Arithmetic Problems on the BIT model

Disclaimer: *These abbreviated notes DO NOT substitute the textbook for this class. They should be used IN CONJUNCTION with the textbook and the material presented in class. If there is a discrepancy between these notes and the textbook, ALWAYS consider the textbook to be correct. Report such a discrepancy to the instructor so that he resolves it. These notes are only distributed to the students taking this class with A. Gerbessiotis in Spring 2012 ; distribution outside this group of students is NOT allowed.*

Integer Multiplication in the BIT model

The arithmetic model that we adapt for the remainder of this discussion on integer multiplication is the bit model. An operation involving two integers costs a number of bit operations proportional to the number of bits of the two integers. The prefix "bit" in "bit operations" will be omitted from now on and so, a term "number of operations" would mean "number of bit operations". A "number" would also loosely mean "(non-negative) integer number" unless otherwise explicitly specified. If the input consists of two numbers x, y one may assume that $x > y$ in general.

Note on number of bits for integer n. The (minimal) number of bits required to represent (natural) integer n is $\lceil \lg (n+1) \rceil$. With this number of bits one can also represent all intermediate (non-negative) integers from 0 through n. One can also write $\lceil \lg (n+1) \rceil$ as $\lfloor \lg n \rfloor + 1$.

Example. For $n = 1$, we have $\lceil \lg (1+1) \rceil = 1$ and thus with 1 bit we can represent also integers $0, \ldots, 1$. Similarly for $n = 7$ we have $\lceil \lg (7+1) \rceil = 3$ and we can represent all integers from $0, \ldots, 7$ with 3 bits. For $n = 16$, we have $\lceil \lg (16+1) \rceil = 5$, and we can represent integers $0, \ldots, 16$ but also $17, \ldots, 31$ with 5 bits. Don't try to figure out what happens with $n = 0$. It's not a natural integer!

We derive the first elementary results. We give two formulations.

Addition/Subtraction: Formulation 1. The addition or subtraction of two n-bit numbers requires $O(n)$ bit operations by the trivial algorithm.

Addition/Subtraction: Formulation 2. The addition/subtraction of a and b requires $O(\lg a + \lg b)$ bit operations.

Addition/Subtraction: Formulation 3 for $a > b$. The addition/subtraction of a and b requires $O(\lg a)$ bit operations.

Multiplication. The naive multiplication of a and b such that $c = ab$ requires $O(\lg a \lg b)$ bit operations. The bit representation of c requires $O(\lg a + \lg b)$ bits.

Corollary Two n-bit integers can be multiplied in $O(n^2)$ bits operations.

Multiplication by 2^n The multiplication of a by 2^n requires $\lg a + n$ operations.

Writing down a requires $\lg a$ bits and multiplication by 2^n is a left-shift operation of n bit positions.

Integer Division by 2^n The integer division of a by 2^n requires $\lg a + n$ operations.

Integer division by 2^n is equivalent to a right-shift operation.

(c) Copyright A. Gerbessiotis. CS 667-101 : Spring 2012 . All rights reserved.

Things to remember.

Remark 1. Most of the time we are going to assume but not explicitly state that we can allocate enough space to store the output i.e. we can allocate space for z whose length is $2 * n$ bits.

Remark 2. In case x or y contains fewer bits than the other we pad the number with fewer bits so that both numbers become have the same length.

Remark 3. For number 10 in binary, i.e. 1010, its most significant bit is 1, its third most significant bit (abbreviated as msb is also the second least significant bit abbreviated as lsb) is also 1, and its least significant bit is 0, and in general for an n-bit number we can represent it in binary in various ways $x_1 \dots x_n$ (Pascal-like and textbook-like) or $x_0 \dots x_{n-1}$ (C/C++-like) or $x_n \dots x_1$ or $x_{n-1} \dots x_0$. Sometimes one or the other representation will be used depending on the circumstances.

(c) Copyright A. Gerbessiotis. All rights reserved.

Karatsuba-Ofman algorithm. The Karatsuba-Ofman algorithm for multiplying two n bit numbers requires $O(n^{\lg 3})$ bit operations.

Proof. In order to simplify exposition, without loss of generality, let n be an even power of two. Let a, b be the two n-bit numbers. The n bit number a consists of $n/2$ least significant bits and $n/2$ most significant bits respectively. Let a_l, a_m be the integer representation of these $n/2$ bits. Then $a = a_m 2^{n/2} + a_l$ and similarly $b = b_m 2^{n/2} + b_l$ Then

$$ab = (a_m 2^{n/2} + a_l)(b_m 2^{n/2} + b_l) = a_m b_m 2^n + (a_m b_l + a_l b_m)2^{n/2} + a_l b_l = a_m b_m 2^n + ((a_m + a_l)(b_m + b_l) - a_l b_l - a_m b_m)2^{n/2} + a_l b_l$$

```
   KaratsubaOfman(a,b)
1. (a_m a_l) =a;
2. (b_m b_l) =b;
3. t1 = KaratsubaOfman((a_m + a_l ),(b_m + b_l));
4. t2 = KaratsubaOfman(a_m , b_m);
5. t3 = KaratsubaOfman(a_l , b_l);
6. t4 = t1 - t2 -t3;
7. t2 = t2 << n; // t2 =t2 * 2**n
8. t4 = t4 << (n/2); // t4 = t4 * 2**(n/2)
6. return ( t2 + t4 + t3);
```

Therefore in order two multiply two n bit numbers a and b we group these two n-bit numbers into groups of $n/2$ bits and thus get a_m, b_m, a_l, b_l. We then perform three multiplications, one is $a_l b_l$ (line 5), the other $a_m b_m$ (line 4) and the third is the $((a_m + a_l)((b_m + b_l)$ (line 3) of $n/2$-bit integers. In addition, the latter operation requires 2 additions of $n/2$ bit numbers in $O(n)$ time and to get ab we need to perform two additional addition and two subtractions for a total of $O(n)$ as well plus two shift operations. Multiplication by 2^n or $2^{n/2}$ is a shift left operation requiring $O(n)$ operations. Therefore the running time $M(n)$ to multiply a and b is

$$M(n) = 3M(n/2) + \Theta(n)$$

where $3M(n/2)$ is due to the three multiplications of $n/2$ bit numbers and $O(n)$ includes additions and shifts. A solution of the recurrence by the master method yields $M(n) = \Theta(n^{\lg 3}) = \Theta(n^{1.58496})$..

Karatsuba-Ofman Multiplication: A missing step

Detail. $(a_m + a_l)$ and $(b_m + b_l)$ may not be $n/2$ bit integers any more when step 3 is performed. In fact they may be $n/2 + 1$-bit integers. How one can fix this "tiny" problem?

This is easily fixable as follows. Write $a_m + a_l$ as follows. $a_m + a_l = a_h 2^{n/2} + a_1$ (similarly for $b_m + b_l$). Note that $a_m + a_l$ can be represented by up to $n/2 + 1$ bits: a_h is either a 0 or 1 i.e. a bit. Then the product $(a_m + a_l)(b_m + b_l) = a_h b_h 2^n + (a_h b_1 + b_h a_1)2^{n/2} + b_1 a_1$. The product $a_h b_h$ is easy to find as it involves two 1-bit numbers. So are the products $a_h b_1$ (the product is either b_1 or 0 depending on whether a_h is 1 or 0) and $b_h a_1$ (either a_1 or 0). The only complication arises from $a_1 b_1$: both numbers are $n/2$ bit integers and the product is a KaratsubaOfman multiplication of two $n/2$-bit integers. The fix thus consists of one $n/2$-bit multiplication plus $O(n)$ bit operations.

