Introduction to Algorithm Design

Lecture Notes 7

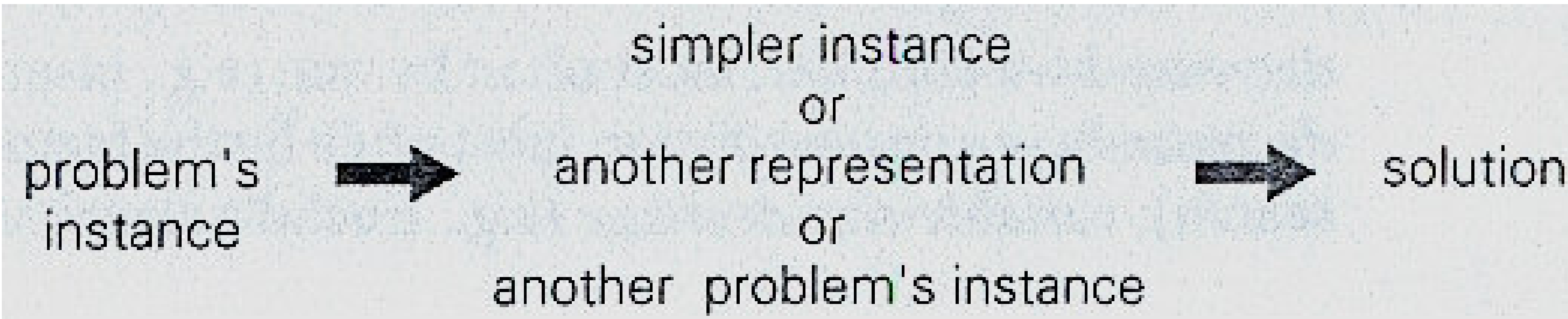
ROAD MAP

- **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 problem instance
- 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
    - Gaussian Elimination
  - 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 $\boldsymbol{\Theta(nlogn)}$

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

- *<u>Approach :</u>*

  1. sort the array

  2. check only its consecutive elements

  If the array has equal elements, they 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

- <u>Example 3 :</u> *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

$$
\begin{aligned}
T(n) &= T_{sort}(n) + T_{search}(n) \\
&= \Theta(n\log n) + \Theta(\log n) = \Theta(n\log n)
\end{aligned}
$$

- 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

ROAD MAP

- **Transform And Conquer**
  - Instance simplification
    - Presorting
    - **Gaussian Elimination**
      - Solving Linear System of Equations
      - LU Decomposition
      - Computing a Matrix Inverse
      - Computing a Determinant
  - Representation change
  - Problem Reduction

Solving Linear System of Equations

- A system of two linear equations in two unknowns

$$a_{11}x + a_{12}y = b_1$$
$$a_{21}x + a_{22}y = b_2$$

- The system has a unique solution
  - unless the coefficients of one equation are proportional to the coefficients of the other

Solving Linear System of Equations

- To find the solution
  - Express one of the variables as a function of the other
  - Substitute the result into the other equation
    - yielding a linear equation
  - Solve it to find the value of the first variable.
  - Use the solution to find the value of the second variable.
- What if the number of variables is large!

Solving Linear System of Equations

- In many applications, we need to solve a system of *n* equations in *n* unknowns

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

- Such a system can be solved by generalizing the *substitution* method
  - However, the resulting algorithm would be extremely *cumbersome*

Solving Linear System of Equations

- In many applications, we need to solve a system of *n* equations in *n* unknowns

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

Solving Linear System of Equations

- **Gaussian elimination** is an elegant algorithm to solve systems of linear equations
- This method is named after Carl Frederich Gauss
- Idea of Gaussian elimination is to tranform a system of n linear equations in n unknowns to an equivalent system
  - with an upper triangular coefficient matrix
    - a matrix with all zeros below its main diagonal
- Mathematical formulation is as follows:

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

$$
Ax = b \Rightarrow A'x = b
$$

$$
A = \begin{bmatrix} a_{11} & a_{12} & \ldots & a_{1n} \\ a_{21} & a_{22} & \ldots & a_{2n} \\ \vdots & & & \\ a_{n1} & a_{n2} & \ldots & a_{nn} \end{bmatrix}, \quad b = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_4 \end{bmatrix} \quad A' = \begin{bmatrix} a'_{11} & a'_{12} & \ldots & a'_{1n} \\ 0 & a'_{22} & \ldots & a'_{2n} \\ \vdots & & & \\ 0 & 0 & \ldots & a'_{nn} \end{bmatrix}, \quad b = \begin{bmatrix} b'_1 \\ b'_2 \\ \vdots \\ b'_n \end{bmatrix}
$$

Gaussian Elimination

- What is the use of such a transformation?
- Easier to solve upper-triangular system by back-substitution
  - Solve the last equation to find the value of x_n
  - Substitute x_n into the next to last equation and find the value of x_{n-1}
  - Continue similarly until the first equation
  - For the first equation, substitute the values of the last n-1 variables into the first equation and find the value of x_1

Gaussian Elimination

- How to transform a system to an equivalent upper-triangular system?
- Perform a series of operations
  - Exchange two equations
  - Replace an equation with its nonzero multiple
  - Replace an equation with a sum or difference of this equation and some multiple of another equation
- These operations do not change the solution of the system

Gaussian Elimination

- How to use operations for a transformation?
- Idea:
  - Use a_{11} as a pivot
  - In each equations except the first, make the coefficient of x_1 to be zero.
    - Perform the last operation on equations *i* with multiple of a_{i1}/a_{11}
  - Do the same for all coefficient on the diagonal.

Gaussian Elimination

- Example :

  Solve the system by Gaussian elimination

$$
\begin{aligned}
2x_1 - x_2 + x_3 &= 1 \\
4x_1 + x_2 - x_3 &= 5 \\
x_1 \quad + x_2 + x_3 &= 0
\end{aligned}
$$

$$\begin{bmatrix} 2 & -1 & 1 & 1 \\ 4 & 1 & -1 & 5 \\ 1 & 1 & 1 & 0 \end{bmatrix} \begin{matrix} \\ \text{row 2 - 4/2 row1} \\ \text{row 3 - 1/2 row 1} \end{matrix}$$

$$\begin{bmatrix} 2 & -1 & 1 & 1 \\ 0 & 3 & -3 & 3 \\ 0 & 3/2 & 1/2 & -1/2 \end{bmatrix} \begin{matrix} \\ \\ \text{row 3 - 1/2 row 2} \end{matrix}$$

$$\begin{bmatrix} 2 & -1 & 1 & 1 \\ 0 & 3 & -3 & 3 \\ 0 & 0 & 2 & -2 \end{bmatrix}$$

$$\begin{bmatrix} 2 & -1 & 1 & 1 \\ 0 & 3 & -3 & 3 \\ 0 & 0 & 2 & -2 \end{bmatrix}$$

Use back-substitution to solve the system

$$x_3 = (-2)/2 = -1$$

$$x_2 = (3 - (-3)x_3)/3 = 0$$

$$x_1 = (1 - x_3 - (-1)x_2)/2 = 1$$

Gaussian Elimination

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

- <u>Observations:</u>
  - What if A[i,i]=0
    - Can not use the *i*th row as a pivot
      - Division by zero
    - Solution: change the *i*th row with some row below it has a nonzero coefficient in the *i*th column
  - What if A[i,i] is too small
    - Round-off error because of division by a small value
    - Solution: change the *i*th row with a row with largest absolute value of the coefficient in the *i*th column
  - This modification is called *partial pivoting*

Gaussian Elimination

```
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 ← 1 to n do A[i, n + 1] ← b[i]   //appends b to A as the last column
    for i ← 1 to n − 1 do
        pivotrow ← i
        for j ← i + 1 to n do
            if |A[j, i]| > |A[pivotrow,i]| pivotrow ← j
        for k ← i to n + 1 do
            swap(A[i, k], A[pivotrow,k])
        for j ← i + 1 to n do
            temp ← A[j, i] / A[i, i]
            for k ← i to n + 1 do
                A[j, k] ← A[j, k] − A[i, k] * temp
```

