PROBLEM # 1

Checking for element uniqueness in an array:

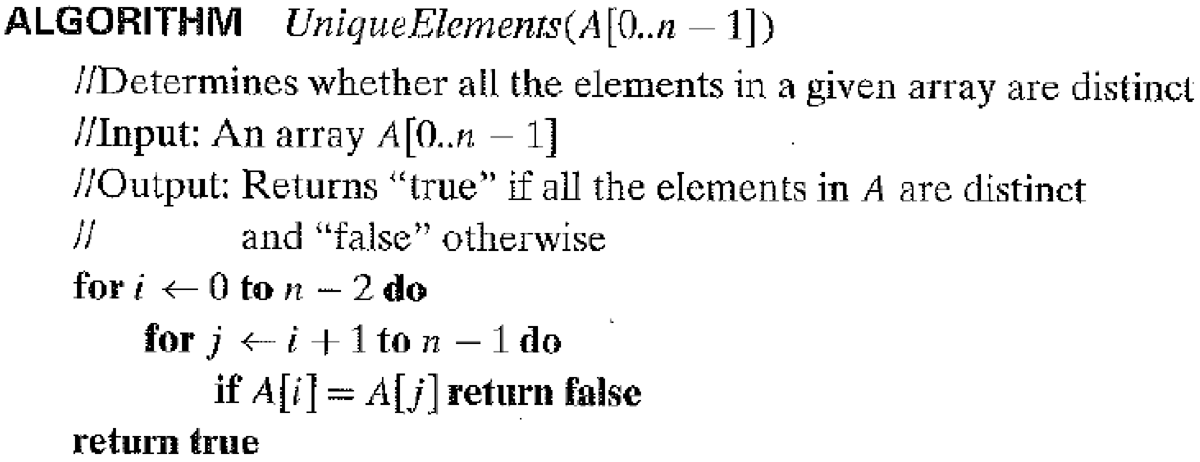
Consider an array of elements and determine if every element in the array is unique.

Left

21	2	15	99	260	20	15	80	No
180	60	30	1	2	4	12	24	Yes

Have we already seen this problem? If yes, how did we solve it?

SOLUTION



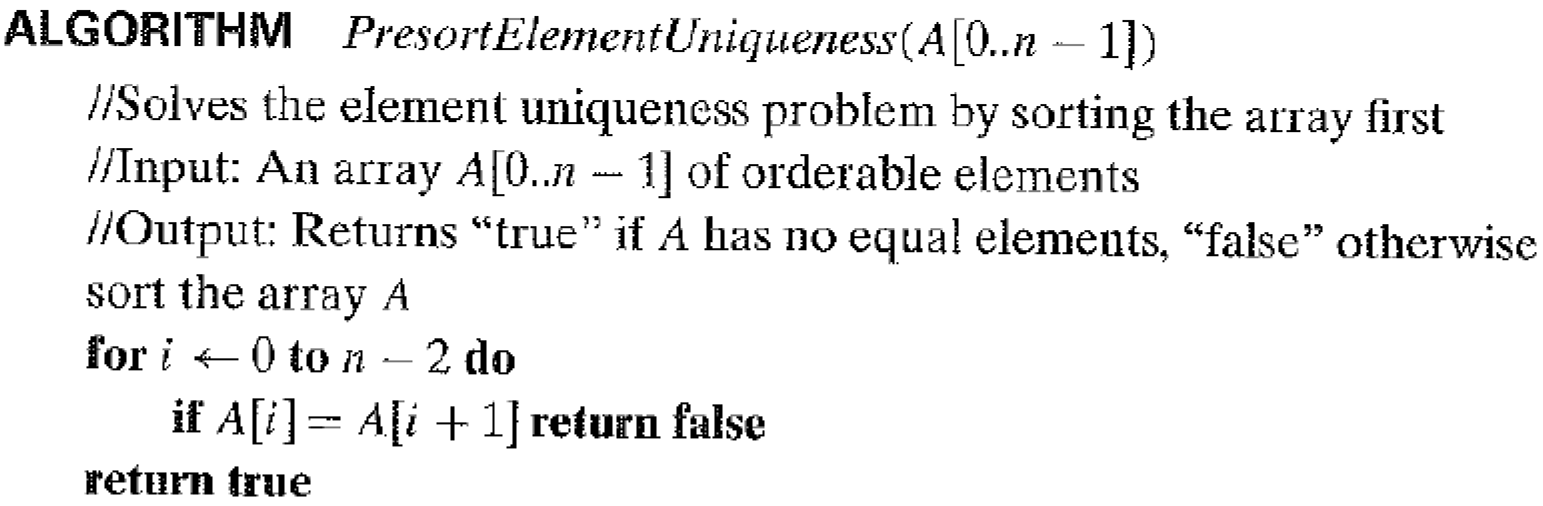
```
ALGORITHM  UniqueElements(A[0..n − 1])
    //Determines whether all the elements in a given array are distinct
    //Input: An array A[0..n − 1]
    //Output: Returns “true” if all the elements in A are distinct
    //          and “false” otherwise
    for i ← 0 to n − 2 do
        for j ← i + 1 to n − 1 do
            if A[i] = A[j] return false
    return true
```

What is the design strategy and time efficiency of this algorithm?

SOLUTION

- ✓ This algorithm is designed using Brute Force design strategy.
- ✓ Its worst – case efficiency is $\Theta(n^2)$.
- ✓ ***Can we improve the efficiency of this algorithm from $\Theta(n^2)$ to $\Theta(n)$?***
- ✓ If so, how?

PRESORTING



```
ALGORITHM PresortElementUniqueness($A[0..n-1]$)
    //Solves the element uniqueness problem by sorting the array first
    //Input: An array $A[0..n-1]$ of orderable elements
    //Output: Returns "true" if $A$ has no equal elements, "false" otherwise
    sort the array $A$
    for $i \leftarrow 0$ to $n-2$ do
        if $A[i] = A[i+1]$ return false
    return true
```

PRESORTING

The worst – case efficiency of this algorithm is:

$$T(n) = T_{sort}(n) + T_{scan}(n)$$

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

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

Thus, this algorithm is better than the brute – force algorithm if a ***n log n*** sorting algorithm is used.

PRESORTING

- ✓ This algorithm solves the problem of Element Uniqueness by first transforming the given input unsorted array to a sorted array and then checks only consecutive elements: if the array has equal elements, a pair of them must be next to each other and vice versa.

- ✓ This algorithm design strategy of transforming the given input instance to something more amenable to the solution and then solving it is called Transform and Conquer.

TRANSFORM AND CONQUER

THE IDEA

- ✓ The ***transform-and-conquer*** algorithms work as two-stage procedures.
- ✓ First, in the **transformation stage**, the problem's instance is modified to be, for one reason or another, more amenable to solution.
- ✓ Then, in the second or **conquering stage**, it is solved.
- ✓ The *transform-and-conquer* idea has three major variations.

VARIATIONS

Instance Simplification

Transformation to a simpler or more convenient instance of the same problem

Representation Change

Transformation to a different representation of the same instance

Problem Reduction

Transformation to an instance of a different problem for which an algorithm is already available

PRESORTING

- ✓ The ***presorting*** idea is an example of the Instance Simplification variety.
- ✓ Many questions about lists are easier to answer if the lists are sorted.
- ✓ The time efficiency of the algorithms that involve sorting will depend on the time efficiency of the sorting algorithm.

PROBLEM # 2

Computing a mode:

A *mode* is a value that occurs most often in a given list of numbers.

1	2	2	4	1	5	4	4	Mode = 4

What is the Brute Force way of solving this problem?

SOLUTION

- ✓ Scan the list and compute the frequencies of all its distinct values, then find the value with the largest frequency.

- ✓ In order to implement this idea, we can store the values already encountered, along with their frequencies, in a separate list.

- ✓ On each iteration, the ith element of the original list is compared with the values already encountered by traversing this auxiliary list.

