Name: ____________________________

Directions: Work only on this sheet (on both sides, if needed); do not turn in any supplementary sheets of paper. There is actually plenty of room for your answers, as long as you organize yourself BEFORE starting writing. Do not use any Python constructs which were not introduced either in lecture, discussion section or our written materials.

1. (20) Write a generator analog of the **cq** iterator class for "circular queues" in our PLN on iterators and generators. Use no more than four lines:

```
def cq(q):
   ------------------------------
   ------------------------------
   ------------------------------
   ------------------------------
```

2. (20) The class **atominc**, usable with the **thread** module, will store variables that can be atomically incremented without the user having to deal him/herself with locks. The value being stored is in the **val** member variable of the class. For example:

```
i = atominc()
...
i.inc()
```

would add 1 to **i.val** in an atomic manner.

```
print i.val
```

would print out the latest value stored (though of course it could be "old" by the time we get it).

We would still need to lock/unlock ourselves for operations other than incrementing. For instance, if we needed to perform some operations atomically if the stored value is 8, we'd write:

```
...
i.lock()
if i.val == 8:
...
i.unlock()
```

Fill in the gaps below.

```
class atominc():
   def __init__(self,initval):
      self.vallock = _______________
      self.val = initval
   def inc(self):
      ---------------
      ---------------
      ---------------
   def lock(self):
      ---------------
   def unlock(self):
      ---------------
```

3. (10) Consider the primes finder code in Sec. 5 of our PLN on threading. Show a single line of code which we could insert somewhere early in **main()** which would result in better load balancing.

4. (20) Consider the functions **ones()** and **ints()** presented in an example in discussion section. Suppose they are in the source file **s.py**, and we execute the following:

```
>>> from s import *
>>> i = ints()
>>> for j in i:
...    if j > 3: break
```

Then the execution of these statements will result in a total of ___________ iterators being created.

5. (20) The function **mrgitrs()** below takes as its argument a list of several iterators, each of which produces an ascending-order sorted sequence (finite or infinite). The function outputs the merge of them, as a generator. It is assumed that each iterator will return at least one item.

The variable **ins** will be such that **ins[2]**, for instance, will initially consist of [x,y,2], where y is **itrs[2]** and x is the first element of the sequence produced by y.

Note that the built-in Python function **min()** does work lexicographically where appropriate; e.g. **min([4,'abc'],[8,5],[3,200])** is [3,200].

Fill in the gaps. (In some cases it is possible to use fewer lines than allotted.)

```
def mrgitrs(itrs):
   tmp = [______________________________]
   ins = [______________________________]
   while ins != []:
      [val,itr,j] = min(ins)
      try:
         ______________________________
         ______________________________
      except ______________________________:
         ______________________________
      ______________________________
```

6. (10) Again consider the **cq** iterator class for "circular queues" in our PLN on iterators and generators. One problem with it is that if the input list is modified in code external to the class, the change won't be reflected in the class' version of the list. For example:

```
>>> import cq
>>> x = [5,12,8]
>>> c = cq.cq(x)
>>> c.next()
5
>>> c.next()
12
>>> c.next()
8
>>> c.next()
5
>>> x[1] = 'abc'
>>> x
[5, 'abc', 8]
>>> c.next()
12
>>> c.next()
8
```

Show how to remedy this problem by changing just one portion of one line in the original class.

Solutions:

1.

```
def cq(q):
   while True:
      q[0:] = q[1:] + [q[0]]
      yield q[-1]
```

2.

```
class atominc():
   def __init__(self,initval):
      self.vallock = thread_allocate_lock()
      self.val = initval
   def inc(self):
      self.vallock.acquire()
      self.val += 1
      self.vallock.release()
   def lock(self):
      self.vallock.acquire()
   def unlock(self):
      self.vallock.release()
```

3. For example,

```
setcheckinterval(5)
```

(Any value less than 10 is an acceptable answer.)

4. There are a total of 9 iterators created. One determines this simply by tracing through the recursion, and remembering that the functions are not actually called at the time the iterators are created; the call occurs when the **.next()** functions in the iterators are called.

5. The original intended solution was

```
def mrgitrs(itrs):
   tmp = [[i.next(),i] for i in itrs]
   ins = [tmp[j]+[j] for j in range(len(itrs))]
   while ins != []:
      [val,itr,j] = min(ins)
      try:
         v = itr.next()
         ins[j] = [v,itr,j]
      except StopIteration:
         del(ins[j])
      yield val
```

However, this does not work if there is a finite iterator and it is not the last element of **itrs**. Full credit was given if the student's code worked in the case in which all input interators are infinite.

6.

```
self.q[0:] = self.q[1:] + [self.q[0]]
```