Gaussian Elimination

Analysis :

- by assuming multiplication as the basic operation

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

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

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

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

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

LU Decomposition

- Gaussian elimination provides more than just a solution to a system of equations
- It also produces LU decomposition of the coefficient matrix as a useful by-product

- What is LU decomposition
  - By an example:

LU Decomposition

$$A = \begin{bmatrix} 2 & -1 & 1 \\ 4 & 1 & -1 \\ 1 & 1 & 1 \end{bmatrix}$$

consider the lower-triangular matrix *L* made up of 1's on its main diagonal and row multiplies used in the Gaussian elimination process

$$L = \begin{bmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ \frac{1}{2} & \frac{1}{2} & 1 \end{bmatrix}$$

consider the upper-triangular matrix U that was the result of the elimination

$$U = \begin{bmatrix} 2 & -1 & 1 \\ 0 & 3 & -3 \\ 0 & 0 & 2 \end{bmatrix}$$

The product LU is equal to matrix A

LU Decomposition

- Solving the system $\mathbf{Ax = b}$ is equivalent to solving the system $\mathbf{LUx = b}$
- The latter system can be solved as follows
  - Denote $\boldsymbol{y=Ux}$, then $\boldsymbol{Ly = b}$
  - First solve the system $\boldsymbol{Ly = b}$
    - It is easy because *L* is a lower-triangular matrix
  - Then solve the system $\boldsymbol{Ux = y}$
    - Upper-triangular matrix *U* to find *x*

- We first solve **Ly = b**

$$\begin{bmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ \frac{1}{2} & \frac{1}{2} & 1 \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix} = \begin{bmatrix} 1 \\ 5 \\ 0 \end{bmatrix}$$

- Its solution is

$$y_1 = 1, \quad y_2 = 5 - 2y_1 = 3, \quad y_3 = 0 - \frac{1}{2}y_1 - \frac{1}{2}y_2 = -2.$$

- Solving Ux=y means solving

$$\begin{bmatrix} 2 & -1 & 1 \\ 0 & 3 & -3 \\ 0 & 0 & 2 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 1 \\ 3 \\ -2 \end{bmatrix}$$

- So the solution is

$$x_3 = (-2)/2 = -1, \quad x_2 = (3 - (-3)x_3)/3 = 0, \quad x_1 = (1 - x_3 - (-1)x_2)/2 = 1$$

LU Decomposition

- Discussion :
  - Once we have the LU decomposition of matrix A, we can solve the systems $Ax = b$ with as many right-hand side vectors b as we want to
    - This is a distinct advantage over the classic Gaussian elimination discussed earlier
  - LU demcoposition does not require extra memory
    - We can store nonzero part of U in the upper triangular part of A
    - We can store nontrivial part of L below the main diagonal of A

Computing Matrix Inverse

Definition :

- The inverse of an n-by-n matrix *A* is an n-by-n matrix denoted $\mathbf{A}^{-1}$ such that

  $$AA^{-1} = I$$

- *I* is the n-by-n identity matrix
  - Matrix with all zero elements except main diagonal elements which are all ones

- *A matrix inverse* can be used to solve a linear system of equations
  - Solution of $Ax=b$ is obtained by $x=A^{-1}b$

Singularity Check

- Not every square matrix has an inverse
  - When it exists the inverse is unique
- If a matrix does not have an inverse, it is called ***singular***
  - A matrix is singular iff one of its rows is a linear combination of other rows
- The way to check whether a matrix is nonsigular is to apply Gaussian elimination
  - Matrix is nonsingular if it yields an upper-triangular matrix with no zero on the main diagonal

Computing Matrix Inverse

- To compute the inverse matrix for a nonsingular *n-by-n* matrix *A* we need to find n^2 numbers x_{ij}, $1 \le i, j \le n$ such that

$$\begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & & & \\ a_{n1} & a_{n2} & \dots & a_{nn} \end{bmatrix} \begin{bmatrix} x_{11} & x_{12} & \dots & x_{1n} \\ x_{21} & x_{22} & \dots & x_{2n} \\ \vdots & & & \\ x_{n1} & x_{n2} & \dots & x_{nn} \end{bmatrix} = \begin{bmatrix} 1 & 0 & \dots & 0 \\ 0 & 1 & \dots & 0 \\ \vdots & & & \\ 0 & 0 & \dots & 1 \end{bmatrix}$$

Computing a Matrix Inverse

- We can find the unknowns by solving *n* systems of linear equations that have the same coefficient matrix *A*
  - the vector of unknowns x^j is the jth column of the inverse
  - the right-hand side vector e^j is the jth column of identity matrix ($1 \leq j \leq n$)

$$Ax^j = e^j.$$

- We can use LU decomposition to solve above equation for all *j* values

OR

- We can solve these systems by applying Gaussian elimination to matrix A augmented by the *n-by-n* identity matrix

Computing a Determinant

- Definition :
  - The determinant of an n-by-n matrix A denoted detA or |A| is a number whose value can be defined recursively as follows :
    - If n=1 i.e., n consists of a single element a_{11}
      - det A = a_{11}
    - For n>1
      - det A is computed by formula

$$\det A = \sum_{j=1}^{n} s_j a_{1j} \det A_j$$

s_j is +1 if j is odd and -1 if j is even
a_{1j} is the element in row 1 and column j
A_j is the (n-1)-by-(n-1) matrix
obtained from matrix A by deleting its row 1 and column j

- For a 2-by-2 matrix

$$\det \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} = a_{11} \det [a_{22}] - a_{12} \det [a_{21}] = a_{11}a_{22} - a_{12}a_{21}$$

- For a 3-by-3 matrix

$$\det \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix}$$

$$= a_{11} \det \begin{bmatrix} a_{22} & a_{23} \\ a_{32} & a_{33} \end{bmatrix} - a_{12} \det \begin{bmatrix} a_{21} & a_{23} \\ a_{31} & a_{33} \end{bmatrix} + a_{13} \det \begin{bmatrix} a_{21} & a_{22} \\ a_{31} & a_{32} \end{bmatrix}$$

$$= a_{11}a_{22}a_{33} + a_{12}a_{23}a_{31} + a_{21}a_{32}a_{13} - a_{31}a_{22}a_{13} - a_{21}a_{12}a_{33} - a_{32}a_{23}a_{11}.$$

What if we need to compute a determinant of a large matrix ?

Computing a Determinant

- The algorithm using the recursive definition requires $O(n!)$ time
- Gaussian elimination comes to resque again!
  - Basic operations used in Gaussian elimination changes the determinant
    - Changes the sign
    - Multiplied by a constant used in elimination
  - The determinant of an upper-triangular matrix is equal to the product of elements on its main diagonal
- The determinant of an n-by-n matrix can be calculated in cubic time

