Unit II

DIVIDE AND CONQUER

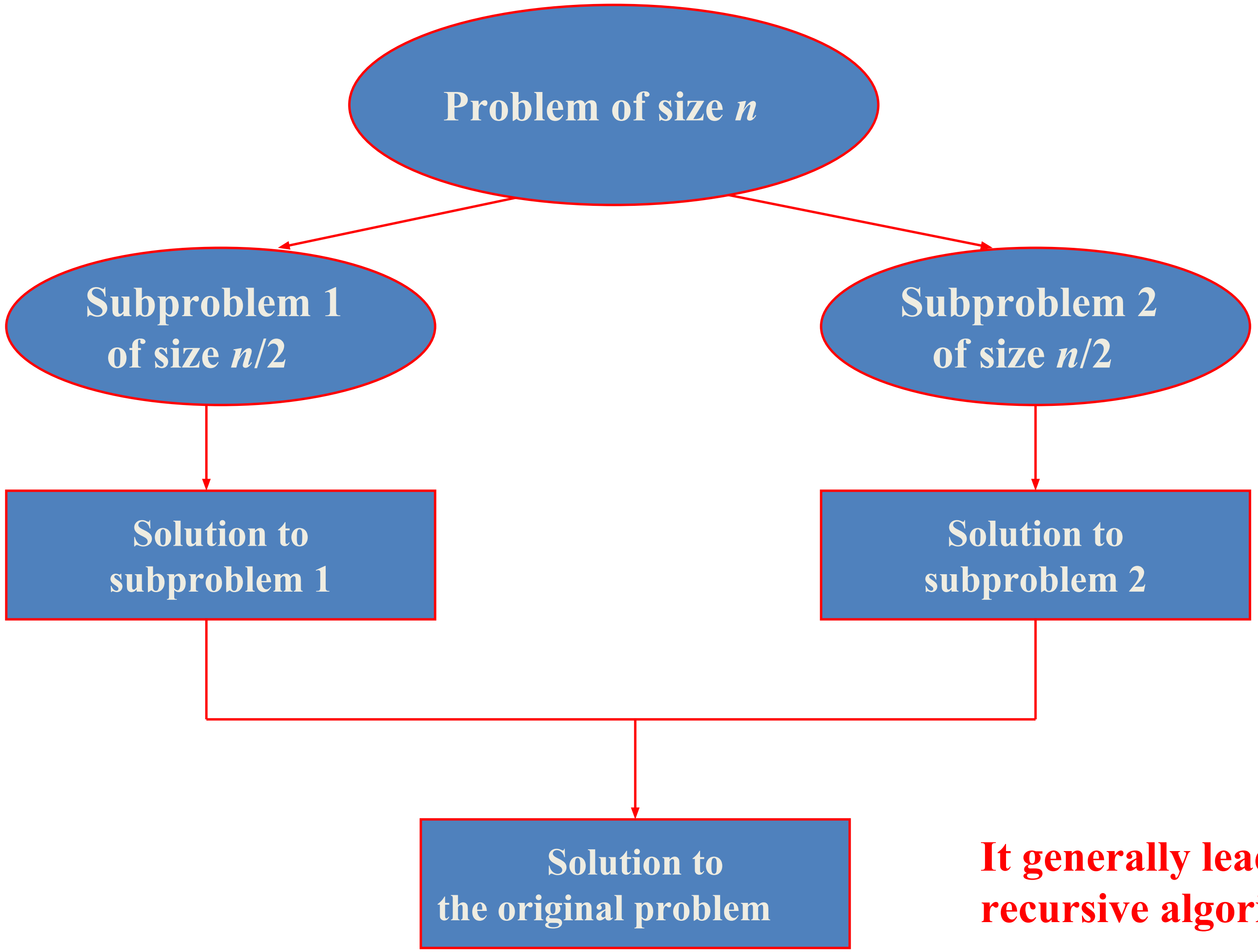
General Method

The most-well known algorithm design strategy or technique:

- **Divide and conquer.**
  - Break up problem into several parts.
  - Solve each part recursively.
  - Combine solutions of sub-problems to obtain solution of the original problem

- **Most common usage.**
  - Break up problem of size n into **two** equal parts of size n/2.
  - Solve two parts recursively.
  - Combine two solutions into overall solution in **linear time**.

General Method(cont.)



It generally leads to a recursive algorithm!

Recurrence Equation for Divide and Conquer

- If the size of problem 'P' is n and the sizes of the 'k' sub problems are n_1, n_2 ....n_k, respectively, then the computing time of divide and conquer is described by the recurrence relation

$$T(n) = \begin{cases} g(n) & n \text{ small} \\ T(n_1) + T(n_2) + \cdots + T(n_k) \ + \ f(n) & \text{otherwise} \end{cases}$$

- Where,
  - T(n) is the time for divide and conquer method on any input of size n and
  - g(n) is the time to compute answer directly for small inputs.
  - $T(n_1)$,$T(n_2)$…$T(n_k)$ is the time taken to solve sub problems of size n_1,n_2…n_k respectively
  - The function f(n) is the time for dividing the problem 'P' and combining the solutions of subproblems

Recurrence Equation for Divide and Conquer [Contd..]

- More generally, an instance of size **n** can be divided into **a** instances of size **n/b**, with **a** of them needing to be solved. (Here, a and b are constants; **a>=1 and b > 1**.)

- Assuming that size **n** is a power of **b** (i.e. $n = b^k$), to simplify our analysis, we get the following recurrence for the running time T(n):

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

- Where f(n) is a function that accounts for the time spent on dividing the problem into smaller ones and on combining their solutions.

Recurrence Equation for Divide and Conquer [Contd..]

- **Substitution Method -** One of the methods for solving the recurrence relation is called the substitution method. This method repeatedly makes substitution for each occurrence of the function T in the right hand side until all such occurrences disappear.
- **Master Theorem** - The efficiency analysis of many divide-and-conquer algorithms is greatly simplified by the master theorem.
  - ❑ It states that, in recurrence equation **T(n) = aT(n/b) + f (n)**,
  - ❑ If $f(n) \in \Theta(n^d)$ where $d \geq 0$ then

$$T(n) \in \begin{cases} \Theta(n^d) & \text{if } a < b^d, \\ \Theta(n^d \log n) & \text{if } a = b^d, \\ \Theta(n^{\log_b a}) & \text{if } a > b^d. \end{cases}$$

Analogous results hold for the *O* and Ω notations, too.

Merge Sort: Design and Implementation

MergeSort

- Merge sort is a perfect example of a successful application of the divide-and conquer technique for sorting purpose.
- It sorts a given array A[1],…..,A[n]
  - ✔ By dividing it into two halves

    A[1],…..,A[n/2] and A[n/2+1],....,A[n],
  - ✔ Sorting each of them recursively,
  - ✔ And then merging the two smaller sorted arrays into a single sorted one.

