Module 3: Binary Search

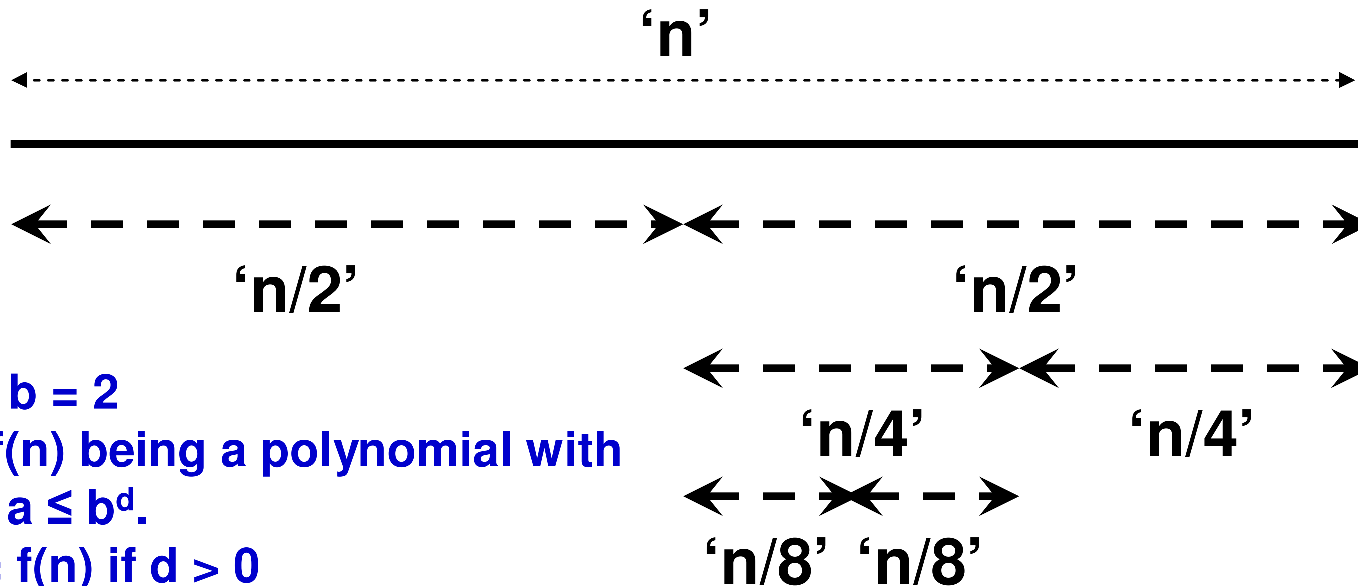
Dr. Natarajan Meghanathan
Professor of Computer Science
Jackson State University
Jackson, MS 39217
E-mail: natarajan.meghanathan@jsums.edu

Binary Search

- Binary search is an approach in which we reduce the problem size (also referred to as the search space) by half in each iteration and we would have found the solution by the time the search space/problem size narrows down to a threshold.

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

where f(n) is the time spent to decide which of the two sub halves (each of size n/2) of the problem (of size n) to choose.



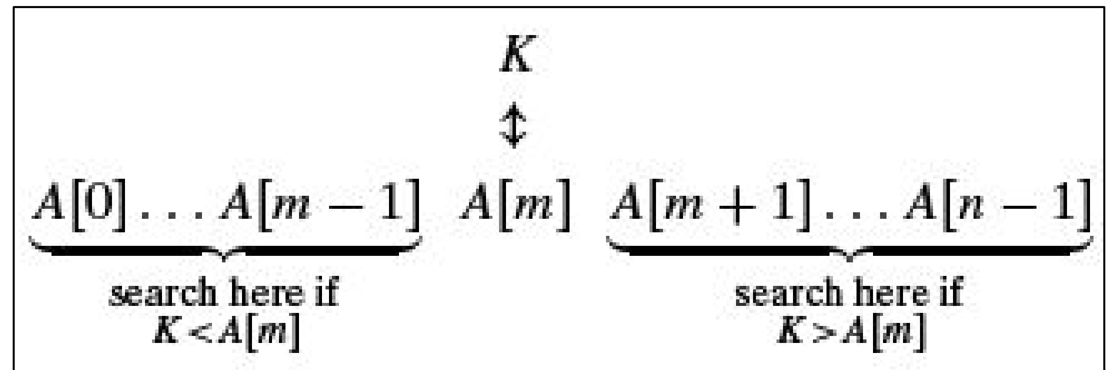
$a = 1; b = 2$
With f(n) being a polynomial with $d \geq 0$, $a \leq b^d$.
$T(n) = f(n)$ if $d > 0$
$T(n) = \Theta(\log n)$ if $d = 0$

