



**A T M E**  
College of Engineering



# **BRUTE FORCE APPROACHES**

## **Module 2- Chapt 1**

**Mrs. Madhu Nagaraj**  
**Assistant Professor**  
**Dept of CSE-Data Science**  
**ATMECE**

## Exhaustive Search

- Exhaustive search is simply a brute-force approach to combinatorial problems.
- It suggests generating each and every element of the problem domain, selecting those of them that satisfy all the constraints, and then finding a desired element.

### Method:

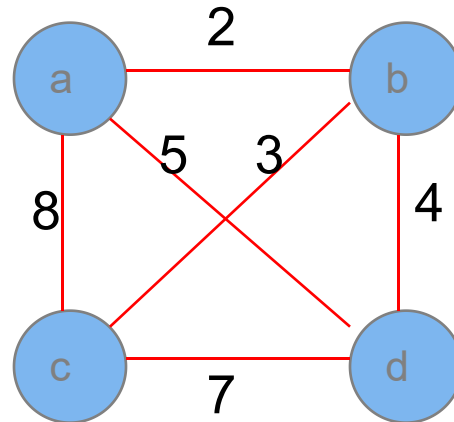
- generate a list of all potential solutions to the problem in a systematic manner.
- evaluate potential solutions one by one, disqualifying infeasible ones and, for an optimization problem, keeping track of the best one found so far
- when search ends, announce the solution(s) found.

Example: Traveling Salesman Problem (TSP) and Knapsack Problem.

## Traveling Salesman Problem

- Given  $n$  cities with known distances between each pair, find the shortest tour that passes through all the cities exactly once before returning to the starting city
- More formally: Find shortest *Hamiltonian circuit* in a weighted connected graph.
- A Hamiltonian circuit is defined as a cycle that passes through all the vertices of the graph exactly once.

Example:





Tour	Cost
$a \rightarrow b \rightarrow c \rightarrow d \rightarrow a$	$2+3+7+5 = 17$
$a \rightarrow b \rightarrow d \rightarrow c \rightarrow a$	$2+4+7+8 = 21$
$a \rightarrow c \rightarrow b \rightarrow d \rightarrow a$	$8+3+4+5 = 20$
$a \rightarrow c \rightarrow d \rightarrow b \rightarrow a$	$8+7+4+2 = 21$
$a \rightarrow d \rightarrow b \rightarrow c \rightarrow a$	$5+4+3+8 = 20$
$a \rightarrow d \rightarrow c \rightarrow b \rightarrow a$	$5+7+3+2 = 17$

Efficiency: # tours =  $O(\# \text{ permutations of } b, c, d) = O(n!)$



**A T M E**  
College of Engineering



## **Advantages:**

- Always gives optimal solution
- Straight forward approach
- Simple technique

## **Disadvantages**

- Inefficient method  $((n-1)!$  possible routes)
- Time consuming to find optimal route.

## Knapsack Problem

Given  $n$  items:

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

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

a knapsack of capacity  $W$

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

Total no of subsets for  $n$  elements is  $2^n - 1$ . So the exhaustive search leads to  $\Omega(2^n)$

Example: Knapsack capacity  $W=16$

<u>item</u>	<u>weight</u>	<u>value</u>
1	2	\$20
2	5	\$30
3	10	\$50
4	5	\$10

## Knapsack: Exhaustive Search

**Efficiency: how many subsets?**

Subset	Total weight	Total value
{1}	2	\$20
{2}	5	\$30
{3}	10	\$50
{4}	5	\$10
{1,2}	7	\$50
{1,3}	12	\$70
{1,4}	7	\$30
{2,3}	15	\$80
{2,4}	10	\$40
{3,4}	15	\$60
{1,2,3}	17	not feasible
{1,2,4}	12	\$60
{1,3,4}	17	not feasible
{2,3,4}	20	not feasible
{1,2,3,4}	22	not feasible



**A T M E**  
College of Engineering



# **DECREASE-AND-CONQUER**

## **Module 2- Chapt 2**

**Mrs. Madhu Nagaraj**  
**Assistant Professor**  
**Dept of CSE-Data Science**  
**ATMECE**



## Decrease-and-Conquer

1. Reduce problem instance to smaller instance of the same problem
2. Solve smaller instance
3. Extend solution of smaller instance to obtain solution to original instance
4. Can be implemented either Recursively(top-down) or Iteratively(bottom-up).
5. Also referred to as *inductive* or *incremental* approach

