

Divide-and-conquer is probably the best-known general algorithm design technique. Though its fame may have something to do with its catchy name, it is well deserved: quite a few very efficient algorithms are specific implementations of this general strategy.

Outline of this lecture

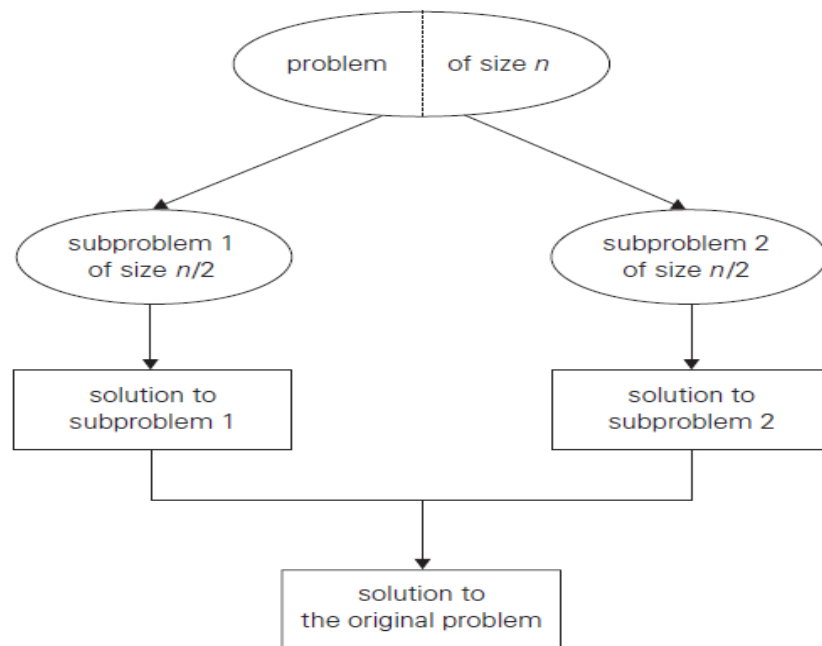
- Principle of Divide and Conquer
- Binary Tree Traversals and Related Properties
- Mergesort
- Quicksort
- The Closest-Pair and Convex-Hull Problems by Divide-and-Conquer
- Multiplication of Large Integers

Principle of Divide and Conquer

Divide-and-conquer algorithms work according to the following general plan:

1. A problem is **divided into several subproblems** of the same type, ideally of about equal size.
2. The **subproblems are solved** (typically recursively, though sometimes a different algorithm is employed, especially when subproblems become small enough).

3. If necessary, **the solutions to the subproblems are combined** to get a solution.



Divide-and-conquer technique (typical case).

As an example, let us consider the **problem of computing the sum of n numbers** a_0, \dots, a_{n-1} .

- If $n > 1$, we can divide the problem into two instances of the same problem:
- compute the sum of the first $n/2$ numbers and
- compute the sum of the remaining $n/2$ numbers.
- Once each of these two sums is computed by applying the same method recursively,
- we can add their values to get the sum in question:

$$a_0 + \dots + a_{n-1} = (a_0 + \dots + a_{\lfloor n/2 \rfloor - 1}) + (a_{\lfloor n/2 \rfloor} + \dots + a_{n-1}).$$

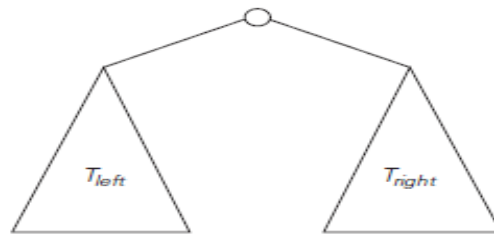
(Of course, if $n = 1$, we simply return a_0 as the answer.)

Binary Tree Traversals and Related Properties

In this section, we see how the divide-and-conquer technique can be applied to **binary trees**.

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.

