Module 2:
Divide and Conquer

Dr. Natarajan Meghanathan
Professor of Computer Science
Jackson State University
Jackson, MS 39217
E-mail: natarajan.meghanathan@jsums.edu

Introduction to Divide and Conquer

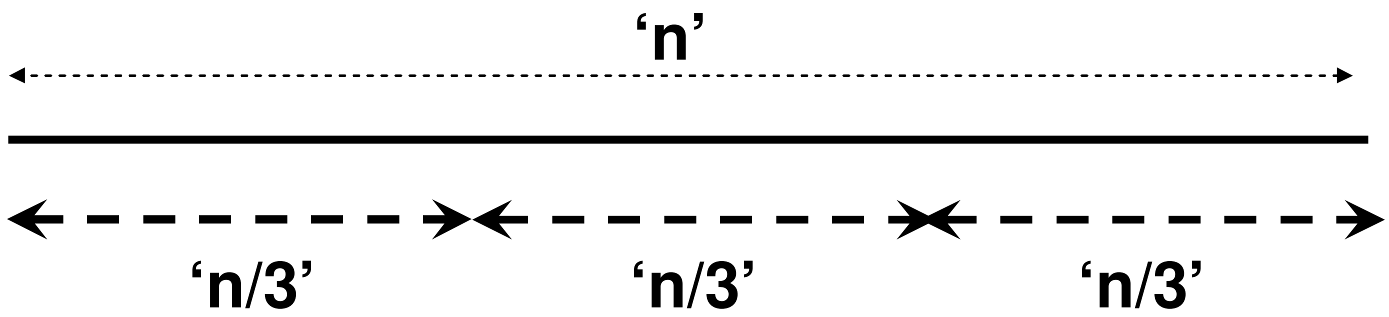
- Divide and Conquer is an algorithm design strategy of dividing a problem into sub problems, solving the sub problems and merging the solutions of the sub problems to get a solution for the larger problem.

- Let a problem space of size ‘n’ (for example: an n-element array used for sorting) be divided into sub problems of size ‘n/b’ each, which could be either overlapping or non-overlapping.
- Let us say we solve ‘a’ of these sub problems of size n/b.
- Let f(n) represent the time complexity of merging the solutions of the sub problems to get a solution for the larger problem.
- The general format of the recurrence relation can be then written as follows: where T(n/b) is the time complexity to solve a sub problem of size n/b and T(n) is the overall time complexity to solve a problem of size n.

$$T(n) = a * T(n/b) + f(n)$$

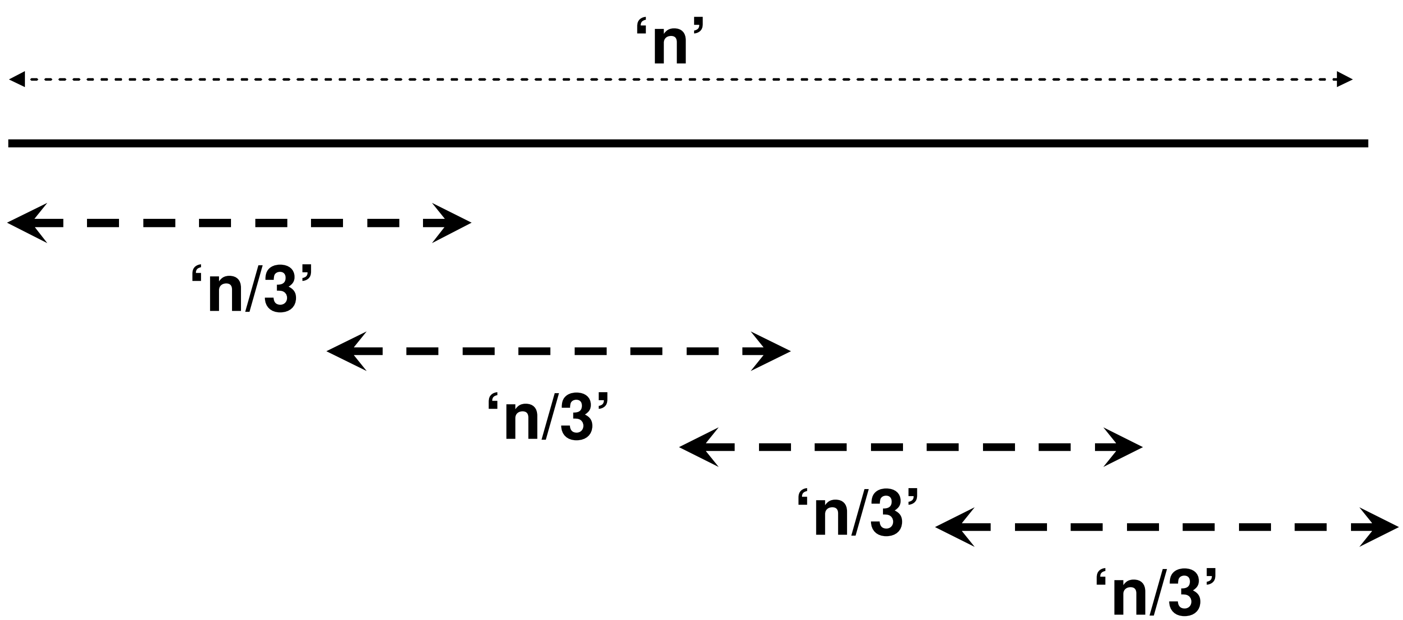
Recurrence Relations for Divide and Conquer

