CS472 Module 7 Part A - Transform and Conquer - Overview

Athens State University

March 4, 2016

Outline

Contents

1 Transform and Conquer - Overview

Transform and Conquer - Say what?

- A meta-heuristic where we solve a problem by transforming the problem or its input into a another form that may be easier to solve.

Transform and Conquer - Three transformations

- Three types of transformations
  - *Instance simplification*: Convert the instance of the problem into a simpler or more convenient instance of the same problem
  - *Representation change*: Use a different representation of the same instance.
  - *Problem reduction*: rephrase the problem so that we can use an algorithm that is already available

2 Instance Simplification

Instance simplification

- Solve a problem's instance by transforming it into another simpler or easier instance of the same problem

3 Instance Simplification: Presorting

Instance simplification - presorting

- Many problems involving lists are easier when the list is sorted
- Topological sorting helps solving some problems for dags
- Presorting is used in many geometric algorithms

Instance simplification - presorting

- *Searching* - binary search is $O(\log_2(n))$, if data is sorted
- *Closet pair* - Given a set of n numbers, how do you find the pair of numbers with the smallest difference between them?
  - Sort the list, then closest pair must lie next to each other in the list
  - Linear search the list, get an $O(n * \log_2(n))$ algorithm!
- *Element uniqueness* - Are there any duplicates in a given set of n items?
- *Frequency distribution* - Given a set of n items, which element occurs the largest number of times in the set?
- *Selection* - What is the k-th item in an array?

Instance simplification: Example: What is the intersection of two sets?
Problem statement
Give an efficient algorithm to determine whether two sets of size m and n are disjoint. Analyze the worst-cast time complexity in terms of m and n, considering the case where m is substantially smaller than n.

Instance simplification: Example: What is the intersection of two sets?
Solutions

1. Sort the big set
   - Set can be sorted in $O(n * lg(n))$ time.
   - Binary search with each of the m elements in the second set
   - Total time required will be $O((n + m) * lg(n))$
2. Sort the small sets
   - Set can be sorted in $O(m * lg(m))$ time.
   - Binary search with each of the n elements in the big set
   - Total time will be $O((n + m) * lg(m))$
3. Sort both sets
   - Sort both sets with $O(n * lg(n))$ and $O(m * lg(m))$ time
   - Compare the smallest element in each set, discard that element if they are identical
   - Repeat recursively. Note this will occur in linear time
   - Total cost is $O(n * lg(n) + m * lg(m) + n + m)$

Instance simplification: Example: What is the intersection of two sets?

Analysis

- Note that $lg(m) < lg(n)$ when $m < n$
- And note that $n + m < 2n$ when $m < n$
- Thus, we can claim that $(n + m) * lg(m)$ is asymptotically less than $n * lg(n)$
- So, sorting the small set is the best option

Instance simplification: How fast can we sort?

- Efficiency of algorithms involving sorting depends on efficiency of sorting.
- We can prove $O(n * log(n))$ comparisons are required in the worst case to sort a list of size n by *any* comparison-based algorithm

4 Polynomial Evaluation

Polynomial Evaluation: Brute Force: Problem

Find the value of the polynomial

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

at a point $x = x_0$

Polynomial Evaluation: Brute Force: Algorithm

Algorithm 1: Brute Force

```
Input: An array containing the polynomial coefficients
Output: The value of p(x) at point
p ← 0.0 ;
for i ← [n..0] do
    power ← 1;
    for j ← [1..i] do
        power ← power * x;
    p ← p + a[i] * power;
return p;
```

Polynomial Evaluation: Brute Force: Analysis

- Notice that the innermost loop of the algorithm is computing x^n
  - And using the brute force method
  - So, can count the number of multiplications

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

Polynomial Evaluation: Brute Force: An improvement?

- The original algorithm computes powers of x without taking into account any relationship between them
- So, suppose we compute from highest term to lowest term
  - Better, but means we have to use division (why? and why bad?)
- Suppose we compute x^i by using x^{i-1}?

Polynomial Evaluation: Brute Force: an improvement?

Algorithm 2: Brute Force, Version 2

```
Input: An array containing the polynomial coefficients
Output: The value of p(x) at point
p ← a[0] ;
power ← 1;
for i ← [1..n] do
    power ← power * x;
    p ← p + a[i] * power;
return p;
```

Polynomial Evaluation: Brute Force: improvement analysis

Again, we count the number of multplications:

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

Victory! We've gone from an $O(n^2)$ algorithm to $O(n)$ algorithm

Can we do better?

No, because for any polynomial of degree n at a point x, we have to process $n+1$ coefficients.

5 Horner's Rule: Applying representation change

Horner's Rule: A different view

Our original problem was to find the value of the polynomial

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

at a point $x = x_0$

Suppose we factor x as common factor in polynomials of diminishing degree:

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

Horner's Rule: A different view

Consider

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

Horner's Rule: A different view

What's interesting about Horner's rule is that you don't need to do all the ugly algebra, you only need to manipulate the coefficients of the polynomial.

Suppose we want the value of the polynomial in the previous slide at $x = 3$:

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

Horner's Rule: Algorithm

Algorithm 3: Evaluate a polynomial at a given point using Horner's Rule

```
Input: An array P[0..n] of coefficients of a polynomial of degree n sorted 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)..0] do
    p ← x*p+P[i];
return p;
```

Horner's Rule: Analysis

- Incredibly simple algorithm
  - Basic operation is the number of multiplications
  - Which is the same as the number of additions
- Which is **linear**!
- Horner's rule has some useful byproducts
  - For example, the intermediate numbers generated by algorithm are the coefficients of the polynomial that results from dividing $p(x)$ by $x - x_0$
  - This fact can be used to produce an algorithm for *synthetic division* that is much simpler to work with than algorithms for long division

Binary Exponentiation

- Horner's rule doesn't gain us anything if we just want to compute a^n
  - In fact, it degenerates into the brute force algorithm
- This is a problem as computation of a^n (most of type using arithmetic modulus n) is a fundamental problem in cryptography
- So, finding an efficient algorithm for this problem is very important

Binary Exponentiation: Left to right binary exponentiation

- Initialize a product accumulator to 1
- Scan n's binary expansion from left to right
  - If the current binary digit is 0, square the accumulator
  - If the binary digit is 1, square the accumulator and multply it by a
- Note the efficiency:
$$b \leq M(n) \leq 2b$$
  where $b = \lfloor log_2 n \rfloor + 1$

6 Key points

Key points

- What is the transform-and-conquer meta-heuristic?
- What are the three types of transform-and-conquer algorithms?
- Why is presorting helpful and how much does it cost?
- Representation change
- Horner's Rule vs. brute force
- Binary exponentiation