Module 2: Classical Algorithm Design Techniques

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

Module Topics

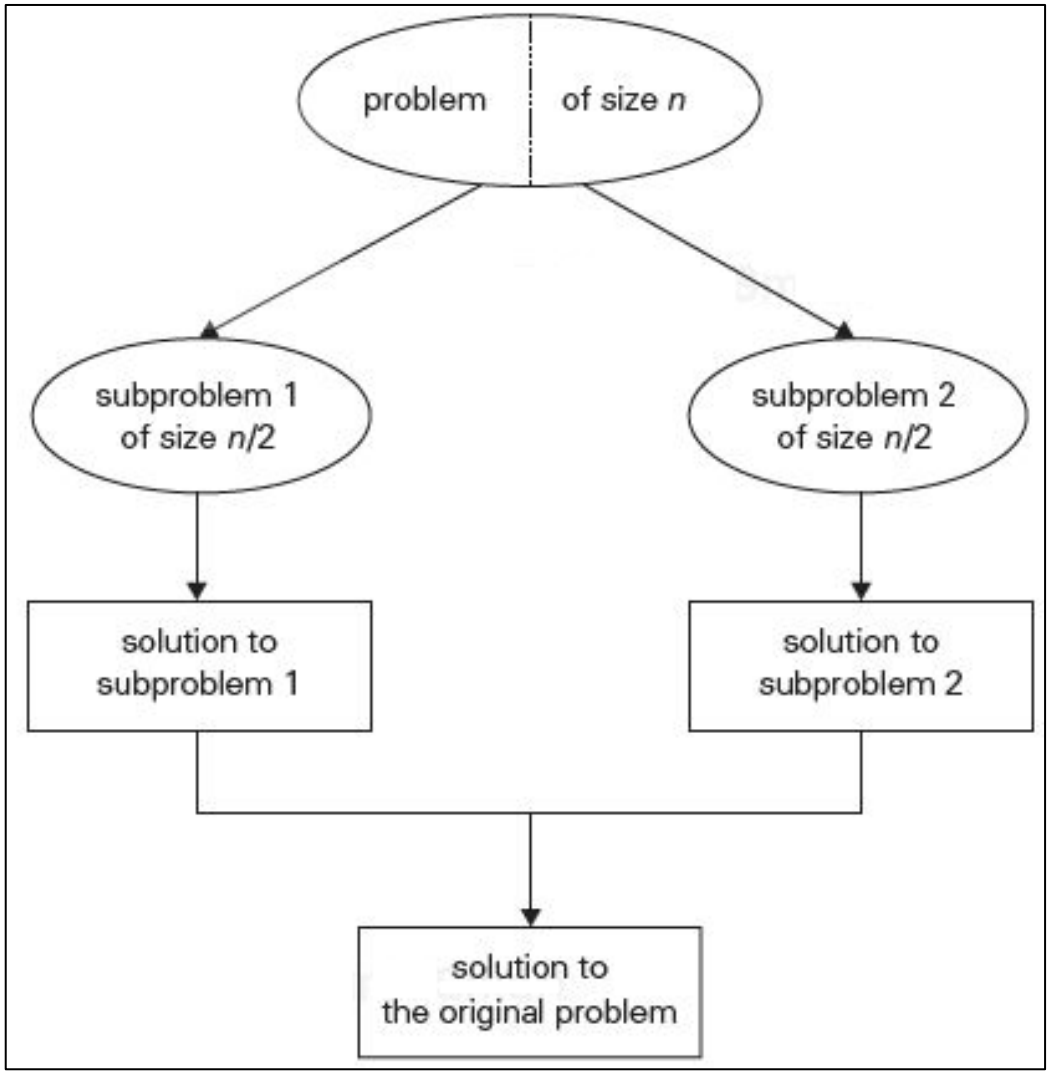
- 2.1 Divide and Conquer
- 2.2 Decrease and Conquer
- 2.3 Transform and Conquer
- 2.4 Space-Time Tradeoff: Sorting and Hashing

2.1 Divide and Conquer

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 ($a \geq 1$; $b > 1$) 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*$(A[0..n-1])$

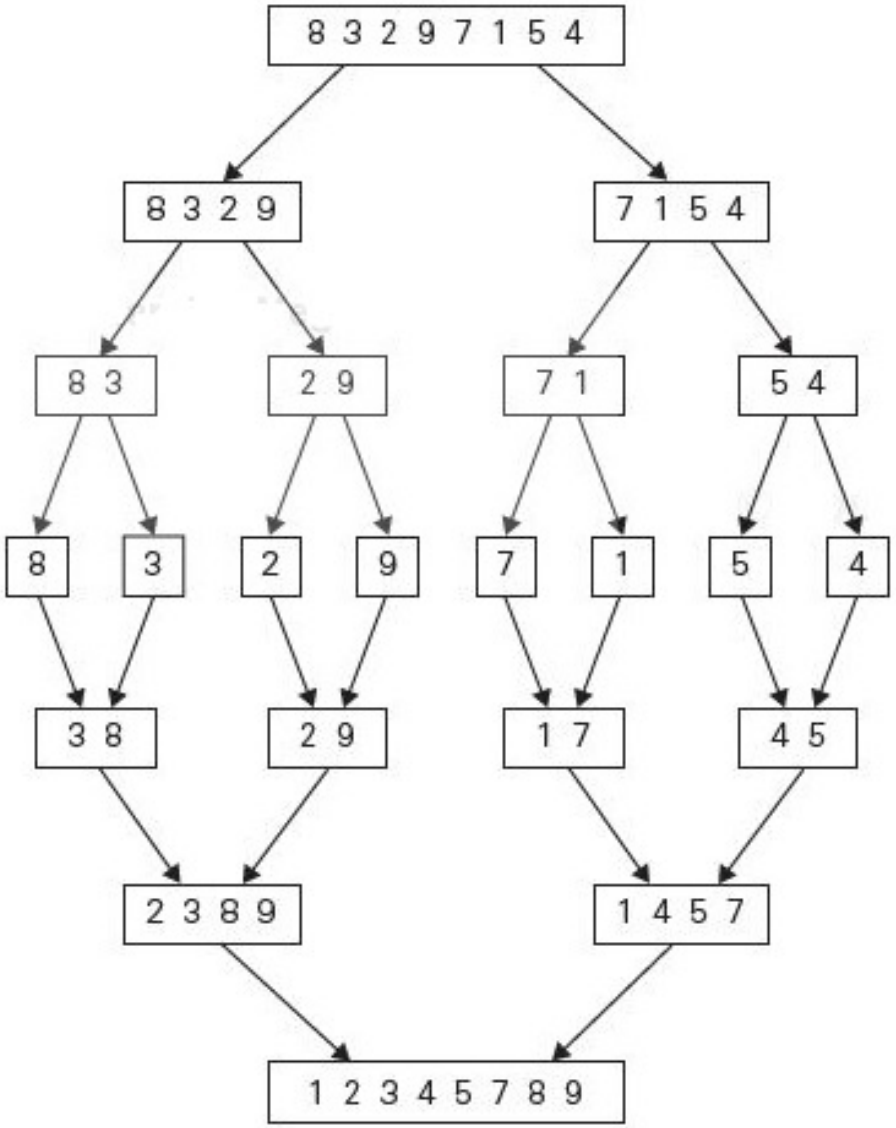
//Sorts array $A[0..n-1]$ by recursive mergesort
//Input: An array $A[0..n-1]$ of orderable elements
//Output: Array $A[0..n-1]$ sorted in nondecreasing order
if $n > 1$
    copy $A[0..\lfloor n/2 \rfloor - 1]$ to $B[0..\lfloor n/2 \rfloor - 1]$
    copy $A[\lfloor n/2 \rfloor..n-1]$ to $C[0..\lceil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



