

INTEGER AND POLYNOMIALS. MATRICES.

The WORD and the BIT model

Chapters 11.2-11.3, 12.1 of GT

DISCLAIMER: *Report an error / typo to the author so that he resolves it.*
These notes are NOT to be distributed outside of NJIT without the expressly written permission of A. Gerbessiotis

Models of Computation

Overview

When we perform arithmetic operations, sometimes we do not distinguish one operation from the other and an operation count is derived that includes additions, subtractions, multiplications, etc. Sometimes we do distinguish between additions and subtractions on one hand, and multiplications on the other, and we do not allow divisions. Such models for arithmetic operations are called word models. The word model of computation is also referred to as the RAM (Random Access Machine) model. We use such models when we discuss operations on polynomials and matrices or generic algorithms. When bit operations are counted and accounted we then deal with a bit model of computation, and operations are reduced to the number of bit operations needed to complete the operation. In the remainder integer are to be unsigned integers unless otherwise noted. Moreover, several times in the word model, we employ what is known as a straight line program (SLP): in simplistic terms it is a program with no branching structures (i.e. if-then-else, and goto statements), or iterative structures (i.e. for-loops, while-loops) and thus consist of straightforward (assignment) statements utilizing binary operators with operands that can be either input parameters, constants or intermediate results (and no indirect access through pointers to main memory).

Definition 1 (Word model: technical definition). In the word model, (main) memory is an array of words that are b -bit wide. The b -bit width can represent all integers that are to be represented. Moreover, $b \geq \lg n$, where n is the problem size. A word can be accessed in time 1 or $\Theta(1)$ and a block of memory of size k words in time $O(k)$. Regular operations such as an addition, a subtraction, a multiplication and similar other operations (e.g. shift left, shift right) are completed in time 1.

Definition 2 (Bit model). In the bit model the cost of access is bit-based. Thus a word b bit wide will be charged of cost b , and operation cost is the bit cost of each operation.

Definition 3 (Straight line program (SLP) model). The model does not allow branching or iterative structures and it is a sequence of assignment statements using binary operators with operands that can be either input parameters, constants or intermediate results.

Number of bit of an unsigned integer

An operation involving two integers costs a number of bit operations proportional to the number of bits of the two integers. If the input consists of two numbers a, b one may assume that $a > b$ in general. If one does not specify the number of bit for a, b , the value of a i.e. a can be used to extract this information. Some other times a, b are just variables holding values and in that case it is specified that say a is an n -bit integer and b is an m -bit integer. (Moreover we may assume $a > b$ and by extension $n > m$ or otherwise we swap the order of the variables listed.) Moreover we might just say that we have two n -bit integers a, b . You can make the smaller to have as many bit as the larger by padding leading zeroes.

One more and final caution: integers means unsigned integers.

Fact 1 (Floors and ceilings). For any real number x , we have that $x \leq \lceil x \rceil$ and $x \geq \lfloor x \rfloor$.

Proposition 1. The number of bit needed to represent $a > 0$ is $\text{bit}(a)$ or $|a|$, where $|a| = \text{bit}(a) = \lceil \lg(a+1) \rceil = \lfloor \lg a \rfloor + 1$.

Proof. Integer 2^{m-1} is represented by an 1 followed by $m-1$ zeroes, i.e. it has m bit. $2^m - 1$ is a sequence of m 1s. So both integers can be represented by m bits. And so are all integers between them.

An integer a can be sandwiched between two consecutive powers of 2. That is, if a is between 2^{m-1} and $2^m - 1 < 2^m$ then a needs m bit for its representation.

$$\begin{aligned} 2^{m-1} &\leq a < 2^m \Leftrightarrow 2^{m-1} \leq a \leq 2^m - 1 \Leftrightarrow 2^{m-1} + 1 \leq a + 1 \leq 2^m \\ 2^{m-1} &< a + 1 \leq 2^m \Leftrightarrow \lg(2^{m-1}) < \lg(a + 1) \leq \lg(2^m) \Leftrightarrow m - 1 < \lg(a + 1) \leq m \end{aligned}$$

From the last one if $\lg(a + 1) \neq m$ then $m = \lceil \lg(a + 1) \rceil$. Use the same or a similar method to show that m is also $\lfloor \lg a \rfloor + 1$. ■

Number of bit of an (unsigned) integer

Computer storage

Example 1. Integer 3 requires 2 bits, integer 15 requires 4 bits, but integer 16 requires 5 bits for its representation and so do 17, ..., 31.

Integer a of $|a|$ bit is stored in an array of length $|a|$. If the array is A , then $A[0]$ stores the rightmost bit, $A[1]$ the second rightmost bit, and so on. This defines a little endian representation of the bit of a . If $A[0]$ were to store the leftmost bit that would lead to a big endian representation of a .

Suppose we have integer $a = 177$. By default it is written in denary (i.e. base-10, or radix-10). Sometimes we say so with a subscript i.e. $a = 177_{10}$. The binary representation of a is $a = 10110001_2 = 0b10110001 = 10110001 = [10110001]$. The former representation with the subscript 2 or the prefix $0b$ is preferable and non confusing, the bracketed one is preferable to the middle third one. One could read the third representation as an integer which is about 10 million: that would be incorrect though.

Example 2. Let $a = 177$, then $a = 0b10110001$ and thus $a = [10110001]$. We may store the bit of a into array A with length $\text{bit}(a) = 8$. Then $A[0] = 1, A[1] = 0, A[2] = 0$ and so on.

Definition 4 (Little endian internal presentation). The bit of integer a are stored in an array so that $A[i]$ stores the i -th rightmost bit.

Example 3. Thus the internal representation stores in the lowest index associated memory the least-significant (i.e. rightmost) bit of the integer. Thus A can be summarily written $A = [1, 0, 0, 0, 1, 1, 0, 1]$. This is the little-endian representation of the bits of a right to left i.e. from least-significant (right-most) to most-significant (left-most).

In a big-endian representation stored in A' , we would have $A' = [1, 0, 1, 1, 0, 0, 0, 1]$ and we would store the bits left to right, most-significant to least-significant.

Definition 5 (Big endian internal presentation). The bit of integer a are stored in an array so that $A[i]$ stores the i -th leftmost bit.

Example 4. Integer $b = 0b11001001$ in binary is $1 + 8 + 64 + 128 = 201$ in denary. The little endian representation of it is $B = [1, 0, 0, 1, 0, 0, 1, 1]$.

Operation : Addition

All integers in the remainder of the discussion on the bit model are assumed to be unsigned integers that is non-negative integers.

Problem 1 (Bit model addition: Formulation 1).

Input. An n -bit integer a , and an m -bit integer b .

Output. Calculate the integer c such that $c = a + b$.

In the previous definition of the problem one may assume that $n \geq m$: if it is the other way around we swap the values of variables a and b . Moreover we may pad the smaller m -bit integer with leading zeroes to make both of the same number of bit n . Thus an alternative formulation is

Problem 2 (Bit model addition: Formulation 2).

Input. An n -bit integer a , and an n -bit integer b .

Output. Calculate the integer c such that $c = a + b$.

In that latter case c has n or $n+1$ bit. Another formulations is the following one that is weaker than the original formulation.

Problem 3 (Bit model addition: Formulation 3).

Input. An integer a , and an integer b .

Output. Calculate the integer c such that $c = a + b$.

Since one input is integer a the number of bit of a is $|a| = \text{bit}(a)$ and likewise of b is $|b| = \text{bit}(b)$. Thus we recover Formulation 1 where $n = |a| = \text{bit}(a)$ and $m = |b| = \text{bit}(b)$. Note that $|a| = \text{bit}(a) = \Theta(\lg a)$ and $|b| = \text{bit}(b) = \Theta(\lg b)$.

Problem 4 (Bit model subtraction: Formulation 1).

Input. An n -bit integer a , and an m -bit integer b .

Output. Calculate the integer c such that $c = a - b$.

Problem 5 (Bit model subtraction: Formulation 2).

Input. An n -bit integer a , and an n -bit integer b .

Output. Calculate the integer c such that $c = a - b$.

Problem 6 (Bit model subtraction: Formulation 3).

Input. An integer a , and an integer b .

Output. Calculate the integer c such that $c = a + b$.

Remark 1 (Subtraction is an Addition). In the remainder the terms addition and subtraction can be used interchangeably. Though $a - b$ is considered a subtraction one might call it an addition since $a - b$ is equal to $a + (-b)$.

Proposition 2 (Bit of $a + b$ and $a - b$). *Let a and b be two (unsigned) integers $a \geq b$, with n and m bit respectively, where $n \geq m$. Then $a + b$ or $a - b$ has at most $\max\{n, m\} + 1$ and $\max\{n, m\}$ bit respectively.*

Note that in this discussion an integer is an unsigned integer. This is implied in most cases though not indicated.

Proposition 3 (Addition and subtraction of integers). *Let a and b be two integers $a \geq b$, with n and m bit respectively, where $n \geq m$. Then $a + b$ or $a - b$ takes time $\Theta(n + m)$.*

Theorem 1. *Let $A(n)$ be the time to add two integers with n bit, and let $A(n, m)$ be the time to add two integers with n, m bit respectively. Let $S(n)$ and $S(n, m)$ be the time to subtract those integers. Then, in the bit model, we have the following*

$$A(n) = S(n) = \Theta(n),$$

$$A(n, m) = S(n, m) = \Theta(n + m) = \Theta(\max\{n, m\}),$$

for the reference elementary school algorithm in the bit model.

Corollary 1. The time to add/subtract integers a and b , where $a \geq b$, is $A(n, m) = S(n, m)$, where $n = \text{bit}(a)$ and $m = \text{bit}(b)$, or $A(n) = S(n)$ as needed.

Problem 7 (Bit model multiplication).

Input. An n -bit integer a , and an m -bit integer b .

Output. Calculate the integer c such that $c = a * b$.

Proposition 4 (Bit of $a * b$). *Let a and b be two integers $a \geq b > 0$, with n and m bit respectively, where $n \geq m$. Then $a * b$ has $n + m$ bit.*

Proposition 5 (Multiplication of integers). *Let a and b be two integers $a \geq b$, with n and m bit respectively, where $n \geq m$. Then, in the bit model, $a * b$ takes time $O(n * m)$, using the elementary school algorithm.*

Corollary 2. Let $M(n)$ be the time to multiply two integers with n bit, and let $M(n,m)$ be the time to multiply two integers with n and m bit respectively. Then we have the following

$$M(n) = O(n^2),$$

and

$$M(n, m) = O(n * m),$$

for the reference elementary school-based multiplication method in the bit model.

Problem 8 (Bit model multiplication by 2^k).

Input. An n -bit integer a , and integer $k > 0$.

Output. Calculate the integer c such that $c = a2^k$.

Problem 9 (Bit model division by 2^k).

Input. An n -bit integer a , and integer $k > 0$.

Output. Calculate the integer c such that $c = a/2^k$.

Definition 6 (Integer multiplication of b by 2^k). $SHL(b,k)$ represents the integer multiplication of b by 2^k i.e. $SHL(b,k) = b \cdot 2^k$. Its binary representation is that of b with k added zeroes to the right of the binary representation of b . Moreover $|SHL(b, 2^k)| = |b| + k$.

Definition 7 (Integer division of b by 2^k). $SHR(b,k)$ represents the integer division of b by 2^k i.e. $SHR(b,k) = \text{int}(b/2^k)$. Its binary representation is that of b with the k rightmost bit truncated (removed). Moreover $|SHR(b, 2^k)| \leq |b|$.

Shift for multiplication and division

Examples

Example 5. Let $b = 5_{10}$. Then $b = [101] = 101_2$. Then $b \times 2^3$ is $b \times 2^3 = 5 \times 8 = 40$. In binary this is $[101000]$ i.e. b shifted left 3 positions, and that 3 is the exponent of 2^3 .

Example 6. Let b be as above. Then $b/2^2$ is equal to $5/4 = 1.25$. For $b = [101]$ the decimal point is omitted but is to the right of the least significant digit i.e. $b = [101.]$. Then $b/2^2$ is equivalent to a shift right by two positions and thus $b/2^2 = [1.01]$ or the integer division leaves $[1.]$ as needed.

If bit positions on the left of the decimal point map to zero or positive powers of two, then to the right of the decimal point would map to negative powers of two. Then, $b/2^2 = [1.01]$ can be interpreted as $1 \times 2^0 + 0 \times 2^{-1} + 1 \times 2^{-2} = 1 + 1/4 = 1.25$.

Word model Polynomials

A polynomial of degree n is represented by the ordered tuple (array) of its coefficients.

Definition 8 (Degree and degree bounds of polynomial). A polynomial $f(x) = f_0 + f_1x + f_2x^2 + \dots + f_nx^n$, where $f_n \neq 0$, has degree $d(f)$ equal to n that is, $d(f) = n$ and degree bound $db(f)$ equal to $n + 1$ that is, $db(f) = n + 1$. In other words, $db(f) = d(f) + 1$. If we store (the coefficients of) $f(x)$ in array F , the length of F is the degree bound of f . Moreover $F = [f_0, f_1, \dots, f_n]$ in little endian representation.

Example 7. A polynomial such as $f(x) = x^7 + x^5 + x^4 + 1$ is of degree 7 as the largest exponent of x is a 7. We have $d(f) = 7$. The degree-bound of f is $db(f) = 8$. In main memory f can be stored in array F of length $db(f)$ and $F = [1, 0, 0, 0, 1, 1, 0, 1]$.

Problem 10 (Polynomial evaluation). .

Input. A polynomial f of degree n , its representation F and an integer number t . ($d(f) = n$ and $db(f) = n + 1$.) Moreover, $F = [f_0, \dots, f_n]$.

Output. Calculate $f(x)|_{x=t} = f(t)$ i.e. $f(t)$.

Example 8. For the polynomial $f(x) = x^7 + x^5 + x^4 + 1$, $f(0) = 1$ and $f(1) = 177$.

Definition 9 (Point value representation of polynomial). A polynomial $g(x)$ of degree $n - 1$ and degree bound n can also be uniquely represented by n different point-value pairs, i.e. the tuple

$$((x_0, g(x_0)), \dots, (x_{n-1}, g(x_{n-1}))) \text{ where } x_i \neq x_j \text{ for all } i, j$$

Theorem 2 (Lagrange). For a polynomial $g(x)$ of degree $n - 1$ represented by n point value pairs, the coefficients of $g(x)$ can be found through an operation known as interpolation using Lagrange's formula in $O(n^2)$ time. In other words the $G = (g_0, \dots, g_{n-1})$ representation is equivalent to the point-value representation.

$$((x_0, g(x_0)), \dots, (x_{n-1}, g(x_{n-1}))) \text{ where } x_i \neq x_j \quad \forall i, j$$

Proof. If we are given the G representation of a polynomial $g(x)$, the point-value representation can be obtained easily in $O(n^2)$ time. However, it is not obvious how to obtain the coefficient vector G of a polynomial $g(x)$ if we are given the point-value representation of $g(x)$. A theorem (formula) due to **Lagrange** is the solution. Consider n distinct point-value pairs $((x_1, b_1), \dots, (x_n, b_n))$. We want to find the $n - 1$ degree polynomial $a(x) = a_{n-1}x^{n-1} + \dots + a_1x + a_0$ such that $a(x_i) = b_i$, $1 \leq i \leq n$.

$$a_i(x) = \frac{(x - x_1)(x - x_2) \dots (x - x_{i-1})(x - x_{i+1}) \dots (x - x_n)}{(x_i - x_1)(x_i - x_2) \dots (x_i - x_{i-1})(x_i - x_{i+1}) \dots (x_i - x_n)}$$

It is easy to verify that $a_i(x_i) = 1$ as the numerator and denominator cancel out, and $a_i(x_j) = 0$ for $j \neq i$ as $(x_j - x_j)$ appears in the numerator and the corresponding term in the denominator is $x_i - x_j$. If we consider the polynomial derived from all $a_i(x)$ such that

$$a(x) = b_1 a_1(x) + \dots + b_n a_n(x)$$

we observe that $a(x_i) = b_i$ for all $i = 1, \dots, n$. Therefore $g(x) = a(x)$. ■

Definition 10 (Duality of BIT and WORD model). We notice that array F representing f is identical to the array A representing $a = 177$. One can verify that $f(2) = f(x)|_{x=2}$ which gives $f(2) = 177$. Thus there is a connection between the bit-model for integer operations and the word-model for polynomial operations (for polynomials with zero and one coefficients).

Example 9. Polynomial $b(x)$ such that $b(x) = 1 + x^3 + x^6 + x^7$ has a little endian representation of the form $B = [1, 0, 0, 1, 0, 0, 1, 1]$.

We have seen before, that integer 201 has the same little endian representation.

Example 10. Integer $b = 0b11001001$ in binary is $1 + 8 + 64 + 128 = 201$ in denary. The little endian representation of it is $B = [1, 0, 0, 1, 0, 0, 1, 1]$.

Corollary 3. Let $b(x)$ be a polynomial with coefficients 0 and 1, and let b be a binary integer (non-negative). If the little-endian representation of $b(x)$ say B is the same as that of b , then $b = b(2)$.

