Assignment 3 – Sets and Sorts

Noah Meisel

CSE 13S – Spring 2023

Purpose

This program tests the effectiveness of five different sorting algorithms;insertion sort, shell sort,heap sort quicksort, and Batchers odd-even merge sort. The effectiveness in this case means the number of comparisons/swaps it takes for an array of size n to be sorted. This data can be printed to the console for each algorithm, along with the sorted array itself.

How to Use the Program

All of the algorithms can be run through the sorting.c test harness program with the following command line options.

- -a, runs all four sorting algorithims
- -i, runs Insertion Sort.
- -s, runs Shell Sort
- -q, runs Quicksort
- -b, runs Batcher Sort
- -r seed: sets the seed for random to seed. Default seed is 13371453
- -n size: sets the size of the array to be sorted to size. Default size is 100
- -p elements: prints elements number of elements from the array.If elements is greater than or equal to the size of the array, all of the array elements will be printed. Default is 100.
- -H, prints out a help message specifying the programs usage.

Program Design

This program revolves around the four sorting algorithms all tied together in sorting.c. The algorithms can be run by the user of the program(see above) through this main c file. The command-line arguments are handled by using a set(see data structures below) to check whether each command-line option is in the set or not in the set. After the command line arguments specified are added to the set through the use of the conventional getopt() switch statement loop, and then passed into a series of if statements to run the desired algorithims. The actual algorithms are also implemented with the help of a small statistics library specified in stats.c through the use of the cmp, move, and swap functions. The specifics of each function can be found below in the Function Descriptions Section. The gist of their use is to replace some common operations in sorting(comparison and swapping) with a function that does that operation while also counting each time that operation is performed. Through these functions, we can compare the performance of each sorting algorithm.

Data Structures

Sets: The Set data structure is designed to act as a simple method of determining whether a command-line option has been specified by the user. The way the set works is by having an 8 bit number where each bits index corresponds to a specific arguments inclusion/exclusion from the set. In this case, -i is index 0, -s is index 1, -h is index 2, -q is index 3, and -b is index 4. When sorting.c runs, the first thing it will do for each command line option is insert it into the set using the set_insert() function. Once this is done for each of the algorithms, we will then have an 8 bit number where the existence of a 1 in a particular bit position signifies that it's corresponding command-line option is to be executed by the rest of the function.

Stats: The Stats structure is a simple structure that stores the number of moves and the number of comparisons made by each sorting algorithm.

Algorithms

The following pseudo code for all of the algorithms is taking from the Assignment 3 document[1]. There were some slight adjustments made to each, mostly to include the statistics functions.

```
INSERTION SORT ALGORITHIM
Insertion Sort(A):
    for k in range 1 to length of the array A:
        j = k
        temporary= A[k]
        while j > 0 and (cmp(pstats,temp, A[j-1])) < 0:
            A[j] = move(pstats, A[j-1])
            j -= 1
        A[j] = temp
```

```
SHELL SORT ALGORITHIM

shell_sort(A):
    for gap_index in range GAPS(142):
        gap = gaps[gap_index]
        for i in range size of A:
            j = i
            temp = A[i]
            while j >= gap and cmp(stats, temp, A[j-gap]) < 0:
                A[j] = move(stats, A[j-gap])
                j  = j - gap
            A[j] = temphttps://www.overleaf.com/project/645492d13caf74a845c44065
```

```
QUICKSORT ALGORITHIM

partition_function(A, low, high):
    i = low -1
    for j in range low to high:
        if cmp(stats, A[j-1], A[high -1]) < 0:
            i + = 1
            swap(stats, A[i-1], A[j-1]
    swap(stats, A[i], A[high-1])
    return i + 1
quick_sorter(A, low, high):
    if low < high:
        p = partition(A, low, high)
        quick_sorter(A, low, p -1)
```

```
        quick_sorter(A, p + 1, high)

quick_sort(A):
    quick_sorter(A, 1, length of A)
```

```
HEAP SORT ALGORITHIM

max_child(A, first, last):
    left = 2 * first
    right - left + 1
    if right <= last and A[right -1 ] > A[left - 1]:
        return right
    return left

fix_heap(A, first, last):
    found = False
    mother = first
    great = max_child(A, mother, last)

    while mother <= int(last / 2) and found = False:
        if cmp(stats, A[mother -1 ], A[great - 1]:
            A[mother-1], A[great - 1] = A[great - 1] = A[great -1], A[mother -1]
            mother = great
            great = max_child(A, mother, last)
        else:
            found = True
build_heap(A, first, last):
    for father in range(int(last/2), first -1, -1):
        fix_heap(A, father, last)
heap_sort(A):
    first = 1
    last = length of A
    build_heap(A, first, last):
    for leaf in rangee(last, first, -1):
        swap(stats, A[first - 1], A[leaf -1])
        fix_heap(A, first, leaf -1)
```

```
BATCHERS SORT ALGORITHIM

comparator(A, x, y):
if cmp(stats, A[y], A[x]) < 0:
    swap(stats, A[x], A[y])

batcher_sort(A):
    if length of A == 0:
        return
    n = length of A
    t = sizeof(n)/8
    p = 1 << (t-1)

    while p > 0:
        q = 1 << (t-1)
        r = 0
```

```
    d = p

    while d > 0:
        for i in range(0, n -d):
            if (i & p) == r:
                comparator(A, i, i + d)
        d = q - p
        q >> = 1
        r - p
    p >>= 1
```

