Lecture 16

Algorithms & Data Structures

Goldsmiths Computing

February 25, 2019

Outline

Outline

Introduction

Fixed point

Multiplication

Floating point

Lecture

1. Random number generation
2. Comparison sorts
   - insertion sort
   - merge sort
   - heap sort
   - quick sort
   - $\Theta(N \log N)$ complexity bound
3. Shuffling
   - several broken shuffle versions
   - one correct $\Theta(N \log N)$ shuffle
   - $\Theta(N)$ Fisher-Yates shuffle

Lab

Catchup and future plans

VLE activities

Graphs quiz

VLE activities (cont'd)

Graphs quiz

VLE activities (cont'd)

Graphs quiz

VLE activities (cont'd)

Graphs submission

Outline

Motivation

Representing:

- integers within a range
- continuous sequence of place-value digits

Useful and practical:

- simple to implement in hardware
- reasonable properties
- ... but some unexpected behaviours too

Definition

- data type for a number
- fixed number of digits
- implicit scaling factor
    - number represented is integer value multiplied by scaling factor

Unsigned integers

Fixed-width of binary digits

- assumed infinite leftward zeros
- radix (decimal) point at the right-hand end of the fixed-width field
- scaling factor of 1

Representation

···	0	0	0	0	0	0	0	0	0	1	1	0	1	0	0	1	0	0	1

(16-bit fixed point integer)

Overflow

What happens when a calculation would put a 1 in the infinite sea of zeros?

hardware wraparound, carry flag

Overflow

What happens when a calculation would put a 1 in the infinite sea of zeros?

hardware	wraparound, carry flag
C++	wraparound

Overflow

What happens when a calculation would put a 1 in the infinite sea of zeros?

hardware	wraparound, carry flag
C++	wraparound
Java	bad luck, no unsigned integers

Signed integers

Many possible representations:

- sign-magnitude

  sign high bit (0: positive; 1: negative)

  magnitude remaining bits

- ones complement

  -x ¬x

- twos-complement

  -x ¬x + 1

Twos complement

What's so good about twos-complement?

- only one zero
- addition, subtraction and multiplication all the same as unsigned

Representation

···	0	0	0	0	0	0	0	0	1	0	1	0	0	1	1	0	1	1	1
···	1	1	1	1	0	0	1	1	1	1	0	1	1	1	0	1	1	1	0

In practice, all current systems use twos-complement.

function NEG(x)
  r ← ¬x
  return r + 1
end function

Overflow

What happens when a calculation would put a non-sign bit in the infinite sea of sign bits?

hardware wraparound, carry flag

Overflow

What happens when a calculation would put a non-sign bit in the infinite sea of sign bits?

hardware wraparound, carry flag

Java wraparound

Overflow

What happens when a calculation would put a non-sign bit in the infinite sea of sign bits?

hardware	wraparound, carry flag
Java	wraparound
C++	bad luck, undefined behaviour

Absolute value

`abs` (C++ cstdlib) or `Math.abs` (Java):

- return (as an `int`) the absolute (non-negative) value of its argument

```
function ABS(x)
    if x < 0 then
        return NEG(x)
    else
        return x
    end if
end function
```

- does this always return a non-negative answer?

Population count

`popcnt` (x86 instruction)

- return how many one bits are set in the (unsigned) integer argument.

Divide-and-conquer implementation:

function POPCNT(x)
  return POPCNTW(x,W) ▷ W is the integer width
end function
function POPCNTW(x,w)
  if w = 1 **then**
    return x
  else
    nw ← $\frac{w}{2}$ ▷ w assumed to be a power of 2
    lo ← POPCNTW(x & $2^{\mathrm{nw}} - 1$, nw)
    hi ← POPCNTW($\left\lfloor \frac{x}{2^{\mathrm{nw}}} \right\rfloor$, nw)
    return lo + hi
  end if
end function

Population count

Parallel divide-and-conquer implementation:

function POPCNT(x) ▷ Assume W = 32

$x \leftarrow (x \,\&\, \texttt{0x55555555}) + (\lfloor \frac{x}{2} \rfloor \,\&\, \texttt{0x55555555})$

$x \leftarrow (x \,\&\, \texttt{0x33333333}) + (\lfloor \frac{x}{4} \rfloor \,\&\, \texttt{0x33333333})$

$x \leftarrow (x \,\&\, \texttt{0x0f0f0f0f}) + (\lfloor \frac{x}{16} \rfloor \,\&\, \texttt{0x0f0f0f0f})$

