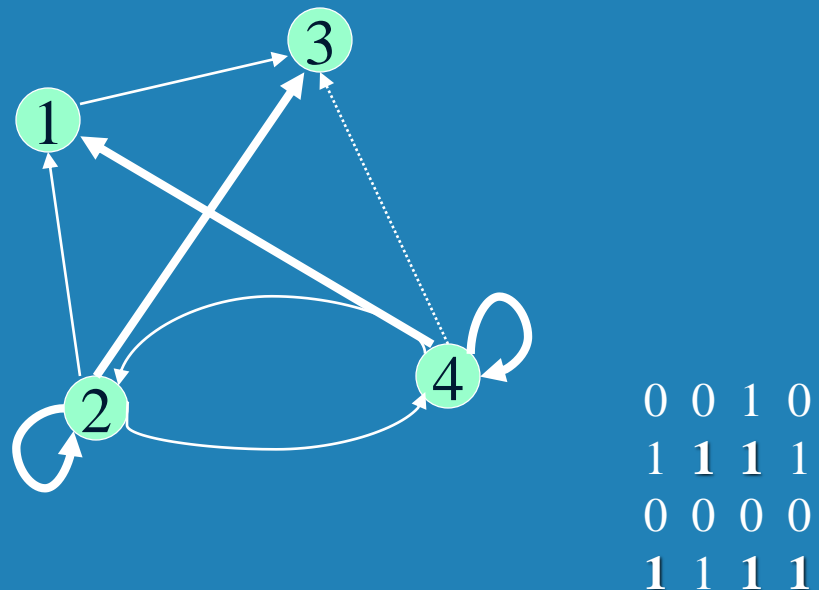
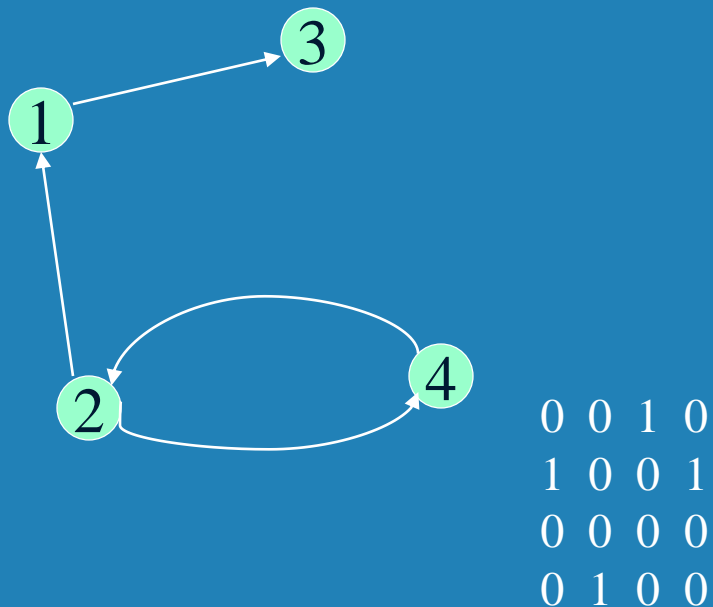
**return**  $C[n, k]$

**Time efficiency:**  $\Theta(nk)$

**Space efficiency:**  $\Theta(nk)$

# Warshall's Algorithm: Transitive Closure

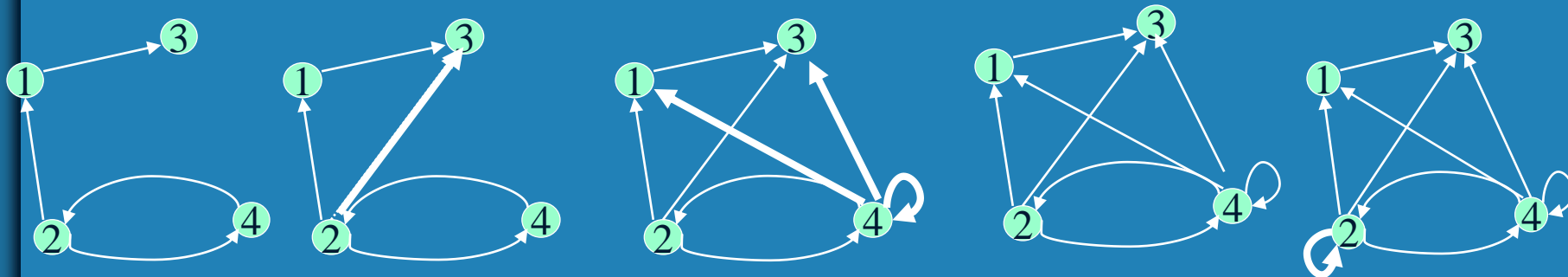
- Computes the transitive closure of a relation
- Alternatively: existence of all nontrivial paths in a digraph
- Example of transitive closure:



# Warshall's Algorithm

Constructs transitive closure  $T$  as the last matrix in the sequence of  $n$ -by- $n$  matrices  $R^{(0)}, \dots, R^{(k)}, \dots, R^{(n)}$  where  $R^{(k)}[i,j] = 1$  iff there is nontrivial path from  $i$  to  $j$  with only the first  $k$  vertices allowed as intermediate

Note that  $R^{(0)} = A$  (adjacency matrix),  $R^{(n)} = T$  (transitive closure)



$R^{(0)}$

|   |   |   |   |
|---|---|---|---|
| 0 | 0 | 1 | 0 |
| 1 | 0 | 0 | 1 |
| 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 0 |

$R^{(1)}$

|   |   |          |   |
|---|---|----------|---|
| 0 | 0 | 1        | 0 |
| 1 | 0 | <b>1</b> | 1 |
| 0 | 0 | 0        | 0 |
| 0 | 1 | 0        | 0 |

$R^{(2)}$

|          |          |          |          |
|----------|----------|----------|----------|
| 0        | 0        | 1        | 0        |
| 1        | 0        | 1        | 1        |
| 0        | 0        | 0        | 0        |
| <b>1</b> | <b>1</b> | <b>1</b> | <b>1</b> |

$R^{(3)}$

|   |   |   |   |
|---|---|---|---|
| 0 | 0 | 1 | 0 |
| 1 | 0 | 1 | 1 |
| 0 | 0 | 0 | 0 |
| 1 | 1 | 1 | 1 |

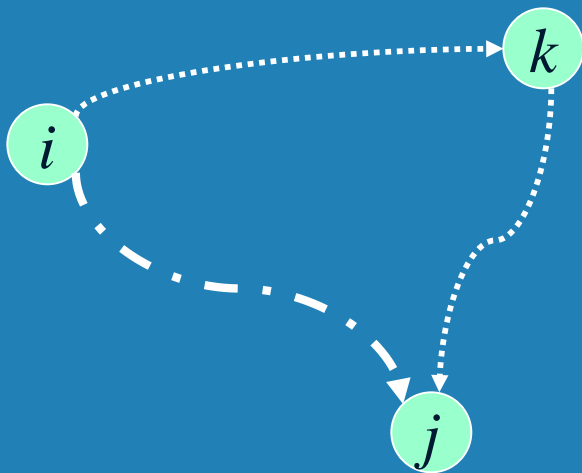
$R^{(4)}$

|   |          |   |   |
|---|----------|---|---|
| 0 | 0        | 1 | 0 |
| 1 | <b>1</b> | 1 | 1 |
| 0 | 0        | 0 | 0 |
| 1 | 1        | 1 | 1 |

# Warshall's Algorithm (recurrence)

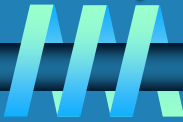
On the  $k$ -th iteration, the algorithm determines for every pair of vertices  $i, j$  if a path exists from  $i$  and  $j$  with just vertices  $1, \dots, k$  allowed as intermediate

$$R^{(k)}[i,j] = \begin{cases} R^{(k-1)}[i,j] & \text{(path using just } 1, \dots, k-1) \\ \text{or} \\ R^{(k-1)}[i,k] \text{ and } R^{(k-1)}[k,j] & \text{(path from } i \text{ to } k \\ & \text{and from } k \text{ to } j \\ & \text{using just } 1, \dots, k-1) \end{cases}$$



Initial condition?

# Warshall's Algorithm (matrix generation)



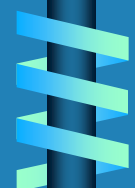
Recurrence relating elements  $R^{(k)}$  to elements of  $R^{(k-1)}$  is:

$$R^{(k)}[i,j] = R^{(k-1)}[i,j] \text{ or } (R^{(k-1)}[i,k] \text{ and } R^{(k-1)}[k,j])$$

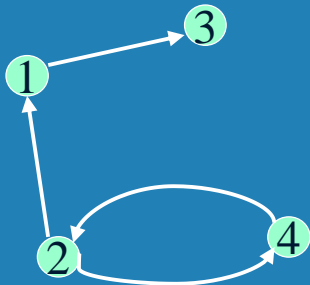
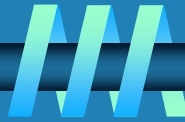
It implies the following rules for generating  $R^{(k)}$  from  $R^{(k-1)}$ :

**Rule 1** If an element in row  $i$  and column  $j$  is 1 in  $R^{(k-1)}$ , it remains 1 in  $R^{(k)}$

**Rule 2** If an element in row  $i$  and column  $j$  is 0 in  $R^{(k-1)}$ , it has to be changed to 1 in  $R^{(k)}$  if and only if the element in its row  $i$  and column  $k$  and the element in its column  $j$  and row  $k$  are both 1's in  $R^{(k-1)}$



# Warshall's Algorithm (example)



$$R^{(0)} = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix}$$

$$R^{(1)} = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix}$$

$$R^{(2)} = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \end{bmatrix}$$

$$R^{(3)} = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \end{bmatrix}$$

$$R^{(4)} = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \end{bmatrix}$$

# Warshall's Algorithm (pseudocode and analysis)

**ALGORITHM** *Warshall*( $A[1..n, 1..n]$ )

//Implements Warshall's algorithm for computing the transitive closure

//Input: The adjacency matrix  $A$  of a digraph with  $n$  vertices

//Output: The transitive closure of the digraph

$R^{(0)} \leftarrow A$

**for**  $k \leftarrow 1$  **to**  $n$  **do**

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

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

$R^{(k)}[i, j] \leftarrow R^{(k-1)}[i, j] \text{ or } (R^{(k-1)}[i, k] \text{ and } R^{(k-1)}[k, j])$

**return**  $R^{(n)}$

**Time efficiency:**  $\Theta(n^3)$

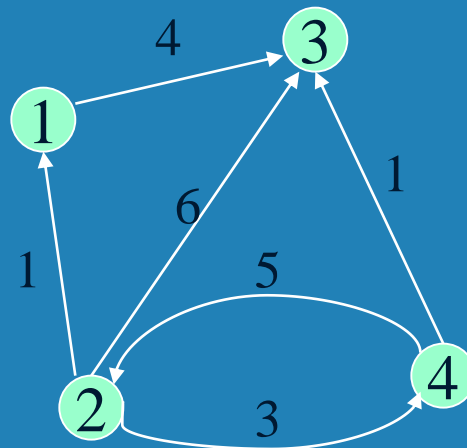
**Space efficiency:** Matrices can be written over their predecessors (with some care), so it's  $\Theta(n^2)$ .

