Binary Hyperbolic Pairing

Here we develop a *pairing function* [2] BPe that encodes a pair of positive ints (a, b) to a single positive int, and a corresponding decoding function BPd:

$$y = BPe(a, b)$$
$$(a, b) = BPd(y)$$

(I call this a "hyperbolic" pairing because pairs with similar products (E.g., (1, 100), (2, 50), (4, 25), (5, 20), (10, 10), (20, 5), (25, 4), (50, 2), (100, 1)) will encode to numbers of the same order of magnitude. On a graph, numbers with similar products will fall into a roughly hyperbolic ($y \approx c/x$) band, in contrast to the famous pairing by Cantor [2], where pairs with similar *sums* fall on 45° diagonals.)

Another way to say this is that the bits from the two numbers get concatenated "with some overhead for the comma." It seems a natural compactness goal.

The motivation to encode *positive* ints (in $\mathbb{N}_{\mathbb{1}}$) is just a technical convenience: they all start with a binary one. (If we need an encoding that accomodates zero, or for that matter negative numbers, we can build wrappers around this encoding.)

Each member of a pair will have its initial one-bit thrown away (to be resupplied on decoding). We'll call the bits-after-the-initial-one **mantissa** bits. The counts of mantissa bits put the positive ints into groups:

```
# of mantissa bits   Range of ints   As binary
      0                1               (1)
      1                2..3            (1)0..(1)1
      2                4..7            (1)00..(1)11
      3                8..15           (1)000..(1)111
     etc.
```

Pairs, in turn, are grouped by their total **pair mantissa** bit counts, and into subgroups according to how those bits are partitioned into a's *vs.* b's mantissas. For n bits there are $n + 1$ ways to partition them, *viz:*

```
# of mant. bits   Partition    Visual   a and b ranges
      0              0+0          |       (1, 1)
      1              0+1          |b      (1, 2..3)
                     1+0         a|       (2..3, 1)
      2              0+2          |bb     (1, 4..7)
                     1+1         a|b      (2..3, 2..3)
                     2+0        aa|       (4..7, 1)
      3              0+3          |bbb    (1, 8..15)
                     1+2         a|bb     (2..3, 4..7)
                     2+1        aa|b      (4..7, 2..3)
                     3+0       aaa|       (8..15, 1)
     etc.
```

The group that distributes n total mantissa bits needs $(n + 1)2^n$ **code points**--positions on the encoded number line; here are the first few groups...

```
 # of mant. bits   # of positions   encoded range
        0                1                1
        1                4              2..5
        2               12              6..17
        3               32             18..49
      etc.?
```

How can we quickly calculate where an arbitrary range starts? And, given an encoded pair, how can we know what range it's in? There's a closed-form answer to the first question, explained at [1] (https://artofproblemsolving.com/wiki/index.php?title=Arithmetico-geometric_series)...

$$(\text{Start of range } n) = (n - 1)2^n + 2$$

...and there's a reasonably fast way (11 μsec. on my computer) to reverse the mapping, explained below. Enough information to start programming.

---

[1] (https://artofproblemsolving.com/wiki/index.php?title=Arithmetico-geometric_series) ArtOfProblemSolving.com: Arithmetico-geometric series

[2] Wikipedia, "Pairing Function"

[3] https://arxiv.org/abs/1706.04129 (https://arxiv.org/abs/1706.04129) A relevant article about various pairing and tupling functions and some of their applications.

In [129]:

```python
from math import log2, ceil

def BPe(a, b):
    """
    Given int a >= 1 and int b >= 1,
    return the encoded pair int y >= 1.
    """
    a_sz,   b_sz   = a.bit_length() - 1, b.bit_length() - 1
    a_mant, b_mant = a - (1 << a_sz),    b - (1 << b_sz)
    pair_mant = (a_mant << b_sz) + b_mant
    pair_sz = a_sz + b_sz
    return pair_mant_start(pair_sz) + (a_sz << pair_sz) + pair_mant

def pair_mant_start(n):
    """
    Given n, the number of mantissa bits in a pair,
    return y_n, the start of the encoded range for such pairs.
    """
    # Have no fear, this works for n == 0.
    return ((n - 1) << n) + 2

def ceil01(x):
    # Ceil to a .01 for printing:
    return ceil(x * 100) * .01

def demo_BPe():
    for (a, b) in (1,1), (1,2), (1,3), (2,1), (3,1), (1, 4), \
                  (65537, 131071), (131071, 65537):
        print((a, b), end=" => ")
        y = BPe(a, b)
        print(y, f" {ceil01(log2(a * b)):.2f} {ceil01(log2(y)):.2f}")
        if a*b > 255:
            print(f"    {y:b}")

demo_BPe()
```

```
(1, 1) => 1  0.00 0.00
(1, 2) => 2  1.00 1.00
(1, 3) => 3  1.59 1.59
(2, 1) => 4  1.00 2.00
(3, 1) => 5  1.59 2.33
(1, 4) => 6  2.00 2.59
(65537, 131071) => 201863593985  33.01 37.56
    10111100000000000000100000000000000001
(131071, 65537) => 206158364675  33.01 37.59
    10111111111111111111110000000000000011
```

Decoding

The first code point for pairs with n total mantissa bits is

$$y_n = (n - 1)2^n + 2.$$

To decode a pair, the trickiest part is to take the encoded value $y : y_n \leq y < y_{n+1}$ and solve that inequality for n.

After going at the original equation with Newton's method (very slow convergence!) I changed variables in the equation.

$$
\begin{aligned}
y &= (n-1)2^n + 2 \\
y - 2 &= (n-1)2^n \\
\frac{y-2}{2} &= (n-1)2^{n-1} \\
\text{let } z &= \frac{y-2}{2} \\
\text{let } m &= n - 1 \\
z &= m2^m
\end{aligned}
$$

(I don't always use "z" and "m" in the code.)

(I should say that this looks like an job for the Lambert W function. The only reason I didn't go there is that getting Lambert W in Python is a little harder than trivial.)

One sequence that converges on the solution starts like this:

$$
\begin{aligned}
m_0 &= \lg(z) \\
m_1 &= m_0 - \lg(m_0)
\end{aligned}
$$

These two steps converge to just below the correct m, especially for large numbers (examples below), and the problem doesn't need more accuracy than m_1.

Some examples (from `quickie_demo_2()` , below):

```
z =  1 * 2**1  =            2  =>  m1 = 1.0
z =  2 * 2**2  =            8  =>  m1 = 1.415
z =  3 * 2**3  =           24  =>  m1 = 2.388
z =  4 * 2**4  =           64  =>  m1 = 3.415
z =  8 * 2**8  =         2048  =>  m1 = 7.541
z = 32 * 2**32 = 137438953472  =>  m1 = 31.79
```

The correct n is either the int part of m_1, or one more than that, so one application of the forward function, and a comparison, determines which pair-mantissa-size group an encoded pair is in.

Let us decode...

In [125]:

```python
from math import modf
from numbers import Integral

_MODF_LG_RES = 52
_MODF_LG_MUL = 2 ** -_MODF_LG_RES

def modf_lg(i):
    """
    Given an integer i >= 1,
    return approximately
            math.modf(math.log2(i)), i.e.
        the fractional and integer parts of the base-2 log:
            log2(i) % 1.0,  floor(log2(i)),
    except
        o  this works with any positive int i >= 1;
        o  the integer part is returned as an int, not float;
        o  the result is always exact for i a power of two;
        o  if 2**n < i < 2**(n + 1)
           and (f, j) = modf_lg(i), then
             o  j == n; and
             o  0 <= f < 1 and always has about
                51 bits of precision.
    """
    assert isinstance(i, Integral) and i >= 1, i

    j = i.bit_length() - 1
    if j < _MODF_LG_RES:
        i <<= (_MODF_LG_RES - j)
    else:
        i >>= (j - _MODF_LG_RES)
    f = log2(i * _MODF_LG_MUL)
    assert f < 1, f
    return f, j

ps = list(range(2, 4 + 1)) + [47, 48, 49, 51, 52, 53, 1021, 1022, 1023]
print("        modf(lg(i))                        modf_lg(i)")
for p in ps:
    i0 = 1<<p
    for i in range(i0 - 1, i0 + 1 + 1):
        f1, j1 = modf(lg(i))
        f2, j2 = modf_lg(i)
        print(f"({f1:.16f}, {j1:6.1f})  ({f2:.16f}, {j2:4d})")
    print()
```

```
        modf(lg(i))                        modf_lg(i)
(0.5849625007211561,    1.0)  (0.5849625007211562,    1)
(0.0000000000000000,    2.0)  (0.0000000000000000,    2)
(0.3219280948873622,    2.0)  (0.3219280948873623,    2)

(0.8073549220576042,    2.0)  (0.8073549220576041,    2)
(0.0000000000000000,    3.0)  (0.0000000000000000,    3)
(0.1699250014423122,    3.0)  (0.1699250014423124,    3)

(0.9068905956085187,    3.0)  (0.9068905956085185,    3)
(0.0000000000000000,    4.0)  (0.0000000000000000,    4)
```

```
(0.0874628412503391,    4.0)  (0.0874628412503394,    4)

(0.9999999999999929,   46.0)  (0.9999999999999898,   46)
(0.0000000000000000,   47.0)  (0.0000000000000000,   47)
(0.0000000000000071,   47.0)  (0.0000000000000103,   47)

(0.9999999999999929,   47.0)  (0.9999999999999949,   47)
(0.0000000000000000,   48.0)  (0.0000000000000000,   48)
(0.0000000000000071,   48.0)  (0.0000000000000051,   48)

(0.0000000000000000,   49.0)  (0.9999999999999974,   48)
(0.0000000000000000,   49.0)  (0.0000000000000000,   49)
(0.0000000000000000,   49.0)  (0.0000000000000026,   49)

(0.0000000000000000,   51.0)  (0.9999999999999993,   50)
(0.0000000000000000,   51.0)  (0.0000000000000000,   51)
(0.0000000000000000,   51.0)  (0.0000000000000006,   51)

(0.0000000000000000,   52.0)  (0.9999999999999997,   51)
(0.0000000000000000,   52.0)  (0.0000000000000000,   52)
(0.0000000000000000,   52.0)  (0.0000000000000003,   52)

(0.0000000000000000,   53.0)  (0.9999999999999999,   52)
(0.0000000000000000,   53.0)  (0.0000000000000000,   53)
(0.0000000000000000,   53.0)  (0.0000000000000000,   53)

(0.0000000000000000, 1021.0)  (0.9999999999999999, 1020)
(0.0000000000000000, 1021.0)  (0.0000000000000000, 1021)
(0.0000000000000000, 1021.0)  (0.0000000000000000, 1021)

(0.0000000000000000, 1022.0)  (0.9999999999999999, 1021)
(0.0000000000000000, 1022.0)  (0.0000000000000000, 1022)
(0.0000000000000000, 1022.0)  (0.0000000000000000, 1022)

(0.0000000000000000, 1023.0)  (0.9999999999999999, 1022)
(0.0000000000000000, 1023.0)  (0.0000000000000000, 1023)
(0.0000000000000000, 1023.0)  (0.0000000000000000, 1023)
```

