

# Linked List Bignums

## CSCI 112, Lab 4

**File names:** Names of files, functions, and variables, when specified, must be EXACTLY as specified. This includes simple mistakes such as capitalization.

**Individual work:** All work must be your own. Do not share code with anyone other than the instructor and teaching assistants. This includes looking over shoulders at screens with the code open. You may discuss ideas, algorithms, approaches, *etc.* with other students but NEVER actual code. Do not use code written by anyone else, in the class or from the internet.

**Documentation:** Each file should begin with a docstring that includes your name, the class number and name, the lab number, and a short description of the lab, as well as documentation pertinent to that particular file.

**Addition and subtraction standard algorithm:** You should be familiar with the standard algorithms for addition and subtraction, at least in base 10. If you are not familiar with other number bases, review them quickly here: <https://www.mathsisfun.com/numbers/bases.html>.

The standard subtraction and addition algorithms work fine in any base. You just have to remember that if you are in base 16, say, and you “borrow one” from the next column, you are borrowing 16, not 10. Likewise, you “carry one” to the next column when you get 16 or more, not 10. Here are some worked examples to get the hang of things:

Base 10 examples:	$\begin{array}{r} 3915 \\ + 1245 \\ \hline 5160 \end{array}$	$\begin{array}{r} 3915 \\ - 1245 \\ \hline 2670 \end{array}$
Base 8 examples:	$\begin{array}{r} 7513 \\ + 2335 \\ \hline 12050 \end{array}$	$\begin{array}{r} 7513 \\ - 2335 \\ \hline 5156 \end{array}$
Base 16 examples:	$\begin{array}{r} 15411 \\ + 41313 \\ \hline 1428 \end{array}$	$\begin{array}{r} 15411 \\ - 41313 \\ \hline 10614 \end{array}$

If you want more examples, just run my program `arithmetic.py` and paste the output into a new project on <https://www.overleaf.com>.

You can get easy base 8 and base 16 examples from python (remember that in base 16 the digits 10 to 15 are: a, b, c, d, e, f):

```
1 >>> hex(0xaaa + 0x123)
2 '0xbcd'
3 >>> oct(0o666 + 0o123)
4 '0o1011'
```

**Bignums:** Python has bignums (arbitrarily high integers) built in. For example, Python has no problem computing:

```
1 >>> 1234567890987654321 * 1234567890987654321
2 1524157877457704723228166437789971041
```

Even though computers only natively support either 32 bit integers, or 64 bit integers. Hence, the maximum integers supported in most programming languages are either  $2^{32} = 4.294.967.296$  or  $2^{64} = 18.446.744.073.709.551.616$ . Integers larger than this are supported in software.

When Python encounters integers that are too big for the 32 or 64 bit registers, it automatically converts them to bignums. Python's bignums are implemented as dynamic arrays of digits in a large, fixed base.

In this lab we will build a Bignum class of our own that supports arbitrarily large integers using linked lists. Each cell in the linked list will hold a single position in the base. For example, if we choose base 10 for our bignums, each cell in the linked list will hold a Python integer from 0 to 9. If we choose base 16 for our bignums, each cell in the linked list will hold a Python integer between 0 and 15. If we choose 10,000 for our base, each cell in the linked list will hold a Python integer between 0 and 9,999.

Here, for example, is a figure of what the number 00374 in base 8 looks like:



Note that the smallest digit will be at the head of the list. This will make arithmetic much easier.

Make your Bignum class general, so it will support any base.

Since there's no such thing as an "empty" bignum, we will not need the header class used for linked lists in the book. A bignum will hold the digit, the sign, and a pointer to the next bigger digit.

**Signs:** Positive and negative numbers will be represented by a separate field in the class called `Bignum.sign`, with the value of either plus or minus one.

**Initializing with Python ints:** To make testing easy, Bignums will be initialized by Python integers. The initializing integer can be any size, of course, using Python's native integers. You will convert this to your linked list representation of Bignums. Here is the initializing routine for bignums that you should use:

```

1  def __init__(self, n, base = 17):
2      self.base = base
3      if n < 0:
4          self.sign = -1
5      else:
6          self.sign = 1
7      self.n = abs(n)
8      self.next = None
9      self.overflow()

```

Needless to say, the `Bignum.overflow` method will take care of normalizing the integer to the range  $[0..(\text{self.base}-1)]$  and pushing any overflows ("carries") along to the next cell in the list.