Types of Binary Search Problems

- Classical binary search: Searching for a key in a sorted array.

Binary Search

3.1 Searching for a Key in a sorted array



- Binary search is a Θ(log n) algorithm
  - **the array needs to be sorted**.
- Working Principle
  - **Define a range of indices, left index to right index, within which the search key could be there.**
    - For an array, left index = 0, right index = n-1
  - Run Iterations: In each iteration
    - find the middle index = (left index + right index) / 2
- 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.

3.1 Searching for a Key in a Sorted Array

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 ← 0;   r ← n − 1
    while l ≤ r do
        m ← ⌊(l + r)/2⌋
        if K = A[m] return m
        else if K < A[m] r ← m − 1
        else l ← m + 1
    return −1
```

$C(n) = C(n/2) + 2$ for $n > 1$
$C(1) = 1$

$C(n) = C(n/2) + \Theta(1)$ for $n > 1$
$a = 1, b = 2, d = 0$
$a = b^d$
$C(n) = \Theta(n^0 \log n) = \Theta(\log n)$

3.1 Searching for a Key in a Sorted Array

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*											

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!!

3.2 Searching for a Threshold Value for a Monotonically Increasing or Decreasing 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

- <u>Solution Approach (for optimization problems)</u>
- **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(Middle Index) < target, we set: Right Index = Middle Index**
    - **If f(Middle Index) >= target, we set: Left Index = 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.

$$\text{\# Iterations} = \log_2 \frac{\text{Initial Right Index - Initial Left Index}}{\text{Min. Allowable Diff. between the Right Index and Left Index}}$$

Function $f(n) = 2/n^2$; for $n > 0$
Target = 0.01

Left Index = 1; f(Left Index) = 2 $>$ target
Right Index = 100; f(Right Index) = $2/100^2$ = 0.0002 $<$ target

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

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

Iterations = log2 ((100-1) / 1) = log2(99) ~ 7

Solution:

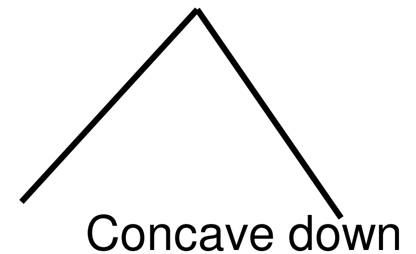
15 is the smallest integer for which the function $f(n) = 2/n^2$ is less than 0.01

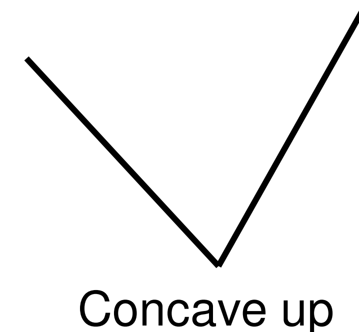
Practice Problem

- Given a monotonically increasing function $f(n) = n^2/10000$ (where 'n' is an integer), use a binary search algorithm to find the largest value of 'n' for which f(n) is less than a target (say, 0.01).

3.3 Finding the Maximum or Minimum in a Unimodal Array

- A (maximum or a concave down) unimodal array is an array that has a sequence of monotonically increasing integers followed by a sequence of monotonically decreasing integers.
- A (minimum or a concave up) unimodal array is an array that has a sequence of monotonically increasing integers followed by a sequence of monotonically decreasing integers.
- All elements in a unimodal array are unique
- Examples for (maximum/concave down) unimodal array.
  - {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.
- Examples for (minimum/concave up) unimodal array.
  - {9, 8, 5, 4, 7, 6, 12, 14}: Min. Element: 4
    - There is an increasing seq. followed by a decreasing seq.
  - {11, 9, 8, 7, 5, 4, 3, 2, 1}: Min. Element: 1
    - There is no increasing seq. It is simply a decreasing seq.
  - {1, 2, 3, 4, 5, 7, 8, 9, 11}: Min. Element: 1
    - There is an increasing seq., but there is no decreasing seq.





3.3 Finding the Maximum Element in a Concave Down Unimodal Array

Search Range: 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]
```

