Course Code

CSC402

Course Name

Analysis of Algorithms

Department of Computer Engineering

AY 2021-2022

Syllabus

Module 2 Divide and Conquer Approach

- General method, Merge sort, Quick sort, Finding minimum and maximum algorithms and their Analysis, Analysis of Binary search.

CE– SE–AOA

Dr. Anil Kale

Associate Professor

Dept. of Computer Engineering,

General method

Divide-and-Conquer

- Divide and Conquer is a method of algorithm design that has created such efficient algorithms as Merge Sort.
- In terms or algorithms, this method has three distinct steps: –
- Divide: If the input size is too large to deal with in a straightforward manner, divide the data into two or more disjoint subsets.
- Recur: Use divide and conquer to solve the subproblems associated with the data subsets.
- Conquer: Take the solutions to the subproblems and “merge” these solutions into a solution for the original problem.

Divide-and-Conquer Algo

Algo DAC(P)
{
 If small (P) then return S(P)
    else
   {
    Divide P into smaller instances P_1 ,P_2 ....P_k , k>=1
     Apply DAC to each of these problem
     Return combine (DAC(P_1), DAC(P_2)…DAC (P_k));
    }
}

General method

Divide-and-Conquer Algo

The complexity of divide and conquer algo is given by recurrence equation

$$T(n) = \begin{cases} T(1) & n=1 \\ aT(n/b) + f(n) & n>1 \end{cases}$$

Merge sort

- An algorithm is a sequence of unambiguous instructions for solving a problem, i.e., for obtaining a required output for any authorized input in a finite amount of time.
- A mathematical relation between an observed quantity and a variable used in a step-by-step mathematical process to calculate a quantity
- Algorithm is any well defined computational procedure that takes some value or set of values as input and produces some value or set of values as output
- A procedure for solving a mathematical problem in a finite number of steps that frequently involves repetition of an operation; broadly : a step-by-step procedure for solving a problem or accomplishing some end (Webster’s Dictionary)

Merge Sort Approach

To sort an array A[p . . r]: A[1…n]=   n=10 n/2=5

- **Divide**
  - Divide the n-element sequence to be sorted into two subsequences of n/2 elements each… till the elements are not divisible.. Or single element
- **Conquer**
  - Sort the subsequences recursively using merge sort
  - When the size of the sequences is 1 there is nothing more to do
- **Combine**
  - Merge the two sorted subsequences

Merge Sort

Alg.: MERGE-SORT(A, p, r)

p q r

1	2	3	4	5	6	7	8
5	2	4	7	1	3	2	6

if p < r ▷ Check for base case

then $q \leftarrow \lfloor (p + r)/2 \rfloor$ ▷ Divide

MERGE-SORT(A, p, q) ▷ Conquer

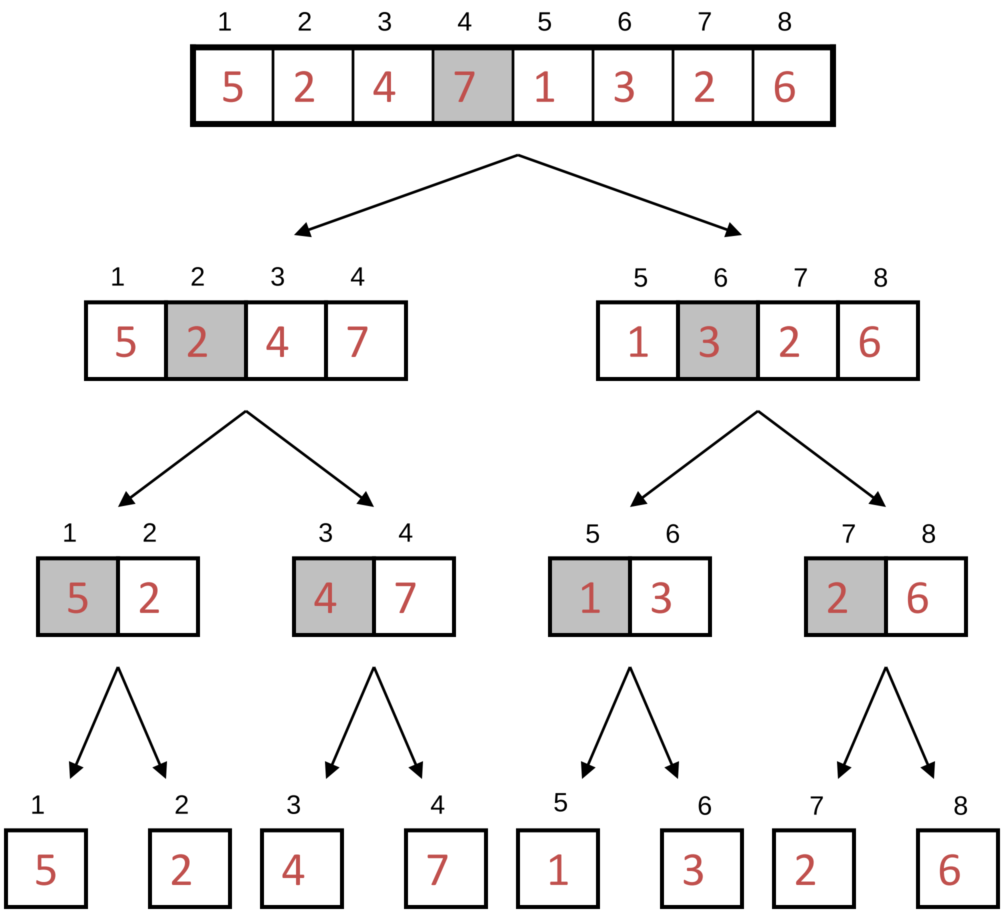
MERGE-SORT(A, q + 1, r) ▷ Conquer

MERGE(A, p, q, r) ▷ Combine

- Initial call: MERGE-SORT(A, 1, n)