# Floyd's Algorithm: All pairs shortest paths

**Problem:** In a weighted (di)graph, find shortest paths between every pair of vertices

**Same idea:** construct solution through series of matrices  $D^{(0)}$ , ...,  $D^{(n)}$  using increasing subsets of the vertices allowed as intermediate

**Example:**

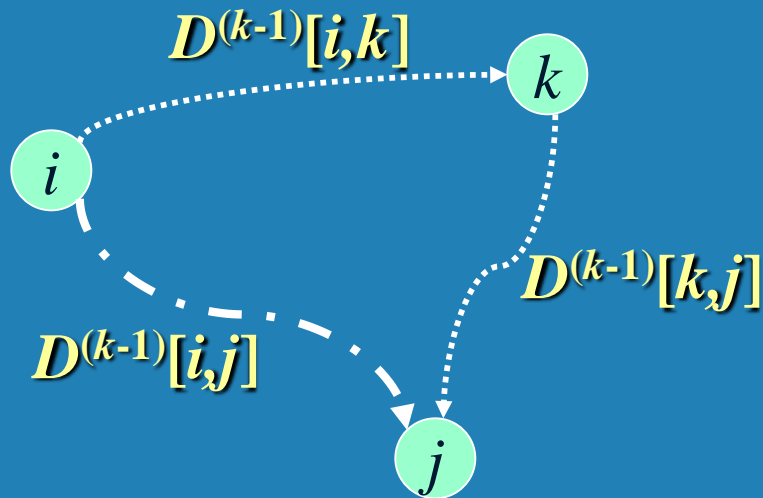


|          |          |   |          |
|----------|----------|---|----------|
| 0        | $\infty$ | 4 | $\infty$ |
| 1        | 0        | 4 | 3        |
| $\infty$ | $\infty$ | 0 | $\infty$ |
| 6        | 5        | 1 | 0        |

# Floyd's Algorithm (matrix generation)

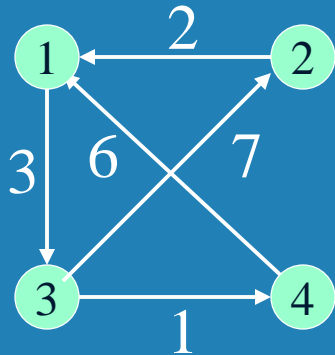
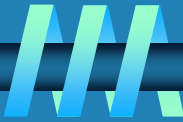
On the  $k$ -th iteration, the algorithm determines shortest paths between every pair of vertices  $i, j$  that use only vertices among  $1, \dots, k$  as intermediate

$$D^{(k)}[i,j] = \min \{D^{(k-1)}[i,j], D^{(k-1)}[i,k] + D^{(k-1)}[k,j]\}$$



Initial condition?

# Floyd's Algorithm (example)



$$D^{(0)} =$$

|          |          |          |          |
|----------|----------|----------|----------|
| 0        | $\infty$ | 3        | $\infty$ |
| 2        | 0        | $\infty$ | $\infty$ |
| $\infty$ | 7        | 0        | 1        |
| 6        | $\infty$ | $\infty$ | 0        |

$$D^{(1)} =$$

|          |          |          |          |
|----------|----------|----------|----------|
| 0        | $\infty$ | 3        | $\infty$ |
| 2        | 0        | <b>5</b> | $\infty$ |
| $\infty$ | 7        | 0        | 1        |
| 6        | $\infty$ | <b>9</b> | 0        |

$$D^{(2)} =$$

|          |          |   |          |
|----------|----------|---|----------|
| 0        | $\infty$ | 3 | $\infty$ |
| 2        | 0        | 5 | $\infty$ |
| <b>9</b> | 7        | 0 | 1        |
| 6        | $\infty$ | 9 | 0        |

$$D^{(3)} =$$

|          |           |   |          |
|----------|-----------|---|----------|
| 0        | <b>10</b> | 3 | <b>4</b> |
| 2        | 0         | 5 | <b>6</b> |
| 9        | 7         | 0 | 1        |
| <b>6</b> | <b>16</b> | 9 | 0        |

$$D^{(4)} =$$

|          |    |   |   |
|----------|----|---|---|
| 0        | 10 | 3 | 4 |
| 2        | 0  | 5 | 6 |
| <b>7</b> | 7  | 0 | 1 |
| 6        | 16 | 9 | 0 |

# Floyd's Algorithm (pseudocode and analysis)

**ALGORITHM** *Floyd*( $W[1..n, 1..n]$ )

//Implements Floyd's algorithm for the all-pairs shortest-paths problem

//Input: The weight matrix  $W$  of a graph with no negative-length cycle

//Output: The distance matrix of the shortest paths' lengths

$D \leftarrow W$  //is not necessary if  $W$  can be overwritten

**for**  $k \leftarrow 1$  **to**  $n$  **do**

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

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

$D[i, j] \leftarrow \min\{D[i, j], D[i, k] + D[k, j]\}$

**return**  $D$

**If**  $D[i,k] + D[k,j] < D[i,j]$  **then**  $P[i,j] \leftarrow k$

**Time efficiency:**  $\Theta(n^3)$

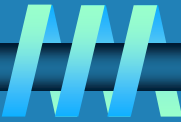
Since the superscripts  $k$  or  $k-1$  make no difference to  $D[i,k]$  and  $D[k,j]$ .

**Space efficiency:** Matrices can be written over their predecessors

**Note:** Works on graphs with negative edges but without negative cycles.

**Shortest paths themselves can be found, too. How?**

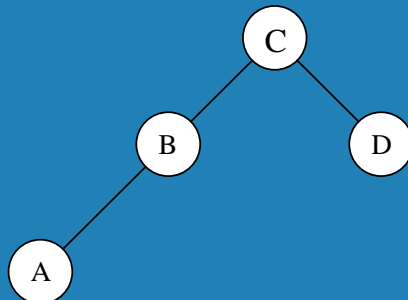
# Optimal Binary Search Trees



**Problem:** Given  $n$  keys  $a_1 < \dots < a_n$  and probabilities  $p_1, \dots, p_n$  searching for them, find a BST with a minimum average number of comparisons in successful search.

Since total number of BSTs with  $n$  nodes is given by  $C(2n, n)/(n+1)$ , which grows exponentially, brute force is hopeless.

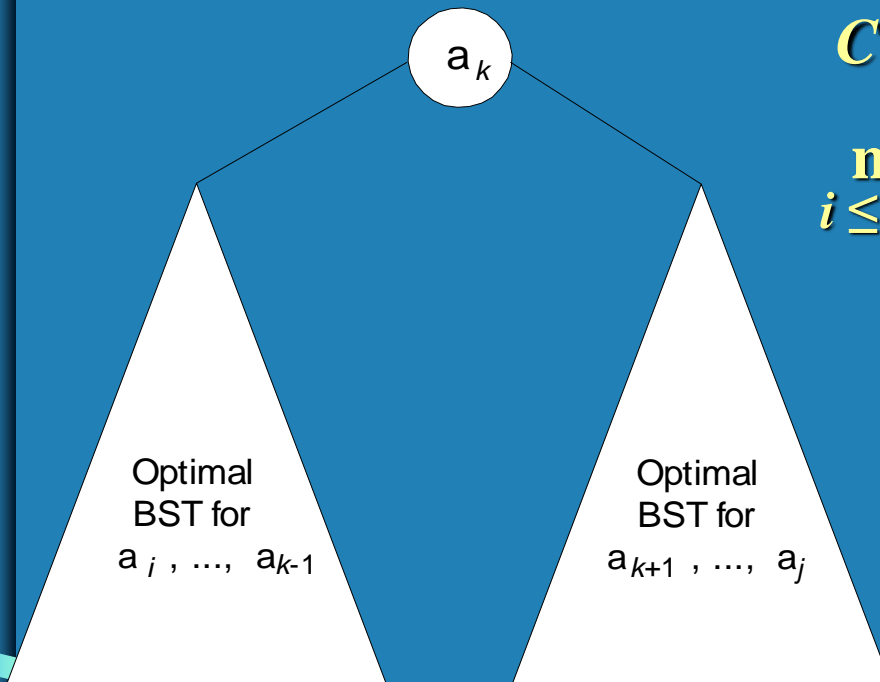
**Example:** What is an optimal BST for keys  $A, B, C$ , and  $D$  with search probabilities 0.1, 0.2, 0.4, and 0.3, respectively?



$$\begin{aligned} \text{Average \# of comparisons} &= 1 * 0.4 + 2 * (0.2 + 0.3) + 3 * 0.1 \\ &= 1.7 \end{aligned}$$

# DP for Optimal BST Problem

Let  $C[i,j]$  be minimum average number of comparisons made in  $T[i,j]$ , optimal BST for keys  $a_i < \dots < a_j$ , where  $1 \leq i \leq j \leq n$ . Consider optimal BST among all BSTs with some  $a_k$  ( $i \leq k \leq j$ ) as their root;  $T[i,j]$  is the best among them.