An easier way to resolve this problem is by rewriting the product as follows.

$$ab = (a_m 2^{n/2} + a_l)(b_m 2^{n/2} + b_l) = a_m b_m 2^n + (a_m b_l + a_l b_m)2^{n/2} + a_l b_l = a_m b_m 2^n + ((a_m - a_l)(b_l - b_m) + a_l b_l + a_m b_m)2^{n/2} + a_l b_l$$

(c) Copyright A. Gerbessiotis. All rights reserved.

Let $X = 1010$ be an integer in binary. Then $X = 10$ in decimal. We say that the precision of X is 4 as 4 is the minimum number of bits needed to represent X. Note also that 00001010 is also 10 in binary. Even if 8 bits are used the preceding zeroes are redundant. If we shift X 4 positions to the right we obtain $x = .1010$. In decimal this number is $1/2 + 1/8 = 5/8$. The 1 on the right of the decimal point guarantees that $1/2 \leq x \leq 1$. The precision of .1010 is 3 (the decimal does not count nor the leading zeroes on the left of the decimal point nor the trailing zeroes after the rightmost one) and we often write instead .101. The precision of 1.101 is of course 4.

Note. On the left of the decimal point the zeroes preceding the leftmost 1 are ignored. On the right of the decimal point (if any) the zeroes trailing the rightmost 1 are also ignored.

Therefore for an integer $X \neq 0$ with precision m, its binary representation requires m bits and the leftmost bit is 1. Let $X = x_1 \dots x_m$ in binary. Then, $x_1 = 1$. If we shift X right m positions we get a fractional number in binary which is $x = .x_1 \dots x_m$, whose digit on the right of the decimal point . is always 1. Such number x whose $x_1 = 1$ is also called a normalized fractional number.

Thm 1. Every precision m integer $X \geq 1$ can be written as the product of a power of 2 and a normalized fractional number x, where $1/2 \leq x \leq 1$.

Proof. $X = 2^m x$, where $x = .x_1 \dots x_m$ with $x_1 = 1$, and the theorem follows.

Corollary 1. Theorem 1 applies even if X is not integer provided that $X \geq 1$. **Question.** What if we have a number (decimal or fractional) that is not normalized, i.e. it does not have one on the right of the decimal point?

If the number is ≥ 1, we can shift it right until it becomes less than 1 and Theorem 1, guarantees it will be normalized (as the number ≥ 1 has an 1 on the left of the decimal point); this also proves Corollary 1.

If the number is < 1 and is of precision n and it is not zero, it must have an 1 on the right of the decimal . which allows the shift of the number left or right so that it becomes normalized. A shift of at most n is required to achieve this.

For example, $100.01 = 2^3 \cdot 0.10001$ and $.01 = 2^{-1} \cdot .1$.

We can thus generalize Theorem 1 to include all possible cases of positive numbers, integer or not.

Thm 2. Every precision m number X can be written in the form $X = 2^k \times .1b_2 \dots b_m$, where $k \leq m$.

Corollary 2. From Thm2 we obtain that $b = .1b_2 \dots b_m$ is such that $1/2 \leq b \leq 1$. This also implies that $1 \leq 1/b \leq 2$.

Thm 3. Let X be an n precision binary integer. Then $x = X2^{-m}$, where $m < n$ is such that $2^{n-m-1} \leq x \leq 2^{n-m}$.

Proof. If $X = x_1 \dots x_n$, then $x = X2^{-m} = x_1 \dots x_{n-m}.x_{n-m+1} \dots x_n$. The smaller value for x is $10 \dots 0.0 \dots 0$ and the largest possible is $11 \dots 1.1 \dots 1$. The former is 2^{n-m-1} and the latter is at most 2^{n-m}.

For example let $X = 10001 = 17$. Then $x = X2^{-3} = 10.001 = 2.125$. It is obvious that $10.000 \leq x \leq 11.111 \dots 1$ i.e. $2 \leq x \leq 3.99999 \dots 9$, i.e. $2 \leq x \leq 4$.

(c) Copyright A. Gerbessiotis. All rights reserved.

Bisection/Binary search

Suppose we have a polynomial f and we want to find the roots of $f(x) = 0$. One way to find a root of f is iterative. First establish two points a, b such that $f(a)f(b) < 0$. This would mean that a solution to $f(x) = 0$ is in the interval $[a, b]$. Then a binary search is performed by inspecting the intervals $[a, c]$ and $[c, b]$ with $c = (a+b)/2$ to further refine the interval containing a solution. The two new intervals are half the size of the original. After m iterations the interval containing a solution is $1/2^m$ the size of the original $[a, b]$ interval. For a predetermined precision ϵ the method stops when $2^{-m}(b - a) < \epsilon$.

A better and faster approximation method is Newton-Raphson approximation. Let r be a root and let the current approximation to the solution be x_0 such that $r = x_0 + \delta$, i.e. we are δ away from the root. Then $f(r) = f(x_0 + \delta) = 0$. By taking a Taylor expansion of $f(x_0 + \delta)$ we have that

$$0 = f(x_0 + \delta) = f(x_0) + \delta f'(x_0) + \delta^2 \Delta$$

where Δ is a tiny quantity. From this approximation we conclude that

$$0 \approx f(x_0) + \delta f'(x_0)$$

i.e. $\delta = -f(x_0)/f'(x_0)$.

In Newton-Raphson we start with an arbitrary approximation of the root r, say x_0. Then, the next approximation becomes $x_1 = x_0 - f(x_0)/f'(x_0)$, and then a refinement of $x_2 = x_1 - f(x_1)/f'(x_1)$ is used , and this is repeated until the obtained approximation x_i is satisfactorily close to r.

(c) Copyright A. Gerbessiotis. All rights reserved.

Let y be a real number. We would use N-R to find $x = 1/y$. Let us consider $f(x) = y - 1/x$. The solution of $f(x) = 0$ for x is $t - 1/x = 0$ therefore $x = 1/y$ is the inverse of y.

Observation. The inverse of y is the solution of $f(x) = 0$.

Using the Newton-Raphson method, we get that $f'(x) = 1/x^2$. If the first approximation to $1/y$ is x_0 (we will discuss how to choose such a x_0 later). Then $x_1 = x_0 - f(x_0)/f'(x_0) = x_0 - (y - 1/x_0)/(1/x_0^2)$ and then we obtain that $x_1 = 2x_0 - yx_0^2$. Therefore,

Lemma APPROX. If x_0 is an initial approximation to $1/y$, then the sequence of Newton-Raphson approximations is given by : $x_i = 2x_{i-1} - yx_{i-1}^2$.

Lemma ERROR. If x_0 is an initial approximation to $1/y$, then in the i-th Newton-Raphson iteration the error is if $y \leq 1$.

$$1/y - x_i \;\; \leq \;\; y(1/y - x_0)^{2^i}. \tag{1}$$

Proof of Lemma Error. Consider first the error of approximating $1/y$ by x_1. We obtain the following error bound.

$$\begin{aligned} 1/y - x_1 &= 1/y - 2x_0 + yx_0^2 \\ &= y(1/y^2 - 2x_0(1/y) + x_0^2) \\ &= y(1/y - x_0)^2 \end{aligned} \tag{2}$$

Induction completes the rest of the proof. □

If $y \leq 1$ and $\mid 1/y - x_0 \mid \leq 1/2$, then after i iterations the error of approximating $1/y$ by x_i is at most $1/2^{2^i}$. For $i = \lg n$, the error is less than $1/2^n$.