In [114]:

```python
from math import floor

PDPMS_TITLE = "Where does (naive) decode_pair_mant_start(y) cross n?"
PDPMS_HEADER = "   n            y  pdpms(y)      y - y_n  fraction"

def pdpms(y):
    # Play Decode Pair Mant Start
    # This roughly corresponds to the part of
    #     decode_pair_mant_start_old(y)
    #     in the cell below,
    # before it converts to an int.
    if y == 1:  return 0, 1
    if y <= 5:  return 1, 2
    m0 = lg(y - 2) - 1
    return m0 - lg(m0) + 1

def pdpmsb(y):
    # Play Decode Pair Mant Start Big
    # This roughly corresponds to the new
    #     decode_pair_mant_start(y)
    # in the cell below.
    if y == 1:  return 0, 1
    if y <= 5:  return 1, 2
    if y < 2**48:  # Arbitrary, no decoding issue there.
        m0 = lg(y - 2) - 1
        return m0 - lg(m0) + 1

    else:
        f, j = modf_lg(y)   # Dropped the -2 on bignum.
        j -= 1
        ff, jj = modf_lg(j) # f not included in lg(j)
        # Deliberately shift the crossover point upwards.
        return j - jj + int(floor(f - ff - .5)) + 1

def demo_W():
    print(PDPMS_TITLE)
    print(PDPMS_HEADER)
    for n in range(6, 33 + 1):
        do_the_stuff(n)

def do_the_stuff(n, do_first=True, do_midpoint=True, delta=0, pdpms=pdp
    fmt = "%4s %12g %9.3f"
    fmt2 = "%12g  %6.4f"
    y_n = pair_mant_start(n)
    y_nn = pair_mant_start(n + 1)
    if do_first:
        print(fmt % (n, y_n, pdpms(y_n)))
    if do_midpoint:
        y_min = y_n + 1
        y_max = y_nn - 1
        target = n + delta
        while y_max - y_min > 0:
            y_mid = (y_max + y_min) // 2
            dpms_mid = pdpms(y_mid)
            if dpms_mid < target:
                y_min = y_mid + 1
            else:
```

```
                y_max = y_mid
        y_min_err = abs(play_dpms(y_min) - target)
        y_max_err = abs(play_dpms(y_max) - target)
        if y_min_err < y_max_err:
            y_mid = y_min
        else:
            y_mid = y_max
        print(fmt  % ("", y_mid, pdpms(y_mid)),
              fmt2 % (y_mid - y_n, (y_mid - y_n) / (y_nn - y_n)), pdpms


def demo_Wp(pdpms=pdpms):
    print(PDPMS_TITLE)
    print(PDPMS_HEADER)
    prev_n = None
    for two_p in range(1*2, 9*2 + 1 + 1):
        n = round(2**(two_p/2))
        do_first = (n != prev_n)
        do_the_stuff(n, do_first=do_first, pdpms=pdpms)
        do_the_stuff(n + 1, do_midpoint=False, pdpms=pdpms)
        prev_n = n + 1

print("y  pdpms(y)")
for y in range(6, 11 + 1):
    print(y, pdpms(y))
print()
demo_Wp(pdpms=pdpmsb)

```

```
y  pdpms(y)
6 2.0
7 1.9192843900317056
8 1.9205137932672667
9 1.9534750766119373
10 2.0
11 2.0522798214071534

Where does (naive) decode_pair_mant_start(y) cross n?
   n          y  pdpms(y)      y - y_n  fraction
   2          6     2.000
             10     2.000            4  0.3333 pdpmsb
   3         18     2.415
             34     3.000           16  0.5000 pdpmsb
   4         50     3.388
             90     4.011           40  0.5000 pdpmsb
   5        130     4.415
   6        322     5.450
            514     6.000          192  0.4286 pdpmsb
   7        770     6.483
   8       1794     7.513
           2659     8.000          865  0.3754 pdpmsb
   9       4098     8.541
  11      20482    10.586
          28234    11.000         7752  0.3154 pdpmsb
  12      45058    11.605
  16     983042    15.666
```

```
      1.26276e+06    16.000       279718  0.2511 pdpmsb
  17  2.09715e+06    16.678
  23  1.84549e+08    22.734
       2.2432e+08    23.000  3.97702e+07  0.1975 pdpmsb
  24  3.85876e+08    23.741
  32  1.33144e+11    31.786
      1.55379e+11    32.000   2.2235e+10  0.1569 pdpmsb
  33  2.74878e+11    32.791
  45  1.54811e+15    44.000
      2.48791e+15    45.000  9.39798e+14  0.5807 pdpmsb
  46  3.16659e+15    45.000
  64  1.16214e+21    63.000
      1.80005e+21    64.000  6.37902e+20  0.5320 pdpmsb
  65  2.36118e+21    64.000
  91  2.22829e+29    90.000
      3.39638e+29    91.000  1.16809e+29  0.5128 pdpmsb
  92   4.5061e+29    91.000
 128  4.32159e+40   127.000
      6.44851e+40   128.000  2.12692e+40  0.4845 pdpmsb
 129  8.71123e+40   128.000
 181  5.51698e+56   180.000
      8.14896e+56   181.000  2.63197e+56  0.4718 pdpmsb
 182  1.10953e+57   181.000
 256   2.9527e+79   255.000
      4.30675e+79   256.000  1.35405e+79  0.4550 pdpmsb
 257  5.92855e+79   256.000
 362  3.3913e+111   361.000
     4.91558e+111   362.000 1.52429e+111  0.4470 pdpmsb
 363 6.80138e+111   362.000
 512 6.85139e+156   511.000
     9.85998e+156   512.000 3.00859e+156  0.4374 pdpmsb
 513 1.37296e+157   512.000
 724 6.38051e+220   723.000
      9.1482e+220   724.000  2.7677e+220  0.4326 pdpmsb
 725 1.27787e+221   724.000
```

Two versions of the decoder

Each version also contains it's own "old" version.

I believe the lower notebook cell contains the code used in `bin_pair.py` , but only for the superficial reason that the tests are run after that one.

