Assignment 3 – Sets and Sorting

Priya Chaudhari

CSE 13S – Spring 2023

Purpose

The purpose of this program is to implement sorting methods like the shell, Batcher, Heapsort, and recursive Quicksort to sort through arrays and set functions. In addition to these files, there will be a test harness named sorting.c that will sort through these algorithms and return the array's output needed based on whatever sorting algorithm the user chooses based on the command-line option they select. There will also be an additional C file named set.c which needs to implement the bit-wise Set operations utilized in our sorting algorithms. In addition to these, there will also be a statistics file named stats.c that will collect statistics on each sort and how it performs like the size of the array it produces, the number of moves required, or the number of comparisons required. There also should be a graph displaying the moves performed and elements of all the elements with different colored lines symbolizing different sorting algorithms.

How to Use the Program

In order to use the program, the user needs to enter the command "make" within the terminal which will compile the Makefile and then they must enter a command line option to print specific statistics employed with my sorting algorithms. Below is a list of all the command line options (inputs and outputs) and what the user would need to enter for a specific purpose.

1. -a: will print all of my sorting algorithms
2. -i: enables Insertion sort
3. -s: enables Shell sort
4. -h: enables Heap sort
5. -q: enables Quicksort
6. -b: enables Batcher sort
7. -r seed: sets random seed to seed, will add an additional statistic to whatever sorting algorithm was specified beforehand
8. -n size: sets array size to size, will add an additional statistic to whatever sorting algorithm was specified beforehand
9. -p elements: prints elements (number of elements) from the array, will add an additional statistic to be printed to whatever sorting algorithm was specified beforehand
10. -h: help message

If the user would like to remove files created within the compilation process (aka the ".o" extension files that are created throughout compilation), the user should enter "make clean" within the terminal. "Make clean" is a command utilized within the Makefile provided to us which will cleanly remove unneeded files like the.o extension files to ensure a clean compilation process and free up disk space that may be used to hold these files.

The Makefile includes the following elements to ensure the compilation process runs smoothly.

1. Sets CC equal to clang
2. Inserts needed CFLAGS and LIBFLAGS (links to math library if needed)
3. Make all and individual executables like sorting
4. Have any necessary or algorithm files set to .o and the next line should include the needed flags in the following format
5. Sets all object files with source code files
6. Makes clean (Helps the compiler get rid of any unnecessary files to ensure that it compiles successfully) and format (Ensure this is in clang-format so that when you run make format in your terminal all files automatically are put in clang-format)

Program Design

The main data structures that will be used in this assignment are array structures and set operations. The 5 main algorithms which avail these data structures are the Heap sort, Batcher sort, Shell sort, recursive Quicksort, and Insertion sort.

There should be two other executables- one named set.c and the other sorting.c. set.c implement the bit-wise set operations utilized by these algorithms and sorting.c which is the file that contains my main function.

Additionally, there is a provided Makefile and 7 source/header files that my 5 algorithm files and my 2 additional .c files will make use of including batcher.h, shell.h, gaps.h, heap.h, quick.h, set.h, and stats.h.

Data Structures

My program utilizes typedef structures including stats and sets as well as arrays which define and allocate memory for each respective sorting algorithm in the program. I also utilized the get opt switch case data structure in my main program to parse through all the command line options. I used a lot of unsigned integer variables with 32-bit sizes throughout my program. The options passed into the function are stored as parameters until modified by me in the main file.

Algorithms

My sorting algorithms are filled with void functions which work through and use arrays and sets to produce statistics that are printed for the user. It goes through bitwise rotations to execute mathematics to evaluate these statistics.

Function Descriptions

insert.c: I followed the psuedocode and it runs through a for loop and takes a list as an input. There is k and a temporary variable for k and the inner loop starts at the current index which is j equal to k. Each element is shifted to the right to make space for the temp variable and then the temp variable is inserted into the sorted position indicated by index j. The outerloop ensures everything is in order.

batcher.c:

Define bit length with the variable x as its parameter Set n equal to 0 Insert a while loop where x is not equal to 0 X should be ¿¿ than 1 Increment n Outside of the while loop but in bit_length return n Define your comparator with 3 variables: a list, and 2 ints (x,y) Put an in-statement with a greater than between the two int variables Then set the array of x and y to y,x Define the batcher sort with one variable that's a list Put an if-statement where the length of this list is set to 0 Return this Set n equal to the length of this list Set t equal to the n.bit_length() Set p equal to 1 with a left shift operator of -1 Put in a while loop where p is greater than 0 Set q equal to 1 with a left shift operator of t-1 Set r to 0 Set d to p Put in a while loop where d is greater than 0 Put in a for loop Put in another if statement where i and p are equal to r

