Quick Sort

Invented by Hoare

- C.A.R. Tony Hoare, at age 26, invented his algorithm in 1960 while trying to sort words for a machine translation project from Russian to English. Says Hoare, “My first thought on how to do this was bubblesort and, by an amazing stroke of luck, my second thought was Quicksort.” It is hard to disagree with his overall assessment: “I have been very lucky. What a wonderful way to start a career in Computing, by discovering a new sorting algorithm!” [Hoa96]. Twenty years later, he received the TuringAward for “fundamental contributions to the definition and design of programming languages”; in 1980, he was also knighted for services to education and computer science.

QuickSort

- Definitely a “greatest hit” algorithm
- Prevalent in practice
- Beautiful analysis
- $O(n \ \log n)$ time “on average”, works in place
  - i.e., minimal extra memory needed
- See course site for optional lecture notes

The Sorting Problem

Input : array of n numbers, unsorted

3	8	2	5	1	4	7	6

Output : Same numbers, sorted in increasing order

1	2	3	4	5	6	7	8

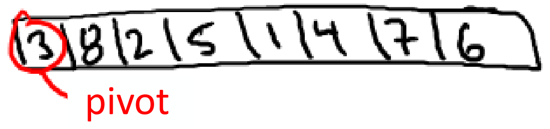
Assume : all array entries distinct.

Exercise : extend QuickSort to handle duplicate entries

Partitioning Around a Pivot

Key Idea : partition array around a pivot element.

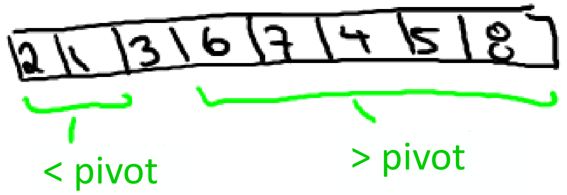
--Pick element of array



--Rearrange array so that

--Left of pivot => less than pivot

--Right of pivot => greater than pivot



Note : puts pivot in its "righful position".

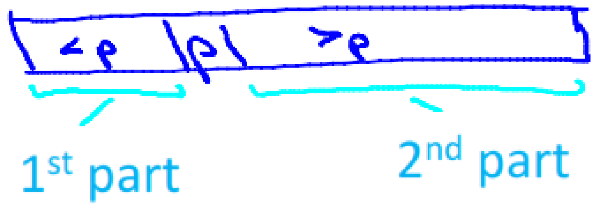
Two Cool Facts About Partition

1. Linear O(n) time, no extra memory
   [see next video]

2. Reduces problem size

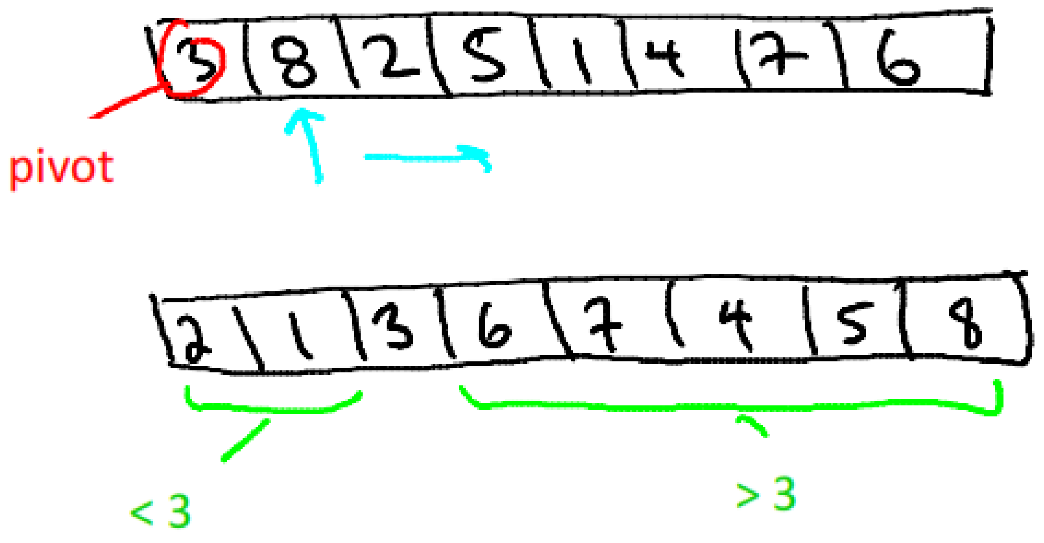
```
QuickSort (array A, length n)
-If n=1   return
-p = ChoosePivot(A,n)
-Partition A around p
-Recursively sort 1st part
-Recursively sort 2nd part
```

