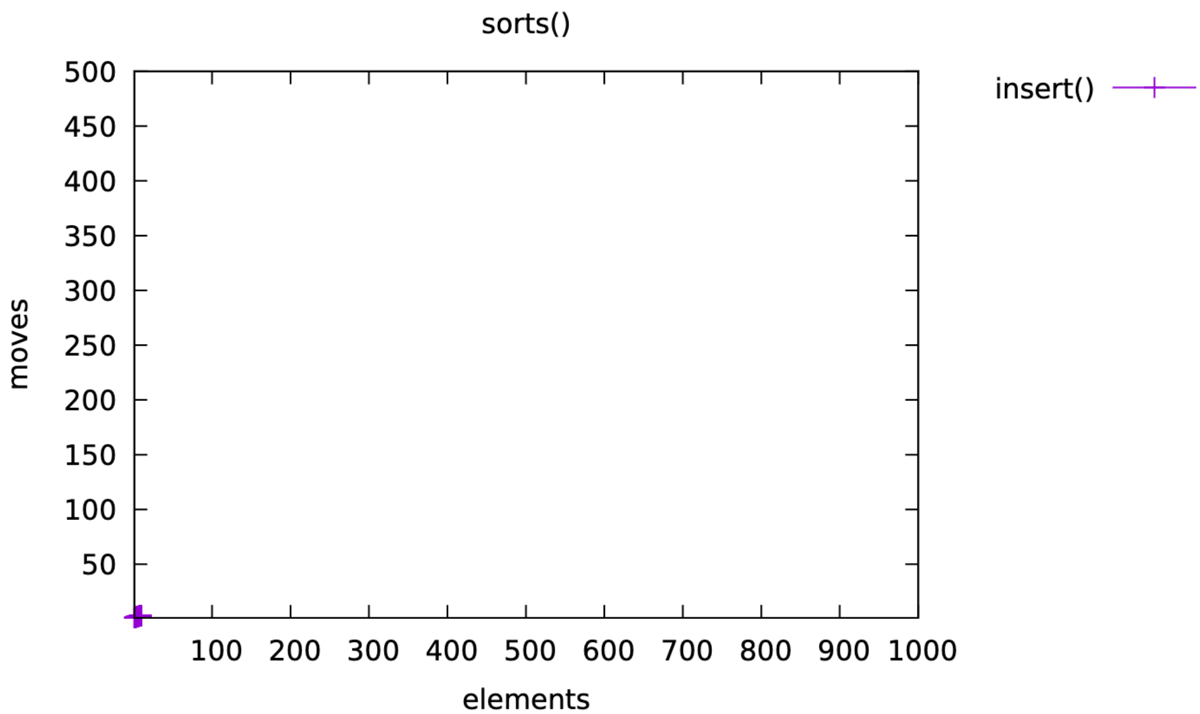
Assignment 3 - Sorting: Putting Your Affairs in Order

I learned that though there are many unique ways to sort an array, plenty of sorting algorithms involve similar techniques. For instance, insertion sort, which I found the most intuitive and straightforward, played a big role in shell sort, which I found really confusing and complex at first. Insertion sort is all about comparing the current element to its previous term. If the previous element is greater than the current element, then the two would move places, and this process would continue until the entire array is sorted in increasing order. Knowing that I had to implement a gap function and find a way to reduce this gap with every call in shell sort originally threw me off, but further contemplation helped me see how similar to insertion it is. In fact, everything following the for loops that keep track of the gap was the same exact process of comparing and moving elements. The gap function was essentially a way to go through the insertion process at a faster, more efficient rate. There are similarities between heap sort and quick sort as well. Both processes consider the array in smaller terms, whether it be by trees and nodes or smaller arrays. Both methods simplify the way we look at arrays (especially big ones) and help sort them efficiently.

Graphs

Unfortunately, I have no knowledge of gnuplot and the research I did on this was futile as I could not figure out how to plot multiple lines on one graph or how to graph over an unidentified number of elements. The graph I managed to create (for insert.c) is shown below but it does not provide any useful data. However, I still believe that the efficiency of my sorting algorithms are similar to the graph provided in the assignment document on page twelve. I know for a fact that my sort algorithms work correctly, so it must be true that insertion sort and shell sort have time complexities of n squared and that heap sort and quick sort have time complexities of n log n. Given how close together the heap and quick sorts are on the given graph, it makes sense that they have similar efficiencies. Quick sort is also known to be one of the most efficient sorts, hence why it has a smaller slope than the others. On the other hand, insertion sort is known to be one of the least efficient sorts, which is why it is steeper than the others. Quick sort must have an uneven line on the graph since it involves partitioning the array, which provides somewhat inconsistent results when sorting arrays of different numbers of elements and thus different sizes.