Have comparator include the parameters(A, i, i + d) Set d equal to q minus p Have q put in a right-shift operator of 1 Set r equal to p Have p contain a right shift operator with an equal sign to 1 shell.c:

Define shell_sort with array as its parameter For loop for the gaps For i in range with gap and the length of the array as the parameters Set j equal to 1 Set temp equal to the array[i] Put in a while loop where j is greater than/equal to gap and temp is less than arr[j-gap] Have arr[j] equal to the array[j - gap] Have j minus and equal to gap Have arr[j] equal to temp

heap.c:

Define build_heap with the parameters (list, int, int) For loop where for the father in range(last divided by 2, first -1 , -1) fix_heap(list, int, int) Define max_child Create 2 variables named left (which should be equal to 2*first) and right (left + 1) An if statement that compares both left and right functions Return left and right Define heap_sort with the only parameter being list Set first equal to 1 Set the variable last equal to the length of the list buildheap(list, int, int) For loop where for leaf is in range(last, first, -1) List[first -1], list[leaf - 1] is set equal to list[leaf-1], a[first - 1] fix_heap(list, first, leaf minus 1) Define fix_heap with A, first, and last parameters contains a while and if statement that swaps the arrays which should be defined which is mother and great

quick.c:

Define quick_sorter with the parameters (list, int1, int2) If list is less than int1 Set p eqal to partition (list, int1, int2) quick_sorter(list, int1, p-1) quick_sorter(list, p +1, int2) Define quick_sort(list) quick_sorter (list, 1, len(list))

Partition function referenced and should also be implemented: Define partition with the parameters (list, int1, int2) Set i equal to int1 minus 1 For j in range (int1, int2) If statement where if list[j-1] ¡ list[int2-1] Increment with i += 1 List[i-1], list[j-1] equal to list[j-1], list[i-1] List[i], list[int2 -1] = List[int2 - 1], list[i] Return i + 1

set.c:

Set set_empty(void) Set bits equal to 0 Set set_universal(void) Sets bits equal to 1 Set set_insert(Set s, uint8_t x) Sets individual bits equal to 1 when called upon with the bit-wise OR operator Set set_remove(Set s, uint8_t x) Will remove x from s and clear it to 0 through the bit-wise AND operator bool set_member(Set s, uint8_t x) Indicates the presence of the given value in a certain set s Set set_union(Set s, Set t) The union of 2 sets by using the OR operator Set set_intersect(set s, set t) The intersection of two sets with elements that are common to both sets by using the AND operator Set set_difference(Set s, Set t) The difference between 2 sets and their elements through the AND operator Set set_complement(Set s) The function is used to return the complement of a given set

Sorting.c:

Will include my main function Include libraries and files needed at the beginning before main Define options with all the command line arguments required Set the default sizes like the seed size and set sets and stat stats so all the uint32_t functions can be defined as well as srandom(seed) Start your switch case within the main function While (op = getopt . . .) Case 'a ' 3 Set functions with set insert and the command and a number from 0-3 since there are 4 sorting algorithms Break statement Set cases for h, b, s, and q Put their individual set insert function and their own number (again only 1 from 0-3) and a break statement Set case for r Set seed equal to uint32_t strtoul(optarg, NULL, 10) Break Set case for n Set size equal to uint32_t strtoul(optarg, NULL, 10) Break Set case for p Set element_num equal to uint32_t strtoul(optarg, NULL, 10) Break Set case for H Do a print statement for what every command does and break Create an if statement where element num is greater than size Create a bunch of if statements with for loops and if statements which print out each individual array as well as all of the arrays The for loops iterate through these arrays and ensure that everything is in proper order Return 0 and ensure all brackets are matched up with each loop, if statement, and while loop in your code

Results

I do believe my code fulfilled its intended purpose. It is lacking the graphs as I did not have enough time to create them. Much could be improved in regards to more effective and quickly executed code but I did the best I could.

