Assignment 4 – Sorting

Jaisuraj Kaleeswaran

CSE 13S – Fall 2023

Purpose

The program stimulates multiple sorting algorithms that takes in an array and prints out the stats behind the array being sorted based on each sorting algorithm. This program is for seeing how each sorting algorithm sorts a 100 element array with various numbers in the array in order to give accurate data on which sorting algorithms are more efficient and which sorting algorithms aren't.

How to Use the Program

In order to use the "Sorting Algorithm" program, you first need to type the commands 'make' or 'make clean all'. Next, you need to type the command './sorting' where sorting.c is the main file where the "Sorting Algorithm" program runs. Then, before clicking enter, you need to type a dash and then a letter(ex: -b) to indicate which sorting algorithm you want the sorting data to be displayed for. However, if you want to display data for all sorting algorithms implemented in the program, then you type '/.sorting -a'. Once you click enter, you will see the expected number of moves and comparisons between the elements the sorting algorithm(s) has implemented in order to sort the array of 100 elements which are displayed in sorted order. However, if you want a different-sized array, you can add '-n' and then the number of elements in an array you want sorted in order to sort an array of any length.

Command-Lines

-a:

When you type '-a' in your command to run the file sorting.c, you're indicating that you want all sorting algorithms implemented in the program to run. Let's say that only Heap, Batcher, and Quick Sort were implemented in the program. Then only those algorithms will be displayed in the output.

-b:

When you type '-b' in your command to run the file sorting.c, you're indicating that you only want the number of moves, element comparisons, and/or elements for the Batcher Sort to be displayed in your output.

-h:

When you type '-h' in your command to run the file sorting.c, you're indicating that you only want the number of moves, element comparisons, and/or elements for the Heap Sort to be displayed in your output.

-i:

When you type '-i' in your command to run the file sorting.c, you're indicating that you only want the number of moves, element comparisons, and/or elements for the Insertion Sort to be displayed in your output.

-n:

When you type '-n' in your command to run the file sorting.c, you're indicating that you want to input a certain number of elements that should be in the array you want sorted. When you don't type this command assuming that you've typed the command necessary to print a sorted array, the output would display a sorted array of 100 elements by default. However, to indicate how many elements in an array you want to be sorted when you add '-n' to your command, you would also type the number of elements in the array you want sorted(ex command: ./sorting -i -n 20).

-p:

When you type '-p' in your command to run the file sorting.c, you're indicating that you want to print the number of elements based on how you define a variable (ex: 'elements') that would represent the number of elements being printed. The default number of elements that should be printed is 100. However, if you don't want the sorted array to display at all, you type '-p 0' indicating that the program will display 0 elements in the sorted array.

-q:

When you type '-q' in your command to run the file sorting.c, you're indicating that you only want the number of moves, element comparisons, and/or elements for the Quick Sort to be displayed in your output.

-r:

When you type 'r' in your command to run the file sorting.c, you're indicating that you want to input a certain seed that you want to be used in the array you want sorted. When you don't type this command assuming that you've typed the command necessary to print a sorted array, the output will display a version of a sorted array based on the inputted seed.

-s:

When you type '-s' in your command to run the file sorting.c, you're indicating that you only want the number of moves, element comparisons, and/or elements for the Shell Sort to be displayed in your output.

-H:

When you type '-H' in your command to run the file sorting.c, you're indicating that you want to get more information on how to use the program.

Program Design

Functions and Algorithms

- insert.c:
  - insertion-sort: The insertion-sort() function takes in a Stats data type, an int array, and the length of the array. This function sorts the array using the insert sorting algorithm and implements methods from the file stats.c.

```
loop through an array from index 1 to index length-1
    set an int j to equal the index of the array
    implement the move() method in stats.c to set a temporary int
    to be equal to an element in the array

    loop while checking if j >= 1 and if the temporary int is
    less than the number in the 'j-1'th index of the array
        decrement j by one
    Using the move() method in stats.c, set the number in the jth index
    of the array to the temporary int.
```

- quick.c:
  - partition: The partition() function takes in a Stats data type, an int lo, and and int hi. This function implements methods from the file stats.c and returns an int after sorting elements in a certain part of the array indicated by the indexes lo and hi.

```
set an int i equal to lo-1
loop through from j = lo to j being 1 less than hi
    check if the number in the jth index of the array is less than
    the number in the hith index of the array
        increment i by 1
        swap the number in the ith index of the array and the number in the jth index of the a
increment i by 1
swap the number in the ith index of the array and the number
in the hith index of the array
return the int i
```