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. 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 $n > 1$.

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

Hence, according to Master Theorem,

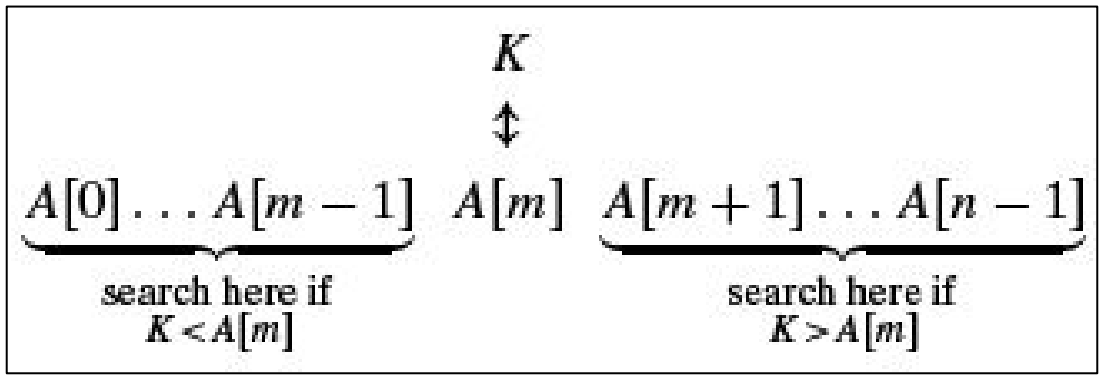
$$a = 2, b = 2, d = 1$$

$$a = b^d$$

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

Binary Search

- Binary search is a Θ(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 in 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	85	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
```

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

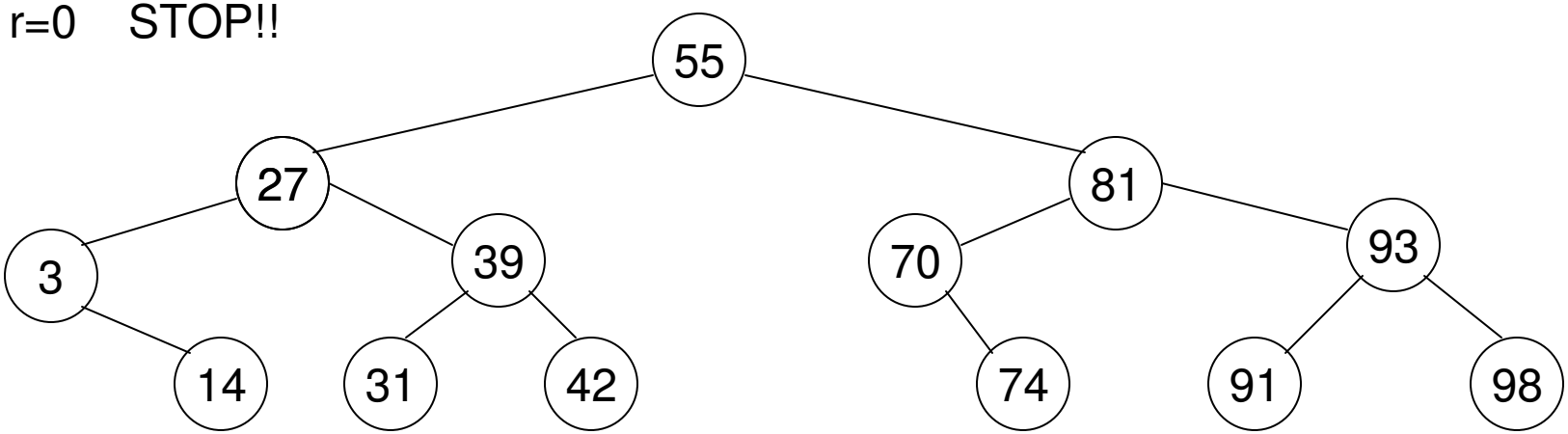
l=0 r=5 m=2

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	85	93	98
iteration 1	*l*						*m*						*r*
iteration 2								*l*		*m*			*r*
iteration 3								*l,m*	*r*				

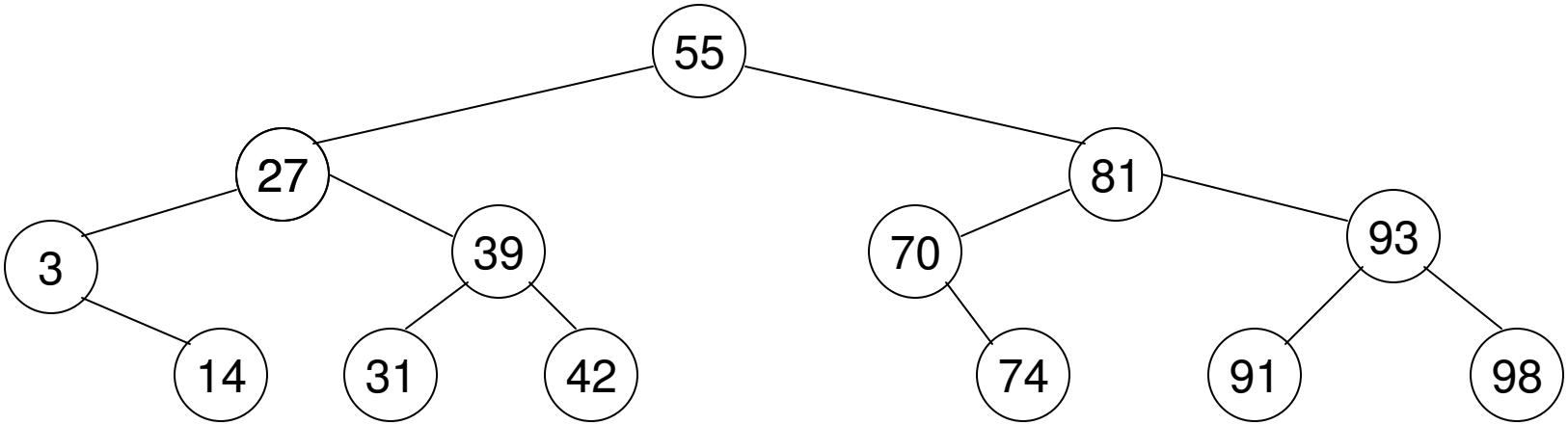


The keys that will require the largest number of comparisons: 14, 31, 42, 74, 91, 98

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) + 1
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

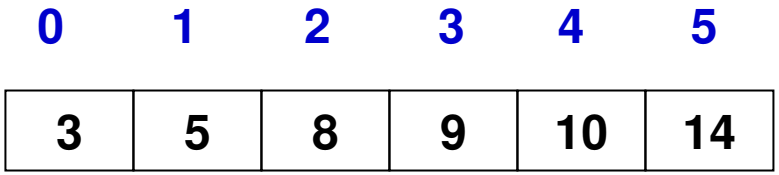
Applications of Binary Search (1)

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



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

- 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.

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: $T(n) = T(n/2) + 1$ for $n > 1$
Basic Condition: $T(1) = 1$
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 $\Theta(1)$.

Applications of Binary Search (2)

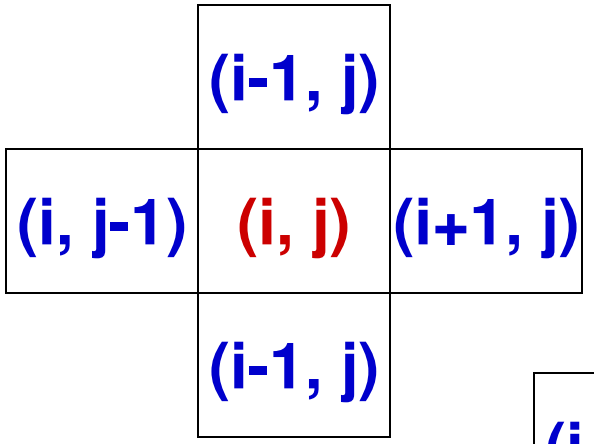