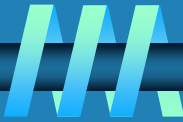
$$C[i,j] =$$

$$\min_{i \leq k \leq j} \{p_k \cdot 1 +$$

$$\sum_{s=i}^{k-1} p_s (\text{level } a_s \text{ in } T[i,k-1] + 1) +$$

$$\sum_{s=k+1}^j p_s (\text{level } a_s \text{ in } T[k+1,j] + 1)\}$$

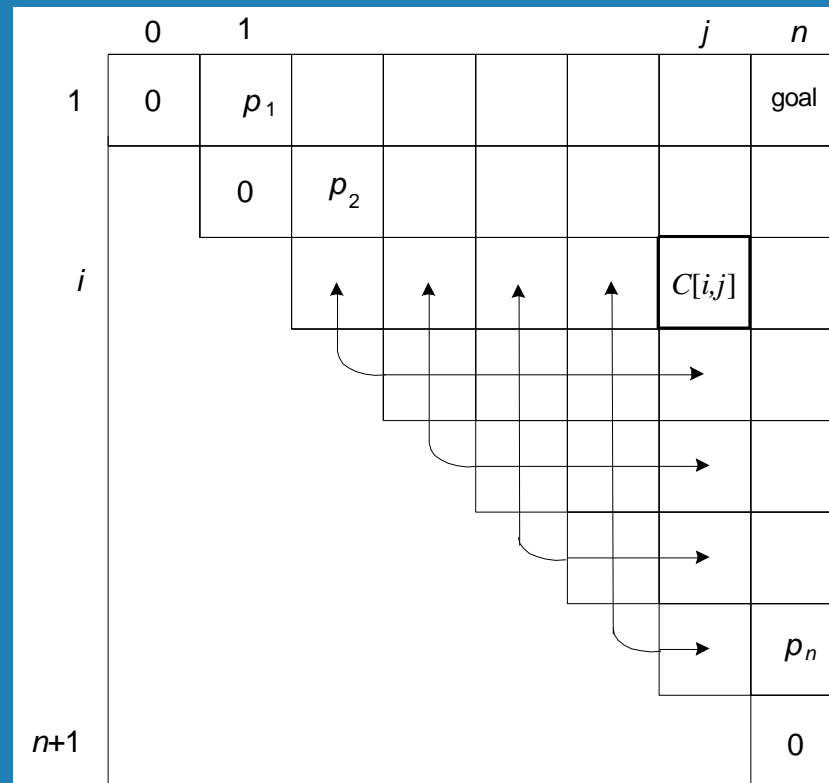
# DP for Optimal BST Problem (cont.)



After simplifications, we obtain the recurrence for  $C[i,j]$ :

$$C[i,j] = \min_{i \leq k \leq j} \{C[i,k-1] + C[k+1,j]\} + \sum_{s=i}^j p_s \quad \text{for } 1 \leq i \leq j \leq n$$

$$C[i,i] = p_i \quad \text{for } 1 \leq i \leq j \leq n$$



|              |          |          |          |          |
|--------------|----------|----------|----------|----------|
| Example: key | <i>A</i> | <i>B</i> | <i>C</i> | <i>D</i> |
| probability  | 0.1      | 0.2      | 0.4      | 0.3      |

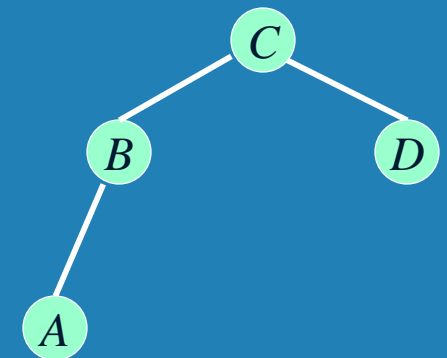
The tables below are filled diagonal by diagonal: the left one is filled using the recurrence

$$C[i,j] = \min_{i \leq k \leq j} \{C[i,k-1] + C[k+1,j]\} + \sum_{s=i}^j p_s, \quad C[i,i] = p_i;$$

the right one, for trees' roots, records *k*'s values giving the minima

| <i>i j</i> | 0 | 1  | 2  | 3   | 4   |
|------------|---|----|----|-----|-----|
| 1          | 0 | .1 | .4 | 1.1 | 1.7 |
| 2          |   | 0  | .2 | .8  | 1.4 |
| 3          |   |    | 0  | .4  | 1.0 |
| 4          |   |    |    | 0   | .3  |
| 5          |   |    |    |     | 0   |

| <i>i j</i> | 0 | 1 | 2 | 3 | 4 |
|------------|---|---|---|---|---|
| 1          |   | 1 | 2 | 3 | 3 |
| 2          |   |   | 2 | 3 | 3 |
| 3          |   |   |   | 3 | 3 |
| 4          |   |   |   |   | 4 |
| 5          |   |   |   |   |   |



optimal BST

# Optimal Binary Search Trees

**ALGORITHM** *OptimalBST*( $P[1..n]$ )

```
//Finds an optimal binary search tree by dynamic programming
//Input: An array  $P[1..n]$  of search probabilities for a sorted list of  $n$  keys
//Output: Average number of comparisons in successful searches in the
//         optimal BST and table  $R$  of subtrees' roots in the optimal BST
for  $i \leftarrow 1$  to  $n$  do
     $C[i, i - 1] \leftarrow 0$ 
     $C[i, i] \leftarrow P[i]$ 
     $R[i, i] \leftarrow i$ 
 $C[n + 1, n] \leftarrow 0$ 
for  $d \leftarrow 1$  to  $n - 1$  do //diagonal count
    for  $i \leftarrow 1$  to  $n - d$  do
         $j \leftarrow i + d$ 
         $minval \leftarrow \infty$ 
        for  $k \leftarrow i$  to  $j$  do
            if  $C[i, k - 1] + C[k + 1, j] < minval$ 
                 $minval \leftarrow C[i, k - 1] + C[k + 1, j]; kmin \leftarrow k$ 
             $R[i, j] \leftarrow kmin$ 
         $sum \leftarrow P[i];$  for  $s \leftarrow i + 1$  to  $j$  do  $sum \leftarrow sum + P[s]$ 
         $C[i, j] \leftarrow minval + sum$ 
return  $C[1, n], R$ 
```

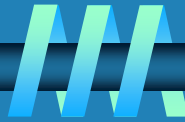
# Analysis DP for Optimal BST Problem

**Time efficiency:**  $\Theta(n^3)$  but can be reduced to  $\Theta(n^2)$  by taking advantage of monotonicity of entries in the root table, i.e.,  $R[i,j]$  is always in the range between  $R[i,j-1]$  and  $R[i+1,j]$

**Space efficiency:**  $\Theta(n^2)$

**Method can be expanded to include unsuccessful searches**

# Knapsack Problem by DP



Given  $n$  items of

integer weights:  $w_1 \ w_2 \ \dots \ w_n$

values:  $v_1 \ v_2 \ \dots \ v_n$

a knapsack of integer capacity  $W$

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

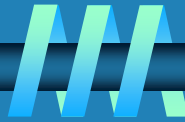
Consider instance defined by first  $i$  items and capacity  $j$  ( $j \leq W$ ).

Let  $V[i,j]$  be optimal value of such an instance. Then

$$V[i,j] = \begin{cases} \max \{V[i-1,j], v_i + V[i-1,j-w_i]\} & \text{if } j-w_i \geq 0 \\ V[i-1,j] & \text{if } j-w_i < 0 \end{cases}$$

Initial conditions:  $V[0,j] = 0$  and  $V[i,0] = 0$

# Knapsack Problem by DP (example)



**Example: Knapsack of capacity  $W = 5$**

| item | weight | value |
|------|--------|-------|
|------|--------|-------|

|   |   |      |
|---|---|------|
| 1 | 2 | \$12 |
|---|---|------|

|   |   |      |
|---|---|------|
| 2 | 1 | \$10 |
|---|---|------|

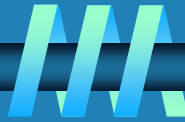
|   |   |      |
|---|---|------|
| 3 | 3 | \$20 |
|---|---|------|

|   |   |      |
|---|---|------|
| 4 | 2 | \$15 |
|---|---|------|

|                     |   | capacity $j$ |    |    |    |    |    |
|---------------------|---|--------------|----|----|----|----|----|
|                     |   | 0            | 1  | 2  | 3  | 4  | 5  |
|                     | 0 | 0            | 0  | 0  |    |    |    |
| $w_1 = 2, v_1 = 12$ | 1 | 0            | 0  | 12 |    |    |    |
| $w_2 = 1, v_2 = 10$ | 2 | 0            | 10 | 12 | 22 | 22 | 22 |
| $w_3 = 3, v_3 = 20$ | 3 | 0            | 10 | 12 | 22 | 30 | 32 |
| $w_4 = 2, v_4 = 15$ | 4 | 0            | 10 | 15 | 25 | 30 | 37 |

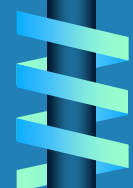
Backtracing  
finds the actual  
optimal subset,  
i.e. solution.

# Example – Dynamic Programming Table

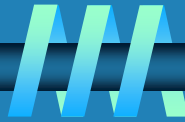


capacity  $W = 5$

|          |   |    | (2,12) | (1,10) | (3,20) | (2,15) |      |  |
|----------|---|----|--------|--------|--------|--------|------|--|
|          |   | 0  | 1      | 2      | 3      | 4      | item |  |
| 0        | 0 | 0  | 0      | 0      | 0      | 0      |      |  |
| 1        | 0 | 0  | 0      | 10     | 10     | 10     |      |  |
| 2        | 0 | 12 | 12     | 12     | 12     | 15     |      |  |
| 3        | 0 | 12 | 22     | 22     | 22     | 25     |      |  |
| 4        | 0 | 12 | 22     | 22     | 30     | 30     |      |  |
| 5        | 0 | 12 | 22     | 32     | 32     | 37     |      |  |
| Capacity |   |    |        |        |        |        |      |  |



# Example

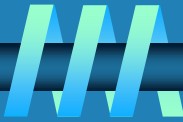


capacity  $W = 5$

|          |   | (2,12) | (1,10) | (3,20) | (2,15) |      |
|----------|---|--------|--------|--------|--------|------|
|          | 0 | 1      | 2      | 3      | 4      | item |
| 0        | 0 | 0      | 0      | 0      | 0      |      |
| 1        | 0 | 0      | 10     | 10     | 10     |      |
| 2        | 0 | 12     | 12     | 12     | 15     |      |
| 3        | 0 | 12     | 22     | 22     | 25     |      |
| 4        | 0 | 12     | 22     | 30     | 30     |      |
| 5        | 0 | 12     | 22     | 32     | 37     |      |
| Capacity |   |        |        |        |        |      |

- Thus, the maximal value is  $V[4, 5] = \$37$ . We can find the composition of an optimal subset by tracing back the computations of this entry in the table.
- Since  $V[4, 5]$  is not equal to  $V[3, 5]$ , item 4 was included in an optimal solution along with an optimal subset for filling  $5 - 2 = 3$  remaining units of the knapsack capacity.

