BITS Pilani
Hyderabad Campus

Data Structures and Algorithms Design

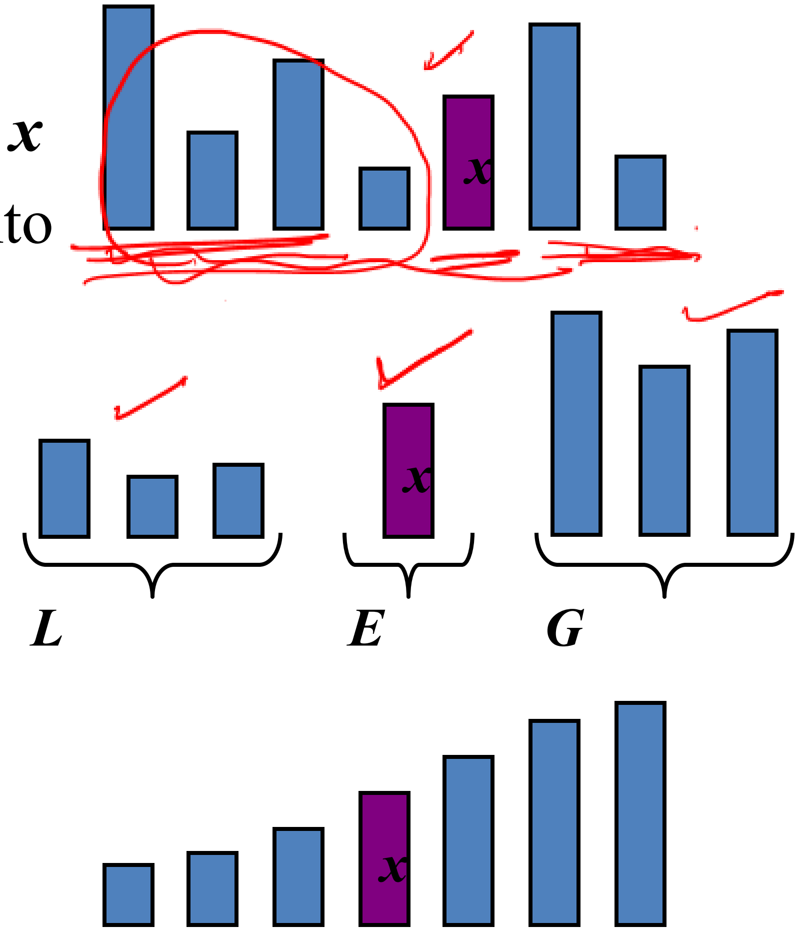
Febin.A.Vahab
2019-20

ONLINE SESSION 11 -PLAN

Sessions(#)	List of Topic Title	Text/Ref Book/external resource
12	Quick Sort Algorithm.	**T1: 5.2, 4.1, 4.3**

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

innovate achieve lead

low, high, pivot

0	1	2	3	4	5	6	7	8
44	75	23	43	55	12	64	77	33
	i							j

Is low ≤ high ? Yes

i ← low + 1 ← 1

j ← high ← 8

pivot ← a[low] ← 44

a) a[i] ≤ pivot ? / pivot > a[i] ?

44 > 75 ? No

b) a[j] > pivot ?

33 > 44 ? No

c) Is i ≤ j ? 1 ≤ 8 ? Yes exchange a[i] and a[j]

75 ↔ 33

0	1	2	3	4	5	6	7	8
44	33	23	43	55	12	64	77	75
	~~i~~	~~i~~	~~i~~	i	j	~~j~~	~~j~~	~~j~~

a) pivot > a[i] ?

44 > 33 Yes

$i \leftarrow i{+}{+} \leftarrow 2$

pivot > a[i]

44 > 23 Yes

$i \leftarrow i+1 \leftarrow 3$

pivot > a[i]

44 > 43 Yes

$i \leftarrow i+1 \leftarrow 4$

pivot > a[i]

44 > 55 No

b) a[j] > pivot ?

75 > 44 ? Yes

$j{-}{-} \leftarrow 7$

a[j] > pivot ?

77 > 44 ? Yes

$j{-}{-} \leftarrow 6$

a[j] > pivot ?

64 > 44 ? Yes

$j{-}{-} \leftarrow 5$

a[j] > pivot ?

12 > 44 ? No

c) Is $i \leq j$? $4 \leq 5$? Yes

$55 \leftrightarrow 12$ //

innovate achieve lead

0	1	2	3	4	5	6	7	8
44	33	23	43	12	55	64	77	75
				i j	i j			

a) pivot > a[i] ?

44 > 12 ? Yes

$i \leftarrow i+1 = 5$

pivot > a[i] ?

44 > 55 ? No

b) a[j] > pivot ?

55 > 44 ? Yes

$j-- \leftarrow 4$

a[j] > pivot ?

12 > 44 ? No

c) Is $i \le j$? $5 \le 4$? No

Exchange a[j] and pivot

$12 \leftrightarrow \underline{44}$

$\underbrace{12 \quad 33 \quad 23 \quad 43}$ $\boxed{44}$ $\underbrace{55 \quad 64 \quad 77 \quad 75}$

77

Partition

- We partition an input sequence as follows:
  - We remove, in turn, each element y from S and
  - We insert y into L, E or G, depending on the result of the comparison with the pivot x
- Each insertion and removal is at the beginning or at the end of a sequence, and hence takes $O(1)$ time
- Thus, the partition step of quick-sort takes $O(n)$ time

Partition

Algorithm *partition*(*S*, *p*)
  Input sequence *S*, position *p* of pivot
  Output subsequences *L*, *E*, *G* of the
    elements of *S* less than, equal to,
    or greater than the pivot, resp.
  $L, E, G \leftarrow$ empty sequences
  $x \leftarrow S.remove(p)$
  while $\neg S.isEmpty()$
    $y \leftarrow S.remove(S.first())$
    if $y < x$
      $L.insertLast(y)$
    else if $y = x$
      $E.insertLast(y)$
    else { $y > x$ }
      $G.insertLast(y)$
  return *L*, *E*, *G*

Quick-Sort

innovate achieve lead

Algorithm inPlaceQuickSort(S, a, b):

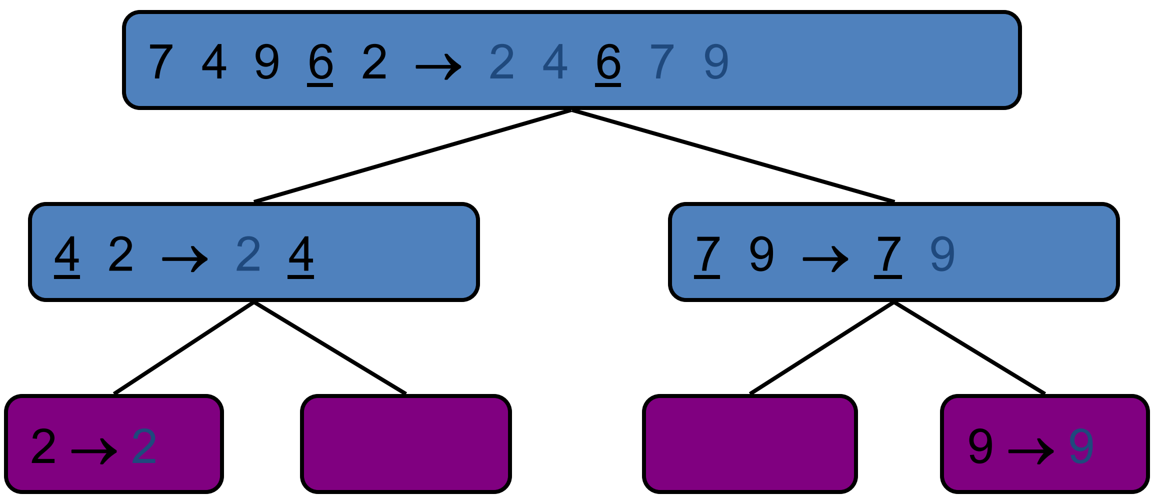
Input: Sequence S of distinct elements; integers a and b

