DIRECTIONS: This exam is closed book, closed note. You have 3 hours to complete the following 5 problems. Each problem is weighted equally unless otherwise specified. Within each problem, each sub-question is also weighted equivalently unless otherwise specified. Even if the question only requests an answer, feel free to explain your reasoning process in a clear manner. All text should be planned, clear, and concise. Writing vague text will rarely help to claim partial credit.

Write your answers on this exam, using the back of pages if necessary. Additional paper can be used for scratch or attached to the exam as necessary.

All responses should consist of:

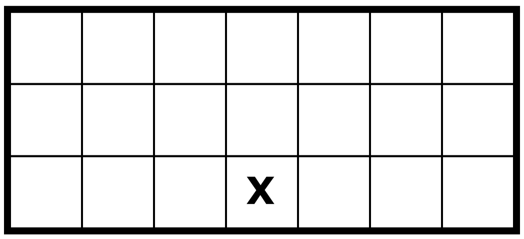
- A conclusive answer to the question (either a single sentence or number)
- A brief description of how you produced that result (or a derivation such as a trace through the program, etc)

NAME: ____

1. Suppose you have a remote controlled robot. This robot accepts four different commands:
   1. Move forward 1 unit (a square in the grid below)
   2. Rotate (in place) right 90 degrees
   3. Rotate (in place) left 90 degrees
   4. Detect_Barrier (see if there is a wall in front of it)

   a) If the robot is placed in a rectangular grid (in any orientation and in any spot), I claim that the following algorithm will eventually result in the robot continually circling the perimeter (border) of the rectangle:

```
Loop Forever
        If (Detect_Barrier) Then
                Rotate_Right
        End If
        Move_Forward
End Loop
```



If we have a rectangular grid as above (Thick line is a wall/barrier), trace what happens if the robot starts at the location marked with an **X (facing towards the top of the page)** and follows my algorithm. You may draw on the grid until the task becomes repetitive. In the space below, trace through the algorithm above (write the robot's location and function calls performed (ie Detect_Barrier or Move_Forward) until it becomes repetitive.

   b) Let's say that Moving_Forward *while facing a barrier* would break the robot's delicate claws and sensors. If the robot can be started *anywhere* in the grid and in *any* orientation (facing any direction), does this algorithm *always* ensure that our robot will not be harmed? If not, how might we change the structure of the If statement above to fix this?

c) Suppose we had a C program that was supposed to perform the above algorithm. The function look_ahead() returns “wall” if there is a wall in front of the robot’s sensors or “space” if the front is clear (as strings). Identify the coding problem here and its cause. Also, write what you would expect the value of wall_ahead to be as the program runs (the problem is a meaning issue, not a syntax issue – the program would compile fine). If you altered this algorithm in part b), do **not** fix that here. The problem is a different one here. The heart of the problem is centered around the call to process_sensor.

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

void process_sensor(int wall_ahead);

int main(void)
{
  int wall_ahead = 0;

  while (1) {
    process_sensor(wall_ahead);
    if (wall_ahead) {
      rotate_right();
    }
    move_forward();

  return 0;
}

void process_sensor(int wall_ahead)
{
  char *input_from_sensor;

  input_from_sensor = look_ahead();
  /* if the value read from look_ahead() is equal to
     “wall” */
  if (strcmp(input_from_sensor, “wall”) == 0)
    wall_ahead = 1;
  else
    wall_ahead = 0;
}
```

d) Fix the previous C program using pointers. The function prototype of process_sensor should be changed. Write the changes below using rough C syntax.

2. Consider the following code:

a)
```
#include <stdio.h>

int input_plus_one(int input);

void foo(int i, int j, int k);

int main(void)
{
  int x;

  x = input_plus_one(5);

  foo(1,1,1);

  printf("%d\n", x);

  return 0;
}

int input_plus_one(int input)
{
  int result;
  result = input + 1;
  return result;
}

/*
 * This function doesn't do anything
 * It takes input, but doesn't produce output
 * it just does some work (messes around with
 * memory)
 */
void foo(int i, int j, int k)
{
  int sum;
  sum = i + j + k;
}
```

What is printed out via the one printf (this is not a trick question)?

b) Now, consider the following code (**note the differences**). There is a problem here. Identify the problem. The program compiles (with 1 warning), runs, and does **not** seg fault. It simply produces the wrong answer. Why? It is related to the reason why we can not create an array inside a function and return it to an outside function. Think about memory, function calls, and the machine's stack architecture.