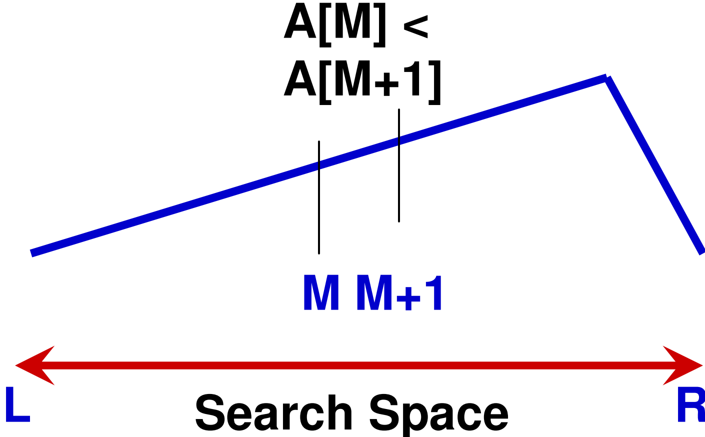
$C(n) = C(n/2) + 2$
Using Master Theorem,
$C(n) = \Theta(\log n)$

Space complexity: $\Theta(1)$

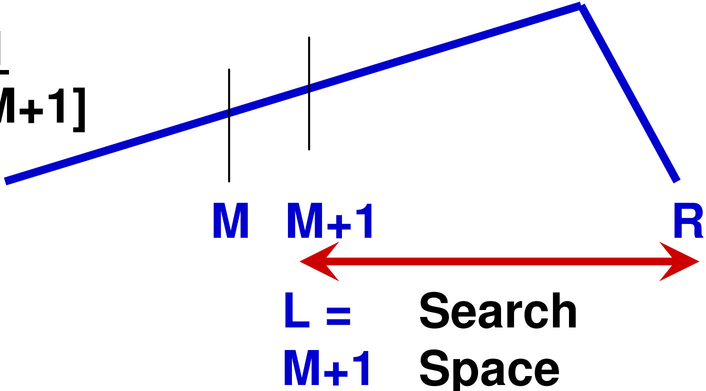
0	1	2	3	4	5	6	7	8	9
3	**5**	**8**	**9**	**10**	**14**	**11**	**4**	**2**	**1**

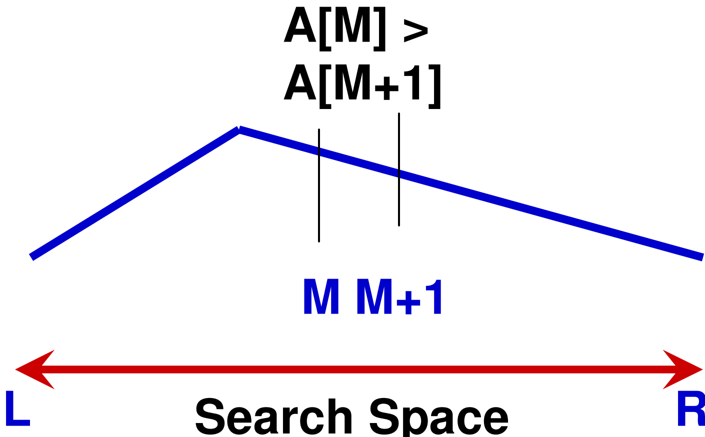
L = 0; R = 9; m = 4: A[m] < A[m+1]
L = 5; R = 9; m = 7: A[m] > A[m+1]
L = 5; R = 7; m = 6: A[m] > A[m+1]
L = 5; R = 6; m = 5: A[m] > A[m+1]
L = 5; R = 5; return A[5] = 14

