Assignment 3 – Sets and Sorting

Mateo Garcia

CSE 13S – Spring 2023

Purpose

The purpose of this program is to allow the user to sort list of random numbers and also have a set of functions to perform set operations. The to-be sorted list will be made up of randomly generated numbers from a seed the user inputs, with a size also inputted from the user. There are 5 different sorts that user can choose: insertion sort, shell sort, heap sort, quick sort, and batcher merge sort. There is also a statistics file that will show the amount of comparisons, and operations, in order to better understand these sorting algorithms. The user can also specify the amount of elements the user wants to print out from the array. As for the sets, the set operations will be used for insertion, removing values, checking for membership, finding union or intersection, or finding the difference or complement. These operations will be used to identify the flags in the command line.

How to Use the Program

First you must go into the cse13s/asgn3 folder where all your files are located. Then run the command **make** and then you can run the main command. By running **./sorting *-[flag1] -[flag2]...*** you can add 10 different options to tell the program which sorting algorithm to use, as well as input the size of your unsorted array, and a seed to fill it with random numbers. There are also flags to print out number of elements of the array, and a help option.

- -a : Employs *all* sorting algorithms implemented
- -i : Enables Insertion Sort
- -s : Enables Shell Sort
- -h : Enables Heap Sort
- -q : Enables Quick Sort
- -b : Enables Batcher Sort
- -r [num] : Set the random seed to num. The default seed is 13371453.
- -n [num] : Set the array size to num. The default size is 100.
- -p [elements] : Enable printing of statistics to see computed terms and factors for all tested functions, elements is the number of computed terms.
- -h : Displays a help message.

Program Design

Data Structures

There are only two big data structures used for this assignment. Sets, and arrays. The arrays will contain the unordered elements to be used for sorting, and will also hold the sorted elements after the algorithms are executed. As for sets, they use 8 bit values that (1 or 0) and denote which operations to execute. Other data structures used are unsigned 32 bit ints for values inside the arrays, and the random function to create the unordered elements in the array. The program will also use getopt() to pass in the flags.

Algorithms

Each sorting algorithm has it's own algorithm to sort the sorted array.

```
def insertion_sort(A: list):
    for k in range(1, len(A)):
        j = k
        temp = A[k]
        while j > 0 and temp < A[j - 1]:
            A[j] = A[j - 1]
            j -= 1
        A[j] = temp
```

```
def shell_sort(arr):
    for gap in gaps:
        for i in range(gap, len(arr)):
            j = i
            temp = arr[i]
            while j >= gap and temp < arr[j - gap]:
                arr[j] = arr[j - gap]
                j -= gap
            arr[j] = temp
```

```
def partition(A: list, lo: int, hi: int):
    i = lo - 1
    for j in range(lo, hi):
        if A[j - 1] < A[hi - 1]:
            i += 1
            A[i - 1], A[j - 1] = A[j - 1], A[i - 1]
    A[i], A[hi - 1] = A[hi - 1], A[i]
    return i + 1
}
# A recursive helper function for Quicksort.
def quick_sorter(A: list, lo: int, hi: int):
    if lo < hi:
        p = partition(A, lo, hi)
        quick_sorter(A, lo, p - 1)
        quick_sorter(A, p + 1, hi)

def quick_sort(A: list):
    quick_sorter(A, 1, len(A))
```

```
def build_heap(A: list, first: int, last: int):
    for father in range(last // 2, first - 1, -1):
```

```
        fix_heap(A, father, last)

def heap_sort(A: list):
    first = 1
    last = len(A)
    build_heap(A, first, last)
    for leaf in range(last, first, -1):
        A[first - 1], A[leaf - 1] = A[leaf - 1], A[first - 1]
        fix_heap(A, firs

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

def fix_heap(A: list, first: int, last: int):
    found = False
    mother = first
    great = max_child(A, mother, last)
    while mother <= last // 2 and not found:
        if A[mother - 1] < A[great - 1]:
            A[mother - 1], A[great - 1] = A[great - 1], A[mother - 1]
            mother = great
            great = max_child(A, mother, last)
        else:
            found = True
```

```
def comparator(A: list, x: int, y: int):
    if A[x] > A[y]:
        A[x], A[y] = A[y], A[x] # Swap A[x] and A[y]

def batcher_sort(A: list):
    if len(A) == 0:
        return
    n = len(A)
    t = n.bit_length()
    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
```

