COMS W3157 Advanced Programming, Lab #2

Please read this assignment carefully and follow the instructions EXACTLY.

Submission:

This lab consists of multiple parts. Your code for each part must be in a subdirectory (named "part1", "part2", etc.)

If a part asks for a program, you must provide a Makefile. Please refer to lab1 for the requirements for Makefiles. The TAs will make no attempt to compile your program other than typing "make".

Please include a README.txt in the top level directory. Please refer to lab1 for the README.txt requirements.

Checking memory errors with valgrind

You will be heavily penalized if your program contains memory errors. Memory errors include (among other things) failure to call free() on the memory you obtained through malloc(), accessing past array bounds, dereferencing uninitialized pointers, etc.

You can use a debugging tool called "valgrind" to check your program:

```
valgrind --leak-check=yes ./your_executable
```

It will tell you if your program has any memory error. See "The Valgrind Quick Start Guide" at http://valgrind.org/docs/manual/quick-start.html for more info.

You must include the output of the valgrind run for EACH PART in your README.txt. In addition, TAs will run valgrind on your program when grading.

Part 1: Sorting an interger array (50 points)

Write a program that dynamically allocates an array of integers. The size of the array (the number of integers, not the byte size) should be read from the user using scanf(). You may assume that the user will input a positive integer (i.e., don't do error checking). The elements of the array should be filled using random() function. After filling the array with random numbers, your program should then make a copy of the array, and sort the new array in ascending order: that is, the first entry of the array should contain the smallest integer in the array, while the last entry should contain the largest integer in the array. Then make a second copy of the original array, and sort it in descending order. Finally your program should print out all three arrays. All three arrays should be allocated using malloc() library function. Don't forget to call free() to deallocate the arrays.

You should always check the return value of malloc(), and if it's NULL, print an error message and quit the program, like this:

```
    p = malloc(1000000000);
    if (p == NULL) {
        perror("malloc returned NULL");
        exit(1);
    }
```

Make sure you do this everytime you call malloc(). This applies to all labs in this class.

Please name your executable program "isort".

See the man page or your textbook for the description of random() function. Type "man 3 random" for the man page. Linux man pages are also available online at

    http://www.kernel.org/doc/man-pages/online_pages.html.

In K&R2, there is no mention of random(). Instead, it describes the older and inferior rand() function. The usage is the same. Use random() in your code.

You will notice that random() function always returns the same sequence of integers. It's just a pseudo-random number generator that simulates randomness. You can "seed" the random number generator by calling srandom() function once in the beginning of your program. Calling it with the return value of time(NULL) will ensure a different sequence of random numbers everytime the program is run. You should do that.

For sorting, you have two options. You can implement any sorting algorithm yourself. You are allowed to look up on the Internet or any book for a sorting algorithm. If you do, please cite the source in the comment for the function.

Or you can use qsort() function provided by the standard C library (just like they provide printf()). Using the qsort() library function is simpler, but requires that you know how to use pointers to functions, which we have not covered yet. Section 5.11 of K&R2 describes pointers to functions. Unfortunately, the section uses a sorting function named "qsort" to illustrate pointers to functions, but it takes a different set of parameters than the real qsort() of standard library. The standard library version of qsort() is described on page 253, K&R2, or in the man page (type "man 3 qsort").

There is, however, one little requirement about calling the sorting function, whether it's your own function or the qsort() function. I want you to call your sorting function indirectly through the following function:

```
    /* This function sorts an integer array.

       begin points to the 1st element of the array.
       end points to ONE PAST the last element of the array.

       If ascending is 1, the array will be sorted in ascending order.
       If ascending is 0, the array will be sorted in descending order.
     */
```

```
    void sort_integer_array(int *begin, int *end, int ascending)
    {
        /* In here, you will call your real sorting function (your own
         * or the qsort()).  Basically, I want to make sure that you
         * know how to translate the begin/end parameter to whatever
         * is required for your sorting function.
         */

         ...
    }
```

Part 2: echo with a twist (50 points)

Write a program, named "twecho", that takes words as command line arguments, and prints each word twice, once as is and once all-capitalized, separated by a space. For example,

```
    ./twecho hello world dude
```

should output:

```
    hello HELLO
    world WORLD
    dude DUDE
```

Your program should handle any number of arguments. You can receive the command line arguments if you start your main() function in the following way:

```
    int main(int argc, char **argv)
```

Please refer to section 5.10 in K&R2. In particular, the picture on page 115 depicts clearly how the command line argument strings are stored in memory.

Here are some requirements and hints:

- You must use the main() function exactly as given below. You CANNOT modify the main function. Your job is to implement other functions that main() calls.

```
    int main(int argc, char **argv)
    {
        if (argc <= 1)
            return 1;

        char **copy = duplicateArgs(argc, argv);
        char **p = copy;

        argv++;
        p++;
        while (*argv) {
            printf("%s %s\n", *argv++, *p++);
        }
```

```
        freeDuplicatedArgs(copy);

        return 0;
    }
```

- You will probably need the following #includes:

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

- You can put duplicateArgs() and freeDuplicatedArgs() in the same .c file as main().

- In duplicateArgs() function, you are making a "copy" of the memory structure shown in the picture on page 115, K&R2. You will call malloc() once for the overall array where each element is of type char*, then you will call malloc() for each element of that array, each of which will hold the all-cap version of each argument. Of course, you will have to copy each string character-by-character, capitalizing as you go.

  Some useful library functions for doing this include strlen() and toupper(). See the textbook.

  Don't forget that the last element of the overall array of char*'s is a NULL pointer (see the picture in page 115, K&R2).

- In freeDuplicatedArgs() function, you must free() everything you malloc()ed. First free() all individual strings, and then free() the overall array.

--

Good luck!