Algorithm Design

(*Divide and Conquer(sorting)-Lecture04*)

Presented
By

Dr. Rajesh Purkait
Asst. Professor
Dept. of CSE ITER
(S`O'A Deemed To be University)
E-mail: rajeshpurkait@soa.ac.in

Outline

- Review of previous sorting algorithm
- What is Divide-and-Conquer?
- Quick Sort Approach
  - Describe Quick Sort Algorithm
    - Divide
    - Conquer/PARTITION
    - Combine
- Analyzing Divide-and Conquer Algorithms
- Time complexity analysis based on partition
- How does partition affect performance?
- Randomized quick sort
- Quick sort in practice

Sorting

- Insertion sort
  - Design approach: incremental
  - Sorts in place: Yes
  - Best case: $\Theta(n)$
  - Worst case: $\Theta(n^2)$

- Bubble Sort
  - Design approach: incremental
  - Sorts in place: Yes
  - Running time: $\Theta(n^2)$

Sorting

- Selection sort
  - Design approach: incremental
  - Sorts in place: Yes
  - Running time: $\Theta(n^2)$

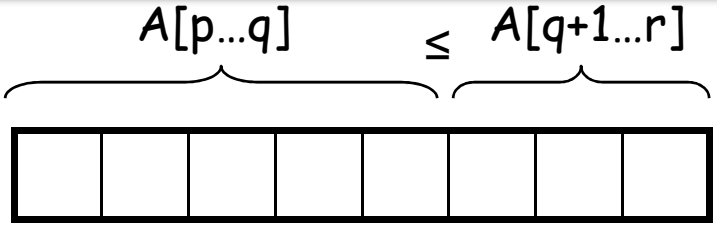
- Merge Sort
  - Design approach: divide and conquer
  - Sorts in place: No O(n)
  - Running time: $O(n \lg n)$

- Quick sort
  - Design approach: divide and conquer
  - Sorts in place: Let's see!!
  - Running time: Let's see!!

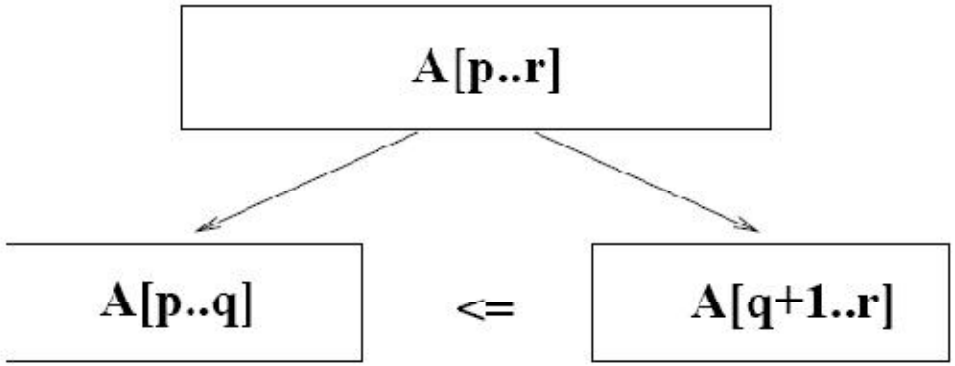
Merge Sort – Discussion

- Running time insensitive of the input
- Advantages:
  - Guaranteed to run in $\Theta(n \lg n)$
- Disadvantage
  - Requires extra space $\approx N$

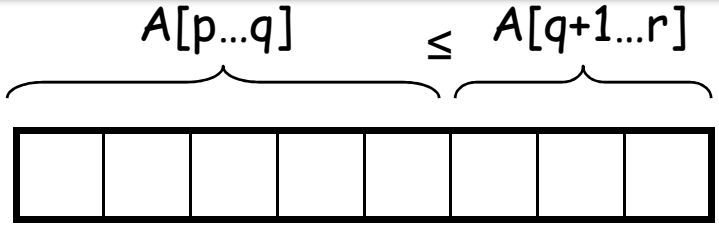
Quicksort



- Sort an array $A[p...r]$
- Divide
  - Partition the array A into 2 subarrays $A[p..q]$ and $A[q+1..r]$, such that each element of $A[p..q]$ is smaller than or equal to each element in $A[q+1..r]$
  - Need to find index q to partition the array



