Chapter 3: algorithmic basics

here

• some very elementary algorithms
• big-$O$, other big things, and complexity
Basic algorithms
consider intuitive algorithm
    that solve simple problems

goal:
    get first grasp of *complexity* of algorithms:
    algorithm behavior with respect to
    usage of time and space ("memory")
    depending on the problem "size"

why?
    to better understand algorithm scalability
    and the "difficulty" of the problems
    (like matrix multiplication: how does effort grow?)
What is an “algorithm”?

“finite set of precise (?) instructions to perform a specified task”:

• to perform a certain computation
• to solve a certain problem
• to cook a certain dish
• to reach a certain destination

needs to satisfy various obvious requirements:

• well-defined input/output behavior
• well-defined steps that always work
• it terminates (“finite” and “effective”)
• must be sufficiently general

(no attempt at a formal definition)
First basic problem: finding the maximum

given set $A = \{a_1, a_2, a_3, \ldots, a_n\}$
the problem: find (index of) “largest” element
(largest with respect to some ordering)

“best solution” minimizes the “cost”:
number of comparisons between elements of $A$
set is “unordered collection”
$\Rightarrow$ as is, all we can do is inspect all elements
(see book page 195/169 for “pseudocode”)
$\Rightarrow n - 1 = |A| - 1$ comparisons

cost is linear function of $|A|$: linear algorithm
(size of elements of $A$ not taken into account in cost!)
Another basic problem: searching

given set \( A = \{a_1, a_2, a_3, \ldots, a_n\} \) and some \( x \)
the problem: if possible, locate \( x \) in \( A \)
(if \( x \in A \) return \( i \) such that \( a_i = x \), else return 0)

again, we like to minimize the cost:
  number of comparisons between \( a \in A \) and \( x \)
set is still an “unordered collection”
⇒ as is, possibly compare \( x \) to all \( a \in A \)
  (see book page 196/170 for pseudocode)
⇒ in the worst case: \( n = |A| \) comparisons

cost is linear function of \( |A| \): linear search

(size of elements of \( A \) again not taken into account in cost)
Can we search \( x \) in \( A \) faster?

only if more is known about \( A \) or \( x \)

\[
A = \{a_1, a_2, a_3, \ldots, a_n\}
\]

could be sorted,

\[
a_1 < a_2 < a_3 < \ldots < a_n:
\]

with \( m = \lfloor n/2 \rfloor \), **compare \( x \) and \( a_m \)**

this suffices to remove \( \{a_1, a_2, a_3, \ldots, a_{m-1}\} \) or \( \{a_{m+1}, a_{m+2}, a_{m+3}, \ldots, a_n\} \) from consideration

\( \Rightarrow \) **cost only 1** to divide problem size by two

\( \Rightarrow \) **total number of comparisons**: about \( \log_2(n) \)

\( \Rightarrow \)**logarithmic search**

(note: finding maximum in \( A \) is now for free)
Another way to search $x$ in $S$ faster
there may be an “index function” $i : A \rightarrow \mathbb{N}_{\geq 0}$
such that if $x \in A$ then $a_{i(x)} = x$

⇒ cost to locate $x$ is at most one comparison
   (plus evaluation of $i(x)$)

⇒ constant cost

seen three types of cost functions so far:
• constant
• logarithmic in problem size
• linear in problem size
all scale well for growing problem sizes
But what about sorting?

the problem:

given a finite sequence of items, “sort” it

intuitively clear what is meant:

input
    25, 16, 32, 33, 8, 3, 17, 6

should be transformed into
    3, 6, 8, 16, 17, 25, 32, 33
Bubble sort

simple iterative solution to sort \( a_1, a_2, a_3, \ldots, a_n \)

for \( i = n \) downto 2:

put max(\( a_1, a_2, a_3, \ldots, a_i \)) in \( a_i \), at cost \( i - 1 \):

for \( k = 1 \) to \( i - 1 \):

if \( a_k > a_{k+1} \) then “swap” \( a_k \) and \( a_{k+1} \)

overall cost \( \sum_{i=2}^{n} (i - 1) = (n - 1)n / 2 \)

\[ \Rightarrow \text{cost function quadratic in problem size} \]

but, how does one “swap” elements?

and, what are we actually counting in our cost?
Other naïve iterative approaches to sorting

- “selection sort”
  for $i = 1$ to $n-1$:
  put $\min(a_i, a_{i+1}, \ldots, a_n)$ in
  $i$th position of $(a_1, a_2, a_3, \ldots, a_n)$

- “insertion sort”
  for $i = 2$ to $n$:
  insert $a_i$ at proper place in
  already sorted list $a_1, a_2, a_3, \ldots, a_{i-1}$

all these approaches have essentially the
same cost function as bubble sort:
  i.e., quadratic in problem size
Other naïve iterative approaches to sorting

- “selection sort”
  for $i = 1$ to $n-1$:
    put $\min(a_i, a_{i+1}, \ldots, a_n)$ in
    $i$th position of $(a_1, a_2, a_3, \ldots, a_n)$

- “insertion sort”
  for $i = 2$ to $n$:
    insert $a_i$ at proper place in
    already sorted list $a_1, a_2, a_3, \ldots, a_{i-1}$

all these approaches have essentially the
same cost function as bubble sort: do they?
  i.e., quadratic in problem size
Faster sorting?

• “bucket sort”
  suppose for each $a_i$ its proper location is a function of just $a_i$:
  to sort $a_1, a_2, a_3, \ldots, a_n$ it suffices to call that function $n$ times:
  linear sorting

• in general:
  faster methods use divide and conquer and smart data structures
Questions?

concludes 1st section of Chapter 3
(with the exception of “greedy”,
which we postpone)
Big-\(O\), Big-Omega, and Big-Theta

motivation:
want to express how the time required by an algorithm depends on the size of the problem

two extremes:
• precise count of everything involved (computer instructions, disk accesses, …) as a function of size:
  inconvenient, not always well-defined
• “it took a few seconds on my laptop” not sufficiently informative:
  what if size doubles?
Example

assume it took $s$ seconds to find
the maximum among $n$ unsorted items

how to predict the time required to find the
maximum among $2n$, $3n$, or $m$ items?

finding the maximum takes linear time
\[ \Rightarrow \text{reasonable to predict} \]
$2s$, $3s$, and $(m/n)s$ seconds
Another example

assume that, for some large $n$, sorting $n$ items using bubble sort took $s$ seconds

how to predict the time required to sort $2n$, $3n$, or $m$ items using bubble sort?

sorting using bubble sort is quadratic

⇒ reasonable to predict $2^2s$, $3^2s$, and $(m/n)^2s$ seconds
Observations on run times

let \( f(n) \) estimate time to solve problem of size \( n \)

if \( f(n) = g(n) + h(n) + \ldots + t(n) \)

for functions \( g, h, \ldots, t: \mathbb{N} \rightarrow \mathbb{R} \)

then the “ultimately largest” of \( g, h, \ldots, t \) determines \( f \)'s behavior when \( n \) gets large

example:

let \( f(n) = 2n^2 + 240n + 9600 \)

then \( g(n) = 2n^2, \ h(n) = 240n, \ t(n) = 9600 \)

for small \( n \): \( t(n) \) most significant

then \( h(n) \) takes over

but ultimately only \( g(n) \) is relevant
Observations on run times

let $f(n)$ estimate time to solve problem of size $n$

if $f(n) = g(n) + h(n) + \ldots + t(n)$

for functions $g, h, \ldots, t: \mathbb{N} \rightarrow \mathbb{R}$

then the “ultimately largest” of $g, h, \ldots, t$ determines $f$’s behavior when $n$ gets large

let $g(n)$ be $f(n)$’s “ultimately most relevant part”

then $f(n)$’s growth rate is independent of multiplicative constants in $g(n)$:

$$\frac{g(m)}{g(n)} = \frac{cg(m)}{cg(n)}$$
Consequences

When considering a runtime function $f(n)$

- Focus on part that grows “fastest” (for $n \to \infty$)
- Forget about multiplicative constants

Examples:

- $f(n) = 2n^2 + 240n + 9600$
  
  $2n^2$ determines behavior, simplify to just $n^2$

- $r(n) = 0.0001n^2 + 24000n + 9600^{9600}$
  
  again, only the $n^2$ is relevant

- $s(n) = 31(\sqrt{n})\log(n) + n\log_{10}(n) + 167n$
  
  $n\log_{10}(n)$ determines behavior: $n\log(n)$

  $f(n)$ is $O(n^2)$, $r(n)$ is $O(n^2)$, $s(n)$ is $O(n\log(n))$
**Big-O**

Let $f, g : \mathbb{R} \rightarrow \mathbb{R}$

We say that “$f(x)$ is $O(g(x))$” if there are constants $C$ and $k$ such that

$$\forall x > k \quad |f(x)| \leq C|g(x)|$$

- $C$ and $k$ are called the *witnesses*
- “$f(x)$ is big-$O$ of $g(x)$”
- “$f$ is big-$O$ of $g$”

**Note:**

big-$O$ takes “focus” and “forget” into account

“$k$”       “$C$”
Earlier examples

\[ f(n) = 2n^2 + 240n + 9600 \text{ is } O(n^2) \]
\[ C = 4, \ k = 240 \text{ are witnesses} \]
\[ \forall \ n > 240 \ |f(n)| \leq 4|n^2| \]

\[ r(n) = 0.0001n^2 + 24000n + 9600^{9600} \text{ is } O(n^2) \]
\[ C = 3, \ k = 9600^{4800} \text{ are witnesses} \]
\[ \forall \ n > 9600^{4800} \ |r(n)| \leq 3|n^2| \]

\[ s(n) = 31(\sqrt{n})\log(n) + n\log_{10}(n) + 167n \text{ is } O(n\log(n)) \]
\[ C = 2, \ k = 10^{167} \text{ are witnesses} \]
\[ \forall \ n > 10^{167} \ |s(n)| \leq 2|n\log(n)| \]
Big-$O$ facts

75 is $O(1)$ and $1$ is $O(75)$

1 is $O(n)$ but $n$ is not $O(1)$

$n$ is $O(n^2)$ but $n^2$ is not $O(n)$

$n^2$ is $O(n^2)$ and $n^2$ is $O(n^3)$

$n^2$ is $O(6n^2+n+3)$ and $6n^2+n+3$ is $O(n^2)$

$O(6n^2+n+3)$ and $O(75)$ are weird&odd, they violate “focus” and “forget”