Non-Overlapping Sub Problems



T(n) = 3 * T (n/3) + f(n)

Overlapping Sub Problems (a ≠ b)



T(n) = 4 * T (n/3) + f(n)

Polynomial Function

- A polynomial is an expression consisting of variables and coefficients, that involves only the operations of addition, subtraction, multiplication, and non-negative integer exponents of variables.
- Example: $f(n) = n^3 + 4n^2 – 2n + 1$ is a polynomial (of degree 3). But $f(n) = n^{-3} + 1$ is not a polynomial (because of the negative exponent).

- A monotonically increasing polynomial function is a polynomial function (say, of an independent variable n) whose value either increases or remains the same with increase in n.
  – That is, the function should be a non-decreasing function.

Master Theorem to Solve Recurrence Relations: T(n) = a * T(n/b) + f(n)

Master Theorem (Θ - version)

If $f(n) \in \Theta(n^d)$ where $d \geq 0$, then

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

Note: To satisfy the definition of a polynomial, 'd' should be a non-negative integer.

Note: To apply Master Theorem, the function **f(n)** should be a **polynomial and should be monotonically increasing**

If $f(n) \notin \Theta(n^d); but$ $f(n) \in O(n^d)$, then

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

Master Theorem (O - version)

where d ≥ 0 and an integer

Note: We will try to apply the **Θ – version** wherever possible. If the **Θ – version** cannot be applied, we will try to apply the **O-version**.

1) $T(n) = 4T(n/2) + n$
can be written as
$T(n) = 4\,T(n/2) + \Theta(n)$
$a = 4$; $b = 2$; $d = 1$ ➔ $a > b^d$

$$T(n) = \Theta\left(n^{\log_2 4}\right) = \Theta(n^2)$$

2) $T(n) = 4T(n/2) + n^2$
Can be written as
$T(n) = 4\,T(n/2) + \Theta(n^2)$
$a = 4$; $b = 2$; $d = 2$ ➔ $a = b^d$

$$T(n) = \Theta\left(n^2 \log n\right)$$

3) $T(n) = 4T(n/2) + n^3$
Can be written as
$T(n) = 4\,T(n/2) + \Theta(n^3)$
$a = 4$; $b = 2$; $d = 3$ ➔ $a < b^d$

$$T(n) = \Theta\left(n^3\right)$$

4) $T(n) = 4T(n/2) + 1$
Can be written as
$T(n) = 4\,T(n/2) + \Theta(n^0)$
$a = 4$; $b = 2$; $d = 0$ ➔ $a > b^d$

$$T(n) = \Theta\left(n^{\log_2 4}\right) = \Theta(n^2)$$

5) $T(n) = 4T(n/2) + (1/n)$
$T(n) = 4T(n/2) + n^{-1}$
$a = 4$, $b = 2$, $d = -1$ (< 0)
$f(n) = 1/n$ is not a polynomial.
Master Theorem cannot be applied.

Master Theorem: More Problems

$T(n) = 3\ T(n/3) + \sqrt{n}$

We cannot write $\sqrt{n} = \Theta(n^d)$,
because d = 1/2 is not an integer.
Hence, we have to use the O-notation.

$\sqrt{n} = O(n)$, the smallest possible integer for which $\sqrt{n}$ can be written as $O(n^d)$.

$T(n) = 3\ T(n/3) + O(n)$
a = 3, b = 3, d = 1
$a = b^d$.
Hence, $T(n) = O(n^d \log n) = O(n \log n)$.

Master Theorem: More Problems

$T(n) = 4\,T(n/2) + \log n$

$\log n \notin \Theta(n^d)$, where 'd' is an integer

But, $\log n \in O(n^d)$, where $d = 1$ is the smallest possible integer for which logn can be written as $O(n^d)$

$a = 4;\ b = 2;\ d = 1$

$a > b^d$; Hence, $T(n) = O\left(n^{\log_b^a}\right)$

$T(n) = O\left(n^{\log_2^4}\right) = O(n^2)$

$T(n) = 6\,T(n/3) + n^2 \log n$

$n^2 \log n \notin \Theta(n^d)$, where 'd' is an integer

But, $n^2 \log n \in O(n^d)$, where $d = 3$ is the smallest possible integer for which logn can be written as $O(n^d)$

$a = 6;\ b = 3;\ d = 3$

$a < b^d$; Hence, $T(n) = O(n^3)$

Merge Sort

- Split array A[0..n-1] in two about equal halves and make copies of each half in arrays B and C
- Sort arrays B and C recursively
- Merge sorted arrays B and C into array A as follows:
  - Repeat the following until no elements remain in one of the arrays:
    - compare the first elements in the remaining unprocessed portions of the arrays
    - copy the smaller of the two into A, while incrementing the index indicating the unprocessed portion of that array
  - Once all elements in one of the arrays are processed, copy the remaining unprocessed elements from the other array into A.

