Assignments are due on the due date before class and have to include this cover page. Plagiarism in assignment answers will not be tolerated. By submitting their answers to this assignment, the authors named above declare that its content is their original work and that they did not use any sources for its preparation other than the class notes, the textbook, and ones explicitly acknowledged in the answers. Any suspected act of plagiarism will be reported to the Faculty’s Academic Integrity Officer and possibly to the Senate Discipline Committee. The penalty for academic dishonesty may range from failing the course to expulsion from the university, in accordance with Dalhousie University’s regulations regarding academic integrity.
**Question 1 (10 marks)** On the slides, I showed you the Python code for generating the sequence of all strings of length at most \( \text{len} \) over the alphabet \{a, b, c, d\} in lexicographically sorted order. The implementation of this iterator relied on Python's support for coroutines. Provide C++ or Java code that implements an iterator object that allows you to iterate over the same sequence of strings. Note that both Java and C++ (via add-on libraries) provide support for threads, which are more general than coroutines and, thus, can be used to simulate coroutines. Do not use these facilities in your implementation but provide an implementation of a single-threaded iterator object. The detailed requirements are as follows.

If you provide C++ code,

- You should provide a class `Lexy` such that `new Lexy(len)` creates an object that represents the lexicographically sorted sequence of all strings over the above alphabet of length at most `len`. The created object should have constant size, independent of `len`. In particular, it must not store an explicit representation of the entire sequence of strings.

- The class should provide methods `begin()` and `end()`. Both return an iterator of type `Lexy::iterator`. The iterator returned by `begin()` points to the first element in the sequence of strings, the one returned by `end()` points beyond the end of the sequence of strings.

- The iterator class itself should provide a dereferencing operator and an increment operator.

With these operations, the equivalent of the Python code on the slides

```cpp
for w in lexy(3):
    print w
```

becomes the standard C++ iteration idiom

```cpp
Lexy l(3);
for (Lexy::iterator it = l.begin(); it != l.end(); ++it)
    cout << *it << endl;
```

Similarly, if you provide Java code,

- You should provide a class `Lexy` such that `new Lexy(len)` creates an object that represents the lexicographically sorted sequence of all strings over the above alphabet of length at most `len`. The created object should have constant size, independent of `len`. In particular, it must not store an explicit representation of the entire sequence of strings.

- The class should provide a method `elements()`, which returns an object of type `LexyEnumeration` that implements the `Enumeration` interface. As such `LexyEnumeration` needs to provide methods `hasMoreElements()` and `nextElement()`, which test whether there are more elements in the sequence to be traversed and return the next element in the sequence if there are any, respectively.

With these operations, the equivalent of the Python code on the slides

```java
for w in lexy(3):
    print w
```

becomes the standard Java iteration idiom

```java
Lexy l = new Lexy(3);
Enumeration<String> e = l.elements();
while (e.hasMoreElements())
    System.out.println(e.nextElement());
```
**Question 2 (10 marks)** In class, we discussed the relationship between recursion and iteration and I pointed out that recursive procedures cannot easily be translated into iterative ones, while the translation in the other direction is straightforward. It is instructive to try to carry out the former translation to learn how much effort is involved. For this question, we consider the standard merge sort algorithm. The following is a C implementation for sorting an array of integers:

```c
int mergesort(int *a, int n)
{
    int *b, *c;
    if (b = malloc(n * sizeof(int))) {
        c = recurse(a, b, n);
        if (c == b) memcpy(a, b, n * sizeof(int));
        free(b);
        return 0;
    }
    else return -1;
}
```

```c
int *recurse(int *a, int *b, int n)
{
    int i, j, k, m, *cl, *cr, *d;
    if (n < 2) return a;
    m = n/2;
    cl = recurse(a, b, m);
    cr = recurse(a + m, b + m, n - m);
    if (cr != cl + m) memcpy(cl + m, cr, (n - m) * sizeof(int));
    d = cl == a ? b : a;
    for (i = k = 0, j = m; i < m && j < n;)
        if (cl[j] < cl[i]) d[k++] = cl[j++];
        else d[k++] = cl[i++];
    if (i == m) memcpy(d + k, cl + j, n - j);
    else memcpy(d + k, cl + i, m - i);
    return d;
}
```

Provide the C code of an iterative implementation of this algorithm. Specifically, the implementation you provide should execute the exact same element comparisons and data moves as the algorithm above, but at any given point in time, its recursion depth (the number of stack frames on the runtime stack) has to be bounded by a constant independent of the input size n.