Algorithm Analysis - Arithmetic Examples

- Addition
- Multiplication
- Bigger Example - RSA cryptography
  - Modular Arithmetic
  - Modular Exponentiation
  - Primality Testing (Fermat’s little theorem) – Probabilistic algorithm
  - Euclid’s Algorithm for gcd (greatest common divisor)
  - Modular division
  - Private key cryptography
  - Public key RSA cryptography
Factoring: Given a number $N$, express it as a product of its prime numbers

Primality: Given a number $N$, determine whether it is prime

Which one is harder?

This gulf will be a key to modern encryption algorithms (RSA, etc), which makes secure communication (internet, etc.) currently reasonable.
Addition

- Addition of two numbers of length $n$
  - Pad smaller number with leading 0’s if necessary
Addition

- Addition of two numbers of length $n$
  - Pad smaller number with leading 0’s if necessary
  - At each step 3 single digit numbers (including carry) are added to create a 2 digit number (could have a leading 0)
  - Time needed: ?
Addition

- Addition of two numbers of length $n$
  - Pad smaller number with leading 0’s if necessary
  - At each step 3 single digit numbers (including carry) are added to create a 2 digit number (could have a leading 0)
  - Time needed: $c_0$ (overhead) + $c_1 \cdot n$ (which is linear)
  - Complexity $O(n)$, where $n$ is the size (in bits) of the numbers
  - Base an issue?
    - Any number $N$ can represented by $\log_b(N+1)$ symbols
    - Difference between decimal and binary (3.32)
    - Constant factor between any two bases - Thus we’ll usually just use binary examples
  - Can we go faster?
Addition

Addition of two numbers of length $n$
- Pad smaller number with leading 0’s if necessary
- At each step 3 single digit numbers (including carry) are added to create a 2 digit number (could have a leading 0)
- Time needed: $c_0$ (overhead) + $c_1 \cdot n$ (which is linear)
- Complexity $O(n)$, where $n$ is the size (in bits) of the numbers
- Base an issue? -
  - Any number $N$ can represented by $\log_b(N+1)$ symbols
  - Difference between decimal and binary (3.32)
  - Constant factor between any two bases - Thus we’ll usually just use binary examples
- Can we go faster?
  - Why not?
  - Thus, addition is $\Theta(n)$
Isn’t addition in a computer just 1 time step?

Can be, if the size of the numbers is fixed
- 32/64/128 bit computer word - 1 step if we assume numbers within the word size because of hardware parallelism.
- Would if bigger (double int), but still fixed. Then still $c \cdot (1)$.
- $O(n)$ when $n$ can be arbitrarily large.

For most applications, the problem size parameter(s) will not be tied to the specific data elements (numbers, characters, etc.) but to the number of data elements.

What is the complexity of adding two arrays each containing $m$ fixed size numbers - common

What is the complexity of adding two arrays each containing $m$ numbers of arbitrary size $n$ - less common

Typically work with fixed size data elements, and our concern is number of elements, rather than varying size of the elements, but for the moment…
Multiplication - Classic

- 2 binary numbers of arbitrary length $n$
- Note that doubling and dividing by two are just a left shift or right shift in binary

```
  1 1 0
*  1 0 1
  ---
  1 1 0
  1 1 0
  0 0 0
  1 1 0
```

- Complexity?
Multiplication - Classic

- 2 binary numbers of arbitrary length $n$
- Note that doubling and dividing by two are just a left shift or right shift in binary
  
  \[
  \begin{array}{c}
  1 & 1 & 0 \\
  \times & 1 & 0 & 1 \\
  \hline
  & 1 & 1 & 0 \\
  & & 1 & 1 & 0 \\
  & 0 & 0 & 0 \\
  \hline
  1 & 1 & 0 \\
  \end{array}
  \]

- Complexity? – $n-1$ adds (two rows at a time) * $n$ rows = $n^2$
- Can we do better?
### Multiplication *a la Francais / Russe*

<table>
<thead>
<tr>
<th>$x$</th>
<th>$y$</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>11</td>
</tr>
</tbody>
</table>

At each step double $x$ and half $y$. Then add up versions of $x$ where $y$ is odd.
11 in binary is 1011 - We show it from bottom to top in binary column
Makes sense since each position in binary is a doubling of the previous position anytime it is a 1 (odd number)
Interesting mix of binary and decimal
Obvious equality, yet it gives us a divide and conquer opportunity since we halve the size at each step.