As for the sorting.c file, this is the pseudocode:

```
int main(void)
    run getopt
        turn on flags using sets, each bit of a set used to represent if a flag was set or not, 1 = on,
    loop from to number of sorts:
        srandom(seed) // to get your random values
        make array using malloc and fill with values using random()
        run sorting algorithms depending on which flags were turned on
        print stats
        free memory so you can make array again in next loop
    return 0;
```

Function Descriptions

There are several modules and functions that are part of this program.

The set module will have these functions:

- **Set set_empty(void)** This function is used to return an empty set. In this context, an empty set would be a set in which all bits are equal to 0.
- **Set set_universal(void)** This function is used to return a set in which every possible member is part of the set.
- **Set set_insert(Set s, int x)** This function takes x and inserts into set s.
- **Set set_remove(Set s, int x)** This function removes x from s.
- **bool set_member(Set s, int x)** This function returns a boolean (true or false), whether x is in set s or not.
- **Set set_union(Set s, Set t)** Using the OR operator this function returns all the values (non-duplicate) appear in each sets.
- **Set set_intersect(Set s, Set t)** Using the AND operator this function returns all the values that are shared between the sets.
- **Set set_difference(Set s, Set t)** This function returns returns a set of the elements of set s that are not in set t.
- **Set set_complement(Set s)** This function is used to return the complement of a set.

The sorting algorithms will have their own files and own sorting functions and all their functions can be found in Algorithms.

- **Insertion Sort** will just have the insertion function.
- **Shell Sort** will just have the shell sort function.
- **Quick Sort** will have a recursive function for quick sort, and a partition function. The partition is used to place all the elements less than the pivot in the left part of the array, and all elements greater than the pivot in the right part of the array.
- **Heap Sort** will have a max_child function, fix_heap function, build-heap function, and the heap_sort function itself.
- **Batcher Sort** will have the batcher sort function and a comparator function.

Results

My code successfully achieves the desired output. If anything could be improved, it would be efficiency with the for loop checking if a flag was turned on. I learned a lot about where the comparisons are made for each sorting algorithm. And especially the more complicated ones, how different each one approaches the problem of sorting. I ran my graphs on desmos so I was not able to put labels on it but the x-axis is Elements, and the y-axis are the number of Moves. So each graph is Num of Moves vs Elements. I have found out that insertion sort takes the longest when you add a lot more elements, and quick sort is very efficient when the partitions are balanced.

Error Handling

The errors I encountered were not properly using my set functions when checking which flags were turned on. I had forgotten that the insert_set() function returned a set instead of changing the actual set because C is call by value. As soon as I realised this my program worked. Thank you TA Ben.

