Homework 2

Jeff Fennell, Misa Pham, Joseph Barbosa, Matt Flickner

March 1, 2015

0.1

$a)(f(n) = n - 100\ g(n) = n - 200.$
$f = \theta(g)$

$b)f(n) = n^{1/2}\ g(n) = n^{2/3}$
$1/2 < 2/3$
$f = O(g)$

$c)f(n) = 100n + logn\ g(n) = n + (logn)^2$
$f = \theta(g)$

$d)f(n) = nlogn\ g(n) = 10nlog10n$
$f = \theta(g)$

$e)f(n) = log(2n)\ g(n)log3n$
$f = \theta(g)$

$f)f(n) = 10logn\ g(n) = log(n^2)$
$f = \theta(g)$

$g)f(n) = n^{1.01}\ g(n) = nlog(n^2)$
$f = \Omega(g)$

$h)f(n) = n^2/logn\ g(n) = n(logn)^2$

$f = \Omega(g)$

$i) f(n) = n^{0.1}\ g(n) = (logn)^10$
$f = \Omega(g)$

$j) f(n) = (logn)^{logn}\ g(n) = n/logn$
$f = \Omega(g)$

$k) f(n) = \sqrt{n}\ g(n) = (logn)^3$
$f = \Omega(g)$

$l) f(n) = n^{1/2}\ g(n) = 5^{logn}$
$f = O(g)$

$m) f(n) = n2^n\ g(n) = 3^n$
$f = O(g)$

$n) f(n) = 2^n\ g(n) = 2^{n+1}$
$f = \theta(g)$

$o) f(n) = n!\ g(n) = 2^n$
$f = \Omega(g)$

$p) f(n) = (logn)^{logn}\ g(n) = 2^{(log_2(n))^2}$
$f = O(g)$

$q) f(n) = \sum_{i=1}^{n} i^k\ g(n) = n^{k+1}$
$f = \theta(g)$

0.4

a) Matrix 1

$$\begin{pmatrix} a & b \\ c & d \end{pmatrix}$$

Matrix 2

$$\begin{pmatrix} e & f \\ g & h \end{pmatrix}$$

Result

$$\begin{pmatrix} i & j \\ k & l \end{pmatrix}$$

The rules of matrix multiplication are as follows:
$i = (a \times e) + (b \times g)$
$j = (a \times f) + (b \times h)$
$k = (c \times e) + (d \times g)$
$l = (c \times f) + (d \times h)$

From this, we see that multiplying two 2 × 2 matrices involves 8 multiplications and 4 additions.

b) Say we have X^8 where X is a 2 × 2 matrix and n = 8. Let's refer to these 8 2 × 2 matrices as a, b, c, d, e, f, g, and h. We begin the matrix computation by multiplying matrix a with matrix b, matrix c with matrix d, matrix e with matrix f, and matrix g with matrix h. This results in 4 new matrices, lets call them i, j, k, and l. The next step would then involve two multiplications (i × j & k × l). This results in 2 matrices that are then multiplied together to obtain the final answer. The number of steps taken to obtain this answer is $log_2(8) = 3$ steps. In each step, the number of multiplications performed is $n/2$ so the first step performs 4 multiplications then 2 and finally 1 for a total of 7 multiplications.

1.2

In order to find the ratio of digits used in binary to express a base ten number, we take the log base 2 of 10. So we get,

$log_2(10) = 3.32$

To represent decimal numbers 8 and 9, one must use 4 binary digits. Numbers less than 8 can be expressed in binary with 3 or less digits. For all other larger decimal numbers, the ratio of binary digits to decimal digits is 3.32:1. This shows that to express a decimal digit in binary, it will take at most 4 binary digits.

