asgn3: Sorting:Putting your affairs in order

Program is divided up into 6 .c files, 4 of which contain code for sorting algorithms, 1 contains functions necessary for keeping track of several efficiency related statistics, and lastly a main file to properly run all of the algorithms and output relevant statistics.

sorting.c:
Acts as the main file that runs all of the other files to properly test and compare their efficiencies on different sized arrays.
Start:
 Use getopt to check which characters the user inputted when running the program
 For each character that the user specified run the corresponding test
  -a: Employs all sorting algorithms.
  -e: Enables Heap Sort.
  -i: Enables Insertion Sort.
  -s: Enables Shell Sort.
  -q: Enables Quicksort.
  -r seed: Set the random seed toseed. Thedefaultseed should be 13371453.
  -n size: Set the array size tosize. Thedefaultsize should be 100.
  -p elements: Print out number of elements from the array. The default number of elements to print out should be 1
00.
  -h: Prints out program usage. See reference program for example of what to print.

insert.c:
Insertion sort that sorts an array one element at a time, by placing them in their correct position.
Insertion sort compares the k-th element with each of the preceding elements in descending order until its position is found.

```
//COPY OF LONG'S PYTHON PSEUDOCODE IN asgn3.pdf
def insertion_sort(A:list):
    for i in range(1,len(A)):
        j=i;
        temp=A[i]
        while j>0 and temp<A[j-i]:
            A[j]=A[j-1]
            j-=1
        A[j]=temp
```

heap.c:
Heap sort is a binary tree where elements are stored in a special order so that the value of a parent node is greater(for
 max heap)
for smaller(for min-heap) than the values in its two children nodes.

```
//COPY OF LONG'S PYTHON PSEUDOCODE IN asgn3.pdf
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 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,first,leaf-1)
```

quick.c:
Quick sort as one of the most commonly used sorting algorithms sorts by dividing and conquering.
Quick sort will partition an array into two smaller arrays, by choosing a pivot element and separating the rest of the elements into the two arrays based on if they are greater than or less than the pivot element.

```
//COPY OF LONG'S PYTHON PSEUDOCODE IN asgn3.pdf
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

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))
```

shell.c:
Shell sort is a variation of shell sort that initially sorts pairs of elements which are as far apart as possible.
With each iteration of Shell sort, the gap becomes smaller and smaller until eventually reaching a gap of 1.
A gap sequence of 1 always produces a sorted array.

```
//COPY OF LONG'S PYTHON PSEUDOCODE IN asgn3.pdf
def gaps(n: int):
    for i in range(int(log(3+2*n)/log(3)),0,-1):
        yield(3**i-1)//2

def shell_sort(A: list):
    for gap in gaps(len(A)):
        for i in range(gap,len(A)):
            j=i
            temp=A[i]
```

```
while j>=gap and temp<A[j-gap]:
    A[j]= A[j-gap]
    j-=gap
A[j]=temp
```

stats.c:
Provides access to functions cmp, move, swap, and reset.
This file is what allows easy tracking of the efficiency of the different sorting algorithms.
The functions this file provides are used in all of the sorting algorithms.
In each of the functions, variables are used which are all a part of an object each of which track a specific form of efficiency: number of compares and number of moves.