Algorithms
By
Sanjoy Dasgupta Christos Papadimitriou Umesh Vazirani
Chapter 1: Algorithms with numbers

Yukteshwar Baranwal

August 19, 2018

Problem 1.1: Show that in any base $b \geq 2$, the sum of any three single-digit numbers is at most two digits long.

Maximum possible number in base b is $b-1$ e.g. 1 in binary($b=2$) and 9 in decimal($b=10$). On adding $b-1$ thrice, the result will be $3 \times (b-1)$, which is at most 2 bit long as $3 \times (b-1) < b^3$.

Problem 1.2: Show that any binary integer is at most four times as long as the corresponding decimal integer. For very large numbers, what is the ratio of these two lengths, approximately?

A number of n in base b has length $log(n)/log(b)$. So , length for binary is $log(n)/log(2)$ & for decimal is $log(n)/log(10)$. Their ratio is given by $log(10)/log(2) = 1/0.3010 > 3$ and its ceil value is 4.

Problem 1.3: A d-ary tree is a rooted tree in which each node has at most d children. Show that any d-ary tree with n nodes must have a depth of $\Omega(log(n)/log(d))$. Can you give a precise formula for the minimum depth it could possibly have?

For n child nodes and for a complete tree, minimum depth would be $log(n)/log(d)$. The sum of nodes over levels is a geometric series sum having value $(d^{k+1}-1)/(d-1)$ where k is depth of tree and this sum would be n. Hence, k would be $\Omega(log(n)/log(d))$. Precise representation would be $\lfloor log_d(n) \rfloor$.

Problem 1.4: Show that $log(n!) = \Theta(nlog(n))$.

Using hint, $n! < n^n$ & $n! > (n/2)^{n/2}$. Hence, $log(n!) < nlog(n)$ & $log(n!) < (n/2)log(n/2)$. The lower bound can also be expressed as $log(n!) < (1/2)(nlog(n) - n)$. Hence, $log(n!) = \Theta(nlog(n))$.

Problem 1.5: Refer question in book.

Based on the hint, we see that the harmonic series is $O(log_2(n))$ & $\Omega(log_4(n))$. Thus, the given harmonic series is $\Theta(log(n))$.

Problem 1.6: Prove that the grade-school multiplication algorithm (Refer book for page #), when applied to binary numbers, always gives the right answer.

It works for binary numbers too. Grad-school multiplication algorithm for $(x \times y)$, multiply x with every bits of y and each level shift multiplications results to left by one decimal places (which is similar to shifting one bit left) followed by addition.

Problem 1.7: How long does the recursive multiplication algorithm (Refer book for page #) take to multiply an n-bit number by an m-bit number? Justify your answer.

At each level, the m-bit number get reduced by one bit (being half) and you either multiply by 2 or add n-bit number with m-k bit number where k is the level. that's $O(max(m, n))$ at each step. So $O(mn)$.

Problem 1.8: Justify the correctness of the recursive division algorithm given in page (Refer book for page #), and show that it takes time $O(n^2)$ on n-bit inputs.

There are n recursive calls because of n halving that need to take place. Then, before each recursive call ends, there are arithmetic operations that are $O(n)$. Thus, the time complexity is $O(n^2)$. This algorithm is correct because what it is essentially doing is dividing 1 by the divisor, then doubling both the quotient and remainder, and then checking to see if the remainder has overflowed. It continues upwards until the original value is reached, at which point the quotient and remainder will be correct.

Problem 1.9: Refer question in book.

Since,

$$\begin{aligned}
x - x' &= Np \\
y - y' &= Nq \\
\Rightarrow (x + y) - (x' + y') &= (x - x') + (y - y') \\
(x + y) - (x' + y') &= N(p + q) \\
\Rightarrow (x - x') \times (y - y') &= N^2pq \\
xy - xy' - x'y + x'y' &= N^2pq \\
xy - x'y' - xy' - x'y + 2x'y' &= N6pq \\
xy - x'y' - (x - x')y' - x'(y - y') &= N^2pq \\
xy - x'y' &= N^2pq + Npy' + Nqx' \\
xy - x'y' &= N(Npq + py' + qx')
\end{aligned}$$

Problem 1.10: Show that if $a \equiv b(modN)$ and if M divides N then $a \equiv b(modM)$.
Since, $a - b = kN$ & $N = pM$, then $a - b = kpM$ & hence, $a \equiv b(modM)$.