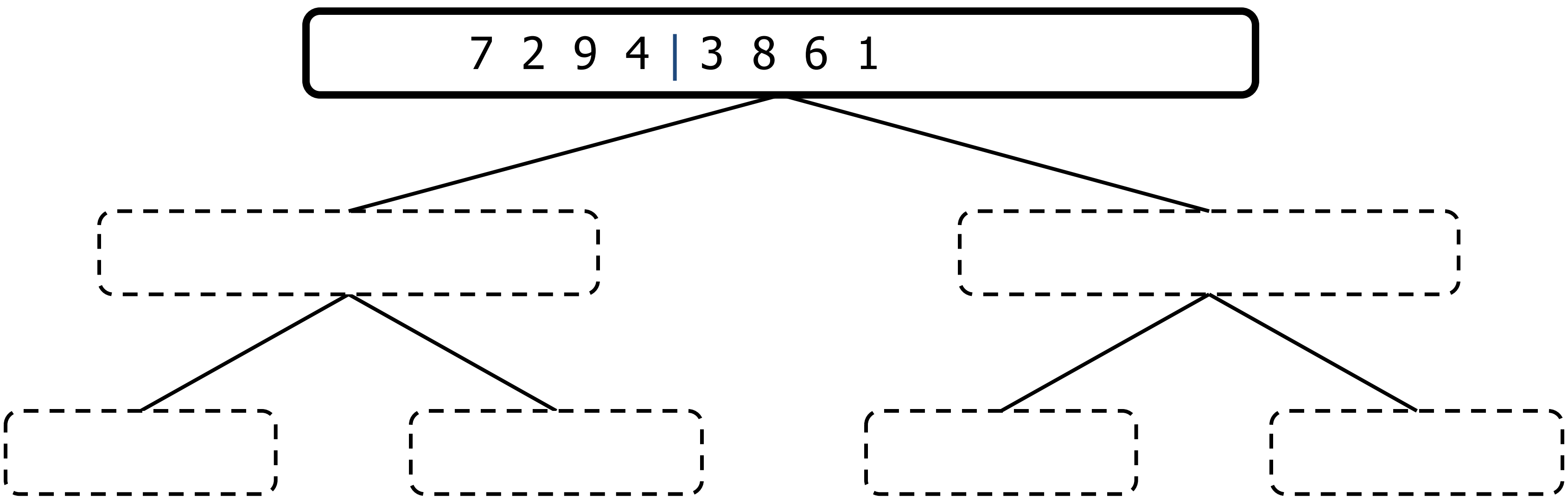
Algorithm for MergeSort

Algorithm Mergesort (L, low, high)

```
// L is list of elements to be sorted, low -> first element, high -> last element
{
    if the list has two elements then
        Compare and sort it
    else
    {
        // Divide L into two sub problems
        mid:=(low+high)/2 ;
        // Solve the sub problems recursively
        Mergesort (L,low, mid); // sorts the first half sub-array L[low:mid]
        Mergesort (L,mid+1,high); //sorts the second half sub-array L[mid+1:high]
        // combine solutions
        Merge (L,low, mid, high);
    }
}
```

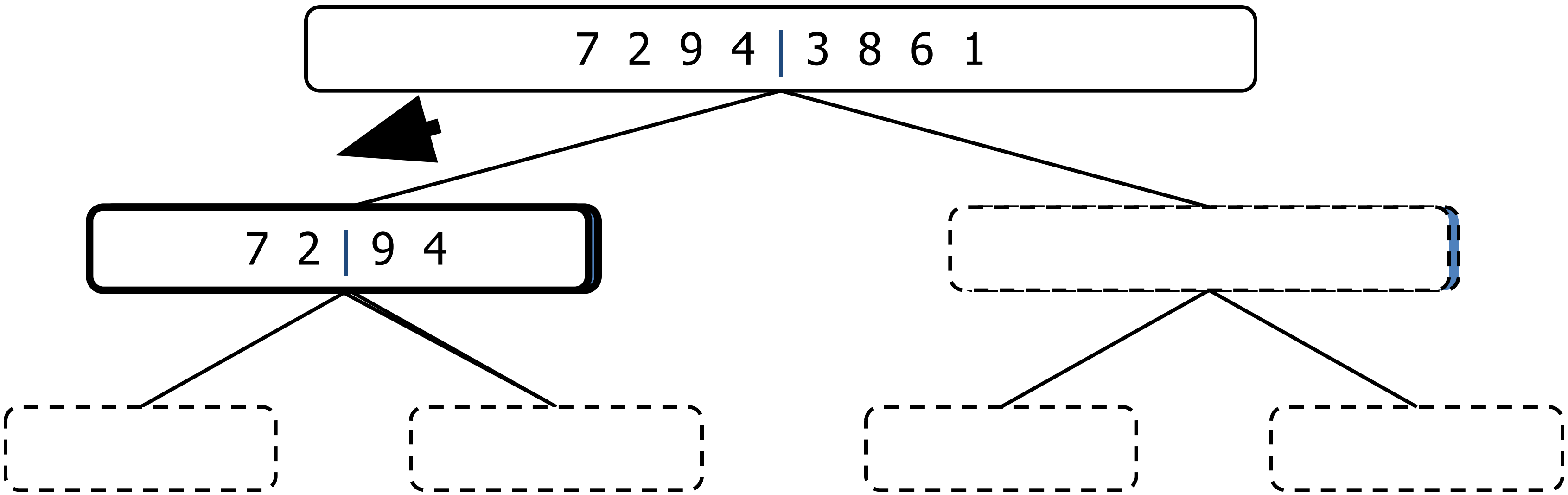
Execution Example

- Initial Call, Partition



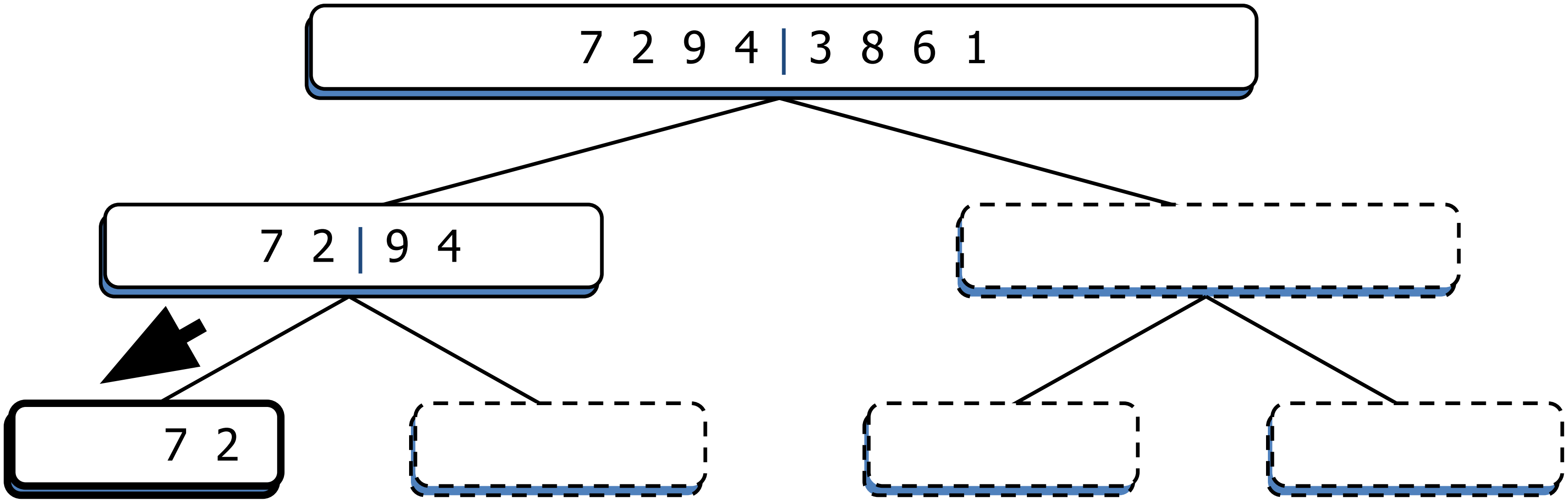
Execution Example (cont.)