Example 11. Polynomial $b(x) = 1 + x^3 + x^6 + x^7$ and integer 201 have the same internal representation using B . Thus $201 = b(2)$, which is obvious.

Problem 11 (Word model polynomial addition).

Input. A polynomial f of degree n in the form of F , and a polynomial g of degree m in the form of G , where $n \geq m$,

Output. Calculate the polynomial h such that $h(x) = f(x) + g(x)$.

Problem 12 (Word model polynomial subtraction).

Input. A polynomial f of degree n in the form of F , and a polynomial g of degree m in the form of G , where $n \geq m$,

Output. Calculate the polynomial j such that $j(x) = f(x) - g(x)$.

For polynomials, how long does it take to add or subtract two polynomials? It depends on the degree bound of the two polynomials. And the degree bound of a polynomial is one more than the degree of it.

Proposition 6 (Degree and degree bound of $f + g$ and $f - g$). *Let f and g be two polynomials of degree n and m respectively, where $n \geq m$. Then $f + g$ or $f - g$ is a polynomial of degree at most $\max\{d(f), d(g)\}$. Moreover $db(f + g) = d(f + g) + 1$ and $db(f - g) = d(f - g) + 1$.*

Polynomial addition and subtraction

Running time

Proposition 7 (Addition and subtraction of polynomials). *Let f and g be two polynomials of degree n and m respectively, where $n \geq m$. Then $f + g$ or $f - g$ takes time $O(\text{db}(f + g)) = O(d(f + g))$ and $O(\text{db}(f - g)) = O(d(f - g))$ respectively.*

Corollary 4. Let $A(n)$ be the time to add two polynomials of degree n , and let $A(n,m)$ be the time to add two polynomials of degree m,n respectively. Let $S(n)$ and $S(n,m)$ be the time to subtract those polynomials. Then we have the following

$$A(n) = S(n) = \Theta(n),$$

$$A(n, m) = S(n, m) = \Theta(n + m) = \Theta(\max\{n, m\}),$$

in the word model for the reference high school algorithm.

Example 12. Let $f(x) = 1 + x^3 + x^6 + x^7$ with $F = [1, 0, 0, 1, 0, 0, 1, 1]$ and $g(x) = 1 + x^2 + x^4$ with $G = [1, 0, 1, 0, 1, 0, 0, 0]$. The degree and degree bounds are 7, 8 for f and 4, 5 for g . Yet we adjust g to use as much as space as f . How much is $f(x) + g(x)$? Well obviously $h(x) = f(x) + g(x) = 2 + x^2 + x^3 + x^4 + x^6 + x^7$, and thus $H = [2, 0, 1, 1, 1, 0, 1, 1]$.

Problem 13 (Word model polynomial multiplication).

Input. A polynomial f of degree n in the form of F , and a polynomial g of degree m in the form of G , where $n \geq m$,

Output. Calculate the polynomial k such that $k(x) = f(x) * g(x)$.

Proposition 8 (Degree and degree bound of $f * g$). *Let f and g be two polynomials of degree n and m respectively, where $n \geq m$. Then $f * g$ is a polynomial of degree $d(f) + d(g) = n + m$, and $db(f * g) = d(f * g) + 1 = n + m + 1$.*

Proposition 9 (Multiplication of polynomials). *Let f and g be two polynomials of degree n and m respectively, where $n \geq m$. Then $f * g$ takes time $O(db(f) * db(g)) = O(d(f) * d(g))$.*

Corollary 5. Let $M(n)$ be the time to multiply two polynomials of degree n , and let $A(n,m)$ be the time to multiply two polynomials of degree m, n respectively. Then we have the following

$$M(n) = O(n^2),$$

and

$$M(n, m) = O(n * m),$$

in the word model for the reference high school-based multiplication method.

Word model

Polynomial multiplication completed

Example 13. Let $f(x) = 1 + x^3 + x^6 + x^7$ with $F = [1, 0, 0, 1, 0, 0, 1, 1]$ and $g(x) = 1 + x^2 + x^4$ with $G = [1, 0, 1, 0, 1, 0, 0, 0]$. If we multiply $f(x) * g(x)$ we need to multiply all terms of f with all terms of g . This will cost us no more than $db(f) * db(g)$ multiplications plus no more than the same number of additions to combine same power monomials (eg. x^7 as the product of x^7 of f and 1 of g or the product of x^3 and x^4 .) We sometimes write that the cost of polynomial multiplication for the reference high school method is $O(d(f)d(g))$ thus simplifying the $O(db(f)db(g)) = O((d(f) + 1)(d(g) + 1))$. Completing the multiplication we conclude that $h(x) = f(x)g(x)$ has $H = [1, 0, 1, 1, 1, 1, 2, 1, 1, 1, 1]$

Moreover we observe again the duality between integers in the bit model and polynomials in the word model.

Example 14 (Nested polynomial multiplication). Say we have three polynomials f, g, h with degree 3, 4, 5. Let the cost of multiplying two polynomials of degrees n, m be $n \times m$. (If this is too simplistic, make the cost $(n + 1)(m + 1)$ for further practice.) What is the best way to multiply the three polynomials? Polynomial and integer multiplication is associative which means $(f(x) * g(x)) * h(x) = f(x) * (g(x) * h(x))$. It is also commutative that is $f(x) * g(x) = g(x) * f(x)$.

Proof. Say we multiply $p_1(x) = f(x)g(x)$. This is at a cost of $3 \times 4 = 12$. Then $d(p_1) = 7$. Finally we multiply $p_1(x)h(x)$ at a cost of $7 \times 5 = 35$. The total cost is $35 + 12 = 47$ and $f(x)g(x)h(x)$ has degree $7 + 5 = 12$.

Say we multiply $q_1(x) = g(x)h(x)$. This is at a cost of $4 \times 5 = 20$. Then $d(q_1) = 9$. Finally we multiply $q_1(x)f(x)$ at a cost of $9 \times 3 = 27$. The total cost is $20 + 27 = 47$ and $f(x)g(x)h(x)$ has degree $7 + 5 = 12$.

Same answer, same degree and same cost though individual costs differ. Do you have a proof? ■

Word model

Operation: multiplication with a monomial

Problem 14 (Word model polynomial multiplication with monomial).

Input. A polynomial f of degree n in the form of F , and a monomial x^k of degree k .

Output. Calculate the polynomial p such that $p(x) = SHL(f, k) = f(x) * (x^k)$.

Some results can translate from one model to the other!

Corollary 6 (SHL polynomials). For $SHL(f, k) = f(x) \cdot x^k$, we have that $db(SHL(f, k)) = db(f) + k$.

DRAFT

Example 15.

$f(x) = 1 + x^{**3} + x^{**6} + x^{**7}$	$d(f)=7$ $db(f)=8$	$F: [1 \ 0 \ 0 \ 1 \ 0 \ 0 \ 1 \ 1]$
$g(x) = 1 + x^{**2} + x^{**4}$	$d(g)=4$ $db(g)=5$	$G: [1 \ 0 \ 1 \ 0 \ 1]$
$h(x) = f(x) + g(x)$ SUM	$d(.) \leq \max(d(f), d(g))$	$F+G: [2 \ 0 \ 1 \ 1 \ 1 \ 0 \ 1 \ 1]$
$j(x) = f(x) - g(x)$ DIF	$db(.) = d(.) + 1$	$F-G: [0 \ 0 \ -1 \ 0 \ -1 \ 0 \ 1 \ 1]$
$k(x) = f(x) * g(x)$ MUL	$d(k) = d(f) + d(g)$	$K: [1 \ 0 \ 1 \ 1 \ 1 \ 1 \ 1 \ 2 \ 1 \ 1 \ 1 \ 1]$
	$db(k) = d(k) + 1$	

Example 16.

$$b = \text{SHL}(a, k) = a \ll k = a * 2^k$$
$$|b| = |a| + k$$

$$a = 5$$
$$b = \text{SHL}(a, 3) = a \ll 3 = a * 2^3$$
$$= 40 = [101000]$$
$$A = [1, 0, 1]$$
$$B = [0, 0, 0, 1, 0, 1]$$

$$c = \text{SHR}(a, k) = a / 2^k = a \gg k$$

$$a = 5 = [101]_2 = [101]_2$$
$$c = \text{SHR}(a, 3) = 5 / 2^3 = 5/8$$
$$= [.101]_2 = 2^{-1} + 2^{-3} = 0.625$$

$$g = \text{SHL}(f, k) = f(x) * x^k$$
$$\text{db}(g) = \text{db}(f) + k$$

$$f(x) = 1 + x^4 + x^5 + x^7$$
$$g(x) = \text{SHL}(f, 3) = f(x) * x^3$$
$$= x^3 + x^7 + x^8 + x^{10}$$
$$F: [1, 0, 0, 0, 1, 1, 0, 1]$$
$$G: [0, 0, 0, 1, 0, 0, 0, 1, 1, 0, 1]$$

Integers and Polynomials

Operation complexity

Example 17. We summarize below the reference algorithm performance.

(unsigned) INTEGERS : Bit model
 a : n bit , b : m bit $n \geq m$
 $|a|=n$, $|b|=m$

Operation	Result length
$a+b$	n or $n+1$ bit
$a-b$	1 to n bit
$a*b$	1 to $n+m$ bit
$a*(2^k)$	$n+k$ bit

Operation	Time Complexity
$a+b$	$A(n,m) = \Theta(n+m)$
$a-b$	$S(n,m) = \Theta(n+m)$
$a*b$	$M(n,m) = \Theta(n*m)$
$a*(2^k)$	$\Theta(n+k)$

For $n \geq m$

$a+b$	$A(n) = \Theta(n)$
$a-b$	$S(n) = \Theta(n)$
$a*b$	$M(n) = \Theta(n*n)$

POLYNOMIALS : Word model
 f : $d(f)=n$, g : $d(g)=m$
 $db(f)=n+1$, $db(g)=m+1$

Operation	Result length
$f(x)+g(x)$	$d(h) \leq n$
$f(x)-g(x)$	$d(j) \leq n$
$f(x)*g(x)$	$d(k) \leq n+m$
$f(x) * (x^k)$	$d(f)+k$

Operation	Time Complexity
$f(x)+g(x)$	$A(n,m) = \Theta(n+m)$
$f(x)-g(x)$	$S(n,m) = \Theta(n+m)$
$f(x)*g(x)$	$M(n,m) = \Theta(n*m)$
$f(x) * (x^k)$	$\Theta(n+k)$

For $n \geq m$

$f(x)+g(x)$	$A(n) = \Theta(n)$
$f(x)-g(x)$	$S(n) = \Theta(n)$
$f(x)*g(x)$	$M(n) = \Theta(n*n)$

Example 18.`c , d, |c| = bit(c) >= |d| = bit(d)`

Operation Result length in bit

`c+d` `<= max(|c|,|d|) +1` bit`c-d` 1 to `|c|` bit`c*d` 1 to `|c|+|d|` bit`c*(2**k)` `|c|+k` bit

Operation Time Complexity

`c+d` `Theta(|c|+|d|)` or `Theta(|c|)` or `Theta(lgc)``c-d`            `Theta(|c|+|d|)` or `Theta(|c|)`       or `Theta(lgc)``c*d`            `Theta(|c|*|d|)` or `Theta(|c|**2)` or `Theta((lgc)**2)``c*(2**k)`       `Theta(|c|+k)`       or                               or `Theta(lgc+k)`

Bit (or Word) model

Complexity results

Theorem 3 (Squaring vs Multiplication). *Integer multiplication is no harder than integer squaring, and integer squaring is no harder than integer multiplication. By the former we have $M(n) = O(Q(n))$, and by the latter $Q(n) = O(M(n))$.*

Proof. Let $M(x, y)$ be the multiplication of x and y i.e. $M(x, y) = x * y$, and let $Q(x)$ be the square of x i.e. $Q(x) = x^2$. Let $Q(n), M(n)$ be the time to square an integer (or real) number and multiply two integer (or real) numbers. We can show that both problems are equivalent. If we can multiply we can square (obviously). If we can square we can also multiply (not very obvious).

Integer squaring is no harder than integer multiplication. Given that $Q(x) = M(x, x)$, the result follows. Moreover if $Q(n), M(n)$ are as defined, then $Q(n) = M(n)$.

Integer multiplication is no harder than integer squaring. If we need to be more careful and precise, we can specify this problem in the bit model by defining by $M(n)$ to be the time to multiply two n -bit numbers and $Q(n)$ to be the time to square an n -bit number. Then $M(n) = O(Q(n) + n)$. Let A, S, SHL, SHR, Q be addition, subtraction, shift left (multiplication by a power of two), shift right (division by a power of two), and squaring. Then

$$a * b = \frac{(a + b)^2 - (a - b)^2}{4} == SHR(S(Q(A(a, b)), Q(S(a, b))), 2).$$

Thus $M(n) = O(Q(n) + n)$, since multiplication requires two square operations of $(a + b)$ and $(a - b)$ and linear time operations such as addition, subtraction used twice, and shift right by two which is the division by four. ■

Theorem 4 (Squaring vs Reciprocal finding). *If $Q(n)$ is the time to square an n -bit number and $R(n)$ is the time to find the reciprocal of an n -bit number $1/a$ then $Q(n) = O(R(n) + n)$.*

Proof. Left as an exercise. ■

Bit (or Word) model
Complexity results

Example 19.

$A(x,y) = x + y$ $M(x,y) = x * y$ $SHL(x,k) = x * 2^{**k}$
 $S(x,y) = x - y$ $Q(x) = x * x$ $SHR(x,k) = x / 2^{**k}$

=====

if $M(x,y)$ then $Q(x)$: Proof $Q(x) = M(x,x)$

if $Q(x)$ then $M(x,y)$: Proof $M(x,y)$
 $t1 = A(x,y)$; // $x+y$
 $t2 = Q(t1)$; // $(x+y)^{**2}$
 $t3 = S(x,y)$; // $(x-y)$
 $t4 = Q(t3)$; // $(x-y)^{**2}$
 $t5 = S(t2,t4)$; // $(x+y)^{**2} - (x-y)^{**2} = 4xy$
 $t6 = SHR(t5,2)$; // $x*y$
 return($t6$) ; // $x*y$

Bit model Complexity results

The following complexity results are applicable.

$M(x, y)$ defines the multiplication of x and y i.e. $M(x, y) = x * y$. The reciprocal $I(x)$ of x is $I(x) = 1/x$. The division $D(x, y)$ of x by y is $D(x, y) = x/y$. For integer division $R(x, y)$ is the remainder of the integer division of x by y and thus $R(x, y) = x - D(x, y) * y$.

Theorem 5 (Multiplication or Squaring and Reciprocal finding). *Multiplication or Squaring is no harder than reciprocal finding.*

Proof. Note that $x^2 = 1/(1/(x-1) - 1/x) + x$. Further details omitted. ■

Theorem 6 (Division vs Reciprocal finding). *Division is no harder than reciprocal finding.*

Proof. $x/y = x \times (1/y)$. Use Newton-Raphson. Details omitted. ■

Theorem 7 (Division vs Reciprocal finding). *$a \bmod b$ 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 $O(\lg q \lg b)$. ■

Word model
Exponentiation: x^n

Problem 15 (Word model x^n).

Input. An integer x and integer $n \geq 0$.

Output. Integer x^n .

Proposition 10 (Naive method). *A naive solution for this problem requires $n - 1$ scalar multiplications.*

We can do better.

Theorem 8 (Exponentiation). *In the word model, x^n can be evaluated in at most $2 \lg n$ (scalar) multiplications. (Other operations are not counted.)*

Proof. Let $k = \lfloor \lg n \rfloor$.

Step 1. We first compute k powers $y_i = x^{2^i}$, $i > 0$, with $y_0 = x$. (Count1). Note that $y_k \leq x^n \leq y_k^2$ implies $k \leq \lg n$.

$$y_0 = x = x^1, y_1 = x^2, y_2 = y_1^2 = x^4, y_3 = y_2^2 = x^8, \dots, y_k = y_{k-1}^2 = x^{2^k}$$

Step 2. Then we get the binary representation of n . ($|n| = k + 1$.) Let it be $n = b_k 2^k + \dots + b_1 2^1 + b_0 2^0$, where $b_i = 0, 1$.

$$x^n = x^{b_0 + b_1 2^1 + \dots + b_k 2^k} = x^{b_0} x^{b_1 2^1} \dots x^{b_k 2^k}$$

Let m of the b_i 's that are one be $b_{l_1}, b_{l_2}, \dots, b_{l_m}$, where $m \leq k + 1$.

Step 3. In the next step we extract the y_j whose corresponding b_j is one. Those y_j 's are all multiplied together to yield x^n . This is because $x^n = x^{2^{l_1}} x^{2^{l_2}} \dots x^{2^{l_m}} = y_{l_1} y_{l_2} \dots y_{l_m}$, i.e. x^n requires $m - 1 \leq k + 1 - 1 = k$ multiplications (Count2) in Step 3. The total multiplication Count is $2k$ i.e. $2 \lfloor \lg n \rfloor \leq 2 \lg n$. ■

Corollary 7 (Exponentiation). In the word model, x^n takes $O(\lg n)$ time to compute from x and n .