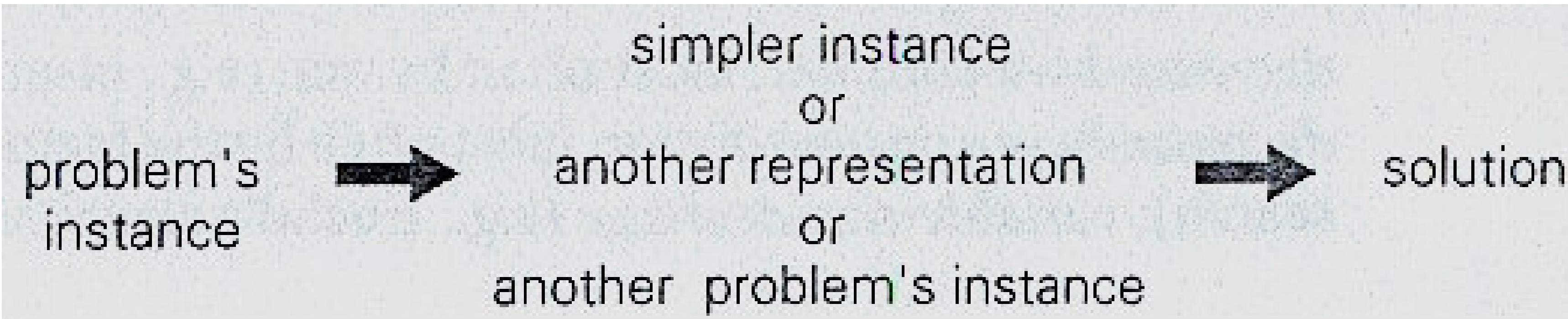
Computing a Determinant

- Discussion :
  - System of n linear equations in n unknowns Ax = b has a unique solution iff the determinant of its coefficient matrix, detA, is not equal to zero
  - This solution can be found by formulas called Cramer's rule

$$x_1 = \frac{\det A_1}{\det A}, \ldots, x_j = \frac{\det A_j}{\det A}, \ldots, x_n = \frac{\det A_n}{\det A}$$

where A_j is obtained by replacing the jth column of A by the column b

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
    - **Balanced Searched Trees**
      - **AVL Trees**
    - Heaps and Heapsort
    - Horner’s Rule and Binary Exponentiation
  - Problem Reduction

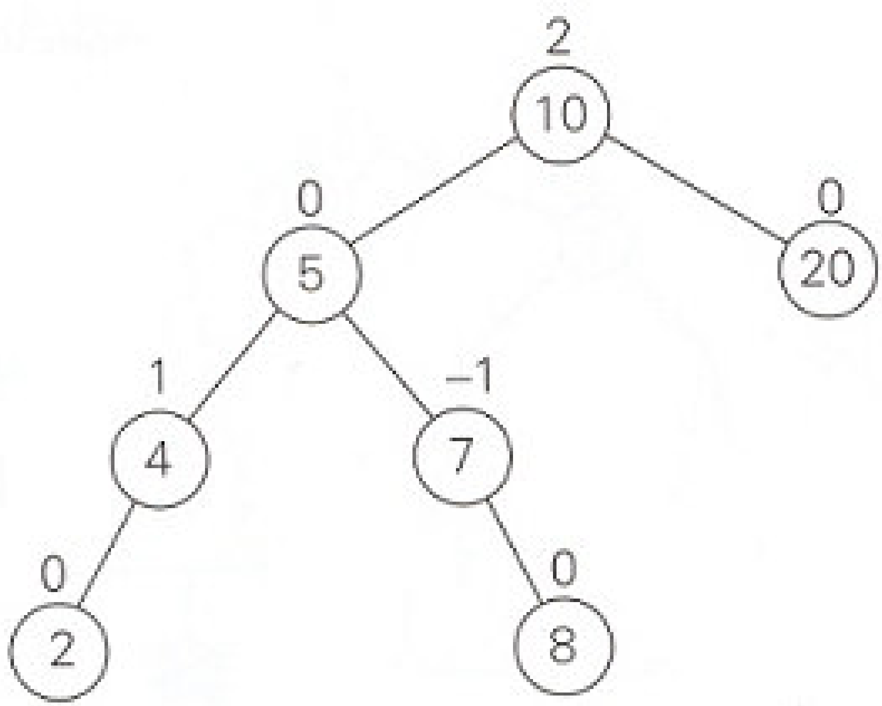
Balanced Search Trees

- BST is a basic data structure to implement dictionaries.
- BST can be considered as an example of representation change
  - Transformation over straightforward array implementation
- By this transformation, better run time efficiency is obtained
  - For insertion, deletion and search operations, logarithmic runtime on average case
  - What about worst case?

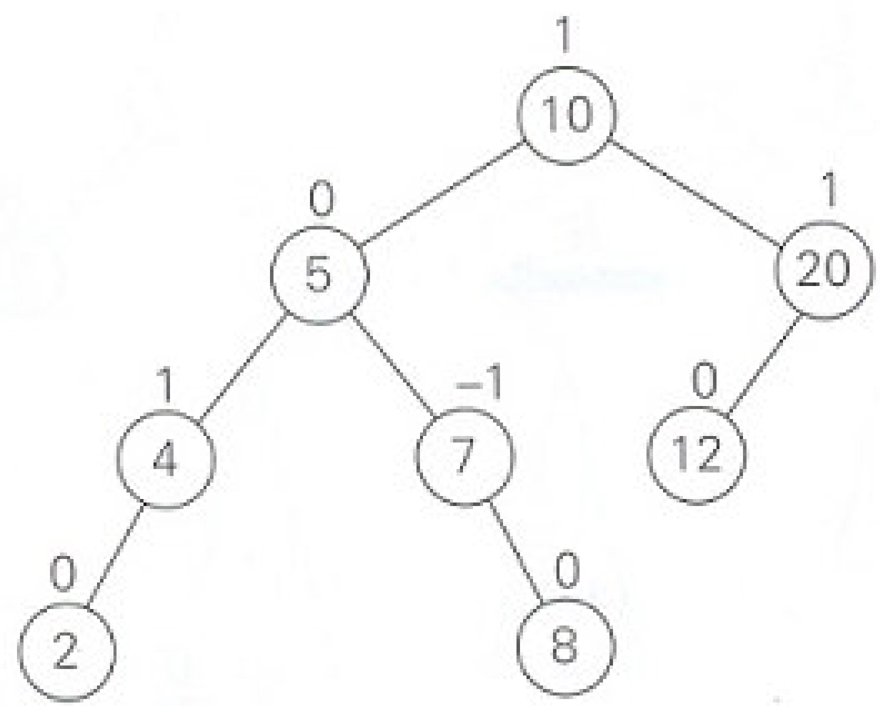
AVL Trees

- BST can be transformed to a balanced tree
  - For better worst case efficiency
- An *AVL tree* is a binary search tree in which the balance factor of every node
  - Balance factor is defined as the difference between the heights of the node's left and right subtrees
    - is either 0 or +1 or -1