Numeric results

Here are screenshots of my outputs and graphs

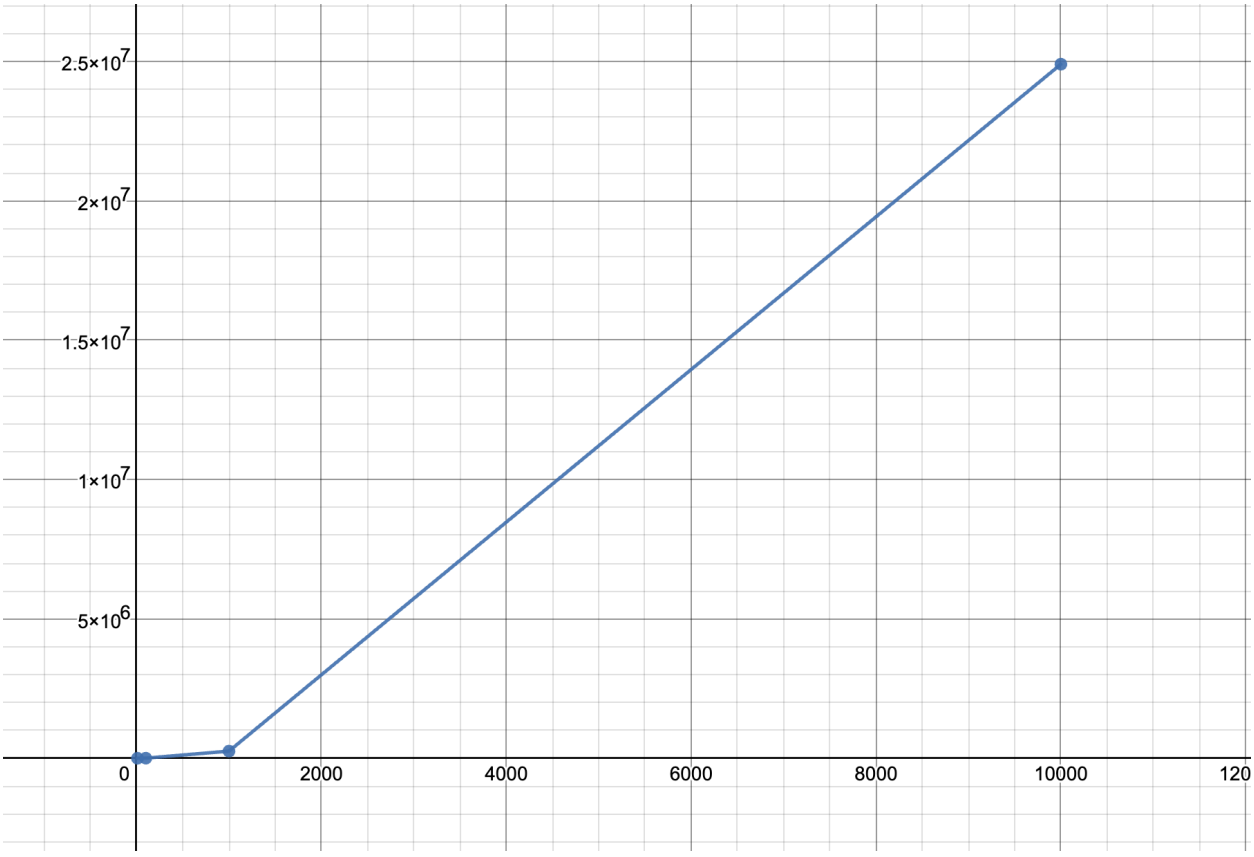

Figure 1: Insertion Sort

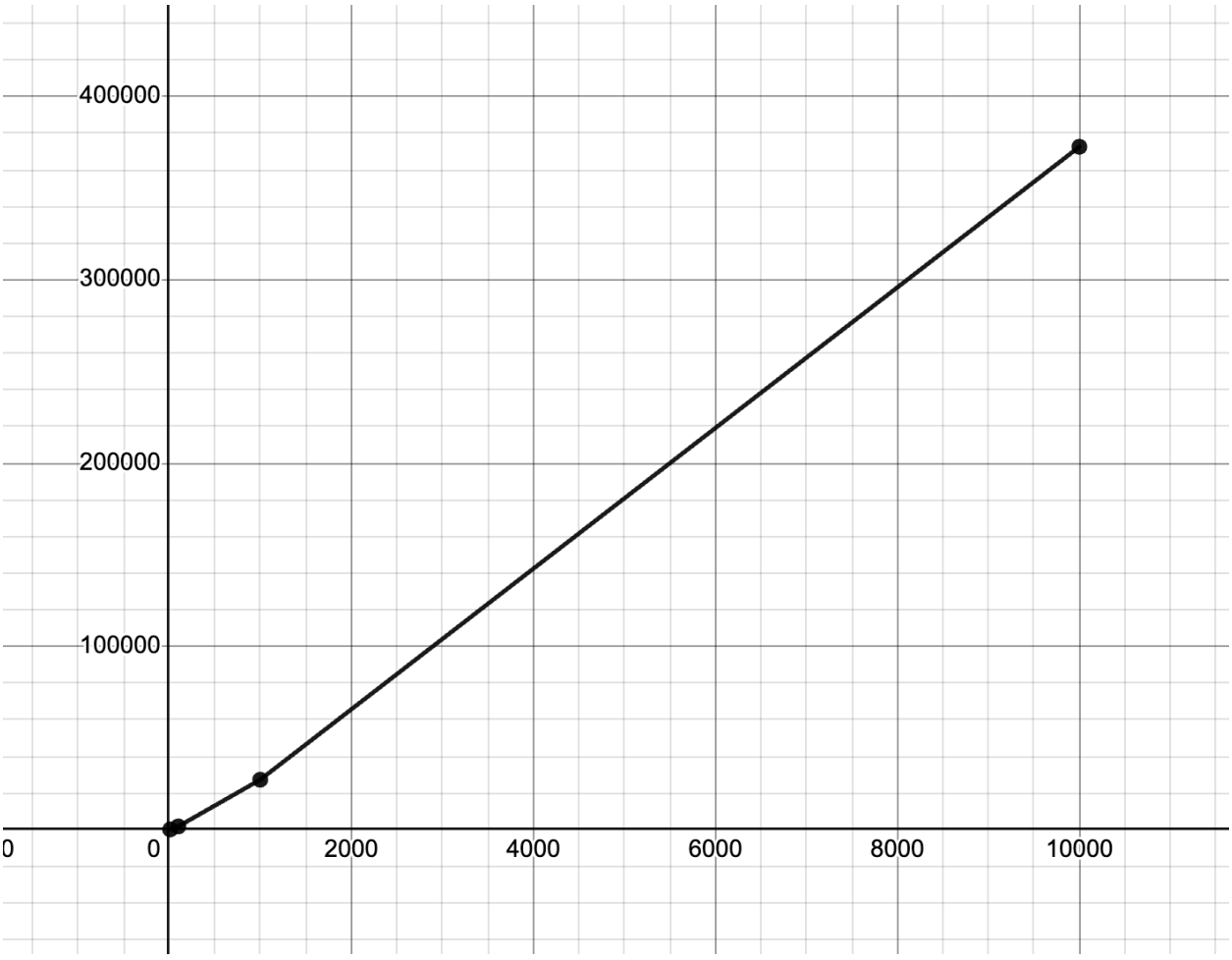

Figure 2: Heap Sort

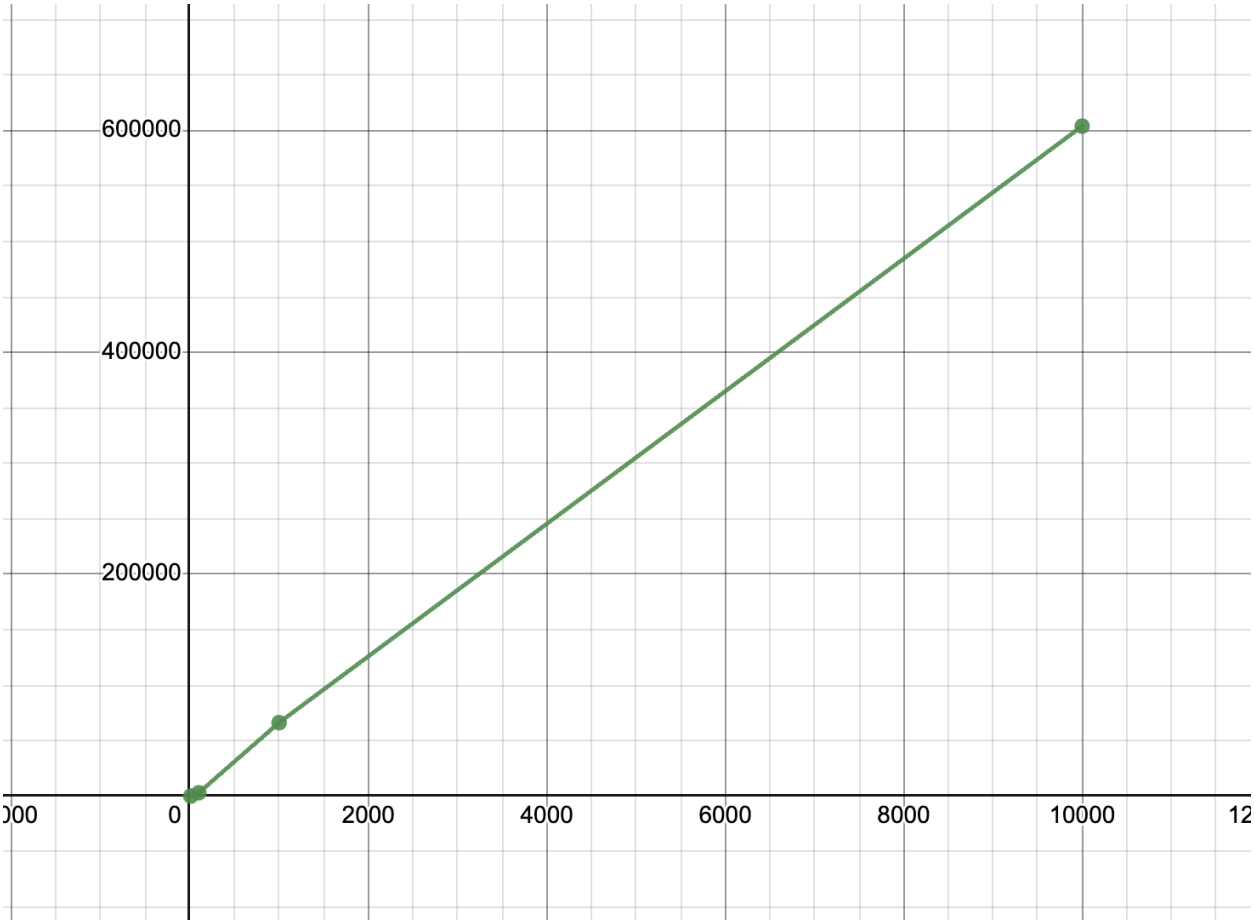

Figure 3: Shell Sort

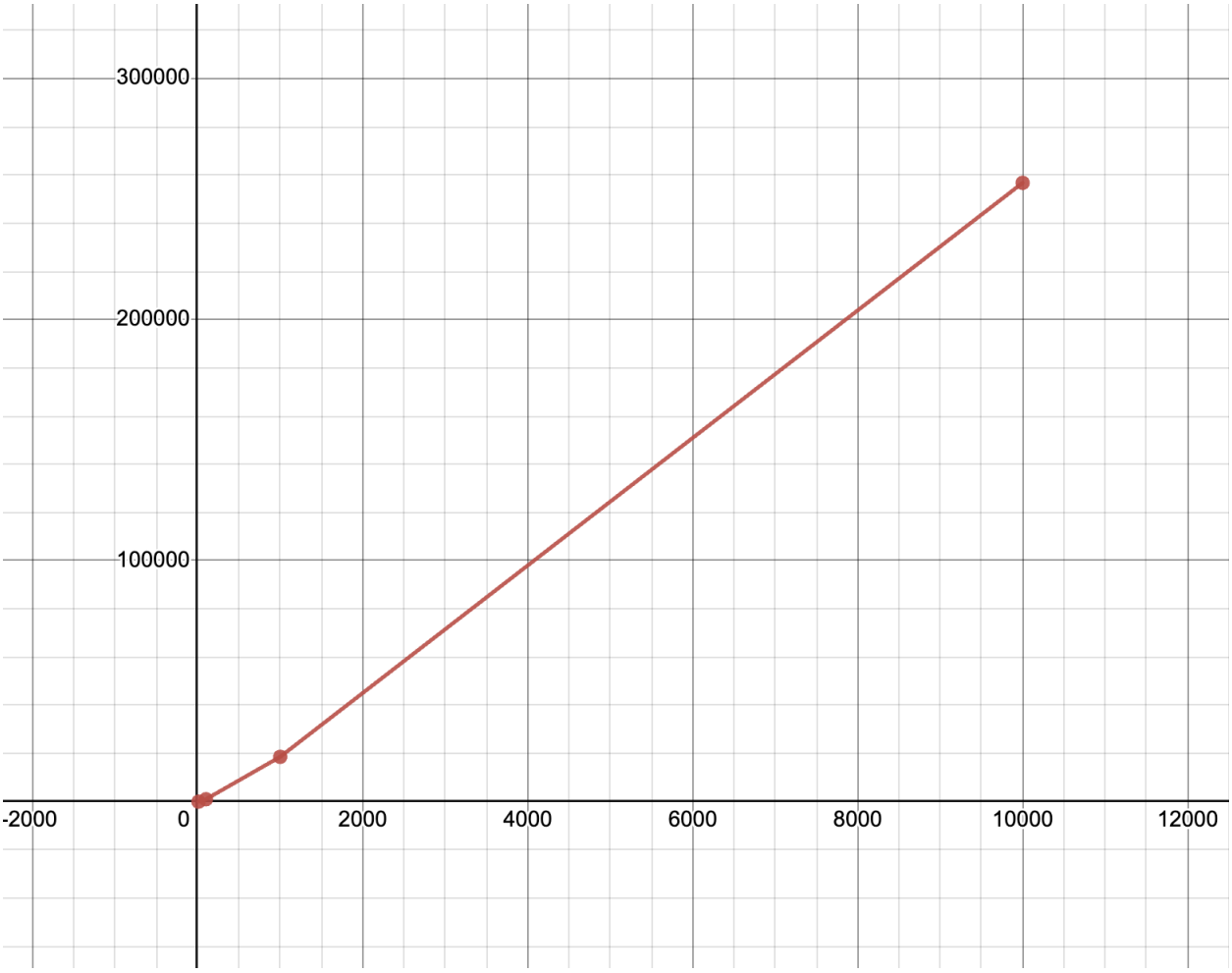

Figure 4: Quick Sort