Merge Sort

ALGORITHM *Mergesort*$(A[0..n-1])$

//Sorts array $A[0..n-1]$ by recursive mergesort
//Input: An array $A[0..n-1]$ of orderable elements
//Output: Array $A[0..n-1]$ sorted in nondecreasing order
if $n > 1$
    copy $A[0..\lfloor n/2 \rfloor - 1]$ to $B[0..\lfloor n/2 \rfloor - 1]$
    copy $A[\lfloor n/2 \rfloor..n-1]$ to $C[0..\lceil n/2 \rceil - 1]$
    Mergesort$(B[0..\lfloor n/2 \rfloor - 1])$
    Mergesort$(C[0..\lceil n/2 \rceil - 1])$
    Merge(B, C, A)

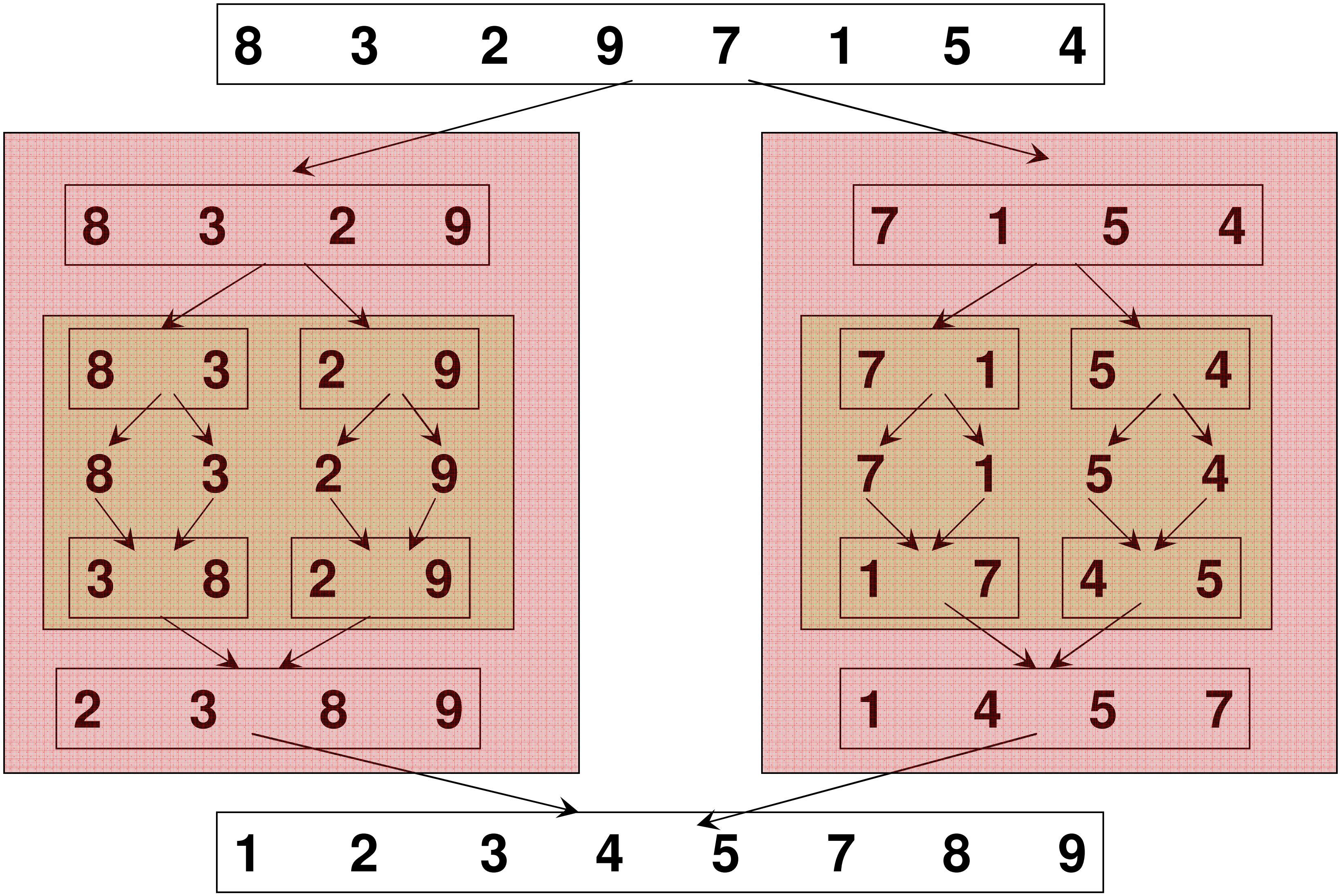
Merge Algorithm

ALGORITHM *Merge*$(B[0..p-1], C[0..q-1], A[0..p+q-1])$