Version one:

In [ ]:

```
def decode_pair_mant_start_old(y):
    """
    Given y, where
        y_n <= y < y_{n + 1},
        y_n = (n - 1) * 2**n + 2
    return
        n -- the number of pair mantissa bits in this group, and
        y_n (a.k.a. pair_mant_start(n))
            -- the first code point of the group.
    This will fail at y = 2**(2**53) at the latest.
    Does an extra operation on a potential bignum,
       so takes some extra time.
    I have no proof this works at all, really.
    """
    if y == 1: return 0, 1
    if y <= 5: return 1, 2

    # z = (y - 2) / 2  # float  z = (n-1) * 2**(n-1)
    # let m = n - 1    #        z =     m * 2**m
    m0 = lg(y - 2) - 1 # float -- subtracts 2 from a bignum!
    m1 = m0 - lg(m0)   # float
    nmax = int(m1) + 2 # int
    y_nmax = pair_mant_start(nmax)
    if y >= y_nmax:
        return nmax, y_nmax
    else:
        n = nmax - 1
        return n, pair_mant_start(n)

def decode_pair_mant_start(y):
    """
    Given y, where
        y_n <= y < y_{n + 1},
        y_n = (n - 1) * 2**n + 2
    return
        n -- the number of pair mantissa bits in this group, and
        y_n (a.k.a. pair_mant_start(n))
            -- the first code point of the group.
    I think this works for any y that Python and the machine can
    handle.  It's specifically fixed to handle y > 2**(2**52).
    I still have no proof.
    """
    if y == 1:  return 0, 1
    if y <= 5:  return 1, 2
    if y < 2**47:  # Arbitrary, semi-superstitious place to
        #             switch strategy.
        m0 = lg(y - 2) - 1
        nmax = floor(m0 - lg(m0)) + 2
    else:
        f, j = modf_lg(y)   # Don't subtract 2 from bignum.
        j -= 1
        ff, jj = modf_lg(j) # f not included in lg(j).
        # Deliberately shift the crossover point upwards.
        nmax = j - jj + floor(f - ff - .5) + 2
    y_nmax = pair_mant_start(nmax)
    if y >= y_nmax:
        return nmax, y_nmax
```

```
    else:
        n = nmax - 1
        return n, pair_mant_start(n)

def test_decode_pair_mant_start(DPMS=decode_pair_mant_start,
                                verbose=False):
    for pair_sz in range(0, 129):
        start = pair_mant_start(pair_sz)
        ymin = ymax = start
        if start > 1:  ymax = start + 1
        if start > 2:  ymin = start - 1
        for y in range(ymin, ymax + 1):
            if y < start:
                if verbose:
                    print("  ", end="  ")
                pair_sz_ok = pair_sz - 1
                start_ok = pair_mant_start(pair_sz - 1)
            else:
                pair_sz_ok = pair_sz
                start_ok = start
                if y == start:
                    if verbose:
                        print(pair_sz, end="  ")
                else:
                    if verbose:
                        print("  ", end="  ")
            pair_sz_d, start_d = DPMS(y)
            if verbose:
                print(y, pair_sz_d, start_d)
            assert pair_sz_d == pair_sz_ok, \
                (y, pair_sz_d, pair_sz_ok)
            assert start_d == start_ok, (y, start_d, start_ok)
    print(DPMS.__name__, "passed.")

def test_decode_pair_mant_start_2(n_max=10000):
    prev_y_n = 1
    for n in range(1, n_max + 1):
        y_n = pair_mant_start(n)
        for y in range(y_n - 1, y_n + 1 + 1):
            if y < y_n:
                correct = n - 1, prev_y_n
            else:
                correct = n, y_n
            ans = decode_pair_mant_start(y)
            if (dn, dyn) != correct:
                print("decode_pair_mant_start(", y, ") =", ans, "inste
                return False

```

Second version of the decoder:

In [134]:

```
def decode_pair_mant_start_old(y):
    """
    Given y, where
        y_n <= y < y_{n + 1},
        y_n = (n - 1) * 2**n + 2
    return
        n -- the number of pair mantissa bits in this group, and
        y_n (a.k.a. pair_mant_start(n))
            -- the first code point of the group.
    This will fail at y = 2**(2**53) at the latest.
    Does an extra operation on a potential bignum,
       so takes some extra time.
    I have no proof this works at all, really.
    """
    if y == 1: return 0, 1
    if y <= 5: return 1, 2

    # z = (y - 2) / 2  # float  z = (n-1) * 2**(n-1)
    # let m = n - 1    #        z =     m * 2**m
    m0 = lg(y - 2) - 1 # float -- subtracts 2 from a bignum!
    m1 = m0 - lg(m0)   # float
    nmax = int(m1) + 2 # int
    y_nmax = pair_mant_start(nmax)
    if y >= y_nmax:
        return nmax, y_nmax
    else:
        n = nmax - 1
        return n, pair_mant_start(n)

def decode_pair_mant_start(y):
    """
    Given y, where
        y_n <= y < y_{n + 1},
        y_n = (n - 1) * 2**n + 2
    return
        n -- the number of pair mantissa bits in this group, and
        y_n (a.k.a. pair_mant_start(n))
            -- the first code point of the group.
    I think this works for any y that Python and the machine can
    handle.  It's specifically fixed to handle y > 2**(2**52).
    I still have no proof.
    """
    if y == 1:  return 0, 1
    if y <= 5:  return 1, 2
    if y < 2**47:  # Arbitrary, semi-superstitious place to
        #             switch strategy.
        m0 = lg(y - 2) - 1
        nmax = floor(m0 - lg(m0)) + 2
    else:
        f, j = modf_lg(y)   # Don't subtract 2 from bignum.
        j -= 1
        ff, jj = modf_lg(j) # f not included in lg(j).
        # Deliberately shift the crossover point upwards.
        nmax = j - jj + floor(f - ff - .5) + 2
    y_nmax = pair_mant_start(nmax)
    if y >= y_nmax:
        return nmax, y_nmax
```

```
    else:
        n = nmax - 1
        return n, pair_mant_start(n)

def test_decode_pair_mant_start_old(DPMS=decode_pair_mant_start,
                                    verbose=False):
    for pair_sz in range(0, 129):
        start = pair_mant_start(pair_sz)
        ymin = ymax = start
        if start > 1:  ymax = start + 1
        if start > 2:  ymin = start - 1
        for y in range(ymin, ymax + 1):
            if y < start:
                if verbose:
                    print("  ", end="  ")
                pair_sz_ok = pair_sz - 1
                start_ok = pair_mant_start(pair_sz - 1)
            else:
                pair_sz_ok = pair_sz
                start_ok = start
                if y == start:
                    if verbose:
                        print(pair_sz, end="  ")
                else:
                    if verbose:
                        print("  ", end="  ")
            pair_sz_d, start_d = DPMS(y)
            if verbose:
                print(y, pair_sz_d, start_d)
            assert pair_sz_d == pair_sz_ok, \
                (y, pair_sz_d, pair_sz_ok)
            assert start_d == start_ok, (y, start_d, start_ok)
    print(DPMS.__name__, "passed.")

from time import process_time as ptime

def test_decode_pair_mant_start(n_max=10000):
    start = ptime()
    prev_y_n = 1
    for n in range(1, n_max + 1):
        y_n = pair_mant_start(n)
        for y in range(y_n - 1, y_n + 1 + 1):
            if y < y_n:
                correct = n - 1, prev_y_n
            else:
                correct = n, y_n
            ans = decode_pair_mant_start(y)
            if ans != correct:
                print("decode_pair_mant_start(", y, ") =", ans, "inste
                return False

        prev_y_n = y_n
    dur = ptime() - start
    print("decode_pair_mant_start() correct for 1 <= n <=", n_max, "in
    return True

def BPd(y):
```

```
    """
    Given y-- an encoded pair,
    return (a, b) -- the pair.
    """
    pair_sz, start = decode_pair_mant_start(y)
    offset = y - start
    # Is there a bit-shift equivalent of divmod?

    a_sz = offset >> pair_sz
    pair_mant = offset - (a_sz << pair_sz)
    # a_sz, pair_mant = divmod(offset, 1 << pair_sz)

    b_sz = pair_sz - a_sz
    top_a = 1 << a_sz
    top_b = 1 << b_sz

    a_mant = pair_mant >> b_sz
    b_mant = pair_mant - (a_mant << b_sz)
    # a_mant, b_mant = divmod(pair_mant, top_b)

    return (top_a + a_mant, top_b + b_mant)

from time import process_time as ptime
from math import sqrt

def test_BPd(max_y=10**5, verbose=False):
    if verbose:
        for y in range(1, 20):
            print(y, "=>", BPd(y))
    start = ptime()
    # "Decode" the first max_y code points to pairs,
    # encode back to code points, and compare.
    for y in range(1, max_y + 1):
        a, b = BPd(y)
        ye = BPe(a, b)
        assert ye == y, (y, a, b, ye)
    dur = ptime() - start
    print("BPd passed. ", f"{max_y:.2e}",
          "(dec->enc)'s averaging", f"{dur/max_y:.2e}", "sec.")
    if verbose:
        for p in range(2, 61, 5):
            for num in 3, 4:
                s = (2. ** (.5 * p) * num) // 4
                sse = BPe(s, s)
                print((s, s), "=>", sse,
                      f"  {lg(sse) - 2 * lg(s):.1f}")

def test_BP_sz(max_y=10**6):
    """
    See how far lg(BPe(a, b)) varies above and below
        lg(a*b) + lg(lg(a*b)).
    """
    print("test_BP_sz(", max_y, ")...")
    min_diff = (0, 1, 1) # lg(lg(1 * 1)) isn't actually defined.
    max_diff = (0, 1, 1)
    sum_diff = 0
    for y in range(2, max_y + 1):
```

```
        a, b = BPd(y)
        lgab = lg(a * b)
        diff = lg(y) - (lgab +lg(lgab))
        min_diff = min(min_diff, (diff, y))
        max_diff = max(max_diff, (diff, y))
        sum_diff += diff
    d, y = min_diff
    a, b = BPd(y)
    print(f"min diff = {d:.4f} at {y:d} = BPe({a:d}, {b:d})")
    avg_diff = sum_diff / (max_y + 1) # float
    print(f"avg diff = {avg_diff:.4f}")
    d, y = max_diff
    a, b = BPd(y)
    print(f"max diff = {d:.4f} at {y:d} = BPe({a:d}, {b:d})")

test_decode_pair_mant_start()
print()

test_BP_sz()
print()

test_BPd()
```

```
decode_pair_mant_start() correct for 1 <= n <= 10000 in 0.8027749999999996
9 sec.

test_BP_sz( 1000000 )...
min diff = -1.8163 at 589825 = BPe(15, 8191)
avg diff = -0.5456
max diff = 1.0000 at 4 = BPe(2, 1)

BPd passed.  1.00e+05 (dec->enc)'s averaging 1.11e-05 sec.
```

In [314]:

```python
for y in range(1, 33):
    print(y, BPd(y))
```

```
1 (1, 1)
2 (1, 2)
3 (1, 3)
4 (2, 1)
5 (3, 1)
6 (1, 4)
7 (1, 5)
8 (1, 6)
9 (1, 7)
10 (2, 2)
11 (2, 3)
12 (3, 2)
13 (3, 3)
14 (4, 1)
15 (5, 1)
16 (6, 1)
17 (7, 1)
18 (1, 8)
19 (1, 9)
20 (1, 10)
21 (1, 11)
22 (1, 12)
23 (1, 13)
24 (1, 14)
25 (1, 15)
26 (2, 4)
27 (2, 5)
28 (2, 6)
29 (2, 7)
30 (3, 4)
31 (3, 5)
32 (3, 6)
```

Combine many numbers into one.