Output: Sequence S with elements originally from ranks from a to b, inclusive, sorted in nondecreasing order from ranks a to b

```
if a ≥ b then return            {empty subrange}
p ← S.elemAtRank(b)             {pivot}
l ← a            {will scan rightward}
r ← b − 1            {will scan leftward}
while l ≤ r do
    {find an element larger than the pivot}
    while l ≤ r and S.elemAtRank(l) ≤ p do
        l ← l + 1
    {find an element smaller than the pivot}
    while r ≥ l and S.elemAtRank(r) ≥ p do
        r ← r − 1
    if l < r then
        S.swapElements(S.atRank(l), S.atRank(r))
{put the pivot into its final place}
S.swapElements(S.atRank(l), S.atRank(b))
{recursive calls}
inPlaceQuickSort(S, a, l − 1)
inPlaceQuickSort(S, l + 1, b)
```

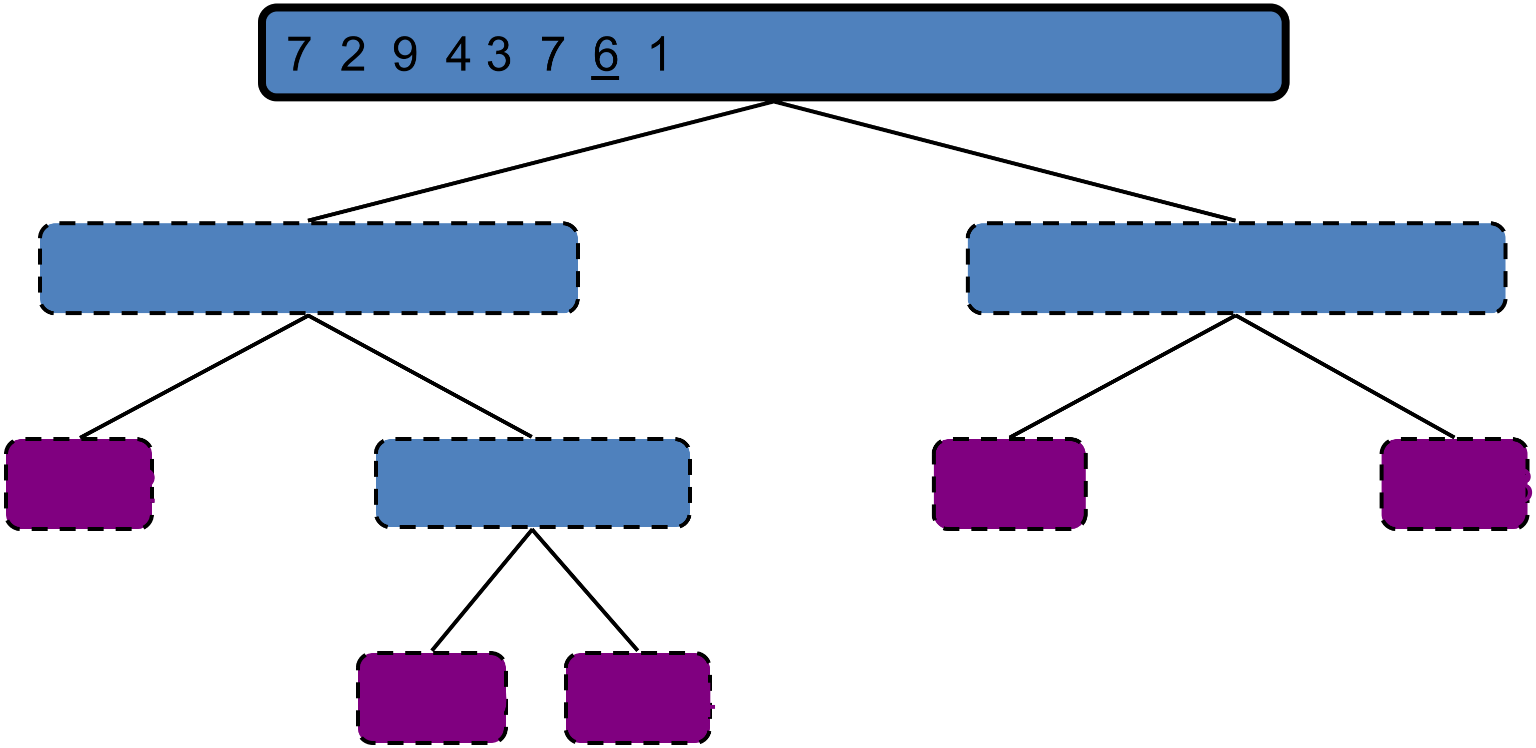
Quick-Sort Tree

- An execution of quick-sort is depicted by a binary tree
  - Each node represents a recursive call of quick-sort and stores
    - Unsorted sequence before the execution and its pivot
    - Sorted sequence at the end of the execution
  - The root is the initial call
  - The leaves are calls on subsequences of size 0 or 1