//Merges two sorted arrays into one sorted array
//Input: Arrays $B[0..p-1]$ and $C[0..q-1]$ both sorted
//Output: Sorted array $A[0..p+q-1]$ of the elements of B and C
$i \leftarrow 0;\ \ j \leftarrow 0;\ \ k \leftarrow 0$
while $i < p$ **and** $j < q$ **do**
    if $B[i] \leq C[j]$
        $A[k] \leftarrow B[i];\ \ i \leftarrow i + 1$
    else $A[k] \leftarrow C[j];\ \ j \leftarrow j + 1$
    $k \leftarrow k + 1$
if $i = p$
    copy $C[j..q-1]$ to $A[k..p+q-1]$
else copy $B[i..p-1]$ to $A[k..p+q-1]$

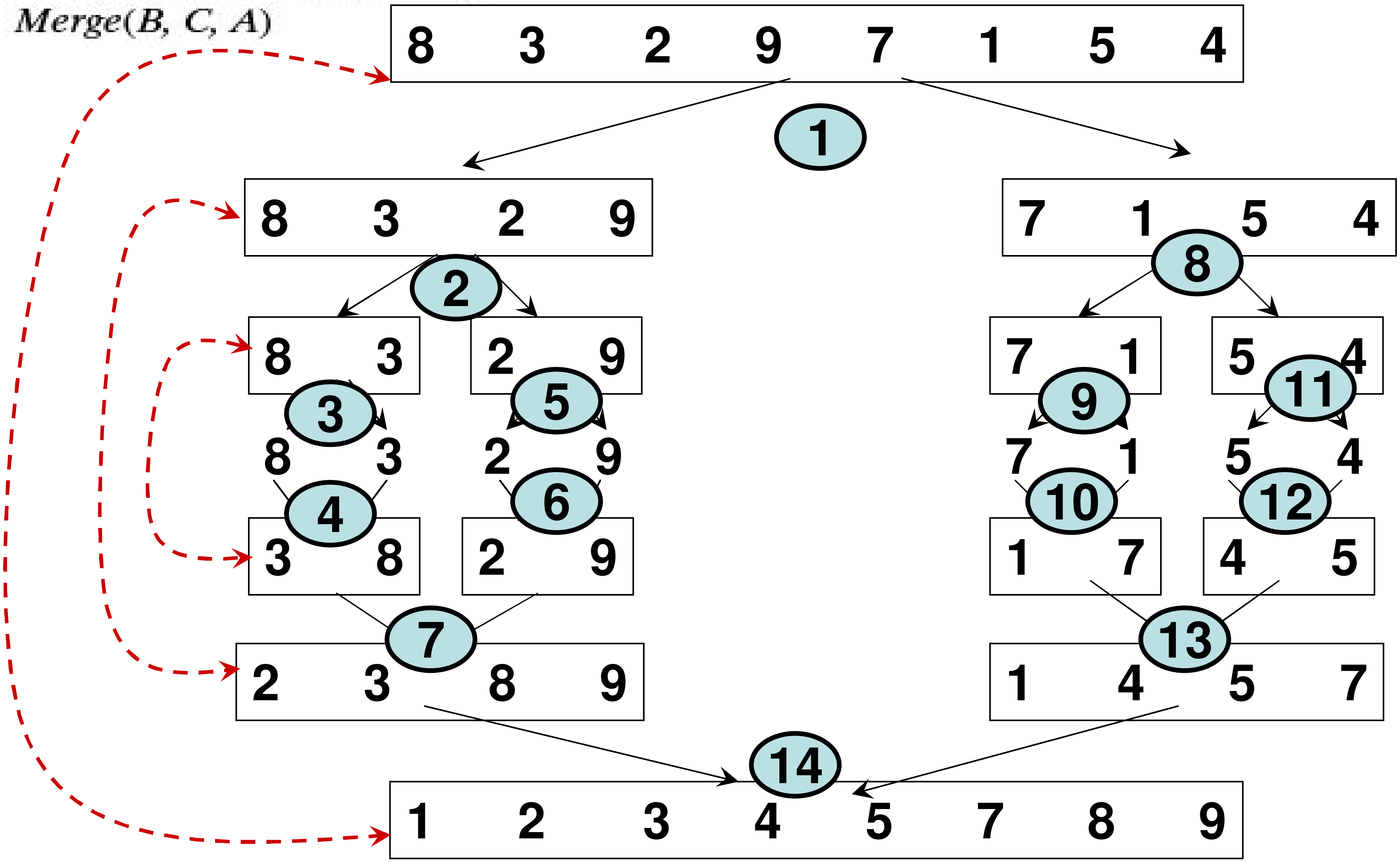
Incase of a tie B[i] = C[j]
Insert the element in the
Left sub array in A.

