Module 2:
Divide and Conquer

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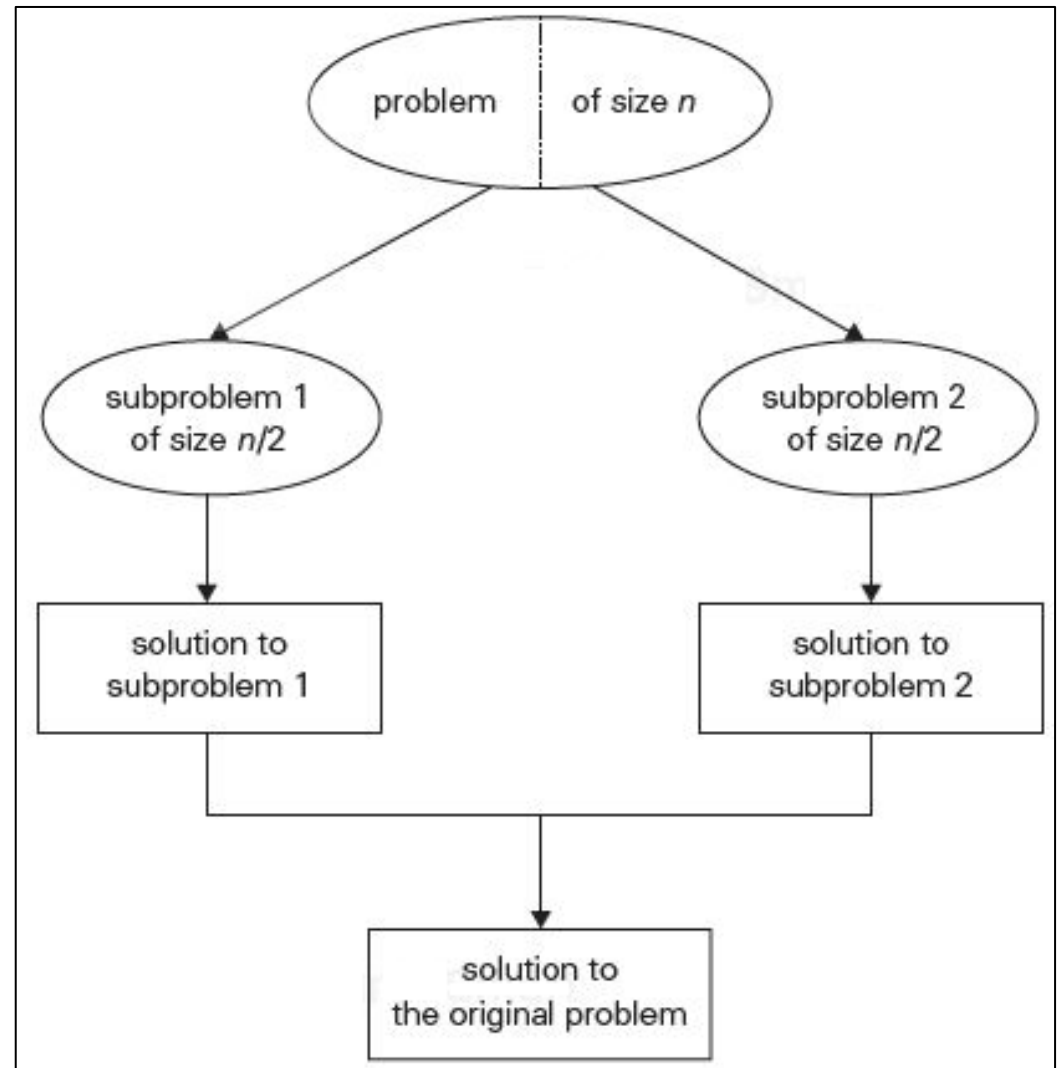Jackson, MS 39217
E-mail: natarajan.meghanathan@jsums.edu

Divide-and-Conquer

The most-well known algorithm design strategy:

1. We divide a problem of instance size 'n' into several sub problems (each of size n/b);

2. Solve 'a' of these sub problems (a ≥ 1; b > 1) recursively and

3. Combine the solutions to these sub problems to obtain a solution for the larger problem.

Typical Case of Divide and Conquer Problems

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.

