

ECE36800 Programming Assignment #1

Due Monday, February 10, 2020, 11:59pm

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:

$$\{h(1) = 1, h(2) = 4, h(3) = 13, \dots, h(i), h(i+1) = 3 \times h(i) + 1, \dots, h(p)\}$$

where $h(p)$ is the largest number in the sequence that is smaller than the size of the array to be sorted.

You are not allowed to pre-compute the sequence and store them in your program. The sequence has to be generated (but not stored) as part of your Shell sort functions.

Functions to be written

We provide you two .h files: `shell_array.h` and `shell_list.h`. You will develop the functions declared in these .h files in the corresponding .c files: `shell_array.c` and `shell_list.c`. These .c files and `pa1.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. It is best that these helper functions be declared as static. Do not name these help functions with a prefix of two underscores “__”.

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 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` 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 array is NULL or size is 0, an empty output file should be created.

```
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 use the sequence $\{h(1) = 1, h(2) = 4, h(3) = 13, \dots, h(i), h(i+1) = 3 \times h(i) + 1, \dots\}$ for sorting. 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 will most likely have high run-time complexity. Why?)

A comparison that involves an item in array, e.g., $\text{temp} < \text{array}[i]$ or $\text{array}[i] < \text{temp}$, corresponds to one comparison. A comparison that involves two items in array, e.g., $\text{array}[i] < \text{array}[i-1]$, 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`. It is best that these helper functions be declared as static. Do not name these help functions with a prefix of two underscores “_”.

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.* 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.

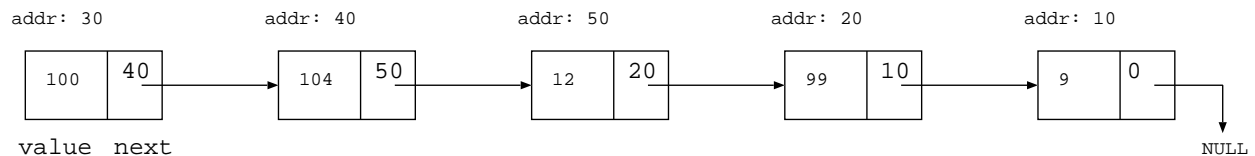
```
int List_Save_To_File(char *filename, Node *list)
```

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.

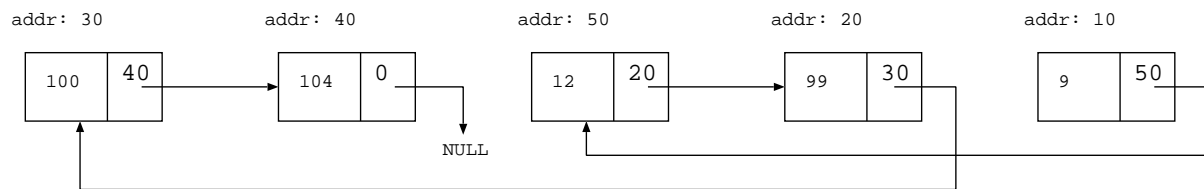
```
Node *List_Shellsort(Node *list, long *n_comp)
```

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 compute the sequence used for sorting accordingly. 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 Nodes, you should return a sorted list of n Nodes.

(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.

You are not allowed to have 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. For example, the following structure can be used to implement a linked-list 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. My implementation uses only the structure Node. You may have a faster run-time if you use a list of linked-lists. Using only the structure Node may slow down the sorting.

If you want to use this structure or any additional structures, they should be defined in `shell_list.c` file. Similarly, additional helper functions must also be defined in `shell_list.c` file. It is best that these helper functions be declared as static. Do not name these helper functions with a prefix of two underscores “__”.

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.

main function you will write 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`.

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

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

The flags used are very similar to the flags used in ECE264, except 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. **It is your responsibility to make sure that your submission can be compiled successfully on ecegrid.** 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, 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 “%ld\n”.

```
./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, 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 “%ld\n”.

What should the main function do when an empty file or an invalid file is given?

For the “-a” option, the load function (`Array_Load_From_File`) returns a NULL address if you could not open the file or fail to allocate sufficient memory. The main function should exit with `EXIT_FAILURE` when the returned address of the load function is NULL.

For an empty file, the load function should return a valid array of size 0. Therefore, an empty output file should be created.

For the “-l” option, the current setup does not allow you to distinguish between an empty file or an invalid input filename (they both will result in an empty linked list from the load function

(List_Load_From_File) for linked-lists). Therefore, you should continue to perform sorting and writing (an empty output file).

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

Submission and Grading:

The assignment requires the submission (electronically) of a zip file called `pa1.zip` through Blackboard. The zip file should contain `shell_array.c`, `shell_list.c`, and `pa1.c`. *Any other files in the zip file will be discarded.*

It is important that if the instructor has a working version of `pa1.c`, it should be compilable with your `shell_array.c` and `shell_list.c` to produce an executable. Similarly, if the instructor has a working version of `shell_array.c`, it should be compilable with your `pa1.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 points. The Shellsort function for arrays will account for 25 points. The Shellsort function for lists will account for 50 points. The main function will account for 5 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. 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.

Given:

We provide the `.h` files and sample input files in `pa1_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`. 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:

```
60
```

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

```
./pa1 -l 15.b 15sl.b  
121
```

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.

Getting started:

Copy over the files from the Blackboard website. Any updates to these instructions will be announced through Blackboard.

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. (This is a rare example when bubble sort is more useful than insertion sort.)

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?

As you have to generate the sequence for Shell sort in `shell_array.c` and `shell_list.c`, there will be similar lines of code in both .c files. While we do not encourage that in general, for the purpose of this assignment, we will reluctantly allow you to copy-and-paste these lines of code.

This assignment is about performing Shell sort. If you implement other sorting algorithms, your submission does not meet the specifications of the assignment. Your program will not earn the relevant credits if it does not meet the specifications.