Two Scenarios

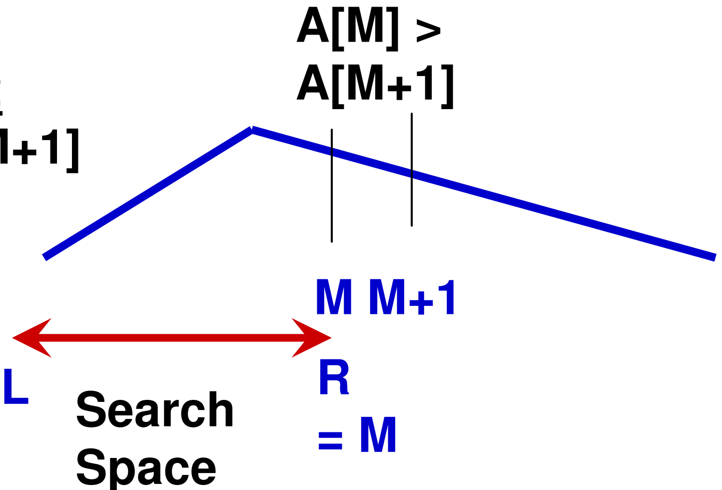


Scenario 1
A[M] < A[M+1]





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



3.3 Finding the Maximum Element in a Concave Down 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.

3.3 Finding the Maximum Element in a Concave Down 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

Practice Problem

- Design a binary search algorithm to find the minimum element in a concave up unimodal array. Run the algorithm on the following array and find the minimum.

  {9, 8, 5, 4, 7, 6, 12, 14}

3.4 Local Minimum in One-Dimension and Two-Dimension Arrays

- Problem: 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.
- 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
- Example:
  - Let A = {8, **5**, 7, **2**, 3, 4, **1**, 9}; the array has several local minimum. 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.

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.

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
- 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]

