Analysis of Algorithms

Chapter 6.1, 6.4, 6.5, 6.6

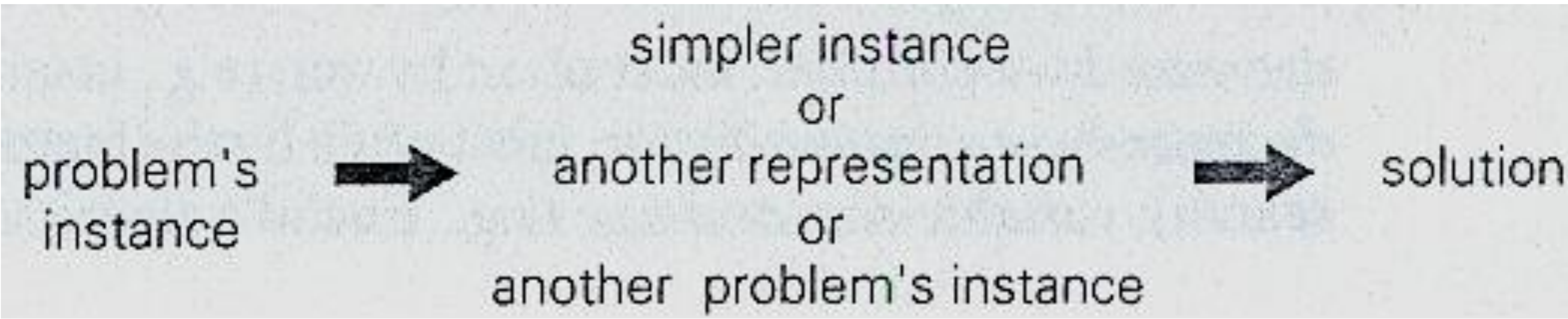
ROAD MAP

- **Transform And Conquer**
  - Instance simplification
  - Representation change
  - Problem reduction

Transform And Conquer

- *Transform and conquer technique* is based on idea of *transformation*
- This method works in two stages
  - Transformation stage
    - The problem is modified to another problem
      - more amenable to solution
  - Conquering stage
    - It is solved

Transform And Conquer Strategy



- Instance simplification
  - Transformation to a simplier instance problem
- Representation change
  - Transformation to a different representation of <u>same</u> instance
- Problem reduction
  - Tranformation to an instance of a different problem for which an algorithm is already available

ROAD MAP

- **Transform And Conquer**
  - Instance simplification
    - Presorting
      - Element Uniqueness
      - Computing Mode
      - Searching
  - Representation change
  - Problem reduction

Presorting

- Presorting is an old idea in computer science
- Many questions about a list are easier to answer if the list is sorted
- Efficiency of sorting algorithms is important
  - The benefits of a sorted list should more than the time spend for sorting.
  - Otherwise, use unsorted list directly
- We will assume that lists are implemented as *arrays*

Sorting

- We discussed three elementary sorting algorithms
  - Selection sort
  - Buble sort
  - Insertion sort

  These algorithms are *quadratic* in worst and average case
- Also discussed two advanced algorithms
  - Merge sort
    - $\Theta(nlogn)$ in worst and average case
  - Quick sort
    - $\Theta(nlogn)$ in average case
    - $\Theta(n^2)$ in worst case
- Are there faster algorithms ?
  - There is no general *comparison-based* sorting algorithm can have better efficiency than $\Theta(nlogn)$

Element Uniqueness

- Example 1 : *Checking element uniqueness in an array*
  - Brute force algorithm compare pairs of array's elements until either two equal elements were found or no pairs were left
  - Its worst case efficiency was $\Theta(n^2)$
  - Alternatively, what can we do ?

Element Uniqueness

- *Approach :*

  1. sort the array

  2. check only its consecutive elements

  If the array has equal elements, a pair of them must be next to each other

Element Uniqueness

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

- What is the running time of the algorithm ?

Element Uniqueness

- Analysis :

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

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

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

More efficient than brute-force algorithm

Computing Mode

- Example 2 : *Computing mode*

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

  For **`5,1,5,7,6,5,7`** the mode is **5**

  - In brute-force approach
    - Scan the list
    - Compute the frequencies of all distinct values
    - Find the value with largest frequency

  - How to implement this idea?