Example – n Power of 2

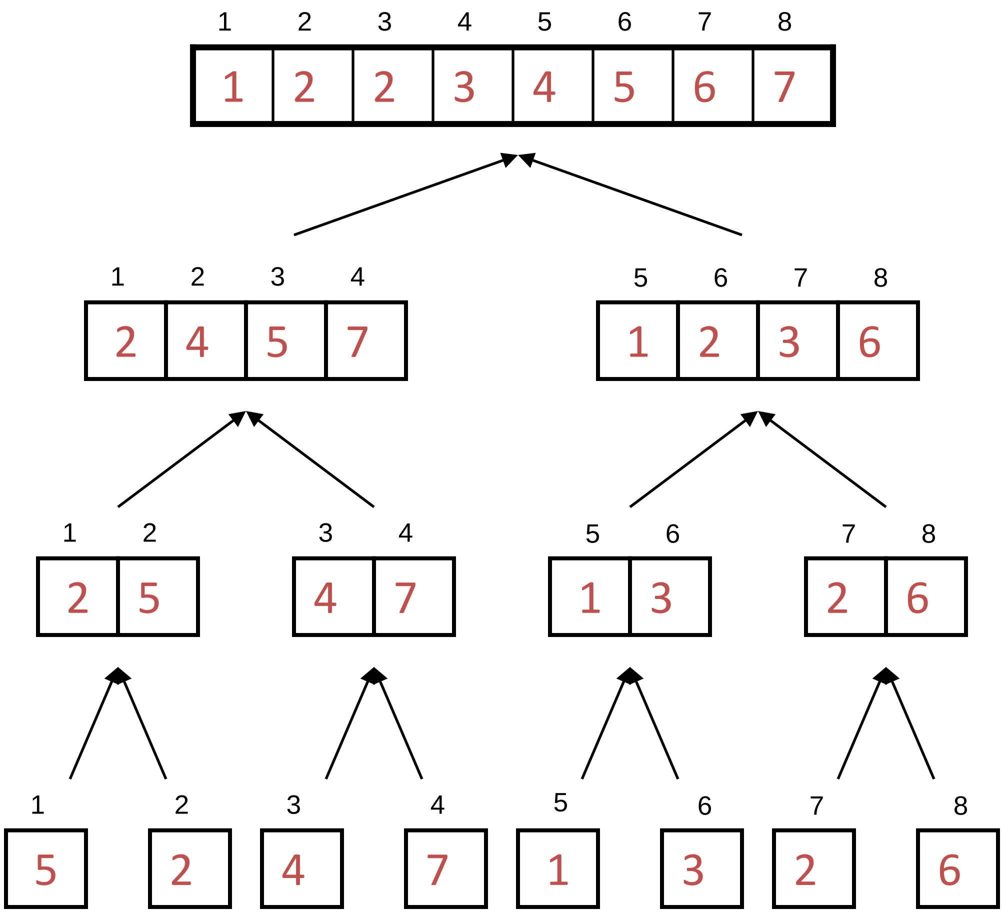
Divide



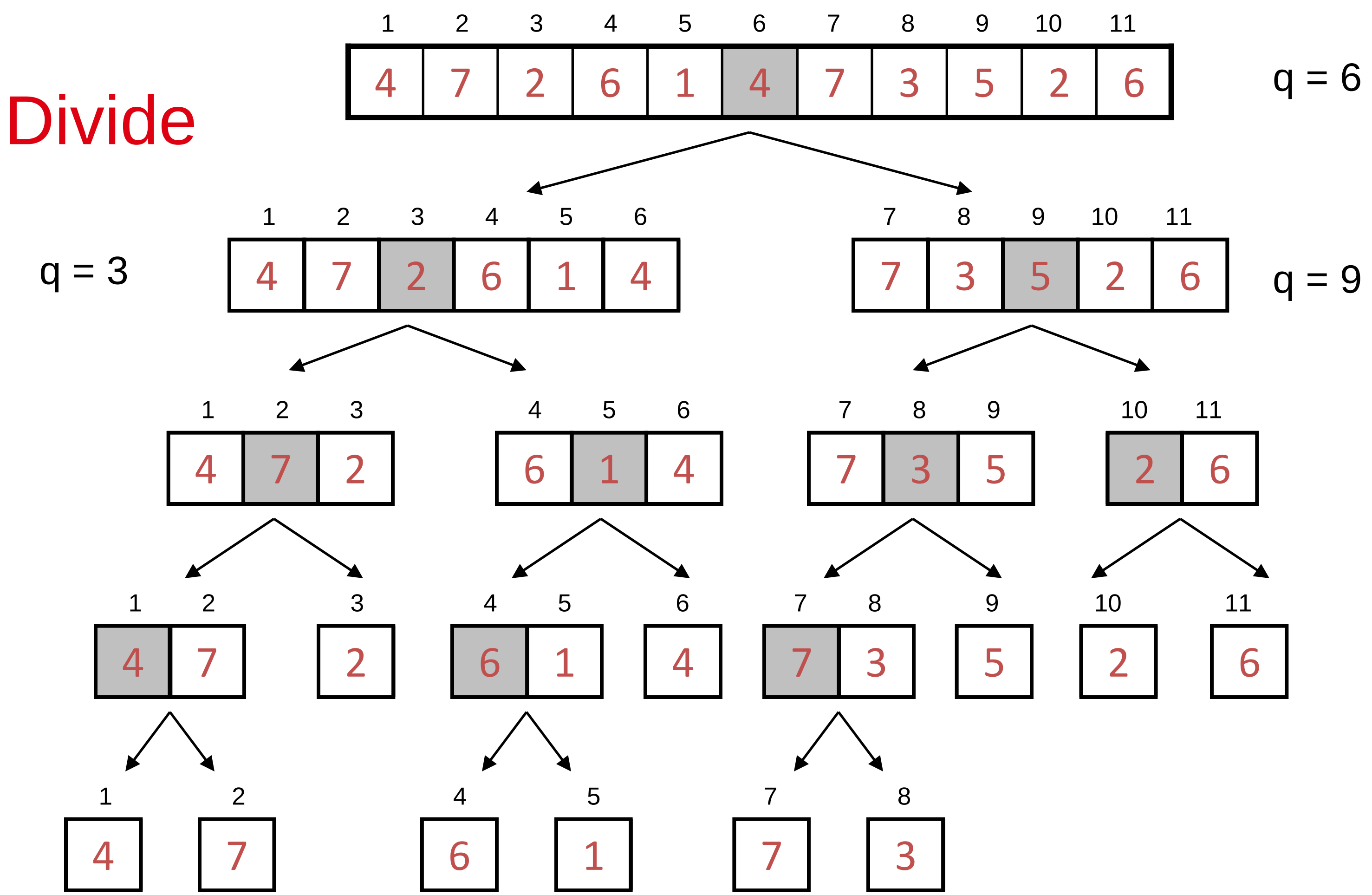
q = 4

Example – n Power of 2

Conquer
and
Merge

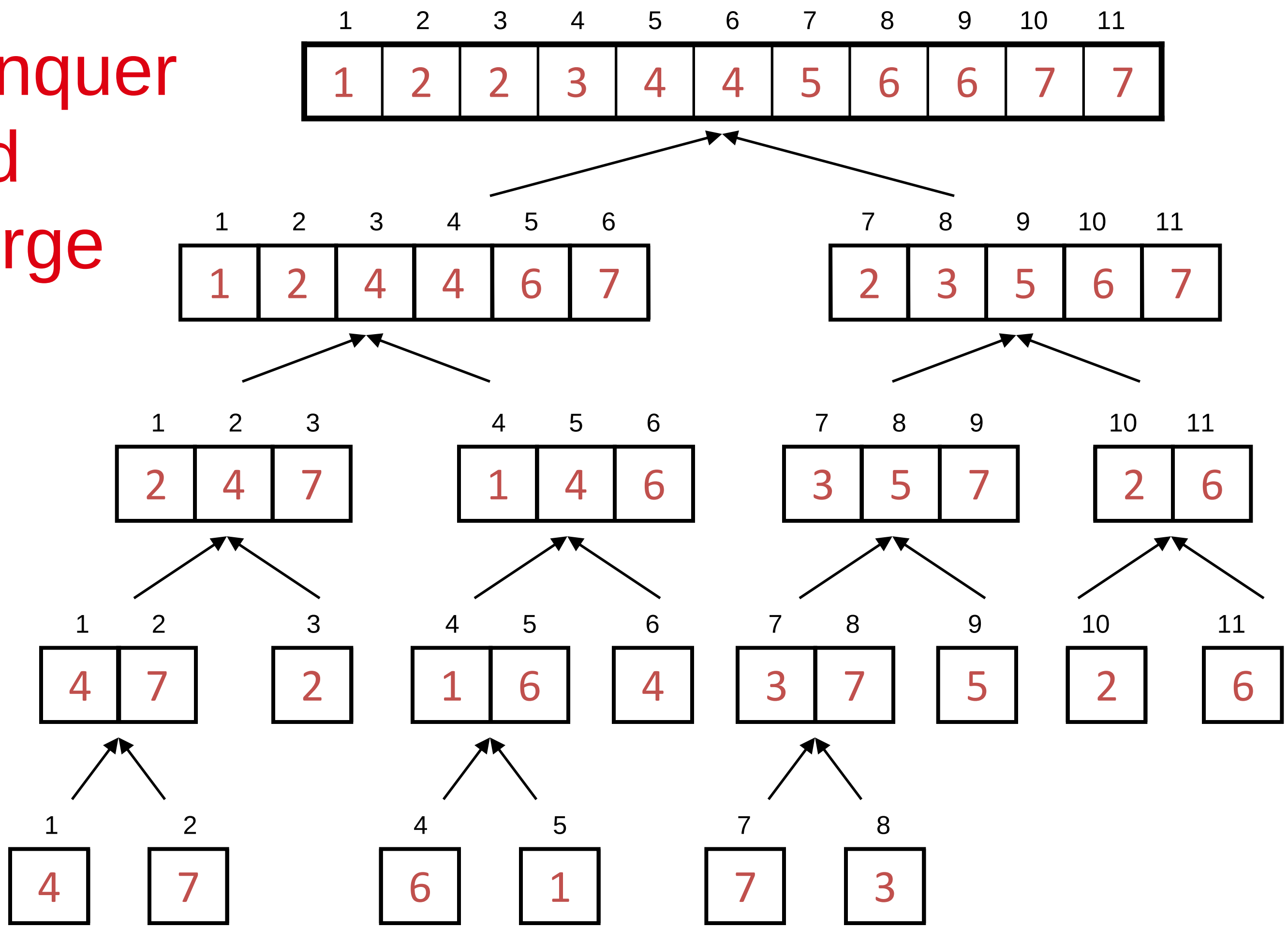


Example – n Not a Power of 2

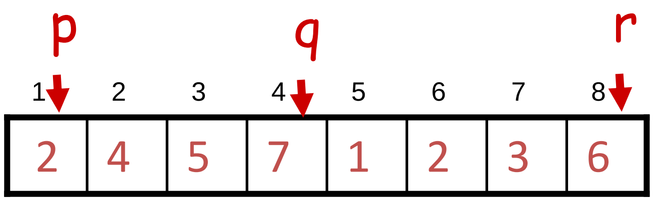


Example – n Not a Power of 2

Conquer and Merge



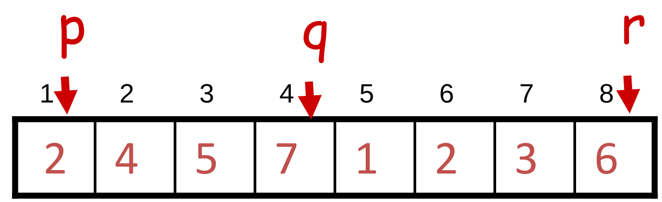
Merging

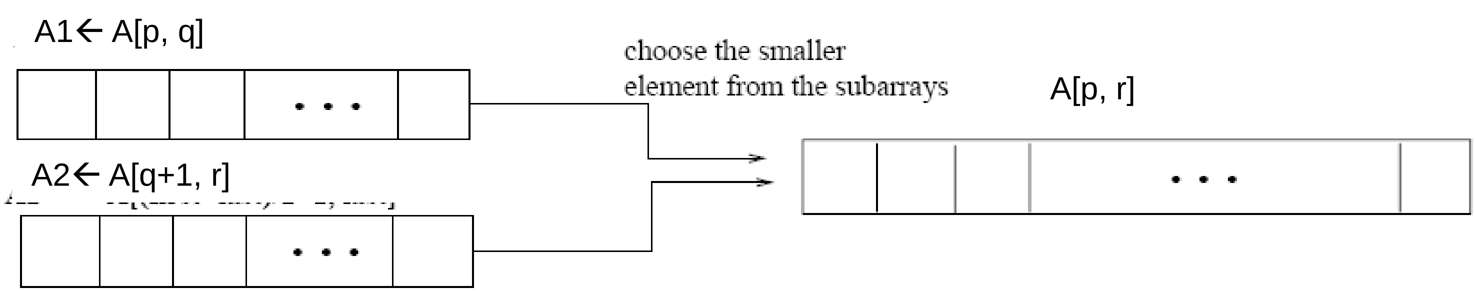


- **Input:** Array A and indices p, q, r such that $p \le q < r$
  - Subarrays $A[p\,..\,q]$ and $A[q+1\,..\,r]$ are sorted
- **Output:** One single sorted subarray $A[p\,..\,r]$

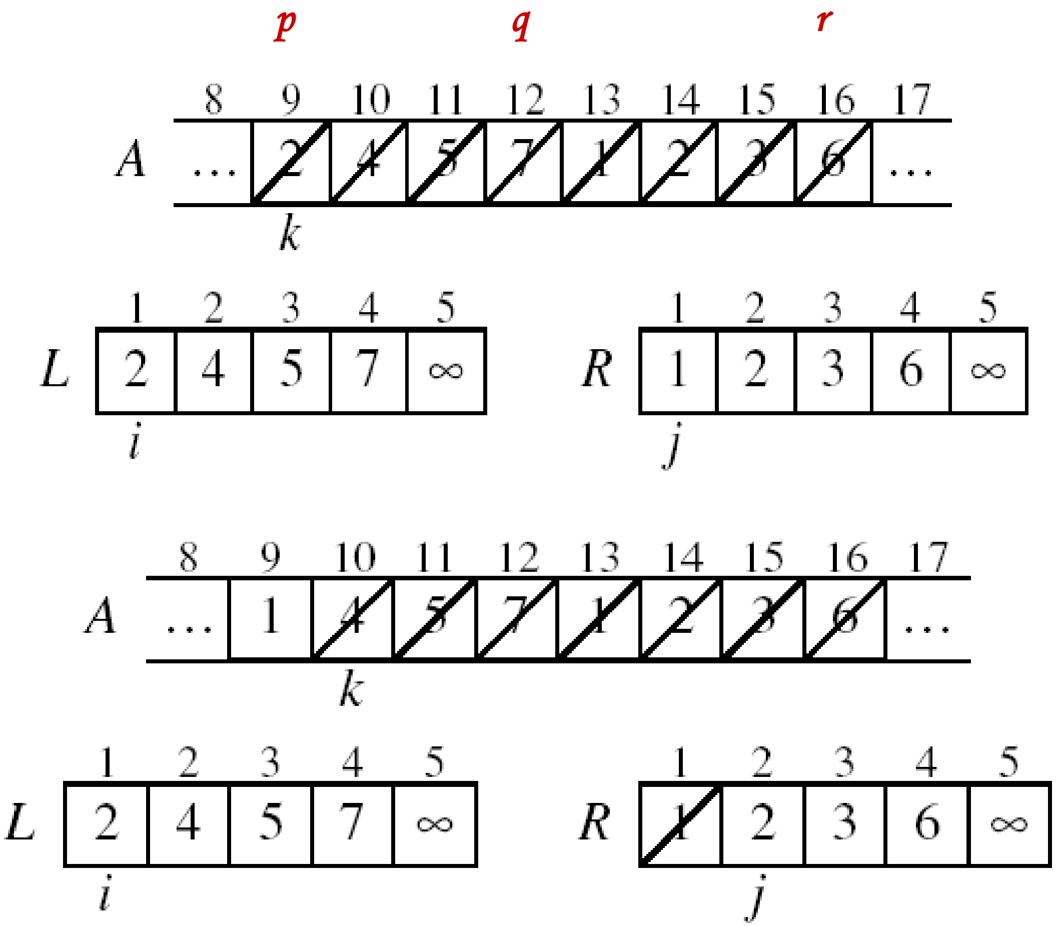
Merging

- Idea for merging:
  - Two piles of sorted cards
    - Choose the smaller of the two top cards
    - Remove it and place it in the output pile
  - Repeat the process until one pile is empty
  - Take the remaining input pile and place it face-down onto the output pile

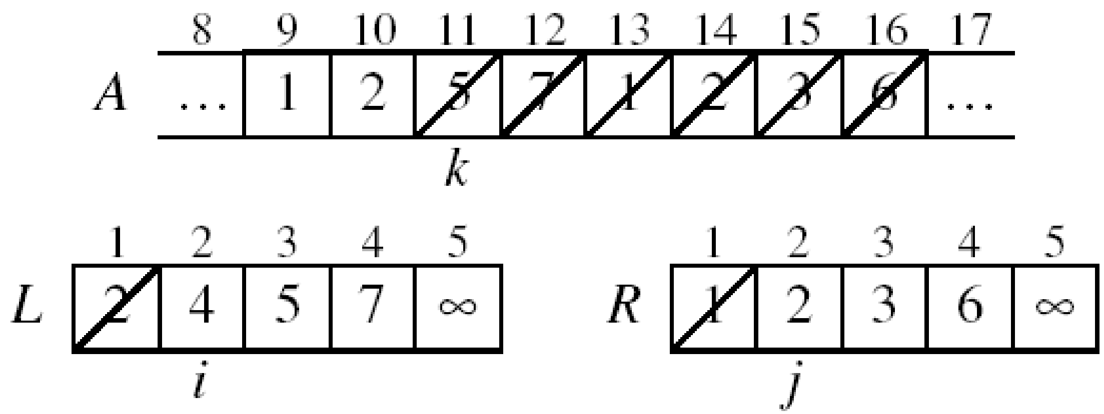


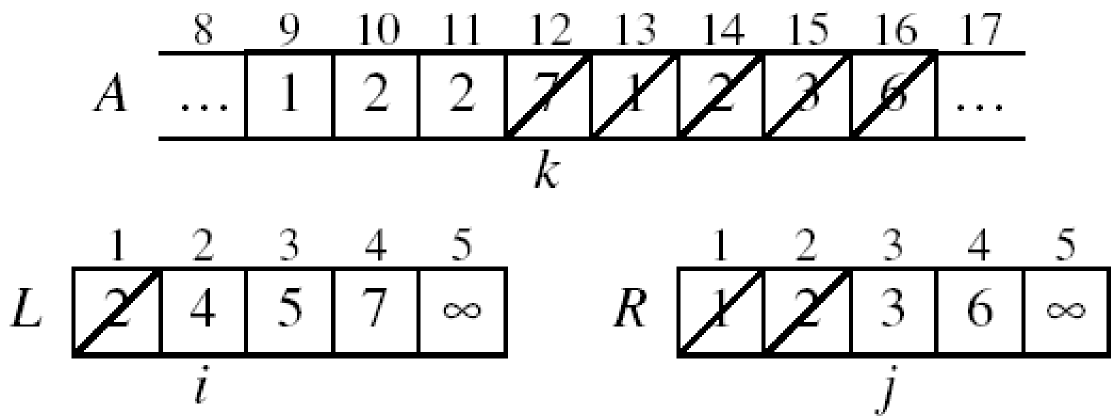


Example: MERGE(A, 9, 12, 16)



Example: MERGE(A, 9, 12, 16)





Example (cont.)