Lemma NR. If $Y \leq 1$, then an approximation to $1/Y$ can be computed by Equation (1) with $Y = y$. After $\lg n$ iterations, the error of this approximation is at most $1/2^n$, i.e. the value returned $x_{\lg n}$ is such that $\mid 1/Y - x_{\lg n} \mid \leq 1/2^n$.

(c) Copyright A. Gerbessiotis. All rights reserved.

Towards this, we have a short excursion on error bounds.

We know that $1/3 = 0.33333\ldots$. If we approximate $1/3$ by 0.3 then the error is $0.0333\ldots$ or less than $4\ 10^{-2}$, if we approximate $1/3$ by 0.33 then the error is $0.00333\ldots$, or less than $4\ 10^{-3}$ and so on. For binary numbers, the representation of integers as binary numbers is quite established. If fractional numbers are represented as binary, 0.11 represents in binary $2^{-1} + 2^{-2} = 3/4$ and $0.111 = 3/4 + 2^{-3} = 3/4 + 1/8 = 7/8$. Therefore approximating 0.111 by 0.11 introduces an error of 2^{-3} (two to the power of the index of the eliminated bit, index 1 starting immediately to the right of the period .). As a conclusion, if the error is less than 2^{-m}, this means that the approximation is accurate to $m-1$ significant digits.

Conclusion. $\lg n$ iterations of Newton-Raphson provide $n-1$ digits to the reciprocal approximation.

Initial Setup of Newton-Raphson. Although g was assumed to be a real number for Lemma NR to work properly y must be no more than 1, i.e. it's not an arbitrary real number.

Let Y be a real number represented in binary as $Y = 2^m \times 0.\epsilon_1\epsilon_2\ldots = 2^m \cdot 0.y$, with $.y = .\epsilon_1\epsilon_2\ldots$, and $\epsilon_1 = 1$. Note that if $\epsilon_1 \neq 1$, we can adjust (decrease) m to achieve this. For example consider $a = 0.25$ which is $a = 0.01$ in binary. Since $\epsilon_1 \neq 1$, shifting Y left one position we obtain $0.1 \times 2^{-1} = 2^{-1}.1$ which is still $(1/2)(2^{-1}) = 1/4 = 0.25$. Because $\epsilon_1 = 1$, we have that $1/2 \leq y \leq 1$. Then $Y^{-1} = 2^{-m}y^{-1}$, where $1 \leq y^{-1} \leq 2$.

Lemma Reciprocal. For an arbitrary real number Y, in order to find an approximation to $1/Y$, we normalize $Y = 2^m \cdot y$, and then find an approximation x' to $1/y$ using Equation (1). An approximation X to $1/G$ is then $X = 2^{-m}x'$.

The number of bit iterations in Newton-Raphson to achieve the desired degree of precision can be further reduced. More details are available in the Appendix, where reciprocal finding become a component of integer division.

Karatsuba-Ofman algorithm. The Karatsuba-Ofman algorithm for multiplying two n bit numbers requires $O(n^{\lg 3})$ bit operations.

1. Power. Given x and n, x^n requires $O(\lg n)$ multiplications each involving $O(n \lg x)$-bit integers.

2. Integer Multiplication is no harder than squaring. If $M(n)$ is the time to multiply two n-bit numbers and $S(n)$ is the time two square an n-bit number (given a, find a^2) then $M(n) = O(S(n) + n)$.

Proof. Consider $ab = ((a+b)^2 - (a-b)^2)/4$.

3. Squaring is no harder than reciprocal finding. If $S(n)$ is the time to square an n-bit number and $R(n)$ is the time to find the reciprocal of an n-bit number (i.e for input a find $1/a$) then $S(n) = O(R(n) + n)$.

Proof. Left as an exercise.

4. Division is no harder than reciprocal finding. $x/y = x \times (1/y)$.

Proof. Use Newton-Raphson.

5. Remainder. $a \bmod b$ requires $O(\lg q \lg b)$ bit operations where $q = \lfloor a/b \rfloor$.

Proof. Since $a \bmod b = a - q * b$, the multiplication of q and b requires $O(\lg q \lg b)$ time and the subtraction $O(\lg q + \lg b)$ time which is also $O(\lg q \lg b)$.

6. Reciprocal. Reciprocal finding is no more complicated than multiplication and subtraction/division.

Proof. Use Newton-Raphson.

(c) Copyright A. Gerbessiotis. All rights reserved.

Fact 0. A positive number a can be represented with $\lfloor \lg a \rfloor + 1 = \lceil \lg (a+1) \rceil$ bits. For example 3, requires 2 bits, 15 requires 4 bits, but 16 requires 5 bits for its representation.

Fact 1. If a and b are positive integers, the division of a by b creates a quotient q and remainder r such that $a = qb + r$, where $0 \leq r < b$, and $q \leq a$. Since a is $\lg a$ bits long, the quotient q of a divided by b is at most $\lg a$ bits long as well. Since $r < b$, r is $\lg b$ bits long.

Fact 2. If $r = 0$, we say b divides a and write b/a and also call b a divisor of a, and a a multiple of b. If the only divisors of a are 1 and a we call a a prime, else we call a a composite. 1 and a are **trivial divisors** of a.

Fact 3. Factorization is the operation that finds all prime divisors of a. In general a can be written as $a = p_1^{b_1} \dots p_n^{b_n}$. The number of divisors of a is represented by $d(a)$. It is easy to find that $d(a) = (1 + b_1) \dots (1 + b_n)$.

Fact 3.a If d/a and d/b, then $d/a \pm b$, and more generally $d/ax + by$ for any integers x, y. If a/b and b/a then $a = \pm b$.

Fact 4. The greatest common divisor of a and b, is an integer m that divides both a and b and every other common divisor of a and b divides d. We represent $d = (a, b)$ or $d = gcd(a, b)$.

Fact 4.a The following properties hold for gcd. $gcd(a, b) = gcd(b, a) = gcd(-a, b) = gcd(|a|, |b|)$, and $gcd(a, 0) = |a|$, $gcd(a, ka) = |a|$.

Fact 4.b If a, b are integers such that $|a|+|b| \neq 0$, then $gcd(a, b)$ is the smallest positive integer of the set $\{ax+by : x, y \in Z\}$.

Fact 4.c For any a, b if d/a and d/b, then $d/gcd(a, b)$.

Fact 4.d $gcd(an, bn) = n \cdot gcd(a, b)$.

Fact 4.e If n/ab and $gcd(a, n) = 1$, then n/b.

Fact 4.f If $gcd(a, p) = 1$ and $gcd(b, p) = 1$, then $gcd(ab, p) = 1$.

Fact 5. If $gcd(a, b) = 1$ we call a and b relatively prime.

(c) Copyright A. Gerbessiotis. All rights reserved.

Fact 6. If $a > 0$, then $\phi(a)$ denotes the number of positive integers b such that $(a, b) = 1$ and $b \leq a$. For example $\phi(4) = 2$ (1, 3 are relatively prime to 4). For the a above, let us filter out all the numbers that are not relatively prime to a. These include the a/p_1 multiples of p_1 less than a, the a/p_2 multiples of p_2 less than a, and so on, for a total of $a - a/p_1 - \ldots - a/p_n$. A multiple of both p_1 and p_2 was counted twice. Therefore we need to add terms of the form $n/(p_1p_2) + \ldots + n/(p_1p_t)$. But a multiple of $p_1p_2p_3$ has been reinserted at least twice. We correct again, and finally by alternating between subtractions and additions we get a term that is $\phi(a) = a(1 - 1/p_1) \ldots (1 - 1/p_n)$.