Computing Mode

- Method:
  - Store values already encountered, along with their frequencies in a separate list
  - On each iteration, the *i*th element of original list is compared with values encountered
  - If a matching value is found, its frequency is incremented
  - Otherwise, current element is added to the list of distinct values seen so far with a frequency of 1

What about analysis?

Computing Mode

- Number of comparisons depends on the input.
  - In the best case: (all the elements are same)

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

  - In worst case: (all the elements are different)

$$C(n) = \sum_{i=1}^{n}(i-1) = 0 + 1 + \ldots + (n-1)$$

$$C(n) = \frac{n(n-1)}{2}$$

$$C(n) \in \Theta(n^2)$$

What can we do as an alternative ?

Computing Mode

- Approach :

1. Sort the input

   Then all equal values will be adjacent to each other

2. Find the longest run of adjacent equal values in the sorted array

Computing Mode

```
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 ← 0                    //current run begins at position i
    modefrequency ← 0        //highest frequency seen so far
    while i ≤ n − 1 do
        runlength ← 1; runvalue ← A[i]
        while i + runlength ≤ n − 1 and A[i + runlength] = runvalue
            runlength ← runlength + 1
        if runlength > modefrequency
            modefrequency ← runlength; modevalue ← runvalue
        i ← i + runlength
    return modevalue
```

Computing Mode

- Analysis :

  - Running time of algorithm depends on the time spent on sorting
    - remainder of the algorithm takes linear time (why ?)

  - So, with an $\Theta(n\log n)$ sort, worst case efficiency will be $\Theta(n\log n)$

Searching Problem

- Example 3 : *Searching Problem*
  - Searching for a given value *v* in a given array of *n* sortable items
  - Brute force solution is sequential search
    - needs *n* comparisons in worst case
  - If the array is sorted, we apply binary search
    - requires only $\lfloor \log_2 n \rfloor + 1$ comparisons in worst case

Searching Problem

- Assume the most efficient $\Theta(n\log n)$ sort is used
- Total running time in worst case and also average case will be

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

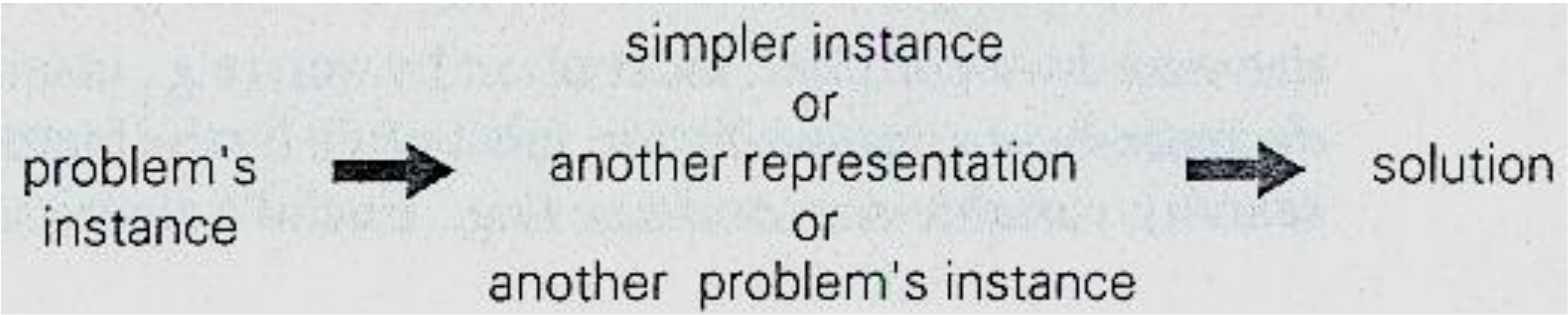
- Worst than sequential search!...
- What if the search will be done several times?...

Presorting

Discussion:

- Geometric algorithms dealing with sets of points use presorting in one way or another
  - Presorting is used in divide and conquer for closest pair problem and convex-hull problem
- Some problems for directed acyclic graphs can be solved more easily after topologically sorting the digraph
  - Finding the shortest and longest paths

Transform And Conquer Strategy



- Instance simplification
  - Transformation to a simplier instance problem
- **Representation change**
  - **Transformation to a different representation of <u>same</u> instance**
- Problem reduction
  - Tranformation to an instance of a different problem for which an algorithm is already available