Exponentiation: x^n Pseudocode and Example

```

Power( x, n)
1. k = floor(lgn); X[0]=x;
2. for (i=1;i<=k;i++)
3.   X[i] = X[i-1]*X[i-1] // X[i]=x**(2^i)
4. n= B[k]B[k-1]...B[0] //bin(n)
5. if (B[0] == 1) P=X;
6.   else          P=1;
7. for(i=1;i<=k;i++);
8.   if (B[i]==1) P= P* X[i];
9. return(P);

```

```

Power(x,n)
1. if (n==0) return(1);
2. if (n==1) return(x);
3. if mod(n,2) ==1
4. return(x*Power(x*x,(n-1)/2));
5. else
6. return( Power(x*x,n/2));

```

Example 20. Let $n = 13$. Compute x^n for some x . Since $n = 13$, then $k = 3$. We first compute, $y_1 = x^2$, $y_2 = y_1^2 = x^4$, and $y_3 = y_2^2 = x^8$ with three M's. Given that the binary representation of 13 is 1101, then $x^{13} = x^{1 \cdot 8 + 1 \cdot 4 + 0 \cdot 2 + 1 \cdot 1} = y_3 \cdot y_2 \cdot y_0$, i.e. x^n can be computed with 2 additional M's for a total of 5 multiplications.

```

x**13      ==>  x , x**2 , x**4 , x**8      Total : 3M      (1,2,4,8 <=13)

          13 = [ 1      1      0      1      ]
                  x**8 x**4 x**2 x**1
Extract      x**8 x**4          x**1  that correspond to 1 bit in 13
Multiply them x**8 * x**4 *    x**1      in : 2M
                        GrandTotal: 5M ( <= 2lg(13)=6.x)

```

Note 1 (The power of division). If divisions are allowed we can do better. Consider x^{31} . Using the previous method it requires 8M operations. If divisions are allowed we first calculate x^{32} in 5M and then with one division by x we obtain x^{31} .

Word model Polynomial evaluation

Problem 16 (Word model polynomial evaluation).

Input. A polynomial f of degree n and an integer number t . ($d(f) = n$ implies $db(f) = n + 1$.) Moreover, $F = [f_0, \dots, f_n]$.

Output. Calculate $f(x)|_{x=t} = f(t)$ i.e. $f(t)$.

Theorem 9 (Naive polynomial evaluation). *A naive algorithm requires n additions and $2n - 1$ multiplications in the word model. The algorithm is addition/subtraction optimal though.*

Proof. Suppose we are given a polynomial $f(x)$ such that

$$f(x) = f_0 + \dots + f_n x^n$$

where the indeterminate is x and the coefficients f_i can also be real even if we deal with the integer version of this problem. Depending on the circumstances f_i could belong to the set of integer numbers (denoted by Z) or the set of rational numbers (denoted by Q). Suppose we are given $t \in R$. Polynomial evaluation deals with finding

$$f(x)|_{x=t} = f(t).$$

In order to compute $f(t)$ the naive method for evaluation first finds

$$1, t, t^2, \dots, t^n \text{ for a total of } (n - 1) \text{ multiplications}$$

then computes all products of t^i with f_i

$$f_0, f_1 t, f_2 t^2, \dots, f_n t^n \text{ for a total of } n \text{ multiplications.}$$

Naive Polynomial evaluation continued

Continued. Finally by adding up the partial products we compute $f(t)$ by performing n additions.

$$f(t) = f_0 + \dots + f_n t^n \text{ for a total of } n \text{ additions}$$

```
Naive(f[0..n],n,c) /f[0..n] stores f(x) ; we shall compute f(c)
1. result = f[0]; tt=c;
2. for(i=1;i<=n;i++) {
3.   result = result + tt * c[i];
4.   if (i<n) tt = tt * c;
5. }
6. return(result);
```

The total count of the Naive method is : $(2n - 1)$ M's (for multiplications) and n A's (for additions). Line 4 gets complicated to avoid a multiplication at the last round! ■

Polynomial evaluation

Horner's rule

The following algorithm is also known as Horner's rule.

Theorem 10 (Horner's method for polynomial evaluation). *Horner's method uses n multiplications and n additions in the word model. The algorithm is addition/subtraction optimal though.*

Proof. There is a faster way to perform polynomial evaluation however using a method that is known as Horner's method/rule or Horner-Newton rule. According to this method.

$$\begin{aligned}f(x) &= f_n x^n + f_{n-1} x^{n-1} + f_{n-2} x^{n-2} + \dots + f_1 x + f_0 \\f(x) &= (f_n x + f_{n-1}) x^{n-1} + f_{n-2} x^{n-2} + \dots + f_1 x + f_0 \\f(x) &= ((f_n x + f_{n-1}) x + f_{n-2}) x^{n-2} + \dots + f_1 x + f_0 \\f(x) &= (((f_n x + f_{n-1}) x + f_{n-2}) x + \dots + f_1) x + f_0\end{aligned}\tag{1}$$

This method requires n M's and n A's, i.e. it is faster than the naive method.

```
Horner(F[0..n],n,t)
1. result = F[n];
2. for(i=n-1;i>=0;i--) {
3.   result = result * t + F[i];
4. }
5. return(result);
```

Line 3 uses overall n Ms and n As. ■

Straight Line Model

Polynomial evaluation complexity

Theorem 11. Any straight line program to compute $a_0 + \dots + a_n$ requires at least n A/S's (additions or subtractions).

Proof. A straight line program is a program that consists of statements of the form $x_i = x_j + x_k$, where x_j, x_k are constant or variables or input arguments (parameters) and x_i is a variable. For example

```
t2 = a3 * c;  
t3 = t2 + a2;  
t4 = t3 * c;  
t5 = t3 + a1;  
t6 = t5 * c;  
t7 = t6 + a0; outputs a0 + a1 c + a2 c**2 + a3 c**3 in t7.
```

We are going to prove the theorem by induction. Let P_n be the proposition that $a_0 + \dots + a_n$ requires at least n A/S's. Let S_n be any straight line program that computes $a_0 + \dots + a_n$ and would thus require if P_n were true at least n A/S's.

Base Case. For $n = 0$ the result is obvious since P_0 is true by default.

Inductive step. The step takes the form “if P_{n-1} is true, then P_n will be true” or equivalently “if S_{n-1} requires at least $n - 1$ A/S's, then S_n will require at least n A/S's”.

Let us consider the evaluation of $a_0 + \dots + a_n$ and let S_n be any straight line program for computing it. Then if in program S_n we substitute 0 for a_n we could get a SLP $S' = S_n(a_n = 0)$ that evaluates $a_0 + \dots + a_{n-1}$. Thus S' is an S_{n-1} program. We will show that if S' requires at least $n - 1$ A/S's P_n requires at least n A/S's.

Let $a = b \text{ op } c$ be the first $\pm$ operation of S_n that uses a_n . We call this operation O . In such an operation one of b or c may be a_n or one of them is a multiple of a_n computed earlier; neither b nor c can be of any of the forms $b = \pm a_n \pm d$ or $c = \pm a_n \pm e$, since then not O but that operation would be the earliest one using a_n .

O MUST EXIST somewhere otherwise the final result is a multiple of a_n (which in general is absurd) as all operations that use a_n would be $*$ operations. This can not be true in general for arbitrary choices of a_0 through a_n .

We then do the following. ■

Polynomial evaluation complexity continued

Continued. **A1.** In all operations preceding O we substitute 0 for a_n .

A2. In O we substitute 0 for b or c depending on whether b or c is the operand that is a_n or uses a_n .

A3. In all operations following O we substitute 0 for a_n .

The cases of interest for O in A2 (be reminded that we are interested in a $\pm$ lower bound) are

1. $a = b + 0$.
2. $a = b - 0$.
3. $a = 0 + c$.
4. $a = 0 - c$.

In each one of cases 1-3 we eliminate O and replace a in the remainder by b (cases 1 and 2) or c (case 3) as required.

For case 4 we replace the step $a = 0 - c$ that involves a subtraction by the step $a = (-1) * c$, that involves a multiplication of c with constant -1 .

In all four cases we reduced A/S's by one (by either eliminating an A/S or replacing in case 4 a subtraction with a multiplication).

The resulting program $S_{n-1} = S'$ is a program that clearly computes $a_0 + \dots + a_{n-1}$. If $S_{n-1} = S'$ requires at least $n - 1$ A/S's, then S_n that generated S' by eliminating a single A/S operation (and substituting 0 for a_n) would require at least n A/S's. This completes the induction. ■

Corollary 8. Horner's rule requires at least n A/S's.

Proof. Consider any straight-line program P that implements Horner's rule. Let it perform N Additions/Subtractions. Consider the case of evaluating $f(1)$ i.e. the special case where $x = 1$. In P it suffices to replace all occurrences of x by 1 (or equivalently rename x by 1) to compute $f(1)$, which incidentally is $a_0 + \dots + a_n$. The number of $\pm$ steps in this renamed version of P remains the same as in the original P . By Lemma 1 the renamed version of P must have at least n additions/subtractions, i.e. $N \geq n$. Since the program is still P (other than the renaming of x by 1), we conclude that any program like P that does Horner's rule must have $N \geq n$ Additions/Subtractions (since otherwise we could have derived a program for evaluating $a_0 + \dots + a_n$ in fewer than n A/S's). ■

Horner's method is optimal as far as additions and subtractions are concerned. Is it optimal in terms of multiplications? No, if for example preprocessing is allowed.

Polynomial multiplication

Using point value representation

The reference method for multiplying polynomials, multiplies two polynomials of degree $n - 1$ (or degree bound n) in $O(n^2)$ time.

We will show how to reduce this to $O(n^{\lg 3})$ by the end of this subject, and later on, how by using FFT, and combining ideas from the discussion below it can be further reduced to $O(n \lg n)$.

Another way to multiply polynomials is through Interpolation.

Example 21. Let $f(x)$ and $g(x)$ be two polynomials each of degree $n - 1$ and thus of degree bound n . The product $h(x) = f(x) * g(x)$ is of degree $2n - 2$ and degree bound $2n - 1$. Therefore the point-value representation of $h(x)$ uses $2n - 1$ pairs on $2n - 1$ distinct values. The point-value representation of $f(x)$ and $g(x)$ uses n pairs on n distinct values. Therefore we will represent $f(x)$ and $g(x)$ in an extended form of $2n - 1$ point-value pairs for this representation to be useful in establishing $h(x)$'s point-value pair representation.

Theorem 12. Let $((x_0, a_0), (x_1, a_1), \dots, (x_{2n-1}, a_{2n-1}))$ be the point-value representation of $f(x)$ in extended form. Let similarly $((x_0, b_0), (x_1, b_1), \dots, (x_{2n-1}, b_{2n-1}))$ be the point-value representation of $g(x)$ in extended form. Then for $h(x) = f(x) * g(x)$ we have $h(x_i) = f(x_i) * g(x_i)$. Therefore the point-value representation of $h(x)$ is $((x_0, a_0 b_0), (x_1, a_1 b_1), \dots, (x_{2n-1}, a_{2n-1} b_{2n-1}))$, and can be obtained from those of $f(x)$, $g(x)$ by performing $2n - 1$ scalar multiplications only which is very efficient compared to performing a polynomial multiplication of $f(x)$ and $g(x)$ to obtain $h(x)$ in $O(n^2)$ operations and then evaluating $h(x)$ at $2n - 1$ points for an additional $O(n^2)$. Given this point-value representation of $h(x)$, $h(x)$ can be derived very easily by an additional interpolation operation.

Polynomial multiplication is Evaluation and Interpolation

Example 22. Consider the two ways to multiply two polynomials.

NaiveMultiply($f(x), g(x)$)	IntelligentMultiply($f(x), g(x)$)
1. $h(x) = f(x) * g(x);$	*1. Evaluate $f(x)$ at $2n$ points x_i .
2. return($h(x)$);	*2. Evaluate $g(x)$ at $2n$ points x_i .
	3. Pairwise multiply $f(x_i) * g(x_i)$
	4. Since $h(x_i) = f(x_i) * g(x_i)$ Interpolate to find $h(x)$
	5. return($h(x)$);

In the naive method step 1 requires $O(n^2)$ time.

In the intelligent method step 3 requires $O(n)$ time. If steps 1, 2, and 4 are performed naively they would all require $O(n^2)$ time and this method becomes less efficient than the naive method. If we choose, however, the x_i intelligently then, steps 1 and 2 can be performed in $O(n \lg n)$ time using a divide-and-conquer method, and step 4 becomes an operation similar to those of steps 1 and 2 and can also be performed in time $O(n \lg n)$. Therefore total running time for the Intelligent method becomes $O(n \lg n)$ compared to the $O(n^2)$ of the naive approach.

As a final note, whether $f(x)$ is evaluated at $2n$ points or $2n - 1$ is not that important. It helps the fact that $2n$ is an even number, and so some extra redundancy makes calculations easier.

Bit and Word model

Integer and Polynomial divide and conquer

Example 23 (Dividing an integer into two parts). Integer $a = 177_{10}$ has $n = 8$ bits. We can split a into two parts or halves, the most-significant half and the least-significant half. We can also call them leftmost and rightmost halves, or high and low halves. Thus using the latter terminology, $a = [10110001]$ becomes $a = a_H a_L$ where $a_H = [1011]$ and $a_L = [0001]$. Converting into denary, $a_H = 11_{10}$ and $a_L = 1_{10}$. How does a, a_L, a_H relate to each other? Well, $a = a_H \times 2^4 + a_L$. Since $a_H = [1011]$ then $a_H \times 2^4 = [10110000]$. Add to it the a_L term and the answer follows. In general, $a = a_H \times 2^{n/2} + a_L$, a has n -bit and a_H, a_L each has $n/2$ -bit.

Example 24 (Dividing a polynomial into two parts). Let $f(x)$ and $g(x)$ be two polynomials adjusted to the degree bound of the larger of the two. Let $f(x) = 1 + x^3 + x^6 + x^7$ with $F = [1, 0, 0, 1, 0, 0, 1, 1]$ Let $g(x) = 1 + x^2 + x^4$ with $G = [1, 0, 1, 0, 1, 0, 0, 0]$ to $n = 8$. We can split $f(x)$ in the high-half and low-half the same way we did this for integers. Likewise for $g(x)$. Then $F_L = [1, 0, 0, 1]$ and $F_H = [0, 0, 1, 1]$. $f_L = 1 + x^3$ and $f_H = x^2 + x^3$. Then $G_L = [1, 0, 1, 0]$ and $G_H = [1, 0, 0, 0]$. $g_L = 1 + x^2$ and $g_H = 1$. Moreover $f = f_L + x^4 f_H$ and $g = g_L + x^4 g_H$.

Complex numbers

Duality of complex numbers and polynomials

Definition 11 (Complex numbers). A complex number z is represented by $z = a + i * b$, where $a, b \in \mathbb{R}$ are real numbers and the symbol i is a number that is not a real number (it is called an imaginary number) with the property that $i^2 = -1$. Sometimes we represent $a + ib$ by the pair (a, b) .

Sometimes i is known as the square root of -1 since $i * i = i^2 = -1$. Adding or multiplying complex numbers is easy if you treat a complex number as a polynomial and its i as the indeterminate x . Thus $z = a + ib$ becomes a $z = a + x * b$ where $x = i$. The only difference later on involves multiplications of complex numbers. An $x * x$ does not become an x^2 but it resolves as $i * i = -1$.

Definition 12 (Addition of complex numbers). For two complex numbers $z_1 = a + ib$ and $z_2 = c + id$, $z_1 + z_2 = (a + c) + i(b + d)$, and it involves two scalar (real) additions/subtractions.

The additions are $(a + c)$ and $(b + d)$.

Definition 13 (Multiplication of complex numbers). The product of $z_1 = a + ib$ and $z_2 = c + id$ is $z_1 z_2 = (ac - bd) + i(ad + bc)$, and it involves two scalar additions/subtractions and four scalar multiplications.

Naturally $z_1 * z_2 = (a + ib) * (c + id)$. We do our $i = x$ conversion and then we polynomial (actually monomial) compute as follows. $(a + ib) * (c + id) = (a + xb) * (c + xd) = ac + bdx^2 + adx + bcx$. We then flip $x = i$ and continue. Naturally $z_1 * z_2 = (a + ib) * (c + id) = ac + bdi^2 + adi + bci$. At that point $i^2 = -1$ and therefore $z_1 * z_2 = (a + ib) * (c + id) = (ac - bd) + (ad + bc)i$. So complex number multiplication and polynomial multiplication are "equivalent".

Proposition 11. *Two complex numbers can be multiplied with only three scalar (real) multiplications (and five additions).*

Proof. The product of $z_1 = a + ib$ and $z_2 = c + id$ which is

$$z_1 * z_2 = (ac - bd) + i(ad + bc)$$

can be rewritten as

$$z_1 * z_2 = (ac - bd) + i[(a + b) * (c + d) - ac - bd]$$

We perform in in three scalar multiplications as follows $M_1 = a * c$, $M_2 = b * d$ and $M_3 = A_1 * A_2 = (a + b) * (c + d)$. The total number of additions/subtractions is the five including $A_1 = a + b$, $A_2 = c + d$, $A_3 = M_1 - M_2$, $A_4 = A_1 * A_2 - M_1$ and $A_5 = A_4 - M_2$. ■

Word Model Faster Polynomial (and Integer) Multiplication

The Karatsuba-Ofman algorithm for multiplying two degree-bound n polynomials requires $O(n^{\lg 3})$ operations, faster than the high school algorithm of that requires $O(n^{\lg 4}) = O(n^2)$ operations. It can be extended to n -bit integer multiplication but the indicated performance refers to bit operations then!