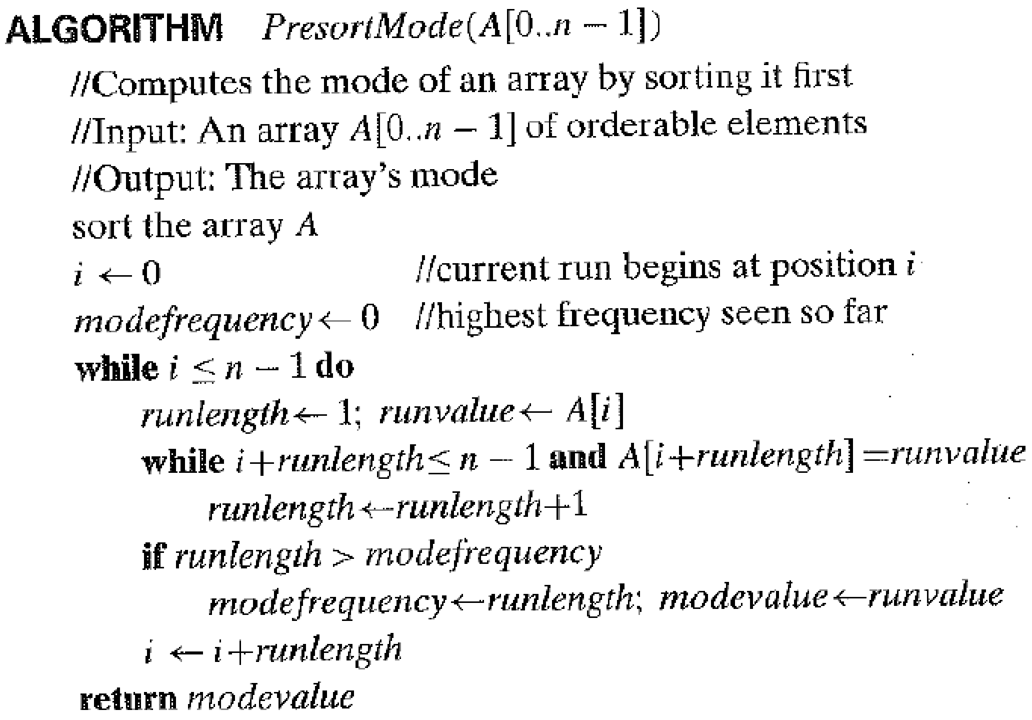
- ✓ If a matching value is found, its frequency is incremented; otherwise, the current element is added to the list of distinct values seen so far with a frequency of 1.

EFFICIENCY

- Worst – case input: A list with no matching elements.
- For such a list, its ith element is compared with i – 1 elements of the auxiliary list of distinct values seen so far before being added to the list with a frequency of 1.
- The number of comparisons made by this algorithm is:

$$C(n) = \sum_{i=1}^{n} (i - 1) = 0 + 1 + \cdots + (n - 1) = \frac{(n-1)n}{2} \in \Theta(n^2).$$

PRESORTING SOLUTION



```
ALGORITHM  $PresortMode(A[0..n-1])$
    //Computes the mode of an array by sorting it first
    //Input: An array $A[0..n-1]$ of orderable elements
    //Output: The array's mode
    sort the array $A$
    $i \leftarrow 0$                       //current run begins at position $i$
    $modefrequency \leftarrow 0$    //highest frequency seen so far
    while $i \leq n-1$ do
        $runlength \leftarrow 1$;  $runvalue \leftarrow A[i]$
        while $i+runlength \leq n-1$ and $A[i+runlength] = runvalue$
            $runlength \leftarrow runlength+1$
        if $runlength > modefrequency$
            $modefrequency \leftarrow runlength$;  $modevalue \leftarrow runvalue$
        $i \leftarrow i+runlength$
    return $modevalue$
```

PRESORTING

The worst – case efficiency of this algorithms is:

$$T(n) = T_{sort}(n) + T_{scan}(n)$$

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

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

Thus, this algorithm is better than the brute – force algorithm if a n log n sorting algorithm is used.

PROBLEM # 3

Searching:

Searching for a given value *v* in a given array of *n* sortable items.

14	32	12	4	100	56	84	93	v = 40	Not Found

What is the Brute Force way of solving this problem?

BRUTE FORCE SOLUTION

- ✓ The Brute Force solution to this problem is Sequential Search.
- ✓ The worst – case efficiency of this algorithms is Θ(n).

PRESORTING SOLUTION

✓ The presorting solution to this problem is Binary Search whose running time will be:

$$T(n) = T_{sort}(n) + T_{search}(n) = \Theta(n \log n) + \Theta(\log n) = \Theta(n \log n)$$

✓ This efficiency is inferior to sequential search.

GAUSSIAN ELIMINATION

✓ In many applications we need to solve a system of linear equations like the one below:

$$\begin{aligned}
a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n &= b_1 \\
a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n &= b_2 \\
\vdots \\
a_{n1}x_1 + a_{n2}x_2 + \cdots + a_{nn}x_n &= b_n
\end{aligned}$$

✓ An elegant solution to solving such a system of linear equations is ***Gaussian Elimination***.

GAUSSIAN ELIMINATION – THE IDEA

- ✓ The idea of Gaussian Elimination is to transform a system of n linear equations in n unknowns to an equivalent system with an upper triangular co – efficient matrix, a matrix with all zeros below its diagonal.

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n &= b_1 \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n &= b_2 \\ &\vdots \\ a_{n1}x_1 + a_{n2}x_2 + \cdots + a_{nn}x_n &= b_n \end{aligned} \quad \Longrightarrow \quad \begin{aligned} a'_{11}x_1 + a'_{12}x_2 + \cdots + a'_{1n}x_n &= b'_1 \\ a'_{22}x_2 + \cdots + a'_{2n}x_n &= b'_2 \\ &\vdots \\ a'_{nn}x_n &= b'_n. \end{aligned}$$

GAUSSIAN ELIMINATION – THE IDEA

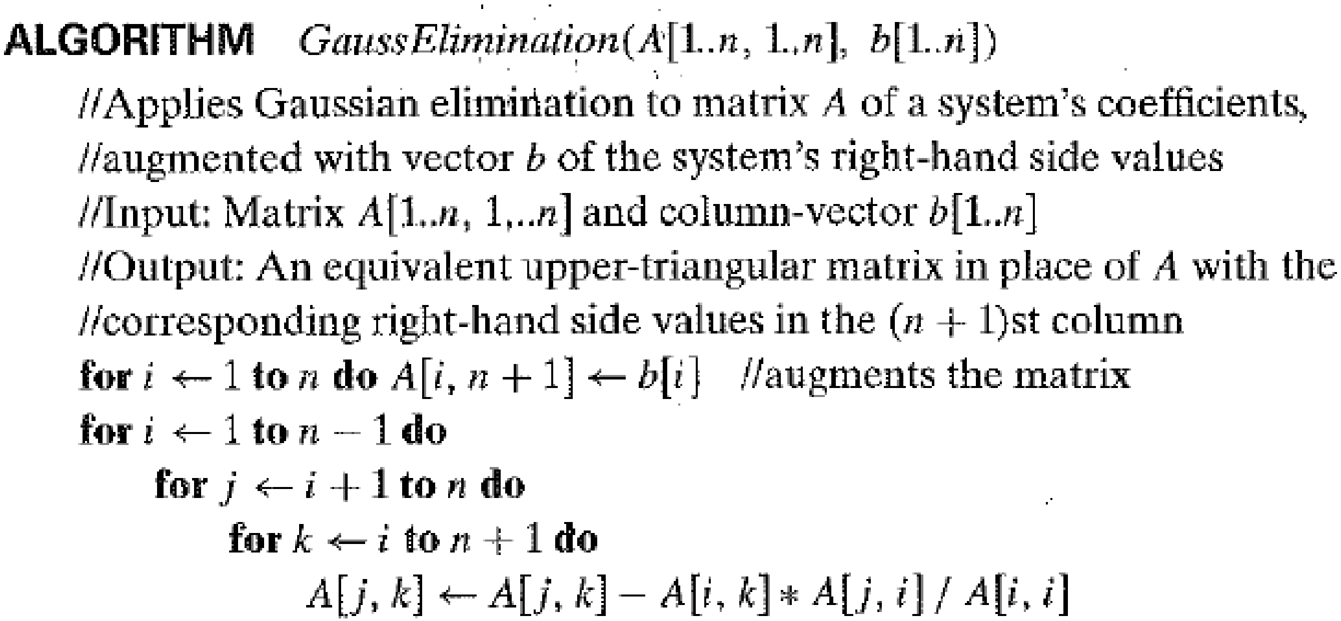
- ✓ The matrix representation of the system is:

$$Ax = b \implies A'x = b',$$

where

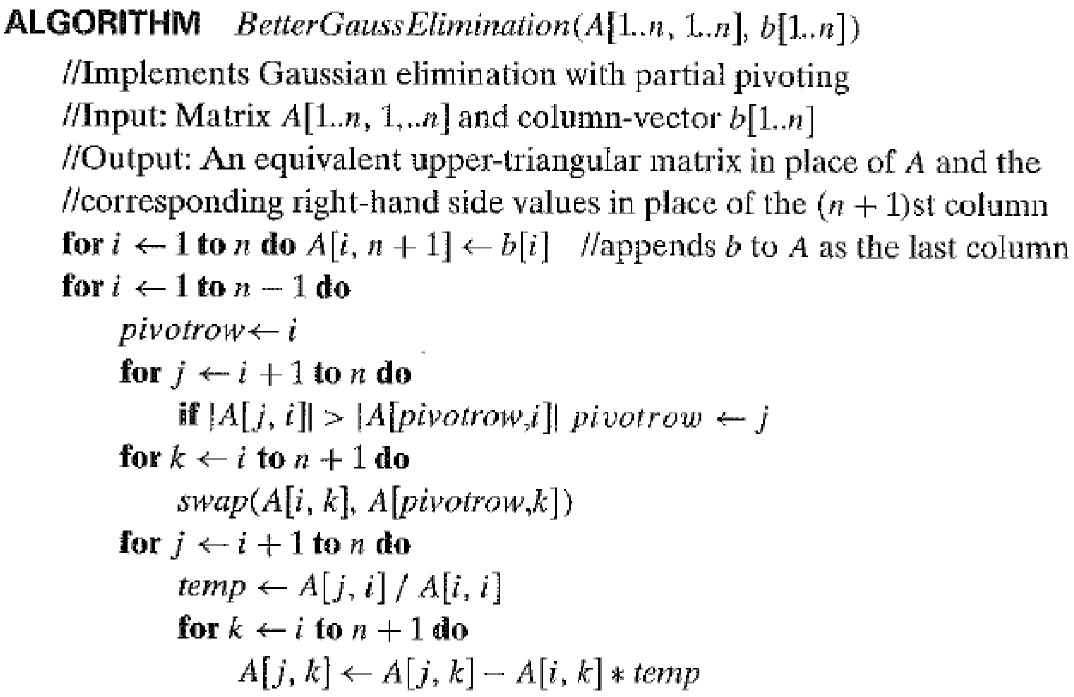
$$A = \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & & & \\ a_{n1} & a_{n2} & \dots & a_{nn} \end{bmatrix}, \; b = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix}, \; A' = \begin{bmatrix} a'_{11} & a'_{12} & \dots & a'_{1n} \\ 0 & a'_{22} & \dots & a'_{2n} \\ \vdots & & & \\ 0 & 0 & \dots & a'_{nn} \end{bmatrix}, \; b = \begin{bmatrix} b'_1 \\ b'_2 \\ \vdots \\ b'_n \end{bmatrix}.$$

GAUSSIAN ELIMINATION – THE ALGORITHM



```
ALGORITHM  $GaussElimination(A[1..n, 1..n], b[1..n])$
    //Applies Gaussian elimination to matrix $A$ of a system's coefficients,
    //augmented with vector $b$ of the system's right-hand side values
    //Input: Matrix $A[1..n, 1,...n]$ and column-vector $b[1..n]$
    //Output: An equivalent upper-triangular matrix in place of $A$ with the
    //corresponding right-hand side values in the $(n+1)$st column
    for $i \leftarrow 1$ to $n$ do $A[i, n+1] \leftarrow b[i]$   //augments the matrix
    for $i \leftarrow 1$ to $n-1$ do
        for $j \leftarrow i+1$ to $n$ do
            for $k \leftarrow i$ to $n+1$ do
                $A[j, k] \leftarrow A[j, k] - A[i, k] * A[j, i] / A[i, i]$
```

GAUSSIAN ELIMINATION – BETTER ALGORITHM



```
ALGORITHM  $BetterGaussElimination(A[1..n, 1..n], b[1..n])$
    //Implements Gaussian elimination with partial pivoting
    //Input: Matrix $A[1..n, 1,..n]$ and column-vector $b[1..n]$
    //Output: An equivalent upper-triangular matrix in place of $A$ and the
    //corresponding right-hand side values in place of the $(n + 1)$st column
    for $i \leftarrow 1$ to $n$ do $A[i, n + 1] \leftarrow b[i]$    //appends $b$ to $A$ as the last column
    for $i \leftarrow 1$ to $n - 1$ do
        $pivotrow \leftarrow i$
        for $j \leftarrow i + 1$ to $n$ do
            if $|A[j, i]| > |A[pivotrow, i]|$ $pivotrow \leftarrow j$
        for $k \leftarrow i$ to $n + 1$ do
            $swap(A[i, k], A[pivotrow, k])$
        for $j \leftarrow i + 1$ to $n$ do
            $temp \leftarrow A[j, i] / A[i, i]$
            for $k \leftarrow i$ to $n + 1$ do
                $A[j, k] \leftarrow A[j, k] - A[i, k] * temp$
```

GAUSSIAN ELIMINATION – TIME EFFICIENCY

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

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

$$= (n+1)(n-1) + n(n-2) + \cdots + 3 \cdot 1$$

$$= \sum_{j=1}^{n-1} (j+2)j = \sum_{j=1}^{n-1} j^2 + \sum_{j=1}^{n-1} 2j = \frac{(n-1)n(2n-1)}{6} + 2\frac{(n-1)n}{2}$$

$$= \frac{n(n-1)(2n+5)}{6} \approx \frac{1}{3}n^3 \in \Theta(n^3).$$

BALANCED SEARCH TREES

- ✓ A Binary Search Tree is a binary tree whose nodes contain elements of a set of orderable items, one element per node, so that all elements in the left subtree are smaller than the element in the subtree's root, and all the elements in the right subtree are greater than it.

- ✓ Example of Representation Change.

BALANCED SEARCH TREES

- ✓ The time efficiency of searching, insertion, and deletion, which are all in Θ(log *n),* but only in the average case.

- ✓ In the worst case, these operations are in Θ(n) because the tree can degenerate into a severely unbalanced one with its height equal to *n* - 1.

- ✓ Computer scientists have come up with two approaches which preserve the logarithmic efficiency of dictionary operations but avoids its worst case degeneracy.

AVL TREES

- ✓ An ***AVL tree*** is a binary search tree in which the ***balance factor*** of every node, which is defined as the difference between the heights of the node's left and right subtrees, is either 0 or + 1 or -1. (The height of the empty tree is defined as -1.)

- ✓ If an insertion of a new node makes an AVL tree unbalanced, we transform the tree by a rotation.

- ✓ A ***rotation*** in an AVL tree is a local transformation of its subtree rooted at a node whose balance has become either +2 or -2; if there are several such nodes, we rotate the tree rooted at the unbalanced node that is the closest to the newly inserted leaf.

AVL TREES

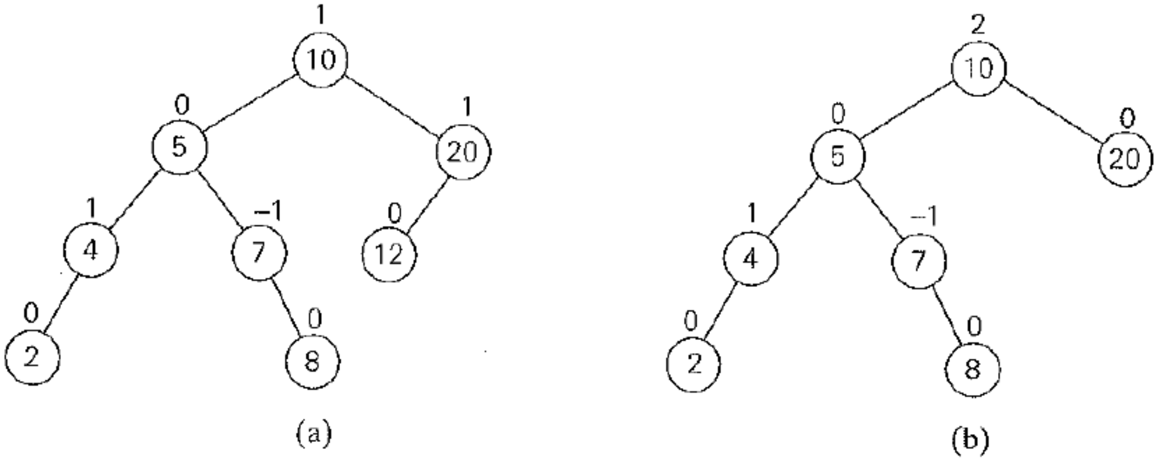


FIGURE 6.2 (a) AVL tree. (b) Binary search tree that is not an AVL tree. The number above each node indicates that node's balance factor.

