CSCI-GA.1170-003/004 Fundamental Algorithms February 21, 2020

Solutions to Problem 1 of Homework 3 (24 points)

Name: Rahul Zalkikar (rz1567) *Due: 8pm on Thursday, February 20*

Collaborators: jni215, ua388, nmd353

The sequence $\{F_n \mid n \geq 0\}$ are defined as follows: $F_0 = 1$, $F_1 = 1$, $F_2 = 2$ and, for $i > 2$, define $F_i := 3F_{i-1} + 3F_{i-2} + 4F_{i-3}$.

(a) (2 Points) We can think of the above recurrence relation as a matrix equation. More specifically, the relation can be represented as an equation of the following form:

$$\mathbf{A} \cdot \begin{pmatrix} F_i \\ F_{i-1} \\ F_{i-2} \end{pmatrix} = \begin{pmatrix} F_{i+1} \\ F_i \\ F_{i-1} \end{pmatrix}$$

What is the satisfying value of $\mathbf{A}$? (**Hint**: Consider the simpler case of a Fibonacci Sequence, i.e, $F_i := F_{i-1} + F_{i-2}$ for $i > 1$ and $F_0 = 0, F_1 = 1$. How would you set up the matrix equation?)

Solution:

We are given that $F_i := 3F_{i-1} + 3F_{i-2} + 4F_{i-3}$, so we know $F_{i+1} = 3F_i + 3F_{i-1} + 4F_{i-2}$. Thus we can see that $A = \begin{bmatrix} 3 & 3 & 4 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}$ □

(b) (3 Points) Use the equation from part (a) and the divide and conquer strategy to build an efficient algorithm for computing F_n. Analyze its runtime in terms of the number of 3×3 matrix multiplications performed (each of which takes a constant number of integer additions/multiplications).

Solution:

I use $\begin{bmatrix} 3 & 3 & 4 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}^{n-1} \cdot \begin{pmatrix} 4 \\ 3 \\ 3 \end{pmatrix} = \begin{pmatrix} F_{n+1} \\ F_n \\ F_{n-1} \end{pmatrix}$

So,

Recursive-Exponentiation(a, n):
  if $n == 0$:
    return a
  else if n is even
    return Recursive-Exponentiation$(a, \frac{n}{2})^2$
  else if n is odd
    return $a \cdot$Recursive-Exponentiation$(a, \frac{n-1}{2})^2$

Since $A = \begin{bmatrix} 3 & 3 & 4 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}$ and $r = \begin{pmatrix} 4 \\ 3 \\ 3 \end{pmatrix}$ we can rewrite as

MYFUNC(a, r, n):
 if $n == 0$ or $n == 1$:
  return 1
 else
  r*Recursive-Exponentiation$(a, n-1)$
  return $r[1]$

It is clear for the exponentiation algorithm that $T(n) = T(n/2) + 1$ which, given constant number of integer operations per multiplications performed, means that the time for our function is $O(log(n-1)) = O(logn)$.

$\square$

(c) (6 Points) Prove by induction that, for some constant $a > 1$, $F_n = \Theta(a^n)$. Namely, prove by induction that for some constant c_1 you have $F_n \leq c_1 \cdot a^n$, and for some constant c_2 you have $F_n \geq c_2 \cdot a^n$. What is the right constant a and the best c_1 and c_2 you can find. Further, use your result to compute the size of F_n in binary. (**Hint**: Pay attention to the base case $n = 0, 1, 2$. Also, you need to do *two* very similar inductive proofs.)

Solution:

First I find values of a, c_1, c_2 using what's given.

$F_n = 3F_{n-1} + 3F_{n-2} + 4F_{n-3}$

$a^n \leq 3a^{n-1} + 3a^{n-2} + 4a^{n-3}$

$a^n \leq a^{n-3}(3a^2 + 3a + 4)$

$a^3 = 3a^2 + 3a + 4$

$a^3 - 3a^2 - 3a + 4 = a^3 - 4a^2 + a^2 - 4a + a - 4 = 0$

$a^2(a-4) + a(a-4) + 1(a-4) = (a^2 + a + 1)(a-4) = 0$, so $a = 4$.

Now let $a = 4, c_1 = 1$, and $c_2 = 1/8$. First, we want to show that $F_n \leq c_1 a^n$.