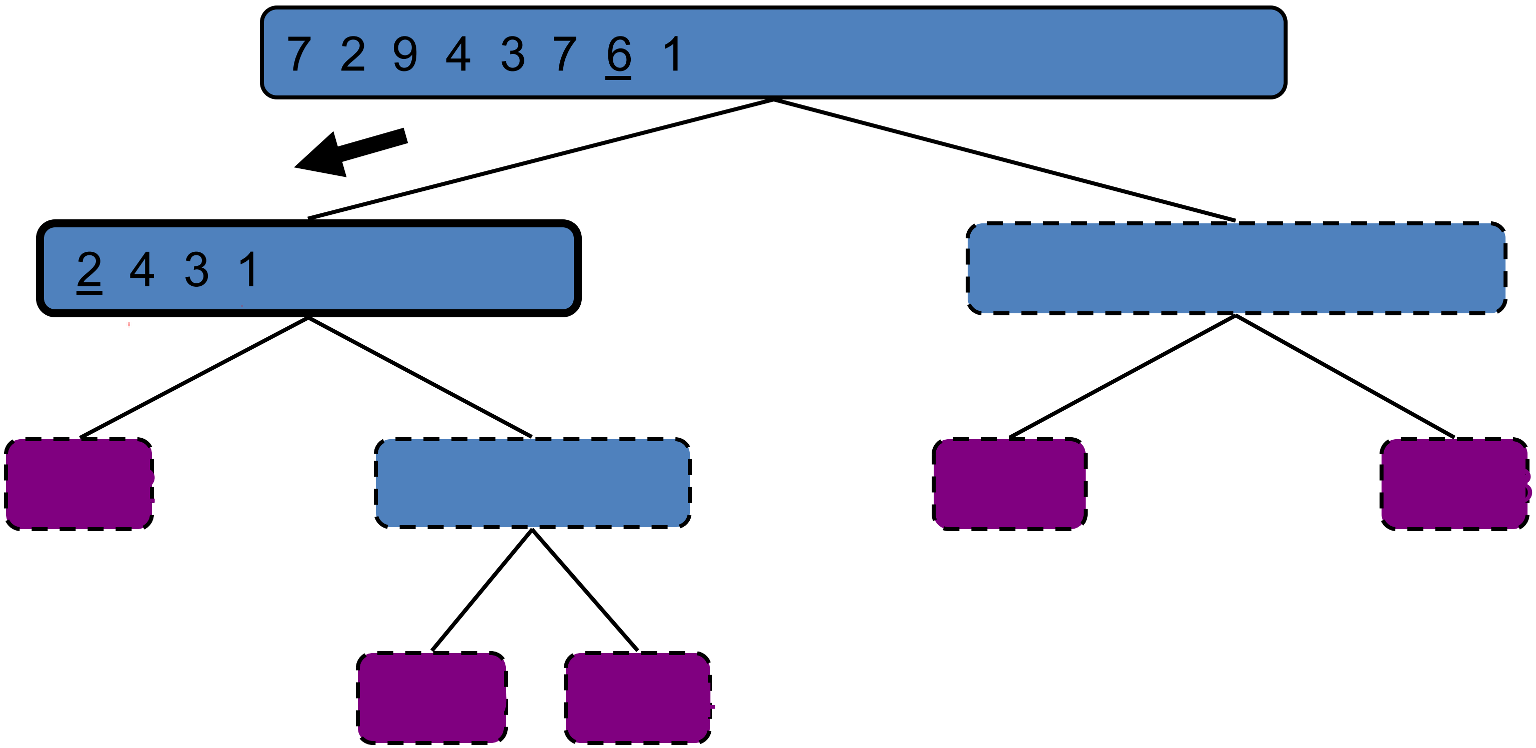
Execution Example

- Pivot selection



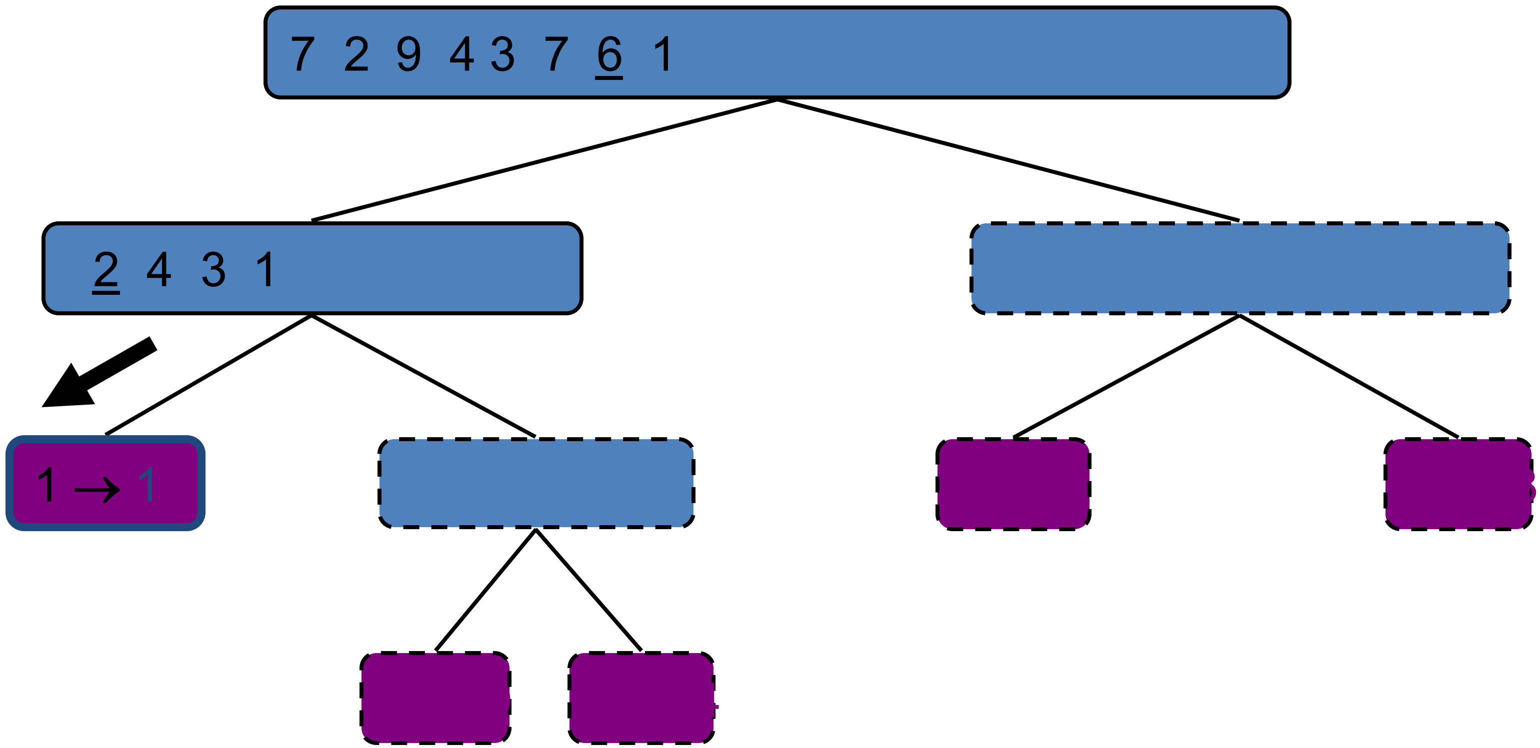
Execution Example (cont.)

- Partition, recursive call, pivot selection



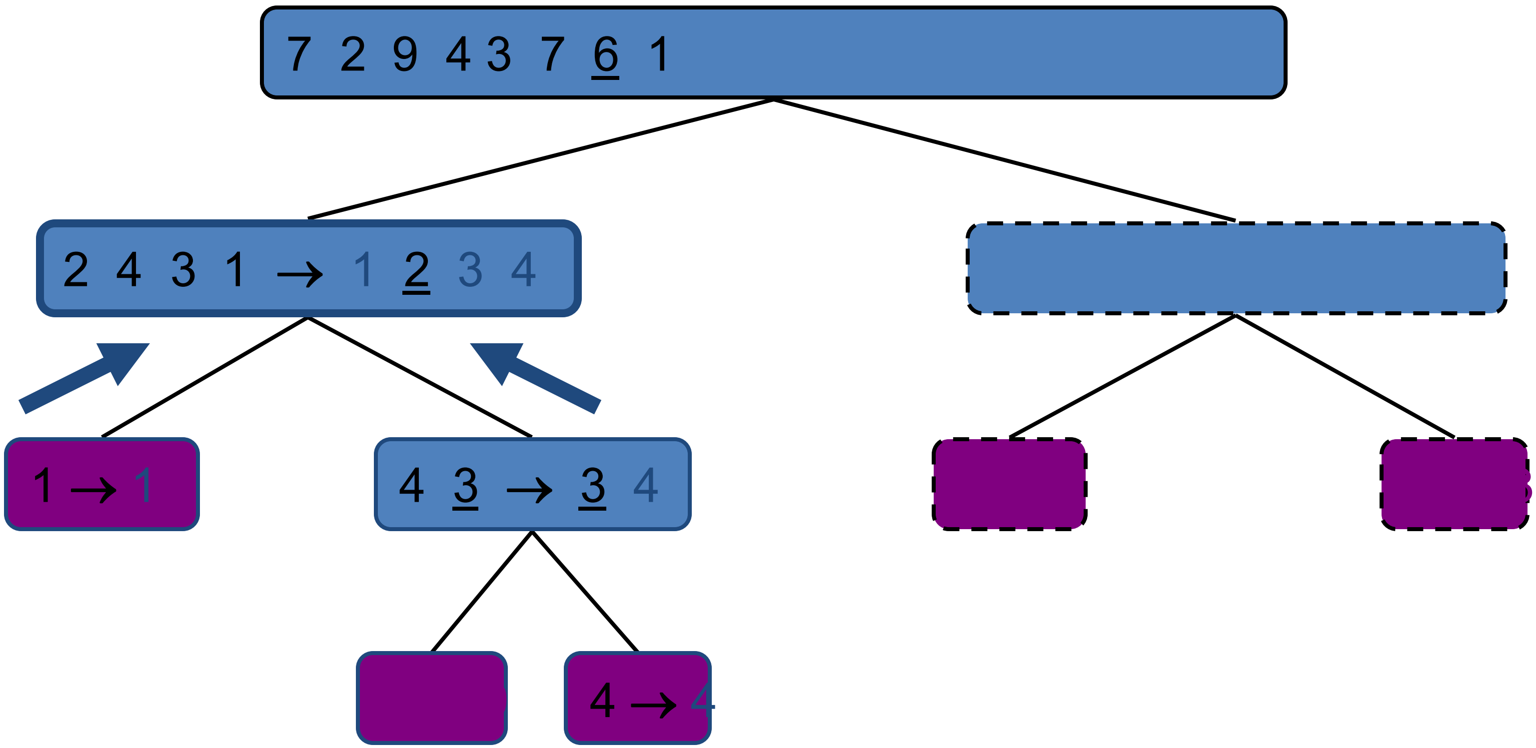
Execution Example (cont.)