AVL TREES - ROTATIONS

✓ There are only four types of rotations

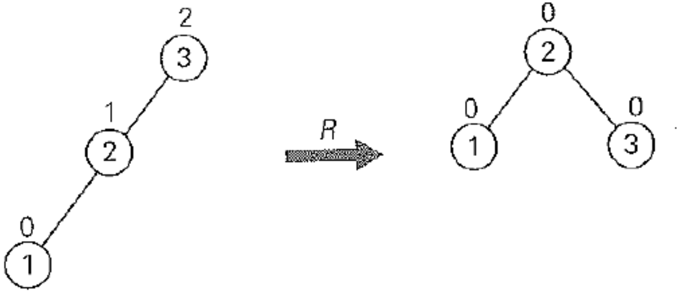
- *Single Right Rotation or R Rotation*
- *Single Left Rotation or L Rotation*
- *Double Left Right Rotation or LR Rotation*
- *Double Right Left Rotation or RL Rotation*

AVL TREES - ROTATIONS

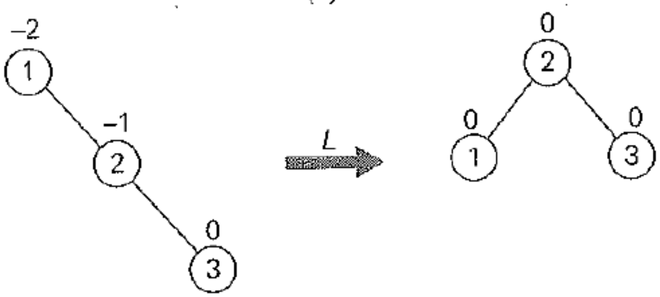
✓ There are only four types of rotations

- *Single Right Rotation or R Rotation*
- *Single Left Rotation or L Rotation*
- *Double Left Right Rotation or LR Rotation*
- *Double Right Left Rotation or RL Rotation*

AVL TREES - ROTATIONS



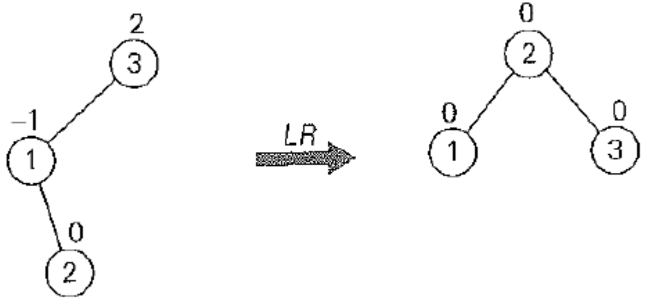
(a)



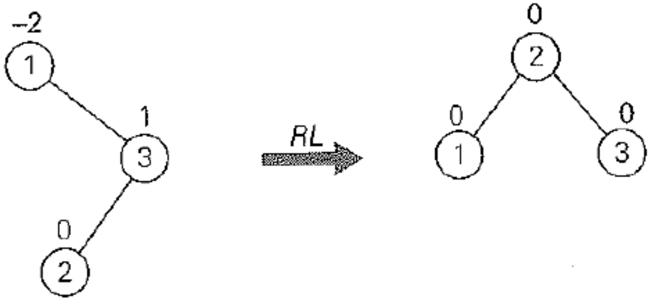
(b)

a) R Rotation
b) L Rotation

AVL TREES - ROTATIONS



(c)



(d)

c) LR Rotation
d) RL Rotation

AVL TREES - EFFICIENCY

- ✓ The operations of insertion, deletion and searching is logarithmic i.e., it belongs to Θ(log n).

2 – 3 TREES

- ✓ A *2-3 tree* is a tree that can have nodes of two kinds: 2-nodes and 3-nodes.
- ✓ A *2-node* contains a single key *K* and has two children: the left child serves as the root of a subtree whose keys are less than K and the right child serves as the root of a subtree whose keys are greater than *K.*
- ✓ A ***3-node*** contains two ordered keys K1 and *K2* (K1 < *K2)* and has three children. The leftmost child serves as the root of a subtree with keys less than K1, the middle child serves as the root of a subtree with keys between K1 and *K2,* and the rightmost child serves as the root of a subtree with keys greater than *K2.*

2 – 3 TREES

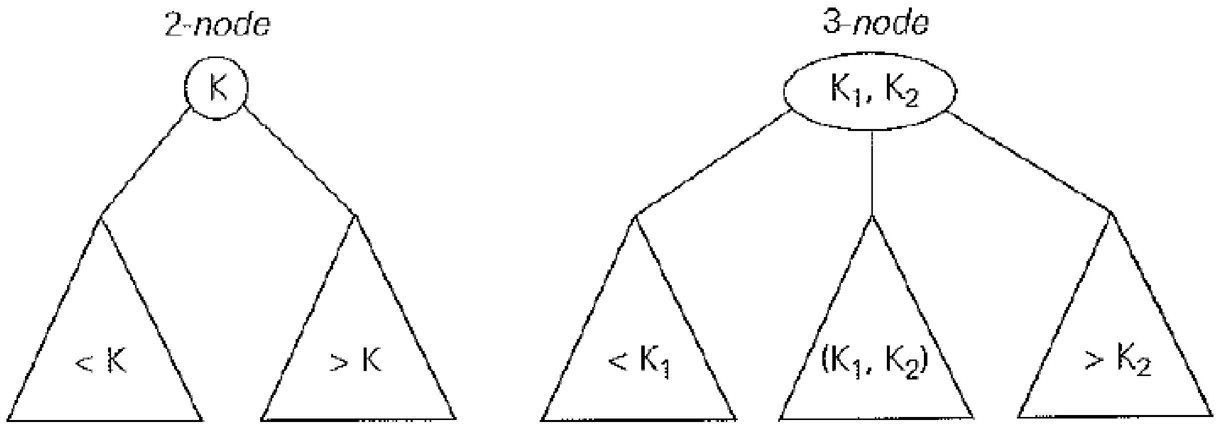


FIGURE 6.7 Two kinds of nodes of a 2-3 tree

2 – 3 TREES

- ✓ The last requirement of the 2-3 tree is that all its leaves must be on the same level, i.e., a 2-3 tree is always perfectly height-balanced: the length of a path from the root of the tree to a leaf must be the same for every leaf.
- ✓ The operations of insertion, deletion and searching can be applied to 2 – 3 Trees.

2 - 3 TREES - EFFICIENCY

- ✓ For any 2-3 tree of height *h* with *n* nodes, we get the inequality

$$n \geq 1 + 2 + \cdots + 2^h = 2^{h+1} - 1,$$

- ✓ And hence:

$$h \leq \log_2(n + 1) - 1.$$

2 - 3 TREES - EFFICIENCY

- ✓ On the other hand, a 2 – 3 tree of height h with the largest number of keys is a full tree of 3 – nodes, each with two keys and three children. Therefore:

$$n \leq 2 \cdot 1 + 2 \cdot 3 + \cdots + 2 \cdot 3^h = 2(1 + 3 + \cdots + 3^h) = 3^{h+1} - 1,$$

- ✓ And hence:

$$h \geq \log_3(n + 1) - 1.$$

- ✓ The lower and upper bounds on height h:

$$\log_3(n + 1) - 1 \leq h \leq \log_2(n + 1) - 1,$$

2 - 3 TREES - EFFICIENCY

- ✓ These upper and lower bounds indicate that the searching, insertion and deletion are all in $\Theta(\log n)$.