For constants $a_i$: $\sum_{i=0}^{d} a_i n^i$ is $O(n^d)$

$\sum_{i=0}^{n} i$ is $O(n^2)$ and $\sum_{i=0}^{n} a_i i^d$ is $O(n^{d+1})$
More big-$O$ facts

$\forall u > v$, $u$, $v$ constant:

$n^v$ is $O(n^u)$ but $n^u$ is not $O(n^v)$

$\forall a > 0$, $b > 0$, $u > v$, $a$, $b$, $u$, $v$ constant:

$\log_b(n^v)$ is $O(\log_a(n^u))$

$\log_a(n^u)$ is $O(\log_b(n^v))$

and they are all $O(\log(n))$

If $f$ is $O(g)$ and $g$ is $O(h)$ then $f$ is $O(h)$
Strictly increasing big-$O$’s

- $\log(n)$ is $O(n)$ but $n$ is not $O(\log(n))$
- important: $\forall t > 0 \ \forall \varepsilon > 0 \ (\log(n))^t$ is $O(n^\varepsilon)$ (any fixed power of $\log n$ loses compared to even the tiniest power of $n$)
- $n$ is $O(n\log(n))$ but $n\log(n)$ is not $O(n)$;
- Constants $b > 1$, $d > 0$:
  - $n^d$ is $O(b^n)$ but $b^n$ is not $O(n^d)$
  - $b^n$ is $O(n!)$ but $n!$ is not $O(b^n)$
- $n!$ is $O(n^n)$ but $n^n$ is not $O(n!)$

$\Rightarrow$ strictly increasing complexities: $O(1)$, $O(\log(n))$, $O(n)$, $O(n\log(n))$, $O(n^d)$ ($d > 1$), $O(b^n)$ ($b > 1$), $O(n!)$, $O(n^n)$
Sometimes confusing big-$O$ facts

- although $n!$ is $O(n^n)$ but $n^n$ is not $O(n!)$:
  \[
  \log(n!) \text{ is } O(n \log(n)) \quad \text{and} \quad n \log(n) \text{ is } O(\log(n!))
  \]

- for constants $a > b$ and $c > 1$:
  \[
  c^{\log_a(n)} \text{ is } O(c^{\log_b(n)})
  \]
  \[
  \text{but } c^{\log_b(n)} \text{ is not } O(c^{\log_a(n)})
  \]

$\Rightarrow$ the base of the logarithm matters when the logarithm is in the exponent,
otherwise the base doesn’t matter
Proofs of some of the big-$O$ facts

- \( \log(n) \) is \( O(n) \)
  As \( n < 2^n \) (formal proof later), we have \( \log(n) < \log(2^n) = n \), so \( \log(n) \) is \( O(n) \) with witnesses \( C=k=1 \).

- \( \forall t>0 \forall \varepsilon>0 \) \( \log(n^t) \) is \( O(n^\varepsilon) \)
  Informally: \( \log(n^t) < n^{\varepsilon/t} \) for \( n \) large, so \( \log(n) < (t/\varepsilon)n^{\varepsilon/t} \) and \( (\log(n))^t < (t/\varepsilon)^t n^\varepsilon \), so \( C = (t/\varepsilon)^t \) and large \( k \).

- \( n \) is \( O(n\log(n)) \) because \( n < n\log(n) \) for \( n > e \) (so, witnesses \( C=1, k=e \))

- \( n\log(n) \) is not \( O(n) \) because \( n\log(n)/n = \log(n) > C \) for \( n > e^C \)

- \( n^k \) is \( O(b^n) \): for \( n \) large enough \( k\log_b(n) < n \), thus for \( n \) large enough \( n^k < b^n \)

- \( b^n \) is not \( O(n^k) \): for any constant \( C > 1 \) and \( n \) large enough \( n\log(b) - k\log(n) > \log(C) \), so \( b^n/n^k > C \)

- \( b^n \) is \( O(n!) \) but \( n! \) is not \( O(b^n) \): \( (1*2*...*n)/(b*b*...*b) \) has fixed number of factors < 2 and growing (with \( n \)) number of factors \( \geq 2 \).

- \( n! \) is \( O(n^n) \):
  \( n!=1*2*...*n \leq n*n*...*n=n^n \), so \( n! \) is \( O(n^n) \) with witnesses \( C=1, k=1 \).

- \( n^n \) is not \( O(n!) \)
  \( \frac{n^n}{n!} = \frac{n}{n} \cdots \frac{n}{n-1} \frac{n}{2} \frac{n}{1} > n \) for \( n > 1 \), so \( n^n > n*n! \) so that \( n^n \) cannot be \( \leq Cn! \) for all large \( n \).

- \( \log(n!) \) is \( O(n\log(n)) \):
  Because \( n! \leq n^n \), we have \( \log(n!) \leq \log(n^n) = n\log(n) \), so \( \log(n!) \) is \( O(n\log(n)) \) with witnesses \( C=1, k=1 \).

- \( n\log(n) \) is \( O(\log(n)) \)
  For \( 0 \leq i < n \) we have that \( (n-i)(i+1) \geq n \), so that \( (n!)^2 \geq n^n \) and \( 2\log(n!) \geq n\log(n) \). It follows that \( n\log(n) \) is \( O(\log(n)) \) with witnesses \( C=2, k=1 \)
Be careful combining big-$O$’s

\[ f_1, f_2, g_1, g_2 : \mathbb{R} \to \mathbb{R}, f_i(x) \text{ is } O(g_i(x)) \text{ for } i = 1, 2 \]

- \((f_1 + f_2)(x) \text{ is } O(\max(g_1(x), g_2(x)))\) (triangle inequality)
- \((f_1f_2)(x) \text{ is } O(g_1(x)g_2(x))\) (trivial)
- but \(f(x) \text{ is } O(g(x))\) does not imply \(bf(x) \text{ is } O(bg(x))\) (any \(b > 1\))

one example we’ve seen already:

\[ n \log(n) \text{ is } O(\log(n!)) \text{ but } n^n \text{ is not } O(n!) \]

an easier example: \(f(x) = 2x, g(x) = x:\)

\[ 2x \text{ is } O(x) \text{ but } 2^{2x} = (2^x)^2 \text{ is not } O(2^x) \]
Big-Omega

seen that for $f, g : \mathbb{R} \to \mathbb{R}$, “$f(x)$ is $O(g(x))$”

if there are constants $C$ and $k$ such that

$$\forall x > k \quad |f(x)| \leq C|g(x)|$$

if there are constants $C > 0$, $k > 0$ such that

$$\forall x > k \quad |f(x)| \geq C|g(x)|$$

then “$f(x)$ is $\Omega(g(x))$”

“$f(x)$ is big-Omega of $g(x)$”
Big-O and big-Omega

“$f(x)$ is $O(g(x))$” $\iff$ “$g(x)$ is $\Omega(f(x))$”

Not necessarily either

“$f(x)$ is $O(g(x))$” or “$g(x)$ is $O(f(x))$”:

$f(x) = \sin(x)$, $g(x) = \cos(x)$ (both $O(1)$)
Big-Omega versus Big-O

- Big-O is an upper bound
  “My algorithm runs in $O(f)$”
  means that it takes at most $Cf(n)$ ($n > k$)

- Big-Omega is a lower bound
  “My algorithm runs in $\Omega(f)$”
  means that it takes at least $Cf(n)$ ($n > k$)

- In literature very often used incorrectly
Big-Theta: both Big-O & Big-Omega

If \( f(x) \) is \( O(g(x)) \) and \( f(x) \) is \( \Omega(g(x)) \) then

\[ “f(x) \text{ is } \Theta(g(x))” \]

“\( f(x) \) is big-Theta of \( g(x) \)”

\( f(x) \) is said to be of order \( g(x) \)

\[ “f(x) \text{ is } \Theta(g(x))” \iff “g(x) \text{ is } \Theta(f(x))” \]

Example: \( n \log(n) \) is of order \( \log(n!) \)

(use \( n^n > n! \) and \( n^n < (n!)^2 \))
Little-o

“$f(x)$ is $o(g(x))$” if $\lim_{x \to \infty} \frac{f(x)}{g(x)} = 0$:

“$f$ is little-o of $g$”

$\Rightarrow \forall$ fixed $d$, $(\log(n))^d = n^{o(1)}$ for $n \to \infty$

Find $f(n)$ with $(\log(n))^d = n^{f(n)}$ and $f(n)$ is $o(1)$:

$(\log(n))^d = e^{d \log(\log(n))}$ and $n^{f(n)} = e^{f(n) \log(n)}$

thus $(\log(n))^d = n^{f(n)}$ for $f(n) = \frac{d \log(\log(n))}{\log(n)}$;

$\lim_{n \to \infty} \frac{f(n)}{1} = 0$, so $f(n) = o(1)$

(any fixed power of $\log n$ loses compared to even the tiniest power of $n$)
Computational “complexity”
worst or average case time used by
algorithms, on input of length $n$:

- $\Theta(1)$ constant complexity (parity check)
- $\Theta(\log n)$ logarithmic complexity (sorted search)
- $\Theta(n)$ linear complexity (search max)
- $\Theta(n \log n)$ $n \log n$ complexity (fast sorting)
- $\Theta(n^2)$ quadratic complexity (bubble sort)
- $\Theta(n^3)$ cubic complexity (basic $n \times n$ matrix multiply)
- $\Theta(n^d)$ polynomial complexity ($d$ fixed)
- $\Theta(\Theta(?))$ sub-exponential complexity (integer factoring)
- $\Theta(c^n)$ exponential complexity ($c > 1$ fixed)
- $\Theta(n!)$ factorial complexity (traveling salesman)
- $\Theta(n^n)$ so bad that it does not have a name
“Easier” separation of the big-$\Theta$’s

Fix $b > 1$, and use $x^y = b^{y\log_b(x)}$

Polynomial $\Theta(n^d) = \Theta(b^{d\log_b(n)})$

Exponential $\Theta(b^n)$:
\[ n \text{ strictly bigger than } d \log_b(n) \]

Factorial $\Theta(n!) = \Theta(\sqrt{n} (n/e)^n)$
\[ = \Theta(\sqrt{n} b^{n\log_b(n/e)}) \]
\[ n\log_b(n/e) \text{ strictly bigger than } n \]