	8	9	10	11	12	13	14	15	16	17
A	…	1	2	2	3	~~1~~	~~2~~	~~3~~	~~6~~	…
						k				

	1	2	3	4	5
L	~~2~~	4	5	7	∞
		i			

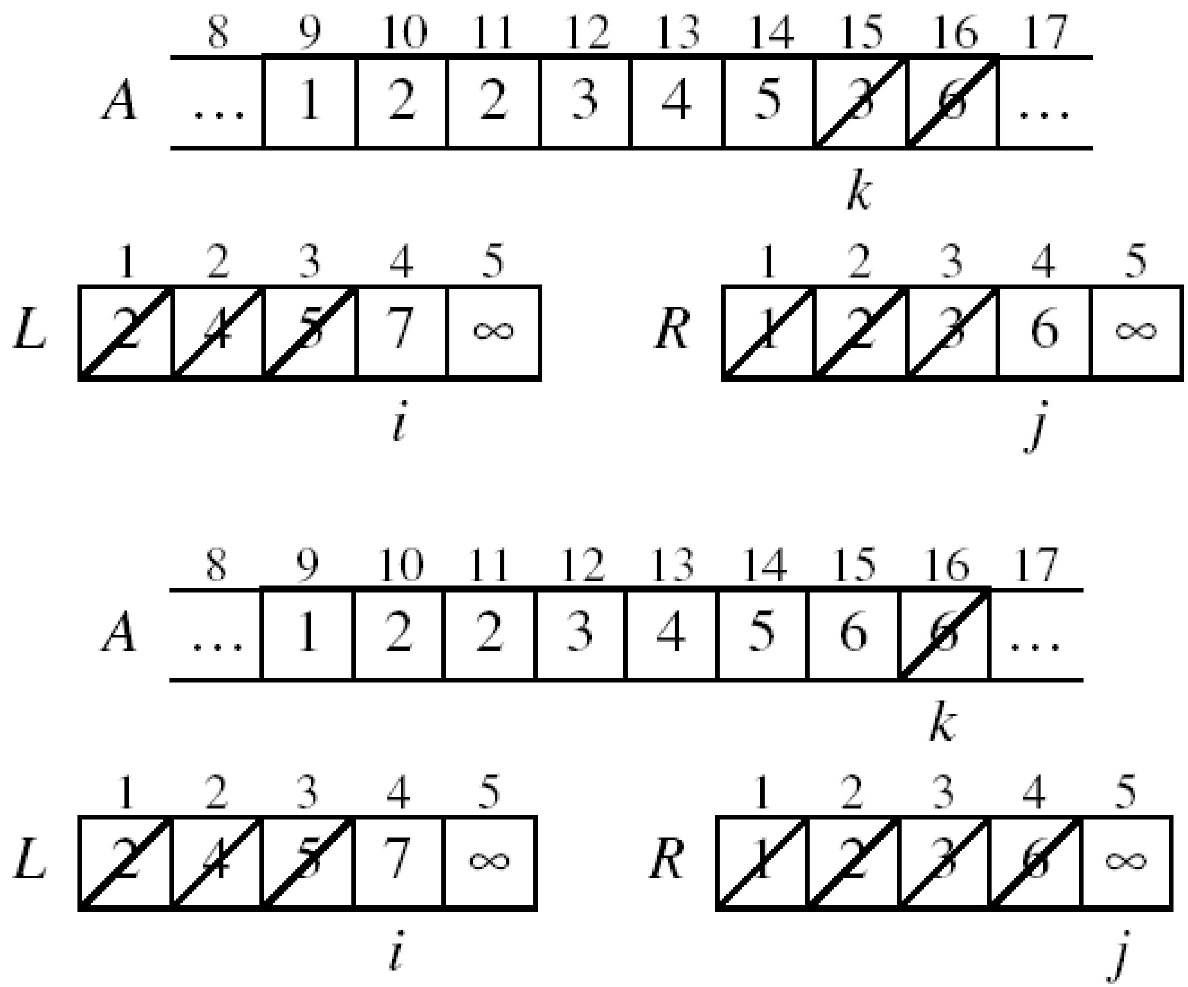
	1	2	3	4	5
R	~~1~~	~~2~~	~~3~~	6	∞
				j	

	8	9	10	11	12	13	14	15	16	17
A	…	1	2	2	3	4	~~2~~	~~3~~	~~6~~	…
							k			

	1	2	3	4	5
L	~~2~~	~~4~~	5	7	∞
			i		

	1	2	3	4	5
R	~~1~~	~~2~~	~~3~~	6	∞
				j	

Example (cont.)



Example (cont.)

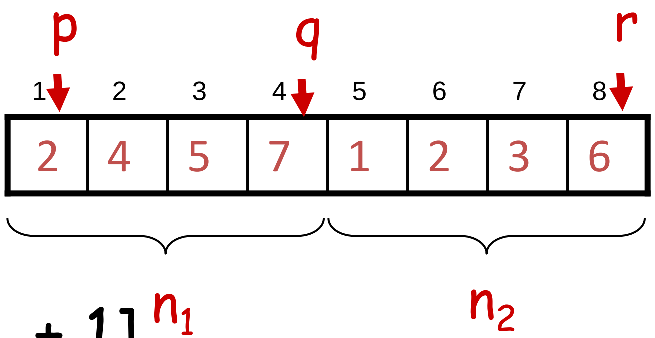
	8	9	10	11	12	13	14	15	16	17
A	…	1	2	2	3	4	5	6	7	…
										k

	1	2	3	4	5
L	~~2~~	~~4~~	~~5~~	~~7~~	∞
					i

	1	2	3	4	5
R	~~1~~	~~2~~	~~3~~	~~6~~	∞
					j

Done!

Merge - Pseudocode



Alg.: MERGE(A, p, q, r)

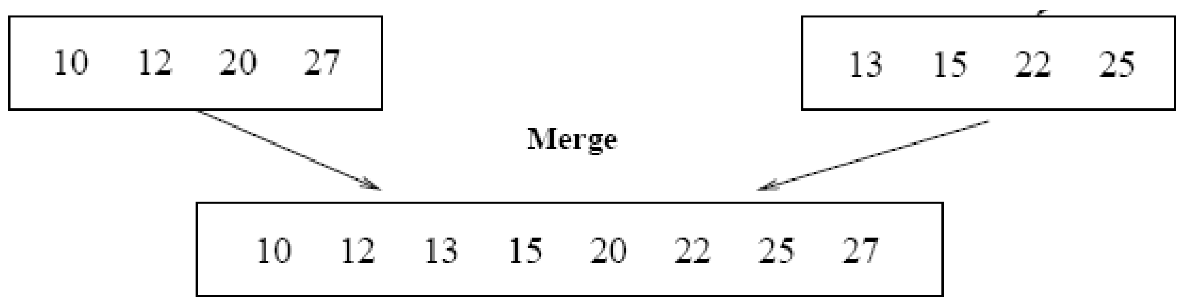
1. Compute n_1 and n_2
2. Copy the first n_1 elements into $L[1\ ..\ n_1 + 1]$ and the next n_2 elements into $R[1\ ..\ n_2 + 1]$
3. $L[n_1 + 1] \leftarrow \infty$; $R[n_2 + 1] \leftarrow \infty$
4. $i \leftarrow 1$; $j \leftarrow 1$
5. **for** $k \leftarrow p$ **to** r
6. **do if** $L[\ i\ ] \leq R[\ j\ ]$
7. **then** $A[k] \leftarrow L[\ i\ ]$
8. $i \leftarrow i + 1$
9. **else** $A[k] \leftarrow R[\ j\ ]$
10. $j \leftarrow j + 1$

	p			q	
L	2	4	5	7	∞

	q + 1			r	
R	1	2	3	6	∞

Running Time of Merge (assume last for loop)

- Initialization (copying into temporary arrays):
  - $\Theta(n_1 + n_2) = \Theta(n)$
- Adding the elements to the final array:
  - n iterations, each taking constant time $\Rightarrow \Theta(n)$
- Total time for Merge:
  - $\Theta(n)$



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 $C(n)$

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

MERGE-SORT Running Time

- **Divide:**
  - compute q as the average of p and r: $D(n) = \Theta(1)$
- **Conquer:**
  - recursively solve 2 subproblems, each of size $n/2 \Rightarrow 2T(n/2)$
- **Combine:**
  - MERGE on an n-element subarray takes $\Theta(n)$ time $\Rightarrow C(n) = \Theta(n)$

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ 2T(n/2) + \Theta(n) + \Theta(1) & \text{if } n > 1 \end{cases}$$

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ 2T(n/2) + \Theta(n) & \text{if } n > 1 \end{cases}$$

Solve the Recurrence

$$T(n) = \begin{cases} c & \text{if } n = 1 \\ 2T(n/2) + cn & \text{if } n > 1 \end{cases}$$

Use Master's Theorem:

Compare n with $f(n) = cn$

Case 2: $T(n) = \Theta(n\lg n)$

Merge Sort - Discussion

- Running time insensitive of the input

- Advantages:
  - Guaranteed to run in $\Theta(nlgn)$

- Disadvantage
  - Requires extra space $\approx$N

Quick sort

- An algorithm is a sequence of unambiguous instructions for solving a problem, i.e., for obtaining a required output for any legitimate input in a finite amount of time.
- A mathematical relation between an observed quantity and a variable used in a step-by-step mathematical process to calculate a quantity
- Algorithm is any well defined computational procedure that takes some value or set of values as input and produces some value or set of values as output
- A procedure for solving a mathematical problem in a finite number of steps that frequently involves repetition of an operation; broadly : a step-by-step procedure for solving a problem or accomplishing some end (Webster's Dictionary)

Quick sort

- Another divide-and-conquer sorting algorithm
- To understand quick-sort, let's look at a high-level description of the algorithm

1) **Divide** : If the sequence S has 2 or more elements, select an element x from S to be your **pivot**. Any arbitrary element, like the last, will do. Remove all the elements of S and divide them into 3 sequences:

   L, holds S's elements less than x

   E, holds S's elements equal to x

   G, holds S's elements greater than x

2) **Recurse**: Recursively sort L and G

3) **Conquer**: Finally, to put elements back into S in order, first inserts the elements of L, then those of E, and those of G.

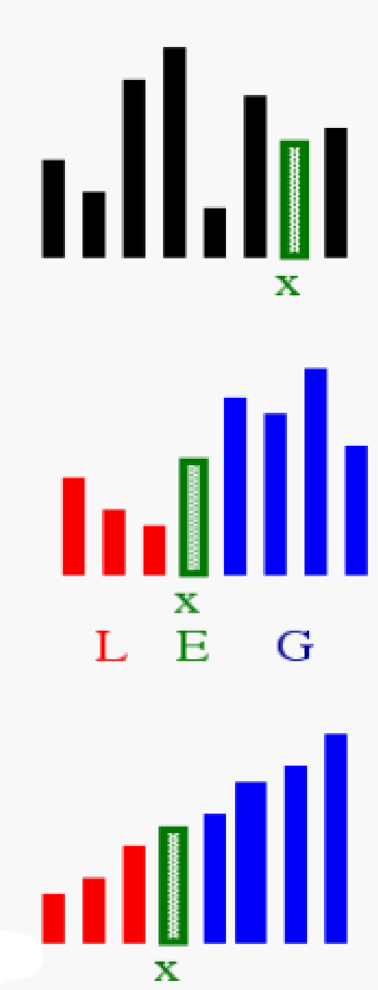
Here are some diagrams....

Idea of Quick Sort

1) **Select**: pick an element

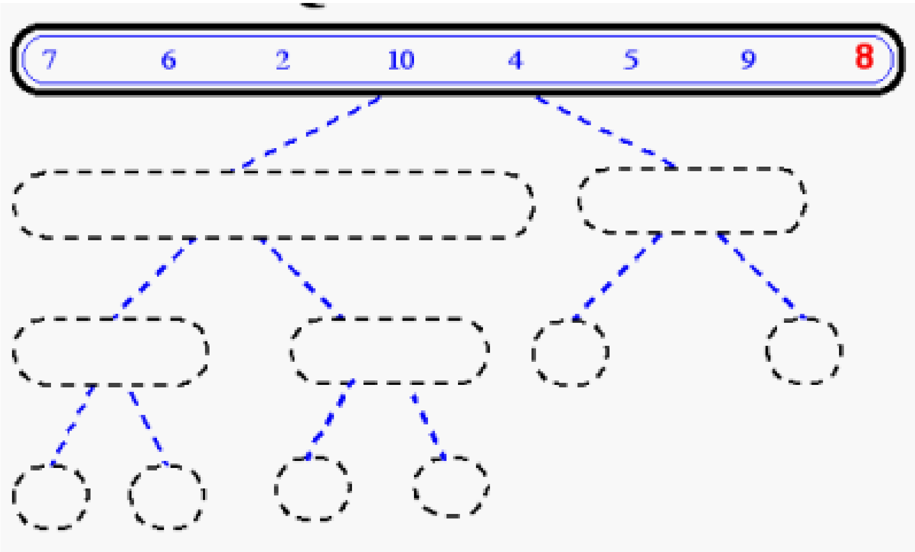
2) **Divide**: rearrange elements so that x goes to its final position E

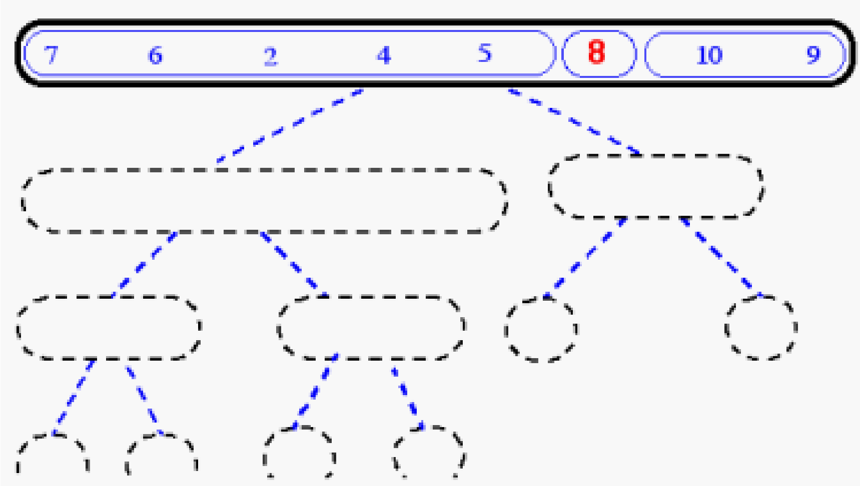
3) **Recurse and Conquer**: recursively sort