## 3 Types of Decrease and Conquer

- Decrease by a constant, usually by 1(insertion sort)
- Decrease by a constant factor (Binary Search)
- Variable size Decrease (Euclid's Algo)

## Insertion Sort

To sort array  $A[0..n-1]$ , sort  $A[0..n-2]$  recursively and then insert  $A[n-1]$  in its proper place among the sorted  $A[0..n-2]$

☞ Usually implemented bottom up (nonrecursively)

Example: Sort 6, 4, 1, 8, 5

6		4		1		8		5
4		<u>6</u>		<u>1</u>		8		5
1		4		<u>6</u>		<u>8</u>		5
1		4		6		<u>8</u>		<u>5</u>
1		4		5		6		<u>8</u>

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

//Sorts a given array by insertion sort

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

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

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

$\text{temp} \leftarrow A[i]$

$j \leftarrow i - 1$

while  $j \geq 0$  and  $A[j] > \text{temp}$  do

$A[j + 1] \leftarrow A[j]$

$j \leftarrow j - 1$

$A[j + 1] \leftarrow v$

## Analysis of Insertion Sort

Time efficiency

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

$$C_{\text{avg}}(n) \approx n^2/4 \in \Theta(n^2)$$

$$C_{\text{best}}(n) = n - 1 \in \Theta(n) \text{ (also fast on almost sorted arrays)}$$

Space efficiency: in-place

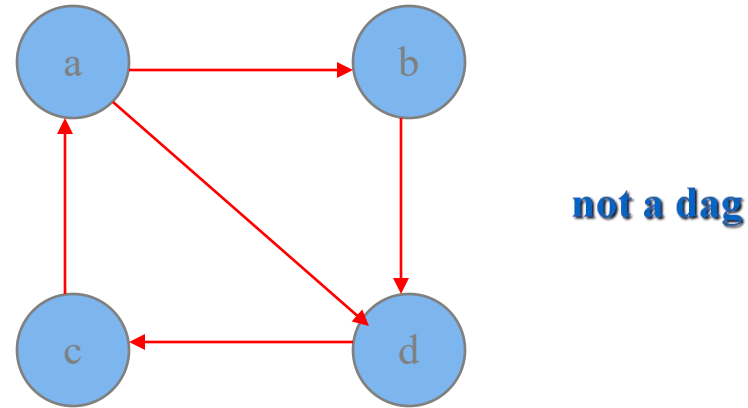
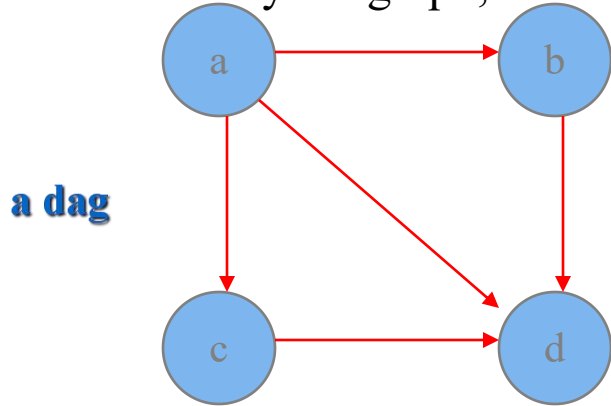
Stability: yes

Best elementary sorting algorithm overall

Binary insertion sort

## Topological Sorting

A dag: a directed acyclic graph, i.e. a directed graph with no (directed) cycles



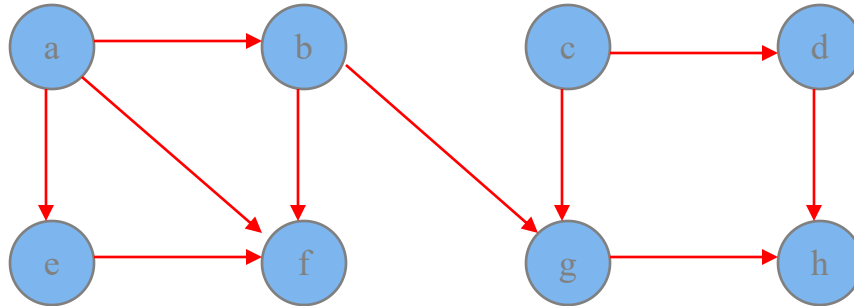
Arise in modeling many problems that involve prerequisite constraints (construction projects, document version control)

Vertices of a dag can be linearly ordered so that for every edge its starting vertex is listed before its ending vertex (topological sorting). Being a dag is also a necessary condition for topological sorting be possible.