# Example



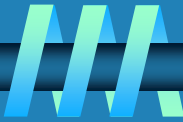
capacity  $W = 5$

|          |   | (2,12) | (1,10) | (3,20) | (2,15) |      |
|----------|---|--------|--------|--------|--------|------|
|          | 0 | 1      | 2      | 3      | 4      | item |
| 0        | 0 | 0      | 0      | 0      | 0      |      |
| 1        | 0 | 0      | 10     | 10     | 10     |      |
| 2        | 0 | 12     | 12     | 12     | 15     |      |
| 3        | 0 | 12     | 22     | 22     | 25     |      |
| 4        | 0 | 12     | 22     | 30     | 30     |      |
| 5        | 0 | 12     | 22     | 32     | 37     |      |
| Capacity |   |        |        |        |        |      |

- ❧ The remaining is  $V[3,3]$
- ❧ Here  $V[3,3] = V[2,3]$  so item 3 is not included
- ❧  $V[2,3] \neq V[1,3]$  so item 2 is included



# Example

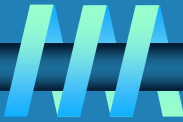


capacity  $W = 5$

|          |   | (2,12) | (1,10) | (3,20) | (2,15) |      |
|----------|---|--------|--------|--------|--------|------|
|          | 0 | 1      | 2      | 3      | 4      | item |
| 0        | 0 | 0      | 0      | 0      | 0      |      |
| 1        | 0 | 0      | 10     | 10     | 10     |      |
| 2        | 0 | 12     | 12     | 12     | 15     |      |
| 3        | 0 | 12     | 22     | 22     | 25     |      |
| 4        | 0 | 12     | 22     | 30     | 30     |      |
| 5        | 0 | 12     | 22     | 32     | 37     |      |
| Capacity |   |        |        |        |        |      |

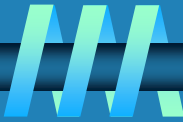
- ❧ The remaining is  $V[1,2]$
- ❧  $V[1,2] \neq V[0,2]$  so item 1 is included
- ❧ The solution is {item 1, item 2, item 4}
- ❧ Total weight is 5
- ❧ Total value is 37

# The Knapsack Problem



- ⌘ The time efficiency and space efficiency of this algorithm are both in  $\theta(nW)$ .
- ⌘ The time needed to find the composition of an optimal solution is in  $O(n + W)$ .

# Knapsack Problem by DP (pseudocode)



**Algorithm DPKnapsack( $w[1..n]$ ,  $v[1..n]$ ,  $W$ )**

**var  $V[0..n, 0..W]$ ,  $P[1..n, 1..W]$ : int**

**for  $j := 0$  to  $W$  do**

**$V[0, j] := 0$**

**for  $i := 0$  to  $n$  do**

**$V[i, 0] := 0$**

**for  $i := 1$  to  $n$  do**

**for  $j := 1$  to  $W$  do**

**if  $w[i] \leq j$  and  $v[i] + V[i-1, j-w[i]] > V[i-1, j]$  then**

**$V[i, j] := v[i] + V[i-1, j-w[i]]$ ;  $P[i, j] := j-w[i]$**

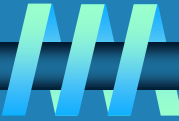
**else**

**$V[i, j] := V[i-1, j]$ ;  $P[i, j] := j$**

**return  $V[n, W]$  and the optimal subset by backtracing**

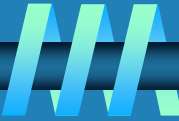
Running time and space:  
 $O(nW)$ .

# Memory Function

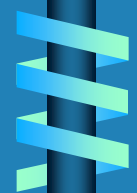


- ❧ The classic dynamic programming approach, fills a table with solutions to *all* smaller subproblems but each of them is solved only once.
- ❧ An unsatisfying aspect of this approach is that solutions to some of these smaller subproblems are often not necessary for getting a solution to the problem given.

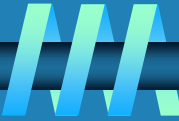
# Memory Function



- ⌚ Since this drawback is not present in the top-down approach, it is natural to try to combine the strengths of the top-down and bottom-up approaches.
- ⌚ The goal is to get a method that solves only subproblems that are necessary and does it only once. Such a method exists; it is based on using *memory functions*



# Memory Function



- Initially, all the table's entries are initialized with a special “null” symbol to indicate that they have not yet been calculated.
- Thereafter, whenever a new value needs to be calculated, the method checks the corresponding entry in the table first: if this entry is not “null,” it is simply retrieved from the table;
- otherwise, it is computed by the recursive call whose result is then recorded in the table.

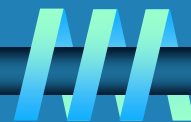


```

//Implements the memory function method for the knapsack problem
//Input: A nonnegative integer  $i$  indicating the number of the first
//       items being considered and a nonnegative integer  $j$  indicating
//       the knapsack's capacity
//Output: The value of an optimal feasible subset of the first  $i$  items
//Note: Uses as global variables input arrays  $Weights[1..n]$ ,  $Values[1..n]$ 
//and table  $V[0..n, 0..W]$  whose entries are initialized with  $-1$ 's except for
//row 0 and column 0 initialized with 0's
if  $V[i, j] < 0$ 
    if  $j < Weights[i]$ 
         $value \leftarrow MFKnapsack(i - 1, j)$ 
    else
         $value \leftarrow \max(MFKnapsack(i - 1, j),$ 
                            $Values[i] + MFKnapsack(i - 1, j - Weights[i]))$ 
     $V[i, j] \leftarrow value$ 
return  $V[i, j]$ 

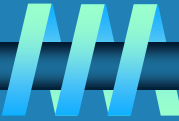
```

# Memory Function for solving Knapsack Problem



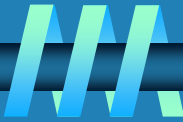
|          |   | (2,12) | (1,10) | (3,20) | (2,15) |      |
|----------|---|--------|--------|--------|--------|------|
|          | 0 | 1      | 2      | 3      | 4      | item |
| 0        | 0 | 0      | 0      | 0      | 0      |      |
| 1        | 0 | 0      | -      | -      | -      |      |
| 2        | 0 | 12     | 12     | -      | -      |      |
| 3        | 0 | -      | 22     | 22     | -      |      |
| 4        | 0 | 12     | -      | -      | -      |      |
| 5        | 0 | 12     | 22     | 32     | 37     |      |
| Capacity |   |        |        |        |        |      |
|          |   |        |        |        |        |      |

# Memory Function

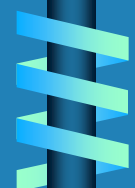


- ❧ In general, we cannot expect more than a constant-factor gain in using the memory function method for the knapsack problem because its time efficiency class is the same as that of the bottom-up algorithm
- ❧ A memory function method may be less space-efficient than a space efficient version of a bottom-up algorithm.

# Conclusion



- ❧ **Dynamic programming is a useful technique of solving certain kind of problems**
- ❧ **When the solution can be recursively described in terms of partial solutions, we can store these partial solutions and re-use them as necessary**



---

# **UNIT-IV**

## **GREEDY ALGORITHM**

**Lecturer:**

**DHANANJAY**

# Greedy Algorithms

---

## ● Optimization problems

- Dynamic programming, but overkill sometime.
- Greedy algorithm:
  - Being greedy for local optimization with the hope it will lead to a global optimal solution, not always, but in many situations, it works.

# An Activity-Selection Problem

---

- Suppose A set of activities  $S = \{a_1, a_2, \dots, a_n\}$ 
  - They use resources, such as lecture hall, one lecture at a time
  - Each  $a_i$ , has a start time  $s_i$ , and finish time  $f_i$ , with  $0 \leq s_i < f_i < \infty$ .
  - $a_i$  and  $a_j$  are compatible if  $[s_i, f_i)$  and  $[s_j, f_j)$  do not overlap
- Goal: select maximum-size subset of mutually compatible activities.
- Start from dynamic programming, then greedy algorithm, see the relation between the two.

# DP solution –step 1

## ● Optimal substructure of activity-selection problem.

- Furthermore, assume that  $f_1 \leq \dots \leq f_n$ .
- Define  $S_{ij} = \{a_k : f_i \leq s_k < f_k \leq s_j\}$ , i.e., all activities starting after  $a_i$  finished and ending before  $a_j$  begins.
- Define two fictitious activities  $a_0$  with  $f_0 = 0$  and  $a_{n+1}$  with  $s_{n+1} = \infty$ 
  - So  $f_0 \leq f_1 \leq \dots \leq f_{n+1}$ .
- Then an optimal solution including  $a_k$  to  $S_{ij}$  contains within it the optimal solution to  $S_{ik}$  and  $S_{kj}$ .

# DP solution –step 2

- A recursive solution
- Assume  $c[n+1,n+1]$  with  $c[i,j]$  is the number of activities in a maximum-size subset of mutually compatible activities in  $S_{ij}$ . So the solution is  $c[0,n+1]=S_{0,n+1}$ .
- $$C[i,j] = \begin{cases} 0 & \text{if } S_{ij} = \emptyset \\ \max\{c[i,k] + c[k,j] + 1\} & \text{if } S_{ij} \neq \emptyset \end{cases}$$

$i < k < j \text{ and } a_k \in S_{ij}$
- How to implement?
  - How to compute the initial cases by checking  $S_{ij} = \emptyset$ ?
  - How to loop to iteratively compute  $C[i,j]$ :
  - For  $i = \dots$  for  $j = \dots$  for  $k = \dots$ ? This is wrong?
  - Need to be similar to MCM:
    - For  $len = \dots$  for  $i = \dots$   $j = i + len$ ; for  $k = \dots$

# Converting DP Solution to Greedy Solution

---

- Theorem 16.1: consider any nonempty subproblem  $S_{ij}$ , and let  $a_m$  be the activity in  $S_{ij}$  with earliest finish time:  $f_m = \min\{f_k : a_k \in S_{ij}\}$ , then
  1. Activity  $a_m$  is used in some maximum-size subset of mutually compatible activities of  $S_{ij}$ .
  2. The subproblem  $S_{im}$  is empty, so that choosing  $a_m$  leaves  $S_{mj}$  as the only one that may be nonempty.
- Proof of the theorem:

# Top-Down Rather Than Bottom-Up

---

- To solve  $S_{ij}$ , choose  $a_m$  in  $S_{ij}$  with the earliest finish time, then solve  $S_{mj}$ , ( $S_{im}$  is empty)
- It is certain that optimal solution to  $S_{mj}$  is in optimal solution to  $S_{ij}$ .
- No need to solve  $S_{mj}$  ahead of  $S_{ij}$ .
- Subproblem pattern:  $S_{i,n+1}$ .

# Optimal Solution Properties

## ● In DP, optimal solution depends:

- How many subproblems to divide. (2 subproblems)
- How many choices to determine which subproblem to use. ( $j-i-1$  choices)

## ● However, the above theorem (16.1) reduces both significantly

- One subproblem (the other is sure to be empty).
- One choice, i.e., the one with earliest finish time in  $S_{ij}$ .
- Moreover, top-down solving, rather than bottom-up in DP.
- Pattern to the subproblems that we solve,  $S_{m,n+1}$  from  $S_{ij}$ .
- Pattern to the activities that we choose. The activity with earliest finish time.
- With this local optimal, it is in fact the global optimal.