Even worse $\Theta(n^n) = \Theta(e^n (n/e)^n)$:
\[ \text{strictly bigger than factorial} \]
\[ \text{because } e^n/\sqrt{n} \text{ is unbounded} \]
Sub-exponential complexity

Input length \( n \), complexity strictly between polynomial=good and exponential=bad

\[ \Theta(n^d) \text{ (fixed } d > 0) \quad \Theta(?) \quad \Theta(b^n) \text{ (fixed } b > 1) \]

\[ n^d = e^{d \log(n)} \quad b^n = e^{\delta n} \quad (\delta = \log(b)) \]

\[ n^d = e^{dn \log(n)} \quad b^n = e^{\delta n \log(n)^0} \]

\[
\Rightarrow \text{ moving from polynomial to exponential}
\]

\[
\Rightarrow \text{ the exponent pair } (0,1) \text{ is transformed into } (1,0)
\]

\[
\Rightarrow ?? = e^{dn^r \log(n)^{1-r}} \text{ with } 0 < r < 1
\]

Example: factoring integer \( m \) takes time

\[ e^{(1.92 + o(1))(\log(m))^{1/3} \left( \log(\log(m)) \right)^{2/3}} \quad (r = 1/3) \]

(input length is \( O(\log(m)) \); all logs natural)
Concludes 3rd section of Chapter 3

On to sections 3.4-3.7: basic number theory

Most already covered in Sciences de l’Information

Thus: here we focus on the missing bits and a quick reminder of known stuff
Integer division facts

Integers $m \neq 0$, $n$, $a$, $b$, $q$, $s$, $t \in \mathbb{Z}$:

- “$m$ divides $n$” or “$m|n$”
  if there is an integer $q$ with $qm=n$:
  “$m$ is a factor of $n$”
  “$n$ is a multiple of $m$”
  “$n$ is divisible by $m$”

- Properties:
  - if $m|a$ and $m|b$ then $m|a+b$
  - if $m|a$ then $\forall b \in \mathbb{Z}$ $m|ab$ (also if $b=0$)
  - if $m|n$ and $n|a$ (with $n \neq 0$) then $m|a$
  - if $m|a$ and $m|b$ then $\forall s, t \in \mathbb{Z}$ $m|sa+tb$
More on division

Integers $m \neq 0, n, q, r \in \mathbb{Z}$:

- “Division algorithm”
  \[ \forall n \in \mathbb{Z} \ \forall m \in \mathbb{Z}_{>0} \ \exists! \ q, r \in \mathbb{Z} \ 0 \leq r < m \text{ s.t.} \]
  \[ n = mq + r \]

- $n$ is the *dividend*, $m$ the *divisor*
- $q = n \div m$, the *quotient* of $n$ and $m$,
- $r = n \mod m$, the *remainder* (upon division of $n$ by $m$)

- $m | n \iff r = n \mod m = 0 \iff m$ divides $n$
- and $m \not| n \iff n \mod m \neq 0 \iff m$ does not divide $n$
Modular arithmetic

Let $a, b, m \in \mathbb{Z}$ with $m > 0$  

• $a$ is congruent to $b$ modulo $m$ if $m \mid a-b$: 
  notation: $a \equiv b \pmod{m}$ (or just $a \equiv b \mod{m}$)  
• if $m \nmid a-b$ (i.e., $a-b \mod m \neq 0$) we write 
  $a \nmid b \pmod{m}$  

• Properties:  
  • $a$ and $b$ are congruent modulo $m$ $\iff$  
    $\exists \ k \in \mathbb{Z}$ s.t. $a = b + km$  
  • $a \equiv c \pmod{m}, b \equiv d \pmod{m}$, then:  
    $a+b \equiv c+d \pmod{m}$, $ab \equiv cd \pmod{m}$  
  • $(a+b) \mod m = ((a \mod m) + (b \mod m)) \mod m$  
  • $ab \mod m = ((a \mod m)(b \mod m)) \mod m$
Notational note on modular arithmetic

- “$a \mod m$” indicates the calculation of the remainder of $a$ upon division by $m$
- “$a \equiv b \pmod{m}$” or “$a \equiv b \mod{m}$” indicates that $a-b$ is divisible by $m$
  (i.e., it says that $(a - b) \mod m = 0)$: $a$ and $b$ are said to be
  “in the same residue class modulo $m$”
- “$a \equiv (a \mod m) \mod m$” is the (true) proposition that
  $a - (a \mod m)$ is divisible by $m$
- $m$ is called the modulus
Toy mod application: Caesar’s cipher

- $f: \{a,b,c,\ldots,z\} \rightarrow \{0,1,2,\ldots,25\}$ bijection mapping $a$ to 0, $b$ to 1, ..., $z$ to 25
- $g: \{0,1,2,\ldots,25\} \rightarrow \{0,1,2,\ldots,25\}$: $n \mapsto (n + 3) \mod 26$
  then $g^{-1}(m) = (m-3) \mod 26$

Caesar’s cipher: $f^{-1} \circ g \circ f$

- encryption: replace each plaintext character $x$ by $f^{-1}(g(f(x)))$
- Decryption: replace each ciphertext character $c$ by $f^{-1}(g^{-1}(f(c)))$

(ciphers of this sort are obviously very weak)
Useful mod application: hash functions

Quick data retrieval while avoiding sorting (or search for specified item):

• Given \( n \) items, each item identified by unique key \( k \in \mathbb{N} \)
• Use \( m \) memory locations \( \{0,1,\ldots,m-1\} \), with \( m \) quite a bit larger than \( n \)
• Store all items: item with key \( k \) stored at location \( k \mod m \) ("the hash")

Once stored, quick retrieval of item with key \( s \): at location \( s \mod m \)

\[ \Rightarrow \text{Data retrieval in time } O(1) \]
(as opposed to \( O(\log n) \))
Collision problem with hash functions

If keys $k_1$ and $k_2$ of different items have same hash: items stored at same location

- Not good: a “collision”
- Collisions will occur if $n$ approaches $\sqrt{m}$ (“birthday paradox”)
  \[ n \text{ approaches } \sqrt{m} \]
  \[ \Rightarrow \text{unavoidable (unless } m \text{ insanely big)} \]
- Requires “collision resolution”:
  - Store at first subsequent free location (leads to hopefully brief linear search)
  - Or use $2^{\text{nd}}$ ($3^{\text{rd}}$, …) hash function
  - Or …
Pseudorandom number generation

With \( a \) (multiplicier), \( c \) (increment), \( m \) (modulus), \( x_0 \) (seed)

and \( x_{i+1} = (ax_i+c) \mod m \)

we get a pseudorandom sequence \( x_0, x_1, \ldots, x_k, \ldots \)

For properly chosen \( a, c, m, x_0 \)

- the resulting sequence looks “random” enough for many purposes
- fast (though it uses a division)
- very bad for cryptography (but widely used)
Remark

hashing and pseudorandom sequences use fact that result of “modding out” by large modulus $m$ looks “unpredictable”

Sequences of mods may cover tracks of a calculation, are thus useful for randomization and data protection

Primes are particularly nice moduli
Concludes 4th section of Chapter 3
Basic results on primes

Why are we interested in primes?

Because they pop up all over the place:

• Hash tables
• Random number generation
• Information security
• Math
• Recreational math
Basic results on primes
Everyone here knows the following:
• a prime is an integer > 1 that has only 1 and itself as positive factors
• non-primes are called *composites*
• \( n \in \mathbb{N}_{>1} \) is prime or can be written as unique product (except for order) of two or more primes (proof later):
  
  the *prime factorization* of \( n \)
  (no unsavory mishaps in \( \mathbb{Z} \) : \( 2 \times 3 = 6 = (1-\sqrt{5}) \times (1+\sqrt{5}) \))
• \( n \) composite \( \iff \) \( n \) has a prime factor \( \leq \sqrt{n} \)
• \( |\text{set of primes}| = \aleph_0 \) (with an easy proof)
• given \( x > 0 \), how many primes \( \leq x \)?
The prime number theorem (PNT)

Less well known (and non-trivial) fact:

- There are *plenty* of primes:
  \[ \pi(x) = \# \{ p \mid p \text{ prime}, p \leq x \} \approx \frac{x}{\log(x)} \]

- “prime counting function” \( \pi(x) \) hard to calculate exactly; current record:
  \( \pi(10^{24}) = 18,435,599,767,349,200,867,866 \)

- Useful consequences of PNT:
  - random \( k \)-bit integer is prime with probability >1/\( k \)
  - random 100-digit \( m \) is prime with probability 1/230
  - different parties probably generate different primes

- But: how do we recognize if \( m \) is prime?
Generating primes

all primes up to a small bound can be generated using **sieve of Eratosthenes**

security applications need primes that are

• very large (hundreds of digits)
• unpredictable by others (“random”)

⇒ sieve of Eratosthenes cannot be used to generate those
Generating large primes
to generate a random $k$-bit prime ($k$ large):
1. pick a random $k$-bit integer $m$
2. if $m$ is composite return to Step 1
3. output $m$ as the desired prime

PNT $\Rightarrow$ “expect” about $k$ jumps to Step 1

how do we:
1. (hard) pick a random number?
2. (easy) check if $m$ composite?
   • try all factors $\leq \sqrt{m}$ of $m$: hopeless
   • use $\approx$ Fermat’s little theorem:
     \[ p \text{ prime} \rightarrow \forall a \in \mathbb{Z} \ a^p \equiv a \pmod{p} \]

one $a$ with $a^m \not\equiv a \pmod{m}$ proves $m$ composite
Applying (variation of) Fermat to prove that large $m$ is composite we need to be able to test if $a^m \not\equiv a \pmod{m}$ for $a \in \mathbb{Z}$:

$m$ does not divide $a^m - a$
$\iff (a^m - a) \mod m \neq 0$
$\iff (a^m \mod m - a \mod m) \mod m \neq 0$
$\iff (\text{use } a = a \mod m)$

$(a^m \mod m - a) \mod m \neq 0$

$a^m \mod m = (a \cdot a \cdot a \cdot \ldots \cdot a) \mod m =
(\ldots(((a \cdot a) \mod m) \cdot a) \mod m) \cdot \ldots \cdot a) \mod m$:

- all intermediate products taken modulo $m$
- repeated product infeasible for large $m$
Modular exponentiation

Calculating $a^e \mod m$ using $e-1$ modular multiplications is infeasible for large $e$ (and would defeat the purpose)

Use binary representation $e = \sum_{i=0}^{L} e_i 2^i$ ($e_i \in \{0,1\}, e_L = 1$) of the exponent $e$

And: $a^e \mod m = a \sum_{i=0}^{L} e_i 2^i \mod m = (a^1)^{e_0} \ast (a^2)^{e_1} \ast (a^2^2)^{e_2} \ast \ldots \ast (a^2^{L-1})^{e_{L-1}} \ast (a^2^L)^{e_L}$

(While computing everything modulo $m$)

This can be used in two ways:

- Right to left: $e_0, e_1, e_2, \ldots, e_{L-1}, e_L$
- Left to right: $e_L, e_{L-1}, e_{L-2}, \ldots, e_1, e_0$
Intermezzo on polynomial evaluation

Compute \( f(c) = \sum_{i=0}^{d} f_i c^i = f_d c^d + ... + f_1 c^1 + f_0 c^0 \)

How not to do it: let \( power = 1, \ result = f_0 \)
for \( i = 1 \) to \( d \) do: ("right to left")
replace \( power \) by \( power \times c \) (\( power = c^i \))
replace \( result \) by \( result + f_i \times power \)
now we have \( result = f(c) \)

How to do it (Horner): let \( result = f_d \)
for \( i = d-1 \) downto 0 do: ("left to right")
replace \( result \) by \( result \times c + f_i \)
now we have \( result = f(c) \)
both \( \Theta(d) \), but Horner twice faster (and fewer variables)
Application of same idea to exponentiation
we can calculate
\[ a^e \mod m = a \sum_{i=0}^{L} e_i 2^i \mod m = \]
\[ (a^{2^0})^{e_0} \ast (a^{2^1})^{e_1} \ast (a^{2^2})^{e_2} \ast \ldots \ast (a^{2^{L-1}})^{e_{L-1}} \ast (a^{2^L})^{e_L} \]

as a product of successive squares
but also as squares of successive products:
\[ (\ldots(((a_e L)^2 \ast a_{e_{L-1}})^2 \ast a_{e_{L-2}})^2 \ast \ldots \ast a_e 1)^2 \ast a_e 0 \]

- unlike Horner, speed remains same
- like Horner: fewer variables
- “*” denotes “modular multiplication”
Right to left modular exponentiation

calculate \(a^e \mod m\) with \(e = \sum_{i=0}^{L} e_i 2^i\)

processing \(e_0, e_1, e_2, \ldots, e_{L-1}, e_L\):

calculate \(a^{2^0}, a^{2^1}, a^{2^2}, \ldots, a^{2^{L-1}}, a^{2^L}\)

multiplying those for which \(e_i = 1\):

let \(result = 1\) and \(power = a \mod m\)

for \(i = 0\) to \(L\) do:

if \(e_i = 1\) then

replace \(result\) by \((result \times power) \mod m\)

replace \(power\) by \(power^2 \mod m\)

now we have \(result = a^e \mod m\)
Right to left exponentiation example

Calculate $3^{23} \mod 47$

with $23 = 2^4 + 2^2 + 2^1 + 2^0 = 10111$ we find $L = 4$ and $e_0 = 1, e_1 = 1, e_2 = 1, e_3 = 0, e_4 = 1$

let $result = 1$ and $power = 3 \mod 47 = 3^1 \mod 47$

for $i = 0$ to 4 do:

$i=0$: $e_0=1$: $result = 1*3 \mod 47 = 3; power = 3^2 \mod 47 = 9$;
   now $result = 3^1 \mod 47, power = 3^{10} \mod 47$

$i=1$: $e_1=1$: $result = 3*9 \mod 47 = 27; power = 9^2 \mod 47 = 34$;
   now $result = 3^{11} \mod 47, power = 3^{100} \mod 47$

$i=2$: $e_2=1$: $result = 27*34 \mod 47 = 25; power = 34^2 \mod 47 = 28$;
   now $result = 3^{111} \mod 47, power = 3^{1000} \mod 47$

$i=3$: $e_3=0$: leave $result$ as is; $power = 28^2 \mod 47 = 32$;
   now $result = 3^{0111} \mod 47, power = 3^{10000} \mod 47$

$i=4$: $e_4=1$: $result = 25*32 \mod 47 = 1; power = 32^2 \mod 47 = 37$;
   now $result = 3^{10111} \mod 47$, done: $result = 1$ ($3^{47} \equiv 3 \mod 47$)
Left to right modular exponentiation

calculate \( a^e \mod m \) with \( e = \sum_{i=0}^{L} e_i 2^i \)

processing \( e_L, e_{L-1}, e_{L-2}, \ldots, e_1, e_0 \):

calculate \( a^{e_L}, (a^{e_L})^2 a^{e_{L-1}}, ((a^{e_L})^2 a^{e_{L-1}})^2 a^{e_{L-2}}, \ldots, \)

using squarings, and multiplies when \( e_i = 1 \):

let \( \text{result} = a \mod m \) (since \( e_L = 1 \))

for \( i = L-1 \) downto 0 do:

replace \( \text{result} \) by \( \text{result}^2 \mod m \)

if \( e_i = 1 \) then

replace \( \text{result} \) by \( (\text{result} \ast a \mod m) \mod m \)

now we have \( \text{result} = a^e \mod m \)
Left to right exponentiation example

Calculate $3^{23} \mod 47$

$23 = 2^4 + 2^2 + 2^1 + 2^0 = 10111$ and we have

$L = 4 \text{ and } e_0 = 1, e_1 = 1, e_2 = 1, e_3 = 0, e_4 = 1$

let $result = 3 \mod 47$

now $result = 3^1 \mod 47$

for $i = 3$ downto 0 do:

$i=3: result = 3^2 \mod 47 = 9; e_3=0$: leave result as is;

now $result = 3^{10} \mod 47$

$i=2: result = 9^2 \mod 47 = 34; e_2=1: result = 34*3 \mod 47 = 8;

now $result = 3^{101} \mod 47$

$i=1: result = 8^2 \mod 47 = 17; e_1=1: result = 17*3 \mod 47 = 4;

now $result = 3^{1011} \mod 47$

$i=0: result = 4^2 \mod 47 = 16; e_0=1: result = 16*3 \mod 47 = 1;

now $result = 3^{10111} \mod 47$, done: $result = 1$ ($3^{47} = 3 \mod 47$)
Speed of modular exponentiation
for both “right to left” and “left to right:”
• # modular squarings: $L+1$ or $L$
• # modular multiplications:
  $\#\{i : e_i = 1\}$ or $\#\{i : e_i = 1\} - 1$
  either way:
  total effort $\Theta(L)$ modular multiplications
schoolbook modular multiplication: $O((\log m)^2)$
overall:
modular exponentiation effort is $O(L(\log m)^2)$
if $L = \log_2(m)$, then this becomes $O((\log m)^3)$
annoying fact: the $\Theta(L)$ is inherently sequential
Speed of prime generation

Generate $k$-bit primes as follows:

1. Pick a random $k$-bit integer $m$ (making it odd helps…)
2. Test if $m$ is composite: pick random $a \in \mathbb{Z}$, check if $a^m \equiv a \pmod{m}$ (actually: slight variant)
   If not return to Step 1
3. Output $m$ as the desired prime

Silent assumption: for randomly selected $a$
   the test $a^m \equiv a \pmod{m}$ fails if $m$ composite:
   incorrect, but in practice okay for large $m$

Overall effort: on average $\approx k$ attempts,
   each attempt $O(k^3) \Rightarrow$ expected overall $O(k^4)$
   (with huge variation; and faster with fast multiplication)
Large primes, for what purpose?
generation of large $k$-bit primes in (expected) $O(k^{\leq 4})$ time allows implementation of

- RSA: security based on the difficulty of inverting integer multiplication (factoring), need $k = 512$ or larger
  as of Jan 1, 2011: RSA no longer approved for US government use

- approved methods based on difficulty of inverting modular exponentiation (discrete logarithm): variants of $ElGamal$, need $k = 160$ or larger
  (using other groups too, principle same)
Skipping

• greatest common divisors
division-free: make odd & subtract \(O((\log(n))^2)\)

• extended Euclidean algorithm / Bezout
easy: maintain \(uv \equiv d \pmod{p}\)

• Chinese remainderring
constructive: \(x = x_1 + p_1[(x_2 - x_1)/p_1 \mod p_2]\)

(all “covered” by Sciences de l’Information)

(some slides will be made available describing the
division-free/easy/constructive methods referred to above:
looks for gcd/etc_slides_0402)

Concludes 7th section of Chapter 3
Section 3.8: matrices

- Read it!
- \( n \times m \) rectangles of numbers:
  \( n \) rows, \( m \) columns
- Originally to represent linear transformations from \( \mathbb{R}^m \) to \( \mathbb{R}^n \)
- Wide variety of applications
Matrix product

\[ \forall m, k, n \in \mathbb{Z}_{>0}: \]

\[ m \times k\] matrix \( A = (a_{ij})_{i=1,j=1}^{m,k} \),

\[ k \times n\] matrix \( B = (b_{jl})_{j=1,l=1}^{k,n} \),

\[ AB = C \] is \( m \times n\) matrix \( (c_{il})_{i=1,l=1}^{m,n} \)

with \( c_{il} = \sum_{j=1}^{k} a_{ij} b_{jl} \)

- Computation in \( m \times k \times n\) multiplications
- Not commutative:
  even if \( AB \) and \( BA \) are both defined, they are not necessarily equal
Concludes Chapter 3

On to Chapter 4: induction & recursion
Modular arithmetic

let $a, b, m \in \mathbb{Z}$ with $m > 0$

• $a$ is congruent to $b$ modulo $m$ if $m \mid a-b$:
  notation: $a \equiv b \pmod{m}$ (or just $a \equiv b \mod{m}$)

• if $m \nmid a-b$ (i.e., $a-b \mod{m} \neq 0$) we write $a \not\equiv b \pmod{m}$

