Commentary: Facebook’s Algorithm vs. Democracy

The social network may feel like a modern town square, but thanks to its tangle of algorithms, it’s nothing like the public forums of the past. The company determines... what we see and learn on its social network. The result has been a loss of focus on critical national issues, an erosion of civil disagreement, and a threat to democracy itself.

Facebook’s reach is expansive. About two-thirds of American adults have a profile on Facebook. They spend an average of 50 minutes a day on the site, and 60% of them count on Facebook to deliver at least some of their political news. Young people, especially, get their news from Facebook, and they are particularly bad at distinguishing real news sources from fake ones.

Administrivia

- Lab 1 due next Thursday (1/26)
- Homework 2 released today, due 1/31
- Lab 0 scores available on Canvas

- Monday’s lecture will be given by Kevin (TA)
Unsigned Multiplication in C

<table>
<thead>
<tr>
<th>Operands:</th>
<th>( u )</th>
<th>( \cdots )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( w ) bits</td>
<td>( \ast )</td>
<td>( v )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>True Product:</th>
<th>( u \cdot v )</th>
<th>( \cdots )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( 2w ) bits</td>
<td>( \cdots )</td>
<td>( \cdots )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Discard ( w ) bits:</th>
<th>( \cdots )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( w ) bits</td>
<td>( \cdots )</td>
</tr>
</tbody>
</table>

- **Standard Multiplication Function**
  - Ignores high order \( w \) bits

- **Implements Modular Arithmetic**
  - \( \text{UMult}_{w}(u, v) = u \cdot v \mod 2^w \)
Multiplication with shift and add

- **Operation** \( u \ll k \) gives \( u \times 2^k \)
  - Both signed and unsigned

**Operands:** \( w \) bits

**True Product:** \( w + k \) bits

**Discard** \( k \) bits: \( w \) bits

- **Examples:**
  - \( u \ll 3 \)  
    \(=\)  \( u \times 8 \)  \(2^3\)
  - \( u \ll 5 - u \ll 3 \)  
    \(=\)  \( u \times 2^4 \)  \((32 - 8)\)
  - Most machines shift and add faster than multiply
    - *Compiler generates this code automatically*
Number Representation Revisited

- What can we represent in one word?
  - Signed and Unsigned Integers
  - Characters (ASCII)
  - Addresses

- How do we encode the following:
  - Real numbers (e.g. 3.14159)
  - Very large numbers (e.g. $6.02 \times 10^{23}$) 
  - Very small numbers (e.g. $6.626 \times 10^{-34}$)
  - Special numbers (e.g. $\infty$, NaN)
Floating Point Topics

- Fractional binary numbers
- IEEE floating-point standard
- Floating-point operations and rounding
- Floating-point in C

- There are many more details that we won’t cover
  - It’s a 58-page standard...
Representation of Fractions

- “Binary Point,” like decimal point, signifies boundary between integer and fractional parts:

  Example 6-bit representation:

  $2^0 \quad 2^{-1} \quad 2^{-2} \quad 2^{-3} \quad 2^{-4}$

  $0.5 \quad 0.25 \quad 0.125 \quad 0.0625$

- **Example:** $10.1010_2 = 1 \times 2^1 + 1 \times 2^{-1} + 1 \times 2^{-3} = 2.625_{10}$

- Binary point numbers that match the 6-bit format above range from $0 (00.0000_2)$ to $3.9375 (11.1111_2)$
Scientific Notation (Decimal)

- **Normalized form**: exactly one digit (non-zero) to left of decimal point

- Alternatives to representing \(1/1,000,000,000\)
  - Normalized:
    - \(1.0 \times 10^{-9}\)
  - Not normalized:
    - \(0.1 \times 10^{-8}, 10.0 \times 10^{-10}\)
Scientific Notation (Binary)

- Computer arithmetic that supports this called floating point due to the “floating” of the binary point
  - Declare such variable in C as `float` (or `double`)
Scientific Notation Translation

