

Module 2: Divide and Conquer

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

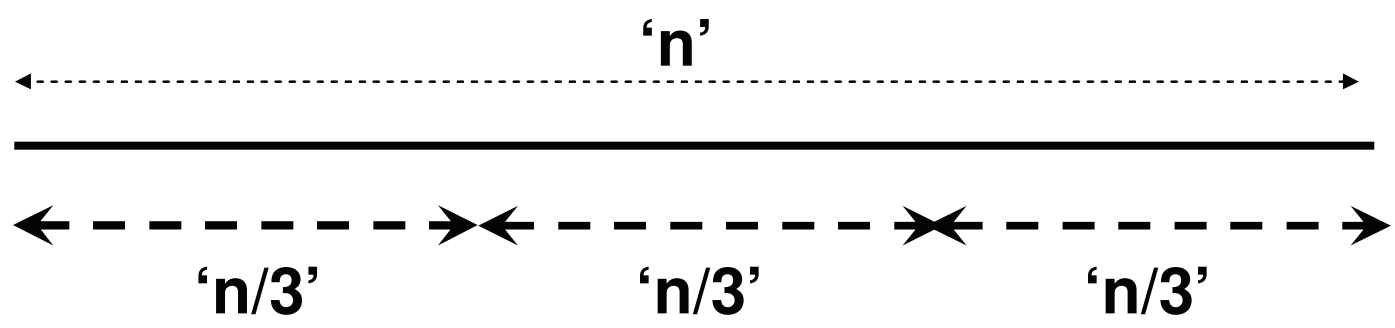
Introduction to Divide and Conquer

- 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.

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

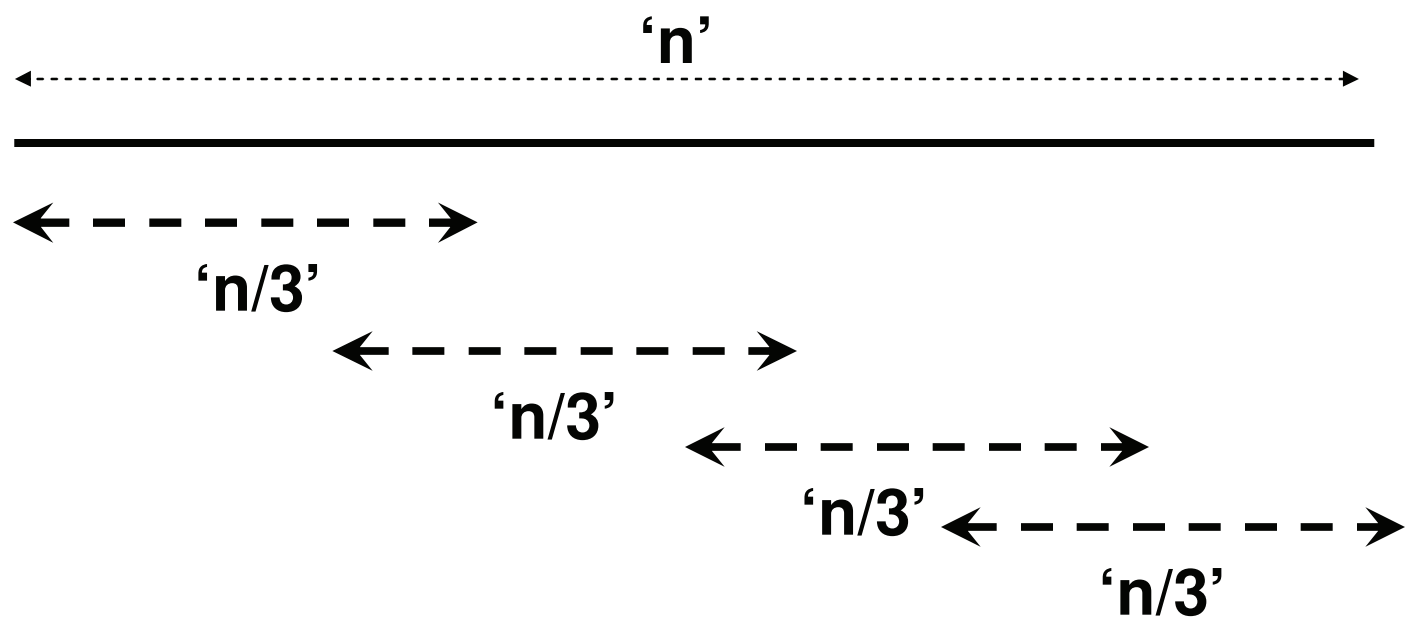
Recurrence Relations for Divide and Conquer

Non-Overlapping Sub Problems



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

Overlapping Sub Problems (a ≠ b)



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

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 $d \geq 0$ 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**.

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 $A[k] \leftarrow C[j]$; $j \leftarrow j + 1$

$k \leftarrow k + 1$

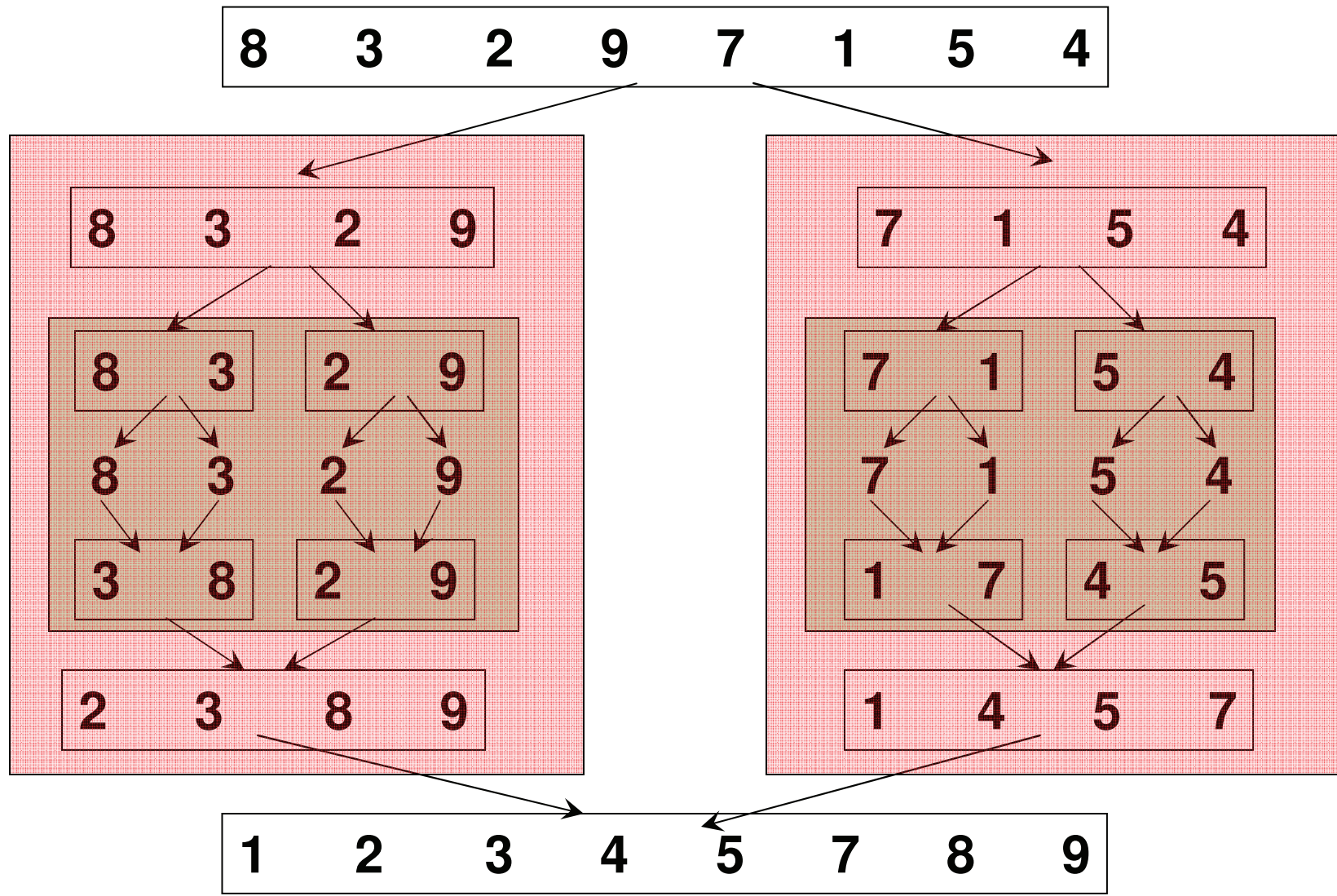
if $i = p$

 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 $B[0.. \lfloor n/2 \rfloor - 1]$

