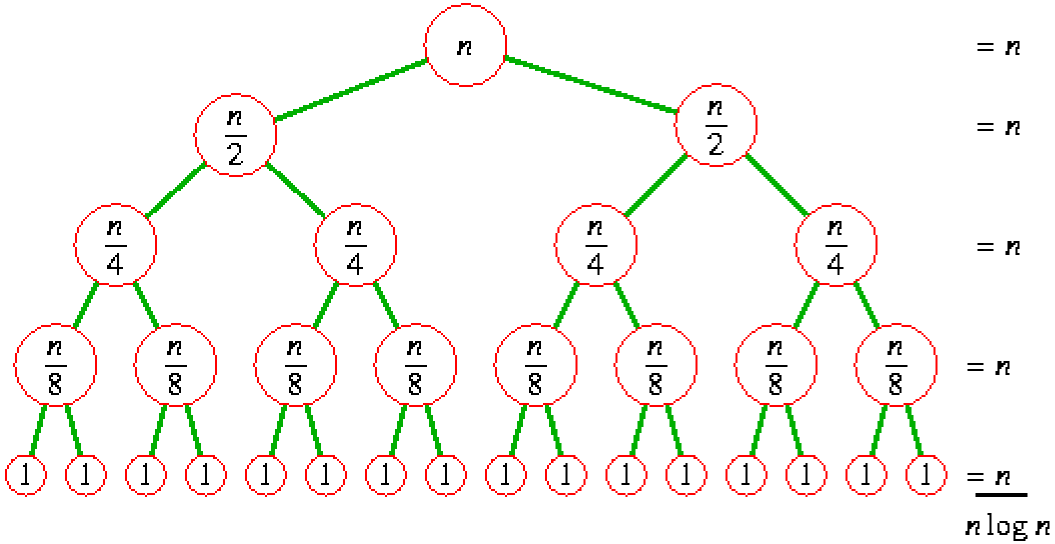
DSE 2256 DESIGN & ANALYSIS OF ALGORITHMS

Lecture 18, 19, 20

Divide-and-Conquer:

Quick Sort
Binary Search
Binary Tree Properties & Traversals



Instructors:

Dr. Savitha G,
Assistant Professor, DSCA, MIT, Manipal

Dr. Abhilash K. Pai,
Assistant Professor, DSCA, MIT, Manipal

Recap of L17

- Divide-and-Conquer
- Master Theorem
- Merge sort using Divide-and-Conquer

Quick sort

- Quick sort divides the input elements according to their position in the array.

- Quick sort works by partitioning an 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:

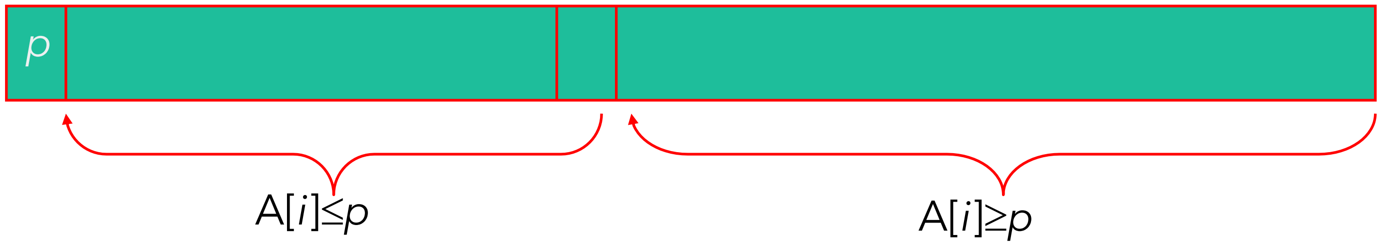
$$\underbrace{A[0]\ldots A[s-1]}_{\text{all are } \le A[s]} \quad A[s] \quad \underbrace{A[s+1]\ldots A[n-1]}_{\text{all are } \ge A[s]}$$

- 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.

