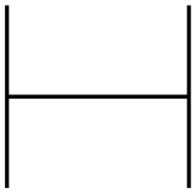
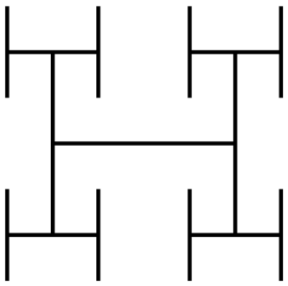
CSC 3002 (Fall 2021) **Assignment 4**

Problem 1 (Exercise 8.16)

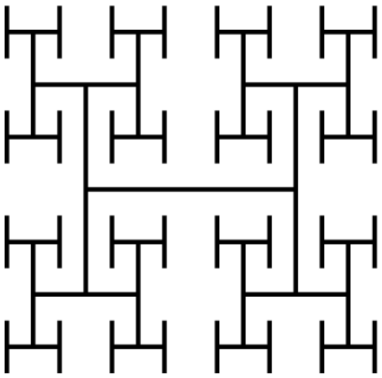
If you search the web for fractal designs, you will find many intricate wonders beyond the Koch snowflake illustrated in this chapter. **H-fractal**, in which the repeated pattern is shaped like an elongated letter **H** in which the horizontal bar and vertical lines on the sides have the same length. Thus, the order-0 Hfractal looks like this:



To create the order-1 fractal, all you do is add four new H-fractalseach one half of the original sizeat each open end of the order-0 fractal, like this:



To create the order-2 fractal, all you have to do is add even smaller H-fractals (again half the size of the fractal to which they connect) to each of the open endpoints. This process gives rise to the following order-2 fractal:



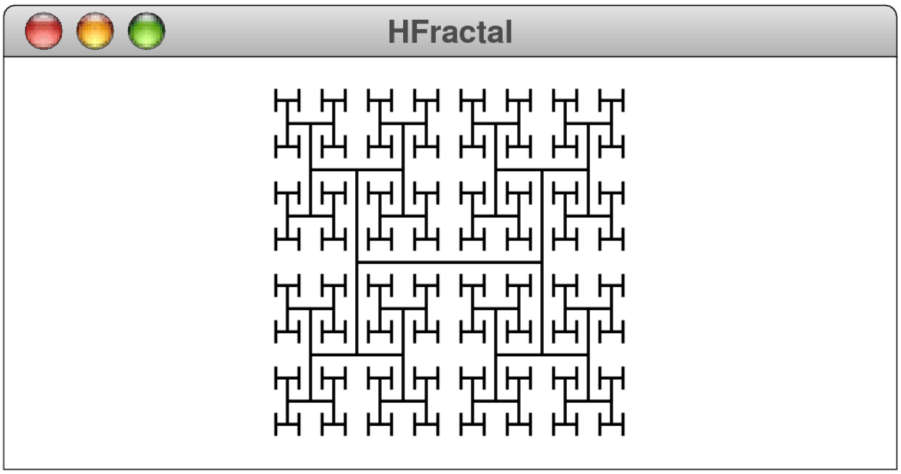
Write a recursive function

```
drawHFractal(GWindow & gw, double x, double y, double size, int order);
```

where `x` and `y` are the coordinates of the center of the H-fractal, size specifies the width and the height, and `order` indicates the order of the fractal. As an example, the main program

```
int main() {
    GWindow gw;
    double xc = gw.getWidth() / 2;
    double yc = gw.getHeight() / 2;
    drawHFractal(gw, xc, yc, 100, 3);
    return 0;
}
```

would draw an order-3 H-fractal at the center of the graphics window, like this:

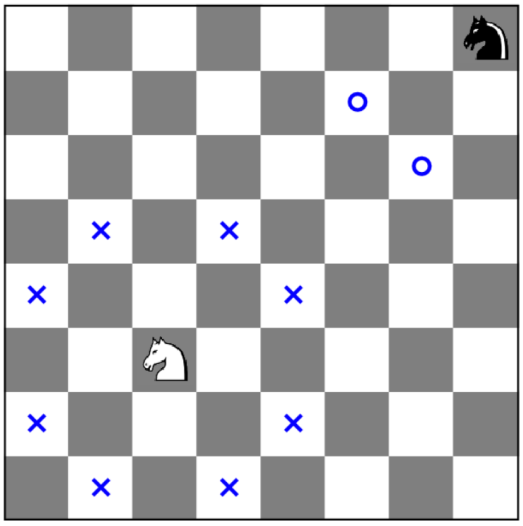


Requirments & Hints:

Please fill in the **TODO** part of **drawHFractal** function in *HFractal.cpp*. You can define your own functions in the codes if necessary, but you need to follow the provided code framework.

Problem 2 (Exercise 9.6)

In chess, a knight moves in an L-shaped pattern: two squares in one direction horizontally or vertically, and then one square at right angles to that motion. For example, the white knight in the upper right side of the following diagram can move to any of the eight squares marked with an ×:



The mobility of a knight decreases near the edge of the board, as illustrated by the black knight in the corner, which can reach only the two squares marked with an ◯.

It turns out that a knight can visit all 64 squares on a chessboard without ever moving to the same square twice. A path for the knight that moves through all the squares without repeating a square is called a knights tour. One such tour is shown in the following diagram, in which the numbers in the squares indicate the order in which they were visited:

52	47	56	45	54	5	22	13
57	44	53	4	23	14	25	6
48	51	46	55	26	21	12	15
43	58	3	50	41	24	7	20
36	49	42	27	62	11	16	29
59	2	37	40	33	28	19	8
38	35	32	61	10	63	30	17
1	60	39	34	31	18	9	64

Write a program that uses backtracking recursion to find a knights tour.