Quicksort



- **Conquer**
  - Recursively sort $A[p..q]$ and $A[q+1..r]$ using Quicksort
- **Combine**
  - Trivial: the arrays are sorted in place
  - No additional work is required to combine them
  - The entire array is now sorted

QUICKSORT

Initially: p=1, r=n

Alg.: QUICKSORT(*A*, p, r)

```
if p < r
   then q ← PARTITION(A, p, r)
        QUICKSORT (A, p, q)
        QUICKSORT (A, q+1, r)
```

Recurrence:

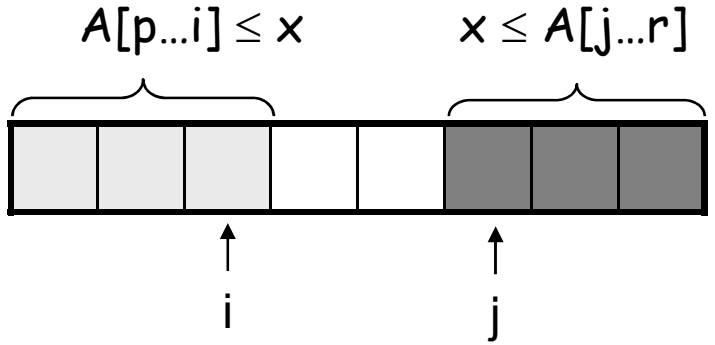
$T(n) = T(q) + T(n - q) + f(n)$ ($f(n)$ depends on PARTITION())

Partitioning the Array

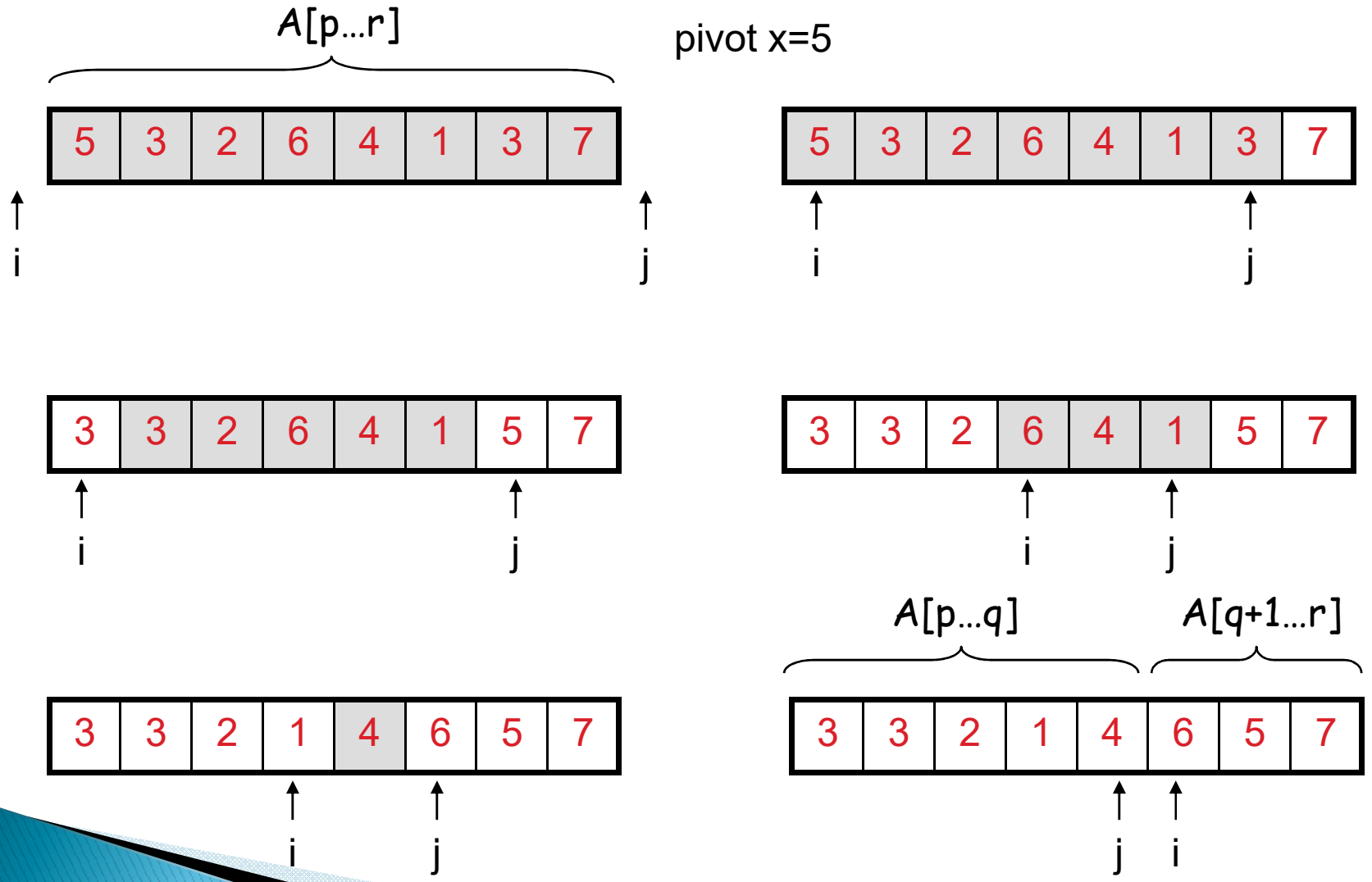
- Choosing PARTITION()
  - There are different ways to do this
  - Each has its own advantages/disadvantages
- Hoare partition
  - Select a pivot element **x** around which to partition
  - Grows two regions

