21:17 9/21/2009

Chapter 3

Algorithms and Integers

coursenotes by Prof. J. L. Gross for Rosen 6th Edition

3.1 ALGORITHMS

DEF: An ***algorithm*** is a finite set of precise instructions for performing a computation or for solving a problem.

Example 3.1.1: A computer program is an algorithm.

Remark: From a mathematical perspective, an algorithm represents a function. The British mathematician Alan Turing proved that some functions cannot be represented by an algorithm.

CLASSROOM PERSPECTIVE

Every computable function can be represented by many different algorithms. Naive algorithms are almost never optimal.

TERMINOLOGY: A ***good pseudocoding*** of an algorithm provides a clear prose representation of the algorithm and also is transformable into one or more target programming languages.

Algo 3.1.1: Find Maximum

Input: unsorted array of integers $a_1, a_2, \ldots, a_n$
Output: largest integer in array

$\{Initialize\}$ $max := a_1$

For $i := 2$ **to** n
  If $max < a_i$ **then** $max := a_i$
  Continue with next iteration of for-loop.

Return (max)

Remark: For a sorted array, there would be a much faster algorithm to find the maximum. In general, the representation of the data profoundly affects both the choice of an algorithm and the execution time.

Algo 3.1.2: Unsorted Sequential Search

Input: unsorted array of integers $a_1, a_2, \ldots, a_n$
target value x
Output: subscript of entry equal to target value,
or 0 if not found

$\{Initialize\}$ $i := 1$

While $i \leq n$ **and** $x \neq a_i$
  $i := i + 1$
  Continue with next iteration of while-loop.

If $i \leq n$ **then** $loc := i$ **else** $loc := 0$
Return (loc)

Remark: If the array were presorted into ascending (or descending) order, then faster algorithms could be used.

(1) linear search could stop sooner

(2) 2-level search could avoid many comparisons

(3) binary search could divide-and-conquer

Algo 3.1.3: Sorted Sequential Search

Input: sorted array of integers $a_1, a_2, \ldots, a_n$
target value x
Output: subscript of entry equal to target value,
or 0 if not found

$\{Initialize\}$ $i := 1$

While $i \leq n$ **and** $x < a_i$
$i := i + 1$
Continue with next iteration of while-loop.

If $(i \leq n$ **cand** $x = a_i)$ **then** $loc := i$ **else** $loc := 0$
Return (loc)

DEF: The logical expression ***conditional-and***

$$boolean1 \textbf{ cand } boolean2$$

is like conjunction, except that $boolean2$ is not evaluated if $boolean1$ is false.

Example 3.1.2: In Algorithm 3.1.3, if $i > n$ then variable a_i does not exist. Since the conditional-and does not evaluate such an a_i, problems are avoided.

Algo 3.1.4: Two-level Search

Input: sorted array of integers $a_1, a_2, \ldots, a_n$
target value x
Output: subscript of entry equal to target value,
or 0 if not found

$\{Initialize\}$ $i := 10$
$\{$*Find target sublist of 10 entries*$\}$
While $i \leq n$ **and** $x < a_i$
$i := i + 10$
Continue with next iteration of while-loop.

$\{$*Linear search target sublist of 10 entries*$\}$
$\{Initialize\}$ $j := i - 9$
While $j < i$ **and** $x < a_j$
$j := j + 1$
Continue with next iteration of while-loop.

If $(j \leq n$ **cand** $x = a_j)$ **then** $loc := j$ **else** $loc := 0$
Return (loc)

Algo 3.1.5: Binary Search

Input: sorted array of integers $a_1, a_2, \ldots, a_n$
target value x
Output: subscript of entry equal to target value,
or 0 if not found

$\{Initialize\}$ $left := 1; right := n$

While $left < right$
$mid := \lfloor (left + right)/2 \rfloor$
If $x > a_{mid}$ **then** $left := mid$ **else** $right := mid$
Continue with next iteration of while-loop.

If $x = a_{left}$ **then** $loc := left$ **else** $loc := 0$
Return (loc)

3.2 GROWTH OF FUNCTIONS

DEF: Let f and g be functions $\mathbb{R} \to \mathbb{R}$. Then f is ***asymptotically dominated*** by g if

$$(\exists K \in \mathbb{R})\,(\forall x > K)\,[f(x) \leq g(x)]$$

NOTATION: $f \preceq g$.

Remark: This means that there is a location $x = K$ on the x-axis, after which the graph of the function g lies above the graph of the function f.

