**Allison Neyer**

**Algorithms Hw#2**

**Exercises**

0.1. In each of the following situations, indicate whether f = O(g), or f = Ω(g), or both (in which case

f = Θ(g)).

|  |  |  |  |
| --- | --- | --- | --- |
|  | f(n) | g(n) |  |
|  | n-100 | n-200 | **f = Θ(g)**. both upper and lower bound are the same. Function of n. |
|  | n1/2 | n2/3 | **f = O (n2/3)** |
|  | 100n+logn | n +(logn)2 | **f = Θ(g)**  upper = lower bound are the same.  n+(logn)2 => log(a)b = blog(a)=> 2 loga |
|  | nlogn | 10n log (10n) =  10n log 10+ 10n log n | **f = O(g)** since g(n) is the upper bound of f |
|  | log(2n) = log (2) + log (n) | log (3n) = log (3) + log(n) | since log(3) is slightly bigger than log (2n) but you can multiply function g by a constant where it is both the upper and lower bound, **f = Θ(g)** |
|  | 10 log n | log(n2) = 2 log (n) | **f = Θ(g)** because 30 \* (2 log (n)) > 10 log n > 2 log (n). |
|  | n1.01 = n × n.01 | nlog2n = n × log(n) × log(n) | **f = O (n.01)**  because log(n) x log(n) > n.01 |
|  | = | n(logn)2 | **f = O ((log(n))2)**  because (log(n))2 canbe multiplied by a constant where (log(n))2 **>** n/log(n) |
|  | n.01 | (logn)10 | **f = Ω(g)** because n.01 is a polynomial which dominates (logn)10 |
|  |  | n/logn | **f = Ω(g)** because f > g ultimately. That is the exponential dominates the polynomial. |
|  |  | (logn)3 | f **= Ω(g)** because f > g since polynomials dominate log |
|  |  |  | since exponential g function dominates the polynomial f function (meaning g > f) then g is the upper bound of f so **f = O(g)**. |
|  | n2n | 3n | **f = O(g)** because f< g since g is the bigger exponential. |
|  | 2n | 2n+1 = 21 \*2n => 2n | **f = Θ(g)**  because it’sbasically the same function and can multiply by a constant so the function g is both the upper and lower bound of f. |
|  | n! | 2n | The exponential 2n is the upper bound for the f function. So **f = O(g)** because f< g. |
|  |  |  | So **f = O(g)** because f< g. Since g is the exponential function of an exponential function and f is an exponential of a logarithmic function so since exponential functions are greater than logarithmic f <g. |
|  |  |  | **f = O(g)** because f< g since g is the bigger polynomial. |

.4a) Show that two 2x2 matrices can be multiplied using 4 additions and 8 multiplications:

X and Y are two 2x2 matrices:

X =

Then their products are:

XY =

p.57 of textbook

So, there are 4 additions, ( 1)AE+BG, 2) AF+BH, 3) CE+DG, and 4) CF+DH), and there are 8 multiplications here, ( 1) AE, 2)BG, 3) AF, 4) BH, 5) CE, 6)DG, 7) CF, and 8) DH ).

.4b)

Show that *O*(log*n*) matrix multiplications suffice for computing Xn. (Hint:Think about computing X8)

<http://math.stackexchange.com/questions/209062/prove-a-bound-on-matrix-multiplication>:

*X*=

Compute the power by using the [Binary Method for Exponentiation.](http://en.wikipedia.org/wiki/Exponentiation_by_squaring). The idea is of great usefulness in computation, so there are many variant executions.

First calculate the binary expansion of the exponent *n*. So we find bits *a*0,*a*1,*a*2,…,*ak* such that

*n=a020+a121+a222+⋯+ak2k. (1)*

Calculate X0 = I, X1, X2, X4, and so on up to by repeated squaring. This works because = )2.

Finally, find the product of all the with *ai*=1. This product is Xn. That follows from the fact that = .

The *k* of Formula (1) is about the size , so the repeated squarings use about matrix multiplications. The multiplications at the end, for the *ai*=1, take (at most) about matrix multiplications.

**Remark:** The powers of the particular matrix mentioned in the problem are intimately connected with the Fibonacci numbers. So in particular the binary method for matrix exponentiation is useful for calculating *Fn* for largish *n*. Because of the rapid growth of the Fibonacci numbers, it is best to use exact integer arithmetic.

1.2)

For any arbitrary integer n we can find a p1 such that . If we take log of base 2 on both sides, then we get Similarly, we can find an integer n such that So p1 ≥ the maximum number of digits needed for binary representation. So if p1 is the smallest integer where , then p1 is equal to the maximum digits needed for n’s binary representation. Just like 99 < 103 for any 3 digit number so 3 is max digits needed for representation of 3 digit number. If you take the ratio of p1 to p2, then

= . Cancel out the log10(n) to get = 3.3. If you round that up that is 4.

P1 is max digits needed for binary. P2 is max digits needed for decimal representation. So the ratio is 3.3 and p1 is at most 4 times p2. Therefore, the binary integer is at most four times as long as the corresponding decimal integer.

1.4)

Log(n!) = logn +log(n-1) +…

Log nn = n logn

Log(n!) > lognn

As <http://stackoverflow.com/questions/2095395/is-logn-%CE%98n-logn> says:

log(n!) = log(1) + log(2) + ... + log(n-1) + log(n)

You can get the upper bound by

log(1) + log(2) + ... + log(n) <= log(n) + log(n) + ... + log(n)

= n\*log(n)

And you can get the lower bound by doing a similar thing after throwing away the first half of the sum:

log(1) + ... + log(n/2) + ... + log(n) >= log(n/2) + ... + log(n)

>= log(n/2) + ... + log(n/2)

= n/2 \* log(n/2)

Using the hint:

n! (= 1\*2\*3\*...\*n) is a product of n numbers each less than or equal to n. Therefore it is less than the product of n numbers all equal to n; i.e., n^n.