- Partition, recursive call, base case



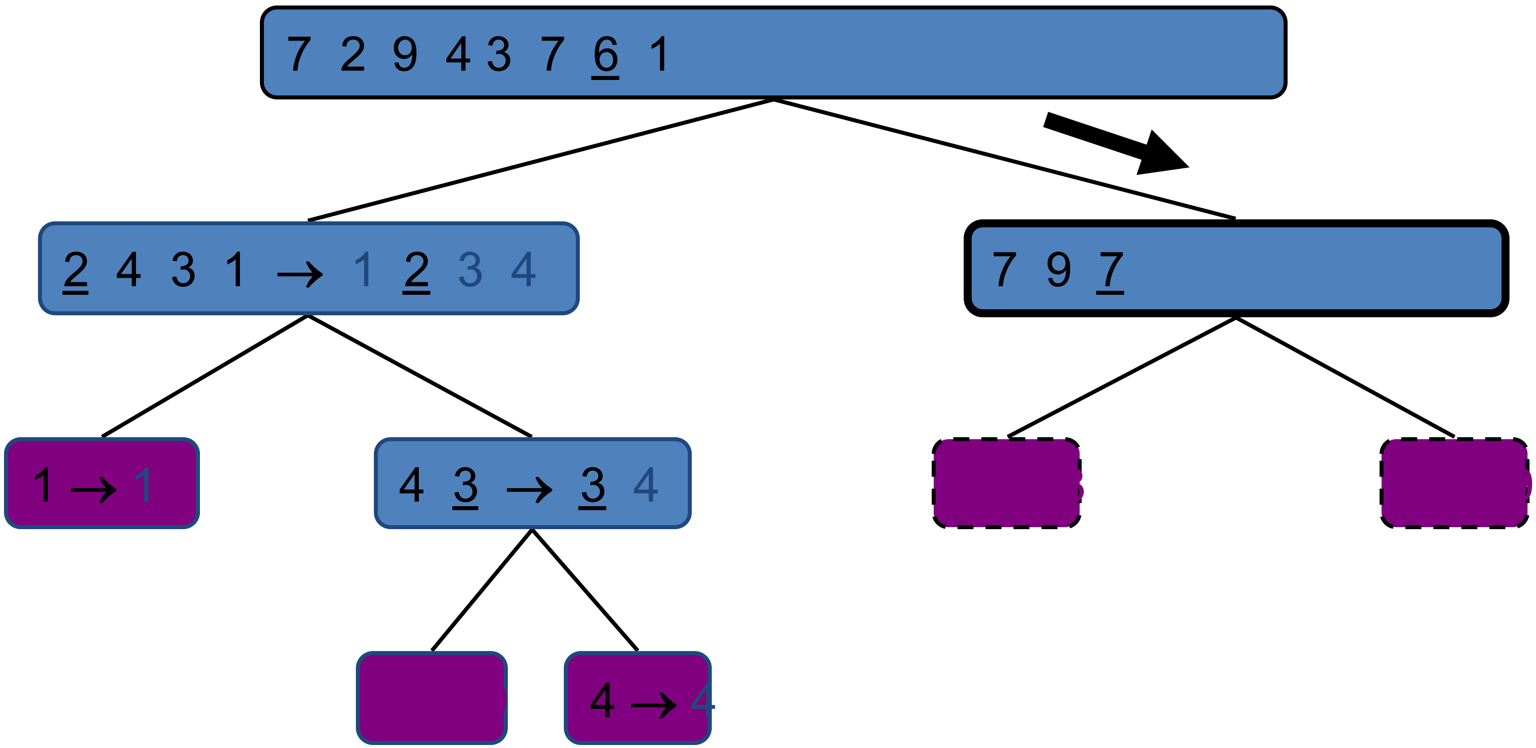
Execution Example (cont.)

- Recursive call, …, base case, join



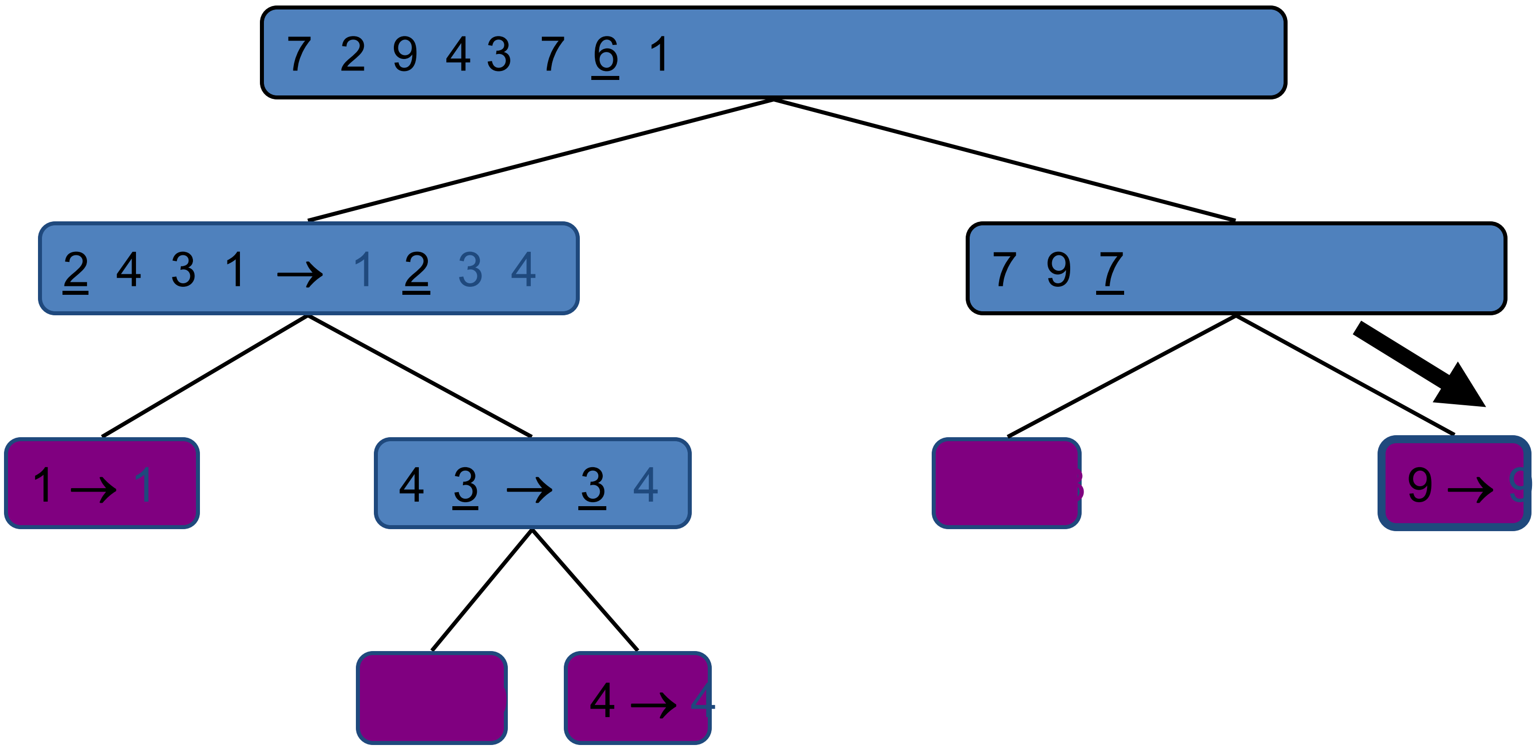
Execution Example (cont.)

- Recursive call, pivot selection



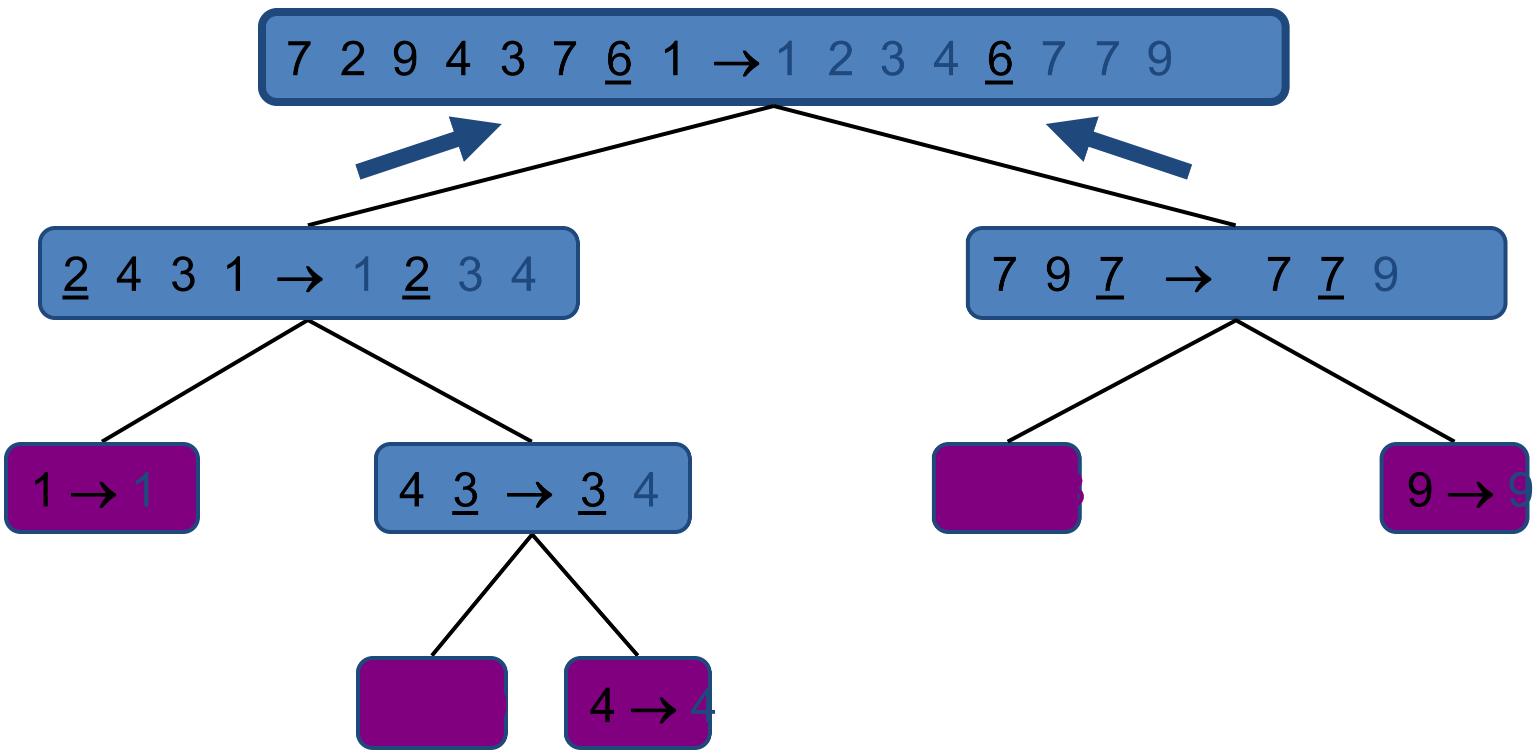
Execution Example (cont.)