Local Minimum in an Array

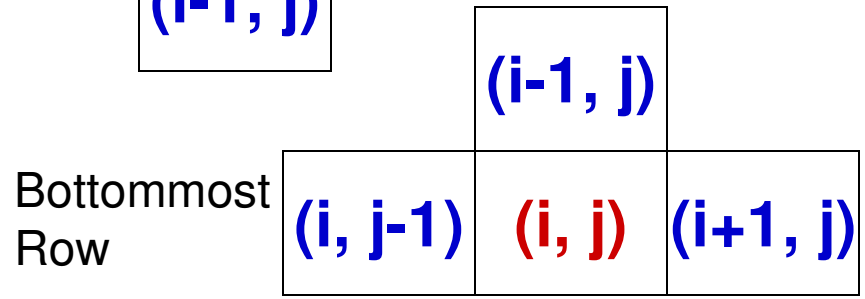
- Constraints:
  - The arrays has at least three elements
  - The first two numbers are decreasing and the last two numbers are increasing.
  - The numbers are unique
- Theorem: If the above three constraints are met for an array, then the array has to have at least one local minimum.
- Proof: Let us 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)

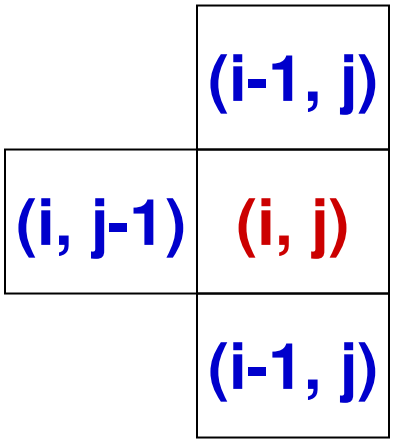
Local Minimum in a Two-Dimensional Array

- 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.

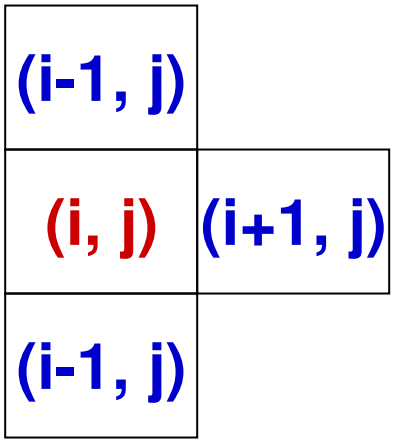




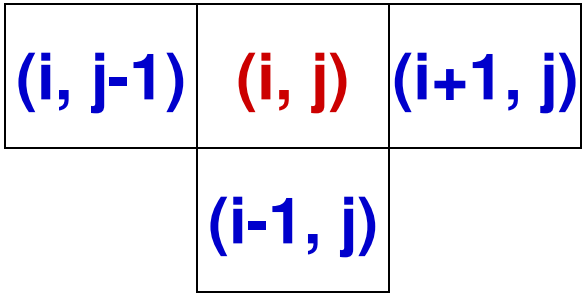
Bottommost Row



Rightmost column



Leftmost Column



Topmost Row

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

Top Row Index →

Mid Row Index →

Bottom Row Index →

	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

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

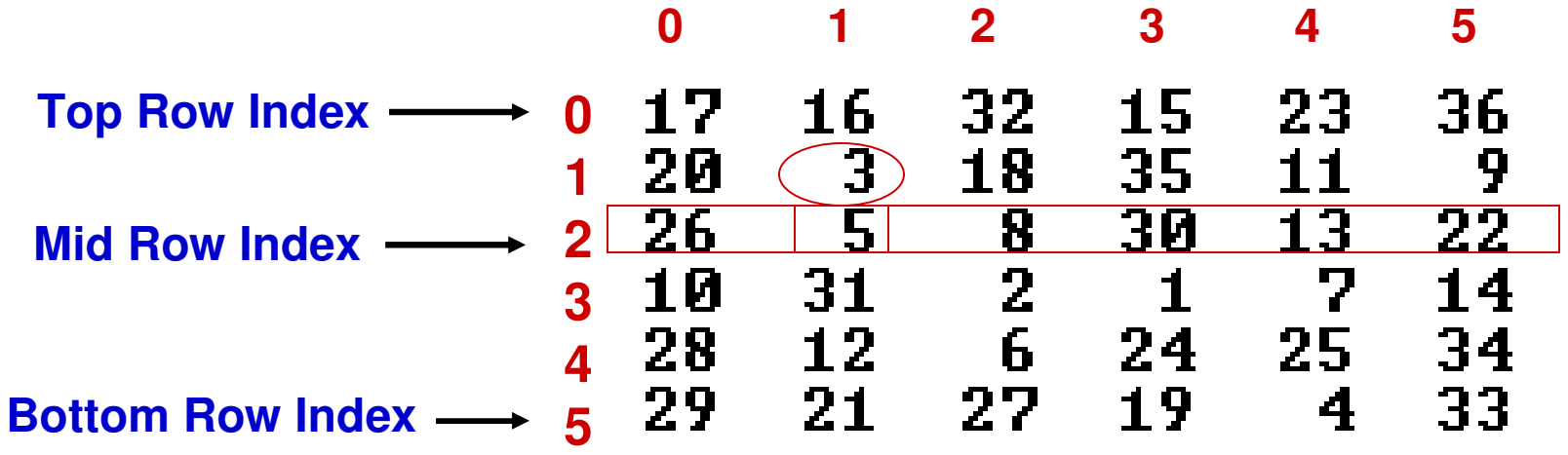
The minimum element 12 in Mid Row is smaller than its immediate top (40) and bottom (33) neighbors

12 at (1, 3) is a local minimum

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

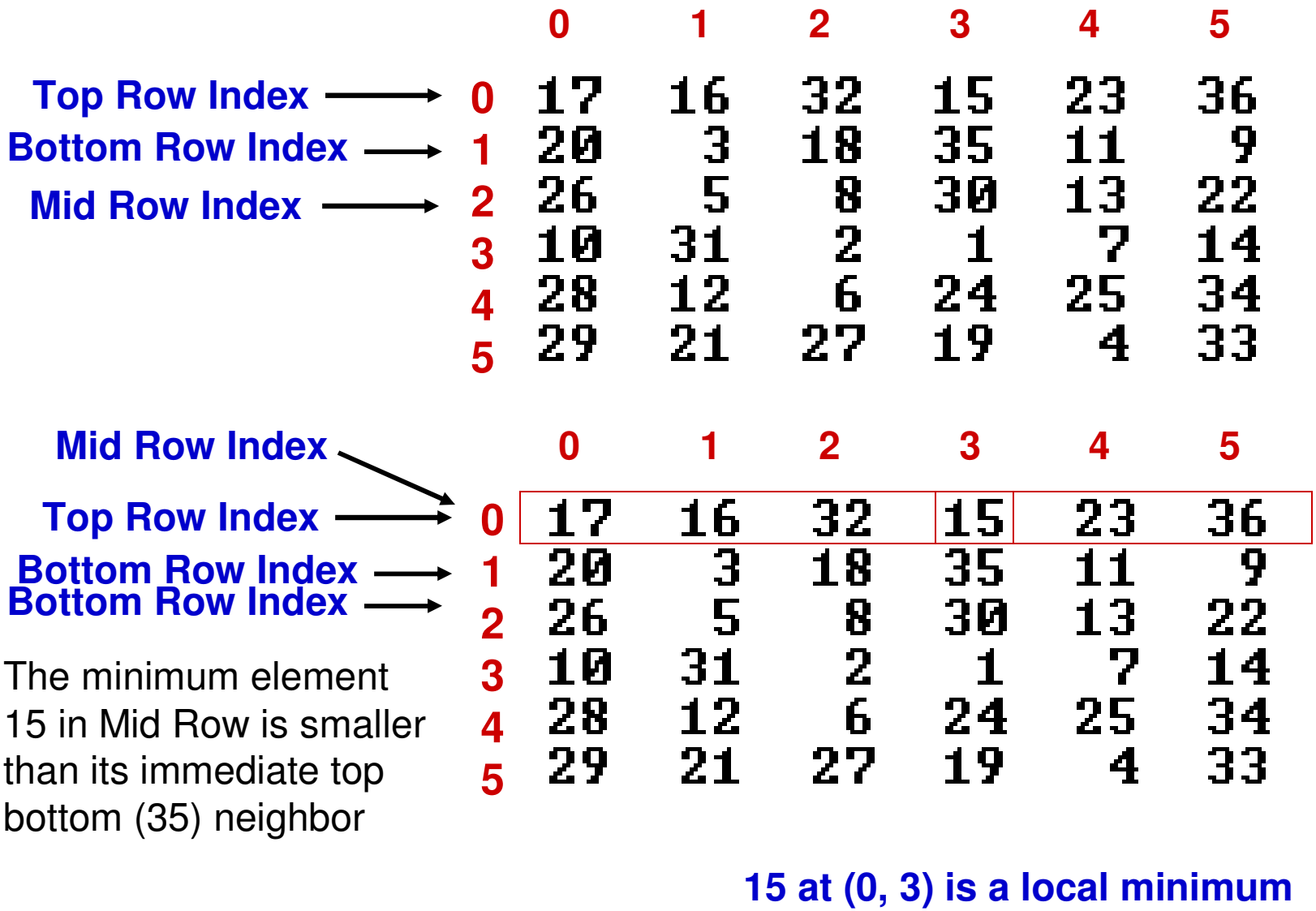
	0	1	2	3	4	5
0	17	16	32	15	23	36
1	20	3	18	35	11	9
2	26	5	8	30	13	22
3	10	31	2	1	7	14
4	28	12	6	24	25	34
5	29	21	27	19	4	33

Iteration 1



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

Iteration 2



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: $a = 1$, $b = 2$, $d = ½$

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