- Convert from scientific notation to binary point
  - Perform the multiplication by shifting the decimal until the exponent disappears
    - Example: $1.011_2 \times 2^4 = 10110_2 = 22_{10}$
    - Example: $1.011_2 \times 2^{-2} = 0.01011_2 = 0.34375_{10}$

- Convert from binary point to normalized scientific notation
  - Distribute out exponents until binary point is to the right of a single digit
    - Example: $1101.001_2 = 1.101001_2 \times 2^3$
    - $2^{-1} = 0.5$, $2^{-2} = 0.25$, $2^{-3} = 0.125$, $2^{-4} = 0.0625$

- Practice: Convert $11.375_{10}$ to binary scientific notation
  - $8 + 2 + 1 + 0.25 + 0.125$
  - $2^3 + 2^1 + 2^0 + 2^{-2} + 2^{-3} = 1011.011_2 = \boxed{1.011011_2 \times 2^3}$

- Practice: Convert $1/5$ to binary
  - $\frac{1}{5} - \frac{1}{8} = \frac{3}{40}$, $\frac{3}{40} - \frac{1}{16} = \frac{1}{80}$, $\frac{1}{80} = \frac{1}{16} \left( \frac{1}{5} \right)$
  - $\frac{1}{5} = \boxed{0.0011_2}$
  - $\uparrow$ same #, but shifted by 4
Floating Point Topics

- Fractional binary numbers
- **IEEE floating-point standard**
- Floating-point operations and rounding
- Floating-point in C

- There are many more details that we won’t cover
  - It’s a 58-page standard...
IEEE Floating Point

- **IEEE 754**
  - Established in 1985 as uniform standard for floating point arithmetic
  - Main idea: make numerically sensitive programs portable
  - Specifies two things: representation and result of floating operations
  - Now supported by all major CPUs

- Driven by numerical concerns
  - **Scientists**/numerical analysts want them to be as **real** as possible
  - **Engineers** want them to be **easy to implement** and **fast**
  - In the end:
    - Scientists mostly won out
    - Nice standards for rounding, overflow, underflow, but...
    - Hard to make fast in hardware
    - Float operations can be an **order of magnitude** slower than integer ops


Floating Point Encoding

- Use normalized, base 2 scientific notation:
  - Value: \( \pm 1 \times \text{Mantissa} \times 2^{\text{Exponent}} \)
  - Bit Fields: \( (-1)^{S} \times 1.M \times 2^{(E-\text{bias})} \)

- Representation Scheme:
  - Sign bit (0 is positive, 1 is negative)
  - Mantissa (a.k.a. significand) is the fractional part of the number in normalized form and encoded in bit vector \( \text{M} \)
  - Exponent weights the value by a (possibly negative) power of 2 and encoded in the bit vector \( \text{E} \)
The Exponent Field

- **Use biased notation**
  - Read exponent as unsigned, but with *bias of* $2^{w-1} - 1 = 127$
  - Representable exponents roughly $\frac{1}{2}$ positive and $\frac{1}{2}$ negative
  - Exponent 0 ($\text{Exp} = 0$) is represented as $E = 0b 0111 1111$

- **Why biased?**
  - Makes floating point arithmetic easier
  - Makes somewhat compatible with two’s complement

- **Practice:** To encode in biased notation, add the bias then encode in unsigned:
  - $\text{Exp} = 1 \quad \rightarrow \quad 128 \quad \rightarrow \quad E = 0b \quad 1000 \quad 0000$
  - $\text{Exp} = 127 \quad \rightarrow \quad 254 \quad \rightarrow \quad E = 0b \quad 1111 \quad 1110$
  - $\text{Exp} = -63 \quad \rightarrow \quad 64 \quad \rightarrow \quad E = 0b \quad 0100 \quad 0000$
The Mantissa (Fraction) Field

\[ (-1)^S \times (1 \cdot M) \times 2^{(E-\text{bias})} \]

- Note the implicit 1 in front of the M bit vector
  - Example: \( 0b\ 0011\ 1111\ 1100\ 0000\ 0000\ 0000\ 0000\ 0000 \) is read as \( 1.1_2 = 1.5_{10}, \text{not } 0.1_2 = 0.5_{10} \)
  - Gives us an extra bit of precision

- Mantissa “limits”
  - Low values near \( M = 0b0...0 \) are close to \( 2^E \)
  - High values near \( M = 0b1...1 \) are close to \( 2^{E+1} \)
Peer Instruction Question

What is the correct value encoded by the following floating point number?