  - quick-sorter: The quick-sorter() function takes in an int array, an int lo, and an int hi. This function is recursive and calls itself twice after calling the partition() function.

```
check if the int lo is less than the int hi
    set an int p equal to the partition method
    with the parameters int array, int lo, and int hi
    call itself with the parameters int array, int lo, and p-1
    call itself again with the parameters int array, p+1, and int hi
```

  - quick-sort: The quick-sort() function takes in a Stats data type, an int array, and the length of the array. This function sorts the array using the quick sorting algorithm.

```
call the quick_sorter() method
with the parameters int array, 0, and 1 less than the length of the array
```

- heap.c:
  - max-child: The max-child() function takes in an int array, an int first, and an int last. Depending on the condition in this function, it returns either the int left which is equivalent to 2*first+1 and and the int right which is equivalent to 2*first+2.

```
set an int left to be equal to 2*first+1
set an int right to be equal to 2*first+2

check if right <= last and the number in the rightth index of the array
is greater than the number in the leftth index of the array
    return the int right
return the int left
```

  - fix-heap: The fix-heap() function takes in an int array, an int first, and an int last. This function 'fixes' the heap so that the array that would be sorted by heap sort would be sorted under the constraints of a heap.

```
set a boolean called done and set it to false
set an int called parent and make it equal to int first

loop while 2*parent+1 is less than or equal to last and done is false
    set an int called largest_child and make that equal to max_child
    with the parameters int array, int parent, and int last

    check if the number in the parentth index of the array is
    less than the number in the largest_childth index of the array
        swap the number in the parentth index of the array with
        the number in the largest_childth index of the array
        set the int parent equal to the int largest_child
    otherwise set the boolean done to be true
```

  - build-heap: The build-heap() function takes in an int array, an int first, and an int last. This function builds the heap based on the constraints of a max heap for our case so that the array can be sorted by heap sort.

```
check if the int last is greater than 0
    loop from i = (int)((last-1)/2) to i = first
    by decrementing i for every iteration
        call the fix_heap function
        with the parameters int array, int i, and int last
```

  - heap-sort: The heap-sort() function takes in a Stats data type, an int array, and the length of the array. This function sorts the array using the heap sorting algorithm.

```
set an int first to be equal to 0
set an int last to be equal to 1 less than the length of the array
call the build_heap function
with the parameters int array, int first, and int last

loop from i = last to i = first+1 by decrementing i for every iteration
```

```
    call the swap function from stats.c and use it to swap the positions
    of the number in the firstth index of the array with the number
    in the ith index of the array
    call the fix_heap function with the parameters int array, int first, and i-1
```

- batcher.c:
  - comparator: The comparator() function takes in an int array, an int x, and an int y. This function swaps the numbers in the xth and yth index of the array if the number in the xth index of the array is greater than the number in the yth index of the array.

```
check if the number in the xth index of the array
is greater than the number in the yth index of the array
    swap the number in the xth index of the array and the number
    in the yth index of the array
```

  - batcher-sort: The batcher-sort() function takes in a Stats data type, an int array, and the length of the array. This function sorts the array using the batcher sorting algorithm and implements methods from the file stats.c.

```
check if the length of the array is equal to 0
    return nothing
set an int t to be equal to zero
set an int m equal to n to calculate the bit length of n

loop while m is true
    increment t by 1
    set m equal to the bit-wise right shift of m by 1

loop while int p is greater than 0
    set an int q equal to the bit-wise left shift of 1 by t-1 times
    set an int r equal to 0
    set an int d equal to an int p

    loop while int d is greater than 0
        loop from int i = 0 to i = n-d-1
            check if the ints i and p are equal to int r
                call the comparator function with the parameters
                int array, int i, and the sum of the ints i and d
        set int d equal to the difference of int q and int p
        set int q equal to the bit-wise right shift of q 1 time
        set int r equal to the new int q
    set int p equal to the bit-wise right shift of p 1 time
```

- shell.c:
  - shell-sort: The shell-sort() function takes in a Stats data type, an int array, and the length of the array. This function sorts the array using the shell sorting algorithm and implements methods from the file stats.c.

```
loop from int i = 0 to i = 141
    loop from j = the array gaps[i] to j = 1 less than n
        set an int k to be equal to int j
        call the move() function and set a temporary int
        to be equal to the move to the number in the jth index of the array
```

```
        loop for when k >= gaps[i] and when the stats.c comparison
        between the temporary int and A[k - gaps[i]] is less than 0
            set the number in the kth index of the array
            equal to the move to A[k - gaps[i]]
            decrement gaps[i] from int k for every iteration
        set the number in the kth index of the array
        equal to the move to the temporary int
```

- set.c:
  - set-empty: The set-empty() function returns an empty set that can be indicated as the hexadecimal 0x00.

