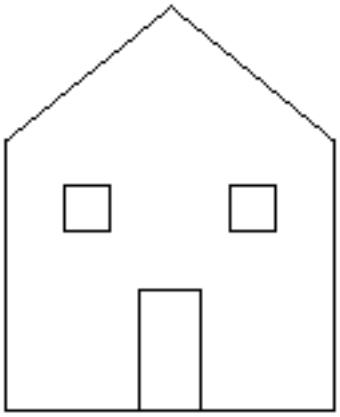
Section Handout #4

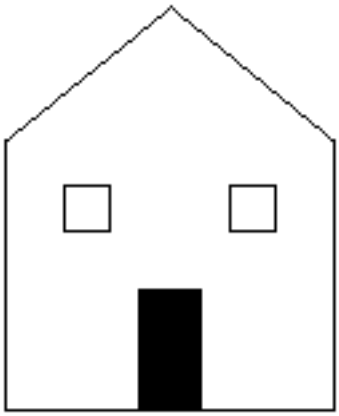
Problem 1: Filling a Region

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:

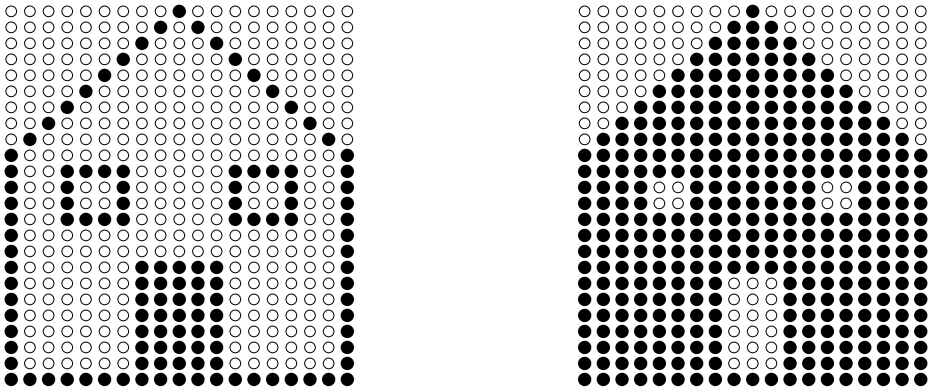


If you select the paint bucket and click inside the door, the drawing program fills the area bounded by the door frame as shown at the left side of the following diagram. If you instead click somewhere on the front wall of the house, the program fills the entire wall space except for the windows and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



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

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

The type **pointT** is defined as follows:

```
struct pointT {
     int row, col;
};
```

A **Grid<pixelStateT>** will be used to represent the screen.

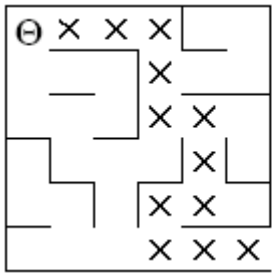
Your task in this problem is therefore to write a function

```
void FillRegion(pointT pt, Grid<pixelStateT> &screen);
```

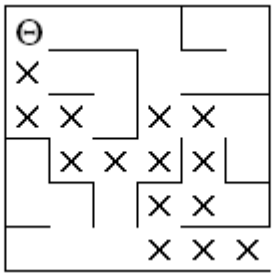
that fills black into all white pixels reachable from the point **pt**.

Problem 2: Shortest Path Through a Maze

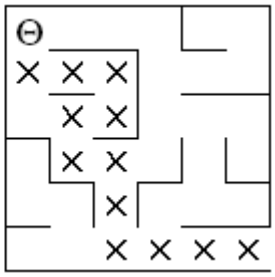
For the second assignment, you generated and solve perfect mazes. However, not all mazes are perfect. In a perfect maze there is exactly one path from each point to each other point. However, in many mazes, there are multiple paths. For example, the diagrams below show three solutions for the same maze:



length = 13

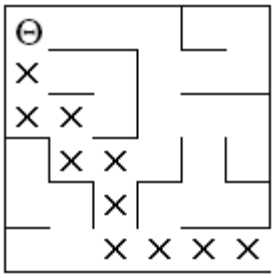


length = 15



length = 13

None of these solutions, however, is optimal. The shortest path through the maze has a path length of 11:



As a starting point, begin by considering this general maze solution:

```
/*
* Function: SolveMaze
* Usage: if (SolveMaze(pt)) . . .
* ------------------------------
* This function attempts to generate a solution to the current
* maze from point pt. SolveMaze returns true if the maze has
* a solution and false otherwise. The implementation uses
* recursion to solve the sub-mazes that result from marking the
* current square and moving one step along each open passage.
*/
bool SolveMaze(pointT pt)
{
  if (OutsideMaze(pt))
    return true;
  if (IsMarked(pt))
    return false;
  MarkSquare(pt);
  for (directionT dir = North; dir <= West; dir=directionT(dir + 1))
  {
    if (!WallExists(pt, dir) && SolveMaze(AdjacentPoint(pt, dir)))
    {
      return true;
    }
  }

  UnmarkSquare(pt);
  return false;
}
```

Write a function

```
int ShortestPathLength(pointT pt);
```

that returns the length of the shortest path in the maze from the specified position to any exit. If there is no solution to the maze, **`ShortestPathLength`** should return the constant **`NoSolution`**, which is defined to have a value larger than the maximum permissible path length, as follows:

```
static const int NoSolution = 10000;
```

Note that the for this problem you will use the Maze library on p.6-6 of the reader, rather than the Maze library from Assignment 2.

Problem 3: Pointers

Suppose you were writing a database for Stanford' library sytem. You have a **`bookT`** object which contains all of the information about a book. It looks like this:

```
struct bookT {
      string author;
      string title;
      string publisher
      (etc.)
};
```

You want people to be able to look up books using different features, such as author or title. To do this, you've created several maps:

```
Map<Vector<bookT> > byAuthor;
Map<Vector<bookT> > byTitle;
Map<Vector<bookT> > byPublisher;
...
```

Draw what this data structure looks like in memory. What happens if you discover that one of the books has the wrong author listed? What needs to be updated? How might you modify your data structure to help resolve this issue? Draw what this new data structure looks like in memory.

Problem 4: Linked List Warmup

Write the following linked list functions. Provide both iterative and a recursive forumlations. Cell is defined as follows:

```
struct Cell {
      Cell *next;
      int value;
};
```

a) Write a function ConvertToList which takes in a Vector of ints and converts it into a linked list. Assume the Vector has at least one element in it.

```
Cell * ConvertToList(Vector<int> vector)
```

b) Write a function which sums the values of a linked list.

```
int SumList(Cell *list)
```

Problem 5: Linked List Trace

You are given the following function below:

```
void PopRocks(Cell * & mikey)
{
  Cell *ptr;

  for (ptr = mikey; ptr->next != NULL; ptr = ptr->next)
  {
    /* Note: loop body intentionally left empty */
  }
  ptr->next = mikey;
  ptr = ptr->next;
  mikey = mikey->next;
  ptr->next = NULL;
}
```

Given a pointer to a linked list, what does the code do to the list? You can assume that the list is properly NULL terminated. Given the following linked lists, if we were to call the function passing in a pointer to the first item in the list, what would the lists look like afterwards?

```
   "15" -> "30" -> "45" -> "60"

   "s" -> "t" -> "a" -> "r"

"Go" -> "hang" -> "a" -> "salami," -> "I'm" -> "a" -> "lasagna" ->
"hog!"
```

Problem 6: Append

Write a function that given two lists will append the second list onto the first. For example, given the first list `(1 4 6)` and the second list `(3 19 2)`, the function would destructively modify the first list to contain `(1 4 6 3 19 2)`. It is easiest to write this function recursively. Be sure to handle the case when one or both lists are empty.\