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.

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

```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;
};
```
class FiniteSequence : public Sequence // all finite sequences
{
public:
    FiniteSequence(T b, T e) : Sequence(b), end(e) {}
    bool hasNext() { return calcNext() < end; }
private:
    T end;
};

class ArithmeticSeq : public FiniteSequence
{
public:
    ArithmeticSeq(int b, int e, int i) : FiniteSequence(b, e),
    inc(i) {}}
protected:
    int calcNext() { return beg + inc; }
private:
    int inc;
};

class GeometricSeq : public FiniteSequence
{
public:
    GeometricSeq(int b, int e, int i) : FiniteSequence(b, e),
    inc(i) {}}
protected:
    int calcNext() { return beg * inc; }
private:
    int inc;
};

b) Write new definitions of the appropriate member functions above to fix the output
values and avoid the two calls to calcNext().
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.
```

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[]`. 
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 \texttt{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.

a) First write a set of CPPUNIT assertions to test that \texttt{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 \texttt{gc()} . Don’t include all the CPPUNIT boilerplate, just the assertions and any Lisp setup code you need.

b) Now define \texttt{gc()} . \textbf{Note in a comment} if \texttt{gc()} needs to be made a friend of any internal classes.
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:

```cpp
template <class Iter1, class Iter2, class OIter>
OIter merge( Iter1 b1, Iter1 e1, Iter2 b2, Iter2 e2, OIter o )
```