Example for Merge Sort



if $n > 1$
    copy $A[0..\lfloor n/2 \rfloor - 1]$ to $B[0..\lfloor n/2 \rfloor - 1]$
    copy $A[\lfloor n/2 \rfloor..n - 1]$ to $C[0..\lceil n/2 \rceil - 1]$
    Mergesort$(B[0..\lfloor n/2 \rfloor - 1])$
    Mergesort$(C[0..\lceil n/2 \rceil - 1])$
    Merge(B, C, A)

The order recursion runs



Analysis of Merge Sort

The recurrence relation for the number of key comparisons C(n) is

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

At each step, exactly one comparison is made, after which the total number of elements in the two arrays still needed to be processed is reduced by one.

<u>Best case:</u> We will encounter n/2 comparisons (when every element in the left sorted sub array is less than or equal to the first element in right sorted sub array)

<u>Worst case:</u> We will encounter (n-1) comparisons (when smaller elements come from the alternating sub arrays; neither of the two sub arrays will become empty before the other sub array contains just one element.

Though best case is different from worst case, both are ~ n, as n increases. Hence, the time complexity to merge: $C_{merge}(n) = \Theta(n)$

$C(n) = 2*C(n/2) + \Theta(n)$ for $n > 1$ and $C(1) = 0$

a = 2; b = 2; d = 1

$a = b^d$

Hence, $C(n) = \Theta(n\log n)$

Merge Sort: Space-time Tradeoff

- Unlike the sorting algorithms (insertion sort, bubble sort, selection sort) we saw in Module 1, Merge sort incurs a time-complexity of $\Theta(n\log n)$, whereas the other sorting algorithms we have seen incur a time complexity of $O(n^2)$ or $\Theta(n^2)$.

- The tradeoff is Merge sort requires additional space proportional to the size of the array being sorted. That is, the space-complexity of merge sort is $\Theta(n)$, whereas the other sorting algorithms we have seen incur a space-complexity of $\Theta(1)$.
  - Algorithms that incur a $\Theta(1)$ space complexity are said to be "**<u>in place</u>**"

Number of Inversions in an Array

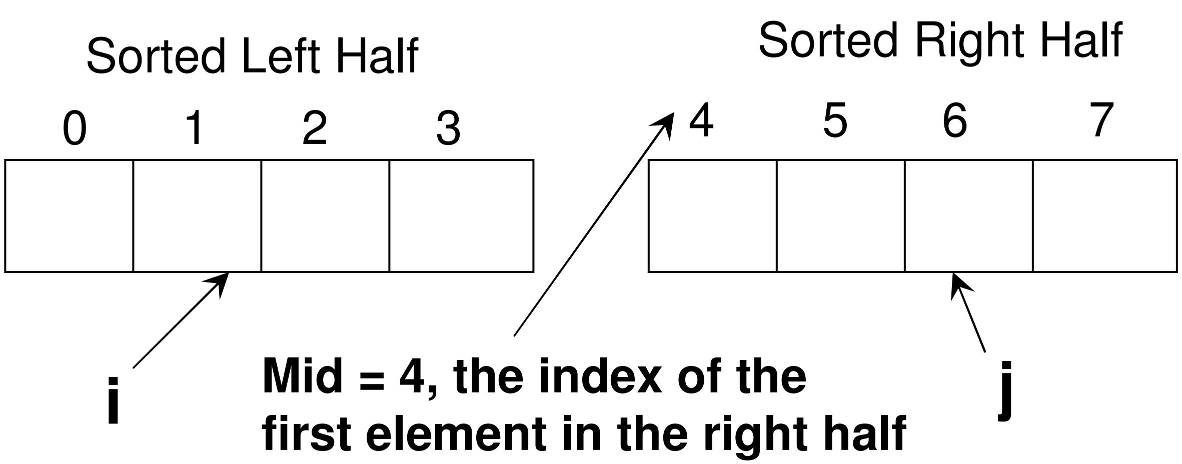
- Given an array A, an inversion is said to have occurred if i < j and A[i] > A[j].

- **<u>Example</u>**

0	1	2	3	4	5
2	**8**	**1**	**9**	**3**	**7**

Inverted Pairs
(2, 1)
(8, 1)
(8, 3)
(8, 7)
(9, 3)
(9, 7)

The number of inversions in an array can be computed as the Sum of the number of inversions encountered in each of the Merging steps of the Merge Sort algorithm.



If A[i] > A[j], then everything to the right of Index in the sorted left half are also going to be greater than A[j]. Hence, the number of inversions due to A[i] > A[j] is: **Mid – i.**

Inversions in the Merge Step (Ex.2)

Mid = 5

0	1	2	3	4	5	6	7	8	9
14	**17**	**19**	**22**	**25**	**13**	**16**	**18**	**20**	**27**

							Sorted Array									
Index i	Index j	A[i]	A[j]	Inv Y/N	# Inv		0	1	2	3	4	5	6	7	8	9
0	5	14	13	Y	5 - 0 = 5		13									
0	6	14	16	N	-		13	14								
1	6	17	16	Y	5 - 1 = 4		13	14	16							
1	7	17	18	N	-		13	14	16	17						
2	7	19	18	Y	5 - 2 = 3		13	14	16	17	18					
2	8	19	20	N	-		13	14	16	17	18	19				
3	8	22	20	Y	5 - 3 = 2		13	14	16	17	18	19	20			
3	9	22	27	N	-		13	14	16	17	18	19	20	22		
4	9	25	27	N	-		13	14	16	17	18	19	20	22	25	
-	9	-	27	N	-		13	14	16	17	18	19	20	22	25	27