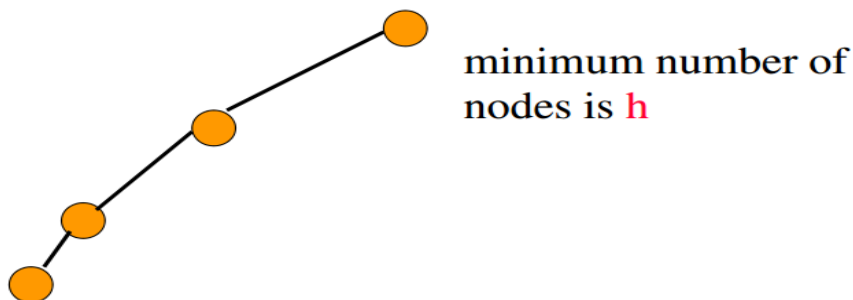
Since the definition itself divides a binary tree into two smaller structures of the same type, the left subtree and the right subtree, many problems about binary trees can be solved by applying the divide-and-conquer technique.



Properties of Binary Trees

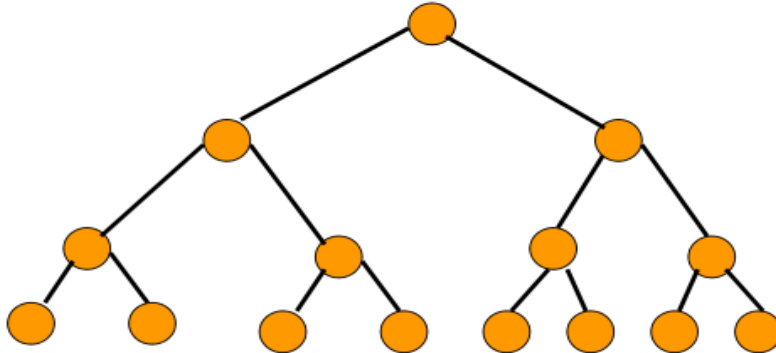
Minimum Number Of Nodes

- Minimum number of nodes in a binary tree whose height is **h** .
- At least one node at each of first **h** levels.



Maximum Number Of Nodes

- All possible nodes at first **h** levels are present.



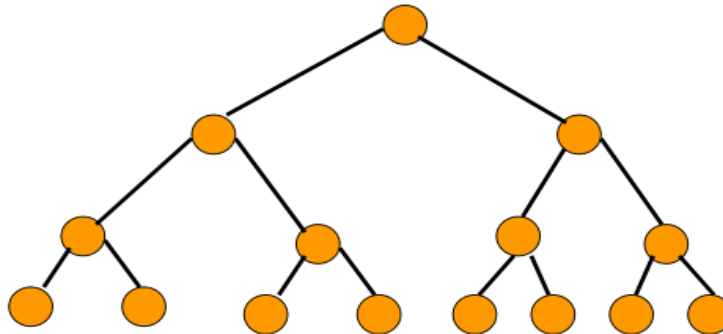
Maximum number of nodes

$$= 1 + 2 + 4 + 8 + \dots + 2^{h-1}$$

$$= 2^h - 1$$

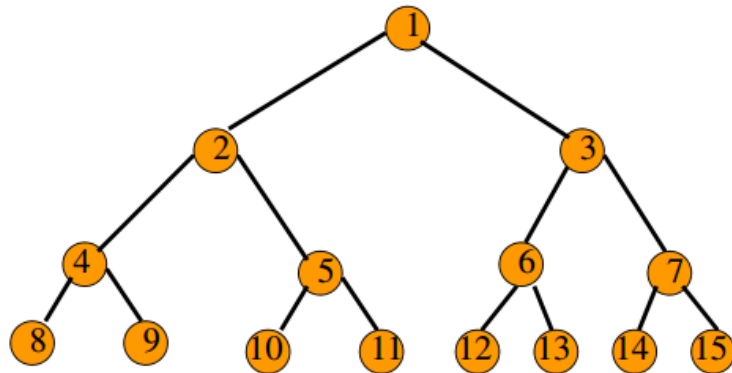
Full Binary Tree

- A full binary tree of a given height **h** has $2^h - 1$ nodes.



