UNIT 2:
Divide and Conquer

Quicksort

Quicksort

- invented by British computer scientist Tony Hoare in 1959
- also called as **partition-exchange sort**
- general-purpose, comparison-based sorting algorithm
- in-place sorting algorithm
- uses divide and conquer strategy

Quicksort v/s Merge sort

Merge sort:

- Divides its input's elements according to their position in the array
- Stable, NOT in-place (Auxiliary array required)
- Merge is required

Quicksort:

- Divides them according to their value in-place sorting algorithm
- NOT stable, but is in-place algorithm
- No merge required

Quicksort is efficient that it runs faster than merge sort and heapsort on randomly ordered arrays

Working of Quicksort (partition-exchange sort)

- Works by selecting a 'pivot' element from the array
- Partitions the other elements into two sub-arrays, according to whether they are less than or greater than the pivot.

$$\underbrace{A[0]\dots A[s-1]}_{\text{all are } \le A[s]} \quad \underset{\substack{\big\uparrow \\ \textbf{Pivot}}}{A[s]} \quad \underbrace{A[s+1]\dots A[n-1]}_{\text{all are } \ge A[s]}$$

- after a partition has been achieved, A[s] will be in its final position in the sorted array,
- continue sorting the two subarrays recursively.

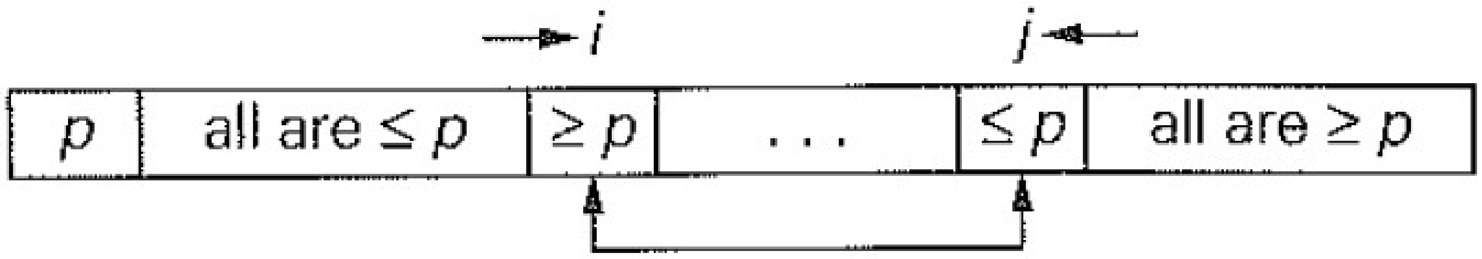
Method: Two scans of the subarray, each comparing the subarray's elements with the pivot

1. **left-to-right scan**, denoted by say, **index i**, starts with the second element of the subarray.

   scan skips over elements that are smaller than the pivot and stops on encountering the first element greater than or equal to the pivot (elements smaller than the pivot will be in the first part of the subarray)

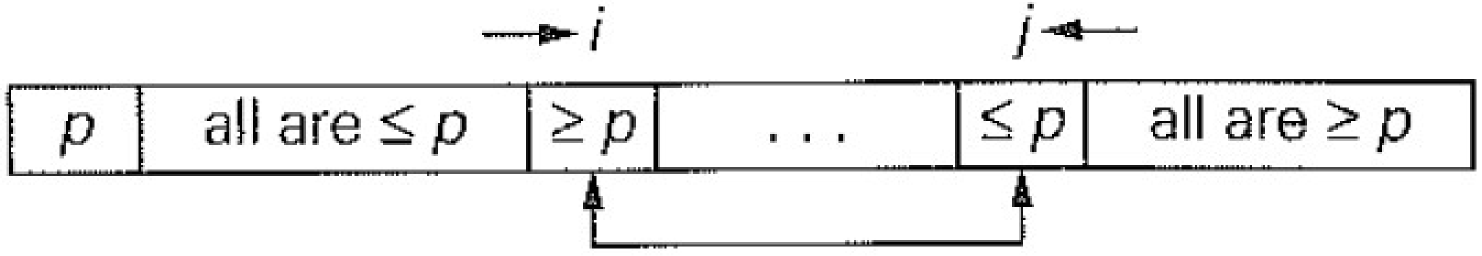
2. **right-to-left scan**, denoted by say, **index j**, starts with the last element of the subarray.