- Recursive call, partition



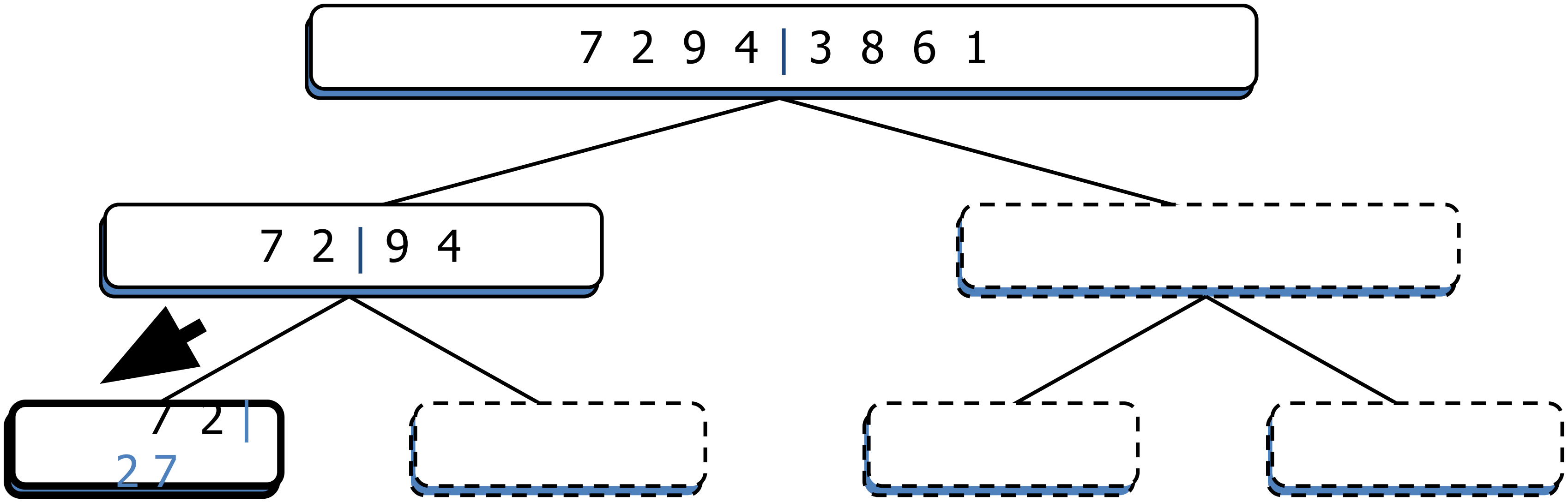
Execution Example (cont.)

- Recursive call, partition



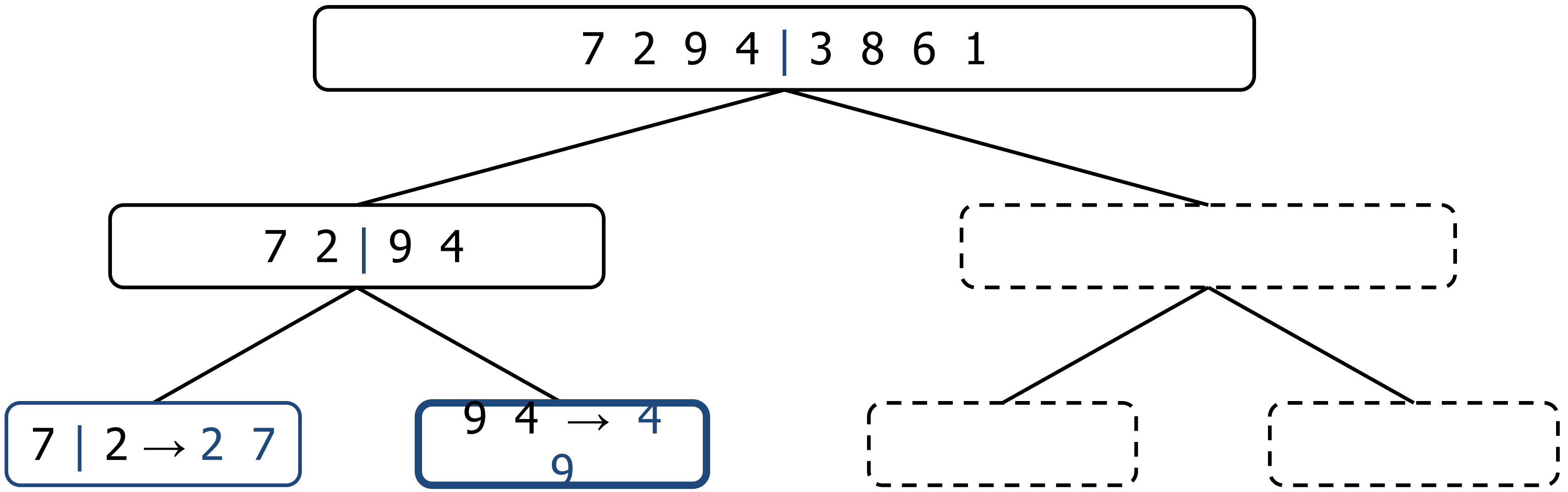
Execution Example (cont.)

- Recursive call, base case



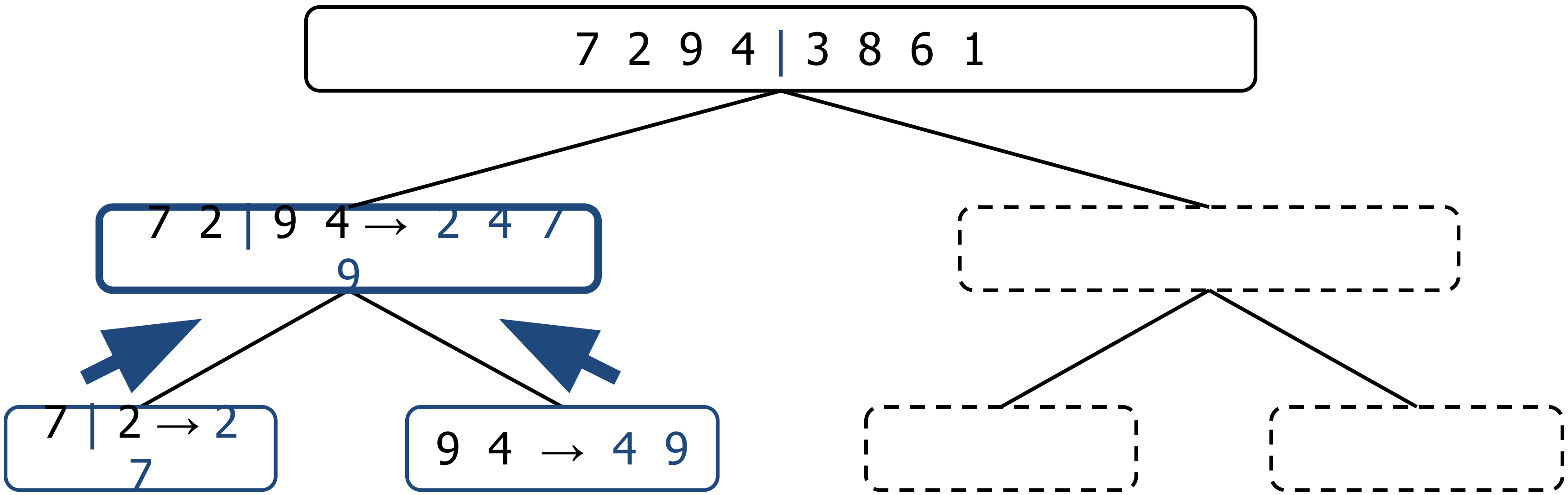
Execution Example (cont.)

- Recursive call, base case



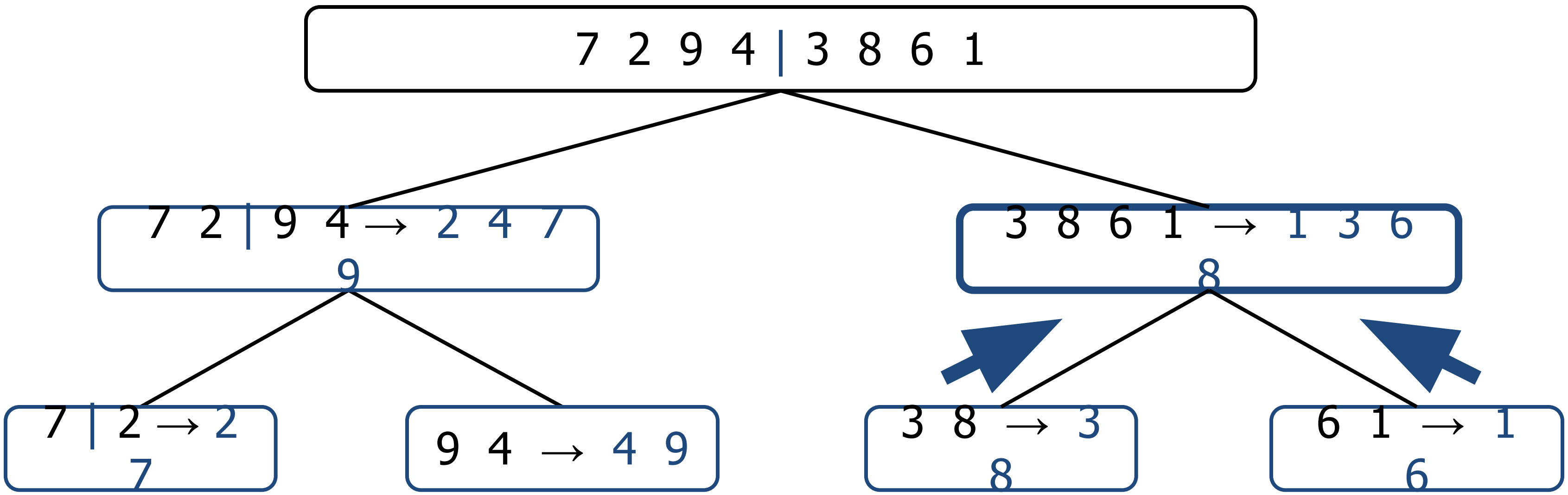
Execution Example (cont.)