• properties:
  • $a$ and $b$ are congruent modulo $m$ $\iff$
    $\exists k \in \mathbb{Z}$ s.t. $a = b + km$
  • $a \equiv c \pmod{m}, b \equiv d \pmod{m}$, then:
    $a+b \equiv c+d \pmod{m}, \ ab \equiv cd \pmod{m}$
  • $(a+b)\mod{m} = ((a \mod{m})+(b \mod{m}))\mod{m}$
  • $ab \mod{m} = ((a \mod{m})(b \mod{m}))\mod{m}$
Notational note on modular arithmetic

• “$a \mod m$” indicates the calculation of the remainder of $a$ upon division by $m$

• “$a \equiv b \ (\text{mod } m)$” or “$a \equiv b \text{ mod } m$” indicates that $a-b$ is divisible by $m$ (i.e., it says that $(a - b) \mod m = 0)$: $a$ and $b$ are said to be “in the same residue class modulo $m$”

• “$a \equiv (a \mod m) \mod m$” is the (true) proposition that $a - (a \mod m)$ is divisible by $m$

• $m$ is called the modulus
Toy mod application: Caesar’s cipher

- $f: \{a,b,c,\ldots,z\} \rightarrow \{0,1,2,\ldots,25\}$ bijection mapping $a$ to 0, $b$ to 1, ..., $z$ to 25
- $g: \{0,1,2,\ldots,25\} \rightarrow \{0,1,2,\ldots,25\}$: 
  \[ n \mapsto (n + 3) \mod 26 \]
  then $g^{-1}(m) = (m - 3) \mod 26$

**Caesar’s cipher**: $f^{-1} \circ g \circ f$

- encryption: replace each plaintext character $x$ by $f^{-1}(g(f(x)))$
- decryption: replace each ciphertext character $c$ by $f^{-1}(g^{-1}(f(c)))$

(ciphers of this sort are obviously very weak)
Useful mod application: hash functions
quick data retrieval while avoiding sorting
(or search for specified item):
• given \( n \) items, each item
  identified by unique key \( k \in \mathbb{N} \)
• use \( m \) memory locations \( \{0,1,\ldots,m-1\} \),
  with \( m \) quite a bit larger than \( n \)
• store all items: item with key \( k \) stored
  at location \( k \mod m \) (“the hash”)
  once stored, quick search for item
  with key \( s \): at location \( s \mod m \)
⇒ data retrieval in time \( O(1) \)
  (as opposed to \( O(\log n) \))
Collision problem with hash functions

if keys $k_1$ and $k_2$ of different items have same hash: items stored at same location
• this is not good: called a “collision”
• for random keys, collisions will occur if $n$ approaches $\sqrt{m}$ (“birthday paradox”)

⇒ unavoidable (unless $m$ insanely big)
• requires “collision resolution”:
  • store at first subsequent free location (leads to hopefully brief linear search)
  • or use 2$^{nd}$ ($3^{rd}$, …) hash function
  • or …
• not to be confused with cryptographic hashing
Pseudorandom number generation

with \( a \) (multiplier), \( c \) (increment), 
\( m \) (modulus), \( x_0 \) (seed)

and \( x_{i+1} = (ax_i + c) \mod m \)

we get a *pseudorandom sequence*

\[ x_0, x_1, \ldots, x_k, \ldots \]

for properly chosen \( a, c, m, x_0 \)

- the resulting sequence looks “random” enough for many purposes
- fast (though it uses a division)
- very bad for information protection (but widely used)
Remark

hashing and pseudorandom sequences
use fact that result of “modding out” by
large modulus $m$ looks “unpredictable”

sequences of $\text{mods}$ may cover tracks
of a calculation, are thus useful for
randomization and data protection

primes are particularly nice moduli
related to one of the hardest practical problems
in data protection: generating random numbers

(notable screw-ups: netscape, debian, playstation3, SSL, X509 certs, …

most recently [http://www.theregister.co.uk/2013/03/26/netbsd_crypto_bug/](http://www.theregister.co.uk/2013/03/26/netbsd_crypto_bug/))
Concludes
first sections of Chapter 4 (7\textsuperscript{th} edition)
4\textsuperscript{th} section of Chapter 3 (6\textsuperscript{th} edition)
Basic results on primes

why are we interested in primes?

because they pop up all over the place:

• hash tables
• random number generation
• information security
• math
• recreational math
Basic results on primes
everyone here knows the following:
• a prime is an integer > 1 that has only 1 and itself as positive factors
• non-primes are called *composites*
• \( n \in \mathbb{N}_{>1} \) is prime or can be written as unique product (except for order) of two or more primes (proof later):
  the *prime factorization* of \( n \)
  (no unsavory mishaps in \( \mathbb{Z} \): \( 2 \times 3 = 6 = (1-\sqrt{-5})(1+\sqrt{-5}) \))
• \( n \) composite \( \iff \ n \) has a prime factor \( \leq \sqrt{n} \)
• \( |\text{set of primes}| = \aleph_0 \) (with an easy proof)
• \( \pi(x) \) is number of primes \( \leq x \) : what is \( \pi(x) \)?
The prime number theorem (PNT) is a less well known (and non-trivial) fact:

- there are *plenty* of primes:
  \[ \pi(x) = \#\{p \mid p \text{ prime}, p \leq x\} \approx \frac{x}{\log(x)} \]

- “prime counting function” \( \pi(x) \) is hard to calculate exactly; current record:
  \[ \pi(10^{24}) = 18,435,599,767,349,200,867,866 \]

- useful consequences of PNT:
  - random \( k \)-bit integer is prime with probability \( >1/k \)
  - random 100-digit \( m \) is prime with probability \( 1/230 \)
  - different parties *probably* generate different primes

- but: how do we recognize if \( m \) is prime?
Generating primes

all primes up to some small bound can be generated using **sieve of Eratosthenes**

security applications need primes that are

- large (hundreds of digits)
- unpredictable by others ("random")

⇒ sieve of Eratosthenes cannot be used to generate those
Generating large primes

to generate a random $k$-bit prime ($k$ large):

1. pick a random $k$-bit integer $m$
2. if $m$ is composite return to Step 1
3. output $m$ as the desired prime

PNT ⇒ “expect” about $k$ jumps to Step 1

how do we:

1. pick a random number? hard or easy?
2. check if $m$ composite? hard or easy?
Generating large primes

to generate a random \( k \)-bit prime (\( k \) large):

1. pick a random \( k \)-bit integer \( m \)
2. if \( m \) is composite return to Step 1
3. output \( m \) as the desired prime

PNT \( \Rightarrow \) “expect” about \( k \) jumps to Step 1

how do we:

1. pick a random number? (this is hard)
2. check if \( m \) composite? (this is easy)

• try all factors \( \leq \sqrt{m} \) of \( m \): hopeless
• use \( \approx \) Fermat’s little theorem:

\[ p \text{ prime } \rightarrow \forall a \in \mathbb{Z} \ a^p \equiv a \pmod{p} \]

one \( a \) with \( a^m \not\equiv a \pmod{m} \) proves \( m \) composite
Applying (variation of) Fermat
proving $m$’s compositeness requires
testing if $a^m \not\equiv a \pmod{m}$ for $a \in \mathbb{Z}$:

$m$ does not divide $a^m - a$
$\iff (a^m - a) \mod m \neq 0$
$\iff (a^m \mod m - a \mod m) \mod m \neq 0$
$\iff \text{(use } a = a \mod m)\frac{}{}
\iff (a^m \mod m - a) \mod m \neq 0$

$a^m \mod m = (a \ast a \ast a \ast \ldots \ast a) \mod m =
(\ldots(((((a \ast a) \mod m) \ast a) \mod m) \ast \ldots \ast a) \mod m$:

• all products taken modulo $m$:
  no intermediate result $> m^2$
• but repeated product infeasible for large $m$
Modular exponentiation

calculating $a^e \mod m$ using $e-1$ modular multiplications is infeasible for large $e$
(and defeats purpose of using Fermat)

from the first semester we know that
“Le calcul d’une puissance en arithmétique modulaire est particulièrement simple, il suffit de décomposer l’exposant.”

example (modulo 7):
$3^{12} = (3^2)^6 = 9^6 \equiv 2^6 = (2^3)^2 = 8^2 \equiv 1^2 = 1$

we also know
“On pense aujourd’hui que la factorisation de nombres entiers très grands est un problème difficile.”
Modular exponentiation

still unclear how to calculate
\( a^e \mod m \) for large \( e \)

use binary representation  \( e = \sum_{i=0}^{L} e_i 2^i \)
\((e_i \in \{0,1\}, e_L = 1)\) of the exponent \( e \)

and : \( a^e \mod m = a^{\sum_{i=0}^{L} e_i 2^i} \mod m = \)
\((a^1)^{e_0} \ast (a^2)^{e_1} \ast (a^2^2)^{e_2} \ast \ldots \ast (a^{2^{L-1}})^{e_{L-1}} \ast (a^{2^L})^{e_L}\)

(while computing everything modulo \( m \))

this can be used in two ways:

• right to left: \( e_0, e_1, e_2, \ldots, e_{L-1}, e_L \)

• left to right: \( e_L, e_{L-1}, e_{L-2}, \ldots, e_1, e_0 \)
Intermezzo on polynomial evaluation

$$f(c) = \sum_{i=0}^{d} f_i c^i = f_d c^d + \ldots + f_1 c^1 + f_0 c^0$$

how not to do it: let $power = 1$, $result = f_0$

for $i = 1$ to $d$ do: ("right to left")

replace $power$ by $power * c$  \hspace{1cm} (power = c^i)

replace $result$ by $result + f_i * power$

now we have $result = f(c)$

how to do it (Horner): let $result = f_d$

for $i = d-1$ downto 0 do: ("left to right")

replace $result$ by $result * c + f_i$

now we have $result = f(c)$

both $\Theta(d)$, but Horner twice faster (and fewer variables)
Application of same idea to exponentiation
we can calculate
\[ a^e \mod m = a \sum_{i=0}^{L} e_i 2^i \mod m = \]
\[(a^{2^0})^{e_0} \ast (a^{2^1})^{e_1} \ast (a^{2^2})^{e_2} \ast \ldots \ast (a^{2^{L-1}})^{e_{L-1}} \ast (a^{2^L})^{e_L} \]
as a product of successive squares
but also as squares of successive products:
\[(...(((a^{e_L})^2 \ast a^{e_{L-1}})^2 \ast a^{e_{L-2}})^2 \ast \ldots \ast a^{e_1})^2 \ast a^{e_0}\]
• unlike Horner, speed remains same
• like Horner: fewer variables
• “*” denotes “modular multiplication”
and all squarings are “modular” too
Right to left modular exponentiation

calculate \( a^e \mod m \) with \( e = \sum_{i=0}^{L} e_i 2^i \)

processing \( e_0, e_1, e_2, \ldots, e_{L-1}, e_L \):

calculate \( a^{2^0}, a^{2^1}, a^{2^2}, \ldots, a^{2^{L-1}}, a^{2^L} \),
multiplying those for which \( e_i = 1 \):

let \( \text{result} = 1 \) and \( \text{power} = a \mod m \)

for \( i = 0 \) to \( L \) do:

\[ \text{if } e_i = 1 \text{ then} \]

\[ \text{replace } \text{result} \text{ by } (\text{result} \times \text{power}) \mod m \]

\[ \text{replace } \text{power} \text{ by } \text{power}^2 \mod m \]

now we have \( \text{result} = a^e \mod m \)
Right to left exponentiation example

calculate $3^{23} \mod 47$

with $23 = 2^4 + 2^2 + 2^1 + 2^0 = 10111$ we find $L = 4$ and $e_0 = 1, e_1 = 1, e_2 = 1, e_3 = 0, e_4 = 1$

let $result = 1$ and $power = 3 \mod 47 = 3^1 \mod 47$

for $i = 0$ to $4$ do:

$i=0$: $e_0=1$: $result = 1 \times 3 \mod 47 = 3$; $power = 3^2 \mod 47 = 9$

now $result = 3^1 \mod 47$, $power = 3^{10} \mod 47$

$i=1$: $e_1=1$: $result = 3 \times 9 \mod 47 = 27$; $power = 9^2 \mod 47 = 34$

now $result = 3^{11} \mod 47$, $power = 3^{100} \mod 47$

$i=2$: $e_2=1$: $result = 27 \times 34 \mod 47 = 25$; $power = 34^2 \mod 47 = 28$

now $result = 3^{111} \mod 47$, $power = 3^{1000} \mod 47$

$i=3$: $e_3=0$: leave $result$ as is; $power = 28^2 \mod 47 = 32$

now $result = 3^{0111} \mod 47$, $power = 3^{10000} \mod 47$

$i=4$: $e_4=1$: $result = 25 \times 32 \mod 47 = 1$; $power = 32^2 \mod 47 = 37$

now $result = 3^{10111} \mod 47$, done: $result = 1$ ($3^{47}=3 \mod 47$)
Left to right modular exponentiation

calculate $a^e \mod m$ with $e = \sum_{i=0}^{L} e_i 2^i$

processing $e_L, e_{L-1}, e_{L-2}, \ldots, e_1, e_0$:

calculate $a^{e_L}, (a^{e_L})^2 a^{e_{L-1}}, (((a^{e_L})^2 a^{e_{L-1}})^2 a^{e_{L-2}}, \ldots$,

using squarings, and multiplies when $e_i = 1$:

let $result = a \mod m$ (since $e_L = 1$)

for $i = L-1$ downto 0 do:

replace $result$ by $result^2 \mod m$

if $e_i = 1$ then

replace $result$ by $(result * a \mod m) \mod m$

now we have $result = a^e \mod m$
Left to right exponentiation example

calculate $3^{23} \mod 47$

$23 = 2^4 + 2^2 + 2^1 + 2^0 = 10111$ and we have $L = 4$ and $e_0 = 1, e_1 = 1, e_2 = 1, e_3 = 0, e_4 = 1$

let \( \text{result} = 3 \mod 47 \)

now \( \text{result} = 3^1 \mod 47 \)

for \( i = 3 \) downto 0 do:

\(i=3: \ \text{result} = 3^2 \mod 47 = 9; \ e_3=0: \ \text{leave result as is};\)

now \( \text{result} = 3^{10} \mod 47 \)

\(i=2: \ \text{result} = 9^2 \mod 47 = 34; \ e_2=1: \ \text{result} = 34*3 \mod 47 = 8;\)

now \( \text{result} = 3^{101} \mod 47 \)

\(i=1: \ \text{result} = 8^2 \mod 47 = 17; \ e_1=1: \ \text{result} = 17*3 \mod 47 = 4;\)

now \( \text{result} = 3^{1011} \mod 47 \)

\(i=0: \ \text{result} = 4^2 \mod 47 = 16; \ e_0=1: \ \text{result} = 16*3 \mod 47 = 1;\)

now \( \text{result} = 3^{10111} \mod 47 \), done: \( \text{result} = 1 \) (\( 3^{47}=3 \mod 47 \))
Speed of modular exponentiation
for both “right to left” and “left to right:”

- # modular squarings: \( L + 1 \) or \( L \)
- # modular multiplications:
  \[
  \#\{ i : e_i = 1 \} \text{ or } \#\{ i : e_i = 1 \} - 1
  \]

either way:

  total effort \( \Theta(L) \) modular multiplications

schoolbook modular multiplication: \( O((\log m)^2) \)

overall:

  modular exponentiation effort is \( O(L(\log m)^2) \)

if \( L = \log_2(m) \), then this becomes \( O((\log m)^3) \)

annoying fact: the \( \Theta(L) \) is inherently sequential
Speed of prime generation

generate $k$-bit primes as follows:
1. pick a random $k$-bit integer $m$ (making it odd helps…)
2. test if $m$ is composite: pick random $a \in \mathbb{Z}$, check if $a^m \equiv a \pmod{m}$ (actually: slight variant)
   if not return to Step 1
3. output $m$ as the desired prime

silent assumption: for randomly selected $a$
   the test $a^m \equiv a \pmod{m}$ fails if $m$ composite:
   incorrect, but in practice okay for large $m$

overall effort: on average $\approx k$ attempts,
each attempt $O(k^3) \Rightarrow$ expected overall $O(k^4)$
(with huge variation; and faster with fast multiplication)
Large primes, for what purpose?
generation of large $k$-bit primes in (expected) $O(k^{\leq 4})$ time allows implementation of

- RSA: security based on the difficulty of inverting integer multiplication (*factoring*), need $k = 512$ and larger
  as of Jan 1, 2011: RSA no longer approved for US government use

- approved methods based on difficulty of inverting modular exponentiation (*discrete logarithm*): variants of *ElGamal*, need $k = 160$ and larger
  (using other groups too, principle same)
Skipping

• greatest common divisors
division-free: make odd & subtract \(O((\log(n))^2)\)

• extended Euclidean algorithm / Bezout
easy: maintain \(uv \equiv d \pmod{p}\)

• Chinese remaindering
constructive: \(x = x_1 + p_1[(x_2-x_1)/p_1 \pmod{p_2}]\)

• all “covered” by Sciences de l’Information

• description of division-free/easy/constructive methods will be made available on slides
Concludes Chapter 4 (7th) / 3 (6th)

on to Chapter 5 (7th) / 4 (6th) :
induction & recursion
Greatest common divisor

Given two integers $a$ and $b$, not both zero; their greatest common divisor is the largest integer $d$ with $d|a$ and $d|b$: $d = \gcd(a,b)$; conversely, least common multiple: smallest $s \in \mathbb{Z}_{>0}$ with $a|s$, $b|s$: $s = \text{lcm}(a,b)$.

- $1|a$ and $1|b$, thus $\gcd(a,b) \geq 1$; also $\gcd(a,b) \leq \min(|a|,|b|)$; thus $\gcd(a,b)$ exists
- $a|ab$ and $b|ab$, thus $\text{lcm}(a,b) \leq |ab|$; also $\text{lcm}(a,b) \geq \max(|a|,|b|)$; thus $\text{lcm}(a,b)$ exists
- If $\gcd(a,b) = 1$, then $a$ and $b$ are coprime.
Computing the gcd and the lcm

\[ a = \prod_{i=1}^{n} p_i^{e_i}, \quad b = \prod_{i=1}^{n} p_i^{d_i} \quad \text{(distinct primes } p_i) \]

\[ \Rightarrow \gcd(a, b) = \prod_{i=1}^{n} p_i^{\min(e_i, d_i)}, \quad \text{lcm}(a, b) = \prod_{i=1}^{n} p_i^{\max(e_i, d_i)} \]

\[ \Rightarrow ab = \gcd(a, b) \ast \text{lcm}(a, b) \]

\[ \Rightarrow \text{lcm}(a, b) \text{ easily follows from } \gcd(a, b) \]

this requires factorization (one suffices): slow
much smarter to use the Euclidean algorithm
Observation underlying Euclidean algorithm

∀k ∈ ℤ: gcd(a, b) = gcd(b, a - kb)

proof

• if d = gcd(a, b) then d|a and d|b,
  and thus ∀s, t ∈ ℤ d|sa + tb;
  take s = 1, t = -k, then d|a - kb. (universal instantiation)
  thus d|b and d|a - kb, thus d|gcd(b, a - kb)

• if d = gcd(b, a - kb) then d|b and d|a - kb,
  and thus ∀s, t ∈ ℤ d|sb + t(a - kb);
  take s = k, t = 1, then d|kb + (a - kb) = a.
  thus d|b and d|a, thus d|gcd(b, a) = gcd(a, b)

⇒ gcd(a, b)|gcd(b, a - kb) and
gcd(b, a - kb)|gcd(a, b), which implies Thm.
Euclidean algorithm

how to best use (with \( a > 0, \ b \geq 0 \))

\[ \forall k \in \mathbb{Z}: \gcd(a,b) = \gcd(b, a - kb) \]

replace problem of computing \( \gcd(a,b) \) by smaller problem of computing \( \gcd(b, a - kb) \),

which \( k \) to use?

three approaches:

**standard:** use \( k = a \ \text{div} \ b \) (and \( \gcd(a,0) = a \))

\( \text{(so } 0 \leq a - kb = a \ \text{mod} \ b < b \) \)

**better:** minimize \( |a - kb| \) (above \( k \) or \( k+1 \))

\( \text{(so } 0 \leq |a - kb| \leq b/2 \) \)

**binary:** \( a, b \) odd: use \( k = 1 \) and

(Division-free!) remove 2s from \( a - b \) (“shift”)
Example

compute \( \text{gcd}(147,91) \)

using factorization (bad idea)