```
#include <stdio.h>

int *input_plus_one(int input);

void foo(int i, int j, int k);

int main(void)
{
  int *x;
  x = input_plus_one(5);
  foo(1,1,1);
  printf("%d\n", *x);
  return 0;
}

int *input_plus_one(int input)
{
  int result;
  result = input + 1;
  return &result;
}

/*
 * This function doesn't do anything
 * It takes input, but doesn't produce output
 * it just does some work (messes around with
 * memory)
 */
void foo(int i, int j, int k)
{
  int sum;
  sum = i + j + k;
}
```

c) Draw a representation of what happens to the machine stack (memory) when input_plus_one is called. What happens to that memory after the function terminates? Could malloc help us here? Where (in the system) does malloc allocate memory? Is that memory subject to the same limitations as the stack?

d) What does the following code produce as output?

```
void swap(int *a, int *b)
{
  int tmp;
  tmp = *a;
  *a = *b;
  *b = tmp;
}
```

if run using the following (in main, etc):

```
  x = 1;
  y = 2;
  swap(&x, &y);
  printf("x is %d and y is %d.\n", x, y);
```

3. Structures

   a) You are charged with creating an electronic dictionary that is to reside in memory. Design a data structure called DICTENTRY that holds information related to a single entry in a dictionary. This is a modeling question: there is no particularly right answer. Try to use struct syntax demonstrated in the attached linkedlist.c.

   b) Let's say that you wanted to store these structs in a "doubly linked list." This means that each element in the list should have a pointer not only to the *next* element, but also to the *previous* element. This allows you to move to the next item as well as the previous (forwards or backwards) using pointers. Below, write the changes you would have to make to the LISTNODE struct to support a doubly linked list. Would you have to change the LIST struct as well?

c) Now, explain (and demonstrate, if necessary) what changes you would have to make in the add_node() function to maintain this doubly linked list. What actions do you have to perform now when adding a new element onto the end of the list? Describe them using English/pictures with accompanying code (if desired) for added clarity. Code is recommended, but not necessary if the description is technically clear.

d) Let's say we wanted to search this doubly linked list. Does having links both ways in our list allow us to do binary search? What conditions are necessary for using binary search? Name two.

4. Taxis and Parking Cars

a. Suppose we live in a city where streets are laid out as square grids. A passenger picks up a taxi at the corner of 1st street and 1st avenue and wants to be driven to **n**th street and **m**th avenue. Regardless of the route taken, what is the best (smallest) number of blocks traversed during the trip (a block is equivalent to driving down one side of any square in the grid).

b. Once arriving, the cab driver wants to park. The parking attendant has some gigantic block available for parking that can accommodate limos and cars. **A limo is twice as long as a car**. How many different ways can vehicles be arranged on a block of size n car lengths? This is very much like counting change, except order DOES matter. For this part, just compute the number of different ways to park cars and limos given a block of **4** car lengths.

c. Sketch an algorithm for solving this. You are effectively solving a permutations problem with some minor constraints. It is also similar to the recursive makechange problem. The algorithm can be recursive or iterative (using loops).

d. What type of running time does your algorithm have? Use “Big O” notation

5. (10% of exam)

a. Write a recursive function int fact(n) that computes the factorial of n. (n * n-1 * n-2 * n-3 …)

b. Comment on its running time in terms of number of recursions. Given an input of size n, how many recursions (calls to itself) take place?
Write the running time using “Big O” notation in terms of n.

```
/*
   LISTNODE (AKA struct listTag) contains an integer AND a pointer
   to another LISTNODE (struct listTag).
*/
typedef struct listTag {
  int data;
  struct listTag *next;
} LISTNODE;

/*
   This is a structure that represents the head of the list (first element).
   Not strictly necessary, but it can decrease the amount of error handling
   we have to do
*/
typedef struct {
  LISTNODE *head;
} LIST;

void add_node(LIST *l, int data) {
  /* Two temporary LISTNODE pointers */
  LISTNODE *cur_node;
  LISTNODE *new_node;

  /* Assume that l is not null... get the head node */
  cur_node = l->head;

  /* if the head was NULL, we have an empty list and do the following */
  if (cur_node == NULL) {
    /* Manufacture a new LISTNODE and put it on the heap */
    new_node = (LISTNODE*)malloc(sizeof(LISTNODE));
    if (!new_node) {
      printf("Unable to allocate memory.\n");
      exit(101);
    }
    /* Set up the values of this newly-created node. */
    new_node->data = data;
    new_node->next = NULL;

    /* Make the head of the list point to this newly-created node */
    l->head = new_node;
    /*
       Done! Since we have modified l, those changes will be visible from
       the calling function (main, in this case)
    */
  }
  else {
    /*
       Find the end of the list by asking if the next element is NULL
       loop setting cur_node to the _next_ node...
    */
    while (cur_node->next != NULL) {
      cur_node = cur_node->next;
    }
    /* Manufacture a new node */
    new_node = (LISTNODE*)malloc(sizeof(LISTNODE));
    if (!new_node) {
      printf("Unable to allocate memory\n");
      exit(101);
    }
    /* Set values */
    new_node->data = data;
    new_node->next = NULL;
    /* link the newly-created node to the end of the list */
    cur_node->next = new_node;
  }
}
```