Master Theorem to Solve Recurrence Relations

- Assuming that size n is a power of b to simplify analysis, we have the following recurrence for the running time, $T(n) = a\,T(n/b) + f(n)$
  - where f(n) is a function that accounts for the time spent on dividing an instance of size n into instances of size n/b and combining their solutions.

- Master Theorem:

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

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

The same results hold good for O and Ω too.

Examples:

1) $T(n) = 4T(n/2) + n$
a = 4; b = 2; d = 1 ➔ a > b^d
$$T(n) = \Theta\!\left(n^{\log_2 4}\right) = \Theta(n^2)$$

2) $T(n) = 4T(n/2) + n^2$
a = 4; b = 2; d = 2 ➔ a = b^d
$$T(n) = \Theta\!\left(n^2 \log n\right)$$

3) $T(n) = 4T(n/2) + n^3$
a = 4; b = 2; d = 3 ➔ a < b^d
$$T(n) = \Theta\!\left(n^3\right)$$

4) $T(n) = 2T(n/2) + 1$
a = 2; b = 2; d = 0 ➔ a > b^d
$$T(n) = \Theta\!\left(n^{\log_2 2}\right) = \Theta(n)$$

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 $B[i] \leq C[j]$
        $A[k] \leftarrow B[i]; \ i \leftarrow i+1$
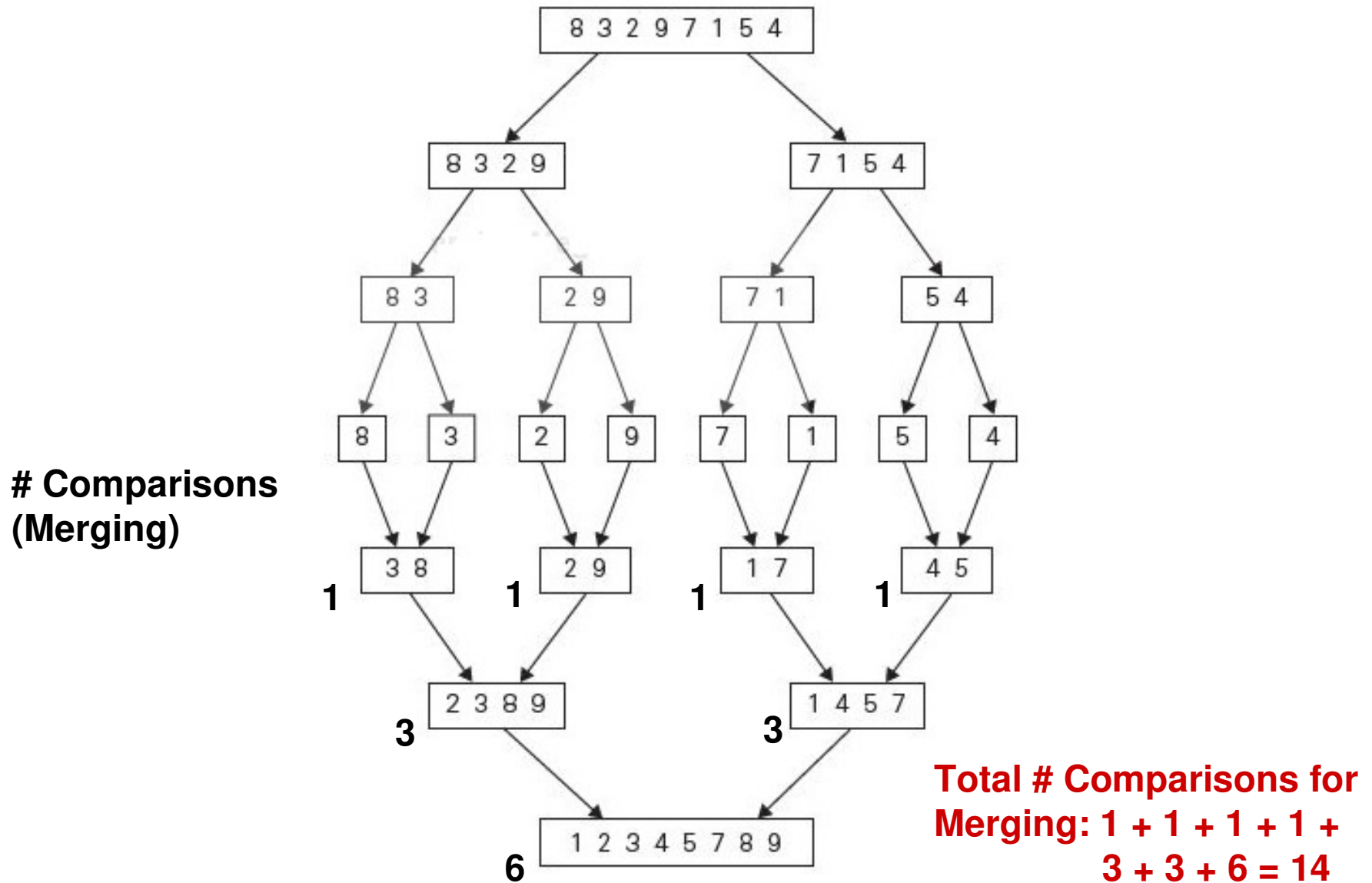   **else** $A[k] \leftarrow C[j]; \ j \leftarrow j+1$
    $k \leftarrow k+1$
if $i = p$
    copy $C[j..q-1]$ to $A[k..p+q-1]$
else copy $B[i..p-1]$ to $A[k..p+q-1]$