\( 147 = 3 \times 7^2, \quad 91 = 7 \times 13 \)

so: \( 147 = 3^1 \times 7^2 \times 13^0, \quad 91 = 3^0 \times 7^1 \times 13^1 \)

thus

\[
\text{gcd}(147,91) = 3^{\min(1,0)} \times 7^{\min(2,1)} \times 13^{\min(0,1)}
\]

\[
= 3^0 \times 7^1 \times 13^0
\]

\[
= 7
\]
Euclidean algorithm examples

compute $\gcd(147, 91)$

standard Euclidean algorithm

\[
\begin{align*}
147 &= 1 \times 91 + 56: \quad \gcd(147, 91) = \gcd(91, 56) \\
91 &= 1 \times 56 + 35: \quad \gcd(91, 56) = \gcd(56, 35) \\
56 &= 1 \times 35 + 21: \quad \gcd(56, 35) = \gcd(35, 21) \\
35 &= 1 \times 21 + 14: \quad \gcd(35, 21) = \gcd(21, 14) \\
21 &= 1 \times 14 + 7: \quad \gcd(21, 14) = \gcd(14, 7) \\
14 &= 2 \times 7 + 0: \quad \gcd(14, 7) = \gcd(7, 0) = 7 \\
\Rightarrow \gcd(147, 91) &= 7,
\end{align*}
\]

after 6 standard division steps:

147, 91, 56, 35, 21, 14, 7, 0

(bounding number of steps is cumbersome)
Euclidean algorithm examples

compute \( \gcd(147, 91) \)

smallest remainder Euclidean algorithm

\[
\begin{align*}
147 &= 2 \times 91 - 35: & \gcd(147, 91) &= \gcd(91, 35) \\
91 &= 3 \times 35 - 14: & \gcd(91, 35) &= \gcd(35, 14) \\
35 &= 2 \times 14 + 7: & \gcd(35, 14) &= \gcd(14, 7) \\
14 &= 2 \times 7 + 0: & \gcd(14, 7) &= \gcd(7, 0) = 7
\end{align*}
\]

\( \Rightarrow \gcd(147, 91) = 7, \)

after 4 division steps:

\[
147, 91, 35, 14, 7, 0
\]

(number of division steps in \( \gcd(n, m) \) is easily bounded by \( \log_2(\min(n, m)) \))
Euclidean algorithm examples

compute gcd(147,91)

**binary Euclidean algorithm**

147 and 91 both odd:

\[
gcd(147,91) = gcd(91, 147 - 91) = gcd(91, 56) = gcd(91, 7) \text{ (removed three 2s)}
\]

\[
gcd(91, 7) = gcd(7, 91 - 7) = gcd(7, 84) = gcd(7, 21) \text{ (removed two 2s)}
\]

\[
gcd(21, 7) = gcd(7, 21 - 7) = gcd(7, 14) = gcd(7, 7) \text{ (removed one 2)}
\]

⇒ gcd(147, 91) = 7, in 3 division-less steps

(147, 91), (91, 7), (21, 7), (7, 7)

Can you figure out how to deal with non-odd inputs?
Another example
compute \( \text{gcd}(127, 91) \)
using factorization
\( 127 = 127^1 \) is prime
thus 127 coprime to any \( a \) with \( 0 < a < 127 \)
\( \Rightarrow \) we find \( \text{gcd}(127, 91) = 1 \)
(remember: gcds with primes are easy)
Euclidean algorithm examples

compute \(\gcd(127, 91)\)

standard Euclidean algorithm

\[127 = 1 \times 91 + 36: \quad \gcd(127, 91) = \gcd(91, 36)\]

\[91 = 2 \times 36 + 19: \quad \gcd(91, 36) = \gcd(36, 19)\]

\[36 = 1 \times 19 + 17: \quad \gcd(36, 19) = \gcd(19, 17)\]

\[19 = 1 \times 17 + 2: \quad \gcd(19, 17) = \gcd(17, 2)\]

\[17 = 8 \times 2 + 1: \quad \gcd(17, 2) = \gcd(2, 1)\]

\[2 = 2 \times 1 + 0: \quad \gcd(2, 1) = \gcd(1, 0) = 1\]

\[\Rightarrow \gcd(127, 91) = 1,\]

after 6 standard division steps:

\[127, 91, 36, 19, 17, 2, 1, 0\]
Euclidean algorithm examples

compute $\text{gcd}(127, 91)$

smallest remainder Euclidean algorithm

$127 = 1 \times 91 + 36$: $\text{gcd}(127, 91) = \text{gcd}(91, 36)$

$91 = 3 \times 36 - 17$: $\text{gcd}(91, 36) = \text{gcd}(36, 17)$

$36 = 2 \times 17 + 2$: $\text{gcd}(36, 17) = \text{gcd}(17, 2)$

$17 = 8 \times 2 + 1$: $\text{gcd}(17, 2) = \text{gcd}(2, 1)$

$2 = 2 \times 1 + 0$: $\text{gcd}(2, 1) = \text{gcd}(1, 0) = 1$

$\Rightarrow \text{gcd}(127, 91) = 1$,

after 5 division steps:

$127, 91, 36, 17, 2, 1, 0$
Euclidean algorithm examples

compute \( \gcd(127, 91) \)

binary Euclidean algorithm

127 and 91 both odd:

\[
gcd(127, 91) = gcd(91, 127 - 91) = gcd(91, 36)
\]

\[
= gcd(91, 9) \text{ (removed two 2s)}
\]

\[
gcd(91, 9) = gcd(9, 91 - 9) = gcd(9, 82)
\]

\[
= gcd(9, 41) \text{ (removed one 2)}
\]

\[
gcd(41, 9) = gcd(9, 41 - 9) = gcd(9, 32)
\]

\[
= gcd(9, 1) \text{ (removed five 2s)}
\]

\[
\Rightarrow \ gcd(127, 91) = 1, \text{ in 3 division-less steps}
\]

(127, 91), (91, 9), (41, 9), (9, 1)

note: binary euclid runs in \( O((\max(\log n, \log m))^2) \) bit operations
Linear congruences (i.e., modular inversion)
given modulus $m$, integers $a, b > 0$,
find integer $x$ such that $ax \equiv b \pmod{m}$

seen that:

$b$ must be a multiple of $\gcd(a,m)$
i.e.: $\gcd(a,m)|b$ is necessary condition
for solution to $ax \equiv b \pmod{m}$ to exist

i.e.: $ax \equiv b \pmod{m}$ solvable $\rightarrow \gcd(a,m)|b$

below constructive proof that $\gcd(a,m)|b$ suffices:
i.e.: $\gcd(a,m)|b \rightarrow ax \equiv b \pmod{m}$ solvable

conclusion:

$ax \equiv b \pmod{m}$ solvable $\iff \gcd(a,m)|b$
Solving \( ax \equiv \gcd(a,m) \pmod{m} \)
(suffices for \( ax \equiv b \pmod{m} \) with \( \gcd(a,m)|b \))

use previous example: \( a = 91, \ m = 127 \)

seen: \( \gcd(91,127) = 1 \),

thus try to solve \( 91x \equiv 1 \pmod{127} \) for \( x \)

combine related identities modulo 127:

1. \( 91 \times 0 \equiv 127 \pmod{127} \), trivially true
2. \( 91 \times 1 \equiv 91 \pmod{127} \), trivially true
3. \( 91 \times (-1) \equiv 36 \pmod{127} \): \((1) - 1 \times (2)\)
4. \( 91 \times 3 \equiv 19 \pmod{127} \): \((2) - 2 \times (3)\)
5. \( 91 \times (-4) \equiv 17 \pmod{127} \): \((3) - 1 \times (4)\)
6. \( 91 \times 7 \equiv 2 \pmod{127} \): \((4) - 1 \times (5)\)
7. \( 91 \times (-60) \equiv 1 \pmod{127} \): \((5) - 8 \times (6)\)

thus \( 91 \times 67 \equiv 1 \pmod{127} \): \( x = 67 \)
Solving $ax \equiv \gcd(a,m) \pmod{m}$

(suffices for $ax \equiv b \pmod{m}$ with $\gcd(a,m)|b$)

use previous example: $a = 91$, $m = 127$

seen: $\gcd(91,127)=1$,

thus try to solve $91x \equiv 1 \pmod{127}$ for $x$

combine related identities modulo 127:

(1) \hspace{1em} 91 \times 0 \equiv 127 \pmod{127}, \text{ trivially true }

(2) \hspace{1em} 91 \times 1 \equiv 91 \pmod{127}, \text{ trivially true }

(3) \hspace{1em} 91 \times (-1) \equiv 36 \pmod{127}: \hspace{1em} (1) - 1 \times (2)

(4) \hspace{1em} 91 \times 3 \equiv 19 \pmod{127}: \hspace{1em} (2) - 2 \times (3)

(5) \hspace{1em} 91 \times (-4) \equiv 17 \pmod{127}: \hspace{1em} (3) - 1 \times (4)

(6) \hspace{1em} 91 \times 7 \equiv 2 \pmod{127}: \hspace{1em} (4) - 1 \times (5)

(7) \hspace{1em} 91 \times (-60) \equiv 1 \pmod{127}: \hspace{1em} (5) - 8 \times (6)

thus $91 \times 67 \equiv 1 \pmod{127}: \hspace{1em} x = 67$
**Euclidean algorithm examples**

compute $\gcd(127,91)$

**standard Euclidean algorithm**

$127 = 1 \times 91 + 36$: $\gcd(127,91) = \gcd(91,36)$

$91 = 2 \times 36 + 19$: $\gcd(91,36) = \gcd(36,19)$

$36 = 1 \times 19 + 17$: $\gcd(36,19) = \gcd(19,17)$

$19 = 1 \times 17 + 2$: $\gcd(19,17) = \gcd(17,2)$

$17 = 8 \times 2 + 1$: $\gcd(17,2) = \gcd(2,1)$

$2 = 2 \times 1 + 0$: $\gcd(2,1) = \gcd(1,0) = 1$

$\Rightarrow \gcd(127,91) = 1$,

after 6 standard division steps:

$127, 91, 36, 19, 17, 2, 1, 0$

---

**Solving** $ax \equiv \gcd(a,m) \pmod m$

(suffices for $ax \equiv b \pmod m$ with $\gcd(a,m)|b$)

use previous example: $a = 91$, $m = 127$

seen: $\gcd(91,127)=1$,

thus try to solve $91x \equiv 1 \pmod{127}$ for $x$

combine related identities modulo 127:

1. $91 \times 0 \equiv 127 \pmod{127}$, trivially true
2. $91 \times 1 \equiv 91 \pmod{127}$, trivially true
3. $91 \times (-1) \equiv 36 \pmod{127}$: (1) - 1\times(2)
4. $91 \times 3 \equiv 19 \pmod{127}$: (2) - 2\times(3)
5. $91 \times (-4) \equiv 17 \pmod{127}$: (3) - 1\times(4)
6. $91 \times 7 \equiv 2 \pmod{127}$: (4) - 1\times(5)
7. $91 \times (-60) \equiv 1 \pmod{127}$: (5) - 8\times(6)

thus $91 \times 67 \equiv 1 \pmod{127}$: $x = 67$

**same sequences**
**Euclidean algorithm examples**

**compute gcd(127,91)**

**standard Euclidean algorithm**

- **127**
  - **91**\*91+36: gcd(127,91) = gcd(91,36)
  - **91**\*36+19: gcd(91,36) = gcd(36,19)
  - **36**\*19+17: gcd(36,19) = gcd(19,17)
  - **19**\*17+2: gcd(19,17) = gcd(17,2)
  - **17**\*8+2: gcd(17,2) = gcd(2,1)
  - **2**\*1+0: gcd(2,1) = gcd(1,0) = 1

⇒ gcd(127,91) = 1,

after 6 standard division steps:

127, 91, 36, 19, 17, 2, 1, 0

---

**Solving** \(ax \equiv \gcd(a,m) \pmod{m}\)

(suffices for \(ax \equiv b \pmod{m}\) with \(\gcd(a,m)\mid b\))

use previous example: \(a = 91, \ m = 127\)

seen: \(\gcd(91,127)=1,\)

thus try to solve \(91x \equiv 1 \pmod{127}\) for \(x\)

combine related identities modulo 127:

1. \(91 * 0 \equiv 127 \pmod{127}\), trivially true
2. \(91 * 1 \equiv 91 \pmod{127}\), trivially true
3. \(91 * (-1) \equiv 36 \pmod{127}\): \((1) - 1\times(2)\)
4. \(91 * 3 \equiv 19 \pmod{127}\): \((2) - 2\times(3)\)
5. \(91 * (-4) \equiv 17 \pmod{127}\): \((3) - 1\times(4)\)
6. \(91 * 7 \equiv 2 \pmod{127}\): \((4) - 1\times(5)\)
7. \(91 * (-60) \equiv 1 \pmod{127}\): \((5) - 8\times(6)\)

thus \(91 * 67 \equiv 1 \pmod{127}\): \(x = 67\)

---

same sequences

and same sequences
Euclidean algorithm examples

compute \( \text{gcd}(127,91) \)

standard Euclidean algorithm

\[
\begin{align*}
\text{gcd}(127,91) &= \text{gcd}(91,36) \\
\text{gcd}(91,36) &= \text{gcd}(36,19) \\
\text{gcd}(36,19) &= \text{gcd}(19,17) \\
\text{gcd}(19,17) &= \text{gcd}(17,2) \\
\text{gcd}(17,2) &= \text{gcd}(2,1) \\
\text{gcd}(2,1) &= \text{gcd}(1,0) = 1
\end{align*}
\]

⇒ \( \text{gcd}(127,91) = 1 \), after 6 standard division steps:

\( 127, 91, 36, 19, 17, 2, 1, 0 \)

Solving \( ax \equiv \text{gcd}(a,m) \pmod{m} \)

(suffices for \( ax \equiv b \pmod{m} \) with \( \text{gcd}(a,m)|b \))

use previous example: \( a = 91, m = 127 \)

seen: \( \text{gcd}(91,127)=1 \),

thus try to solve \( 91x \equiv 1 \pmod{127} \) for \( x \)

combine related identities modulo 127:

\[
\begin{align*}
(1) \quad 91 \times 0 & \equiv 127 \pmod{127}, \text{ trivially true} \\
(2) \quad 91 \times 1 & \equiv 91 \pmod{127}, \text{ trivially true} \\
(3) \quad 91 \times (-1) & \equiv 36 \pmod{127}: \quad (1) - 1 \times (2) \\
(4) \quad 91 \times 3 & \equiv 19 \pmod{127}: \quad (2) - 2 \times (3) \\
(5) \quad 91 \times (-4) & \equiv 17 \pmod{127}: \quad (3) - 1 \times (4) \\
(6) \quad 91 \times 7 & \equiv 2 \pmod{127}: \quad (4) - 1 \times (5) \\
(7) \quad 91 \times (-60) & \equiv 1 \pmod{127}: \quad (5) - 8 \times (6)
\end{align*}
\]

thus \( 91 \times 67 \equiv 1 \pmod{127} \): \( x = 67 \)

Thus in identities \( ay \equiv t \pmod{m} \):

- \( t \) follows sequence of Euclidean algorithm
- Euclidean sequence terminates at \( t = \text{gcd}(a,m) \) with \( y \) equal to \( x \) s.t. \( ax \equiv \text{gcd}(a,m) \pmod{m} \).

- this is constructive proof of
  
  \( \text{gcd}(a,m)|b \rightarrow ax \equiv b \pmod{m} \) solvable

- multipliers are quotients in Euclidean algorithm
Example $ax \equiv \gcd(a,m) \pmod{m}$ with $\gcd \neq 1$

let $a = 91$, $m = 147$ try to find $x$ such that $91x \equiv \gcd(91,147) \pmod{147}$ (known to be 7)

combine related identities modulo 147 and use the standard Euclidean algorithm:

(1) $91 \times 0 \equiv 147 \pmod{147}$, trivially true

(2) $91 \times 1 \equiv 91 \pmod{147}$, trivially true

(3) $91 \times (-1) \equiv 56 \pmod{147}$: $(1) - 1 \times (2)$

(4) $91 \times 2 \equiv 35 \pmod{147}$: $(2) - 1 \times (3)$

(5) $91 \times (-3) \equiv 21 \pmod{147}$: $(3) - 1 \times (4)$

(6) $91 \times 5 \equiv 14 \pmod{147}$: $(4) - 1 \times (5)$

(7) $91 \times (-8) \equiv 7 \pmod{147}$: $(5) - 1 \times (6)$

Thus $91 \times 139 \equiv 7 \pmod{147}$: $x = 139$
Same example again
using the smallest remainder variant:

(1) $91 \times 0 \equiv 147 \pmod{147}$
(2) $91 \times 1 \equiv 91 \pmod{147}$
(3) $91 \times (-2) \equiv -35 \pmod{147}$: \[(1) - 2 \times (2)\]
(4) $91 \times (-5) \equiv -14 \pmod{147}$: \[(2) + 3 \times (3)\]
(5) $91 \times 8 \equiv -7 \pmod{147}$: \[(3) - 2 \times (4)\]

thus $91 \times 139 \equiv 7 \pmod{147}$: $x = 139$

(sequence of multipliers as before, up to sign)
Same example again
using the binary variant, gets a bit messy:

(1) \( 91 \times 0 \equiv 147 \pmod{147} \)

(2) \( 91 \times 1 \equiv 91 \pmod{147} \)

(3) \( 91 \times (-1) \equiv 56 \pmod{147} \):
\( (1) - 1 \times (2) \)

\( \text{divide by 2, with } -1/2 \equiv (-1+147)/2 = 73: \)
\( 91 \times 73 \equiv 28 \pmod{147} \)

\( \text{divide by 2, with } 73/2 \equiv (73+147)/2 = 110: \)
\( 91 \times 110 \equiv 14 \pmod{147} \)

\( \text{divide by 2:} \)
\( 91 \times 55 \equiv 7 \pmod{147} \)

thus \( 91 \times 55 \equiv 7 \pmod{147} \): \( x = 55 \)

\( \Rightarrow \) other solution than before … (147 not prime)
Remarks
for prime $p$ and all $a$ with $0 < a < p$:

• $\text{gcd}(a,p) = 1$
• therefore $\exists x \text{ s.t. } ax \equiv 1 \pmod{p}$, the *multiplicative inverse* of $a$ modulo $p$
• careful runtime analyses of (all) Euclids:
  time $O((\log p)^2)$ to calculate $a^{-1}$ modulo $p$
• $a^p \equiv a \pmod{p}$ (Fermat) $\rightarrow$
  $a^p * a^{-2} \equiv a * a^{-2} \pmod{p} \rightarrow a^{p-2} \equiv a^{-1} \pmod{p}$
  $\Rightarrow a^{-1}$ modulo $p$ in time $O((\log p)^3)$ using
  modular exponentiation, only for prime $p$

given $ax \equiv b \pmod{m}$,
  $k$ with $ax + km = b$ follows as $(ax - b)/m$
Application: Chinese remaindering

thm. Let \( p \) and \( q \) be coprime integers and let \( x_p, x_q \in \mathbb{Z} \) with \( 0 \leq x_p < p \) and \( 0 \leq x_q < q \). then:

\[ \exists! \; x \in \mathbb{Z} \text{ with } 0 \leq x < pq \text{ such that } \]

\[ x \equiv x_p \pmod{p} \quad \text{and} \quad x \equiv x_q \pmod{q} \]

proof by unique construction: if \( x \) exists, then \( x \equiv x_p \pmod{p} \rightarrow x = x_p + kp \rightarrow \)

\[ (\text{with } x \equiv x_q \pmod{q}) \quad x_p + kp \equiv x_q \pmod{q} \rightarrow \]

\[ (\text{since } \gcd(p, q) = 1) \quad k \equiv (x_q - x_p) p^{-1} \pmod{q} \rightarrow \]

\[ x = x_p + p((x_q - x_p) p^{-1} \pmod{q}) \]. This \( x \) works and \( 0 \leq x \leq p - 1 + p(q - 1) = pq - 1 \)
Applications of Chinese remaindering

- alternative arithmetic with large integers: let $p_i$ be $i$th prime. Represent large $n$ as
  \[(n \mod p_1, n \mod p_2, \ldots, n \mod p_k)\]
  for some $k$ such that $n < p_1 * p_2 * \ldots * p_k$.
  allows components-wise $+$, $-$, $*$
  (not at all widely used)

- the RSA public key cryptosystem:
  both in proof that it works
  and to make it fast

- counting