Function Descriptions

Set Functions: Set set_empty(void) -Take no arguments and returns an empty set(0).

Set set_universal(void) -Takes no argument and returns a full set(255).

Set set_insert(Set s, uint8_t x) -Takes a Set and a number that represents the element you want to be included in the set. For instance, if I wanted to include -i in the set I would run set_insert(set_name, 0). The returned value is the set with the new value in it.

Set set_remove(Set, uint8_t x) –Takes a Set and a number that represents the element you want to be removed from the set. For instance, if I wanted to remove -i from the set I would run set_remove(set_name, 0). The returned value is the set with specified value removed.

Set set_member(Set s, uint8_t x) -takes a Set and checks if a number that represents the an element is inside the set.Returns True if true, otherwise False

Set set_union(Set s, Set t) -Combines two different Sets, s and t and returns them as one set.

Set set_intersect(Set s, Set t) -combines two different Sets, s and t, and returns their intersection

Set set_different(Set s, Set t) -takes a Set s and a Set t and removes all elements of Set t from Set s. Returns this new set.

Set set_complement(Set s) -takes a Set s and returns it's complement, or in other words the set where all things not in S are in this new set, and all things in S are not in this new set.

Sorting Algorithms: Quicksort, Insertionsort, Shell Sort, Batcher Sort, and Heap sort all take in a Stats pointer, a pointer to an array, and the length of the array being pointed to. None of the above functions return anything, but they do modify the variable stats and sort the array passed by reference to the function with the pointer.

Statistics Functions:

int cmp(Stats *stats, uint32_t x, uint32_t y) -takes in stats instance, and two integers. Returns -1 if x ¡ y, 0 if x == y, and 1 if x ¿ y.

uint32_t move(Stats *stats, uint32_t x) -Takes in stats instance and an element to be swapped with another variable. Increments the moves field in stats, and then returns x.

void swap(Stats *stats, uint32_t *x, uint32_t *y) -Swaps the elements pointed to by x and y. Increments the move field in stats by 3.

void reset(Stats *stats) -sets moves and comparisons to 0 in stats.

void print_stats(Stats *stats, const char *algorithm_name, uint32_t num_elements) -takes in a pointer stats to a statistics instance, a sorting algorithm name, and the number of elements in the array. Prints this information out in readable format.

void make_array(uint32_t seed, uint32_t size, uint32_t *ArrayMemory) -takes in a seed value to pass to the srandom function, the size of the array, and a chunk of memory to create the array at. This function returns nothing, but modifies the already existing array.

Results

All five sorting algorithms were implemented as intended and sort the arrays correctly, as shown by figure 1. The pseudo code provided made the implementation of all the algorithms a relatively simple process. The most challenging part of the project was the creation of the test bench to run the algorithms from. It took quite a while to understand exactly how to make the arrays using malloc()/calloc() but after that everything was relatively simple. One issue with the programming I would like to fix with more time is the if statement block I created. I recognize that could likely be put into a function to significantly decrease the code bloat, but I left it as was because I had already finished it before I realized I should change that.