[currently unimplemented]



Not a Inplace Implementation

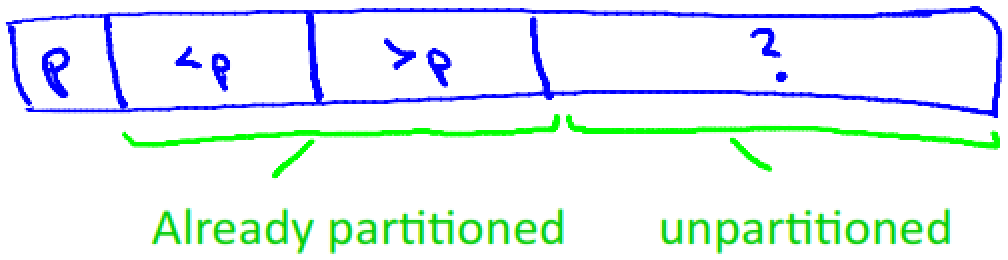
Note : Using O(n) extra memory, easy to partition around pivot in O(n) time.



Inplace Implementation

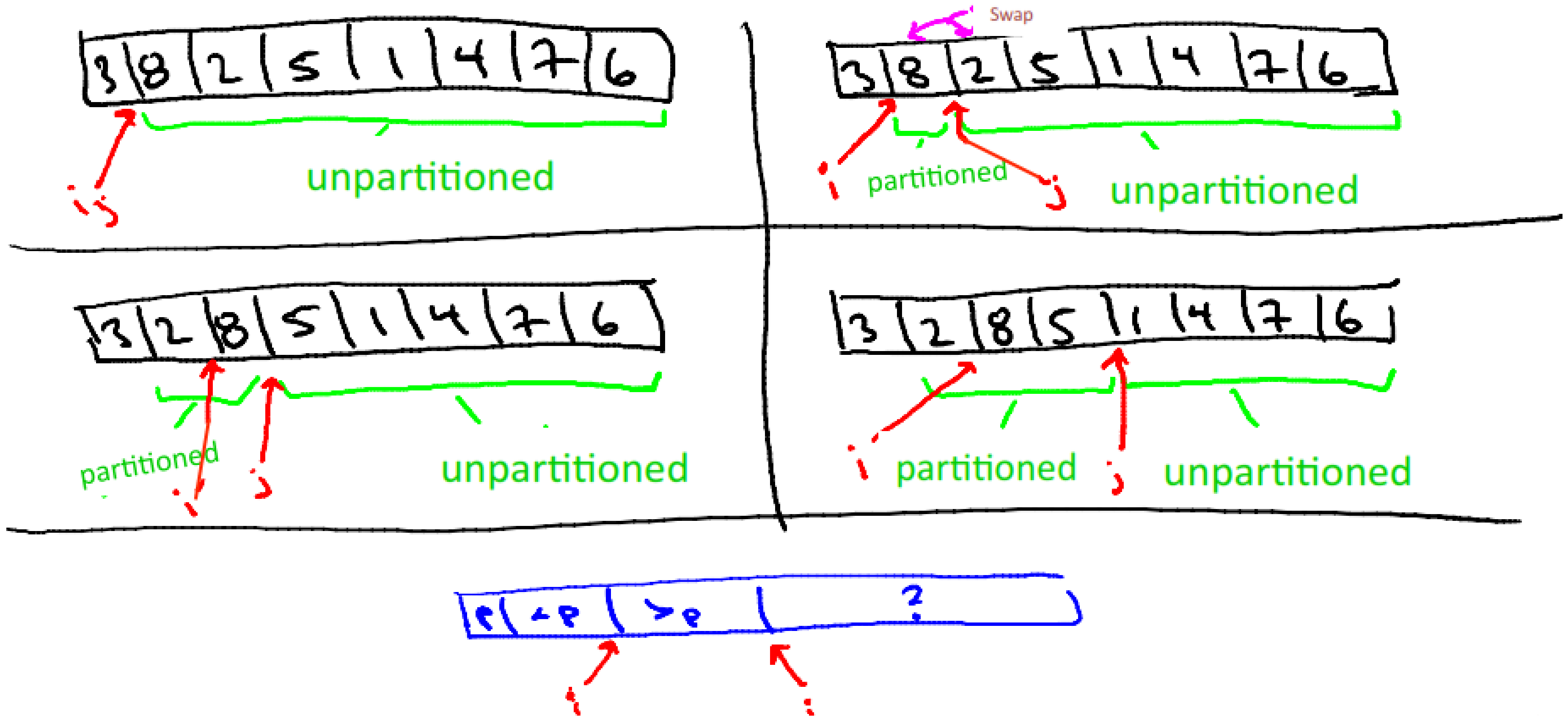
Assume : pivot = 1st element of array
[if not, swap pivot <--> 1st element as preprocessing step]