Theorem 13 (Karatsuba-Ofman integer multiplication). *In the bit model two n -bit integers can be multiplied in $T(n) = O(n^{\lg 3})$ bit operations.
(We assume scalar addition, subtraction, and shift operations are linear to the number of bit of the relevant integer(s) operations.)*

Theorem 14 (Karatsuba-Ofman polynomial multiplication). *In the word model two degree bound n polynomials can be multiplied in $T(n) = O(n^{\lg 3})$ time.
(We assume scalar addition, subtraction, multiplication, and shift operations are constant time operations.)*

Naive Divide and Conquer

Polynomial multiplication: preliminaries

Proposition 12. *The multiplication of two n -bit integers or two degree bound n polynomials using naive divide and conquer leads to $T(n) = O(n^{\lg 4}) = O(n^2)$ performance.*

Proof. (We show the result re the polynomial multiplication.) We refer to the preprocessing step outlined below. For two polynomials of degree bound n , $f(x)$ and $g(x)$ let F and G be their coefficient vectors. The length of F, G is n , the degree bound of f or g . For simplicity we assume that n is a power of two.

We can split F into F_L, F_H and G into G_L, G_H . The latter correspond to polynomial of half the degree bound of f or g , f_L, f_H, g_L, g_H . Based on prior discussion $f = f_L + x^{n/2}f_H$ and so is $g = g_L + x^{n/2}g_H$. For the product $f * g$ we have the following.

$$f * g = (f_L + x^{n/2}f_H) * (g_L + x^{n/2}g_H) = f_L * g_L + (f_H * g_L + f_L * g_H) \cdot x^{n/2} + f_H * g_H \cdot x^n$$

The multiplication $f * g$ involves two polynomials of degree bound n . Let the time for multiplication be $M(n)$.

The multiplication of $f_L * g_L$ and $f_H * g_L$ and $f_L * g_H$ and $f_H * g_H$ involve polynomials of degree bound $n/2$ in all cases. Let the time for all those four multiplications be $4M(n/2)$.

There are three addition operations. These involve polynomials whose degree bound never exceeds $2n$. This is because f and g are degree bound n i.e. their degrees are no more than $n - 1$, and thus the product $f * g$ has degree no more than $2n - 2$ and thus its degree bound is no more than $2n - 1$ i.e. at most $2n$. The collection running time of those additions is $3\Theta(n)$. (The 3 is to signify the number of additions involved and serves no other purpose.)

Moreover there are two $\cdot$ multiplications one by $x^{n/2}$ and the other by x^n . These are $SHL(., n/2)$ and $SHL(., n)$ operations. Each one and collectively both of them are linear thus contributing $2\Theta(n)$.

Naive Divide-and-Conquer Analysis. Overall

$$M(n) = 4M(n/2) + 5\Theta(n) = 4M(n/2) + \Theta(n)$$

This divide-and-conquer approach is not better than the high school method. Master method gives $M(n) = \Theta(n^{\lg 4})$ i.e.

$$M(n) = \Theta(n^2).$$

■

Karatsuba-Ofman (polynomial version)

Divide and conquer that saves time

Proposition 13. *The multiplication of two degree bound n polynomials using Karatsuba-Ofman divide and conquer leads to $T(n) = O(n^{\lg 3})$ performance.*

Proof (Turning a $\lg 4$ exponent into a $\lg 3$). To improve performance we aim at reducing the 4 of $M(n/2)$ into a 3. For this we observe the following.

$$f * g = f_L * g_L + ((f_H + f_L) * (g_H + g_L) - f_H \cdot g_H - f_L \cdot g_L) \cdot x^{n/2} + f_H * g_H \cdot x^n$$

The number of $*$ multiplications has been reduced by one although the number of auxiliary operations has increased a lot. The auxiliary operations are linear however. Note that $f_H \cdot g_H$ and $f_L \cdot g_L$ are not multiplications but cut-and-paste (or memory copies) of results first computed for $f_L * g_L$ and $f_H * g_H$. There are two SHL operations as before, and six more addition/subtraction operations. Overall.

Karatsuba-Ofman Analysis. Overall

$$M(n) = 3M(n/2) + 10\Theta(n) = 3M(n/2) + \Theta(n)$$

A solution of the recurrence by the master method yields

$$M(n) = \Theta(n^{\lg 3}) = \Theta(n^{1.58496}).$$

■

Remark 2. Java's BigInteger uses a variant of Karatsuba-Ofman for integer multiplication. This beats the naive method for $n > 100 - 200$ usually.

Karatsuba-Ofman (integer version)

Divide and conquer that saves time

Proposition 14. *The multiplication of two n -bit integers using Karatsuba-Ofman divide and conquer leads to $T(n) = O(n^{\lg 3})$ performance.*

Proof (From a $\lg 4$ exponent into a $\lg 3$). 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_H, a_L be the integer representation of these $n/2$ bits. Then $a = a_H 2^{n/2} + a_L$ and similarly $b = b_H 2^{n/2} + b_L$. Then

$$ab = (a_H 2^{n/2} + a_L)(b_H 2^{n/2} + b_L) = a_H b_H 2^n + (a_H b_L + a_L b_H) 2^{n/2} + a_L b_L = a_H b_H 2^n + ((a_H + a_L)(b_L + b_H) - a_L b_L - a_H b_H) 2^{n/2} + a_L b_L$$

```
KaratsubaOfman(a,b)
1. (a_H a_L) =a;
2. (b_H b_L) =b;
3. t1 = KaratsubaOfman((a_H + a_L ),(b_H + b_L));
4. t2 = KaratsubaOfman(a_H , b_H);
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 to 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_H, b_H, a_L, b_L . We then perform three multiplications, one is $a_L b_L$ (line 5), the other $a_H b_H$ (line 4) and the third is the $((a_H + a_L)(b_H + b_L))$ (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 (integer version)

Some details

Note 2. The integers $(a_H + a_L)$ and $(b_H + b_L)$ may not be $n/2$ bit integers any more when step 3 of KaratsubaOfman 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 $a_H + a_L$ as follows. $a_H + a_L = a_h 2^{n/2} + a_1$ (similarly for $b_H + b_L$). Note that $a_H + a_L$ 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 $(a_H + a_L)(b_H + b_L) = a_h b_h 2^n + (a_h b_1 + b_h a_1) 2^{n/2} + b_1 a_1$. 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.

$$ab = (a_H 2^{n/2} + a_L)(b_H 2^{n/2} + b_L) = a_H b_H 2^n + (a_H b_L + a_L b_H) 2^{n/2} + a_L b_L = a_H b_H 2^n + ((a_H - a_L)(b_L - b_H) + a_L b_L + a_H b_H) 2^{n/2} + a_L b_L$$

The ordering of the $-$ terms might need to change though!

Example 25.

$a = 1234$ $aH = 12$, $aL=34$ $|a|=4$ $|aH|=|aL|=2$
 $b = 5678$ $bH = 56$, $bL=78$ $|b|=4$ $|bH|=|bL|=2$

STEP 1: $A=[4,3, 2,1] = [AL \ AH]$ $a = aH*10^{n/2} + aL$

$B=[7,8, 6,5] = [GL \ GH]$ $b = bH*10^{n/2} + bL$

$a = aH * 10^{n/2} + aL$ $b = bH * 10^{n/2} + bL$

[if binary $|a|=|b|=n$ bit]

$a = aH * 2^{n/2} + aL$ $b = bH * 2^{n/2} + bL$

Let $n= 8$ $|a|=|b|=n$, $|aH|=|aL|=|bH|=|bL|=n/2$

$M(n)$: time to multiply two n digit (or n bit) integers

$M(n/2)$: time to multiply two $n/2$ digit (or $n/2$ bit) integers

$$a * b = [aL + 10^{n/2} * aH] * [bL + 10^{n/2} * bH] =$$

$$1234 * 5678$$

$| = (aH * bH) . 10^n + (aH * bL + aL * bH) . 10^{n/2} + (aL * bL)$

$| = (12 * 56) . 10^n + (12 * 78 + 34 * 56) . 10^{n/2} + (34 * 78)$

$| \quad \quad \quad / \quad \quad \quad | \quad \quad \quad / \quad \quad \quad | \quad \quad \quad / \quad \quad \quad | \quad \quad \quad /$

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

$M(n) = 4 M(n/2) + 5 \Theta(n) \Leftrightarrow M(n) = \Theta(n^2)$

Karatsuba-Ofman

Example: Conquered

Example 26.

$M(n)$: time to multiply two n digit (or n bit) integers
 $M(n/2)$: time to multiply two $n/2$ digit (or $n/2$ bit) integers

$$\begin{aligned}
 & \begin{array}{ccccccc}
 n & & n & & n/2 & & n/2 & & n/2 & & n/2 \\
 a * b = & [& a_L & +10^{**n/2} * & a_H &] * [& b_L & +10^{**n/2} * & b_H &] = \\
 & | & & & & & & & & & \\
 & | & = & (a_H * b_H) & .10^{**n} + & (a_H * b_L + & a_L * b_H) & .10^{**n/2} + & (a_L * b_L) \\
 & | & = & (a_H*b_H)10^{**n} + & [(a_H+a_L)* & (b_H+b_L)-a_Hb_H-a_Lb_L)10^{**n/2} + & (a_L * b_L) \\
 & | & = & (12*56)10^{**4} + & [(46) * & (1 34) - 672-2652)10^{**4/2} + & (34 * 78) \\
 & | & & / & | & & / & | & | & | & | & | & / \\
 & | & = & (a_H*b_H)10^{**n} + & [-(a_H-a_L)* & (b_H-b_L)-a_Hb_H-a_Lb_L)10^{**n/2} + & (a_L * b_L) \\
 & | & = & (12*56)10^{**4} + & [-(22) * & (22) - 672-2652)10^{**4/2} + & (34 * 78) \\
 & | & & / & | & & / & | & | & | & | & | & / \\
 M(n) & = & M(n/2) + 3 * & \Theta(n) + M(n/2) + 4 * & \Theta(n) + 3 * & \Theta(n) + M(n/2) <=> \\
 M(n) & = & 3 M(n/2) + 10 \Theta(n) <=> M(n) = \Theta(n^{**\lg(3)})
 \end{array}
 \end{aligned}$$

Example 27.

```

f(x) = 1 + x**3 + x**6 + x**7    d(f) = 7 , db(f)=8    F: [1,0,0,1,0,0,1,1]
g(x) = 1 + x**2 + x**4           d(g) = 4 , db(g)=5    G: [1,0,1,0,1      ]
                                Pad with zeroes    G: [1,0,1,0,1,0,0,0]

STEP 1:  F=[1001 0011] = [FL FH]    FL = [1001]    fl = 1 + x**3
                                FH = [0011]    fh = x**2+x**3
                                G=[1010 1000] = [GL GH]    GL = [1010]    gl = 1 + x**2
                                GH = [1000]    gh = 1
f(x) = fh * x**n/2 + fl          g(x) = gh * x**n/2 + gl