**Extracting Python ints:** Also to make testing easy, include a `Bignum.int` method that converts your bignum into a Python int. This will support easy testing expressions such as:

```

1  n = 2**100
2  self.assertEqual(n, Bignum(n).int())

```

**String method:** While it will be nice to see the bignums converted to Python integers for testing, I find it easier to debug if I can see what the data structure looks like. Build a `__str__` method that shows the contents of bignums like this:

```

1  >>> print(Bignum(20,8))
2  Bignum base 8: 4:2:+
3  >>> print(Bignum(20,2))
4  Bignum base 2: 0:0:1:0:1:+
5  >>> print(Bignum(-123,10))
6  Bignum base 10: 3:2:1:-

```

```

7 >>> print(Bignum(50,16))
8 Bignum base 16: 2:3:+
9 >>>

```

**Addition and subtraction:** Once you've got your Bignum class working, implement the standard algorithms for addition and subtraction, as illustrated above, to support the `__add__` and `__sub__` methods for your Bignums.

You will have to do a little sign checking at the beginning. For example, to subtract a negative number from a positive number, just convert the negative to a positive and then add. Or, for another example, to subtract a larger number,  $a$ , from a smaller number,  $b$ , use the formula:

$$b - a = -(a - b)$$

So you will always be subtracting the smaller from the larger. There are a few other cases for you to work out: each operand can be positive or negative, and either one could be the larger. There should be eight cases in all, then, right?

**Leading zeros:** Some operations may result in leading zeros, for example,  $12345 - 12300$  in base 10. Your Bignum class should clean up any result by removing these unnecessary zeros.

```

1 >>> print(Bignum(123456, 10) - Bignum(123000, 10))
2 Bignum base 10: 6:5:4:+
3 >>>

```

**Recursion:** Note, if you use recursion in your bignum calculations, you may run into recursion depth limits with very large numbers. This will not be considered a bug.

**Calculator:** Use your infix calculator from the previous lab to make a Bignum calculator!

**Turn in:** Files `bignum.py` and `bignum_test.py` in folder `csci112lab04yourname`, zipped and turned into Canvas.

**Optional additions:**

- Multiplication can be done with the standard algorithm, or using the following facts:

$$ab = \begin{cases} 0 & \text{if } b = 0 \\ a + a(b-1) & \text{if } b \text{ is odd} \\ 2(a(b/2)) & \text{if } b \text{ is even} \end{cases}$$

You will have to implement a divide by 2 function, but the rest can be done with recursion and addition.

- Division, exponentiation, *etc.*
- Run some experiments to see how the timing of your Bignums changes with the size of the base. Note that Python will not use native integers if the size of the integer is greater than the size of a native integer. Since your calculations can result in numbers in the cells up to  $2(\text{base})$ , or  $\text{base}^2$  if you do multiplication, your base should probably not exceed  $2^{16}$  for 32 bit computers, or  $2^{32}$  for 64 bit computers. (Why?)

Once you find the best base for your implementation, compare times on your implementation with those for native Python bignums. How much slower are you? Do you think it makes a big-O difference, or not? Why? Can you prove it with some data?

Write up your experiments and conclusions in a short document.

**My implementation headers:** You do not have to follow my implementation here, but in my Bignum class I used the following approach. Some of my functions are recursive, and some are iterative.

I have also supplied my unit test file, as a start, for yours.

```
1 def __init__(self, n, base = 17):
2 def __str__(self):
3 def int(self):
4     '''convert to Python integer'''
5 def copy(self, end=None):
6 def add(self, other):
7     '''add bodies of numbers without regard to signs'''
8 def __add__(self, other):
9     '''figure out signs and sizes, then add or sub and fix result'''
10 def sub(self, other):
11     '''subtract bodies without regard to signs but allow larger and smaller each way'''
12 def __sub__(self, other):
13     '''figure out signs then add or sub and fix result'''
14 def cmp(self, other):
15     '''compare size of bodies, no signs, return -1,0,1 for less, equal, greater'''
16 def __lt__(self, other):
17     '''consider signs in comparison'''
18 def __eq__(self, other):
19     '''needed for unit tests'''
20 def remove_zeros(self):
21     '''remove leading zeros'''
22     def inc_next(self, amt=1):
23         '''if amt > 0 increment the next cell. if absent, create it'''
24 def overflow(self):
25     '''any cell that is larger than the base is reduced and carried to the next cell'''
26 def dec_next(self, amt=1):
27     '''if amt > 0 decrement next cell, if absent, raise error'''
28 def underflow(self):
29     '''any cell that is less than 0 is incremented by borrowing from the next'''
```