Quick sort

- 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



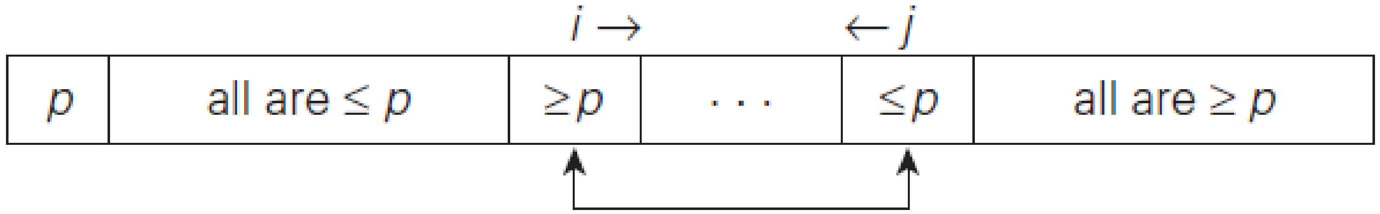
- Exchange the pivot with the last element in the first (i.e., ≤) subarray — the pivot is now in its final position

- Sort the two subarrays recursively

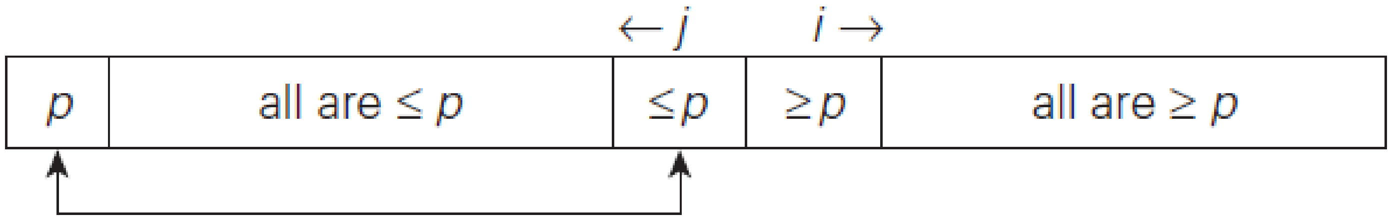
Quick sort

- After both scans stop, three situations may arise, depending on whether or not the scanning indices have crossed :

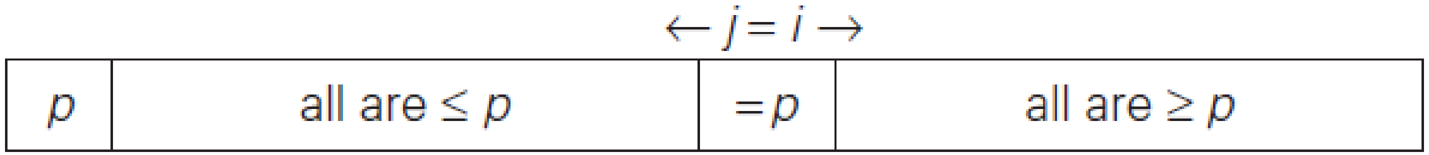
1. i < j



2. i > j



3. i = j



Algorithm for Quicksort

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])
```

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 ← A[l]
i ← l; j ← r + 1
repeat
    repeat i ← i + 1 until A[i] ≥ p  ⎤
    repeat j ← j − 1 until A[j] ≤ p  ⎦ Basic operation
    swap(A[i], A[j])
until i ≥ j
swap(A[i], A[j])  //undo last swap when i ≥ j
swap(A[l], A[j])
return j
```

Analysis of Quicksort

- The number of key comparisons in the **best case** satisfies the recurrence:

$$C_{best}(n) = 2C_{best}(n/2) + n \quad \text{for } n > 1, \quad C_{best}(1) = 0.$$

According to the Master Theorem, $C_{best}(n) \in \Theta(n \log_2 n)$

- The total number of key comparisons for **worst case** will be equal to :

$$C_{worst}(n) = (n+1) + n + \cdots + 3 = \frac{(n+1)(n+2)}{2} - 3 \in \Theta(n^2)$$