- Merge



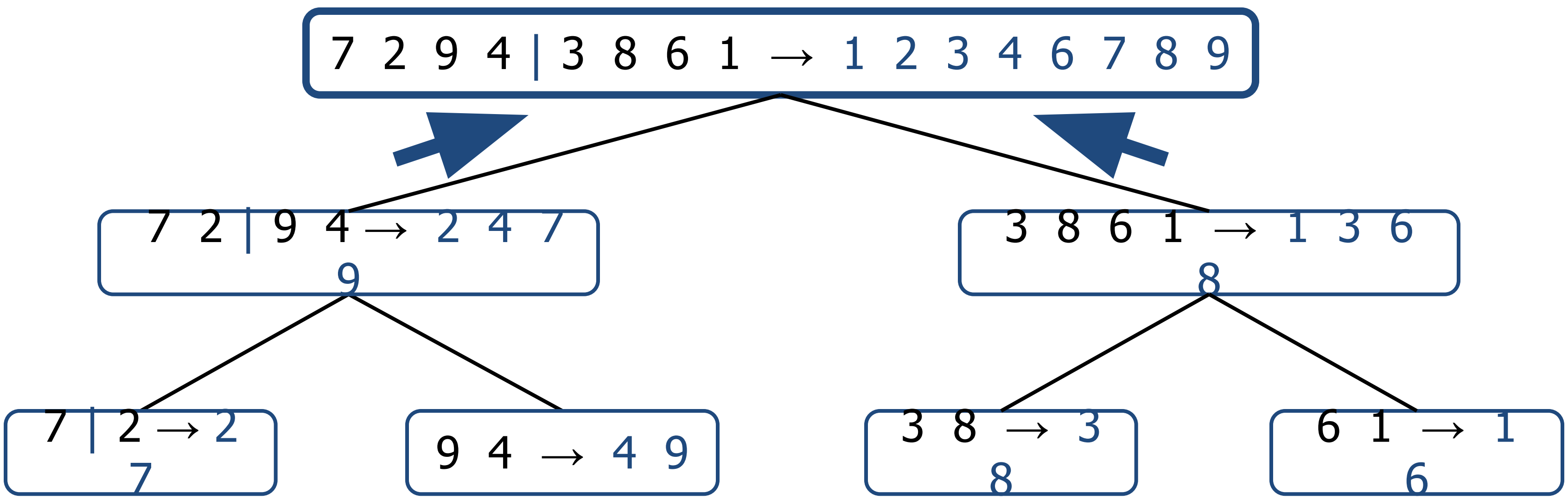
Execution Example (cont.)

- Recursive call, …, merge



Execution Example (cont.)

- Merge



MergeSort: Step Through

- Start with two sorted sets of values

```
a:   3  7  8  19  24  25

b:   2  5  6  10

c:
```

MergeSort: Step Through

- Merge

```
a:    3  7  8  19  24  25

b:    _  5  6  10

c:    2
```

MergeSort: Step Through

- Merge

```
a:    _  7  8  19  24  25

b:    _  5  6  10

c:    2  3
```

MergeSort: Step Through

- Merge

```
a:    _   7   8   19   24   25

b:    _   _   6   10

c:    2   3   5
```

MergeSort: Step Through

- Merge

```
a:    _  7  8  19  24  25

b:    _  _  _  10

c:    2  3  5  6
```

MergeSort: Step Through

- Merge

```
a:   _  _  8  19  24  25

b:   _  _  _  10

c:   2  3  5  6  7
```

MergeSort: Step Through

- Merge

```
a:   _  _  _  19  24  25

b:   _  _  _  10

c:   2  3  5  6  7  8
```

MergeSort: Step Through

- Merge

```
a:    _   _   _   19   24   25
b:    _   _   _    _
c:    2   3   5   6   7   8   10
```

Second sub-array exhausted

MergeSort: Step Through

- Merge

```
a:   _   _   _   _   24   25

b:   _   _   _   _

c:   2   3   5   6   7   8   10   19
```

MergeSort: Step Through

- Merge

```
a:    _   _   _   _   _  25

b:    _   _   _   _

c:    2   3   5   6   7   8   10   19   24
```

MergeSort: Step Through

- Merge

Copied a-values to c

```
a:   _   _   _    _    _    _

b:   _   _   _    _

c:   2   3   5   6   7   8   10   19   24   25
```

MergeSort: Step Through

- Merge

```
a:    _   _   _    _    _    _

b:    _   _   _    _

c:    2   3   5   6   7   8   10   19   24   25
```

Algorithm for Merge

Algorithm Merge (a,low, mid, high)
// a (low: high) is an array containing two sorted subsets in a (low: mid) //and in a(mid+1, high). The goal is to merge these two subsets in to a single set residing in a(low: high)
// b[ ] is a temporary global array.

```
{
    h:=low, i:=low; j=mid+1;
    while ((h≤mid) and (j≤ high)) do
    {
        if (a[h] ≤ a[j]) then
        {
            b[i] = a[h];
            h:=h+1;
        }
        else
        {
            b[i]:=a[j];
            j:=j+1;
        }
        i:=i+1;
    }
```

Algorithm for Merge

```
if (h>mid) then  //  ???
for k=j to high do
{
        b[i]: = a[k];
         i: = i+1;
}
else //  ???
for k:=h to mid do
{
   b[i]:=a[k];
    i:=i+1;
}
for k: = low to high do
       a[k]: = b[k];
}
```

Analysis

- Input Size: **n- elements in the input array**
- Here the basic operation is the **comparisons made**.
- **No best ,worst and average case.**
- Let T(n) denote the number of comparisons made to sort n elements, Assuming n is a even number of elements
- Hence,

$$\mathbf{T(n)=T(n/2)+T(n/2)+C_{merge}(n)\quad for\ n>2}$$

$$\mathbf{T(2)=1\ for\ n=2,\ T(1)=0\quad for\ n=1}$$

Where T(n/2)= Time taken to sort n/2 subproblem
and $C_{merge}(n)$=No of comparisons made to merge 2 sorted Subproblems.