Prep us for Modular exponentiation which uses the same basic approach.

\[
x \cdot y = \begin{cases} 
2(x \cdot \lfloor y/2 \rfloor) & \text{if } y \text{ is even} \\
 x + 2(x \cdot \lfloor y/2 \rfloor) & \text{if } y \text{ is odd}
\end{cases}
\]
Function `multiply(x, y)`

- **Input:** Two $n$-bit integers $x$ and $y$, where $y \geq 0$
- **Output:** Their product

1. If $y = 0$: return 0
2. $z = \text{multiply}(x, \text{floor}(y/2))$
3. If $y$ is even: return $2z$
4. Else: return $x + 2z$

\[
x \cdot y = \begin{cases} 
2(x \cdot \lfloor y/2 \rfloor) & \text{if } y \text{ is even} \\
x + 2(x \cdot \lfloor y/2 \rfloor) & \text{if } y \text{ is odd}
\end{cases}
\]
### Multiplication *a la Francais / Russe (15 * 8)*

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>y_{binary}</th>
<th># of x’s</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>8</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>30</td>
<td>4</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>60</td>
<td>2</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>120</td>
<td>1</td>
<td>1</td>
<td>8</td>
</tr>
<tr>
<td>120</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Any $x$ value added (when $y$ is odd) in keeps getting doubled to its appropriate contribution as we unfold the recursion.
function multiply(x, y)
   Input: Two $n$-bit integers $x$ and $y$, where $y \geq 0$
   Output: Their product

   if $y = 0$: return 0
   $z = \text{multiply}(x, \text{floor}(y/2))$
   if $y$ is even: return $2z$
   else: return $x + 2z$

- Is it better than classical multiplication?
function multiply(x, y)
    Input: Two $n$-bit integers $x$ and $y$, where $y \geq 0$
    Output: Their product

    if $y = 0$: return 0
    $z = \text{multiply}(x, \text{floor}(y/2))$
    if $y$ is even: return $2z$
    else: return $x + 2z$

Is it better than classical multiplication?
Don’t need to know all the times tables, just need to double and half, which could be an advantage - especially easy in binary
Complexity?
- $n$ function calls where $n$ is the length in bits of the binary $y$ ($n = \log_2 y$)
- Each call has how many $n$-bit adds?
- Thus big-OH complexity is?
Multiplication a la Francais / Russe

\[
\begin{array}{cc}
  x & y \\
  15 & 11 \\
  1111 & 1011 \\
\end{array}
\]

At each step double \( x \) and half \( y \). Then add up versions of \( x \) where \( y \) is odd.

In fact when numbers are in binary, they are basically the same algorithm.
Complexity of Multiplication

- Is multiplication $O(n^2)$?
  - Could we come up with a multiplication algorithm which is slower than $O(n^2)$?
  - Know we can do at least this well, real question is can we come up with a faster one

- Is multiplication $\Theta(n^2)$?
  - In other words, is this the best we can do
  - Multiplication problem vs particular algorithms
Complexity of Multiplication

- Is multiplication $O(n^2)$?
  - Could we come up with a multiplication algorithm which is slower than $O(n^2)$?
  - Know we can do at least this well, real question is can we come up with a faster one

- Is multiplication $\Theta(n^2)$?
  - In other words, is this the best we can do
  - Multiplication problem vs particular algorithms

- Not $\Theta(n^2)$. It turns out we can do better, as we will see later
  - Can we prove lower bounds? - Sometimes (e.g. addition)

- Division is also $O(n^2)$
Convenient in many cases when we want to restrict potentially large numbers to a specific range: time of day

- Can work with numbers in the computer word range (e.g. 64 bits) while still working with numbers which could initially be much larger

- $8 \pmod{3} = 2 = 11 \pmod{3} = 14 \pmod{3}$: Congruent or in the same equivalence class. Three equivalence classes for mod 3 (those with remainder 0, 1, or 2). All congruent numbers can be substituted for each other in modular arithmetic.
  - $8 + 4 = 14 + 7 = 8 + 1 = 0 \pmod{3}$
  - $8 \cdot 4 = 14 \cdot 7 = 8 \cdot 1 = 2 \pmod{3}$

- Simplified form is a number between 0 and mod-1, (not negative, etc.)

