ECE26400 Programming Exercise #1

This exercise includes the following learning goals:

1. How to "iterate" over arrays in C.
2. How to use `argc` and `argv` in `main`.
3. How to compile, run, and test your code.
4. How to run your code under `valgrind`, in order to find memory, and more generally, programming errors.
5. How to use `make`.

1 Getting started

You should unzip on eceprog.ecn.purdue.edu the zip file `pe01_files.zip` using the following command:

```
unzip pe01_files.zip
```

The zip file `pe01_files.zip` contains a folder, named `PE01`, and nine files within the folder:

1. `answer01.c`: This is the file that you should modify and you have to submit the modified file.
2. `answer01.h`: This is a "header" file and it contains a complete explanation of the functions you will be writing for this exercise. ***Do not modify this file.***
3. `pe01.c`: With proper arguments, your code should call the correct functions written in `answer01.c`, and you are supposed to modify the `main` function in `pe01.c` to properly parse the arguments. You could also use this file to write testing code that runs the functions in answer01.c, in order to ensure their correctness. `pe01.c` comes with some example testing code to help you get started.
4. `utility01.o`: This file contains the object code for the `read_in_array` function. ***Do not modify this file.***
5. `utility01.h`: This is the "header" file for the function in `utility01.o`. ***Do not modify this file.***
6. `inputfile`: An input file for testing your main function in `pe01.c`. It is a file that contains only numbers.
7. `invalid_inputfile`: An input file that is of incorrect format.
8. `empty_inputfile`: An input file that corresponds to an empty array.
9. `Makefile`: This is a file that helps you compile the C source programs or object codes into an executable, among other things.

To get started, read this document in its entirety. You will be writing code in the `answer01.c` file. Moreover, you will write code in the `main` function in `pe01.c` to call the functions in `answer01.c` correctly. You are encouraged to also write code in the `pe01.c` file to test the code written in the `answer01.c` file, even though this part of `pe01.c` would not be graded.

2 Functions you have to write

In programming, iterating means visiting every element of a "collection" once, and only once. An array is the most fundamental type of collection: It contains a certain number of elements and store the elements sequentially.

The easiest way to iterate over an array is to use an "index" variable in a for-loop:

```
// Assume we have an array "array", with length "len"
int ind; // the index variable
for (ind = 0; ind < len; ++ind) {
   // do something with array[ind]
}
```

You should be familiar with this programming motif from CS 15900.

2.1 `answer01.c`

Both functions `smallest_partial_sum` and `largest_difference` are declared in `answer01.h`, and this exercise requires you to define these two functions in `answer01.c`.

Note that in the declarations in `answer01.h`, "`__attribute__((nonnull (1)));`" implies that we expect these two functions to be called with non-NULL first parameter. In other words, this attribute tells the caller function to make sure that it does not pass NULL address to the functions because it is not the responsibility of these two functions to check for that.

Both functions require you to iterate over an array to compute the smallest partial sum and the largest difference.

2.1.1 Partial sums

We define the partial sums of an array of length `len` as follows: The $(\texttt{i},\texttt{j})$-th partial sum is the sum of `array[i]` through `array[j]`, $0 \le \texttt{i} \le \texttt{j} < \texttt{len}$. There are $(\texttt{len})(\texttt{len}+1)/2$ partial sums: $(0,0)$-th, ..., $(0,\texttt{len}-1)$-th, $(1,1)$-th, ..., $(1,\texttt{len}-1)$-th, ..., $(\texttt{len}-1,\texttt{len}-1)$-th.

Consider an array that contains the 5 integers: {`1, -1, 3, 5, 4`}, The $(0,0)$-th partial sum is 1, the $(0,1)$-th partial sum is 0, ..., the $(0,4)$-th partial sum is 12, ..., the $(3,4)$-th partial sum is 9, and the $(4,4)$-th partial sum is 4.

2.1.2 `smallest_partial_sum`

The function `smallest_partial_sum` computes the smallest among these partial sums.

For the special case when the array is of length 0, the smallest partial sum is defined to be 0.

2.1.3 `largest_difference`

The function `largest_difference` finds the largest element and the smallest element in any array and calculates the difference as (largest element − smallest element). If the array is of length 1, the largest difference is 0 because the only element in the array is largest and smallest at the same time.

