Algorithms with numbers (1)
CISC5835, Computer Algorithms
CIS, Fordham Univ.

Instructor: X. Zhang

Fall 2018

Acknowledgement

- The set of slides have used materials from the following resources
  - Slides for textbook by Dr. Y. Chen from Shanghai Jiaotong Univ.
  - Slides from Dr. M. Nicolescu from UNR
  - Slides sets by Dr. K. Wayne from Princeton
    - which in turn have borrowed materials from other resources
  - Other online resources

Outline

- Motivation
- Algorithm for integer addition
- Algorithms for multiplication
  - grade-school algorithm
  - recursive algorithm
  - divide-and-conquer algorithm
- Division
- Exponentiation

Algorithms for integer arithmetics

- We study adding/multiplying two integers
  - earliest algorithms!
  - mostly what you learned in grade school!

- Analyze running time of these algorithms by counting number of elementary operations on individual bits when adding/multiplying two N-bits long ints (so called bit complexity)
  - input size N: the length of operands
  - for example, to add two N-bits integer numbers, we need to O(n) bit operations (such as adding three bits together).

Practical consideration

- But, why bother?
  - Given that with a single (machine) instruction, one can add/subtract/multiply integers whose size in bits is within word length of computer – 32, or 64.
  - i.e., they are implemented in hardware
- Bit complexity of arithmetic operations algorithms captures amount of hardware (transistors and wires) necessary for implementing algorithm using digital logic circuit.
  - e.g., number of logic gates needed …

Support for Big Integer

- What if we need to handle numbers that are several thousand bits long?
  - need to implement arithmetic operations of large integers in software.

- Ex: Use an array of ints to store the (decimal or binary) digits of integer,
  - int digits[3]={2,4,6}; //represents 642
  - int digits1[10]={3,4,5,7,0,7,8}; //represent 8707543
  - int bindigits[4]={1,0,1,0}; //represent 0101, i.e., 3
- Algorithms studied here are presented assuming base 2
  - those for other base (e.g., base 10) are similar

Support for Big Integer*

- But notice that each int variable can store up to 64 bits, and we can add/subtract/multiple 64-bits ints in one machine instruction
- To save space and time, one could divide big integer into chunks of 63 bits long, and store each chunk in a int

- 101100… 10 1111110…101110 111111…0000110011110111

  63 bits 63 bits 63 bits

  - int chunks[3]={32, 121254, 145246};

    //represent a value of

$$32 \times 2^{126} + 121254 \times 2^{63} + 145246$$

When adding two numbers, adding corresponding chunks together, carry over are added to the next chunks…

Outline

- Motivation
- Algorithm for integer addition
- Algorithms for multiplication
  - grade-school algorithm
  - recursive algorithm
  - divide-and-conquer algorithm
- Division
- Exponentiation

Adding two binary number

Carry:	1			1	1	1		
		1	1	0	1	0	1	(53)
		1	0	0	0	1	1	(35)
	1	0	1	1	0	0	0	(88)

Algorithm for adding integers

- Sum of any three single-digit numbers is at most two digits long. (holds for any base)
  - In binary the largest possible sum of three single-bit numbers is 3, which is a 2-bit number.
  - In decimal, the max possible sum of three single digit numbers is 27 (9+9+9), which is a 2-digit number
- **Algorithms for addition (in any base):**
  - align their right-hand ends,
  - perform a single right-to-left pass
    - the sum is computed digit by digit, maintaining overflow as a carry
    - since we know each individual sum is a two-digit number, the carry is always a single digit, and so at any given step, three single-digit numbers are added)

Question

Given two binary numbers x and y, how long does our algorithm take to add them?

We want the answer expressed as a **function of the size of the input**: the number of bits of x and y.

Suppose x and y are each n bits long. Then the sum of x and y is $n+1$ bits at most, and each individual bit of this sum gets computed in a fixed amount of time.
The total running time for the addition algorithm is therefore of the form $c_0 + c_1 n$, where c_0 and c_1 are some constants, i.e., $O(n)$.

Question

Is there a faster algorithm?

In order to add two n-bit numbers we must at least read them and write down the answer, and even that requires n operations.
So the addition algorithm is *optimal*, up to multiplicative constants!

Ubiquitous $\log_2 N$

- $\log_2 N$ is the power to which you need to raise 2 in order to obtain N.
  - e.g., $\log_2 8=3$ (as $2^3=8$), $\log_2 1024=10$(as $2^{10}=1024$).
- Going backward, it can be seen as the number of times you must halve N to get down to 1, more precisely: $\lceil log_2 N \rceil$
  - e.g., N=10, $\lceil log_2 10 \rceil = 4$; N=8, $\lceil log_2 8 \rceil = 3$
- It is the number of bits in binary representation of N, more precisely: $\lceil log_2 (N+1) \rceil$
  - e.g., hw1 questions
- It is the depth of a complete binary tree with N nodes, more precisely: $\lfloor log_2 N \rfloor$
  - height of a heap with N nodes …
- It is even the sum 1+1/2+1/3+...+1/N, to within a constant factor.