Output (for -a)

```
tkjoseph@leaf:~/cse13s/asgn3$ ./sorting -a
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
Shell Sort, 100 elements, 1183 moves, 801 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
Insertion Sort, 100 elements, 2741 moves, 2638 compares
```

```
   1037686539   1048807596   1054405046   1057925624   1072766566
Insertion Sort, 100 elements, 2741 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
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
tkjoseph@leaf:~/cse13s/asgn3$
```

insert.c

```c
#include "stats.h"

#include <stdio.h>

// Insertion Algorithm Implementation
// Compares with previous element(s) in array
// Follows assignment document pseudocode

void insertion_sort(Stats *stats, uint32_t *A, uint32_t n) {
    for (int i = 1; i < n; i++) {
        int j = i;
        int temp = move(stats, A[i]);
        while (j > 0 && cmp(stats, temp, A[j - 1]) < 0) {
            A[j] = move(stats, A[j - 1]);
            j -= 1;
        }
        A[j] = move(stats, temp);
    }
}
```

heap.c

```c
#include "stats.h"

#include <stdbool.h>
#include <stdio.h>

// Heap Algorithm Implementation
// Compares parents and children in max heap form
// Finds the max child, fixes the heap, and then builds it
// Follows assignment document pseudocode

uint32_t max_child(Stats *stats, uint32_t *A, uint32_t first, uint32_t last) {
    uint32_t left = 2 * first;
    uint32_t right = left + 1;
    if (right <= last && cmp(stats, A[right - 1], A[left - 1]) > 0) {
        return right;
    }
    return left;
}

void fix_heap(Stats *stats, uint32_t *A, uint32_t first, uint32_t last) {
    bool found = false;
    uint32_t mother = first;
    uint32_t great = max_child(stats, A, mother, last);
    while (mother <= (last / 2) && !found) {
        if (cmp(stats, A[mother - 1], A[great - 1]) < 0) {
            swap(stats, &A[mother - 1], &A[great - 1]);
            mother = great;
            great = max_child(stats, A, mother, last);
        } else {
            found = true;
        }
    }
}

void build_heap(Stats *stats, uint32_t *A, uint32_t first, uint32_t last) {
    for (uint32_t father = (last / 2); father > first - 1; --father) {
        fix_heap(stats, A, father, last);
    }
}

void heap_sort(Stats *stats, uint32_t *A, uint32_t n) {
    uint32_t first = 1;
    uint32_t last = n;
    build_heap(stats, A, first, last);
    for (uint32_t leaf = last; leaf > first; --leaf) {
        swap(stats, &A[first - 1], &A[leaf - 1]);
        fix_heap(stats, A, first, leaf - 1);
    }
}
```

shell.c

```c
#include "stats.h"

#include <math.h>
#include <stdio.h>

// Shell Algorithm Implementation
// Creates gap that is then reduced until gap is one
// Very similar to Insertion Algorithm Implementation
// Follows assignment document pseudocode

// Creates static variables for gap implementation
static uint32_t i = 0;

static uint32_t gap_array[32];

uint32_t array_size;

void gaps(uint32_t n) {
    uint32_t range = (uint32_t)(log(3 + 2 * n) / log(3));
    uint32_t index = 0;
    for (uint32_t i = range; i > 0; --i) {
        gap_array[index++] = (uint32_t)((pow(3, i) - 1) / 2);
    }
    array_size = range;
}

void shell_sort(Stats *stats, uint32_t *A, uint32_t n) {
    gaps(n);
    for (uint32_t ii = 0; ii < array_size; ii++) {
        uint32_t gap = gap_array[ii];
        for (uint32_t i = gap; i < n; i++) {
            uint32_t j = i;
            uint32_t temp = move(stats, A[i]);
            while (j >= gap && cmp(stats, temp, A[j - gap]) < 0) {
                A[j] = move(stats, A[j - gap]);
                j -= gap;
            }
            A[j] = move(stats, temp);
        }
    }
}
```

quick.c

```c
#include "stats.h"

#include <stdio.h>

// Quick Algorithm Implementation
// Partitions array into two separate arrays and compares elements to pivot
// Follows assignment document pseudocode

uint32_t partition(Stats *stats, uint32_t *A, uint32_t low, uint32_t high) {
    uint32_t i = low - 1;
    for (uint32_t j = low; j < high; j++) {
        if (cmp(stats, A[j - 1], A[high - 1]) < 0) {
            i++;
            swap(stats, &A[i - 1], &A[j - 1]);
        }
    }
    swap(stats, &A[i], &A[high - 1]);
    return i + 1;
}

void quick_sorter(Stats *stats, uint32_t *A, uint32_t low, uint32_t high) {
    if (low < high) {
        uint32_t p = partition(stats, A, low, high);
        quick_sorter(stats, A, low, p - 1);
        quick_sorter(stats, A, p + 1, high);
    }
}

void quick_sort(Stats *stats, uint32_t *A, uint32_t n) {
    quick_sorter(stats, A, 1, n);
}
```