## DFS-based algorithm for topological sorting

- Perform DFS traversal, noting the order vertices are popped off the traversal stack
- Reverse order solves topological sorting problem
- Back edges encountered? → NOT a dag!

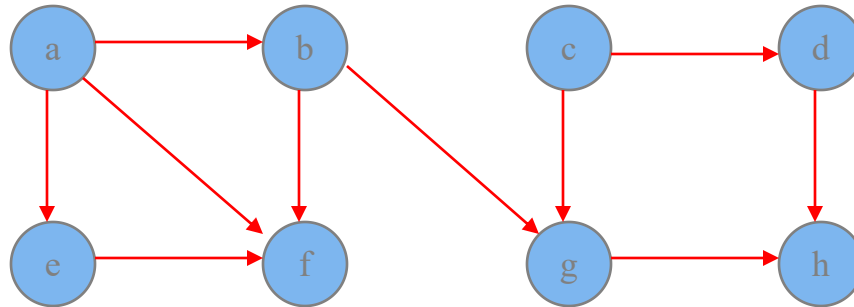
Example:



## Source removal algorithm

Repeatedly identify and remove a *source* (a vertex with no incoming edges) and all the edges incident to it until either no vertex is left (problem is solved) or there is no source among remaining vertices (not a dag)

Example:



Efficiency: same as efficiency of the DFS-based algorithm



- Applications of Topological Sorting
  - Instruction scheduling in program compilation
  - Cell evaluation ordering in spreadsheet formula
  - Resolving symbol dependencies in linkers





**A T M E**  
College of Engineering



# **DIVIDE-AND-CONQUER**

## **Module 2- Chapt 3**

**Mrs. Madhu Nagaraj**  
**Assistant Professor**  
**Dept of CSE-Data Science**  
**ATMECE**

## Divide-and-Conquer

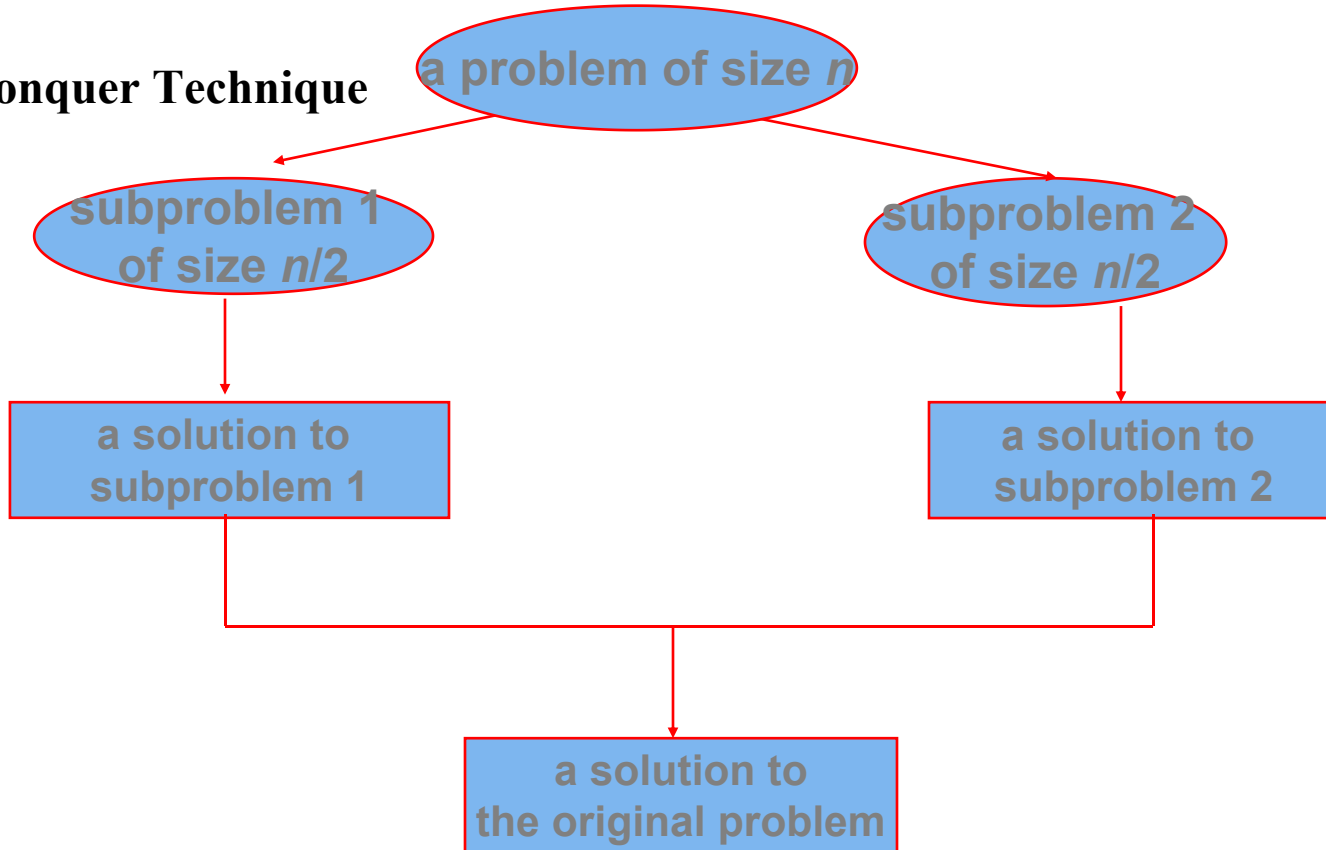
The most-well known algorithm design strategy:

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 Examples

- Sorting: mergesort and quicksort
- Binary tree traversals
- Multiplication of large integers
- Matrix multiplication: Strassen's algorithm

## Divide-and-Conquer Technique



## Mergesort

- ❧ Split array  $A[0..n-1]$  in two about equal halves and make copies of each half in arrays B and C
- ❧ Sort arrays B and C recursively
- ❧ 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, copy the remaining unprocessed elements from the other array into A.

```
ALGORITHM Mergesort(A[0..n - 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..[n/2] - 1] to B[0..[n/2] - 1]  
    copy A[[n/2]..n - 1] to C[0..[n/2] - 1]  
    Mergesort(B[0..[n/2] - 1])  
    Mergesort(C[0..[n/2] - 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  $A[k..p + q - 1]$ 
```

## Analysis of Mergesort

∴ Number of key comparisons – recurrence:

$$C(n) = 2C(n/2) + C'(n) = 2C(n/2) + n-1$$

For merge sort – worst case:

$$C'(n) = n-1$$

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

Number of comparisons in the worst case is close to theoretical minimum for comparison-based sorting:

- Theoretical min:  $\lceil \log_2 n! \rceil \approx n \log_2 n - 1.44n$

## General Divide-and-Conquer Recurrence

$$T(n) = aT(n/b) + f(n) \quad \text{where } f(n) \in \Theta(n^d), \quad d \geq 0$$

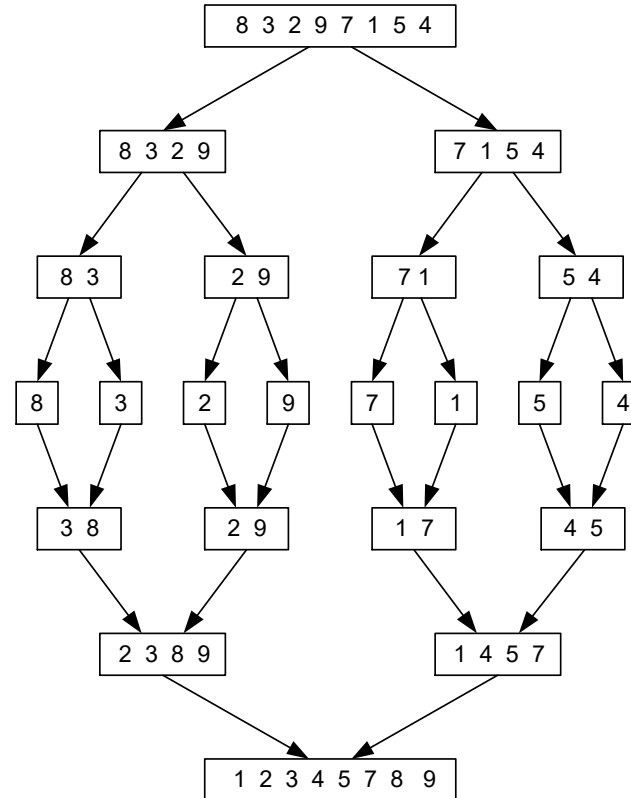
Master Theorem:    If  $a < b^d$ ,     $T(n) \in \Theta(n^d)$   
                              If  $a = b^d$ ,     $T(n) \in \Theta(n^d \log n)$   
                              If  $a > b^d$ ,     $T(n) \in \Theta(n^{\log_b a})$