HEAPS AND HEAPSORT

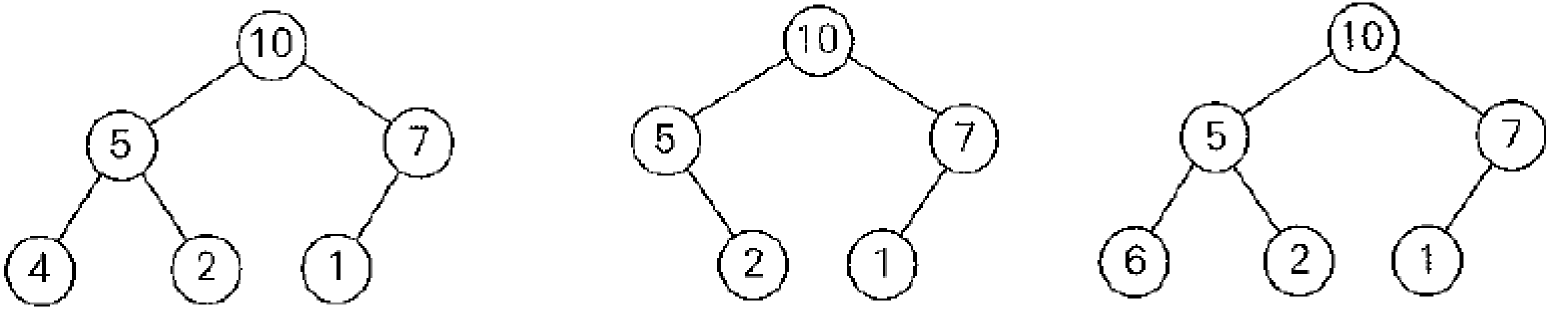
Definition:

A *heap* can be defined as a binary tree with keys assigned to its nodes (one key per node) provided the following two conditions are met:

1. ***The tree's shape requirement*** - The binary tree is *essentially complete* (or simply *complete),* that is, all its levels are full except possibly the last level, where only some rightmost leaves may be missing.

2. ***The parental dominance requirement*** - The key at each node is greater than or equal to the keys at its children. (This condition is considered automatically satisfied for all leaves.)

HEAPS AND HEAPSORT



Only the leftmost tree is a heap. Why?

PROPERTIES OF HEAPS

1. There exists exactly one essentially complete binary tree with *n* nodes. Its height is equal to floor($\log_2 n$).

2. The root of a heap always contains its largest element.

3. A node of a heap considered with all its descendants is also a heap.

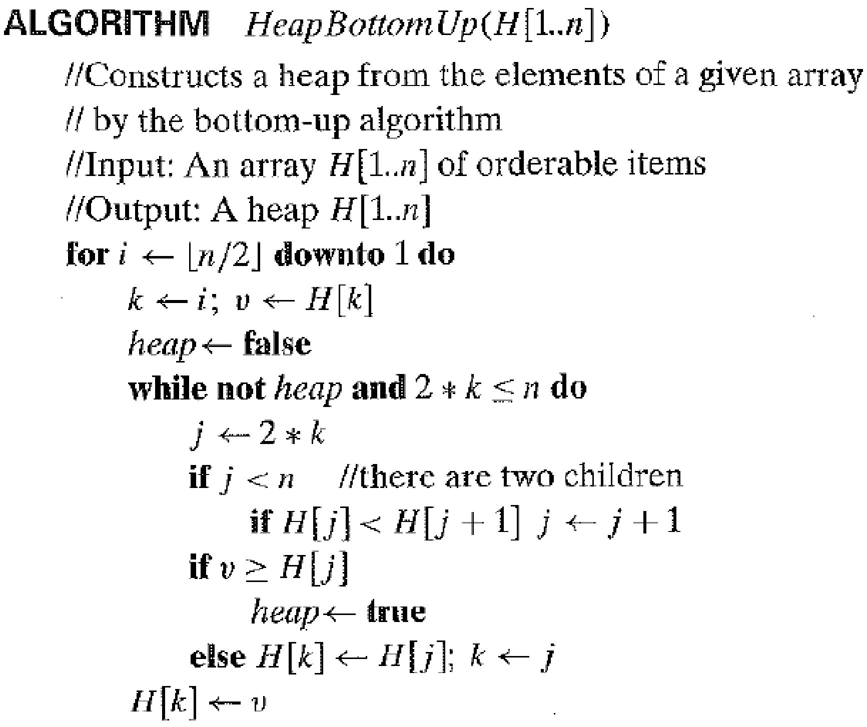
4. A heap can be implemented as an array by recording its elements in the topdown, left-to-right fashion. It is convenient to store the heap's elements in positions 1 though *n* of such an array, leaving H[O] either unused or putting there a sentinel whose value is greater than every element in the heap. In such a representation,

PROPERTIES OF HEAPS

a) the parental node keys will be in the first floor(n/2) positions of the array, while the leaf keys will occupy the last ceil(n/2) positions;

b) the children of a key in the array's parental position *i* (1 <= *I* <= *floor*(n/2)) will be in positions *2i* and *2i* + 1, and, correspondingly, the parent of a key in position *i* (2 <= *i* <= *n)* will be in position floor(i /2).

HEAP CONSTRUCTION – BOTTOM UP



```
ALGORITHM HeapBottomUp($H[1..n]$)
    //Constructs a heap from the elements of a given array
    // by the bottom-up algorithm
    //Input: An array $H[1..n]$ of orderable items
    //Output: A heap $H[1..n]$
    for $i \leftarrow \lfloor n/2 \rfloor$ downto 1 do
        $k \leftarrow i$; $v \leftarrow H[k]$
        $heap \leftarrow$ false
        while not $heap$ and $2 * k \leq n$ do
            $j \leftarrow 2 * k$
            if $j < n$    //there are two children
                if $H[j] < H[j + 1]$ $j \leftarrow j + 1$
            if $v \geq H[j]$
                $heap \leftarrow$ true
            else $H[k] \leftarrow H[j]$; $k \leftarrow j$
        $H[k] \leftarrow v$
```

HEAP CONSTRUCTION – BOTTOM UP (EFFICIENCY)

$$C_{worst}(n) = \sum_{i=0}^{h-1} \sum_{\text{level } i \text{ keys}} 2(h-i) = \sum_{i=0}^{h-1} 2(h-i)2^i = 2(n - \log_2(n+1)),$$

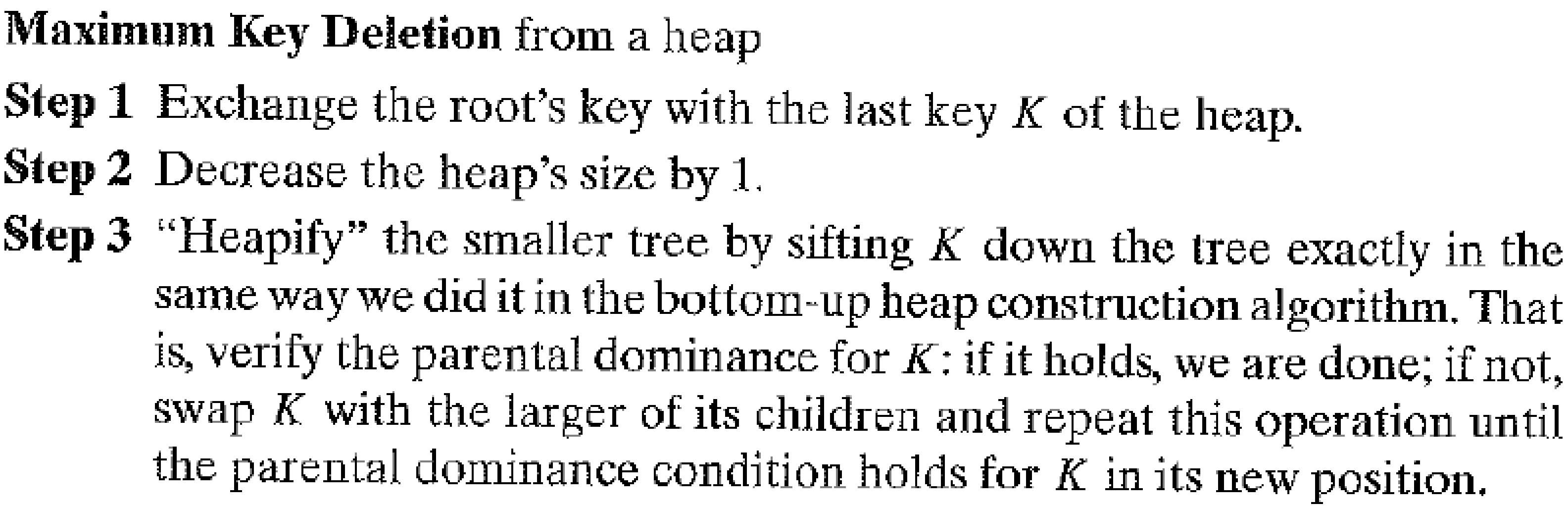
HEAP CONSTRUCTION – TOP DOWN

1. First, attach a new node with key *K* in it after the last leaf of the existing heap.
2. Then sift *K* up to its appropriate place in the new heap as follows.
3. Compare *K* with its parent's key: if the latter is greater than or equal to *K*, stop (the structure is a heap);
4. otherwise, swap these two keys and compare *K* with its new parent.
5. This swapping continues until *K* is not greater than its last parent or it reaches the root.
6. In this algorithm, too, we can sift up an empty node until it reaches its proper position, where it will get *K* 's value.