Space Complexity: $\Theta(1)$

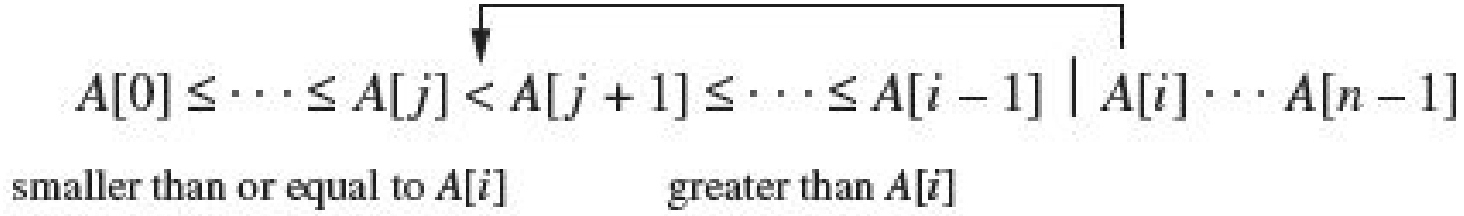
Applications of Binary Search (3)
Local Minimum in a Two-Dimensional Array

- 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).
    - This implies, there should be an element above the topmost row that is less than the minimum element in the topmost row.
    - Such a row (that is above the topmost 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.

2.2 Decrease and Conquer

Decrease by One: Insertion Sort

- Given an array A[0…n-1], at any time, we have the array divided into two parts: A[0,…,i-1] and A[i…n-1].
  - The A[0…i-1] is the sorted part and A[i…n-1] is the unsorted part.
  - In any iteration, we pick an element *v* = A[i] and scan through the sorted sequence A[0…i-1] to insert *v* at the appropriate position.
    - The scanning is proceeded from right to left (i.e., for index j running from i-1 to 0) until we find the right position for *v*.
    - During this scanning process, v = A[i] is compared with A[j].
    - If A[j] > v, then we v has to be placed somewhere before A[j] in the final sorted sequence. So, A[j] cannot be at its current position (in the final sorted sequence) and has to move at least one position to the right. So, we copy A[j] to A[j+1] and decrement the index j, so that we now compare v with the next element to the left.



    - If A[j] ≤ v, we have found the right position for v; we copy v to A[j+1]. This also provides the stable property, in case v = A[j].

Insertion Sort
Pseudo Code and Analysis

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

