**γ code**

This week’s exercise is on γ-codes (book paragraph 5.3.2).

You’ll implement encoding and decoding γ-codes. Initially, the focus will be on encoding a single code. The second part requires you to write functions that decode a (non-delimited) sequence of codes.

We provide the following function, that returns the index of the most significant bit (MSB) of the binary representation of num (from the right):

```python
In [ ]:

```def` msb_index(num):
    if num == 0:
        return 0
    return int(math.log(num, 2))
```

print(msb_index(1))
print(msb_index(2))
print(msb_index(3))
print(msb_index(16))
print(msb_index(17))

Implement the function `gamma_encode` that encodes `number` as a γ code:

```python
In [ ]:

def gamma_encode(number):
    if number <= 0:
        raise ValueError("Cannot gamma-encode number < 0")
    # TODO: assignment
    return 0
```

Here are some tests.
In [ ]:
```python
# From book, table 5.5
# (number, gamma code, length)
values = [
    (1, 0b00, 1),
    (2, 0b100, 3),
    (3, 0b101, 3),
    (4, 0b11000, 5),
    (9, 0b1110001, 7),
    (13, 0b1110101, 7),
    (24, 0b111101000, 9),
    (511, 0b11111111111111111, 7),
    (1025, 0b111111111100000000001, 21)
]
for n, g, b in values:
    assert gamma_encode(n) == g
```

Now implement the function `gamma_decode` that decodes a γ-encoded number. The function should accept a sequence of γ-codes, decode the first one and return (decoded number, number of bits decoded). For example `gamma_decode(0b1110101)` should return `(13, 7)` because the number 13 is encoded with 7 bits in γ-code.

In [ ]:
```python
def gamma_decode(gcode):
    if gcode < 0:
        raise ValueError("gcode must be >= 0")
    # TODO: assignment
    return None
```

Tests:

In [ ]:
```python
for n, g, b in values:
    assert gamma_decode(g) == (n, b)
```

To make the last part of the exercise easier, `gamma_decode` should support decoding the prefix of a bit string, i.e. the most significant bits of a bit string, as a γ-code:

In [ ]:
```python
assert gamma_decode(0b100101110001111010000) == (2, 3)
```

In this test, the first γ-code in the bitstring should be decoded (2) and the length of its encoding returned (3 bits).

We can also verify that numbers correctly round-trip through a pair of encode and decode operations:

In [ ]:
```python
for x in range(1, 2049):
    assert gamma_decode(gamma_encode(x))[0] == x
```
Now that we have a function to decode a single $\gamma$-code, we can decode a bitstring containing a sequence of codes.

Implement the following function that accepts a bitstring with a sequence of codes and its length in bits; it should return a list of the decoded numbers.

```python
In [ ]:

def gamma_decode_stream(gcodes, slen):
    if number < 0:
        raise ValueError('Cannot decode "negative" bitstring')
    # TODO: assignment
    return None
```

Tests for decoding a bitstring:

```python
In [ ]:

assert(gamma_decode_stream(0b100101110001111010000, 21) == [ 2, 3, 4, 24 ])
assert(gamma_decode_stream(0b11111110111111111000101, 23) == [ 511, 1, 1, 1, 3 ])
assert(gamma_decode_stream(0b0000, 4) == [ 1, 1, 1, 1 ])
assert(gamma_decode_stream(0b0000101, 7) == [ 1, 1, 1, 1, 3 ])
assert(gamma_decode_stream(0b1111010000101100111001, 23) == [ 24, 1, 3, 2, 9 ])
```

```python
In [ ]:

# Construct Posting list from gamma coded gaps
# The first element of the gamma coded gaps list is document ID, rest are gaps
# construct the posting list with document IDs by adding subsequent gaps to the init

def construct_posting_list(gamma_decoded_gaps_list):
    # TODO: assignment
    return(posting_list)
```

```python
In [ ]:

assert(construct_posting_list(gamma_decode_stream(0b11111110111111111000101, 23)) ==
```