High – Level Idea :

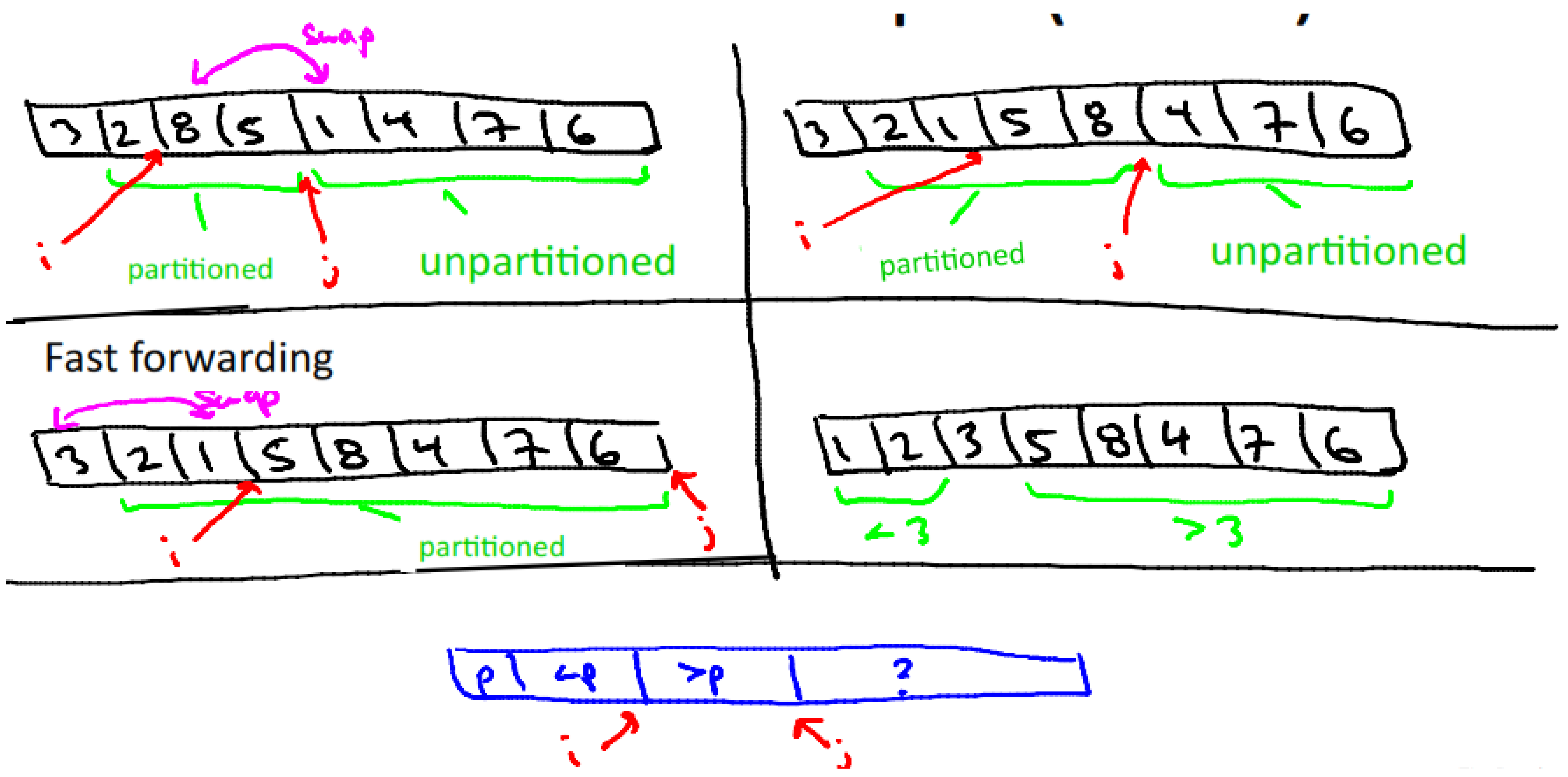


-Single scan through array
- invariant : everything looked at so far is partitioned