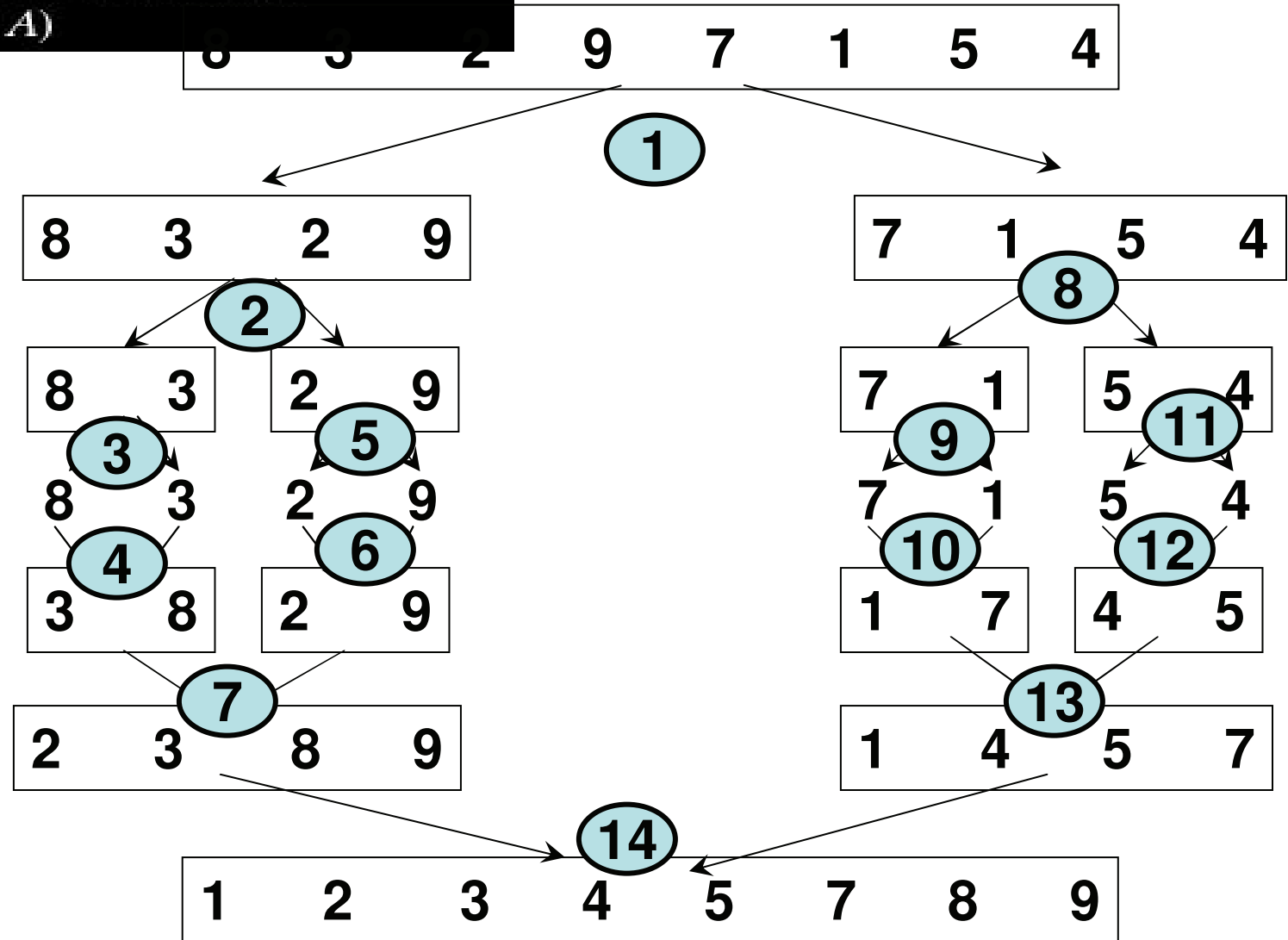
copy $A[\lfloor n/2 \rfloor .. n - 1]$ to $C[0.. \lfloor n/2 \rfloor - 1]$

Mergesort($B[0.. \lfloor n/2 \rfloor - 1]$)

Mergesort($C[0.. \lfloor n/2 \rfloor - 1]$)

Merge(B, C, A)

The order recursion runs



Analysis of Merge Sort

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

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

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. In the worst case, neither of the two arrays becomes empty before the other one contains just one element (e.g., smaller elements may come from the alternating arrays). Hence, the worst case of $C_{\text{merge}}(n) = n - 1$ for $n > 1$.

$$C(n) = 2C(n/2) + (n - 1) \quad f(n) = n - 1 = \Theta(n)$$

Hence, according to Master Theorem,

$$a = 2, b = 2, d = 1$$

$$C(n) \in \Theta(n \log n)$$

$$a = b^d$$

Merge Sort: Space-time Tradeoff

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

Number of Inversions in an Array

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

Inverted Pairs

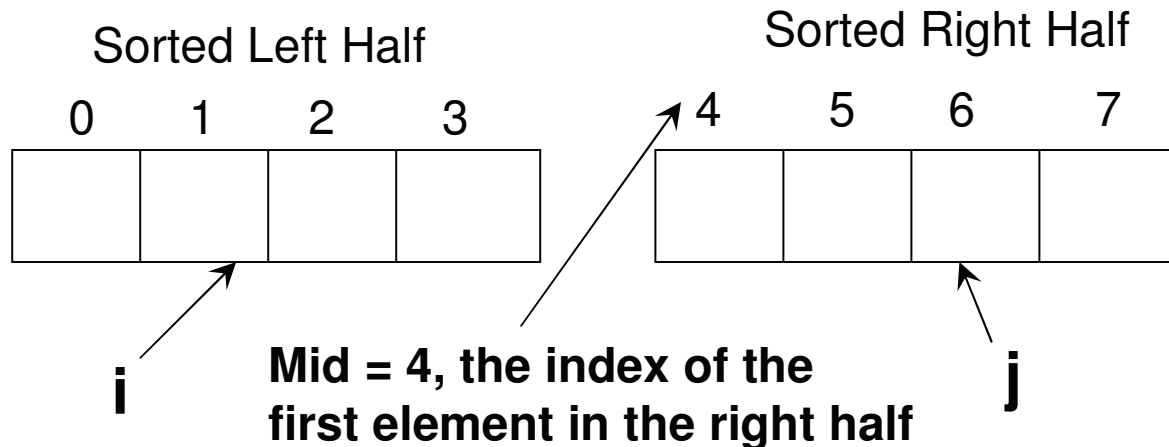
- (2, 1)
- (8, 1)
- (8, 3)
- (8, 7)
- (9, 3)
- (9, 7)

- Example**

0	1	2	3	4	5
2	8	1	9	3	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 i 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>i</i>	Index <i>j</i>	A[<i>i</i>]	A[<i>j</i>]	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)