Example for Merge Sort

 # Comparisons (Merging)

8 3 2 9 7 1 5 4

8 3 2 9   7 1 5 4

8 3   2 9   7 1   5 4

8   3   2   9   7   1   5   4

3 8   2 9   1 7   4 5

1   **1**   **1**   **1**

2 3 8 9   1 4 5 7

3   **3**

1 2 3 4 5 7 8 9

6

Total # Comparisons for Merging: 1 + 1 + 1 + 1 + 3 + 3 + 6 = 14

Analysis of Merge Sort

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

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

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. 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 $C_{merge}(n) = n - 1$ for n > 1.

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

Hence, according to Master Theorem,

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

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

$$a = b^d$$

Binary Search

- Binary search is a Θ(log n), highly efficient search algorithm, <u>in a sorted array</u>.

- 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 in based on a recursive idea, it can be easily implemented as a non-recursive algorithm.

$$K$$
$$\updownarrow$$
$$\underbrace{A[0]\dots A[m-1]}_{\substack{\text{search here if}\\ K<A[m]}} \quad A[m] \quad \underbrace{A[m+1]\dots A[n-1]}_{\substack{\text{search here if}\\ K>A[m]}}$$

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

Worst-case # Key Comparisons

$$C_{worst}(n) = C_{worst}(\lfloor n/2 \rfloor) + \Theta(1) \quad \text{for } n > 1, \quad C_{worst}(1) = 1$$