 # Inversions in the Merging Step **= 5 + 4 + 3 + 2**
= 14

Inversions in the Merge Step (Ex.2)

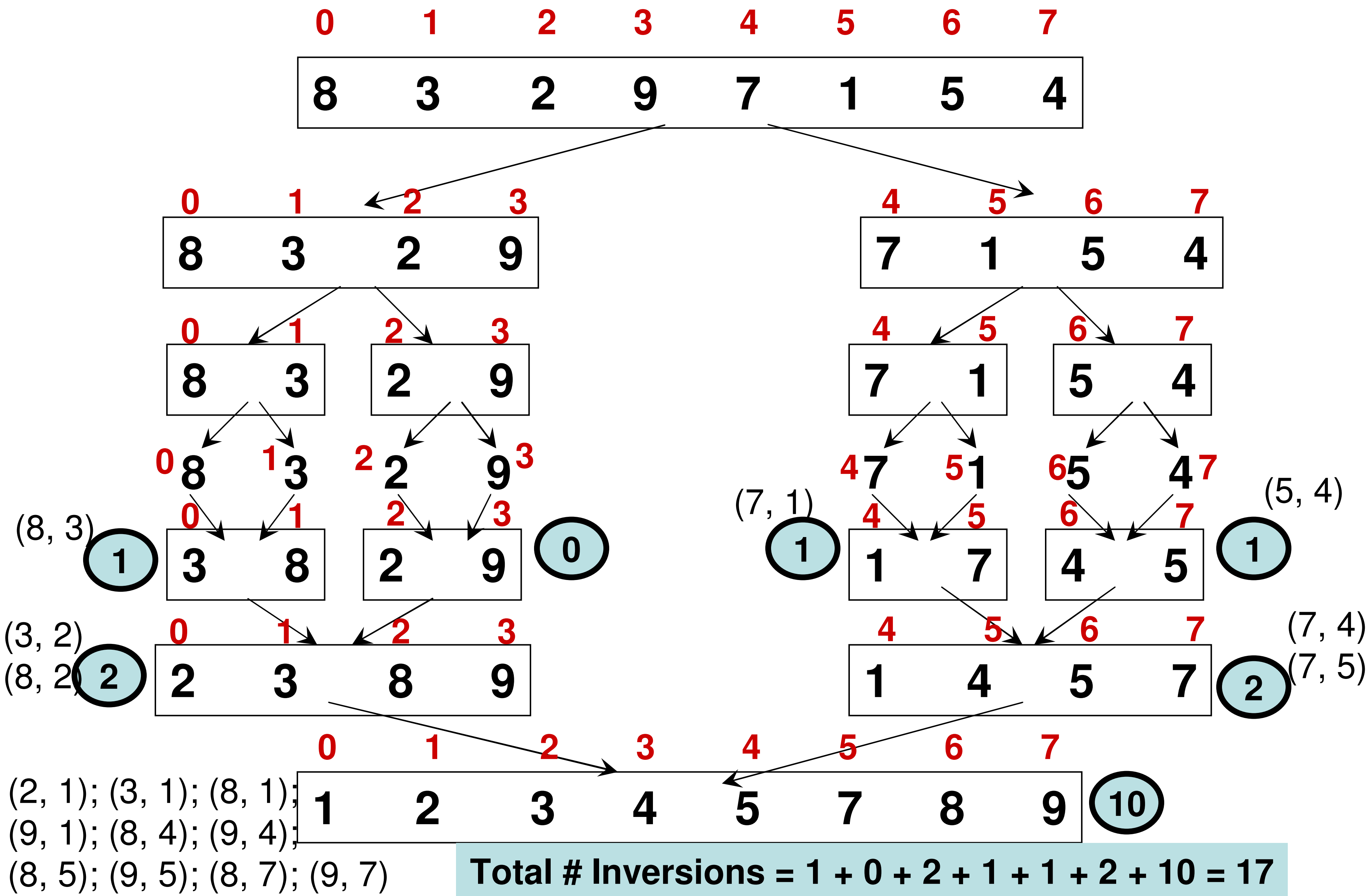
Mid = 5

0	1	2	3	4	5	6	7	8	9
2	**3**	**8**	**9**	**10**	**1**	**4**	**5**	**7**	**8**

							Sorted Array									
Index i	Index j	A[i]	A[j]	Inv Y/N	# Inv		0	1	2	3	4	5	6	7	8	9
0	5	2	1	Y	5 - 0 = 5		1									
0	6	2	4	N	-		1	2								
1	6	3	4	N	-		1	2	3							
2	6	8	4	Y	5 - 2 = 3		1	2	3	4						
2	7	8	5	Y	5 - 2 = 3		1	2	3	4	5					
2	8	8	7	Y	5 - 2 = 3		1	2	3	4	5	7				
2	9	8	8	N	-		1	2	3	4	5	7	8			
3	9	9	8	Y	5 - 3 = 2		1	2	3	4	5	7	8	8		
3	-	9	-	-	-		1	2	3	4	5	7	8	8	9	10