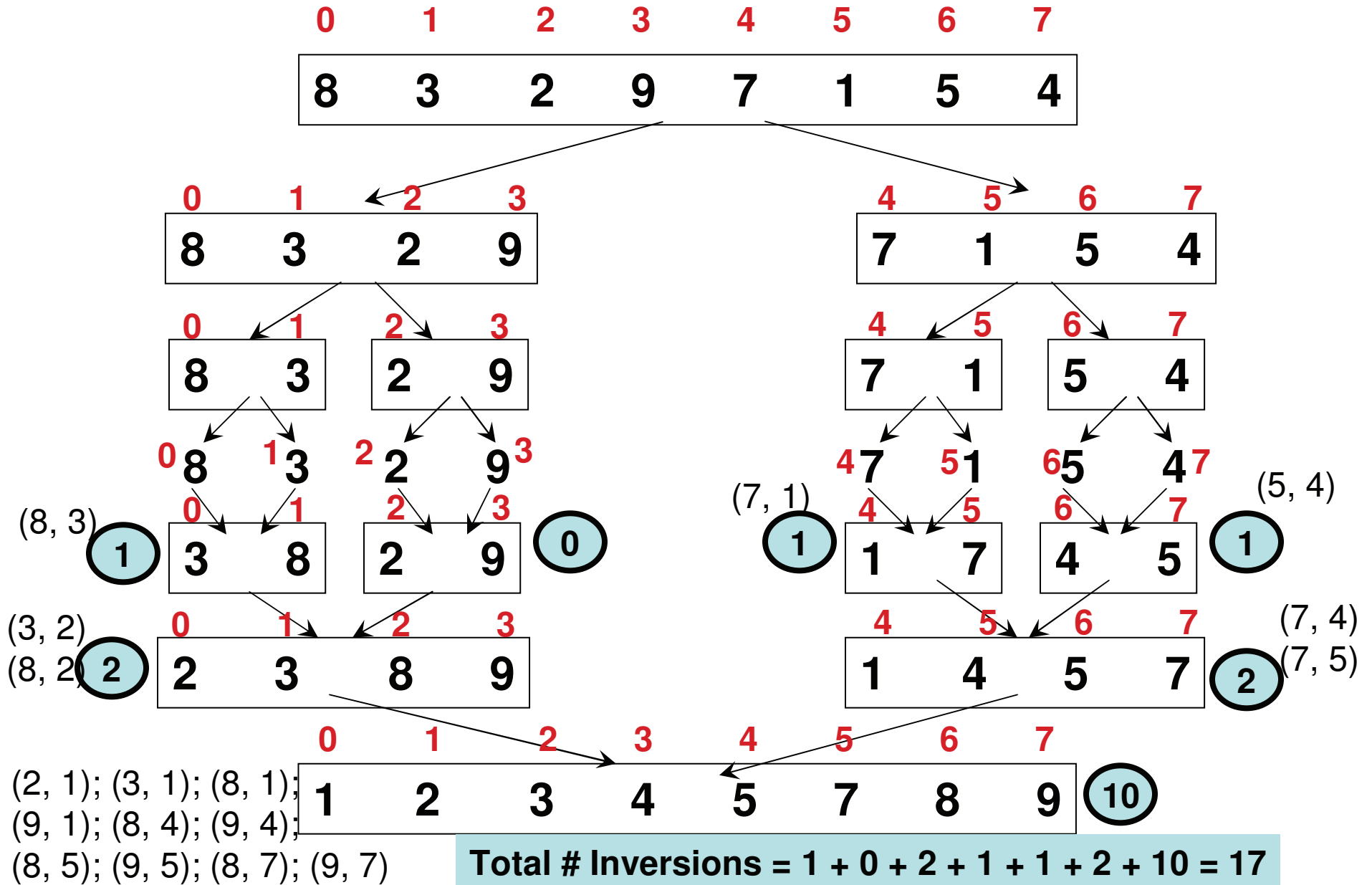
Mid = 5

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

						Sorted Array									
Index	Index	A[i]	A[j]	Inv	# Inv	0	1	2	3	4	5	6	7	8	9
<u>i</u>	<u>j</u>			Y/N											
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

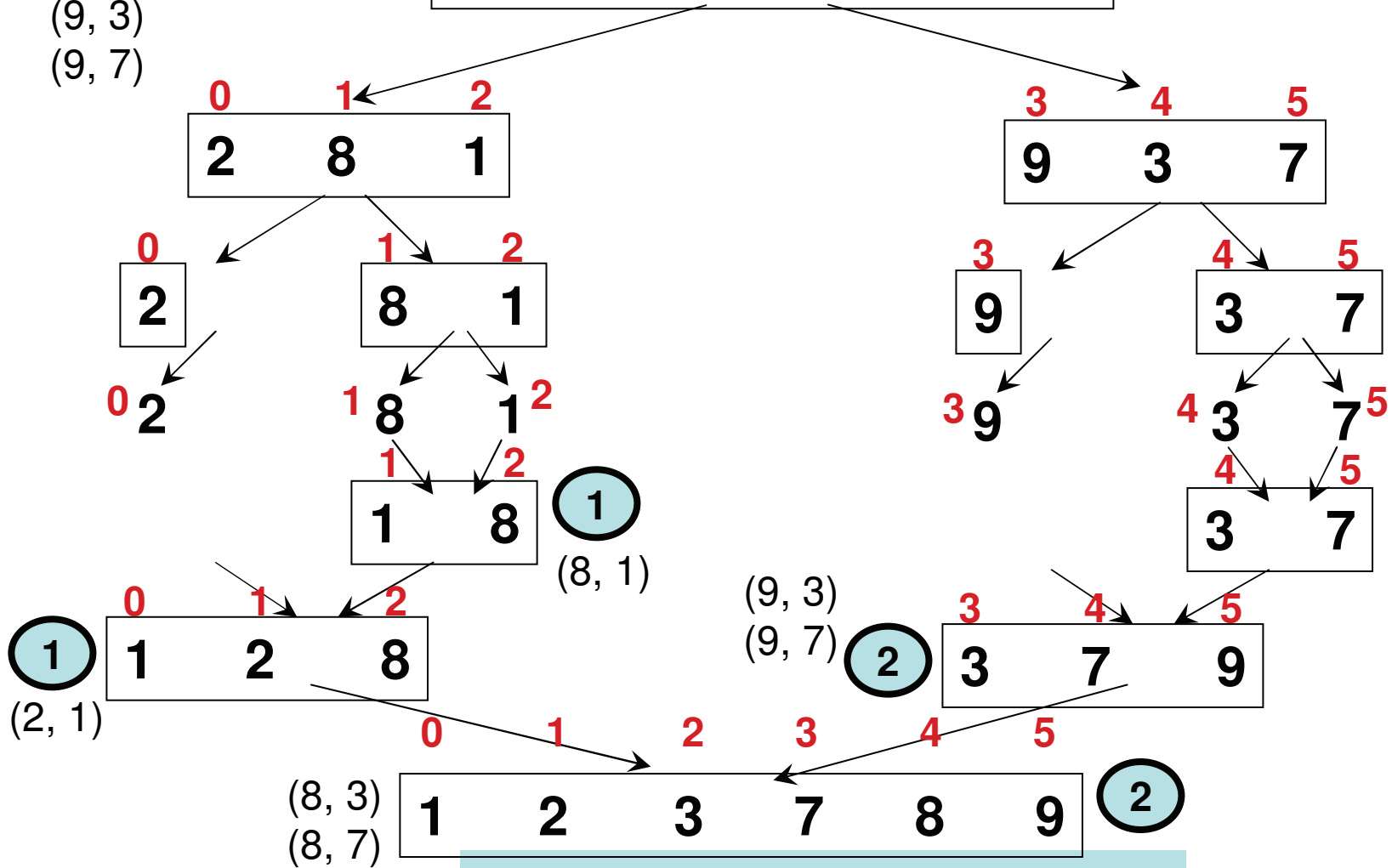
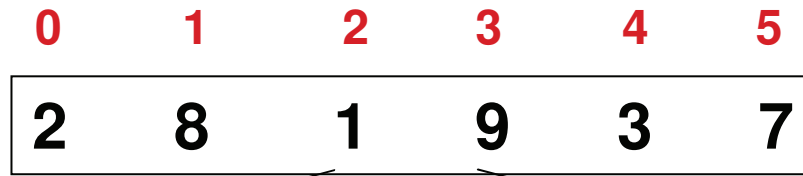
Total # Inversions (all Merging Steps): Ex. 3



Inverted Pairs

- (2, 1)
- (8, 1)
- (8, 3)
- (8, 7)
- (9, 3)
- (9, 7)

