CSCI-GA.1170-001/002 Fundamental Algorithms	September 10, 2014

Solutions to Problem 1 of Homework 1 (16 (+4) points)

Name: Keeyon Ebrahimi *Due: Tuesday, September 10*

A degree-n polynomial $P(x)$ is a function

$$P(x) = a_0 + a_1 x + \ldots + a_{n-1}x^{n-1} + a_n x^n = \sum_{i=0}^{n} a_i x^i$$

(a) (2 points) Express the value $P(x)$ as

$$P(x) = a_0 + a_1 x + \ldots + a_{n-2}x^{n-2} + b_{n-1}x^{n-1} = \sum_{i=0}^{n-1} b_i x^i$$

where $b_0 = a_0, \ldots, b_{n-2} = a_{n-2}$. What is b_{n-1} as a function of the a_i's and x?

Solution:

For all numbers, b_n equals a_n, so $P(x)$ in regards to b is

$$P(x) = b_0 + b_1 x + \ldots + b_{n-2}x^{n-2} + b_{n-1}x^{n-1} = \sum_{i=0}^{n-1} b_i x^i$$

What is b_{n-1} as a function of the a_i's and x?

We now want to isolate b_{n-1} in order to do this, *in terms of a_i's and x*, we do this

$$\sum_{i=0}^{n-1} a_i x^i - \sum_{i=0}^{n-2} a_i x^i$$

$\sum_{i=0}^{n-1} a_i x^i$ and $\sum_{i=0}^{n-2} a_i x^i$ have all similar terms, except $\sum_{i=0}^{n-1} a_i x^i$ has b_{n-1} where $\sum_{i=0}^{n-2} a_i x^i$ does not.

By subtracting the two, we can isolate b_{n-1}

□

(b) (5 points) Using part (a) above write a recursive procedure $\mathsf{Eval}(A, n, x)$ to evaluate the polynomial $P(x)$ whose coefficients are given in the array $A[0 \ldots n]$ (i.e., $A[0] = a_0$, etc.). Make sure you do not forget the base case $n = 0$.

Solution:

We want to express the equation as $a_0 + x(a_1 + x)(a_2 + x) \dots (a_n + x)$ in order to get away from the n^2 running time. Now we don't have to have an exponent calculation with every pass, and instead we just have an individual multiplication operation with each pass.

$\text{Eval}(A, n, x)$
  If $n == 0$
    Return $A[0] + x$
  Return $(A[n] + x)*$ $\text{Eval}(A, n-1, x)$

□

(c) (3 points) Let $T(n)$ be the running time of your implementation of Eval. Write a recurrence equation for $T(n)$ and solve it in the $\Theta(\cdot)$ notation.

Solution:
We need to solve for $T(n) = aT(\frac{n}{b}) + D(n) + C(n)$ where a is the number of subproblems, $\frac{n}{b}$ is the size of each subproblem, $D(n)$ is the running time for the dividing step and $C(n)$ is the running time for the combining step.

If $n \leq 1$, $\Theta(1)$

Else
$T(n) = 1T(n) + \Theta(1) + \Theta(1)$

- $a = 1$ because we are only having one subproblem, evident by the only one call back into $\text{Eval}(A, n, x)$
- $\frac{n}{b} = n$ because each subset only decreases by 1 no matter how large n is, which makes the subset size stay at n
- $D(n) = 1$ because to divide, we just take our subset and decrease its size by one, which can be done in constant time.
- $C(n) = 1$ because we are not doing any iteration with the combining, we are just returning the conquered results, which is done with a simple multiplication, and runs in constant time

This makes the running time of this algorithm $\Theta(n)$ □

(d) (6 points) Assuming n is a power of 2, try to express $P(x)$ as $P(x) = P_0(x) + x^{n/2} P_1(x)$, where $P_0(x)$ and $P_1(x)$ are both polynomials of degree $n/2$. Assuming the computation of $x^{n/2}$ takes $O(n)$ times, describe (in words or pseudocode) a recursive procedure Eval_2 to compute $P(x)$ using two recursive calls to Eval_2. Write a recurrence relation for the running time of Eval_2 and solve it. How does your solution compare to your solution in part (c)?

Solution:

Assuming n is a power of 2, try to express $P(x)$ as $P(x) = P_0(x) + x^{n/2} P_1(x)$, where $P_0(x)$ and $P_1(x)$ are both polynomials of degree $n/2$

$$P_0(x) = \sum_{i=0}^{\frac{n}{2}} a_i x^i$$

$$P_1(x) = \sum_{i=\frac{n}{2}+1}^{n} a_i x^{i-\frac{n}{2}}$$

With these summations for $P_0(x)$ and $P_1(x)$, we can have $P_0(x) + x^{n/2} P_1(x) = \sum_{i=0}^{n} a_i x^i$

Write a recursive procedure Eval_2 *to compute $P(x)$ using two recursive calls to* Eval_2

$\text{E}{\scriptsize\text{VAL}}2(A, x, n)$
  If $n == 0$
    Return $A[0]$
  If $n == 1$
    Return $\text{E}{\scriptsize\text{VAL}}2(A[0], x, n-1) + A[1]x$
  Return $\text{E}{\scriptsize\text{VAL}}2(A[0 : \frac{n}{2}], x, \frac{n}{2}) + (x^{\frac{n}{2}} * \text{E}{\scriptsize\text{VAL}}2(A[\frac{n}{2}+1 : n], x, n - (\frac{n}{2}+1)))$

We have base cases for when n is 0 or 1. When n is 1, we just basically add $A[1]x$ to the $n == 0$ base case

then we run Eval2 twice with half of the array given to us, and we multiply $x^{\frac{n}{2}}$ to the second part of the array.