```
//Sorts a given array by insertion sort
//Input: An array A[0..n − 1] of n orderable elements
//Output: Array A[0..n − 1] sorted in nondecreasing order
for i ← 1 to n − 1 do
    v ← A[i]
    j ← i − 1
    while j ≥ 0 and A[j] > v do
        A[j + 1] ← A[j]
        j ← j − 1
    A[j + 1] ← v
```

Best Case (if the array is already sorted): the element v at A[i] will be just compared with A[i-1] and since A[i-1] ≤ A[i] = v, we retain v at A[i] itself and do not scan the rest of the sequence A[0…i-1]. There is only one comparison for each value of index i.

$$\sum_{i=1}^{n-1} 1 = (n-1) - 1 + 1 = (n-1)$$

The comparison A[j] > v is the basic operation.

Worst Case (if the array is reverse-sorted): the element v at A[i] has to be moved all the way to index 0, by scanning through the entire sequence A[0…i-1].

$$\sum_{i=1}^{n-1} \sum_{j=i-1}^{0} 1 = \sum_{i=1}^{n-1} \sum_{j=0}^{i-1} 1 = \sum_{i=1}^{n-1} (i-1) - 0 + 1 = \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2}$$

Insertion Sort: Analysis and Example

Average Case: On average for a random input sequence, we would be visiting half of the sorted sequence A[0…i-1] to put A[i] at the proper position.

$$C(n) = \sum_{i=1}^{n-1} \sum_{j=i-1}^{(i-1)/2} 1 = \sum_{i=1}^{n-1} \frac{(i-1)}{2} + 1 = \sum_{i=1}^{n-1} \frac{(i+1)}{2} = \Theta(n^2)$$

Example: Given sequence (also initial): **45** 23 8 12 90 21

Index -1

Iteration 1 (v = 23):

45	**45**	8	12	90	21
23	**45**	8	12	90	21

Iteration 2 (v = 8):

23	**45**	**45**	12	90	21
23	**23**	**45**	12	90	21
8	**23**	**45**	12	90	21

Iteration 3 (v = 12):

8	**23**	**45**	**45**	90	21
8	**23**	**23**	**45**	90	21
8	**12**	**23**	**45**	90	21

Iteration 4 (v = 90):

8	**12**	**23**	**45**	90	21
9	**12**	**23**	**45**	**90**	21

Iteration 5 (v = 21):

9	**12**	**23**	**45**	**90**	**90**
9	**12**	**23**	**45**	**45**	**90**
9	**12**	**23**	**23**	**45**	**90**
9	**12**	**21**	**23**	**45**	**90**

Overall time complexity

$$\lim_{n->\infty} \frac{Best-case}{Worst-case}$$

$$= \lim_{n->\infty} \frac{n-1}{\left(\frac{n(n-1)}{2}\right)}$$

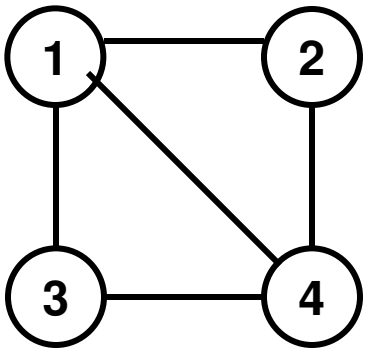
$$= \lim_{n->\infty} \frac{2}{n} = 0$$

O(n²)

The **colored** elements are in the sorted sequence and the circled element is at index *j* of the algorithm.

2.3 Transform and Conquer

Walks of Certain Length in a Graph



	1	2	3	4
1	0	1	1	1
2	1	0	0	1
3	1	0	0	1
4	1	1	1	0

A Walk is a sequence of vertices connecting source and destination such that any vertex (including the end vertices) could occur even more than once. In a path, an intermediate vertex (if any is present) could occur only once.

Walk (also a path): 2 – 1 – 3 Length: 2
Walk (not a path): 2 – 1 – 2 – 4 – 3 Length: 4

$A^2 =$

	1	2	3	4
1	0	1	1	1
2	1	0	0	1
3	1	0	0	1
4	1	1	1	0

×

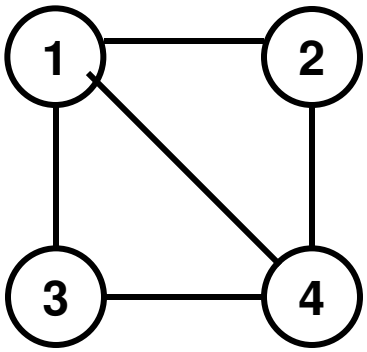
	1	2	3	4
1	0	1	1	1
2	1	0	0	1
3	1	0	0	1
4	1	1	1	0

=

	1	2	3	4
1	3	1	1	2
2	1	2	2	1
3	1	2	2	1
4	2	1	1	3

Number of Walks in a Graph

 # walks of Length 4:
Find $A^4 = A^2 * A^2$.



	1	2	3	4
1	3	1	1	2
2	1	2	2	1
3	1	2	2	1
4	2	1	1	3

X

	1	2	3	4
1	3	1	1	2
2	1	2	2	1
3	1	2	2	1
4	2	1	1	3

$A^4 =$

	1	2	3	4
1	15	9	9	14
2	9	10	10	9
3	9	10	10	9
4	14	9	9	15

To find the number of walks length 4 between vertices b and c, just simply do a pair-wise multiplication and addition of the elements corresponding to the row for vertex ‘b’ in A^2 with the elements corresponding to the column for vertex ‘c’ in A^2.

Note: Rule for Matrix Multiplication
To find the value of an entry in cell (i, j) in the product matrix P = A * B,
Do a pair-wise multiplication and addition of the elements in row ‘i’ of the first matrix A and the elements in column ‘j’ of the second matrix B.

2.4 Space-Time Tradeoff

In-place vs. Out-of-place Algorithms

