# ECE264 Fall 2021 Exam 2

# 6:30-7:30PM, November 4

Please write your answers on the answer sheets only and **return the answer sheet**. Do not return the questions. Please keep the answer sheet clean. Do not use the answer sheet as your scratch space. The answer sheet should contain **only** the answers.

Do not write anything that is not the answers. Otherwise, you may lose points.

Please use $\boxed{\textbf{DARK}}$ ink. If your pen is too light, your answer may not be graded.

When you take this exam, you agree with the Purdue Honor Pledge: **advancing a culture of academic integrity, seeing academic dishonesty and cheating as threats to Purdues reputation**.

| Value | Character | Value | Character | Value | Character |
|---|---|---|---|---|---|
| 48 | 0 | 65 | A | 97 | a |
| 49 | 1 | 66 | B | 98 | b |
| 50 | 2 | 67 | C | 99 | c |
| 51 | 3 | 68 | D | 100 | d |
| 52 | 4 | 69 | E | 101 | e |
| 53 | 5 | 70 | F | 102 | f |
| 54 | 6 | 71 | G | 103 | g |
| 55 | 7 | 72 | H | 104 | h |
| 56 | 8 | 73 | I | 105 | i |
| 57 | 9 | 74 | J | 106 | j |
|  |  | 75 | K | 107 | k |
|  |  | 76 | L | 108 | l |
|  |  | 77 | M | 109 | m |
|  |  | 78 | N | 110 | n |
|  |  | 79 | O | 111 | o |
|  |  | 80 | P | 112 | p |
|  |  | 81 | Q | 113 | q |
|  |  | 82 | R | 114 | r |
|  |  | 83 | S | 115 | s |
|  |  | 84 | T | 116 | t |
|  |  | 85 | U | 117 | u |
|  |  | 86 | V | 118 | v |
|  |  | 87 | W | 119 | w |
|  |  | 88 | X | 120 | x |
|  |  | 89 | Y | 121 | y |
|  |  | 90 | Z | 122 | z |

## 1 Merge Sort

Consider the following program for merge sort.

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdbool.h>

// input: arr is an array and its two parts
// arr[l..m] and arr[m+1..r] are already sorted
// l, m, r mean left, middle, and right respectively
//
// You can assume that l <= m <= r.
//
// output: arr is an array and all elements in arr[l..r] are sorted
// Do not worry about the elements in arr[0..l-1] or arr[r+1..]

void merge(int * arr, int l, int m, int r, int * mergecount)
{
  (* mergecount) ++;
  if ((l > m) || (m > r)) { return; } // one or both are empty
  int * temparr = malloc (sizeof (* temparr) * (r - l + 1));
  if (temparr == NULL) { return; } // malloc fail

  int ind1 = l;
  int ind2 = m + 1;
  int ind3 = 0;
  while ((ind1 <= m) && (ind2 <= r))
    {
      if (arr[ind1] < arr[ind2])
        {
          temparr[ind3] = arr[ind1];
          ind1 ++;
        }
      else
        {
          temparr[ind3] = arr[ind2];
          ind2 ++;
        }
      ind3 ++;
    }
  // copy the remaining elements
  while (ind1 <= m)
    {
       temparr[ind3] = arr[ind1];
       ind1 ++;
       ind3 ++;
    }
  while (ind2 <= r)
    {
       temparr[ind3] = arr[ind2];
       ind2 ++;
       ind3 ++;
    }

  // copy temparr to array
  ind3 = 0;
  for (ind1 = l; ind1 <= r; ind1 ++)
    {
      arr[ind1] = temparr[ind3];
      ind3 ++;
    }
  free (temparr); // release temporary memory
}

// merge sort has the following steps:
// 1. if the input array has one or no element, it is already sorted
// 2. break the input array into two arrays. Their sizes are the same,
//    or differ by one
// 3. send each array to the mergeSort function until the input array
//    is small enough (one or no element)
// 4. sort the two arrays

void mergeSort(int * arr, int l, int r,
               int * sortcount, int * mergecount)
{
  (* sortcount) ++;
  if (l > r)  { return; } // empty array, nothing to do
  if (l == r) { return; } // only one element, already sorted
  int m = (l + r)/2;
  mergeSort(arr, l, m, sortcount, mergecount);
  mergeSort(arr, m+1, r, sortcount, mergecount);
  merge(arr, l, m, r, mergecount);
}
```

