ECE220 Midterm 2 Review

Author: Eisa Kamran, David Zheng, Kyle Chung

February 8, 2025

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 must 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 example_grid[N * N] =
{
0, 0, 0, 0,
0, 1, 0, 0,
0, 1, 1, 0,
0, 0, 0, 0
}

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

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

  int paths = 0;
  // Reached destination
  if (x == N-1 && y == N-1){
    return 1;
  }
  // Out of bounds
  if (x >= N || y >= N) {
    return 0;
  }
  // Obstacle
  if(array[x*N+y]==1){
    return 0;
  }

  // Recursive Cases
  paths += num_paths(array, x, y+1)
  paths += num_paths(array, x+1, y);

  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().

```
    /*
        toFill -- array represented in row-major where all cells are initialized with -1s.
        Your job is to fill in the corresponding cells.
        hiddenGrid -- array represented in row-major array with locations of the bombs
        r -- row index of current cell
        c -- column index of current cell
        rows -- number of rows in the grid
        columns -- number of columns in the grid

        You are given a helper function, countBombs().
        Given a cell at (r,c), it returns the number of neighboring bombs at that cell.
        Function signature:


        int countBombs(int * hiddenGrid, int r, int c, int rows, int columns)
        The parameters are the same as above.
    */



    void fillPlayerGrid(int * toFill, int * hiddenGrid, int r, int c, int rows, int columns)
    {

        //Base Case: There is a bomb AT the cell
        if (hiddenGrid[r * columns + c])
        {
            return;
        }
        //Base Case: The cell has at least one neighboring bomb cell
        // get number of neighboring bombs using countBombs()
        int centerBombs = countBombs(hiddenGrid, r,c,rows,columns);

        if (centerBombs != 0)  // If number of bombs is not zero
        {
            // Set toFill at (r,c) to the number of bombs
            toFill[r * columns + c] = centerBombs;

            return;
        }

        // Recursive case: The cell has zero neighboring bomb cells; check its neighbors
        //go from the previous row to the next row
        for (int x = r-1; x <= r + 1; x++)
        {
            if ( x >= 0 && x < rows ) // check if we are in row bounds
            {
                // Go from the previous column to the next column
                for (int y = c - 1; y <= c + 1 ; y ++ )

                {
                    if ( y >= 0 && y < columns)  // check if we are in column bounds

                    {
                        if ( x != r&& y != c ) // check if this is not the same cell at (r,c)
                        {
                            //  get count of number of neighboring bomb cells for cell (x,y)
                            int bombs = countBombs(hiddenGrid, x,y,rows,columns);

                            // If we haven't already updated toFill at (x,y)

                            if (toFill[x * columns + y]  == -1  )
```

```
61                                  {
62                                      if ( bombs > 0)
63                                      // if the cell at (x,y) has at least one
    neighboring bomb
64                                      {
65                                          toFill[x * columns + y] = bombs;
66                                      } else
67                                      {
68                                          toFill[x * columns + y] = 0;
69                                          //Recursively call using x and y
70                                          fillPlayerGrid(toFill, hiddenGrid, x, y, rows,
    columns);
71                                      }
72                                  }
73
74                              }
75                          }
76                      }
77                  }
78              }
79          }
80
```

2 C to Lovely LC3 Conversion

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);

}
```

Gana() runs as stated and hits the "score" function. Please complete the following function

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:

Value	
8	sleep (arg) ← R6
7	study (arg)
3	food (arg)
3	food (local)
7	study (local)
8	sleep (local)

LC3 code:

; Please load sleep, study, and food into R1, R2, and R3 ; respectively

- LDR R1, R6, #2
- LDR R2, R6, #1
- LDR R3, R6, #0

; Push Score arguments to stack (only update stack ; pointer once)

- ADD R6, R6, #-3
- STR R1, R6, #0
- STR R2, R6, #1
- STR R3, R6, #2

; Call score

- JSR score

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

RTS:

Value	
	score (local) ← R6, R5
Old R5	dyn link
R7	ret addr
	ret val
8	s (arg)
7	ss (arg)
3	f (arg)
3	food (local)
7	study (local)
8	sleep (local)

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

- ADD R6, R6, #-4
- STR R7, R6, #2
- STR R5, R6, #1
- ADD R5, R6, #0

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

- LDR R2, R6/R5, #4
- LDR R3, R6/R5, #5
- LDR R4, R6/R5, #6
- NOT R2, R2
- ADD R2, R2, #1
- ADD R4, R4, R2
- ADD R4, R3, R4
- STR R4, R6, #0

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:

Value	
2	score (local)
Old R5	dyn link
R7	ret addr
2	ret val ← R6
8	s (arg)
7	ss (arg)
3	f (arg)
3	food (local)
7	study (local)
8	sleep (local) ← R5

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

- LDR R0, R6/R5, #0
- STR R0, R6, #3

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

- LDR R7, R6, #2
- LDR R5, R6, #1
- ADD R6, R6, #3
- RET

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:

Value	
2	score (local)
Old R5	dyn link
R7	ret addr
2	ret val
8	s (arg)
7	ss (arg)
3	f (arg)
3	food (local) ← R6
7	study (local)
8	sleep (local) ← R5

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

- LDR R3, R6, #0
- ADD R6, R6, #4

3 Conceptual

(a) **Stack**: When using a stack, when you pop an item off the stack, is it removed from memory?
Nothing, the top of stack pointer is instead moved down.

(b) **Pointers**: Assume we declare 4 separate pointer variables in a C program on a 32-bit system 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

e. is correct. All of the variables are pointer types, so they all take up the same size in memory (the word size of the system, 32 bits in this case). Although it is true that the values they point to take up different sizes in memory, with char* being the smallest (8 bits).

(c) **Recursion**: What are some potential downsides to a recursive solution?
1) Stack Overflows. A recursive function uses the run time stack for each function call. Depending on the depth of the recursion, this may cause the stack size to grow greater than the amount of memory on the system, resulting in various problems.
2) Performance. Similar to stack overflows, there is a lot of additional overhead in setting up and tearing down stack frames for each function call. This can cause a slow program if the compiler is unable to optimize it.

(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	9	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.

Col-major order: array[3][2], array[3 * 4 + 2];

(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  }
```