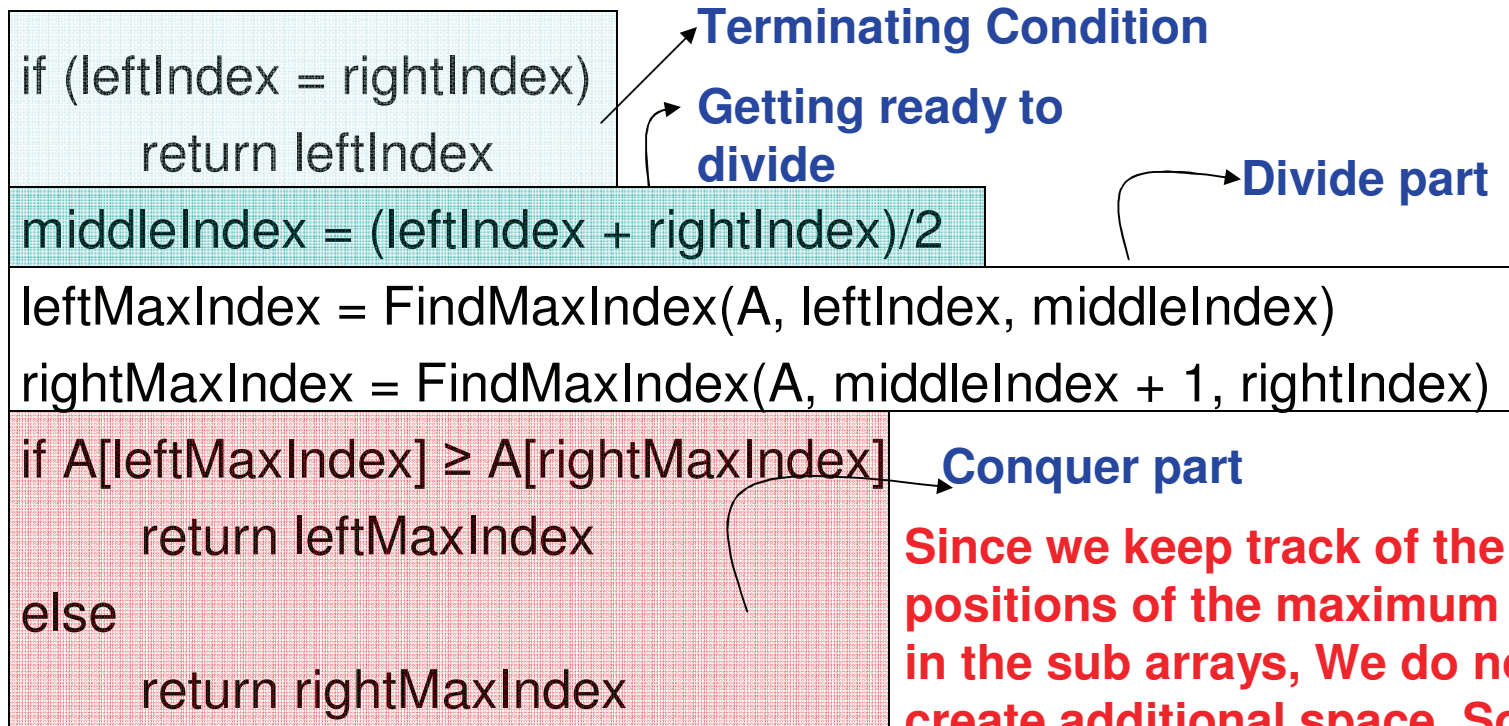
Total # Inversions: Ex. 4



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



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

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

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

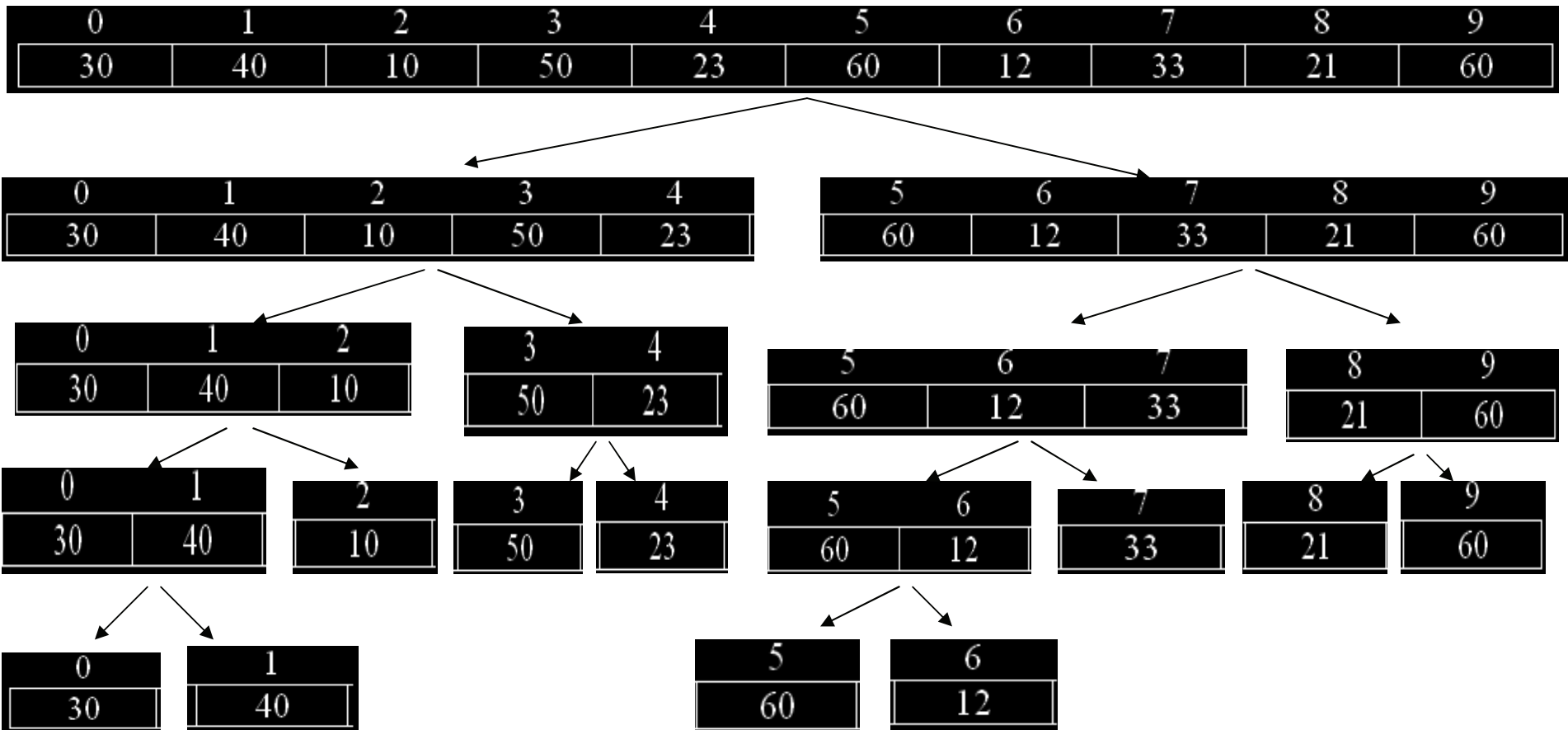
$$a = 2, b = 2, d = 0$$

$$b^d = 2^0 = 1. \text{ Hence, } a > b^d$$

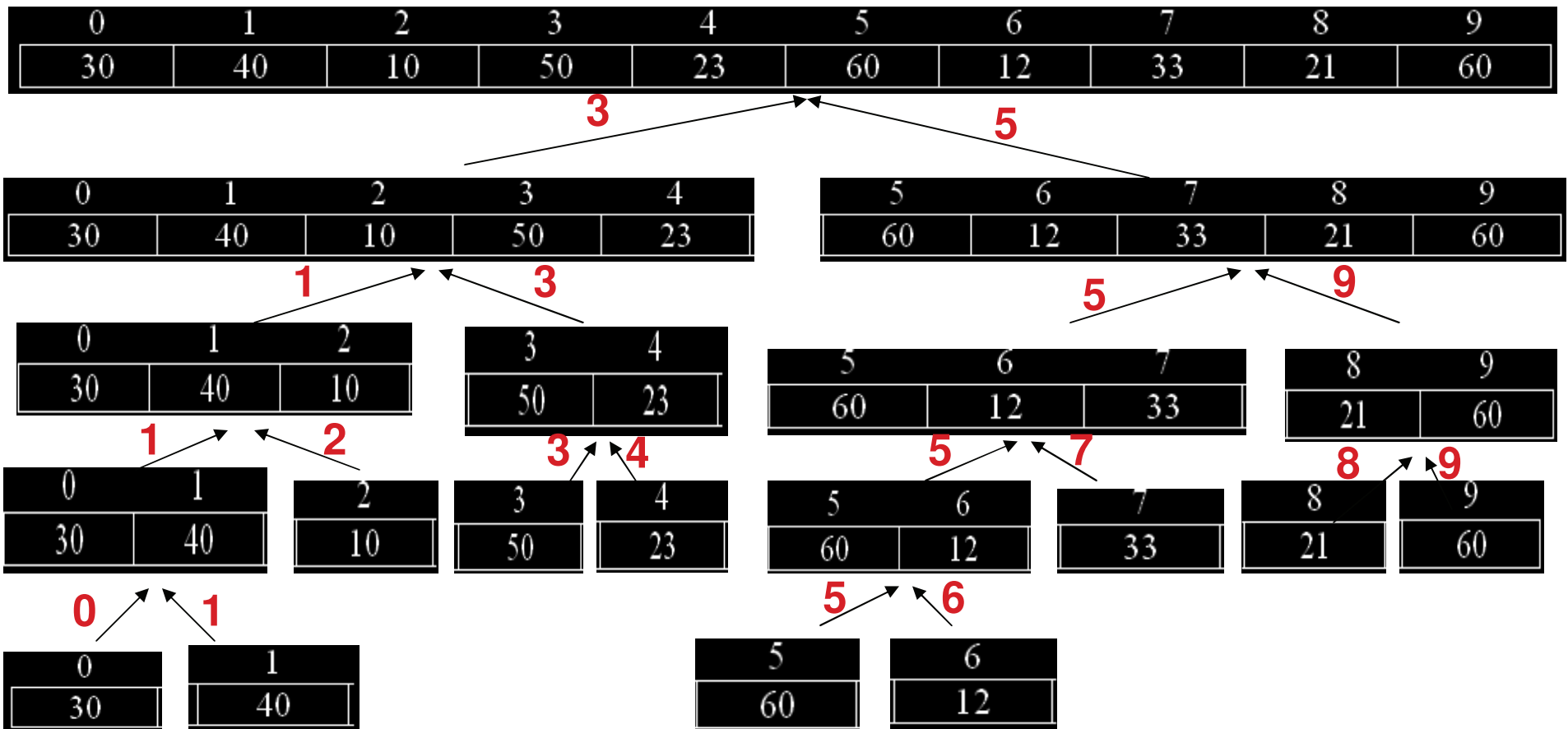
$$\mathbf{T(n) = \Theta(n^{\log_b(a)}) = \Theta(n^{\log_2(2)}) = \Theta(n)}$$