Best Case Occurs When all the elements in a subproblem is smaller than the other subproblem

10	22	33

56	87	92	97

If
- Size of I subproblem is m=3
- Size of II problem is n=4
- No of comparisons made =3 =m

34	45	56	67

10	22	33

If
- Size of I subproblem is m=4
- Size of II problem is n=3
- No of comparisons made =3 =n

Worst Case Occurs When smaller elements may come from the alternating subproblems

34	45	56	67

42	54	65

If
- Size of I subproblem is m=4
- Size of II problem is n=3
- No of comparisons made =6 =m+n-1

In MergeSort, $C_{merge}(n)=n/2+n/2-1=n-1$

Analysis

- Hence,

  $T(n) \le T(n/2) + T(n/2) + C_{merge}(n)$ for $n>2$

  $T(n) \le 2T(n/2) + (n-1)$ for $n>2$

  $T(n) \le 2T(n/2) + O(n)$ for $n>2$

  $T(n) \le 2T(n/2) + cn$ for $n>2$

and

  $T(2) \le c$ for $n=2$

Solving the recurrence equation using **master theorem:**

- Here $a = 2$, $b = 2$, $f(n) = n$, $d = 1$.
- Therefore $2 = 2^1$, case 2 holds in the master theorem
- $\boldsymbol{T}(n) \in \Theta(n^d \log n) = \Theta(n^1 \log n) = \Theta(n \log n)$
- Therefore $\boldsymbol{T(n) \in \Theta(n \log n)}$

Analysis

Two ways of Analyzing Recursion:

- ✔ Unrolling the Recursion method
- ✔ Substituting a solution method

Refer to class notes/ Text book1 page no-212 and 213

Comparison between Sorting Algorithms

	Worst Case	Average Case	Best Case
Bubble Sort	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n)$
Insertion Sort	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n)$
Merge Sort	$\mathcal{O}(n \log n)$	$\mathcal{O}(n \log n)$	$\mathcal{O}(n \log n)$
Heap Sort	$\mathcal{O}(n \log n)$	$\mathcal{O}(n \log n)$	$\mathcal{O}(n \log n)$
Quick Sort	$\mathcal{O}(n^2)$	$\mathcal{O}(n \log n)$	$\mathcal{O}(n \log n)$

QuickSort - Introduction

- Quicksort is the other important sorting algorithm that is based on the divide and conquer approach.
- Unlike mergesort, which divides its input elements according to their position in the array, quicksort divides them according to their value.
- A partition is an arrangement of the array's elements so that all the elements to the left of some element A[s] are less than or equal to A[s], and all the elements to the right of A[s] are greater than or equal to it

$$\underbrace{A[0] \ldots A[s-1]}_{\text{all are } \leq A[s]} \quad A[s] \quad \underbrace{A[s+1] \ldots A[n-1]}_{\text{all are } \geq A[s]}$$

QuickSort- Introduction

$$\underbrace{A[0] \dots A[s-1]}_{\text{all are } \leq A[s]} \quad A[s] \quad \underbrace{A[s+1] \dots A[n-1]}_{\text{all are } \geq A[s]}$$

- Obviously, after a partition is achieved, A[s] will be in its final position in the sorted array, and we can continue sorting the two subarrays to the left and to the right of A[s] independently (e.g., by the same method).
- Note the difference with mergesort: there, the division of the problem into two subproblems is immediate and the entire work happens in combining their solutions; here, the entire work happens in the division stage, with no work required to combine the solutions to the subproblems.

Algorithm for QuickSort

ALGORITHM *Quicksort*$(A[l..r])$

//Sorts a subarray by quicksort
//Input: Subarray of array $A[0..n-1]$, 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
    $Quicksort(A[l..s-1])$
    $Quicksort(A[s+1..r])$

Pick Pivot Element

There are a number of ways to pick the pivot element. In this example, we will use the first element in the array:

40	20	10	80	60	50	7	30	100

Partitioning Array

Given a pivot, partition the elements of the array such that the resulting array consists of:

1. One sub-array that contains elements >= pivot
2. Another sub-array that contains elements <= pivot

The sub-arrays are stored in the original data array.

Pivot = 40

40	20	10	80	60	50	7	30	100	∞
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
i									**j**

1. Repeat i++ until a[i]>=pivot

Pivot=40

40	20	10	80	60	50	7	30	100	∞
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]

i

j

1. Repeat i++ until a[i]>=pivot

Pivot=40

40	20	10	80	60	50	7	30	100	∞
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]

i

j

1. Repeat i++ until a[i]>=pivot

Pivot=40

40	20	10	80	60	50	7	30	100	∞
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]

i

j

1. Repeat i++ until a[i]>=pivot
2. Repeat j-- until a[j]<=pivot

Pivot=40

40	20	10	80	60	50	7	30	100	∞
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]

i **j**

1. Repeat i++ until a[i]>=pivot
2. Repeat j-- until a[j]<=pivot

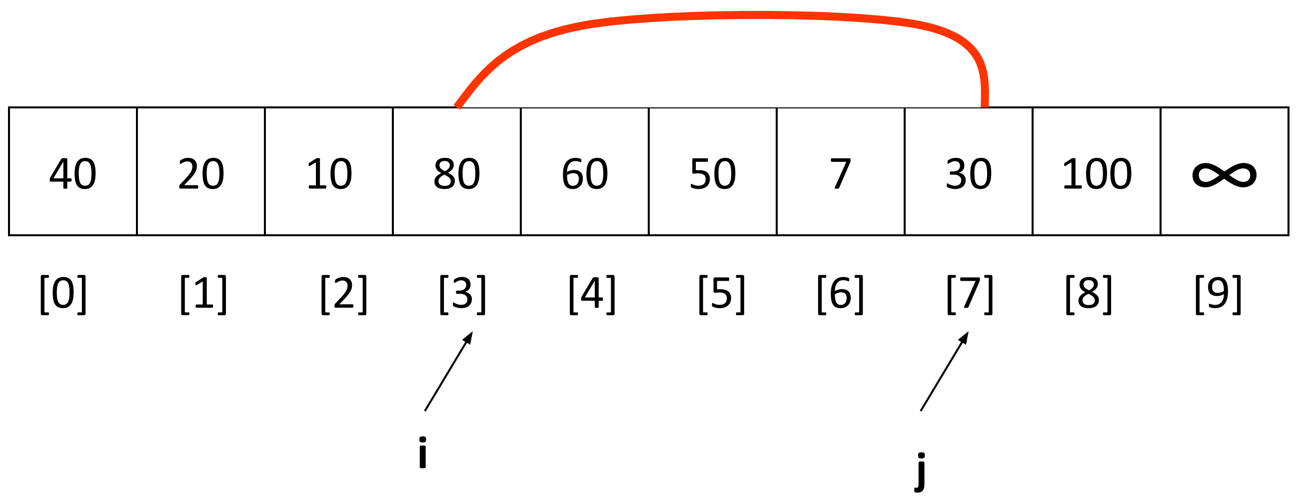
pivot_index = 0

40	20	10	80	60	50	7	30	100	∞
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
			i				**j**		

1. Repeat i++ until a[i]>=pivot
2. Repeat j-- until a[j]<=pivot
3. If i<j

   swap a[i] and a[j]

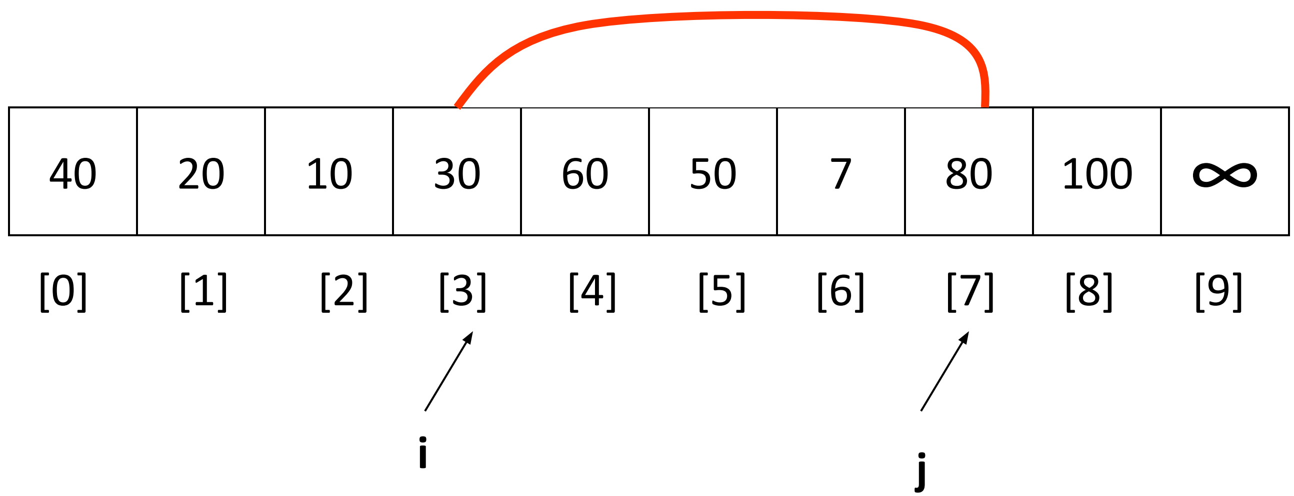
pivot_index = 0



1. Repeat i++ until a[i]>=pivot
2. Repeat j-- until a[j]<=pivot
3. If i<j

   swap a[i] and a[j]

Pivot=40



1. Repeat i++ until a[i]>=pivot
2. Repeat j-- until a[j]<=pivot
3. If i<j

    swap a[i] and a[j]
4. if i <j, go to 1.