BIG OH CLASSES

DEF: Let f and g be functions $\mathbb{R} \to \mathbb{R}$. Then f is in the ***class*** $\mathcal{O}(g)$ ("***big-oh of g***") if

$$(\exists C \in \mathbb{R})\,[f \preceq Cg]$$

NOTATION: $f \in \mathcal{O}(g)$.

DISAMBIGUATION: Properly understood, $\mathcal{O}(g)$ is the class of all functions that are asymptotically dominated by any multiple of g.

TERMINOLOGY NOTE: The idiomatic phrase

"f is big-oh of g"

makes sense if one imagines either
that the word "in" precedes the word "big-oh",
or that "big-oh of g" is an adjective.

Example 3.2.1: $4n^2 + 21n + 100 \in \mathcal{O}(n^2)$

Pf: First suppose that $n \geq 0$. Then

$$\begin{aligned} 4n^2 + 21n + 100 &\leq 4n^2 + 24n + 100 \\ &\leq 4(n^2 + 6n + 25) \\ &\leq 8n^2 \text{ which holds whenever} \end{aligned}$$

$n^2 \geq 6n + 25$, which holds whenever

$n^2 - 6n + 9 \geq 34$, which holds whenever

$n - 3 \geq \sqrt{34}$, which holds whenever $n \geq 9$.

Thus,

$$(\forall n \geq 9)[4n^2 + 21n + 100 \leq 8n^2] \qquad \diamondsuit$$

Remark: We notice that n^2 itself is asymptotically dominated by $4n^2 + 21n + 100$. However, we proved that $4n^2 + 21n + 100$ is asymptotically dominated by $8n^2$, a multiple of n^2.

WITNESSES

This operational definition of membership in a big-oh class makes the definition of asymptotic dominance explicit.

DEF: Let f and g be functions $\mathbb{R} \to \mathbb{R}$. Then f is in the ***class*** $\mathcal{O}(g)$ (“***big-oh of g***”) if

$$(\exists C \in \mathbb{R})\,(\exists K \in \mathbb{R})\,(\forall x > K)\,[f(x) \leq Cg(x)]$$

DEF: In the definition above, the multiplier C and the location K on the x-axis after which $Cg(x)$ dominates $f(x)$ are called the ***witnesses*** to the relationship $f \in \mathcal{O}(g)$.

Example 3.2.1, continued: The values

$$C = 8 \quad \text{and} \quad K = 9$$

are witnesses to the relationship

$$4n^2 + 21n + 100 \in \mathcal{O}(n^2)$$

Larger values of C and K could also serve as witnesses. However, a value of C less than or equal to 4 could not be a witness.

CLASSROOM EXERCISE

If one chooses the witness $C = 5$, then $K = 30$ could be a co-witness, but $K = 9$ could not.

Lemma 3.2.1. $(x+1)^n \in \mathcal{O}(x^n)$.

Pf: Let C be the largest coefficient in the (binomial) expansion of $(x+1)^n$, which has $n+1$ terms. Then

$$(x+1)^n \leq C(n+1)x^n \qquad \diamondsuit$$

Example 3.2.2: The proof of Lemma 3.2.1 uses the witnesses

$$C = \binom{n}{\lfloor \frac{n}{2} \rfloor} \quad \text{and} \quad K = 0$$

Theorem 3.2.2. *Let $p(x)$ be any polynomial of degree n. Then $p(x) \in \mathcal{O}(x^n)$.*

Pf: Apply the method of Lemma 3.2.1. $\diamondsuit$

Example 3.2.3: $100n^5 \in \mathcal{O}(e^n)$. Observing that $n = e^{\ln n}$ inspires what follows.

Pf: Taking the upper Riemann sum with unit-sized intervals for $\ln x = \int_1^n \frac{dx}{x}$ implies for $n > 1$ that

$$\begin{aligned}
\ln(n) &< \frac{1}{1} + \frac{1}{2} + \cdots + \frac{1}{n} \\
&\le \left(\frac{1}{1} + \cdots + \frac{1}{5}\right) + \frac{1}{6} + \cdots + \frac{1}{n} \\
&\le \left(\frac{1}{1} + \cdots + \frac{1}{5}\right) + \frac{1}{6} + \cdots + \frac{1}{6} \\
&\le 5 + \frac{n-5}{6}
\end{aligned}$$

Therefore, $6 \ln n \le n + 25$, and accordingly,

$$100n^5 = 100 \cdot e^{5 \ln n} < 100 \cdot e^{n+25} < e^{32} \cdot e^n$$

We have used the witnesses $C = e^{32}$ and $K = 0$. $\diamondsuit$

Example 3.2.4: $2^n \in \mathcal{O}(n!)$.