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

FindMaxIndex: Example

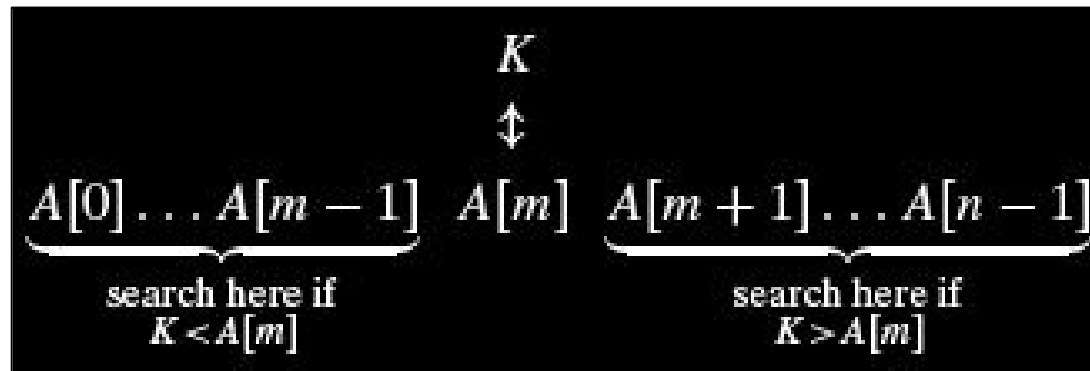


FindMaxIndex: Example (contd..)



Binary Search

- Binary search is a $\Theta(\log n)$, highly efficient search algorithm, in a sorted array.
- It works by comparing a search key K with the array's middle element $A[m]$. If they match, the algorithm stops; otherwise, the same operation is repeated recursively for the first half of the array if $K < A[m]$, and for the second half if $K > A[m]$.
- Though binary search is based on a recursive idea, it can be easily implemented as a non-recursive algorithm.



Binary Search

Example

Search Key K = 70

l=0	r=12	m=6
l=7	r=12	m=9
l=7	r=8	m=7

index	0	1	2	3	4	5	6	7	8	9	10	11	12	
value	3	14	27	31	39	42	55	70	74	81	91	93	98	
iteration 1	l		m					r						
iteration 2								l		m		r		
iteration 3								l,m		r				

ALGORITHM *BinarySearch*(A[0..n - 1], K)

//Implements nonrecursive binary search

//Input: An array A[0..n - 1] sorted in ascending order and

// a search key K

//Output: An index of the array's element that is equal to K

// or -1 if there is no such element

$l \leftarrow 0; \quad r \leftarrow n - 1$

while $l \leq r$ **do**

$m \leftarrow \lfloor (l + r)/2 \rfloor$

if $K = A[m]$ **return** m

else if $K < A[m]$ $r \leftarrow m - 1$

else $l \leftarrow m + 1$

return -1

$$C(n) = C(n/2) + 1 \text{ for } n > 1$$

$$C(1) = 1$$

$$C(n) = C(n/2) + \Theta(1) \text{ for } n > 1$$

$$a = 1, b = 2, d = 0$$

$$a = b^d$$

$$C(n) = \Theta(n^0 \log n) = \Theta(\log n)$$

Binary Search

index	0	1	2	3	4	5	6	7	8	9	10	11	12
value	3	14	27	31	39	42	55	70	74	81	91	93	98
iteration 1	l						m						r
iteration 2	l		m			r							
iteration 3	l, m	r							r				
iteration 4		l											
		m											
		r											
iteration 5	r	l											

$l=0$ $r=5$ $m=2$

$l=0$ $r=1$ $m=0$

$l=1$ $r=1$ $m=1$

$l=1$ $r=0$ STOP!!

Applications of Binary Search (1)

Searching for a Threshold Value for a Function

- Sample Scenario
- Consider a monotonically decreasing function $f(n) = 2/n^2$, where n is a positive integer ($n > 0$).
- We need to develop a $\Theta(\log n)$ algorithm that would determine the smallest value of n (called the threshold value) for which $f(n)$ would be less than a target value 't' (say, $t = 0.01$).

n	$f(n) = 2/n^2$	n	$f(n) = 2/n^2$
1	2	9	0.0247
2	0.5	10	0.02
3	0.222	11	0.0165
4	0.125	12	0.0139
5	0.08	13	0.0118
6	0.0556	14	0.0102
7	0.0408	15	0.0089
8	0.0313	16	0.0078

- Solution Approach
- Invariants (something that will remain true throughout the algorithm):
 - We will keep the Left Index as an 'n' value for which $f(n)$ is always going to be greater than or equal to the threshold value
 - We will keep the Right Index as an 'n' value for which $f(n)$ is always going to be less than the threshold value.
- We will go through a sequence of iterations of Binary Search until the difference between the Right Index and Left Index is greater than ONE (note: we are dealing with integers here).
 - In each iteration, the middle index is the average of the Left Index and Right Index.
 - If $f(\text{Middle Index}) < \text{target}$, we set: $\text{Right Index} = \text{Middle Index}$
 - If $f(\text{Middle Index}) \geq \text{target}$, we set: $\text{Left Index} = \text{Middle Index}$
 - In each iteration, either the Left Index increases or the Right Index decreases.
 - The moment the difference between the Left Index and Right Index is equal to 1, we will exit from the loop and say that the value of the Right Index is the threshold (smallest integer) value of 'n' for which the function value is less than the target.

Function $f(n) = 2/n^2$; for $n > 0$

Target = 0.01

Left Index = 1; $f(\text{Left Index}) = 2 > \text{target}$

Right Index = 100; $f(\text{Right Index}) = 2/100^2 = 0.0002 < \text{target}$

It #	Left Index	Right Index	Middle Index	f(Middle Index)
1	1	100	$(1 + 100)/2 = 50$	$0.0008 < \text{target}$
2	1	50	$(1 + 50)/2 = 25$	$0.0032 < \text{target}$
3	1	25	$(1 + 25)/2 = 13$	$0.0118 > \text{target}$
4	13	25	$(13 + 25)/2 = 19$	$0.0055 < \text{target}$
5	13	19	$(13 + 19)/2 = 16$	$0.0078 < \text{target}$
6	13	16	$(13 + 16)/2 = 14$	$0.0102 > \text{target}$
7	14	16	$(14 + 16)/2 = 15$	$0.0089 < \text{target}$
8	14	15	STOP!	

Threshold = Value of Right Index when we stop the iterations
= 15

Iterations = $\log_2(\text{the starting Right Index}) = \log_2(100) \sim 7$

Solution:

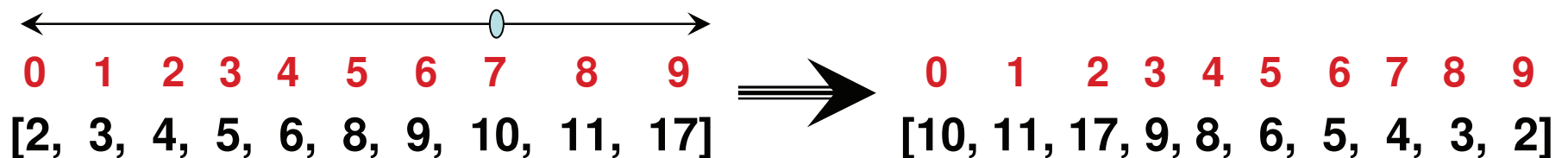
15 is the smallest integer for which the function $f(n) = 2/n^2$ is less than 0.01

Algorithm to Generate a (Maximum) Unimodal Array

- A (maximum) unimodal array is an array that has a sequence of monotonically increasing integers followed by a sequence of monotonically decreasing integers.
- All elements in the array are unique
- Examples
 - {4, 5, 8, 9, 10, 11, 7, 3, 2, 1}: Max. Element: 11
 - There is an increasing seq. followed by a decreasing seq.
 - {11, 9, 8, 7, 5, 4, 3, 2, 1}: Max. Element: 11
 - There is no increasing seq. It is simply a decreasing seq.
 - {1, 2, 3, 4, 5, 7, 8, 9, 11}: Max. Element: 11
 - There is an increasing seq., but there is no decreasing seq.