Basic Condition: T(3) = 2

Using Master Theorem, we have

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

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

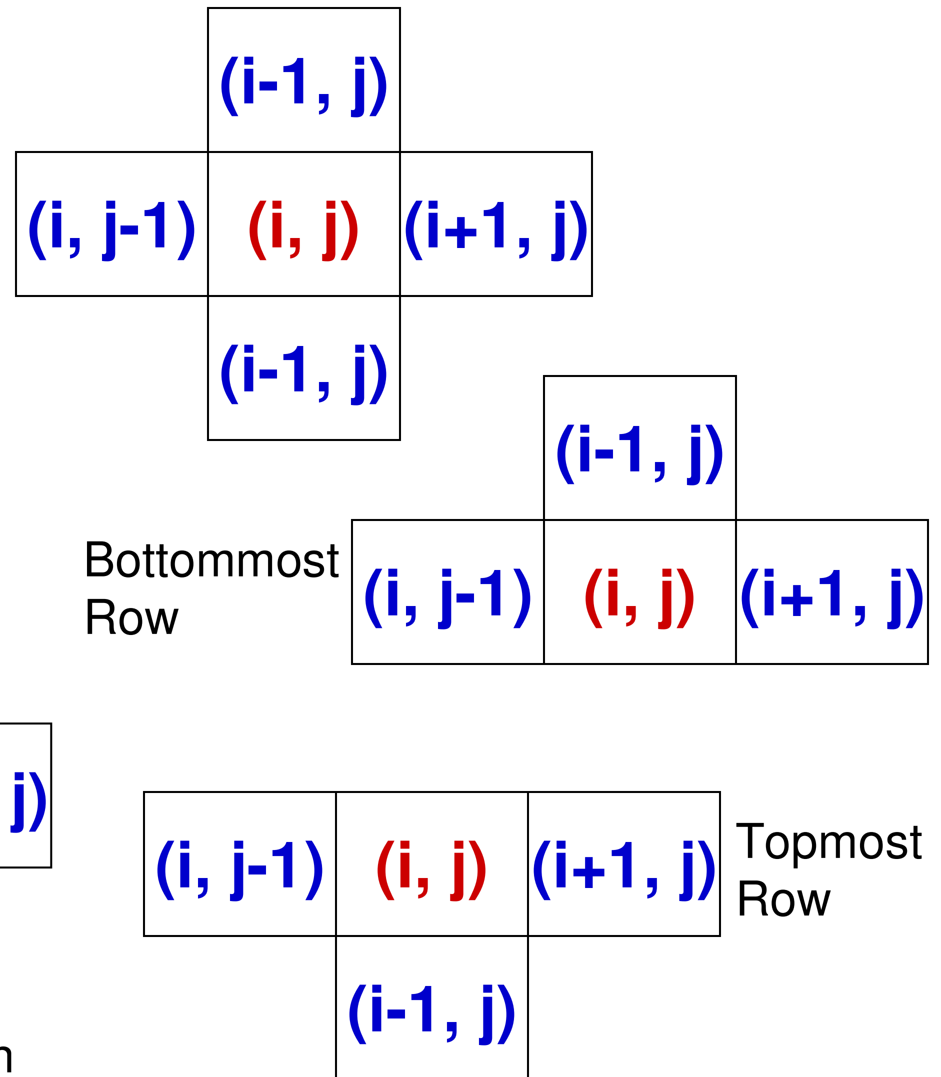
Space Complexity: As all evaluations are done on the input array itself, no extra space proportional to the input is needed. Hence, space complexity is $\Theta(1)$.

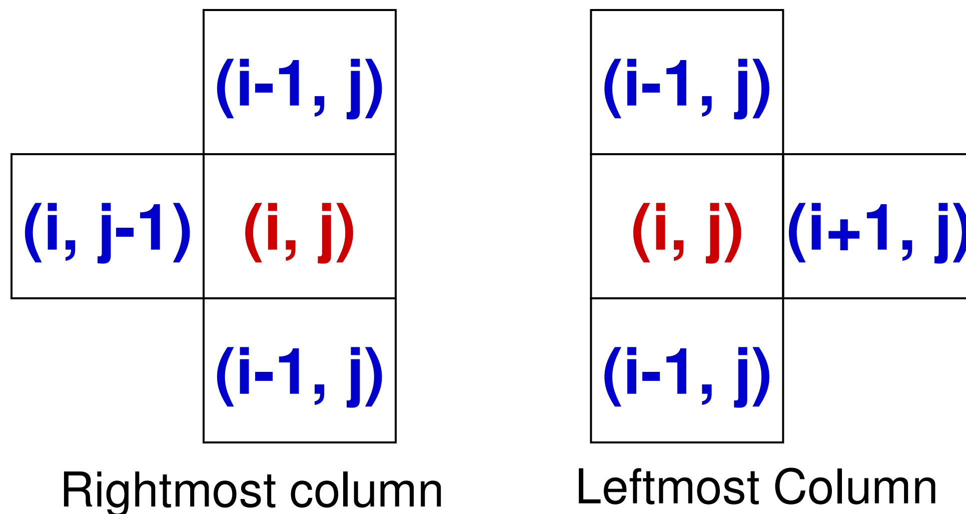
Local Minimum in an Array

- 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.

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.





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

Top Row Index → 0
Bottom Row Index → 2

	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

Top Row Index → 0
Mid Row Index → 1
Bottom Row Index → 2

	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

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

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: $\Theta(1)$