$$C_{worst}(2^k) = k + 1 = \log_2 n + 1.$$

$$C_{worst}(n) = \Theta(\log n)$$

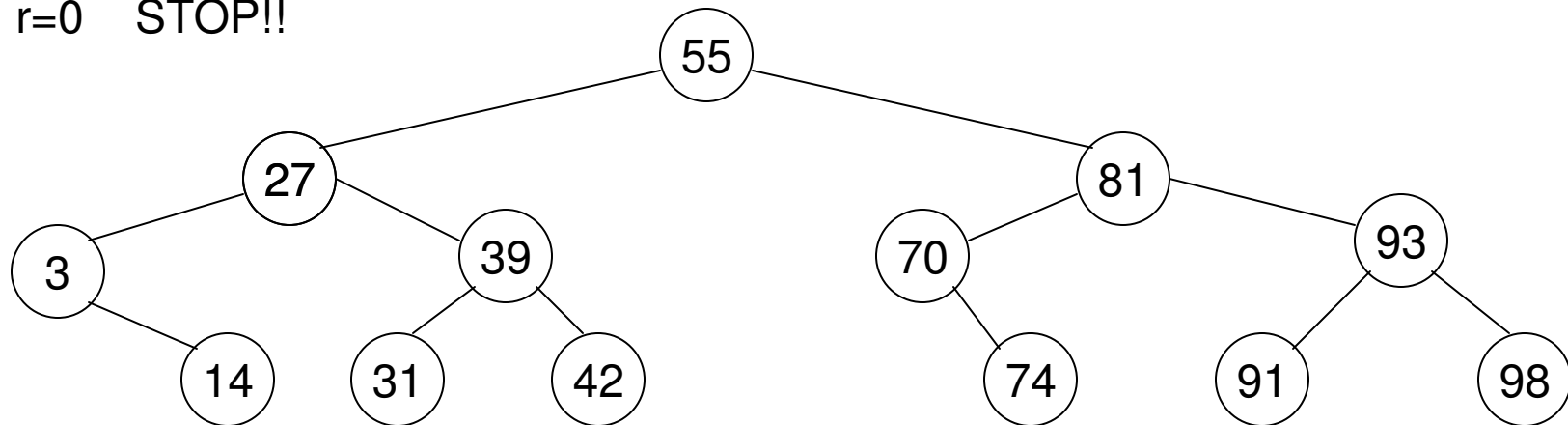**Unsuccessful Search**

Search K = 10
l=0 r=12 m=6
l=0 r=5   m=2
l=0 r=1   m=0
l=1 r=1   m=1
l=1 r=0   STOP!!

| 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*				

The keys that will require the largest number of comparisons: 14, 31, 42, 74, 91, 98

Average # Comparisons for Successful Search

Keys	# comparisons
55	1
27, 81	2
3, 39, 70, 93	3
14, 31, 42, 74, 91, 98	4

Avg # comparisons
= [Sum of the product of the # keys with certain # comparisons] / [Total Number of keys]
= [(1)(1) + (2)(2) + (3)(4) + (4)(6)] /13
= 3.15
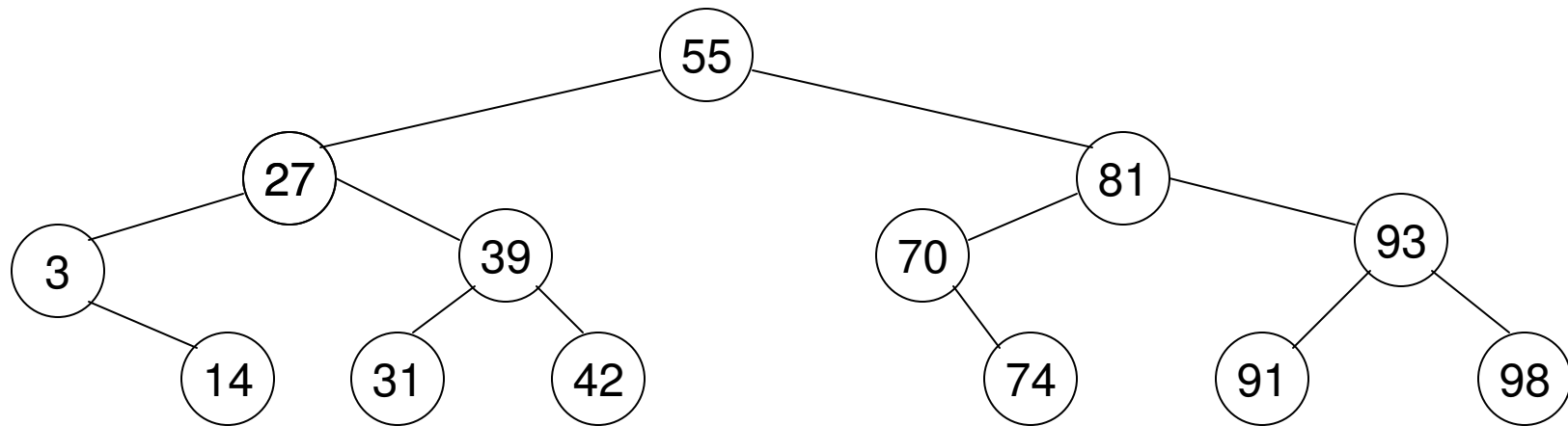
Average # Comparisons for Unsuccessful Search