 # Inversions in the Merging Step **= 5 + 3 + 3 + 3 + 2**
= 16

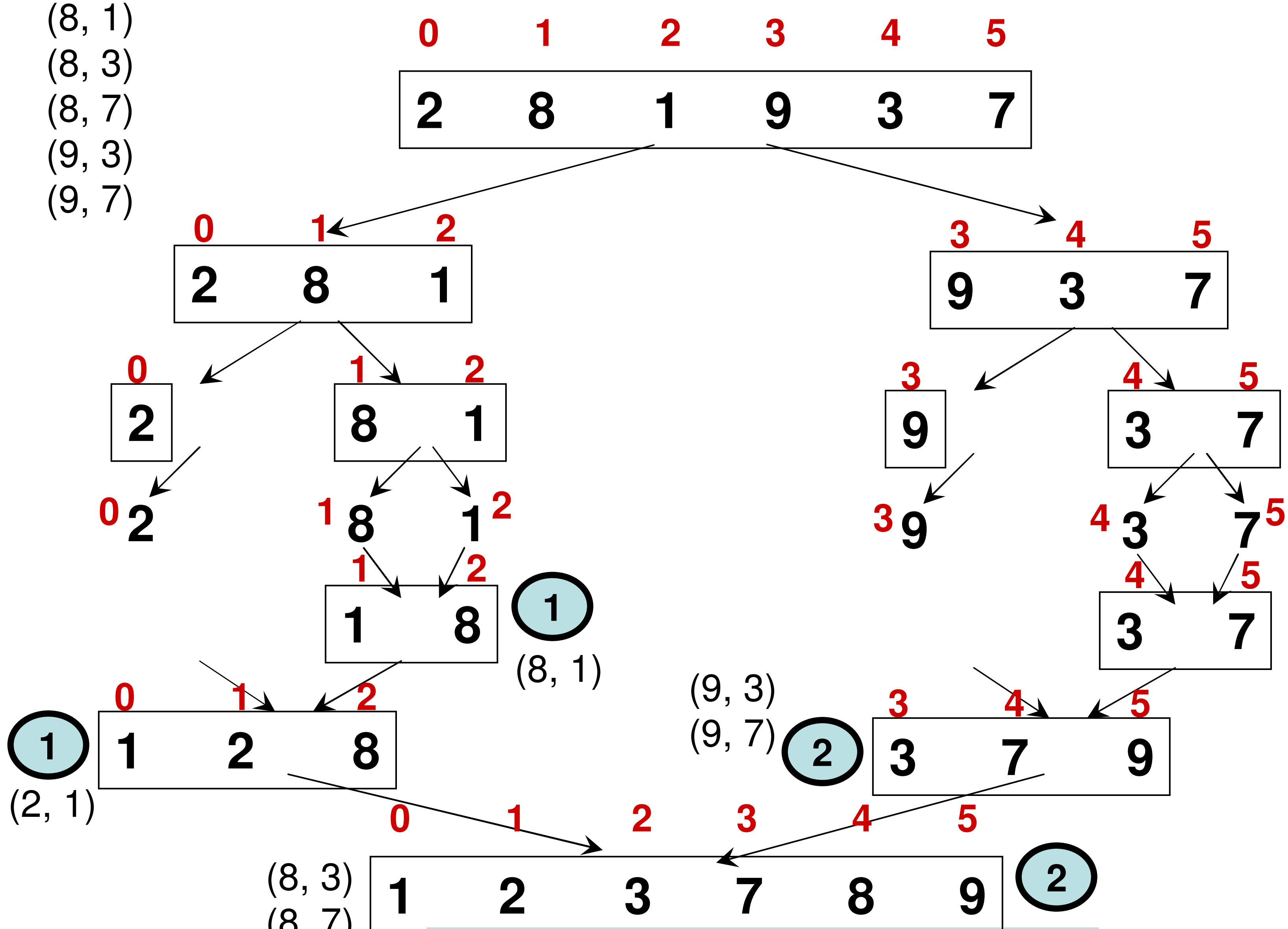
Total # Inversions (all Merging Steps): Ex. 3



Total # Inversions = 1 + 0 + 2 + 1 + 1 + 2 + 10 = 17

Total # Inversions: Ex. 4

Inverted Pairs
(2, 1)
(8, 1)
(8, 3)
(8, 7)
(9, 3)
(9, 7)



Total # Inversions = 1 + 1 + 2 + 2 = 6

Finding the Maximum Integer in an Array: Recursive Divide and Conquer

Algorithm FindMaxIndex(Array A, int leftIndex, int rightIndex)

// returns the index of the maximum left in the array A for //index positions ranging from leftIndex to rightIndex

```
if (leftIndex = rightIndex)
        return leftIndex
middleIndex = (leftIndex + rightIndex)/2
leftMaxIndex = FindMaxIndex(A, leftIndex, middleIndex)
rightMaxIndex = FindMaxIndex(A, middleIndex + 1, rightIndex)
if A[leftMaxIndex] ≥ A[rightMaxIndex]
        return leftMaxIndex
else
        return rightMaxIndex
```

Terminating Condition

Getting ready to divide

Divide part

Conquer part

Since we keep track of the index positions of the maximum element in the sub arrays, We do not need to create additional space. So, this algorithm is in-place.