Having parameters as pointers allow the data pointed to to change from within the function. If the variables were passed in as just their values, only the run time stack variables would be edited and the new values would not be saved. Note how this function doesn't return anything, but is still useful.

(f) **Arrays**: in C, Arrays in function arguments / parameters are...
a. Pass by reference
b. Pass by value
c. Either/or
The answer is a. Arrays in C always decay to a pointer to their first element when passed in functions. This means the entire array isn't copied onto the run time stack every time it is passed into a function. Because arrays represent contiguous regions in memory, just having a pointer to the first element is enough to access any element in the array. Note: The only way to pass an entire array is to wrap it in a struct.

(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)
The goal of each recursive step is to make the problem slightly simpler. An example is factorial, you calculate number * factorial(number-1). By decrementing the number each time, we make the problem easier, until we get to the base case.

(h) **Sorting algorithms**: Aaron writes the following algorithm. Which sorting algorithm did he use?

```
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]);
}
```

Bubblesort

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

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

There are two main things wrong. The first, is that were is no return type for this function, we need to make this return a float. The second, is that we need to deal with numbers that are odd-length apart, for example 4 and 7. To do this we can add: else if (a+1 == b) return ((a+b)/2); as another condition to check.

(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)

```
1  void ReverseArray(int array[], int size) {
2      int start = 0;
3      int end = size - 1;
4      int temp;
5
6      if (start < end) {
7          // Swap First and Last
8          temp = array[start];
9          array[start] = array[end];
10         array[end] = temp;
11
12         ReverseArray(array, size-1);
13     }
14 }
15 int main(){
16     int array[5], i;
17     for (i = 0; i<5; i++){
18         array[i] = i;
19     }
20     ReverseArray(array, 5);
21     printf("Reversed Array: ");
22
23     for (i = 0; i<5; i++){
24         printf("%d ", array[i]);
25     }
26     printf("\n");
27     return 0;
28 }
```

The program is basically just flipping the first and last elements, and then flipping the first and second to last and so on. So in the end, the only thing that actually changes is the first element goes to the back and the rest is shifted down. To fix this we need to change the recursive call to ReverseArray(array +1, size-2)

(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.

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

Value	
local variable	← R5, R6
caller frame pointer	
return address	R7
return value	
pointer to arg[0]	arr(arg)
1	agi(arg)
'R'	arr[0] (local)
'A'	arr[1] (local)
'H'	arr[2] (local)
'U'	arr[3] (local)
'L'	arr[4] (local)
1	agi(local)

(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[ j ][ i ] = i + j;
    array2[2*i+j]=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

```c
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 = &(all_students[0]) ;
    find_best(all_students, 5, &best_student );
    printf("Best GPA:%f\n", best_student.GPA);
}

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

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

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

```
ADD R6, R6, #-1          ; Allocate spot for return value
ADD R6, R6, #-1
STR R7, R6, #0           ; Push R7 (Return Address)
ADD R6, R6, #-1
STR R5, R6, #0           ; Store R5 (Caller's Frame Pointer)
ADD R5, R6, #-1          ;Set frame pointer for Callee
LDR R1, R5, #2           ;Allocate memory for local variables (x)
LDR R2, R5, #1           ; Load a
ADD R0, R1, R2           ; Load b
STR R0, R5, #0
STR R0, R5, #3           ; Store result into x
ADD R6, R5, #1           ; Store result into return value slot
LDR R5, R6, #0           ; Pop local variables
ADD R6, R6, #1           ; Pop Frame Pointer
LDR R7, R6, #0
ADD R6, R6, #1           ; Pop Return Address
RET
```