Assignment 4 – Sets and Sorting

Asaveri Rao

CSE 13S – Fall 2023

Purpose

This assignment is a program that sorts through sets through various formats. There are two main types of functions that need to be implemented, sets and sorting algorithms. Size bit wise operators are used to implement nine different functions within sets to define nine different functionalities. An empty set, a full set, set insertion, removal, member, union, intersection, complement, and difference. These are the fundamental functions that need to be defined for set, in order for the sorting algorithms to execute smoothly. Without getting into too many details about the sorting algorithms in this particular section, the main purpose of them is to help organize items in the sets in an organized manner.

How to Use the Program

The basic premises of the of the assignment is pretty simple, which is why the way to use the program is also relatively simple. There are a few elementary steps that need to take place in order for the program to execute properly. One of them being the compilation of the files, this is done through the Makefile. To run the makefile so that the code can execute, the programmer had to run make, make format, make clean, and make release in order for the file to be cleaned and compiled correctly. After these commands are run in the command line and no errors are output, that means the files were successfully compiled. Now the user can run the program by running ./sorting in the command line to get an option of all the different commands they can ask for. Depending on what type of sorting algorithm they want to run, they can run -i, -a, -q, etc.

You should also describe any optional flags that your program uses, and what they do.

Program Design

This program has two major function types that are implemented as mentioned before. The first one are sets. Their main purpose is to keep track of the command line options and to work with them, bit wise operations can be used to do so. The second major type of algorithm that is used are sorting algorithms. In this specific program, all five sorting algorithms are utilized. The general structure of the program is as follows, with the predefined header files, they are called in their corresponding c files, which contain the functions to execute the sorting algorithms. There is also a sets.h file which has a corresponding sets.c file which contains code that has functions that runs the different operations.

Data Structures

As mentioned before, the data structures used are mainly sets, arrays and within those integers, strings, and functions. Here are the brief descriptions of each function used in set.c:

```
Define the the Set data type

Set set_empty(){
    function that returns empty set
}
```

```
Set set_universal(){
    function that returns a universal set(a set that has all 1 bits)

}

bool set_member(Set s, int x){
    function that checks if an element is apart of the set
}

Set set_insert(Set s, int x){
    function that adds/inserts an element into the set
}

Set set_remove(Set s, int x){
    function that removes an element from the set
}

Set set_union(Set s, Set t){
    function that calculates the union between two sets
}

Set set_intersect(Set s, Set t){
    function that calculates the intersection of two sets
}

Set set_difference(Set s, Set t){
    function that calculates the difference between two sets
}
Set set_complement(Set s){
    function that calcualates the complement between two sets
}
```

Algorithms

This section will show pseudo code for each sorting algorithm. Insert.c:

```
procedure insertion_sort(stats, arr, length)
    for k from 1 to length - 1
        key = arr[k]
        j = k - 1

        while j >= 0 and cmp(stats, arr[j], key) > 0
            arr[j + 1] = move(stats, arr[j])
            j = j - 1

        arr[j + 1] = move(stats, key)
    end for
end procedure
```

Shell.c:

```
procedure shell_sort(stats, A, length)
    for gap_index from 0 to GAPS - 1
```

```
        gap = gaps[gap_index]

        for k from gap to length - 1
            temp = move(stats, A[k])
            j = k

            while j >= gap and cmp(stats, A[j - gap], temp) > 0
                A[j] = move(stats, A[j - gap])
                j = j - gap

            A[j] = move(stats, temp)
        end for
    end for
end procedure
```

Heap.c:

```
procedure max_child(stats, A, first, last)
    left = 2 * first + 1
    right = 2 * first + 2
    if right <= last and cmp(stats, A[right], A[left]) > 0
        return right
    return left
end procedure

procedure fix_heap(stats, A, first, last)
    done = false
    parent = first
    while 2 * parent + 1 <= last and not done
        largest_child = max_child(stats, A, parent, last)
        if cmp(stats, A[parent], A[largest_child]) < 0
            swap(stats, A[parent], A[largest_child])
            parent = largest_child
        else
            done = true
        end if
    end while
end procedure

procedure build_heap(stats, A, first, last)
    if last > 0
        for parent from (last - 1) / 2 down to first
            fix_heap(stats, A, parent, last)
        end for
    end if
end procedure

procedure heap_sort(stats, A, n)
    first = 0
    last = n - 1
    build_heap(stats, A, first, last)
    for leaf from last down to first + 1
        swap(stats, A[first], A[leaf])
        fix_heap(stats, A, first, leaf - 1)
```

```
    end for
end procedure
```