AVL Trees



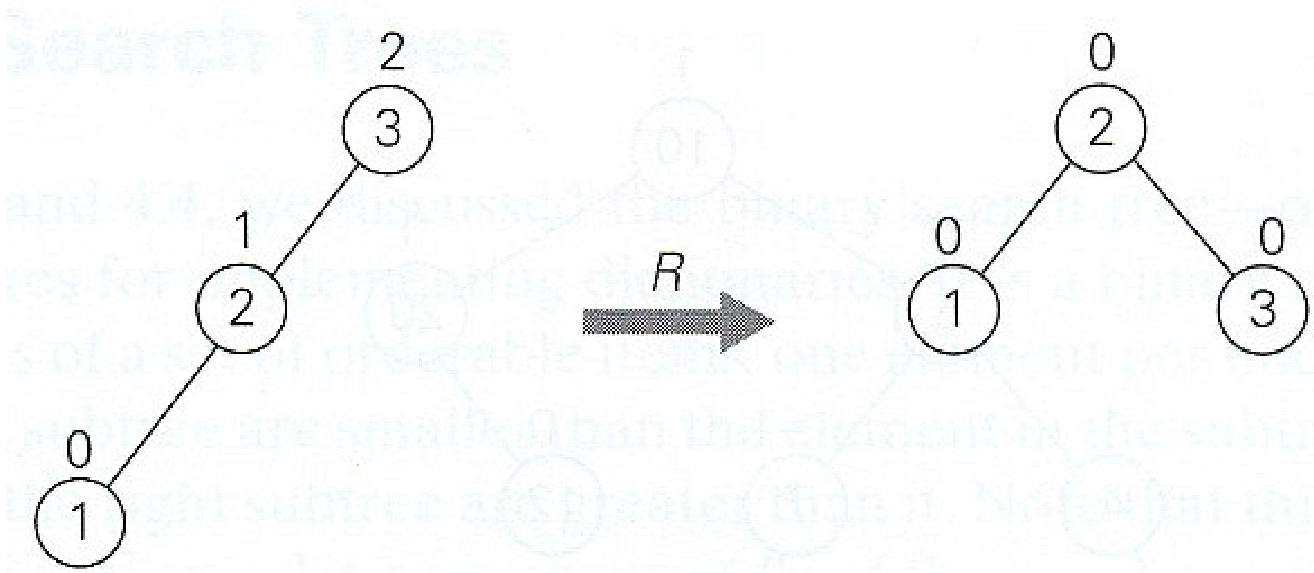
An AVL tree



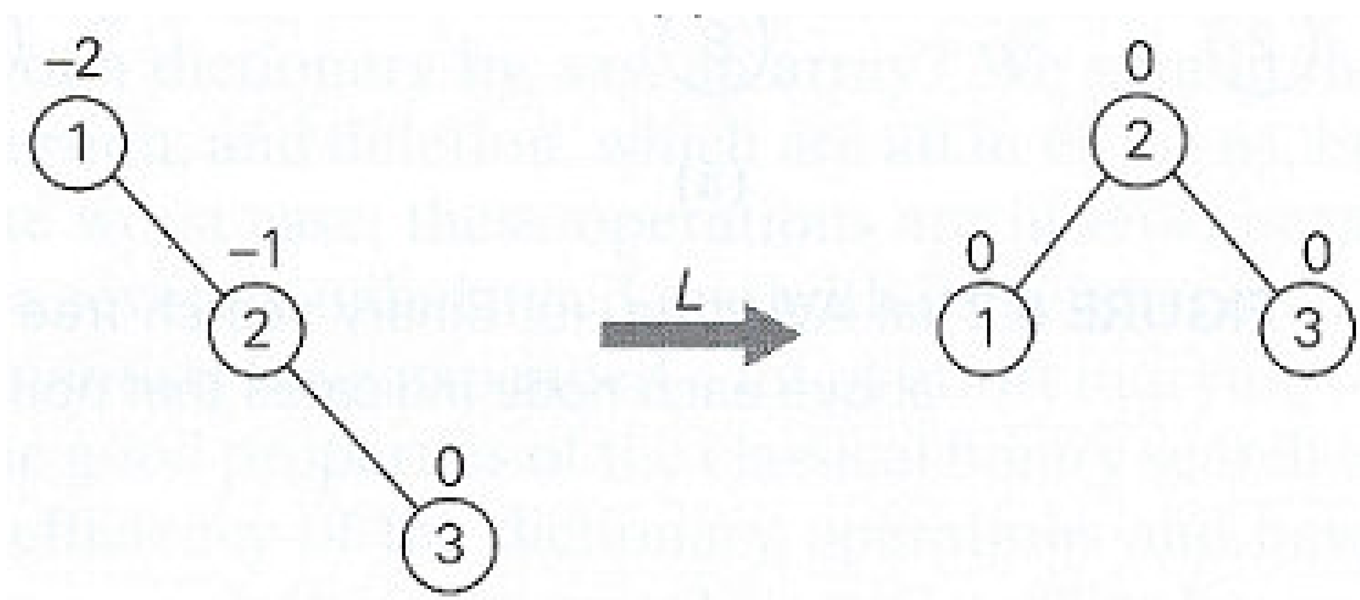
Not an AVL tree

AVL Trees

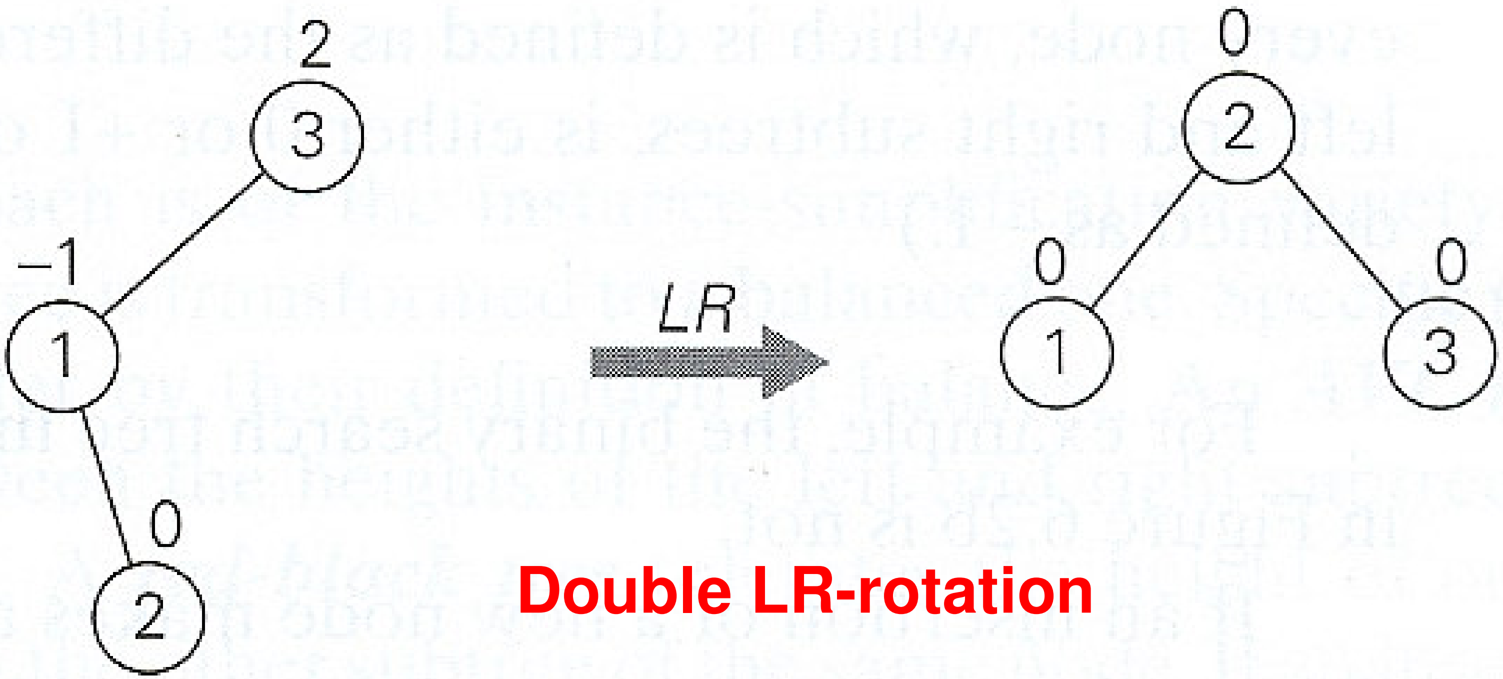
- If an insertion of a new node makes an AVL tree unbalanced we transform the tree by rotations
  - Single rotation
    - Left rotation
    - Right rotation
  - Double rotation
    - Left-right rotation
    - Right-left rotation