Fact 7. Suppose the division of a by b yields $a = qb + r$. The integers can then be divided into b equivalence classes depending on the remainder of their division by b. The **equivalence class modulo b** containing an integer a is

$$[a]_b = \{a + kb : k \in Z\}$$

contains all the integers whose division by b yield the same remainder as a. For two integers a, c we can write $c \in [a]_n$ by writing equivalently that $a \equiv c \pmod{b}$. The set consisting of all equivalent classes modulo b is denoted $Z_b = \{[a]_b : 0 \leq a \leq b - 1\}$. Most of the times from such a representation we drop the brackets and denote the elements of Z_b as $Z_b = \{0, 1, \ldots, b - 1\}$.

Example. Let $b = 3$. Then any integer n belongs to one of three classes (sets) $[0]_3, [1]_3, [2]_3$ depending on what the remainder of the division of n by 3 is. Integers 0, 3, 6, 9, etc belong to the same set $[0]_3$. When we add elements of Z_3 we must never forget that additions is performed modulo 3, i.e. $[1]_3 + [2]_3 = [3]_3 = [0]_3$. Of course we simplify notation to write $1 + 2 = 3 = 0$ implying modulo 3 addition.

(c) Copyright A. Gerbessiotis. All rights reserved.

Euclid's Algorithm

Theorem. For any $a \geq 0, b > 0$, we have that $gcd(a, b) = gcd(b, a \bmod b)$. Euclid's algorithm (circa 300BC) finds the gcd of a and b as follows recursively.

```
   Euclid(a,b)
1. if (b==0) return(a);
2.  else return(Euclid(b,a mod b));
```

Lemma. The algorithm eventually terminates.

Proof. Let $a_0 = a$ and $a_1 = b$, and $a_2 = a \bmod b = a_0 \bmod a_1$. It is obvious that $a_2 < a_1$. Then $gcd(a_0, a_1) = gcd(a_1, a_2)$. Continuing the same way and defining $a_i = a_{i-2} \bmod a_{i-1}$ with $a_i < a_{i-1}$, we obtain that

$$gcd(a_0, a_1) = gcd(a_1, a_2) = gcd(a_2, a_3) = \ldots = gcd(a_{i-2}, a_{i-1}) = gcd(a_{i-1}, a_i) = \ldots = gcd(a_n, 0) = a_n.$$

Because $a_i < a_{i-1}$ we have that $a_i < a_{i-1} < a_{i-2} < \ldots < a_2 < a_1$. Given that a_1 is finite, and the sequence of the a_i's is strictly decreasing, there will be an $a_{n+1} = 0$ after at most $a_1 = b$ steps.

$$\begin{aligned} a_0 &= a_1 q_1 + a_2 \\ a_1 &= a_2 q_2 + a_3 \\ a_2 &= a_3 q_3 + a_4 \\ \ldots & \\ a_{n-2} &= a_{n-1} q_{n-1} + a_n \\ a_{n-1} &= a_n q_n \end{aligned} \tag{3}$$

Give eventuall $a_{n+1} = 0$, we have that $gcd(a, b) = gcd(a_{n-1}, a_n) = gcd(a_n, 0) = a_n$. □

Remark. We also have that $q_i = \lfloor a_{i-1}/a_i \rfloor$, i.e. $q_i \leq a_{i-1}$. In addition $q_n \geq 2$, since otherwise $q_n = 1$, $a_{n-1} = a_n$, and then the remainder of the division of a_{n-2} by a_{n-1} is $a_n = a_{n-1}$ which contradicts the fundamental $a_n < a_{n-1}$ fact.

(c) Copyright A. Gerbessiotis. All rights reserved.

The number of steps (divisions)

In general we may assume **without loss of generality (w.l.o.g.)** that $a > b$, since otherwise we can exchange the a and b and the gcd remains the same since $gcd(a, b) = gcd(b, a)$.

Let (see page 56) F_n be the n-th Fibonacci number, i.e. $F_n = F_{n-1} + F_{n-2}$, $n > 1$, where $F_0 = 0$ and $F_1 = 1$.

It is known that $F_{n+1} \approx \phi^{n+1}/\sqrt{5}$ where $\phi = (1 + \sqrt{5})/2$ is the golden ratio. Therefore if $b = F_n$, then $n = O(\lg b)$ as well.

Lemma 1. In Euclid's algorithm for $a > b \geq 1$, we have that $a \geq F_{n+2}$ and $b \geq F_{n+1}$, where n is the number of division steps of the algorithm.

Proof. The proof is by induction.

For $n = 1$, we show that in an one-step division algorithm it is true that $a \geq F_3$ and $b \geq F_2$. Note that $F_2 = F_1 + F_0 = 1$ and $F_3 = F_2 + F_1 = 2$. Since $b \geq 1$ we have that $b \geq F_2 = 1$. Since $a > b$, we have that $a \geq 2 = F_3$. This proves the base case of the induction.

We then prove the inductive step, i.e. that if an i division step GCD computation of A, B, requires $A \geq F_{i+2}$ and $B \geq F_{i+1}$, for any $i = 1, \ldots, n-1$, then an n division step GCD computation of a, b requires $a \geq F_{n+2}$ and $b \geq F_{n+1}$.

An n step division consists of the first step and the $n-1$ remaining steps. The first step of an n step division algorithm is say $a_0 = a = bq_1 + a_2 = a_1q_1 + a_2$ Subsequently the gcd of a_1 and a_2 is computed in the remaining $n-1$ steps. The inductive hypothesis for $i = n-1$, would yield $b = a_1 \geq F_{n+1}$ and $a_2 \geq F_n$.

Since $q_1 = \lfloor a/b \rfloor$ and $a > b$ we have that $q_1 \geq 1$ and thus $a_0 = a_1q_1 + a_2 \geq a_1 \cdot 1 + a_2 \geq F_{n+1} + F_n = F_{n+2}$. Therefore, $a = a_0 \geq F_{n+2}$ and we also have by the inductive hypothesis that $b = a_1 \geq F_{n+1}$ as required. This proves the inductive step.□.

Corollary. If $a > b$ and $b < F_{k+1}$, then the GCD algorithm consists of less than k division steps.

Proof. Suppose that the GCD algorithm would consist of at least k steps, and say $j \geq k$ is the number of steps. Then by Lemma 1, $b \geq F_{k+1}$. This contradicts the fact that $b < F_{k+1}$.

Corollary 2. If $b < F_{k+1}$, then the number of division steps is $\Theta(\lg b)$.

Proof. This is a consequence of $F_{n+1} \approx \phi^{n+1}/\sqrt{5}$.

The runing time (bit operations)

Corollary 3. The bit complexity of GCD is $O(\lg b M(\lg a, \lg a))$ which is also $O(\lg a M(\lg a, \lg a))$.

Proof. The number of steps if $\Theta(\lg b)$ and each a_i, q_i is no more than a, i.e. all intermediate results are $O(\lg a)$ bits long and of course $\lg b \leq \lg a$. Division and/or Multiplication requires $O(M(n, n))$ bit operations for n bit numbers.