Quick sort

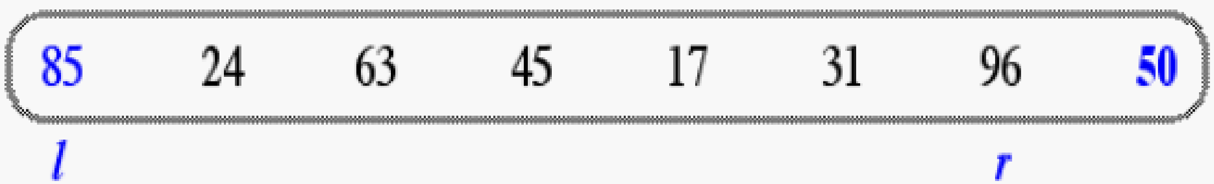
Quick-Sort Tree



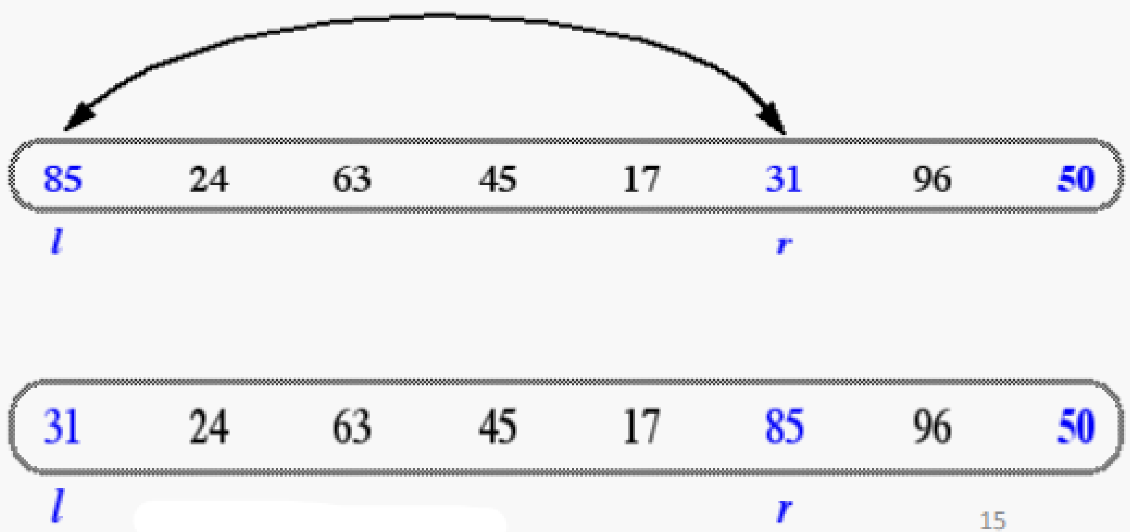


In-Place Quick-Sort

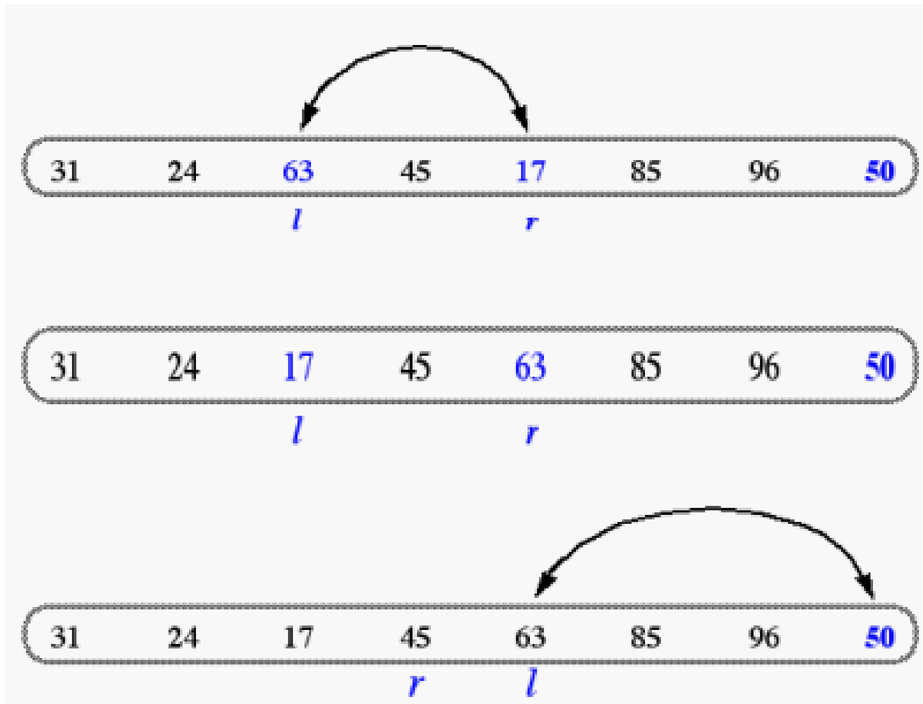
Divide step: l scans the sequence from the left, and r from the right.



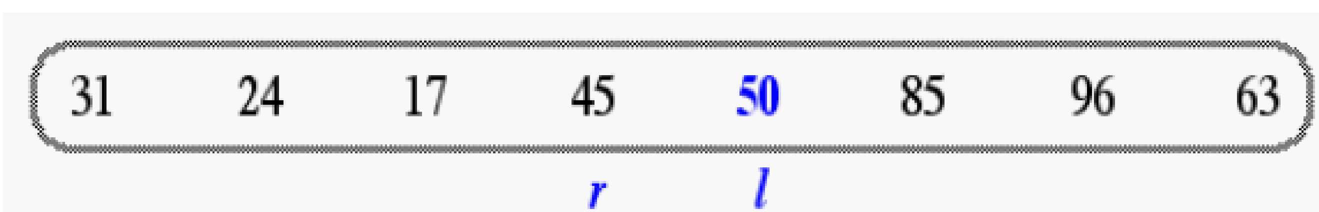
A swap is performed when l is at an element larger than the pivot and r is at one smaller than the



In Place Quick Sort (cont’d)



A final swap with the pivot completes the divide step



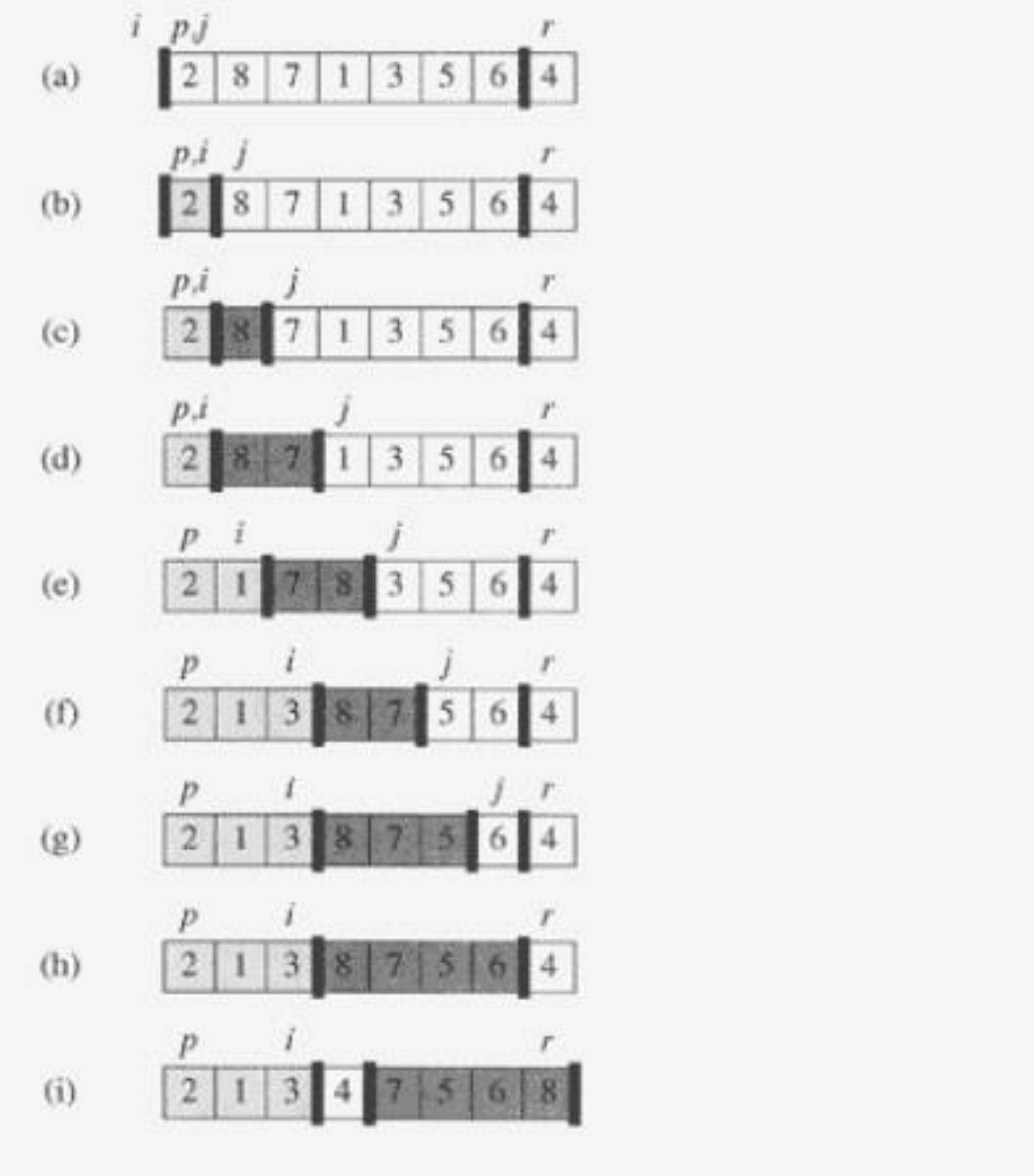
Quick sort

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

Quick sort

```
PARTITION(A, p, r)
  x ← A[r]
  i ← p − 1
  for j ← p to r − 1
       do if A[j] ≤ x
            then i ← i + 1
                 exchange A[i] ↔ A[j]
  exchange A[i + 1] ↔ A[r]
  return i + 1
```

Quick sort



(a) i p,j r
2 8 7 1 3 5 6 4
(b) p,i j r
2 8 7 1 3 5 6 4
(c) p,i j r
2 8 7 1 3 5 6 4
(d) p,i j r
2 8 7 1 3 5 6 4
(e) p i j r
2 1 7 8 3 5 6 4
(f) p i j r
2 1 3 8 7 5 6 4
(g) p i j r
2 1 3 8 7 5 6 4
(h) p i r
2 1 3 8 7 5 6 4
(i) p i r
2 1 3 4 7 5 6 8

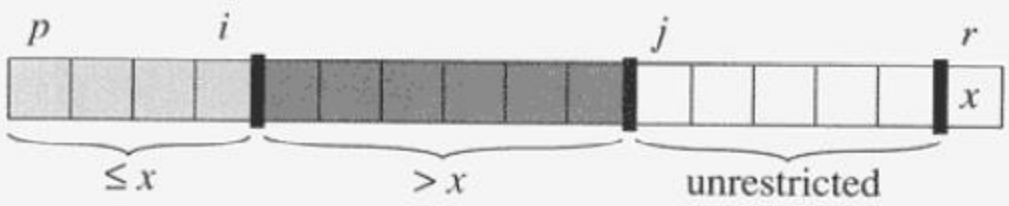


Figure 7.2 The four regions maintained by the procedure PARTITION on a subarray $A[p..r]$. The values in $A[p..i]$ are all less than or equal to x, the values in $A[i+1..j-1]$ are all greater than x, and $A[r] = x$. The values in $A[j..r-1]$ can take on any values.

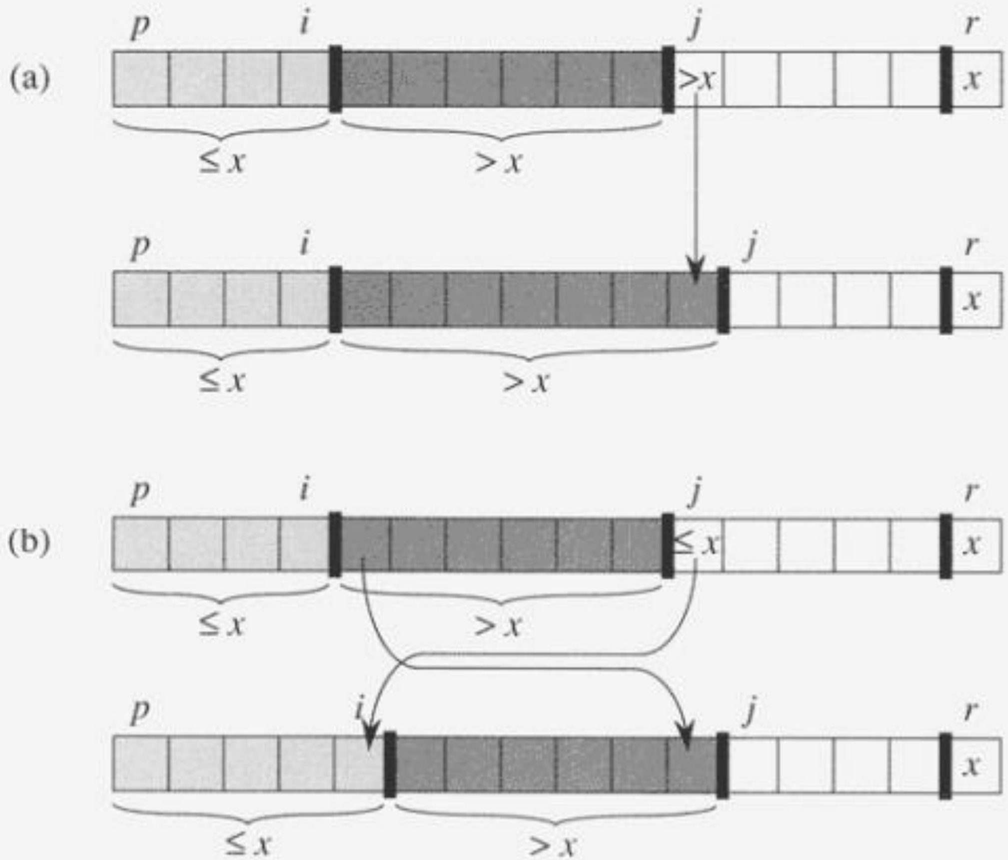


Figure 7.3 The two cases for one iteration of procedure PARTITION. **(a)** If $A[j] > x$, the only action is to increment j, which maintains the loop invariant. **(b)** If $A[j] \le x$, index i is incremented, $A[i]$ and $A[j]$ are swapped, and then j is incremented. Again, the loop invariant is maintained.