Multiplication in base 2

$$
\begin{array}{cccccccc r}
 & & & & 1 & 1 & 0 & 1 & \text{(binary 13)} \\
 & & & \times & 1 & 0 & 1 & 1 & \text{(binary 11)} \\
\hline
 & & & & 1 & 1 & 0 & 1 & \text{(1101 times 1)} \\
 & & & 1 & 1 & 0 & 1 & & \text{(1101 times 1, shifted once)} \\
 & & 0 & 0 & 0 & 0 & & & \text{(1101 times 0, shifted twice)} \\
+ & 1 & 1 & 0 & 1 & & & & \text{(1101 times 1, shifted thrice)} \\
\hline
1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & \text{(binary 143)}
\end{array}
$$

The grade-school algorithm for <u>multiplying two numbers x and y:</u>

- create an array of intermediate sums, each representing the product of x by a single digit of y.
- these values are appropriately left-shifted and then added up.

Multiplication in base 2

				1	1	0	1	(binary 13)
			×	1	0	1	1	(binary 11)
				1	1	0	1	(1101 times 1)
			1	1	0	1		(1101 times 1, shifted once)
		0	0	0	0			(1101 times 0, shifted twice)
+	1	1	0	1				(1101 times 1, shifted thrice)
1	0	0	0	1	1	1	1	(binary 143)

If x and y are both n bits, then there are n intermediate rows, with lengths of up to 2n bits.

The total time taken to add up these rows, doing two numbers at a time, is

$$\underbrace{O(n) + \ldots + O(n)}_{n-1 \text{ times}}.$$

which is $O(n^2)$.

Multiplication: top-down approach

MULTIPLY(x, y)
// Two n-bit integers x and y, where $y \geq 0$.

1. **if** $y = 0$ **then** return 0
2. $z =$ MULTIPLY$(x, \lfloor y/2 \rfloor)$
3. **if** y is even **then** return $2z$
4. **else** return $x + 2z$

- Totally, n recursive calls, because at each call y is halved (i.e., n decreases by 1)
- In each recursive call: a division by 2 (right shift), a test for odd/even (looking up the last bit); a multiplication by 2 (left shift); and possibly one addition => a total of O(n) bit operations.
- The total time taken is thus $O(n^2)$.

Divide-and-conquer

Suppose x and y are two n-bit integers, and assume for convenience that n is *a power of* 2.

Lemma

For every n there exists an n' with $n \leq n' \leq 2n$ such that n' a power of 2.

As a first step toward multiplying x and y, we split each of them into their **left and right halves**, which are $n/2$ bits long:

$$x = \boxed{x_L}\,\boxed{x_R} = 2^{n/2}x_L + x_R$$
$$y = \boxed{y_L}\,\boxed{y_R} = 2^{n/2}y_L + y_R.$$

$$xy = (2^{n/2}x_L + x_R)(2^{n/2}y_L + y_R) = 2^n x_L y_L + 2^{n/2}(x_L y_R + x_R y_L) + x_R y_R.$$

The additions take linear time, as do the multiplications by powers of 2. The significant operations are the four $n/2$-**bit multiplications**; these we can handle by *four recursive calls*.

Running time

$$xy = (2^{n/2}x_L + x_R)(2^{n/2}y_L + y_R) = 2^n x_L y_L + 2^{n/2}(x_L y_R + x_R y_L) + x_R y_R.$$

Our method for multiplying n-bit numbers

1. making recursive calls to multiply these four pairs of n/2-bit numbers,

2. evaluates the above expression in O(n) time

Writing T(n) for the overall running time on n-bit inputs, we get the recurrence relation:

$$T(n) = 4T(n/2) + O(n)$$

By master theorem, $T(n)=(n^2)$

Can we do better?

$$xy = (2^{n/2}x_L + x_R)(2^{n/2}y_L + y_R) = 2^n x_L y_L + 2^{n/2}(x_L y_R + x_R y_L) + x_R y_R.$$

By **Gauss**'s trick, three multiplications, $x_L y_L$, $x_R y_R$, and $(x_L + x_R)(y_L + y_R)$, suffice, as

$$x_L y_R + x_R y_L = (x_L + x_R)(y_L + y_R) - x_L y_L - x_R y_R.$$

The recurrence relation:

$$T(n) = 3T(n/2) + O(n)$$

By Master Theorem: $T(n) = n^{log_2 3}$

Integer Division

$\textsc{Divide}(x, y)$
// Two n-bit integers x and y, where $y \geq 1$.

1. **if** $x = 0$ **then** return $(q, r) = (0, 0)$
2. $(q, r) = \textsc{Divide}(\lfloor x/2 \rfloor, y)$
3. $q = 2 \cdot q$, $r = 2 \cdot r$
4. **if** x is odd **then** $r = r + 1$
5. **if** $r \geq y$ **then** $r = r - y$, $q = q + 1$
6. return (q, r)

Readings

- Chapter 1.1