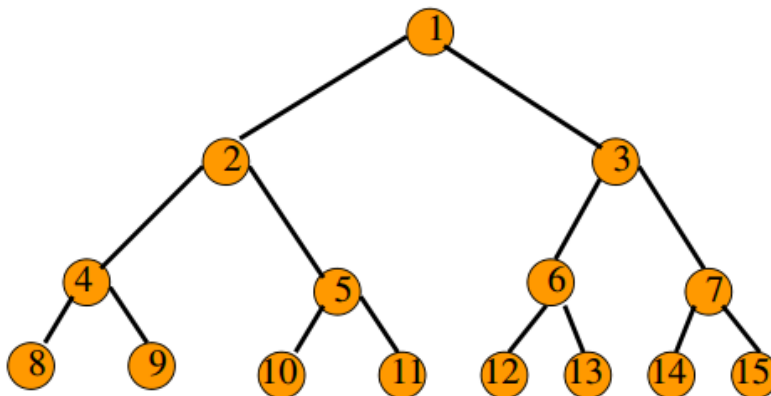
Height **4** full binary tree.

Node Number Properties



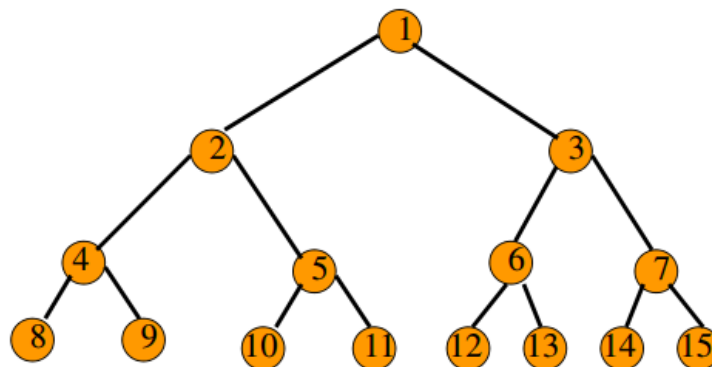
- Parent of node i is node $i / 2$, unless $i = 1$.
- Node 1 is the root and has no parent.

Node Number Properties



- Left child of node i is node $2i$, unless $2i > n$, where n is the number of nodes.
- If $2i > n$, node i has no left child.

Node Number Properties



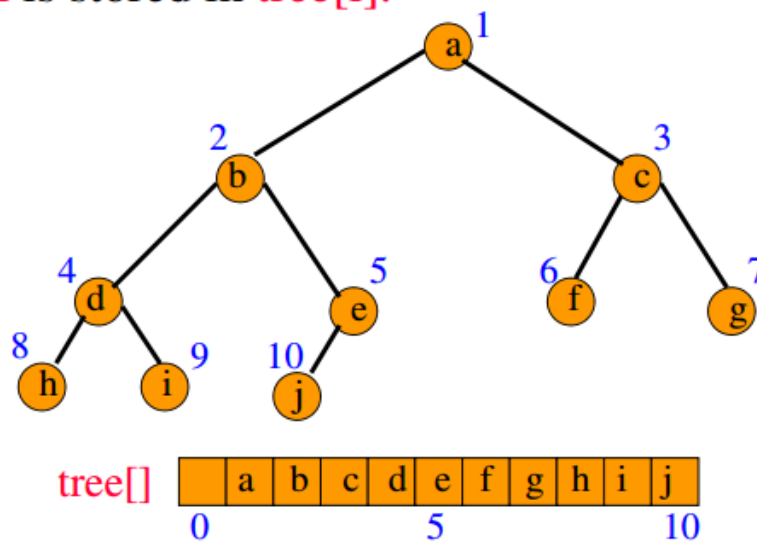
- Right child of node i is node $2i+1$, unless $2i+1 > n$, where n is the number of nodes.
- If $2i+1 > n$, node i has no right child.