Using log – Take  $\log_b$  of both sides:

- Case 1:  $\log_b a < d$ ,     $T(n) \in \Theta(n^d)$
- Case 2:  $\log_b a = d$ ,     $T(n) \in \Theta(n^d \log_b n)$
- Case 3:  $\log_b a > d$ ,     $T(n) \in \Theta(n^{\log_b a})$

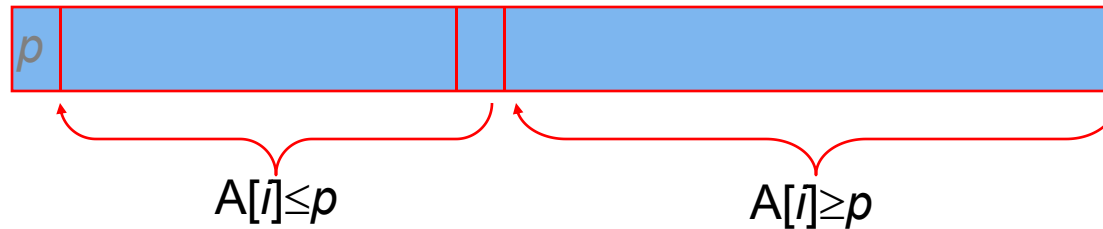


## Mergesort Example



# Quicksort

- ⌚ Select a *pivot* (partitioning element) – here, the first element
- ⌚ Rearrange the list so that all the elements in the first  $s$  positions are smaller than or equal to the pivot and all the elements in the remaining  $n-s$  positions are larger than or equal to the pivot (see next slide for an algorithm)



- ⌚ Exchange the pivot with the last element in the first (i.e.,  $\leq$ ) subarray — the pivot is now in its final position
- ⌚ Sort the two subarrays recursively

ALGORITHM Quicksort( $A[l..r]$ )

//Sorts a subarray by quicksort

//Input: Subarray of array  $A[0..n - 1]$ , defined by its left and right

// indices  $l$  and  $r$

//Output: Subarray  $A[l..r]$  sorted in nondecreasing order

**QUICK\_SORT( $A$ ,  $low$ ,  $high$ ):**

**if  $low < high$ :**

**pivotIndex = PARTITION( $A$ ,  $low$ ,  $high$ )**

**QUICK\_SORT( $A$ ,  $low$ , pivotIndex - 1) # Sort left partition**

**QUICK\_SORT( $A$ , pivotIndex + 1,  $high$ ) # Sort right partition**

**PARTITION(A, low, high):**

**pivot = A[low] # Choosing the first element as pivot**

**i = low + 1**

**j = high**

**while i <= j:**

**while i <= j and A[i] <= pivot:**

**i = i + 1**

**while i <= j and A[j] > pivot:**

**j = j - 1**

**if i < j:**

**swap(A[i], A[j])**

**swap(A[low], A[j])**

**return j**

**# Move pivot to correct position**

**# Return partition index**

# Analysis of Quicksort

## Recurrences:

- Best case:  $T(n) = T(n/2) + \Theta(n)$
- Worst case:  $T(n) = T(n-1) + \Theta(n)$

## Performance:

- Best case: split in the middle —  $\Theta(n \log n)$
- Worst case: sorted array! —  $\Theta(n^2)$
- Average case: random arrays —  $\Theta(n \log n)$

# Analysis of Quicksort

- Problems:
  - Duplicate elements
- Improvements:
  - better pivot selection: median of three partitioning
  - switch to insertion sort on small sub-arrays
  - elimination of recursion
    - These combine to 20-25% improvement
- Improvements: Dual Pivot
- Method of choice for internal sorting of large files ( $n \geq 10000$ )

## Binary Tree Traversals

Binary tree is a divide-and-conquer ready structure!