HEAP CONSTRUCTION – TOP DOWN EFFICIENCY

Efficiency of insertion is O(log n)

HEAP DELETION



Maximum Key Deletion from a heap

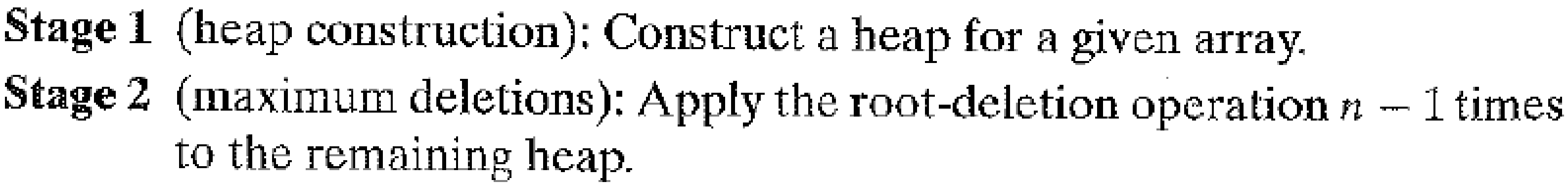
Step 1 Exchange the root's key with the last key K of the heap.

Step 2 Decrease the heap's size by 1.

Step 3 "Heapify" the smaller tree by sifting K down the tree exactly in the same way we did it in the bottom-up heap construction algorithm. That is, verify the parental dominance for K: if it holds, we are done; if not, swap K with the larger of its children and repeat this operation until the parental dominance condition holds for K in its new position.

The efficiency of deletion is O(log n).

HEAP SORT



Stage 1 (heap construction): Construct a heap for a given array.

Stage 2 (maximum deletions): Apply the root-deletion operation $n - 1$ times to the remaining heap.

HEAP SORT – EFFICIENCY

The number of key comparisons:

$$C(n) \le 2\lfloor \log_2(n-1) \rfloor + 2\lfloor \log_2(n-2) \rfloor + \cdots + 2\lfloor \log_2 1 \rfloor \le 2\sum_{i=1}^{n-1} \log_2 i$$

$$\le 2\sum_{i=1}^{n-1} \log_2(n-1) = 2(n-1)\log_2(n-1) \le 2n\log_2 n.$$

C(n) belongs to O(n log n) for the second stage of Heap Sort.

HEAP SORT – EFFICIENCY

For both stages:

O(n) + O(n log n) belongs to O(n log n).

HORNER’S RULE AND BINARY EXPONENTIATION

PROBLEM

Computing the value of a polynomial:

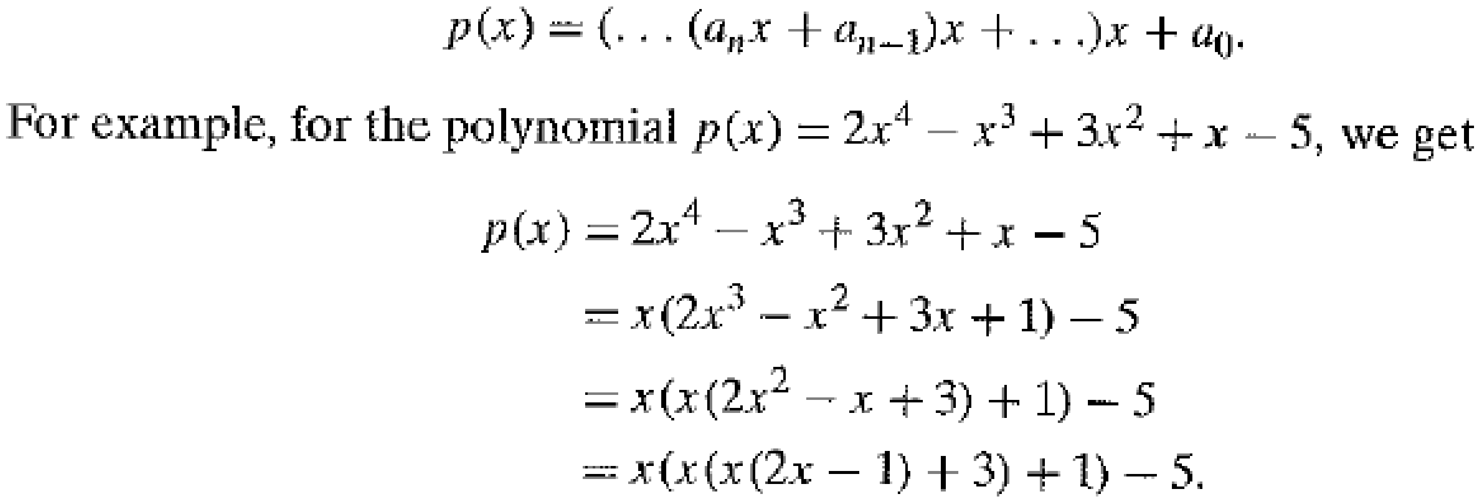
$$p(x) = a_n x^n + a_{n-1} x^{n-1} + \dots. + a_1 x + a_0$$

at a given point x and its special case of computing x^n.

HORNER’S METHOD

- ✓ Is an algorithm for calculating polynomials.
- ✓ Named after British mathematician, ***William George Horner***, who published it in the 19th century.
- ✓ An example of Representation Change variety of Transform – and – Conquer.