- The total number of key comparisons for **average case** will be equal to :

$$C_{avg}(n) \approx 2n \ln n \approx 1.39 n \log_2 n.$$

Better technique for initial pivot selection

⇓

Median-of-three Partitioning

Here, Pivot = Median of Leftmost, Rightmost, Middle element of the i/p array

Binary Search

- Very efficient algorithm for searching in sorted array:

$$\begin{array}{c} K \\ \updownarrow \\ \underbrace{A[0]\ldots A[m-1]}_{\substack{\text{search here if}\\ K<A[m]}} \quad A[m] \quad \underbrace{A[m+1]\ldots A[n-1]}_{\substack{\text{search here if}\\ K>A[m]}}. \end{array}$$

- If $K = A[m]$, stop (successful search).
- Otherwise, continue searching by the same method in $A[0..m\text{-}1]$ if $K < A[m]$ and in $A[m\text{+}1..n\text{-}1]$ if $K > A[m]$

Algorithm for Binary Search

ALGORITHM *BinarySearch*$(A[0..n-1], K)$
//Implements nonrecursive binary search
//Input: An array $A[0..n-1]$ sorted in ascending order and
//   a search key K
//Output: An index of the array's element that is equal to K
//   or -1 if there is no such element
$l \leftarrow 0$; $r \leftarrow n-1$
while $l \le r$ **do**
    $m \leftarrow \lfloor (l+r)/2 \rfloor$
    if $K = A[m]$ **return** m
    else if $K < A[m]$ $r \leftarrow m-1$
    else $l \leftarrow m+1$
return -1

Analysis of Binary Search

- Time efficiency : **worst-case** recurrence:

$$C_{worst}(n) = C_{worst}(\lfloor n/2 \rfloor) + 1 \quad \text{for } n > 1, \quad C_{worst}(1) = 1. \quad (1)$$

$$C_{worst}(2^k) = k + 1 = \log_2 n + 1.$$

$$C_{worst}(n) = \lfloor \log_2 n \rfloor + 1 = \lceil \log_2(n+1) \rceil.$$

can also be obtained by applying Master Theorem to Eq. (1)

- Worst case time efficiency of Binary search is Θ(log n).
- Average case efficiency is slightly less than the worst case : $C_{avg}(n) \approx \log_2 n.$
- Limitations: must be a sorted array
- Bad (degenerate) example of divide-and-conquer because only one of the sub-instances is solved. Rather, it uses a decrease-by-constant-factor strategy.

Binary Tree Travels and Related Properties

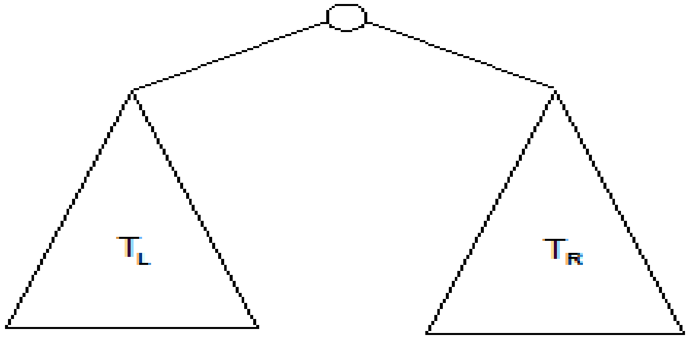
A ***binary tree*** *T* is defined as a finite set of nodes that is either empty or consists of a root and two disjoint binary trees *TL* and *TR* called, respectively, the left and right subtree of the root.

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

Example 1: Computing the height of a binary tree

A tree height is defined as the length of the longest path from the root to a leaf.

Computing : Maximum of the heights of the root's left and right subtrees plus 1.