Example

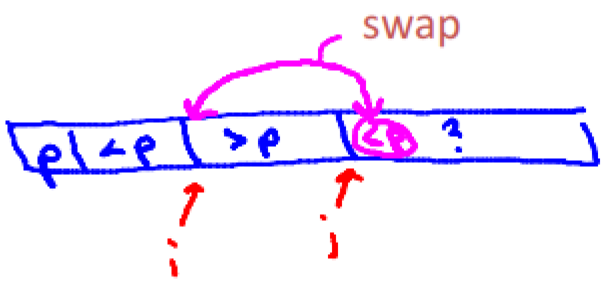


3 8 2 5 1 4 7 6
i j
unpartitioned
Swap
3 8 2 5 1 4 7 6
i
partitioned
j
unpartitioned
3 2 8 5 1 4 7 6
partitioned
i
j
unpartitioned
3 2 8 5 1 4 7 6
i
partitioned
j
unpartitioned
p <p >p ?
i
j


swap
3 2 8 5 1 4 7 6
i
partitioned
j
unpartitioned
3 2 1 5 8 4 7 6
i
partitioned
j
unpartitioned
Fast forwarding
swap
3 2 1 5 8 4 7 6
i
partitioned
j
1 2 3 5 8 4 7 6
<3
>3
p
<p
>p
?
i
j

Pseudo code for partition

```
Partition (A,l,r)                    [ input corresponds to A[l…r]]
        - p:= A[l]
        - i:= l+1
        - for j=l+1 to r
                - if A[j] < p           [if A[j] > p, do nothing ]
                  -swap A[j] and A[i]
                  - i:= i+1
        - swap A[l] and A[i-1]
```



Running time

Running time = O(n), where n = r – l + 1 is the length of the input (sub) array.

Reason : O(1) work per array entry.

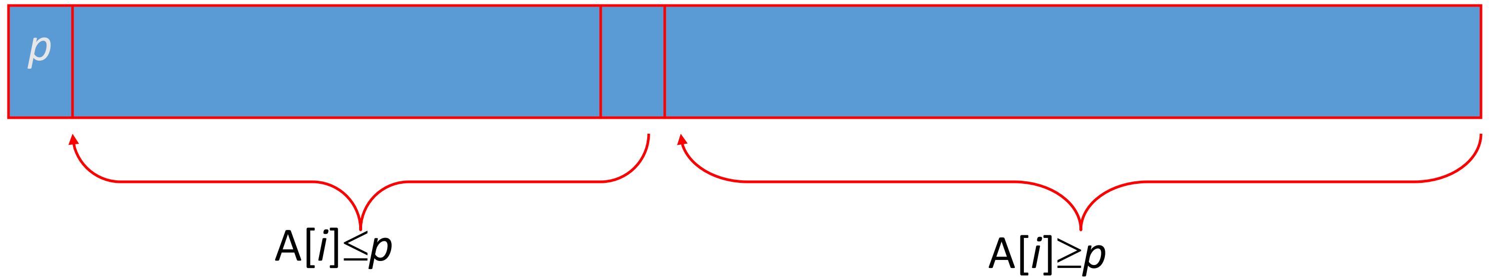
Also : clearly works in place (repeated swaps)

Outline of QuickSort

- The Partition subroutine
- Correctness proof [optional]
- Choosing a good pivot
- Randomized QuickSort
- Analysis
  - A Decomposition Principle
  - The Key Insight
  - Final Calculations