HORNER'S METHOD – Representation Change

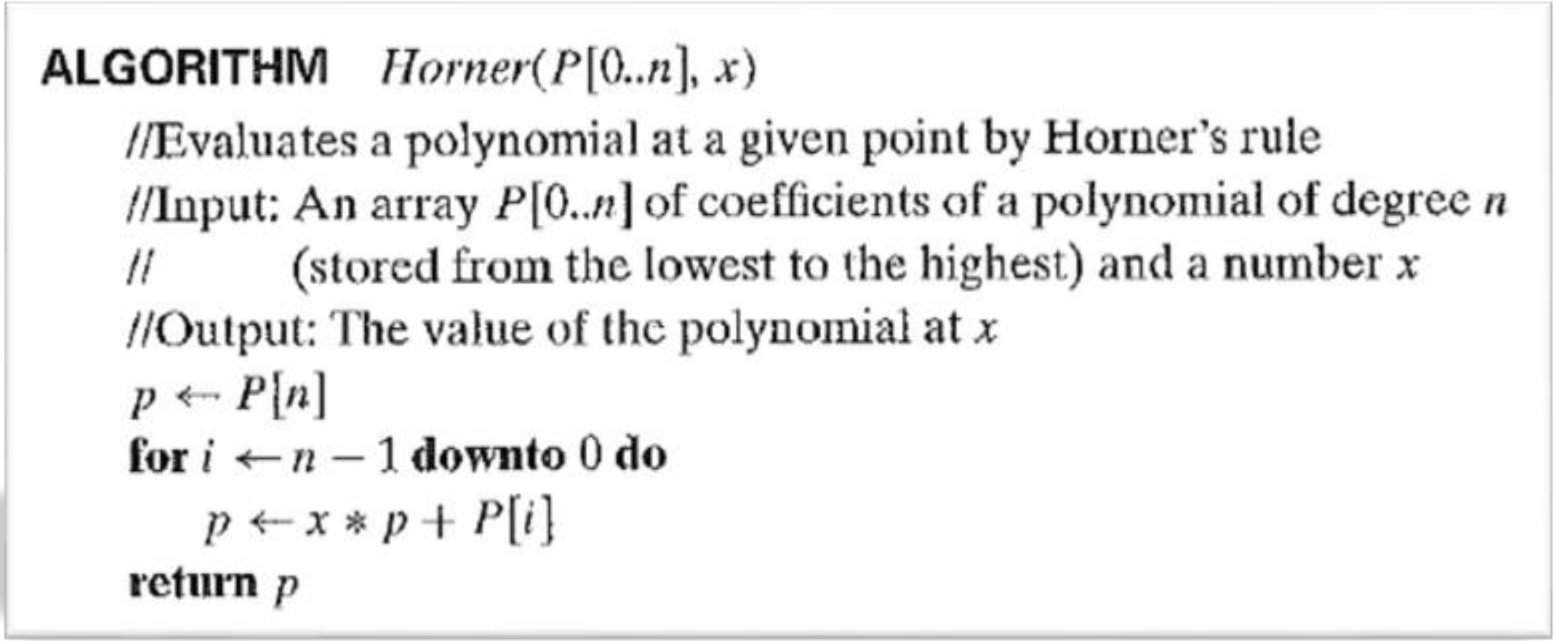


$$p(x) = (\ldots(a_n x + a_{n-1})x + \ldots)x + a_0.$$

For example, for the polynomial $p(x) = 2x^4 - x^3 + 3x^2 + x - 5$, we get

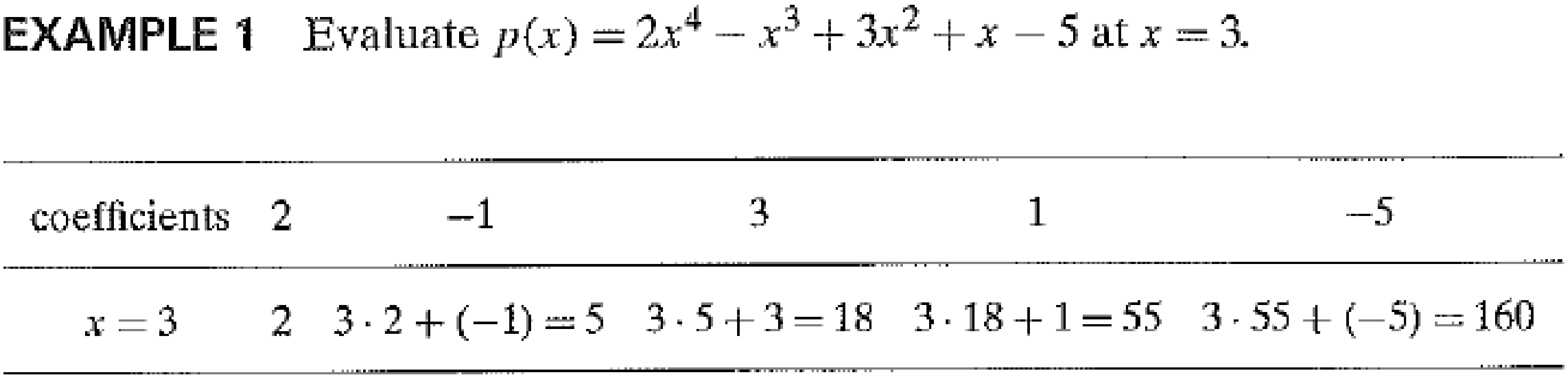
$$\begin{aligned}
p(x) &= 2x^4 - x^3 + 3x^2 + x - 5 \\
&= x(2x^3 - x^2 + 3x + 1) - 5 \\
&= x(x(2x^2 - x + 3) + 1) - 5 \\
&= x(x(x(2x - 1) + 3) + 1) - 5.
\end{aligned}$$

HORNER'S METHOD – Algorithm



```
ALGORITHM  Horner(P[0..n], x)
    //Evaluates a polynomial at a given point by Horner's rule
    //Input: An array P[0..n] of coefficients of a polynomial of degree n
    //          (stored from the lowest to the highest) and a number x
    //Output: The value of the polynomial at x
    p ← P[n]
    for i ← n − 1 downto 0 do
        p ← x * p + P[i]
    return p
```

HORNER’S METHOD – Pen and Pencil Evaluation



EXAMPLE 1 Evaluate $p(x) = 2x^4 - x^3 + 3x^2 + x - 5$ at $x = 3$.

coefficients	2	-1	3	1	-5
$x = 3$	2	$3 \cdot 2 + (-1) = 5$	$3 \cdot 5 + 3 = 18$	$3 \cdot 18 + 1 = 55$	$3 \cdot 55 + (-5) = 160$

HORNER’S METHOD – Number of Multiplications and Divisions

$$M(n) = A(n) = \sum_{i=0}^{n-1} 1 = n.$$

Comparison: Using Brute Force design strategy, just to compute a single term a^n would have required n multiplications.

PROBLEM

Computing a^n.

The Horner's Rule for computing a^n degenerates to the Brute Force multiplication of a by itself.

This can be improvised.

We see two algorithms for computing a^n which use the binary representation of the exponent.

1. The Left – to – right Binary Exponentiation
2. The Right – to – left Binary Exponentiation

SOLUTION

Computing a^n.

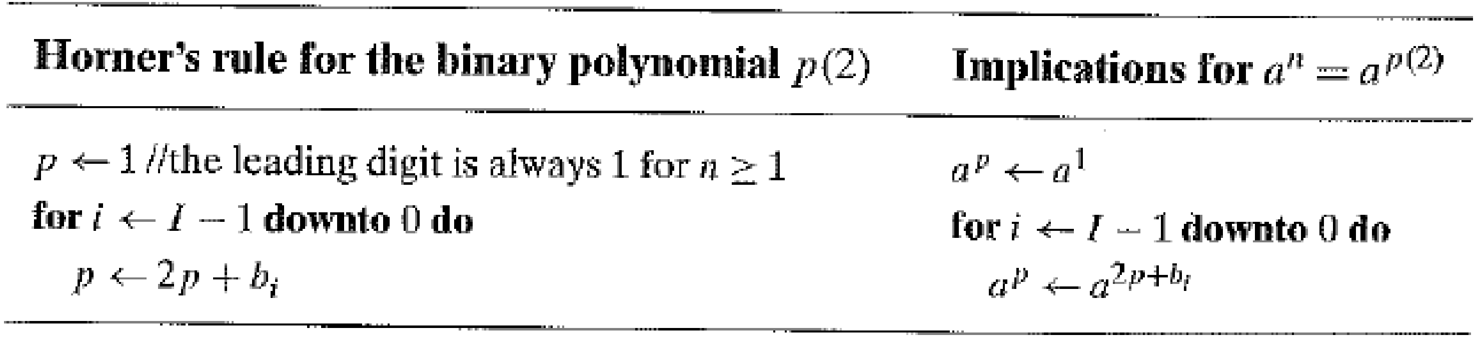
Let $n = b_1 \ldots b_i \ldots B_0$ be a bit string representing a positive integer n in the Binary Number System.

Value of n can be computed as the polynomial:

$$p(x) = b_I x^I + \cdots + b_i x^i + \cdots + b_0$$

SOLUTION

Let us now compute this polynomial by Horner’s Rule:

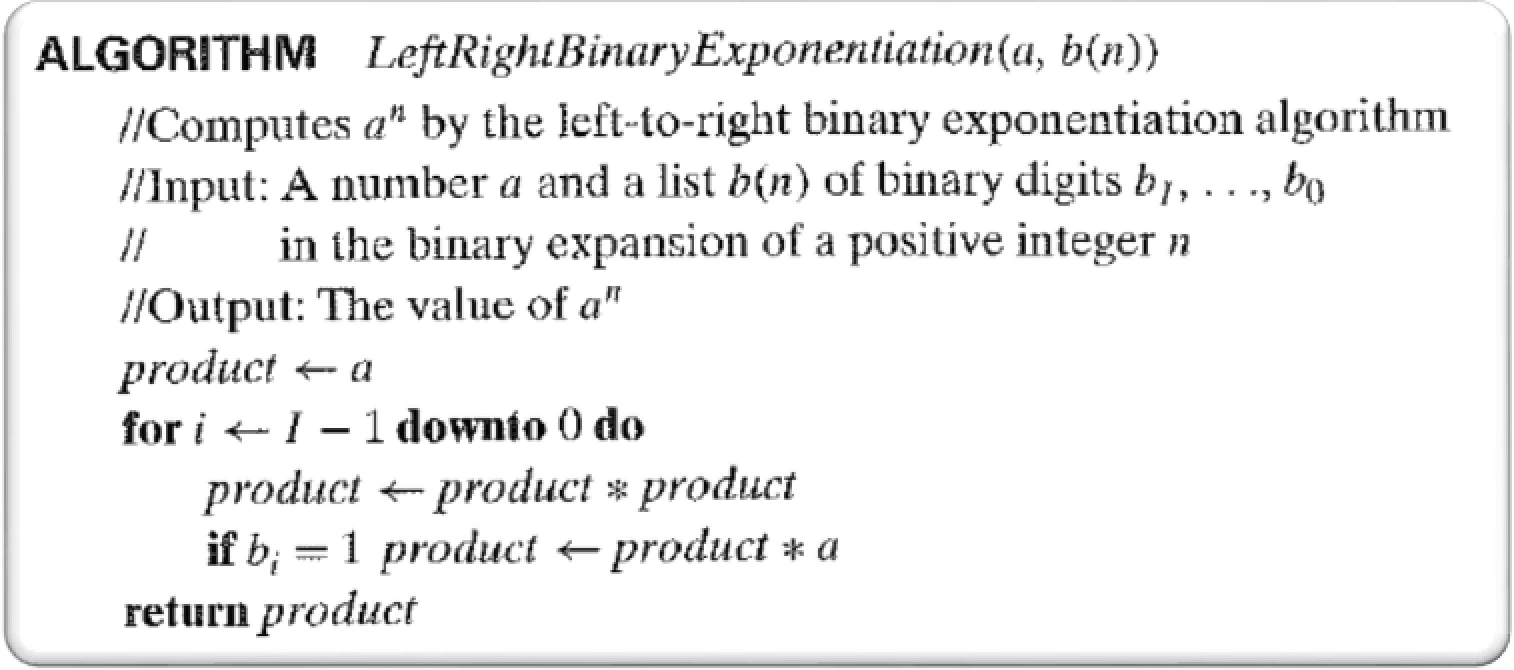


Horner’s rule for the binary polynomial $p(2)$	**Implications for** $a^n = a^{p(2)}$
$p \leftarrow 1$ //the leading digit is always 1 for $n \geq 1$ **for** $i \leftarrow I - 1$ **downto** 0 **do** $p \leftarrow 2p + b_i$	$a^p \leftarrow a^1$ **for** $i \leftarrow I - 1$ **downto** 0 **do** $a^p \leftarrow a^{2p+b_i}$

But:

$$a^{2p+b_i} = a^{2p} \cdot a^{b_i} = (a^p)^2 \cdot a^{b_i} = \begin{cases} (a^p)^2 & \text{if } b_i = 0 \\ (a^p)^2 \cdot a & \text{if } b_i = 1 \end{cases} .$$

Left – To – Right Binary Exponentiation



```
ALGORITHM LeftRightBinaryExponentiation($a$, $b(n)$)
    //Computes $a^n$ by the left-to-right binary exponentiation algorithm
    //Input: A number $a$ and a list $b(n)$ of binary digits $b_I, \dots, b_0$
    //          in the binary expansion of a positive integer $n$
    //Output: The value of $a^n$
    $product \leftarrow a$
    for $i \leftarrow I - 1$ downto 0 do
        $product \leftarrow product * product$
        if $b_i = 1$ $product \leftarrow product * a$
    return $product$
```

Left – To – Right Binary Exponentiation Efficiency

The number of multiplications M(n) is given by:

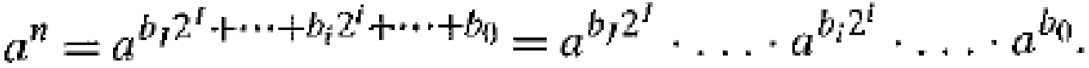
$$(b-1) \leq M(n) \leq 2(b-1),$$

b – 1 = floor(log$_2$n)

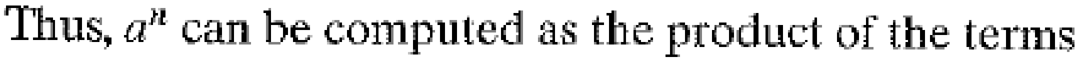
The efficiency of this algorithm is logarithmic.

YET ANOTHER SOLUTION

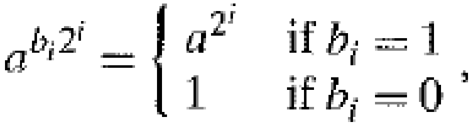
Computing a^{n}.



$$a^n = a^{b_I 2^I + \cdots + b_i 2^i + \cdots + b_0} = a^{b_I 2^I} \cdot \ldots \cdot a^{b_i 2^i} \cdot \ldots \cdot a^{b_0}.$$

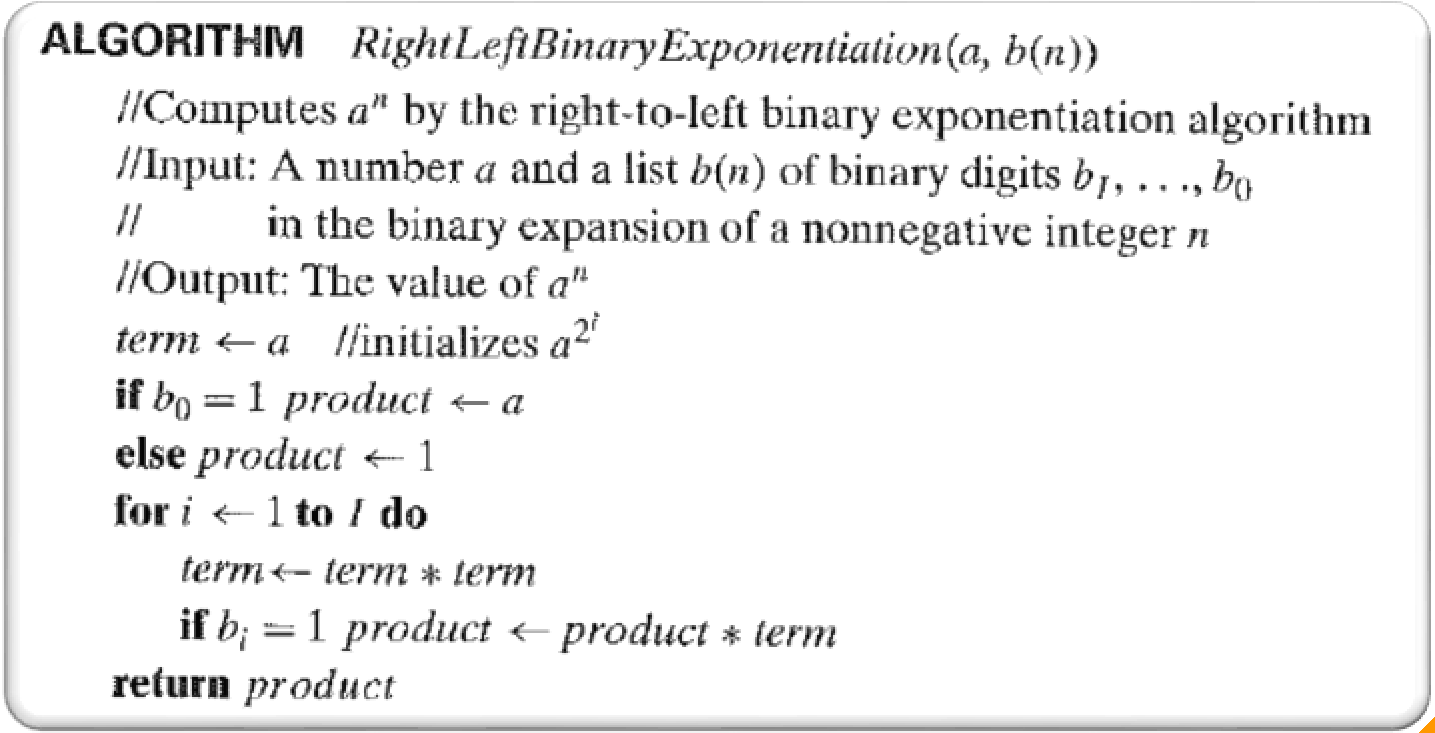


Thus, a^n can be computed as the product of the terms



$$a^{b_i 2^i} = \begin{cases} a^{2^i} & \text{if } b_i = 1 \\ 1 & \text{if } b_i = 0 \end{cases},$$

Right – To – Left Binary Exponentiation



```
ALGORITHM  RightLeftBinaryExponentiation(a, b(n))
    //Computes $a^n$ by the right-to-left binary exponentiation algorithm
    //Input: A number $a$ and a list $b(n)$ of binary digits $b_I, \ldots, b_0$
    //          in the binary expansion of a nonnegative integer $n$
    //Output: The value of $a^n$
    $term \leftarrow a$   //initializes $a^{2^i}$
    if $b_0 = 1$ $product \leftarrow a$
    else $product \leftarrow 1$
    for $i \leftarrow 1$ to $I$ do
        $term \leftarrow term * term$
        if $b_i = 1$ $product \leftarrow product * term$
    return $product$
```

Left – To – Right Binary Exponentiation Efficiency

The efficiency of this algorithm is logarithmic.