Assignment 4 – Sets and Sorting

Collin Wen

CSE 13S – Fall 2023

Purpose

The purpose for this assignment is to create a program that implements all the sorting algorithms(insertion sort, shell sort, heap sort, quick sort, and Batcher's odd-even merge sort). The program uses an array with random elements and prints the elements of the array sorted by whichever sorting algorithms are selected. The program also prints out which sort was used, the elements sorted, comparisons used, and moves used. For knowing which algorithms have been selected a Set is used which is represented by an unsigned 8 bit integer. The bits of the set and if the bit has a member determines which sorting algorithms are used.

How to Use the Program

To make the program using the Makefile you just enter "make" and the program executable sorting should be compiled. To use the program the user must enter in the command line "./sorting" followed by the following:
-H prints out program usage
-i selects insertion sort to be applied
-s selects shell sort to be applied.
-h selects heap sort to be applied
-q selects quick sort to be applied
-a selects all sorting algorithms to be applied
-r followed by a valid number above 0 for the random seed used in the program
-n followed by a number for the size of the array
-p followed by a number for the number of elements to be printed out in the array

After typing that in the command line the program will print out the sorted array's elements with the type of sort used, number of elements in the array, number of moves used by the sorting algorithm and the number of comparisons used by the sorting algorithm. Multiple sorting algorithms can be selected.

Program Design

The program is designed with header files containing helper functions and a main program "sorting.c" for printing out the array and sorts used and their stats.

set.h and set.c:

set.h contains the struct set which is just an unsigned 8 bit integer. Its functions are used to modify the Set by setting all members, inserting a member, removing a member, clearing the Set, checking for a member in the set, and checking for same/different elements in the set.

The main file uses a Set and getopt to tell which sorting algorithm is selected by putting a member in a certain bit place.

insert.h and insert.c:

These are the files for insertion sort that sorts an array by comparing one item at a time in a list with all of its previous items replacing that item if it is smaller.

shell.h and shell.c:

Files for the shell sort that sorts an array by sorting elements far apart from each other using and gradually decreasing the gap of the elements that are being compared.

heap.h and heap.c:

Files for heap sort that sorts an array by organizing elements in heaps and moving the largest element in the heap to the end of the array.

quick.h and quick.c:

Files for quick sort that sorts an array by breaking up an array into smaller ones and then sorting the left and right parts of that array using recursion.

batcher.h and batcher.c:

Files for batcher sort that uses sorting networks and comparators to sort and array.

stats.h and stats.c:

Files for the stats structure that is used to record stats like comparisons and moves used by the sorting algorithms.

sorting.c:

sorting.c is the main program file that creates a random array and depending on which sorts are chosen they print that sorted array with the number of moves and comparisons used.

Data Structures

The struct set is a unsigned 8 bit integer that has members representing if something is selected or not. To interact with the set for doing things like inserting a member bitwise operators are used.

Another struct “stats” is used to track the number of moves and comparisons used by a sorting algorithm.

An array is used in sorting.c that is randomized to a 30 bit integer using the random() function.

Algorithms

Insertion Sort:

```
def insertion_sort(A: list):
# Sort values in A[0] through A[len(A) - 1]
    for k in range(1, len(A)):
# k will take on values from 1 through len(A) - 1.
# A[0] through A[k - 1] already are sorted.
# Shift values until A[0] through A[k] are sorted.
# Example with k == 3, so A[0] through A[2] already are sorted:
# A[0] A[1] A[2] A[3] temp
# 20 40 50 30 --
# 20 40 50 -- 30 temp = A[3]
# 20 40 -- 50 30 A[3] = A[2]
# 20 -- 40 50 30 A[2] = A[1]
# 20 30 40 50 -- A[1] = temp
j = k
temp = A[k] # move A[k]
while j >= 1 and temp < A[j - 1]: # compare temp and A[j - 1]
A[j] = A[j - 1] # move A[j - 1]
j -= 1
A[j] = temp # move temp
```

Shell sort:

```
def shell_sort(A: list):
# Sort values in A[0] through A[len(A) - 1]
for gap in gaps:
# gap will take on a series of non-contiguous, decreasing values ending with 1.
# (We provide you with an array of these "gap" values.)
for k in range(gap, len(A)):
# k will take on values from gap through len(A) - 1.
# Sort several interleaved lists, one list for each value of gap.
# For example, when gap == 5:
# The list ending ..., A[k-15], A[k-10], A[k-5] already is sorted.
# Shift values until ..., A[k-15], A[k-10], A[k-5], A[k] are sorted.
# (Compare these statememnts to those of insertion_sort().)
j = k
temp = A[k] # move A[k]
while j >= gap and temp < A[j - gap]: # compare temp and A[j - gap]
A[j] = A[j - gap] # move A[j - gap]
j -= gap
A[j] = temp # move temp
```