    ```
    this function returns the hexadecimal 0x00
    ```

  - set-universal: The set-universal() function returns 255 which is the number of elements in a universal set.

    ```
    this function returns the hexadecimal 0xFF which equals 255
    ```

  - set-member: The set-universal() function returns 255 which is the number of elements in a universal set.

    ```
    this function returns the hexadecimal 0xFF which equals 255
    ```

  - set-member: The set-member() function checks if int x is in the set. It it is, the function returns true. Otherwise, the function returns false.

    ```
    this function returns if s and 0x00 bit-shifted to the left is not equal to 0
    ```

  - set-insert: The set-insert() function inserts 1 to the xth position of the set. Then it returns the new set.

    ```
    create a set called val equal to 0x01 bit-shifted to the left by x
    make the set s equal to the value of s bit-wise or val
    return the set s
    ```

  - set-remove: The set-remove() function goes to the digit in the xth index of the set and removes it. Then, the function returns the new set.

    ```
    create a set called val equal to 0x00 bit-shifted to the left by x
    make the set s equal to the value of s bit-wise or val
    return the set s & not 0x01 bit-shifted to the left by x
    ```

  - set-union: The set-union() function returns both sets in the parameters.

    ```
    return set s bitwise-and return set t
    ```

  - set-intersection: The set-intersection() function returns both sets under the bit-wise or operator.

    ```
    return set s bitwise-or set t
    ```

  - set-difference: The set-difference() function returns the difference between both sets in the parameters.

    ```
    return set s and not set t
    ```

– set-complement: The set-complement() function returns the negation or 'not' the set.

```
return not set s
```

- sorting.c: The file sorting.c defines commands to call functions based on their commands. This would make the program add additional outputs based on the commands that the user types.

```
define a string Options which stores all the possible commands
the user can type to run the program
set an int seed to be 13371453
set the int 'length' of an array to be 100
set the default number of elements to print to be 100
set a type enumerator to be equal to the options
that print the sorted algorithms
define a Stats datatype from the file stats.c
initialize a set equal to an empty set
set an int opt

make a switch case for each command (ex: -a, -b, -h, etc.)
using getopt and store the string Options = "Hahbsqi:n:p:r"
    call the corresponding functions depending on the command in Options
    check for case -a:
        call set_insert and add the parameters set and a
        break from the switch statement
    check for case -h:
        call set_insert and add the parameters set and h
        break from the switch statement
    check for case -b:
        call set_insert and add the parameters set and b
        break from the switch statement
    check for case -s:
        call set_insert and add the parameters set and s
        break from the switch statement
    check for case -q:
        call set_insert and add the parameters set and q
        break from the switch statement
    check for case -i:
        call set_insert and add the parameters set and i
        break from the switch statement
    check for case -n:
        store the number of elements for arr_ints that the user wants
        break from the switch statement
    check for case -p:
        print the number of elements for arr_ints that the user wants
        break from the switch statement
    check for case -r:
        store the seed for arr_ints that the user wants
        break from the switch statement
    check for case -H:
        store the program help and usage menu
        return 0 to exit the program
    Move to the default case:
        store the program help and usage menu
        return 0 to exit the program
check if the set is empty or false, meaning that the commands haven't been used
```