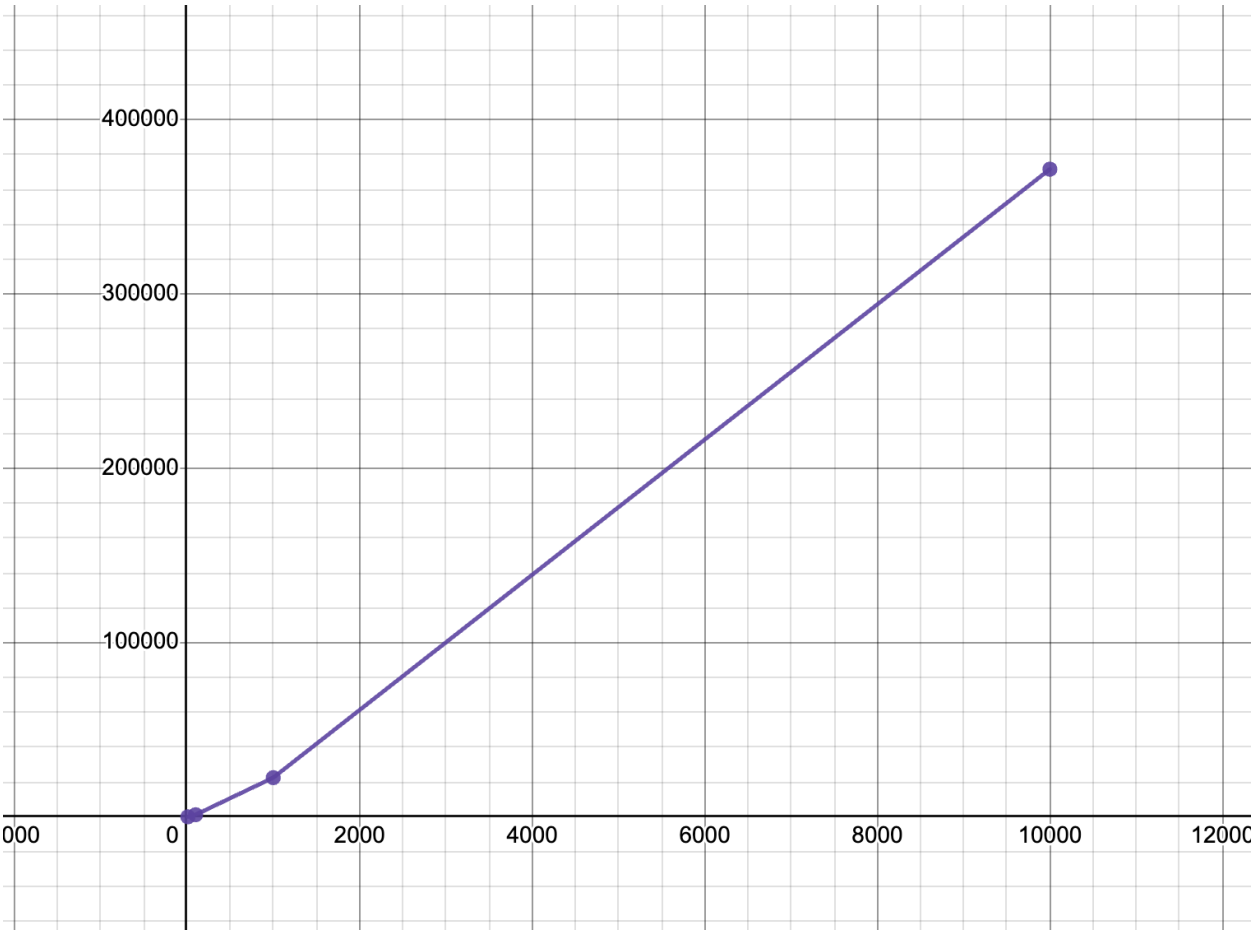

Figure 5: Batcher Sort

```
mgarc318@mgarc318:~/cse13s/asgn3$ ./sorting -h -s -q -n 20 -p 30
Heap Sort, 20 elements, 228 moves, 120 compares
     34732749     42067670     54998264    102476060    104268822
    134750049    182960600    194989550    451764437    538219612
    607875172    629948093    783585680    954916333    966879077
    989854347    994582085   1025188081   1037686539   1072766566
Shell Sort, 20 elements, 273 moves, 141 compares
    200592044    251593342    261742721    391223417    426152680
    444703321    460885430    500293632    510040157    521864874
    579453371    616902904    620182312    738166936    782250002
    868766010    908068554    935579555    950136224   1054405046
Quick Sort, 20 elements, 216 moves, 118 compares
      8032304     56499902     73647806     75442881    111498166
    243082246    398173317    438071796    447975914    464871224
    527207318    648567958    689665138    708948898    783550802
    920038191    934604298    988526615    999105042   1037080358
mgarc318@mgarc318:~/cse13s/asgn3$
```

Figure 6: Screenshot of the program running with Heap, Shell, Quick, and number of elements 20 and num elements printed 30 but reduced to number of elements.