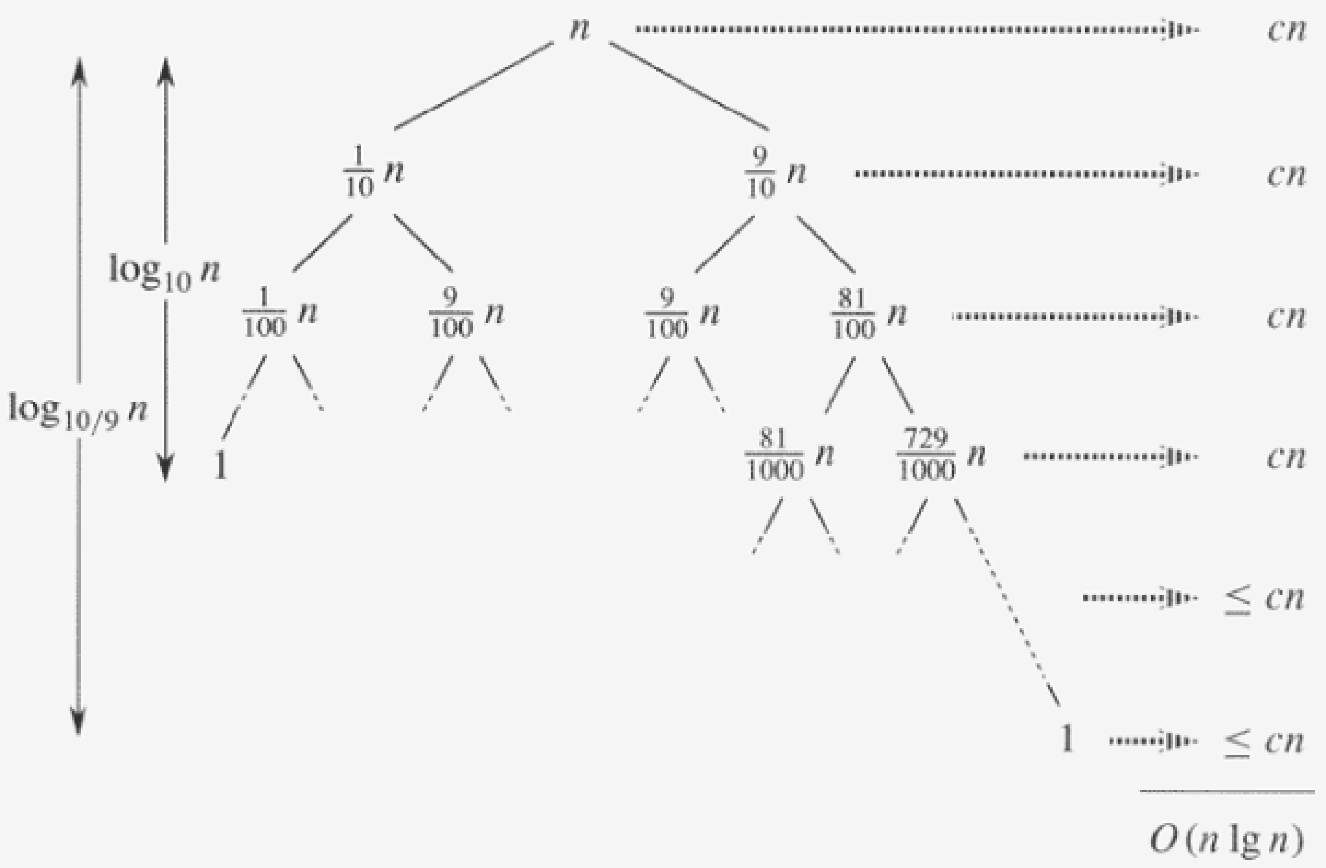
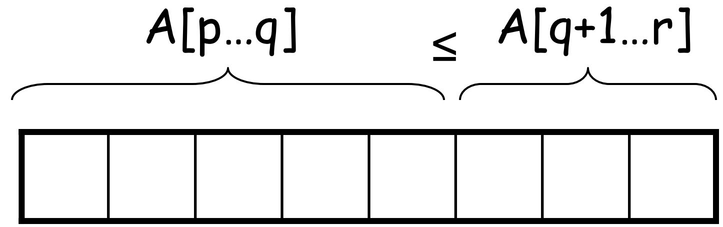


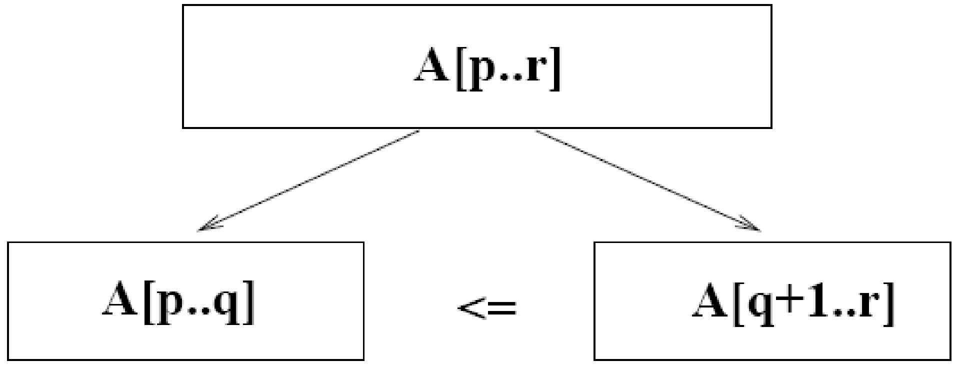
Figure 7.4 A recursion tree for QUICKSORT in which PARTITION always produces a 9-to-1 split, yielding a running time of $O(n \lg n)$. Nodes show subproblem sizes, with per-level costs on the right. The per-level costs include the constant c implicit in the $\Theta(n)$ term.

Quicksort

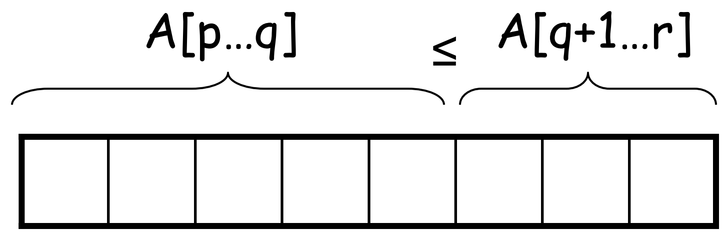
- **Sort an array $A[p...r]$**



- **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

Alg.: QUICKSORT(A, p, r)

Initially: p=1, r=n

if p < r

then $q \leftarrow$ PARTITION(A, p, r)

QUICKSORT (A, p, q)

QUICKSORT (A, q+1, r)

Recurrence:

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

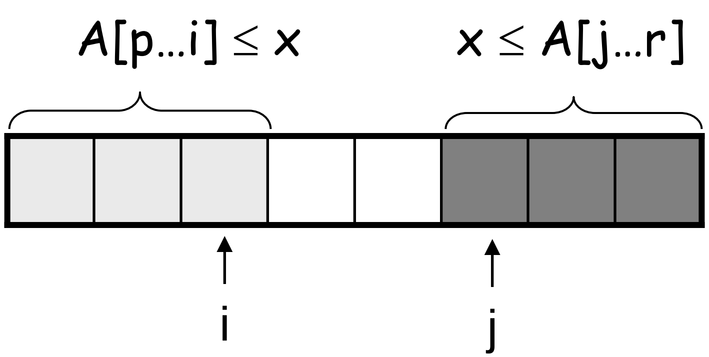
($f(n)$ depends on PARTITION())

Partitioning the Array

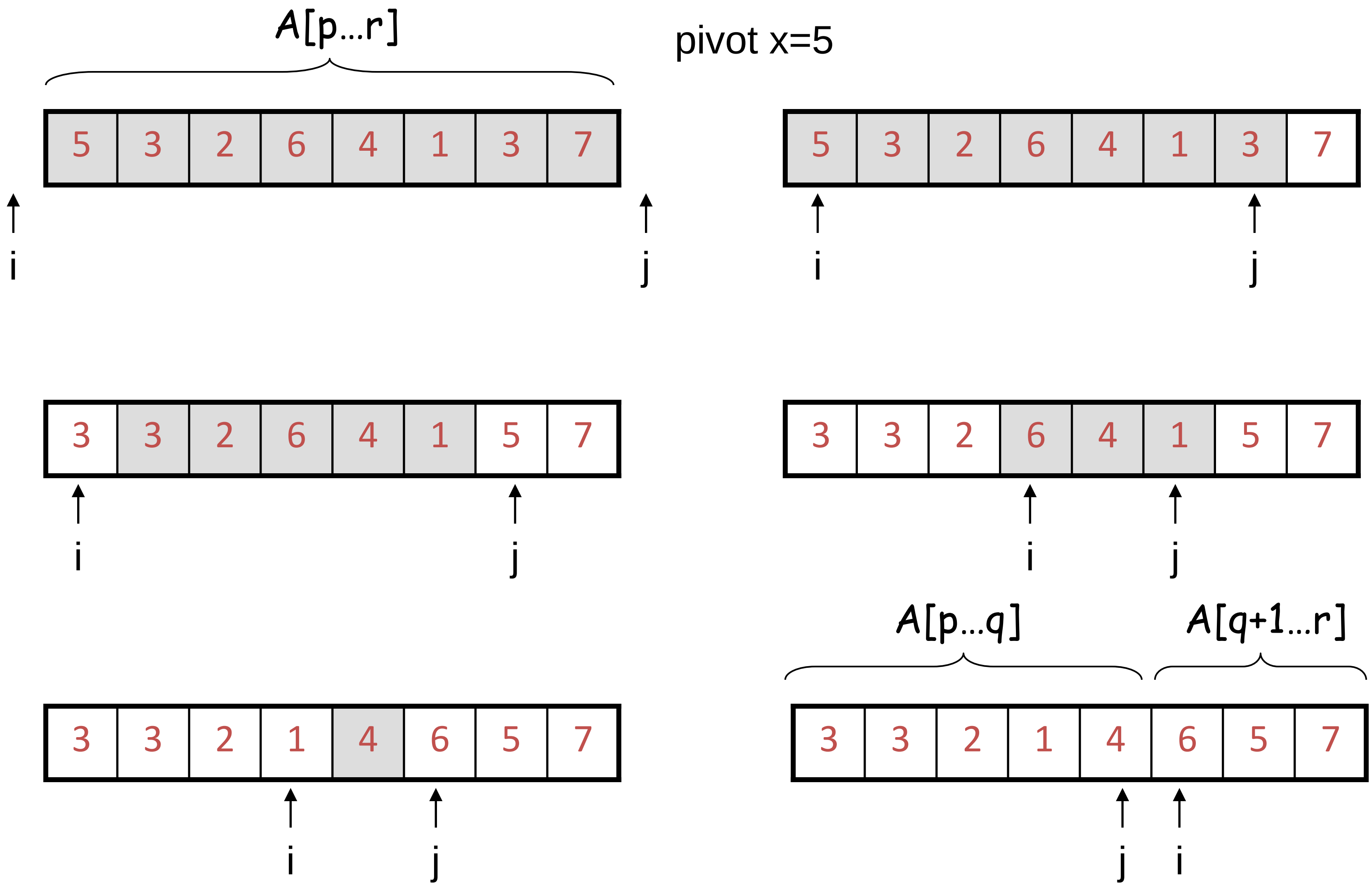
- Choosing PARTITION()
  - There are different ways to do this
  - Each has its own advantages/disadvantages
- How are partition
  - Select a pivot element **x** around which to partition
  - Grows two regions