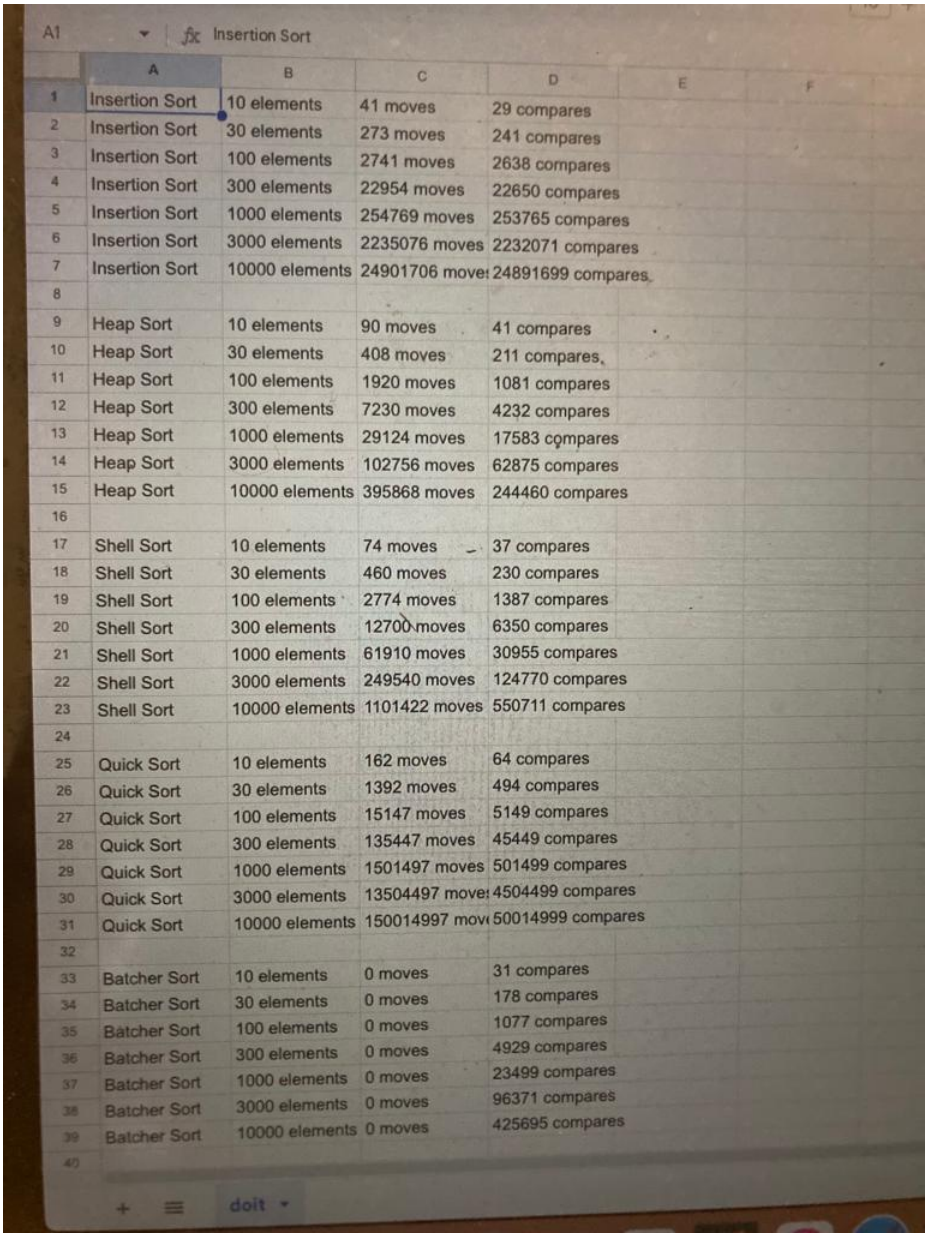
```
    return 0 to exit the program
check if the number of elements needed to print is less than 'length'
    set the number of elements needed to print is equal to 'length'
define an array with the length of the variable 'length'

iterate through the array
    set each element in the array to be bitmasked by 30 bits.
check if the command a is typed by the user
    reset the stats of insertion sort
    print the stats of the insertion sort

    iterate through the array
        print enter for every 5 elements in the sorted array
        print each number in the sorted array
    check if the number of elements needed to be printed is not 0
        print enter
    reset the stats of heap sort
    print the stats of the heap sort

    iterate through the array
        print enter for every 5 elements in the sorted array
        print each number in the sorted array
    check if the number of elements needed to be printed is not 0
        print enter
    reset the stats of shell sort
    print the stats of the shell sort

    iterate through the array
        print enter for every 5 elements in the sorted array
        print each number in the sorted array
    check if the number of elements needed to be printed is not 0
        print enter
    reset the stats of quick sort
    print the stats of the quick sort

    iterate through the array
        print enter for every 5 elements in the sorted array
        print each number in the sorted array
    check if the number of elements needed to be printed is not 0
        print enter
    reset the stats of batcher sort
    print the stats of the batcher sort

    iterate through the array
        print enter for every 5 elements in the sorted array
        print each number in the sorted array
    check if the number of elements needed to be printed is not 0
        print enter
check if the command h is typed by the user
    reset the stats of heap sort
    print the stats of the heap sort

    iterate through the array
        print enter for every 5 elements in the sorted array
```

```
        print each number in the sorted array
    check if the number of elements needed to be printed is not 0
        print enter
check if the command b is typed by the user
    reset the stats of batcher sort
    print the stats of the batcher sort

    iterate through the array
        print enter for every 5 elements in the sorted array
        print each number in the sorted array
    check if the number of elements needed to be printed is not 0
        print enter
check if the command s is typed by the user
    reset the stats of shell sort
    print the stats of the shell sort

    iterate through the array
        print enter for every 5 elements in the sorted array
        print each number in the sorted array
    check if the number of elements needed to be printed is not 0
        print enter
check if the command q is typed by the user
    reset the stats of quick sort
    print the stats of the quick sort

    iterate through the array
        print enter for every 5 elements in the sorted array
        print each number in the sorted array
    check if the number of elements needed to be printed is not 0
        print enter
check if the command i is typed by the user
    reset the stats of insertion sort
    print the stats of the insertion sort

    iterate through the array
        print enter for every 5 elements in the sorted array
        print each number in the sorted array
    check if the number of elements needed to be printed is not 0
        print enter
free the memory in the array
return 0 to exit the program
```

