1. To refresh your memory, `UnorderedList` has methods `isEmpty()`, `length()`, `add()`, `search()`, and `remove()`, in addition to the constructor. In our implementation from class we had to recompute the length every time the user asked for it, which was inefficient. Eric came up with the idea of keeping track of the length as items were added and removed from the list; then we could just return the current length whenever asked.

A. Write a subclass of `UnorderedList` called `FasterUnorderedList` that uses Eric's suggestion. (If you subclass well, then you do not need to write much code.)

Answer:

```
from linkedlist import *

class FasterUnorderedList(UnorderedList):
    def __init__(self):
        UnorderedList.__init__(self)
        self.count = 0

    def length(self):
        return self.count

    def add(self,item):
        UnorderedList.add(self, item)
        self.count += 1

    def remove(self,item):
        UnorderedList.remove(self, item)
        self.count -= 1
```

[Remark: `self.count` cannot be called `self.length`, because that would conflict with the `length()` method.]

B. How would the abstract data type (ADT) specification for `FasterUnorderedList` differ from that for `UnorderedList`?

Answer: The two ADT specs are identical. `UnorderedList` and `FasterUnorderedList` respond to exactly the same methods. The only difference between the classes is that `FasterUnorderedList` uses slightly more memory (to store `self.count`) and performs `length()` in time $O(1)$ instead of $O(n)$. Such issues of memory and time efficiency are not part of the ADT spec.

2. A counterintuitive aspect of the radix sort we've studied is that it sorts by digits from right to left, even though the left-most digits are the most significant. To correct this "defect" I

altered the code — only a few lines needed changing — to sort by digit left-to-right instead of right-to-left. (To understand this, it is helpful to assume that all numbers are of the same length; i.e. shorter numbers have leading 0s.) Here is its output:

```
unsorted:
[32, 127, 5, 3456, 6, 1, 99, 245]
sorting...
[32, 127, 5, 6, 1, 99, 245, 3456]
[32, 5, 6, 1, 99, 127, 245, 3456]
[5, 6, 1, 127, 32, 245, 3456, 99]
[1, 32, 5, 245, 6, 3456, 127, 99]
sorted:
[1, 32, 5, 245, 6, 3456, 127, 99]
```

Is this thing working as I described? Why don't I end up with a sorted list? Why does the standard radix sort start with the least significant digits? Discuss.

Answer: Yes, it is working as I described, but clearly the left-to-right algorithm is wrong. The left-to-right algorithm essentially sorts by the 1s digit, breaking ties using the 10s digit, breaking those ties using the 100s digit, and so on. In contrast, the usual right-to-left algorithm sorts by 1000s digit (say), breaks ties using the 100s digit, breaks those ties using the 10s digit, and breaks those ties using the 1s digit; this produces the correct sorted order. To put it another way, on each iteration the radix sort the list is resorted, with the results of the previous iteration partially overridden. By sorting right-to-left, the more significant digits are able to override the less significant digits, as desired.

3. The *Collatz function* is defined recursively by the following Python code. In the right margin of this page, from the bottom of the page up, draw the call stack for `collatz(6)` when it is at its deepest. On the rest of this page, write an iterative version of `collatz()`.

```
def collatz(n):
    if n == 1:
        return n
    else:
        if n % 2 == 0:
            return collatz(n / 2)
        else:
            return collatz(3 * n + 1)
```

Answer: The iterative version is

```
def collatz(n):
    while n != 1:
        if n % 2 == 0:
            n = n / 2
        else:
            n = 3 * n + 1
    return n
```

The call stack for `collatz(6)` at its deepest is

```
collatz(1)
collatz(2)
collatz(4)
collatz(8)
collatz(16)
collatz(5)
collatz(10)
collatz(3)
collatz(6)
```

[Remark: The only value that the Collatz function can ever return is 1, so a simpler implementation would be

```
def collatz(n):
    return 1
```

However, this assumes that the original `collatz()` function always terminates. In fact, it is not currently known whether the Collatz function terminates for all positive n. Experiments show that it terminates for all n less than about 10^{18}, but little is known beyond that.]

4. In our radix sort function we used the Python operator `%`, which computes the remainder when one integer is divided by another. For example, `3456 % 100` returns `56`, `25 % 4` returns `1`, and `36 % 6` returns `0`. In this problem you will write this remainder function yourself. Let's just call it `remainder()`. It takes in two positive integers `a` and `b`, and returns the remainder for `a` divided by `b`. So `remainder(25, 4)` returns `1`. One approach is to repeatedly subtract 4 from 25 until you have the remainder; how do you know when that is?

A. Write an iterative version of `remainder()` that uses this repeated subtraction idea.

Answer:

```
def remainder(a, b):
```

```
    while a >= b:
        a = a - b
    return a
```

B. Write a recursive version of `remainder()` that uses repeated subtraction.

Answer:

```
def remainder(a, b):
    if a < b:
        return a
    else:
        return remainder(a - b, b)
```

C. Describe the running time of either function using big-O notation.

Answer: The number of subtractions involved is exactly $\lfloor a/b \rfloor$, which is approximately a/b. (To be precise, $a/b - 1 < \lfloor a/b \rfloor \leq a/b$.) Therefore the algorithm is $O(a/b)$.

5. We've discussed using queues for scheduling jobs on a printer and tasks on a processor. In Real Life situations we don't always want all tasks to be scheduled fairly; we want the ability to specify that some jobs have priority over others, so that they get handled first. (Jobs that have equal priority should be handled in a FIFO manner.) Let us assume that every item that will be put in our queue responds to a `priority()` message that returns its assigned priority (an integer, say). So, given an item `item`, its priority is `item.priority()`.

Write `enqueue()` and `dequeue()` for this new kind of queue. Also describe the complexity of your `enqueue()` and `dequeue()` using big-O notation.

Answer: I will only show `enqueue()`; `dequeue()` (and all other methods) are identical to those in `Queue`, given this `enqueue()`.

```
class PriorityQueue():
    def enqueue(self, item):
        # Scan through the list until we reach a high-priority item, or the end.
        i = 0
        while i < self.length() and item.priority() > self.items[i].priority():
            i += 1
        if i == self.length():
            # We reached the end; just append.
            self.items.append(item)
        else:
            # i is the index of the higher-priority item; insert behind it.
            self.items.insert(i, item)
```

[Remark: Because almost all of the methods are identical to our usual `Queue` class, I was tempted to write a subclass. However, the subclass would not have access to the queue's underlying list, and I want access. So I had to write a new class from scratch.]

The algorithm scans through the list. If the list has n items, then this scan takes time $O(n)$. Additionally, the algorithm has to perform an `append()` or `insert()`; I don't remember whether those are $O(1)$ or $O(n)$ in Python, but even if they are $O(n)$ they still don't take any more time than the scan. So the algorithm is $O(n)$ overall.