   scan skips over elements that are larger than the pivot and stops on encountering the first element smaller than or equal to the pivot (elements larger than the pivot to be in the second part of the subarray)



Both scans stop:
3 situations may arise, depending on whether or not the scanning indices have crossed

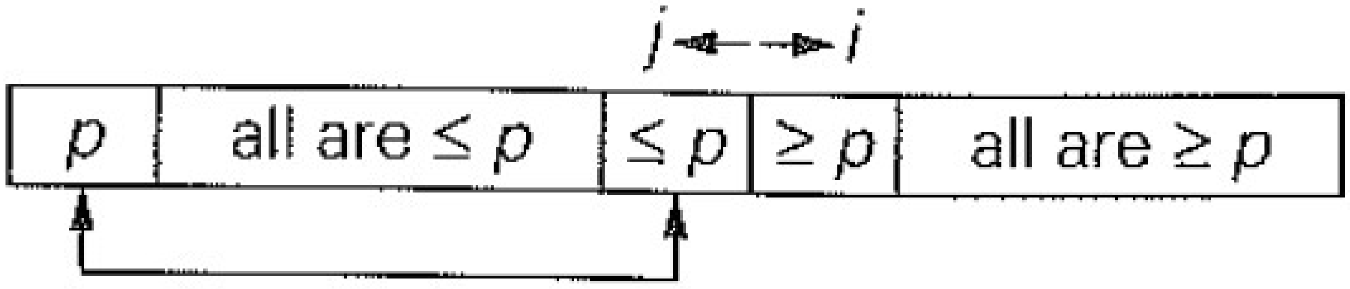
1. scanning indices i and j have not crossed : i < j



Exchange A[i] and A[j] and resume the scans by incrementing i and decrementing j, respectively

Both scans stop…

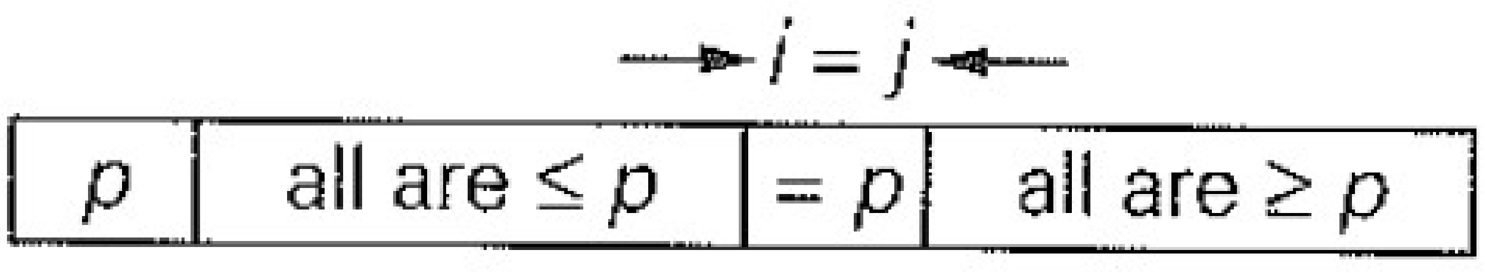
2. scanning indices i and j have crossed over : i > j



The array is partitioned after exchanging the pivot with A[j]

Both scans stop…

3. scanning stops pointing to the same element: i = j,



The value both indices are pointing must be equal pivot. Thus, array is partitioned, with the split position