1.4

To show that $log(n!) = \theta(nlogn)$ we can show the upper and lower bounds to be equal to be $nlogn$. One way to do this is to show how both can be equal to $nlogn$, as follows:
Log rules state that $log(n^n) = nlogn$. So for the case of upper bound being n^n, $log(n!) = log(1) + ... + log(n)$ $which\ is$ $<= log(n) + ... + log(n)$ therefore, the upper bound is nlogn. Likewise, for the case of $(n/2)^{n/2}, log(1) + ... + log(n/2) + ... + log(n)$ $which\ is$ $>= log(n/2) + ...log(n)$ $since$ $log(1) + ... + log(n/2) >= 0$. Thus, we can say that it is also greater than $log(n/2) + ... + log(n/2)$ $which\ is$ $log(n/2)$ $n/2$ $times$. So we can say that it is in essence equal to the $nlogn$ function we have defined as our upper limit. So, we can say that $log(n!) = \theta(nlogn)$

1.11

Q: Is $4^{1536} - 9^{4824}$ divisible by 35?

A: Yes. $(4^{1536} - 9^{4824})\text{mod}35 = 0$.

1.13

Q: Is $5^{30000} - 6^{123,456}$ a multiple of 31?

A: Yes. $(5^{30000} - 6^{123,456})$mod31 is 0, so $31n$ where n is an integer gives us $5^{30000} - 6^{123,456}$.

1.16

When $a^b \mod c$ and a = 2, b = 11, and c = 16 $2^{11} \mod 16 (2^4 \times 2^7) \mod 16 (2^4 \mod 16) + (2^7 \mod 16) 0 + (2^7 \mod 16)(2^4 \times 2^3) \mod 16 (2^4 \mod 16) + (2^3 \mod 16) 0 + (2^3 \mod 16) 8 \mod 16$

1.33

```
class lcm {
    public static void main(String[] args) {
        int x = Integer.parseInt(args[0]);
        int y = Integer.parseInt(args[1]);
        int gcd = gcd(x, y);

        System.out.println((x * y) / gcd);
    }

    public static int gcd(int x, int y) {
        return y == 0 ? x : gcd(y, x % y);
    }

}
```

The runtime of the lcm as a whole can be reduced to a multiplication and a division, which are both $\theta(1)$ + the complexity of the GCD algorithm.

The GCD algorithm continually divides the first number, x, by the second number,y. The GCD algorithm breaks out of the recursive call when x % y is zero. Therefore, it will break out of the recursive call and return when we have done $log_y(x)$ divisions. There is a constant ration between the logs of different bases, so the magnitude of y doesn't matter. So the complexity of the GCD function is log (n), where n is the magnitude of x.

The total complexity is: $\theta(1)$ - for multiplication $\theta(1)$ - for division $\theta(log(n))$ for the gcd function

Therefore, the complexity of the problem is $\theta(log(n))$, where n is the magnitude of x.

1.35d

Due to the fact that Wilson's theorem is an if-and-only-if condition, one must solve for (N-1)! factorial, where N is the number in question, in order to determine primality. The left hand side of the equation must be solved first in order to determine primality but it is difficult to calculate (N-1)! especially when N is large number. With a^b, it can easily be made more efficient by using the binary representation of b. This allows you to do $logb$ calcuations instead of a very large b number of calculations. Because factorial multiplies by n-1 each time, this efficient method used on a^b cannot be used. Therefore,

a primality test cannot be immediately based on this rule because it would take too long to calculate $(N-1)!$.

1.39

Give a polynomial-time algorithm for computing $(a^b)^c \text{mod} p$ given a, b, c, and prime p.

```
import sys

#primality test grabbed from:
#http://en.wikipedia.org/wiki/Primality_test#Python_implementation
def is_prime(n):
    if n <= 3:
        return n >= 2 #1 is not prime
    if n % 2 == 0 or n % 3 == 0:
        return False
    for i in range(5, int(n ** 0.5) + 1, 6):
        if n % i == 0 or n % (i + 2) == 0:
            return False
    return True


def modpow(x, y, n):
    if y==0 or x==1:
        return 1
    elif x==0:
        return 0
    else:
        output = x
        while y>1:
            output = (output*x)%n
            y = y-1
    return output

try:
    a= int(sys.argv[1])
```

```
    b= int(sys.argv[2])
    c= int(sys.argv[3])
    p= int(sys.argv[4])
except ValueError:
    print "enter numbers ya big dummy"
    sys.exit(1)
if(not is_prime(p)):
    print "p has to be prime ya big dummy"
    sys.exit(1)

print(modpow(modpow(a,b,p),c,p))
```