Computer Science I: Quiz 1 (Programs 1 - 3) Solution

1) (15 pts) A knight in chess can move in one of eight directions, all of which form an 'L' shape. If give each square on a board (x, y) coordinates, x for the row number and y for the column number, a knight on square (x, y) can move directly to

(x-2, y-1), (x-2, y+1), (x-1, y-2), (x-1, y+2), (x+1, y-2), (x+1, y+2), (x+2, y-1) and (x+2, y+1).

(a) Write two constant arrays, DX and DY, that encapsulate the relative movement of a knight. Please order your arrays in the manner that the eight ordered pairs above are listed (for ease of grading).

```
const int DX[] = {-2,-2,-1,-1,1,1,2,2}; // 1/2 pt per pair

const int DY[] = {-1,1,-2,2,-2,2,-1,1}; // 1 pt for order
```

(b) Assume that a knight is on an infinite chess board, starting at (0, 0). Write a function that takes in an array of moves, where the moves are all integers in between 0 and 7, inclusive, representing the movement specified by the DX, DY arrays and prints out each location the knight visits from (0,0), NOT including the starting spot. As an example, if the move array stored [3, 2, 5, 5], then your function should print out

```
(-1, 2)
(-2, 0)
(-1, 2)
(0, 4)
```

Write the printMoves function as specified. (The input parameters are the array of moves, moves and length, the length of the array moves.)

```
void printMoves(int* moves, int length) {

    int x = 0, y = 0, i;


    for (i=0; i<length; i++) {              // 2 pts
        x = x + DX[moves[i]];               // 3 pts
        y = y + DY[moves[i]];               // 3 pts
        printf("(%d, %d)\n", x, y);         // 2 pts
    }


}
```

2) (10 pts) In the solution to the birthday problem, a function studentcmp is written. The function takes in pointers to two struct students and returns a negative integer if the first one comes before the second one, 0 if they are identical and a positive integer if the first one comes after the second one. Why is it better to write a function like this than embedding the logic of the function in the merge or partition functions?

One important reason is code reuse. Let's say that someone wanted to add other functions separate to a sort that needed to compare two student structs in the future. If the logic for this task was embedded in merge or partition, it would not be reusable. One would have to redo the whole logic again. But, if this comparison task was written in a separate function, new code could just call that comparison function. This sort of task is so common that even C defines a similar function for strings, since comparing strings lexicographically is useful to solve many other problems.

A second reason is flexibility. Let's say that for whatever reason, somewhere down the line, it is decided that the method by which we want to compare two students will change. If the code for the comparison is embedded in multiple functions, the code editing might be very difficult and may introduce many bugs. But, if a separate compare function is written, which other functions call, all we have to do is rewrite the compare function to the new specification, test the function by itself thoroughly, and then the rest of our code will naturally work.

A third reason is to reduce debugging time or the chance of bugs. If we separate out the logic of the compare into a separate function, it's easier to test on its own than if it's embedded in multiple places. If we are certain that our general sort logic is correct but our sort isn't working, then we can narrow down and just look at our compare function to find the error.

There are certainly other reasons than these, but these are three major areas that I feel are relatively clear advantages for writing such a compare function.

Grading: Students should have at least two reasons thoroughly fleshed out. Give up to 5 pts per reason. Give partial credit (1 - 5 pts) based on the depth and clarity of their response. In some sense, you are grading relatively here, so read through a few papers before you assign any grades, to get a feel for the quality of the student responses. Pick your best to be a 5 and your worst to be a 1 or 2 and use those two as anchors for grading the rest.

3) (15 pts) Consider a different prompt for the mastermind assignment that takes in the same identical input: Instead of outputting the number of possible color combinations that are consistent with the current transcript of the game, output the ***first*** possible color combination (in lexicographical ordering) that is consistent with the current transcript of the game.One way to edit the currently posted solution to accomplish this task would be to rewrite the solve function. The solve function for the assignment takes in the combo array and an integer k, and returns the number of possible solutions where the first k slots of combo are fixed. In the edited version, this function would return null if there were no solutions under the given constraints and return an int* storing the appropriate color combination otherwise.

Here is the solve function posted in the solution:

```
int solve(int* combo, int k) {
    if (k == numSlots) return eval(combo);

    int i, sum = 0;
    for (i=0; i<numColors; i++) {
        combo[k] = i;
        sum += solve(combo, k+1);
    }
    return sum;
}
```

Recall that currently, eval returns 1 if combo is consistent with the guesses and feedback and 0 otherwise. Rewrite this function to do the alternative task at hand. Some of the edited function has been given to you.

```
int* solve(int* combo, int k) {

    int i;

    if (k == numSlots) {
        int res = eval(combo);
        if (res == 0)
            return NULL ;                                   // 1 pt
        else {
            int* arr = malloc(numSlots*sizeof(int));        // 4 pts

            for (i=0; i<numSlots; i++)                      // 3 pts
                arr[i] = combo[i];                          // 2 pts
            return arr;
        }

    }

    for (i=0; i<numColors; i++) {
        combo[k] = i;
        int* arr = solve(combo, k+1);

        if ( arr != NULL )                                  // 2 pts

            return arr ;                                    // 2 pts
    }
    return NULL ;                                           // 1 pt
}
```