Binary Tree Representation

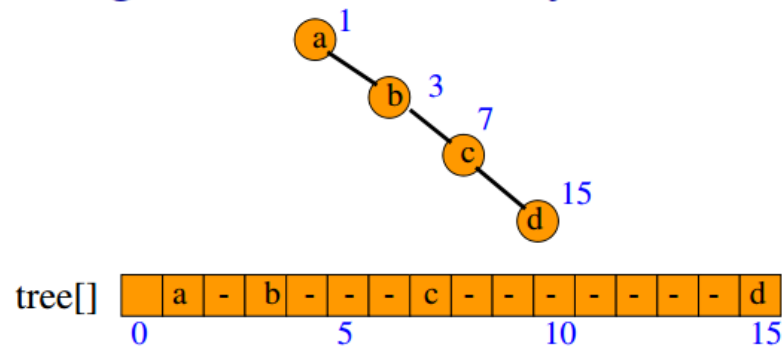
- Array representation.
- Linked representation.

Array Representation

- Number the nodes using the numbering scheme for a full binary tree. The node that is numbered i is stored in $tree[i]$.



Right-Skewed Binary Tree

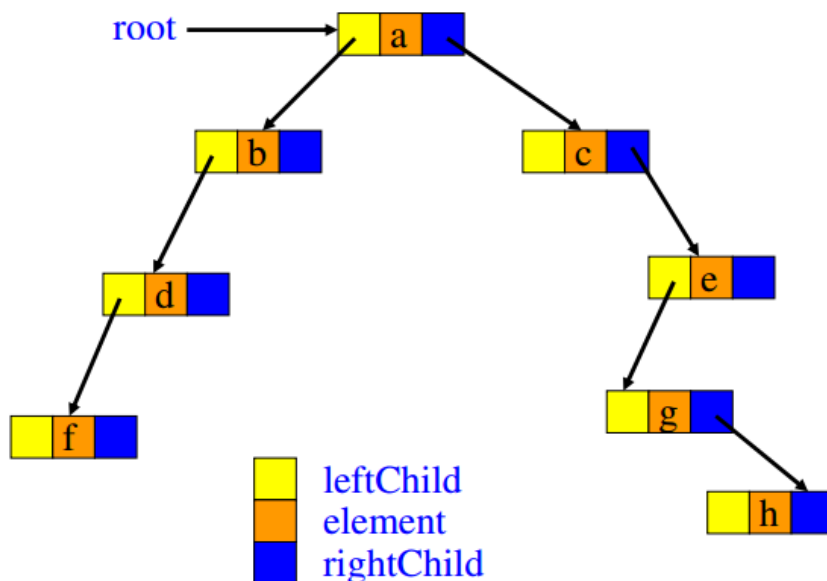


- An **n** node binary tree needs an array whose length is between **n+1** and **2ⁿ**.

Linked Representation

- Each binary tree node is represented as an object whose data type is **binaryTreeNode**.
- The space required by an **n** node binary tree is **n * (space required by one node)**.

Linked Representation Example



Binary Tree Traversal

- Many binary tree operations are done by performing a **traversal** of the binary tree.
- In a traversal, each element of the binary tree is **visited** exactly once.
- During the **visit** of an element, all action (make a clone, display, evaluate the operator, etc.) with respect to this element is taken.

As an example, let us consider a recursive algorithm for computing the height of a binary tree. Recall that the **height is defined as the length of the longest path from the root to a leaf**. Hence, it can be computed as the maximum of the heights of the root's left and right subtrees plus 1.