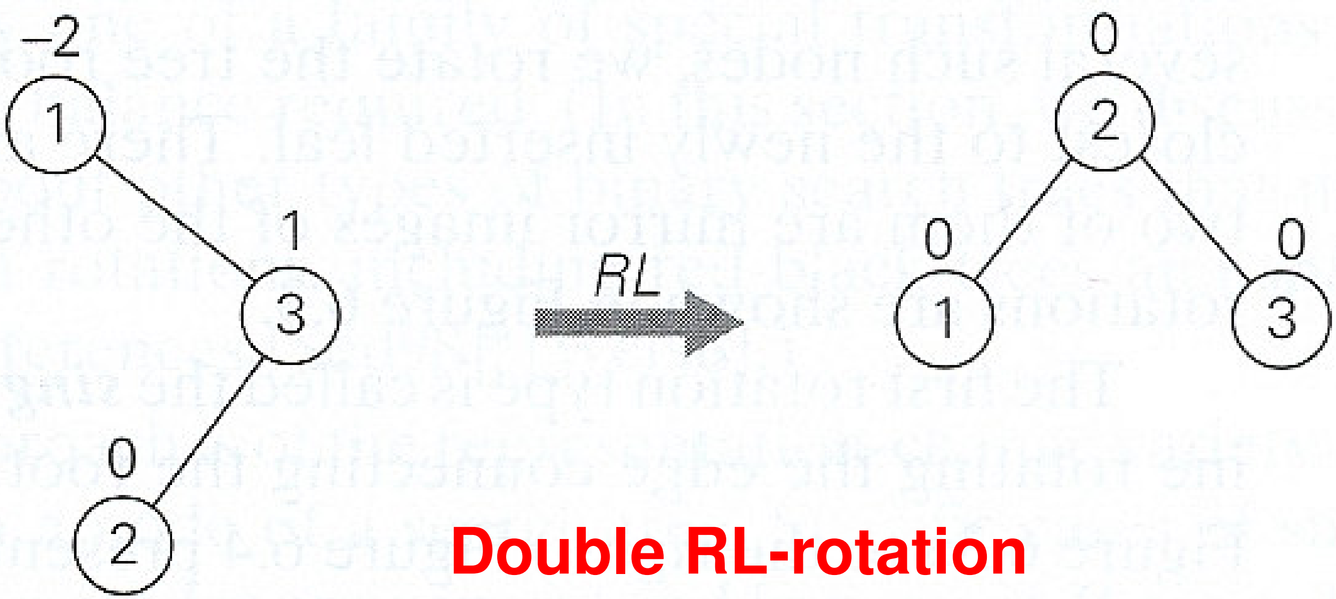
Single R-rotation



Single L-rotation

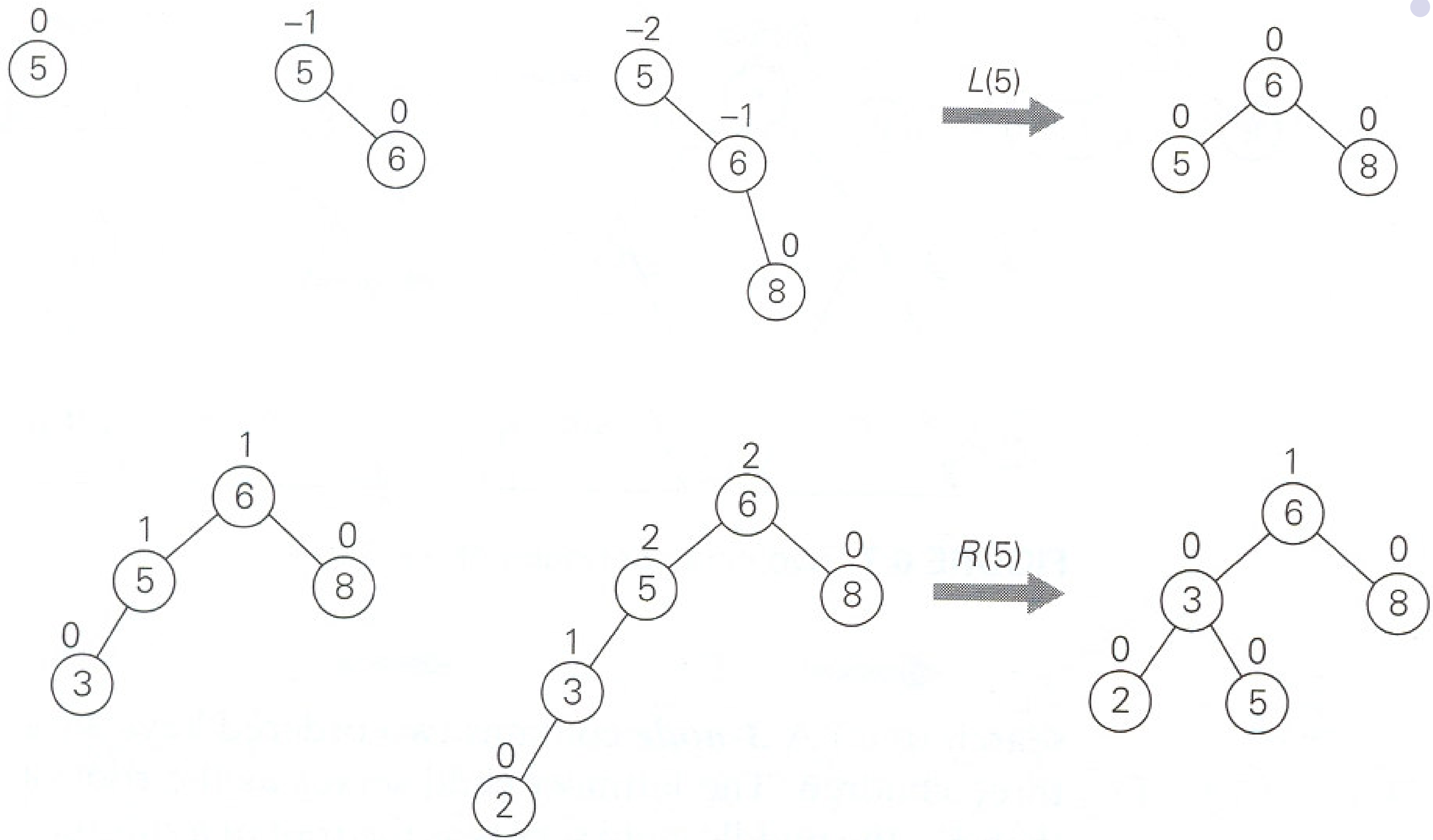


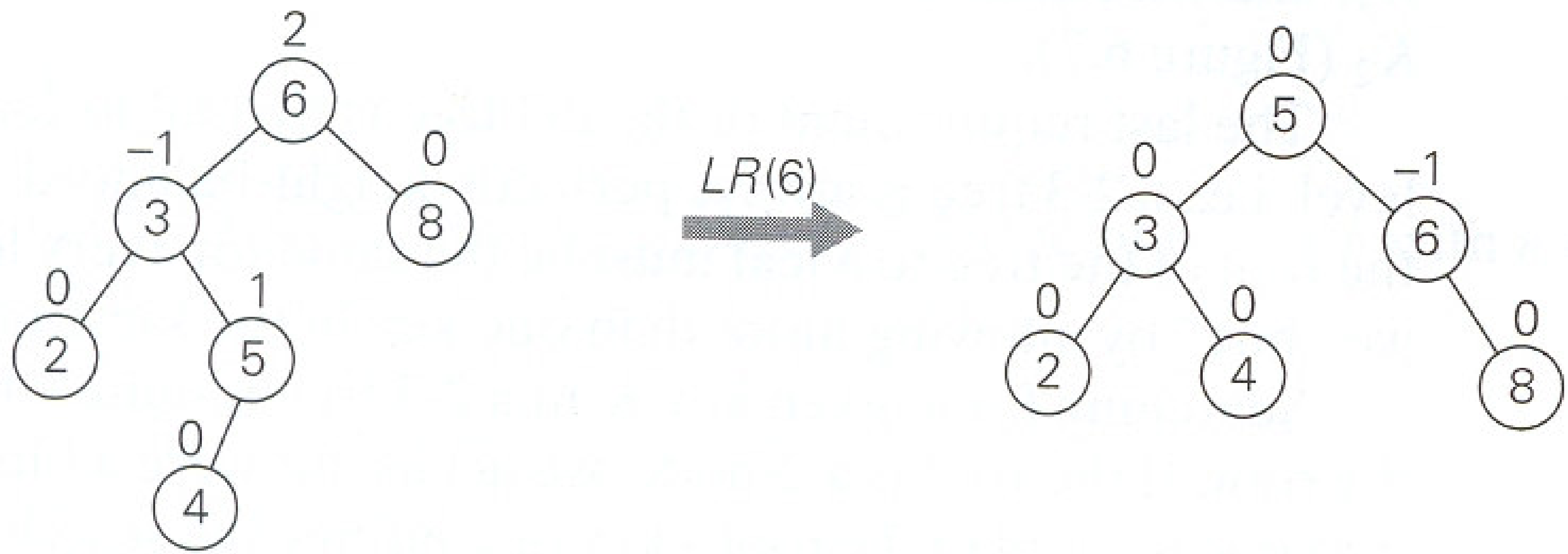
Double LR-rotation

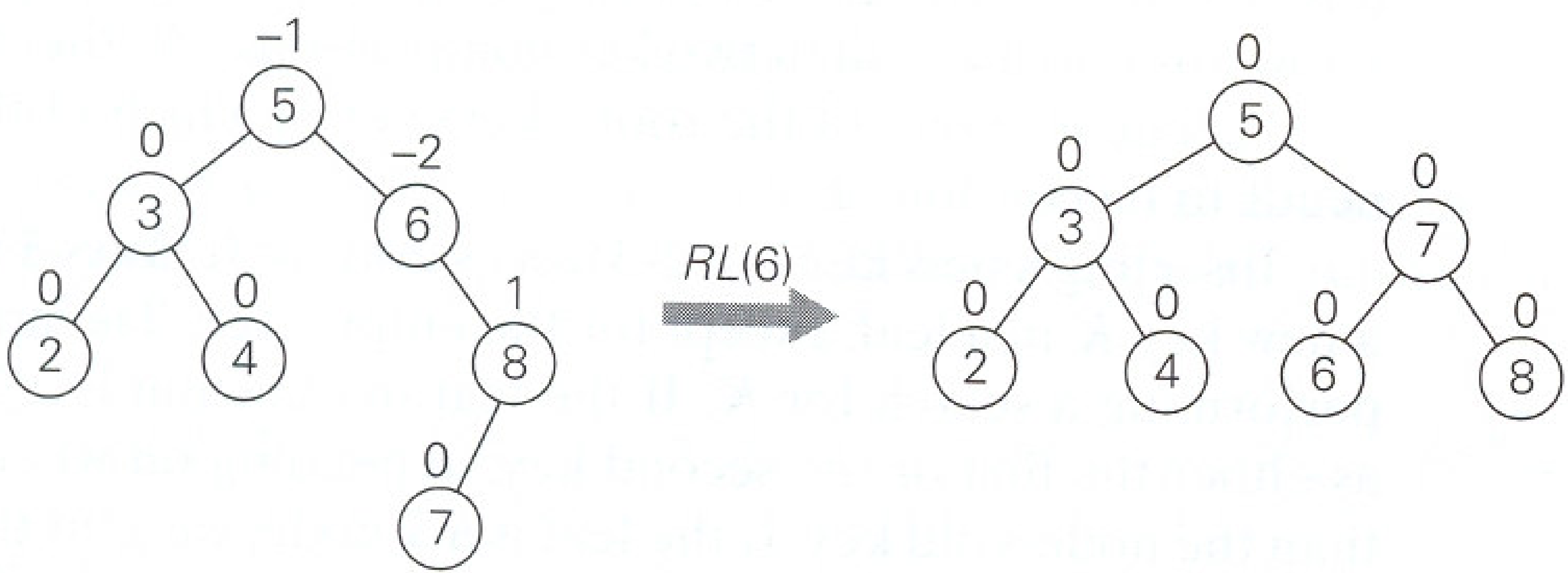


Double RL-rotation

Constructing of an AVL tree for the list 5, 6, 8, 3, 2, 4, 7




LR(6)


RL(6)

AVL Trees

- Analysis :
  - Searching and insertion and deletion operations are Θ`(logn)` in worst case
  - Searching in an AVL tree requires on average almost the same number of comparisons as searching in a sorted array by binary search

ROAD MAP

- **Transform And Conquer**
  - Instance simplification
  - Representation change
    - Balanced Searched TreesAVL Trees
    - **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*

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 ?

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

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

  - 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
    - Balanced Searched TreesAVL Trees
    - Heaps and Heapsort
    - **Horner's Rule and Binary Exponentiation**
  - Problem Reduction

Horner’s Rule

<u>Problem Definition:</u>

- 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

$p \leftarrow P[n]$
for $i \leftarrow n - 1$ **downto** 0 **do**
    $p \leftarrow x * p + P[i]$
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

