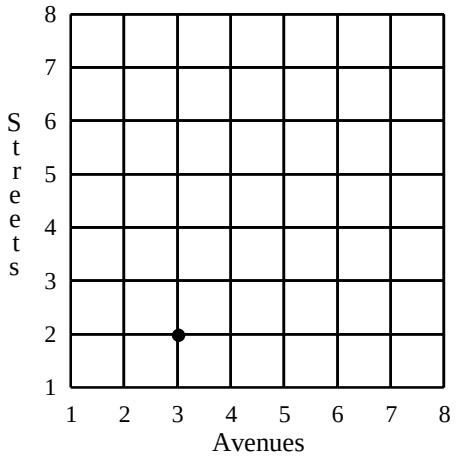
Section Handout #2

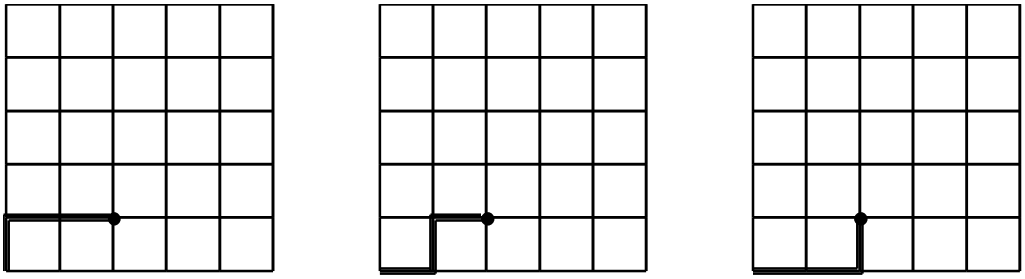
Problem 1. Counting paths

Note: Although Karel's world is used as the setting for this problem, you don't have to know anything about Karel programming to solve it.

As most of you know, Karel the Robot lives in a world composed of streets and avenues laid out in a regular rectangular grid that looks like this:



Suppose that Karel is sitting on the intersection of 2nd Street and 3rd Avenue—the intersection marked by the dot in the diagram—and wants to get back to the origin at 1st Street and 1st Avenue. Even if Karel wants to avoid going out of the way, there are still several equally short paths. For example, in this diagram there are three possible routes, as follows:



Your job in this problem is to write a recursive function

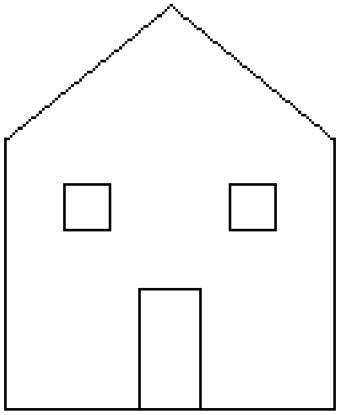
```
int CountPaths(int street, int avenue)
```

that returns the number of paths Karel could take back to the origin from the specified starting position, subject to the condition that Karel always wants to take the shortest possible routes and can therefore only move west or south (left or down in the diagram). If you tried to solve this problem with the maze-like try all directions method, the function would infinitely recurse, since Karel can go infinitely out of his way before reaching the final destination.

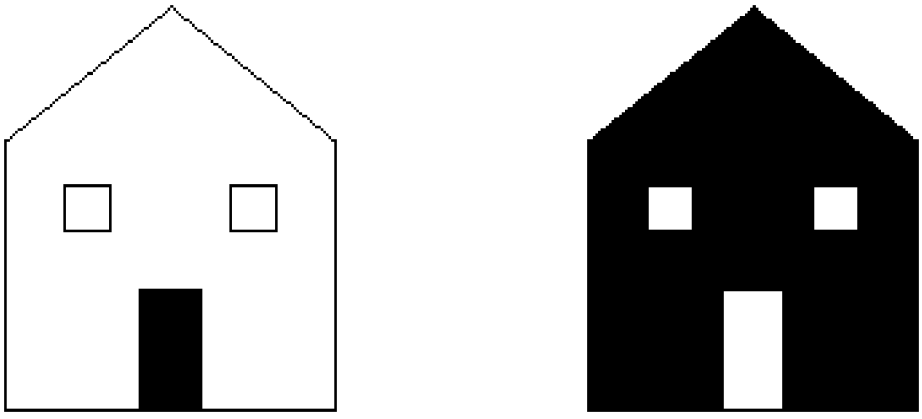
Problem 2: Filling A Region (Chapter 6, exercise 6, page 276-278)

Most drawing programs for personal computers make it possible to fill an enclosed region on the screen with a solid color. Typically, you invoke this operation by selecting a paint-bucket tool and then clicking the mouse, with the cursor somewhere in your drawing. When you do, the paint spreads to every part of the picture it can reach without going through a line.