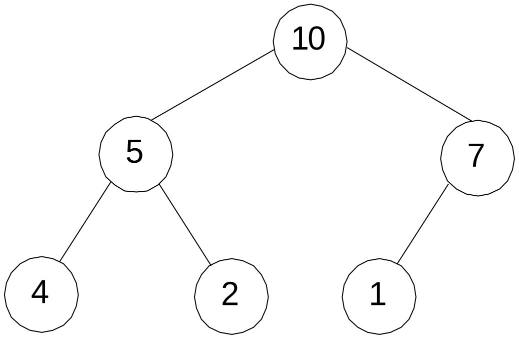
ROAD MAP

- **Transform And Conquer**
  - Instance simplification
  - Representation change
    - **Heaps and Heapsort**
    - Horner’s Rule and Binary Exponentiation
  - Problem Reduction

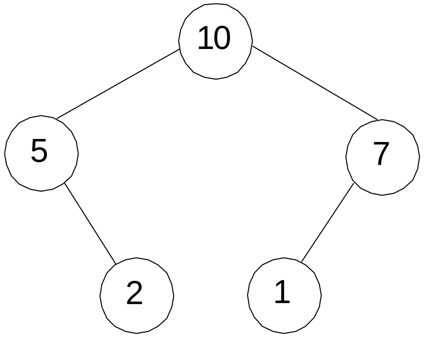
Heaps

- Heap is an important data structure, suitable for implementing *priority queues*
- Priority queue is a multiset of items with an orderable charecteristics called an item's *priority*
  - find an item with highest priority
  - delete an item with highest priority
  - add a new item to multiset
- Heap is representation change over regular list
  - Provides efficient algorithms for basic operations
- Heap also serves an important sorting algorithm called *heapsort*

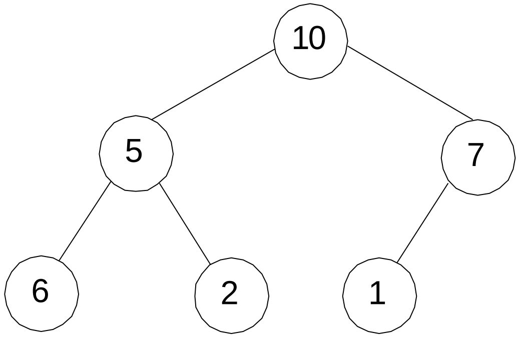
Illustration of the heap's definition



a heap



not a heap



not a heap

Some Important Properties of a Heap

- Given *n,* there exists a unique binary tree with *n* nodes that

  is essentially complete, with $h = \lfloor \log_2 n \rfloor$

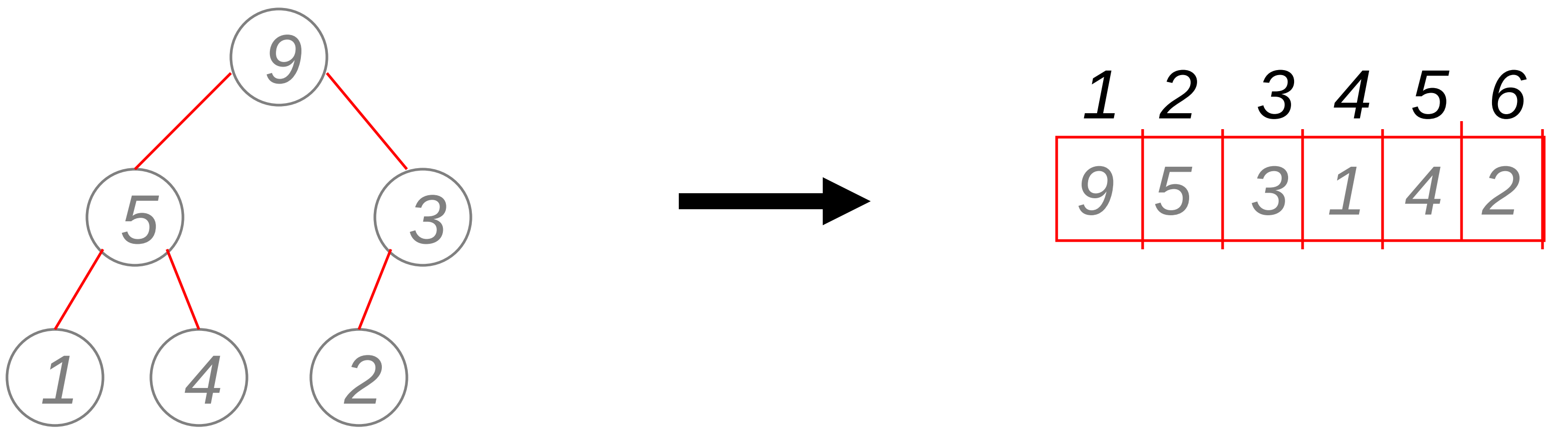
- The root contains the largest key

- The subtree rooted at any node of a heap is also a heap

- A heap can be represented as an array

Heap's Array Representation

Store heap's elements in an array (whose elements indexed, for convenience, 1 to *n*) in top-down left-to-right order. Example:



- Left child of node *j* is at 2*j*
- Right child of node *j* is at 2*j*+1
- Parent of node *j* is at $\lfloor j/2 \rfloor$
- Parental nodes are represented in the first $\lfloor n/2 \rfloor$ locations

Heap Construction (bottom-up)

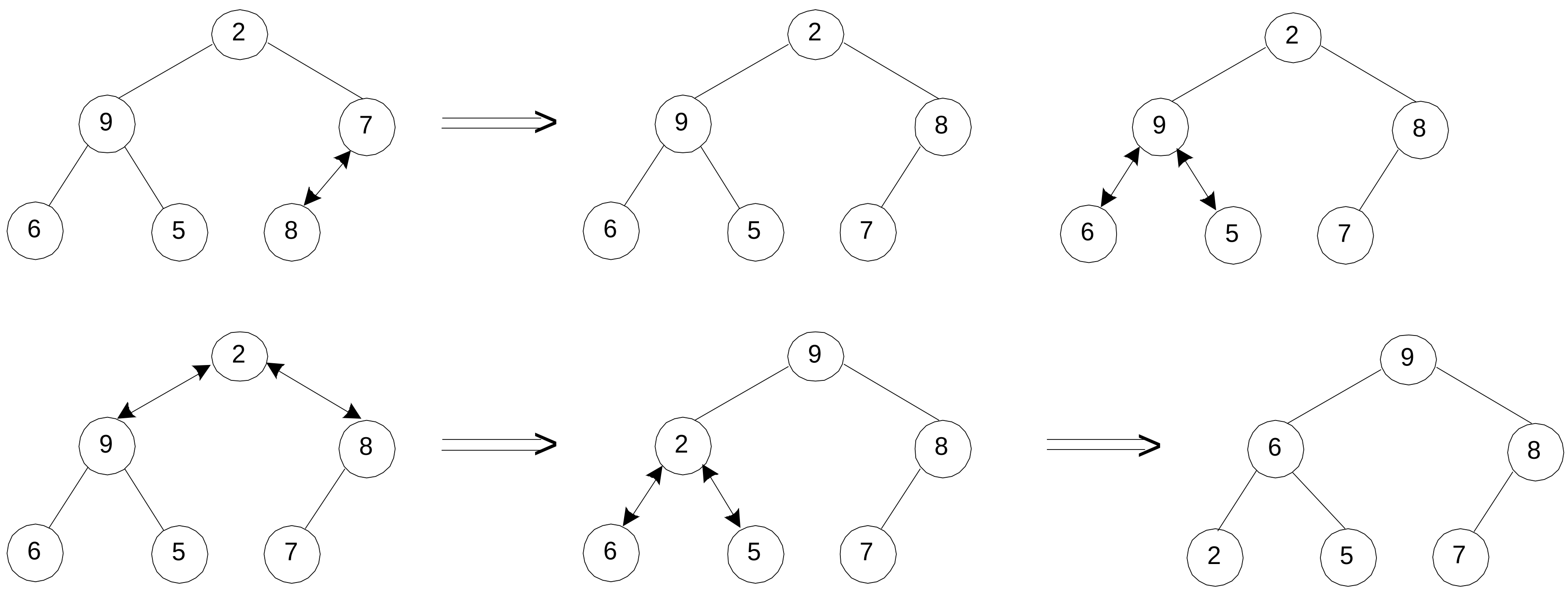
Step 0: Initialize the structure with keys in the order given

Step 1: Starting with the last (rightmost) parental node, fix the heap rooted at it, if it doesn't satisfy the heap condition: keep exchanging it with its largest child until the heap condition holds