For the special case when the array is of length 0, the largest difference is defined to be 0.

These two functions are to be written in `answer01.c`. You may assume that the address passed to each of these two functions as the first parameter is non-NULL and correct. You may also assume that the correct length of the array is passed to each of these two functions as the second parameter.

2.2 `pe01.c`

In `pe01.c`, you also have to modify the `main` function with such that it performs the following tasks:

- We expect the executable to accept two arguments (not counting the name of the executable).
- If the first argument `argv[1]` is `"0\0"` (character zero and NULL character), the executable should use the function `read_in_array` in `utility01.o` to read and store in an array the set of integers in the file specified by `argv[2]`, print to the `stdout` the smallest partial sum of the array using the format `"%d\n"`, and return `EXIT_SUCCESS`. In other words, the `main` function must call the `smallest_partial_sum` function with the correct parameters.
- If the first argument `argv[1]` is `"1\0"` (character one and NULL character), the executable should use the function `read_in_array` in `utility01.o` to read and store in an array the set of integers in the file specified by `argv[2]`, print to the `stdout` the largest difference of the array using the format `"%d\n"`, and return `EXIT_SUCCESS`. In other words, the `main` function must call the `largest_difference` function with the correct parameters.

The `main` function should return `EXIT_SUCCESS` only when the executable is successful in executing these tasks. When it receives the wrong number of arguments or when the input file is not of the expected format, for example, the executable is unsuccessful. When the executable is unsuccessful in executing these tasks, the `main` function should return `EXIT_FAILURE`.

2.3 Function you will need from `utility01.o`

Note that the first parameter of `read_in_array` does not have the `nonnull` attribute. Therefore, it is the responsibility of the function `read_in_array` to check whether a NULL address has been passed into the function.

`read_in_array` always return a non-NULL address (even if it returns a zero-length array) when the input file is of the correct format. If first parameter is NULL, the input file is non-existent, or it is of the incorrect format, the function returns a NULL address. The function also "returns" the number of integers in the array through second parameter passed to the function.

Suppose we have the following statements in the program:

```
int array_len;
int *array = read_in_array("inputfile", &array_len);
```

`array` will store a non-NULL address, and `array_len` will store `9`. If we have the following statements in the program:

```
int array_len;
int *array = read_in_array("invalid_inputfile", &array_len);
```

`array` will store a NULL address, and `array_len` will store `0`. If we have the following statements in the program:

```
int array_len;
int *array = read_in_array("empty_inputfile", &array_len);
```

`array` will store a non-NULL address, and `array_len` will store `0`.

2.4 Expected output to `stdout`

All debugging, error, or log statements should be printed to `stderr`. The only item printed to `stdout` should be the smallest partial sum or the largest difference using the format `"%d\n"`. If you print other items to `stdout`, we would not be able to evaluate your submission properly.

In the partially completed `main` function in `pe01.c`, we also provided some test functions. Note that all these functions print to `stderr`. In other words, you can print as many messages to `stderr`. We will evaluate your program by looking at the output printed to `stdout` and the status returned by `main` function.

2.5 Helper functions and macros

You may define your own helper functions in `pe01.c` and `answer01.c`. You may view those test functions provided in `pe01.c` as helper functions. All these functions are declared and defined as `static` functions. In other words, the scope of these functions are only within the files that they are found. Declaring and defining these functions as `static` eliminate the conflicts in function names across different files.

You should not modify any of the `.h` files.

You also should not use macros that start with `T_` (Upper case T and underscore) in their names. We will use such macros in the evaluation of your submission. If you use such macros, we may not be able to evaluate your submission properly.

3 Compiling your program

In this course we use the compiler `gcc`. You can compile your program by typing the following into the terminal (make sure you are in your `PE01` directory):

```
gcc -std=c99 -Wall -Wshadow -Wvla -Werror -g -pedantic -fstack-protector-strong \
    --param ssp-buffer-size=1 pe01.c answer01.c utility01.o -o pe01
```

Here, \ at the end of first line means the next line belongs to the same command. `gcc` takes a wide variety of arguments, and these are the most important arguments that we will use in this course. The arguments mean:

- `-std=c99`, use the standard defined in 1999. *Although there are newer standards, most programming fundamentals that we will learn are independent of standards.*
- `-Wall`, turn on all common compilation warnings.
- `-Wshadow`, in addition to common warnings, warn if a variable declaration shadows the declaration of another variable.
- `-Wvla`, warn if a variable-length array is used in the code.