```

int main(int argc, char * * argv)
{
  int size;
  for (size = 2; size < 16; size *= 2)
    {
      int * arr = malloc(sizeof(int) * size);
      int ind;
      for (ind = 0; ind < size; ind ++)
        {
          arr[ind] = size - ind; // descending order
        }
      int sortcount = 0;  // how many times mergeSort is called
      int mergecount = 0; // how many times merge is called
      mergeSort(arr, 0, size - 1, & sortcount, & mergecount);
      printf("sortcount=%d, mergecount=%d\n", sortcount, mergecount);
      free (arr);
    }
  return EXIT_SUCCESS;
}
```

The `mergeSort` function is caleld three times using arrays of sizes 2, 4, and 8. This question asks you how many times `mergeSort` and `merge` are called.
This is the program's output:

```
sortcount= <<ANSWER A>>, mergecount=<<ANSWER B>>
sortcount= XXX, mergecount=XXX
sortcount= <<ANSWER C>>, mergecount= <<ANSWER D>>
```

What are the answers marked as `A - D`? Do not worry about `XXX`
This is `<<ANSWER E>>`: If the input array is already sorted in the ascending order, will the answer for `C` change? Please answer **YES** if the answer for `C` changes. Please answer **NO** if the answer for `C` does not chang.
**Answer:** A: 3, B: 1, C: 15, D: 7, No

## 2 Depth First Search

Here is a portion of depth-first-search code from HW9:

```c
bool depthFirstSolve(Maze *m, MazePos curpos, char *path, int step) {
  // Base case 1. if curpos is at the end, append \0 to the path,
  //   return true
  if (atEnd(curpos, m)) {
    path[step] = '\0';
    return true;
  }
  // Base case 2. if curpos is invalid (wall, out of bounds, or
  //   already visited) return false
  if (!inBounds(curpos, m)
    || hitWall(curpos, m)
    || isVisited(curpos, m)) {
    return false;
  }

  // inductive case: check each possible direction from curpos
  char direction[4] = {NORTH, SOUTH, EAST, WEST};
  MazePos steps[4] = {
      {.xpos = curpos.xpos, .ypos = curpos.ypos - 1}, // NORTH
      {.xpos = curpos.xpos, .ypos = curpos.ypos + 1}, // SOUTH
      {.xpos = curpos.xpos + 1, .ypos = curpos.ypos}, // EAST
      {.xpos = curpos.xpos - 1, .ypos = curpos.ypos}  // WEST
  };

  for (int i = 0; i < 4; i++) {
    // Try to make a move and see if it's successful
    if (depthFirstSolve(m, steps[i], path, step + 1)) {
      // If successful, record this step in the path
      path[step] = direction[i];
      return true;
    }
  }

  // Set this square to visited ** THIS IS A BUG **
  m->maze[curpos.ypos][curpos.xpos].visited = true;

  // If no move is successful, return false
  return false;
}
```

There is a bug in this code (line 35 should happen *before* the for loop in line 25).

**ANSWER A**: Consider the following maze (remember: # is a wall, **.** is an open space, **s** is the start space, and **e** is the end space.

```
#e##
#.##
#.##
s..#
```

What will happen when you try to solve this maze? Your answer should be the number of the result below.

1. The program will say there is no solution
2. The program will go into an infinite loop/stack overflow
3. The program will say the maze is solved with the path **wsss**
4. The program will solve the maze with the path **ennn**

**Answer: 4**
**ANSWER B**: Consider the following slightly different maze (note that the start position has changed)

```
#e##
#.##
#.##
#..s
```

What will happen when you try to solve this maze? Your answer should be the number of the result below.

1. The program will say there is no solution
2. The program will go into an infinite loop/stack overflow
3. The program will say the maze is solved with the path **wwnnn**
4. The program will solve the maze with the path **sssee**

**Answer: 2**
**ANSWER C**: Suppose we change lines 17–23 to:

```
17   char direction[4] = {NORTH, SOUTH, WEST, EAST};
18   MazePos steps[4] = {
19       {.xpos = curpos.xpos, .ypos = curpos.ypos - 1}, // NORTH
20       {.xpos = curpos.xpos, .ypos = curpos.ypos + 1}, // SOUTH
21       {.xpos = curpos.xpos - 1, .ypos = curpos.ypos}, // WEST
22       {.xpos = curpos.xpos + 1, .ypos = curpos.ypos}  // EAST
23   };
```

In other words, the new search order is NORTH, SOUTH, WEST, EAST. What happens when we try to solve the maze from part B with the new program? Your answer should be the number of the result below.

1. The program will say there is no solution
2. The program will go into an infinite loop/stack overflow
3. The program will say the maze is solved with the path **wwnnn**
4. The program will solve the maze with the path **sssee**

**Answer: 3**
**ANSWER D**: Suppose we fix the `visited` bug (by setting the visited flag before the for loop), but then we replace lines 10–14 with the following:

```
10 if (hitWall(curpos, m)
11   || isVisited(curpos, m)) {
12
13   return false;
14 }
```

In other words, we forget to check whether the current position is within the maze boundary as part of our base case. Consider the following four mazes. Which of them could result in a segmentation fault when you run the program because the solver goes out of bounds? Your answer should be the number(s) of the maze(s) where the search goes out of bounds. (Assume the search order is as in the original code: NORTH, SOUTH, EAST, WEST)

1.
   ```
   #e#
   #.#
   #s#
   ```

2.
   ```
   ####
   ...e
   s###
   ```

3.
   ```
   ####
   e...
   ###s
   ```

4.
   ```
   .###
   ...e
   s###
   ```

**Answer: 3, 4**

## 3 Recursive Function

Consider the following program.