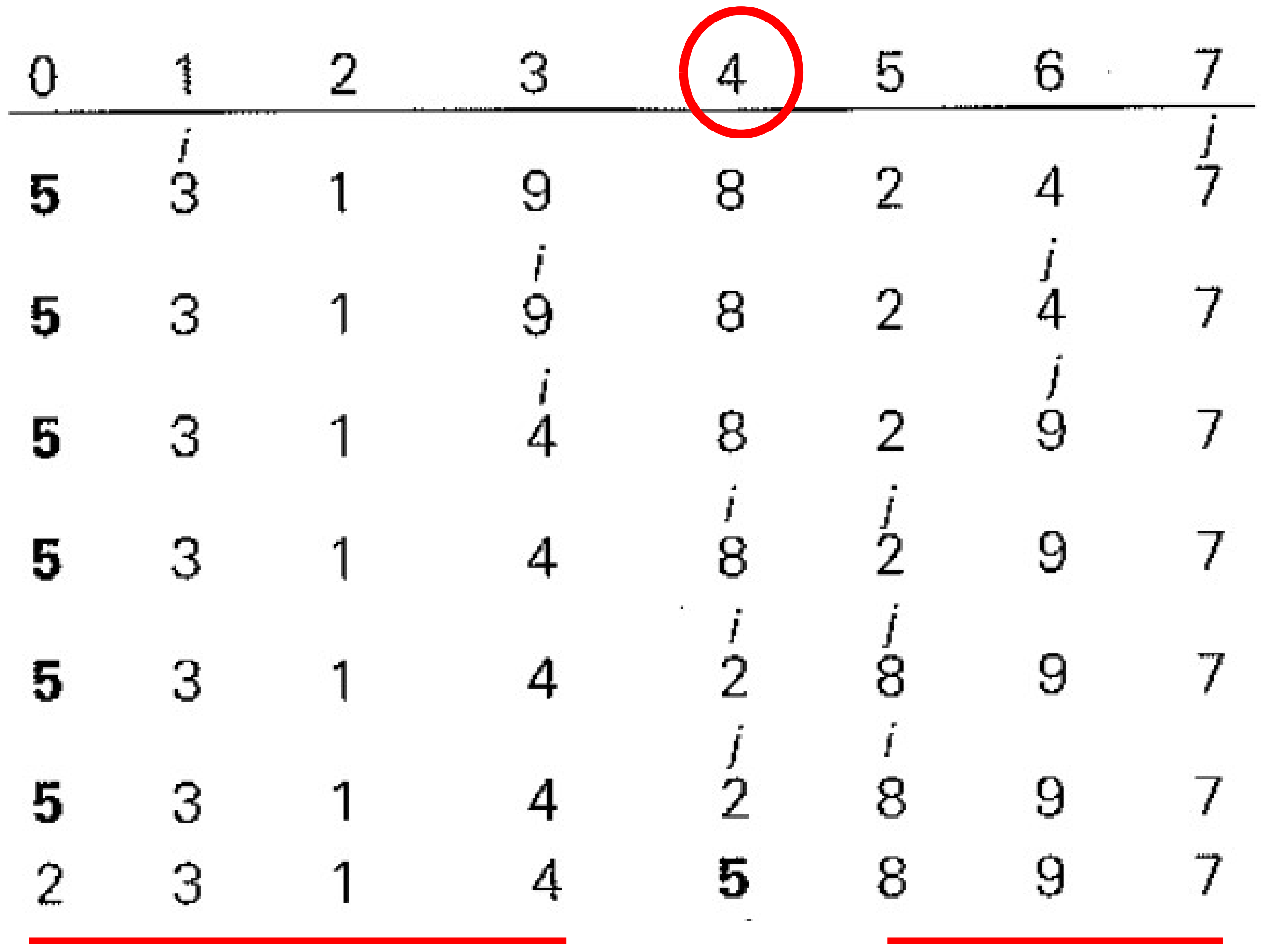
s = i = j

Note: we can combine case 2 and 3

Quicksort

```
ALGORITHM  Quicksort(A[l..r])
    //Sorts a subarray by quicksort
    //Input: A subarray A[l..r] of 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])
```