- `-Werror`, turn warnings into errors. In other words, you will not get an executable when your program has warnings from the compiler. **Therefore, you have to remove all warnings.**
- `-g`, turn on debugging symbols so that you can see the corresponding line numbers in a debugger. Without the `-g` flag, the debugger would not be able to display the corresponding line numbers in your program.
- `-pedantic`, issue all the warnings demanded by strict ISO C, and reject all programs that use forbidden extensions.
- `-fstack-protector-strong --param ssp-buffer-size=1`, emit extra code to check for buffer overflows, such as stack smashing attacks. The minimum size of buffers protected is 1 byte. This option will inflate the compiled code and make the execution of the code less efficient. When you are confident that your code does not have buffer overflow vulnerability, this option could be removed. However, for this course, we will always compile with this option.
- `pe01.c answer01.c utility01.o`. These are the files that you are compiling and linking into a computer program. Each `.c` file is compiled separately (`gcc` does this internally), and then linked with the `.o` files into a single computer program. Compiling and linking are two different steps, but here we are doing both steps with a single command.
- `-o pe01`, create an executable `pe01`. By default `gcc` will produce a file called `a.out`, and we are just telling `gcc` to name that file `pe01` instead.

Note that when we evaluate your program, we also use the optimization flag `-O3` (uppercase letter `O` and number three) instead of the `-g` flag as follows:

```
gcc -std=c99 -Wall -Wshadow -Wvla -Werror -O3 -pedantic -fstack-protector-strong \
    --param ssp-buffer-size=1 pe01.c answer01.c utility01.o -o pe01
```

While developing your code, you should use the `-g` flag. Once you have debugged your program and are confident that you have removed all bugs, you should compile with the `-O3` option for further testing.

4 Running and testing your program

The file (technically, compilation unit) `pe01.c` "knows" about the functions in `answer01.c` and `utility01.o`, because it "includes" the files `answer01.h` and `utility01.h`, which contain the declarations for those functions.

Declarations describe the existence of some compilable function in some compilation unit somewhere. The functions declared in `answer01.h` are to be defined by you in the compilation unit `answer01.c`. The function declared in `utility01.h` is already defined in some file and compiled into some object code in `utility01.o`.

Suppose you simply compile the given files (without modifying `answer01.c` and `pe01.c`) into an executable `pe01`. To run the executable, simply type into the terminal:

```
./pe01
```

This should print to `stderr` the following messages:

```
Welcome to ECE264, we are working on PE01.

Testing smallest_partial_sum(...)
{-4, -1, 0, 1, 5, 10, 20, 21}. partial sum = 0, expected = -5. FAIL
{1, 4, -1, 6, -5, 4}. partial sum = 0, expected = -5. FAIL
{-1, -2, -3, -4}. partial sum = 0, expected = -10. FAIL
{}. partial sum = 0, expected = 0.

Testing largest_difference(...)
```

As the functions in answer01.c are incomplete, the test cases should fail in many, if not all, instances. These messages appear on the terminal, but they are actually printed to `stderr`.

To know the exit status, i.e., the value returned by `main`, you type the following immediately after running the executable:

```
echo $?
```

You should get `1` at the terminal, because `1` is equivalent to `EXIT_FAILURE`.

If you run the same executable as follows:

```
./pe01 0 inputfile
```

You will get

```
Welcome to ECE264, we are working on PE01.

{}. number of elements in array: 0
0
```

Here, the first three lines are printed to `stderr`, and the last line is printed to `stdout`. If you run the following command immediately after running the executable:

```
echo $?
```

You should get `0` at the terminal, because `0` is equivalent to `EXIT_SUCCESS`.