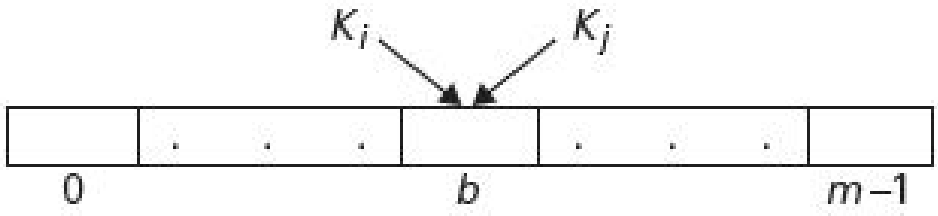
- An algorithm is said to be “in-place” if it uses a minimum and/or constant amount of extra storage space to transform or process an input to obtain the desired output.
  - Depending on the nature of the problem, an in-place algorithm may sometime overwrite an input to the desired output as the algorithm executes (as in the case of in-place sorting algorithms); the output space may sometimes be a constant (for example in the case of string-matching algorithms).
- Algorithms that use significant amount of extra storage space (sometimes, additional space as large as the input – example: merge sort) are said to be out-of-place in nature.
- Time-Space Complexity Tradeoffs of Sorting Algorithms:
  - In-place sorting algorithms like Selection Sort, Bubble Sort, Insertion Sort and Quick Sort have a worst-case time complexity of $\Theta(n^2)$.
  - On the other hand, Merge sort has a space-complexity of $\Theta(n)$, but has a worst-case time complexity of $\Theta(n\log n)$.

Hashing

- A very efficient method for implementing a *dictionary,* i.e., a set with the operations: find, insert and delete
- Based on representation-change and space-for-time tradeoff ideas
- We consider the problem of implementing a dictionary of *n* records with keys K_1, K_2, …, K_n.
- Hashing is based on the idea of distributing keys among a one-dimensional array H[0…m-1] called a hash table.
  - The distribution is done by computing, for each of the keys, the value of some pre-defined function *h* called the ***hash function***.
  - The hash function assigns an integer between 0 and *m*-1, called the hash address to a key.
  - The size of a hash table *m* is typically a prime integer.
- Typical hash functions
  - For non-negative integers as key, a hash function could be h(K)=K mod m;
  - If the keys are letters of some alphabet, the position of the letter in the alphabet (for example, A is at position 1 in alphabet A – Z) could be used as the key for the hash function defined above.
  - If the key is a character string $c_0\ c_1\ \ldots\ c_{s-1}$ of characters from an alphabet, then, the hash function could be: $(\sum_{i=0}^{s-1} ord(c_i)) \bmod m$

Collisions and Collision Resolution

If $h(K_1) = h(K_2)$, there is a *collision*



- Good hash functions result in fewer collisions but some collisions should be expected
- In this module, we will look at open hashing that works for arrays of any size, irrespective of the hash function.

Open Hashing

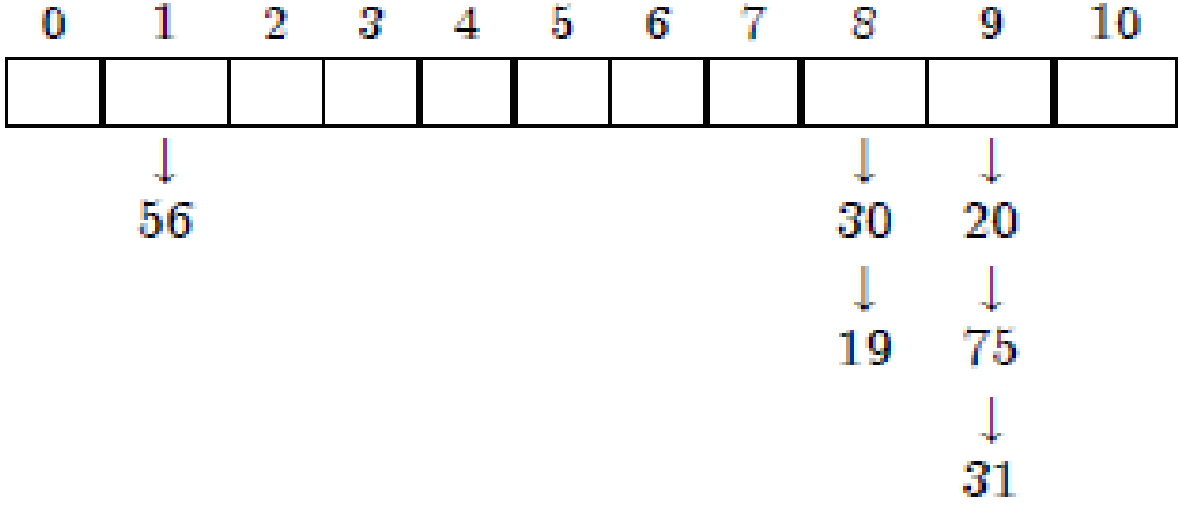
The list of keys: 30, 20, 56, 75, 31, 19

The hash function: $h(K) = K \bmod 11$

The hash addresses:

K	30	20	56	75	31	19
$h(K)$	8	9	1	9	9	8

The open hash table:



The largest number of key comparisons in a successful search in this table is 3 (in searching for $K = 31$).

The average number of key comparisons in a successful search in this table, assuming that a search for each of the six keys is equally likely, is

$$\frac{1}{6} \cdot 1 + \frac{1}{6} \cdot 1 + \frac{1}{6} \cdot 1 + \frac{1}{6} \cdot 2 + \frac{1}{6} \cdot 3 + \frac{1}{6} \cdot 2 = \frac{10}{6} \approx 1.7.$$

Open Hashing

- Inserting and Deleting from the hash table is of the same complexity as searching.
- If hash function distributes keys uniformly, average length of linked list will be $\alpha = n/m$. This ratio is called *load factor*.
- Average-case number of key comparisons for a successful search is $\alpha/2$; Average-case number of key comparisons for an unsuccessful search is α.
- Worst-case number of key comparisons is $\Theta(n)$ – occurs if we get a linked list containing all the n elements hashing to the same index. To avoid this, we need to be careful in selecting a proper hashing function.