    $A[p\ldots i] \le \mathbf{x}$

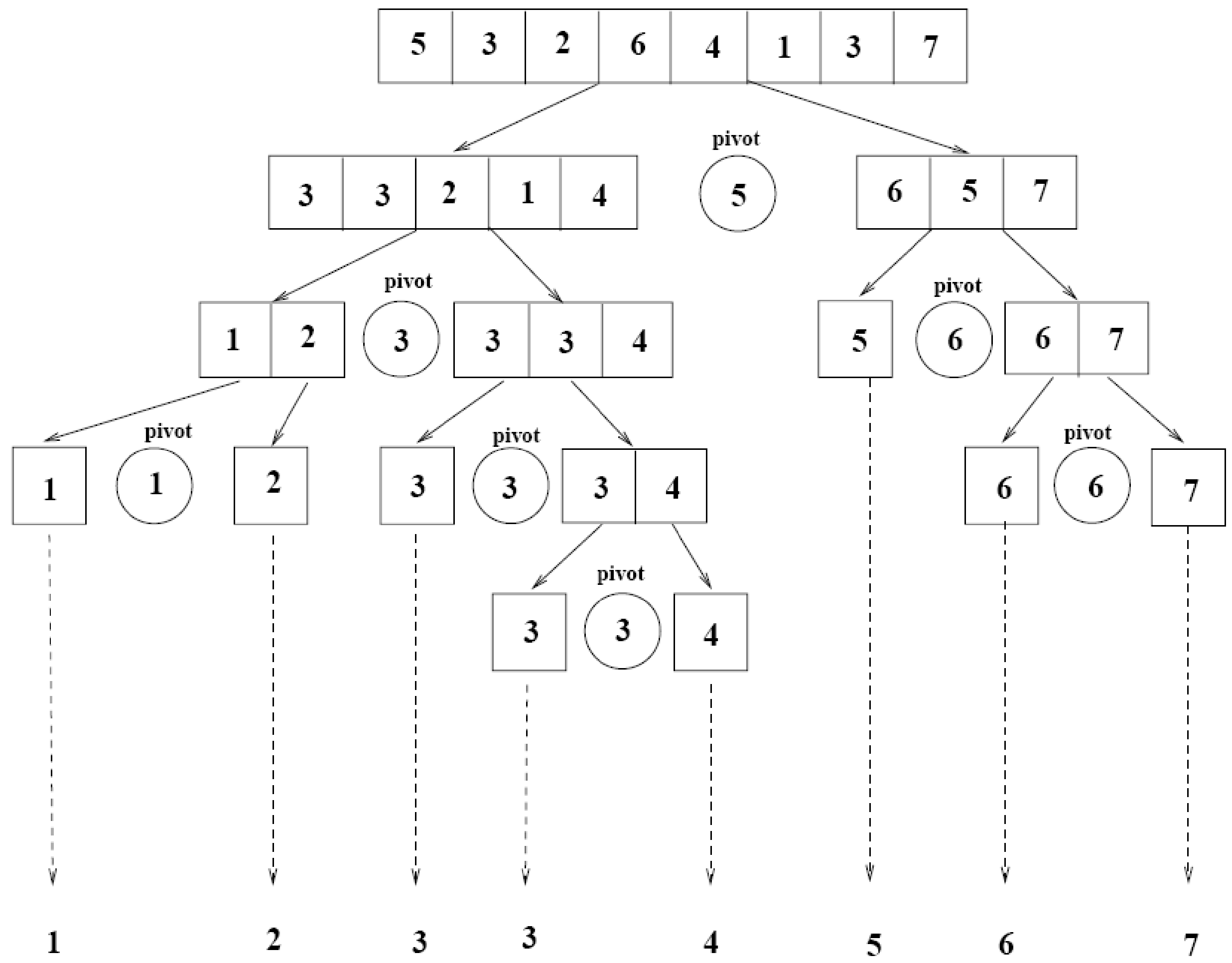
    $\mathbf{x} \le A[j\ldots r]$



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

Example

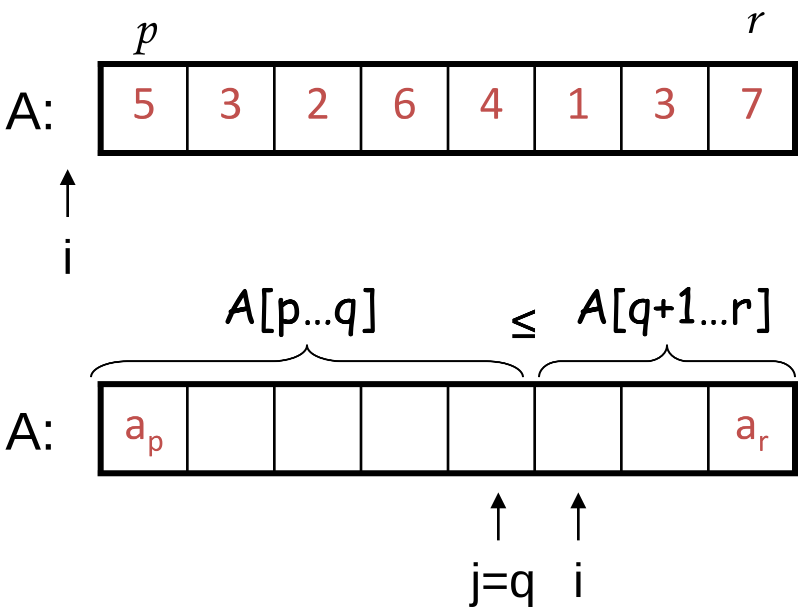


5 3 2 6 4 1 3 7
pivot
5
3 3 2 1 4
6 5 7
pivot
3
1 2
3 3 4
pivot
6
5
6 7
pivot
1
1
2
pivot
3
3
3 4
pivot
6
6
7
pivot
3
3
4
1 2 3 3 4 5 6 7

Partitioning the Array

Alg. PARTITION (A, p, r)

```
x ← A[p]        x=5
i ← p - 1
j ← r + 1
while TRUE
        do repeat j ← j - 1
              until A[j] ≤ x
        do  repeat i ← i + 1
              until A[i] ≥ x
        if i < j
              then exchange A[i] ↔ A[j]
        else return j
```



Each element is visited once!

Running time: $\Theta(n)$

$n = r - p + 1$

Recurrence

Alg.: QUICKSORT(A, p, r)

Initially: p=1, r=n

if p < r

then $q \leftarrow$ PARTITION(A, p, r)

QUICKSORT (A, p, q)

QUICKSORT (A, q+1, r)

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

https://www.youtube.com/watch?v=cnzIChso3cc

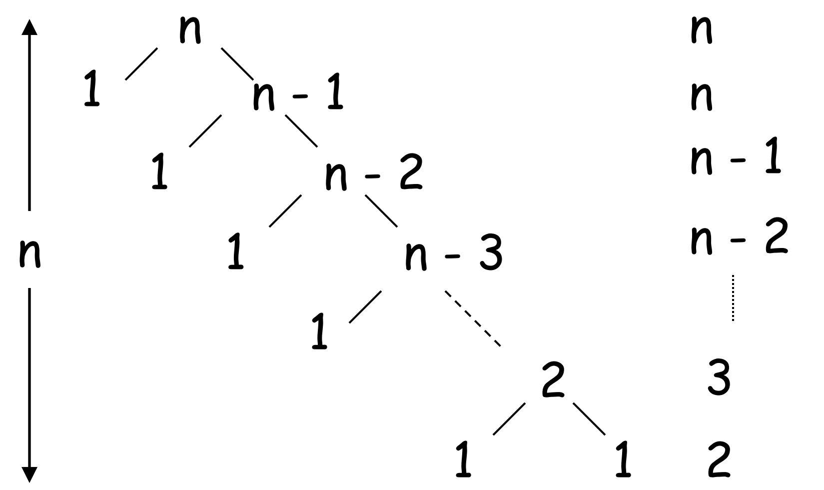
Worst Case Partitioning

- Worst-case partitioning   n(n-1)/2= n^2 -n
  - One region has one element and the other has n – 1 elements
  - Maximally unbalanced
- Recurrence: q=1

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

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

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



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

$\Theta(n^2)$

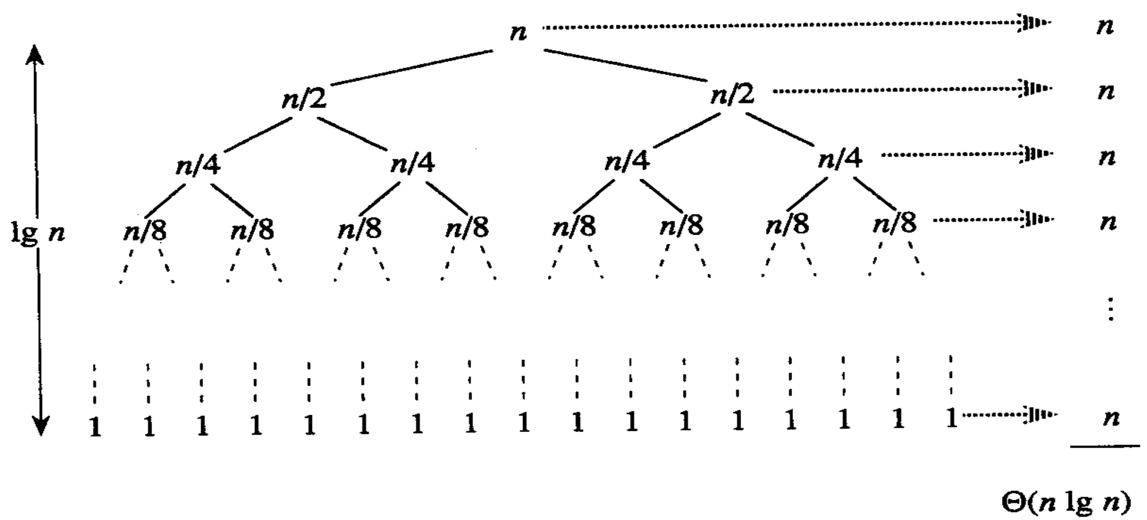
When does the worst case happen?

Best Case Partitioning

- Best-case partitioning
  - Partitioning produces two regions of size $n/2$
- Recurrence: $q=n/2$
- $T(n) = T(q) + T(n – q) + n = T(n/2) + T(n/2) + n$

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

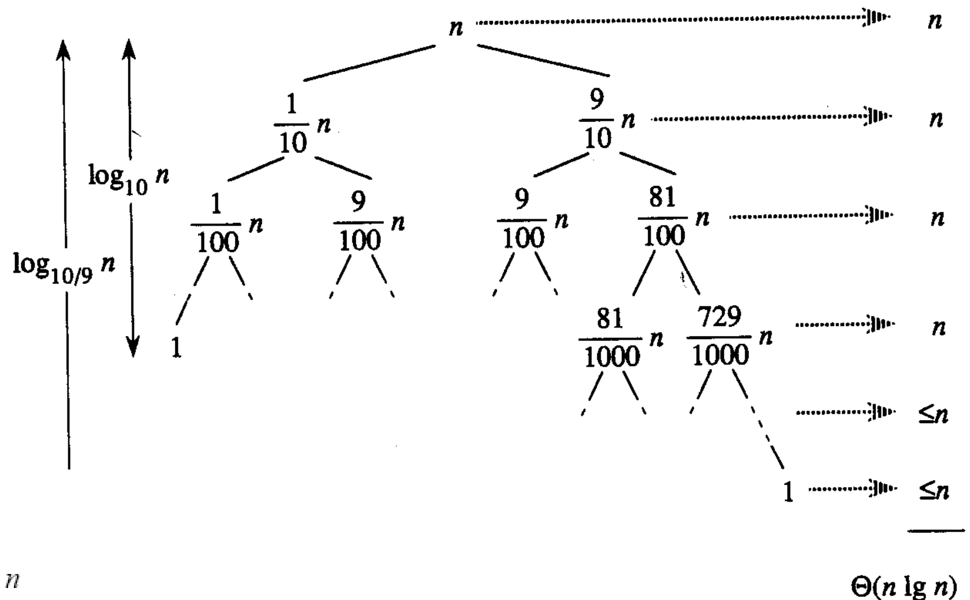
  $T(n) = \Theta(n\lg n)$ (Master theorem)



Case Between Worst and Best

- 9-to-1 proportional split

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



- Using the recursion tree:

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

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

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

How does partition affect performance?

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

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

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

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

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

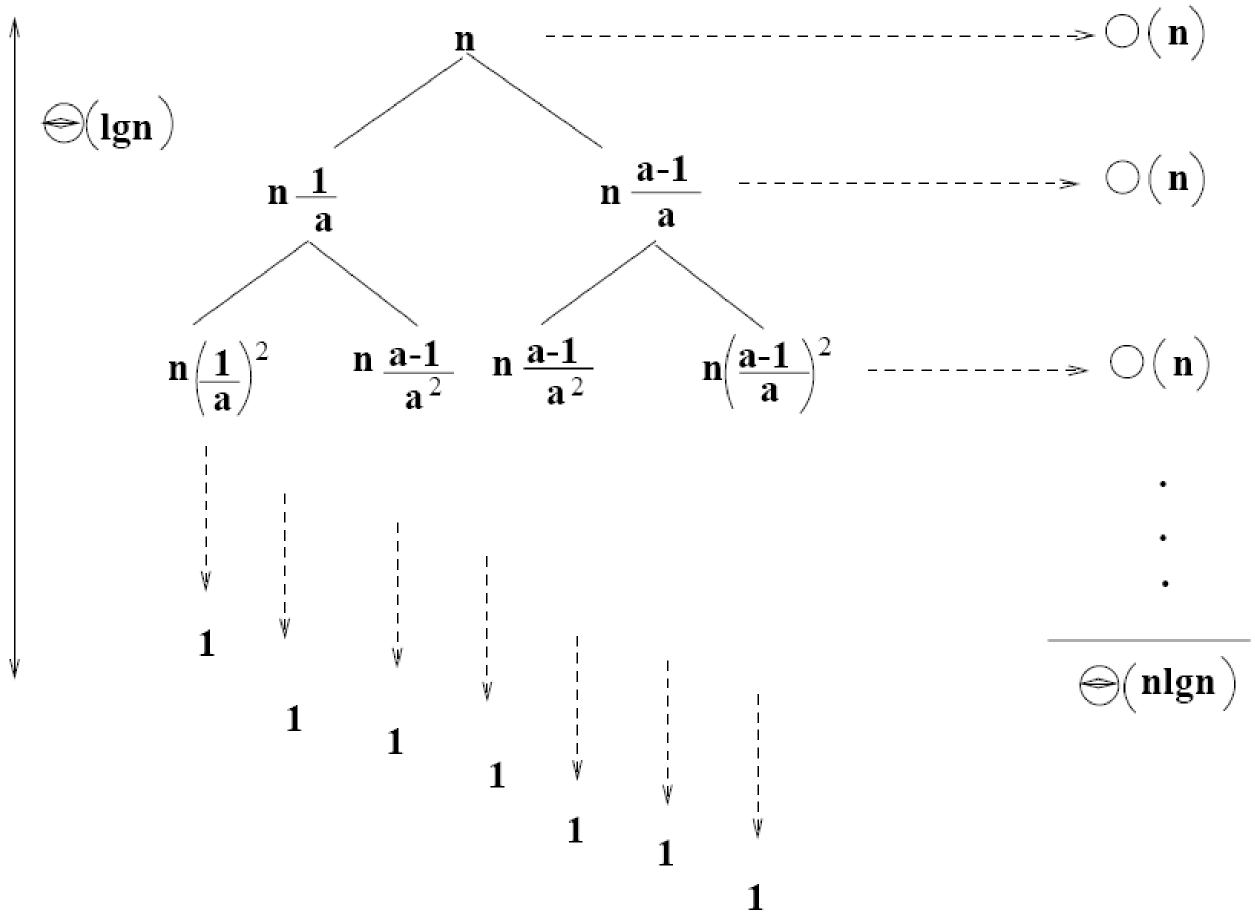
- Consider the $(9n/10 : n/10)$ splitting:

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

How does partition affect performance?