I want to use pairs of pairs, etc., to pack a list of numbers into a single number. Here's an example of the principle:

In [199]:

```python
from math import log2

# Here L is the Python list and nabcdefgh is the encoded list.

L = [2, 3, 5, 7, 11, 13, 17, 19]
print(L)
# Delta encode.
for i in range(len(L) - 1, 0, -1):
    L[i] -= L[i - 1]
# Combine 2**3 ints into pairs three times.
for p in range(3):
    L = [BPe(L[i], L[i + 1]) for i in range(0, len(L), 2)]
nabcdefgh = BPe(8 + 1, L[0])
print(nabcdefgh, log2(nabcdefgh), "bits after delta coding and BPe().")

del L # Nothing up my sleeve...

n, abcdefgh = BPd(nabcdefgh)
assert n == 8 + 1
L = [abcdefgh]
# Split ints into pairs three times => 2**3 ints.
for p in range(3):
    L = list(sum((BPd(x) for x in L), tuple()))
# Delta decode.
for i in range(1, len(L)):
    L[i] += L[i - 1]
print(L, "log2(product) =", sum(log2(x) for x in L), "bits.")

```

```
[2, 3, 5, 7, 11, 13, 17, 19]
3920140360 31.868258164717233 bits after delta coding and BPe().
[2, 3, 5, 7, 11, 13, 17, 19] log2(product) = 23.209507209138437 bits.
```

Defining the list encoding and `BP_list` class.

Let's formalize this list-of-ints encoding. Here I'll say "pair" when I mean BP-encoded pair.

A **BP_list** is a pair (length + 1, root). **No longer true**. the first number of that pair is a more complicated thing now. See "THE RULE TO HANDLE EMPTY LISTS" in the `__init__` method of the `BP_list` class code in a cell below.

The **root** is a node, or just one if the list is empty, i.e., if length + 1 == 1. Thus the empty list is `BPe(1, 1) = 1`.

A **node** is interpreted as either an element of the list or a pair of nodes, depending on its position relative to the root, and the length. If the number of elements under a node is $n > 1$, then

$$\text{number of elements on the left } = 2^{\lceil \lg(n)-1 \rceil}$$
$$\text{number on the right} = n - 2^{\lceil \lg(n)-1 \rceil}$$

A node with one element is just the element itself.

Use

To create a list, `L = BP_list()` , and to add an element, `L.append(x)` .

To decode a list from an int, `L = BP_list(i)` . The default is 1, the code for the empty list.

To encode the list as an int, `i = L.as_int()` .

To extract the data it's *recommended* to use `L.pop(0)` , because that way the memory is deallocated as it's used. The methods `__iter__` and `__reversed__` are supposed to give forward and reverse iterators, but I would like deallocate-as-you-go versions, hmm.

The opposite of `pop` is `insert(position, value)` .

Internal representation

Inside the `BP_list` class, the working representation of the list is a Python list of Python two-tuples, where each tuple is

```
(number of elements, BP-encoded node).
```

For an empty list, this is an empty Python list. Initialized from an encoded int, it starts out

```
[ (# of elements, root node) ]
```

From there, nodes can be split and combined, and single-element nodes can be added or removed on either end. Somewhere in the list is the node with the most elements, and the sizes decrease toward either end. A list constructed by appending to the right side has the longest-length node on the left, and only that arrangement can be finally encoded as an int without an extra-large amount of work.

In case I am taken away from this project for a while

I would like to allow pushing and popping on both ends of the list, but that would allow states that could not easily be encoded into the current BPe representation. I believe this representation is more flexible:

```
BPe(BPe(left # of elements + 1, right # of elements + 1), root)
```

This can be broken down into two back-to-back descending lists of nodes each of which has a power-of-two number of entries, and entries could then be pushed and popped on both ends while maintaining that condition, and finally encoded into the thing above. I think. The empty list would be BPe(BPe(1, 1), 1) == 1.

In [313]:

```
from numbers import Integral as _Int

class BP_list(object):
    """
    A list-like object that encodes a sequence of zero
    or more ints >= 1 as a single int >= 1, and/or decodes
    an int as a sequence of ints, using BPe() and BPd().
        >>> L = BP_list( [123, 456, 1492] )
        >>> y = L.as_int()  # encode
        >>> M = BP_list(y)  # decode
        >>> M.append(1776)
        >>> M.pop(0)
        123
        >>> M.pop(0)
        456
        >>> list(x for x in M.reversed_pop())
        [1776, 1492]
        >>> len(M)
        0
    BP_list doesn't support random access,
    only append at the end, and pop or iterated pop
    at either end (positions 0 and -1).
    """
    # There are many sequences like this here:
    #     a, b = BPd(y);  del y
    # or
    #     y = BPe(a, b);  del a, b
    # or
    #     chunks.append( (length, node) );  del node
    # or
    #     (length, node) = chunks.pop(0) # (pop deletes.)
    # The general idea is to let go of extra links
    # to possibly huge items in memory as soon as possible.
    # "del var" is done even if we're about to leave the
    # scope of var, as a way to remember that that var
    # needs to be deleted right away even if the source
    # code is edited.

    def __init__(self, y=1):
        self.chunks = []
        self.length = 0
        if y == 1:
            return

        try:
            self.extend(y)  # Maybe it's an iterable...
            return

        except TypeError:
            pass            # Never mind.

        assert isinstance(y, _Int) and y >= 1, "init with an int >= 1

        lengthoid, root = BPd(y);    del y
        # THE RULE TO HANDLE EMPTY LISTS:
        if root == 1:
            length = lengthoid - 1  # So (1,         1) = empty list.
```

```python
                                    #    (n > 1,     1) = list of n -
        else:
            length = lengthoid      #    (1,     r > 1) = list of just
                                    #    (n > 1, r > 1) = list of n no

        chunks = [ (length, root) ];  del root
        # Break down into power-of-two chunks.
        while True:
            last_len, last_node = chunks[-1]
            pow2 = 1 << (last_len.bit_length() - 1)
            if last_len > pow2:
                left_node, right_node = BPd(last_node);  del last_node
                chunks[-1] = (pow2, left_node)
                chunks.append( (last_len - pow2, right_node) )
                del left_node, right_node
            else:
                del last_node
                break

        self.chunks = chunks; del chunks
        self.length = length

    def as_int(self):
        """
        If possible, encode self as an int and empty the list.
        WARNING: If elements have been popped from the front
        of the list, it becomes impossible to encode the internal
        data structure as an int in a reasonable way.
        """
        # Check that the chunks can be encoded.
        prev_len = None
        total = 0
        for chunk in self.chunks:
            chunk_len = chunk[0];  del chunk
            pow2 = 1 << (chunk_len.bit_length() - 1)
            assert chunk_len == pow2, "Broken chunks structure"
            assert not prev_len or prev_len > chunk_len, \
                "as_int() after pop(0)"

            total += chunk_len
            prev_len = chunk_len
        # Combine chunks right-to-left.
        assert self.length == total, "total length mismatch"

        if self.length == 0:
            node = 1
        else:
            node = self.chunks.pop(-1)[1]
            while self.chunks:
                node = BPe(self.chunks.pop(-1)[1], node)

        # See "THE RULE TO HANDLE EMPTY LISTS" in __init__(), above.
        if node == 1:
            return BPe(self.length + 1, node)
        else:
            return BPe(self.length, node)