Results

- scan-build: Unfortunately, for scan-build, it didn't find any errors in my code even though I had problems with my print statements and had errors like missing brackets. Hence, I had to type 'make' to find and resolve my errors.
- Performance: From what I've observed, the sorting algorithms have performed well when there are fewer elements in the array. When there are too many elements in the array, the sorting algorithms have to use up much more moves and compares than expected. This seems to be a reoccurrence for each sorting algorithm, so even the algorithms that have the least number of moves and compares have a lot of moves and compares.

- Below I've shown a table of points for each of the sorting algorithms I've implemented. Each sorting algorithm worked successfully.



A1 | Insertion Sort

	A	B	C	D	E	F
1	Insertion Sort	10 elements	41 moves	29 compares		
2	Insertion Sort	30 elements	273 moves	241 compares		
3	Insertion Sort	100 elements	2741 moves	2638 compares		
4	Insertion Sort	300 elements	22954 moves	22650 compares		
5	Insertion Sort	1000 elements	254769 moves	253765 compares		
6	Insertion Sort	3000 elements	2235076 moves	2232071 compares		
7	Insertion Sort	10000 elements	24901706 moves	24891699 compares		
8						
9	Heap Sort	10 elements	90 moves	41 compares		
10	Heap Sort	30 elements	408 moves	211 compares		
11	Heap Sort	100 elements	1920 moves	1081 compares		
12	Heap Sort	300 elements	7230 moves	4232 compares		
13	Heap Sort	1000 elements	29124 moves	17583 compares		
14	Heap Sort	3000 elements	102756 moves	62875 compares		
15	Heap Sort	10000 elements	395868 moves	244460 compares		
16						
17	Shell Sort	10 elements	74 moves	37 compares		
18	Shell Sort	30 elements	460 moves	230 compares		
19	Shell Sort	100 elements	2774 moves	1387 compares		
20	Shell Sort	300 elements	12700 moves	6350 compares		
21	Shell Sort	1000 elements	61910 moves	30955 compares		
22	Shell Sort	3000 elements	249540 moves	124770 compares		
23	Shell Sort	10000 elements	1101422 moves	550711 compares		
24						
25	Quick Sort	10 elements	162 moves	64 compares		
26	Quick Sort	30 elements	1392 moves	494 compares		
27	Quick Sort	100 elements	15147 moves	5149 compares		
28	Quick Sort	300 elements	135447 moves	45449 compares		
29	Quick Sort	1000 elements	1501497 moves	501499 compares		
30	Quick Sort	3000 elements	13504497 moves	4504499 compares		
31	Quick Sort	10000 elements	150014997 moves	50014999 compares		
32						
33	Batcher Sort	10 elements	0 moves	31 compares		
34	Batcher Sort	30 elements	0 moves	178 compares		
35	Batcher Sort	100 elements	0 moves	1077 compares		
36	Batcher Sort	300 elements	0 moves	4929 compares		
37	Batcher Sort	1000 elements	0 moves	23499 compares		
38	Batcher Sort	3000 elements	0 moves	96371 compares		
39	Batcher Sort	10000 elements	0 moves	425695 compares		
40						

doit