ECE368 Programming Assignment (PA) #2

First Deadline: Monday, July 13, 2020 at 11:59 PM
Peer Review Deadline: Wednesday, July 15, 2020 at 11:59 PM
BRAC & Final Deadline: Friday, July 17, 2020 at 11:59 PM

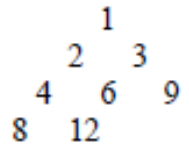
Important: By submitting your code, you certify that the work is your own and that you have not copied the code of any other student (past or current) while completing it. Code will be checked with a plagiarism checker and any failure to honor these requirements will be subject to disciplinary action as outlined in the course policy.

First submission: 5 points for your code → 5% of the final grade.
The next part of this assignment will include a peer review for the other 5 points of this programming assignment for a total of 10 points → 10% of the final grade.
The final (optional) deadline will be for resubmitting your code along with a Bug Report and Correction (BRAC) document for the possibility of earning back points for your code.

You will implement Shell sort on an array. *For extra credit you may implement Shell sort on a linked list.* In both cases you will use the following sequence for Shell sort:

`{1,2,3,4,6,...,2p3q,...}`

where every integer in the sequence is of the form `2p3q` 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 16. 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} would be organized as follows, with an incomplete row containing the integers 8 and 12 in the sequence:



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. 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, ..., 2-sorting, and 1-sorting.

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 `pa2.c` are the only files you will submit for this assignment. If you need additional structures and helper functions, you should define them in the corresponding `.c` files. Do not add anything to the `.h` files.

1.1 Function you will write for `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. For `n` = 16, the sequence should contain 8 elements. The function should allocate space to store the elements of the sequence as `long` integers (even when the sequence is empty). Moreover, these elements must be stored in ascending order. 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. Any support functions for `Generate_2p3q_Seq`, if any, must reside in `sequence.c`. It is best that these helper functions be declared as static. Do not name these helper functions with a prefix of two underscores "`__`".

1.2 Three Functions you will write for `shell_array.c`

There are three functions that deal with performing Shell sort on an array. The first two functions `Array_Load_From_File` and `Array_Save_To_File`, are not for sorting, but are needed to transfer the `long` integers to be sorted 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. The function should return the address of the memory allocated for the `long` integers.

You may NOT 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`, you can google to find it). 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` if you could not open the file or fail to allocate sufficient memory.

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

The function saves array to an external 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 the size of the array is 0, an empty output file should be created.

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

The function takes in an array of `long` integers and sort them. `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. 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.

Any support functions for these three functions, if any, must reside in `shell_array.c`.

2 Extra Credit: Using Linked Lists (Up to 2 points)

2.1 Three Functions you will have to write for `shell_list.c`

There are also a set of 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 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)`

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 Nodes as the number of `long` integers in the file. 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

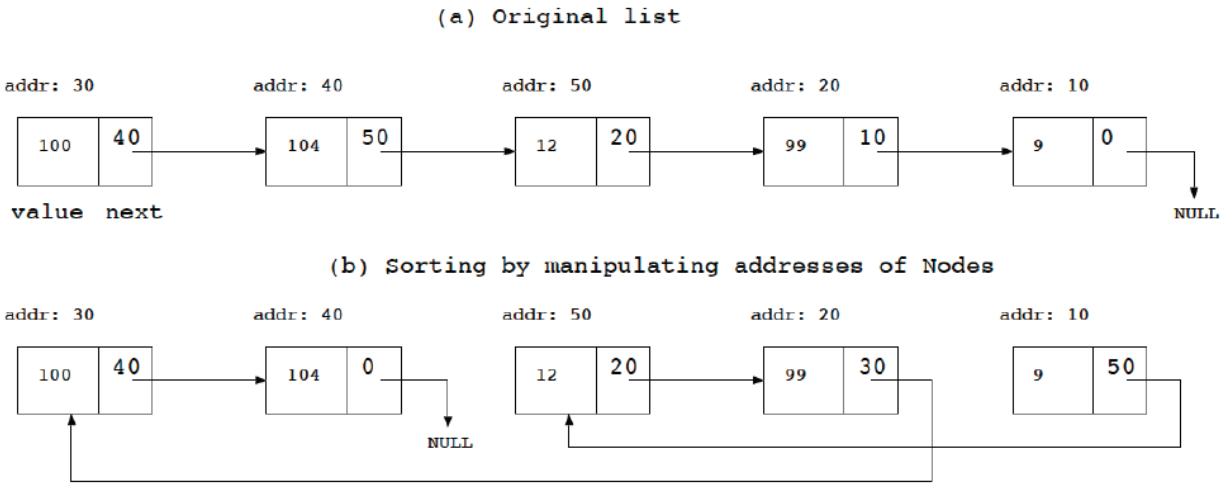
first (last) `long` integer in the input file is the `long` integer stored in the first (last) node of the list.

```
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.

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

The Shell sort function takes in a list of `long` integers and sorts them. To correctly apply Shell sort, you would have to know the number of elements in the list and generate the sequence accordingly (by calling `Generate_2p3q_Seq`). 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`.



The `List_Shellsort` function must perform sorting by manipulating the `next` fields of the `Node`s. 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 `Node`s. 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 is the sequence generated by `Generate_2p3q_Seq`. You are not allowed to have other arrays (of any types) in this file. 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`. If you want to divide a list into

sub-lists, you must use a list of linked-lists to maintain these sub-lists. You may use the following user-defined type to store a linked-list of linked-lists. To be exact, the following structure can be used to implement a linkedlist of addresses pointing to the `Node` structure.