Let n= 8    d(f) , d(g)  <= n-1      d(fl) , d(fh) , d(gl) , d(gh)  <= n/2 -1
          db(f), db(g) <=n          df(fl) , df(fh) , db(gl) , db(gh) <= n/2 -1
M(n) : time to multiply two degree bound n (atmost) polynomials
M(n/2): time to multiply two degree bound n/2 (atmost) polynomials
      n      n      n/2          n/2      n/2          n/2
f * g = [ fl + x**n/2 * fh ] * [ gl + x**n/2 * gh ] =
      |      = (fh * gh) . x**n + (fh * gl  +  fl * gh) . x**n/2+ (fl * gl)
      |          /      |      |      /      |      /      |      |      /
M(n)   = M(n/2)+ 2Theta(n)+M(n/2)+Theta(n)+M(n/2)+2Theta(n)+ M(n/2) <=>
M(n)   = 4 M(n/2) + 5 Theta(n) <=> M(n) = Theta( n**2)

```

Karatsuba-Ofman Example: Conquered

Example 28.

$M(n)$: time to multiply two degree bound n (atmost) polynomials

$M(n/2)$: time to multiply two degree bound $n/2$ (atmost) polynomials

$$\begin{aligned}
 & \begin{array}{ccccccc}
 n & & n & & n/2 & & n/2 \\
 f * g = & [f_l + x^{n/2} * f_h] & * & [g_l + x^{n/2} * g_h] & = & & \\
 & | & = & (f_h * g_h) \cdot x^n + (f_h * g_l + f_l * g_h) \cdot x^{n/2} + (f_l * g_l) \\
 & | & = & (f_h * g_h) \cdot x^n + [(f_h + f_l) * (g_h + g_l) - f_h g_h - f_l g_l] \cdot x^{n/2} + (f_l * g_l) \\
 & | & & / & | & & / \\
 M(n) & = & M(n/2) + 3 * \Theta(n) + M(n/2) + 4 * \Theta(n) + 3 * \Theta(n) + M(n/2) <=> \\
 M(n) & = & 3 M(n/2) + 10 \Theta(n) <=> M(n) = \Theta(n^{\lg(3)})
 \end{array}
 \end{aligned}$$

Karatsuba-Ofman Note for integer multiplication

Given the duality of polynomial and integers, the discussion of the previous pages also applies to integer multiplication.

In integer multiplication however, we can have the following abnormality that is not present with polynomials.

(In the discussion below, f, g become n bit integers and thus f_H, f_L, g_H, g_L become $n/2$ bit integers.)

For polynomial f, g , the additions $(f_H + f_L)$ and $(g_H + g_L)$ do not change the degree of the sums relative to the degree of the constituent terms.

$$((f_H + f_L) * (g_H + g_L) - f_H \cdot g_H - f_L \cdot g_L)$$

Thus if f_H, f_L, g_H, g_L are of degree bound (at most) $n/2$ so are $(f_H + f_L)$ and $(g_H + g_L)$.

For integer f, g however, this preservation might fail. Thus if f_H, f_L, g_H, g_L become $n/2$ bit integer, then $(f_H + f_L)$ and $(g_H + g_L)$ have the potential of becoming $n + 1$ bit integer.

Thus in the recurrence for the running time the contribution of the multiplication $(f_H + f_L) * (g_H + g_L)$ could become (or should become) $M(n/2 + 1)$ instead of $M(n/2)$.

If we want to avoid this nuance, we could resort to a variety of tricks. One of them is shown below.

Write $f_H + f_L$ as follows. $f_H + f_L = f_h 2^{n/2} + f_l$ (similarly for $g_H + g_L$). Note that $f_H + f_L$ can be represented by up to $n/2 + 1$ bits: f_h is either a 0 or 1 i.e. a bit. Then the product $(f_H + f_L) * (g_H + g_L) = f_h g_h 2^n + (f_h g_l + g_h f_l) 2^{n/2} + f_l * g_l$. The product $f_h g_h$ is easy to find as it involves two 1-bit numbers. So are the products $f_h g_l$ (the product is either g_l or 0 depending on whether f_h is 1 or 0) and $g_h f_l$ (either f_l or 0). The only complication arises from $f_l g_l$: both numbers are $n/2$ bit integers and the product is a Karatsuba-Ofman multiplication of two $n/2$ -bit integers. The fix thus consists of one $n/2$ -bit multiplication plus $O(n)$ bit operations but the third multiplication is the one involving $f_l * g_l$ rather than $(f_H + f_L) * (g_H + g_L)$.

Polynomials

Interpolation and secret sharing

Definition 14 (Secret sharing). There is a group of N people who want to share a numeric secret s in such a way that the presence of at least n of the N people would be required to recover the value of s .

The scheme below is not just for sharing secrets. It can be used to redundantly store information. Think of having N hard disk-drives (or SSD) and of a record s of a file. We would like s to be stored redundantly so that even if $N - n$ of the hard disk-drives fail redundancy would allow the recovery of s from the remaining n hard disk-drives.

Theorem 15 (Shamir's scheme). *There exists a very simple secret sharing scheme of low computational complexity.*

Proof. The idea is to split secret s into N pieces $s_1, \dots, s_N$ so that a quorum of n pieces is needed to retrieve s . Choose randomly a polynomial of degree $n - 1$. Let $f(x)$ be such a polynomial.

$$f(x) = a_{n-1}x^{n-1} + \dots + a_1x + s$$

Choosing a polynomial randomly is equivalent to choosing its $n - 1$ coefficients $a_1, \dots, a_{n-1}$ randomly. The 0-th degree coefficient will be our secret s . The i -th piece of s is $s_i = f(i)$, $1 \leq i \leq N$. Piece s_i is assigned to person i . Polynomial $f(x)$ satisfies the requirements of secret sharing. Suppose a quorum of n is gathered. Let these be $j_1, j_2, \dots, j_n$. Then we have n point-value pairs $(j_i, f(j_i) = s_{j_i})$ of polynomial $f(x)$ and therefore by Lagrange's formula we derive that $f(x) = g(x)$, where

$$g(x) = b_1g_1(x) + \dots + b_ng_n(x)$$

In order to determine the secret s we evaluate $g(x)$ as found by Lagrange's formula at $x = 0$. Then, $s = g(0) = f(0)$. ■

Note 3. In practice the polynomial $f(x)$ is constructed over a finite field. What this means is that s , N , and all important parameters (e.g. a_i) are values up to say a prime number P and all computations are mod P .

Word model Matrices

In the next few pages we review some definitions from matrix theory. A matrix is a rectangular array of numbers. For example, A is a 3×4 matrix, 3 indicating the number of rows and 4 the number of columns.

$$A = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & 2 & 4 & 8 \\ 1 & 3 & 9 & 27 \end{pmatrix}$$

In A , a_{ij} represents the entry in row i and column j . The transpose of A denoted by A^T is A with rows becoming columns and columns becoming rows.

$$A^T = \begin{pmatrix} 1 & 1 & 1 \\ 1 & 2 & 3 \\ 1 & 4 & 9 \\ 1 & 8 & 27 \end{pmatrix}$$

A (column) vector x is an $n \times 1$ matrix. A row vector x^T is an $1 \times n$ matrix. A unit vector e_i is one whose i -th row is 1 and all other entries 0. A zero matrix is a matrix with all elements zeros. A square matrix is an $n \times n$ matrix.

$$x = \begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix}, x^T = (1 \ 2 \ 3).$$

Special cases of matrices are (a) a diagonal (only a_{ii} elements are non-zero), (b) the identity matrix whose i -th column is e_i , $i = 1, \dots, n$, (c) a tridiagonal where $a_{ij} = 0$ for $|i - j| > 1$, (d) an upper triangular ($a_{ij} = 0, i > j$) and a lower triangular ($a_{ij} = 0, i < j$), (e) a permutation matrix whose entries are zero except for a single one in every row and column.

A matrix A is symmetric if and only if $A = A^T$.

Word Model

Matrix addition, subtraction

Definition 15 (Matrix addition). For two matrices A and B of dimension $n \times m$, matrix C of dimension also $n \times m$ is called the sum of A and B and we write $C = A + B$ if and only if $c_{ij} = a_{ij} + b_{ij}$.

Theorem 16 (Matrix addition (word model)). *If A, B, C are $n \times m$ matrices, then addition requires nm additions and thus $O(nm)$ time.*

Definition 16 (Matrix subtraction). For two matrices A and B of dimension $n \times m$, matrix C of dimension also $n \times m$ is called the difference of A and B and we write $C = A - B$ if and only if $c_{ij} = a_{ij} - b_{ij}$.

Theorem 17 (Matrix subtraction (word model)). *If A, B, C are $n \times m$ matrices, then subtraction requires nm subtractions and thus $O(nm)$ time.*

Definition 17 (Matrix multiplication). For two matrices A, B , matrix $C = A \times B$ is defined as long as the column size of A equals the row size of B , i.e. if A is an $n \times m$ and B an $m \times p$ matrix, then C is an $n \times p$ matrix such that $c_{ij} = \sum_{k=1}^m a_{ik}b_{kj}$.

Theorem 18 (Matrix multiplication (word model)). If A, B, C are $n \times m, m \times p$ and $n \times p$ matrices, then multiplication $C = A \times B$ requires nmp multiplications and $np(m - 1)$ additions.

DRAFT

Time complexity of matrix operations

Addition	:	A,B:nxm	A+B : nxm	Ops : n*m additions
Addition	:	A,B:nxm	A+B : nxm	Time: Theta (nm)
Addition	:	A,B:nxn	A+B : nxn	Time: Theta (n*n)
Subtraction:		A,B:nxm	A+B : nxm	Ops : n*m subtractions
Subtraction:		A,B:nxm	A-B : nxm	Time: Theta (nm)
Subtraction:		A,B:nxn	A-B : nxn	Time: Theta (n*n)
Multiplication:	A : nxm,	C= A*B : nxp	Ops : n*p*m	multiplications
[Binet's method]	B : mxp,		n*p*(m-1)	additions
	C : nxp			
Multiplication:	A : nxm,	C= A*B : nxp	Time: Theta (n*m*p)	
[Binet's method]	B : mxp,			
	C : nxp			
Multiplication:	A : nxn,	C= A*B : nxn	Ops : n*n*n	multiplications
[Binet's method]	B : nxn,		n*n*(n-1)	additions
	C : nxn			
Multiplication:	A : nxn,	C= A*B : nxn	Time: Theta (n*n*n)	
[Binet's method]	B : nxn,			
	C : nxn			

Optional: Matrix Multiplication

Winograd's method

Theorem 19 (Winograd). *Winograd's method for matrix multiplications yields $n^3/2 + n^2$ M's and $3n^3/2 + n^2 + n^2 - 2n = 3n^3/2 + 2n^2 - 2n$ A's.*

Proof. Let us consider again the case of matrix multiplication for two $n \times n$ square matrices A and B and let n be even. Consider the vector product of two vectors $a = (a_1, \dots, a_n)$, and $b = (b_1, \dots, b_n)$.

$$\begin{aligned} ab &= a_1b_1 + \dots + a_nb_n \\ &= ((a_1 + b_2)(a_2 + b_1) + (a_3 + b_4)(a_4 + b_3) + \dots + (a_{n-1} + b_{n-1})(a_n + b_n)) \\ &\quad - (a_1a_2 + a_3a_4 + \dots + a_{n-1}a_n) - (b_1b_2 + b_3b_4 + \dots + b_{n-1}b_n). \end{aligned}$$

Step 1. We precompute the $a_1a_2 + \dots + a_{n-1}a_n$ once per row of A and similarly $b_1b_2 + b_3b_4 + \dots + b_{n-1}b_n$ once per column of B . We have $2n$ such rows/columns of A and B . For each such product-sum we perform $n/2$ M's (short for scalar multiplications) and $n/2 - 1$ A's (short for additions). The total preprocessing count is thus $2nn/2 = n^2$ M's and $2n(n/2 - 1) = n^2 - 2n$ A's.

Step 2. The preprocessing is performed once. In order to find the product $A \times B$ we need to find n^2 vector-products involving rows of A and columns B using the information collected in the preprocessing step. This is equivalent to performing the following computation, where a', b' are pieces of preprocessed information obtained in Step 1. For each one of n^2 vector products we perform: $n/2$ M's for the products like $(a_1 + b_2)(a_2 + b_1)$, n A's for the additions (each of $n/2$ products like $(a_1 + b_2)(a_2 + b_1)$ requires two additions), $n/2 - 1$ additions to combine the products, and 2 A's (in fact subtractions) of the a', b' . The total count is thus $n/2$ M's and $n + (n/2 - 1) + 2 = 3n/2 + 1$ A's per single vector product. The total count is thus $n^2n/2 = n^3/2$ M's and $(3n/2 + 1)n^2$.

$$ab = ((a_1 + b_2)(a_2 + b_1) + (a_3 + b_4)(a_4 + b_3) + \dots + (a_{n-1} + b_{n-1})(a_n + b_n)) - a' - b'$$

The overall cost is the sum of the two individual ones of steps 1 and 2. This yields $n^3/2 + n^2$ M's and $3n^3/2 + n^2 + n^2 - 2n = 3n^3/2 + 2n^2 - 2n$ A's. ■

Matrix Multiplication

Divide and conquer

Theorem 20 (Blocked Binet's method). *For n a power of two, two matrices A, B of geometry $n \times n$ can be multiplied in time*

$$T(n) = O(n^3).$$

Proof. Suppose we view matrix multiplication not the traditional way but devise for it a divide and conquer algorithm. Let us assume we are given an $n \times n$ matrix A , and let us assume without loss of generality that n is a power of two. Then we can view A as consisting of four $n/2 \times n/2$ matrices $A_{12}, A_{11}, A_{21}, A_{22}$. Then the product of two matrices A and B can be viewed as a blocked matrix product.

$$C = A \times B = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \times \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix} = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}$$

where, $C_{11} = A_{11}B_{11} + A_{12}B_{21}$, $C_{12} = A_{11}B_{12} + A_{12}B_{22}$, $C_{21} = A_{21}B_{11} + A_{22}B_{21}$, $C_{22} = A_{21}B_{12} + A_{22}B_{22}$. It is clear that for the calculation of C , 8 $n/2 \times n/2$ multiplications are required and 4 additions. Each addition requires $n^2/4$ additions for a total of n^2 operations. Then the total number of operations (additions and multiplications).

$$T(n) = 8T(n/2) + n^2$$

By the master method we derive that $T(n) = \Theta(n^3)$. ■

Matrix Multiplication

Strassen's divide and conquer

In the late 60s Strassen reformulated the calculations for C so that only 7 matrix multiplications (including its additions), and 18 additions and subtraction were required. The following was then derived.

Theorem 21 (Strassen). *For n a power of two, two matrices A, B of geometry $n \times n$ can be multiplied in time $T(n) = O(n^{\lg 7}) = O(n^{2.807})$ using Strassen's method, that uses 7 multiplications of $n/2 \times n/2$ matrices and 18 additions and subtractions.*

Proof. Strassen's original method involved 18 additions thus giving a recurrence of

$$T(n) = 7T(n/2) + 18n^2/4$$

for matrix multiplication. For Strassen's method to work as claimed n must be a power of two and the base case is for $n = 2$. The number of operations required to multiply two 2×2 matrices is given either by the traditional method (known also as Cannon's method) of 12 operations (8 multiplications, 4 additions) or Strassen's method itself of 25 operations (7 multiplications, and 18 additions).

$$\begin{aligned} m_1 &= (A_{12} - A_{22}) * (B_{21} + B_{22}) \\ m_2 &= (A_{11} + A_{22}) * (B_{11} + B_{22}) \\ m_3 &= (A_{11} - A_{21}) * (B_{11} + B_{12}) \\ m_4 &= (A_{11} + A_{12}) * B_{22} \\ m_5 &= A_{11} * (B_{12} - B_{22}) \\ m_6 &= A_{22} * (B_{21} - B_{11}) \\ m_7 &= (A_{21} + A_{22}) * B_{11} \\ C_{11} &= m_1 + m_2 - m_4 + m_6 \\ C_{21} &= m_6 + m_7 \\ C_{12} &= m_4 + m_5 \\ C_{22} &= m_2 - m_3 + m_5 - m_7. \end{aligned}$$

The solution of this recurrence using the master method is $T(n) = O(n^{\lg 7}) = O(n^{2.807})$. ■

Matrix Multiplication Beyond Strassen

Corollary 9. For n a power of two, two matrices A, B of geometry $n \times n$ can be multiplied in time $T(n) = O(n^{\lg 7}) = O(n^{2.807})$ using a variation of Strassen's method, that uses 7 multiplications of $n/2 \times n/2$ matrices and 15 additions and subtractions.

Proof. Thus the 18 of the recurrence that analyzed Strassen's original schedule can become a 15. Then

$$T(n) = 7T(n/2) + 15n^2/4$$

still has the same solution asymptotically. ■

The algorithm can be extended for non power of two geometry.

There has been much activity in reducing the exponent of matrix multiplication to something close to 2. The table below shows the progress in reducing Strassen's matrix multiplication exponent from $t = 2.80\dots$ indicating a running time of $T(n) = O(n^t)$.

Matrix Multiplication Exponent		
1812	Binet	$t=3$
1969	Strassen	$t=2.807\dots$
1990	Coppersmith-Winograd	$t=2.376$
2013	Strothers	$t=2.3737$
2013	Williams	$t=2.3729$
2014	Le Gall	$t=2.3728639$

Method 1 (Dynamic Programming). The development of a dynamic programming algorithm is broken into a sequence of steps.

- (1) Characterize the structure of an optimal solution.
- (2) Recursively define the value of an optimal solution.
- (3) Compute the value of an optimal solution in a bottom-up fashion.
- (4) Construct an optimal solution from computed information.

We have already used dynamic programming in the context of all pairs-shortest paths, transitive closure and also early in the semester for the Fibonacci sequence problem.

We are going to use dynamic programming techniques in solving the following problem.

Word model

Nested matrix multiplication

The nested matrix multiplication problem is also known as the matrix chain product, chained matrix multiplication, and also known as iterated matrix multiplication.

Problem 17 (Nested matrix multiplication).

Input: Dimensions $d_0, d_1, \dots, d_{n-1}, d_n$ corresponding to the dimensions of matrices involved in the calculation of

$$M = M_{0n-1} = M_0 \times M_1 \times \dots \times M_{n-1}$$

where M_i is an $d_i \times d_{i+1}$ matrix. (Note that the number of dimensions utilized is $(n + 1)$.)

Output: An (implied) schedule for the multiplication of $M = M_{0n-1} = M_0, M_1, \dots$ to determine the product M_{0n-1} using the minimal number of scalar arithmetic operations (i.e. scalar multiplications). The output is the **minimal number of scalar multiplications** used, and (optionally) the multiplication schedule.

Note that matrix multiplication is an associative operation i.e. $A \times (B \times C) = (A \times B) \times C$, but not necessarily commutative i.e. $B \times C$ might not be equal to $C \times B$. Dynamic programming, like the divide-and-conquer method, solves problems by dividing a problem into subproblems, solving the subproblems, and combining the solutions of subproblems. In divide and conquer subproblems are usually independent of each other. Dynamic programming however is used in cases where subproblems are not independent. A dynamic programming algorithm solves subproblems once, saves the solutions in a table and avoids recomputing the answer for a problem that has already been solved before.

Word model Nested matrix multiplication

Example 29.

res: resulting product's geometry/shape
ops: # of Ms to compute resulting product

$$\begin{array}{lcl}
 10 \times 20 & 20 \times 2 & 2 \times 30 \\
 M1 \times M2 \times M3 & = & (M1 \times M2) \times M3 = M1 \times (M2 \times M3) \\
 & | & / & | & / \\
 & \text{ops: } 10 \times 20 \times 2 & / & | & \text{ops: } 20 \times 2 \times 30 \\
 & = 400 & / & | & = 1200 \\
 & \text{res: } 10 \times 2 & / & | & \text{res: } 20 \times 30 \\
 & | & / & | & / \\
 & | & / & | & / \\
 & \text{ops: } 10 \times 2 \times 30 & / & | & \text{ops: } 10 \times 20 \times 30 \\
 & = 600 & / & | & = 6000 \\
 & \text{res: } 10 \times 30 & / & | & \text{res: } 10 \times 30 \\
 \text{Total ops} & = 400 + 600 = 1000 & & \text{Total ops} & = 1200 + 6000 = 7200
 \end{array}$$

Theorem 22 (Nested matrix multiplication). *Dynamic Programming allows for the solution of the nested matrix multiplication problem in $O(n^3)$ time, where n is the number of matrices of the chain products and also $n + 1$ is the number of dimensions involved. The problem involves finding the best cost schedule to compute M defined as follows.*

$$M = M_{0n-1} = M_0 \times \dots \times M_{n-1},$$

where M_i is an $d_i \times d_{i+1}$ matrix. The cost of multiplying two matrices of geometry $m \times n$ and $n \times p$ is defined as mnp which is the number of scalar multiplications involved.

Proof. Towards this we will also compute the (cost of the) best schedule for all $M_{ij} = M_i \times \dots \times M_j$, i.e. the minimal or optimal cost. Moreover, note that M_{ii} would denote the trivial product $M_{ii} = M_i$, and M_{i+1} the following product $M_{i+1} = M_i \times M_{i+1}$.

In order to compute the optimal/minimal cost of M_{ij} we will examine exhaustively for every k that lies between i and j the costs associated with all calculations for picking a given k i.e. all schedules

$$M_{ij} = M_{ik} \times M_{k+1j},$$

for $i \leq k < j$ will be examined.

This way, a larger problem (longer chain of matrices) would be split into smaller subproblems (shorter chains), and in turn optimal solutions would be sought for those problems as well. The cost of the optimal solution, i.e. optimal/minimal cost, for M_{ij} would be maintained in variable $N_{i,j}$. Moreover retrieving the schedule of the optimal multiplication would require that the index k for the best/minimal cost of M_{ij} be stored into an auxiliary variable s_{ij} .

Therefore in an optimal computation of M_{ij} , the best schedules for M_{ik} and M_{k+1j} are first computed and then the best schedule for M_{ij} as the product of the two partial results is derived for the optimal k .

Given that i and j are indices and describe multiple subproblems, both N and s would be matrices of dimension $n \times n$.

Proof continued.

Step-1. $M_{ii} = M_i$. Establish $N_{i,i}$ for all $\forall i : 0 \leq i \leq n - 1$. ($s_{i,i} = i$.)

$$N_{i,i} = 0 \text{ for all } 0 \leq i < n.$$

Step-2. $M_{i,i+1} = M_i \times M_{i+1}$. Establish $N_{i,i+1}$ and $s_{i,i+1}$ for all $\forall i : 0 \leq i < n - 1$.

$$N_{i,i+1} = d_i d_{i+1} d_{i+2} \text{ and } s_{i,i+1} = i, \text{ for all } 0 \leq i < n - 1.$$

Step-3. For generic M_{ij} calculate $N_{i,j}$ and store also information re $s_{i,j}$. You may substitute (to follow the textbook faithfully) $j = i + b$ in this calculation.

$$N_{i,j} = \min_{i \leq k < j} \{N_{i,k} + N_{k+1,j} + d_i d_{k+1} d_{j+1}\} \text{ for all } 0 \leq i < n - 1 \text{ and } i < j < n \text{ with}$$

$$s_{ij} = \text{value of } k \text{ that obtains the above minimum}$$

$$N_{i,i+b} = \min_{i \leq k < i+b} \{N_{i,k} + N_{k+1,i+b} + d_i d_{k+1} d_{i+b+1}\} \text{ for all } 0 \leq i < n - b \text{ and } 0 < b < n \text{ with}$$

$$s_{i,i+b} = \text{value of } k \text{ that obtains the above minimum}$$

■

Nested matrix multiplication

Pseudocode