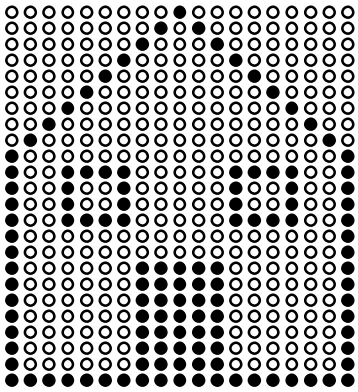
For example, suppose you have just drawn the following picture of a house:

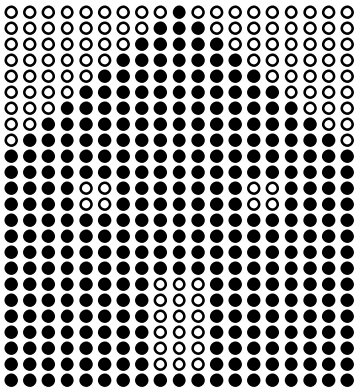


If you select the paint bucket and click inside the door, the drawing program fills the area bounded by the door frame as shown at the left side of the following diagram. If you instead click somewhere on the front wall of the house, the program fills the entire wall space except for the windows and doors, as shown on the right:



In order to understand how this process works, it is important to understand that the screen of the computer is actually broken down into an array of tiny dots called **pixels.** On a monochrome display, pixels can be either white or black. The paint-fill operation consists of painting black the starting pixel (i.e., the pixel you click while using the paint-bucket tool) along with any pixels connected to that starting point by an unbroken chain of white pixels. Thus, the patterns of pixels on the screen representing the preceding two diagrams would look like this:





Write a program that simulates the operation of the paint-bucket tool. To simplify the problem, assume that you have access to the enumerated type

```
typedef enum { White, Black } pixelStateT;
```

The type `pointT` is defined in Chapter 6 as follows:

```
typedef struct{
     int x, y;
} pointT;
```

Assume you also have the following functions:

```
pixelStateT GetPixelState(pointT pt);
void SetPixelState(pointT pt, pixelStateT state);
bool OutsidePixelBounds(pointT pt);
```

The first function is used to return the state of any pixel, given its coordinates in the pixel array. The second function allows you to change the state of any pixel to a new value. The third makes it possible to determine whether a particular coordinate is outside the pixel array altogether, so that the recursion can stop at the edges of the screen.

Your task in this problem is therefore to write a function

```
static void FillRegion(pointT pt);
```

that fills black into all white pixels reachable from the point `pt`.

Problem 3: Big-O Notation

a) (Chapter 7, Review Question 12) What is the computational complexity of the following function:

```
int Mystery1 (int n)
{
       int i, j, sum;

       sum = 0;
       for (i = 0;i < n; i++) {
              for (j = 0; j < i; j++){
                     sum += i * j;
              }
       }
       return (sum);
}
```

b) (Chapter 7, Review Question 12) What is the computational complexity of this function:

```
int Mystery2 (int n)
{
       int i, j, sum;

       sum = 0;
       for (i = 0;i < 10; i++) {
              for (j = 0; j < i; j++){
                     sum += j * n;
              }
       }
       return (sum);
}
```

c) What is the computational complexity of this function:

```
int Mystery3 ( int n )
{
   if( n <= 1 ) return 1;
   return (Mystery3( n / 2 ) + 1);
}
```

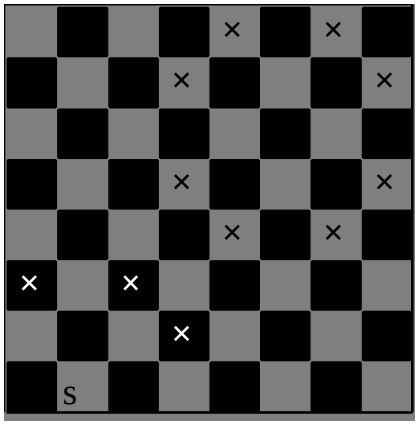
d) What is the computational complexity of this function:

```
int Mystery4 ( int n )
{
   int i, sum = 0;
   if( n <= 0 ) return 1;
   for( i = 0; i < n; i++ ) {
          sum += i;
   }
   return ( Mystery4( n - 1 ) + sum );
}
```

More Recursion for your personal enjoyment . . .

Problem 4: Knights Tour (Chapter 6, exercise 7, page 278)

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 black knight in the following diagram can move to any of the eight squares marked with a black cross:



The mobility of the knight decreases toward the edges of the board, as illustrated by the position of the white knight, which can move only to the three squares marked by white crosses.

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 **knight's 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 knight's tour.