ECE36800 Programming Assignment #1

This assignment covers learning objective 1: An understanding of basic data structures, including stacks, queues, and trees; learning objective 3: An ability to apply appropriate sorting and search algorithms for a given application.

This assignment is to be completed on your own. You will implement Shell sort on an array and Shell sort on a linked list. In both cases, you will use the following sequence for Shell sort:

$$\{1, 2, 3, 4, 6, \ldots, 2^p 3^q, \ldots\},$$

where every integer in the sequence is of the form $2^p 3^q$ and is smaller than the size of the array to be sorted. Note that most of the integers in this sequence, except perhaps for some, can always be used to form a triangle, as shown in Lecture slides on insertion sort and Shell sort. There may be incomplete rows of integers in the sequence below the triangle. For example, if there are 15 integers to be sorted, the corresponding sequence $\{1, 2, 3, 4, 6, 9, 8, 12\}$ (in the form of a triangle) would be organized as follows, with an incomplete row containing the integers 8 and 12 in the sequence:

```
          1

      2       3

  4       6       9

8   12
```

You are not allowed to pre-compute the sequence and store them in your program. The sequence has to be generated as part of your Shell sort functions. Moreover, **you have to generate the sequence such that the numbers in the sequence are sorted in ascending order**. For the sequence generated for sorting 15 numbers, the **sorted sequence** is $\{1, 2, 3, 4, 6, 8, 9, 12\}$. Your Shell sort will perform 12-sorting, 9-sorting, 8-sorting, 6-sorting, 4-sorting, 3-sorting, 2-sorting, and 1-sorting. In other words, Shell sort performs k-sorting in the reverse order of the sorted sequence.

1 Functions to be written

We provide you three `.h` files: `sequence.h`, `shell_array.h`, and `shell_list.h`. You will develop the functions declared in these `.h` files in the corresponding `.c` files: `sequence.c`, `shell_array.c`, and `shell_list.c`.

These `.c` files and `pa1.c` are the only files you will submit for this assignment.

For this assignment, you are not allowed to define additional structures (`struct`). **If your submission contains additional structures, your submission will receive a grade of 0.**

You are also not allowed to use mathematical functions declared in `math.h` and defined in the math library. **If your submission includes `math.h` or calls math functions declared in `math.h` and defined in the math library, your submission will receive a grade of 0.**

However, you are allowed to define additional helper functions. These helper functions should be declared as static so that the scope of each helper function stays within that of the `.c` file that contains the helper function.

Do not modify the provided `.h` files because you are not submitting them. **Any modifications you have made to the provided `.h` will not be reflected in the `.h` files that we use to evaluate your submission.**

1.1 Function in `sequence.c`

```
long *Generate_2p3q_Seq(int n, int *seq_size)
```

Here, `n` is the number of long integers to be sorted. You should determine the number of elements in the sequence and store that number in `*seq_size`. For example, if `n` is 0 or 1, the sequence should contain 0 elements. Therefore, `*seq_size` should store 0. For $n = 16$, the sequence should contain 8 elements. Therefore, `*seq_size` should store 8. You should not expect the caller function to initialize `*seq_size`.

The function should allocate adequate and not excessive amount of memory to store the elements of the sequence as `long` integers. Even when the sequence is empty, you should allocate a space of 0 elements. Yes, it is fine to call `malloc` with 0 as the input parameter. A non-NULL address pointing to a memory location with 0 bytes allocated will be returned if the allocation is successful (and you would have to free the allocated 0-byte memory in a "clean" program).

Moreover, these elements must be stored in **ascending order**, with 1 being stored at position 0 of the sequence (array), if the sequence is not empty. The address of the `long` array is returned. If `malloc` fails, you should return NULL and store 0 in `*seq_size`.

This function will be called by the `Array_Shellsort` and `List_Shellsort` functions. It is important that the caller function, e.g., `Array_Shellsort` or `List_Shellsort` function, has an integer variable to store the size of the sequence, and pass the address of this variable into `long *Generate_2p3q_Seq(int n, int *seq_size)` using the `seq_size` parameter.

Any helper functions for `Generate_2p3q_Seq`, if any, must reside in `sequence.c`. These helper functions should be declared as static.