```
NestedMatrixMult(d[0..n-1],n)
1. for(i=0;i< n;i++)
2.   N[i,i]=0;
3.   for(i=0;i<n-1;i++)
4.     s[i,i+1]=i;
5.     for(i=0;i<n-1;i++)
6.       N[i,i+1]= d[i]*d[i+1]*d[i+2];
7.       for(b=1;b<n;b++) {
8.         for(i=0;i<n-b;i++) {
9.           N[i,i+b] = infinity; // j=i+b
10.          for(k=i;k<i+b;k++) {
11.            temp=N[i,i+b]; // N[i,j]                j=i+b
12.            //N[i,j] =
13.            N[i,i+b]= MIN{ N[i,i+b],
14.                           N[i,k]+N[k+1,i+b]+d[i]*d[k+1]*d[i+b+1] };
15.            if (N[i,i+b] < temp)
16.              s[i,i+b] = k;
17.          }
18.        }
19.      }
```

There are three nested loops. These determine the $O(n^3)$ in performance.
(See textbook pages 274-277.)

Nested matrix multiplication Example

Example 30. Let us have 5 matrices $M_0 - M_4$ of dimensions 10×5 , 5×1 , 1×10 , 10×2 , 2×10
i.e. $d_0=10$, $d_1=5$, $d_2=1$, $d_3=10$, $d_4=2$, $d_5=10$

		j=i+b				
		0	1	2	3	4
i	0	0	50	150	90	190
	1		0	50	30	90
	2			0	20	40
	3				0	200 <-----N[i,i+1]
	4					0 <-----N[i,i]

Proof. Example Calculation

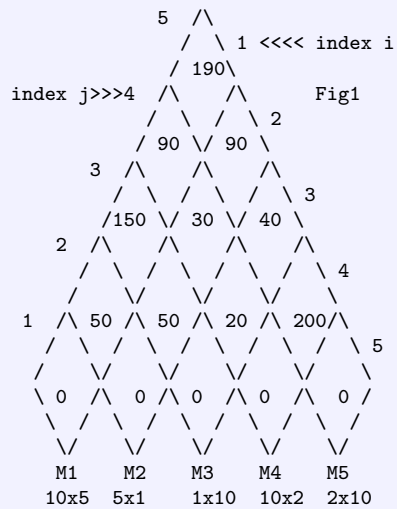
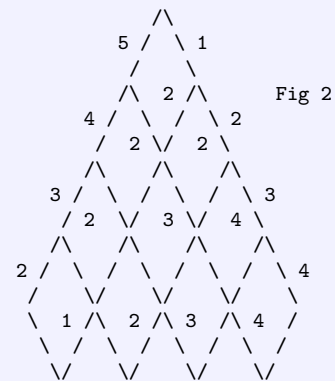
- Diagonal elements are 0 by $N[i,i]$ initialization
- Elements over diagonal are computed by

$$N[i,i+1] = d_i d_{i+1} d_{i+2}$$
- Elements below the diagonal are irrelevant
- Consider $N[0,2]$ (the entry that will become 150);
 This is $N[i,j] = N[i,i+b]$
- Its entry is determined by elements of the same row and column
- One choice is $M[0,0] * M[1,2]$ with $N[0,0]=0$ $N[1,2]=50$
- Another choice is $M[0,1] * M[2,2]$ with $N[0,1]=50$ $N[2,2]=0$
- First choice gives $0 + 50 + 10 \times 5 \times 10 = 550$
- Second choice gives $50 + 0 + 10 \times 1 \times 10 = 150$
- The smallest of the two and original value (infinity) is 150
- Therefore so far $(M_0 \times M_1) \times M_2$ is the way to go.
- For an element $N[i,j]$
- the element of the same row used is $N[i,k]$ and d_i
- the element of the same column used is $N[k+1,j]$ and d_{k+1} and d_{j+1}



Nested matrix multiplication Example

Example 31. Same example as before different structure



1. STANDARD VIEW

2. ALTERNATIVE VIEW

		1	2	3	4	5
		0	50	150	90	190
1						
2						
3						
4						
5						

j=i+s

i

0 <-----c[i,i]

200 <-----c[i,i+1]

Nested matrix multiplication

Example continued

Example 32.

	M0	x	M1	x	M2	x	M3	
	10x5		5x1		1x10		10x2	
	0		1		2		3	

0	0		50_0		150_1		90_1	<----M03 best cost is 90 Schedule (M0 x M1) (M2 x M3)

1	void		0		50_1		30_1	M01=M0 x M1 : N01 = 10x5x1=50 [0] M12=M1 x M2 : N12 = 5x1x10=50 [1] M23=M2 x M3 : N23 = 1x10x20=20 [2]

2	void		void		0		20_2	M02=(M0xM1xM2): N02 = min(M00 N00 + M12 N12 + M00xM12 10x5x10, M01 N01 + M22 N22 + M01xM22 10x1x10)

3	void		void		void		0	

								More on next page = 150_1

Nested matrix multiplication

Example continued

Calculations. With reference to the figure the main diagonal elements are 0. The diagonal over the main diagonal contains $d_i d_{i+1} d_{i+2}$ products.

In order to fill the third diagonal (that will become 150, 30, 40). we have $N[0, 2] = \min\{N[0, 0] + N[1, 2] + 10 \cdot 5 \cdot 10, N[0, 1] + N[2, 2] + 10 \cdot 1 \cdot 10\} = \min\{0 + 50 + 500, 50 + 0 + 100\} = 150$ and record $s[0, 2] = 1$. Similarly $N[1, 3] = \min\{N[1, 1] + N[2, 3] + 5 \cdot 1 \cdot 2, N[1, 2] + N[3, 3] + 5 \cdot 10 \cdot 2\} = \min\{0 + 20 + 10, 50 + 100\} = 30$ and record $s[1, 3] = 2$. $N[2, 4] = \min\{N[2, 2] + N[3, 4] + 1 \cdot 10 \cdot 10, N[2, 3] + N[4, 4] + 1 \cdot 2 \cdot 10\} = \min\{0 + 200 + 100, 20 + 0 + 20\} = 40$ and record $s[2, 4] = 3$.

Then we fill the fourth row from the bottom. $N[0, 3] = \min\{N[0, 0] + N[1, 3] + 10 \cdot 5 \cdot 2, N[0, 1] + N[2, 3] + 10 \cdot 1 \cdot 2, N[0, 2] + N[3, 3] + 10 \cdot 10 \cdot 10\} = \min\{0 + 30 + 100, 50 + 20 + 20, 150 + 0 + 1000\} = 90$ and record $s[0, 3] = 1$. $N[1, 4] = \min\{N[1, 1] + N[2, 4] + 5 \cdot 1 \cdot 10, N[1, 2] + N[3, 4] + 5 \cdot 10 \cdot 10, N[1, 3] + N[4, 4] + 5 \cdot 2 \cdot 10\} = \min\{0 + 40 + 50, 50 + 200 + 500, 30 + 0 + 100\} = 90$ and record $s[1, 4] = 1$.

Finally, $N[0, 4] = \min\{N[0, 0] + N[1, 4] + 10 \cdot 5 \cdot 10, N[0, 1] + N[2, 4] + 10 \cdot 1 \cdot 10, N[0, 2] + N[3, 4] + 10 \cdot 10 \cdot 10, N[0, 3] + N[4, 4] + 10 \cdot 2 \cdot 10\} = \min\{0 + 90 + 500, 50 + 40 + 100, 150 + 200 + 1000, 90 + 0 + 200\} = 190$ and record $s[0, 4] = 1$. Summarizing, in order to compute $M_0 \dots M_4$ we need to perform 190 operations. We first compute $M_0 \times M_1$ and then $M_2 \times M_3 \times M_4$. The first product is computed directly in 50 multiplications by reading entry $N[0, 1]$. For M_{24} we obtain that we need to perform 40 operations since $N[2, 4] = 40$, and the way is by finding $M_2 \times M_3$ first and then multiplying the result with M_4 as derived from the $s[2, 4]$ entry recorded. The first product $M_2 \times M_3$ requires 20 operations, and the second the remaining twenty multiplications. Therefore $M_{04} = (M_0 \times M_1) \times ((M_2 \times M_3) \times M_4)$. ■

Optional: Linear Equations

Introduction

A set of linear equations with unknowns $x_1, x_2, \dots, x_n$ is

$$\begin{aligned}a_{11}x_1 + a_{12}x_2 + \dots + a_{1n}x_n &= b_1, \\a_{21}x_1 + a_{22}x_2 + \dots + a_{2n}x_n &= b_2, \\&\dots = \dots \\a_{n1}x_1 + a_{n2}x_2 + \dots + a_{nn}x_n &= b_n,\end{aligned}\tag{2}$$

A set of values for $x_1, \dots$ that satisfy all Eq. 2 is said to be a solution to these equations. We can rewrite this system as follows in matrix-vector product form or equivalently letting $A = (a_{ij})$ and $x = (x_i)$ and $b = (b_i)$, we can write $Ax = b$. Then $x = A^{-1}b$, as long as A has an inverse A^{-1} . In that case vector x is unique.

$$\begin{pmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \dots & \dots & \dots & \dots \\ a_{n1} & a_{n2} & \dots & a_{nn} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \dots \\ x_n \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ \dots \\ b_n \end{pmatrix}$$

A (column) vector x is an $n \times 1$ matrix. A row vector x^T is an $1 \times n$ matrix.

Optional: Linear Equations

LUP decomposition

The idea behind the LUP decomposition is to find three matrices L, U, P such that $PA = LU$, where P is a permutation matrix of zeroes and ones, U is an upper-triangular matrix and L is a lower triangular matrix. Then solving A reduces to the problem of solving triangular matrices which is simpler.

Suppose we have determined the LUP decomposition of A . Then $Ax = b$ yields an equivalent system $PAx = Pb$, where PA permutes the rows of A . It is easy to see that a 1 in position (i, j) in P has the effect of moving row j of A to row i in PA . Similar observations can be derived for columns of A and AP respectively.

Since $PA = LU$ from the decomposition, we obtain that $PAx = Pb$ can be rewritten as $LUx = Pb$. We solve such a system in two steps. Let us write $Ux = y$. Then $Ly = Pb$.

Solve(A,x,b)

1. Find LUP decomposition of A (i.e. $PA=LU$).
2. Solve $Ly=Pb$ by forward substitution.
3. Solve $Ux=y$ by backward substitution.
4. return(x)

We first solve the system $Ly = Pb$ by forward substitution to find y .

Then we solve the system $Ux = y$ by backward substitution to find x , the solution of $Ax = b$.

This is so because by $Ax = b$ and $PA = LU$, we get $A = P^{-1}LU$.

$$Ax = P^{-1}L(Ux) = P^{-1}Ly = P^{-1}Pb = b,$$

as required.

LUP Decomposition

Forward and Backward Substitution

Forward substitution solves a linear system of n unknowns in $O(n^2)$ time. Note that for the system of equations below, we have $b_{j[i]}$ instead of say b_i because of the product Pb : this permutes the rows (i.e. elements) of the column vector b by moving to row i the element previously in row $j[i]$. We also note that the diagonal elements of L are units (i.e. one) where as the diagonal elements of U are not necessarily one.

$$\begin{array}{cccccc}
 y_1 + & & & \dots & & = b_{j[1]} \\
 l_{21}y_1 + & y_2 + & & \dots & & = b_{j[2]} \\
 l_{31}y_1 + & l_{32}y_2 + & y_3 + & \dots & & = b_{j[3]} \\
 & & & \dots & & \\
 l_{n1}y_1 + & l_{n2}y_2 + & l_{n3}y_3 + & \dots & y_n & = b_{j[n]}
 \end{array}$$

We first solve for y_1 from the first equation. We have that $y_1 = b_{j[1]}$.

Substituting y_1 into the second equation and solving for y_2 we find that $y_2 = b_{j[2]} - l_{21}y_1$.

Similarly $y_i = b_{j[i]} - \sum_{k=1}^{i-1} l_{ik}y_k$.

Back substitution works upwards by first solving for x_n in $Ux = y$. Then $x_i = (y_i - \sum_{k=i+1}^n u_{ik}x_k) / u_{ii}$.

LUP Decomposition

Finding the LUP decomposition

In the absence of P , the LUP decomposition can be found through the LU decomposition algorithm, which is often called Gaussian elimination. In Gaussian elimination, we first subtract multiples of the first equation from the others so that the first variable is eliminated from the second third and so on equations. After we accomplish this we then subtract multiples of the second equation from the others so that the second variable is eliminated from the third fourth and so on equations, and so on. This way we end up with an upper triangular matrix in place of A . Matrix L is made up of the multipliers we used in variable elimination. An LU decomposition description is a recursive procedure.

If $n = 1$, we are done, since we can choose $L = I_1$ and $U = A$.

For $n > 1$ we break A into four pieces.

$$A = \left(\begin{array}{c|ccc} a_{11} & a_{12} & \dots & a_{1n} \\ \hline & & & \\ a_{21} & a_{22} & \dots & a_{2n} \\ \hline \dots & \dots & \dots & \dots \\ a_{n1} & a_{n2} & \dots & a_{nn} \end{array} \right) = \begin{pmatrix} a_{11} & w^T \\ v & A' \end{pmatrix}$$

where v is $n - 1$ element column vector, w^T is an $n - 1$ element row vector, and A' is an $(n - 1) \times (n - 1)$ matrix. Then we can verify that

$$A = \begin{pmatrix} a_{11} & w^T \\ v & A' \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ v/a_{11} & I_{n-1} \end{pmatrix} \begin{pmatrix} a_{11} & w^T \\ 0 & A' - vw^T/a_{11} \end{pmatrix}$$

In the latter expression, the 0's in the first row and first column span $n - 1$ columns and rows respectively. The term $A' - vw^T/a_{11}$ is an $(n - 1) \times (n - 1)$ matrix called the **Schur complement** of A with respect to a_{11} .

We now claim if A is nonsingular so is the Schur complement. We prove it by way of contradiction. We thus assume that the Schur complement is singular. Then its row rank is strictly less than $n - 1$. Then because the bottom $n - 1$ rows have 0 in the first column, the first row is independent of the $n - 1$ other rows, i.e. the row rank of the entire matrix is less than $n - 1$, and so is that of the product, i.e. rank of A is strictly less than $n - 1$.

LUP Decomposition

Finding the LUP decomposition (continued)

Since the Schur complement is nonsingular we can recursively establish its own complement by further decomposing $A' - vw^T/a_{11}$. Let the end result of the recursion that LU decomposes the Schur complement of A with respect to a_{11} be $A' - vw^T/a_{11} = L'U'$.

$$\begin{aligned} A &= \begin{pmatrix} 1 & 0 \\ v/a_{11} & I_{n-1} \end{pmatrix} \begin{pmatrix} a_{11} & w^T \\ 0 & A' - vw^T/a_{11} \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ v/a_{11} & I_{n-1} \end{pmatrix} \begin{pmatrix} a_{11} & w^T \\ 0 & L'U' \end{pmatrix} \\ &= \begin{pmatrix} 1 & 0 \\ v/a_{11} & L' \end{pmatrix} \begin{pmatrix} a_{11} & w^T \\ 0 & U' \end{pmatrix} = LU. \end{aligned}$$

If a_{11} is zero, the whole scheme breaks down. We call such elements (the diagonal elements) the **pivots**. In order to make sure that a pivot is not zero, we can preprocess A , by permuting its rows so that a_{11} is non-zero. The same applies to the Schur complement of A . The row exchanges in pivoting operations are kept track in the permutation matrix P .