Pf:

$$\begin{aligned}
\overbrace{2 \cdot 2 \cdots 2}^{n \text{ times}} &= 2 \cdot 1 \cdot \overbrace{2 \cdot 2 \cdots 2}^{n-1 \text{ times}} \\
&\le 2 \cdot 1 \cdot 2 \cdot 3 \cdots n = 2n!
\end{aligned}$$

We have used the witnesses $C = 2$ and $K = 0$. $\diamondsuit$

BIG-THETA CLASSES

DEF: Let f and g be functions $\mathbb{R} \rightarrow \mathbb{R}$. Then f is ***in the class*** $\Theta(g)$ ("***big-theta of g***") if $f \in \mathcal{O}(g)$ and $g \in \mathcal{O}(f)$.

3.3 COMPLEXITY

DISAMBIGUATION: In the early 1960's, Chaitin and Kolmogorov used ***complexity*** to mean measures of complicatedness. However, most theoretical computer scientists have used it in a jargon sense that means measures of resource consumption.

DEF: ***Algorithmic time-complexity measures*** estimate the time or the number of computational steps required to execute an algorithm, given as a function of the size of the input.

TERMINOLOGY: The resource for a complexity measure is implicitly time, unless space or something else is specified.

DEF: A ***worst-case complexity measure*** estimates the time required for the most time-consuming input of each size.

DEF: An ***average-case complexity measure*** estimates the average time required for input of each size.

Example 3.3.1: In searching and sorting, complexity is commonly measures in terms of the number of comparisons, since total computation time is typically a multiple of that.

Algo 3.1.1: Find Maximum

Input: unsorted array of integers $a_1, a_2, \ldots, a_n$
Output: largest integer in array

$\{Initialize\}$ $max := a_1$

For $i := 2$ **to** n
  If $max < a_i$ **then** $max := a_i$
  Continue with next iteration of for-loop.

Return (max)

Big-Oh:

Always takes $n - 1$ comparisons.

Time complexity is in $\mathcal{O}(n)$.

Example 3.3.2:

Algo 3.1.2: Unsorted Sequential Search

Input: unsorted array of integers $a_1, a_2, \ldots, a_n$
target value x
Output: subscript of entry equal to target value,
or 0 if not found

$\{Initialize\}$ $i := 1$

While $i \leq 2$ **and** $x \neq a_i$
$i := i + 1$
Continue with next iteration of while-loop.

If $i \leq n$ **then** $loc := i$ **else** $loc := 0$
Return (loc)

Target in or not in Array:

Worst case takes n comparisons.

Average case takes $n/2$ comparisons.

Target not in Array:

Every case takes n comparisons.

Big-Oh:

Time complexity is in $\mathcal{O}(n)$.

Example 3.3.3:

Algo 3.1.3: Sorted Sequential Search

Input: sorted array of integers $a_1, a_2, \ldots, a_n$
target value x
Output: subscript of entry equal to target value,
or 0 if not found

$\{Initialize\}$ $i := 1$

While $i \leq n$ **and** $x < a_i$
  $i := i + 1$
  Continue with next iteration of while-loop.

If $(i \leq n$ **cand** $x = a_i)$ **then** $loc := i$ **else** $loc := 0$
Return (loc)

Target in or not in Array:

Worst case takes n comparisons.

Average case takes $n/2$ comparisons.

Big-Oh:

Time complexity is in $\mathcal{O}(n)$.

Example 3.3.4:

Algo 3.1.4: Two-level Search

Input: sorted array of integers $a_1, a_2, \ldots, a_n$
target value x
Output: subscript of entry equal to target value,
or 0 if not found

$\{Initialize\}$ $i := 10$
$\{$*Find target sublist of 10 entries*$\}$
While $i \leq 2$ **and** $x \leq a_i$
  $i := i + 10$
  Continue with next iteration of while-loop.
$\{$*Linear search target sublist of 10 entries*$\}$
$\{Initialize\}$ $j := i - 9$
While $j \leq i$ **and** $x < a_j$
  $j := j + 1$
  Continue with next iteration of while-loop.
If ($j \leq n$ **cand** $x + a_j$) **then** $loc := j$ **else** $loc := 0$
Return (loc)

Target in or not in Array:

Worst case takes $(n/10) + 10$ comparisons.