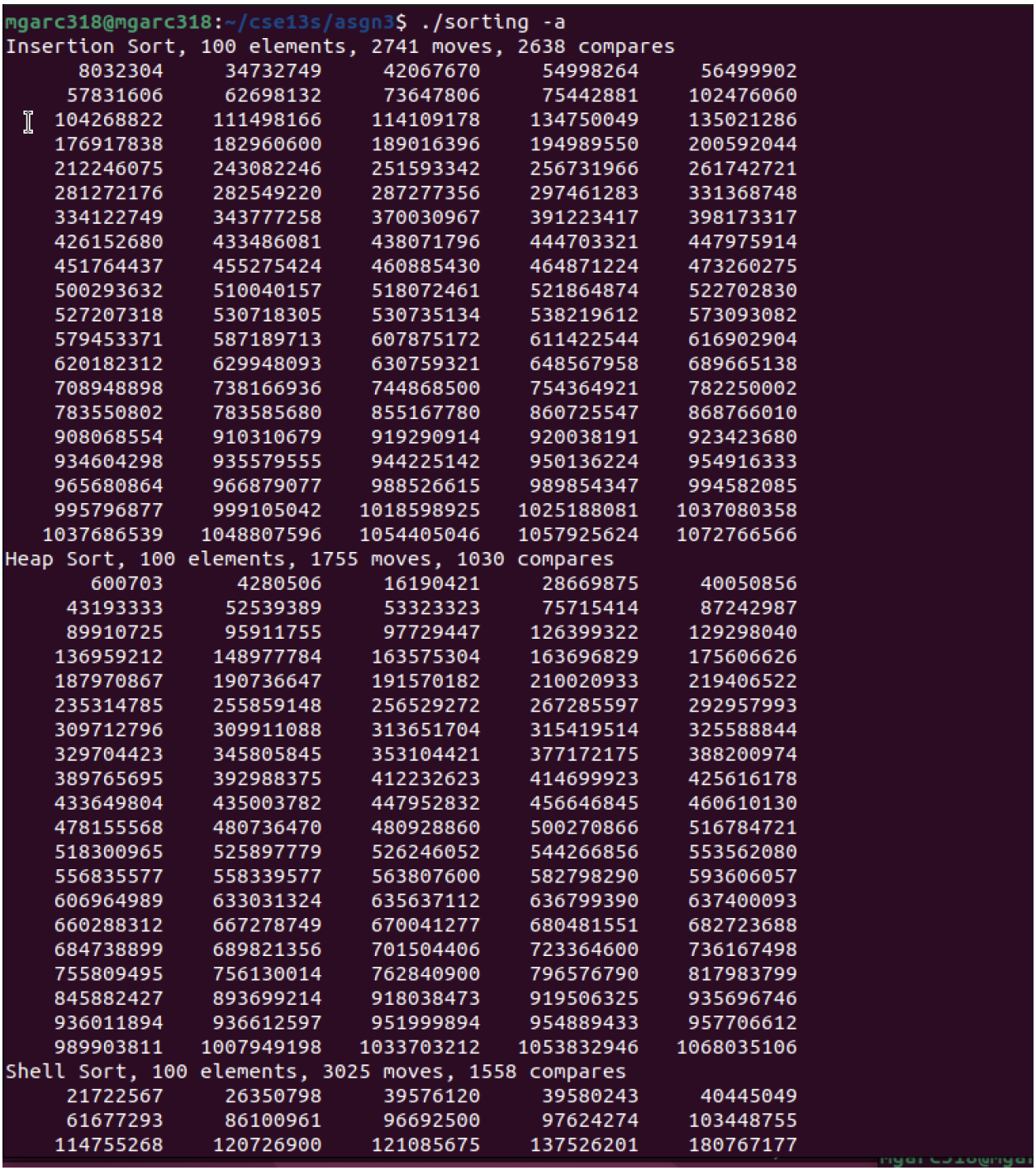

Figure 7: Screenshot of the program running with all sorts.

```
ngarc318@ngarc318:~/cse13s/asgn3$ ./sorting -H
By running './sorting <flag1> <flag2> ..' you can input the following flags to tell the program which sorting algorithmy you want to run and change the size,
seed, and if you want to print the number of elements or not:
-a : Employs all sorting algorithms
-i : Enables Insertion Sort
-s : Enables Shell Sort
-h : Enables Heap Sort
-q : Enables Quick Sort
-b : Enables Batcher Sort
-r <num> : Sets the seed to num. The default seed is 13371453
-n <num> : Sets the size of array to num. The default size is 100
-p <elements> : Enables printing of statistics to see computed terms and factors for all tested functions, elements is the number of computed terms
-h : Displays this help message again
ngarc318@ngarc318:~/cse13s/asgn3$
```

Figure 8: Screenshot of the program running with help message.

```
ngarc318@ngarc318:~/cse13s/asgn3$ valgrind ./sorting
==9062== Memcheck, a memory error detector
==9062== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==9062== Using Valgrind-3.18.1 and LibVEX; rerun with -h for copyright info
==9062== Command: ./sorting
==9062==
By running './sorting <flag1> <flag2> ..' you can input the following flags to tell the program which sorting algorithmy you want to run and change the size,
seed, and if you want to print the number of elements or not:
-a : Employs all sorting algorithms
-i : Enables Insertion Sort
-s : Enables Shell Sort
-h : Enables Heap Sort
-q : Enables Quick Sort
-b : Enables Batcher Sort
-r <num> : Sets the seed to num. The default seed is 13371453
-n <num> : Sets the size of array to num. The default size is 100
-p <elements> : Enables printing of statistics to see computed terms and factors for all tested functions, elements is the number of computed terms
-h : Displays this help message again
==9062==
==9062== HEAP SUMMARY:
==9062==     in use at exit: 0 bytes in 0 blocks
==9062==   total heap usage: 1 allocs, 1 frees, 1,024 bytes allocated
==9062==
==9062== All heap blocks were freed -- no leaks are possible
==9062==
==9062== For lists of detected and suppressed errors, rerun with: -s
==9062== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
ngarc318@ngarc318:~/cse13s/asgn3$
```

Figure 9: Screenshot of valgrind producing no errors (no memory leaks).