```

```
    def __len__(self):
        return self.length

    def append(self, item):
        chunks = self.chunks
        chunks.append( (1, item) )
        self.length += 1
        # Merge same-size chunks on the right end.
        while len(chunks) >= 2 and chunks[-2][0] == chunks[-1][0]:
            right_len, right_node = chunks.pop(-1)
            left_len, left_node = chunks.pop(-1)
            chunks.append( (left_len + right_len, BPe(left_node, right_
            del left_node, right_node

    def __iadd__(self, L):
        """
        self += L => self.extend(L)
        """
        self.extend(L)
        # += is a command, not a function, but it must return
        # a value for the variable.
        return self

    def extend(self, L):
        """
        Append the elements of L to self one by one.
        """
        for x in L:
            self.append(x)

    def pop(self, idx):
        """
        Remove and return the first (if idx==0) or last (if idx=-1)
        element of the list.  No other positions can be popped.

        WARNING: once any elements are popped from the beginning of
        the list, the remainder of the list can't be encoded with
        self.as_int().
        """
        if self.length == 0:  raise IndexError("pop from empty list")

        end_len, end_node = self.chunks.pop(idx)
        if idx == 0:
            # Prepare to break down left-most chunks.
            outer = 0   # The left member of first pair is on left end
            inner = 1   # The right member of first pair is further in
            ins_pt = 0  # Insert leftovers before the beginning.
        elif idx == -1:
            # Prepare to break down right-most chunks.
            outer = 1             # Right member of last pair is right
            inner = 0             # Left member of last pair is furthe
            ins_pt = len(self.chunks)  # Insert "before past the end."
        else:
            raise IndexError("pop index must be 0 or -1")

        pow2 = 1 << (end_len.bit_length() - 1)
        assert end_len == pow2, ("chunk len isn't a power of two", end_
```

```

        while end_len > 1:
            # Split the (end_len, end_node) chunk.
            nodes = BPd(end_node);  del end_node
            end_node, inner_node = nodes[outer], nodes[inner];  del no
            end_len >>= 1
            # Return the unused portion.
            self.chunks.insert(ins_pt, (end_len, inner_node)); del inn
            if idx == -1:  ins_pt += 1
        self.length -= 1
        return end_node


    def iter_pop(self):
        while self:
            yield self.pop(0)

    def reversed_pop(self):
        while self:
            yield self.pop(-1)

def BP_list_demo():
    L = BP_list()
    L += ( [  2,  3,  5,  7, 11, 13, 17,
               19, 23, 29, 31, 37, 41, 43,
               47, 53, 59, 61, 67,
             ] )  # 19 = 0b10011
    y = L.as_int()
    print("y =", y)
    for x in BP_list(y).iter_pop():
        print(x, end=" ")
    print()
    for x in BP_list(y).reversed_pop():
        print(x, end=" ")
    del y
    print()

def BP_list_as_int_demo():
    L = BP_list()
    L.append(123)
    L.append(456)
    L.append(1492)
    y = L.as_int() # encode
    print("y =", y)
    del L
    M = BP_list(y); del y # decode
    M.append(1776)
    print(M.pop(0))
    print(M.pop(0))
    print(list(x for x in M.reversed_pop()))
    del M

def decode_some_lists(n=20):
    for y in range(1, n + 1):
        length, root = BPd(y)
        if length == 1 and root > 1:
            # print(f"{y:}: not a valid list")
```

```python
            pass
        else:
            print(f"{y:}: {list(BP_list(y).iter_pop()):}")

from time import process_time as _ptime

def test_BP_list(n=2000):
    L = BP_list( [1, 2, 3] )
    L.pop(0)
    try:
        y = L.as_int()
        print("Didn't get AssertionError after pop(0) then as_int().")
        # NB: pop(0) on *some* BP_lists would leave them okay:
        # 2 or 1 elements => 1 or 0 elements => as_int() works.
        return False
    except AssertionError as e:
        pass  # Assertion error is what we were looking for.

    start = _ptime()
    for y in range(1, n + 1):
        L = list(BP_list(y).iter_pop())  # Decode.
        if BP_list(L).as_int() != y:     # Encode and compare.
            print("test_BP_list() failed at y =", y)
            return False

    dur = _ptime() - start
    print(f"BP_list passed, {n:g} decode=>encodes in {dur:.2f} sec.")
    return True


BP_list_demo()
print("-----")
print()
BP_list_as_int_demo()
print("-----")
print()
test_BP_list()
# L = list(BP_list(1).iter_pop())
# print(L)

```

```
y = 26804811408642678179802297158505326654846328777
2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67
67 61 59 53 47 43 41 37 31 29 23 19 17 13 11 7 5 3 2
-----

y = 596261153240
123
456
[1776, 1492]
-----

BP_list passed, 2000 decode=>encodes in 1.04 sec.
```

Out[313]: True

In [299]:

```python
for y in range(1, 100):
    print(y, ":", list(BP_list(y).iter_pop()))
```

```
1 : []
2 : [2]
3 : [3]
4 : [1]
5 : [1, 1]
6 : [4]
7 : [5]
8 : [6]
9 : [7]
10 : [1, 2]
11 : [1, 3]
12 : [1, 1, 2]
13 : [1, 1, 3]
14 : [1, 1, 1]
15 : [1, 1, 1, 1]
16 : [1, 1, 1, 1, 1]
17 : [1, 1, 1, 1, 1, 1]
18 : [8]
19 : [9]
20 : [10]
21 : [11]
22 : [12]
23 : [13]
24 : [14]
25 : [15]
26 : [2, 1]
27 : [3, 1]
28 : [1, 4]
29 : [1, 5]
30 : [1, 2, 1]
31 : [1, 3, 1]
32 : [1, 1, 4]
33 : [1, 1, 5]
34 : [1, 1, 1, 2]
35 : [1, 1, 1, 3]
36 : [1, 1, 1, 1, 2]
37 : [1, 1, 1, 1, 3]
38 : [1, 1, 1, 1, 1, 2]
39 : [1, 1, 1, 1, 1, 3]
40 : [1, 1, 1, 1, 1, 1, 2]
41 : [1, 1, 1, 1, 1, 1, 3]
42 : [1, 1, 1, 1, 1, 1, 1]
43 : [1, 1, 1, 1, 1, 1, 1, 1]
44 : [1, 1, 1, 1, 1, 1, 1, 1, 1]
45 : [1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
46 : [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
47 : [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
48 : [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
49 : [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
50 : [16]
51 : [17]
52 : [18]
53 : [19]
```

```
54 : [20]
55 : [21]
56 : [22]
57 : [23]
58 : [24]
59 : [25]
60 : [26]
61 : [27]
62 : [28]
63 : [29]
64 : [30]
65 : [31]
66 : [1, 6]
67 : [1, 7]
68 : [2, 2]
69 : [2, 3]
70 : [3, 2]
71 : [3, 3]
72 : [4, 1]
73 : [5, 1]
74 : [1, 1, 6]
75 : [1, 1, 7]
76 : [1, 2, 2]
77 : [1, 2, 3]
78 : [1, 3, 2]
79 : [1, 3, 3]
80 : [2, 1, 1]
81 : [3, 1, 1]
82 : [1, 2, 1, 1]
83 : [1, 3, 1, 1]
84 : [1, 1, 2, 1]
85 : [1, 1, 3, 1]
86 : [1, 1, 1, 2, 1]
87 : [1, 1, 1, 3, 1]
88 : [1, 1, 1, 1, 4]
89 : [1, 1, 1, 1, 5]
90 : [1, 1, 1, 2, 1, 1]
91 : [1, 1, 1, 3, 1, 1]
92 : [1, 1, 1, 1, 2, 1]
93 : [1, 1, 1, 1, 3, 1]
94 : [1, 1, 1, 2, 1, 1, 1]
95 : [1, 1, 1, 3, 1, 1, 1]
96 : [1, 1, 1, 1, 1, 2, 1]
97 : [1, 1, 1, 1, 1, 3, 1]
98 : [1, 1, 1, 1, 1, 1, 1, 2]
99 : [1, 1, 1, 1, 1, 1, 1, 3]
```

In [220]:

```python
from math import log, exp
def geomean(L):
    return sum(log2(x) for x in L) / len(L)
print(BPe(1,1), BPe(1,2), BPe(2,1), BPe(2,2), "bits", geomean([1,2,4,10
# Pairs w/ smallest products: (1,1), (1,2), (2,1), (1,3), (3,1), (1,4),
# In that ordering, the pairs above are 1, 2, 3, 7.
# There's even an argument for ...(3,1), (2,2), (1,4), (4,1) in which c
print(1,2,3,6, "bits", geomean([1,2,3,6]))
```

```
1 2 4 10 bits 1.5804820237218404
1 2 3 6 bits 1.292481250360578
```

Fiddling with data-collector schemes

(From before I decided to mostly-emulate a list object.)

I was going to use a Python old-style generator (as opposed to a new-style "coroutine") with the `send()` feature, but their use- pattern is nasty and this thing doesn't really have a complicated internal control flow, just a state.

A simpler pattern is to create a closure (instance of a function) with state that hangs around. Here's an example of that pattern:

In [157]:

```python
def Collector():
    """
    Collector produces a function that starts out with
    and empty list inside.  You call the function to append
    to the list, and if you call the function with the
    argument "None", it returns *a copy of* the present
    state of the list.  (There is no way to append None
    to the list.)

    Used like this:

        >>> c = Collector()
        >>> c("Polly")
        >>> c("Molly")
        >>> c("Sue")
        >>> c("Brady")
        >>> print(c(None))
        ["Polly", "Molly", "Sue", "Brady"]
    """
    collection = []

    def c(item):
        nonlocal collection # New feature in Python 3.

        if item != None:
            collection.append(item)
        else:
            return list(collection)

    return c

c = Collector()
c("haploid")
c("mongoloid")
c("diploid")
some = c(None)
c("and more")
more = c(None)

print(some)
print(more)
```

```
['haploid', 'mongoloid', 'diploid']
['haploid', 'mongoloid', 'diploid', 'and more']
```

Stuff about using generators as data collectors.