# Elements of greedy strategy

---

- ⦿ Determine the optimal substructure
- ⦿ Develop the recursive solution
- ⦿ Prove one of the optimal choices is the greedy choice yet safe
- ⦿ Show that all but one of subproblems are empty after greedy choice
- ⦿ Develop a recursive algorithm that implements the greedy strategy
- ⦿ Convert the recursive algorithm to an iterative one.

# Greedy vs. DP

---

- ⊙ Knapsack problem

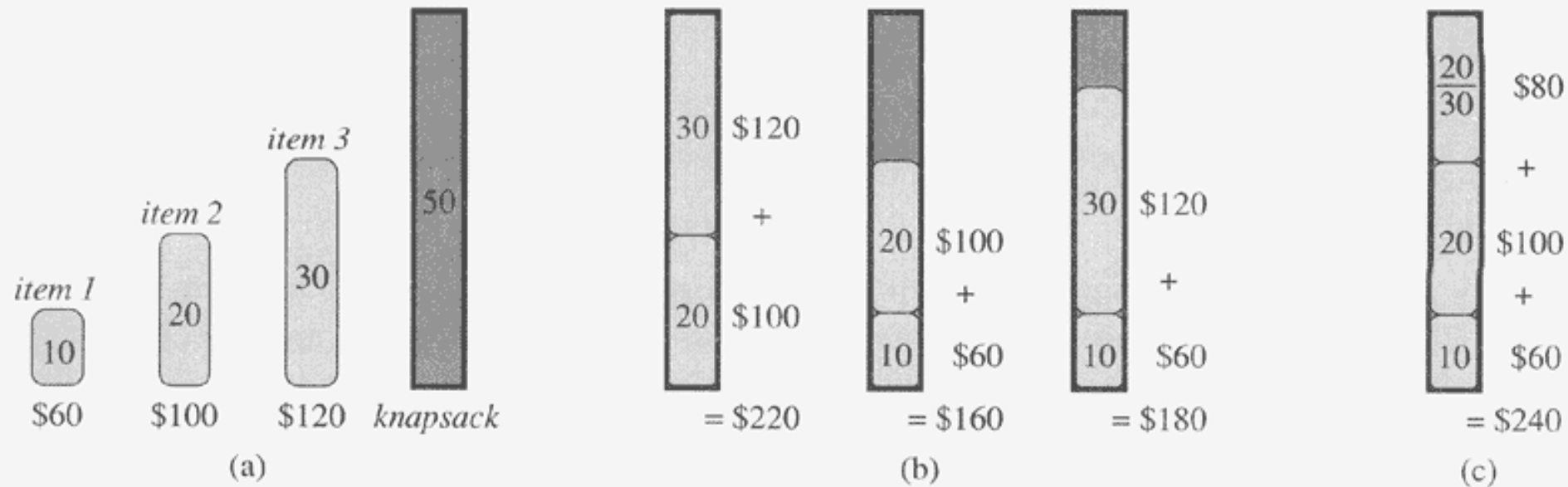
- $I_1(v_1, w_1), I_2(v_2, w_2), \dots, I_n(v_n, w_n)$ .
- Given a weight  $W$  at most he can carry,
- Find the items which maximize the values

- ⊙ Fractional knapsack,

- Greedy algorithm,  $O(n \log n)$

- ⊙ 0/1 knapsack.

- DP,  $O(nW)$ .
- Questions: 0/1 knapsack is an NP-complete problem, why  $O(nW)$  algorithm?



**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.

# Maximum attendance

- A little bit change to the previous activity selection.  
For each activity  $a_i$  such as a talk, there is an associated attendance  $att_i$ .
  - i.e., given  $\{a_1, a_2, \dots, a_n\} = \{(s_1, f_1, att_1), (s_2, f_2, att_2), \dots, (s_n, f_n, att_n)\}$ , compute its maximum attendance of compatible activities.
- Use the one similar to the previous activity-selection:
- $$C[i,j] = \begin{cases} 0 & \text{if } S_{ij} = \emptyset \\ \max\{c[i,k] + c[k,j] + att_k\} & \text{if } S_{ij} \neq \emptyset \end{cases}$$
$$i < k < j \text{ and } a_k \in S_{ij}$$

# Maximum attendance (cont.)

## • New analysis (easy to think sort activities):

- For any  $a_i$ , which is the most recent previous one compatible with it?
  - so, define  $P(i) = \max\{k: k < i \text{ \&\& } f_k \leq s_i\}$ .
  - compute  $P(i)$  is easy.  $P(1) = 0$ . (for easy coding, a dummy  $a_0 = (0, 0, 0)$ )
- Also, define  $T(i)$ : the maximum attendance of all compatible activities from  $a_1$  to  $a_i$ .  $T(n)$  will be an answer.
- Consider activity  $a_i$ , two cases:
  - $a_i$  is contained within the solution, then only  $a_{P(i)}$  can be included too.
  - $a_i$  is not included, then  $a_{i-1}$  can be included.
- $$T(i) = \begin{cases} 0 & \text{if } i=0 \\ \max\{T(i-1), att_i + T(P(i))\} & \text{if } i>0 \end{cases}$$

# Typical tradition problem with greedy solutions

---

- Coin changes
  - 25, 10, 5, 1
  - How about 7, 5, 1
- Minimum Spanning Tree
  - Prim's algorithm
    - Begin from any node, each time add a new node which is closest to the existing subtree.
  - Kruskal's algorithm
    - Sorting the edges by their weights
    - Each time, add the next edge which will not create cycle after added.
- Single source shortest pathes
  - Dijkstra's algorithm
- Huffman coding
- Optimal merge

# UNIT-II

## DISJOINT SET

**Lecturer:**

**Dhananjay**

# Disjoint Sets

- Some applications require maintaining a collection of disjoint sets.
- A Disjoint set  $S$  is a collection of sets  $S_1, \dots, S_n$  where  $\forall_{i \neq j} S_i \cap S_j = \emptyset$
- Each set has a representative which is a member of the set (Usually the minimum if the elements are comparable)

# Disjoint Set Operations

- Make-Set( $x$ ) – Creates a new set where  $x$  is its only element (and therefore it is the representative of the set).  $O(1)$  time.
- Union( $x, y$ ) – Replaces  $S_x, S_y$  by  $S_x \cup S_y$  one of the elements of  $S_x \cup S_y$  becomes the representative of the new set.  $O(\log n)$  time.
- Find( $x$ ) – Returns the representative of the set containing  $x$   $O(\log n)$  time

# Analyzing Operations

- We usually analyze a sequence of  $m$  operations, of which  $n$  of them are Make\_Set operations, and  $m$  is the total of Make\_Set, Find, and Union operations
- Each union operations decreases the number of sets in the data structure, so there can not be more than  $n-1$  Union operations

# Applications

- Equivalence Relations (e.g Connected Components)
- Minimal Spanning Trees

# Connected Components

- Given a graph  $G$  we first preprocess  $G$  to maintain a set of connected components.

$\text{CONNECTED\_COMPONENTS}(G)$

- Later a series of queries can be executed to check if two vertexes are part of the same connected component

$\text{SAME\_COMPONENT}(U,V)$

# Connected Components

*CONNECTED\_COMPONENTS( $G$ )*

*for each vertex  $v$  in  $V[G]$*   
*do  $MAKE\_SET(v)$*

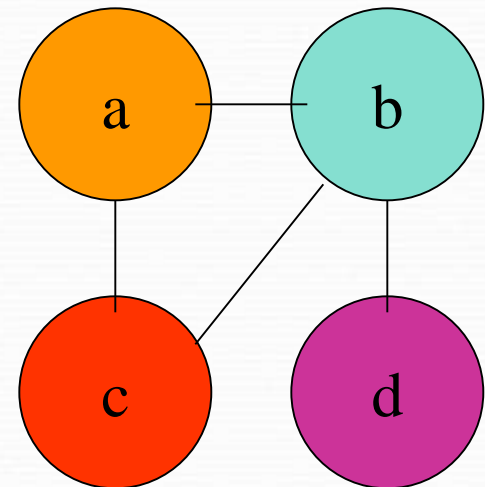
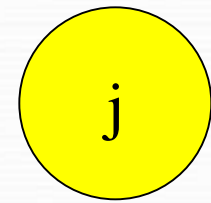
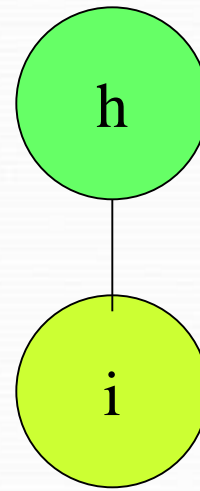
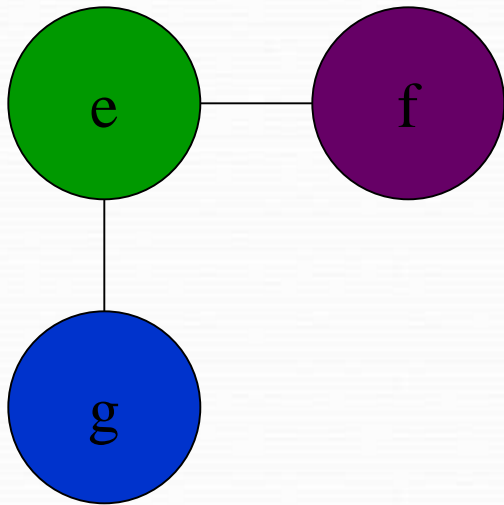
*for each edge  $(u,v)$  in  $E[G]$*   
*do if  $FIND\_SET(u) \neq FIND\_SET(v)$*   
*then  $UNION(u,v)$*