$x \leftarrow (x \,\&\, \texttt{0x00ff00ff}) + (\lfloor \frac{x}{256} \rfloor \,\&\, \texttt{0x00ff00ff})$

$x \leftarrow (x \,\&\, \texttt{0x0000ffff}) + (\lfloor \frac{x}{65536} \rfloor \,\&\, \texttt{0x0000ffff})$

return x

end function

Work

1. Reading
   - Henry S. Warren, Jr. *Hacker's Delight*, Addison-Wesley (2003)
     - sections 5-1, 7-1

Outline

Motivation

- working with numbers as a data structure
- everyone knows how to multiply
- almost no-one knows how to multiply efficiently

Previously

Numbers as array of digits (binary: bits)

- numbers have a *width* w, at least $1 + \log_b(n)$

Logical operations

and(x,y)	return the bitwise logical and of x and y
xor(x,y)	return the bitwise exclusive-or of x and y

Previously

Numbers as array of digits (binary: bits)

- numbers have a *width* w, at least $1 + \log_b(n)$

Logical operations

and(x,y)	return the bitwise logical and of x and y
xor(x,y)	return the bitwise exclusive-or of x and y

Arithmetic operations

add(x,y)	return the sum of x and y
sub(x,y)	return the difference between x and y
shift(x,n)	return x multiplied by the base n times

Previously

Numbers as array of digits (binary: bits)

- numbers have a *width* w, at least $1 + \log_b(n)$

Logical operations

and(x,y)	return the bitwise logical and of x and y
xor(x,y)	return the bitwise exclusive-or of x and y

Arithmetic operations

add(x,y)	return the sum of x and y
sub(x,y)	return the difference between x and y
shift(x,n)	return x multiplied by the base n times

Complexity

- until now, $\Theta(1)$
- in fact, $\Theta(w) \sim \Theta(log(n))$

(logarithmic factor is usually irrelevant, or width is taken as constant)

Problem

Given these basic operations:

- how do we implement multiplication?
- how efficient is it?

Example

123×135:

Example

123×135:

Primary (old-)school multiplication

$$
\begin{array}{cccccc}
 & & & 1 & 2 & 3 \\
\times & & & 1 & 3 & 5 \\
\hline
 & & & & 1 & 5 \\
 & & & 1 & 0 & \\
 & & & 5 & & \\
 & & & & 9 & \\
 & & & 6 & & \\
 & & 3 & & & \\
 & & & 3 & & \\
 & & 2 & & & \\
 & 1 & & & & \\
\hline
 & 1 & 6 & 6 & 0 & 5
\end{array}
$$

Example

123×135:

Primary school multiplication

$\times$	100	20	3
100	10000	2000	300
30	3000	600	90
5	500	100	15

Complexity analysis

time

For each digit in x

- multiply with each digit in y.

Assume x and y are each of width w

$$\Longrightarrow \Theta(w^2)$$

Divide and conquer

Write

- $x = x_{hi} \times b^{w/2} + x_{lo}$
- $y = y_{hi} \times b^{w/2} + y_{lo}$

Then $x \times y = x_{hi}y_{hi}\, b^{w} + (x_{hi}y_{lo} + x_{lo}y_{hi})b^{w/2} + x_{lo}y_{lo}$

Example

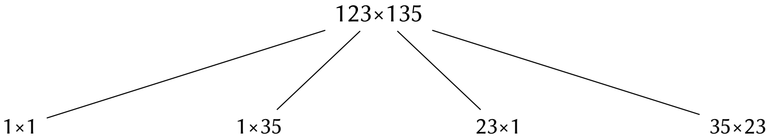
$$x \times y = x_{hi}y_{hi}\, b^{w} + (x_{hi}y_{lo} + x_{lo}y_{hi})b^{w/2} + x_{lo}y_{lo}$$

$$\begin{aligned}
123 \times 135 &= (1 \times 100 + 23) \times (1 \times 100 + 35) \\
&= 1 \times 1 \times 10000 + (23 \times 1 + 1 \times 35) \times 100 + 23 \times 35 \\
&= 10000 + 5800 + 805
\end{aligned}$$