Corollary 4. In the i-th division step $a_{i-1} = a_i q_i + a_{i+1}$, The multiplication involved requires $\lg a_i \lg q_i \leq \lg b \lg q_i \leq \lg b(\lg a_{i-1} - \lg a_i)$ bit operations, since $q_i \leq a_{i-1}/a_i$.

Corollary 5. The bit complexity of GCD can be improved to $O(\lg a \lg b)$.

Proof. Let $M(p, q) = pq$ (i.e. ordinary multiplication). Use the observation of Corollary 4 to reduce the running time of $O(\lg^3 a)$ of Corollary 3 to $O(\lg a \lg b)$. (You also need a proof that you can find q in $O(\lg a \lg b)$.)

(c) Copyright A. Gerbessiotis. All rights reserved.

Extended form of Euclid's Algorithm

A variation of Euclid's algorithm in addition to $gcd(a, b)$ also computes the x, y such that $gcd(a, b) = ax + by$.

We can rewrite Eq. 3 in matrix form as follows.

$$\begin{bmatrix} a_0 \\ a_1 \end{bmatrix} = \begin{bmatrix} q_1 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} a_1 \\ a_2 \end{bmatrix}$$

and

$$\begin{bmatrix} a_1 \\ a_2 \end{bmatrix} = \begin{bmatrix} q_2 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} a_2 \\ a_3 \end{bmatrix}$$

all the way down to

$$\begin{bmatrix} a_{n-1} \\ a_n \end{bmatrix} = \begin{bmatrix} q_n & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} a_n \\ a_{n+1} \end{bmatrix}$$

where, however $a_{n+1} = 0$, and $a_n = gcd(a, b)$.

We next define matrix D_k such that

$$D_k = \begin{bmatrix} q_1 & 1 \\ 1 & 0 \end{bmatrix} \cdots \begin{bmatrix} q_k & 1 \\ 1 & 0 \end{bmatrix}$$

Then,

$$\begin{bmatrix} a \\ b \end{bmatrix} = D_n \begin{bmatrix} gcd(a, b) \\ 0 \end{bmatrix}$$

and equivalently

$$\begin{bmatrix} gcd(a, b) \\ 0 \end{bmatrix} = D_n^{-1} \begin{bmatrix} a \\ b \end{bmatrix}$$

(c) Copyright A. Gerbessiotis. All rights reserved.

Extended form of Euclid's Algorithm (continued)

If D_n is such that $$D_n = \begin{bmatrix} e & f \\ c & d \end{bmatrix}$$

then we can easily find D_n^{-1} from D_n as follows.

$$D_n^{-1} = \begin{bmatrix} x & y \\ z & w \end{bmatrix} = (-1)^n \begin{bmatrix} d & -f \\ -c & e \end{bmatrix}$$

then we have that $gcd(a,b) = ax + by$, where x and y is the first row of D_n^{-1}. The x, y can be expressed in term of the elements of D_n as follows: $x = (-1)^n d$ and $y = (-1)^{n+1} f$. The following pseudocode finds the x and y through D_n.

```
  ExtendedEuclid(a,b)
  // Computes gcd(a,b) and x,y : gcd(a,b)=ax+by.
1.  D = [1  0];
        [0  1]
2.  n = 1;
3.  while ( b != 0) {
4.   q = floor(a/b);  // q_i
5.          |q 1|
5.   D = D x|   | ;  // Find D_n
            |1 0|
6.   [a] = [b    ];  // Lines 6 and 7 are performed simultaneously...
7    [b] = [a-q*b];  // b now holds the remainder a2, a3, .... an
8.   n++;
    } // D_n found
9.  return(gcd=b,x= (-1)**n D[1,2], y=(-1)**(n+1) D[1,2] );
```

Theorem. The running time of ExtendedEuclid is $O(\lg a \lg b)$.

(c) Copyright A. Gerbessiotis. All rights reserved.

An alternative method

Suppose we want to find x, y such that $d = ax + by$. After one iteration of Euclid's algorithm it suffices to find x', y' of $gcd(b, a \bmod b) = d$. We note that

$$\begin{aligned} d &= bx' + (a \bmod b)y' \\ &= bx' + (a - \lfloor (a/b) \rfloor b)y' \\ &= ay' + b(x' - \lfloor (a/b) \rfloor y') \end{aligned}$$

Therefore the following recursive pseudocode determines x, y.

```
  ExtendedEuclid(a,b)
1. if (b==0)
2.  return (a,1,0);
3. (d',x',y' ) = ExtendedEuclid(b, a mod b);
4. (d, x ,y  ) = (d', y', x' - floor(a/b)* y' );
5. return(d,x,y);
```

```
                      (17,5)=
  17 = 5 x 3 + 2      (5, 2)=
   5 = 2 x 2 + 1      (2, 1)=
   2 = 1 x 2 + 0      (1, 0)=1

  (1,0) --> (1,1,0)
  (2,1) --> (1,0,1-2   x   0)=(1,0,1)
  (5,2) --> (1,1,0-2   x   1)=(1,1,-2)
  (17,5)--> (1,-2,1-3(-2))=(1,-2,7)
  Thus 1 = 17 (-2) + 5 (7)
```

(c) Copyright A. Gerbessiotis. All rights reserved.

Introduction

Modular arithmetic is usual arithmetic except that every result x we compute is replaced by $x \bmod N$. Sometimes we choose N artificially high so that its choice does not affect computations at all. For example consider the case where one wants to compute y^n and y is a $\lg b$-bit integer. Then, $y \leq 2^{\lg b} = b$, and therefore $y^n \leq b^n$. Therefore if we choose N to be number greater than b^n no computation will be affected by doing modular arithmetic since all intermediate results are less than N.

The existence of modular arithmetic makes sense in practice as in a computer a number in binary is represented by a limited number of bits (8, 16, 32, 64, or 128) which implies that all operations are modular arithmetic operations for an appropriate N.

For the set Z_n defined earlier, a representative of this set is denoted by a, where $0 \leq a \leq n-1$, and a represents all integers whose remainder if divided by n is a. When we add two elements of Z_n the operations we perform are $\pmod n$. It is easy to show that $+$ is well-defined in Z_n, and so is $-$. In addition, the identity element is 0 and every element a has an inverse that is called **additive inverse** $-a$, which is $[-a]_n$. For example in Z_9 -4 is $[5]_n$ usually written as 5 since $(-4)+4 = 5+4 \equiv 0 \bmod 9$.

A multiplicative inverse of a is represented by a^{-1} and is such that $aa^{-1} \equiv 1 \bmod n$. For example in Z_9 the inverse of 2 is 5 since $2 \cdot 5 \equiv 1 \bmod 9$. 0 does not have a multiplicative inverse. There are some elements of Z_9 other than 0 that also do not have an inverse. For example 3 does not have a multiplicative inverse either. If n is prime all elements of Z_n other than 0 have a multiplicative inverse. These elements then form **the multiplicative group modulo** n denoted by Z_n^* i.e.

$$Z_n^* = \{[a]_n \in Z_n : gcd(a, n) = 1\}.$$

The size of Z_n^* is $\phi(n)$. For a prime p, we have that $\phi(p) = p-1$. For $n = pq$, where p, q are primes $\phi(n) = (p-1)(q-1)$. For a subset of Z_n^* and operation $+$, if the operation is closed, commutative (i.e. $a+b = b+a$), associative ($a+(b+c) = (a+b)+c$), has identity 0, and every element has an additive inverse, then that subset is called a subgroup of Z_n^*. In that case the subgroup S is such that $|S|/|Z_n^*|$. If S is a proper subgroup, then $|S| \leq |Z_n^*|/2$.