sorting.c

```c
#include "heap.h"
#include "insert.h"
#include "quick.h"
#include "set.h"
#include "shell.h"

#include <inttypes.h>
#include <math.h>
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

#define OPTIONS       "aeisqr:n:p:h"
#define HEAP_BIT      0
#define INSERTION_BIT 1
#define SHELL_BIT     2
#define QUICK_BIT     3
#define HEX_30_BITS   0x3FFFFFFF

// Creates static variables of default values (for -r, -n, and -p)
static uint32_t seed = 13371453;
static uint32_t size = 100;
static uint32_t print_size = 100;

// Creates array of random elements
// ptr = pointer (here and everywhere else)
uint32_t *create_array(uint32_t size, uint32_t seed) {
    uint32_t *ptr;
    ptr = malloc(size * sizeof(uint32_t));
    srandom(seed);
    for (uint32_t i = 0; i < size; i++) {
        *(ptr + i) = random() & HEX_30_BITS;
    }
    return ptr;
}

// Creates random elemets (for -a, -e, -i, -s, and -q)
void init_random(uint32_t *ptr, uint32_t size, uint32_t seed) {
    srandom(seed);
    for (uint32_t i = 0; i < size; i++) {
        *(ptr + i) = random() & HEX_30_BITS;
    }
}
```

```c
// Prints statistics (and elements if asked) for each sort
void print_result(Stats *stat_ptr, char *label, uint32_t *data_ptr) {
    printf("%s, %d"
           " elements,",
        label, size);
    printf(" %lu"
           " moves,",
        stat_ptr->moves);
    printf(" %lu"
           " compares\n",
        stat_ptr->compares);
    if (print_size > size) {
        print_size = size;
    }
    for (uint32_t i = 0; i < print_size; i++) {
        if (i > 0 && i % 5 == 0) {
            printf("\n");
        }
        printf("%13" PRIu32, *(data_ptr + i));
    }
    printf("\n");
}

// Main function
// Makes use of set.h's inserts and members
// Makes use of create_array, init_random, and print_results
int main(int argc, char **argv) {
    Stats insert_stats;
    Stats shell_stats;
    Stats heap_stats;
    Stats quick_stats;
    int opt = 0;
    Set s = empty_set();
    while ((opt = getopt(argc, argv, OPTIONS)) != -1) {
        switch (opt) {
        case 'a':
            // all sorts
            s = insert_set(HEAP_BIT, s);
            s = insert_set(INSERTION_BIT, s);
            s = insert_set(SHELL_BIT, s);
            s = insert_set(QUICK_BIT, s);
            break;
        case 'e':
            // heap
            s = insert_set(HEAP_BIT, s);
            break;
```

```
        case 'i':
            // insertion
            s = insert_set(INSERTION_BIT, s);
            break;
        case 's':
            // shell
            s = insert_set(SHELL_BIT, s);
            break;
        case 'q':
            // quick
            s = insert_set(QUICK_BIT, s);
            break;
        case 'r':
            // random seed
            seed = atoi(optarg);
            break;
        case 'n':
            // array size
            size = atoi(optarg);
            break;
        case 'p':
            // # of elements
            print_size = atoi(optarg);
            break;
        case 'h':
            // program usage
            printf("SYNOPSIS\n   A collection of comparison-based sorting algorithms.\n\n");
            printf("USAGE\n   ./sorting [-haeisqn:p:r:] [-n length] [-p elements] [-r seed]\n\n");
            printf("OPTIONS\n");
            printf("   -h              display program help and usage\n");
            printf("   -a              enable all sorts.\n");
            printf("   -e              enable Heap Sort.\n");
            printf("   -i              enable Insertion Sort.\n");
            printf("   -s              enable Shell Sort.\n");
            printf("   -q              enable Quick Sort.\n");
            printf("   -n length       specify number of array elements (default: 100).\n");
            printf("   -p elements     specify number of elements to print (default: 100).\n");
            printf("   -r seed         specify random seed (default: 13371453).\n");
            break;
        }
    }

    uint32_t *test_pattern;
```

```c
#if 1
    test_pattern = malloc(size * sizeof(uint32_t));
    if (test_pattern == NULL) {
        return -1;
    }
#else
    test_pattern = create_array(size, seed);
    if (test_pattern == NULL) {
        return -1;
    }
#endif

    if (member_set(HEAP_BIT, s)) {
        init_random(test_pattern, size, seed);
        reset(&heap_stats);
        heap_sort(&heap_stats, test_pattern, size);
        print_result(&heap_stats, "Heap Sort", test_pattern);
    }

    if (member_set(SHELL_BIT, s)) {
        init_random(test_pattern, size, seed);
        reset(&shell_stats);
        shell_sort(&shell_stats, test_pattern, size);
        print_result(&shell_stats, "Shell Sort", test_pattern);
    }

    if (member_set(INSERTION_BIT, s)) {
        init_random(test_pattern, size, seed);
        reset(&insert_stats);
        insertion_sort(&insert_stats, test_pattern, size);
        print_result(&insert_stats, "Insertion Sort", test_pattern);
    }

    if (member_set(QUICK_BIT, s)) {
        init_random(test_pattern, size, seed);
        reset(&quick_stats);
        quick_sort(&quick_stats, test_pattern, size);
        print_result(&quick_stats, "Quick Sort", test_pattern);
    }

    free(test_pattern);
    return 0;
}
```