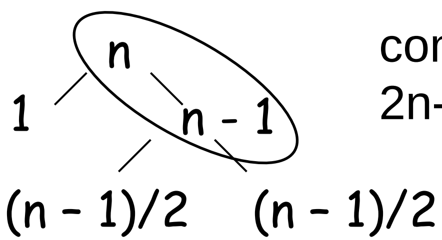
- Any $((a-1)n/a : n/a)$ splitting:

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



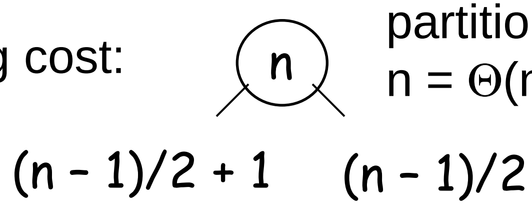
Performance of Quicksort

- 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:
$2n-1 = \Theta(n)$

Alternate of a good and a bad split



partitioning cost:
$n = \Theta(n)$

Nearly well balanced split

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

Sorting Challenge 1

Problem: Sort a file of huge records with tiny keys

Example application: Reorganize your MP-3 files

Which method to use?

A. merge sort, guaranteed to run in time ~NlgN
B. selection sort
C. bubble sort
D. a custom algorithm for huge records/tiny keys
E. insertion sort

Sorting Files with Huge Records and Small Keys

- Insertion sort or bubble sort?
  - NO, too many exchanges
- Selection sort?
  - YES, it takes linear time for exchanges –O(n)
- Merge sort or custom method?
  - Probably not: selection sort simpler, does less swaps

Sorting Challenge 2

Problem: Sort a huge randomly-ordered file of small records

Application: Process transaction record for a phone company

Which sorting method to use?

A. Bubble sort
B. Selection sort
C. Mergesort guaranteed to run in time ~NlgN
D. Insertion sort

Sorting Huge, Randomly - Ordered Files

- Selection sort?
  - NO, always takes quadratic time
- Bubble sort?
  - NO, quadratic time for randomly-ordered keys
- Insertion sort?
  - NO, quadratic time for randomly-ordered keys
- Mergesort?
  - YES, it is designed for this problem

Sorting Challenge 3

Problem: sort a file that is already almost in order

Applications:

- Re-sort a huge database after a few changes
- Doublecheck that someone else sorted a file

Which sorting method to use?

A. Mergesort, guaranteed to run in time ~NlgN
B. Selection sort
C. Bubble sort
D. A custom algorithm for almost in-order files
E. Insertion sort

Sorting Files That are Almost in Order

- Selection sort?
  - NO, always takes quadratic time
- Bubble sort?
  - NO, bad for some definitions of “almost in order”
  - Ex: B C D E F G H I J K L M N O P Q R S T U V W X Y Z A
- Insertion sort?
  - YES, takes linear time for most definitions of “almost in order”
- Mergesort or custom method?
  - Probably not: insertion sort simpler and faster

Finding minimum and maximum algorithms

- Problem : Find Minimum and Maximum number from the given list.

Example

50	40	-5	-9	45	90	65	25	75

Finding minimum and maximum algorithms

- Problem : Find Minimum and Maximum number from the given list.

Example

50	40	-5	-9	45	90	65	25	75

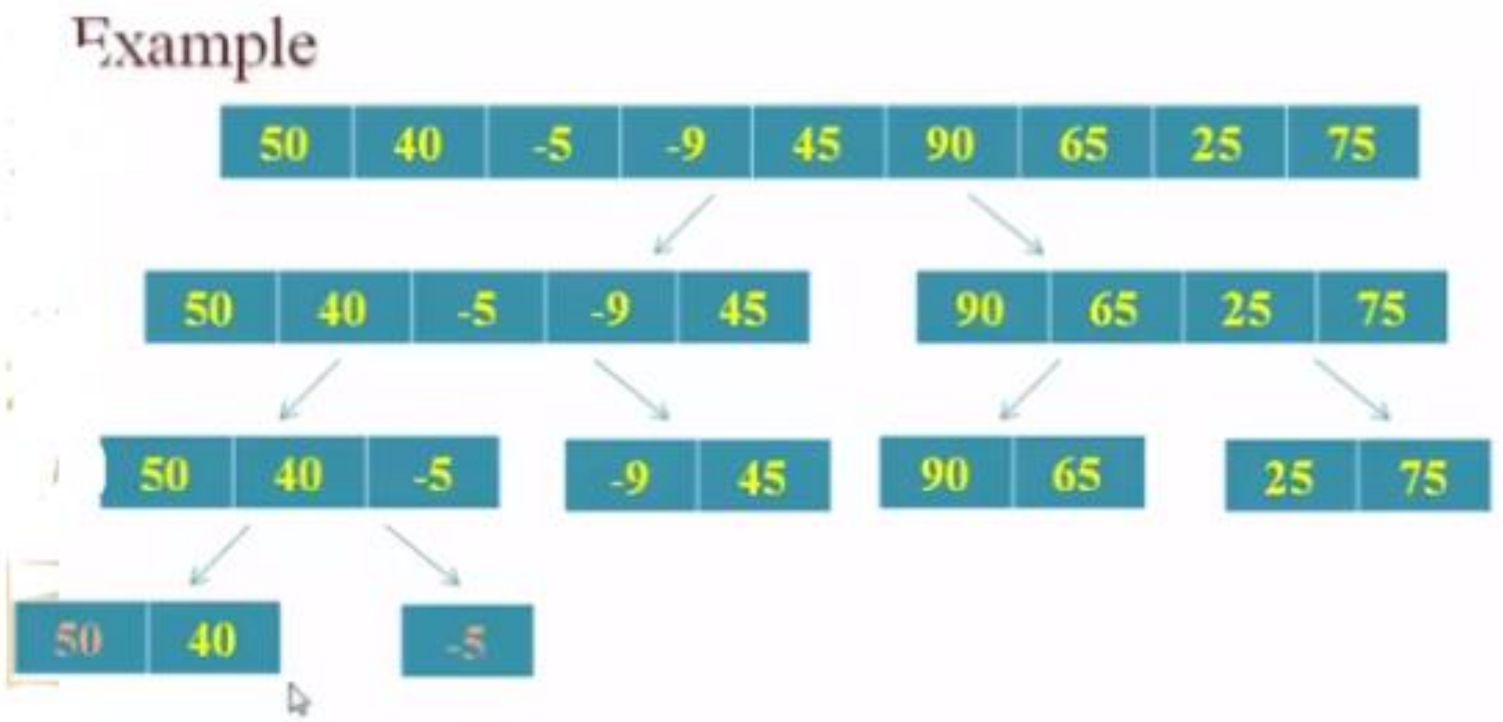
n-1 comparisons to find min value

n-1 comparisons to find max value

So 2n-2 comparisons in Classes method.

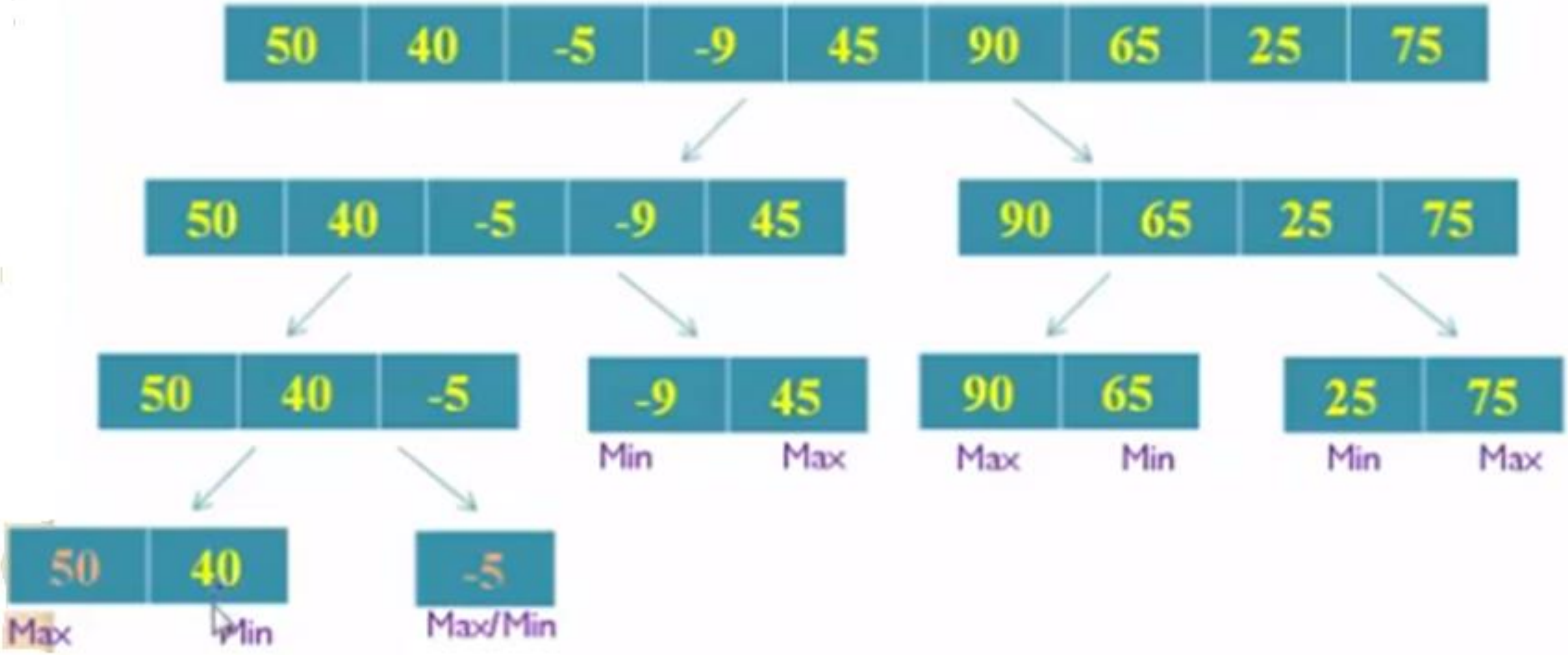
To reduce number of comparisons we can use **Divide and Conquer strategy**.