Step 2: Repeat Step 1 for the preceding parental node

Example of Heap Construction

Construct a heap for the list 2, 9, 7, 6, 5, 8



Bottom-up heap construction

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**
    $k \leftarrow i; \quad v \leftarrow H[k]$
    $heap \leftarrow$ **false**
    while not $heap$ **and** $2 * k \leq n$ **do**
        $j \leftarrow 2 * k$
        if $j < n$ //there are two children
            if $H[j] < H[j+1] \quad j \leftarrow j+1$
        if $v \geq H[j]$
            $heap \leftarrow$ **true**
        else $H[k] \leftarrow H[j]; \quad k \leftarrow j$
    $H[k] \leftarrow v$

Heapsort

- Heapsort is an interesting two-stage algorithm
  - 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
  - As a result the array's elements are eliminated in decreasing order
  - Since under the array implementation, an element being deleted is placed last, the resulting array will be exactly the original array sorted in ascending order

Stage 1 (heap construction)

2 9 **7** 6 5 8

2 **9** 8 6 5 7

2 9 8 6 5 7

9 **2** 8 6 5 7

9 6 8 2 5 7

Stage 2 (maximum deletions)

9 6 8 2 5 7

7 6 8 2 5 | **9**

8 6 7 2 5

5 6 7 2 | **8**

7 6 5 2

2 6 5 | **7**

6 2 5

5 2 | **6**

5 2

2 | **5**

2

Sorting array 2, 9, 7, 6, 5, 8 by heapsort

Heapsort

- Analysis :
  - We know that heap construction stage is **O(n)**
    - Stage 1
  - What about stage 2 ?

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

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

  - For both stages *O(n)* + *O(nlogn)* = *O(nlogn)*

Heapsort

- <u>Discussion :</u>
  - Average case efficiency is also ***Θ(nlogn)***
    - such as mergesort
  - Heapsort does not require an extra storage
  - Timing experiments on random files show that heapsort runs more slowly than quicksort but it is competitive with mergesort

ROAD MAP

- **Transform And Conquer**
  - Instance simplification
  - Representation change
    - Heaps and Heapsort
    - **Horner’s Rule and Binary Exponentiation**
  - Problem Reduction

Horner's Rule

Problem Definition:

- Compute the value of a polynomial

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

  at a given point *x*

- Polynomials constitute the most important class of functions
  - They posses a wealth of good properties
  - Can be used for approximating other types of functions
- Manipulating polynomials efficiently is an important problem

Horner's Rule

- Horner's rule provides elegant method for evaluating a polynomial
- It is a good example of representation change technique since it is based on representing P(x) by a formula

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

Horner's Rule

- Example :

For example, for the polynomial

$$p(x) = 2x^4 - x^3 + 3x^2 + x - 5$$ we get

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

Horner's Rule

- The pen-and-pencil calculation can be conveniently organized with a two row table
  - First row contains the polynomial's coefficients listed from the highest a_n to the lowest a_0
  - Second row is filled from left to right as follows (except its first entry which is a_n)
    - Next entry is computed as the *x*'s value times the last entry in the second row plus the next coefficient from first row
  - Final entry is the value being sought

Horner's Rule

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

P(3) = 160

3.2+(-1) → 2x-1 at x=3

3.5+3 = 18 → x(2x-1)+3 at x=3

3.18+1 = 55 → x(x(2x-1)+3)+1 at x=3

3.55+(-5) = 160 → x(x(x(2x-1)+3)+1)-5 = p(x)