The base case is:

$F_0 \leq c_1 a^0$, $1 \leq c_1(1)$, subbing $c_1 = 1$ we have $1 \leq 1$.

$F_1 \leq c_1 a^1$, $1 \leq c_1(4)$, subbing $c_1 = 1$ we have $1 \leq 4$.

$F_2 \leq c_1 a^2$, $2 \leq c_1(16)$, subbing $c_1 = 1$ we have $2 \leq 16$.

To satisfy all the base case we can see that the tightest value for c_1 is 1.

$F_0 \geq c_2 a^0$, $1 \geq c_2(1)$, subbing $c_2 = 1/8$ we have $1 \geq 1/8$.

$F_1 \geq c_2 a^1$, $1 \geq c_2(4)$, subbing $c_2 = 1/8$ we have $1 \geq 1/2$.

$F_2 \geq c_2 a^2$, $2 \geq c_2(16)$, subbing $c_2 = 1/8$ we have $2 \geq 2$.

and just because, $F_3 \geq c_2 a^3$, $13 \geq c_2(64)$, subbing $c_2 = 1/8$ we have $13 \geq 8$. (we can also see that $1/8 > 13/64$).

To satisfy all the base case we can see that the tightest value for c_2 is $1/8$.

We know that $F_{i+1} = 3F_i + 3F_{i-1} + 4F_{i-2}$

So the inductive step (Assuming the inequality holds for $n \geq 3$):

$F_{i+1} \leq 3c_1 a^n + 3c_1 a^{n-1} + 4c_1 a^{n-2}$

$F_{i+1} = 3(4^n) + 3(4^{n-1}) + 4(4^{n-2}) = 3(4^n) + 3(4^{n-1}) + (4^{n-1})$

$F_{i+1} = 3(4^n) + 4(4^{n-1}) = 3(4^n) + (4^n) = 4(4^n) = 4^{n+1}$

$F_{i+1} = c_1 a^{n+1}$

and

$F_{i+1} \geq 3c_2 a^n + 3c_2 a^{n-1} + 4c_2 a^{n-2}$

$F_{i+1} = (1/8)(3)(4^n) + (1/8)(3)(4^{n-1}) + (1/8)(4)(4^{n-2}) = (1/8)[3(4^n) + 3(4^{n-1}) + (4^{n-1})]$

$F_{i+1} = (1/8)[3(4^n) + 4(4^{n-1})] = (1/8)[3(4^n) + (4^n)] = (1/8)[4(4^n)] = (1/8)[4^{n+1}]$

$F_{i+1} = c_2 a^{n+1}$

So the size of F_n in binary is $2n - 3 + 1 = 2n - 2$.

□

(d) (6 points) In your algorithm of part (b) you only counted the number of 3×3 matrix multiplications. However, the integers used to compute (F_i, F_{i-1}, F_{i-2}) grow in size as shown in part (c). Thus, the 3×3 matrix multiplication used at that level of recursion will not take $O(1)$ time. Show that using Karatsuba's multiplication the runtime of last matrix multiplication to compute (F_i, F_{i-1}, F_{i-2}) is $O(i^{\log_2 3})$. Using this more realistic estimate, analyze the actual running time $T(n)$ of your algorithm in part (b).

Solution:

Now that we know the matrix multiplication is not constant time (instead Karatsuba's multiplication run time), we have $T(n) = T(n/2) + O(n^{log_2 3})$, so our exponentiation algo time becomes $O(n^{log_2 3})$ (case 1 of the Master Theorem), and our function time becomes $O((n-1)^{log_2 3}) = O(n^{log_2 3})$. □

(e) (3 points) Finally, let us look at the naive sequential algorithm which computes $F_3, F_4, \ldots, F_n$ one-by-one. Assuming each F_i takes $\Theta(i)$ bits to represent, and that integer addition/subtraction takes time $O(i)$ (multiplication by two can be implemented by addition), analyze the actual running time of the naive algorithm. How does it compare to your answer in part (d)?

Solution:

From expanding a little we have $T(n) = T(n-1) + n = T(n-2) + (n-1) + n = T(n-3) + (n-2) + (n-1) + n$. Assuming that $T(1) = 1$, we can see that $T(n) = \frac{n(n+1)}{2}$, meaning the time is clearly $O(n^2)$, which is slower than $O(n^{log_2 3})$ from (d). □