CSCI 136
Data Structures & Advanced Programming

Bitwise Operations

Today's Outline

- Bit operations
  - Useful for data structures in general
- `BIterator.java`: an iterator for enumerating the individual bits in the binary representation of an Integer

Representing Numbers

Humans usually think of numbers in *base 10*

- E.g.: 3,470,265, -4312, 0

3470265 is shorthand for

$$3 \cdot 10^6 + 4 \cdot 10^5 + 7 \cdot 10^4 + 0 \cdot 10^3 + 2 \cdot 10^2 + 6 \cdot 10^0 + 5 \cdot 10^0$$

- Each power of 10 has a coefficient in range 0-9
  - A "digit"
- Negative numbers have a distinguishing mark "-"
- A "carry" happens when two digits sum to more than 9

Representing Numbers

But we could do this with powers of *any integer*

Ex: Base 2 (binary)

- Powers of 2 instead of powers of 10
- Only two "digits" (bits): 0 and 1

$147_{10} = 128_{10} + 16_{10} + 2_{10} + 1_{10} =$

$1 * 2^7 + 0 \cdot 2^6 + 0 \cdot 2^5 + 1 \cdot 2^4 + 0 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0$

$= 10010011_2$

- So, $147_{10} = 10010011_2$

Representing Numbers in Hardware

Hardware stores numbers as *fixed width values*

- Every value has same number of bits (say 32 or 64)
- Ex: $23_{10} = 10111_2$ has form
  - 00000000 00000000 00000000 00010111

In lab, we converted from base 10 to base 2

```
public static String numInBinary(int n) {
        if (n <= 1)
            return "" + n%2;

        return numInBinary(n/2) + n%2;
}
```

numInBinary(int n)

- What was our strategy for writing (recursive) `numInBinary`?
  - Use mod to isolate the least significant bit
  - Divide by 2 and recurse

```
public static String numInBinary(int n) {
        if (n <= 1)
            return "" + n%2;

        return numInBinary(n/2) + n%2;
}
```

Bitwise Operations

- We can use *bitwise* operations to manipulate the 1s and 0s in the binary representation:
  - Bitwise 'and': &
    - b_1 & b_2 is 1 if b_1=b_2=1 and 0 otherwise
  - Bitwise 'or': |
    - b_1 | b_2 is 0 if b_1=b_2=0 and 1 otherwise
- Also useful: bit shifts
  - Bit shift left: << (fills 'holes' on right with 0s)
  - Bit shift right: >> (fills 'holes' on left with 0s)
    - But only for non-negative numbers

& and |

- Given two integers a and b, the *bitwise or* expression `a | b` returns an integer s.t.
  - At each bit position, the result has a 1 if that bit position had a 1 in EITHER a OR b
  - `6 | 12 = ?` 0110 | 1100 = 1110
- Given two integers a and b, the *bitwise and* expression `a & b` returns an integer s.t.
  - At each bit position, the result has a 1 if that bit position had a 1 in BOTH a AND b
  - `6 & 12 = ?` 0110 & 1100 = 0100

>> and <<

- Given two integers `a` and `i`, the expression
  `(a << i)` returns `(a * 2^i)`
  - Why? It shifts all bits left by `i` positions
  - `1 << 4 = ?` 00001 << 4 = 10000
- Given two integers `a` and `i`, the expression
  `(a >> i)` returns `(a / 2^i)`
  - Why? It shifts all bits right by `i` positions
  - `1 >> 4 = ?` 00001 >> 4 = 00000
  - `97 >> 3 = ?` (97 = 1100001)
    1100001 >> 3 = 1100
- Be careful about shifting left and "overflow"!!!

What About Negative Numbers?

With 32-bit representation we could store values from 0 up to $2^{32} - 1$.

What if we want negative numbers?

Idea:

- Use highest-order/most-significant/leftmost bit to encode *sign* of number
  - 0 for non-negative, 1 for negative
- Example: 4-bit numbers
  - 1111 is no longer 15
  - It's "negative something"…but what??

Two's-Complement

Java stores negative values in *two's-complement* representation

- Take a positive number in binary
  - 23_{10} = 00000000 00000000 00000000 00010111
- Flip all of the bits
  - 11111111 11111111 11111111 11101000
- Add 1
  - 11111111 11111111 11111111 11101001
- Note: left-most bit becomes 1
- "Negative 0" equals 0
- For negative numbers, >> fills with 1s, not 0s!
  - To fill with 0s, use >>> instead!

Revisiting numInBinary(int n)

- How would we rewrite a recursive `numInBinary` using bit shifts and bitwise operations?

```
public static String numInBinary(int n) {
    if (n <= 1) // no non-zero digits
        return "" + n;
    return numInBinary(n >> 1) + (n & 1);
}
```

Revisiting numInBinary(int n)

- How would we write an iterative `numInBinary` using bit shifts and bitwise operations?

```
public static String numInBinary(int n,
                                 int width) {
    String result = "";
    for(int i = 0; i < width; i++)
        if ((n & (1<<i)) == 0)
            result = 0 + result;
        else
            result = 1 + result;
    return result;
}
```

BIterator.java

- Goal:
  - Take a number `n`, and yield its bits (0 or 1) from least significant bit to most significant bit
  - For example, 1011 would yield: 1, 1, 0, 1
  - We return the bits as single-character Strings
- Implementation:
  - Store `n`
  - Each `next()` isolates the LSB and shifts
  - `hasNext()`?
  - `reset()`?