(c) Copyright A. Gerbessiotis. All rights reserved.

Consider : $ax \equiv b \bmod n$

Theorem. Consider the set $< a >= \{ax \bmod n : x > 0\}$. If $b \in < a >$, then the equation above has a solution for x. Let $d = gcd(a, n)$. Consider set $D = \{0, d, 2d, \ldots, ((n/d) - 1)d\} =< d >$. Then, $D =< a >$.

Proof. Since $d = gcd(a, n)$, there exist x, y such that $ax + ny = d$. Therefore $ax \equiv d \bmod n$. Therefore $d \in < a >$. Then, every multiple of d like kd is such that $kd \equiv a(kd) \bmod n$, i.e. $kd \in < a >$. Therefore $D =< d > \subseteq < a >$.

If $m \in < a >$, then $m \equiv ax \bmod n$ and thus $m = ax + nw$. Since d is the smallest positive integer such that $d = ax + ny$, and since d/a and d/n, we have that d/m and thus m is a multiple of d i.e. $m \in < d >= D$. This means that $< a > \subseteq < d >= D$.

Therefore $D =< a >$. □

This leads to the following theorem.

Theorem 1. $ax \equiv b \bmod n$ has a solution for x if and only if $gcd(a, n)/b$.

Theorem 2. If $ax \equiv b \bmod n$ has a solution for x then it has $d = gcd(a, n)$ solutions. One solution is $x_0 = x'(b/d) \bmod n$, where $d = ax' + ny'$, and the other solutions are $x_0 + i(n/d)$ for $1 \leq i \leq d - 1$.

Remark. $ax \equiv b \bmod n$ implies $ax - b = kn$ or $ax + (-k)n = b$. Since $d = ax' + ny'$ multiplying both sides by (b/d) we have $b = a(b/d)x' + n(b/d)y'$. One x such that $ax - b = kn$ is the $x_0 = (b/d)x'$. For the other solutions (which are distinct) note that $a(x_0 + i(n/d)) = ax_0 + ian/d$. Since d divides both a and n, we have that (a/d) is an integer and an/d is multiple of n, i.e. $a(x_0 + in/d) \equiv b \bmod n$. This concludes the proof.

```
  SOLVE_AXB (a,b,n)
1. (d,x',y')= ExtendedEuclid(a,n);
2. if (d divides b) {
3.  x0 = x'  (b/d) mod n;
4.  for(i=0;i<d;i++) {
5.   print(" i-th  solution is ", x0+i(n/d) mod n);
6.  }
7. }
8. else error("No solution exists as b not multiple of d");
```

(c) Copyright A. Gerbessiotis. All rights reserved.

Introduction

In the introduction for modular arithmetic we mentioned the case of performing computations modulo a very big number N. Say N is such that $N = n_1 n_2 \dots n_k$, where the n_i's are pairwise prime. The Chinese Remainder Theorem (CRT for short) tells us that all computations modulo N (or of precision N) can be reduced to k simultaneous computations of lower precision $n = \max\{n_i\}$. This also means that instead of working with x modulo N we will be working with its representation $(x_1, \dots, x_k)$, where $x_i = x \bmod n_i$. Therefore CRT presents a mapping of Z_n to $Z_{n_1} \times \dots \times Z_{n_k}$.

Theorem CRT. Let $n = n_1 n_2 \dots n_k$, where the n_i's are pairwise relatively prime i.e. $(n_i, n_j) = 1$. Consider the correspondence $a \leftrightarrow (a_1, \dots, a_k)$ where $a \in Z_n$ and $a_i \in Z_{n_i}$, and $a_i = a \bmod n_i$. The mapping/correspondence is one-to-one. If in addition $b \leftrightarrow (b_1, \dots, b_k)$ then $(a + b) \bmod n \leftrightarrow (a_1 + b_1 \bmod n_1, \dots, a_k + b_k \bmod n_k)$, $(a - b) \bmod n \leftrightarrow (a_1 - b_1 \bmod n_1, \dots, a_k - b_k \bmod n_k)$, $(ab) \bmod n \leftrightarrow (a_1 b_1 \bmod n_1, \dots, a_k b_k \bmod n_k)$,

Proof.

Obtaining $(a_1, \dots, a_k)$ from a requires k divisions as $a = a_i \bmod n_i$.

Computing a from the a_i's is also easy in terms of computations involved; it requires however a series of steps. Let $m_i = n/n_i$. Then $m_i = 0 \bmod n_j$ for all $j \neq i$, and m_i, n_i are relatively prime. Then by letting,

$$c_i = m_i (m_i^{-1} \bmod n_i)$$

for $i = 1, \dots, k$, we obtain that

$$a \equiv (a_1 c_1 + \dots + a_k c_k) \pmod{n}$$

The existence of the multiplicative inverse of m_i follows from the fact that m_i and n_i are relatively prime. In addition c_j is a multiple of m_j, i.e. $c_j \equiv 0 \pmod{n_i}$, $j \neq i$. For $j = i$, however, we have that $c_j \equiv 1 \pmod{n_i}$, by construction of c_i. Therefore for c_j's we establish the correspondence $c_i \bmod n \leftrightarrow (0, \dots, 0, 1, 0, \dots, 0)$, where all entries are 0 except for the i-th entry that is 1. All the c_i's are then linearly independent which implies that $a \equiv a_i c_i \pmod{n_i} = a_i m_i (m_i^{-1} \pmod{n_i}) \equiv a_i \pmod{n_i}$.

Corollary. If n_i, $i = 1, \dots, k$ are pairwise relatively prime, and $n = n_1 \dots n_k$, then the system of equations $x \equiv a_i \pmod{n_i}$ has a unique solution for x.

(c) Copyright A. Gerbessiotis. All rights reserved.

An Example

Let $n = 210$. Then $n = 2 \cdot 3 \cdot 5 \cdot 7$.
Consider 182 and 8.
We then have

```
182                 8
182  % 2  = 0      8 % 2=0
182  % 3  = 2      8 % 3=2
182  % 5  = 2      8 % 5=3
182  % 7  = 0      8 % 7=1
```

Thus we can write $182 = (0, 2, 2, 0)$ and $8 = (0, 2, 3, 1)$. If we add 182 and 8 by adding pairwise their representations modulo the 4 prime factors of 210 we have $(0, 1, 0, 1)$. Then we can recover from the quadruplet the corresponding integer mod 210 as follows.

```
m1 = 210/2 =105       inverse(m1) % n1 = 1 x m1 = 105
m2 = 210/3 = 70       inverse(m2) % n2 = 1 x m2 =  70
m3 = 210/5 = 42       inverse(m3) % n3 = 3 x m3 = 126
m4 = 210/7 = 30       inverse(m4) % n4 = 4 x m4 = 120

Therefore
(0,1,0,1)= 0 x 105 + 1 x 70 + 0 x 126 + 1 x 120 = 190
    which is the sum of 182=(0,2,2,0) and 8=(0,2,3,1).
```

(c) Copyright A. Gerbessiotis. All rights reserved.

We are interesting in raising an element a of Z_n^* to a power k. The following theorem was proved by Euler.

Theorem Euler. For any integer $n > 1$, we have that

$$a^{\phi(n)} \equiv 1 \pmod{n}$$

for all $a \in Z_n^*$.