```
LUDecompose(A)
1. for(k=1;k<=n;k++) {
2.     u_kk = a_kk;
3.     for (i=k+1;i<=n;i++) {
4.         l_ik = a_ik / u_kk; //l_ik holds v_i
                                T
5.         u_ki = a_ki;        //u_ki holds w_i
6.     }
7.     for(i=k+1;i<=n;i++){      // The Schur complement of A
8.         for(j=k+1;j<=n;j++){  // with respect to a_kk
9.             a_ij = a_ij - l_ik * u_kj ; // is computed
10.        }
11.    }
12. return(L=l[ ][ ],U=u[ ][ ]);
```

Because of the triple loop of lines 1, 7, and 8, running time is $O(n^3)$.

Finding the LUP decomposition (continued)

The pivoting variant is the same except that before an elimination takes place (i.e. the Schur complement is established), we permute rows so that in the k -th iteration the element with largest absolute value is located in a_{kk} .

For example, before we start dividing by the input matrix's a_{11} we exchange the first row with that row l whose a_{l1} has the largest absolute value, and after exchanging the l -th and 1-st rows, then and only then the Schur complement calculation is performed: the a_{11} used is the original a_{l1} because of the row exchange. This pivoting operation requires for the i column $n - i$ comparisons to establish the element with the largest absolute value and $n - i + 1$ swaps for row exchange. The total cost of pivoting over the n columns is thus $O(n^2) = o(n^3)$.

We keep track of row exchanges not in a permutation matrix which is a sparse matrix (only n of n^2 entries are non-zero) and wastes space but in an one-dimensional matrix instead to save space.

Initially the i -th entry of this matrix is set to i . If at some point we exchange row k and row l , we also exchange the k -th and l -th entries of the matrix.

DRAFT

Matrix Inversion

LUP-based algorithm

On the same 4 need to be solved. The LUP decomposition of A requires $O(n^3)$ operations. This cost can be considered a preprocessing cost if many equations

After we have established this decomposition, we can solve the system $Ax = b$ in $O(n^2)$ additional time (the cost of forward and backward substitution). If another system $Ax = c$ is given we can solve this system in additional $O(n^2)$ time as A is already decomposed.

If k systems of the form $Ax = D_i$, $i = 1, \dots, k$ are given, all these systems can be solved in $O(kn^2)$ time given the availability of LUP of A . If we have n such systems the total time to solve these is $O(n \cdot n^2) = O(n^3)$ plus the time to decompose A which is also $O(n^3)$ for a total of $O(n^3)$.

If the n systems are the n columns of I_n , the x column vectors that we calculate are the column vectors of a matrix X such that $AX = I_n$, i.e. we thus can calculate the inverse of A in total $O(n^3)$ time, provided that A exists.

DRAFT

Matrix Inversion and Multiplication Relationships

Theorem. Matrix multiplication is no harder than inversion as long as the time $In(n)$ it takes to invert an $n \times n$ matrix is $\Omega(n^2)$, and $In(3n) = O(In(n))$.

Proof. Suppose A, B, C are three matrices for which the product $C = A \times B$ needs to be established. We then create the matrix

$$D = \begin{pmatrix} I_n & A & 0 \\ 0 & I_n & B \\ 0 & 0 & I_n \end{pmatrix}$$

whose inverse is,

$$D^{-1} = \begin{pmatrix} I_n & -A & AB \\ 0 & I_n & -B \\ 0 & 0 & I_n \end{pmatrix}$$

Forming D from A and B takes $O(n^2)$ time. Finding the inverse of D which is a $3n \times 3n$ matrix takes time $In(3n)$, where $In(n)$ is the time it takes to invert a matrix of size $n \times n$. Subsequently AB can be read off D^{-1} in $O(n^2)$ time. Therefore $M(n)$ the time it takes to multiply two $n \times n$ matrices is given by the following expression

$$M(n) = O(n^2) + In(3n)$$

Since $In(3n) = O(In(n))$ and $In(n) = \Omega(n^2)$, we have that

$$M(n) = O(In(n)). \square$$

Theorem. Inversion is no harder than matrix multiplication (under certain “innocent conditions”).

Proof. Perform a similar trick by forming a matrix and then end up with the following recurrence

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

The master method (case 3) provides the solution to this recurrence as long as $2M(n/2) < M(n)$. The solution is $In(n) = O(M(n))$.

Matrix Inversion and Multiplication Relationships (continued)

An easier bound to derive is by noting that an $n \times n$ matrix A can be written in terms of its $n/2 \times n/2$ submatrices $A_{11}, A_{12}, A_{21}, A_{22}$ as follows.

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} = \begin{pmatrix} I & 0 \\ A_{21}A_{11}^{-1} & I \end{pmatrix} \begin{pmatrix} A_{11} & 0 \\ 0 & D \end{pmatrix} \begin{pmatrix} I & A_{11}^{-1}A_{12} \\ 0 & I \end{pmatrix}$$

where $D = A_{22} - A_{21}A_{11}^{-1}A_{12}$. We denote by M matrix multiplications of $n/2 \times n/2$ matrices and with I inversions of $n/2 \times n/2$ matrices.

In order to compute D we need 2 M and 1 I (of A_{11}^{-1}) plus $O(n^2)$ time for the subtraction.

The triple matrix form of A requires 1 additional M to compute $A_{11}^{-1}A_{12}$ or $A_{21}A_{11}^{-1}$ as the other one is a partial product in D 's computation.

Therefore the total count is 3 M 's, 1 I , plus $O(n^2)$ work in order to establish the triple matrix representation of A .

After this representation of A has been established, the inverse of A is then given by

$$A^{-1} = \begin{pmatrix} I & -A_{11}^{-1}A_{12} \\ 0 & I \end{pmatrix} \begin{pmatrix} A_{11}^{-1} & 0 \\ 0 & D^{-1} \end{pmatrix} \begin{pmatrix} A_{11}^{-1} & 0 \\ 0 & D^{-1} \end{pmatrix} \begin{pmatrix} I & 0 \\ -A_{21}A_{11}^{-1} & I \end{pmatrix}$$

In order to multiply the second and third matrices we need to perform 1 M . In order to multiply the first with the previous result we need 2 M 's. The inverse of A_{11} needs also to be established.

The total count therefore is

$$In(n) = 2In(n/2) + 6M(n/2) + \Theta(n^2).$$

This method is slightly better than the one in the textbook where the $6M(n/2)$ term becomes $4M(n)$.

Matrix Inversion through Matrix Multiplication

The transformation that proves that matrix inversion is no harder than matrix multiplication can be made to work without a need of checking whether A_{11} and D are singular or not (as long as we know that A is nonsingular), if instead of finding the inverse of A we find the inverse of $A^t A$, which is $(A^t A)^{-1}$. It is easy to show that

$$A^{-1} = (A^t A)^{-1} A^t$$

and therefore given the inverse of $A^t A$ we can recover A^{-1} by one extra matrix multiplication operation for a total additional cost of $M(n)$. $A^t A$ is a positive definite matrix and as such is guaranteed that A_{11} and D are nonsingular and positive definite.

DRAFT

Optional: Binary numbers

Integer and Real

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 4. 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.

Theorem 23. *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 10. The theorem above (Theorem 23) applies even if X is not integer provided that $X \geq 1$.

Question 1. 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 23 to include all possible cases of positive numbers, integer or not.

Theorem 24. 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 11. From Theorem 24 we obtain that $b = .1b_2 \dots b_m$ is such that $1/2 \leq b \leq 1$. This also implies that $1 \leq 1/b \leq 2$.

Theorem 25. *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} . ■

Example 33. 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. $2 \leq x \leq 4$.

DRAFT

Optional: Root finding

Bisection is 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 .

Optional: Reciprocal Using Newton-Raphson method

Problem 18 (Reciprocal Unrestricted).

Input. An integer y .

Output. Calculate using Newton-Raphson $x = 1/y$, the reciprocal of y .

In general y is unrestricted and therefore $x = 1/y$ is also unrestricted in value. If y is m bit long then $y = 2^m \cdot Y$ or equivalently $Y = 2^{-m} \cdot y$ leads to an Y such that $1/2 \leq Y < 1$. In other words we shift y to the right m bit positions so that the integer part is a zero and the bit next to the decimal point is a one! This guarantees that Y is $Y \geq 1/2$. Moreover because the integer part is zero we have that $Y < 1$ as well. This way we are going to reduce the original problem into the following one.

Problem 19 (Reciprocal Restricted).

Input. An integer $y = 2^m \cdot Y$ and its normalized fractional part Y , where $1/2 \leq Y < 1$.

Output. Calculate using Newton-Raphson $X = 1/Y$, the reciprocal of Y . Then output X shifted to the right m positions. (Since $x = 1/y = 1/Y \cdot 2^{-m} = X \cdot 2^{-m}$, the output is the reciprocal of y .)

Optional: Reciprocal using Newton-Raphson Preliminaries

Theorem 26. Let Y be $1/2 \leq Y < 1$. The n bit of the approximation of the reciprocal X of Y can be calculated in $\lg n$ iterations of a Newton-Raphson method, with initial approximation $X_0 = 1.5$.

Proof. Let Y be the input for which $1/2 \leq Y < 1$. This also implies that $1 < 1/Y \leq 2$. We are to compute $X = 1/Y$ and X is such that $1 < X \leq 2$. Consider the function $f(x) = Y - 1/x$. A solution for $f(x) = 0$ is $x = 1/Y$. Moreover $f'(x) = 1/x^2$. If we apply Newton-Raphson the i -st approximation of $1/Y$ is x_i such that

$$x_i = x_{i-1} - \frac{f(x_{i-1})}{f'(x_{i-1})} = x_{i-1} - \frac{Y - 1/x_{i-1}}{1/x_{i-1}^2} = 2x_{i-1} - Yx_{i-1}^2.$$

Let $x_0 = 1.5$. Then $-0.5 < 1/Y - x_0 \leq 0.5$ since $1 < 1/Y \leq 2$. Moreover from the recurrence for x_i we have that

$$1/Y - x_1 = 1/Y - 2x_0 + Yx_0^2 = Y(1/Y^2 - 2x_0/Y + 1/x_0^2) = Y(1/Y - x_0)^2$$

By induction on i we conclude that

$$1/Y - x_i = Y^{2^i-1} (1/Y - x_0)^{2^i}.$$

The term $Y^{2^i-1} \leq Y$ for $i \geq 1$. Furthermore $Y^{2^i-1} < 1$ and so is Y^{2^i-2} for $i > 1$. Moreover $1/Y - x_0 \leq 1/2$. Therefore

$$|1/Y - x_i| \leq 1/2^{2^i}$$

If $i = \lg n$ i.e. we stop the approximation at the conclusion of iteration i we have then

$$|1/Y - x_{\lg n}| \leq 1/2^n.$$

■

Optional: Division using Newton-Raphson Preliminaries

Theorem 27. *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 $O(M(n) \lg n)$ bit operations, where $M(k)$ is the number of bit operations to multiply two k precision (i.e. k bit) integers.*

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 $1/2 \leq y \leq 1$.

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 $O(M(n) \lg n)$. ■

Optional: Division using Newton-Raphson Continued

Note 5. The bit count bound for the Newton-Raphson approach to finding 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.

Optional: Division using Newton-Raphson Continued

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.

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

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 .

Optional: Division using Newton-Raphson Continued

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 3. 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'.

$$\begin{aligned}\bar{x}_i &= 2x_{i-1} - (.y_1y_2 \dots y_{2^i+4})x_{i-1}^2 \\ x_i &= \lceil \bar{x}_i \rceil\end{aligned}$$

Remark 4. With Step 1 changed as in Remark 3, the i -th iteration of Newton Raphson requires time $M(2^{i+4})$.

Optional: Division using Newton-Raphson Optimality

Remark 5. The modified Newton-Raphson according to Remark 3 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 $T(n) = O(M(n))$. The proof of Remark 5 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.

$$e_i = 1/y - x_i = 1/y - 2x_{i-1} + y_{i-1}x_{i-1}^2 = 1/y - (2(1/y - e_{i-1}) - y_{i-1}(1/y - e_{i-1})^2) = y_{i-1}e_{i-1}^2 - \frac{y - y_{i-1}}{y^2}(1 - 2e_{i-1}y)$$

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

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

■

Optional: Division using Newton-Raphson

Example

Example 34. Suppose that $X = 20$ and $Y = 5$. Then $X = 10100$ and $Y = 101$. We normalize X and Y so that $X = x2^3 = 10.100 \times 2^3$ 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 $2^1 - 1$ bit 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 3 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 approximation to $1/y$ 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.

Proposition 15. *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 $\lg a$ bit long, the quotient q of a divided by b is at most $\lg a$ bit long as well. Since $r < b$, r is $\lg b$ bit 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 4. 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 5. 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 6. 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 7. If a, b are integers such that $|a| + |b| \neq 0$, then $\gcd(a, b)$ is the smallest positive integer of the set $\{ax + by : x, y \in \mathbb{Z}\}$.

Fact 8. The following also hold.

- For any a, b if d/a and d/b , then $d/\gcd(a, b)$.
- $\gcd(an, bn) = n \cdot \gcd(a, b)$.
- If n/ab and $\gcd(a, n) = 1$, then n/b .
- If $\gcd(a, p) = 1$ and $\gcd(b, p) = 1$, then $\gcd(ab, p) = 1$.

Definition 18. If $\gcd(a, b) = 1$ we call a and b relatively prime.

Fact 9. 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 $a - a/p_1 - \dots - a/p_n$. A multiple of both p_1 and p_2 was counted twice. Therefore we need to add terms of the form $n/(p_1 p_2) + \dots + n/(p_1 p_t)$. But a multiple of $p_1 p_2 p_3$ has been reinserted at least twice. We correct again, and finally by alternating between subtractions and additions we get a term that is $\phi(a) = a(1 - 1/p_1) \dots (1 - 1/p_n)$.

Fact 10. 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

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

contains all the integers whose division by b yield the same remainder as a . For two integers a, c we can write $c \in [a]_b$ 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 $Z_b = \{0, 1, \dots, b - 1\}$.

Example 35. 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.

Optional: Number Theory

Euclid's Algorithm

Theorem 28. *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 16. *The algorithm eventually terminates.*

The proof is on the next page.

Optional: Number Theory

Euclid's Algorithm

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

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

Because $a_i < a_{i-1}$ we have that $a_i < a_{i-1} < a_{i-2} < \dots < a_2 < a_1$. 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.

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

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 6. 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.

Optional: Euclid's Algorithm

Division steps

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 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 $F_1 = 1$.

Fact 11. 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 17. 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 $i = 1, \dots, n-1$, 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. ■

Optional: Euclid's Algorithm

Corollaries

Corollary 12. 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 17 $b \geq F_{k+1}$. This contradicts the fact that $b < F_{k+1}$. ■

Corollary 13. If $b < F_{k+1}$, then the number of division steps is $\Theta(\lg b)$.

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

Corollary 14. 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 is $\Theta(\lg b)$ 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 15. 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 16. 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 15 to reduce the running time of $O(\lg^3 a)$ of Corollary 14 to $O(\lg a \lg b)$. (You also need a proof that you can find q in $O(\lg a \lg b)$.) ■

DRAFT

Optional: Extended GCD Preliminaries

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

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

Optional: Extended GCD

Preliminaries 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 29. *The running time of EXTENDED EUCLID is $O(\lg a \lg b)$.*

Optional: Extended GCD

Alternative view

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)
```

Optional: Modular Arithmetic

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 $(\bmod n)$. 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 $|S| \mid |Z_n^*|$. If S is a proper subgroup, then $|S| \leq |Z_n^*|/2$.

Optional: Modular Arithmetic

Solving modular linear equations

Consider : $ax \equiv b \pmod{n}$

Theorem 30. Consider the set $\langle a \rangle = \{ax \pmod{n} : x > 0\}$. If $b \in \langle a \rangle$, then the equation above has a solution for x . Let $d = \gcd(a, n)$. Consider set $D = \{0, d, 2d, \dots, ((n/d) - 1)d\} = \langle d \rangle$. Then, $D = \langle a \rangle$.

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

If $m \in \langle a \rangle$, then $m \equiv ax \pmod{n}$ 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. $m \in \langle d \rangle = D$. This means that $\langle a \rangle \subseteq \langle d \rangle = D$.

Therefore $D = \langle a \rangle$. ■

This leads to the following theorem.

Theorem 31. $ax \equiv b \pmod{n}$ has a solution for x if and only if $\gcd(a, n) \mid b$.

Theorem 32. If $ax \equiv b \pmod{n}$ has a solution for x then it has $d = \gcd(a, n)$ solutions. One solution is $x_0 = x'(b/d) \pmod{n}$, where $d = ax' + ny'$, and the other solutions are $x_0 + i(n/d)$ for $1 \leq i \leq d - 1$.