- Partition, …, recursive call, base case



Execution Example (cont.)

- Join, join



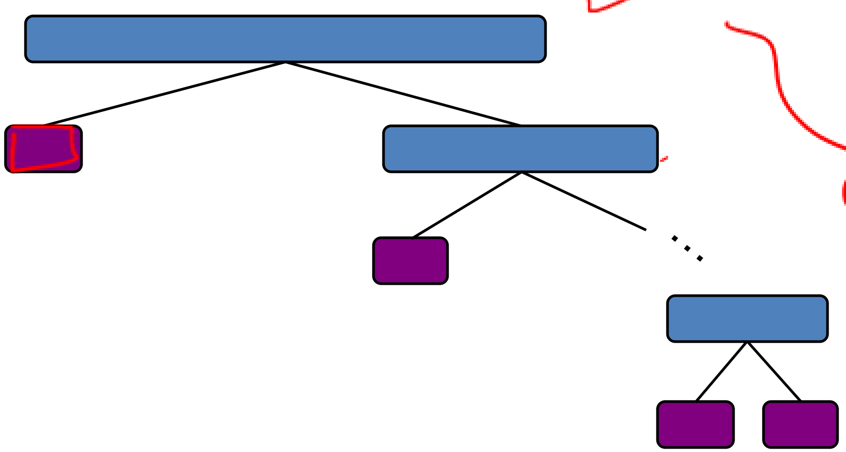
Worst-case Running Time

- The worst case for quick-sort occurs when the pivot is the unique minimum or maximum element
- One of $\boldsymbol{L}$ and $\boldsymbol{G}$ has size $\boldsymbol{n} - 1$ and the other has size 0
- The running time is proportional to the sum

$$\boldsymbol{n} + (\boldsymbol{n} - 1) + \ldots + 2 + 1$$

- Thus, the worst-case running time of quick-sort is $\boldsymbol{O}(\boldsymbol{n}^2)$

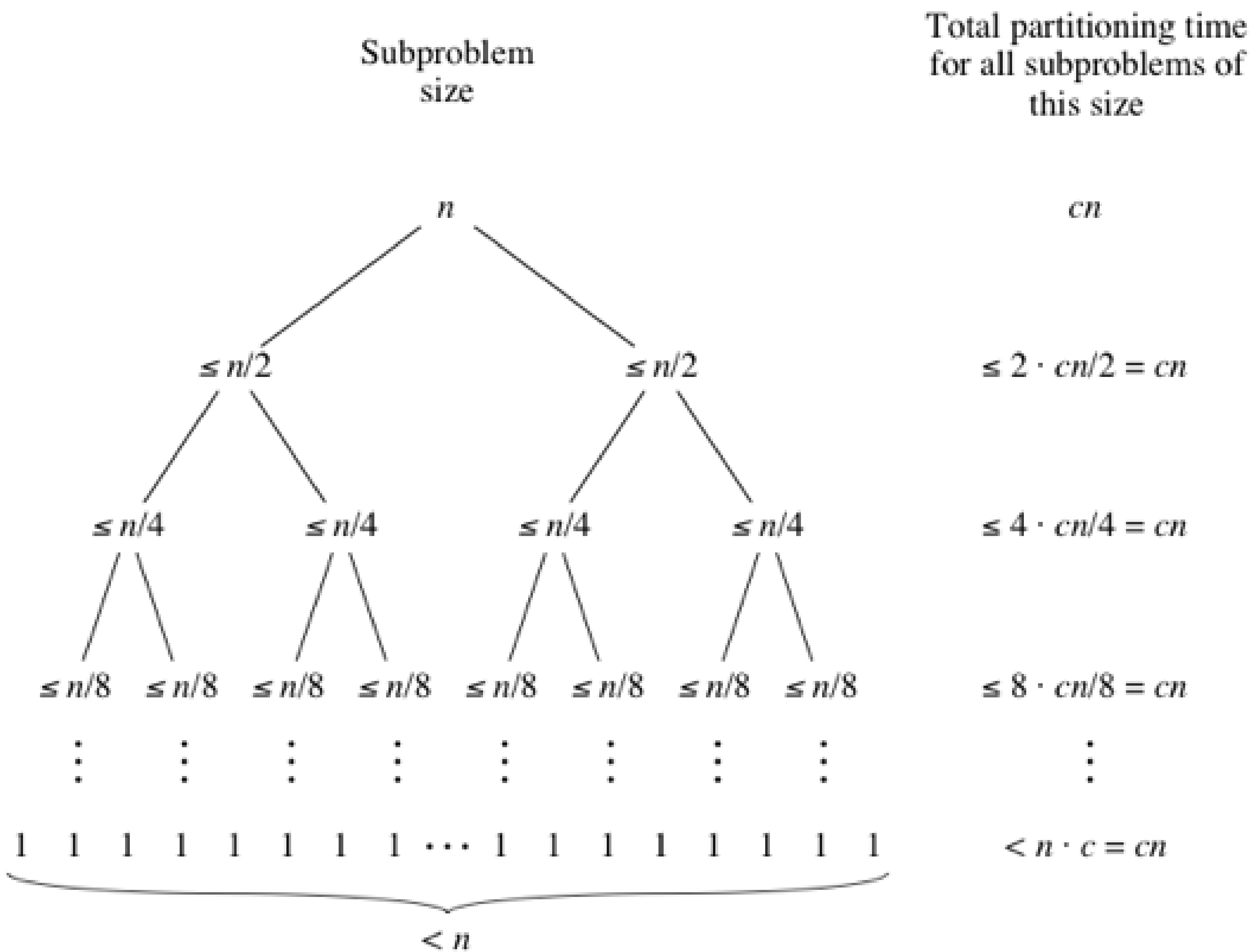
depth	time
0	$\boldsymbol{n}$
1	$\boldsymbol{n} - 1$
…	…
$\boldsymbol{n} - 1$	1