Range of Keys for Unsuccessful search	# comparisons
< 3	3
> 3 and < 14	4
> 14 and < 27	4
> 27 and < 31	4
> 31 and < 39	4
> 39 and < 42	4
> 42 and < 55	4
> 55 and < 70	3
> 70 and < 74	4
> 74 and < 81	4
> 81 and < 91	4
> 91 and < 93	4
> 93 and < 98	4
> 98	4

$$Avg = [4*12 + 3*2] / 14$$
$$= 3.86$$

Applications of Binary Search (1)
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 (1)

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) = Θ(logn)

Space complexity: Θ(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]
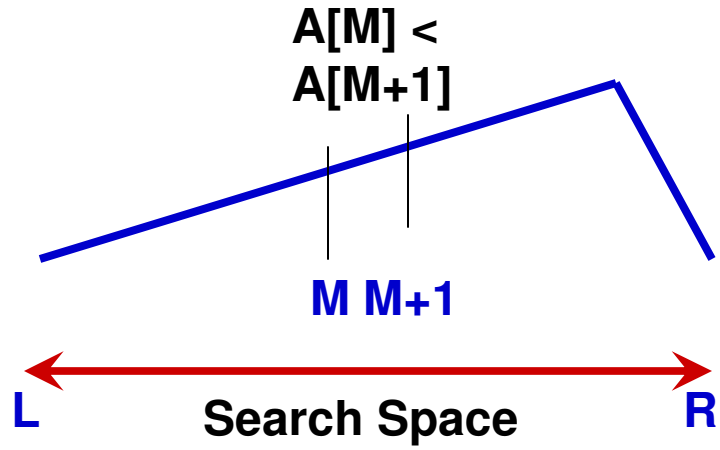L = 5; R = 9; m = 7: A[m] > A[m+1]
L = 5; R = 7; m = 6: A[m] > A[m+1]
L = 5; R = 6; m = 5: A[m] > A[m+1]
L = 5; R = 5; return A[5] = 14