# Connected Components

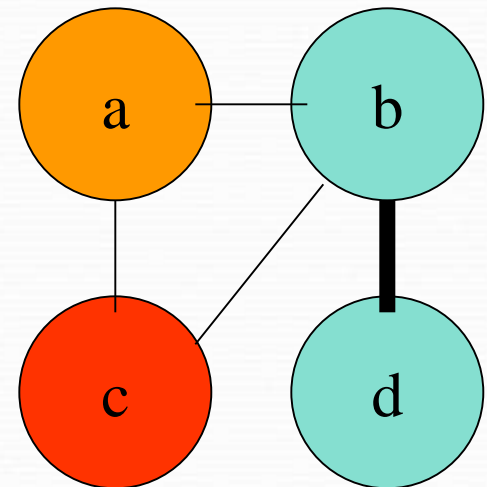
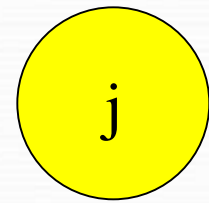
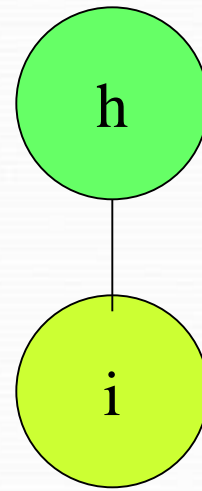
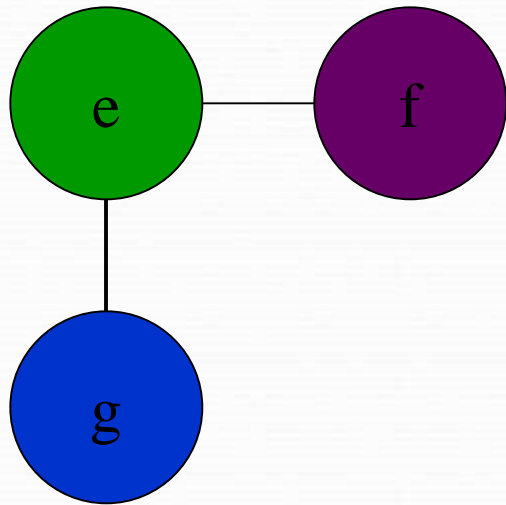
*SAME\_COMPONENT(u,v)*

*return FIND\_SET(u) == FIND\_SET(v)*

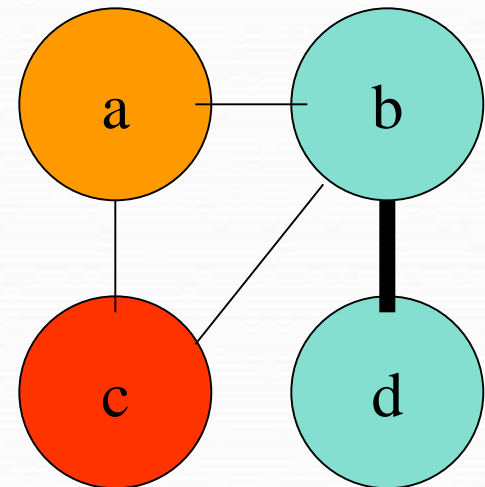
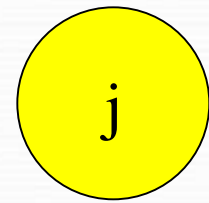
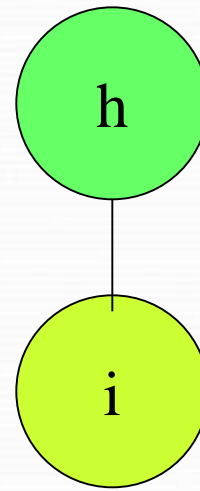
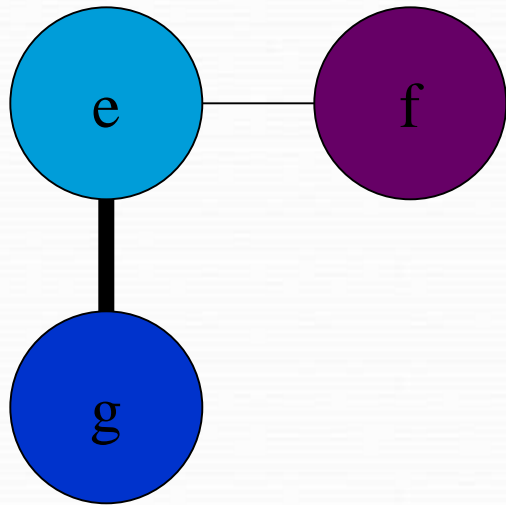
# Example



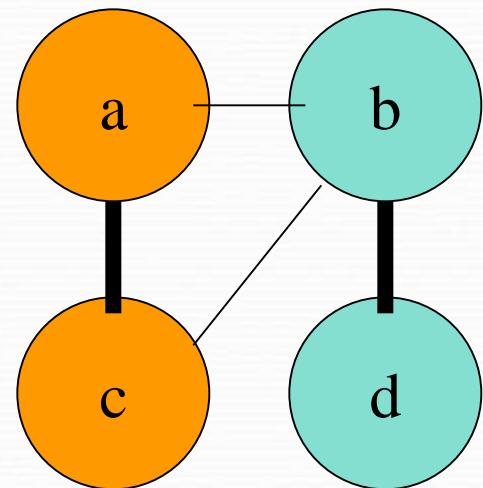
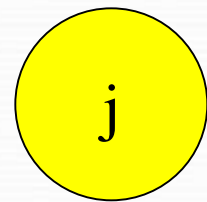
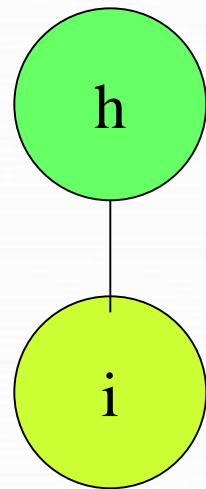
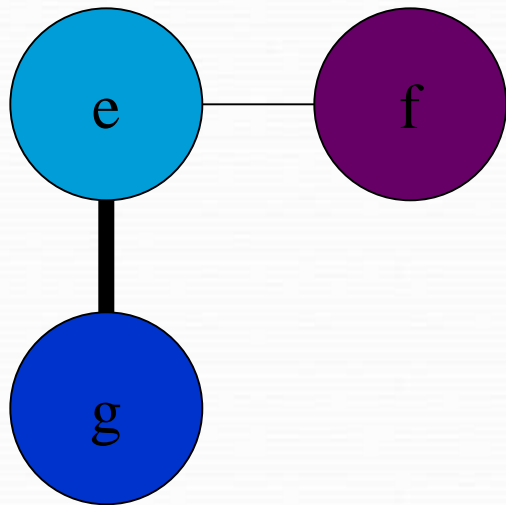
(b,d)



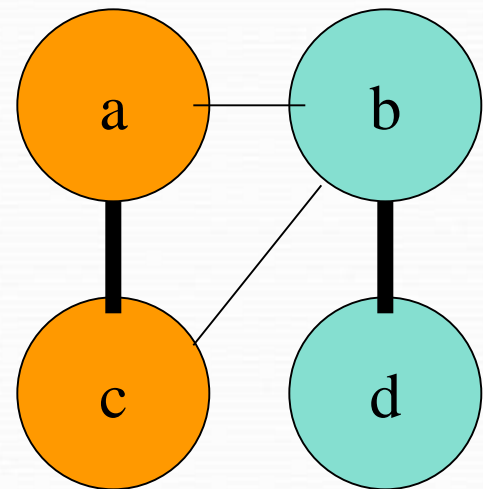
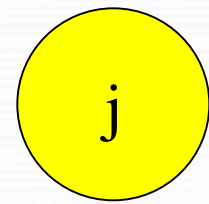
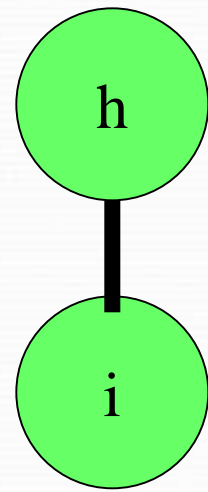
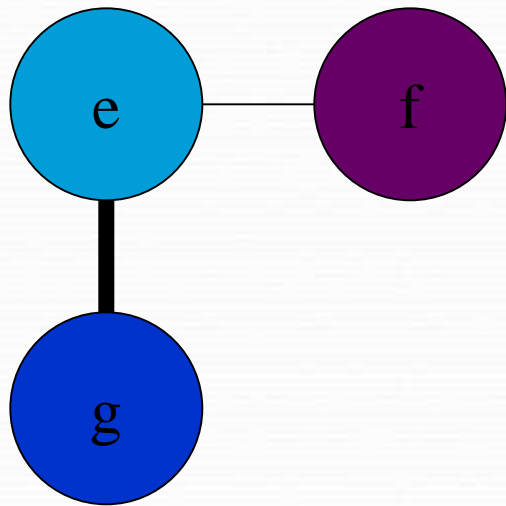
(e,g)



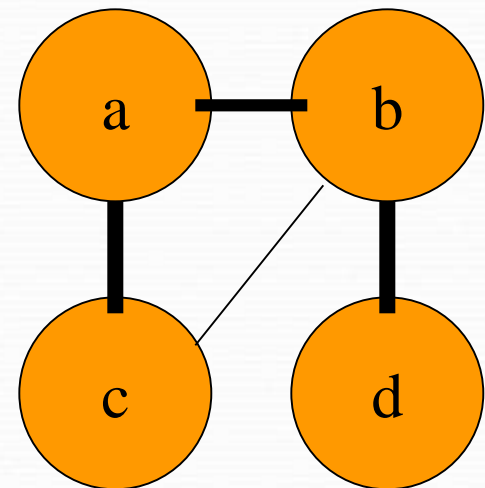
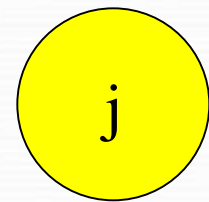
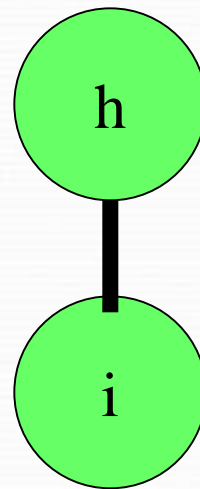
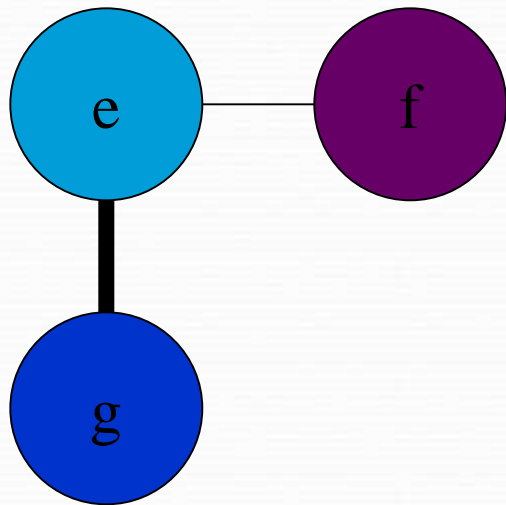
(a,c)