1.2 Functions in `shell_array.c`

There are three functions that deal with performing Shell sort on an array. The `Array_Load_From_File` and `Array_Save_To_File` functions are needed to transfer `long` integers from and to a file in **binary form** to and from an array, respectively.

```
long *Array_Load_From_File(char *filename, int *size)
```

The size of the binary file whose name is stored in the `char` array pointed to by `filename` should determine the number of `long` integers in the file. The size of the **binary** file should be a multiple of `sizeof(long)`. You should allocate sufficient memory to store all `long` integers in the file into an array and assign to `*size` the number of integers you have in the array. You should not expect the caller function to initialize `*size`. The function should return the address of the memory allocated for the `long` integers.

You may assume that all input files that we will use to evaluate your code will be of the correct format.

Note that we will not give you an input file that stores more than `INT_MAX` `long` integers (see `limits.h` for `INT_MAX`). If the input file is empty, an array of size 0 should still be created and `*size` be assigned

0. You should return a NULL address and assign 0 to `*size` only if you could not open the file or fail to allocate sufficient memory.

It is important that the caller function, e.g., the `main` function, has an `int` variable to store the size of the array, and pass the address of this variable into `long *Array_Load_From_File(char *filename, int *size)` using the `size` parameter.

```
int Array_Save_To_File(char *filename, long *array, int size)
```

The function saves `array` to a file specified by `filename` in **binary format**. The output file and the input file have the same format. The integer returned should be the number of `long` integers in the `array` that have been successfully saved into the file.

If `array` is NULL or the output file cannot be opened for writing, the function should return a value of -1 and no output file should be created.

Note that this function asks you to save an array to a file. **It does not ask you to free the array.**

```
void Array_Shellsort(long *array, int size, long *n_comp)
```

The function takes in an `array` of `long` integers and sort them (using the Shell sorting algorithm). `size` specifies the number of integers to be sorted, and `*n_comp` should store the number of comparisons involving items in `array` throughout the entire process of sorting. This function will have to call `Generate_2p3q_Seq` to obtain the sequence of numbers to be used for Shell sort. You may choose to use insertion sort or bubble sort to sort each sub-array. (If you use selection sort to sort each sub-array, your program may have high run-time complexity if you do not take advantage of the properties of the sequence.) If `Generate_2p3q_Seq` fails to allocate memory for the sequence, you should use a regular insertion sort or bubble sort to sort the input array.

A comparison that involves an item in `array`, e.g., `temp < array[i]` or `array[i] < temp`, corresponds to one comparison. A comparison that involves two items in `array`, e.g., `array[i] < array[i-1]`, also corresponds to one comparison. Comparisons such as `i < j` where `i` or `j` are indices are not considered as comparisons for this programming assignment.

It is important that the caller function, e.g., the `main` function, has a `long` integer variable to store the number of comparisons, and pass the address of this variable into `void Array_Shellsort(long *array, int size, long *n_comp)` using the `n_comp` parameter. You should not expect the caller function to initialize `*n_comp`.

Any support functions for these three functions, if any, must reside in `shell_array.c`. These helper functions should be declared as static.

1.3 Functions in `shell_list.c`

There are also three functions that deal with performing Shell sort on a linked list. In this assignment, you will use the following user-defined type to store `long` integers in a linked list:

```
typedef struct _Node {
   long value;
   struct _Node *next;
} Node;
```

This structure has been defined in `shell_list.h`. Given the definition of the structure `Node`, these are the three functions you have to write to deal with performing Shell sort on a linked list:

`Node *List_Load_From_File(char *filename, int *status)`

The load function should read all (`long`) integers in the input file into a linked list and return the address pointing to the first node in the linked list. *The linked list must contain as many `Node`s as the number of `long` integers in the file.*

We expect you to make as many `malloc` or `calloc` function calls as the number of `long` integers in the file, with each `malloc` or `calloc` accounting for the allocation of the memory for a `Node`.

You should not have additional nodes in the linked list. **Moreover, the `long` integers should be stored in the same order in the linked list as they are stored in the file.** In other words, the first (last) `long` integer in the input file is the `long` integer stored in the first (last) node of the list.

You should return an empty list (`NULL`) if you could not open the file or fail to allocate sufficient memory. Moreover, you should store in `*status` a value of -1 if you could not open the file or fail to allocate sufficient memory.

If you could open the file and allocate sufficient memory to create the linked list, you should store in `*status` a value of 0. If the input file is empty, you should return an empty list.