l = 0 and r=7

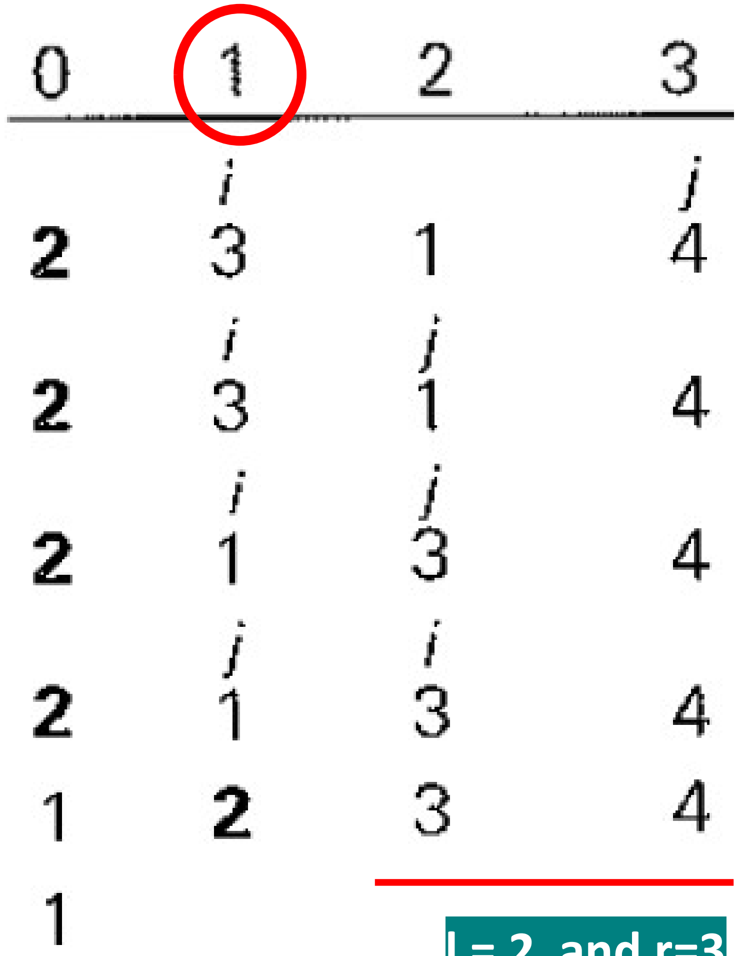


0	1	2	3	4	5	6	7
5	$\overset{i}{3}$	1	9	8	2	4	$\overset{j}{7}$
5	3	1	$\overset{i}{9}$	8	2	$\overset{j}{4}$	7
5	3	1	$\overset{i}{4}$	8	2	$\overset{j}{9}$	7
5	3	1	4	$\overset{i}{8}$	$\overset{j}{2}$	9	7
5	3	1	4	$\overset{i}{2}$	$\overset{j}{8}$	9	7
5	3	1	4	$\overset{j}{2}$	$\overset{i}{8}$	9	7
2	3	1	4	**5**	8	9	7