The special case of n being a prime was proved by Fermat. If $n = p$ is a prime $\phi(n) = n - 1 = p - 1$.

Theorem Fermat. If p is prime,

$$a^{p-1} \equiv 1 \pmod{p}$$

for all $a \in Z_p^*$.

It is also a consequence of the Theorem that $a^p \equiv a \pmod{p}$. An element g that generates all elements of Z_n^* through its powers g^i is called a primitive root or generator of Z_n^*, and Z_n^* is called cyclic.

Theorem Cyclic The values of n for which Z_n^* is cyclic are $2, 4, p^e, 2p^e$ for all odd primes p and all positive integers e.

The following is called the Discrete Logarithm Theorem.

Theorem DLT. If g is a primitive root of Z_n^* then the equation $g^x \equiv g^y \pmod{n}$ holds iff the equation $x \equiv y \pmod{\phi(n)}$ holds.

Theorem. If p is an odd prime and $e \geq 1$, then the equation $x^2 \equiv 1 \pmod{p^e}$, has only two solutions, namely $x = 1$ and $x = -1$.

Proof. Let $n = p^e$. Then Z_n^* has a primitive root g. Then x can be written $x = g^a \pmod{n}$ for some a, and the equation becomes $g^{2a} \equiv g^0 \pmod{n}$. Since Z_n^* is cyclic with period $\phi(n)$ we have that $2a \equiv 0 \pmod{\phi(n)}$. Let $d = gcd(2, \phi(n)) = gcd(2, (p-1)p^{e-1}) = 2$, and since $d/0$ we conclude that the equation has 2 solutions for a. By inspection, they are $+1$ and -1.

A non trivial square root of 1 modulo n is a number that is neither 1 nor -1.

(c) Copyright A. Gerbessiotis. All rights reserved.

The Algorithm

We present an algorithm for modular exponentiation, i.e. the problem of $a^b \bmod n$ by using repeated squaring using the binary representation of b.

```
  NaiveExponent (a,b,n)
1. result=1;
2. for(i=0;i<b;i++) {
3.  result = (result*a) mod n;
4. }
5. return(result);

  RepeatedSquaring(a,b,n)
1. c=0;
2. d=1;
3. (bk,bk-1,...,b1,b0) = BinaryRepresentation(b);
4. for(i=k;i>=0;i--) {
5.   c = 2c;
6.   d = (d d) mod n;
7.   if (bi == 1) {
8.    c++;
9.    d= (d a) mod n;
10.  }
11.}
12.return(d)
```

The total number of bit operation is $O(\lg^2 n)$, if all a, b, n are at most $O(\lg n)$ bits.

(c) Copyright A. Gerbessiotis. All rights reserved.

Appendix

Integer division (Newton-Raphson based)

Thm 1. Let X and Y be two precision n and m integers such that $n \geq m$. Then we can find integers Q and R such that $X = YQ + R$, $0 \leq R < Y$ in $O(M(n) \lg n)$ time (i.e. bit operations), where $M(k)$ is the time to multiply two k precision (i.e. bit) numbers.

Note. $M(k)$ counts bit operations.

Proof. Let $x = X \times 2^{-m}$ and $y = Y \times 2^{-m}$ and also let $r = R \times 2^{-m}$. Let $X = x_1 \dots x_n$ in binary. Then $x = x_1 \dots x_{n-m}.x_{n-m+1} \dots x_n$. Let $Y = y_1 \dots y_m$ in binary. Then $y = .y_1 \dots y_m$, with $y_1 = 1$.

The first observation is that since $X = YQ + R$, if we know Q we can find R by multiplying Y and Q and then subtracting YQ from X to yield R. This cost is also $O(M(n))$. Therefore, we will ignore the cost of finding R from the remainder of this discussion, and we will concentrate on finding Q only.

Then

$$\begin{aligned} X &= YQ + R \\ X2^{-m} &= (Y2^{-m})Q + R2^{-m} \\ x &= yQ + r \end{aligned}$$

Therefore the problem of finding the quotient of arbitrary numbers has been reduced to that of finding the quotient Q of x and y, where y is a normalized number such that $1/2 \leq y \leq 1$ (Theorem 2, Corollary 2, page 21).

Suppose we can find an approximation z to $1/y$ such that $|1/y - z| \leq 2^{-(n-m)}$. We shall discuss how to find such an approximation later using the Newton-Raphson method. The running time of the method for finding z will be proved to be $O(M(n) \lg n)$.

Note. This bound for the Newton-Raphson application to find z can be improved to $O(M(n))$. This discussion however will be deferred until the very end.

Consider the value $Q' = xz$. Given x and z, Q' can be computed in $O(M(n))$ time. $x = X2^{-m}$ and X is an n precision number. By an immediate application of Theorem 3 we obtain that $2^{n-m-1} \leq x \leq 2^{n-m}$. If we multiply both sides of $|1/y - z| \leq 2^{-(n-m)}$, which is equivalent to $-2^{-(n-m)} \leq 1/y - z \leq 2^{-(n-m)}$, with $2^{n-m-1} \leq x \leq 2^{n-m}$ we obtain that $|x(1/y) - xz| \leq 2$, i.e.. $|x/y - xz| \leq 2$, and finally $|x/y - Q'| = |Q - Q'| \leq 2$.

Q is an integer and Q' is at most 2 away from Q (or Q is also two away from Q'). In order to find Q from Q' we can try (by trial and error) each one of the four or five integers that are within 2 from Q'. One of them would be Q. The best such choice is the one that give $0 \leq X - YQ = R < Y$. This trial and error step requires no more than 10, i.e. $O(1)$ multiplications and subtractions/additions.

The only step still missing is finding the reciprocal z of y, where $1/2 \leq y \leq 1$, i.e. $1 \leq (1/y) \leq 2$.

Note that reciprocal finding depends on the degree of precision required.

(c) Copyright A. Gerbessiotis. All rights reserved.

Appendix
Binary Numbers: Reciprocal Finding

Let $f(x) = 1/x - y$. The solution for x of the equation $f(x) = 0$ is $x = 1/y$, i.e. the reciprocal of y. Also $f'(x) = -1/x^2$.

Step 1. According to Newton-Raphson, successive approximation to the reciprocal of y. can be established by the following iterative method.

$$x_i = x_{i-1} - f(x_{i-1})/f'(x_{i-1}) = 2x_{i-1} - yx_{i-1}^2 \qquad x_0 = 1.5$$

Note that $1 \leq 1/y \leq 2$. We choose $x_0 = 1.5$ in this iterative method. Such a choice for x_0 guarantees that $|1/y - x_0| \leq 1/2$. This is to be used in Steps 6 below.

Step 2. The error bound of the approximation is given by the following formula

$$1/y - x_i = y(1/y - x_{i-1})^2.$$

This was proved on page 24 (Proof of Lemma Error) for $i = 1$.

Step 3. For $y \leq 1$ the error bound for any integer $i \geq 1$ is (by Step 2 and $y \leq 1$)

$$1/y - x_i \leq (1/y - x_{i-1})^2.$$

Step 4. For any $i \geq 0$, the error bound, since $1/y - x_0 \leq 1/2$ by Step 1, is

$$1/y - x_i \leq (1/y - x_0)^{2^i} \leq 2^{-2^i}.$$

Step 5. For $k = \lg(n - m)$ we obtain that

$$1/y - x_k \leq (1/y - x_0)^{(n-m)} \leq 2^{-(n-m)}$$

Substitute $k = \lg(n - m)$ in Step 4.