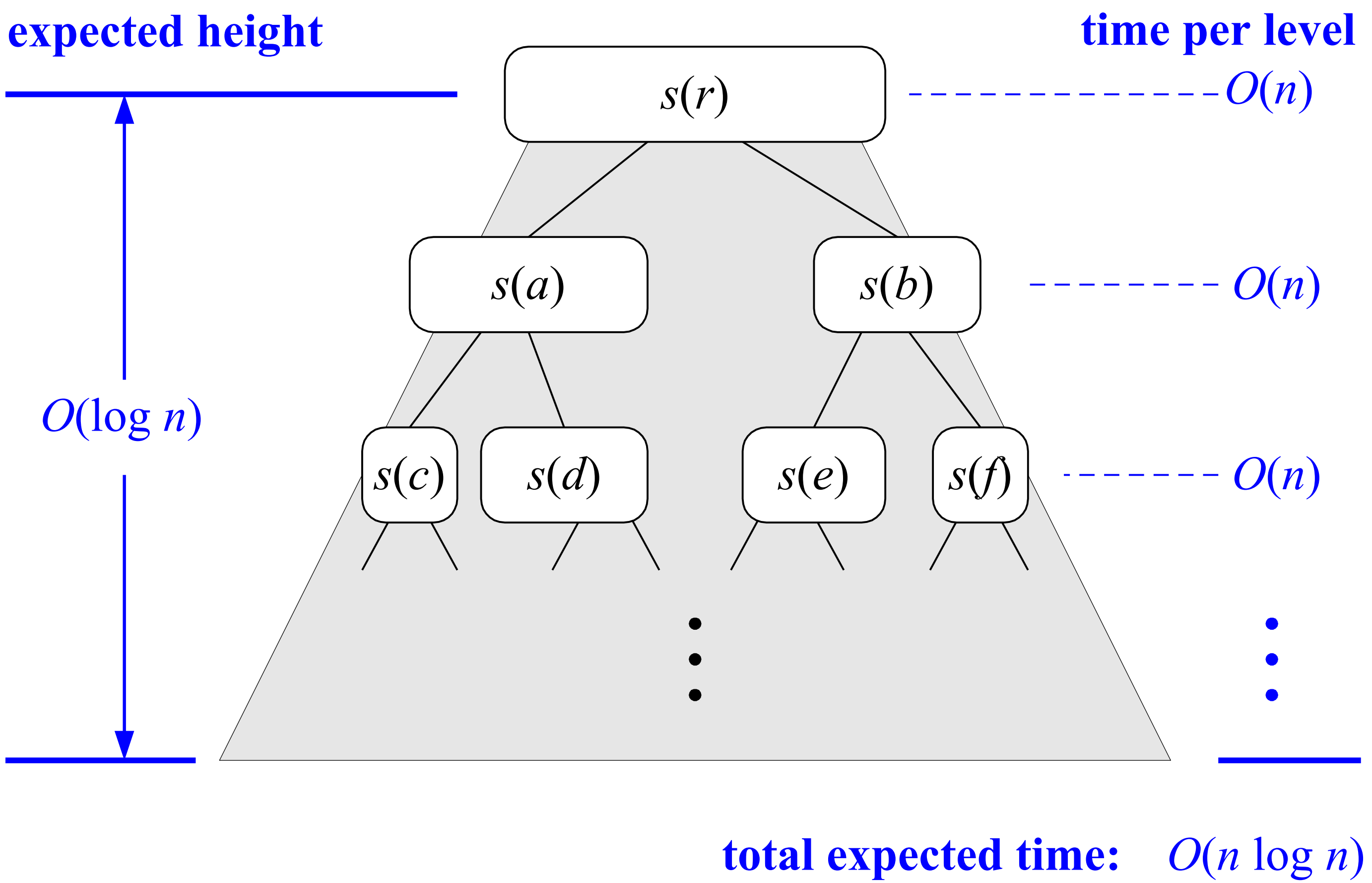
Best Case Analysis

- Best case occurs when the partitions are as evenly balanced as possible
- If the subarray has an odd number of elements and the pivot is right in the middle after partitioning, then each partition has (n-1)/2 elements.
- If the subarray has an even number n of elements and one partition has n/2 and other having n/2-1.
- In either of these cases, each partition has at most n/2 elements

Best Case Analysis



A visual time analysis of the quick-sort tree T

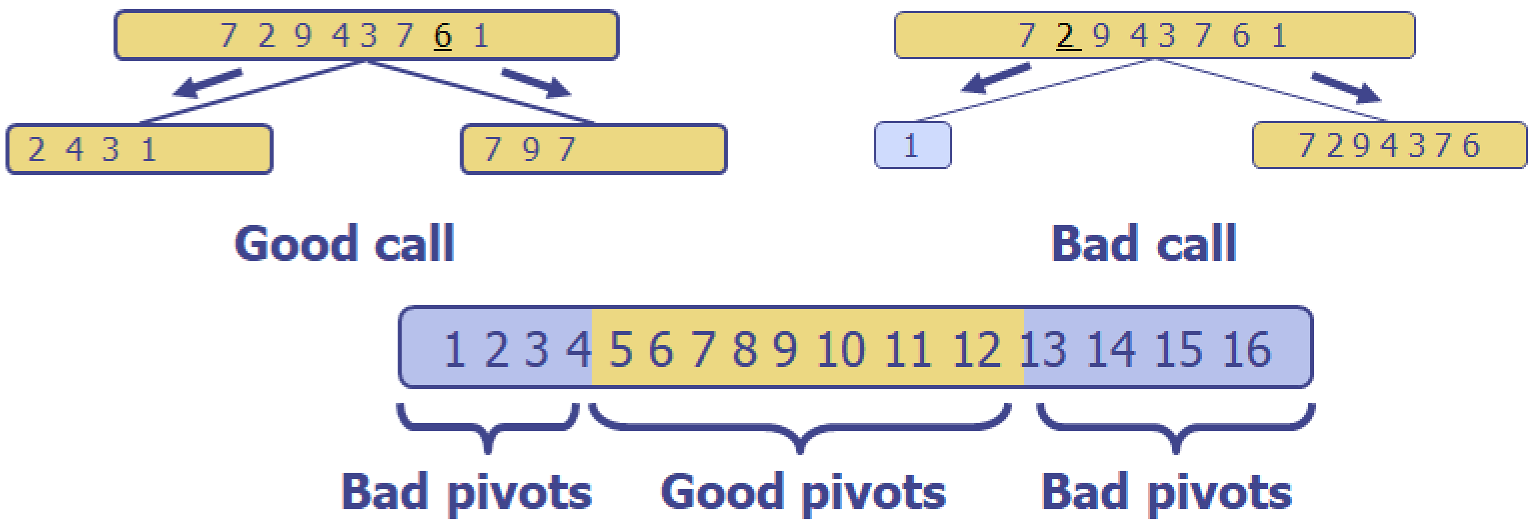


Best Case Running Time

- The amount or work done at the nodes of the same depth is O(n)
- The expected height of the quick-sort tree is O(log n)
- Thus, the expected running time of quick-sort is O(n log n)