Requirments & Hints:

Please fill in the **TODO** part of **findKnightsTour** function in *KnightsTour.cpp*. You can define your own functions in the codes if necessary, but you need to follow the provided code framework.

Problem 3 (Exercise 11.7)

Rewrite the implementation of the merge sort algorithm from Figure 10-3 (shown as follows) so that it sorts an array rather than a vector.

FIGURE 10-3 Implementation of the merge sort algorithm

```
/*
 * Function: sort
 * --------------
 * This function sorts the elements of the vector into increasing order
 * using the merge sort algorithm, which consists of the following steps:
 *
 * 1. Divide the vector into two halves.
 * 2. Sort each of these smaller vectors recursively.
 * 3. Merge the two vectors back into the original one.
 */

void sort(Vector<int> & vec) {
   int n = vec.size();
   if (n <= 1) return;
   Vector<int> v1;
   Vector<int> v2;
   for (int i = 0; i < n; i++) {
      if (i < n / 2) {
         v1.add(vec[i]);
      } else {
         v2.add(vec[i]);
      }
   }
   sort(v1);
   sort(v2);
   vec.clear();
   merge(vec, v1, v2);
}

/*
 * Function: merge
 * ---------------
 * This function merges two sorted vectors, v1 and v2, into the vector
 * vec, which should be empty before this operation.  Because the input
 * vectors are sorted, the implementation can always select the first
 * unused element in one of the input vectors to fill the next position.
 */

void merge(Vector<int> & vec, Vector<int> & v1, Vector<int> & v2) {
   int n1 = v1.size();
   int n2 = v2.size();
   int p1 = 0;
   int p2 = 0;
   while (p1 < n1 && p2 < n2) {
      if (v1[p1] < v2[p2]) {
         vec.add(v1[p1++]);
      } else {
         vec.add(v2[p2++]);
      }
   }
   while (p1 < n1) vec.add(v1[p1++]);
   while (p2 < n2) vec.add(v2[p2++]);
}
```

As in the reimplementation of the selection sort algorithm on page 499 (shown as follows),

```
void sort(int array[], int n) {
    for (int lh = 0; lh < n; lh++) {
        int rh = lh;
        for (int i = lh + 1; i < n; i++) {
            if (array[i] < array[rh]) rh = i;
        }
        swap(array[lh], array[rh]);
    }
}
```

your function should use the prototype

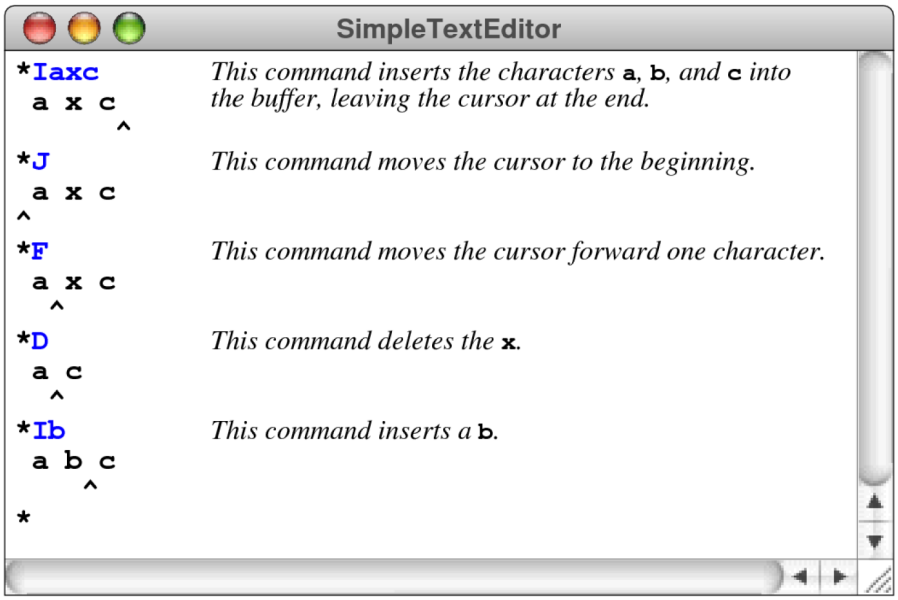
```
void sort(int array[], int n)
```

Requirments & Hints:

Please fill in the **TODO** part of **sort** function in *MergeSort.cpp*. You can define your own functions in the codes if necessary, but you need to follow the provided code framework.

Problem 4 (Exercise 13.12)

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. Simple testing results are shown as follows.



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. Methods need to be implemented:

- The destructor that delete all cells;
- Methods move the cursor;
- 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.

Implement the **EditorBuffer** class using this representation (which is, in fact, the design strategy used in many editors today). Make sure that your program continues to have the same computational efficiency as the two-stack implementation in the text and that the buffer space expands dynamically as needed.

Requirments & Hints:

Please fill in the **TODO** part of the methods in *buffer.cpp*. You can define your own functions in the codes if necessary, but you need to follow the provided code framework.

Requirements for Assignment

I've provided a project named as *Assignment4.pro*. You should write **TODO** part in each cpp file according to the problem requirements. The test file is provided under *src* folder named *main.cpp*. Please pack your whole project files into a single .zip file, name it using your student ID (e.g. if your student ID is 123456, hereby the file should be named as 123456. zip), and then submit the .zip file via BB system.

Please note that, the teaching assistant may ask you to explain the meaning of your program, to ensure that the codes are indeed written by yourself. Please also note that we may check whether your program is too similar to your fellow students' code using BB.

Reminder: For windows users, please switch you input language to English before interacting in Stanford console. Or, you will get no response.