l = 0 and r=3

l = 5 and r=7

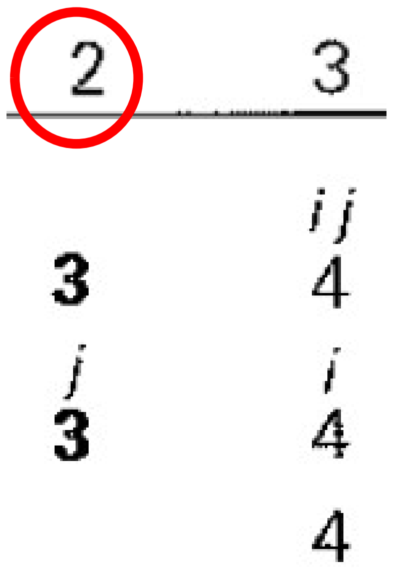
l = 0 and r=3



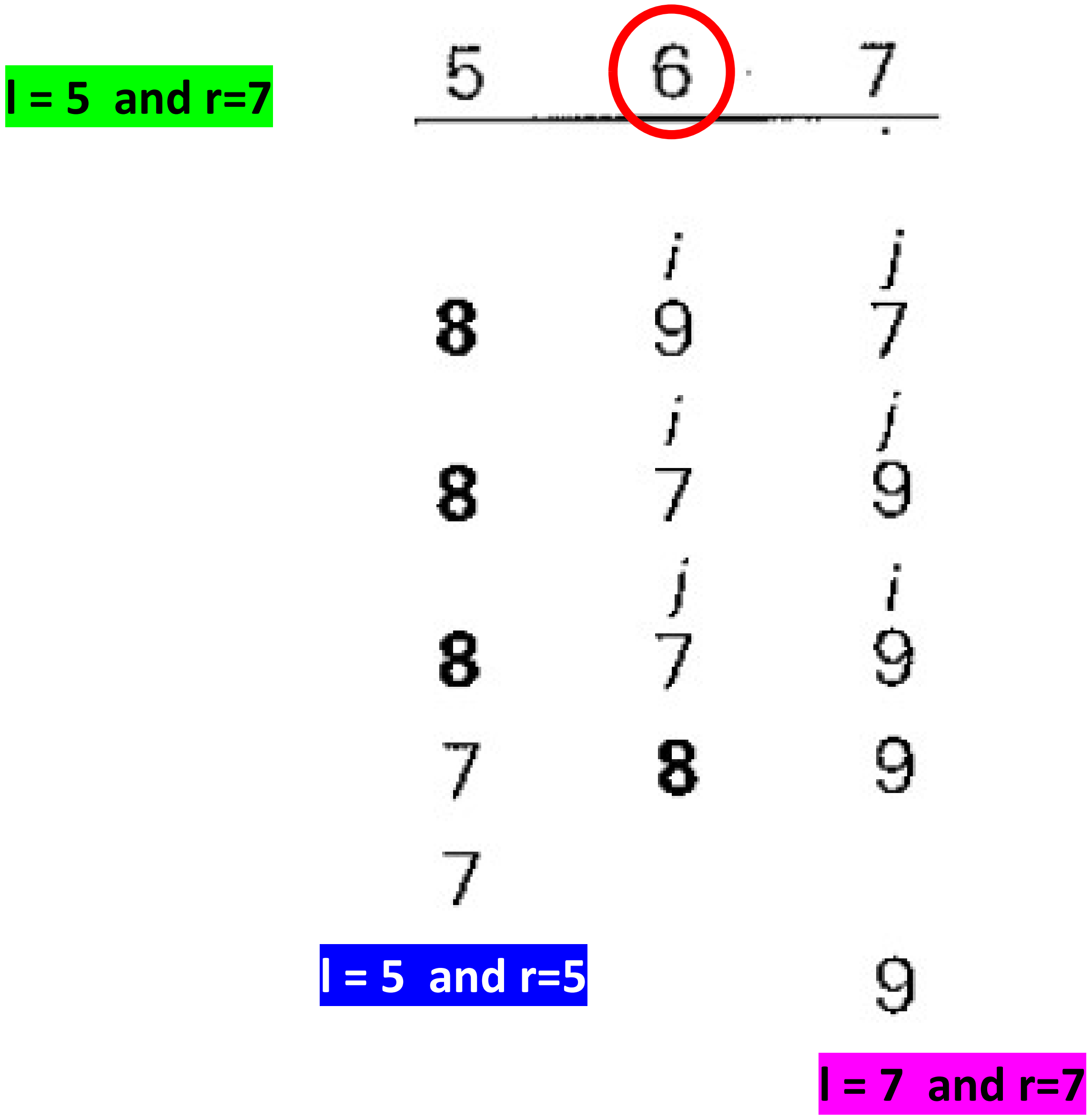
l = 2 and r=3

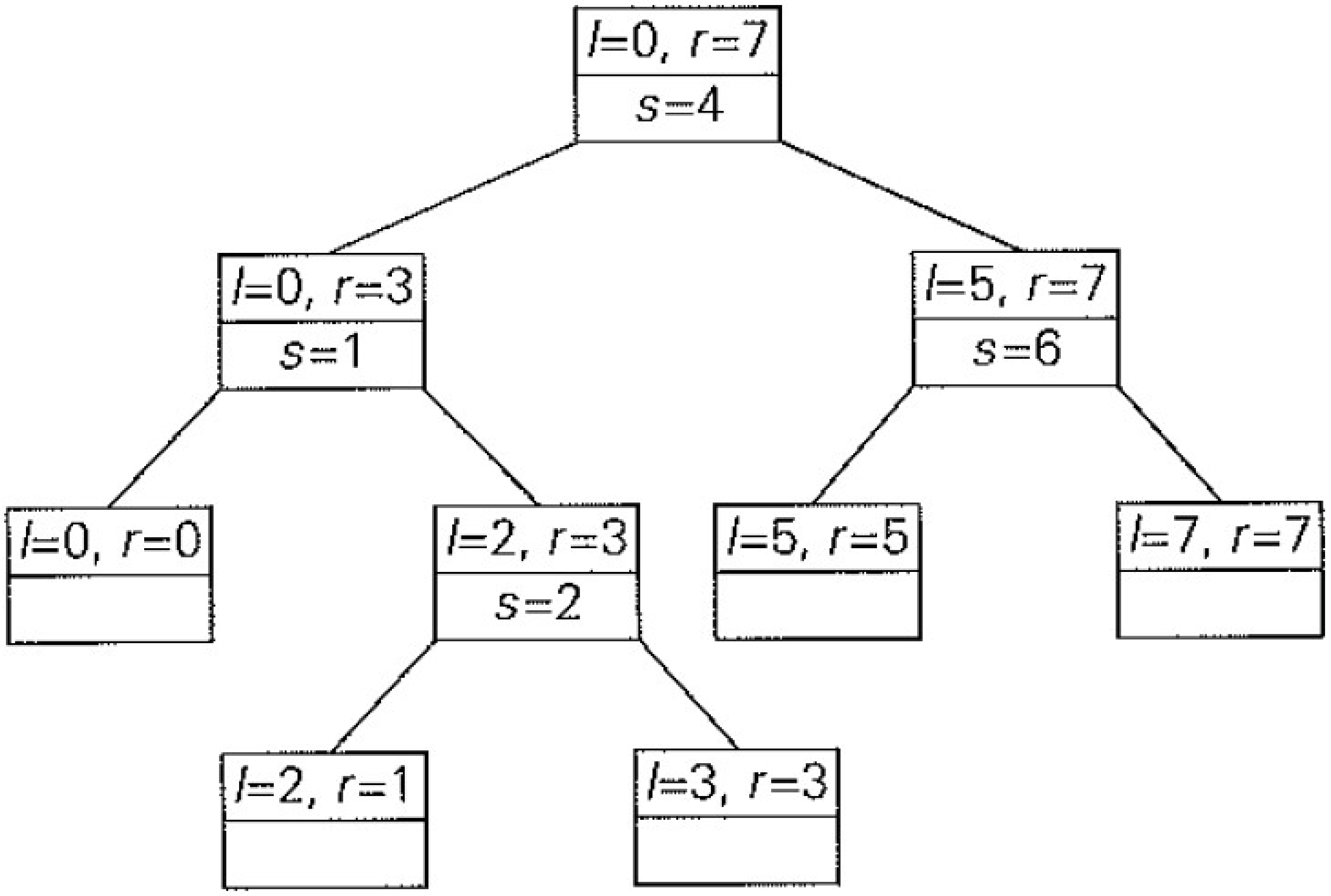
l = 0 and r=0

l = 2 and r=3



l = 3 and r=3


l = 5 and r=7
5
6
7
i
j
8
9
7
i
j
8
7
9
j
i
8
7
9
7
8
9
7
l = 5 and r=5
9
l = 7 and r=7



The tree of recursive calls to Quicksort with input values l and r of subarray bounds and split positions of a partition obtained

Quicksort…

```
ALGORITHM  Partition(A[l..r])
    //Partitions a subarray by using its first element as a pivot
    //Input: A subarray A[l..r] of A[0..n − 1], defined by its left and right
    //        indices l and r (l < r)
    //Output: A 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 $j$
```

Quicksort : Partition schemes

- **Lomuto partition scheme** (Nico Lomuto and popularized by Bentley)
  chooses a pivot that is typically the last element in the array.