Big-Oh: Time complexity is in $\mathcal{O}(n)$.

To optimize the two-level search, minimize

$$\frac{n}{x} + x$$

as in differential calculus.

$$\frac{-n}{x^2} + 1 = 0 \quad \Rightarrow \quad x = \sqrt{n}$$

Target in or not in Array:

Worst case takes $2\sqrt{n}$ comparisons.

Big-Oh: Time complexity is in $\mathcal{O}(\sqrt{n})$.

Increasing to k levels further decreases the execution time to $\mathcal{O}(\sqrt[k]{n})$,
provided that k is not too large.

Example 3.3.5:

Algo 3.1.5: Binary Search

Input: sorted array of integers $a_1, a_2, \ldots, a_n$
target value x
Output: subscript of entry equal to target value,
or 0 if not found

$\{Initialize\}$ $left := 1; right := n$

While $left < right$
$mid := \lfloor (left + right)/2 \rfloor$
If $x > a_{mid}$ **then** $left := mid$ **else** $right := mid$
Continue with next iteration of while-loop.

If $x = a_{left}$ **then** $loc := left$ **else** $loc := 0$
Return (loc)

Target in or not in Array:

Every case takes $\lg n$ comparisons.

Big-Oh: Time complexity is in $\mathcal{O}(\lg n)$.

COMPLEXITY JARGON

DEF: A problem is ***solvable*** if it can be solved by an algorithm.

Example 3.3.6: Alan Turing defined the ***halting problem*** to be that of deciding whether a computational procedure (e.g., a program) halts for all possible input. He proved that the halting problem is unsolvable.

DEF: A problem is in ***class* P** if it is solvable by an algorithm that runs in polynomial time.

DEF: A problem is ***tractable*** if it is in class **P**.

DEF: A problem is in ***class* NP** if an algorithm can decide in polynomial time whether a putative solution is really a solution.

Example 3.3.7: The problem of deciding whether a graph is 3-colorable is in class **NP**. It is believed not to be in class **P**.

3.4 THE INTEGERS AND DIVISION

In mathematics, specifying an axiomatic model for a system precedes all discussion of its properties. The number system serves as a foundation for many other mathematical systems.

Elementary school students learn algorithms for the arithmetic operations without ever seeing a definition of a "number" or of the operations that these algorithms are modeling.

These coursenotes precede discussion of division by the construction of the number system
(see Appendix A1 of Rosen, 6th Edition)
and of the usual arithmetic operations.

AXIOMS for the NATURAL NUMBERS

DEF: The ***natural numbers*** are a mathematical system

$$\{\mathbb{N},\ 0 \in \mathbb{N},\ s : \mathbb{N} \to \mathbb{N}\}$$

with a number ***zero*** 0 and a ***successor*** operation $s : \mathbb{N} \to \mathbb{N}$ such that

(1) $(\nexists n)\,[0 = s(n)]$.
Zero is not the successor of any number.

(2) $(\forall m, n \in \mathbb{N})\,[m \neq n \Rightarrow s(m) \neq s(n)]$.
Different numbers cannot have the same successor.

(3) Given a subset $S \subseteq \mathbb{N}$ with $0 \in S$

$$\text{if } (\forall n \in S)\,[s(n) \in S] \text{ then } S = \mathbb{N}$$

Given a subset S of the natural numbers, suppose that it contains the number 0, and suppose that whenever it contains a number, it also contains the successor of that number. Then $S = \mathbf{N}$.

Remark: Axiom (1) implies that $\mathbb{N}$ has at least one other number, namely, the successor of zero. Let's call it ***one***. Using Axioms (1) and (2) together, we conclude that $s(1) \notin \{0, 1\}$. Etc.

ARITHMETIC OPERATIONS

DEF: The ***predecessor*** of a natural number n is a number m such that $s(m) = n$.
NOTATION: $p(n)$.

DEF: ***Addition*** of natural numbers.

$$n + m = \begin{cases} n & \text{if } m = 0 \\ s(n) + p(m) & \text{otherwise} \end{cases}$$

DEF: ***Ordering*** of natural numbers.

$$n \geq m \text{ means } \begin{cases} \quad m = 0 \quad \text{or} \\ \quad p(n) \geq p(m) \end{cases}$$

DEF: ***Multiplication*** of natural numbers.

$$n \times m = \begin{cases} 0 & \text{if } m = 0 \\ n + n \times p(m) & \text{otherwise} \end{cases}$$

OPTIONAL:

(1) Define ***exponentiation***.

