4005-800 Algorithms

Homework 2

Christopher Wood

March 31, 2012

PROBLEM 1.

Solution.

The time complexity of the recurrence F_n can be characterized using a recurrence relation that defines the number of recursive calls made by F_n, as shown below.

$$
\begin{aligned}
T_F(0) &= 1 \\
T_F(1) &= 1 \\
T_F(n) &= T_F(n-1) + T_F(n-2)
\end{aligned}
$$

The solution to $T_F(n)$ can be solved using the method of homogeneous equations, which yields the result that $T_F(n) = \frac{1}{\sqrt{5}}(\phi^n - \phi'^n)$, where $\phi = \frac{1+\sqrt{5}}{2}$ and $\phi' = \frac{1-\sqrt{5}}{2}$. Thus, we can express this solution, and subsequentially the time complexity of F_n, as a function of exponential growth of n, as shown below.

$$
T_F(n) \in \Theta(\phi^n) \implies T_F(n) = \Theta(\phi^n)
$$

PROBLEM 2.

Solution.

The time complexity of the $fibIt$ routine can be found by solving the recurrence relation that defines $fibIt$. Such a recurrence relation can be defined by analyzing the number of additions performed during each call to $fibIt$, which is captured in the following set of equations.

$$
\begin{aligned}
T_f(0) &= 0 \\
T_f(1) &= 0 \\
T_f(n) &= T_f(n-1) + 1
\end{aligned}
$$

This is because there is only one addition made in each recursive call from $f(n; a, b)$ to $f(n-1; b, a+b)$, and there are no additions made in the two cases where $n = 0$ and $n = 1$.

In order to solve this recurrence relation we can expand out the expression and attempt to

identify the pattern (i.e. the **method of backwards substitution**). This process is shown below.

$$
\begin{aligned}
T_f(n) &= T_f(n-1) + 1 \\
&= (T_f(n-2) + 1) + 1 = T_f(n-2) + 2 \\
&= (T_f(n-3) + 1) + 2 = T_f(n-3) + 3 \\
&= \ldots \\
&= (T_f(n-k) + 1) + k = T_f(n-k) + k
\end{aligned}
$$

Based on this pattern, we can reach the first base case of this recurrence relation $(T_f(1))$ when $(n-k) = 1$, meaning that $k = (n-1)$. Thus, we have the following.

$$
\begin{aligned}
T_f(n) &= T_f(n - (n-1)) + (n-1) \\
&= T_f(1) + (n-1) \\
&= 0 + (n-1) \\
&= n - 1
\end{aligned}
$$

Based on this observation we can see that $T_f(n) \in \Theta(n)$, or simply $T_f(n) = \Theta(n)$.

PROBLEM 3.

Solution.

  Base $(n = 0)$

When $n = 0$, we know that $L^0(a, b) = (a, b)$ because the operator L is applied 0 times to (a, b). Furthermore, by definition of f, we know that $(f(0; a, b), f(1; a, b)) = (a, b)$. Thus, $L^0(a, b) = (f(0; a, b), f(1; a, b))$.

  Induction $(n > 0)$

First, we assume that $L^n(a, b) = (f(n; a, b), f(n+1; a, b))$. Now we show that $L^{n+1}(a, b) = (f(n+1; a, b), f(n+2; a, b))$.

$$
\begin{aligned}
L^{n+1}(a, b) &= L(L^n(a, b)) \text{ (by law of exponents)} \\
&= L(f(n; a, b), f(n+1; a, b)) \text{ (by the induction hypothesis)} \\
&= (f(n+1; a, b), f(n; a, b) + f(n+1; a, b)) \text{ (by definition of } L) \\
&= (f(n+1; a, b), f(n+2; a, b)) \text{ (by Theorem 1)}
\end{aligned}
$$

Thus, $L^{n+1}(a, b) = (f(n+1; a, b), f(n+2; a, b))$, as desired. Therefore, we know that $f(n; a, b) = (L^n(a, b))_1$.

PROBLEM 4-a.

Solution.

L can be represented as the product of two matrices, as shown below.

$$L \begin{pmatrix} a \\ b \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} \begin{pmatrix} a \\ b \end{pmatrix} = \begin{pmatrix} b \\ a + b \end{pmatrix}$$

PROBLEM 4-b.

Solution.

We can use the method of repeated squaring to achieve fast exponentiation in $\Theta(\log n)$ time. The source code for this routine is shown below.

```
1  def power(o, p):
2          if (p == 0):
3                  return IDENTITY
4          elif (p == 1):
5                  return o
6          elif ((p % 2) == 0):
7                  return power(o * o, p / 2)
8          else:
9                  return o * power(o * o, (p - 1) / 2)
```

PROBLEM 4-c.

Solution.

Using the representation for L and the *power* function described above, we can implement fibPow as follows:

```
1  def fibPow(n):
2          o = L(0,1)
3          o = power(o, n)
4          return o[0]
```

PROBLEM 4-d.

Solution.

Since the time to perform matrix multiplication with a $2x2$ and $2x1$ matrix is constant time (i.e. $\Theta(1)$), and the multiplication routine of repeated squares that surrounds this constant operation runs in $\Theta(\log n)$ time, we can conclude that the time complexity of $fibPow$ is $\Theta(\log n)$.

PROBLEM 5-a. *Write down the definition of pseudo-polynomial time.*

Solution.

Definition 1. A pseudo-polynomial-time algorithm is one that runs in time polynomial in the dimension of the problem and the magnitudes of the data involved (provided these are given as integers), rather than the base-two logarithms of their magnitudes .

PROBLEM 5-b. *Is fib a pseudo-polynomial time algorithm? Explain.*

Solution. No - exponential in value of input (not polynomial).

No, fib has a time complexity of $\Theta(\phi_n)$, where n is the input magnitude, which means that it is exponential in the magnitude of the input, which further implies that it is not polynomial in the magnitude of the input. Therefore, by definition, fib cannot be a pseudo-polynomial time algorithm.

PROBLEM 5-c. *Is $fibIt$ a pseudo-polynomial time algorithm? Explain.*

Solution. Yes - polynomial in value of input, which is exponential in number of bits in the input (base 2 logarithm)

Yes, $fibIt$ has a time complexity of $\Theta(n)$, which can also be defined as $\Theta(2^{\lg n})$, where n is the input magnitude, which means that it also has polynomial time complexity in the magnitude of n but exponential time in the bit size of of n. Therefore, by definition, $fibIt$ must be a pseudo-polynomial time algorithm.

PROBLEM 5-d. *Is $fibPow$ a pseudo-polynomial time algorithm? Explain.*

Solution. Don't believe so, it isn't exponential in the magnitude of the input (see below) Is 2**(lglgn) exponential? If so, then yes, if not, then no.

No, $fibPow$ has a time complexity of $\Theta(\lg n)$, which can also be defined as $\Theta(2^{\lg \lg n})$, where n is the input magnitude, which means that it is sub-polynomial in the magnitude of n and sub-exponential in the bit size of n. Therefore, by definition, $fibPow$ must be a pseudo-polynomial time algorithm.