Batcher.c:

```
procedure comparator(stats, A, x, y)
    if cmp(stats, A[x], A[y]) > 0
        swap(stats, A[x], A[y])
    end if
end procedure

procedure batcher_sort(stats, A, n)
    if n = 0
        return
    end if

    t = ceil(log2(n))
    p = 2^(t - 1)

    while p > 0
        q = 2^(t - 1)
        r = 0
        d = p

        while d > 0
            for i from 0 to n - d - 1
                if (i & p) = r
                    comparator(stats, A, i, i + d)
                end if
            end for
            d = q - p
            q = q / 2
            r = p
        end while
        p = p / 2
    end while
end procedure
```

Quick.c:

```
procedure partition(stats, A, lo, hi)
    pivot = A[hi]
    i = lo - 1

    for j from lo to hi - 1
        if cmp(stats, A[j], pivot) < 0
            i = i + 1
            swap(stats, A[i], A[j])
        end if
    end for
    i = i + 1
    swap(stats, A[i], A[hi])
    return i
end procedure
```

```
procedure quick_sorter(stats, A, lo, hi)
    if lo < hi
        p = partition(stats, A, lo, hi)
        quick_sorter(stats, A, lo, p - 1)
        quick_sorter(stats, A, p + 1, hi)
    end if
end procedure

procedure quick_sort(stats, A, n)
    quick_sorter(stats, A, 0, n - 1)
end procedure
```

Function Descriptions

I have a main function in a file named sorting.c that primarily executes all the sorting algorithms along with the bit wise operators on the sets given. This is the pseudo code for the current main function, labeled sorting.c.

```
function print_help(program_name):
    Print program usage and options.

function print_array(arr, arr_size, print_count):
    For i in range 0 to min(arr_size, print_count):
        Print arr[i] with formatting
        If (i + 1) is a multiple of 5:
            Print a newline
    If arr_size is not a multiple of 5:
        Print a newline

function main(argc, argv):
    Initialize arr_size to 100
    Initialize print_count to 100
    Initialize seed to 13371453
    Initialize run_algorithms as an array of 5 booleans, all set to false

    Process command line options using getopt:
    While there are more options to process:
        If option is 'H':
            Call print_help(argv[0])
            Return 0
        Else if option is 'a':
            Set all elements in run_algorithms array to true
        Else if option is 'i':
            Set run_algorithms[0] to true
        Else if option is 'h':
            Set run_algorithms[1] to true
        Else if option is 's':
            Set run_algorithms[2] to true
        Else if option is 'q':
            Set run_algorithms[3] to true
        Else if option is 'b':
            Set run_algorithms[4] to true
        Else if option is 'n':
```

```
            Set arr_size to atoi(optarg)
        Else if option is 'p':
            Set print_count to atoi(optarg)
        Else if option is 'r':
            Set seed to atoi(optarg)
        Else:
            Print usage message
            Exit with failure

    If no sorting algorithm is selected:
        Call print_help(argv[0])
        Return 1

    Allocate memory for an integer array arr of size arr_size
    Create a Stats object called stats
    Seed the random number generator with seed

    Fill the array arr with random integers

    If run_algorithms[0]:
        Call insertion_sort with stats, arr, and arr_size
        Print sorting statistics for Insertion Sort
        Call print_array(arr, arr_size, print_count)

    If run_algorithms[1]:
        Reset the statistics in stats
        Call heap_sort with stats, arr, and arr_size
        Print sorting statistics for Heap Sort
        Call print_array(arr, arr_size, print_count)

    If run_algorithms[2]:
        Reset the statistics in stats
        Call shell_sort with stats, arr, and arr_size
        Print sorting statistics for Shell Sort
        Call print_array(arr, arr_size, print_count)

    If run_algorithms[3]:
        Reset the statistics in stats
        Call quick_sort with stats, arr, and arr_size
        Print sorting statistics for Quick Sort
        Call print_array(arr, arr_size, print_count)

    If run_algorithms[4]:
        Reset the statistics in stats
        Call batcher_sort with stats, arr, and arr_size
        Print sorting statistics for Batcher Sort
        Call print_array(arr, arr_size, print_count)

    Free the memory allocated for arr
    Return 0
```