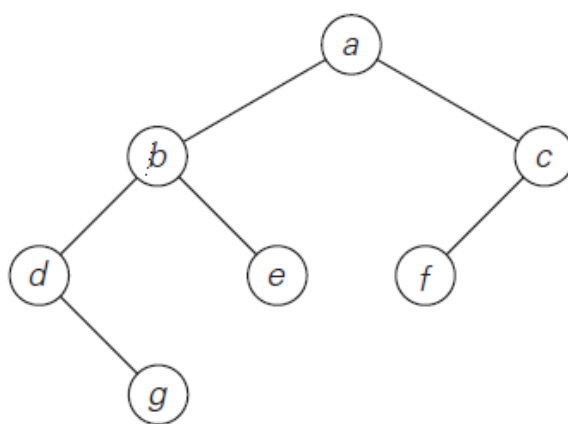
ALGORITHM *Height(T)*

```
//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  $\max\{\text{Height}(T_{\text{left}}), \text{Height}(T_{\text{right}})\} + 1$ 
```

The most important divide-and-conquer algorithms for binary trees are the three classic traversals: **preorder, inorder, and postorder**.

All three traversals visit nodes of a binary tree recursively, i.e., by visiting the tree's root and its left and right subtrees. They differ only by the timing of the root's visit:

- In the preorder traversal, the root is visited before the left and right subtrees are visited (in that order). **Root, Left, Right**
- In the inorder traversal, the root is visited after visiting its left subtree but before visiting the right subtree. **Left, Root, Right**
- In the postorder traversal, the root is visited after visiting the left and right subtrees (in that order). **Left, Right, Root**



preorder: a, b, d, g, e, c, f
inorder: d, g, b, e, a, f, c
postorder: g, d, e, b, f, c, a

Mergesort

Mergesort is a perfect example of a successful application of the divide-and-conquer technique. It sorts a given array $A[0..n-1]$ by dividing it into two halves $A[0..\lfloor n/2 \rfloor - 1]$ and $A[\lfloor n/2 \rfloor..n-1]$, sorting each of them recursively, and then merging the two smaller sorted arrays into a single sorted one.

