Assignment 5

Question 1: Merge sort algorithm with an array

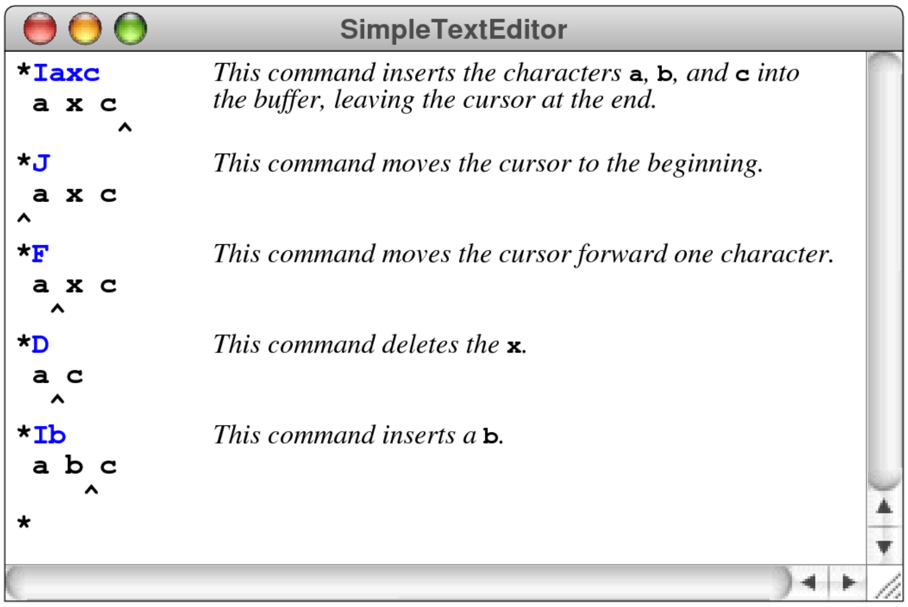
Rewrite the implementation of the merge sort algorithm from Figure 10-3 (Page 446) so that it sorts an array rather than a vector. Your function should use the prototypes:

void sort(int array[], int n)
void merge(int array[], int n, int array1[], int n1, int array2[], int n2)

Do not use other sort algorithms! Do not modify the header file and the implementation should be in the .cpp file. We will check your implementation and make sure it uses exactly the merge and sort strategy.

Question 2: EditorBuffer using double linked list

Implement the **EditorBuffer** class using the strategy described in the section entitled "Doubly linked lists" (Page 606). Be sure to test your implementation as thoroughly as you can. In particular, make sure that you can move the cursor in both directions across parts of the buffer where you have recently made insertions and deletions.



```
*Iaxc          This command inserts the characters a, b, and c into
 a x c         the buffer, leaving the cursor at the end.
      ^
*J             This command moves the cursor to the beginning.
 a x c
^
*F             This command moves the cursor forward one character.
 a x c
  ^
*D             This command deletes the x.
 a c
  ^
*Ib            This command inserts a b.
 a b c
    ^
*
```

In this implementation, the ends of the linked list are joined to form a ring, with the dummy cell at both the beginning and the end. This representation makes it possible to implement the **moveCursorToEnd** method in constant time, and reduces the number of special cases in the code. The constructor is already given. Five methods need to be implemented:

- The destructor that delete all cells
- A **insertCharacter** method that inserts one character into the buffer on the **cursor**
- A **deleteCharacter** method that deletes one character after the **cursor**
- A **getText** method that returns the content in buffer
- A **getCursor** method that returns the index of the **cursor**

Do not modify the header file and the implementation should be in the .cpp file. We will check all these methods' implementation.

Question 3: Priority queue implementation using queue

For some programming applications, it is useful to extend the simple queue abstraction into a **priority queue**, in which the order of the items is determined by a numeric priority value. When an item is enqueued in a priority queue, it is inserted in the list ahead of any lower priority items. If two items in a queue have the same priority, they are processed in the standard first-in/first-out order. Using the linked-list implementation of queues as a model, design and implement a **pqueue.h** interface that exports a class called **PriorityQueue**, which exports the same methods as the traditional **Queue** class with the exception of the **enqueue** method, which now takes an additional argument, as follows:

void enqueue(ValueType value, double priority)

The parameter **value** is the same as for the traditional versions of **enqueue**; the **priority** argument is a numeric value representing the priority. As in conventional English usage, smaller integers correspond to higher priorities, so that priority 1 comes before priority 2, and so forth. Besides the **enqueue** method, three more methods need to be implemented:

- A **dequeue** method that pops the value on the top.
- A **peek** method that returns the value on the top.
- A **deepcopy** method that copies the data from another **priority queue**.

C++ requires that the implementation for a template class be available to the compiler whenever that type is used. The effect of this restriction is that header files must include the implementation. So there is no .cpp file and you need to write the implementation in the header file.

Question 4: BigInt

On newer machines, the data type **long** is stored using 64 bits, which means that the largest positive value of type **long** is 9,223,372,036,854,775,807 or $2^{63} - 1$. While this number seems enormous, there are applications that require even larger integers. For example, if you were asked to compute the number of possible arrangements for a deck of 52 cards, you would need to calculate 52!, which works out to be

80658175170943878571660636856403766975289505440883277824000000000000

If you are solving problems involving integer values on this scale (which come up often in cryptography, for example), you need a software package that provides **extended-precision arithmetic,** in which integers are represented in a form that allows them to grow dynamically. Although there are more efficient techniques for doing so, one strategy for implementing extended-precision arithmetic is to store the individual digits in a linked list. In such representations, it is conventional—mostly because doing so makes the arithmetic operators easier to implement—to arrange the list so that the units digit comes first, followed by the tens digit, then the hundreds digit, and so on. Thus, to represent the number 1729 as a linked list, you would arrange the cells in the following order: Design and implement a class called **BigInt** that uses this representation to implement extended-precision arithmetic, at least for nonnegative values.

At a minimum, your **BigInt** class should support the following operations:

- A constructor that creates a **BigInt** from an **int** or from a **string** of digits.
- The destructor that delete all cells.
- A **toString** method that converts a **BigInt** to a **string**.
- The operators **+** and ***** for addition and multiplication, respectively.

You can implement the arithmetic operators by simulating what you do if you perform these calculations by hand. Addition, for example, requires you to keep track of the carries from one digit position to the next. Multiplication is trickier, but is still straightforward to implement if you find the right recursive decomposition. Your **BigInt** class should be able to calculate the value of *n*! for all values of *n* between 0 to 52, inclusive.

You can modify both the header file and .cpp file. For the implementation of constructor, we only consider nonnegative integers and normal strings. Illegal inputs like "0033" or "=32" are not in our test.*