Two Scenarios
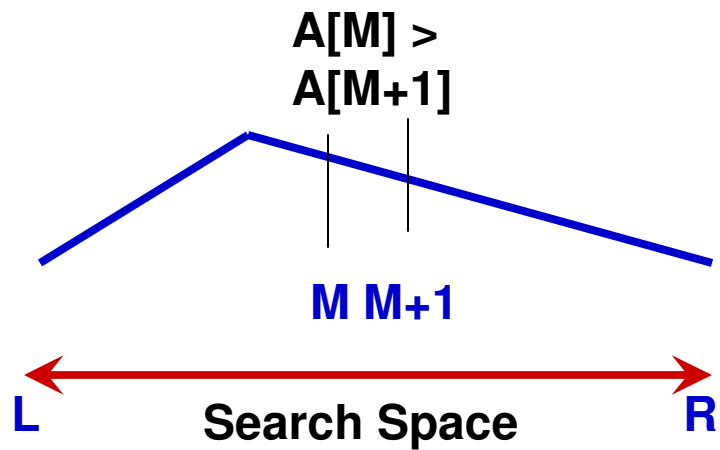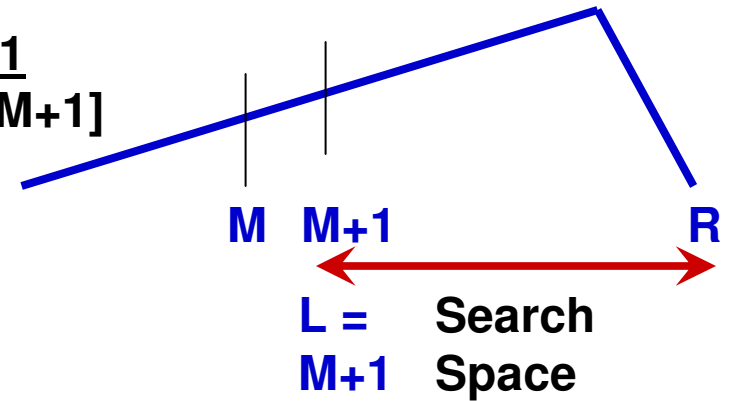
A[M] < A[M+1]

M M+1

L Search Space **R**

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

M M+1 **R**

L = M+1 Search Space

A[M] > A[M+1]

M M+1

L Search Space **R**

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

A[M] > A[M+1]