    $A[p...i] \leq x$

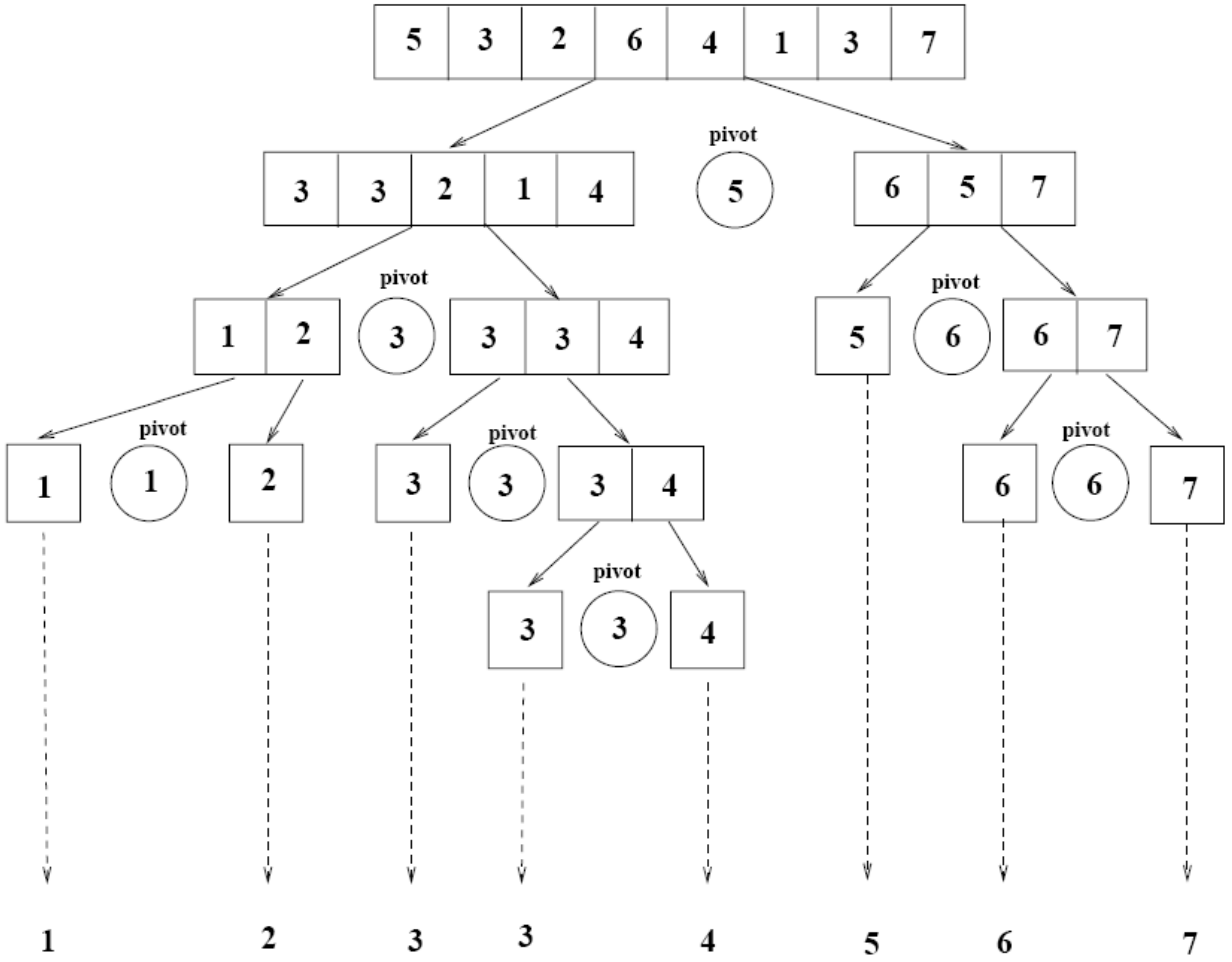
    $x \leq A[j...r]$



Example


A[p…r]
pivot x=5
5 3 2 6 4 1 3 7
i
j
5 3 2 6 4 1 3 7
i
j
3 3 2 6 4 1 5 7
i
j
3 3 2 6 4 1 5 7
i
j
A[p…q]
A[q+1…r]
3 3 2 1 4 6 5 7
i
j
3 3 2 1 4 6 5 7
j
i

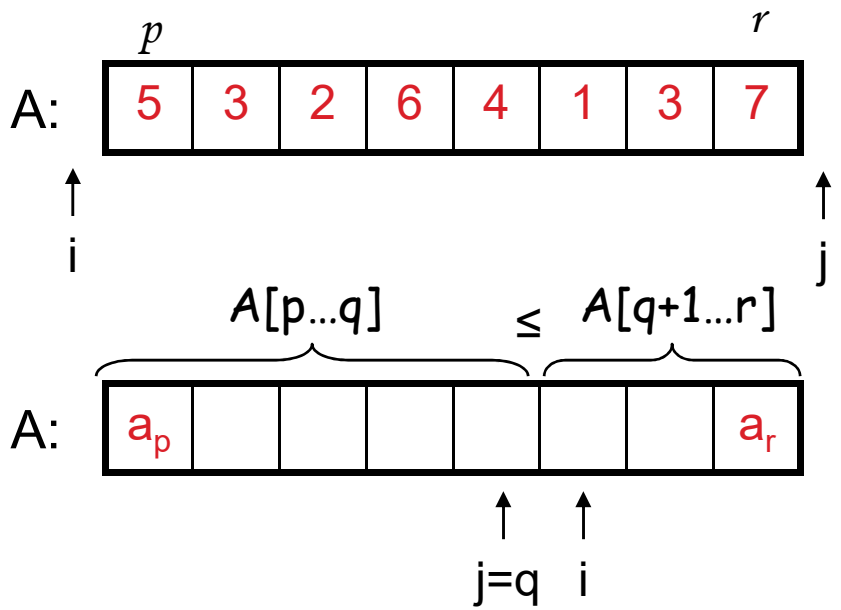
Example


5 3 2 6 4 1 3 7
pivot
5
3 3 2 1 4
6 5 7
pivot
3
1 2
3 3 4
pivot
6
5
6 7
pivot
1
1
2
3
3
3 4
6
6
7
pivot
pivot
3
3
4
1 2 3 3 4 5 6 7

Partitioning the Array

Alg. PARTITION (A, p, r)

1. $x \leftarrow A[p]$
2. $i \leftarrow p - 1$
3. $j \leftarrow r + 1$
4. while TRUE
5. do repeat $j \leftarrow j - 1$
6. until $A[j] \leq x$
7. do repeat $i \leftarrow i + 1$
8. until $A[i] \geq x$
9. if $i < j$
10. then exchange $A[i] \leftrightarrow A[j]$
11. else return j



Each element is visited once!

Running time: $\Theta(n)$
$n = r - p + 1$

Recurrence

Initially: p=1, r=n

Alg.: QUICKSORT(*A*, p, r)

if p < r

then *q* ← PARTITION(*A*, p, r)

QUICKSORT (*A*, p, *q*)

QUICKSORT (*A*, *q*+1, r)

Recurrence:

$$T(n) = T(q) + T(n - q) + n$$

Analyzing Divide-and Conquer Algorithms

- The recurrence is based on the three steps of the paradigm:
  - T(n) – running time on a problem of size n
  - **Divide** the problem into **a** subproblems, each of size **n/b**: takes D(n)
  - **Conquer** (solve) the subproblems aT(n/b)
  - **Combine** the solutions C(n)

$$T(n) = \begin{cases} \Theta(1) & \text{if } n \le c \\ aT(n/b) + D(n) + C(n) & \text{otherwise} \end{cases}$$

