Bits

COMPSCI 230 — Discrete Math

January 26, 2017
Co-Primes

- Two positive integers $m$ and $n$ are co-primes iff $\text{GCD}(m, n) = 1$
- Not co-primes: $m = 24 = 2^3 \cdot 3$ and $n = 15 = 3 \cdot 5$
- Co-primes: $m = 24 = 2^3 \cdot 3$ and $n = 35 = 5 \cdot 7$
- 1 is co-prime with every positive integer: $\text{GCD}(1, n) = \text{GCD}(n, 1) = 1$
- ... including itself: $\text{GCD}(1, 1) = 1$
- If $m, n > 1$, then $m$ and $n$ are co-prime iff they share no prime factors (proper or otherwise)
Outline

1. Binary Representation of Integers

2. Binary Multiplication
Integers in Decimal and Binary

- Decimal (base 10): Ten digits: 0, ..., 9
  
  \[ 5307 = 5 \cdot 10^3 + 3 \cdot 10^2 + 0 \cdot 10^1 + 7 \cdot 10^0 \]

- Binary (base 2): Two digits: 0, 1
  
  \[ 1101 = 1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 \]

- Is 1101 decimal or binary?
- Say 1101\{2\} for binary, 1101\{10\} for decimal when ambiguous
Computers do Binary

- Integers are internally represented in binary
- Python prints in decimal for readability
- When Python prints 6, the computer sees 110
- \texttt{0bxxx} is a \textit{binary literal} (a number, not a string!)
- \texttt{0b110} is a different way to write 6
- They are represented the same way internally
- To convert from binary to integer just say
  >>> \texttt{0b1101}
  13
- To find out the binary equivalent of a decimal integer:
  >>> \texttt{bin(13)}
  '0b1101' \# A string, not a number!
- But how does Python convert?
Binary Representation of Integers

**Binary to Decimal**

- \(1101_2 = (1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0)_{10}\)

- We know how to do arithmetic:
  \[
  1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = 2^3 + 2^2 + 2^0 = 8 + 4 + 1 = 13
  \]

- Fruitful observation: splitting off the LSB
  \[
  1101_2 = 2 \cdot 110_2 + 1
  \]
- More generally:
  \[
  b_p \cdot 2^p + \ldots b_1 \cdot 2^1 + b_0 = 2(b_p \cdot 2^{p-1} + \ldots b_1) + b_0
  \]
  \[
  b_p \ldots b_1 b_0_2 = 2^{10} \cdot b_p \ldots b_1_2 + b_0
  \]

- The Least Significant Bit (LSB) \(b_0\) tells whether the number is even or odd (parity, or number \(\text{MOD } 2\))
Binary to Decimal in Python (str to num)

- To understand what Python does, imagine converting a string of bits (e.g., '1101') to a decimal.
- ['1101' is represented internally as ASCII codes 49, 49, 48, 49]
- The easy Python way:  
  \[
  \text{int('1101', 2)} \text{ yields 13}
  \]
- The instructive Python way:  
  Let \( b = 'b_p \ldots b_0' \) be the string of bits. Then  
  \[
  \text{b2d('b_p \ldots b_1 b_0')} = 2 \text{b2d('b_p \ldots b_1')} + b_0
  \]
  - ... and if we have a single bit \( (p = 0) \)  
    \[
    \text{b2d('b_0')} = b_0
    \]
  - So we have a recursion
Binary to Decimal in Python (str to num)

\[
b2d(\text{'}b_p \ldots b_1 b_0\text{'}) = \begin{cases} 
    b_0 & \text{if } p = 0 \\
    2 \cdot b2d(\text{'}b_p \ldots b_1\text{'}) + b_0 & \text{otherwise}
\end{cases}
\]

def b2d(b):
    if len(b) == 1: return 0 if b == '0' else 1
    else:
        last = len(b) - 1
        return 2 * b2d(b[:last]) + b2d(b[last])

[May want to add some input checking, especially against '' and non-binary inputs]
Binary Representation of Integers

Decimal to Binary (num to str)