Problem 1.11: Is $4^{1536} - 9^{4824}$ divisible by 35?
Since, $4^6 = 1(mod35) \Rightarrow 4^6 - 1 = 35p$ & $9^6 = 1(mod35) \Rightarrow 9^6 - 1 = 35q$. Thus, $4^6 - 9^6 = 35(p-q) \Rightarrow 4^6 - 9^6 = 0(mod35)$ & $4^{1536} - 9^{4824} = (4^6)^{256} - (9^6)^{804} = 0(mod35)$.

Problem 1.12: What is $2^{2^{2006}}(mod3)$?
$2 = -1(mod3) \Rightarrow (-1)^{2^{2006}} = 1(mod3)$. Hence, answer is 1.

Problem 1.13: Is the difference of $5^{30,000}$ and $6^{123,456}$ a multiple of 31?
Since, $5^4 = 5(mod31) \Rightarrow 5^4 - 5 = 31p$ & $6^2 = 5(mod31) \Rightarrow 6^2 - 5 = 31q$. Thus, $5^4 - 6^2 = 31(p-q) \Rightarrow 5^4 - 6^2 = 0(mod31)$ & $5^{30,000} - 6^{123,456} = (5^4)^{7500} - (6^2)^{61728} = 0(mod35)$.

Problem 1.14: Suppose you want to compute the nth Fibonacci number F_n , modulo an integer p. Can you find an efficient way to do this?
Using following equation:

$$\begin{pmatrix} F_n \\ F_{n+1} \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}^n \times \begin{pmatrix} F_0 \\ F_1 \end{pmatrix}$$

Since $F_n(mod(p))$ can be obtained by taking the first term of the matrix $\begin{pmatrix} F_n \\ F_{n+1} \end{pmatrix}(mod(p))$. As the matrix $\begin{pmatrix} F_0 \\ F_1 \end{pmatrix}$ is a constatnt matrix, computing $\begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}^n (mod(p))$ is sufficient to compute $F_n(mod(p))$. Let matrix $\begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}$ be X. The modular exponentiation algorithm can be extended to solve this problem once following theorem can be proved:
Theorem: Given a 2-D matrix X, $(X\ mod\ p) \times (X\ mod\ p) = X^2\ mod\ p$
Proof: Let $X = \begin{pmatrix} a & b \\ c & d \end{pmatrix}$, then

$$X\ mod\ p = \begin{pmatrix} a + k_1 p & b + k_2 p \\ c + k_3 p & d + k_4 p \end{pmatrix}$$

Multiplying $(X\ mod\ p) \times (X\ mod\ p)$, the first term in the final matrix can be written as $b^2 + ac + pC$ where C is any constant which is equivalent to the first term of the matrix $X^2\ mod\ p$. Same for other terms. Hence, $(X^n\ mod\ p) \times (X\ mod\ p) = X^{n+1}\ mod\ p$ can be established as well. The extended modular exponentiation algorithm is given by:

function Modified-modexp(X, p, n)
Input: 2×2 array X where each element is of n-bits, p (n bit) and integer exponent n

```
Output: $(X^n \; mod \; p)$
if n = 0: return $I_2$
z = Modified-modexp(X, p, $\lfloor \frac{n}{2} \rfloor$)
temp = $(z \; mod \; p) \times (z \; mod \; p)$
if y is even: return temp
else: return $(X \; mod \; p) \times temp$
```

Complexity Analysis: Number of recursive calls are analogous to Modular Exponentiation presented at page 19. Matrix multiplication is $O(n^2)$ (Chapter 3 presents a better bound). Hence the complexity of the algorithm is $O(n^3)$.