0b 0 10000000 11000000000000000000000

\[ \text{Exp} = E - 127 = 1 \]

\[ \text{Man} = 1.111_2 \]

Vote at http://PollEv.com/justinh

A. +0.75
B. +1.5
C. +2.75
D. +3.5
E. We’re lost...

\[ 1.11_2 \times 2^1 = 11.1_2 = 2 + 0.5 = 3.5 \]
Precision and Accuracy

- **Precision** is a count of the number of bits in a computer word used to represent a value
  - Capacity for accuracy

- **Accuracy** is a measure of the difference between the actual value of a number and its computer representation
  - High precision permits high accuracy but doesn’t guarantee it. It is possible to have high precision but low accuracy.

**Example:** float pi = 3.14;
- \( \pi \) will be represented using all 24 bits of the mantissa (highly precise), but is only an approximation (not accurate)
Need Greater Precision?

- Double Precision (vs. Single Precision) in 64 bits

- C variable declared as `double`
- Exponent bias is now $2^{10} - 1 = 1023$, $bias = 2^{\omega_d} - 1$
- Advantages: greater precision (larger mantissa), greater range (larger exponent)
- Disadvantages: more bits used, slower to manipulate
Representing Very Small Numbers

- But wait... what happened to zero?
  - Using standard encoding 0x00000000 = \( 1.0 \times 2^{-127} \neq 0 \)
  - **Special case:** \( E \) and \( M \) all zeros = 0
    - Two zeros! But at least 0x00000000 = 0 like integers
      \( 0x80000000 = -\infty \)

- New numbers closest to 0:
  - \( a = 1.0...0_2 \times 2^{-126} = 2^{-126} \)
  - \( b = 1.0...01_2 \times 2^{-126} = 2^{-126} + 2^{-149} \)
  - Normalization and implicit 1 are to blame
  - **Special case:** \( E = 0, M \neq 0 \) are **denormalized numbers**
Denorm Numbers

- Denormalized numbers
  - No leading 1
  - Uses implicit exponent of –126 even though \( E = 0x00 \)

- Denormalized numbers close the gap between zero and the smallest normalized number
  - Smallest norm: \( \pm 1.0...0_{\text{two}} \times 2^{-126} = \pm 2^{-126} \)
  - Smallest denorm: \( \pm 0.0...01_{\text{two}} \times 2^{-126} = \pm 2^{-149} \)
    - There is still a gap between zero and the smallest denormalized number

This is extra (non-testable) material
Other Special Cases

- **E = 0xFF, M = 0**: ±∞
  - e.g. division by 0
  - Still work in comparisons!

- **E = 0xFF, M ≠ 0**: Not a Number (NaN)
  - e.g. square root of negative number, 0/0, ∞–∞
  - NaN propagates through computations
  - Value of M can be useful in debugging

- New largest value (besides ∞)?
  - E = 0xFF has now been taken!
  - E = 0xFE has largest: 1.1…12×2^{127} = 2^{128} – 2^{104}
## Floating Point Encoding Summary

<table>
<thead>
<tr>
<th>Exponent</th>
<th>Mantissa</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00</td>
<td>0</td>
<td>± 0</td>
</tr>
<tr>
<td>0x00</td>
<td>non-zero</td>
<td>± denorm num</td>
</tr>
<tr>
<td>0x01 - 0xFE</td>
<td>anything</td>
<td>± norm num</td>
</tr>
<tr>
<td>0xFF</td>
<td>0</td>
<td>± ∞</td>
</tr>
<tr>
<td>0xFF</td>
<td>non-zero</td>
<td>NaN</td>
</tr>
</tbody>
</table>