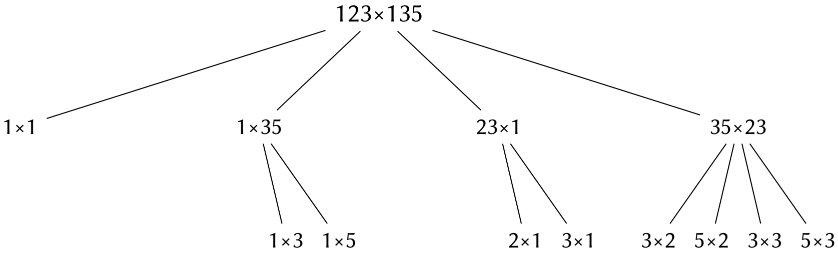
Recursion tree

123×135

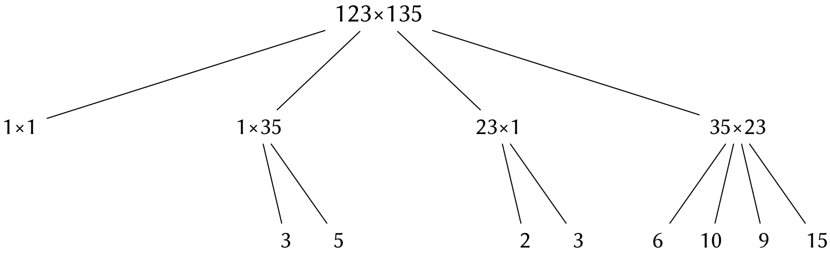
Recursion tree



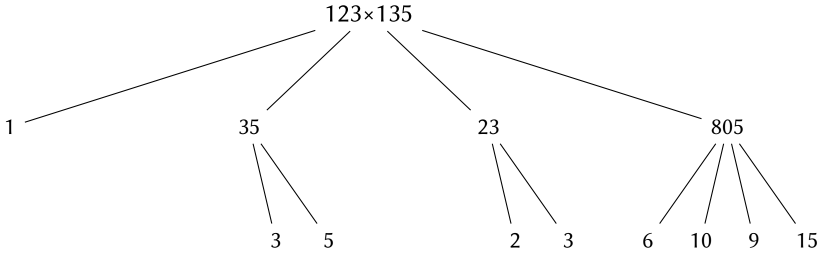
Recursion tree



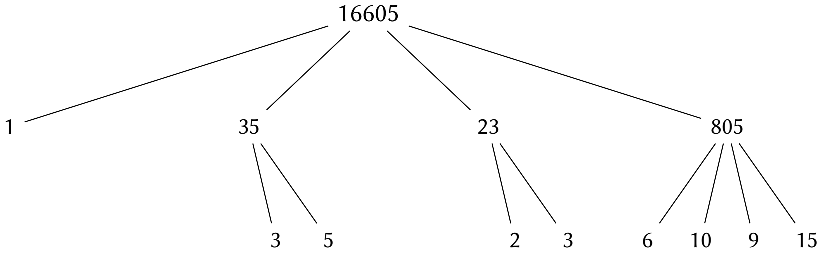
Recursion tree



Recursion tree



Recursion tree



Complexity analysis

time

$\mathrm{x} \times \mathrm{y} = \mathrm{x_{hi}y_{hi}}\ \mathrm{b^{w}} + (\mathrm{x_{hi}y_{lo}} + \mathrm{x_{lo}y_{hi}})\mathrm{b^{w/2}} + \mathrm{x_{lo}y_{lo}}$

- $T_w = 4T_{w/2} + O(w)$

Complexity analysis

time

$x \times y = x_{hi}y_{hi}\, b^{w} + (x_{hi}y_{lo} + x_{lo}y_{hi})b^{w/2} + x_{lo}y_{lo}$

- $T_w = 4T_{w/2} + O(w)$

solution

Use the master theorem:

- $a = 4, b = 2, c = 1 \implies$ case 1
- $\log_b a = \log_2(4) = 2$

$$\implies T(w) \in \Theta(w^2)$$

Divide and conquer, Karatsuba

Still with

- $x = x_{hi} \times b^{w/2} + x_{lo}$
- $y = y_{hi} \times b^{w/2} + y_{lo}$

Calculate

- $z_{hi} = x_{hi} \times y_{hi}$
- $z_{lo} = x_{lo} \times y_{lo}$
- $z_c = (x_{hi} + x_{lo}) \times (y_{hi} + y_{lo})$