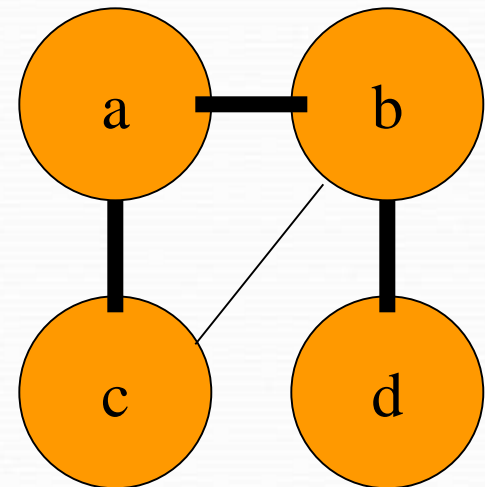
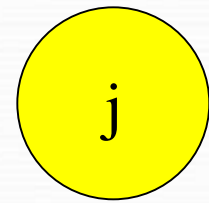
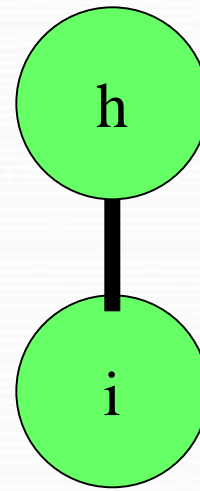
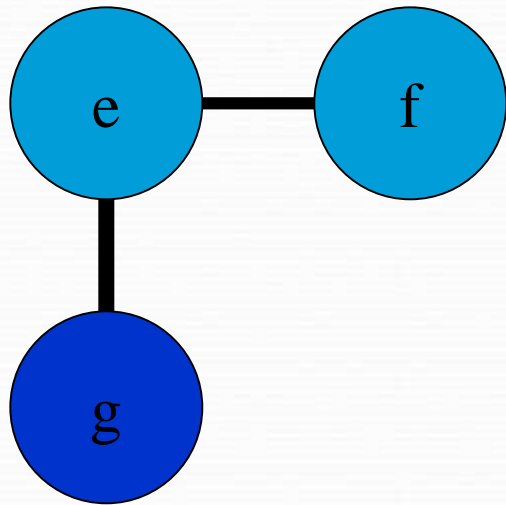
$(h,i)$



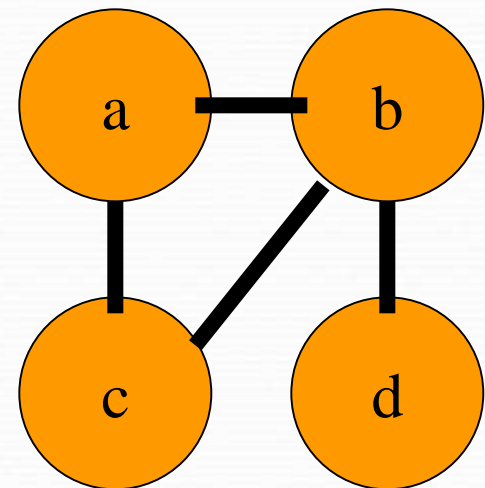
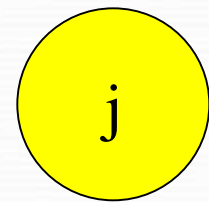
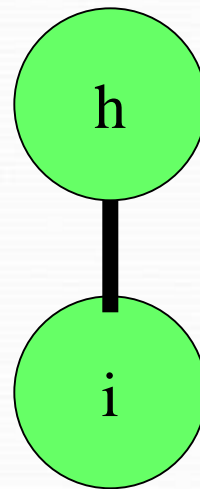
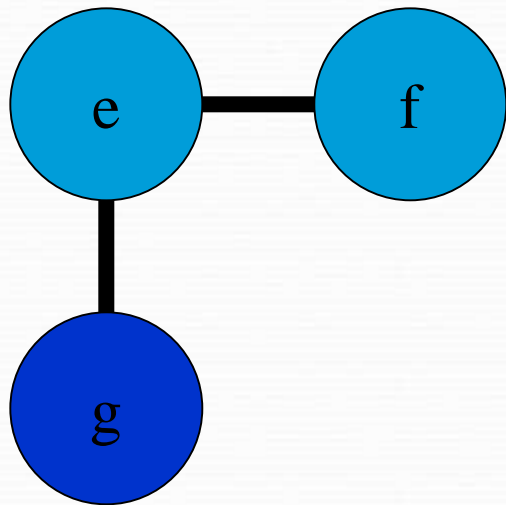
(a,b)



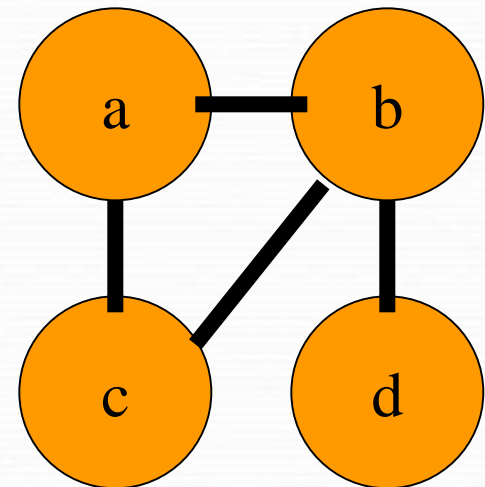
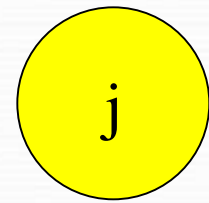
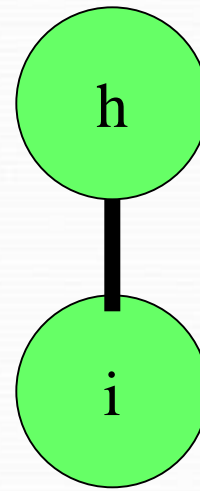
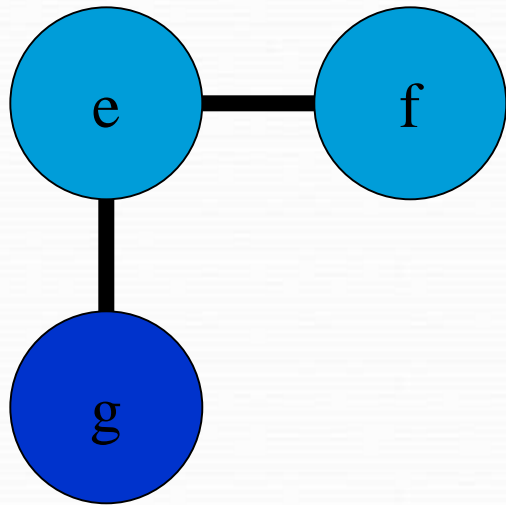
(e,f)



(b,c)



# Result



# Connected Components

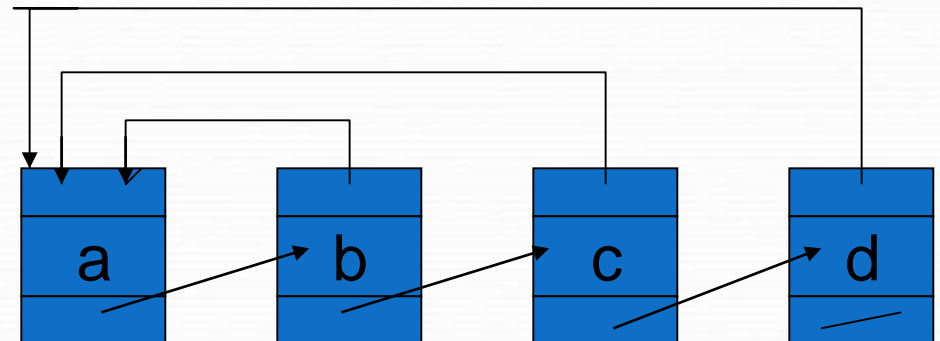
- During the execution of CONNECTED-COMPONENTS on a undirected graph  $G = (V, E)$  with  $k$  connected components, how many time is FIND-SET called? How many times is UNION called? Express you answers in terms of  $|V|$ ,  $|E|$ , and  $k$ .

# Solution

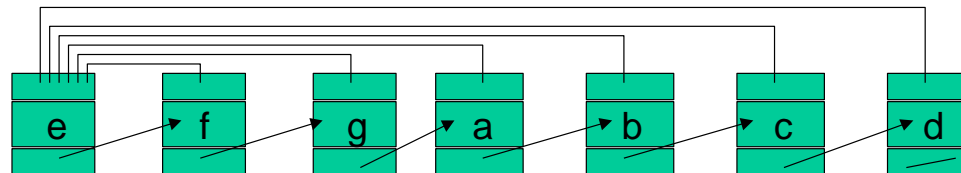
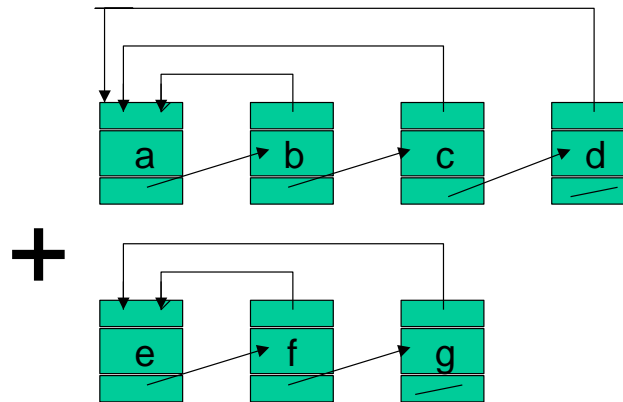
- FIND-SET is called  $2|E|$  times. FIND-SET is called twice on line 4, which is executed once for each edge in  $E[G]$ .
- UNION is called  $|V| - k$  times. Lines 1 and 2 create  $|V|$  disjoint sets. Each UNION operation decreases the number of disjoint sets by one. At the end there are  $k$  disjoint sets, so UNION is called  $|V| - k$  times.