When it comes to the performance of the algorithms, the Figure 2 in Numeric Results highlights how both Insertion Sort and Shell sort perform very well for smaller arrays(10-40 elements) but very quickly begin to struggle as the number of elements increases. Batch er Sort seems to perform well at all array sizes, though is slightly outperformed by quick sort as we move closer to 100 elements. Heap Sort performs almost exactly in the middle for the Shell/Insertion sort pairing and the Batcher/Quick sort group. In terms of the graphs shape, Insertion sort and Shell sort look parabolic, Heap sort look linear, and Quick sort and Batcher sort look slightly logarithmic. The scale of this graph makes it easier to see how quickly Shell/Insertion sort get outperformed by the other sorts, but the trade off is the shape is slightly undefined still.

Error Handling

The pipeline passes with no errors, but there are some limitations to the program. When given 1000000000 as the number of elements, the program will experience a seg-fault and crash. This is almost certainly due to the computer not having enough memory to run that large of a sort though, as numbers slightly smaller than that run though it can take a while to run depending on how many sorts are occurring. Another potential issue with the program is that a negative input for the seed, size, or print will return an error.

Numeric results

References

[1] Darrell Long Kerry Veenstra. *Sets and Sorts*. UCSC, Santa Cruz, CA, Spring, 2023.

