Homework 1

Michael Anderson

April 6, 2011

CS517

Prof. Cull

1

a) Show log n! $= O$(n log n):

$$\log n! = \log(1 \times 2 \times \ldots \times n) = \log 1 + \log 2 + \ldots + \log n =$$

$$(\frac{\log 1}{\log n} + \frac{\log 2}{\log n} + \ldots + \frac{\log(n-1)}{\log n} + \frac{\log n}{\log n}) \log n$$

Now since $\log x$ increases monotonically as x increases:

$$(\frac{\log 1}{\log n} + \frac{\log 2}{\log n} + \ldots + \frac{\log(n-1)}{\log n} + \frac{\log n}{\log n}) \log n < (1_1 + 1_2 + \ldots + 1_n) \log n = n \log n$$

b) Show log n! $= \Omega$(n log n): Not sure about this one. Even though it seems clear that $\log n!$ approaches $n \log n$, as n goes to infinity, I do not know how to formally bound the ratio between them by a constant.

2

Since the algorithm is recursive and the proof method is induction, work from the bottom up. Suppose that $(x_n, x_{n-1}, \ldots, x_1, x_0)$ is the set of all a and b values given by the algorithm. In other words x_n is the initial a, x_{n-1} is the initial b, x_1 is the final a and the algorithm's return value, and $x_0 = 0$.

- **Basis step.** The GCD of x_1 and $x_0 = 0$ is clearly x_1, as the algorithm returns.
- **Inductive step.** Assume that Euclid(x_n, x_{n-1}) is the correct GCD of x_n and x_{n-1}. Euclid(x_{n+1}, x_n) will the return the same value, must show that this is also the GCD of x_{n+1} and x_n.

  By the algorithm we have:

  $$x_{n-1} = x_{n+1} \mod x_n$$

  or alternatively:

  $$x_{n+1} = kx_n + x_{n-1} \text{ for some natural number k}$$

  Let g be the GCD of x_n and x_{n-1}. Since g is a common divisor of these two values, then it must also divide $kx_n + x_{n-1}$, because it divides both kx_n and x_{n-1} it must also divide their sum.

  Suppose there is some natural number $h > g$ that divides both x_{n+1} and x_n, which would invalidate the algorithm. This implies that it divides

$x_{n+1} = kx_n + x_{n-1}$, but since it divides kx_n it must also divide x_{n-1}, else it could not divide x_{n+1}. This means that h is a divisor of x_n and x_{n-1} that is larger than the GCD of those two values, which is a contradiction. Therefore such an h cannot exist, and if g is the GCD of x_n and x_{n-1}, it is also the GCD of x_{n+1} and x_n.

3

Since $nn^K = n^{K+1}$, and since n is not upper bounded by any constant as it goes to infinity, there can be no constant c such that $cn^K \geq n^{K+1}$ as n goes to infinity. This means that $n^K \neq \Omega(n^{K+1})$, and furthermore $n^K \neq \Theta(n^{K+1})$

4

If n is a natural number, then its positive square root lies in the interval [0,n]. Iff n is a perfect square, then its positive square root is also a natural number. The algorithm iterates through all of the natural numbers in the interval [0,n], squares them for comparison with n, returns YES when a match is found, and NO if a match is not found anywhere in the interval [0,n]. Therefore, the algorithm is a recognizer for the set of perfect squares.

Assume that the I^2 operation is performed using naive schoolbook multiplication, which requires k^2 time for k bit operands. This is the meat of the computation. Since in the worst case the algorithm iterates over $n = \Theta(2^k)$ values and squares each of them, the algorithm has a runtime of $\Theta(2^k k^2)$.

This algorithm is not reasonable in the sense that it does not execute in polynomial time as a function of the bit length of its input. It has exponential runtime in k.

5

To figure out if n is a perfect square, simply throw it into MAGIC(). Then n is a perfect square if MAGIC outputs $\sqrt{n}$. Otherwise, n is not a perfect square.

```
SQ_RECOGNIZER(n):
    IF MAGIC(n)^2 = n
        RETURN YES
    ELSE
        RETURN NO
```

Here MAGIC(n) takes $\log n$ time, and squaring a value again takes $(\log n)^2$ time. So the total runtime is $\Theta((\log n)^2) + \Theta(\log n) = \Theta((\log n)^2) = \Theta(k^2)$, where k is again the size of the input in bits.

6

Since the Fibonacci recurrence has a closed-form solution, there are more efficient methods for identifying Fibonacci numbers than the following, but the following is simple. Just compute the values of Fibonacci sequence iteratively. Since the Fibonacci sequence is monotonically increasing, if the at any point the input is less than the last member of the sequence computed, it is safe to return NO. If a member of the sequence is found to be equal to the input, return YES.

```
FIB_RECOGNIZER(x):
    a = 0
    b = 1
    WHILE a < x
        temp := a + b
        a := b
        b := temp
    IF a == x
        RETURN YES
    ELSE IF a > x
        RETURN NO
```

In order to relate the size of the input to the rate at which the Fibonacci sequence grows, solve the characteristic equation of the Fibonacci recurrence:

$$r^2 = r^1 + r^0$$

The quadratic formula yields:

$$r = \frac{1+\sqrt{5}}{2}, r = \frac{1-\sqrt{5}}{2}$$

Let $\varphi = \frac{1+\sqrt{5}}{2}$. So the Fibonacci sequence is $\Theta(\varphi^n)$, where n is the index of the sequence. The bit length k of the nth member of the Fibonacci sequence is then given as:

$$k = \lceil \log_2 \varphi^n \rceil = \lceil \frac{n}{\log_\varphi 2} \rceil = \Theta(n)$$

FIB_RECOGNIZER(x) runs until a value of the Fibonacci sequence is generated that is $\geq x$, so it must run $\Theta(n)$ times if F_n is the first Fibonacci number larger than x. Since $k = \Theta(n)$, FIB_RECOGNIZER() runs in linear time relative to the bit length of the input.