# Linked List implementation

- We maintain a set of linked list, each list corresponds to a single set.
- All elements of the set point to the first element which is the representative
- A pointer to the tail is maintained so elements are inserted at the end of the list



# Union with linked lists



# Analysis

- Using linked list, MAKE\_SET and FIND\_SET are constant operations, however UNION requires to update the representative for at least all the elements of one set, and therefore is linear in worst case time
- A series of m operations could take

$$\Theta(m^2)$$

# Analysis

- Let  $q = m - n + 1 = \lfloor m/2 \rfloor$ ,  $n = \lceil m/2 \rceil$ . Let  $n$  be the number of make set operations, then a series of  $n$  MAKE\_SET operations, followed by  $q-1$  UNION operations will take  $\Theta(m^2)$  since

$$n + 1 + 2 + 3 + \dots + q - 1 = n + \sum_{i=1}^{q-1} i = n + q^2$$

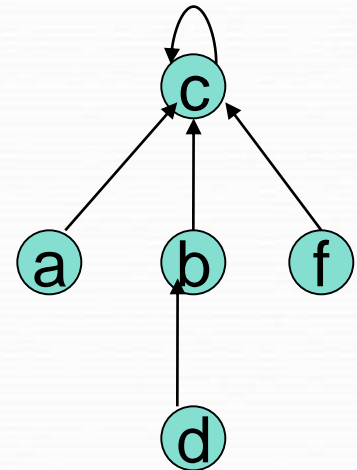
- $q, n$  are an order of  $m$ , so in total we get  $\Theta(m^2)$  which is an amortized cost of  $m$  for each operations

# Improvement – Weighted Union

- Always append the shortest list to the longest list.  
A series of operations will now cost only  $\Theta(m + n \log n)$
- MAKE\_SET and FIND\_SET are constant time and there are  $m$  operations.
- For Union, a set will not change its representative more than  $\log(n)$  times. So each element can be updated no more than  $\log(n)$  time, resulting in  $n \log n$  for all union operations

# Disjoint-Set Forests

- Maintain A collection of trees, each element points to it's parent. The root of each tree is the representative of the set
- We use two strategies for improving running time
  - Union by Rank
  - Path Compression

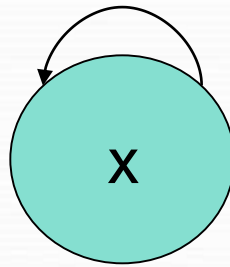


# Make Set

- **MAKE\_SET (x)**

$p(x) = x$

$\text{rank}(x) = 0$



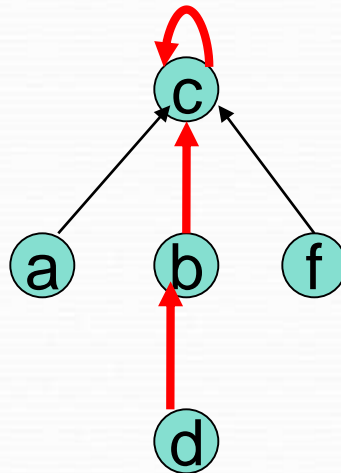
# Find Set

- **FIND\_SET(d)**

```
if d != p[d]
```

```
    p[d] = FIND_SET(p[d])
```

```
return p[d]
```



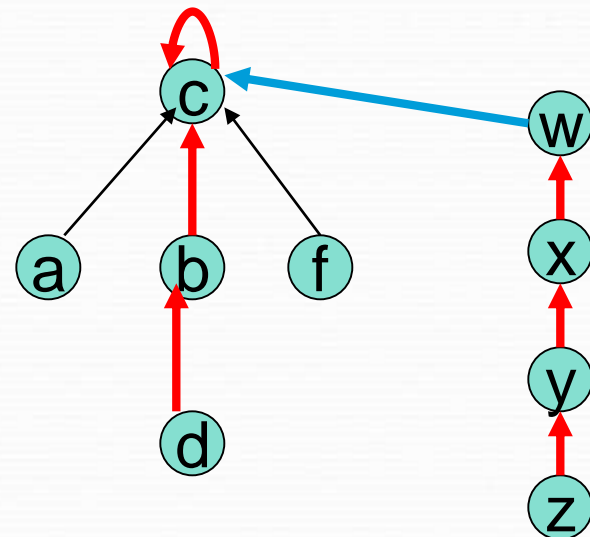
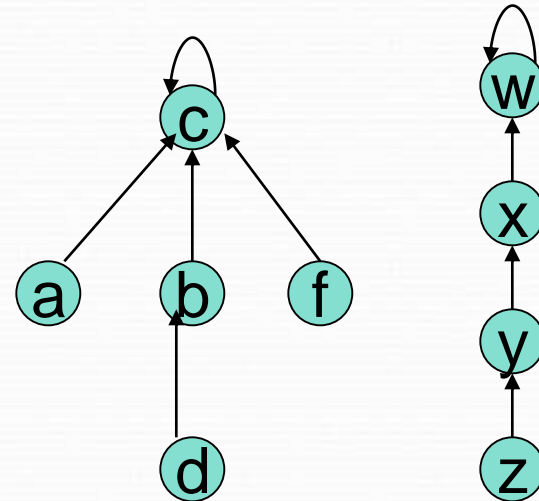
# Union

- **UNION (x, y)**

```
link (findSet (x),  
      findSet (y))
```

- **link (x, y)**

```
if rank(x) > rank(y)  
  then p(y) = x  
else  
  p(x) = y  
  if rank(x) = rank(y)  
    then rank(y) ++
```



# Analysis

- In Union we attach a smaller tree to the larger tree, results in logarithmic depth.
- Path compression can cause a very deep tree to become very shallow
- Combining both ideas gives us (without proof) a sequence of  $m$  operations in  $O(m\alpha(m, n))$

# Exercise

- Describe a data structure that supports the following operations:
  - $\text{find}(x)$  – returns the representative of  $x$
  - $\text{union}(x,y)$  – unifies the groups of  $x$  and  $y$
  - $\text{min}(x)$  – returns the minimal element in the group of  $x$

# Solution

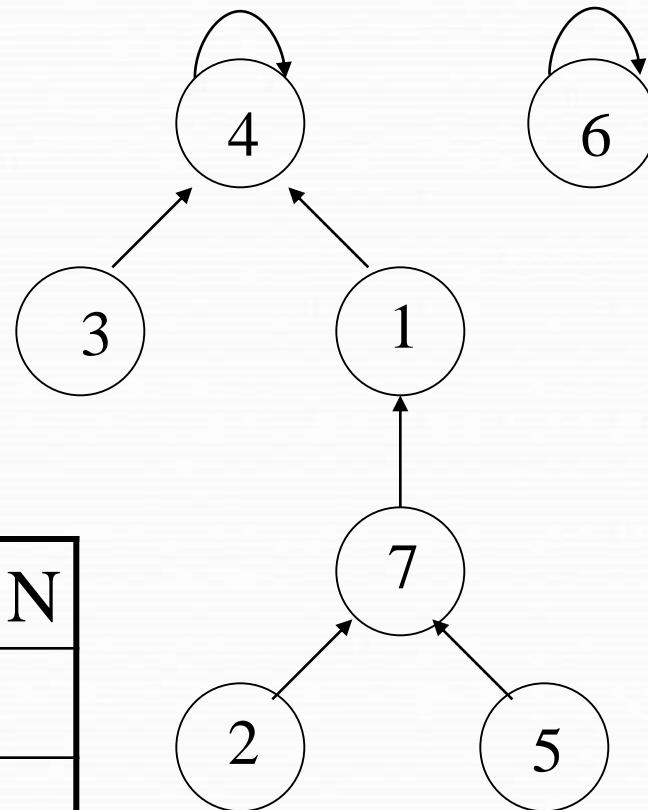
- We modify the disjoint set data structure so that we keep a reference to the minimal element in the group representative.
- The find operation does not change ( $\log(n)$ )
- The union operation is similar to the original union operation, and the minimal element is the smallest between the minimal of the two groups

# Example

- Executing find(5)

$7 \rightarrow 1 \rightarrow 4 \rightarrow 4$

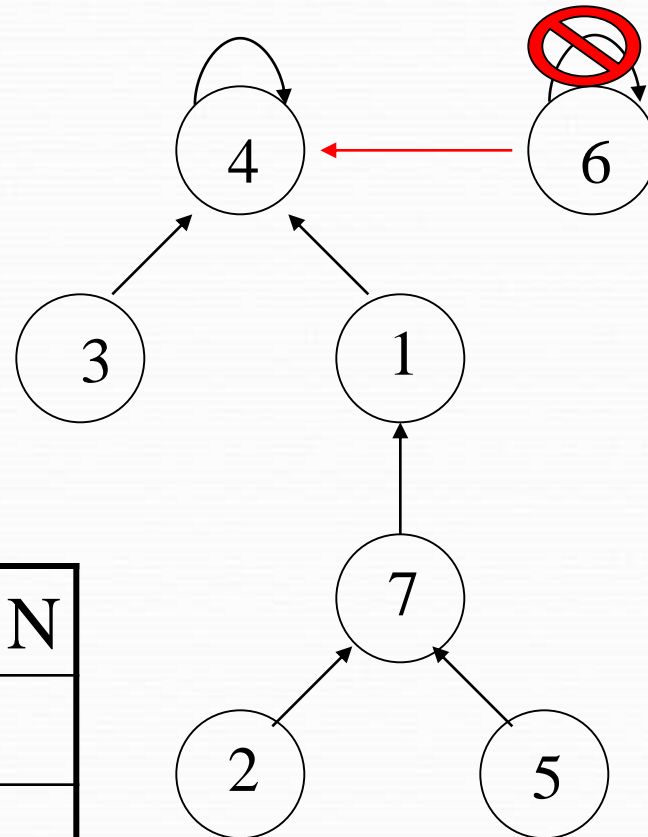
|        | 1 | 2 | 3 | 4 | 5 | 6 | .. | N |
|--------|---|---|---|---|---|---|----|---|
| Parent | 4 | 7 | 4 | 4 | 7 | 6 |    |   |
| min    |   |   |   | 1 |   | 6 |    |   |



# Example

- Executing union(4,6)

|        | 1 | 2 | 3 | 4 | 5 | 6 | .. | N |
|--------|---|---|---|---|---|---|----|---|
| Parent | 4 | 7 | 4 | 4 | 7 | 4 |    |   |
| min    |   |   |   | 1 |   | 1 |    |   |



# Exercise

- Describe a data structure that supports the following operations:
  - $\text{find}(x)$  – returns the representative of  $x$
  - $\text{union}(x,y)$  – unifies the groups of  $x$  and  $y$
  - $\text{deUnion}()$  – undo the last union operation

# Solution

- We modify the disjoint set data structure by adding a stack, that keeps the pairs of representatives that were last merged in the union operations
- The find operations stays the same, but we can not use path compression since we don't want to change the modify the structure after union operations

# Solution

- The union operation is a regular operation and involves an addition push  $(x,y)$  to the stack
- The deUnion operation is as follows
  - $(x,y) \leftarrow s.pop()$
  - $parent(x) \leftarrow x$
  - $parent(y) \leftarrow y$

# Example

- Example why we can not use path compression.
  - Union (8,4)
  - Find(2)
  - Find(6)
  - DeUnion()

|        |   |   |   |   |   |   |   |   |   |    |  |
|--------|---|---|---|---|---|---|---|---|---|----|--|
|        | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |  |
| parent | 4 | 7 | 7 | 4 | 8 | 1 | 5 | 8 | 1 | 4  |  |