Ex. 1: Classic traversals (preorder, inorder, postorder)

Algorithm *Inorder*( $T$ )

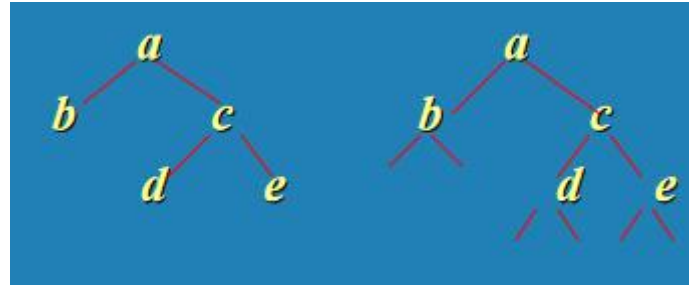
if  $T \neq \emptyset$

*Inorder*( $T_{\text{left}}$ )

    print(root of  $T$ )

*Inorder*( $T_{\text{right}}$ )

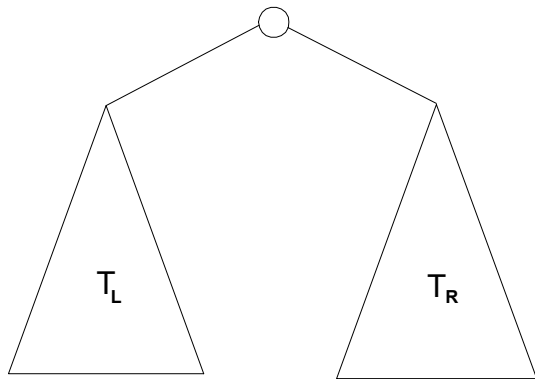
Efficiency:  $\Theta(n)$



## Binary Tree Algorithms (cont.)

Ex. 2: let us consider a recursive algorithm for computing the height of a binary tree.

Can be computed as the maximum of the heights of the root's left and right subtrees plus 1.



**ALGORITHM** Height( $T$ )

//Computes recursively the height of a binary tree

//Input: A binary tree  $T$

//Output: The height of  $T$

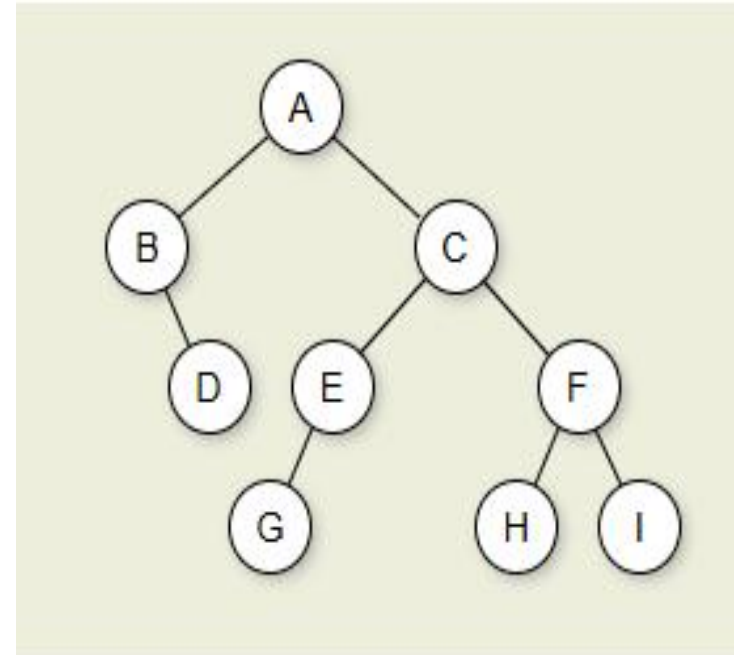
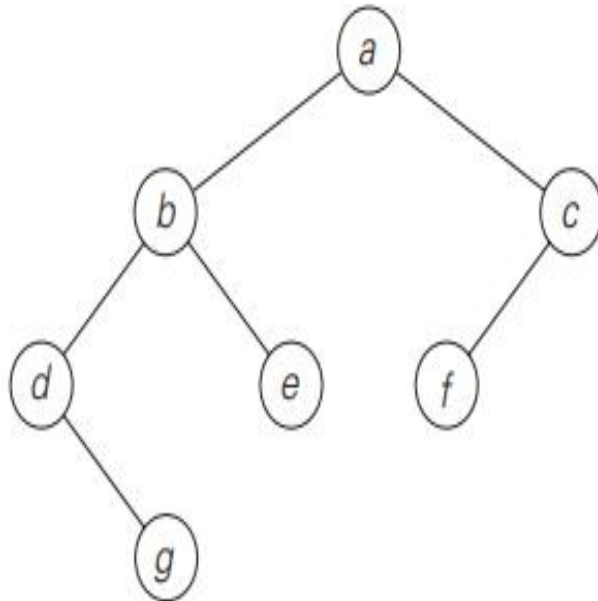
if  $T = \emptyset$  return  $-1$