Before I decided on a closure with attached state, I was considering using a generator to construct pair-based lists.

Generators have a couple tricks that turn them into coroutines with input and output. (N.b. Python now uses the word "coroutine" for a different construct.)

1. `yield(output)` acts like a function that returns a result.

2. Inside the generator, `yield()` returns what the caller sends with `g.send(value)` (instead of having called `next(g)`).
3. The result from calling `g.send()` is the output of the *next* `yield`.
4. You have to send `None` to the generator at the beginning to let it find its way to the first `yield` statement. The `None` doesn't go anywhere but it has to be a `None`. The first value `yield` ed is returned from that first `send()`.
5. Just as with regular generators, returning from the generator raises "StopIteration".

Patterns of `send()` and `yield()`

In [168]:

```python
# Just showing what immediately comes out of yields,
# not building a collection.
# In this example, the generator finally returns,
# which raises a StopIteration exception.

def guffaw():
    output = "first output"
    while True:
        input = yield(output)
        if input == None:
            break

        output = input

g = guffaw()
print("Initial send(None) returns", repr(g.send(None))) # to get starte

print("send(1) returns", repr(g.send(1)))
print("send(\"b\") returns", repr(g.send("b")))
try:
    print("Now I will send(None)...")
    print(g.send(None))
    assert False, "I shouldn't have gotten here."

except StopIteration:
    print("Oh good, we're done.")
```

```
Initial send(None) returns 'first output'
send(1) returns 1
send("b") returns 'b'
Now I will send(None)...
Oh good, we're done.
```

In [169]:

```
# Here we use an extra yield just so it doesn't return.

def harrumph():
    input1 = yield()
    input2 = yield()
    input3 = yield()
    yield( (input1, input2) )
    yield(None)  # The extra unused yield.

h = harrumph()
h.send(None)  # To get started.
h.send("a")
h.send("b")
print("When I send(None) I get", h.send(None))
```

```
When I send(None) I get ('a', 'b')
```

In [171]:

```
# Wrapping the generator with functions.

def collector():
    def collect():
        collection = []
        input = yield()
        while input != None:
            collection.append(input)
            input = yield()
        yield(collection)
        yield(None)

    gen = collect()

    def result():
        return gen.send(None)

    def c(input):
        gen.send(input)

    c.result = result
    c(None)
    return c

c = collector()
c(1)
c("b")
c("zebra")
print(c.result())

```

```
[1, 'b', 'zebra']
```

In [120]:

```python
# Noticed a pattern in the min diff record breakers...
a, b = BPd(589825)
print(f"Most negative diff with BPe({a:}, {b:})")
y = BPe(a, b)
la, lb, ly = lg(a), lg(b), lg(y)
lo = ly - la - lb
nums = la, lb, ly, lo, lg(lo)
print(*("%.2f" % n for n in nums))
print()
for p in range(1, 127 + 1):
    a = 15
    b = 2**p - 1
    y = BPe(a, b)
    assert BPd(y) == (a, b)
    lgab = lg(a * b)
    diff = lg(y) - lgab - lg(lgab)
    print(f"({a:}, 2**{p:} - 1) => {diff:}")
```

```
(15, 2**108 - 1) => -1.8928666950468909
(15, 2**109 - 1) => -1.8929903062644078
(15, 2**110 - 1) => -1.8931117574060883
(15, 2**111 - 1) => -1.893231104602105 8
(15, 2**112 - 1) => -1.8933484020546008
(15, 2**113 - 1) => -1.89346370211 9714
(15, 2**114 - 1) => -1.8935770553855598
(15, 2**115 - 1) => -1.8936885107461725
(15, 2**116 - 1) => -1.8937981154718857
(15, 2**117 - 1) => -1.8939059152761528
(15, 2**118 - 1) => -1.8940119543791276
(15, 2**119 - 1) => -1.8941162755681535
(15, 2**120 - 1) => -1.8942189202553061
(15, 2**121 - 1) => -1.8943199285322105
(15, 2**122 - 1) => -1.894419339222286
(15, 2**123 - 1) => -1.894517189930478
(15, 2**124 - 1) => -1.894613517090697
(15, 2**125 - 1) => -1.8947083560110185
(15, 2**126 - 1) => -1.8948017409168996
```

Bug at BPe($15, 2^{48} - 1$), problem with log

Note: the encoding problem is fixed by using `int.bit_length()` , available since Python 3.1. There was a decoding problem starting at $y = 2^{2^{52}}$, fixed using `modf_lg()` , above.

On 2021-02-08 I hit a bug encoding the pair $(15, 2^{48} - 1)$, due to using the Python expression

```
int(log(2**48 - 1, 2))
```

and expecting to get 47. The difference caused by the "-1" above is roughly the slope of the base-2 log at 2^{48} ...

```
>>> s = log(2) / 2**48
>>> print(s)
2.462534697973183e-15
```

The ratio between the 48 itself and that tiny decrement, as a number of bits...

```
>>> print(log(48 / s, 2))
54.113728873666055
```

...is just below the resolution of a 64-bit floating point mantissa. Being *just* below means rounding happens, and the original expression gives

```
>>> print(int(log(2**48 - 1, 2)))
48
```

Here's a demo of the problem not happening, almost happening, and happening (prettied-up):

```
>>> for p in 46, 47, 48:
...     a = 2**p
...     print((log(a - 1, 2)), log(a, 2), log(a + 1, 2))
...
45.99999999999998 46.0              46.00000000000002
46.99999999999999 47.00000000000001 47.000000000000014
48.0              48.0              48.00000000000001
```

First, doing the `int(log(x, 2))` right

I want to pair numbers that could themselves represent large-ish amounts of information, like, say, the 2^{44} bits my backup drive holds. I've already hit a snag with 48-bit numbers. Technology improves exponentially. If I fix this bug with 48 bits, can I expect smooth sailing until, say, 2^{48} bits?

The Python `log` function works with any positive integer, and Python integers are limited only by virtual memory size (which on a 64-bit computer is 2^{67} bits) and swap space on the hard drive (2^{35} bits on my computer). But the output of `log` is a standard 64-bit floating-point number. I got in trouble counting on the part of a float below the decimal point. If this `BPe()/BPd()` code can work without depending on the fractional parts of floats, then I only need to know how large the integer part of a float can be before it too looses accuracy.

```
>>> for p in 51, 52, 53:
...     x = float(2**p)
...     print(p, end=" bits: ")
...     for step in range(3):
...         print(x, end=" ")
...         x += 1
...     print()
...
51 bits: 2251799813685248.0 2251799813685249.0 2251799813685250.0
52 bits: 4503599627370496.0 4503599627370497.0 4503599627370498.0
53 bits: 9007199254740992.0 9007199254740992.0 9007199254740992.0
```

Notice that on the last line, the numbers don't increment. Floats can represent all the integers up to 2^{52}.

Right now I can test with some pretty big numbers (here a 128 MB int-- big enough for an mp3 music album!-- that took about 20 seconds to create),

```
>>> (log(2**(2**30), 2), 2**30)
(1073741824.0, 1073741824)
```

but above that, let's assume there aren't problems with Python's `log()` function, only the limitations of float representation. The remaining question is how those limitations impact `BPe()` and `BPd()` . But first, a rounded-down base-two log function that I believe is robust up to $n = 2^{\text{(the largest float) / 2}}$.

In [20]:

```python
from math import log

_2M50 = float(2) ** -50

# Note, again: this is replaced by Python 3.1's int.bit_length().

def repaired_floor_lg(n):
    """
    Given int n: 1 <= n <= 2 ** (2 ** 1023),
        a computer with big enough memory, and
        at least Python 2.7's log() behavior,
        including 64-bit standard float result,
    return the true int(log(n, 2)).
    I.e., find the top bit.
    """
    # Assumptions:
    # o  For 2**48 <= n <= 2**(2**52),
    #    int() combined with float rounding
    #    can create errors up to +/- 1.
    # o  For (L = log(n, 2)) > 2**52,
    #    errors in L can be expected to be
    #    about +/- (L / 2**52).
    # The code below makes more cautious assumptions.
    shifted = 0
    L = log(n, 2)
    while L >= 47:
        # Get a new int L <= the true lg(n), but close.
        # For, e.g., n = 2**(2**60) (an exabit-sized
        # number), this underestimates by about 1024.
        L = int(L - L * _2M50)
        shift = L - 2
        n >>= shift
        shifted += shift
        L = log(n, 2)
    return shifted + int(L)

def naive_floor_lg(x): return int(log(x, 2))

def test_floor_lg(floor_lg):
    print("Testing", floor_lg.__name__, "...")
    for pp in list(range(46, 65 + 1)) + [1<<30]:
        n0 = 1<<pp
        print(pp, end=": ")
        results = []
        for n in n0 - 1, n0, n0 + 1:
            L = floor_lg(n)
            results.append(L)
            print(L, end=" ")
        assert tuple(results) == (pp - 1, pp, pp), (pp, results)
        print()
    print("Passed.")
    return True

# Fast to fail:
try:
    test_floor_lg(naive_floor_lg)
except:
```

```
        print("Failed.")

print()

# This test takes about 30 seconds (on my laptop).
if test_floor_lg(repaired_floor_lg):
    floor_lg = repaired_floor_lg
```

```
Testing naive_floor_lg ...
46: 45 46 46
47: 46 47 47
48: 48 48 48 Failed.

Testing repaired_floor_lg ...
46: 45 46 46
47: 46 47 47
48: 47 48 48
49: 48 49 49
50: 49 50 50
51: 50 51 51
52: 51 52 52
53: 52 53 53
54: 53 54 54
55: 54 55 55
56: 55 56 56
57: 56 57 57
58: 57 58 58
59: 58 59 59
60: 59 60 60
61: 60 61 61
62: 61 62 62
63: 62 63 63
64: 63 64 64
65: 64 65 65
1073741824: 1073741823 1073741824 1073741824
Passed.
```

What's the impact on `BPe()` and `BPd()` ?