Horner’s Rule

```
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 Rule

- Analysis :
  - Number of multiplications and number of additions

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

  - So how efficient is Horner's rule ?

Horner's Rule

- Analysis :
  - Consider only the first term of a polynomial of degree n : $a_n x^n$
  - Just computing this term with brute force approach requires n multiplications
    - Horner's rule computes *n-1* other terms in addition to this and still uses the same number of multiplications
  - So it is an optimal algorithm for polynomial evaluation

Horner's Rule

- Discussion:
  - Horner's rule also has some useful by-products
  - The intermediate numbers generated by the algorithm in the process of evaluating P(x) at some point x_0 turn out to be the coefficient to the quotient of the division of P(x) by $x-x_0$
    - While the final result, in addition to being $P(x_0)$ is equal to the remainder of this division of

$$P(x) = P'(x)\,(x-x_0) + P(x_0)$$

$$\mathbf{2x^4 - x^3 + 3x^2 + x - 5} \quad \text{by} \quad \mathbf{x-3}$$

$$\mathbf{2x^3 + 5x^2 + 18x + 55} \quad \text{and} \quad \mathbf{160}$$

  - This division algorithm is known as *synthetic division*
    - It is more convenient than long division

Exponentiation

- Problem Definition :
  - Compute a^n

  - Computing a^n in an essential operation in *primality-testing* and *encryption methods*
  - The brute-force algorithm takes linear time
  - Designing other algorithms for computing a^n is important
  - For example, based on the representation change idea

Binary Exponentiation

- We will consider two algorithms for computing a^n
- Both of them exploit the binary representation of exponent n
  - One of them processed this processes this binary string left to right
  - The second does it right to left

Binary Exponentiation

- Let

$$n = b_I \; \dots \; b_i \; \dots \; b_0$$

be the string representation of a positive integer *n* in binary system

The value of *n* can be computed as the value of polynomial at $x = 2$

$$P(x) = b_I x^I + \dots + b_i x^i + \dots + b_0$$

If $n = 13$ its binary representation is *1101* and

$13 = 1.2^3 + 1.2^2 + 0.2^1 + 1.2^0$

Binary Exponentiation

- If we compute the value of *P(x)* with Horner's rule

$$a^n = a^{p(2)} = a^{b_I 2^I + \cdots + b_i 2^i + \cdots + b_0}$$

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 $n \geq 1$	$a^p \leftarrow a^1$
for $i \leftarrow I - 1$ **downto** 0 **do**	**for** $i \leftarrow I - 1$ **downto** 0 **do**
$p \leftarrow 2p + b_i$	$a^p \leftarrow a^{2p+b_i}$

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

Binary Exponentiation

- After initializing the accumulator's value to *a*,
  - the bit string representing the exponent is always square the last value of accumulator
  - if the current binary digit is *1*, also multiply it by *a*
- These observations lead to *left-to-right exponentiation* method of computing an

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 $b_I, \ldots, b_0$
// 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

- <u>Example :</u>
  - Compute a^{13} by left-right binary exponentiation
    - Here $n = 13 = (1101)_2$
    - So

binary digits of n	1	1	0	1
product accumulator	a	$a^2 \cdot a = a^3$	$(a^3)^2 = a^6$	$(a^6)^2 \cdot a = a^{13}$

Left-to-right binary exponentiation

- Analysis :

Total number of multiplications M(n)

$$(b-1) \le M(n) \le 2(b-1)$$

- b is the length of bit string representing exponent n
- b-1 = logn

So efficiency is ***Θ(logn)***

Left-to-right binary exponentiation

- Discussion :
  - This algorithm is better efficiency class than brute-force exponentiation
    - requires *n-1* multiplications

Right-to-left binary exponentiation

- <u>Definition:</u>
  - Right-to-left binary exponentiation uses same binary polynomial p(2) yielding value of n
  - But it does not apply Horner's rule
    - Exploits it differently

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

Right-to-left binary exponentiation

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

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

  - The product of consecutive terms a^{2i}, skipping those for which the binary digit bi is zero
  - We can compute a^{2i} by simply squaring the same term we computed for the previous value of *i* since

$$a^{2i} = (a^{2^{i-1}})^2$$

  - We compute powers of a right to left (smallest to largest)

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 ← a    //initializes a^(2^i)
if b_0 = 1 product ← a
else product ← 1
for i ← 1 to I do
    term ← term * term
    if b_i = 1 product ← product * term
return product
```

Right-to-left binary exponentiation

- Example :
  - Compute a^{13} by right-to-left binary exponentiation
    - Here n = 13 = 1101
    - So

1	1	0	1	binary digits of n
a^8	a^4	a^2	a	terms a^{2^i}
$a^5 \cdot a^8 = a^{13}$	$a \cdot a^4 = a^5$		a	product accumulator

Right-to-left binary exponentiation

- Analysis :
  - Efficiency is *logaritmic*
    - Same as left-to-right binary multiplications