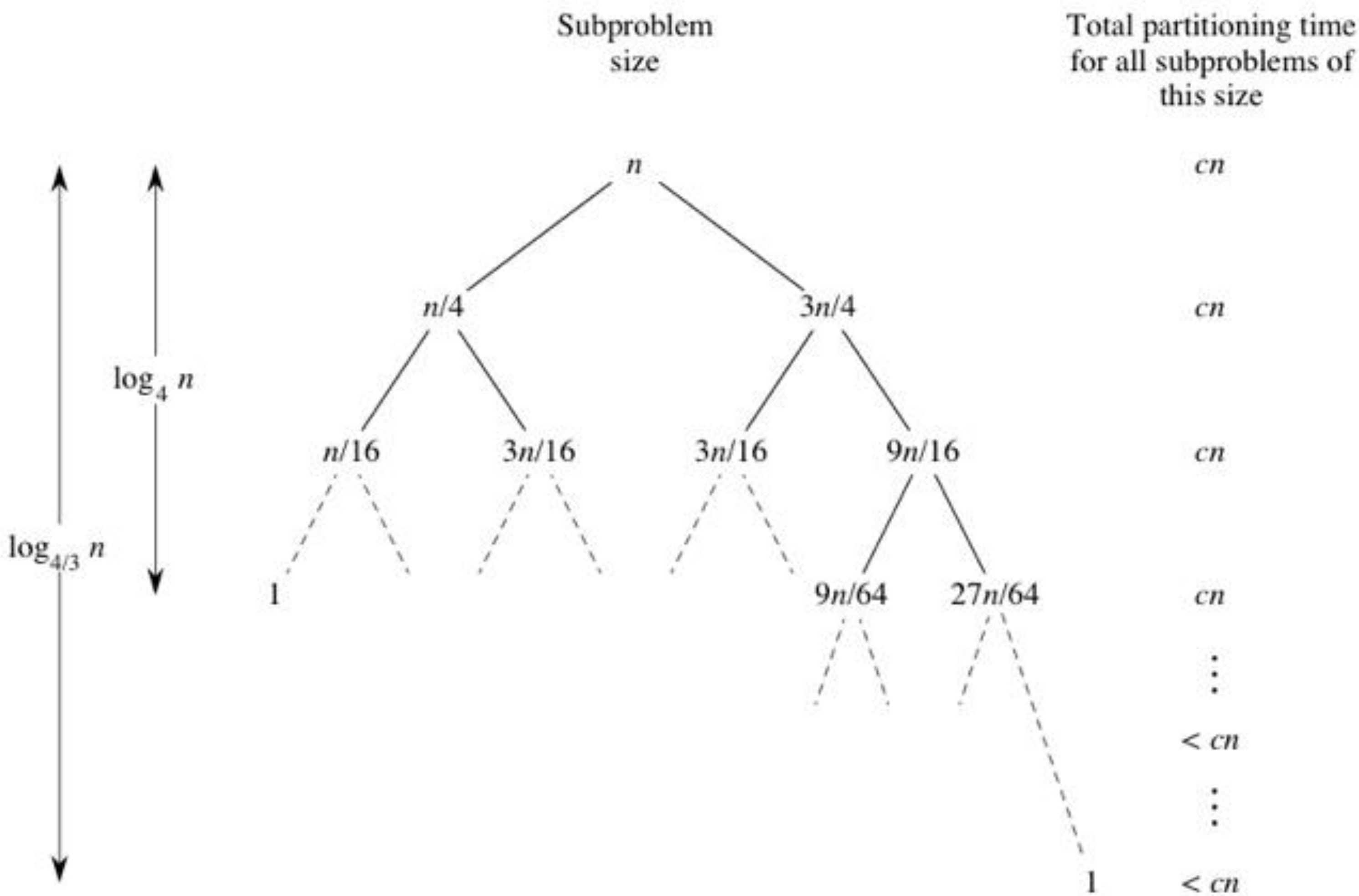
Expected Running Time

- Consider a recursive call of quick-sort on a sequence of size ***s***
  Good call: the sizes of ***L*** and ***G*** are each less than 3***s***/4
  Bad call: one of ***L*** and ***G*** has size greater than 3***s***/4

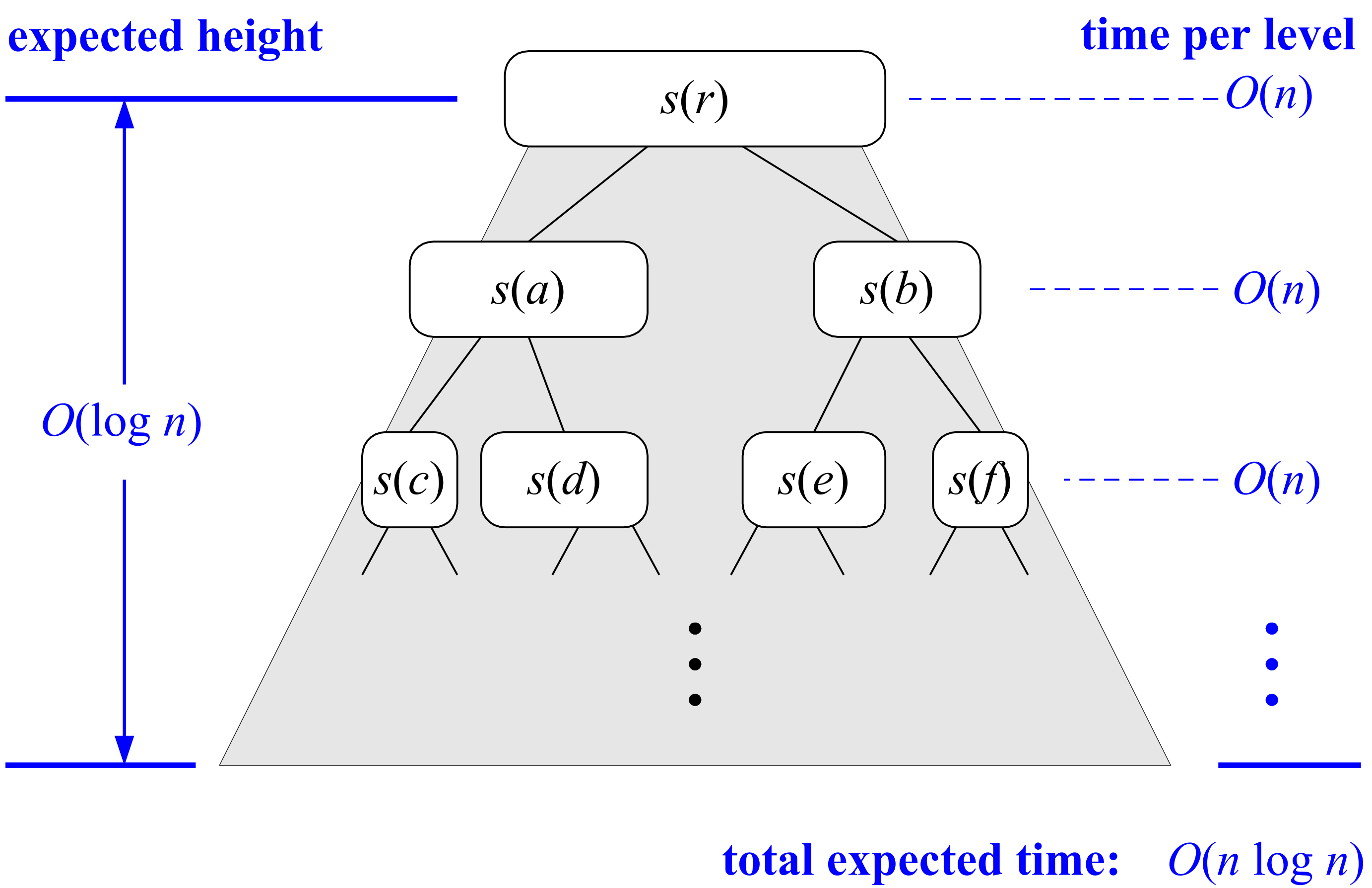


- A call is good with probability 1/2
  - 1/2 of the possible pivots cause good calls:

Expected Running Time



A visual time analysis of the quick-sort tree T



Expected Running Time

Intuitively,

- (In best case) For the tree to have height $\log_2 n$, with each step, the size of the sequence must shrink to at most (n/2).
  - That is, $\log_2 n$ invocations
- For each call to be good,
  - the input size should shrink to atmost (¾)n
  - If all calls are good calls, the expected height of tree is $\log_{(4/3)} n$
  - In other words, $\log_{(4/3)} n$ good calls are needed to get O(log n) height
  - If pivots are randomly chosen, we expect that, out of 2*log (4/3) n calls, log (4/3) n calls are good !
  - $\log_{(4/3)} n$ and log n differs by only a factor of $\log_2(4/3)$ which is a constant.
  - This implies that the height of tree is O(log n)

- Expected complexity of quick sort is O (n log n)

Expected Running Time

- The amount or work done at the nodes of the same depth is O(n)
- The expected height of the quick-sort tree is O(log n)
- Thus, the expected running time of quick-sort is O(n log n)

In-Place Quick-Sort

- A sorting algorithm is in-place if it uses only a constant amount of memory in addition to that needed for the objects being sorted themselves

In-Place Partitioning

- Perform the partition using two indices to split S into L and E U G (a similar method can split E U G into E and G).

j k

3	2	5	1	0	7	3	5	9	2	7	9	8	9	7	**6**	9

(pivot = 6)

- Repeat until j and k cross:
  - Scan j to the right until finding an element $\geq$ x.
  - Scan k to the left until finding an element < x.
  - Swap elements at indices j and k

$\longrightarrow$ j k $\longleftarrow$

3	2	5	1	0	7	3	5	9	2	7	9	8	9	7	**6**	9

In-Place Quick-Sort

- Quick-sort can be implemented to run in-place
- In the partition step, we use replace operations to rearrange the elements of the input sequence such that
  - the elements less than the pivot have rank less than h
  - the elements equal to the pivot have rank between h and k
  - the elements greater than the pivot have rank greater than k
- The recursive calls consider
  - elements with rank less than h
  - elements with rank greater than k

In-Place Quick-Sort

Algorithm inPlaceQuickSort(S, a, b):