(2) Define ***positional representation*** of numbers.

(3) Verify that the usual base-ten methods for addition, subtraction, etc. produce correct answers.

DIVISION

DEF: Let n and d be integers with $d \neq 0$. Then we say that d ***divides*** n if there exists a number q such that $n = dq$. NOTATION: $d\backslash n$.

DEF: The integer d is a ***factor*** of n or a ***divisor*** of n if $d\backslash n$.

DEF: A divisor d of n is ***proper*** if $d \neq n$.

DEF: The number 1 is called a ***trivial divisor***.

DIVISION THEOREM

Theorem 3.4.1. *Let n and d be positive integers. Then there are unique nonnegative integers q and $r < d$ such that $n = qd + r$.*

TERMINOLOGY: $n =$ ***dividend***, $d =$***divisor***, $q =$ ***quotient***, and $r =$ ***remainder***.

Algo 3.4.1: Division Algorithm

Input: dividend $n > 0$ and divisor $d > 0$
Output: quotient q and remainder $r : 0 \leq r < d$

$q := 0; r := n$
While $n \geq d$
    $q := q + 1$
    $r := r - d$
    Continue with next iteration of while-loop.
Return (quotient: d; remainder: r)

Time-Complexity: $\mathcal{O}(n/d)$.

Remark: ***Positional representation*** uses only $\Theta(\log n)$ digits to represent a number. This facilitates a faster algorithm to calculate division.

Example 3.4.1: divide 7 into 19

n	d	q
19	7	0
12	7	1
5	7	2

MODULAR ARITHMETIC

DEF: Let n and $m > 0$ be integers. The ***residue*** of dividing n by m is, if $n \geq 0$, the remainder, or otherwise, the smallest nonnegative number obtainable by adding an integral multiple of m.

DEF: Let n and $m > 0$ be integers. Then $n \bmod m$ is the residue of dividing n by m. This is called the ***mod operator***.

Prop 3.4.2. *Let n and $m > 0$ be integers. Then $n - (n \bmod m)$ is a multiple of m.*

Example 3.4.2:

$$\begin{aligned} 19 \bmod 7 &= 5 \\ 17 \bmod 5 &= 2 \\ -17 \bmod 5 &= 3 \end{aligned}$$

DEF: Let b, c, and $m > 0$ be integers. Then b is ***congruent to*** c ***modulo*** m if m divides $b - c$.
NOTATION: $b \equiv c \mod m$.

Theorem 3.4.3. *Let $a, b, c, d, m > 0$ be integers such that $a \equiv b \mod m$ and $c \equiv d \mod m$. Then*

$$a + c \equiv b + d \bmod m \quad \text{and} \quad ac \equiv bd \bmod m$$

Pf: Straightforward. ◊

CAESAR ENCRYPTION

DEF: ***Monographic substitution*** is enciphering based on permutation of an alphabet

$$\pi : A \to A$$

Ciphertext is obtained from plaintext by replacing each occurrence of each letter by its substitute.

letter	A	B	C	D	E	F	⋯	X	Y	Z
subst	Q	W	E	R	T	Y	⋯	B	N	M

DEF: A monographic substitution cipher is called ***cyclic*** if the letters of the alphabet are represented by numbers 0, 1, ..., 25 and there is a number m such that $\pi(n) = m + n \mod 26$.

An ancient Roman parchment is discovered with the following words:

HW WX EUXWH

What can it possibly mean?

Hint: Julius Caesar encrypted military messages by cyclic monographic substitution.

3.5 PRIMES AND GCD'S

DEF: An integer $p \geq 2$ is ***prime*** if p has no nontrivial proper divisors, and ***composite*** otherwise.

Algo 3.5.1: Naive Primality Algorithm

Input: positive integer n
Output: smallest nontrivial divisor of n

For $d := 2$ **to** n
  If $d \backslash n$ **then exit**
  Continue with next iteration of for-loop.

Return (d)

Time-Complexity: $\mathcal{O}(n)$.