Findings

From creating and implementing the different sorting algorithms I learned much about the importance of matters regarding syntax but also more foundational concepts like the structure of your for and while loops, if statements, function and variable declarations, as well as the importance of ensuring that you properly translate python to C.

One thing I learned and found really interesting with completing assignment 3 is really how different the C language is from Python and the importance of going line by line to ensure that you are properly interpreting the translation but also implementing this translation properly in your C code.

For my shell sort, I felt it was the easiest to translate but it was also the first sorting algorithm I translated so there was a lot of novelty in the process once I got the hang of translating this sort, the others came much easier. One key thing I learned is the translation can sometimes be more complicated with the C language like the simple line of for gap in gaps: in python requires the implementation of yet another for loop and also a variable declaration within this for loop. This made me realize how different the syntax is in comparison to Python.

For my batcher sort, I felt it was definitely one of the harder sorting algorithms to complete because there are so many moving parts. With the function declaration needed of bit length and the void functions of comparator and batcher sort. Another key thing that I found to be really important with batcher.sort is implementing the statistics functions like cmp, move, swap, and effectively doing so. I encountered the most issues with batcher and found it was infinite looping for quite some time and it was the only algorithm whose statistics were not fully working so this sort helped me pay special attention to the placement of where I put my code in loops.

For my Quicksort, it felt a little easier than batcher because I did batcher before quick so it felt more doable with fewer moving parts with their only being partition, quick sorter, and quick sort. However, I felt it was testing to ensure that both variables lo and hi were separated and compared properly. The structure of quicksort definitely felt like it was an algorithmic system rather than an algorithm with the structure and implementation.

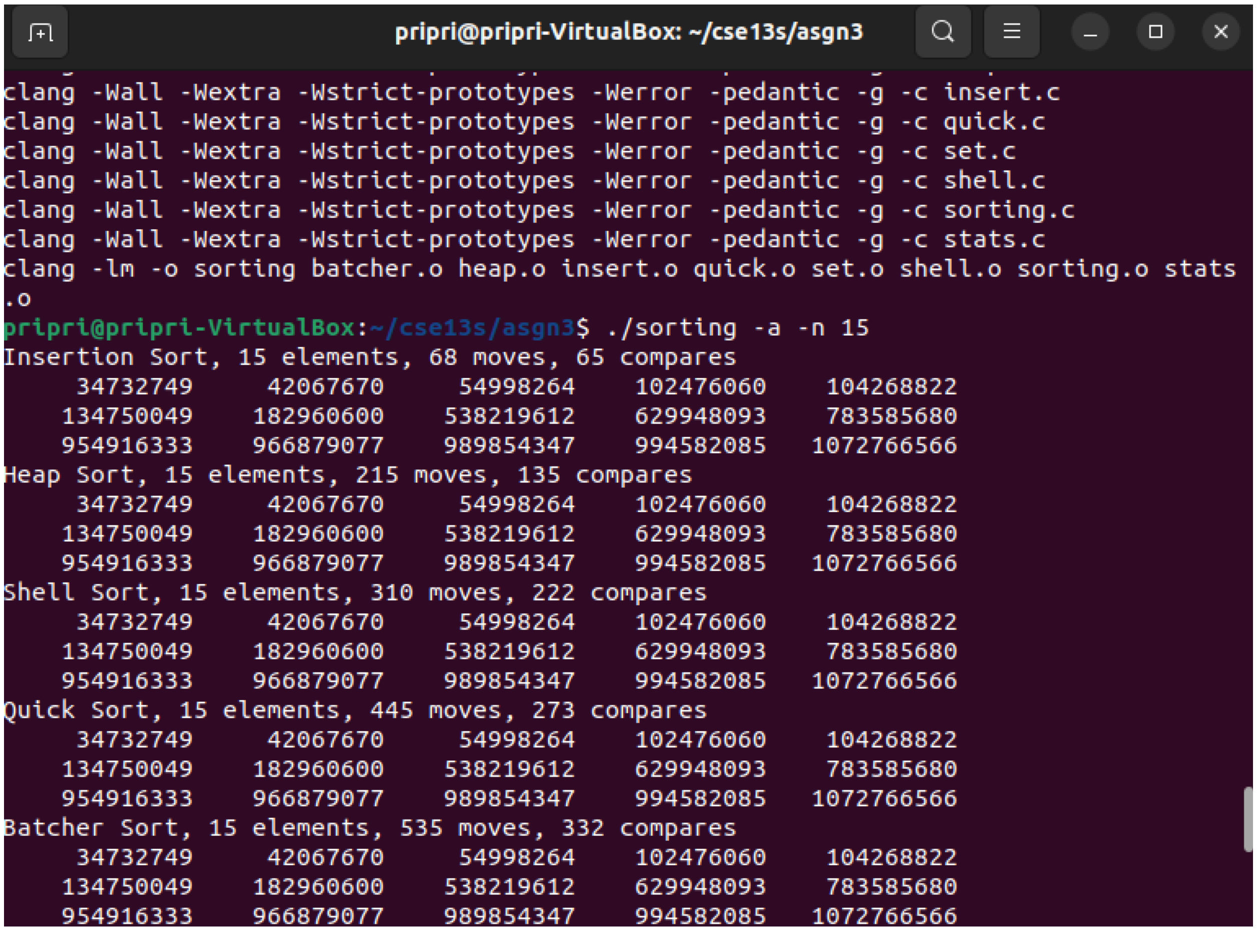
For my heap sort, it was also one of the harder algorithms to complete but I felt the translation from the python scripts to C made it a little more doable, and also with having done similar basic translations it kept getting more of second nature. Heap sort, like batcher had many moving parts and functions which made it naturally harder to organize and conceptualize in my head.

