You have 70 minutes.

Show your work and explain all of your answers. Good work often earns partial credit. A correct answer with no explanation often earns little or no credit.

If you are asked to write code but you do not know the exact Python required, then try to write code that is approximately correct. If you think that your code does not demonstrate that you understand the solution, then describe your idea in English as well. Be precise enough that I cannot misinterpret your solution.

If you have no idea how to solve a problem, or if you have forgotten a key concept that you think you need to know, you may ask me for a hint. The hint will cost you some points (to be decided unilaterally by me as I grade your paper), but it may help you earn more points overall.

Good luck.

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.)

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

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.

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)
```

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.

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

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

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.