Optional: Modular Arithmetic

Solving modular linear equations

Remark 7. $ax \equiv b \pmod n$ 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. $a(x_0 + in/d) \equiv b \pmod n$. 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");
```

Optional: Chinese Remainder Theorem

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 33 (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 $m_i \equiv 0 \bmod n_j$ 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 . ■

Optional: Chinese Remainder Theorem

Example

Example 36. 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.

$m1 = 210/2 = 105$	$\text{inverse}(m1) \% n1 = 1 \times m1 = 105$
$m2 = 210/3 = 70$	$\text{inverse}(m2) \% n2 = 1 \times m2 = 70$
$m3 = 210/5 = 42$	$\text{inverse}(m3) \% n3 = 3 \times m3 = 126$
$m4 = 210/7 = 30$	$\text{inverse}(m4) \% n4 = 4 \times m4 = 120$

Therefore

$(0, 1, 0, 1) = 0 \times 105 + 1 \times 70 + 0 \times 126 + 1 \times 120 = 190$
which is the sum of $182 = (0, 2, 2, 0)$ and $8 = (0, 2, 3, 1)$.

Optional: Modular Powers

Introduction

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

Theorem 34 (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 35 (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 36 (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 .*

Optional: Modular Powers

Introduction continued

The following is called the Discrete Logarithm Theorem.

Theorem 37 (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 38. *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 $d \neq 0$ 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.

Optional: Modular Powers (revisited)

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.

Notice Problem Set with Solutions

These Problem Sets are not for credit. They are provided as are for your practice. Several problems are provided with solutions. These can be discussed with the instructor after a student has studied the relevant material. Some other problems are provided with either no solutions at all or a hint of a solution. They are there for your reference and will not be discussed further. For some solutions provided you might have to fill in details related to calculations or intermediate steps.

Before you try these problems: (a) Study the designated textbook, (b) use the Subject notes (typewritten notes available through the course web-page from the password-protected section C) to review your study of the designated textbook material, (c) resolve the examples done in class, (d) then practice with some of these problems PS (Problem Set), and then if you have time, (e) solve some of the problems of the designated textbook, and (f) solve more problems from the PS especially the ones with minimal solution information.

Several PS problems appeared in past exams or homeworks.

Copyright (c) 2020-2024. Alex. Gerbessiotis. All rights reserved.
Not to be displayed outside of copyright holder's web-page

Problem Set
Problem 1.

TRUE or FALSE? Anything other than a TRUE or FALSE is considered wrong!

1. If we have t^2 matrices $M_1, M_2, \dots$, of geometry $r_1, r_2, \dots$ we can determine the optimal schedule to multiply these matrices using the dynamic programming algorithm for the matrix chain product available in the textbook and Subject 9 in time $O(t^3)$.
2. If we have n matrices $M_i, i = 1, \dots, n$, each one of geometry $n \times n$, we can find the optimal schedule (requiring the minimal amount of scalar operations) to compute the product $M_1 \times M_2 \times \dots \times M_n$ of the n matrices in $\Theta(n^3)$ time.
3. If we have the same n matrices $M_i, i = 1, \dots, n$, each one of geometry $n \times n$, the optimal schedule to compute the product $M_1 \times M_2 \times \dots \times M_n$ requires $O(n^4)$ scalar operations.
4. If we have n matrices $M_i, i = 1, \dots, n$, each one of geometry $n \times n$, the optimal schedule to compute the product $M_1 \times M_2 \times \dots \times M_n$ can be done in $O(n^{3.9})$ scalar operations.

Proof. 1. FALSE, 2.FALSE, 3. TRUE, 4. TRUE. ■

Problem Set Problem 2.

You are given six polynomials $f_1, \dots, f_6$ of degrees 1, 2, 3, 1, 4, 5 respectively. We are interested in finding the product $f = f_1 f_2 f_3 f_4 f_5 f_6$. Assume that the cost of multiplying two polynomials of degree a and b is $a \cdot b$. Find a schedule for multiplying the six polynomials that is of the lowest possible total cost.

Proof. Multiply them any way you like. The final result is a polynomial of degree 16 and the total cost is 100. Note that 100 is the sum of all products $d_i \cdot d_j$ for every combination of two polynomials f_i and f_j among the six and d_i and d_j are respectively the degrees of f_i and f_j . In our example we have $6(6-1)/2$ such pairs of polynomials.

$$\begin{aligned} 100 &= d_1(d_2 + d_3 + d_4 + d_5 + d_6) + d_2(d_3 + d_4 + d_5 + d_6) + d_3(d_4 + d_5 + d_6) + d_4(d_5 + d_6) + d_5d_6 \\ &= 1(2 + 3 + 1 + 4 + 5) + 2(3 + 1 + 4 + 5) + 3(1 + 4 + 5) + 1(4 + 5) + 4 \cdot 5 \\ &= 15 + 26 + 30 + 9 + 20 \end{aligned}$$

This sum is independent of the order of the evaluation of the products; an easy proof is by induction (eg. with reference to the 6 polynomials above, the product of five of them $f_2, \dots, f_6$ takes time equal to the 4 summands except the first one $d_1(d_2 + d_3 + d_4 + d_5 + d_6)$; in the last step of an inductive proof the polynomial f_1 whose degree is d_1 is multiplied by the result $f_2 \dots f_6$ whose degree is $d_2 + \dots + d_6$ and the cost of that multiplication is the missing/absent (first) term. Permute the polynomial and the result remains the same except that the pairs of the products $d_i d_j$ get rearranged.).

Note. One of the reasons that this is so is that polynomials satisfy commutativity therefore $f_1 f_2 = f_2 f_1$. What happens if instead of polynomials you have matrices and the operation is matrix multiplication? The cost then depends on the specific schedule that you choose. This is the matrix chain product; on how to determine the optimal schedule using dynamic programming, See the textbook or Subject 9. Problem 1 relates to this question!

■

Problem Set

Problem 3.

- a. What is the largest k such that if you can multiply 3×3 matrices using k multiplications (not assuming commutativity of multiplication), then you can multiply $n \times n$ matrices in time $O(n^{\lg 7})$? What would the running time of this algorithm be?
- b. V. Pan has discovered a way of multiplying 68×68 matrices using 132464 multiplications, 70×70 matrices using 143640 multiplications, 72×72 matrices using 155424 multiplications. Which method yields the best asymptotic running time when used in a divide and conquer matrix multiplication algorithm? How does it compare to Strassen's algorithm?

Proof. a. Strassen's algorithm uses the pattern of multiplication for 2×2 matrices and divide-and-conquer by splitting a matrix multiplication of $n \times n$ matrices into a matrix multiplication of $n/2 \times n/2$ matrices to derive a time bound for matrix multiplication given by the recurrence $T(n) = 7T(n/2) + O(n^2)$. In such a recurrence the 7 indicates the number of scalar multiplications in multiplying 2×2 matrices or the number of $n/2 \times n/2$ matrix multiplications in the recursive (divide-and-conquer) decomposition. Note that Strassen's method on 2×2 matrices cannot be improved. We cannot multiply 2×2 matrices with say 6 (scalar) multiplications.

Now if we can multiply 3×3 matrices and k is the number of scalar multiplications required and we use a recursive decomposition of an $n \times n$ matrix into $n/3 \times n/3$ matrices we get a recurrence for the running time of this method of

$$T(n) = kT(n/3) + O(n^2).$$

The solution of this recurrence (master method) yields $T(n) = \Theta(n^{\log_3 k})$. Therefore if we compare the exponent $\log_3 k$ to $\lg 7$, then for $k = 21$ we have $\log_3 21 \approx 2.771 < 2.8073 = \lg 7$.

For that value of $k = 21$ or smaller a by 3 decomposition beats Strassen's method.

- b. This is similar to part (a). If we decompose not by-2 or by-3, but by- t , where t now is 68, 70, 72 we obtain that the running time for such a decomposition is given by the recurrence,

$$T(n) = kT(n/t) + O(n^2).$$

with an asymptotically tight solution given by $T(n) = \Theta(n^{\log_t k})$. Therefore if we compare the exponent $\log_t k$ to $\lg 7$, then for $t = 68$, $\log_{68} 132464 = 2.795128$, $t = 70$, $\log_{70} 143640 = 2.795122$, $t = 72$, $\log_{72} 155424 = 2.795147$. All three of them beat Strassen's exponent. The best of the three is the $t = 70$ split. Calculations used the Unix `bc` calculator and hopefully are correct! ■

Problem Set Problem 4.

For the remainder of this problem we assume that the cost of multiplying two polynomials of degrees m and n respectively, is $m + n$. You are given six polynomials $f_1, f_2, f_3, f_4, f_5, f_6$ of degrees 1, 2, 3, 1, 4, 5 respectively. We are interested in finding the product $f = f_1 f_2 f_3 f_4 f_5 f_6$ by minimizing the total cost of all intermediate polynomial multiplications involved in this computation. The total cost of finding the product is thus the sum of the costs of all polynomial multiplications involved. What is the best schedule to find f and what is the cost associated with this best schedule? Explain.

Example Let g_1, g_2, g_3 be three polynomials of degree 3, 2, 6 respectively. If we first multiply $h_1 = g_2 \cdot g_3$ the cost of this multiplication is $8 = 2 + 6$, because the two polynomials g_2, g_3 are of degrees 2 and 6 respectively. If we then find $h_2 = h_1 \cdot g_1 = g_1 \cdot g_2 \cdot g_3$ the cost of this second multiplication is $11 = 3 + 8$, because the degree of g_1 is 3 and the degree of h_1 is $8 = 2 + 6$, i.e. the sum of the degrees of g_2, g_3 . This schedule results to a total cost of $8 + 11 = 19$ for computing the product $g_1 \cdot g_2 \cdot g_3$. It remains to be seen whether it is the best schedule or not.

Proof. Multiply f_1 by f_4 . The result with f_2 . Then say we multiply f_3 with f_5 . The product of f_1, f_4, f_2 with f_6 , and then the last remaining polynomials together. This is a greedy, ala Huffman approach. You fill in the details. ■

Problem Set
Problem 5.

- (a) What's the best way to find the product $M_0 \times M_1 \times M_2 \times M_3$ where the matrices are of dimension 100×200 , 200×500 , 500×10 , and 10×1000 respectively? (In a prior problem, you had to deal with polynomials. The polynomials have now become matrices.)
- (b) You are given 4 matrices M_0, M_1, M_2, M_3 and you are asked to determine the optimal schedule for the product $M_0 \times M_1 \times M_2 \times M_3$ that minimizes the number of operations (multiplications) involved. The dimensions of the four matrices are respectively 10×1 , 1×2 , 2×1 , 1×10 . What is the best (cheapest) schedule to multiply all the matrices together and compute $M_0 \times M_1 \times M_2 \times M_3$? What is the total cost for this schedule?
- (c) You are given 4 matrices M_1, M_2, M_3, M_4 and you are asked to determine the optimal schedule for the product $M_1 \times M_2 \times M_3 \times M_4$ that minimizes the number of operations (addition/multiplication) involved. The dimensions of the four matrices are respectively 100×20 , 20×500 , 500×30 , and 30×100 . What is the best (cheapest) schedule to multiply all the matrices together and compute $M_1 \times M_2 \times M_3 \times M_4$? What is the total cost for this schedule?
- (d) You are given 4 matrices M_1, M_2, M_3, M_4 and you are asked to determine the optimal schedule for the product $M_1 \times M_2 \times M_3 \times M_4$ that minimizes the number of operations (addition/multiplication) involved. The dimensions of the four matrices are respectively 100×50 , 50×200 , 200×50 , and 50×10 . What is the best (cheapest) schedule to multiply all the matrices together and compute $M_1 \times M_2 \times M_3 \times M_4$? What is the total cost for this schedule?

Proof. (a) May be the cost is 2,200,000. You find the schedule! Practice with the other ones, if you need to! ■

Problem Set
Problem 6.

You are given an integer n . We would like to determine whether n is the integer power of an integer i.e. whether there exist two integer numbers a, b , $a > 1$ such that $n = a^b$. Give an algorithm that determines whether n is the integer power of an integer (and in that case one set a, b is printed out) or not in time that is $O(\lg^3 n)$. Use the word model for operation counting i.e. the cost of one addition, subtraction, multiplication is one. Divisions are not allowed unless they are a division by two (i.e. a shift).

Hint: If $n = a^b$ how big can b be relative to n ? And if you have spare time, can you beat the $O(\lg^3 n)$ time bound and by how much?

Proof. Binary search on the integer range from 1 to n ! ■

Copyright (c) 2020-2024. Alex. Gerbessiotis. All rights reserved.
Not to be displayed outside of copyright holder's web-page

Problem Set

Problem 7.

The Fibonacci sequence is defined as $F(n) = F(n-1) + F(n-2)$, for $n \geq 2$, and $F(1) = F(0) = 1$ otherwise.

- (a) Write a recursive program (pseudocode) that finds the n -th Fibonacci number (i.e. $F(n)$).
- (b) Write an iterative program (pseudocode) for $F(n)$.
- (c) Which of the two programs is asymptotically faster? Explain.
- (d) Give an algorithm that is $O(\lg n)$ in the WORD model.
- (e) Show how to compute the n -th Fibonacci number with the minimal number of bit operation in the BIT model. Say $O(n^2)$ bit operations.

Hint: $F_n \leq 2^n$, if we recall induction and its applications of Subject 1.

Proof.

```
Fibonacci-Recur(n) {  
1. if (n==0) return(1);  
2. if (n==1) return(1);  
3. return(Fibonacci-Recur(n-1)+Fibonacci-Recur(n-2));  
}
```

(b)

```
Fibonacci-Iterate(n) {  
0 if (n==0) or (n==1) return(1)  
1. f_n_minus2 = 1;  
2. f_n_minus1 = 1;  
3. result = 0;  
4. for (i=2; i<=n; i++) {  
5.   result = f_n_minus1 + f_n_minus2;  
6.   f_n_minus2 = f_n_minus1;  
7.   f_n_minus1 = result;  
8. }  
9. return(result);  
}
```

Copyright (c) 2020-2024. Alex. Gerbessiotis. All rights reserved.
Not to be displayed outside of copyright holder's web-page

Problem Set Continued

This is Subject 1 material. By now you should be able to solve (a), (b) and (c).

For (d) think! The running time of Fibonacci-Iterate is easy to find. The for loop runs for $\Theta(n)$ iterations for a total time of $\Theta(n)$ operations. The running time of Fibonacci-Recur can be estimated as follows. To find $F(1)$ or $F(0)$ it takes one operation (line 1 or 2). For $F(2)$ we need to find recursively $F(1)$ and $F(0)$, i.e. two operations total. For $F(3)$ we need to find recursively $F(2)$ and $F(1)$. Finding $F(1)$ takes $F(1) = 1$ operations. Finding $F(2)$ also takes $F(2) = 2$ operations. Therefore by an inductive argument, finding $F(n)$ takes $F(n)$ operations. From a discrete math book $F(n) = \Theta(((1+\sqrt{5})/2)^n) = \Theta((1.61803)^n)$. This is a tighter bound than the one we derived in class or Subject 1. The latter number is asymptotically larger than n . Therefore Fibonacci-Iterate is asymptotically faster. Of course memoization can improve the recursive algorithm. ■

Copyright (c) 2020-2024. Alex. Gerbessiotis. All rights reserved.
Not to be displayed outside of copyright holder's web-page

Problem Set Problem 8.

What does the following recursive program do?

```
int Findme ( int x , int y) {  
    if ( y == 0) return (0);  
    else  
        return( Findme( 2*x, floor(y/2)) + x * (y % 2 ) );  
}
```

Proof. Suppose we want to divide an integer y by 2. y is the dividend and 2 is the divisor. By the elementary school division we have that

$$y = 2Q + R$$

where Q is the quotient and R is the remainder. R is 0 or 1. The quotient is $Q = \text{floor}(y/2)$, and the remainder is $R = y\%2$. Therefore,

$$y = 2Q + R = 2\text{floor}(y/2) + (y\%2)$$

If we multiply by x both sides of the equality we get that

$$x * y = x * (2\text{floor}(y/2) + y\%2) = (2 * x) * (\text{floor}(y/2)) + x * (y\%2).$$

Therefore the product $x * y$ is equal to the sum of the product of $2x$ times $\text{floor}(y/2)$ and $x * (y\%2)$. Note that the product $x * (y\%2)$ is easy to find. If the $y\%2$ is 0 the product is 0, if $y\%2$ is 1 the product is x .

Therefore Findme finds the product $x * y$ recursively. ■

Problem Set

Problem 9.

- (a) You multiply $x = x_1 \dots x_k$, k (non-negative) integers $x_1, \dots, x_k$. In the bit model asymptotically estimate the number of bit operations and show it is $O((\lg x)^2)$, i.e. dependent on the result x and independent of the intermediate numbers.
- (b) In the bit model estimate the number of bit operations for $x = y^n$. Input is y and n and output is x . Assume all x, y, n are non-negative integer.
- (c) You are given x and you are asked to determine whether x is of the form $x = y^n$ for some integers y, n . Give an efficient algorithm for that. (Testing for all y and all n is not an efficient solution.)

Proof. No solutions provided for the other problem(s). Think! and practice! ■

Copyright (c) 2020-2024. Alex. Gerbessiotis. All rights reserved.
Not to be displayed outside of copyright holder's web-page

Problem Set
Problem 10.

For an $n \times n$ matrix A , the inverse of A is $i(A) = A^{-1}$ which is a matrix such that $i(A) \times A = A \times i(A) = I$, where I is the identity matrix (diagonals one, everything else zero). The cost of inversion of A is let it be $I(n)$. If we have a recursive approach that has cost

$$I(3n) \leq 20I(n) + 31n^2$$

what would the cost of $i(A)$ be ?

Proof. No solutions provided. Master method. ■

Copyright (c) 2020-2024. Alex. Gerbessiotis. All rights reserved.
Not to be displayed outside of copyright holder's web-page

Problem 11.

Let $J(n)$ be an upper bound for the number of operations to invert a positive definite matrix B that is $n \times n$, and let $M(n)$ be the cost of matrix multiplication. Show a lower bound for $I(n)$ to invert an arbitrary matrix A that is $n \times n$. (Use info from a previous problem, if necessary.)

Proof. Hint: Let $A^{-1} = (A^t \times A)^{-1} \times A^t$. ■

Copyright (c) 2020-2024. Alex. Gerbessiotis. All rights reserved.
Not to be displayed outside of copyright holder's web-page