Finding minimum and maximum algorithms



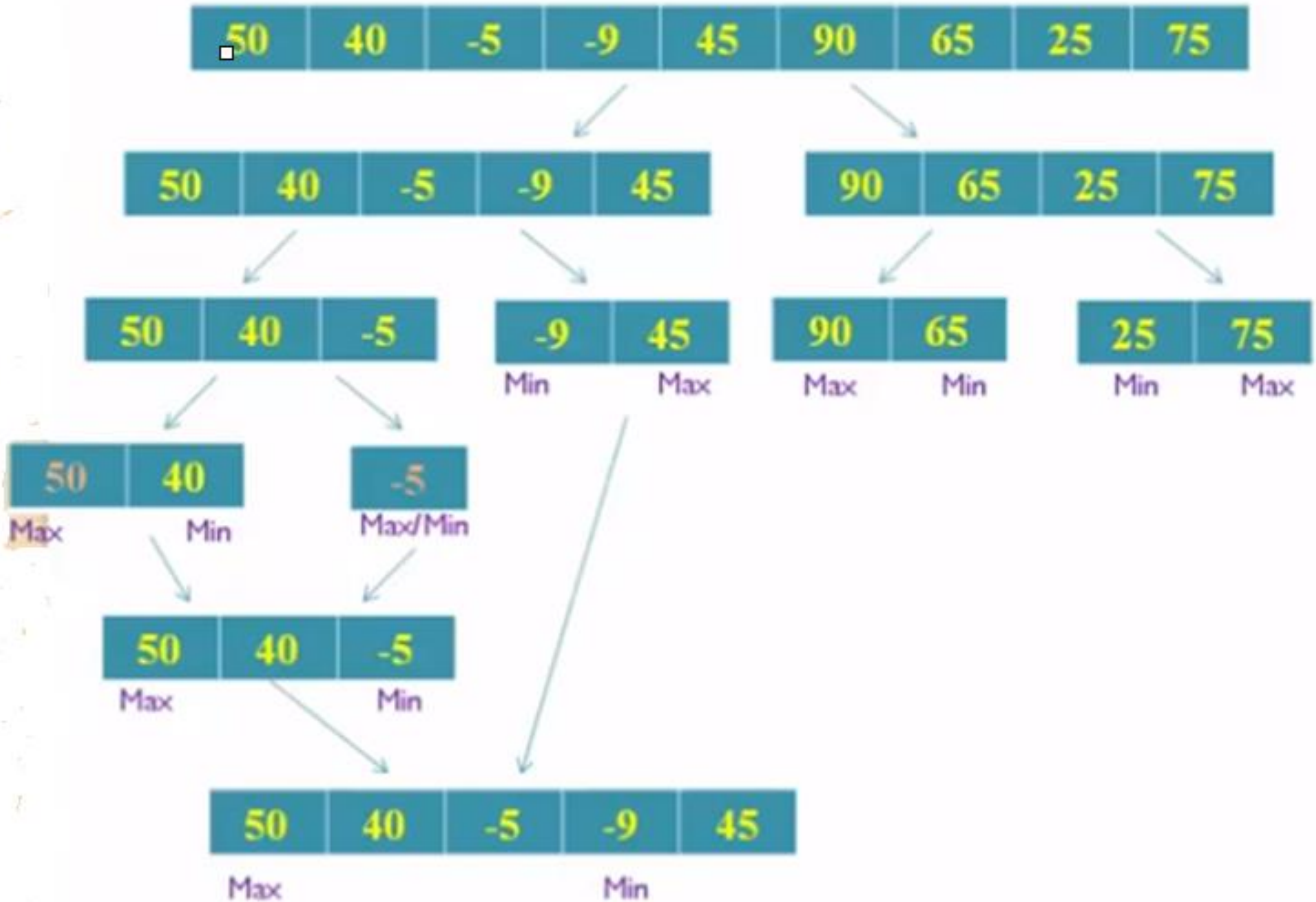
Finding minimum and maximum algorithms

Example



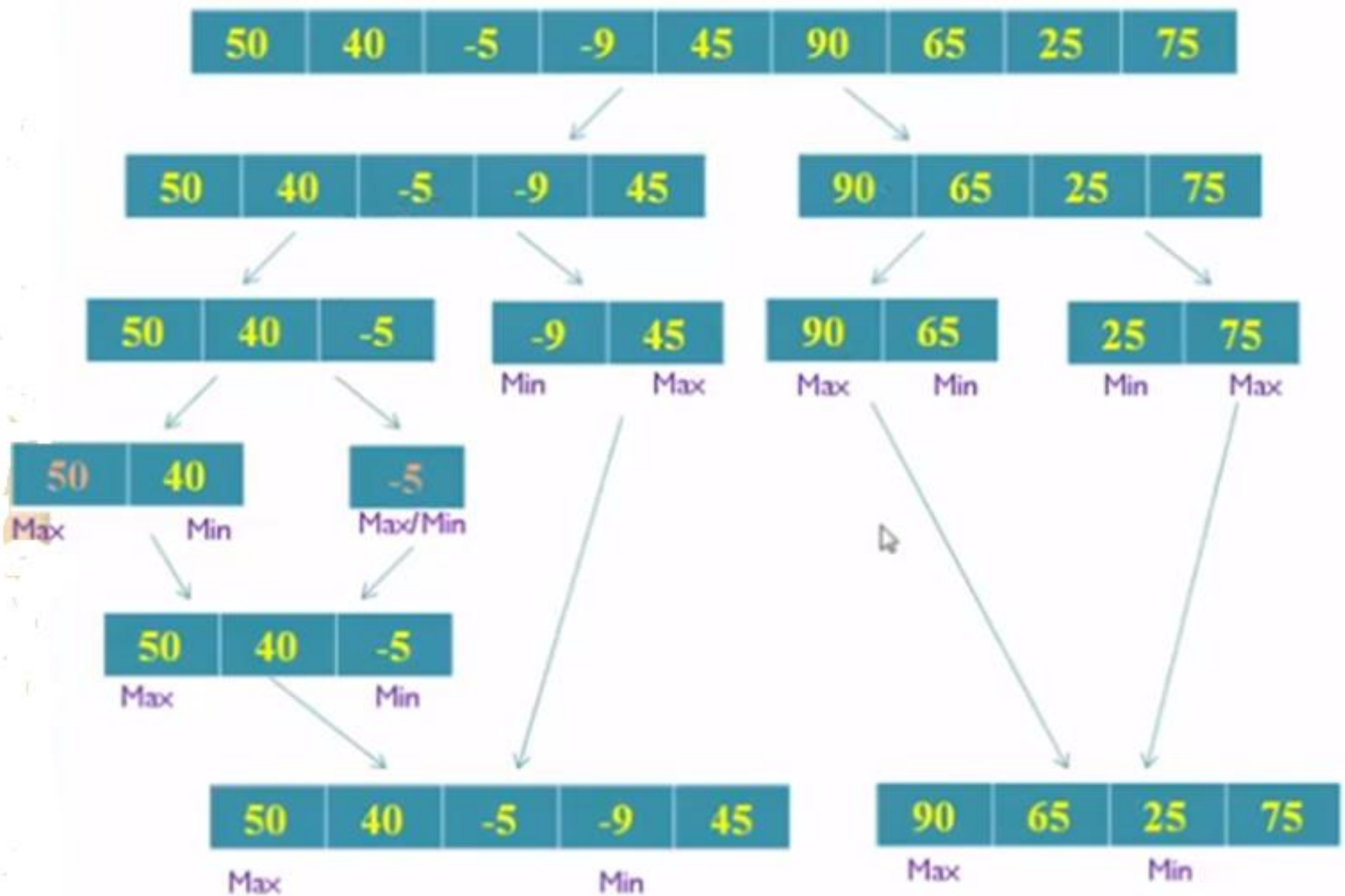
Finding minimum and maximum algorithms

Example



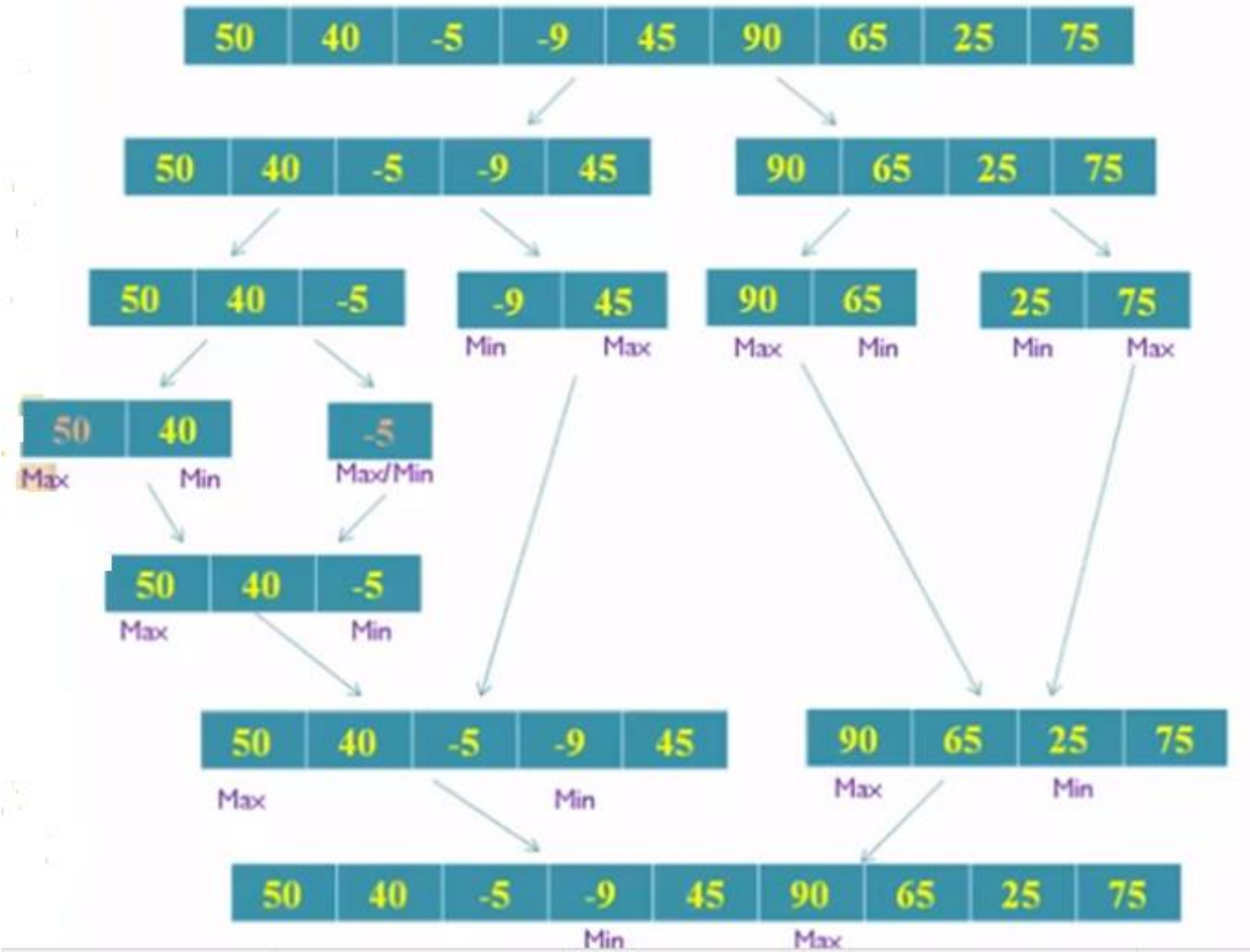
Finding minimum and maximum algorithms

Example



Finding minimum and maximum algorithms

Example



Finding minimum and maximum algorithms

Min Max algorithm

```
Algorithm Max_Min(i
    ,j,max,min)
{       if ( i == j )
        {       max ← A[i]
                min ← A[j]
        }
        else if (i = j – 1) then
        {   if (A[i] < A[j]) then
            {       max ← A[j]
                    min ← A[i]
            }
            else
            {       max ← A[i]
                    min ← A[j]
            }
        }
```

```
else
{   mid ← (i + j) / 2
    Max_Min(i , mid, max , min)
    Max_Min(mid+1 , j, max_new , min_new)

    if (max < max_new ) then
            max ← max_new

    if (min > min_new ) then
            min ← min_new
}
```

Analysis of MaxMin Algorithm

In analyzing the time complexity of this algorithm, concentrate on the number of element comparisons.

- If only one element is present in given list then no comparison required. $T(n)= 0$ ....$n=1$
- If there are two elements in given list then we require one comparison. $T(n)= 1$ ....$n=2$
- If there are more than two elements in given list then we require to implement given algorithm and it takes

  $T(n) = T(n/2)+T(n/2)+2$ ....$n>2$

Finding minimum and maximum algorithms

Analysis of MaxMin Algorithm

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

$=2[2T(n/4)+2]+2$

$=4T(n/4)+4+2$

$= 4\ [2T(n/8)+2]+4+2$

$= 8T(n/8)+ 8+4+2$ ....assume here k=4 and $n=2^k$

$= 2^{4-1}T(2^4/2^3) + 2^3+2^2+2^1$

$=2^{k-1}T(2) + \sum_{1\le i \le k-1}2^i$

Finding minimum and maximum algorithms

Analysis of MaxMin Algorithm

$T(n) = 2^{k-1}T(2) + \sum_{1 \le i \le k-1} 2^i$

$= 2^{k-1} + 2^k - 2$ ……………………$T(2)=1$

$=(2^k/2) + 2^k - 2$

$=(n/2) + n - 2$ ………………$n = 2^k$

$T(n)=(3n/2)-2$

Time Complexity for MaxMin Algorithm using Divide & Conquer Method is (3n/2)-2

Analysis of Binary search.

- Problem Statement : Binary search can be performed on a sorted array. In this approach, the index of an element **x** is determined if the element belongs to the list of elements. If the array is unsorted, linear search is used to determine the position.
- Solution : In this algorithm, we want to find whether element **x** belongs to a set of numbers stored in an array ***numbers[]***. Where ***l*** and ***r*** represent the left and right index of a sub-array in which searching operation should be performed.
- **Algorithm: Binary-Search(numbers[], x, l, r)**

```
if l = r then
        return l
else
        m := ⌊(l + r) / 2⌋
        if x ≤ numbers[m] then
            return Binary-Search(numbers[], x, l, m)
        else
            return Binary-Search(numbers[], x, m+1, r)
```

Analysis of Binary search.

Analysis

Linear search runs in ***O(n)*** time. Whereas binary search produces the result in ***O(log n)*** time

Let **T(n)** be the number of comparisons in worst-case in an array of **n** elements.

Hence,

$$T(n) = \begin{cases} 0 & if\, n = 1 \\ T(\frac{n}{2}) + 1 & otherwise \end{cases}$$

Using this recurrence relation $T(n) = log\, n$.

Therefore, binary search uses $O(log\, n)$ time.

Analysis of Binary search.

Example

In this example, we are going to search element 63.

First **m** is determined and the element at index **m** is compared to **x**.

5	13	27	30	50	57	63	76
l=0			m=3				r=7

As x > numbers[3], the element may reside in numbers[4...7]. Hence, the first half is discarded and the values of l, m and r are updated as shown below.

~~5~~	~~13~~	~~27~~	~~30~~	50	57	63	76
				L=4	m=5		r = 7

Now the element **x** needs to be searched in numbers[4...7]. As x > numbers[5], new values of l, m and r are updated in a similar way.

~~5~~	~~13~~	~~27~~	~~30~~	~~50~~	~~57~~	63	76
						l=m=6	r = 7

Now, comparing **x** with numbers[6], we get the match. Hence, the position of x = 63 have been determined.