Theorem 3.5.1. *Let n be a composite number. Then n has a divisor d such that $1 < d \leq \sqrt{n}$.*

Pf: Straightforward. ◊

Algo 3.5.2: Less Naive Primality Algorithm

Input: positive integer n
Output: smallest nontrivial divisor of n

For $d := 2$ **to** $\sqrt{n}$
  If $d \backslash n$ **then exit**
  Continue with next iteration of for-loop.

Return (d)

Time-Complexity: $\mathcal{O}(\sqrt{n})$.

Example 3.5.1: Primality Test 731.

Upper Limit: $\lfloor \sqrt{731} \rfloor = 27$, since $729 = 27^2$.

$\neg(2 \backslash 731)$: leaves $3, 5, 7, 9, 11, \ldots, 25, 27$ 13 cases

$\neg(3, 5, 7, 9, 11, 13, 15 \backslash 731)$: however, $17 \backslash 731$

AHA: $731 = 17 \times 43$.

N.B. To accelerate testing, divide only by primes 2, 3, 5, 7 ,11, 13, 17.

MERSENNE PRIMES

Prop 3.5.2. *If $m, n > 1$ then $2^{mn} - 1$ is not prime.*

Pf:

$$\begin{array}{lrrrrr}
 & & 2^{m(n-1)} & +\cdots & +2^m & +1 \\
\text{(times)} & & & \times & 2^m & -1 \\
\hline
 & 2^{mn} & +2^{m(n-1)} & +\cdots & +2^m & \\
 & & -2^{m(n-1)} & -\cdots & -2^m & -1 \\
\hline
 & 2^{mn} & & & & -1
\end{array}$$

Example 3.5.2:

$$\begin{aligned}
2^6 - 1 = 2^{3\cdot 2} - 1 \\
&= (2^{3\cdot 1} + 1)(2^3 - 1) = 9 \cdot 7 = 63 \\
&= 2^{2\cdot 3} - 1 \\
&= (2^{2\cdot 2} + 2^{2\cdot 1} + 1)(2^2 - 1) = 21 \cdot 3 = 63
\end{aligned}$$

Mersenne studied the CONVERSE of Prop 3.5.2:

Is $2^p - 1$ prime when p is prime?

DEF: A ***Mersenne prime*** is a prime number of the form $2^p - 1$, where p is prime.

Example 3.5.3: primality of $2^p - 1$ vsa

prime p	$2^p - 1$	Mersenne?
2	$2^2 - 1 = 3$	yes (1)
3	$2^3 - 1 = 7$	yes (2)
5	$2^5 - 1 = 31$	yes (3)
7	$2^7 - 1 = 127$	yes (4)
11	$2^{11} - 1 = 2047 = 23 \cdot 89$	no
11213	$2^{11213} - 1$	yes (23)
19937	$2^{19937} - 1$	yes (24)
3021377	$2^{3021377} - 1$	yes (37)

Fundamental Theorem of Arithmetic

Theorem 3.5.3. *Every positive integer can be written uniquely as the product of nondecreasing primes.*

Pf: §3.5 proves this difficult lemma:
if a prime number p divides a product mn of integers, then it must divide either m or n. $\diamondsuit$

Example 3.5.4: $720 = 2^4\, 3^2\, 5^1$ is written as a ***prime power factorization***.

GREATEST COMMON DIVISORS

DEF: The ***greatest common divisor*** of two integers m, n, not both zero, is the largest positive integer d that divides both of them.
NOTATION: $\gcd(m, n)$.

Algo 3.5.3: Naive GCD Algorithm

Input: integers $m \leq n$ not both zero
Output: $\gcd(m, n)$

$g := 1$
For $d := 1$ **to** m
  If $d \backslash m$ **and** $d \backslash n$ **then g:=d**
  Continue with next iteration of for-loop.
Return (g)

Time-Complexity: $\Omega(m)$.

Algo 3.5.4: Primepower GCD Algorithm

Input: integers $m \leq n$ not both zero
Output: $\gcd(m, n)$
(1) Factor $m = p_1^{a_1} p_2^{a_2} \cdots p_r^{a_r}$ into prime powers.
(2) Factor $n = p_1^{b_1} p_2^{b_2} \cdots p_r^{b_r}$ into prime powers.
(3) $g := p_1^{\min(a_1,b_1)} p_2^{\min(a_2,b_2)} \cdots p_r^{\min(a_r,b_r)}$
Return (g)

Time-Complexity:
depends on time needed for factoring

DEF: The ***least common multiple*** of two positive integers m, n is the smallest positive integer d divisible by both m and n.
NOTATION: $\mathrm{lcm}(m, n)$.

