Edison Chen

Assignment 3: Sorting

## Program Description:

The purpose of this assignment is to implement various sorts in order to gain a better understanding of various algorithms and computational complexity. The program implements four sorting algorithms: insertion sort, shell sort, heapsort, and recursive quicksort. A test harness will also be implemented for the sorting algorithms and will be creating an array of pseudorandom elements used for testing each of the sorts. A set will be used to track command-line options of the implemented sorting algorithms. Statistics for each sort must also be implemented in the test harness, in particular: the size of the array, the number of moves required, and the number of comparisons required.

## Files Included:

- insert.c - c file that implements Insertion Sort
- insert.h - interface specified for insert.c
- shell.c - c file that implements Shell Sort
- shell.h - interface specified for shell.c
- heap.c - c file that implements Heap Sort
- heap.h - interface specified for heap.c
- quick.c - c file that implements Quicksort
- quick.h - interface specified for quick.c
- stats.c - c file that implements the statistics module
- stats.h - interface specified for the statistics module
- set.h - implements and specifies the interface for the set

- sorting.c - contains main() and implements the testing harness used to test and compare sorts along with their statistics
- Makefile - file that builds the program, runs the program, and formats all files to clang format
- README.md - text file in markdown format that describes: the program, how to build the program, and how to run the program
- DESIGN.pdf - pdf file that describes the design and program
- WRITEUP.pdf - pdf file that describes and compares the performances of each sorting algorithm with graphs

## Notes and Pseudocode:

### Insertion Sort:

- Considers elements one at a time
- For an array of size n, for each k in increasing value form 1 <= k <= n, compares the kth element with each of the preceding ones
- Starts from A[k] and checks if it is in correct order by comparing it to A[k-1]
- Two possibilities during comparison
    - A is in the right place: move to sort the next comparison
    - A is in the wrong place: A[k] is in the wrong place (less than A[k-1]) and A[k-1] is shifted up to A[k] and original value of A[k] is compared to A[k-2] and so on

**Pseudocode:**

for each iteration from 1 to the length of the array - 1

    set variable j to the current iteration count

    set variable temp to the list element whose index is the current iteration count

// while loop checks if the current value is greater than all the preceding ones

while j is greater than 0 and temp is less than the list element of index j - 1

set the list element of index j to the list element of index j -1

subtract 1 from j

// this compares the next preceding element

set the value of the list of index j to the value of variable temp

**Shell Sort:**

- Variation of insertion sort, sorts pairs of elements far apart from each other in what is called a gap
- Each interaction decreases the gap until a gap of 1 is used, which means the elements are sorted
- Need to find a way around not having a generator like in Python
- Gaps are computed as [(3^k) -1 / 2], largest k is [log (2n + 3) / log (3)], n is the length of the array to sort
- Start with the largest (k) and go down to a gap size of 1

  **Pseudocode:**

  function gaps:

  for each iteration from largest k to 0, incrementing by - 1 each iteration

  generate ((3^k)-1 / 2)

  function shell_sort:

  for each value gap returned by function gaps

  for each iteration, i from gap to length of the array - 1

  set variable j to i

set temp to value of index i of array

while j >= gap and temp is less than value of array at index j - gap

value of array at index j = value of array at index j - gap

j is equal to j minus the value of gap

value of the array at index j  = value of temp

**Heap Sort:**

- Max heap: parent node must have a value that is greater than or equal to the values of its children
- Min heap: parent node must have a value that is less than or equal to the values of its children
- Typically represented as an array for any index k, the index of the left child is 2k and the index of the right is 2k + 1
- Heapsort sorts its elements using 2 ways
    - Building a heap: taking the array to sort and building a heap from it. The constructed heap will be a max heap. The largest element, (root) is the first element of the array from which the heap is built.
    - Fixing a heap: largest array elements removed from the top of the heap and placed at the end of the sorted array. After removing the largest element from the heap, the heap needs to be fixed so that it once again obeys the constraints of a heap.

**Pseudocode:**

function build_heap: takes in A(list), first(int), last(int)

for variable father in (last divided by 2 to first - 1) increment by -1 each iteration

fix the heap using arguments (A, father, and last)

function heap_sort: takes in A(list)

set int first to 1

set int last to the length of A

build the heap using arguments (A, first, last)

for variable leaf from (last to first) incrementing by -1 each iteration

swap value A with index (first - 1) with value of A with index (leaf - 1)

fix the heap using (A, first, leaf - 1) as arguments

## Quicksort (recursive):

- Splits arrays into two subarrays by selecting an element from the array and designating it as a pivot
- Elements less than pivot go to left side and elements greater than or equal to the pivot go to the right side
- Returns the index that indicates the division between partitioned parts of the array

### Pseudocode:

function partition: takes in arguments A(list), lo(int), hi(int)

int i equals lo - 1

for int j from lo to hi

if value of A of index (j - 1) is less than value of A of index (hi - 1)

increment i by 1

swap value of A of index (i - 1) with value of A of index (j - 1)

swap value of A of index (i) with value of A of index (hi - 1)

return value of (i + 1)

function quick_sorter: takes in arguments A(list), lo(int), hi(int)

if lo is less than hi

define p as the result returned by calling function partition

recursively call quick_sorter using arguments (A, lo, p - 1)

recursively call quick_sorter using arguments(A, p + 1, hi)

function quick_sort: takes in arguments A(list)

call function quick_sorter using arguments (A, 1, length of A)

**Sorting:**

define OPTIONS aeisqhr:n:p:

while command line arguments in OPTIONS

implement switch cases for each of the options

implement a set and include sorts if their command line arguments are inputted

use optarg to take in inputs in the case of commands -r -n -p

set default cases for size(n), random seed(r), and number of elements to print(p)

print out the sorted array and statistics of each sort after they are called

reset the structure stats and refill the array after it is sorted with the same pseudorandom numbers generated by srandom() each time a function is called

free dynamically allocated memory once main() is finished running

## Credit:

- Code for the sorts are translated to C code from the python code given in assignment 3 pdf by Professor Long

- I watched Eugene's section video on 10/12/21 for learning how to use sets and for general guidance on assignment 3
- I modified the plot.sh file shown in the lecture on 10/11/21 by Professor Long to produce the graphs in my writeup