- **smallest E (all 0's)**
- **everything else**
- **largest E (all 1's)**
Distribution of Values

- What ranges are NOT representable?
  - Between largest norm and infinity: Overflow
  - Between zero and smallest denorm: Underflow
  - Between norm numbers? Rounding

- Given a FP number, what’s the bit pattern of the next largest representable number?
  - What is this “step” when Exp = 0? $2^{-23}$
  - What is this “step” when Exp = 100? $2^{77}$

- Distribution of values is denser toward zero
Floating Point Topics

- Fractional binary numbers
- IEEE floating-point standard
- **Floating-point operations** and rounding
- Floating-point in C

- There are many more details that we won’t cover
  - It’s a 58-page standard...
Floating Point Operations: Basic Idea

Value = \((-1)^s \times \text{Mantissa} \times 2^{\text{Exponent}}\)

- \(x +_f y = \text{Round}(x + y)\)
- \(x \times_f y = \text{Round}(x \times y)\)

**Basic idea for floating point operations:**
- First, **compute the exact result**
- Then **round** the result to make it fit into desired precision:
  - Possibly over/underflow if exponent outside of range
  - Possibly drop least-significant bits of mantissa to fit into M bit vector
Floating Point Addition

\[ (-1)^{S_1} \times \text{Man}_1 \times 2^{E_1} + (-1)^{S_2} \times \text{Man}_2 \times 2^{E_2} \]

- Assume \( E_1 > E_2 \)

**Exact Result:** \((-1)^{S} \times \text{Man} \times 2^{E}\)

- Sign \( S \), mantissa \( \text{Man} \):
  - Result of signed align & add
- Exponent \( E \): \( E_1 \)

**Adjustments:**

- If \( \text{Man} \geq 2 \), shift \( \text{Man} \) right, increment \( E \)
- if \( \text{Man} < 1 \), shift \( \text{Man} \) left \( k \) positions, decrement \( E \) by \( k \)
- Over/underflow if \( E \) out of range
- Round \( \text{Man} \) to fit mantissa precision

Line up the binary points!

\[
\begin{align*}
1.010 * 2^2 & \quad + \quad 1.000 * 2^{-1} \\
1.010 * 2^2 & \quad + \quad 0.0001 * 2^2
\end{align*}
\]
Floating Point Multiplication

- \((-1)^{S_1} \times M_1 \times 2^{E_1} \times (-1)^{S_2} \times M_2 \times 2^{E_2}\)

- Exact Result: \((-1)^S \times M \times 2^E\)
  - Sign \(S\): \(s_1 \oplus s_2\)
  - Mantissa \(Man\): \(M_1 \times M_2\)
  - Exponent \(E\): \(E_1 + E_2\)

- Adjustments:
  - If \(Man \geq 2\), shift \(Man\) right, increment \(E\)
  - Over/underflow if \(E\) out of range
  - Round \(Man\) to fit mantissa precision
Mathematical Properties of FP Operations

- Exponent overflow yields $+\infty$ or $-\infty$
- Floats with value $+\infty$, $-\infty$, and NaN can be used in operations
  - Result usually still $+\infty$, $-\infty$, or NaN; but not always intuitive
- Floating point operations do not work like real math, due to rounding
  - Not associative: $(3.14+1e100)-1e100 \neq 3.14+(1e100-1e100)$
    - $0 \neq 3.14$
  - Not distributive: $100*(0.1+0.2) \neq 100*0.1+100*0.2$
    - $30.000000000000003553 \neq 30$
  - Not cumulative
    - Repeatedly adding a very small number to a large one may do nothing
Summary