Theorem 3.5.4. *Let m and n be positive integers. Then $mn = \gcd(m, n)\mathrm{lcm}(m, n)$.*

Pf: The Primepower LCM Algorithm uses max instead of min. $\diamondsuit$

RELATIVE PRIMALITY

DEF: Two integers m and n, not both zero, are ***relatively prime*** if $\gcd(m, n) = 1$.
NOTATION: $m \perp n$.

Proposition 3.5.5. *Two numbers are relatively prime if no prime has positive exponent in both their prime power factorizations.*

Pf: Immediate from the definition above. $\diamondsuit$

Remark: Proposition 3.5.5 is what motivates the notation $m \perp n$. Envision the integer n expressed as a tuple in which the kth entry is the exponent (possibly zero) of the kth prime in the prime power factorization of n. The dot product of two such representations is zero iff the numbers represented are relatively prime. This is analogous to orthogonality of vectors.

3.6 INTEGERS AND ALGORITHMS

We accelerate evaluation of gcd's, of arithmetic operations, and of monomials and polynomials.

POSITIONAL REPRESENTATION of INTEGERS

Arithmetic algorithms are much more complicated for numbers in positional notation than for numbers in monadic notation. However, they pay benefits in execution time.

(1) Addition algorithm execution time decreases from $\mathcal{O}(n)$ to $\mathcal{O}(\log n)$.

(2) Multiplication algorithm execution time decreases from $\mathcal{O}(nm)$ to $\mathcal{O}(\log n \log m)$.

Theorem 3.6.1. *Let $b > 1$ and $n \geq 0$ be integers. Let k be the maximum integer such that $b^k \leq n$. Then there is a unique set of nonnegative integers $a_k, a_{k-1}, \ldots, a_0 < b$ such that*

$$n = a_k b^k + a_{k-1} b^{k-1} + \cdots + a_1 b^1 + a_0$$

Pf: Apply the division algorithm to n and b to obtain a quotient and remainder a_0. Then apply the division algorithm to that quotient and b to obtain a new quotient and remainder a_1. Etc. $\Diamond$

NUMBER BASE CONVERSION

The algorithm in the proof of Theorem 3.6.1 provides a method to convert any positive integer from one base to another.

Example 3.6.1: Convert 1215_{10} to base-7.

n	d	q	r
1215	7	173	4
173	7	24	5
24	7	3	3
3	7	0	3

Solution: 3354_7

EVALUATION OF MONOMIALS

Example 3.6.2: Calculate 13^n, e.g. 13^{19}.

Usual method: $13 \times 13 \times 13 \times \cdots \times 13$
time $= \Theta(n)$.

Better method:
$13, 13^2, 13^4, 13^8, 13^{16}$ takes $\Theta(\log n)$ steps
$13 \times 13^2 \times 13^{16}$ takes $\Theta(\log n)$ steps

EVALUATION OF POLYNOMIALS

Evaluate $f(x) = a_n x^n + a_{n-1} x^{n-1} + ... + a_1 x + a_0$

Usual method of evaluation takes $\Theta(n)$:
n multiplications to calculate n powers of x
n multpplications by coefficients
n additions

Horner's method (due to __________):
$a_n x + a_{n-1}$
$(a_n x + a_{n-1})x + a_{n-2}$ etc.

requires only n multiplications and n additions.

EUCLIDEAN ALGORITHM

Lemma 3.6.2. *Let* $d \backslash m$ *and* $d \backslash n$.
Then $d \backslash m - n$ *and* $d \backslash m + n$.

Pf: Suppose that $m = dp$ and $n = dq$.
Then $m - n = d(p - q)$ and $m + n = d(p + q)$. $\diamondsuit$

Corollary 3.6.3. $\gcd(m, n) = \gcd(m - n, n)$.
Pf: *In three steps.*

A1. $\gcd(m, n)$ is a common div of $m - n$ and n, and $\gcd(m - n, n)$ is a common div of m and n.

Pf. Both parts by Lemma 3.6.2.

A2. $\gcd(m, n) \leq gcd(m - n, n)$
and $\gcd(m - n, n) \leq gcd(m, n)$.

Pf. Both parts by A1 and def of gcd ("greatest").

A3. $\gcd(m, n) = \gcd(m - n, n)$.
Pf. Immediate from A2. $\diamondsuit$ Cor 3.6.3

Cor 3.6.4. $\gcd(m, n) = \gcd(n, m \bmod n)$.