To convince you that indeed the messages are printed to `stderr` and `stdout`, run the following command:

```
./pe01 0 inputfile > outputfile
```

Here, "`> outputfile`" means that we want to redirect the output printed to `stdout` to the file called `outputfile`. Now, you will see the following being printed to the terminal:

```
Welcome to ECE264, we are working on PE01.

{}. number of elements in array: 0
```

You will also find a file called `outputfile` created in the folder. The file should contain `0` (and a newline). You can also redirect the output printed to `stderr` to a file. Try the following:

```
./pe01 0 inputfile 2> errorfile
```

Here, "2> errorfile" means messages printed to stderr should be redirected to errorfile. You can also redirect all output printed to stderr and stdout to a single file. Try the following:

```
./pe01 0 inputfile > all_output 2>&1
```

Which file contains all the messages?

What happens when you run the following command:

```
./pe01 0 inputfile > outputfile 2> errorfile
```

4.1 Running valgrind

Now, run the following command:

```
valgrind ./pe01 0 inputfile
```

What you do here is to supply your executable path and the necessary arguments to valgrind. valgrind is an extremely useful tool for finding problems in C programs. It is a core goal of this course that you learn how to use valgrind in order to ensure that your exercises/assignments do not have memory errors or memory leaks.

valgrind has many functions. In particular, we will use it to check whether you have memory errors such as accessing any memory that you should not or using uninitialized memory to perform conditional jump. We also use valgrind to check whether your program has memory leaks, i.e., whether all allocated memory have been freed when your program terminates.

You can redirect the output of valgrind to a file:

```
valgrind ./pe01 0 inputfile > log.txt 2>&1
```

If there are no memory errors/leaks, you should have something similar to the following two statements in the report from valgrind. (The number between == will most likely be different.)

```
==18067== All heap blocks were freed -- no leaks are possible

==18067== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

You can also run valgrind with more flags in order to have more details in the report:

```
valgrind --leak-check=full --show-leak-kinds=all --track-origins=yes \
         --verbose --log-file=log.txt ./pe01 0 inputfile
```

Here, \ at the end of first line means the next line belongs to the same command. The flags mean the following:

- --leak-check=full: Show each leak in detail.
- --show-leak-kinds=all: Show all definite, indirect, possible, and reachable leaks.
- --track-origins=yes: Track the origins of uninitialized values.
- --verbose: Provide more verbosity.

- `--log-file=log.txt`: Write `valgrind` output to `log.txt`. Output from program not included in the report.

When there are memory leaks, the `valgrind` report will suggest that you run `valgrind` again with the flag `--leak-check=full`.

`valgrind` may report the presence of suppressed errors. You do not have to worry about suppressed errors.

4.2 Interpreting and debugging `valgrind` errors

If you have errors from `valgrind`, you should fix them. Look for the *first* error, and debug that error. Sometimes you will get hundreds (or even hundreds of thousands) or errors; however, they may all be caused by the same line of code. Therefore, you always fix the first problem you encounter in the log file.

Here are some categories of the memory errors:

- "Invalid read of size X," where X is a number. This means that you are reading X bytes from memory that you should not have access.
- "Invalid write of size X," as above, except that you are writing (or assigning some value) to memory that you should not have access.
- "X bytes in 1 blocks are definitely lost in loss record U of V," which means that you have asked for X bytes of memory, but never freed them. Sometimes, you may get a message that says that certain number of bytes are "indirectly lost," "possible lost," or "still reachable." A clean program should not have any of these messages.
- "Conditional jump or move depends on uninitialized value(s)." A conditional jump or move means that a branch occurs when some condition in `if` or `for`, for example, is met or not met. If the condition involves some memory that is uninitialized (e.g., `x` has not been assigned any value), the program behavior is unpredictable. When you see this error, look at the involved statement and ask yourself how any involved variables could be initialized properly.

4.3 Intentionally injecting an error

We will inject an error in `pe01.c`. Take a look at the at the following function in `pe01.c`:

```
static void print_array(int *array, int len)
```

You change the iteration from "`for (ind = 0; ind < len; ++ind)`" to "`for (; ind < len; ++ind)`." Compile the code and run `valgrind` on `pe01`.