Then $x \times y = z_{hi}\, b^{w} + (z_c - z_{hi} - z_{lo}) b^{w/2} + z_{lo}$

Divide and conquer, Karatsuba

Still with

- $x = x_{hi} \times b^{w/2} + x_{lo}$
- $y = y_{hi} \times b^{w/2} + y_{lo}$

Calculate

- $z_{hi} = x_{hi} \times y_{hi}$
- $z_{lo} = x_{lo} \times y_{lo}$
- $z_c = (x_{hi} + x_{lo}) \times (y_{hi} + y_{lo})$

Then $x \times y = z_{hi}\, b^w + (z_c - z_{hi} - z_{lo}) b^{w/2} + z_{lo}$

Example

123×135:

- $z_{hi} = 1$; $z_{lo} = 35 \times 23 = 805$; $z_c = 36 \times 24 = 864$

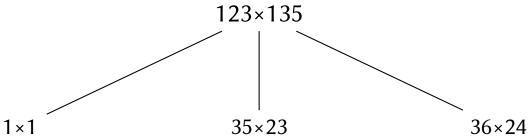
so:

$$\begin{aligned} 123 \times 135 &= 1 \times 10000 + (864 - 805 - 1) \times 100 + 805 \\ &= 10000 + 5800 + 805 \\ &= 16605 \end{aligned}$$

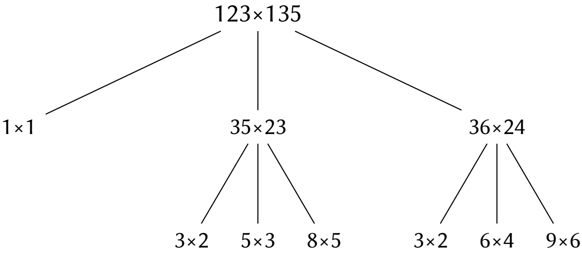
Recursion tree

123×135

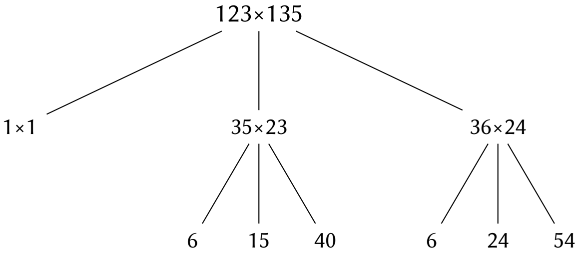
Recursion tree



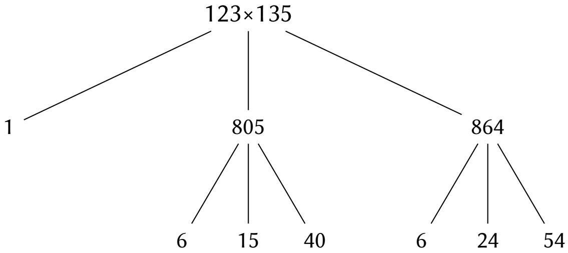
Recursion tree



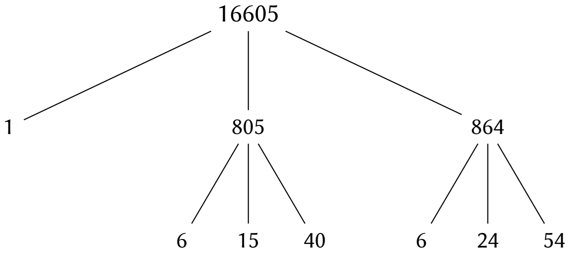
Recursion tree



Recursion tree



Recursion tree



Complexity analysis

time

$x \times y = z_{hi}\, b^{w} + (z_{c} - z_{hi} - z_{lo})b^{w/2} + z_{lo}$

- $T_w = 3T_{w/2} + O(w)$

Complexity analysis

time

$x \times y = z_{hi}\, b^{w} + (z_c - z_{hi} - z_{lo}) b^{w/2} + z_{lo}$

- $T_w = 3T_{w/2} + O(w)$

solution

Use the master theorem:

- $a = 3, b = 2, c = 1 \Longrightarrow$ case 1
- $\log_b a = \log_2(3) = 1.58$

$$\Longrightarrow T(w) \in \Theta(w^{1.58})$$

Work

1. Reading
    - CLRS, section 4.2: matrix multiplication

Outline

Motivation

Represent a wider range of numbers than with fixed point

- fixed point: constant absolute precision
- floating point: constant relative precision

Convenient approximation to Real numbers

- but only an approximation
- ... some unexpected behaviours too

Definition

Floating point behaviour is defined by an engineering standard:

- IEEE 754 (1985, revised 2008)

Implemented by most hardware platforms:

- floating-point units in CPUs
  - (software support for FPU features varies)
- graphics cards (CUDA, OpenGL)
  - and other coprocessors

Operations

Operations on floats:

- **add** return the sum of two floating point numbers
- **sub** return the difference of two floating point numbers
- **mul** return the product of two floating point numbers
- **div** return the quotient of two floating point numbers
- **sqrt** return the square root of one floating point number

Operations on floating point units:

- **rounding mode** should rounding go towards $+\infty$, 0, $-\infty$ or even?
- **trapping** should the FPU generate an exception for conditions such as overflow or divide by zero?

