Problem 1 (20 points): Testing and Debugging

(10 points) The recursive function shown below is meant to return the factorial of the number N, but it has a bug. In NO MORE THAN 15 WORDS, describe the bug. Then fix the program.

(the bug) _____

_____

```
int factorial (int N)
{
    int next = factorial (N - 1);
    if (1 >= N) {
        return 1;
    }
    return N * next;
}
```

(10 points) The function below is meant to return the length of a player's name (or -1 if a NULL name or player is passed), but it has a bug. In NO MORE THAN 15 WORDS, describe the bug. Then fix the program.

(the bug) _____

_____

```
int player_name_length (const player_t* p)
{
    const char* s;
    int         count = 0;

    if (NULL == p->name || NULL == p) {
        return -1;
    }
    for (s = p->name; '\0' != *s; s++) {
        ++count;
    }
    return count;
}
```

Problem 2 (30 points): Linked Lists and Recursion

This problem deals with a linked data structure that is used to store a database of people and their friends. The list of people is stored as a linked list of `person_t`'s, each of which includes a pointer to a string containing the person's name, a pointer to the person's list of friends, and a pointer to the next person in the list:

```
typedef struct friend_t friend_t;
typedef struct person_t person_t;
struct person_t {
    char*     name;
    friend_t* friend_list;
    person_t* next;
};
```

In the linked list of friends, each `friend_t` includes a pointer to a `person_t` (the particular friend), and a pointer to the next friend in the linked list of friends:

```
struct friend_t {
    person_t* friend;
    friend_t* next;
};
```

For this problem, you must complete a recursive function called `circle_n` to determine whether two people are within distance N of one another in the circle of friends. A person is distance 0 from himself, distance 1 from everyone in his `friend_list`, distance 2 from friends of friends, and so forth.

On the next page, you will see that Prof. Lumetta has started to write the function for you. The three arguments to the function are `p`, a pointer to the person of interest, `f`, a pointer to the person's target friend, and `N`, the maximum distance between the person and the target friend. The function must return 1 if the distance constraint is satisfied, and 0 otherwise.

Problem 2, continued:

```
int circle_n (person_t* p, person_t* f, int N)
{
    friend_t* check;

    if ( ________________ ) { return 0; }            // blank #1

    if ( ________________ ) { return 1; }            // blank #2

    for (check = p->friend_list; NULL != check;

         ________________ ) {                        // blank #3

        if (circle_n ( ________________ ,            // blank #4

                       ________________ ,            // blank #5

                       ________________ )) {         // blank #6
            return 1;
        }
    }
    return 0;
}
```

Circle EXACTLY ONE ANSWER to indicate what should appear in each blank in the code above.

(5 points): blank #1

A) `p != f` B) `0 == N` C) `0 > N` D) `0 >= N` E) `p == f`

(5 points): blank #2

A) `0 == N` B) `NULL == p` C) `f != p` D) `p == f` E) `0 >= N`

(5 points): blank #3

A) `p = p->next` B) `check = check->next` C) `check = f->friend_list` D) `check++` E) `f = f->next`

(5 points): blank #4

A) `check->friend` B) `f` C) `p` D) `check` E) `p->next`

(5 points): blank #5

A) `check->friend` B) `p` C) `f` D) `N + 1` E) `p->next`

(5 points): blank #6

A) `N` B) `check` C) `1` D) `N - 1` E) `p->next`

Problem 3 (20 points): Data Structures in Memory

This problem refers to the same data structures that were used in Problem 2. Shown below on the left is C code defining these data structures along with the variables `i`, `ptr`, `x`, and `str_p`. Shown on the right is the LC-3 memory at runtime.

C code:

```
struct person_t {
    char*     name;
    friend_t* friend_list;
    person_t* next;
};
struct friend_t {
    person_t* friend;
    friend_t* next;
};
int       i;
int*      ptr;
friend_t* x;
char**    str_p;
```

LC-3 Memory

Address	Data	Comments
x4001	x3050	
x4002	x4008	
x4003	x400B	
x4004	x4008	
x4005	x0000	
x4006	x400B	
x4007	x400E	
x4008	x3021	
x4009	x400B	
x400A	x4001	
x400B	x3073	
x400C	x4008	
x400D	x4001	
x400E	x4001	
x400F	x4004	
x4010	x1337	`i`
x4011	x4010	`ptr`
x4012	x4006	`x`
x4013	x400B	`str_p`

Complete the table below by indicating the value and C type for each expression in the left column. If the expression is a structure, provide the start and end address of the structure (for example, Mem[start_addr : end_addr]) instead of a value. The first two rows of the table have been completed for you.