M M+1

L **R = M**
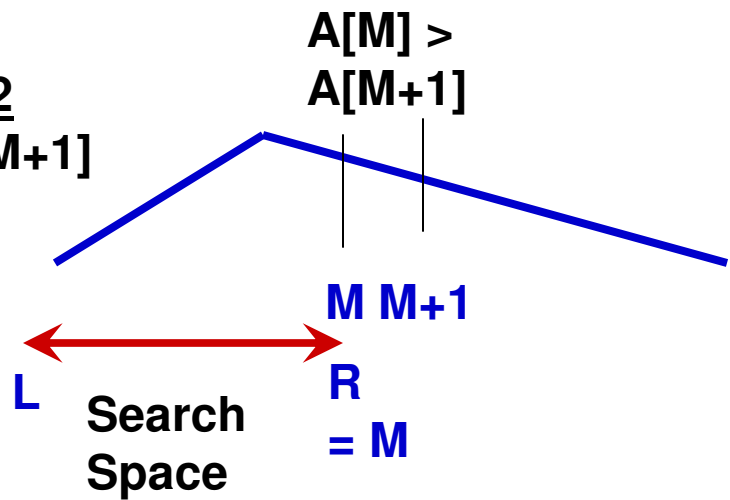
Search Space

Applications of Binary Search (1)
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…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…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...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 (1)
Finding the Maximum Element in a Unimodal Array

```
         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

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]

Applications of Binary Search (2)
Local Minimum in an Array

- <u>Problem:</u> Given an array A[0,…, 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.

- <u>Constraints:</u>
  - The arrays has at least three elements
  - The first two numbers are decreasing and the last two numbers are increasing.
  - The numbers are unique

- <u>Example:</u>
  - Let A = {8, **5**, 7, **2**, 3, 4, **1**, 9}; the array has several local minimum. These are: 5, 2 and 1.

- <u>Algorithm:</u> 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 (2)
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 (2)
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: $T(3) = 2$ $\longrightarrow$ One comparison for A[M] with A[M+1]
Using Master Theorem, we have $\longrightarrow$ One comparison for A[M] with A[M-1]
$a = 1, b = 2, d = 0$ ➜ $a = b^d$. $\longrightarrow$ One comparison for A[M-1] with A[M+1]
Hence, $T(n) = \Theta(n^d \log n) = \Theta(n^0 \log n) = \Theta(\log n)$

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 $\Theta(1)$.
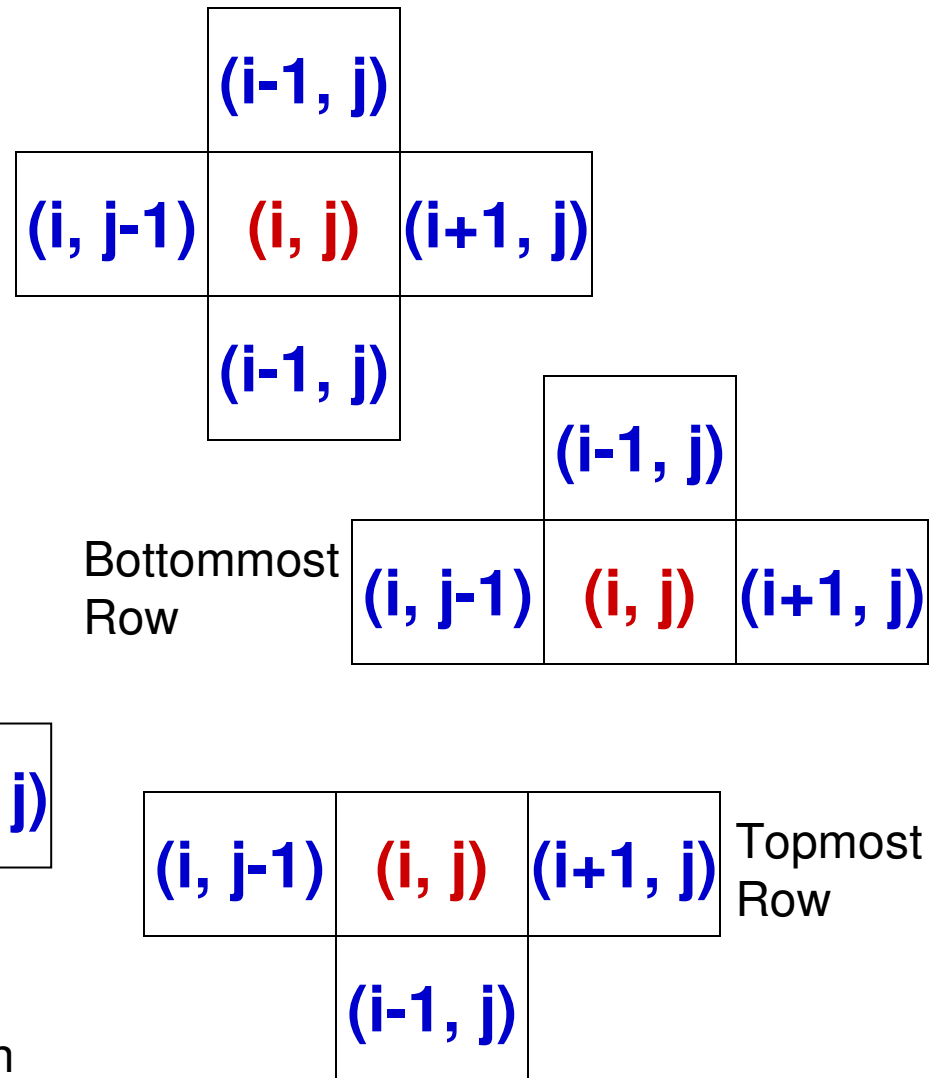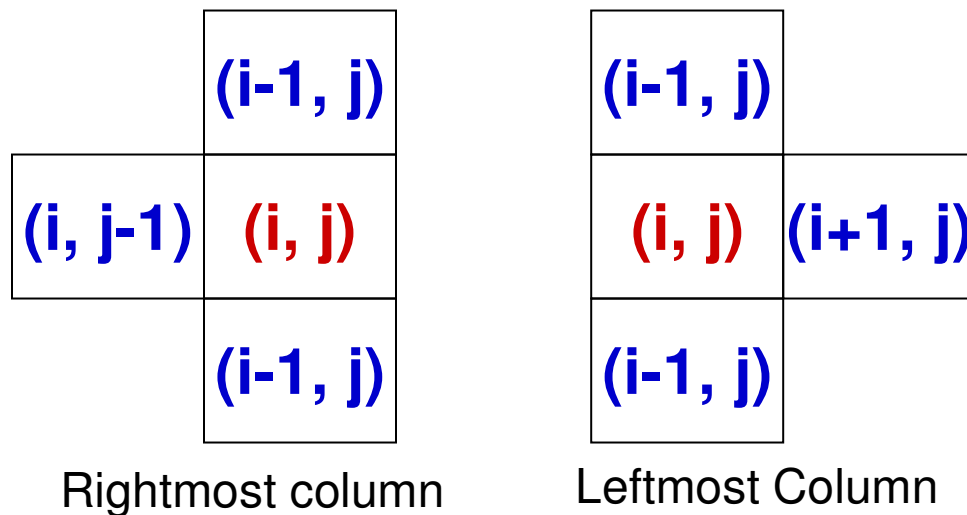
Applications of Binary Search (2)
Local Minimum in an Array

- Constraints:
  - The arrays 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 (3)
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.

(i-1, j)

(i, j-1) (i, j) (i+1, j)

(i-1, j)

Bottommost Row

(i-1, j)

(i, j-1) (i, j) (i+1, j)

(i-1, j)

(i, j-1) (i, j)

(i-1, j)

Rightmost column

(i-1, j)

(i, j) (i+1, j)

(i-1, j)

Leftmost Column

(i, j-1) (i, j) (i+1, j) Topmost Row

(i-1, j)

Applications of Binary Search (3)
Local Minimum in a Two-Dimensional Array

Given an array A[0…numRows-1][0…numCols-1]
TopRowIndex = 0
BottomRowIndex = numRows – 1
while (TopRowIndex ≤ BottomRowIndex) do
