ECE220 Midterm 2 Review

Author: Members of HKN

Fall 2024

1 Recursion and Arrays

(a) **All Possible Paths**: Given a 2D square grid with obstacles, write a recursive function to count the total number of paths from the top-left cell to the bottom-right cell. You can only move right or down, and you can only move into squares on the grid marked as '0', and most not leave the bounds of the grid. N is provided as the length of the side of the square grid.

A function skeleton for num_paths() has been provided for you, fill in the blanks. You must also fill in the blanks in the main() function to the first call of num_paths().

```
int array[N * N] =
{
0, 0, 0, 0,
0, 1, 0, 0,
0, 1, 1, 0,
0, 0, 0, 0
}

int main() {
  answer = num_paths(|array, 0, 0|);
  printf("Num paths = %d", answer );
  return 0;
}

int num_paths(int array[], int x, int y) {

  int paths = 0;
  // Reached destination
  if (------------------) {
    return 1;
  }
  // Out of bounds
  if (------------------) {
    return 0;
  }
  // Obstacle
  if (array[------------------] == 1) {
    return 0;
  }

  // Recursive Cases
  paths += num_paths(------------------, ------------------, ------------------);
  paths += num_paths(------------------, ------------------, ------------------);

  return paths;
}
```

(b) **Minesweeper**: Minesweeper is a game where you are presented with a grid and you try not to step on a bomb. We present you a simplified version of minesweeper. When you tap on a cell, there are three possibilities.

1) if the cell is surrounded by at least one bomb, the cell is marked with the number of bombs it neighbors.
2) The cell itself has a bomb, in which case the game is over.
3) If no bombs surround the cell, then all of its neighboring cells are marked by how many bombs neighbor them. Of course, if a neighboring cell has no bombs, the process repeats for that cell. This is where recursion comes into play. For example, given a sample hidden grid and the corresponding sample blank player grid:

Hidden grid:

x		x	x	x
			x	

Player grid:

If you tap on the cell at row 1, column 1, the output player grid should look like

	2			

However, if you tap on the cell at row 2, column 1, the output player grid should look like

1	2	3		
0	0	1		
0	0	1		

A function skeleton for fillPlayerGrid() has been provided for you. It takes in 6 parameters:

toFill: this is an int array in row-major representing what the player grid should look like after the tap. All cells are initialized to -1, indicating unmarked cells. You are tasked with filling in this array so that it represents the player grid after a tap at a certain cell.

hiddenGrid: This is an int array in row-major where a 1 represents a bomb in that cell and a 0 represents no bombs. We have also provided a helper function, countBombs(), which takes in a hidden grid and indices r and c and returns you how many bombs neighbor that cell.

r: this is an integer representing the row of the cell tapped. You may assume it is in-bounds.

c: this is an integer representing the column of the cell tapped. You may assume it is in-bounds.

rows: this is an integer representing the number of rows of the grid.

columns: this is an integer representing the number of columns of the grid.

Fill in the blanks for fillPlayerGrid().

```
1
2
3       /*
4           toFill -- array represented in row-major where all cells are initialized
    with -1s.
5           Your job is to fill in the corresponding cells.
6           hiddenGrid -- array represented in row-major with locations of the bombs
7           r -- row index of current cell
8           c -- column index of current cell
9           rows -- number of rows in the grid
10          columns -- number of columns in the grid
11
12          You are given a helper function, countBombs().
13          Given a cell at (r,c), it returns the number of neighboring bombs at that
    cell.
14          Function signature:
15
16
17          int countBombs(int * hiddenGrid, int r, int c, int rows, int columns)
18          The parameters are the same as above.
19      */
20
21
22
23      void fillPlayerGrid(int * toFill, int * hiddenGrid, int r, int c, int rows, int
     columns)
24      {
25
26          //Base Case: There is a bomb AT the cell
27          if (hiddenGrid[r * columns + c])
28          {
29              return;
30          }
31          //Base Case: The cell has at least one neighboring bomb cell
32          // get number of neighboring bombs using countBombs()
33          int centerBombs = __________;
34          if (___________ )  // If number of bombs is not zero
35          {
36              // Set toFill at (r,c) to the number of bombs
37              toFill[________] = centerBombs;
38              return;
39          }
40
41          /* Recursive case: The cell has zero neighboring bomb cells; check its
    neighbors */
42          // go from the previous row to the next row
43          for (int x = _________; x <= ________; x++)
44          {
45              if ( ________________ ) // check if we are in row bounds
46              {
47                  // Go from the previous column to the next column
48                  for (int y = ____ ; y <= _____ ; y ++ )
49                  {
50                      if ( _________)  // check if we are in column bounds
51                      {
52                      /* check if this is not the same cell at (r,c) */
53                          if ( ___________ )
54                          {
55                              /*  get count of number of neighboring bomb cells for
    cell (x,y) */
56                              int bombs = _________________________;
57                              // If we haven't already updated toFill at (x,y)
```

```
58                                  if (toFill[____________] == -1  )
59                                  {
60                                      /* if the cell at (x,y) has at least one
    neighboring bomb */
61                                      if (__________ )
62                                      {
63                                          toFill[____________] = bombs;
64                                      } else
65                                      {
66                                          toFill[____________] = _______;
67                                          //Recursively call using x and y
68                                          fillPlayerGrid(_______, ______, ______, _____,
    ______, _______);
69                                      }
70                                  }
71
72                              }
73                          }
74                      }
75                  }
76              }
77          }
78
```

2 C to Lovely LC3 Conversion

(a) **1.**: Gana decides to write a function to calculate his score on STAT 400 exams based on a measure of how much sleep, studying, and food he has. He came up with the code below:

```
int score(s, ss, f) {
    int score;
    score = ss + f - s;
    return score;
}

int Gana()
{
    int sleep = 8;
    int study = 7;
    int food = 3;

    return score(sleep, study, food);

}
```

Caller setup: Please complete the Lc3 code and how the RTS diagram looks like after caller setup. Assume that sleep, study, and food are stored in R1, R2, and R3 respectively. Please ensure to draw where R6 and R5 are after caller setup ends. Assume the values for the variables have already been set in the stack.