Binary Exponentiation

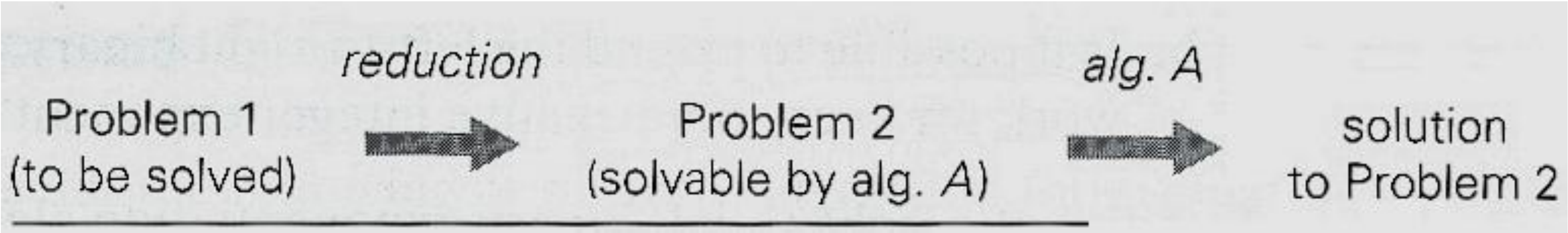
- Discussion:
  - Both binary exponentiation algorithm discussed reduce somewhat by their reliance on availability of the explicit binary expansion of exponent *n*

ROAD MAP

- **Transform And Conquer**
  - Instance simplification
  - Representation change
  - **Problem Reduction**
    - **Computing The Least Common Multiple**
    - **Counting Paths in A Graph**

Problem Reduction

- **Definition:**
  - Problem reduction is to reduce a problem you need to solve to *another* problem that you know how to solve
    1. Find a problem to reduce onto
    2. Perform reduction



  - The reduction worth if the reduction operations and algorithm A takes less time than solving the original problem directly

Least Common Multiple

- Definition :
  - Computing the least common multiple of two integers *m* and *n* is denoted *lcm(m,n)*
  - *lcm* is defined as the smallest integer that is divisible by both *m* and *n*
    - lcm (24, 60) = 120
    - lcm (11,5) = 55
  - It is an important notion in arithmetic and algebra

Computing the Least Common Multiple

- Approach :
  - Given the prime factorizations of m and n, $lcm(m,n)$ can be computed as the product of all the common prime factors of m and n times the product of m's prime factors that are not in n times n's prime factors that are not in m

    $24 = 2 . 2 . 2 . 3$

    $60 = 2 . 2 . 3 . 5$

    $lcm(24, 60) = (2 . 2 . 3) . 2 . 5 = 120$

Computing the Least Common Multiple

- As a computational procedure, this algorithm has the same drawbacks as middle-school algorithm for computing greatest-common-divisor

  How can we design a more efficient algorithm by using problem reduction ?

Computing the Least Common Multiple

- Product of lcm(m,n) and gcd(m,n) includes every factor of m and n exactly once
- So,

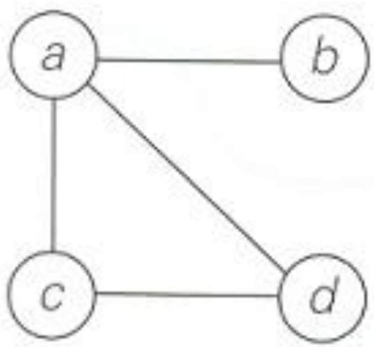
$$\operatorname{lcm}(m,n) = \frac{m.n}{\gcd(m,n)}$$

- This formula reduces lcm calculation to gcd calculation
- gcd(m,n) can be computed with Euclid’s algorithm efficiently

Counting Paths in a Graph

- Definition:
  - Counting different paths between two vertices in a graph
  - It is easy to prove that number of different paths of length $k>0$ from the ith vertex to the jth vertex of a graph equals the (i,j) th element of A^k where A is the adjacency matrix of the graph

Counting Paths in a Graph



$$A = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{bmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \end{bmatrix} \end{array} \qquad A^2 = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{bmatrix} 3 & 0 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 1 & 1 & 2 & 1 \\ 1 & 1 & 1 & 2 \end{bmatrix} \end{array}$$

a graph its adjacency matrix A its square A^2

Elements of A and A^2 indicate the number of paths of lengths 1 and 2

Counting Paths in a Graph

- So, the problem can be solved with an algorithm for computing an appropriate power of its adjacency matrix
- Problem is reduced to matrix exponentiation
  - How to calculate A^k

Problem Reduction

- Discussion:
  - Plays a central role in theoretical computer science
    - where it is used to classify problems according to their complexity
  - The practical difficulty is finding a problem to which the problem at hand should be reduced
  - If we want our efforts to be of practical value, we need our reduction-based algorithm to be more efficient than solving the original problem directly