```
valgrind --log-file=log.txt ./pe01
```

Note that we use only one of the many flags in the command. Take a look at the report. Do you see that the error messages somehow point you to the source of the error? In particular, a subset of error messages may look like the following:

```
==13092== Conditional jump or move depends on uninitialised value(s)
==13092==    at 0x108BC5: print_Array (pe01.c:24)
==13092==    by 0x108C0F: test_smallest_partial_sum (pe01.c:41)
==13092==    by 0x108D0F: test_00_smallest_partial_sum (pe01.c:61)
==13092==    by 0x108F01: main (pe01.c:141)
==13092==
==13092== Use of uninitialised value of size 8
==13092==    at 0x108B73: print_Array (pe01.c:25)
==13092==    by 0x108C0F: test_smallest_partial_sum (pe01.c:41)
==13092==    by 0x108D0F: test_00_smallest_partial_sum (pe01.c:61)
==13092==    by 0x108F01: main (pe01.c:141)
==13092==
```

You should pay attention to the messages that correspond to functions in your program. In this case, you should look at `print_array` in `pe01.c` at line 24. In other words, valgrind actually tells you where the problems are. Because of the change, the conditional check of "`ind < len`" is now using an uninitialized `ind` in line 24. The messages also tell you the order in which functions are called and how the corresponding stack frames appear in the call stack.

Now, change the iteration to "`for (ind = 0; ind <= len; ++ind)`". Although we initialize `ind` properly here, the for-loop goes beyond the array by one position. Compile the code and run `valgrind` on `pe01`.

Now, the error messages may look like the following:

```
==13740== Conditional jump or move depends on uninitialised value(s)
==13740==    at 0x4E9A9DA: vfprintf (vfprintf.c:1642)
==13740==    by 0x4E9C76F: buffered_vfprintf (vfprintf.c:2329)
==13740==    by 0x4E99825: vfprintf (vfprintf.c:1301)
==13740==    by 0x4EA2F43: fprintf (fprintf.c:32)
==13740==    by 0x108B96: print_Array (pe01.c:25)
==13740==    by 0x108C16: test_smallest_partial_sum (pe01.c:41)
==13740==    by 0x108D16: test_00_smallest_partial_sum (pe01.c:61)
==13740==    by 0x108F08: main (pe01.c:141)
==13740==
==13740== Use of uninitialised value of size 8
==13740==    at 0x4E9696B: _itoa_word (_itoa.c:179)
==13740==    by 0x4E9A00D: vfprintf (vfprintf.c:1642)
==13740==    by 0x4E9C76F: buffered_vfprintf (vfprintf.c:2329)
==13740==    by 0x4E99825: vfprintf (vfprintf.c:1301)
==13740==    by 0x4EA2F43: fprintf (fprintf.c:32)
==13740==    by 0x108B96: print_Array (pe01.c:25)
==13740==    by 0x108C16: test_smallest_partial_sum (pe01.c:41)
==13740==    by 0x108D16: test_00_smallest_partial_sum (pe01.c:61)
==13740==    by 0x108F08: main (pe01.c:141)
```

Some functions showed in the messages, e.g., `fprintf`, `buffered_vfprintf`, and `vfprintf` are provided by the system and they are out of your control. You should pay attention to the messages that corre-

spond to functions in your program. In this case, the error messages indicate you are printing an element outside the array. Although the error is from line 25 of `pe01.c`, you have to correct the problem in line 24.

The preceding error messages are for arrays such as `array1`, `array2`, and `array3`, all of which has the memory allocated within a stack frame. `array4` is technically not an array, but an address pointing to memory allocated on the heap. The error messages for such an array may look like the following:

```
==13740== Invalid read of size 4
==13740==    at 0x108B7A: print_Array (pe01.c:25)
==13740==    by 0x108C16: test_smallest_partial_sum (pe01.c:41)
==13740==    by 0x108DC1: test_00_smallest_partial_sum (pe01.c:75)
==13740==    by 0x108F08: main (pe01.c:141)
==13740==  Address 0x522f040 is 0 bytes after a block of size 0 alloc'd
==13740==    at 0x4C31B0F: malloc (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==13740==    by 0x108DA0: test_00_smallest_partial_sum (pe01.c:73)
==13740==    by 0x108F08: main (pe01.c:141)
```

Here, it says that the program is trying to read 4 bytes (an `int`). As the 4 bytes reside outside the allocated memory (of 0 bytes), it is an invalid read. The address at which the program is performing an invalid read is "`0 bytes after a block of size 0 alloc'd`" because we are trying to read right beyond the last element in the array.

Next, revert to the original `print_array` function and make a change to the following function in `pe01.c`:

```
static void test_00_smallest_partial_sum()
```

Immediately after the statement "`int *array4 = malloc(sizeof(*array4) * 0);`", add the statement "`array4[0] = 4;`" Compile, and run `valgrind` to get a new report. Do you see error messages that correspond to an invalid write? It is a write because you are making an assignment to store some value into a memory location. However, it is invalid because the memory should have 0 bytes.

Again, revert to the original `test_00_smallest_partial_sum` function. Comment out the "`free(array4);`" statement, compile, and run `valgrind` to get a new report. You will see the following summary:

```
==18970== HEAP SUMMARY:
==18970==     in use at exit: 0 bytes in 1 blocks
==18970==   total heap usage: 1 allocs, 0 frees, 0 bytes allocated
==18970==
==18970== LEAK SUMMARY:
==18970==    definitely lost: 0 bytes in 1 blocks
==18970==    indirectly lost: 0 bytes in 0 blocks
==18970==      possibly lost: 0 bytes in 0 blocks
==18970==    still reachable: 0 bytes in 0 blocks
==18970==         suppressed: 0 bytes in 0 blocks
==18970== Rerun with --leak-check=full to see details of leaked memory
```

Now, follow the recommendation of the report and rerun `valgrind` with the flag `--leak-check=full` to see whether you can trace where the problem is and remove the problem.

4.4 Writing and testing `answer01.c` and `pe01.c`

It is your responsibility to test your program and ensure that it works. A program that is correct 99% of the time is still non-functional. It is very difficult to write programs correctly, and large programs almost always contain subtle bugs. For this reason, software engineers should adopt a zero-tolerance policy for software defects.

Please note that receiving a passing grade for a function/program does not mean your exercise is perfect, but it does mean that your exercise passes the tests that we put it through.

Testing your own program is a fundamental skill if you want to be a programmer. How do you write and test your code? The most important thing is that you should test your code as you go. After every 10 (or 20, or 30) lines of new code, compile and test the new code that you have written.

Because you are just starting, `pe01.c` includes a little bit of testing code to start you off. However, in future exercises/assignments, you will have to write all of the testing code yourself.

The sample testing routine provided is named `test_00_smallest_partial_sum()`. You probably want to write `test_01_largest_difference()`.

To write your own testing code, you have to think about the function you are developing, and then you have to write code that demonstrates that it always works. This is a core skill to develop if you are interested in being a competent computer programmer.

Suppose you have made all the necessary changes to `answer01.c` and `pe01.c`. Assuming that all `fprintf` statements to `stderr` and `stdout` are not modified, you may get the following screen output when you type in the following commands after compilation:

```
./pe01 0 inputfile
Welcome to ECE264, we are working on PE01.

{142, 44, 333, 33, 246, 40, -10, -204, 190}. number of elements in array: 9
-214

./pe01 1 inputfile
Welcome to ECE264, we are working on PE01.

{142, 44, 333, 33, 246, 40, -10, -204, 190}. number of elements in array: 9
537

./pe01 193 inputfile
Welcome to ECE264, we are working on PE01.


./pe01 inputfile
Welcome to ECE264, we are working on PE01.
```

In the second to last command, the first argument is invalid. In the last command, We are short of 1 argument. Here, we assume that the correct code in `pe01.c` would terminate the program and return `EXIT_FAILURE` immediately for these two commands.

You can also create other input files for your own testing.

4.5 Running `./pe01` with `valgrind`

You should also run `./pe01` with/without arguments under `valgrind`. I suggest that you intentionally inject (and then remove) those errors in Section 4.3 after you have completed both `answer01.c` and `pe01.c`.

You should also compile the program with the `-O3` option, and then repeat the testing process. With such a flag, the report will no longer provide you the offending line numbers. However, `valgrind` may identify problems when the compiler optimizes your code, pointing to some issues that could not be identified when the program is run in the debug mode.

5 Submission

You must submit a zip file called `PE01.zip`, which contains two files:

1. `answer01.c`
2. `pe01.c`

Assuming that you are in the folder that contains `answer01.c` and `pe01.c`, use the following command to zip your files:

```
zip PE01.zip answer01.c pe01.c
```

Make sure that you name the zip file as `PE01.zip`. Moreover, the zip file should not contain any folders. Submit `PE01.zip` through Brightspace. You may submit as many versions as you like. Brightspace will keep only the most recent submission, and we will grade only the final submission.

6 Grading

It is important that if the instructor has a working version of `pe01.c`, it should be compilable with your `answer01.c` to produce an executable. Similarly, if the instructor has a working version of `answer01.c`, it should be compilable with your `pe01.c` to produce an executable. For evaluation purpose, we will use different combinations of your submitted `.c` files and our `.c` files to generate different executables. If a particular combination does not allow an executable to be generated, you do not get any credit for the function(s) that the executable is supposed to evaluate. We use both `-g` and `-O3` flags (separately) to compile your submission. Your submission is evaluated only when compilation using each of the flags is successful.

Be aware that we set a time-limit for each test case based on the size of the test case. If your program does not complete its execution before the time limit for a test case, it is deemed to have failed the test case. We will not announce the time-limits that we will use. They will be very reasonable.

The `smallest_partial_sum` function accounts for 35%, the `largest_difference` function accounts for 30%, and the `main` function accounts for 35%.

The occurrence of any memory issues (memory errors or memory leaks flagged in a `valgrind` report) will result in 50-point penalty.

7 Makefile

It appears that we have to remember many flags for `gcc` or `valgrind`. We can use the `make` utility to make your experience slightly better.

Take a look at the `Makefile`. A line starting with `#` means a comment. `CC` and `CFLAGS` define (as indicated by `=`) the compiler and the flags used by the compiler, respectively.

The label (target) `pe01` (as indicated by `:`) defines the rule to build the executable `pe01`. Immediately after the label, the files that `pe01` is dependent on are listed: `pe01.o` and `answer01.o`. Note that `utility01.o` is also required. However, this is a file that should not be changed; we therefore choose not to include it in the dependency list.

Following the dependency list, the next line(s) should state the actions to be taken. Each (action) line should be indented by a Tab character. For the label `pe01`, the action is the linking of the object codes `pe01.o`, `answer01.o`, and `utility01.o` to produce `pe01`.

The rule for producing the `.o` file is listed at the end of the `Makefile` as follows:

```
%.o: %.c
        $(CC) $(CFLAGS) -c $<
```

The dependency says that `%.o` is dependent on `%.c`, where `%` refers to any string, i.e., a wild card. The `-c` flag in the action line states that object code should be produced, and `$<` states that the name of the source code is the same as the first one in the dependency list.

The first build rule is the default rule to follow when you type "`make`" at the command line. Of course, you can also type in "`make pe01`" at the terminal to make `pe01`. In both cases, the `make` command looks for `pe01.o` and `answer01.o`. Since both are absent, `make` looks for the rule to make them, and found at the end of `Makefile` the rule to make `pe01.o` and `answer01.o`. It creates `pe01.o` and `answer01.o`, and uses them to create `pe01`.

If you type "`make clean`", the rule for `clean` will be executed, which removes all object codes and the executable.

We can also specify rules to run tests. Note that it is possible to have empty actions (see `test_all`).

Note that `make` terminates when an action results in an exit status that does not correspond to `EXIT_SUCCESS`. If you run "`make test_all`", for example, it will terminate after running `test3` because `./pe01` returns `EXIT_FAILURE`.

To force `make` to ignore the error and continue with the remaining actions when it encountered a failed action, you can run the command "`make -i test_all`".

8 A few points to remember

We only accept one output printed to `stdout`. All other messages should be printed to `stderr`. If your output to `stdout` does not match our expected output, you will get 0 for that particular test case.

You are not submitting `answer01.h` or `utility01.h`. Therefore, you should not make changes to `answer01.h` or `utility01.h`.

You can declare and define additional static functions that you have to use in `pe01.c` and `answer01.c`.

You should not use macros that start with `T_`.

Grading of programming exercises and assignments is performed on machines with similar setup as eceprog.ecn.purdue.edu. You should perform testing of your work on eceprog.ecn.purdue.edu before submission. Correct output on your computer does not translate into correct output on eceprog.ecn.purdue.edu.