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

```

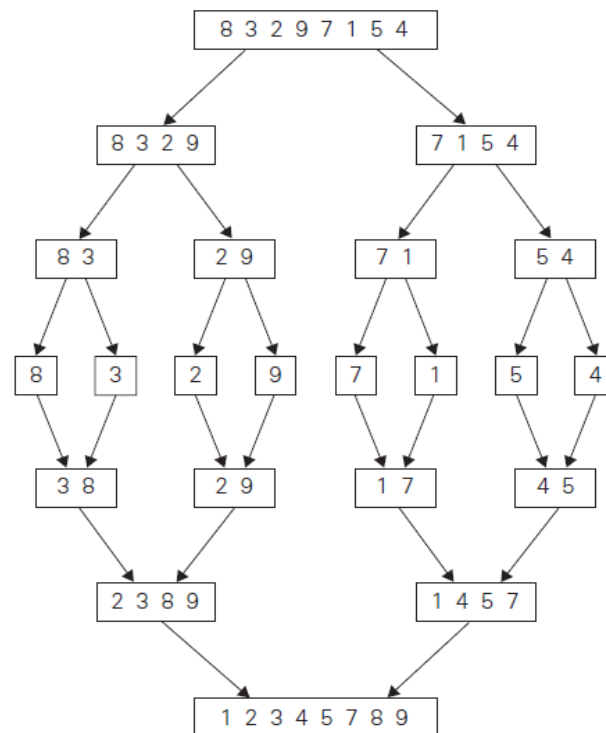
//Sorts array  $A[0..n-1]$  by recursive mergesort
//Input: An array  $A[0..n-1]$  of orderable elements
//Output: Array  $A[0..n-1]$  sorted in nondecreasing order
if  $n > 1$ 
    copy  $A[0..\lfloor n/2 \rfloor - 1]$  to  $B[0..\lfloor n/2 \rfloor - 1]$ 
    copy  $A[\lfloor n/2 \rfloor..n-1]$  to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

The merging of two sorted arrays can be done as follows:

- Two pointers (array indices) are initialized to point to the first elements of the arrays being merged.
- The elements pointed to are compared, and the smaller of them is added to a new array being constructed;
- After that, the index of the smaller element is incremented to point to its immediate successor in the array it was copied from.
- This operation is repeated until one of the two given arrays is exhausted, and then the remaining elements of the other array are copied to the end of the new array.

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 $A[k..p+q-1]$

The operation of the algorithm on the list **8, 3, 2, 9, 7, 1, 5, 4** is illustrated in the following figure:



Home work: Apply mergesort to sort the list E, X, A, M, P, L, E, S in alphabetical order.

Quicksort

Quicksort is the other important sorting algorithm that is based on the divide-and conquer approach. Unlike **mergesort, which divides its input elements according to their position** in the array, **quicksort divides them according to their value**.

A partition is an arrangement of the array's elements so that all the elements to the left of some element $A[s]$ are less than or equal to $A[s]$, and all the elements to the right of $A[s]$ are greater than or equal to it:

$$\underbrace{A[0] \dots A[s-1]}_{\text{all are } \leq A[s]} \quad A[s] \quad \underbrace{A[s+1] \dots A[n-1]}_{\text{all are } \geq A[s]}$$

After a partition is achieved, **A[s]** will be in its final position in the sorted array, and we can continue sorting the two subarrays to the left and to the right of **A[s]** independently (e.g., by the same method).

Note **the difference with mergesort**: there, the division of the problem into two subproblems is immediate and the entire **work happens in combining** their solutions; here, the entire **work happens in the division stage**, with no work required to combine the solutions to the subproblems.

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

if *l* < *r*

s ← *Partition*(A[l..r]) // *s* is a split position

Quicksort(A[l..*s* - 1])

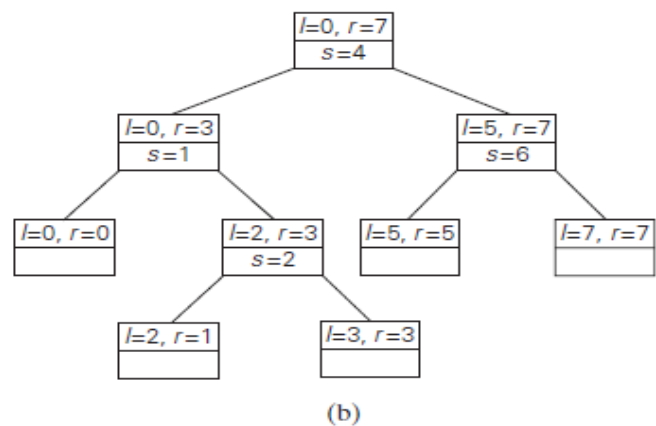
Quicksort(A[*s* + 1..*r*])

We start by selecting a pivot—an element with respect to whose value we are going to divide the subarray. There are several different strategies for selecting a pivot; For now, we use the simplest strategy of selecting the subarray's first element: ***p* = A[l]**.

ALGORITHM *HoarePartition*($A[l..r]$)

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

0	1	2	3	4	5	6	7
5	i 3	1	9	8	2	4	j 7
5	3	1	i 9	8	2	j 4	7
5	3	1	i 4	8	2	j 9	7
5	3	1	4	i 8	j 2	9	7
5	3	1	4	i 2	j 8	9	7
5	3	1	4	j 2	i 8	9	7
2	3	1	4	5	8	9	7
2	i 3	1	j 4				
2	i 3	j 1	4				
2	i 1	j 3	4				
2	j 1	i 3	4				
1	2	3	4				
1		3	i, j 4				
		3	j 4				
			4				
				8	i 9	j 7	
				8	i 7	j 9	
				8	j 7	i 9	
				7	8	9	
				7			
						9	



Next Lecture

Continue ... Divide and Conquer Setratige