Transform-and-Conquer Strategy

CMSC 142: Design and Analysis of Algorithms

Department of Mathematics and Computer Science
College of Science
University of the Philippines Baguio

Overview

Transform-and-Conquer Strategy

Instance Simplification

Representation Change

Problem Reduction

Transform-and-Conquer Strategy

The **transform-and-conquer** works as two-stage procedures:

(i) transformation stage - the problem's instance is modified to be more amenable to solution

(ii) conquering stage - transformed problem is solved

Three major variations of transform-and-conquer:

- *instance simplification* - transformation to a simpler or more convenient instance of the same problem
- *representation change* - transformation to a different representation of same instance
- *problem reduction* - transformation to an instance of a different problem for which an algorithm is already available

Overview

Instance Simplification: Presorting

Interest in sorting algorithms is due to the fact that many questions about a list are easier to answer if the list (array) is sorted.

Element Uniqueness Problem

- *Brute Force*: compare pairs of the array elements until either two equal elements are found or no more pairs were left; $C(n) \in \mathcal{O}(n^2)$
- Transform and conquer: sort then check for possibility of consecutive elements that are equal

What is the overall efficiency of this transform-and-conquer algorithm?

Instance Simplification: Presorting

Finding the Mode

- A **mode** is a value that occurs most often in a list.
- *Brute Force*: scan list, compute frequencies of all distinct values, then find the value with largest frequency
- Transform and conquer: *Sort* then find the longest run of adjacent equal values in sorted array

What is the overall efficiency of this transform-and-conquer algorithm?

```
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
```

Figure: Pseudocode of the transform-and-conquer algorithm in finding mode

Instance Simplification: Gaussian Elimination

Solving a System of Linear Equations

- standard method: substitution/ elimination of variables
- transform and conquer: transform system of n linear equations in n unknowns to an equivalent system with an upper triangular coefficient matrix using **Gaussian elimination**, then solve sytem by *backward-substitution*
- Gaussian elimination is a series of execution of elementary row operations (EROs): (i) exchanging two equations of the system, (ii) replacing an equation with its nonzero multiple, (iii) replacing an equation with a sum or difference of this equation and some multiple of another equation

Remark. Any system that is obtained through a series of EROs will have the same solution as the original one.

ALGORITHM *ForwardElimination*$(A[1..n, 1..n], b[1..n])$

//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**
      $A[j, k] \leftarrow A[j, k] - A[i, k] * A[j, i] \,/\, A[i, i]$

Figure: Pseudocode for the elimination stage of Gaussian elimination

Pseudocode for the Gaussian elimination with **partial pivoting**

ALGORITHM *BetterForwardElimination*$(A[1..n, 1..n], b[1..n])$

```
//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
```

Overview

Representation Change

Horner's Rule (William George Horner, *early 19th century*)

- efficient algorithm for evaluating a polynomial
- problem: find $p(x) = a_n x^n + a_{n-1}x^{n-1} + \ldots + a_1 x + a_0$ at $x = x_0$
- representation change: represent $p(x)$ as

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

Example

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

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

Figure: Pseudocode for the Horner's algorithm

Take multiplication as basic operation.
What is the overall efficiency of Horner's rule?

Representation Change: Binary Exponentiation a^n

- Let $n = b_I \dots b_i \dots b_0$ be the bit string representing a positive integer n in the binary number system. The value of n is equal to $p(x)$ at $x = 2$ where

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

- representation change: $a^n = a^{p(2)} = a^{b_I x^I + \dots + b_i x^i + \dots + b_0}$

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**
$p \leftarrow 2p + b_i$	$a^p \leftarrow a^{2p+b_i}$

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 $product$

Figure: Pseudocode for the left-to-right binary exponentiation method

What is the overall efficiency of the transform-and-conquer algorithm?

Right-to-left Binary Exponentiation

Let $n = b_I \dots b_i \dots b_0$ be the binary representation of $n \in \mathbb{N}$ and $p(2)$ be the binary polynomial associated to n. Then,

$$\begin{aligned} a^n &= a^{b_I x^I + \dots + b_i x^i + \dots + b_0} \\ &= a^{b_I 2^I} \cdot \dots a^{b_i 2^i} \cdot \dots \cdot a^{b_0} \end{aligned}$$

ALGORITHM *RightLeftBinaryExponentiation*$(a, b(n))$

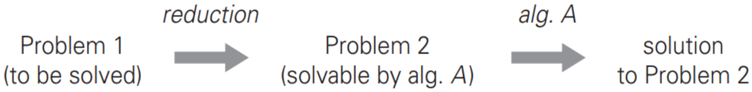
//Computes a^n by the right-to-left binary exponentiation algo
//Input: A number a and a list $b(n)$ of binary digits $b_I, \dots, b_0$
// in the binary expansion of a nonnegative integer n
//Output: The value of a^n
$term \leftarrow a$ //initializes a^{2^i}
if $b_0 = 1$ $product \leftarrow a$
else $product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

What is the overall efficiency of the transform-and-conquer algorithm?

Overview

Problem Reduction

Main idea: "If you need to solve a problem, reduce it to another problem that you know how to solve."



Examples

1. Computing the Least Common Multiple
2. Counting Paths in a Graph
3. Linear Programming
4. Reduction of Optimization Problems and Reduction to Graphs

End of Lecture