Pivot=40

40	20	10	30	60	50	7	80	100	∞
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
			i				j		

1. Repeat i++ until a[i]>=pivot
2. Repeat j-- until a[j]<=pivot
3. If i<j

   swap a[i] and a[j]
4. if i <j, go to 1.

Pivot=40

40	20	10	30	60	50	7	80	100	∞
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
			i				j		

1. Repeat i++ until a[i]>=pivot
2. Repeat j-- until a[j]<=pivot
3. If i<j

   swap a[i] and a[j]
4. if i <j, go to 1.

Pivot=40

40	20	10	30	60	50	7	80	100	∞
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
				i			**j**		

1. Repeat i++ until a[i]>=pivot
2. Repeat j-- until a[j]<=pivot
3. If i<j

   swap a[i] and a[j]
4. if i <j, go to 1.

Pivot=40

	40	20	10	30	60	50	7	80	100	∞
	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
					i			j		

1. Repeat i++ until a[i]>=pivot
2. Repeat j-- until a[j]<=pivot
3. If i<j

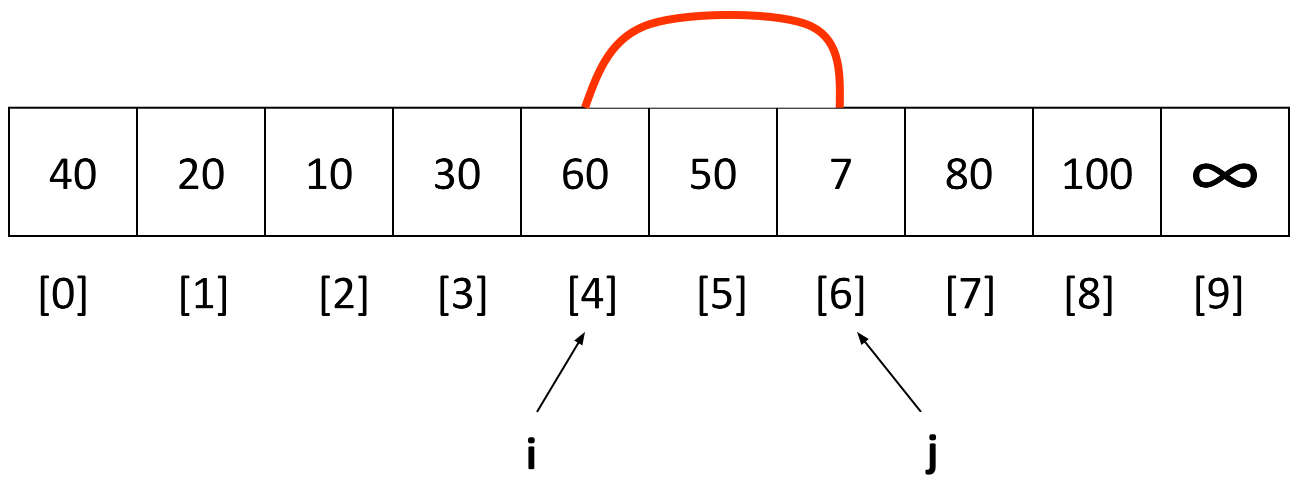
   swap a[i] and a[j]
4. if i <j, go to 1.

Pivot=40

40	20	10	30	60	50	7	80	100	∞
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
				i		**j**			

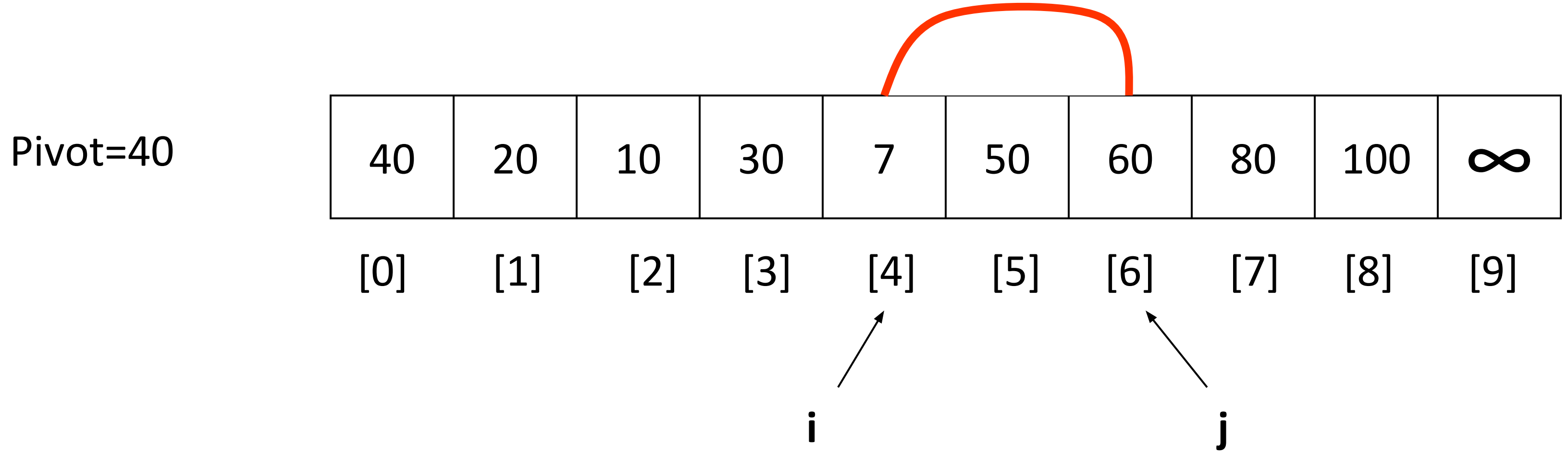
1. Repeat i++ until a[i]>=pivot
2. Repeat j-- until a[j]<=pivot
3. If i<j
   swap a[i] and a[j]
4. if i <j, go to 1.

Pivot=40



1. Repeat i++ until a[i]>=pivot
2. Repeat j-- until a[j]<=pivot
3. If i<j

    swap a[i] and a[j]
4. if i <j, go to 1.



1. Repeat i++ until a[i]>=pivot
2. Repeat j-- until a[j]<=pivot
3. If i<j

   swap a[i] and a[j]

4. if i <j, go to 1.

Pivot=40

40	20	10	30	7	50	60	80	100	∞
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
				i		j			

1. Repeat i++ until a[i]>=pivot
2. Repeat j-- until a[j]<=pivot
3. If i<j

   swap a[i] and a[j]
4. if i <j, go to 1.

Pivot=40

40	20	10	30	7	50	60	80	100	∞
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
				i		j			

1. Repeat i++ until a[i]>=pivot
2. Repeat j-- until a[j]<=pivot
3. If i<j

   swap a[i] and a[j]
4. if i <j, go to 1.

Pivot=40

40	20	10	30	7	50	60	80	100	∞
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
					i	j			

1. Repeat i++ until a[i]>=pivot
2. Repeat j-- until a[j]<=pivot
3. If i<j

   swap a[i] and a[j]
4. if i <j, go to 1.

Pivot=40

40	20	10	30	7	50	60	80	100	∞
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]

i

j

1. Repeat i++ until a[i]>=pivot
2. Repeat j-- until a[j]<=pivot
3. If i<j

   swap a[i] and a[j]
4. if i <j, go to 1.