Step 6. For $k = \lg(n - m)$ we obtain that

$$1/y - x_k \leq 2^{-(n-m)}$$

Step 7. Return $z = x_k$ as a solution to the reciprocal of y.

Appendix
Analysis of Reciprocal Finding

The algorithm requires $O(\lg n)$ iterations since $k \leq \lg n$ and in each iteration a constant number of multiplications and less expensive additions and subtractions are performed. The cost per iteration is $O(M(n))$ and the cost of all $\lg n$ iterations is $O(M(n) \lg n)$.

Observation 1. At iteration i the approximation of $1/y$ offered by x_i is correct to $2^i - 1$ bits (on the right of the decimal point).

Proof. Since $1/y - x_i \leq 2^{-2^i}$ then $1/y$ and x_i agree in the first $2^i - 1$ digits.

For example $a = .111 = 7/8$ and $b = .11 = 3/4$ agree in two decimal digits and $a - b \leq 2^{-3}$.

In order to achieve the optimal $O(M(n))$ bound instead of the $O(M(n) \lg n)$ one, the following modification of the algorithm is required.

Remark 1. Instead of step 1 run the following step to obtain x_i from x_{i-1}. At iteration i, x_i we maintain $2^i + 1$ bits of x_i and $2^i + 4$ bits of y.

Step 1'.

$$\begin{aligned} \bar{x_i} &= 2x_{i-1} - (.y_1 y_2 \ldots y_{2^i+4}) x_{i-1}^2 \\ x_i &= \lceil \bar{x_i} \rceil \end{aligned}$$

Observation 2. With Step 1. changed as in Remark 1., the i-th iteration of Newton Raphson requires time $M(2^{i+4})$.

Observation 3. The modified Newton-Raphson (according to Remark 1) which follows Step 1', requires a total of $O(M(n))$ operations rather than $O(M(n) \lg n)$.

Proof. Let the time for iteration $i - 1$ in obtaining x_{i-1} be $T(2^{i-1})$. After x_{i-1} is obtained a constant number of multiplications and additions/subtractions is required to obtain x_i. These operations involve 2^{i+4} precision/bit numbers. Therefore, $T(2^i) = T(2^{i-1}) + O(M(2^{i+4}))$. The solution is $T(2^i) = O(M(2^i))$ and thus $T(n) = O(M(n))$.

Appendix
Analysis of Reciprocal Finding: Optimality

The Proof of Observation 3 is involved. We present a highlight of it. Let $e_i = 1/y - x_i$ be the error rate in the i-th approximation using the formula $x_i = 2x_{i-1} - yx_{i-1}^2$. We showed earlier that $e_i = 1/y - x_i = y(1/y - x_{i-1})^2 = ye_{i-1}^2$. Given that $1/2 \leq y \leq 1$, we derived $e_i = e_{i-1}^2 = e_0^{-2^i} = 2^{-2^i}$.

In order to prove the bound we use the slightly modified formula, $x_i = 2x_{i-1} - y_{i-1}x_{i-1}^2$, where y_{i-1} is y rounded down (floor) to $2^i + 4$ bits. Then the error rate e_i of the approximation becomes.

$$\begin{aligned} e_i = 1/y - x_i &= 1/y - 2x_{i-1} + y_i x_{i-1}^2 \\ &= 1/y - \left(2(1/y - e_{i-1}) - y_i(1/y - e_{i-1})^2\right) \\ e_i &= y_{i-1}e_{i-1}^2 - \frac{y - y_{i-1}}{y^2}(1 - 2e_{i-1}y) \end{aligned}$$

We shall prove by induction on i that $e_i \leq 2^{-2^i}$ as before (the $O(M(n)\lg n)$ case). As soon as x_i is found it is then rounded up (ceiling) to the $2^i + 1$-st bit.

From the last expression the error bound is (including the round up which causes an error of at most 2^{-2^i-1}) given by the expression below

$$|e_i| \leq \max\{y_{i-1}e_{i-1}^2, \frac{y - y_{i-1}}{y^2}(1 + 2e_{i-1}y) + 2^{-2^i-1}\}$$

We examine separately the various error contributions.

By the inductive step $e_{i-1} \leq 2^{-2^{i-1}}$. Also, y_{i-1} is y rounded down, i.e. $y_{i-1} \leq y$ and $y \leq 1$. We thus get

$$y_{i-1}e_{i-1}^2 \leq ye_{i-1}^2 \leq e_{i-1}^2 \leq 2^{-2^i}$$

For the second term, we note that $y - y_{i-1} \leq 2^{-2^i-4}$, $1/y \leq 2$, i.e. $1/y^2 \leq 4$, and $(1 + 2e_{i-1}y) \leq 2$. Then

$$\frac{y - y_{i-1}}{y^2}(1 + 2e_{i-1}y) + 2^{-2^i-1} \leq 2^{-2^i-4} \cdot 4 \cdot 2 + 2^{-2^i-1} = 2^{-2^i-1} + 2^{-2^i-1} = 2^{-2^i}.$$

The two term of the maximum have the same upper bound 2^{-2^i} and thus $e_i \leq 2^{-2^i}$ as required.

Appendix
An Example

Suppose that $X = 20$ and $Y = 5$. Then $X = 10100$ and $Y = 101$. Then $X = x2^3 = 10.100 \times 2^3$ and $Y = y2^3 = 0.101 \times 2^3 = 5/8 \times 2^3$. $y = 5/8$ is between $1/2$ and 2 and we can Newton-Raphson find an approximation to the inverse of y, which is $1/y$. Let $x_0 = 1.5$ as directed by the method. Then $x_1 = 2x_0 - yx_0^2 = 1.59375 = 1.10011$. If we only maintain $2^1 - 1$ bits on the right of . for x_1 even that value $\bar{x_1} = 1.100$ is a good approximation of $1/y$ which is 1.6 since by Remark 1 we can maintain only $2^i + 1$ bits of x_i in iteration i. In fact $1/y - \bar{x_1} < 2^{-2^1} = 2^{-2}$ according to Newton Raphson. Then for x_2, $2^2 + 1$ bits to the right of . are also a good approximation to $1/y$ and $\bar{x_2} = 1.1001$ is such that $1/y - \bar{x_2} \leq 2^{-2^2} = 1/16$ since $x_2 = 2x_1 - yx_1^2 = 1.5999755859 = 1.1001100110011$.

Therefore the approximationto $1/y$ we have established is $z = 1.1001$ which is 5 bits The $1/Y = 1/y * 2^{-3} = 0.0011001 = 0.19531250$, which is close enough to 0.2. Then we form the product $xz = 10.100 * 1.1001$ which is $Q' = xz = 11.11101 = 3.90625$; note that both x and z have as many digits as X. Then the quotient Q of X by Y is between 1.90625 and 5.90625 since Q' is 2 away from Q. The only integers in that range are $t_1 = 2, t_2 = 3, t_3 = 4, t_4 = 5$. We try each one in turn as a candidate for Q by computing $r_i = X - Yt_i$ and checking whether the result r_i is between 0 (inclusive) and Y (exclusive). We find $r_1 = 10$, $r_2 = 5$ $r_3 = 0$ and $r_4 = -5$. The only value that satisfies the latter requirement is $r_3 = 0$ and we declare $Q = t_3 = 4$ the quotient of the division of X by Y and the remainder $R = r_3 = 0$.

It is easy to verify the validity of these claims.