Results

After fixing minor errors, the main function in the sorting.c file is executing successfully with no errors, meaning the program is also executing successfully. Some errors that could have taken place There are a few false positives that can occur as a result of scan-build. They consist of are false positives due to the scan build function. However this was mitigated through making sure there were no uninitialized variables, array bound violations, and memory or resource leaks. Each sorting algorithm had differing outputs. For example, the insertion sort seemed to be a little more efficient in terms of smaller arrays, however quicksort seemed to be more efficient and faster with longer arrays. The heap, shell, and batcher sorting algorithm seemed to be in a similar ballpark in terms of performance. They were still all efficient, however somewhat slower than the others. I observed that when the arrays were already somewhat sorted, it made the performance time much shorter and much more efficient especially for insert sort. As for the conditions in which the sorts perform the worst under, I would say when there is a reverse ordered array, it makes it harder for the sort algorithms to execute efficiently. Overall, I would say based off this assignment, not for all scenarios, but for most, insertion sort is the most efficient and fastest sort algorithm. I've also included screenshots of the results down below. The graph I've inserted below is one that compares all five different sorting algorithms and their data for a few specific test cases, like a reverse array, a long array, and a short array. As can be seen by the graph for this particular instance, the insert sorting algorithm performs exponentially better than the other algorithms.

```
asa@asarao:~/cse13s/asgn4$ ./sorting -i
Insertion Sort, 100 elements, 2642 moves, 2638 compares
      8032304     34732749     42067670     54998264     56499902
     57831606     62698132     73647806     75442881    102476060
    104268822    111498166    114109178    134750049    135021286
    176917838    182960600    189016396    194989550    200592044
    212246075    243082246    251593342    256731966    261742721
    281272176    282549220    287277356    297461283    331368748
    334122749    343777258    370030967    391223417    398173317
    426152680    433486081    438071796    444703321    447975914
    451764437    455275424    460885430    464871224    473260275
    500293632    510040157    518072461    521864874    522702830
    527207318    530718305    530735134    538219612    573093082
    579453371    587189713    607875172    611422544    616902904
    620182312    629948093    630759321    648567958    689665138
    708948898    738166936    744868500    754364921    782250002
    783550802    783585680    855167780    860725547    868766010
    908068554    910310679    919290914    920038191    923423680
    934604298    935579555    944225142    950136224    954916333
    965680864    966879077    988526615    989854347    994582085
    995796877    999105042   1018598925   1025188081   1037080358
   1037686539   1048807596   1054405046   1057925624   1072766566
asa@asarao:~/cse13s/asgn4$ ./sorting -h
Heap Sort, 100 elements, 1755 moves, 1029 compares
      8032304     34732749     42067670     54998264     56499902
     57831606     62698132     73647806     75442881    102476060
    104268822    111498166    114109178    134750049    135021286
    176917838    182960600    189016396    194989550    200592044
    212246075    243082246    251593342    256731966    261742721
    281272176    282549220    287277356    297461283    331368748
    334122749    343777258    370030967    391223417    398173317
    426152680    433486081    438071796    444703321    447975914
    451764437    455275424    460885430    464871224    473260275
    500293632    510040157    518072461    521864874    522702830
    527207318    530718305    530735134    538219612    573093082
    579453371    587189713    607875172    611422544    616902904
    620182312    629948093    630759321    648567958    689665138
    708948898    738166936    744868500    754364921    782250002
    783550802    783585680    855167780    860725547    868766010
    908068554    910310679    919290914    920038191    923423680
    934604298    935579555    944225142    950136224    954916333
    965680864    966879077    988526615    989854347    994582085
    995796877    999105042   1018598925   1025188081   1037080358
   1037686539   1048807596   1054405046   1057925624   1072766566
asa@asarao:~/cse13s/asgn4$
```