Half of the numbers -- i.e. n/2 of them -- in the n! product are greater than or equal to n/2. Therefore their product is greater than the product of n/2 numbers all equal to n/2; i.e. (n/2)^(n/2).

Take logs throughout to establish the result.

<http://stackoverflow.com/questions/2095395/is-logn-%CE%98n-logn>

1.11) Is 41536 *−* 94824 divisible by 35? **yes**.

First, irb ruby says: (4\*\*1536 - 9\*\*4824) % 35 = 0 so it is divisible.

page 6 of <http://cs.gmu.edu/~lifei/teaching/cs483_fall08/assignment3.pdf>

note: • = x = multiplication

Proof:

35 = 5 x 7; 5 and 7 are primes.

**By Fermat’s Little Theorem**, for any prime p and 1 ≤ a < p, ap−1  ≡ 1 mod p.

Thus, a5−1 ≡ 1 mod 5 and a7−1 ≡ 1 mod 7. Furthermore, we have (a5−1)7−1 = (a4)6 = a24 ≡ 1 mod (5 • 7). That is a24 ≡ 1 mod 35, for all 1 ≤ a < 35. Therefore, 41536 = 424•64 ≡ 1 mod 35 and 94824 = 924•201 ≡ 1 mod 35. We conclude that 41536 ≡ 94824 mod 35. So the difference is divisible by 35.

1.13) Is the difference of 530,000 *–* 6123,456 a multiple of 31? **yes**.

<http://userpages.umbc.edu/~kavitad1/HW2_Solutions.pdf> :

By Fermat’s little theorem (see fermat’s little theorem in problem 1.11) , p = 31 is prime and so 5^30 = 1 (mod 31).

Likewise, 6^30 = 1 (mod 31)

Now, 5^30000 = 5 ^ rem(30000,30) (mod 31) = 1

6^123456 = 6^6\*6^123450 = 6^6 \*6^rem(123450,30) (mod 31) = 6^6 (mod 31) = 1 Hence 5^30000 – 6^123456 = (1-1) (mod 31) = 0.

Hence the difference is a multiple of 31.

1.16) The algorithm for computing *ab* mod *c* by repeated squaring does not necessarily lead to the minimum number of multiplications. Give an example of *b >* 10 where the exponentiation can be performed using fewer multiplications, by some other method.

**An example: Here, there is 1 multiplication since 0 times anything is 0 and anything mod with 0 is 0 so 1 step multiplication as long as mod is greater than 0 and is an integer. In this example, a just has to be 0.**

1.25) Calculate 2125 mod 127 using any method you choose. (*Hint:* 127 is prime.)

[**http://www.wolframalpha.com/input/?i=Calculate+2^125+mod+127**](http://www.wolframalpha.com/input/?i=Calculate+2%5e125+mod+127)**+ says:**

**2^125 mod 127 is 64.**

**Also, irb says: (2\*\*125) % 127 = 64.**

<https://answers.yahoo.com/question/index?qid=20090317054940AA4jB4p>

127 = 2^7 - 1

2^125= 2^7 \* 2^118

= 2^118 \* 2^7 - 2^118 + 2^118

=2^118(2^7 -1) +2^118

=2^118(2^7 -1) +2^111 \* 2^7 -2^111 +2^111

= (2^118+2^111)((2^7 -1)+2^104 \* 2^7-2^104 + 2^104

=(2^118+2^111+2^104)((2^7 -1)+2^104.....

=(2^118 + 2^111 ...2^13)(2^7-1) + 2^13]

Divide by 2^7 - 1

=(2^118 + 2^111 ...2^13)+ 2^13/(2^7-1)

2^13 mod 127 = **64**

1.33)

**see Lcm.rb file for more info.**

Ruby Gcd function modified slightly from (<http://cs.lmu.edu/~ray/notes/numalgs/> ):

def gcd(x, y)

return y == 0 ? x : gcd(y, x % y)

end

The least common multiple of a and b is the product divided by the greatest common divisor.   
I.e. lcm(a, b) = ab/gcd(a, b). (<http://stackoverflow.com/questions/3154454/what-is-the-most-efficient-way-to-calculate-the-least-common-multiple-of-two-int>)

Complexity time :

ab which is constant complexity of 1 (because no matter how many digits the input numbers are it will still be one operation (multiplication for ab and division for division operation). This constant time is overwhelmed by the log time of the gcd function. Hence, the running time of the algorithm is **log a.**

Lcm program (also **see Lcm.rb file**) :

class Lcm

def gcd(x, y)

return y == 0 ? x : gcd(y, x % y)

end

def lcm(x,y)

return (x\*y)/gcd(x,y)

end

# for quick testing purposes

# x = Lcm.new

# puts "hi"

# puts "gcd(2,3): "

# puts "gcd(2,3): #{x.gcd(2,3)}."

# puts "lcm(12,18): #{x.lcm(12,18)}."

end

1.35d) Since Wilson’s theorem is an if-and-only-if condition for primality, we can’t immediately base a primality test on this rule because it is too slow to do it immediately because it is too complicated and hard to calculate whether something is exactly prime. This takes too much memory and also because it is an n!, one can’t group the factorial. Therefore, we can’t immediately base a primality test on this rule.

1.39)

See computing.py