- Floating point approximates real numbers:
  - Handles large numbers, small numbers, special numbers
  - Exponent in biased notation (bias = $2^{w-1}-1$)
    - Outside of representable exponents is overflow and underflow
  - Mantissa approximates fractional portion of binary point
    - Implicit leading 1 (normalized) except in special cases
    - Exceeding length causes rounding

<table>
<thead>
<tr>
<th>Exponent</th>
<th>Mantissa</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00</td>
<td>0</td>
<td>± 0</td>
</tr>
<tr>
<td>0x00</td>
<td>non-zero</td>
<td>± denorm num</td>
</tr>
<tr>
<td>0x01 – 0xFE</td>
<td>anything</td>
<td>± norm num</td>
</tr>
<tr>
<td>0xFF</td>
<td>0</td>
<td>± ∞</td>
</tr>
<tr>
<td>0xFF</td>
<td>non-zero</td>
<td>NaN</td>
</tr>
</tbody>
</table>
An example that applies the IEEE Floating Point concepts to a smaller (8-bit) representation scheme. These slides expand on material covered today, so while you don’t need to read these, the information is “fair game.”
Tiny Floating Point Example

8-bit Floating Point Representation
- The sign bit is in the most significant bit (MSB)
- The next four bits are the exponent, with a bias of $2^{4-1} - 1 = 7$
- The last three bits are the mantissa

Same general form as IEEE Format
- Normalized binary scientific point notation
- Similar special cases for 0, denormalized numbers, NaN, $\infty$
# Dynamic Range (Positive Only)

<table>
<thead>
<tr>
<th>S E M</th>
<th>Exp</th>
<th>Value</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0000 000</td>
<td>-6</td>
<td>0</td>
<td>closest to zero</td>
</tr>
<tr>
<td>0 0000 001</td>
<td>-6</td>
<td>1/8*1/64 = 1/512</td>
<td></td>
</tr>
<tr>
<td>0 0000 010</td>
<td>-6</td>
<td>2/8*1/64 = 2/512</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 0000 110</td>
<td>-6</td>
<td>6/8*1/64 = 6/512</td>
<td>largest denorm</td>
</tr>
<tr>
<td>0 0000 111</td>
<td>-6</td>
<td>7/8*1/64 = 7/512</td>
<td></td>
</tr>
<tr>
<td>0 0001 000</td>
<td>-6</td>
<td>8/8*1/64 = 8/512</td>
<td>smallest norm</td>
</tr>
<tr>
<td>0 0001 001</td>
<td>-6</td>
<td>9/8*1/64 = 9/512</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 0110 110</td>
<td>-1</td>
<td>14/8*1/2 = 14/16</td>
<td>closest to 1 below</td>
</tr>
<tr>
<td>0 0110 111</td>
<td>-1</td>
<td>15/8*1/2 = 15/16</td>
<td></td>
</tr>
<tr>
<td>0 0111 000</td>
<td>0</td>
<td>8/8*1 = 1</td>
<td>closest to 1 above</td>
</tr>
<tr>
<td>0 0111 001</td>
<td>0</td>
<td>9/8*1 = 9/8</td>
<td></td>
</tr>
<tr>
<td>0 0111 010</td>
<td>0</td>
<td>10/8*1 = 10/8</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 1110 110</td>
<td>7</td>
<td>14/8*128 = 224</td>
<td>largest norm</td>
</tr>
<tr>
<td>0 1110 111</td>
<td>7</td>
<td>15/8*128 = 240</td>
<td></td>
</tr>
<tr>
<td>0 1111 000</td>
<td>n/a</td>
<td>inf</td>
<td></td>
</tr>
</tbody>
</table>
Special Properties of Encoding

- Floating point zero ($0^+$) exactly the same bits as integer zero
  - All bits = 0

- Can (Almost) Use Unsigned Integer Comparison
  - Must first compare sign bits
  - Must consider $0^- = 0^+ = 0$
  - NaNs problematic
    - Will be greater than any other values
    - What should comparison yield?
  - Otherwise OK
    - Denorm vs. normalized
    - Normalized vs. infinity