Unimodal Array Generation Algorithm Steps

- Step 1: For a given array size, generate a random array of unique elements in a certain range
 - Ex: [4, 11, 8, 10, 5, 6, 17, 3, 2, 9]
- Step 2: Sort (in increasing order) the array of Step 1
 - Ex: [2, 3, 4, 5, 6, 8, 9, 10, 11, 17]
- Step 3: Pick a random index (call it the break index) such that the contents of the array are rearranged around that index, as shown in the example below (let the break index be '7'):



Step 1: To generate a random array of unique elements.

Let 'N' be the size of the targeted unimodaly array (A) and 'M' be the maximum value for an element in the array.

Let B be an array of unique elements with values [1, 2, 3, ..., M] with the accessible indexes 0, 1, 2, ..., M-1.

Go through a sequence of iterations as follows.

- Iteration 1:
 - Generate a random index (say, r_1) in the range [0, ..., M-1].
 - Copy B[r_1] to A[0]
 - Swap B[r_1] and B[M-1].
 - The accessible indexes of B for further iterations are now: [0, 1, 2, ..., M-2]
- Iteration 2:
 - Generate a random index (say, r_2) in the range [0, ..., M-2].
 - Copy B[r_2] to A[1]
 - Swap B[r_2] and B[M-2].
 - The accessible indexes of B for further iterations are now: [0, 1, 2, ..., M-3]
-
-
- Iteration j (in general):
 - Generate a random index (say, r_j) in the range [0, ..., M-j]
 - Copy B[r_j] to A[j-1]
 - Swap B[r_j] and B[M-j].
 - The accessible indexes of B for further iterations are now: [0, 1, 2, ..., M-(j+1)]

Let 'N' = 10; 'M' = 20

Initial

A = []

0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19
B = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20]

Accessible indexes of B: [0, ..., 19]

Iteration 1

Random Index r1: 7

A = [8]

0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19
B = [1, 2, 3, 4, 5, 6, 7, 20, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 8]

Accessible indexes of B: [0, ..., 18]

Iteration 2

Random Index r2: 11

A = [8, 12]

0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19
B = [1, 2, 3, 4, 5, 6, 7, 20, 9, 10, 11, 19, 13, 14, 15, 16, 17, 18, 12, 8]

Accessible indexes of B: [0, ..., 17]

Iteration 3

Random Index r3: 8

A = [8, 12, 9]

0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19

B = [1, 2, 3, 4, 5, 6, 7, 20, 18, 10, 11, 19, 13, 14, 15, 16, 17, 9, 12, 8]

Accessible indexes of B: [0, ..., 16]

Iteration 4

Random Index r4: 0

A = [8, 12, 9, 1]

0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19

B = [17, 2, 3, 4, 5, 6, 7, 20, 18, 10, 11, 19, 13, 14, 15, 16, 1, 9, 12, 8]

Accessible indexes of B: [0, ..., 15]

Iteration 5

Random Index r5: 15

A = [8, 12, 9, 1, 16]

0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19

B = [17, 2, 3, 4, 5, 6, 7, 20, 18, 10, 11, 19, 13, 14, 15, 16, 1, 9, 12, 8]

Accessible indexes of B: [0, ..., 14]

Iteration 6

Random Index r6: 4

A = [8, 12, 9, 1, 16, 5]

0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19

B = [17, 2, 3, 4, 15, 6, 7, 20, 18, 10, 11, 19, 13, 14, 5, 16, 1, 9, 12, 8]

Accessible indexes of B: [0, ..., 13]

Iteration 7

Random Index r7: 3

A = [8, 12, 9, 1, 16, 5, 4]

0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19

B = [17, 2, 3, 14, 15, 6, 7, 20, 18, 10, 11, 19, 13, 4, 5, 16, 1, 9, 12, 8]

Accessible indexes of B: [0, ..., 12]

Iteration 8

Random Index r8: 7

A = [8, 12, 9, 1, 16, 5, 4, 20]

0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19

B = [17, 2, 3, 14, 15, 6, 7, 13, 18, 10, 11, 19, 20, 4, 5, 16, 1, 9, 12, 8]

Accessible indexes of B: [0, ..., 11]

Iteration 9

Random Index r9: 11

A = [8, 12, 9, 1, 16, 5, 4, 20, 19]

0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19

B = [17, 2, 3, 14, 15, 6, 7, 13, 18, 10, 11, 19, 20, 4, 5, 16, 1, 9, 12, 8]

Accessible indexes of B: [0, ..., 10]

Iteration 10

Random Index r10: 10

A = [8, 12, 9, 1, 16, 5, 4, 20, 19, 10]

0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19

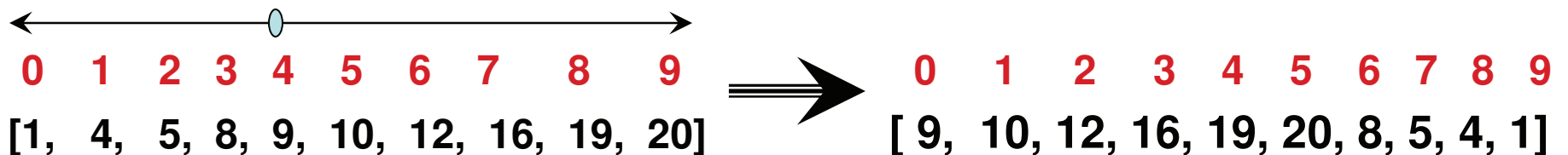
B = [17, 2, 3, 14, 15, 6, 7, 13, 18, 11, 10, 19, 20, 4, 5, 16, 1, 9, 12, 8]

Accessible indexes of B: [0, ..., 9]

Step 1: Random array of unique elements: [8, 12, 9, 1, 16, 5, 4, 20, 19, 10]

Step 2: Sort the random array of Step 1: [1, 4, 5, 8, 9, 10, 12, 16, 19, 20]

Step 3: Generate a random index (break index) in the range [0, ..., 9]. Let it be '4'



Applications of Binary Search (2)

Finding the Maximum Element in a Unimodal Array

- A unimodal array is an array that has a sequence of monotonically increasing integers followed by a sequence of monotonically decreasing integers.
- All elements in the array are unique
- Examples
 - {4, 5, 8, 9, 10, 11, 7, 3, 2, 1}: Max. Element: 11
 - There is an increasing seq. followed by a decreasing seq.
 - {11, 9, 8, 7, 5, 4, 3, 2, 1}: Max. Element: 11
 - There is no increasing seq. It is simply a decreasing seq.
 - {1, 2, 3, 4, 5, 7, 8, 9, 11}: Max. Element: 11
 - There is an increasing seq., but there is no decreasing seq.
- Algorithm: Modified binary search.

Applications of Binary Search (2)

Finding the Maximum Element in a Unimodal Array

L = 0; R = n-1

while (L < R) do

 m = (L+R)/2

 if A[m] < A[m+1]

 L = m+1 // max. element is from m+1 to R

 else if A[m] > A[m+1]

 R = m // max. element is from L to m

end while

return A[L]

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

Using Master Theorem,

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

Space complexity: $\Theta(1)$

0 1 2 3 4 5 6 7 8 9

3	5	8	9	10	14	11	4	2	1
---	---	---	---	----	----	----	---	---	---

L = 0; R = 9; m = 4: A[m] < A[m+1]

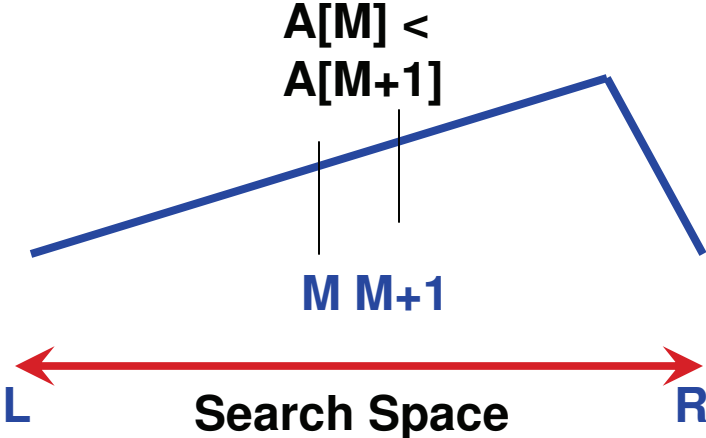
L = 5; R = 9; m = 7: A[m] > A[m+1]

L = 5; R = 7; m = 6: A[m] > A[m+1]

