I'm going to ask you two unrelated questions about our `gcd` function from class:

```
def gcd(a, b):
    """The arguments a and b are integers with a >= b >= 0. Returns the greatest
    common divisor of a and b."""
    if b == 0:
        return a
    else:
        return gcd(b, a % b)
```

First, what happens if we accidentally call `gcd` with `b` greater than `a` (and both still nonnegative)?

Answer: If `b` is greater than `a`, then `a % b` and `a` have the same value. So the first recursive call is `gcd(b, a)`. This effectively switches the arguments, so that the first argument is greater than the second. The rest of the recursion works correctly, and the function returns the correct answer.

Second, let m be the number of decimal digits in `a`, and let n be the number of decimal digits in `b`. Using big-$\mathcal{O}$ notation, describe the running time of `gcd` in terms of m and/or n.

THIS PART HAS BEEN CANCELLED; DO NOT DO IT.

Answer: We know from class that `gcd` is $\mathcal{O}(\log a)$. That is, the running time is proportional to $\log_2 a$. The number of decimal digits in `a` is $m = \log_{10} a$, which is proportional to $\log_2 a$. Therefore the running time is proportional to m, and `gcd` is $\mathcal{O}(m)$.

Perform a "queue-based radix sort" on the following list of numbers, showing each intermediate result along the way. `[313, 7, 21, 3241, 3420, 27]`

Answer:

```
[313, 7, 21, 3241, 3420, 27]
[3420, 21, 3241, 313, 7, 27]
[7, 313, 3420, 21, 27, 3241]
[7, 21, 27, 3241, 313, 3420]
[7, 21, 27, 313, 3241, 3420]
```

Perform a "stack-based radix sort" on the same list of numbers, showing each intermediate result along the way. `[313, 7, 21, 3241, 3420, 27]`

Answer: At each step, ties are broken by reversing the ordering from the previous step.

```
[313, 7, 21, 3241, 3420, 27]
[3420, 3241, 21, 313, 27, 7]
```

```
[7, 313, 27, 21, 3420, 3241]
[21, 27, 7, 3241, 313, 3420]
[313, 7, 27, 21, 3420, 3241]
[313, 7, 27, 21, 3420, 3241]
```

Remember the `BetterSet` class, which was a subclass of `Set`? I just remembered another method that I wanted for `BetterSet`. Please implement this method recursively.

```
def subset(self, f):
    """The argument f is a predicate (a function of one argument, that outputs
    either True or False). Returns a new BetterSet, consisting of all of the
    elements in this set for which f is True. This set is left unaltered."""
```

Answer:

```
    if self.getSize() == 0:
        return BetterSet()
    else:
        # Recurse on a slightly smaller set.
        e = self.getElement()
        self.remove(e)
        result = self.subset(f)
        self.add(e)
        # Adjust for the final element.
        if f(e):
            result.add(e)
        return result
```

Remark: This is just the filter operation on sets.

Using big-$\mathcal{O}$ notation, describe the running time of `subset`.

Answer: Through recursion it traverses the entire set. On each recursive call, some of the work (getting an element, removing an element just gotten) is constant-time, but some of the work (adding) is linear-time. On a set of size n, the linear-time work done in each recursive call is (no worse than) $\mathcal{O}(n)$, and the recursion goes n steps deep. Thus the running time is $\mathcal{O}(n^2)$.

Consider the following implementation of `List`. As usual, there is a big linked list, based at `self.head`, containing all of the data nodes, and there is a `self.count` member that tracks the length n of the data list. But there is a second linked list, based at `self.shortcuts`, of length

$\sqrt{n}$. (For simplicity, assume that n is a perfect square, so that $\sqrt{n}$ is an integer.) Each node in this shortcut list contains, as its data element, a pointer to a node in the big linked list. These shortcut pointers are spaced evenly throughout the big linked list. For example, if $n = 100$, then the shortcut list has 10 nodes, which point to nodes $0, 10, 20, ..., 80, 90$ of the data list.
Implement `__getitem__`. Do not include a docstring.

Answer:

```
def __getitem__(self, i):
    if i >= 0 and i < self.count:
        # Compute how many shortcut steps and regular steps to take.
        root = int(math.sqrt(self.count))
        q = i / root
        r = i % root
        # Traverse the shortcut list.
        current = self.shortcuts
        while q > 0:
            current = current.getNext()
            q -= 1
        # Jump over to the regular data list.
        current = current.getData()
        # Traverse the data list.
        while r > 0:
            current = current.getNext()
            r -= 1
        return current.getData()
```

Using big-$\mathcal{O}$ notation, describe the running time of `__getitem__`.

Answer: Computing the numbers of steps to take is constant-time. We may have to take as many as $\sqrt{n}$ shortcut steps, then one step over to the data list, then as many as $\sqrt{n}$ regular steps, each of which is constant-time. So the work seems proportional to $2\sqrt{n}$, which is $\mathcal{O}(\sqrt{n})$.

In this question, an $m \times n$ *grid* is a `List` of m items, each of which is a `List` of n numbers. For example, `((3, -1, 2.1), (14, 3.25, 3))` is a 2×3 grid. You are writing a program that contains a grid `g` of indeterminate size. Using `listMap`, `listFilter`, and `listFold`, find the smallest number in `g`. (You may not need to use all three functions. However, you should try to get as much work done using these functions as you can, rather than writing your own custom code. My solution fits in one line.)

Answer: `listFold(min, listMap((lambda l: listFold(min, l)), g))`

Using big-$\mathcal{O}$ notation, describe how much time that computation takes.

Answer: The interior `listFold` call is operating on a list of length n, and hence is $\mathcal{O}(n^2)$. This gets mapped across a list of m elements, so the total cost of the `listMap` is $\mathcal{O}(mn^2)$. The output of the `listMap` call is a list of m elements, so the exterior `listFold` call is $\mathcal{O}(m^2)$. The total cost is $\mathcal{O}(mn^2 + m^2)$. If for simplicity we let $N = \max(m, n)$, then the total cost is $\mathcal{O}(N^3)$.

Remark: A custom-made algorithm — either iterative or recursive — could do all of this in $\mathcal{O}(mn)$. The solution given above is slower because `listFold` is so slow ($\mathcal{O}(n^2)$ on a `List` of n elements). The `listFold` we've been using is called *left-folding*; there is also a notion of *right-folding*, which would work just as well in this problem, but much more quickly ($\mathcal{O}(n)$ on a list of n elements), for a total solution time of $\mathcal{O}(mn)$. You could read about left- vs. right-folding on Wikipedia, implement right-folding, and figure out how it allows for $\mathcal{O}(mn)$.