1. I'll give two answers. The first answer begins with `Queue`, as implemented in class, with constant-time enqueueing and dequeueing. We can define `RoundRobin` as a subclass of `Queue`, with methods

```
def add(self, obj):
    self.enqueue(obj)

def next(self):
    if self.size() == 0:
        return None
    else:
        temp = self.dequeue()
        self.enqueue(temp)
        return temp
```

These `next()` and `add()` methods are just sequences of constant-time operations, so they are also constant-time — that is, $\mathcal{O}(1)$.

For the second answer, instead of subclassing `Queue` imagine a linked list with `head` and `tail` pointers, but such that the next-pointer on the `tail` node points back to the `head` node. Then my methods are

```
def __init__(self):
    self.head = None
    self.tail = None

def add(self, obj):
    if self.tail == None:
        # This is the first object in the mix.
        self.tail = Node(obj)
        self.tail.setNext(self.tail)
        self.head = self.tail
    else:
        newNode = Node(obj)
        newNode.setNext(self.head)
        self.tail.setNext(newNode)

def next(self):
    if self.head == None:
        return None
```

```
    else:
        self.tail = self.head
        self.head = self.head.getNext()
        return self.tail.getData()
```

Again each method is constant-time (that is, $\mathcal{O}(1)$) because it consists of a sequence of constant-time operations. In practice this second implementation will be somewhat faster than the first one, because the first one requires us to destroy and create `Node` objects on each call to `next()`.

2. The stack is at its deepest on three separate occasions; all of these are when the stack looks like

```
string
array
dict
array
dict
plist
```

3. With the addition of a `middle` pointer, the worst case for accessing an array element occurs when the index is just before the `middle` or just before the `tail`. In these cases, we have to scan about half of the linked list (starting at `head` or at `middle`, respectively) before getting to the array entry we wanted. So we have to traverse about $n/2$ nodes. This is about half as bad as the worst-case access time without the `middle` pointer, but it is still $\mathcal{O}(n)$. Halving the worst-case access time is a practical improvement, but the big-O notation reveals that it is not a qualitative improvement.

[Remark: Some students said that $n/2$ approaches n, as n goes to infinity. That is not true.]

4. [This is easier with a picture; I leave that to you.] The $\sqrt{n}$ pointers point to data nodes evenly spaced throughout the linked list of n data nodes, so each of these pointers is responsible for about $n/\sqrt{n} = \sqrt{n}$ of the data nodes. The worst case occurs when we have to access the second-to-last item in the array. Then we must traverse about $\sqrt{n}$ pointers, follow the data pointer into the list of data nodes, and traverse about $\sqrt{n}$ data nodes. In all there are at most $2\sqrt{n}$ steps, which is $\mathcal{O}(\sqrt{n})$.

[Remark: This is a vast improvement over the $\mathcal{O}(n)$ method we used earlier; for example, on an array of 1000000 elements it may be 1000 times as fast as the earlier method. On the other hand, it uses up some extra memory, but only $\mathcal{O}(\sqrt{n})$ extra memory; it doesn't even double the memory requirements of the data structure.]

5. There are two popular answers, depending on whether you work from the left or the right. Here is the one that works from the right.

```
def number(numeral):
    n = len(numeral)
    if n == 1:
        return num(numeral)
    else:
        return num(numeral[n - 1]) + 10 * number(numeral[0:n - 1])
```

At its deepest, the call stack for `number('32767')` looks like this:

```
num('3')
number('3')
number('32')
number('327')
number('3276')
number('32767')
```

Here is the one that works from the left. It's slightly less efficient, because it has to raise numbers to powers instead of just multiplying them by 10.

```
def number(numeral):
    n = len(numeral)
    if n == 1:
        return num(numeral)
    else:
        return num(numeral[0]) * 10**(n - 1) + number(numeral[1:])
```

At its deepest, the call stack for `number('32767')` looks like this:

```
num('7')
number('7')
number('67')
number('767')
number('2767')
number('32767')
```

6. The basic idea is to implement `Set` using a linked list with `head` and `tail` pointers. The inclusion of a tail pointer requires only minor modifications to the original methods of `Set`, and it does not qualitatively slow them down at all. The inclusion of the tail pointer lets us

implement `unite()` very rapidly — namely, $\mathcal{O}(1)$. Specifically, when one calls `a.unite(b)`, the `a` set gets the head and tail of the `b` set using the extra methods `getHead()` and `getTail()`, and then just updates a couple of pointers. I do not tell my user about the extra methods, because she should not have to know how `Set` is implemented and I do not want her to use them (in case I change the implementation later). They are private methods, existing solely to implement the public methods of the ADT.

```
def unite(self, set):
    """Absorbs the elements of the given set into the receiving set."""
    if self.tail == None:
        # This set is currently empty.
        self.head = set.getHead()
        self.tail = set.getTail()
    elif set.getHead() != None:
        # Neither set is empty; absorb the second set.
        self.tail.setNext(set.getHead())
        self.tail = set.getTail()

def getHead(self):
    """Private method. Returns the head Node of the underlying linked
    list, or None if the set is empty."""
    return self.head

def getTail(self):
    """Private method. Returns the tail Node of the underlying linked
    list, or None if the set is empty."""
    return self.tail
```

[Remark: Python lets you access another object's members directly — as in `set.head` instead of `set.getHead()` — but this is generally dangerous. For many objects to work correctly, they must keep the information stored in various members synchronized. In other words, changes to one member may require changes to other members. By directly accessing a member, you may make a change that puts the members out of sync. Do not do it.]