You should not expect the caller function to initialize `*status`.

`int List_Save_To_File(char *filename, Node *list)`

The save function should write all (`long`) integers in a linked list into the output file in the order in which they are stored in the linked list. This function returns the number of integers successfully written into the file.

If the output file cannot be opened for writing, the function should return a value of -1.

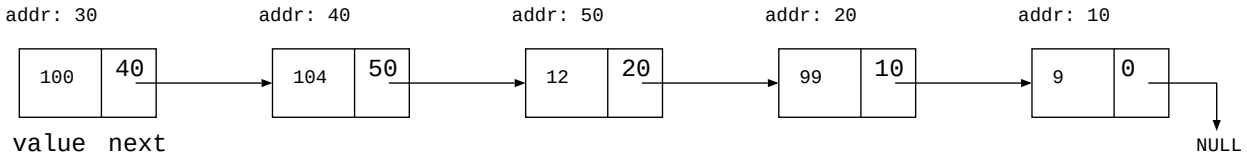
Note that this function asks you to save a list to a file. **It does not ask you to free the list.**

`Node *List_Shellsort(Node *list, long *n_comp)`

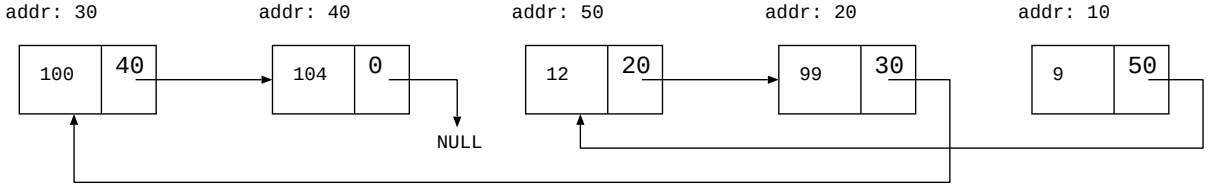
The Shell sort function takes in a `list` of long integers and sort them. To correctly apply Shell sort, you would have to know the number of elements in the `list` and generate the sequence used for sorting accordingly (by calling `Generate_2p3q_Seq`). If `Generate_2p3q_Seq` fails to allocate memory for the sequence, you should use a regular insertion sort or bubble sort to sort the input list.

The address pointing to the first node of the sorted list is returned by the function. Similar to the case of an array, a comparison here is defined to be any comparison that involves the field `value` in the structure `Node`. Note that if you are given a list of n `Node`s, you should return a sorted list of n `Node`s.

(a) Original list



(b) Sorting by manipulating addresses of Nodes



The List_Shellsort function must perform sorting by manipulating the next fields of the Nodes. Figure (a) shows an original list that is unsorted. Figure (b) shows how the list is sorted by storing the correct addresses in the next fields. The long integers stored in the value fields remain in the original Nodes. For example, the integer 99 is stored in a Node with an address 20 in the original list. The field of the same Node stores the address 10, allowing it to point to the Node storing the value 9.

After sorting, 99 is still stored in the value field of the Node with address 20. However, the next field of the Node now stores 30, allowing it to point to the Node storing the value 100.

In other words, each long integer must reside in the same Node in the original list before and after sorting.

The only array that appears in this function (and in the file shell_list.c) is the sequence generated by Generate_2p3q_Seq. You are not allowed to have other arrays (of any types) in the file shell_list.c. Therefore, you cannot divide a list into sub-lists and use an array to store these sub-lists. This restriction also applies to all helper functions of List_Shellsort and all functions in shell_list.c.

You should not expect the caller function to initialize *n_comp.

Any additional helper functions should be defined in shell_list.c file. These helper functions should be declared as static.

It is important that the linked list returned from and/or passed into these three functions in shell_list.c contain only nodes that store valid values.

In the file shell_list.c, you are expected to use the type Node, instead of the equivalent type struct _Node when you want to declare variables (or parameters). The keyword struct should not be present in the file.

1.4 main function in pa1.c

You have to write another file called pa1.c that would contain the main function to invoke the functions in shell_array.c and shell_list.c. Note that the function in sequence.c is invoked indirectly by the two Shellsort functions in shell_array.c and shell_list.c.