General idea

Represent a number n as:

$$n = \text{sign} \times \text{significand} \times 2^{\text{exponent}}$$

- **sign** 1 or -1
- **significand** number in $[1, 2)$
- **exponent** $\lfloor \log_2 n \rfloor$

Represent this in a fixed-size field using:

- **sign bit** 0 (positive) or 1 (negative)
- **mantissa** *fractional* part of significand
- **exponent** exponent + bias

$$n = (-1)^s \times (1 + m) \times 2^{e-B}$$

Single-precision

- 32-bit quantity:
  - 1 sign bit
  - 8 exponent bits
    - bias is 127, range is ± 2^{-126} to 2^{127}
  - 23 mantissa bits
    - plus “hidden bit” gives 24 binary (~7 decimal) digits of precision

Single-precision

- 32-bit quantity:
  - 1 sign bit
  - 8 exponent bits
    - bias is 127, range is $\pm 2^{-126}$ to 2^{127}
  - 23 mantissa bits
    - plus “hidden bit” gives 24 binary (~7 decimal) digits of precision

Representation

Single-precision

- 32-bit quantity:
  - 1 sign bit
  - 8 exponent bits
    - bias is 127, range is ± 2^{-126} to 2^{127}
  - 23 mantissa bits
    - plus "hidden bit" gives 24 binary (~7 decimal) digits of precision

Representation

Example

$$0.5 = 1 \times (1 + 0) \times 2^{-1}$$

sign	0
mantissa	0
exponent	126 (`0x7e`)
overall	`0x3f000000`

Zero?

No representation for zero in this scheme

$$\lfloor \log_2(x) \rfloor = -\infty$$

Special representation of zero:

exponent field 0

mantissa 0

sign 0 or 1

Double-precision

- 64-bit quantity:
  - 1 sign bit
  - 11 exponent bits
    - bias is 1023, range is ± 2^{-1022} to 2^{1023}
  - 52 mantissa bits
    - plus “hidden bit” gives 53 binary (~16 decimal) digits of precision

Double-precision

- 64-bit quantity:
  - 1 sign bit
  - 11 exponent bits
    - bias is 1023, range is ± 2^{-1022} to 2^{1023}
  - 52 mantissa bits
    - plus “hidden bit” gives 53 binary (~16 decimal) digits of precision

Representation

Double-precision

- 64-bit quantity:
  - 1 sign bit
  - 11 exponent bits
    - bias is 1023, range is ± 2^{-1022} to 2^{1023}
  - 52 mantissa bits
    - plus “hidden bit” gives 53 binary (~16 decimal) digits of precision

Representation

Example

$$0.75 = 1 \times (1 + 0.5) \times 2^{-1}$$

sign	0
mantissa	shift(1, 51)
exponent	1022 (`0x3fe`)
overall	`0x3fe8000000000000`

Epsilon

Floating point has a larger *range* than *precision*

- calculations with floating points will usually not give an exactly representable answer
  - (even if the input numbers were exact)

Epsilon

ε is the smallest float which you can add to 1.0 and get an answer that isn't 1.0:

single-precision $2^{-24} + 2^{-47}$

double-precision $2^{-53} + 2^{-105}$

For all $0 < x < \varepsilon$

$$1 + x \longrightarrow 1$$

Inverse square root

Game programming history

- need to take $f(x) = \frac{1}{\sqrt{x}}$ often and quickly

Use $\log_2((1 + m) \times 2^{e-B}) \approx e - B + m$:

```
int i = *(int *) &f;
i = 0x5f3759df - (i >> 1);
f = *(float *) &i;
```

This was good in 1999 (*Quake III Arena*)

- nowadays we have hardware to do this
- SSE `rsqrtss`

Work

1. Reading
   - David Goldberg, *What every computer scientist should know about floating point arithmetic*, Computing (1991)