4005-800 Algorithms

Homework 2

Christopher Wood

April 3, 2012

PROBLEM 1.

Solution.

The time complexity of the recurrence fib 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 by making the observation that $T_F(n) = F_{n+1}$, which is proved below using strong induction.

Base $(n = 0)$

By definition we know that $T_F(0) = 1$ and $F_{0+1} = F_1 = 1$. Thus, $T_F(0) = F_1$.

Base $(n = 1)$

By definition we know that $T_F(1) = 1$ and $F_{1+1} = F_2 = F_1 + F_0 = 0 + 1 = 1$. Thus, $T_F(1) = F_2$.

Induction $(n > 1)$

Assume that $T_F(k) = F_{k+1}$ for all k such that $2 \leq k < n$. We will show that $T_F(n) = F_{n+1}$.

$$\begin{aligned} T_F(n) &= T_F(n-1) + T_F(n-2) \text{ (by definition of } T_F) \\ &= F_n + F_{n-1} \text{ (by the induction hypothesis)} \\ &= F_{n+1} \text{ (by definition of } F_n) \end{aligned}$$

The solution to the recurrence $T_F(n) = F_{n+1}$ can then be solved by treating it as a homogeneous second-order linear recurrence with constant coefficients, which yields the result that $T_F(n) = F_{n+1} = \frac{1}{\sqrt{5}}(\phi^{n+1} - \phi'^{n+1})$, where $\phi = \frac{1+\sqrt{5}}{2}$ and $\phi' = \frac{1-\sqrt{5}}{2}$. Thus, we can express this solution, and subsequently the time complexity of fib, as a function of exponential growth of n, as shown below.

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

However, without loss of generality, we can also solve $T_F(n)$ using the substitution approach,

knowing that $T_F(n) = \Theta(a^n)$ for some a, where a is a real number.

$$\begin{aligned} T_F(n) &= T_F(n-1) + T_F(n-2) \\ a^n &= a^{n-1} + a^{n-2} \text{ (by substitution)} \\ a^2 &= a + 1 \text{ (divide by } a^{n-2}) \\ a^2 - a - 1 &= 0 \text{ (move terms to one side)} \end{aligned}$$

At this point one can see we have a very simply quadratic equation with two roots. We can solve for these roots using the quadratic equation, which yields $a = \frac{1 \pm \sqrt{5}}{2}$. Thus, since $a = \frac{1+\sqrt{5}}{2} = \phi$ is the larger of the two roots and $a = \frac{1-\sqrt{5}}{2}$ is negative, we can conclude that $a = \frac{1+\sqrt{5}}{2} = \phi$, which implies that $T_F(n) = \Theta(\phi^n)$.

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 iteration**). This process is shown below.

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

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 clearly see that $T_f(n) \in \Theta(n-1) = \Theta(n)$, or simply $T_f(n) = \Theta(n)$. Therefore, the time complexity of $fibIt$ is $\Theta(n)$.

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.

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

$$L^n\begin{pmatrix} a \\ b \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}^n\begin{pmatrix} a \\ b \end{pmatrix} = \begin{pmatrix} f(n;a,b) \\ f(n+1;a,b) \end{pmatrix}$$

PROBLEM 4-b.

Solution.

We can use the method of repeated squaring to achieve fast exponentiation of these matrices in $\Theta(\log n)$ time. The recursive definition for this algorithm is shown below.

$$a^n = \begin{cases} (a^{\frac{n}{2}})^2, & \text{if n is even and positive,} \\ (a^{\frac{(n-1)}{2}})^2 * a, & \text{if n is odd,} \\ 1 & \text{if n = 0} \end{cases}$$

Careful analysis reveals that this algorithm runs in $\Theta(\log n)$ time because, on each iteration, the size is reduced by about a half at the expense of one or two multiplications [1]. This assumes of course that the time for multiplication is $\Theta(1)$.

The source code for this routine (which can be implemented both recursively and iteratively) is shown below.

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

```
1  def matrixPower(base, p):
2          result = IDENTITY
3          while (p != 0):
4                  if ((p % 2) != 0):
5                          result = result * base
6                          p = p - 1
7                  base = base * base
8                  p = p / 2
9          return result
```

Finally, we can implement a single *power* routine that uses either one of these multiplication functions to raise objects of type L to the nth power that is $\Theta(tlogn)$, as shown below.

```
1  def power(base, p):
2          matrix = matrixPower(base.m, p)
3          v1 = (matrix[0] * base.a) + (matrix[1] * base.b)
4          v2 = (matrix[2] * base.a) + (matrix[3] * base.b)
5          return (v1, v2)
```

PROBLEM 4-c.

Solution.

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

```
1  def fibPow(n, a, b):
2          base = L(a,b,0,1,1,1)
3          return base.power(n)
```

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 utilizes 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 polynomial time (i.e. $\Theta(n^k), n \geq 1$) with respect to the value n of its input, but does not run in polynomial time with respect to the size b of its input (i.e. it must run in exponential time with respect to the size of the input, or simply $\Theta(m^b), m > 1$). In other words, when working with numerical input n to an algorithm, we can say that such an algorithm runs in pseudo-polynomial time if its time complexity is polynomial with respect to n and exponential with respect to $b = \lg(n)$, because b represents the approximate number of bits used to represent n (i.e. the size of n).

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

Solution.

No, fib has a time complexity of $\Theta(\phi^n)$, where n is the input value, which means that it is exponential with respect to the value of the input, which further implies that it is not polynomial with respect to the value of the input. Therefore, by definition, fib is not a pseudo-polynomial time algorithm.

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

Solution.

Yes, $fibIt$ has a time complexity of $\Theta(n)$, which can also be defined as $\Theta(2^{\lg n})$, where n is the input value and $\lg(n)$ is the bit size of n. This means that it $fibIt$ has polynomial time complexity

with respect to the value of n and exponential time with respect to the size of n. Therefore, by definition, $fibIt$ is a pseudo-polynomial time algorithm.

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

Solution.

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 value and $\lg n$ is the bit size of n. This means that $fibPow$ has a sub-linear time complexity with respect to the value of n (i.e. it does not have a time complexity of $\Theta(n^k), k \geq 1$) and thus sub-exponential time complexity with respect to the size of n. Therefore, since $fibPow$ does not have polynomial runtime in the value of n, nor does it have exponential time complexity in the size of n, it is not a pseudo-polynomial time algorithm.

References

[1] Levitin, Anany. *Introduction to the Design and Analysis of Algorithms.* Pearson, Boston: 2012. Print.