The conditions in which the sorting algorithms perform well is when there is enough memory allocation (as I did get an error saying my segmentation fault(core dumped) which meant there was not enough memory within my program), simple syntax is of course cohesive and makes sense with brackets matching up for instance, and a simple number of functions and voids the algorithm must go through. Making your code as simplistic as possible helps sorts perform better which makes sense since if one were to have complex code with messy variables and functions that could be simplified their run time would be larger, their memory would be taken up more, and their code would overall not be in its most efficient form for the sort to go through what it needs. Another factor that helps is when there is a large array/input because more numbers within an array help dictate exactly how well your sort executes.

In conclusion, from this assignment, I have learned that ensuring your code is not only sensible and contain the correct syntax and formatting, but it is also best to have enough memory allocated to your program, a lesser number of swaps and statistical functions, and simplified, schematic coherent code for your sorts to work.

Numeric results

You can include screenshots of program output, as I have in Fig. **??**.



pripri@pripri-VirtualBox: ~/cse13s/asgn3

```
clang -Wall -Wextra -Wstrict-prototypes -Werror -pedantic -g -c insert.c
clang -Wall -Wextra -Wstrict-prototypes -Werror -pedantic -g -c quick.c
clang -Wall -Wextra -Wstrict-prototypes -Werror -pedantic -g -c set.c
clang -Wall -Wextra -Wstrict-prototypes -Werror -pedantic -g -c shell.c
clang -Wall -Wextra -Wstrict-prototypes -Werror -pedantic -g -c sorting.c
clang -Wall -Wextra -Wstrict-prototypes -Werror -pedantic -g -c stats.c
clang -lm -o sorting batcher.o heap.o insert.o quick.o set.o shell.o sorting.o stats
.o
pripri@pripri-VirtualBox:~/cse13s/asgn3$ ./sorting -a -n 15
Insertion Sort, 15 elements, 68 moves, 65 compares
     34732749     42067670     54998264    102476060    104268822
    134750049    182960600    538219612    629948093    783585680
    954916333    966879077    989854347    994582085   1072766566
Heap Sort, 15 elements, 215 moves, 135 compares
     34732749     42067670     54998264    102476060    104268822
    134750049    182960600    538219612    629948093    783585680
    954916333    966879077    989854347    994582085   1072766566
Shell Sort, 15 elements, 310 moves, 222 compares
     34732749     42067670     54998264    102476060    104268822
    134750049    182960600    538219612    629948093    783585680
    954916333    966879077    989854347    994582085   1072766566
Quick Sort, 15 elements, 445 moves, 273 compares
     34732749     42067670     54998264    102476060    104268822
    134750049    182960600    538219612    629948093    783585680
    954916333    966879077    989854347    994582085   1072766566
Batcher Sort, 15 elements, 535 moves, 332 compares
     34732749     42067670     54998264    102476060    104268822
    134750049    182960600    538219612    629948093    783585680
    954916333    966879077    989854347    994582085   1072766566
```

Figure 1: Screenshot of the program running.