```
#include <stdio.h>
#include <stdlib.h>

int func(int n){
  printf("Function call\n");
  if(n <= 1){
    return n;
  }
  else{
    return func(n-1) + func(n-2);
  }
}

int main(int argc, char ** argv){
  if(argc != 2){
    printf("Wrong number of input.\n");
    return EXIT_FAILURE;
  }
  int n = (int) strtol(argv[1], NULL, 10);
  printf("func(%d) = %d\n", n, func(n));
  return EXIT_SUCCESS;
}
```

Suppose the executable program is named `recprog`.
Hint: If we run the program with

```
./recprog 1
```

the output of the program will be

```
Function call
func(1) = 1
```

**ANSWER A**: Run the program with

```
./recprog 8
```

What is the value of `func(8)`?
Your answer should only contain an integer, that is `func(8) = [ANSWER A]`
**Answer: 21**
**ANSWER B**: If we run the program with

```
./recprog 5
```

How many lines of **Function call** are there in the output? For example, there is one line of **Function call** in the output if we run the program with `./recprog 1`. Your answer should only contain an integer.
**Answer: 15**
**ANSWER C**: In general, denote `g(n)` as the number of lines of **Function call** in the output by running the program with integer `n`. What is the relation between `g(n)` and `g(n-1)` and `g(n-2)`? Your answer should be the number of the statement below.

1. `g(n) = g(n-1) + g(n-2)`
2. `g(n) = g(n-1) + g(n-2) + 1`
3. `g(n) = g(n-1) + 1`
4. `g(n) = g(n-1) + 2`

**Answer: 2**

## 4 Linked List

Consider the following definition of two structures in `list.h`:

```
#ifndef _LIST_H
#define _LIST_H

typedef struct node {
    int value;
    struct node * next;
} Node;

typedef struct {
    Node * head;
    Node * tail;
} List;

#endif
```

Consider the following program.

```
#include <stdio.h>
#include <stdlib.h>
#include "list.h"

// Create a new node
static Node * node_construct(int val){
    Node * nd = malloc(sizeof (Node));
    nd -> next = NULL;
    nd -> value = val;
    return nd;
}

// Insert a node to the end of a linked list
void list_insert(List * list, int val){
    Node * ptr = node_construct(val);
    if(list -> head == NULL){
        list -> head = ptr;
        list -> tail = ptr;
    }
    //----QUESTION A: Fix the following code----
    else{
        [  ANSWER A  ] = ptr;
        list -> tail = ptr;
    }
```

```
}

// Print the values of all nodes in the linked list
void list_print(List * list){
    Node * nd = list -> head;
    while (nd != NULL){
        printf ("%d ", nd -> value);
        nd = nd -> next;
    }
    printf("\n");
}

// Destroy the entire linked list
void list_destroy(List * list){
    Node * nd = list -> head;
    Node * ptr;
    while (nd != NULL){
        ptr = nd -> next;
        free (nd);
        nd = ptr;
    }
    //----QUESTION B: Fix the following code----
    free([  ANSWER B  ]);
}

int main(int argc, char * * argv) {
    if (argc < 2) {
        printf("Not enough inputs.\n");
        return EXIT_FAILURE;
    }

    // Create a linked list with the input values
    List * llist = malloc(sizeof(*llist));
    llist -> head = NULL;
    llist -> tail = NULL;
    int value;
    for(int i = 1; i < argc; i++) {
        value = (int) strtol(argv[i], NULL, 10);
        list_insert(llist, value);
    }
    list_print(llist);
    list_destroy(llist);
    return EXIT_SUCCESS;
}
```

**ANSWER A**: Fix the code in function `list_insert`.
**Answer: Acceptable answers include:**

```
list -> tail -> next
(list -> tail) -> next
(*list).tail -> next
((*list).tail) -> next
```

**ANSWER B**: Fix the code in function `list_destroy`.
**Answer: list**
**ANSWER C**: Suppose the executable program is named `listprog`. If we run the program with `./listprog 10 20 30`, what is the output? Your answer should be the number of the statement below.

1. `10 20 30`
2. `30 20 10`
3. `10 30 20`
4. `30 10 20`

**Answer: 1**