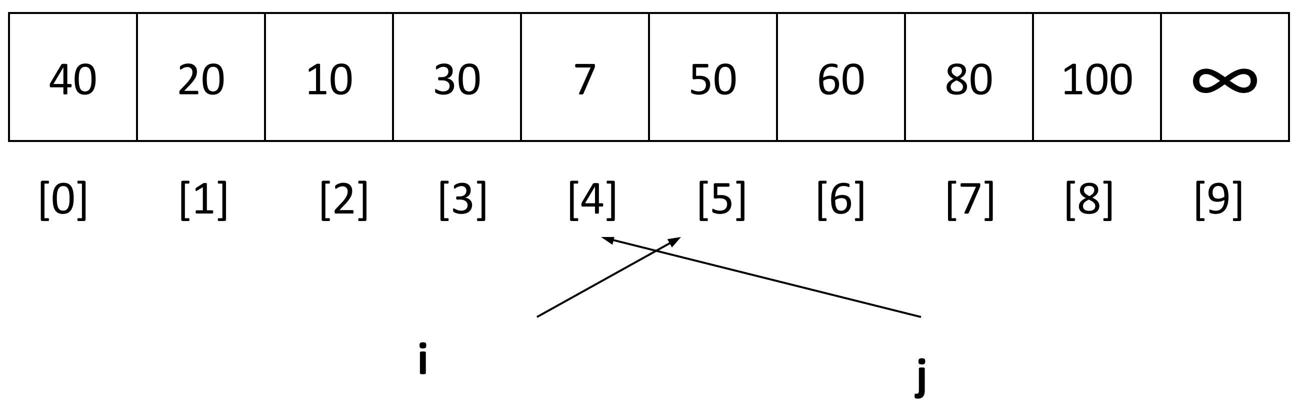
Pivot=40

40	20	10	30	7	50	60	80	100	∞
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]

i **j**

1. Repeat i++ until a[i]>=pivot
2. Repeat j-- until a[j]<=pivot
3. If i<j
   swap a[i] and a[j]
4. if i <j, go to 1.

Pivot=40



1. Repeat i++ until a[i]>=pivot
2. Repeat j-- until a[j]<=pivot
3. If i<j

   swap a[i] and a[j]
4. if i <j, go to 1.

Pivot=40

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
40	20	10	30	7	50	60	80	100	∞

i **j**

1. Repeat i++ until a[i]>=pivot
2. Repeat j-- until a[j]<=pivot
3. If i<j

   swap a[i] and a[j]
4. if i <j, go to 1.

Pivot=40

40	20	10	30	7	50	60	80	100	∞
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]

i **j**

1. Repeat i++ until a[i]>=pivot
2. Repeat j-- until a[j]<=pivot
3. If i<j

   swap a[i] and a[j]
4. if i <j, go to 1.
5. If i>=j then swap a[low] and a[j]

Pivot=40

40	20	10	30	7	50	60	80	100	∞
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]

i **j**

1. Repeat i++ until a[i]>=pivot
2. Repeat j-- until a[j]<=pivot
3. If i<j

   swap a[i] and a[j]
4. if i <j, go to 1.
5. If i>=j then swap a[low] and a[j]

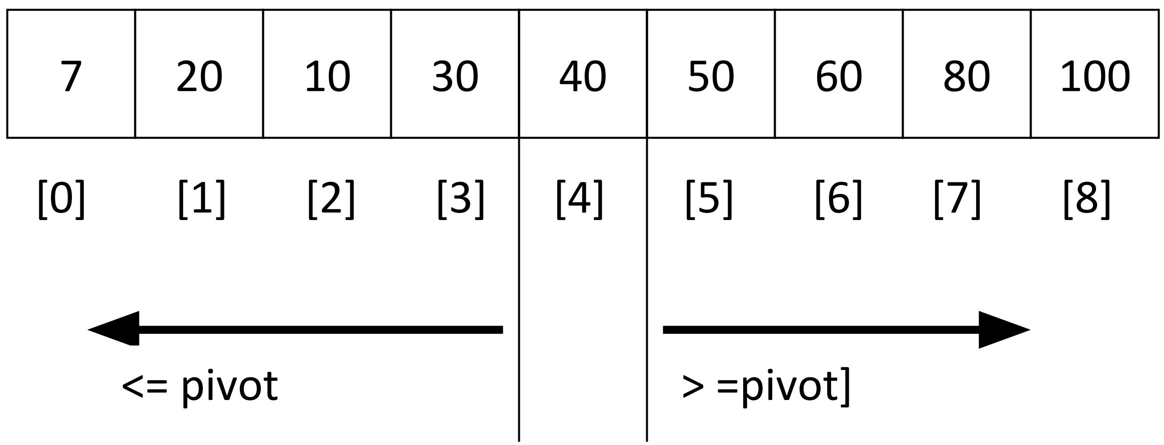
Pivot=40
and returns 4

7	20	10	30	40	50	60	80	100	∞
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]

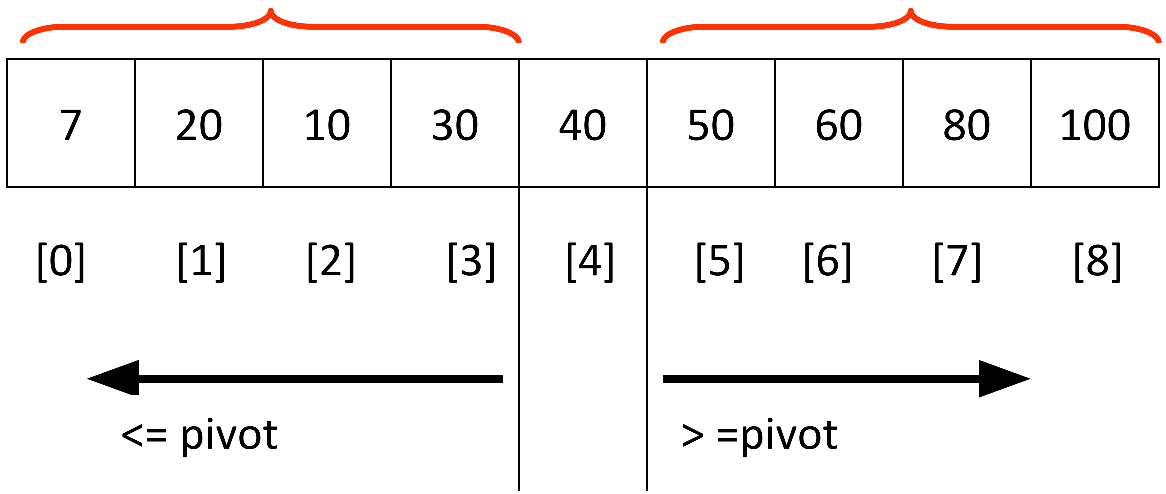
i

j

Partition Result



Recursion: Quicksort Sub-arrays



Algorithm for Partition

Alg. PARTITION (A[*l*…r])

```
pivot ← A[l]
 i ← l
j ← r + 1
repeat
        repeat
          i ← i + 1
        until A[i] ≥ pivot

    repeat
      j ← j – 1
              until A[j] ≤ pivot
    if i < j
            then exchange A[i] ↔ A[j]
until i>=j
 exchange A/l] ↔ A[j]
return j
```

Quicksort Analysis

- Assume that keys are random, uniformly distributed.
- What is best case running time?

Quicksort Analysis

- Assume that keys are random, uniformly distributed.
- What is best case running time?
  - Recursion:
    1. Partition splits array in two sub-arrays of size n/2
    2. Quicksort applied recursively to each sub-array
- The recurrence relation to compute time complexity

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

Quicksort Analysis

- Assume that keys are random, uniformly distributed.
- Best case running time: $\Omega(n \log_2 n)$

Quicksort Analysis

- Assume that keys are random, uniformly distributed.
- Best case running time: $\Omega(n \log_2 n)$
- Worst case running time?