You should be able to obtain the executable pa1 with the following command:

```
gcc -O3 -std=c99 -Wall -Wshadow -Wvla -pedantic sequence.c shell_array.c shell_list.c
pa1.c -o pa1
```

The flags used are a subset of the flags used in ECE26400. Note that the -Werror flag has been taken out. Also, the optimization flag -O3 is used. It is recommended that while you are developing your program, you use the "-g" flag instead of the "-O3" flag for compilation so that you can use a debugger if necessary. The "-g" flag also allows valgrind to report the line numbers of offending statements that cause memory issues in your executable.

Note that we do not allow the use of math library in this assignment, as the gcc command does not have "-lm" at the end of the command.

It is your responsibility to make sure that your submission can be complied successfully on eceprog. Just to be sure, you should type in alias gcc at the command line and check whether your gcc uses the correct set of flags.

When the following command is issued,

```
./pa1 -a input.b output.b
```

the program should load from `input.b` the `long` integers to be sorted and store them in an array, run Shell sort on the array, print the number of comparisons performed to the standard output with the format `"%ld\n"`, and save the sorted `long` integers in `output.b`.

When the following command is issued,

```
./pa1 -l input.b output.b
```

the program should load from `input.b` the `long` integers to be sorted and store them in a linked list, run Shell sort on the linked list, print the number of comparisons performed to the standard output with the format `"%ld\n"`, and save the sorted `long` integers in `output.b`.

The `main` function should immediately exit with `EXIT_FAILURE` when it detects an anomaly (after cleaning up allocated memory). Examples of an anomaly includes an incorrect number of expected arguments, an input file that cannot be opened for reading, the memory required for the `long` integers to be sorted cannot be allocated, the output file cannot be opened for writing. Note that your sorting functions should handle failure of memory allocation in the function `Generate_2p3q_Seq` as described earlier.

You may declare and define other static help functions in `pa1.c`.

In the file `pa1.c`, you are expected to use the type `Node`, instead of the equivalent type `struct _Node` when you want to declare variables (or parameters). The keyword `struct` should not be present in the file.

2 Submission and grading

The assignment requires the submission (electronically) of a zip file called `pa1.zip` through Brightspace. The zip file should contain `sequence.c`, `shell_array.c`, `shell_list.c`, and `pa1.c`. We do not expect you to turn in a `Makefile` because we are going to evaluate your functions individually. `Any other files in the zip file will be discarded.` **Your zip file should not contain a folder (that contains the source files)**. Assuming that your folder contains `sequence.c`, `shell_array.c`, `shell_list.c`, and `pa1.c` (and no other `.c` files), you can create `pa1.zip` as follows:

```
zip pa1.zip sequence.c shell_array.c shell_list.c pa1.c
```

It is important that if the instructor has a working version of `pa1.c`, it should be compilable with your `sequence.c`, `shell_array.c` and `shell_list.c` to produce an executable. Similarly, if the instructor has a working version of `sequence.c`, it should be compilable with your `pa1.c`, `shell_array.c` and `shell_list.c` to produce an executable. For evaluation purpose, we will use different combinations of your submitted `.c` files and our `.h` and `.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.

The loading and saving functions will account for 20%. The sequence generation function will account for 20%. The Shell sort function for arrays will account for 20%. The Shell sort function for lists will account for 40%. The main function does not account for any points. However, if your main function does not work properly, we will deduct up to 10 points.

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. You should instead analyze whether your implementation has the expected time complexity through the numbers of comparisons or the runtimes for various test cases. You can obtain the runtime using the command `time`, e.g., `time ./pa1 -l input.b output.b`.

It is important all the files that have been opened are closed and all the memory that have been allocated are freed before the program exits. A caller function that receives heap memory should be responsible for freeing it. For example, if the instructor's `main` function calls the `Array_Load_From_File` function, it is the responsibility of the `main` function to free the returned array. It is not the responsibility of the `Array_Shellsort` or `Array_Save_To_File` to free the array. Memory issues will result in 50-point penalty.

3 What you are given

We provide `.h` files, namely, `sequence.h`, `shell_array.h`, and `shell_list.h`. We also provide sample input files in `pa1_examples.zip`. (Use the command `unzip pa1_examples.zip` to unzip the zip file.) All ".b" files are binary files. The number in the name refers to the number of `long` integers the file is associated with. For example, `15.b` contains 15 `long` integers, `15sa.b` contains 15 sorted `long` integers from `15.b`. In particular, `15sa.b` is created by `pa1` by the following command:

```
./pa1 -a 15.b 15sa.b
```

My implementation of `pa1` prints the following output to the screen when the above command is issued:

```
67
```

(The count of 67 is based on an implementation that uses insertion sort as the basic sorting algorithm in Shell sort. If we implement Shell sort with bubble sort as the basic sorting algorithm, the count is 80.)

