Representing Numbers

We are all familiar with representing numbers in **decimal** (base-10). We can also represent numbers in other bases. For example, it is very common in computer science to use **binary** (base-2). Using **octal** (base-8) and **hexadecimal** (base-16) is also common. For hexadecimal, we use digits 0–9 and letters A–F where $A = 10$ and $F = 16$.

In decimal, we can represent 10 different numbers using a single digit (0–9). If we add another digit, we can represent 100 different numbers (00–99). Adding a third digit gives us 1000 different numbers (000–999). With $n \geq 1$ digits, we can represent 10^n unique numbers.

Binary numbers work in exactly the same way, but with powers of 2 instead of powers of 10: each binary digit can only represent two values.

If we want to represent a number n in base b, we will need at least $\log_b(n)$[1] digits. Think about why this might be the case.

Counting.

Let's think about counting in decimal. We start with the lowest natural number (0), and we count by *incrementing* (adding 1). We always increment the least significant digit until it reaches its maximum value (9).

When digit exceeds its maximum value, we reset that digit to the lowest value, and *carry*—increment the next-least signifcant digit. This gives us: 00, 01, 02, 03, . . ., 09, 10, 11, . . . 19, 20, 21, . . ., 98, 99.

For any base b, the lowest value a digit can have is 0 and the maximum value of any digit is $b - 1$. Thus, with one binary digit, we can represent two numbers: 0 and 1. With two digits, we can represent four numbers: 00, 01, 10, 11. With three digits, eight numbers, and with $n \geq 1$ digits, 2^n numbers.

Counting in binary follows the same principles as counting in decimal. We always increment the least significant digit, and when any digit overflows, we reset that digit and carry (increment the next-least significant digit). If we want to count the first 8 numbers in binary (from decimal 0 to decimal 7), we would get: 000, 001, 010, 011, 100, 101, 110, 111.

Converting from Decimal to Binary

To convert a decimal number to a binary number, it's useful to think about polynomial expansions. For example, 23 is

$$(2 \times 10^1) + (3 \times 10^0).$$

Because $23 = 10111$ in binary, the binary polynomial expansion is

$$(1 \times 2^4) + (0 \times 2^3) + (1 \times 2^2) + (1 \times 2^1) + (1 \times 2^0).$$

- Notice that a number is odd if and only if the last binary bit is a 1 so checking for even/odd values tells us the last bit of the binary representation.
- Now imagine dividing the decimal number by 2 using integer division and think about how this affects the the polynomial expansion. It essentially chops off the last bit and shifts the binary number to the right by one place.
- If this new number is even or odd tells us what the next-to-last bit of the binary representation is. We can repeat this!

Practice

Question 1. *Given an integer d in base-10, write a function that represents d in binary as a list of 0s and 1s.*

[1] technically there are $\lfloor \log_b(n) \rfloor + 1$ digits

```
def num_to_binary(num):
    """
    return the binary representation of num as a list of bits (i.e., the integers 0 and 1)
    """
    if num == 0:
        return [0]

    bits = []
    while num > 0:
        if num % 2 == 0 :
            bits.append(0)
        else :
            bits.append(1)
        num = num // 2
    bits.reverse()
    return bits
```

How would we generalize this function to other bases?

```
1  def num_to_baseb(num, b) :
2      """
3      return the b-ary representation of num as a list of base-b integers
4      """
5      if num == 0 :
6          return [0]
7
8      digits = []
9      while num > 0:
10         digits.append(num % b)
11         num = num // b
12     digits.reverse()
13     return digits
```

These functions work well for most inputs but it prints the minimum number of digits needed to display the number. How would we modify the code to pad our strings to be a fixed width?

Question 2. *Can you think of any inputs that the code does not handle? What about small or large values of n?*

```
1  def num_to_padded_base(num, b, width) :
2      digits = []
3      while num > 0:
4          digits.append(num % b)
5          num = num // b
6
7      digits.extend([0]*(width - len(digits)))
8      digits.reverse()
9      return digits
```

Let's use this padded function to iterate through the first 8 binary numbers:

```
1  import sys
2  from math import log, ceil
```

```
3
4 n = int(sys.argv[1])
5 b = int(sys.argv[2])
6 width = ceil(log(n-1, b))
7 for i in range(n):
8     print(num_to_padded_base(i, b, width))
```

```
$ python3 printbinary.py 8 2
[0, 0, 0]
[0, 0, 1]
[0, 1, 0]
[0, 1, 1]
[1, 0, 0]
[1, 0, 1]
[1, 1, 0]
[1, 1, 1]
```