else return  $\max \{ \text{Height}(T_{\text{left}}), \text{Height}(T_{\text{right}}) \} + 1$

**Efficiency:  $\Theta(n)$**



Write the binary tree traversals for the below tree



## Multiplication of Large Integers

- Some applications like modern cryptography require manipulation of integers that are over 100 decimal digits long.
- such integers are too long to fit in a single word of a modern computer, they require special treatment.
- In the conventional pen-and-pencil algorithm for multiplying two  $n$ -digit integers, each of the  $n$  digits of the first number is multiplied by each of the  $n$  digits of the second number for the total of  $n^2$  digit multiplications.
- The divide-and-conquer method does the above multiplication in less than  $n^2$  digit multiplications.

## Multiplication of Large Integers

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

$A = 12345678901357986429$     $B = 87654321284820912836$

### Divide & Conquer Method

1. Consider elements  $a_1, a_0$  &  $b_1, b_0$ , where all are integers.
2. Recursive Calculation -  
    Calculate  $C_2 = a_1 * b_1$  (product of first halves)  
    Calculate -  $a_0 * b_0$  (second halves)  
    Calculate  $c_1 = (a_1 + a_0) * (b_1 + b_0) - (C_2 + C_0)$
3. Combine results=product  $C = C_2 * 10^n + C_1 * 10^{n/2} + C_0$
4. Repeat the process recursively until  $n=1$ .

## First Divide-and-Conquer Algorithm

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

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

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

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

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

$$\text{Solution: } M(n) = n^2$$

## Strassen's Matrix Multiplication

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

$$\begin{pmatrix} C_{00} & C_{01} \\ C_{10} & C_{11} \end{pmatrix} = \begin{pmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{pmatrix} * \begin{pmatrix} B_{00} & B_{01} \\ B_{10} & B_{11} \end{pmatrix}$$

$$= \begin{pmatrix} M_1 + M_4 - M_5 + M_7 & M_3 + M_5 \\ M_2 + M_4 & M_1 + M_3 - M_2 + M_6 \end{pmatrix}$$

Example:  $M_2 + M_4 = (A_{10} + A_{11}) * B_{00} + A_{11} * (B_{10} - B_{00})$   
 $= A_{10}B_{00} + A_{11}B_{10}$

## Formulas for Strassen's Algorithm

$$M_1 = (A_{00} + A_{11}) * (B_{00} + B_{11})$$

$$M_2 = (A_{10} + A_{11}) * B_{00}$$

$$M_3 = A_{00} * (B_{01} - B_{11})$$

$$M_4 = A_{11} * (B_{10} - B_{00})$$

$$M_5 = (A_{00} + A_{01}) * B_{11}$$

$$M_6 = (A_{10} - A_{00}) * (B_{00} + B_{01})$$

$$M_7 = (A_{01} - A_{11}) * (B_{10} + B_{11})$$

## Analysis of Strassen's Algorithm

If  $n$  is not a power of 2, matrices can be padded with zeros.

Number of multiplications:

$$M(n) = 7M(n/2), \quad M(1) = 1$$

Solution:  $M(n) = 7^{\log_2 n} = n^{\log_2 7} \approx n^{2.807}$  vs.  $n^3$  of brute-force alg.

Algorithms with better asymptotic efficiency are known but they are even more complex.