Thus during arithmetic we can reduce intermediate results to their remainder Modulo $N$ at any stage, which leads to significant efficiencies
  - $2^{600} \pmod{31} = (2^5)^{120} = 32^{120} = 1^{120} = 1 \pmod{31}$
Modular Arithmetic Complexity

- Assume we start with numbers Mod $N$ - Thus they are numbers between 0 and $N-1$ which means length of numbers is $n = \log(N)$
  - If not already in Modulus range then Modular reduction requires an initial standard division which is $O(n^2)$
- Modular Addition is $O(n)$ since it requires one addition and possibly one subtraction if sum is greater than $N$.
- Modular Multiplication is $O(n^2)$
  - Just standard multiplication followed by a division if product exceeds $N$
- Modular division is $O(n^3)$ – more later
For encryption we need to compute $x^y \mod N$ for values of $x$, $y$, and $N$ which are several hundred bits long:

$$38295034738204523523\ldots \mod 5345098234509345823\ldots$$

Way too big (and slow) unless we keep all intermediate numbers in the modulus range $N$.

Algorithm to solve $x^y \mod N$?

How about multiplying by $x \mod N$, $y$ times, each time doing modular reduction to keep the number less than $N$?

- Multiplication is $n^2$, where $n (\approx \log(N))$ is length of \{x, y, N\} in bits. Relatively fast since each intermediate result is less than $N$ thus each multiplication time is $\log^2(N)$. Much better than the huge number you would end up with for non-modular exponentiation.

- However, must still do $y$ multiplies where $y$ can be huge. Requires $2^y$ multiplications, with $y$ being potentially hundreds of bits long.
There is a faster algorithm

- In decimal $x^{347} = x^{300} \cdot x^{40} \cdot x^7$
- $x^{21} = x^{20} \cdot x^1$
  - change the 21 to binary: $10101_2$
- $x^{10101} = x^{10000} \cdot x^{100} \cdot x^1$ (binary) = $x^{16} \cdot x^4 \cdot x^1 = x^{21}$ (powers of 2)
- Start bottom up and square $x$ each time and multiply the powers corresponding to 1’s in the binary representation of $y$.
- Keep results in the mod range as you go (optional)
- Thus there are only $\log_2(y)$ multiplications
Modular Exponentiation

Note that we can use the same style trick as in multiplication a la Francais

\[ x^y = \begin{cases} 
(x^{\lfloor y/2 \rfloor})^2 & \text{if } y \text{ is even} \\
 x \cdot (x^{\lfloor y/2 \rfloor})^2 & \text{if } y \text{ is odd} 
\end{cases} \]
Recursive Modexp Algorithm

\[
x^y = \begin{cases} 
(x^{\lfloor y/2 \rfloor})^2 & \text{if } y \text{ is even} \\
(x^{\lfloor y/2 \rfloor})^2 \cdot x & \text{if } y \text{ is odd}
\end{cases}
\]

**function modexp** (x, y, N)

Input: Two \( n \)-bit integers \( x \) and \( N \), an integer exponent \( y \) (arbitrarily large)

Output: \( x^y \mod N \)

if \( y = 0 \): return 1

\[ z = \text{modexp}(x, \text{floor}(y/2), N) \]

if \( y \) is even: return \( z^2 \mod N \)

else: return \( x \cdot z^2 \mod N \)
Example $2^{25}$ mod 20

modexp ($x$, $y$, $N$)
if $y = 0$: return 1
$z = \text{modexp}(x, \ \text{floor}(y/2), \ N)$
if $y$ is even: return $z^2 \mod N$
else: return $x \cdot z^2 \mod N$

<table>
<thead>
<tr>
<th>$x$</th>
<th>$y$</th>
<th>$y_{\text{binary}}$</th>
<th>power of $x$</th>
<th>$z$</th>
<th>return value</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>25</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Example $2^{25}$ mod 20

modexp $(x, y, N)$
if $y = 0$: return 1
z = modexp$(x, \text{ floor}(y/2), N)$
if $y$ is even: return $z^2 \mod N$
else: return $x \cdot z^2 \mod N$

<table>
<thead>
<tr>
<th>$x$</th>
<th>$y$</th>
<th>$y_{binary}$</th>
<th>power of $x$</th>
<th>$z$</th>
<th>return value</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>25</td>
<td>1</td>
<td>$x^1$</td>
<td>16</td>
<td>512 mod 20 = 12</td>
</tr>
<tr>
<td>2</td>
<td>12</td>
<td>0</td>
<td>$x^2$</td>
<td>4</td>
<td>16</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
<td>0</td>
<td>$x^4$</td>
<td>8</td>
<td>64 mod 20 = 4</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>1</td>
<td>$x^8$</td>
<td>2</td>
<td>8</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>1</td>
<td>$x^{16}$</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>25</td>
<td></td>
<td>$x^{25}$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Note that if we drop the mod the algorithm does regular exponentiation
Algorithm Analysis