`int(log(x, 2))` was used in the first step of `BPe(a, b)` : finding the lengths of a and b, which is straightforward, and `repaired_floor_lg()` fixes that.

The other place involving logs is the guts of `decode_pair_mant_start(y)` :

```
z = (y - 2) / 2     # float    z = (n-1) * 2**(n-1)
# let m = n - 1     #          z =     m * 2**m
m0 = lg(z)          # float
m1 = m0 - lg(m0)    # float
nmax = int(m1) + 2 # int
y_nmax = pair_mant_start(nmax)
if y >= y_nmax:
    return nmax, y_nmax
else:
    n = nmax - 1
    return n, pair_mant_start(n)
```

Although decoding is more complicated than encoding and seems to be relying on floatiness, there are some saving graces here.

1. m_1 gets closer to the correct m the higher m is (but never reaches it).
2. We need only get the correct int n or one less, then the code tests with `pair_mant_start()` (i.e. int arithmetic) and chooses the correct answer.
3. At our target, $y = 2^{2^{48}}$, the float $m_0 = \lg((y-1)/2)$ still has five bits after the decimal point.
4. Because we're working with logs here, we can simulate testing code on hypothetical ints so large we can't actually represent them as Python ints at the moment.

In [104]:

```python
def quickie_3():
    ns = list(range(2, 50 + 1))
    # ns += list(range(12, 20 + 1, 2))
    # ns += list(range(25, 50 + 1, 5))
    ys = [pair_mant_start(n) for n in ns]
    more_ys = [y - 1 for y in ys[1:]]
    ys = sorted(set(ys + more_ys))
    print("  n                          y    \"n1\"")
    for y in ys:
        n, y_n = decode_pair_mant_start(y)
        assert isinstance(n, int)
        if y_n == y:
            nstr = " %2d " % n
        else:
            nstr = "(%2d)" % n
        # z = (y - 2) / 2
        m0 = log(y - 2, 2) - 1
        m1 = m0 - log(m0, 2)
        iz = y - 2
        im0 = repaired_floor_lg(iz) - 1
        im1 = im0 - (log(im0, 2) + .775)  # .72 .. .83
        yes =  (int(im1) + 2) in {n - 1, n}
        print(nstr, f"{y:18d} {m1 + 1:5.2f} {im1 + 2:5.2f} {yes:}")

quickie_3()
```

```
  n                   y    "n1"
  2                   6  2.00  2.23 True
( 2)                 17  2.37  2.23 True
  3                  18  2.42  2.64 True
( 3)                 49  3.37  3.23 True
  4                  50  3.39  3.23 True
( 4)                129  4.41  3.90 True
  5                 130  4.42  4.64 True
( 5)                321  5.45  5.42 True
  6                 322  5.45  5.42 True
( 6)                769  6.48  6.22 True
  7                 770  6.48  6.22 True
( 7)               1793  7.51  7.06 True
  8                1794  7.51  7.06 True
( 8)               4097  8.54  7.90 True
  9                4098  8.54  8.77 True
( 9)               9217  9.56  9.64 True
 10                9218  9.56  9.64 True
(10)              20481 10.59 10.52 True
 11               20482 10.59 10.52 True
(11)              45057 11.61 11.42 True
 12               45058 11.61 11.42 True
(12)              98305 12.62 12.32 True
 13               98306 12.62 12.32 True
(13)             212993 13.64 13.22 True
 14              212994 13.64 13.22 True
(14)             458753 14.65 14.14 True
 15              458754 14.65 14.14 True
(15)             983041 15.67 15.06 True
 16              983042 15.67 15.06 True
(16)            2097153 16.68 15.98 True
```

17	2097154	16.68	16.90	True
(17)	4456449	17.69	17.83	True
18	4456450	17.69	17.83	True
(18)	9437185	18.70	18.77	True
19	9437186	18.70	18.77	True
(19)	19922945	19.71	19.70	True
20	19922946	19.71	19.70	True
(20)	41943041	20.72	20.64	True
21	41943042	20.72	20.64	True
(21)	88080385	21.73	21.58	True
22	88080386	21.73	21.58	True
(22)	184549377	22.73	22.52	True
23	184549378	22.73	22.52	True
(23)	385875969	23.74	23.47	True
24	385875970	23.74	23.47	True
(24)	805306369	24.75	24.42	True
25	805306370	24.75	24.42	True
(25)	1677721601	25.75	25.37	True
26	1677721602	25.75	25.37	True
(26)	3489660929	26.76	26.32	True
27	3489660930	26.76	26.32	True
(27)	7247757313	27.77	27.27	True
28	7247757314	27.77	27.27	True
(28)	15032385537	28.77	28.23	True
29	15032385538	28.77	28.23	True
(29)	31138512897	29.78	29.18	True
30	31138512898	29.78	29.18	True
(30)	64424509441	30.78	30.14	True
31	64424509442	30.78	30.14	True
(31)	133143986177	31.79	31.10	True
32	133143986178	31.79	31.10	True
(32)	274877906945	32.79	32.06	True
33	274877906946	32.79	33.02	True
(33)	566935683073	33.79	33.98	True
34	566935683074	33.79	33.98	True
(34)	1168231104513	34.80	34.94	True
35	1168231104514	34.80	34.94	True
(35)	2405181685761	35.80	35.90	True
36	2405181685762	35.80	35.90	True
(36)	4947802324993	36.81	36.87	True
37	4947802324994	36.81	36.87	True
(37)	10170482556929	37.81	37.83	True
38	10170482556930	37.81	37.83	True
(38)	20890720927745	38.81	38.80	True
39	20890720927746	38.81	38.80	True
(39)	42880953483265	39.82	39.77	True
40	42880953483266	39.82	39.77	True
(40)	87960930222081	40.82	40.73	True
41	87960930222082	40.82	40.73	True
(41)	180319906955265	41.82	41.70	True
42	180319906955266	41.82	41.70	True
(42)	369435906932737	42.83	42.67	True
43	369435906932738	42.83	42.67	True
(43)	756463999909889	43.83	43.64	True
44	756463999909890	43.83	43.64	True
(44)	1548112371908609	44.83	44.61	True
45	1548112371908610	44.83	44.61	True

```
(45)   3166593487994881 45.83 45.58 True
 46    3166593487994882 45.83 45.58 True
(46)   6473924464345089 46.84 46.55 True
 47    6473924464345090 46.84 46.55 True
(47)  13229323905400833 47.84 47.52 True
 48   13229323905400834 47.84 47.52 True
(48)  27021597764222977 48.84 48.50 True
 49   27021597764222978 48.84 48.50 True
(49)  55169095435288577 49.84 49.47 True
 50   55169095435288578 49.84 49.47 True
```

In [126]:

```
for i in 50, 51, 52, 53, 54:
    j = 2**i - 1
    print(i, j / 2**i, log(j/2**i, 2))

```

```
50 0.9999999999999991 -1.2813706015259676e-15
51 0.9999999999999996 -6.406853007629837e-16
52 0.9999999999999998 -3.2034265038149176e-16
53 0.9999999999999999 -1.6017132519074588e-16
54 1.0 0.0
```

In [232]:

```
def debug_BPe(a, b):
    """
    Given int a >= 1 and int b >= 1, return the encoded pair
          as int y >= 1.
    """
    print(f"debug_BPe({a:}, {b:}):")
    sz_mant_a = int(log(a, 2)); mant_a = a - (1 << sz_mant_a)
    assert mant_a >= 0
    hsz_m_a = (sz_mant_a + 3) // 4
    print(f"    mant_a = 0x{mant_a:0{hsz_m_a}x} {mant_a:} sz {sz_mant_a
    sz_mant_b = int(log(b, 2)); mant_b = b - (1 << sz_mant_b)
    hsz_m_b = (sz_mant_b + 3) // 4
    print(f"    mant_b = 0x{mant_b:0{hsz_m_b}x} {mant_b:} sz {sz_mant_b
    assert mant_b >= 0
    pair_mant = (mant_a << sz_mant_b) + mant_b
    sz_pair_mant = sz_mant_a + sz_mant_b
    start = pair_mant_start(sz_pair_mant)
    return start + (sz_mant_a << sz_pair_mant) + pair_mant


a, b = 15, 2**48 - 1
y = debug_BPe(a, b)
print(f"y = BPe(15, 2**48 - 1) = {y:} = 0x{y:x}")
print("lg(y) =", lg(y))
dsz = decode_pair_mant_start(y)
dn, dy_dn = dsz
print(f"decode_pair_mant_start(y) = (dn, dy_dn)   = ({dn:}, 0x{dpmsn:x}
y_dn = pair_mant_start(dn)
if dy_dn == y_dn:
    # print("y_dn = dy_dn = pair_mant_start(dn))")
    pass
else:
    print(f"pair_mant_start(dn)) = 0x{y_d:x})")
    assert False

assert y >= y_dn, "y < y_dn"

y_dnp1 = pair_mant_start(dn + 1)
print(f"(dn + 1, pms(dn + 1) =                   ({dn + 1:}, 0x{y_dnp1:

print(f" a,  b = 0x{a:x}, 0x{b:x}")
da, db = BPd(y)
print(f"da, db = 0x{da:x}, 0x{db:x}")