```
typedef struct List {
   Node *node;
   struct List *next;
} List;
```

This structure is probably useful for you to maintain `k` linked-lists, where `k` is a number in your sequence. However, it is not necessary that you use this structure in your implementation. The solution implementation uses only the structure `Node`. However, you can have a much shorter run-time if you use a list of linked-lists. Using only the structure `Node` can really slow down the sorting.

Any additional structures and helper functions should be defined in `shell_list.c` file.

3 The `main` function you will write in `pa2.c`

You have to write another file called `pa2.c` that would contain the main function to invoke the functions in `shell_array.c` and `shell_list.c`. Note that the function in `generate.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 pa2 with the following command:

```
gcc -O3 -std=c99 -Wall -Wshadow -Wvla -pedantic generate.c shell_array.c
shell_list.c pa2.c -o pa2
```

When the following command is issued,

```
./pa2 -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, and save the sorted `long` integers in `output.b`. The program should also print the number of comparisons performed to the standard output with the format `"%le\n"`.

```
./pa2 -l input.b output.b
```

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

You may declare and define other help functions in `pa2.c`.

4 Submission and Grading

The assignment requires the submission (electronically) of a zip file called `pa2.zip` through Brightspace. The zip file should contain `sequence.c`, `shell_array.c`, `shell_list.c`, and `pa2.c`. Please make sure when you zip this, it does not include any hidden files, such as `.DS_Store`. Also make sure when you unzip it does not contain nested folders.

We do not expect you to turn in a Makefile because we are going to evaluate your functions individually. It is important that if the instructor has a working version of `pa2.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 `pa2.c`, `shell_array.c`, and `shell_list.c` to produce an executable.

The loading and saving functions will account for 1.1 points. The sequence generation function will account for 1.75 points. The Shellsort function for arrays will account for 1.75 points.The main function will account for 0.4 points. For Extra Credit, The Shellsort function for lists will account for up to 2 points.

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. Memory issues will result in 2-point penalty.

4.1 Given

We provide `.h` files and sample input files in `pa2_examples.zip`. 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`. (The 's' means **s**orted and the 'a' means that it used the **a**rray option.) In particular, `15sa.b` is created by `pa2` by the following command:

```
./pa2 -a 15.b 15sa.b
```

The solution implementation of `pa2` prints the following output to the screen when the above command is issued:

```
7.100000e+01
```

The solution implementation of `pa2` prints the following output to the screen when the following command is issued:

```
./pa2 -l 15.b 15sl.b
1.060000e+02
```

The solution implementation of `pa2` also created `1000sa.b` and `1000sl.b`. Of course, `15sa.b`

and `15sl.b` are identical and `1000sa.b` and `1000sl.b` are also identical. For the input files `10000.b`, `100000.b`, and `1000000.b`, the output files of the solution implementation of `pa2` are not included. Your implementation should not try to match the number of comparisons that the solution implementation reported. That is not the purpose of the assignment.

5 Getting started

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. If you want to perform Shell sort on a linked list without dividing the list into several sub-lists, it is easier to implement bubble sort in your Shell sort routine. If you want to divide a linked list into several sub-lists, you should ask yourself the question of how the "sortedness" of a linked list affect the time complexity of insertion sort. You also have to ask the question of whether you have performed (Shell) sorting correctly. If the array of `long` integers is in ascending order after sorting, have you sorted correctly?

Start sorting!

*****Check out the Brightspace website and Piazza for any updates to these instructions.*****