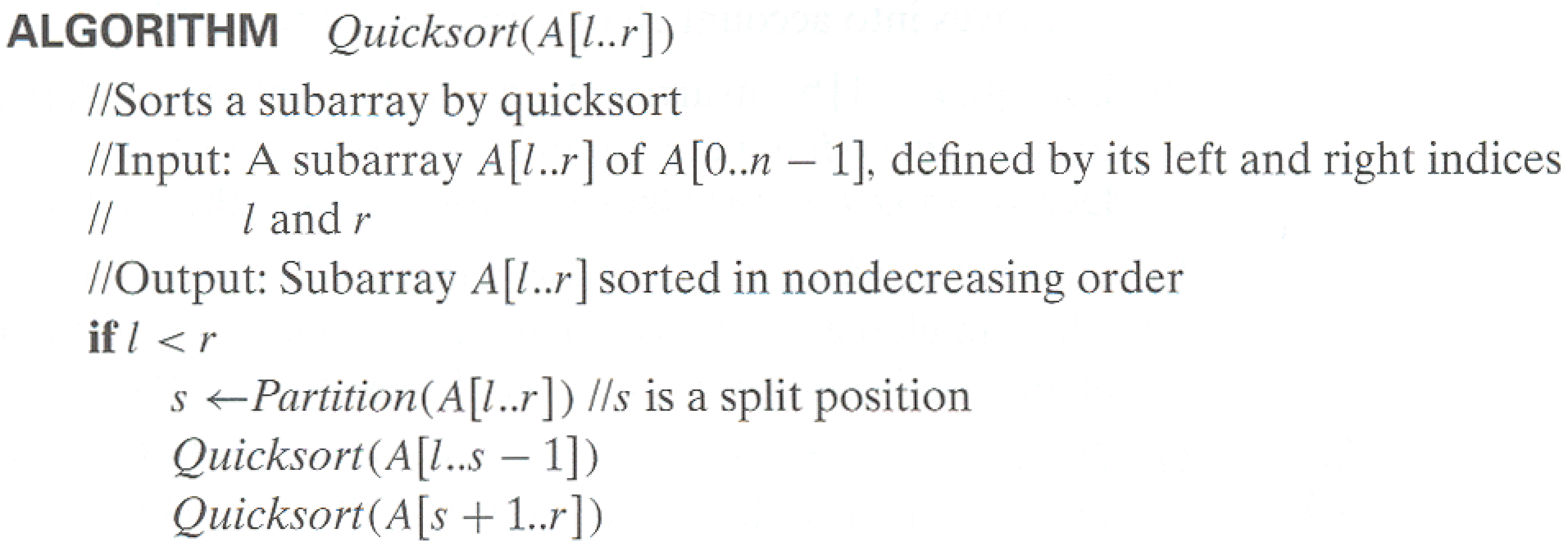
Quicksort

- 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 (see next slide for an algorithm)



- 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

Algorithm



```
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 $l < r$
        $s \leftarrow$ Partition($A[l..r]$) //$s$ is a split position
        Quicksort($A[l..s-1]$)
        Quicksort($A[s+1..r]$)
```

```
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
    $p \leftarrow A[l]$
    $i \leftarrow l; \quad j \leftarrow r + 1$
    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 $j$
```

Best case

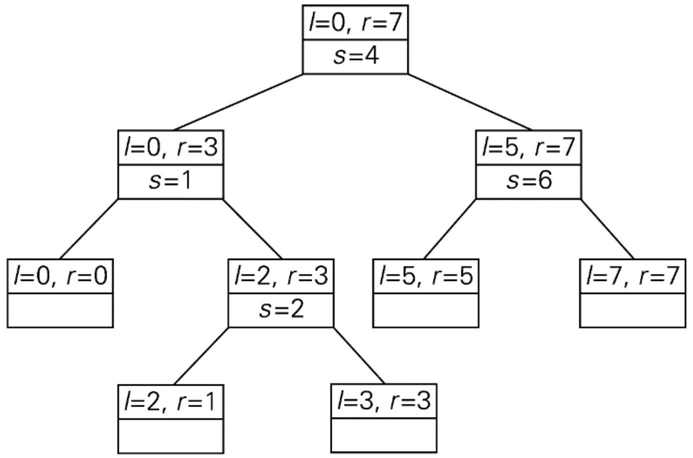
$$C_{best}(n) = 2C_{best}(n/2) + n \quad \text{for } n > 1, \quad C_{best}(1) = 0.$$

Quicksort Example

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

(a)



(b)

FIGURE 4.3 Example of Quicksort operation. (a) The array's transformations with pivots shown in bold. (b) The tree of recursive calls to *Quicksort* with input values l and r of subarray bounds and split position s of a partition obtained.

Analysis of Quicksort

- Best case: split in the middle — $\Theta(n \log n)$
- Worst case: sorted array! — $\Theta(n^2)$
- Average case: random arrays — $\Theta(n \log n)$

- Improvements:
  - better pivot selection: median of three partitioning
  - switch to insertion sort on small subfiles
  - elimination of recursion

  These combine to 20-25% improvement

- Considered the method of choice for internal sorting of large files ($n \geq 10000$)

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