Section Solutions #2

Problem 1. Counting paths

The recursive insight in this problem is that the number of paths—assuming that paths only move west and south—from any point in the grid to the origin is the sum of the number of paths from the two adjacent intersections. The simple case is that there is only one path from any point along the western and southern edge of the grid. The code for the problem can therefore be written as follows:

```
/*
 * Function: CountPaths
 * Usage: paths = CountPaths(street, avenue);
 * ------------------------------------------
 * This function computes the number of distinct paths from the
 * specified intersection to the origin, assuming that all motion
 * goes either west or south.
 */

int CountPaths(int street, int avenue)
{
    if (street == 1 || avenue == 1) {
        return (1);
    } else {
        return (CountPaths(street - 1, avenue)
                + CountPaths(street, avenue - 1));
    }
}
```

Problem 2: Filling a region (Chapter 6, exercise 6, page 276-278)

```
/*
 * Function: FillRegion
 * Usage: FillRegion(pt);
 * ----------------------
 * This function paints black pixels everywhere inside the
 * region surrounding the point.
 */

void FillRegion(pointT pt)
{
    if (OutsidePixelBounds(pt)) return;
    if (GetPixelState(pt) == Black) return;
    SetPixelState(pt, Black);
    FillRegion(AdjacentPoint(pt, North));
    FillRegion(AdjacentPoint(pt, East));
    FillRegion(AdjacentPoint(pt, South));
    FillRegion(AdjacentPoint(pt, West));
}
/* The type pointT and AdjacentPoint are defined in Chapter 6 */
```

Problem 3: Big-O Notation

a) O(n^2). The outer loop will run n times. Each time through the outer loop, the inner loop will run i times, where i runs from 0 to n-1. A constant amount of work is done in the body of the inner loop. This leads to the series: 0 + 1 + 2 + ... n-1 which, as we know, equals (n-1)(n-2)/2. Keeping only the highest order term and throwing away any constant factors, we arrive at our answer of O(n2) computational complexity.

b) O(1). The outer loop executes 10 times, the inner loop i times where i runs from 0 to 9, so there are ~100 multiply/add operations, but it does that same amount of work for any value of n. Thus the computational complexity is *constant* with respect to n. The constant "1" in O(1) signifies this. Constant time doesn't necessarily mean that a function is trivial or computes its result instantly, but the function always does the same amount of work, regardless of the inputs.

c) O(log n). Each recursive call divides the values of n by 2, so if we double the value of n we will only increase the number of recursive calls by 1. Note the similarity in structure between this function and binary search (which is also O(log n).)

d) O(n^2). We will make a total of n recursive calls before we reach the simple case. During each recursive call, the loop to calculate the sum will execute n times. Counting the total number of times the line within the for loop is executed gives us the following formula:

$$n + (n - 1) + (n - 2) + ... + 1$$

This simplifies to n * (n + 1) / 2, which indicates that the function is O(n^2). Expanding n * (n + 1) / 2 gives us n^2 / 2 + n / 2. In “Big-O” notation we ignore constant factors and lower-order terms; this means that we can ignore the n / 2 term as well as the factor of 1/2 in the n^2 term.

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

```
/*
 * File: knight.c
 * --------------
 * This program finds and displays a knight's tour, which is
 * defined to be a path in which a knight visits every square
 * on a chessboard without ever visiting the same square twice.
 * The knight moves in an L-shaped pattern, moving two squares
 * horizontally or vertically and then one square at right
 * angles to the original motion.
 */

#include <stdio.h>
#include "genlib.h"

/*
 * Constants
 * ---------
 * N -- Size of each dimension of the chessboard
 */

#define N 8
```

```
/* Private function prototypes */

static void InitBoard(int board[N][N]);
static void DisplayBoard(int board[N][N]);
static bool FindTour(int board[N][N], int x, int y, int turn);
static bool OffBoard(int x, int y);

/* Main program */

main()
{
    int board[N][N];

    InitBoard(board);
    if (FindTour(board, 0, 0, 1)) {
        DisplayBoard(board);
    } else {
        printf("No tour exists for this board\n");
    }
}

/*
 * Function: InitBoard
 * Usage: InitBoard(board);
 * ------------------------
 * This function initializes the board so that all of its
 * squares contain 0.  The value of 0 in a square is used
 * to indicate that the square has not been visited.
 */

static void InitBoard(int board[N][N])
{
    int x, y;

    for (x = 0; x < N; x++) {
        for (y = 0; y < N; y++) {
            board[x][y] = 0;
        }
    }
}

/*
 * Function: DisplayBoard
 * Usage: DisplayBoard(board);
 * ---------------------------
 * This function displays each of the squares in the board.
 * In the finished board, each square is numbered with its
 * position in the tour.
 */

static void DisplayBoard(int board[N][N])
{
    int x, y;

    for (x = N - 1; x >= 0; x--) {
        for (y = 0; y < N; y++) {
            printf(" %2d", board[x][y]);
        }
        printf("\n");
    }
}
```

```
/*
 * Function: FindTour
 * Usage: flag = FindTour(board, x, y, seq);
 * -----------------------------------
 * This function looks for a tour on the board, starting at
 * the position indicated by x and y.  The seq parameter is
 * the sequence number of this move and is needed to keep
 * track of the steps on the tour.  The function returns TRUE
 * if a tour is found and FALSE otherwise.
 *
 * This function begins by checking for three simple cases:
 *
 * 1. The position is off the board.
 * 2. The position has been previously visited.
 * 3. The tour is complete.
 *
 * In the first two cases, there can be no tour that begins
 * from that position on the board.  In the last case, the
 * function can immediately return TRUE.
 *
 * If the simple cases do not apply, the function marks the
 * current square and then tries all possible moves.  If it
 * finds a tour from the resulting position, the function
 * returns TRUE to the next highest level.  If no moves lead
 * to a tour, the function must back out of this move and try
 * again at a higher level of the recursion.
 */

static bool FindTour(int board[N][N], int x, int y, int seq)
{
    int option, nx, ny;

    if (OffBoard(x, y) || board[x][y] != 0) return (FALSE);
    board[x][y] = seq;
    if (seq == N * N) return (TRUE);
    for (option = 0; option < 8; option++) {
        switch (option) {                      // Note: the reason we need this
          case 0: nx = x + 2; ny = y + 1; break;// switch statement is that
          case 1: nx = x + 1; ny = y + 2; break;// there is simple loop that we
          case 2: nx = x - 1; ny = y + 2; break;// could iterate over to cover
          case 3: nx = x - 2; ny = y + 1; break;// all of the knights possible
          case 4: nx = x - 2; ny = y - 1; break;// moves.  If we were modeling
          case 5: nx = x - 1; ny = y - 2; break;// a piece that had simpler moves,
          case 6: nx = x + 1; ny = y - 2; break;// a double for loop would be more
          case 7: nx = x + 2; ny = y - 1; break;// appropriate.
        }
        if (FindTour(board, nx, ny, seq + 1)) return (TRUE);
    }
    board[x][y] = 0;
    return (FALSE);
}

/*
 * Function: OffBoard
 * Usage: if (OffBoard(x, y)) . . .
 * -------------------------------
 * This function TRUE if the position (x, y) is off the board.
 */

static bool OffBoard(int x, int y)
{
    return (x < 0 || x >= N || y < 0 || y >= N);
}
```