```

```
debug_BPe(15, 281474976710655):
    mant_a = 0x7 7 sz 3
    mant_b = 0x-000000000001 -1 sz 48

------------------------------------------------------------------------
----
AssertionError                            Traceback (most recent call l
ast)
<ipython-input-232-652f524bafaa> in <module>
     20
     21 a, b = 15, 2**48 - 1
---> 22 y = debug_BPe(a, b)
```

```
     23 print(f"y = BPe(15, 2**48 - 1) = {y:} = 0x{y:x}")
     24 print("lg(y) =", lg(y))

<ipython-input-232-652f524bafaa> in debug_BPe(a, b)
     12     hsz_m_b = (sz_mant_b + 3) // 4
     13     print(f"    mant_b = 0x{mant_b:0{hsz_m_b}x} {mant_b:} sz {s
z_mant_b}")
---> 14     assert mant_b >= 0
     15     pair_mant = (mant_a << sz_mant_b) + mant_b
     16     sz_pair_mant = sz_mant_a + sz_mant_b

AssertionError:
```

Equation-solving attempts

It turns out that "solving" the equation to the point where one can decide between two adjacent ints with int arithmetic is relatively simple (two steps) regardless of the size of y. Further convergence is unnecessary. But the problem (solved above, I *think*) turns out to be handling the big numbers where the *math* part of the problem keeps getting *easier*.

In [107]:

```python
from math import log

import changing
from changing import Changing, ChangingIndep
# help(changing)

# ==== Solving the original equation with Newton. ====

y = 32 * 2**32
n = log(y, 2)
n = ChangingIndep(n=log(y, 2))

def f(n):
    return (n - 1) * 2**n

for i in range(11):
    f_n = f(n)
    err = f_n - y
    print("n =", n)
    print("f_n =", f_n)
    print(f_n / n, "= f_n / n")
    print(y, "= y")
    print(f_n / y, "= f_n / y")
    print(err, "= err")
    print(err.slope, "= err.slope")
    n = ChangingIndep(n=float(n) - float(err) / float(err.slope))
    print()
    if abs(err) < .5:
        break


print("n =", n)



if False:
    print("    err", float(err)) # 7177177277359.105   5316624969192.3
    print("    slope", float(err.slope))
    n = ChangingIndep(n=float(n) - float(err / err.slope))

```

```
n = 37.0
f_n = 4947802324992.0
133724387161.94595 = f_n / n
137438953472 = y
36.0 = f_n / y
4810363371520.0 = err
3566994185008.147 = err.slope

n = 35.651423825769704
f_n = 1870118145524.9382
52455636965.98204 = f_n / n
137438953472 = y
13.606900360354762 = f_n / y
1732679192052.9382 = err
1350236565842.3826 = err.slope
```

```
n = 34.368182625016146
f_n = 739922817802.0034
21529297195.465416 = f_n / n
137438953472 = y
5.3836470600945425 = f_n / y
602483864330.0034 = err
535049916555.3013 = err.slope

n = 33.24214962194823
f_n = 327573035801.5438
9854147205.488262 = f_n / n
137438953472 = y
2.3834075240414228 = f_n / y
190134082329.54382 = err
237216102747.9818 = err.slope

n = 32.44062695916533
f_n = 183270679727.2692
5649418550.324608 = f_n / n
137438953472 = y
1.3334696976181963 = f_n / y
45831726255.269196 = err
132862658784.58908 = err.slope

n = 32.09567129837597
f_n = 142711748998.8018
4446448484.348198 = f_n / n
137438953472 = y
1.0383646367612658 = f_n / y
5272795526.801788 = err
103509687458.3573 = err.slope

n = 32.044731181984226
f_n = 137534987522.45428
4291968833.8586354 = f_n / n
137438953472 = y
1.0006987396806237 = f_n / y
96034050.45428467 = err
99762208782.68161 = err.slope

n = 32.04376855243115
f_n = 137438986926.09744
4289101848.342678 = f_n / n
137438953472 = y
1.000000243410595 = f_n / y
33454.09744262695 = err
99692711188.45654 = err.slope

n = 32.043768216859
f_n = 137438953472.00372
4289100849.2469926 = f_n / n
137438953472 = y
1.000000000000027 = f_n / y
0.00372314453125 = err
99692686970.06017 = err.slope
```

```
n = 32.043768216858965
```

In [169]:

```python
# Solve x * 2**x = y for x and look at convergence.

def lg(x): return log(x, 2)

def solve(x):
    y = x * 2**x
    x_est = lg(y)
    for i in range(10):
        errbits = lg(abs(x_est - x) / x)
        print(i, x_est, errbits)
        x_est = lg(y / x_est)

def solve_log(x):
    lgy = lg(x) + x
    x_est = lgy
    for i in range(10):
        err = lg(x_est) + x_est - lgy
        if abs(err) > 0:
            errbits = lg(abs(err) / lgy)
        else:
            errbits = "oo"
        print(i, x_est, errbits)
        if err == 0:
            break

        slope = 1 / x_est + 1
        x_est -= err / slope

def quickie(y): return lg(y / lg(y / lg(y)))

def quickier(y): return lg(y / lg(y))

# solve(32)
# print()
# solve_log(32)
def blodge():
    for p in 2, 3, 4, 8, 16, 32:
        x = 2**p
        y = x * 2**x
        q = quickie(y)
        print(x, lg(y / lg(y)), q, lg(y / q))

def nosler():
    # Interesting pattern, off topic.
    for p in range(9):
        x = 3 * 2**p
        print(p, x, 2**x / x, x - lg(x), x + lg(x))
    print("2**(1-.4150374992788437) =", 2**(1-.4150374992788437))
# nosler()

def quickie_demo():
    print("quickie demo.")
    for n in 32, 16, 8, 7, 6, 5, 4, 3, 2:
        print(f"    For n =", n)
        y = (n - 1) * 2**n + 2
        print(f"    y = {(n - 1):d} * 2**{n:d} + 2 = {y:d}")
        z = (y - 2) // 2
```

```
        print(f"    z = ({y:.1f} - 2) / 2 = {z:d}")
        print(f"    m = {n:d} - 1 = {n - 1:d}")
        # z = m * 2**m
        # print(f"    For z = {m:d} * 2**{m:d}")
        m0 = lg(z)
        print(f"    m0 = lg(z) = {m0:.2f}")
        m1 = lg(z / m0)
        print(f"    m1 = lg(z / m0) = {m1:.2f}")
        m2 = lg(z / m1)
        print(f"    m2 = lg(z / m1) = {m2:.2f}")
        print()

from prcol import print_in_columns

def quickie_demo_2():
    print("quickie demo 2.")
    for n in 128, 64, 32, 16, 8, 7, 6, 5, 4, 3, 2:
        rows = []
        row = "    ", f"For n = {n:d}", "", ""
        rows.append(row)

        row = "    ", "y", "m1", "m2"
        rows.append(row)

        ymin = (n - 1) * 2**n + 2
        ymax = n * 2**(n+1) + 2 - 1
        for y in ymin, ymax:
            z = (y - 2) // 2
            m0 = lg(z)
            m1 = lg(z / m0)
            m2 = lg(z / m1)
            row = "    ", y, f"{m1:.2f}", f"{m2:.2f}"
            rows.append(row)
        print_in_columns(rows)
        print()

quickie_demo_2()
```

```
quickie demo 2.
                                  For n = 128
                                            y      m1      m2
    43215860598959184859848575143834562854914 126.92 127.00
    87112285931760246646623899502532662132737 127.92 128.00

                For n = 64
                         y     m1     m2
    1162144876643701751810 62.87 63.00
    2361183241434822606849 63.87 64.00

      For n = 32
                  y     m1     m2
    133143986178 30.79 31.01
    274877906945 31.79 32.01

    For n = 16
            y     m1     m2
```

```
    983042 14.67 15.03
   2097153 15.68 16.03

For n = 8
        y   m1   m2
     1794 6.51 7.10
     4097 7.54 8.08

For n = 7
        y   m1   m2
      770 5.48 6.13
     1793 6.51 7.10

For n = 6
        y   m1   m2
      322 4.45 5.17
      769 5.48 6.13

For n = 5
        y   m1   m2
      130 3.42 4.23
      321 4.44 5.16

For n = 4
        y   m1   m2
       50 2.39 3.33
      129 3.40 4.21

For n = 3
        y   m1   m2
       18 1.42 2.50
       49 2.35 3.29

For n = 2
        y   m1   m2
        6 1.00 1.00
       17 1.32 2.41
```