Expression	Value	C type
`i`	x1337	int
`&i`	x4010	int *
`ptr`		
`*ptr`		
`&ptr`		
`x->friend`		
`*str_p`		
`*(x->friend)`		
`x->friend->next`		
`&(x->friend->friend_list)`		
`&(x->friend)`		
`x+1`		

Problem 4 (30 points): Dynamic Allocation and I/O

Prof. Lumetta has started to implement the Codebreaker server that we discussed in class. As a starting point, he created the `person_t` structure below to hold a player's name and password. Players are to be kept in a cyclic, doubly-linked list using the `prev` and `next` fields of the structure.

```
typedef struct person_t person_t;
struct person_t {
    char*     name;     // dynamically-allocated copy of name
    char*     password; // dynamically-allocated copy of password
    person_t* prev;     // previous player in list
    person_t* next;     // next player in list
};
```

Next, Prof. Lumetta tried to write a subroutine to read player names and passwords from a file. Each player appears as two consecutive lines in the file. The player's name in on the first line, and their password is on the second line. Names and passwords are not allowed to include white space (space, tab, carriage return, new line), so he realized that he can use `fgets` together with `sscanf` to read the first word—either a name or a password—from each line.

Lumetta's subroutine (shown on the next page) is called `read_player_list`. The parameters are a fake player that forms the sentinel for the cyclic, doubly-linked list (as discussed in class) and a filename. **The fake player has already been initialized to serve as an empty list.** The function reads all players from the named file, dynamically allocates players, and inserts them at the end of the list. In other words, after the routine executes, following the `next` links from `fake` should produce the same order of players as found in the file.

The function **must perform all error checks and cleanup**, such as freeing unused memory and closing files.

The function returns the number of players read and inserted into the list, or -1 if an error occurs before any players are inserted (if an error occurs after inserting players, the function still returns the number of players inserted). As you might expect, Lumetta has left blanks for you to fill in. **Fill in the blanks—note that this problem is NOT multiple choice. Write your code directly in the blanks.** You may not modify any code outside of the blanks.

An example of the player file format appears below.

```
Lumetta
Secret!
Dang
H4ck4r
Lam
9999998999
Huy
default
```

Problem 4, continued:

```
// (structure definition replicated for your convenience)
typedef struct person_t person_t;
struct person_t {
    char*     name;     // dynamically-allocated copy of name
    char*     password; // dynamically-allocated copy of password
    person_t* prev;     // previous player in list
    person_t* next;     // next player in list
};

int read_player_list (person_t* fake, const char* filename)
{
    FILE*     f;         // the file to read
    char      buf[200];  // one line from the file
    char      word[200]; // one word from a line
    person_t* p;         // a new player (must be dynamically allocated)
    int n_players = 0;   // number of players inserted into list

    f = fopen ( _____________________________ , "r");
    if (NULL == f) {
        return -1;
    }
    while (NULL != fgets (buf, 200, f) && 1 == sscanf (buf, "%s", word)) {
        p = malloc ( _____________________________ );
        if (NULL != p) {
            p->name = _____________________________ ;
            if (NULL != p->name) {
                if (NULL != fgets (buf, 200, f) &&
                    1 == sscanf (buf, "%s", word)) {
                    p->password = strdup (word);
                    if (NULL != p->password) { // insert p into list

                        _____________________________ ;

                        _____________________________ ;

                        _____________________________ ;

                        _____________________________ ;

                        _____________________________ ;
                        n_players++;
                        continue;
                    }
                }

                _____________________________ ;
            }
            free (p);
        }
        if (0 == n_players) {
            fclose (f);
            _____________________________ ;
        }
        break;
    }
    fclose (f);
    return n_players;
}
```