Algorithm for Computing Height of Binary Tree

ALGORITHM *Height*(T)
    //Computes recursively the height of a binary tree
    //Input: A binary tree T
    //Output: The height of T
    if $T = \varnothing$ **return** -1 → Basic Operation
    else return $\max\{Height(T_{left}), Height(T_{right})\} + 1$

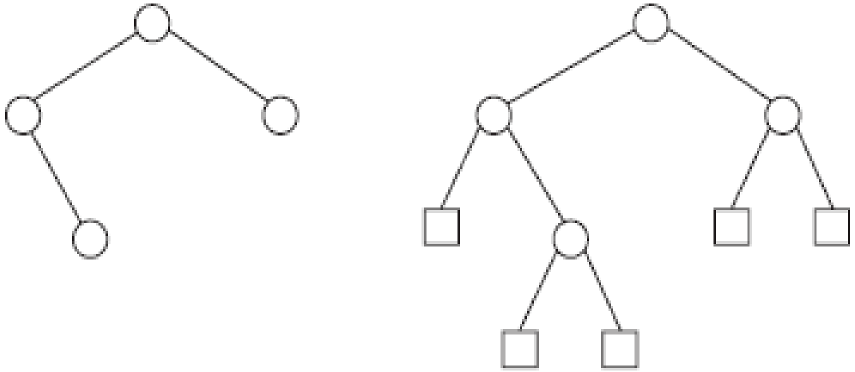
- Problem's instance size is measured by the number of nodes **n(T)** in a given binary tree **T** .

- Number of comparisons made to compute the maximum of two numbers and the number of additions **A(n(T))** made by the algorithm are the same.

$$A(n(T)) = A(n(T_{left})) + A(n(T_{right})) + 1 \quad \text{for } n(T) > 0,$$
$$A(0) = 0.$$

Internal and External nodes

- The analysis of tree algorithms may be done by drawing the tree's extension by replacing the empty subtrees by special nodes.
- The extra nodes (shown by little squares) are called ***external***; the original nodes (shown by little circles) are called ***internal***.



(a) (b)

Analysis : Height of Binary Tree

- The *Height* algorithm makes exactly one addition for every internal node of the extended tree, and it makes one comparison to check whether the tree is empty for every internal and external node.

- Therefore, to ascertain the algorithm's efficiency, we need to know how many external nodes an extended binary tree with n internal nodes can have.

- The number of external nodes $\boldsymbol{x}$ is always **1** more than the number of internal nodes $\boldsymbol{n}$:

$$\mathbf{x = n + 1}.$$

- To prove this equality, consider the total number of nodes, both internal and external. Since every node, except the root, is one of the two children of an internal node, we have the equation : $\boldsymbol{2n + 1 = x + n}$

$\therefore C(n) = 2n + 1$

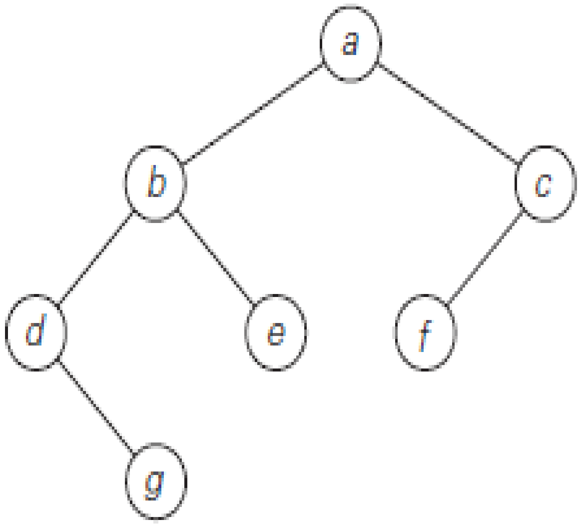
Binary Tree Traversals

- 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.

  - **Preorder traversal:** the root is visited before the left and right subtrees are visited (in that order).
  - **Inorder traversal:** the root is visited after visiting its left subtree but before visiting the right subtree.
  - **Postorder traversal:** the root is visited after visiting the left and right subtrees (in that order).

Binary Tree Traversals

Example :



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

- Efficiency analysis is identical to the above analysis of the ***Height* algorithm** because a recursive call is made for each node of an extended binary tree.

Thank you!

Any queries?