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

      $2x^4 - x^3 + 3x^2 + x - 5$ by $x-3$

      $2x^3 + 5x^2 + 18x + 55$ and 160
  - This division algorithm is known as *synthetic division*
    - It is more convenient than long division

Exponentiation

- <u>Problem Definition :</u>
  - 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

- Example :
  - 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

- <u>Analysis :</u>

  Total number of multiplications M(n)

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

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

  So efficiency is $\boldsymbol{\Theta}$ ***(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

- Definition:
  - 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

$term \leftarrow a$ //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 $product$

Right-to-left binary exponentiation

- <u>Example :</u>
  - 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

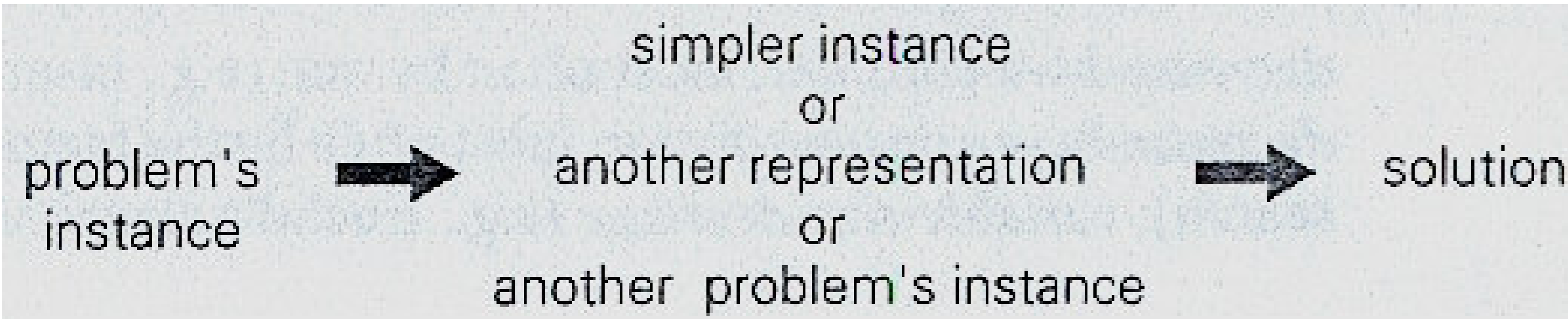
Binary Exponentiation

- 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**
    - **Reduction of Optimization Problems**
    - **Linear Programming**
    - **Reduction to Graph Problems**

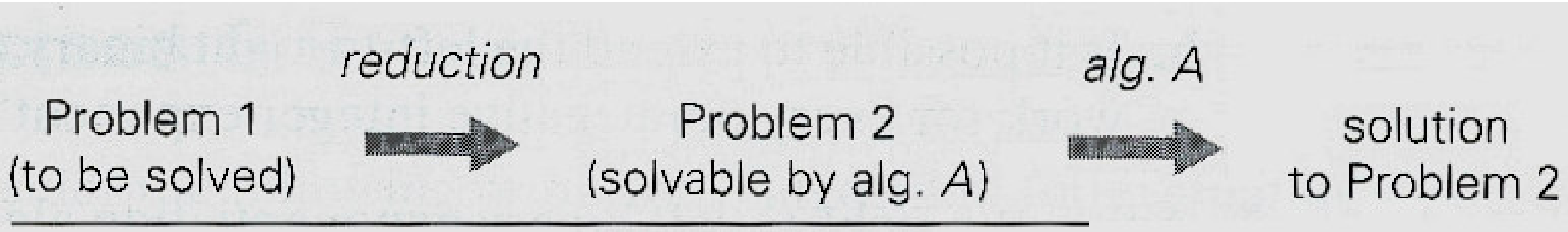
Transform And Conquer Strategy



- Instance simplification
  - Transformation to a simplier problem instance
- 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**