Max Integer Index Problem: Time Complexity

$T(n) = 2*T(n/2) + 1$

i.e., $T(n) = 2*T(n/2) + \Theta(n^0)$

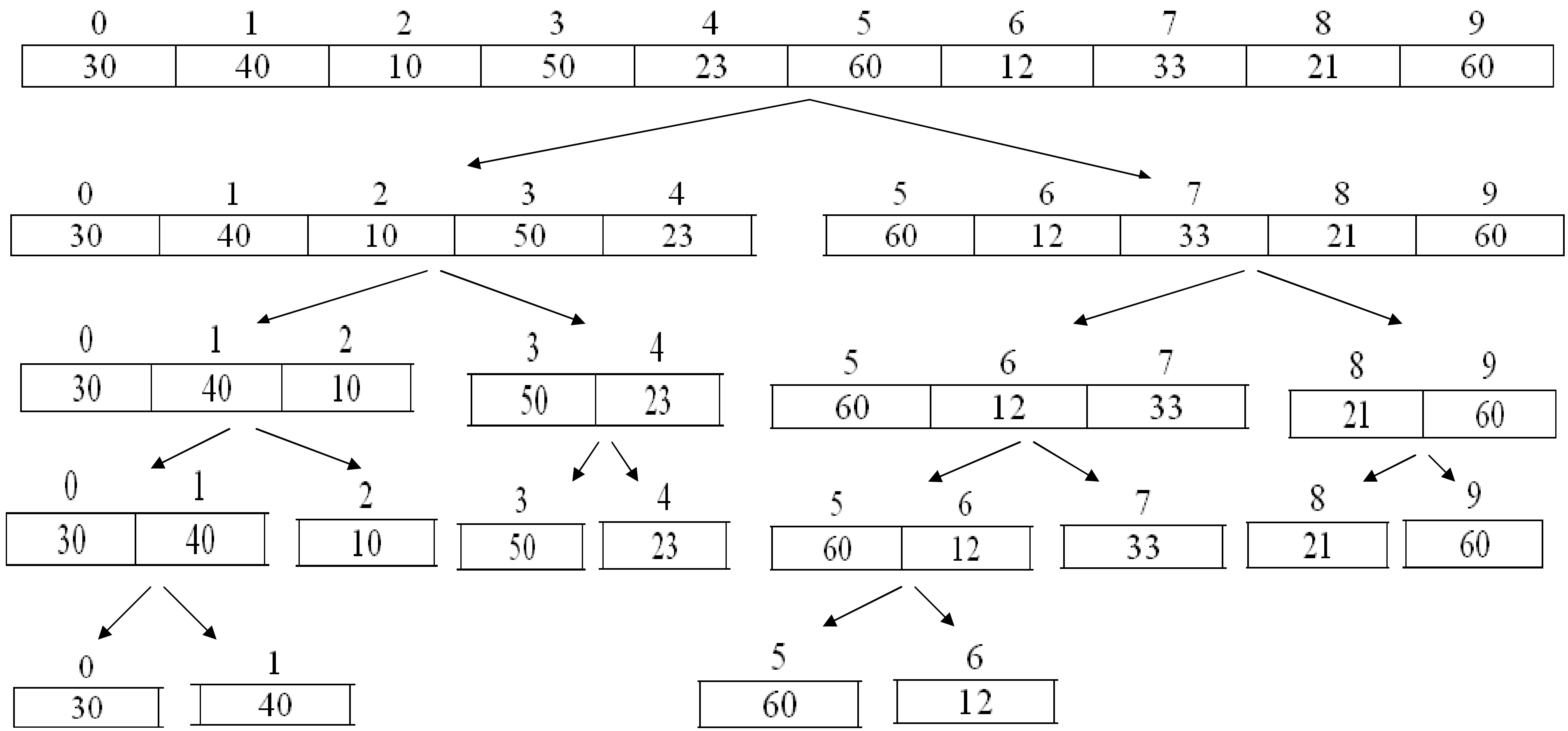
$a = 2$, $b = 2$, $d = 0$

$b^d = 2^0 = 1$. Hence, $a > b^d$

$$\mathbf{T(n) = \Theta(n^{\log_b(a)}) = = \Theta(n^{\log2(2)}) = \Theta(n)}$$

Note that even an iterative approach would take Θ(n) time to compute the time-complexity. The overhead comes with recursion.

FindMaxIndex: Example



FindMaxIndex: Example (contd..)