QUICK-SORT Running Time

- **Divide:**
  - compute q as per the partition method: $D(n) = \Theta(n)$
- **Conquer:**
  - recursively solve 2 sub-problems, each of size $\Rightarrow$ $T(q)$ and $T(n - q)$
- **Combine:**
  - No need for combination. $\Rightarrow$ $C(n) = 0$

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

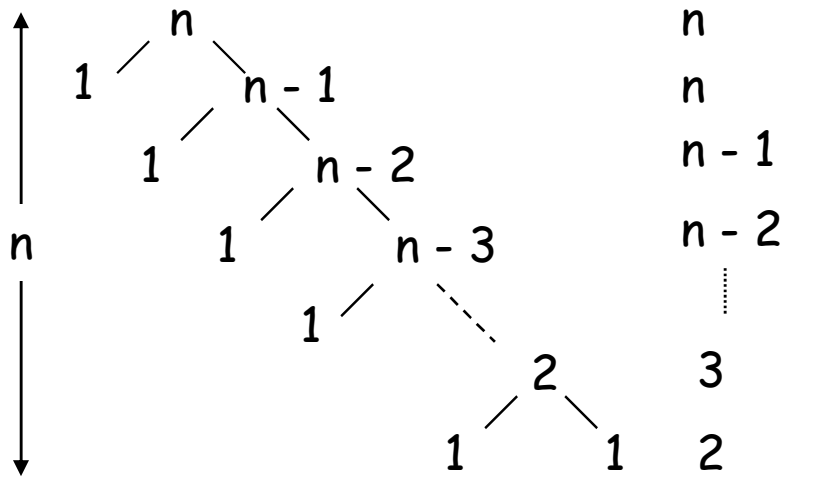
Worst Case Partitioning

- Worst-case partitioning
  - One region has one element and the other has n – 1 elements
  - Maximally unbalanced
- Recurrence: q=1

$T(n) = T(1) + T(n-1) + n,$

$T(1) = \Theta(1)$

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



$$= n + \left(\sum_{k=1}^{n} k\right) - 1 = \Theta(n) + \Theta(n^2) = \Theta(n^2)$$

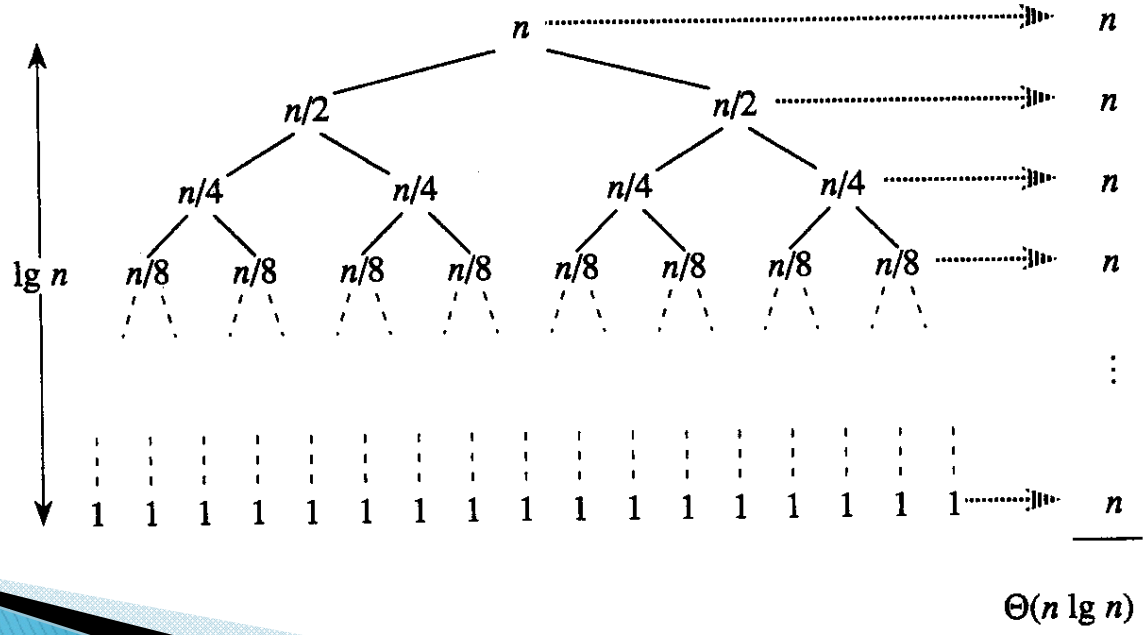
When does the worst case happen?