Write a recurrence relation for the running time of Eval_2 *and solve it. How does your solution compare to your solution in part (c)?*

If $n \leq 1$, $\Theta(1)$

Else
$T(n) = 2T(\frac{n}{2}) + \Theta(1) + \Theta(n)$

This gives this algorithm a running time of $\Theta(n\ log\ n)$

$\square$

(e) (**Extra Credit.**) Explain how to fix the slow "conquer" step of part (d) so that the resulting solution is as efficient as "expected".

Solution: So right now, each conquer step takes n steps to complete because we have the $x^{n/2}$ being multiplied by $P_1(x)$, and the $x^{n/2}$ takes $O(n)$ times. Although we are reducing size exponentially, we still have $O(n)$ operations per conquer. Instead, we can calculate develop and add onto the $x^{n/2}$ value with each iteration, instead of recalculating at each pass. Horners Algorithm, which is the equation used for part c, is a method of decreasing the Conquer stop $\square$

CSCI-GA.1170-001/002 Fundamental Algorithms	September 10, 2014

Solutions to Problem 2 of Homework 1 (10 Points)

Name: Keeyon Ebrahimi *Due: Tuesday, September 10*

For each of the following pairs of functions $f(n)$ and $g(n)$, state whether f is $O(g)$; whether f is $o(g)$; whether f is $\Theta(g)$; whether f is $\Omega(g)$; and whether f is $\omega(g)$. (More than one of these can be true for a single pair!)

f is $O(g)$ f is $o(g)$ f is $\Theta(g)$ f is $\Omega(g)$ f is $\omega(g)$

(a) $f(n) = 32n^{21} + 2$; $g(n) = \frac{n^{22}+3n+4}{111} - 52n$.

Solution: f is $\Omega(g)$ and f is $\omega(g)$ □

(b) $f(n) = \log(n^{21} + 3n)$; $g(n) = \log(n^2 - 1)$.

Solution: f is $O(g)$, f is $\Theta(g)$, and f is $\Omega(g)$ □

(c) $f(n) = \log(2^n + n^2)$; $g(n) = \log(n^{22})$.

Solution: f is $O(g)$ and f is $o(g)$ □

(d) $f(n) = n^3 \cdot 2^n$; $g(n) = n^2 \cdot 3^n$.

Solution: f is $O(g)$, f is $\Theta(g)$, and f is $\Omega(g)$ □

(e) $f(n) = (n^n)^3$; $g(n) = n^{(n^3)}$.

Solution:

f is $\Omega(g)$ and f is $\omega(g)$

□

CSCI-GA.1170-001/002 Fundamental Algorithms	September 10, 2014

Solutions to Problem 3 of Homework 1 (10 points)

Name: Keeyon Ebrahimi — *Due: Tuesday, September 10*

The following two functions both take as arguments two n-element arrays A and B:

Magic-1(A, B, n)
  For $i = 1$ **to** n
    For $j = 1$ **to** n
      If $A[i] \geq B[j]$ **Return** FALSE
  Return TRUE

Magic-2(A, B, n)
  $temp := A[1]$
  For $i = 2$ **to** n
    If $A[i] > temp$ **Then** $temp := A[i]$
  For $j = 1$ **to** n
    If $temp \geq B[j]$ **Return** FALSE
  Return TRUE

(a) (2 points) It turns out both of these procedures return TRUE if and only if the same 'special condition' regarding the arrays A and B holds. Describe this 'special condition' in English.

Solution: The special condition is when elements 1 and on in Array B is larger than elements 1 and on in Array A. Every of these elements in Array B has to be larger than every single one of the A array elements above index 1. □

(b) (5 points) Analyze the worst-case running time for both algorithms in the Θ-notation. Which algorithm would you chose? Is it the one with the shortest code (number of lines)?

Solution: Magic-1 has a running time of $Theta(0(n^2))$
Magic-2 has a running time of $Theta(0(n))$
I would choose Magic-2 as it has a shorter running time. The one with the shortest code is not the one that runs shortest. □

(c) (3 points) Does the situation change if we consider the best-case running time for both algorithms?

Solution: The best-case running time for both algorithms does change things. The best case running time for Magic-1 is $\Theta(1)$, which happens when $A[1] \geq B[1]$

The best case running time for MAGIC-1 is $\Theta(1)$, which happens when $A[1] \geq B[1]$

The best case running time for MAGIC-2 is $\Theta(n)$ because no matter what, we will always iterate through all of Array A.

$\square$