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

- <u>Definition :</u>
  - 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,

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

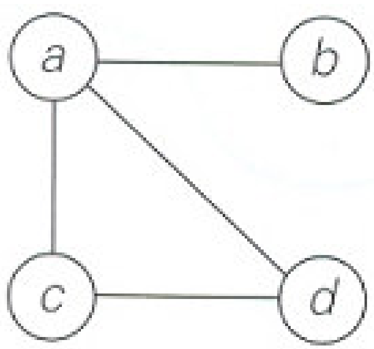
- This formula reduces lcm calculation to gcd calculation
- gcd(m,n) can be computed with Euclid's algorithm efficiently

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

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



a graph

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

its adjacency matrix A

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

its square A^2

Elements of A^2 indicate the number of paths of length 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 ?...

Reduction of Optimization Problems

Problem Definition:

- Find a maximum (minimum) of some function,
  - *maximization (minimization) problem*

- Suppose that you know an algorithm for maximizing a function, but you want to minimize it
- How can you take advantage of the latter ?

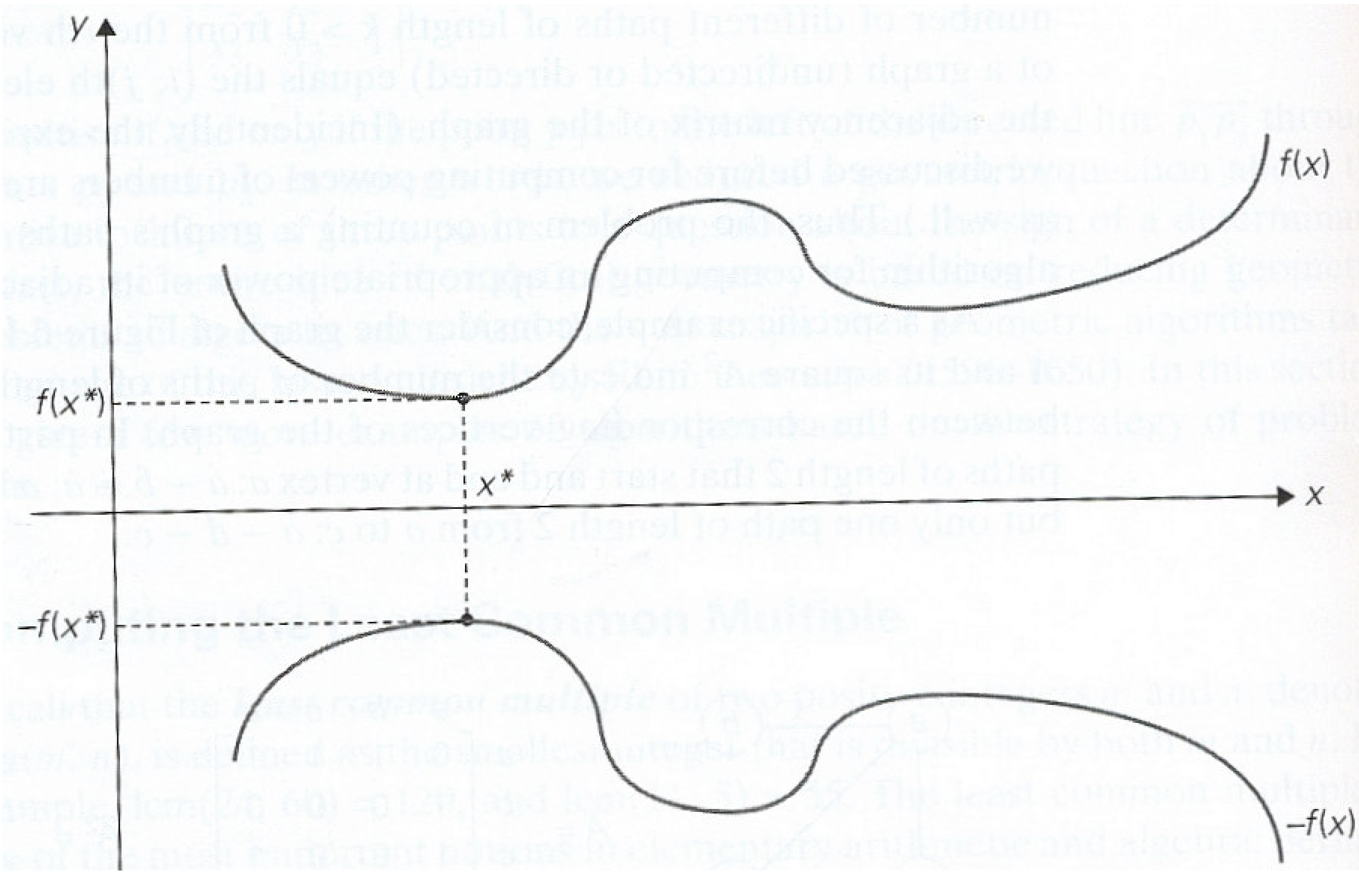
Reduction of Optimization Problems

- Approach :
  - To minimize a function
    - we can maximize its negative instead
    - get a correct minimal value of the function itself
    - change the sign of the answer

$$\min\ f(x) = -\max[-f(x)]$$

$$\max\ f(x) = -\min[-f(x)]$$ is also correct

Reduction of Optimization Problems



Relationship between minimizing and maximizing problems

$$\min \; f(x) = -\max\left[-f(x)\right]$$

Reduction of Optimization Problems

- Example :
  - To find extremum points of a function based on problem reduction
    - Find the function's derivative $f'(x)$
    - Solve the equation $f'(x) = 0$
    - Find the function's critical points
  - Optimization problem is reduced to the problem of solving an equation as the principal part of finding extremum points

Linear Programming

- Definition:
  - Linear programming is a problem of optimizing a linear function of several variables subject to constraints in the form of linear equations and linear inequalities
  - Many problems of optimal decision making can be reduced to an instance of the linear programing problem

Linear Programming

- Example 1:
  - Consider a university endowment that needs to invest $100 million
  - This sum must be split between three types of investments
    - stocks, bonds and cash
  - Endowment managers expect an annual return of 10%, 7% and 3% for their stock, bond and cash investments
  - Since stocks are most risky than bonds, endowment rules require the amount invested in stocks to be no more than one third of the moneys invested in bonds
  - In addition, at least 25% of the total amount invested in stocks and bonds must be invested in cash
  - How should the managers invest the money to maximize the return ?