Problem 1.15: Determine necessary and sufficient conditions on x and c so that the following holds: for any a, b, if $ax \equiv bx$ mod c, then $a \equiv b$ mod c.

$ax \equiv bx$ mod $c \Rightarrow c$ must divide $(a-b)x$ and $a \equiv b$ mod $c \Rightarrow c$ must divide $(a-b)$. Hence, necessary and sufficient conditions on x and c is $gcd(c,x) = 1$.

Problem 1.16: The algorithm for computing a^b 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.

Refer solution to problems 1.11, 1.12 & 1.13.

Problem 1.17 Consider the problem of computing x^y for given integers x and y: we want the whole answer, not modulo a third integer. We know two algorithms for doing this: the iterative algorithm which performs y - 1 multiplications by x; and the recursive algorithm based on the binary expansion of y.

Compare the time requirements of these two algorithms, assuming that the time to multiply an n-bit number by an m-bit number is $O(mn)$.

x and y are each n-bit long. We are performing complexity analysis of 2 algorithms for ^{y} computation.

```
function Iterative-Exponentiation(x,y)
Input: x and y each n-bit long
Output: $x^y$
prod = x
for i = 1 to y - 1
    prod = x × prod
return prod
```

Complexity Analysis: After each multiplication, the size of product becomes $i \times n$ where i is the current iteration. Total time is given by $\sum_{i=0}^{y-1} O(in \cdot n) = \frac{(y-1)y}{2} O(n^2)$. The complexity is $O(y^2 n^2)$ where y is $O(2^n)$.

```
function Recursive-Exponentiation(x,y)
Input: x and y each n-bit long
Output: $x^y$
if y == 0: return 1
z = Recursive-Exponentiation(x, $\lfloor \frac{y}{2} \rfloor$)
if y is even: return z×z
else: return x×z×z
```

Complexity Analysis: Since y is n-bit long, the number of iterations is bounded by $O(n)$. Size of z on each return from recursive call increases by a factor of 4.Total running time is given by

$$O(n^2) + O(4n^2) + \cdots + O((2^{n-2}n)^2) = O(n^2 2^{2n})$$

Problem 1.31: Consider the problem of computing $N! = 1 \cdot 2 \cdot 3 \cdots N$.

(a) If N is an n-bit number, how many bits long is $N!$, approximately (in $\Theta(\cdot)$ form)?

Upper Bound: Assuming each number 1,2,3,...,N is represented by n bits, the result of multiplying N n-bit number will give a number of $N \times n$ bits where $N = 2^n$. Hence it in $O(N \times n)$.

Lower Bound: Since each number $i \in (1, 2, 3, ..., N)$ can be optimally represented by log i bits, total number of bits in N! is given by $\sum_{i=1}^{N} log(i)$ which is $log(N!)$. Using Sterling's approximation or using factor argument, $N! \geq (N/2)^{N/2}$ which implies that total number of bits in N! is lower bound by $Nlog(N)$. It turns out to be $\Omega(N \times n)$.

Combining bpth, we get $\Theta(N \times n)$.

(b) Give an algorithm to compute N! and analyze its running time.

A simple iterative algorithm to solve the problem is given by:

```
Input : N
Output : N!
prod = 1
for i = 1 to N
    prod = prod × i
return prod
```

Complexity analysis:We present a worst case bound. Assuming each of the number 1,2,3,...,N are n-bit long each multiplication computes product of a $i \times n$ bit number with n bit number. Therefore total time taken by the for-loop is given by $\sum_{i=1}^{N}(n \times in)$ which turns out to be $O(N^2 \times n^2)$.

Problem 1.33: Give an efficient algorithm to compute the least common multiple of two n-bit numbers x and y, that is, the smallest number divisible by both x and y. What is the running time of your algorithm as a function of n?

Given numbers X and Y, apply Euclid algorithm to compute GCD(X,Y). Followed by that get LCM(X,Y) by $\frac{X \times Y}{GCD(X,Y)}$. GCD computation takes $O(n^3)$ where n is number of bits in X, Y. Final LCM computation takes $O(n^2)$. Therefore the algorithm is bounded by $O(n^3)$.