    MidRowIndex = (TopRowIndex + BottomRowIndex) / 2
    MinColIndex = FindMinColIndex(A[MidRowIndex][ ])

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

    MinRowIndex = FindMinRowIndexNeighborhood (A, MidRowIndex,
                                    MinColIndex)

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

    if (MinRowIndex == MidRowIndex)
        return A[MinRowIndex][MinColIndex]
    else if (MinRowIndex < MidRowIndex)
        BottomRowIndex = MidRowIndex – 1
    else if (MinRowIndex > MidRowIndex)
        TopRowIndex = 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
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

The minimum element 12 in Mid Row is smaller than its immediate top (40) and bottom (33) 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

	0	1	2	3	4	5
Mid Row Index / **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 (3)
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: $a = 1, b = 2, d = \frac{1}{2}$

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

Space Complexity: $\Theta(1)$

Proof of Correctness

- We will prove by contradiction.
- Assume the local minimum is not in the top half (as well as in the bottom half) and not in the middle row either.
- If the local minimum is not in the middle row and there is an element in the immediate top row of the middle row that is less than the minimum element in the middle row, then we move the search space to the top half (or likewise to the bottom half).
- If there is no local minimum in the top half, then for every row in the top half: for the minimum element in this row, there is an element that is lower than it in the immediate top row (recursively starting from the row above the middle row) or the immediate bottom row.
  - This implies, there should be an element above the topmost row that is less than the minimum element in the topmost row (or) there is an element in the initial middle row that is lower than the element in the immediate top row.
  - Such a row (that is above the topmost row or that is the initial middle row) does not exist. (A contradiction)
  - Hence, there should be some element in the top half (or the bottom half) that should be a local minimum, if a local minimum does not exist in the middle row.