- What is \( d = 5307 \{10\} \) in binary?
- Produce a string of bits
- The LSB tells whether the number is even or odd
- Conversely, the LSB is 0 if the number is even, 1 otherwise
- So we can find the LSB: \( b_0 = d \ MOD \ 2 \)
- What then?
  - In
    \[
    b_p \cdot 2^p + \ldots b_1 \cdot 2^1 + b_0 = 2(b_p \cdot 2^{p-1} + \ldots b_1) + b_0
    \]
    \[
    (b_p \cdot 2^{p-1} + \ldots b_1) \text{ is quotient, } b_0 \text{ is remainder:}
    \]
    \[
    b_p \cdot 2^{p-1} + \ldots b_1 = (b_p \cdot 2^p + \ldots b_1 \cdot 2^1 + b_0) \ \text{DIV} \ 2 = d \ \text{DIV} \ 2
    \]
  - We have exposed \( b_1 \), so we can repeat!
Binary Representation of Integers

Decimal to Binary in Python (num to str)

• Recipe:
  \[ \begin{align*}
  b_0 & \leftarrow d \mod 2 \\
  d & \leftarrow d \div 2 \\
\end{align*} \]
  and repeat

• Recursively,
  \[
  b_0 = d \mod 2 \\
  b = d2b(d) = \begin{cases} 
    'b_0' & \text{if } d \leq 1 \\
    d2b(d \div 2) \| 'b_0' & \text{otherwise}
  \end{cases}
\]
  where \(\|\) denotes string concatenation

• In Python,

```python
def d2b(d):
    lsb = d % 2
    if d <= 1: return str(lsb)
    else: return d2b(d // 2) + str(lsb)
```

[Again, add input-value checks]
Decimal to Binary in Python (num to list of nums)

\[
\text{bits}(26)
\]
\[
[1, 1, 0, 1, 0]
\]

We did one recursively, let’s try iterative

```python
def bits(d):
    assert(type(d) == int and d >= 0)
    if d == 0: return [0]
    b = []
    while d:
        b.insert(0, d % 2)
        d //= 2  # or d >>= 1
    return b
```
Bit Manipulation in Python

- Integers are represented internally as vectors of bits:
  35 is 0b100011

- Can manipulate these vectors bitwise (right-aligned)
  - Flip all bits: ~b
    ~0b100011 is 0b011100
  - Bitwise AND: a & b
    0b100011 & 0b110101 is 0b100001
  - Bitwise OR: a | b
    0b100011 | 0b110101 is 0b110111

- ... and shifted around:
  - Left shift: a << n (zeros come in from the right)
    0b100011 << 3 is 0b100011000
  - Right shift: a >> n (LSBs fall off the cliff)
    0b100011 >> 3 is 0b100
Weird Math?

```python
>>> 19 & 53
17
>>> d2b(19)
'10011'
>>> d2b(53)
'110101'
>>> 0b10011 & 0b110101
17
>>> d2b(17)
'10001'
```
Multiplication

Decimal:

\[
\begin{array}{c}
24 \cdot 35 \\
\hline
120 \\
72- \\
840
\end{array}
\]

- 24 times 5
- 24 times 3 shifted left by 1 \((i.e., \text{times 10})\)
- sum (with carry)

Binary:

\[
\begin{array}{c}
11000 \cdot 100011 \\
\hline
11000 \\
11000- \\
0-- \\
0--- \\
0---- \\
11000-----
\end{array}
\]

- 11000 times 1
- 11000 times 1 shifted left by 1 \((i.e., \text{times 2})\)
- 11000 times 0 shifted left by 2
- 11000 times 0 shifted left by 3
- 11000 times 0 shifted left by 4
- 11000 times 1 shifted left by 5
- sum (with carry)
Binary Multiplication Written in Decimal

Binary:

\[ 11000 \cdot 100011 \]

\[ \begin{array}{r}
11000 \\
11000 - \\
0-- \\
0--- \\
0----- \\
11000----- \\
\hline
1101001000
\end{array} \]

\[ 24 \cdot 35 \]

\[ \begin{array}{r}
24 \\
48 - \\
96 - \\
192 - \\
384 - \\
768 - \\
\hline
840
\end{array} \]

Looks familiar?

Russian Peasant Multiplication is binary multiplication disguised under decimal notation
Example: RPM with Bit Operations (Inputs are Integers)

def rpm(a, b):
    assert(type(a) == int)
    assert(type(b) == int and b >= 0)
    s = 0
    while b:
        if b & 1: s += a
        a <<= 1
        b >>= 1
    return s

>>> rpm(24, 35)
840