My implementation of `pa1` prints the following output to the screen when the following command is issued:

```
./pa1 -l 15.b 15sl.b
104
```

My implementation of `pa1` also created `1Ksa.b` and `1Ksl.b`. Of course, `15sa.b` and `15sl.b` are identical and `1Ksa.b` and `1Ksl.b` are also identical. For the input files `10K.b`, `100K.b`, and `1M.b`, the output files of my implementation of `pa1` are not included.

Your implementation should not try to match the number of comparisons that my implementation reported. That is not the purpose of the assignment.

4 Getting started

Download all files associated with the assignment from the Brightspace website. As the link to the assignment will disappear after the due date, you would not be able to access any of these files from Brightspace after the deadline. Any updates to these instructions will be announced through Brightspace.

Given that the input files are in binary format, you probably want to write some helper functions to print the array of `long` integers before and after sorting in text (instead of binary) for debugging purpose. Keep in mind that `fread` and `fwrite` for binary files are analogous to `fscanf` and `fprintf` for text files.

You probably want to write and test the load and save functions together because if you call the load and save functions without sorting the array or linked list, the output file should match the input file.

The main challenge of this assignment is to perform Shell sort on a linked list. When we perform Shell sort on an array, we do not have to divide the array in sub-arrays. This assignment challenges you to perform Shell sort on a linked list without dividing the list into several sub-lists. This is a rare example when bubble sort is more useful than insertion sort.

My suggestion is that you first implement Shell sort of an array using bubble sort as the basic sorting algorithm. It is important that the bubble sort routine terminates early when it detects that an array is sorted. (In the case of Shell sort, the bubble sort should terminate early when it detects that all sub-arrays are sorted.)

The following algorithm implements bubble sort on an array `A` of n integers.

```
sorted = false
last_exchange = n
while (not sorted)
   sorted = true
   last_element = last_exchange - 1
   for i = 1 to last_element
      if A[i - 1] > A[i]
         exchange A[i-1] and A[i]
         last_exchange = i
         sorted = false
```

The algorithm employs two techniques to gain some efficiency. It uses a flag `sorted` to detect sortedness of an array. It also uses `last_exchange` to record the index at which the most recent last exchange occurs. As all elements at and after the `last_exchange` index are already at their correct positions at the end of the `for`-loop, the next iteration of `while`-loop should not examine those elements again. Therefore, `last_element` is assigned `last_exchange - 1`.

Your Shell sort of an array based on bubble sort should add an outer-loop to account for the sequence. The body of the loop should be very similar to the bubble sort provided above, just like how Shell sort of an array based on insertion sort is developed in class.

You probably also want to implement the above bubble sort algorithm on a linked list.

Once you have these implementations, you may be able to identify a better connection between Shell sorting an array and Shell sorting a linked list.

This assignment is about performing Shell sort. **If you implement other sorting algorithms, your submission does not meet the specifications of the assignment, and it will receive a grade of 0.**

Other than the required output to `stdout` as specified, do not print other messages to `stdout`. If you want to print error messages for debugging purposes, use `fprintf` to print the messages to `stderr`. If your program produces messages that are not expected, your submission does not meet the specifications of the assignment and it will not earn the relevant credits.

We will use `valgrind` to check for memory issues. While you are most familiar with memory leaks, `valgrind` can be useful in helping you find the cause of a segmentation fault and identify allocated memory locations that have not been initialized properly. The tool is useful only when you pay attention to all messages that valgrind reports. One useful programming habit is to keep your code `valgrind`-clean at any stage of programming. In other words, do not leave any memory issues unresolved at any stage of programming.

Another good habit to cultivate is to pay attention to the number of memory allocations made by your program. Does the number of memory allocations reported by `valgrind` match your expectation? For example, I expect my `pa1` to make 4 or more allocations for Shell sorting an array, and 18 or more allocations

for Shell sorting a list of 15 integers. Why do we need 4 or more allocations and 18 or more allocations, respectively?