Linear Programming

- First create a mathematical model of this problem
- Let, x, y, z be amounts invested in stocks, bonds and cash
- So we can pose the following optimization problem

$$
\begin{aligned}
\text{maximize} \quad & 0.10x + 0.07y + 0.03z \\
\text{subject to} \quad & x + y + z = 100 \\
& x \le \frac{1}{3} y \\
& z \ge 0.25(x + y) \\
& x \ge 0, \quad y \ge 0, \quad z \ge 0.
\end{aligned}
$$

General Linear Programming Problem

- Optimal decision making can be reduced to an instance of the general linear programming problem

$$
\begin{aligned}
\text{maximize (or minimize)} \quad & c_1x_1 + \cdots + c_nx_n \\
\text{subject to} \quad & a_{i1}x_1 + \cdots + a_{in}x_n \le (\text{or } \ge \text{ or } =)\ b_i \quad \text{for } i = 1, \ldots, m \\
& x_1 \ge 0, \ldots, x_n \ge 0.
\end{aligned}
$$

- Linear programming has proved to be flexible enough to model a wide variety of important applications
  - airline crew scheduling
  - transportation and communication network planning
  - oil exploration and refining
  - industrial production optimization

Linear Programming

- Classical algorithm for this problem *is simplex method*
  - Worst case efficiency of this algorithm is known to be exponential but performs very well on typical inputs
- There exist a few other algorithms
  - best known of them is discovered by Narendra Karmarkar
- Their polynomial worst case efficiency was proven
  - Karmarkar algorithm is competetive with simplex method in empirical tests

Linear Programming

- When variables of a linear programming problem are required to be integers, is said to be *integer linear programming* problems
  - known to be much more difficult
  - no polynomial time algorithm for solving them is known

Linear Programming

- Example 2:
  - Let see how knapsack problem can be reduced to a linear programming problem
    - Given a knapsack capacity W
    - n items of weights $w_1, w_2, \ldots, w_n$
  - We consider first the continuous version of the problem
    - Any fraction of any given item can be taken into the knapsack

Linear Programming

- Example 2:
  - Let x_j, $j = 1, \dots, n$ be a variable representing a fraction of item j taken to knapsack
  - x_j must satisfy the inequality $0 \le x_j \le 1$
  - Then total weight of selected items is $\sum_{j=1}^{n} w_j x_j$
  - Their total value is $\sum_{j=1}^{n} v_j x_j$
  - So the knapsack problem can be posed as the following linear programing problem

$$\text{maximize} \quad \sum_{j=1}^{n} v_j x_j$$

$$\text{subject to} \quad \sum_{j=1}^{n} w_j x_j \le W$$

$$0 \le x_j \le 1 \quad \text{for } j = 1, \dots, n$$

Linear Programming

- In the *discrete* (0 or 1) version of knapsack problem
  - either to take an item entirely or not to take it
  - we have the following integer linear programming problem

$$\text{maximize} \qquad \sum_{j=1}^{n} v_j x_j$$

$$\text{subject to} \qquad \sum_{j=1}^{n} w_j x_j \leq W$$

$$x_j \in \{0,1\} \qquad \text{for } j = 1, \ldots, n$$

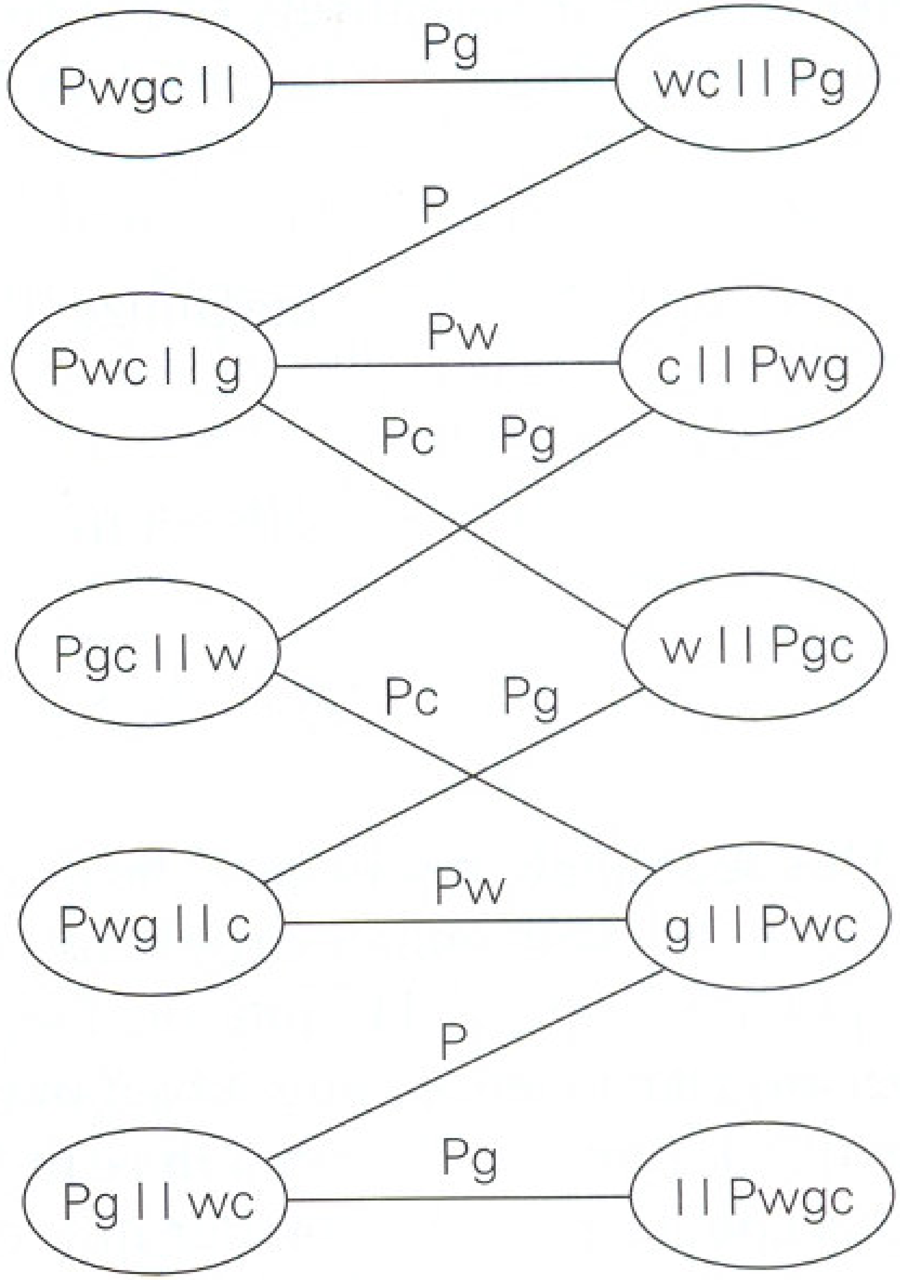
  - this minor modification makes a drastic difference on the complexity
  - discrete version seems to be easier, but actually more complicated

Reduction to Graph Problems

- Many problems can be solved by a reduction to one of the standart graph problems
  - Vertices typically represent possible states of the problem
    - One is initial state, another is final state
  - Edges indicate permitted transmissions among states
  - Such a graph is called state-space-graph
- The transformation reduces the problem to question about a path from the initial-state vertex to a goal-state vertex

Reduction to Graph Problems

- Example :
  - Consider river-crossing puzzle problem
  - A peasant finds himself on a river bank with a wolf, a goat and a head of cabbage
  - He needs to transport all three on the other side of the river in his boat.
  - However, the boat has room only peasant himself and one other item
  - In his absence the wolf would eat the goat, the goat will eat the cabbage
  - Find a way for the peasant to solve his problem or prove that it has no solution



- P,w,g,c stand for peasant, wolf, goat and cabbage
- | | represent the river
- The edges are labeled by indicating the boat's occupants for each crossing
- We are inretested in finding a path from the initial-state vertex, Pwgc || to the final state ||Pwgc

State-space graph for the *peasant*, *wolf*, *goat* and *cabbage* puzzle

Reduction to Graph Problems

- It is easy to see that there exist two distinct simple paths from initial state vertex to final state vertex
  - What are they ?
- If we find them by applying breadth-first search, we get a formal proof that these paths have the smallest number of edges possible
- This puzzle has two solutions, each of which requires seven river crossings

Reduction to Graph Problems

- Discussion:
  - Generating and investigating state-space graphs are not always a straightforward task such as in the given example
  - To get a better appreciation of them, consult books on AI (artificial intelligence)

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