Input: Sequence S of distinct elements; integers a and b

Output: Sequence S with elements originally from ranks from a to b, inclusive, sorted in nondecreasing order from ranks a to b

```
if a ≥ b then return            {empty subrange}
p ← S.elemAtRank(b)             {pivot}
l ← a                 {will scan rightward}
r ← b − 1                  {will scan leftward}
while l ≤ r do
    {find an element larger than the pivot}
    while l ≤ r and S.elemAtRank(l) ≤ p do
        l ← l + 1
    {find an element smaller than the pivot}
    while r ≥ l and S.elemAtRank(r) ≥ p do
        r ← r − 1
    if l < r then
        S.swapElements(S.atRank(l), S.atRank(r))
{put the pivot into its final place}
S.swapElements(S.atRank(l), S.atRank(b))
{recursive calls}
inPlaceQuickSort(S, a, l − 1)
inPlaceQuickSort(S, l + 1, b)
```

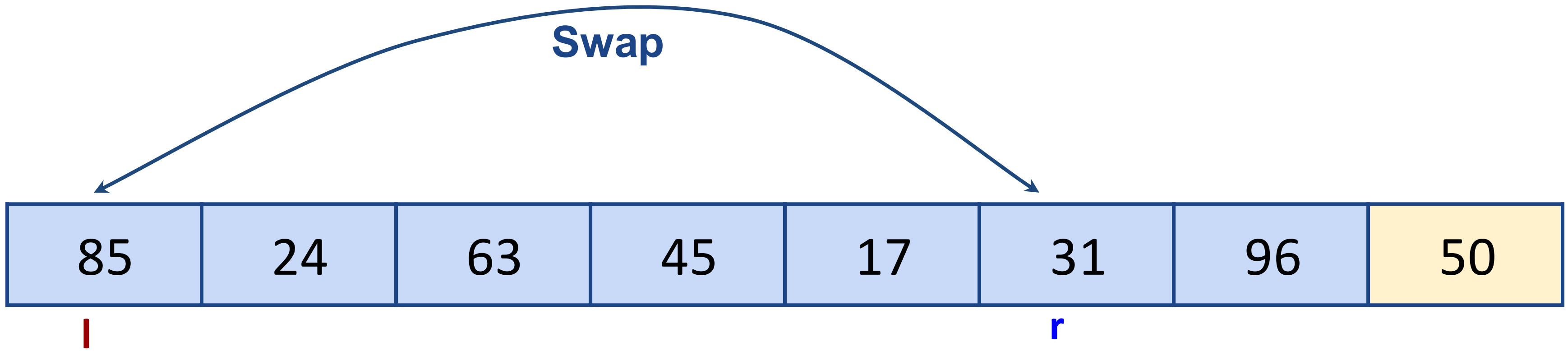
In-Place Quick-Sort

85	24	63	45	17	31	96	50
l						r	

In-Place Quick-Sort

85	24	63	45	17	31	96	50
l					r		

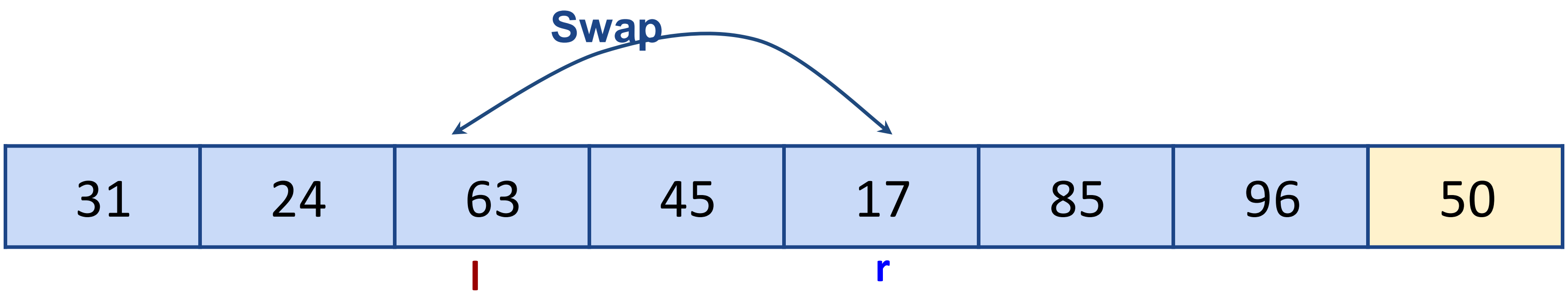
In-Place Quick-Sort



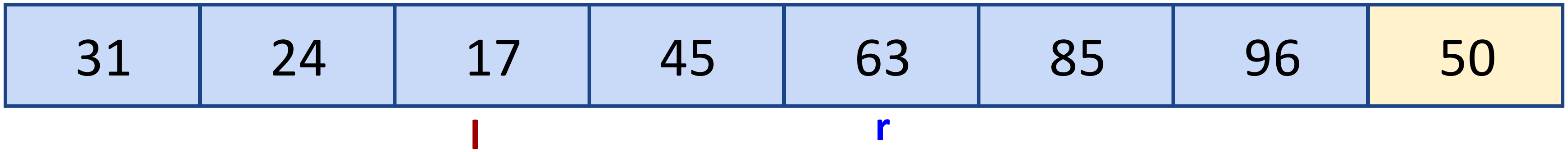
In-Place Quick-Sort

31	24	63	45	17	85	96	50
		l		r			

In-Place Quick-Sort



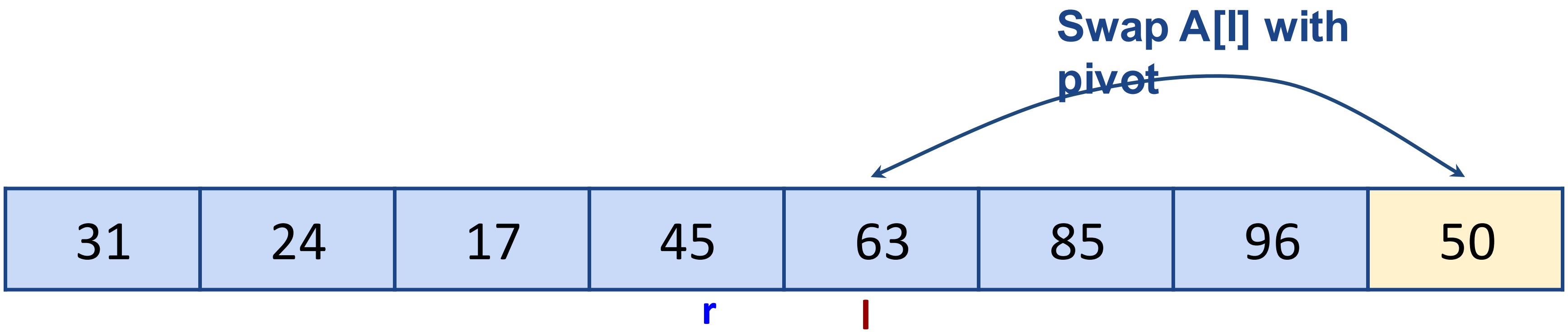
In-Place Quick-Sort



In-Place Quick-Sort

31	24	17	45	63	85	96	50
			r	l			

In-Place Quick-Sort



In-Place Quick-Sort

31	24	17	45	50	85	96	63
			r	l			

Next Recursive Calls:

inPlaceQuickSort(A, a, l – 1)
inPlaceQuickSort(A, l + 1, b)

Quick-Sort Properties

- **Not Adaptive**
- **Not Stable-Can be made stable.**
- **Not Incremental** : Incremental versions possible
- **Not online**
- **In Place**

QuickSort

- **Advantages of Quicksort**
- Its average-case time complexity to sort an array of n elements is O(nlogn).
- It requires no additional memory.
- **Disadvantages of Quicksort**
- Its worst-case running time, $O(n^2)$ to sort an array of n elements, happens when pivot is an extreme
- It is not stable.

QuickSort and Mergesort

- Quick Sort: traditionally built-in for many runtimes, hence used by programs that call the default. *Can't be used where worst-case behavior could be exploited or cause significant ramifications*, such as services that might receive denial-of-service attacks, or real-time systems.
- Java's systems programmers have chosen to use quicksort (with 3-way partitioning) to implement the primitive-type methods, and merge sort for reference-type methods. The primary practical implications of these choices are to trade speed and memory usage (for primitive types) for stability and guaranteed performance (for reference types).
- Merge Sort: used in *database scenarios*, because stable and external (results don't all fit in memory).

QuickSort and MergeSort

- In most practical situations, quicksort is the method of choice.
- If stability is important and space is available, merge sort might be best.
- In some performance-critical applications, the focus may be on just sorting numbers, so it is reasonable to avoid the costs of using references and sort primitive types instead.

THANK YOU!

BITS Pilani
Hyderabad Campus