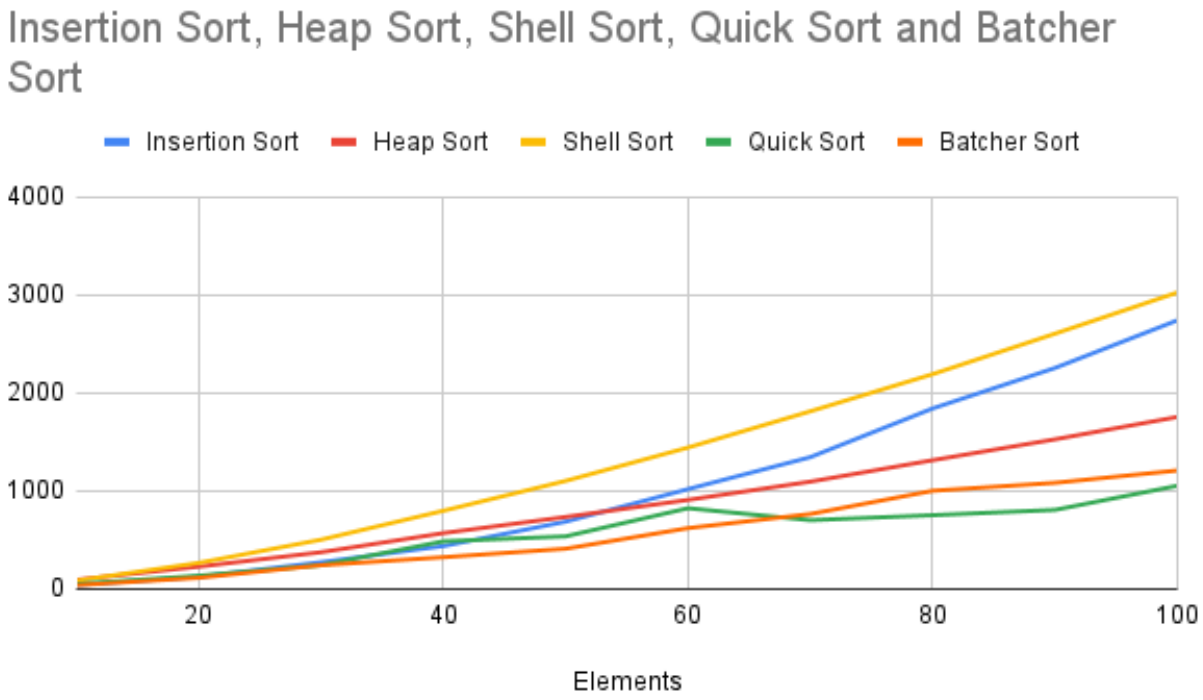

Figure 1: Graph of the algorithms moves in relation to number of elements

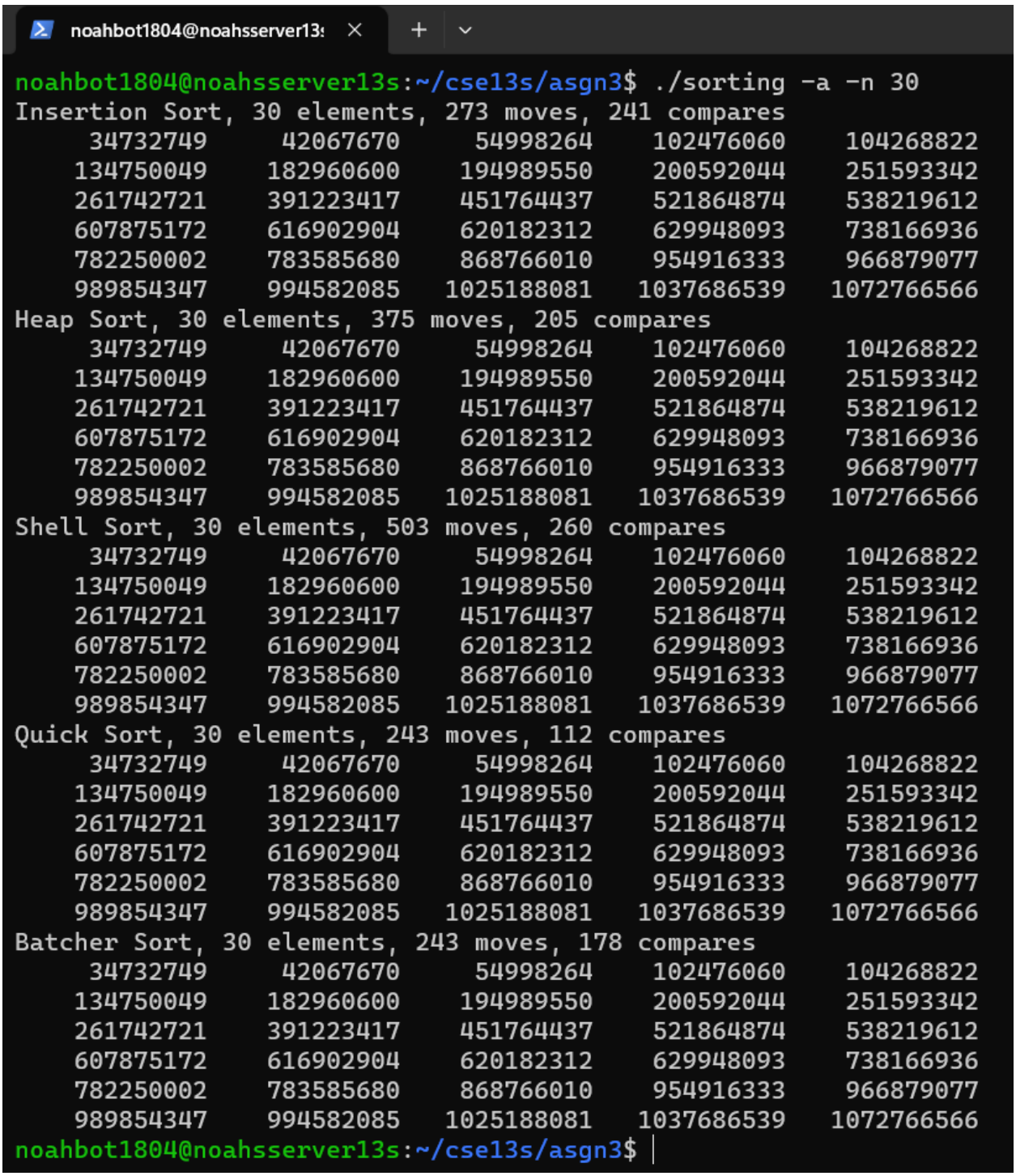

Figure 2: Image of program output for all algorithims doing a 30 element sort