Best Case Partitioning

- Best-case partitioning
  - Partitioning produces two regions of size n/2
- Recurrence: q=n/2

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

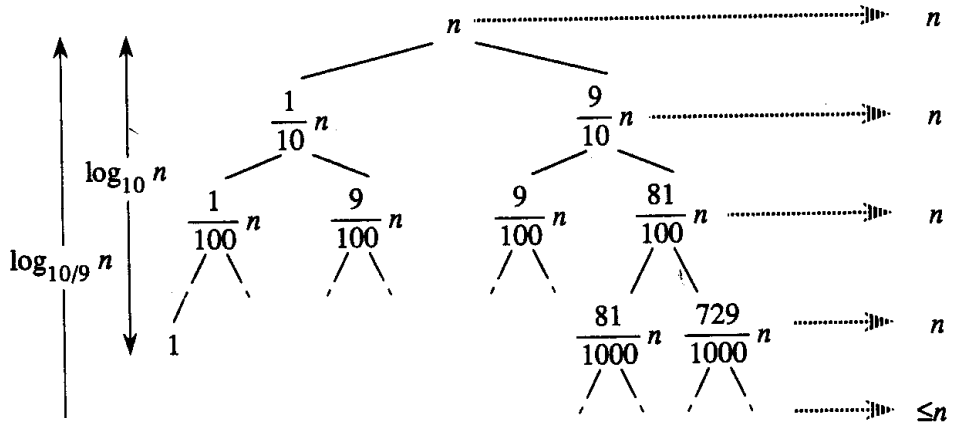
  $T(n) = \Theta(n \lg n)$ (Master theorem)



Case Between Worst and Best

- 9-to-1 proportional split

Q(n) = Q(9n/10) + Q(n/10) + n



- Using the recursion tree:

longest path: $Q(n) \le n \sum_{i=0}^{\log_{10/9} n} 1 = n(\log_{10/9} n + 1) = c_2 nlgn$

shortest path: $Q(n) \ge n \sum_{i=0}^{\log_{10} n} 1 = n \log_{10} n = c_1 nlgn$

Thus, $Q(n) = \Theta(nlgn)$

How does partition affect performance?

- **Any splitting of constant proportionality** yields $\Theta(nlgn)$ time !!!

- Consider the $(1 : n-1)$ splitting:

$$\text{ratio}=1/(n-1) \text{ not a constant !!!}$$

- Consider the $(n/2 : n/2)$ splitting:

$$\text{ratio}=(n/2)/(n/2) = 1 \text{ it is a constant !!}$$

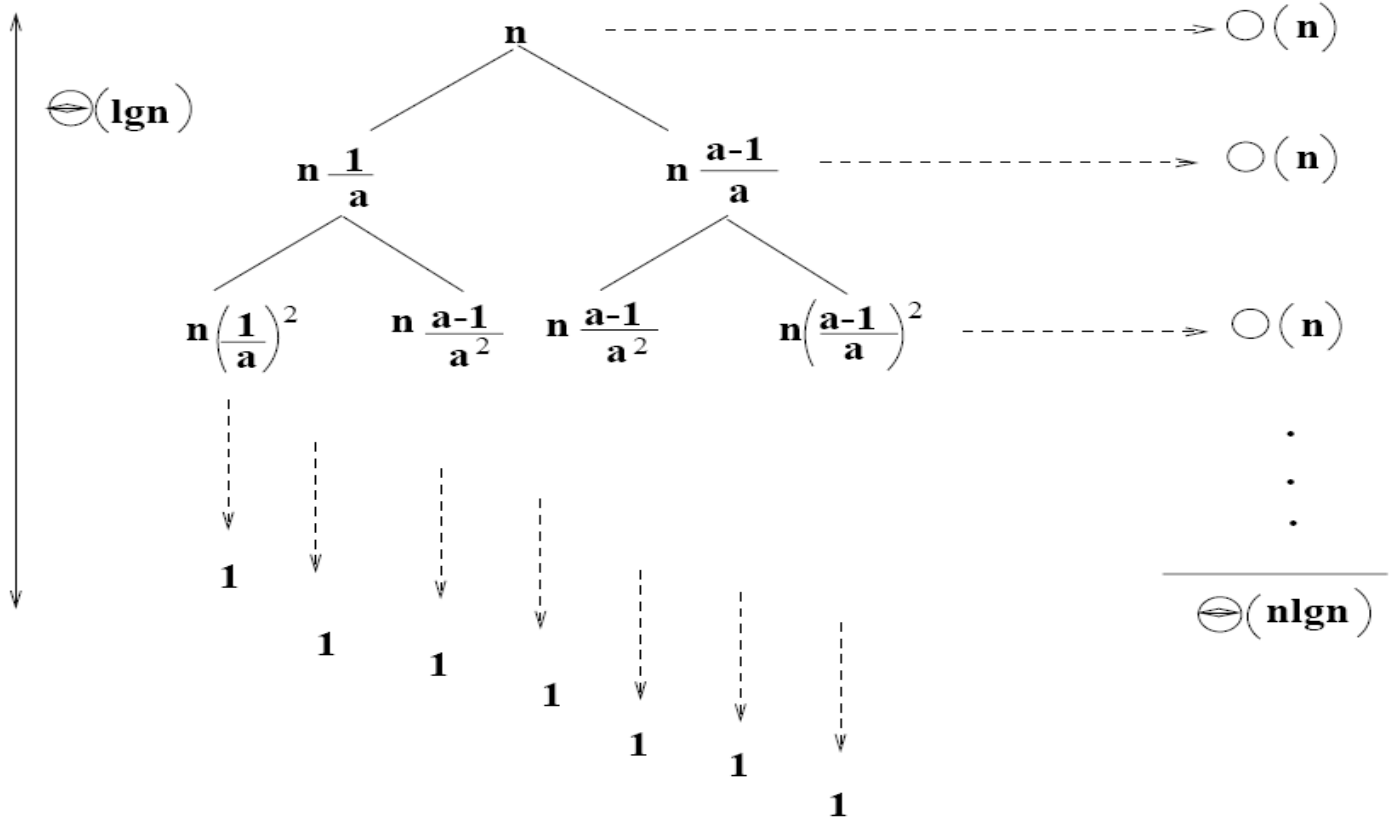
- Consider the $(9n/10 : n/10)$ splitting:

$$\text{ratio}=(9n/10)/(n/10) = 9 \text{ it is a constant !!}$$

How does partition affect performance?

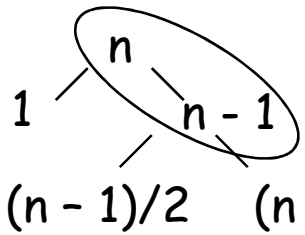
- Any $((a-1)n/a : n/a)$ splitting:

  ratio=$((a-1)n/a)/(n/a) = a-1$ it is a constant !!

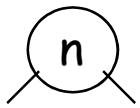


Performance of Quicksort

- Average case
  - All permutations of the input numbers are equally likely
  - On a random input array, we will have a mix of well balanced and unbalanced splits
  - Good and bad splits are randomly distributed across throughout the tree



combined partitioning cost:
2n-1 = Θ(n)



partitioning cost:
n = Θ(n)

Alternate of a good and a bad split

Nearly well balanced split

- Running time of Quicksort when levels alternate between good and bad splits is $O(n \lg n)$

Randomized Quicksort

IDEA: **Partition around a random element.**

- Running time is independent of the input order.
- No assumption need to be made about the input distribution.
- No specific input elicits the worst case behavior.
- The worst case is determined only by the output of a random-number generator.

Randomized Quicksort: Algorithm

RANDOMIZED-PARTITION(A, p, r)

```
1  i ← RANDOM(p, r)
2  exchange A[p] ↔ A[i]
3  return PARTITION(A, p, r)
```

We now make the new quicksort call RANDOMIZED-PARTITION in place of PARTITION:

RANDOMIZED-QUICKSORT(A, p, r)

```
1  if p < r
2     then q ← RANDOMIZED-PARTITION(A, p, r)
3          RANDOMIZED-QUICKSORT(A, p, q)
4          RANDOMIZED-QUICKSORT(A, q + 1, r)
```

We analyze this algorithm in the next section.

Quicksort in practice

- Quicksort is a great general-purpose sorting algorithm.
- Quicksort is typically over twice as fast as merge sort.
- Quicksort can benefit substantially from code tuning.
- Quicksort behaves well even with caching and virtual memory.

Thank You