Heap sort

```
def max_child(A: list, first: int, last: int):
left = 2 * first + 1
right = 2 * first + 2
if right <= last and A[right] > A[left]:
return right
return left
def fix_heap(A: list, first: int, last: int):
done = False
parent = first
# Move the parent down until the max-heap condition is met.
while 2 * parent + 1 <= last and not done:
# parent has at least one child
largest_child = max_child(A, parent, last)
if A[parent] < A[largest_child]:
# This Python code swaps parent and largest_child.
A[parent], A[largest_child] = A[largest_child], A[parent]
parent = largest_child
else:
done = True

def build_heap(A: list, first: int, last: int):
if last > 0:
for parent in range((last - 1) // 2, first - 1, -1):
fix_heap(A, parent, last)
def heap_sort(A: list):
first = 0
last = len(A) - 1
build_heap(A, first, last)
for leaf in range(last, first, -1):
# Visit all non-root nodes starting with the last.
# That is, the range() above ensures that leaf never will equal first.
# This Python code swaps first and leaf.
A[first], A[leaf] = A[leaf], A[first]
fix_heap(A, first, leaf - 1)
```

Quick sort:

```
def partition(A: list, lo: int, hi: int):
# Use element hi as the pivot.
# Divide the subarray into two partitions.
i = lo - 1
for j in range(lo, hi):
# j takes values from lo to hi - 1
if A[j] < A[hi]:
i += 1
```

```
# Swap elements i and j.
A[i], A[j] = A[j], A[i]
i += 1
# Swap the pivot and element i.
A[i], A[hi] = A[hi], A[i]
# Return where the pivot is now.
return i

# 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, 0, len(A) - 1)
```

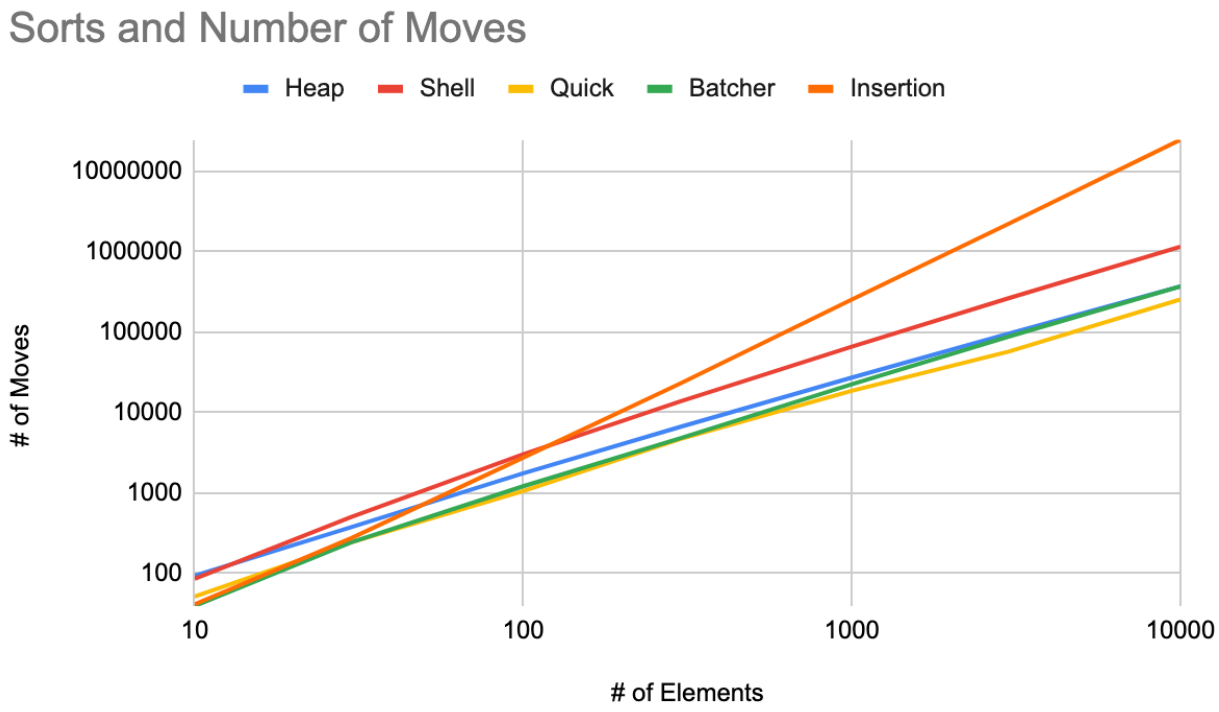
Batcher Sort:

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

Taken from assignment 4 pdf

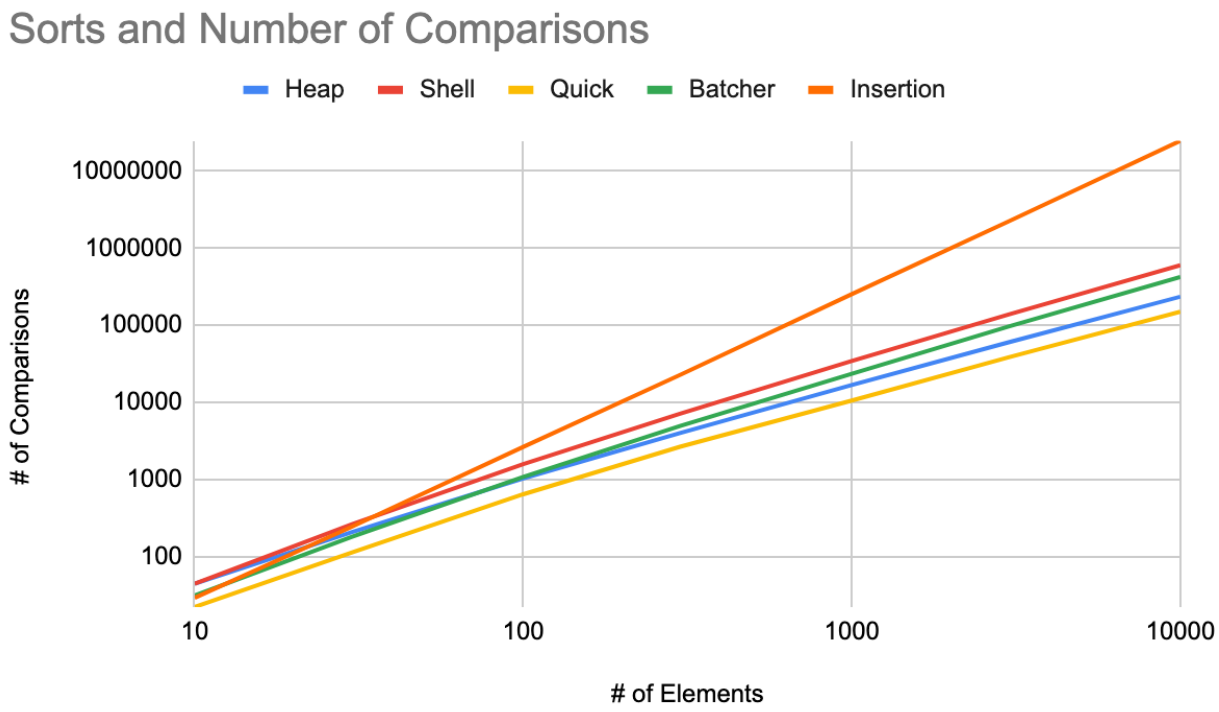
Results

In general, sorts perform more better when given smaller arrays that already have some elements sorted. Sorts perform worse when they are tasked with larger more random arrays especially arrays that are in reverse order which requires more comparisons and moves.



Graph of sorts and the number of moves required.

In the above graphs plotting the number of moves required by each type of sort with increasing number of elements you can see that the quick sort and batcher sort require the least amount of moves when the number of elements is larger. The opposite is true for shell and insertion sort. Insertion sort you can see specifically struggles pretty bad when the number of elements are larger. The number of moves required with the number of elements after 1000 is quite far off from teh other sorts.



Graph of sorts and the number of comparisons required.

The above graph shows the number of comparisons used by each sort as the number of elements in the array increases. Insertion sort like with the number of moves needed requires much more comparisons than other arrays when there are a larger amount of elements. You can especially see this after 1000 elements. Quicksort is the most efficient with comparisons as the graph shows it use the least amount for any number of elements.