Quicksort: Worst Case

- Assume first element is chosen as pivot.
- Assume we get array that is already in order:

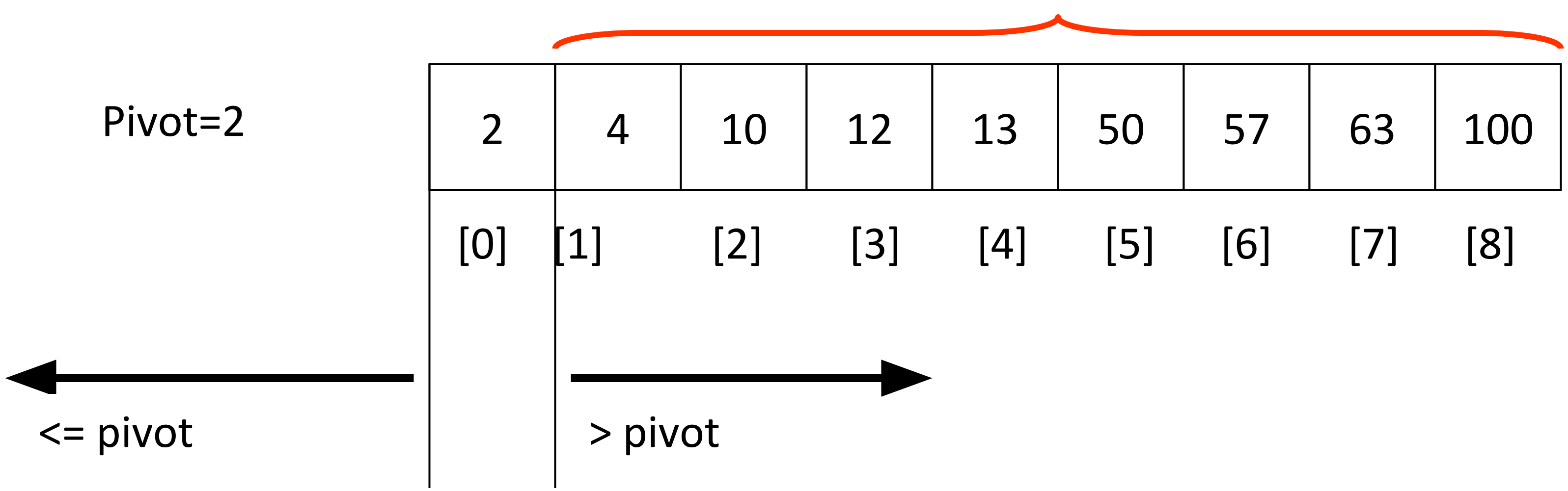
Pivot= 2

2	4	10	12	13	50	57	63	100
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]

i **j**

1. Repeat i++ until a[i]>=pivot
2. Repeat j-- until a[j]<=pivot
3. If i<j

    swap a[i] and a[j]
4. if i <j, go to 1.
5. If i>=j then swap a[low] and a[j]



Quicksort Analysis

- Assume that keys are random, uniformly distributed.
- Best case running time: $\Omega(n \log_2 n)$
- Worst case running time?
  - Recursion:
    1. Partition splits array in two sub-arrays:
       - one sub-array of size 0
       - the other sub-array of size n-1
    2. Quicksort each sub-array.
- The recurrence relation to compute time complexity $T(n) = T(n-1) + T(0) + cn$

Quicksort Analysis

- The recurrence relation to compute time complexity

$$T(n) = T(n-1)+T(0)+n$$

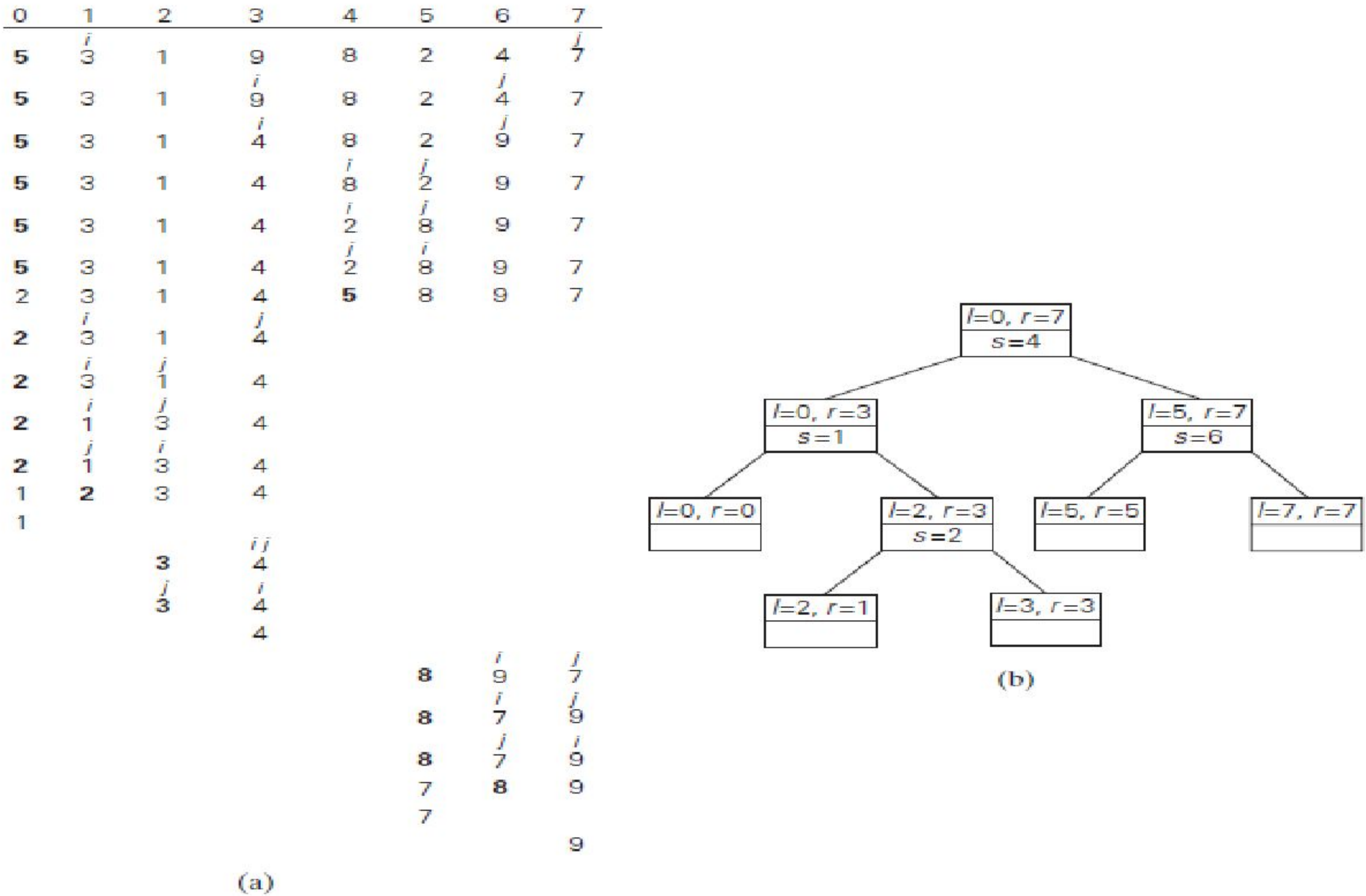
$$T(n) = T(n-1)+n$$

Solving the recurrence relation using backward substitution method will result to $T(n)=O(n^2)$

Quicksort Analysis

- Assume that keys are random, uniformly distributed.
- Best case running time: $\Omega(n \log_2 n)$
- Worst case running time: $O(n^2)$

Example of quicksort operation. (a) Array‘s transformations with pivots shown in bold. (b) Tree of recursive calls to Quicksort with input values l and r of subarray bounds and split position s of a partition obtained



0	1	2	3	4	5	6	7
	i						j
5	3	1	9	8	2	4	7
			i			j	
5	3	1	9	8	2	4	7
			i			j	
5	3	1	4	8	2	9	7
				i	j		
5	3	1	4	8	2	9	7
				i	j		
5	3	1	4	2	8	9	7
				j	i		
5	3	1	4	2	8	9	7
2	3	1	4	5	8	9	7

	i		j
2	3	1	4
	i	j	
2	3	1	4
	i	j	
2	1	3	4
	j	i	
2	1	3	4
1	2	3	4

1

	i	j
8	9	7
	i	j
8	7	9
	j	i
8	7	9
7	8	9

7		9

	i,j
3	4
j	i
3	4
3	4