Figure 1: Screenshot of -i and -h command

```
asa@asarao:~/cse13s/asgn4$ ./sorting -q
Quick Sort, 100 elements, 1053 moves, 640 compares
      8032304     34732749     42067670     54998264     56499902
     57831606     62698132     73647806     75442881    102476060
    104268822    111498166    114109178    134750049    135021286
    176917838    182960600    189016396    194989550    200592044
    212246075    243082246    251593342    256731966    261742721
    281272176    282549220    287277356    297461283    331368748
    334122749    343777258    370030967    391223417    398173317
    426152680    433486081    438071796    444703321    447975914
    451764437    455275424    460885430    464871224    473260275
    500293632    510040157    518072461    521864874    522702830
    527207318    530718305    530735134    538219612    573093082
    579453371    587189713    607875172    611422544    616902904
    620182312    629948093    630759321    648567958    689665138
    708948898    738166936    744868500    754364921    782250002
    783550802    783585680    855167780    860725547    868766010
    908068554    910310679    919290914    920038191    923423680
    934604298    935579555    944225142    950136224    954916333
    965680864    966879077    988526615    989854347    994582085
    995796877    999105042   1018598925   1025188081   1037080358
   1037686539   1048807596   1054405046   1057925624   1072766566
asa@asarao:~/cse13s/asgn4$ ./sorting -b
Batcher Sort, 100 elements, 1209 moves, 1077 compares
      8032304     34732749     42067670     54998264     56499902
     57831606     62698132     73647806     75442881    102476060
    104268822    111498166    114109178    134750049    135021286
    176917838    182960600    189016396    194989550    200592044
    212246075    243082246    251593342    256731966    261742721
    281272176    282549220    287277356    297461283    331368748
    334122749    343777258    370030967    391223417    398173317
    426152680    433486081    438071796    444703321    447975914
    451764437    455275424    460885430    464871224    473260275
    500293632    510040157    518072461    521864874    522702830
    527207318    530718305    530735134    538219612    573093082
    579453371    587189713    607875172    611422544    616902904
    620182312    629948093    630759321    648567958    689665138
    708948898    738166936    744868500    754364921    782250002
    783550802    783585680    855167780    860725547    868766010
    908068554    910310679    919290914    920038191    923423680
    934604298    935579555    944225142    950136224    954916333
    965680864    966879077    988526615    989854347    994582085
    995796877    999105042   1018598925   1025188081   1037080358
   1037686539   1048807596   1054405046   1057925624   1072766566
```

Figure 2: Screenshot of -q and -h command

```
asa@asarao:~/cse13s/asgn4$ ./sorting -s
Shell Sort, 100 elements, 3025 moves, 1575 compares
      8032304     34732749     42067670     54998264     56499902
     57831606     62698132     73647806     75442881    102476060
    104268822    111498166    114109178    134750049    135021286
    176917838    182960600    189016396    194989550    200592044
    212246075    243082246    251593342    256731966    261742721
    281272176    282549220    287277356    297461283    331368748
    334122749    343777258    370030967    391223417    398173317
    426152680    433486081    438071796    444703321    447975914
    451764437    455275424    460885430    464871224    473260275
    500293632    510040157    518072461    521864874    522702830
    527207318    530718305    530735134    538219612    573093082
    579453371    587189713    607875172    611422544    616902904
    620182312    629948093    630759321    648567958    689665138
    708948898    738166936    744868500    754364921    782250002
    783550802    783585680    855167780    860725547    868766010
    908068554    910310679    919290914    920038191    923423680
    934604298    935579555    944225142    950136224    954916333
    965680864    966879077    988526615    989854347    994582085
    995796877    999105042   1018598925   1025188081   1037080358
   1037686539   1048807596   1054405046   1057925624   1072766566
```

Figure 3: Screenshot of -s command

```
asa@asarao:~/cse13s/asgn4$ ./sorting -a -p 1
Insertion Sort, 100 elements, 2642 moves, 2638 compares
      8032304
Heap Sort, 100 elements, 1920 moves, 1081 compares
      8032304
Shell Sort, 100 elements, 2774 moves, 1387 compares
      8032304
Quick Sort, 100 elements, 15147 moves, 4950 compares
      8032304
Batcher Sort, 100 elements, 0 moves, 1077 compares
      8032304
```

Figure 4: Screenshot of one of the test cases in harness

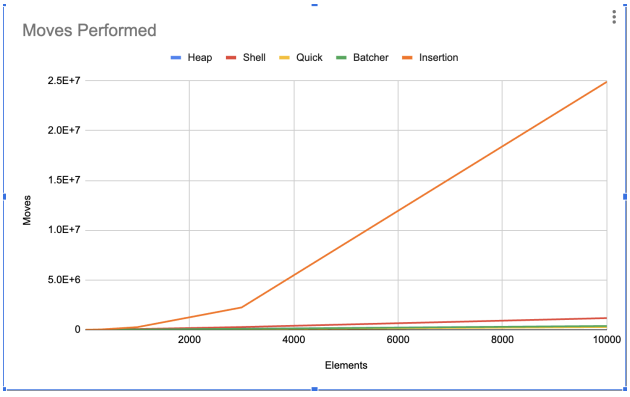

Figure 5: This is a graph that compares all the sort algorithms together.