RTS:

LC3 code:

; assume variables have already been set ; Please load sleep, study, and food into R1, R2, and R3 ; respectively

-
-
-

; Push score() arguments (only update stack pointer once)

-
-
-
-

; call score()

-

Callee setup: After callee setup ends, Please draw what the stack looks like, including updated R6 and R5 values

RTS:

LC3 code:
; Set up bookkeeping information and allocate space
; for the local variable (only update stack pointer once)

-
-
-
-

Function Logic:
; Pop out variables into R2, R3, R4
; add up everything, store result into R4
; store R4 into score
; return to caller

-
-
-
-
-
-
-
-

Callee Teardown: Please write the code for callee teardown, and then fill in the RTS after Callee Teardown has occurred. Make sure to include where R5 and R6 are as well! RTS:

LC3:
; store score in return value, using R0 as a temp register

-
-

; restore R7, restore CFP, and deallocate space on the stack

-
-
-
-

Caller Teardown: Complete caller teardown, and show what RTS looks like after Caller teardown completes (make it look like how it was when we started the first question, not full caller teardown)
RTS:

LC3:
; Pop Return value into R3
; finish tearing down stack

-
-

3 Conceptual

(a) **Stack**: When using a stack, when you pop an item off the stack, is it removed from memory?

(b) **Pointers**: Assume we declare 4 separate pointer variables in a C program as shown below.

```
1    int* david;
2    char* eisa;
3    float* kyle;
4    void* xavier;
5
```

Which one of these variables takes up the most space in memory?
a. david
b. eisa
c. kyle
d. xavier
e. they are the same size
f. They take up no memory space

(c) **Recursion**: What are some potential downsides in a recursive solution?

(d) **2D Arrays**: Suppose we have a 2d array stored in column major order as.

123	93	0	76	67
921	82	10	4	5
42	13	43	1	65
100	2	54	10	32

How do you access the array to retrieve the element '9'? Assume the array is declared as array[4][5]. Write two answers using both 1d and 2d array notation.

(e) **Passing Pointers**: Explain the importance of having the parameters of this function be pointers.

```
1 void swap(int* a, int* b)
2 {
3     int temp;
4     temp = *a;
5     *a = *b;
6     *b = temp;
7 }
```

(f) **Arrays**: in C, Arrays in functions are...
a. Pass by reference
b. Pass by value
c. Either/or

(g) **Rambunctious Recursion**: When writing a recursive algorithm, what is the goal of each recursive step? (Hint: The base case represents the simplest form of the problem)

(h) **Sorting algorithms**: Which type of sorting algorithm is written below?

```
void sort(int arr[], int n) {
    int i, j;
    for (i = 0; i < n - 1; i++)
        for (j = 0; j < n - i - 1; j++)
            if (arr[j] > arr[j+1])
                swap(&arr[j], &arr[j + 1]);
}
```

(i) **Midpoint**: What is wrong with Lucas' recursive midpoint function?

```
find_midpoint(int a, int b) {
    if (a == b) { return a; }
    else { return find_midpoint(a+1, b-1); }
}
```

(j) **Recursive Reversal**: What is the output of this program? If there is an error in ReverseArray, identify the line and fix it? (Hint: it might be nice and helpful to print every step of Reverse Array)

```
void ReverseArray(int array[], int size) {
    int start = 0;
    int end = size - 1;
    int temp;

    if (start < end) {
        // Swap First and Last
        temp = array[start];
        array[start] = array[end];
        array[end] = temp;

        ReverseArray(array, size-1);
    }
}
int main(){
    int array[5], i;
    for (i = 0; i<5; i++){
        array[i] = i;
    }
    ReverseArray(array, 5);
    printf("Reversed Array: ");

    for (i = 0; i<5; i++){
        printf("%d ", array[i]);
    }
    printf("\n");
    return 0;
}
```

(k) **Arrays on stack**: Rahul initializes an array and calls a function"print" on it. Please write what the RTS looks like after Callee setup.

```
int main{
    int agi = 1;
    char arr[5] = {'R', 'A', 'H', 'U', 'L'};
    print(char, agi);
}
```

(l) **Alternative Indexing**: Fill in the blanks to make the two arrays have identical values.

```
int array1[4][2];
int array2[8];
int i, j;
for (i = 0; i < 2; i++) {
    for(j = 0; j < 4; j++) {
    array1[______][______] = i + j;
    array2[_______] = i + j;
    }
}
```

4 Extra Problems

(a) **Difficult Student Sort**: Fill In the blanks to find the student with the highest GPA and store a pointer to them in best_student

```
typedef struct StudentStruct {
    int UIN;
    float GPA;
} Student;

int main () {
    Student all_students[5];
    // Load data into all students:
    load_students(all_students, 5);
    // Find the student with the highest GPA:
    Student* best_student = _________
    find_best(all_students, 5, __________ );
    printf("Best GPA:%f\n",  __________);
}

void find_best(Student* all, int num_students, Student** best) {
    for (int i = 0; i < num_students; i++) {
        if (all[i].GPA > __________) {
            __________//Fill:
        }
    }
}
```

(b) **C to LC3**: Convert the following C function into LC3 using Callee Setup and Teardown.

```
int foo(int a, int b) {
    int x;
    x = a + b;
    return x;
}
```