In all code below, don’t worry about `#include` files and using declarations.

1) We want code to make it easy to define any kind of finite or infinite sequence. For example, the arithmetic sequence below would go up from 10 by 5’s until 100 is reached or passed, and the geometric one would do powers of 2. (10 pts total)

```cpp
int main ()
{
    ArithmeticSeq a(10, 100, 5);
    GeometricSeq g(1, 100, 2);
    cout << "Arithmetic: ";
    while (a.hasNext()) { cout << a.next() << " "; }
    cout << endl << "Geometric: ";
    while (g.hasNext()) { cout << g.next() << " "; }
}
```

The output should be

Arithmetic: 10 15 20 25 30 35 40 45 50 55 60 65 70 75 80 85 90 95
Geometric: 1 2 4 8 16 32 64

The code below makes it pretty easy to define a sequence, but it doesn’t compile, it doesn’t get exactly the right output values, and it calls `calcNext()` in both `hasNext()` and `next()` which will cause problems if `calcNext()` is expensive or has side-effects. First, mark and fix the syntactic bugs in this code. Then, separately, write new definitions for the relevant member functions so that the correct output is generated and `calcNext()` is called only when `next()` is called.

a) Mark and fix the syntactic errors. (5 pts)

```cpp
template <class T> class Sequence  // all sequences
{
public:
    Sequence(T b) : beg(b) {}  
    bool hasNext() { return true; }
    T next() { return beg = calcNext(); }
protected:
    virtual T calcNext() = 0; // pure virtual
private:
    T beg;
};
```
b) Write new definitions of the appropriate member functions above to fix the output values and avoid the two calls to calcNext(). (5 pts)

Two “bugs” to fix: first value in sequence not returned and calcNext() called twice. Two simple changes can do this, with no new data members required. In Sequence:

```cpp
virtual T next() { T v = beg; beg = calcNext(); return v; }
```

And in FiniteSequence:

```cpp
bool hasNext() { return beg < end; }
```

Now beg is always the next value to show, and calcNext() is called only in next().

Changes that required every subclass like ArithmeticSeq to do redundant work lost points.
2) Using STL containers and algorithms as much as possible, define

    print_unique(int a[], int len) to print the unique numbers in a[] in sorted
    order, e.g., if a = { 3, 1, 2, 1, 3, 5, 2 } it should print 1 2 3 5. Do not modify a[].
    Don’t worry about includes and using declarations. (6 pts)

The simplest approach is to copy into an STL set, because sets store only unique
elements. This is N log N. To use the STL unique algorithm, you need to copy and sort
first, which is more code. copy_if() can also work but it’s N-squared.

```cpp
void print_unique( int a[], int len )
{
    set<int> s( a, a + len );
    cout << "Unique items = ";
    copy( s.begin(), s.end(), ostream_iterator<int>( cout, " " ) );
    cout << endl;
}
```

3) Using STL containers and algorithms as much as possible, define get_mode(int

    a[], int len) to find the number in a[] that occurs most often, e.g., for a = { 3,
    1, 2, 1, 3, 5, 2, 1 } it should return 1. If more than one such number exists, it doesn’t
    matter which one is returned. Do not modify a[]. (8 pts)

We need to count the different values, then pick an element with the maximum count. A
map can store the counts in STL pairs. The STL max_element() algorithm can find the
pair with the greatest count (second element) if we pass it the appropriate predicate.
max_elements() will return a pointer to that pair so we just need to return it’s first
element (the key).

```cpp
bool pairGreater( const pair<int, int> &p1, const pair<int, int> &p2 )
{
    return p1.second < p2.second;
}

// undefined for empty arrays
int get_mode( int a[], int len )
{
    map<int, int> counts;

    for (int i = 0; i < len; ++i)
    {
        ++counts[ a[ i ] ];
    }

    return max_element( counts.begin(), counts.end(),
                        pairGreater )->first;
}
```
4) Your Lisp list code in the last assignment calls `new` to allocate memory for pairs and primitive data expressions but never releases this memory. Define `gc(const Exp *)` to take a pointer to any Lisp expression, delete the memory your functions allocated to build it, if any, and finally return the number of items deleted. (10 pts total)

a) First write a set of CPPUNIT assertions to test that `gc()` at least returns the right count. You shouldn’t need very many tests, but your tests should cover both typical and special cases. For each test, explain the reason for the return value you assert for `gc()`. Don’t include all the CPPUNIT boilerplate, just the assertions and any Lisp setup code you need. (4 pts)

```cpp
testGC()
{
    CPPUNIT_ASSERT_EQUAL(0, gc(nil)); // nothing allocated
    CPPUNIT_ASSERT_EQUAL(1, gc(cons(nil, nil))); // 1 pair
    CPPUNIT_ASSERT_EQUAL(2, gc(cons(1, nil))); // 1 pair, 1 data
    CPPUNIT_ASSERT_EQUAL(4, gc(cons(1, cons(2, nil)))); // a list
    CPPUNIT_ASSERT_EQUAL(5, gc(cons(cons(1, nil), cons(2, nil)))); // a pair w/data in car of list
}
```

b) Now define `gc()`. Note in a comment if `gc()` needs to be made a friend of any internal classes. (6 pts)

Uses just public API so no need to be a class friend.

```cpp
const int gc(const Exp *p)
{
    if (p == nil)
    {
        return 0;
    }
    else if (isAtom(p))
    {
        delete p;
        return 1;
    }
    else
    {
        int n = gc(car(p)) + gc(cdr(p));
        delete p;
        return n + 1;
    }
}
```

Comment: Even if your code had a NIL object, as opposed to using the NULL pointer, nil should be unique so that == nil works, which means that it should never be deleted.

Comment: The counts from the car and cdr need to be included.

Comment: Don’t forget the pair!
5) Define a simple version of the STL merge algorithm. `merge(b1, e1, b2, e2, o)` takes 2 begin/end iterators (b1/e1 and b2/e2) pointing to already sorted data, and sends the sorted merger of the data to an output iterator o. E.g., if a is [ 0, 4, 4, 8, 10 ] and b is [ 1, 2, 5, 5, 11, 12, 15 ], then `merge(a, a + 5, b, b + 7, ostream_iterator<int>(cout, " "))` will print 0 1 2 4 4 5 5 8 10 11 12 15.

Merging sorted data is fairly simple. No intermediate containers or calls to `sort()` are needed. Here’s the start of the definition: (8 pts)

```cpp
template <class Iter1, class Iter2, class OIter>
OIter merge( Iter1 b1, Iter1 e1, Iter2 b2, Iter2 e2, OIter o )
{
    while ( b1 < e1 && b2 < e2 )
    {
        if ( *b1 < *b2 )
            *o = *b1;
        ++b1;
        else
        {
            *o = *b2;
            ++b2;
        }
        ++o;
    }
    copy( b1, e1, o );
    copy( b2, e2, o );
    return o;
}
```

The algorithm is to simply pick the smallest at the front until one or the other range runs out. You don’t know which ran out, but copy will test for empty so it’s easiest to just copy both to the end. Don’t forget to return the output iterator!

Recursion in C++ is a poor choice. The stack would overflow on very long ranges.

Functions like `push_back()` are for containers, not iterators.