- $x$ and $N$ are integers of length $n$ bits, $y$ is an integer of length $m$ bits (i.e. $m = \log_2 y$, $n = \log_2 x = \log_2 N$)
- Each multiply and divide is $n^2$
- How many multiplies/divides - calling depth which is $m$
- Thus Complexity is $O(n^2m)$
- If we assume $y$ is of length similar to $x$ and $N$ then complexity is $O(n^3)$
- Very efficient compared to the exponential alternatives
function modexp(x, y, N)  //Iterative version
Input: Two \( n \)-bit integers \( x \) and \( N \), an integer exponent \( y \) (arbitrarily large)
Output: \( x^y \mod N \)

if \( y = 0 \): return 1
\( i = y; \ r = 1; \ z = x \mod N \)
while \( i > 0 \)
  if \( i \) is odd: \( r = r \cdot z \mod N \)
  \( z = z^2 \mod N \)
  \( i = \text{floor}(i/2) \)
return \( r \)

- Same big-O complexity
- Iteration can save overhead of calling stack (optimizing compilers)
- Some feel that recursion is more elegant
- Either way is reasonable
Given an integer $p$, we want to state if it is prime or not (i.e. it is only divisible by 1 and itself)

Could check all factors, but...

All known approaches to factoring take exponential time

How about trying to divide by all numbers less than $p/2$?

We have Fermat’s little theorem

If $p$ is prime, then $a^{p-1} \equiv 1 \mod p$

for any $a$ such that $1 \leq a < p$

Some examples
function primality($N$)
Input: Positive integer $N$
Output: yes/no

// $a$ is random positive integer between 2 and $N$-1
$a = \text{uniform}(2 \ldots N-1)$
if (modexp($a$, $N$-1, $N$) == 1): return yes
else: return no

Is this correct?
function primality($N$)
Input: Positive integer $N$
Output: maybe/no

// $a$ is random positive integer between 2 and $N$-1
$a = \text{uniform}(2 \ldots N\text{-}1)$
if (modexp($a$, $N$-1, $N$) == 1): return maybe a prime
else: return no
Primality

- How often does a composite number $c$ pass the test (i.e. does $a^{c-1} \equiv 1 \mod c$ for a random $a$ such that $1 < a < c$)
  - Note always passes if we use $a = 1$
- Fairly rare, and becomes increasingly rare with the size of the number
- We can prove that the probability is less than .5 (very conservative)
- So how do we extend the algorithm to give us more confidence?
function primality2(N)
Input: Positive integer N
Output: yes/no

Choose $a_1 \ldots a_k \ (k < N)$ unique random integers between 2 and $N-1$
if $a_i^{N-1} \equiv 1 \mod N$ for all $a_i$ then
  return yes with probability $\geq 1 - 1/(2^k)$
else:
  return no

- Is this algorithm correct? - Randomized Algorithm
- What is its complexity?
- Demo
Primality notes

- Primality testing is efficient! - $O(n^3)$
- Carmichael numbers
  - There are an infinite but rare set of composite numbers which pass the Fermat test for all $a_i$
  - These can be dealt with by a more refined primality test
- Generating random primes
  - An $n$ bit random number has approximately a 1 in $n$ chance of being prime
- Random Prime Generation Algorithm
Primality notes

● Primality testing is efficient! - $O(n^3)$
● Carmichael numbers
  - There are an infinite but rare set of composite numbers which pass the Fermat test for all $a_i$
  - These can be dealt with by a more refined primality test
● Generating random primes
  - An $n$ bit random number has approximately a 1 in $n$ chance of being prime
● Random Prime Generation Algorithm
  - Randomly choose an $n$ bit number
  - Run Primality Test
  - If passes, return the number, else choose another number and repeat
  - $O(n)$ average tries to find a prime, times the Primality test complexity $O(n^3)$: Total is $O(n^4)$
● In practical algorithms $a=2$ is sufficient or $a=\{2, 3, 5\}$ for being really safe since for large numbers it is extremely rare for a composite to pass the test with $a=2$. 