  - Mod-based hashing functions with a prime integer as the divisor are more likely to result in hash values that are evenly distributed across the keys.
- Open hashing still works if the number of keys, $n >$ the size of the hash table, m.

Applications of Hashing (1)

Finding whether an array is a Subset of another array

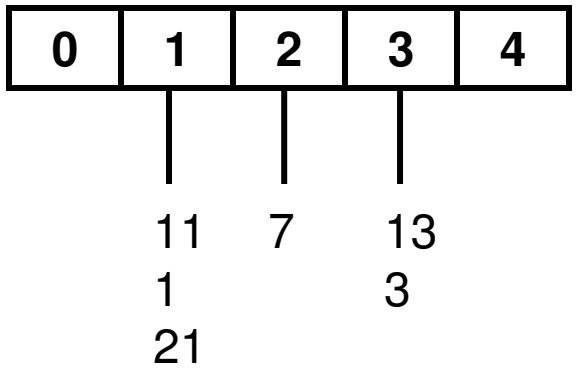
- Given two arrays A_L (larger array) and A_S (smaller array) of distinct elements, we want to find whether A_S is a subset of A_L.
- Example: $A_L = \{11, 1, 13, 21, 3, 7\}$; $A_S = \{11, 3, 7, 1\}$; A_S is a subset of A_L.
- Solution: Use (open) hashing. Hash the elements of the larger array, and for each element in the smaller array: search if it is in the hash table for the larger array. If even one element in the smaller array is not there in the larger array, we could stop!
- Time-complexity:
  - $\Theta(n)$ to construct the hash table on the larger array of size n, and another $\Theta(n)$ to search the elements of the smaller array.
  - A brute-force approach would have taken $\Theta(n^2)$ time.
- Space-complexity: $\Theta(n)$ with the hash table approach and $\Theta(1)$ with the brute-force approach.

- Note: The above solution could also be used to find whether two sets are disjoint or not. Even if one element in the smaller array is there in the larger array, we could stop!

Applications of Hashing (1)

Finding whether an array is a Subset of another array

- **Example 1**: A_L = {11, 1, 13, 21, 3, 7};
- A_S = {11, 3, 7, 1}; A_S is a subset of A_L.
- Let H(K) = K mod 5.



Hash table approach

comparisons = 1 (for 11) + 2 (for 3) +
1 (for 7) + 2 (for 1) = 6

Brute-force approach: Pick every element in the smaller array and do a linear search for it in the larger array.

comparisons = 1 (for 11) + 5 (for 3) +
6 (for 7) + 2 (for 1) = 14

- **Example 2**: A_L = {11, 1, 13, 21, 3, 7};
- A_S = {11, 3, 7, 4}; A_S is NOT a subset of A_L.
- Let H(K) = K mod 5.

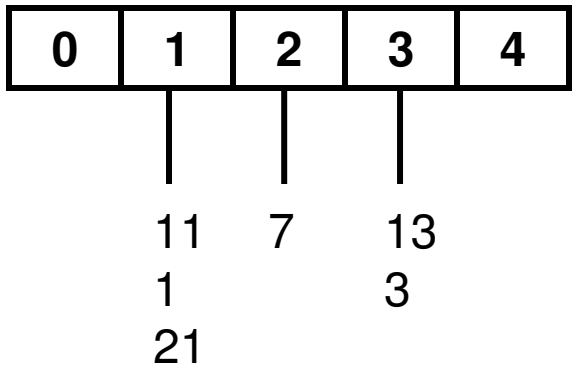
The **hash table approach** would take just 1 (for 11) + 2 (for 3) + 1 (for 7) + 0 (for 4) = 4 comparisons

The **brute-force approach** would take: 1 (for 11) + 5 (for 3) + 6 (for 7) + 6 (for 4) = 18 comparisons.

Applications of Hashing (1)

Finding whether two arrays are disjoint are not

- **Example 1**: A_L = {11, 1, 13, 21, 3, 7};
- A_S = {22, 25, 27, 28}; They are disjoint.
- Let H(K) = K mod 5.



Hash table approach

comparisons = 1 (for 22) + 0 (for 25) + 1 (for 27) + 3 (for 28) = 5

Brute-force approach: Pick every element in the smaller array and do a linear search for it in the larger array.

comparisons = 6 comparisons for each element * 4 = 24

- **Example 2**: A_L = {11, 1, 13, 21, 3, 7};
- A_S = {22, 25, 27, 1}; They are NOT disjoint.
- Let H(K) = K mod 5.

The **hash table approach** would take just 1 (for 22) + 0 (for 25) + 1 (for 27) + 2 (for 1) = 4 comparisons

The **brute-force approach** would take: 6 (for 22) + 6 (for 25) + 6 (for 27) + 2 (for 1) = 20 comparisons.

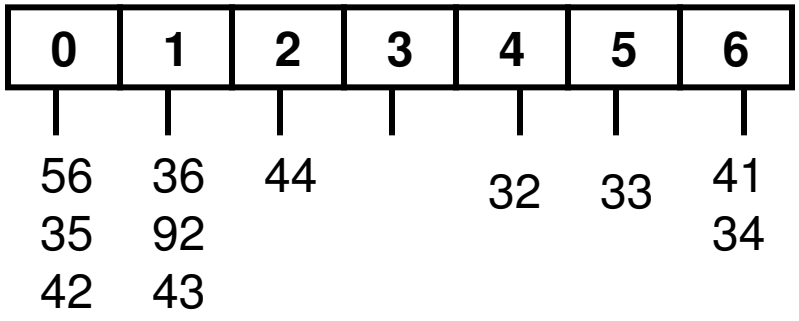
Applications of Hashing (2)

Finding Consecutive Subsequences in an Array

- Given an array A of unique integers, we want to find the contiguous subsequences of length 2 or above as well as the length of the largest subsequence.
- Assume it takes Θ(1) time to insert or search for an element in the hash table.

36 41 56 35 44 33
34 92 43 32 42

H(K) = K mod 7



36	41	56	35	44	33	34	92	43	32	42
35	40	55	34	43	32	33	91	42	31	41
	42	57					93		33	
	43								34	
	44								35	
	45								36	
									37	
	41								**32**	
	42								**33**	
	43								**34**	
	44								**35**	
									36	

Applications of Hashing (1)

Finding Consecutive Subsequences in an Array

- Algorithm