- **Hoare partition scheme (original partition)**
  uses two pointers (indices into the range) that start at both ends of the array being partitioned, then move toward each other, until they detect an inversion

Note:

Hoare's scheme is more efficient than Lomuto's partition scheme because it does three times fewer swaps on average.

Quicksort : Applications

- Commercial Computing
- Information searching
- Operational research and event-driven simulation (cache-friendly)

Quicksort variants

- Multi-pivot quicksort
- Partial and incremental quicksort
- Block Quicksort
- External quicksort - for disk files
- Three-way radix quicksort
- Quick radix sort - parallel PRAM algorithm

Quicksort : Implementation issues (1)

Choice of pivot

- leftmost element:
  causes worst-case behavior on already sorted arrays

Solution:

- random index for the pivot,
- choosing the middle index of the partition
- (especially for longer partitions) choosing the median of the first, middle and last element of the partition for the pivot.

Quicksort : Implementation issues (2)

Repeated elements (all elements are equal)

- **Hoare partition scheme** needlessly swaps elements, but the partitioning itself is best case
- **Lomuto partition scheme problem** (also called the Dutch national flag problem): when all the input elements are equal, at each recursion, the left partition is empty (no input values are less than the pivot), and the right partition has all the (n-1) elements

Solution: Fat partition

Alternative linear-time partition routine can be used that separates the values into three groups: values less than the pivot, values equal to the pivot, and values greater than the pivot. ("**fat partition**" - implemented in the **qsort of Version 7 Unix**). The values equal to the pivot are already sorted, so only the less-than and greater-than partitions need to be recursively sorted.

Quicksort : Interesting facts (1)

At that time, **Hoare** was working on a **machine translation project** for the National Physical Laboratory. As a part of the **translation process**, he needed **to sort the words in Russian sentences** before looking them up in a Russian-English dictionary, which was in alphabetical order on magnetic tape. After recognizing that his first idea**, insertion sort, would be slow**, he came up with a new idea. He wrote the partition part in Mercury Autocode but had trouble dealing with the list of unsorted segments. On return to England, he was asked to write code for **Shellsort**. Hoare mentioned to his boss that he knew of a faster algorithm and his boss bet sixpence that he did not. His boss ultimately accepted that he had lost the bet. Later, Hoare learned about ALGOL and its ability to do recursion that enabled him to publish the code in Communications of the Association for Computing Machinery, the premier computer science journal of the time.

Source: https://en.wikipedia.org/wiki/Quicksort

Quicksort : Interesting facts (2)

Quicksort gained widespread adoption, appearing, for example, in **Unix** as the **default library sort** subroutine.

Hence, it lent its name to the **C standard library** subroutine **qsort** and in the reference implementation of **Java**.

Source: https://en.wikipedia.org/wiki/Quicksort

Competitors for Quicksort

- **Heapsort**

  but its average running time is usually considered slower than in-place quicksort.
- **Introsort**

  variant of quicksort that switches to heapsort when a bad case is detected to avoid quicksort's worst-case running time.
- **Merge sort**

  stable sort, has excellent worst-case performance, works well on linked lists, good choice for external sorting of very large data sets stored on slow-to-access media such as disk storage or network-attached storage.
- **Bucket sort**

  with two buckets, it is very similar to quicksort; the pivot is effectively the value in the middle of the value range, which does well on average for uniformly distributed inputs.

Let’s check our understanding

QUIZ time!!!

Attempt the quiz using the given link:

https://forms.gle/yFW6Dd4yPuqsZvaq8

Time: 15 min

Marks: 10

No. of questions: 10

Next session…

Divide and Conquer:

    Quicksort analysis