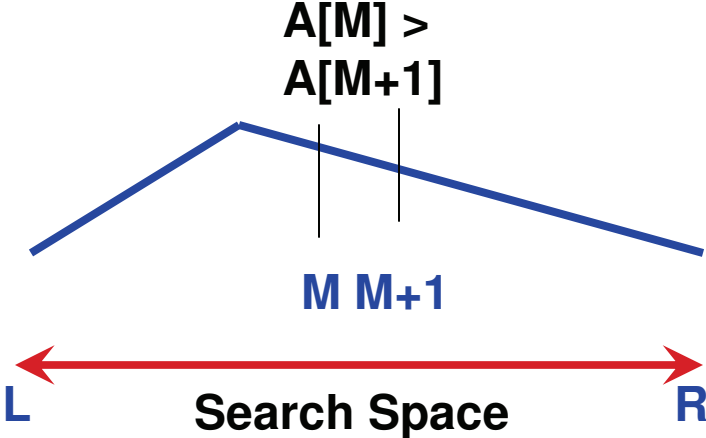
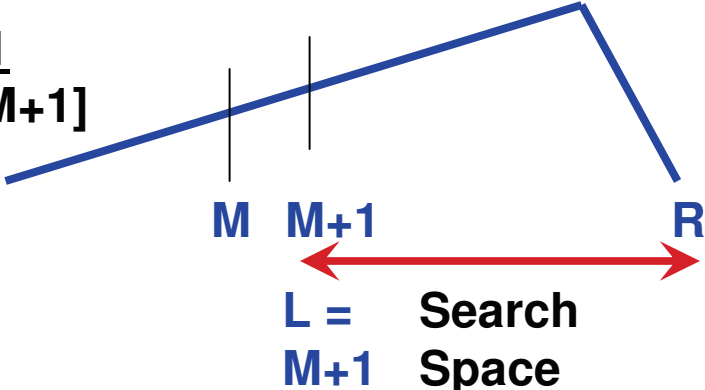
L = 5; R = 6; m = 5: A[m] > A[m+1]

L = 5; R = 5; return A[5] = 14

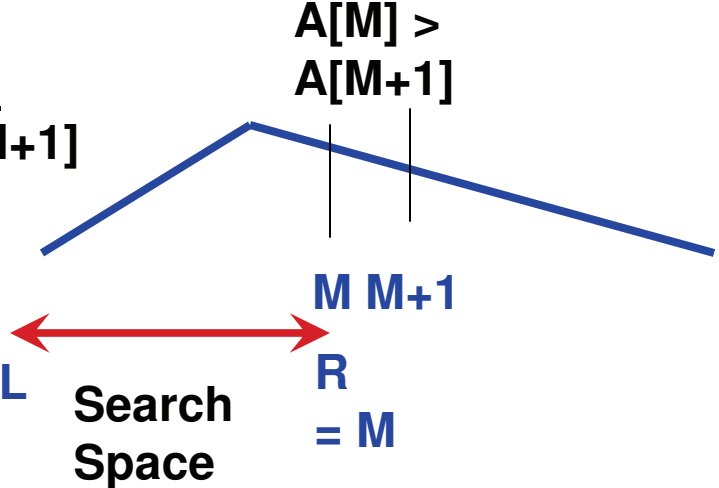
Two Scenarios



Scenario 1
 $A[M] < A[M+1]$



Scenario 2
 $A[M] > A[M+1]$



Applications of Binary Search (2)

Finding the Maximum Element in a Unimodal Array

- Proof of Correctness
 - We always maintain the invariant that the maximum element lies in the range of indexes: $L \dots R$.
 - If $A[m] < A[m+1]$, then, the maximum element has to be either at index $m+1$ or to the right of index $m+1$. Hence, we set $L = m+1$ and retain R as it is, maintaining the invariant that the maximum element is in the range $L \dots R$.
 - If $A[m] > A[m+1]$, then, the maximum element is either at index m or before index m . Hence, we set $R = m$ and retain L as it is, maintaining the invariant that the maximum element is in the range $L \dots R$.
 - The loop runs as long as $L < R$. Once $L = R$, the loop ends and we return the maximum element.

Applications of Binary Search (2)

Finding the Maximum Element in a Unimodal Array

L = 0; R = n-1

while (L < R) do

 m = (L+R)/2

 if A[m] < A[m+1]

 L = m+1 // max. element is from m+1 to R

 else if A[m] > A[m+1]

 R = m // max. element is from L to m

end while

return A[L]

0 1 2 3 4 5

3	5	8	9	10	14
---	---	---	---	----	----

L = 0; R = 5; m = 2: A[m] < A[m+1]

L = 3; R = 5; m = 4: A[m] < A[m+1]

L = 5; R = 5; return A[5] = 14

Applications of Binary Search (3)

Local Minimum in an Array

- Problem: Given an array $A[0, \dots, n-1]$, an element at index i ($0 < i < n-1$) is a local minimum if $A[i] < A[i-1]$ as well as $A[i] < A[i+1]$. That is, the element is lower than the element to the immediate left as well as to the element to the immediate right.
- Constraints:
 - The array has at least three elements
 - The first two numbers are decreasing and the last two numbers are increasing.
 - The numbers are unique
- Example:
 - Let $A = \{8, 5, 7, 2, 3, 4, 1, 9\}$; the array has several local minimums. These are: 5, 2 and 1.
- Algorithm: Do a binary search and see if every element we index into is a local minimum or not.
 - If the element we index into is not a local minimum, then we search on the half corresponding to the smaller of its two neighbors.

Applications of Binary Search (3)

Local Minimum in an Array

Examples

1)

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

Iteration 1: $L = 0$; $R = 7$; $M = (L+R)/2 = 3$ Element at $A[3]$ is a local minimum.

2)

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

Iteration 1: $L = 0$; $R = 7$; $M = (L+R)/2 = 3$ Element at $A[3]$ is NOT a local minimum.
Search in the space $[0...2]$ corresponding to the smaller neighbor '2'

Iteration 2: $L = 0$; $R = 2$; $M = (L+R)/2 = 1$ Element at $A[1]$ is NOT a local minimum.
Search in the space $[2...2]$ corresponding to the smaller neighbor '2'

Iteration 3: $L = 2$; $R = 2$; $M = (L+R)/2 = 2$. Element at $A[2]$ is a local minimum.

Applications of Binary Search (3)

Local Minimum in an Array

Examples

3)

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

Iteration 1: $L = 0$; $R = 10$; $M = (L+R)/2 = 5$ Element at $A[5]$ is NOT a local minimum.

Search in the space $[6...10]$ corresponding to the smaller neighbor '1'

Iteration 2: $L = 6$; $R = 10$; $M = (L+R)/2 = 8$ Element at $A[8]$ is NOT a local minimum.

Search in the space $[9...10]$ corresponding to the smaller neighbor '-8'

Iteration 3: $L = 9$; $R = 10$; $M = (L+R)/2 = 9$. Element at $A[9]$ is a local minimum. STOP

Time-Complexity Analysis

Recurrence Relation: $T(n) = T(n/2) + 3$ for $n > 3$

Basic Condition: $T(3) = 2$

Using Master Theorem, we have

$a = 1$, $b = 2$, $d = 0 \rightarrow a = b^d$.

Hence, $T(n) = \Theta(n^d \log n) = \Theta(n^0 \log n) = \Theta(\log n)$

- One comparison for $A[M]$ with $A[M+1]$
- One comparison for $A[M]$ with $A[M-1]$
- One comparison for $A[M-1]$ with $A[M+1]$

Space Complexity: As all evaluations are done on the input array itself, no extra space proportional to the input is needed. Hence, space complexity is $\Theta(1)$.

Applications of Binary Search (3)

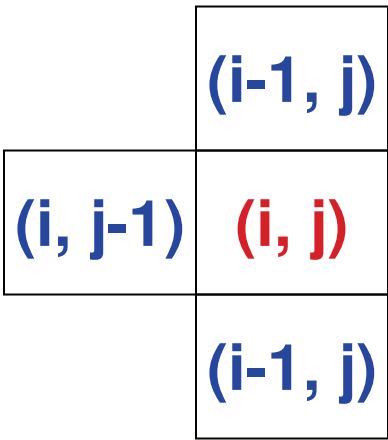
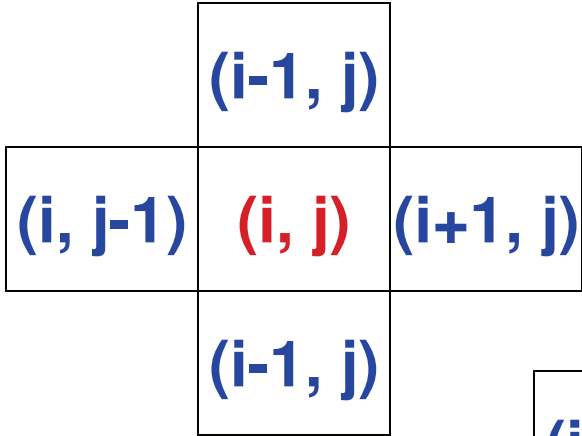
Local Minimum in an Array

- Constraints:
 - The array has at least three elements
 - The first two numbers are decreasing and the last two numbers are increasing.
 - The numbers are unique
- Theorem: If the above three constraints are met for an array, then the array has to have at least one local minimum.
- Proof: Let us prove by contradiction.
 - If the second number is not to be a local minimum, then the third number in the array has to be less than the second number.
 - Continuing like this, if the third number is not to be a local minimum, then the fourth number has to be less than the third number and so on.
 - Again, continuing like this, if the penultimate number is not to be a local minimum, then the last number in the array has to be smaller than the penultimate number. This would mean the second constraint is violated (and also the array is basically a monotonically decreasing sequence). A contradiction.