```
Insert the elements of A in a hash table H
Largest Length = 0
for i = 0 to n-1 do
    if (A[i] – 1 is not in H) then
        j = A[i]   // A[i] is the first element of a possible cont. sub seq.
        j = j + 1
        while ( j  is in H) do
            j = j + 1
        end while
        if ( j – A[i] > 1) then  // we have found a cont. sub seq. of length > 1
            Print all integers from A[i] … (j-1)
            if (Largest Length < j – A[i]) then
                Largest Length = j – A[i]
            end if
        end if
    end if
end for
```

L searches in the Hash table H for sub sequences of length L

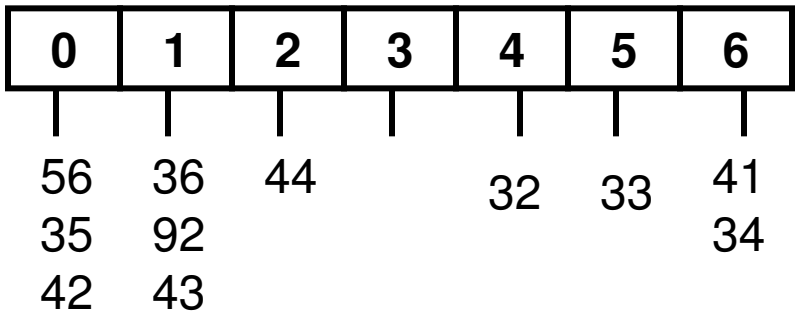
Applications of Hashing (2)

Finding Consecutive Subsequences in an Array

- **Time Complexity Analysis**
- For each element at index i in the array A we do at least one search (for element A[i] – 1) in the hash table.
- For every element that is the first element of a sub seq. of length 1 or above (say length L), we do L searches in the Hash table.
- The sum of all such Ls should be n.
- For an array of size n, we do n + n = 2n = Θ(n) hash searches. The first 'n' corresponds to the sum of all the lengths of the contiguous sub sequences and the second 'n' is the sum of all the 1s (one 1 for each element in the array)

36 41 56 35 44 33
34 92 43 32 42

H(K) = K mod 7



36	41	56	35	44	33	34	92	43	32	42
35	40	55	34	43	32	33	91	42	31	41
	42	57					93		33	
	43								34	
	44								35	
	45								36	
									37	