Pf: The number $m \bmod n$ is obtained from m by subtracting a multiple of n. Iteratively apply Cor 3.6.3. $\diamondsuit$

Algo 3.6.1: Euclidean Algorithm

Input: positive integers $m \geq 0, n > 0$
Output: $\gcd(n, m)$

If $m = 0$ **then return**(n)
  else return $\gcd(m, n \bmod m)$

Time-Complexity: $\mathcal{O}(\log(\min(n, m)))$.
Much better than Naive GCD algorithm.

Example 3.6.3: Euclidean Algorithm

$$\begin{aligned}
\gcd(210, 111) &= \gcd(111, 210 \bmod 111) = \\
\gcd(111, 99) &= \gcd(99, 111 \bmod 99) = \\
\gcd(99, 12) &= \gcd(12, 99 \bmod 12) = \\
\gcd(12, 3) &= \gcd(3, 12 \bmod 3) = \\
\gcd(3, 0) &= 3
\end{aligned}$$

Example 3.6.4: Euclidean Algorithm

$$
\begin{aligned}
\gcd(42,26) &= \gcd(26, 42 \bmod 26) = \\
\gcd(26,16) &= \gcd(16, 26 \bmod 16) = \\
\gcd(16,10) &= \gcd(10, 16 \bmod 10) = \\
\gcd(10,6) &= \gcd(6, 10 \bmod 6) = \\
\gcd(6,4) &= \gcd(4, 6 \bmod 4) = \\
\gcd(4,2) &= \gcd(2, 4 \bmod 2) = \\
\gcd(2,0) &= 2
\end{aligned}
$$

3.7 NUMBER THEORY

EXTENDED EUCLIDEAN ALGORITHM

Given two integers a and b, the ***extended Euclidean algorithm*** produces numbers s and t such that $sa + tb = gcd(a, b)$. We describe it by example.

Example 3.7.1: Euclidean Algorithm

$$
\begin{aligned}
312 &= 2 \cdot 111 + 90 \\
111 &= 1 \cdot 90 + 21 \\
90 &= 4 \cdot 21 + 6 \\
21 &= 3 \cdot 6 + 3 \\
6 &= 2 \cdot 3 + 0 \quad \text{now start back-substitution} \\
3 &= 21 - 3 \cdot 6 \\
&= 21 - 3 \cdot [90 - 4 \cdot 21] = 13 \cdot 21 - 3 \cdot 90 \\
&= 13 \cdot [111 - 90] - 3 \cdot 90 = 13 \cdot 111 - 16 \cdot 90 \\
&= 13 \cdot 111 - 16 \cdot [312 - 2 \cdot 111] \\
&= \underline{45} \cdot 111 - \underline{16} \cdot 312 = 4995 - 4992 = 3
\end{aligned}
$$

EXPONENTIATION MOD a PRIME

Problem: Evaluate $x^k \bmod p$, with p prime.

FACT 1: $x^k \bmod n = (x \bmod n)^k \bmod n$.

Pf: If $x = qn + (x \bmod n)$, then

$$
\begin{aligned}
x^k \bmod n &= (qn + (x \bmod n))^k \bmod n \\
&\quad \text{do a binomial expansion} \\
&= Bn + (x \bmod n)^k \bmod n \\
&= (x \bmod n)^k \bmod n \quad \diamondsuit
\end{aligned}
$$

Example 3.7.2: $12^3 \bmod 5 = 1728 \bmod 5 = 3$

$12^3 \bmod 5 = 2^3 \bmod 5 = 3$

FACT 2. Fermat's Little Theorem

Let p be prime. Then $x^{p-1} = 1 \bmod p$.

Pf: See Exercise 17 of §2.6. $\diamondsuit$

Example 3.7.3: $2^6 \bmod 7 = 64 \bmod 7 = 1$

$7^4 \bmod 5 = 2401 \bmod 5 = 1$

Example 3.7.4: Calculate $16^{20} \bmod 7$.

Using fast monomial evaluation, this looks like $\lg n$ mults and 1 division. Not bad, unless you want the answer by hand computation.

Pure Algebra to the Rescue

$$\begin{aligned} 16^{20} \bmod 7 &= 2^{20} \bmod 7 \quad \text{by FACT 1} \\ &= 2^{2} \bmod 7 \quad \text{by FACT 2} \\ &= 4 \end{aligned}$$