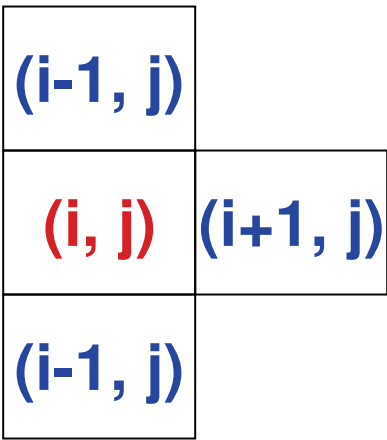
Applications of Binary Search (4)

Local Minimum in a Two-Dimensional Array

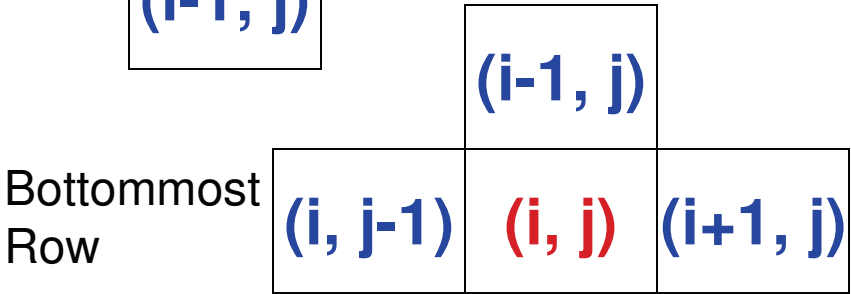
- An element is a local minimum in a two-dim array if the element is the minimum compared to the elements to its immediate left and right as well as to the elements to its immediate top and bottom.
 - If an element is in the edge row or column, it is compared only to the elements that are its valid neighbors.



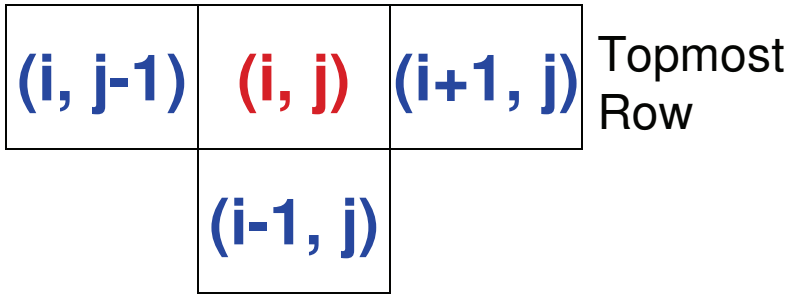
Rightmost column



Leftmost Column



Bottommost Row



Topmost Row

Applications of Binary Search (4)

Local Minimum in a Two-Dimensional Array

Given an array $A[0 \dots \text{numRows}-1][0 \dots \text{numCols}-1]$

$\text{TopRowIndex} = 0$

$\text{BottomRowIndex} = \text{numRows} - 1$

while ($\text{TopRowIndex} \leq \text{BottomRowIndex}$) **do**

$\text{MidRowIndex} = (\text{TopRowIndex} + \text{BottomRowIndex}) / 2$

$\text{MinColIndex} = \text{FindMinColIndex}(A[\text{MidRowIndex}][\])$

/ Finds the col index with the minimum element in the row
corresponding to MidRowIndex */*

$\text{MinRowIndex} = \text{FindMinRowIndexNeighborhood}(A, \text{MidRowIndex},$
 $\text{MinColIndex})$

/ Finds the min entry in the column represented by MinColIndex
and the rows MidRowIndex, MidRowIndex - 1,
MidRowIndex + 1, as appropriate */*

if ($\text{MinRowIndex} == \text{MidRowIndex}$)

return $A[\text{MinRowIndex}][\text{MinColIndex}]$

else if ($\text{MinRowIndex} < \text{MidRowIndex}$)

$\text{BottomRowIndex} = \text{MidRowIndex} - 1$

else if ($\text{MinRowIndex} > \text{MidRowIndex}$)

$\text{TopRowIndex} = \text{MidRowIndex} + 1$

end While

Local Minimum in a Two-Dim Array: Ex. 1

	0	1	2	3	4	5	6
0	30	19	18	40	16	45	13
1	43	14	15	12	25	34	17
2	24	1	32	33	31	36	11
3	44	6	48	46	39	27	8
4	29	20	49	26	28	22	7
5	38	4	47	5	10	23	3
6	42	41	37	2	9	35	21

Iteration 1

Use the function
FindMinRowIndexNeighborhood

Use the
FindMinColIndex
function

	0	1	2	3	4	5	6	
Top Row Index →	0	30	19	18	40	16	45	13
	1	43	14	15	12	25	34	17
	2	24	1	32	33	31	36	11
Mid Row Index →	3	44	6	48	46	39	27	8
	4	29	20	49	26	28	22	7
	5	38	4	47	5	10	23	3
Bottom Row Index →	6	42	41	37	2	9	35	21

Local Minimum in a Two-Dim Array: Ex. 1 (1)

Iteration 2

		0	1	2	3	4	5	6	
Top Row Index	→	0	30	19	18	40	16	45	13
		1	43	14	15	12	25	34	17
Bottom Row Index	→	2	24	1	32	33	31	36	11
		3	44	6	48	46	39	27	8
		4	29	20	49	26	28	22	7
		5	38	4	47	5	10	23	3
		6	42	41	37	2	9	35	21

		0	1	2	3	4	5	6	
Top Row Index	→	0	30	19	18	40	16	45	13
Mid Row Index	→	1	43	14	15	12	25	34	17
Bottom Row Index	→	2	24	1	32	33	31	36	11
The minimum element		3	44	6	48	46	39	27	8
12 in Mid Row is smaller		4	29	20	49	26	28	22	7
than its immediate top		5	38	4	47	5	10	23	3
(40) and bottom (33)		6	42	41	37	2	9	35	21

neighbors

12 at (1, 3) is a local minimum

Local Minimum in a Two-Dim Array: Ex. 2

	0	1	2	3	4	5
0	17	16	32	15	23	36
1	20	3	18	35	11	9
2	26	5	8	30	13	22
3	10	31	2	1	7	14
4	28	12	6	24	25	34
5	29	21	27	19	4	33

Iteration 1

	0	1	2	3	4	5	
Top Row Index →	0	17	16	32	15	23	36
	1	20	3	18	35	11	9
Mid Row Index →	2	26	5	8	30	13	22
	3	10	31	2	1	7	14
	4	28	12	6	24	25	34
Bottom Row Index →	5	29	21	27	19	4	33

Local Minimum in a Two-Dim Array: Ex. 2 (1)

Iteration 2

	0	1	2	3	4	5	
Top Row Index →	0	17	16	32	15	23	36
Bottom Row Index →	1	20	3	18	35	11	9
Mid Row Index →	2	26	5	8	30	13	22
	3	10	31	2	1	7	14
	4	28	12	6	24	25	34
	5	29	21	27	19	4	33

Mid Row Index		0	1	2	3	4	5
Top Row Index →	0	17	16	32	15	23	36
Bottom Row Index →	1	20	3	18	35	11	9
Bottom Row Index →	2	26	5	8	30	13	22
	3	10	31	2	1	7	14
	4	28	12	6	24	25	34
	5	29	21	27	19	4	33

The minimum element
15 in Mid Row is smaller
than its immediate top
bottom (35) neighbor

15 at (0, 3) is a local minimum

Applications of Binary Search (4)

Local Minimum in a Two-Dimensional Array

- Time Complexity Analysis

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

Time complexity to search for the minimum element in a row

The search space reduces by half

Let $N = n^2$.

$$T(N) = T(N/2) + \Theta(N^{1/2})$$

Use Master Theorem: $a = 1$, $b = 2$, $d = 1/2$

We have $a < b^d$. Hence, $T(N) = \Theta(N^{1/2}) = \Theta(n)$

Space Complexity: $\Theta(1)$