Design for Sorting

TASK:
Implement a testing harness for sorting algorithms using getopt.
Implement the four sorting algorithms Bubble Sort, Shell Sort, Quicksort, and Binary Insertion Sort.
Gather statistics about each sort and its performance such as the size of the array that is being sorted, the number of moves required, and the number of comparisons required.

APPROACH:
Use Python pseudocode provided in lab document as a guide to generate to each sort. Implement each sort into their own header and C files and have them also return how many elements, moves, and compares they make. (Quicksort may require extern variables as it is a recursive function)
Use srand and rand to generate a psuedo-random array, allocated by calloc, and then have it be sorted given arguments parsed through by getopt.

PRE-LAB Part 1:
1. 6 swaps
2. n^2 comparisons

PRE-LAB Part 2:
1. The time complexity for shell sort depend on the size of the gap as it is the first for loop; increasing the amount of times that for loop loops causes an exponential increase in the amount of times the other for loops run. To improve the time complexity, you need to decrease the gap size.
2. You could swap multiple elements at once to avoid changing the gap size.

PRE-LAB Part 3:
1. Quicksort isn't doomed by its worst case scenario as you can use a random index or the middle index of the partition for the pivot variable
Source: https://www.geeksforgeeks.org/when-does-the-worst-case-of-quicksort-occur/

PRE-LAB Part 4:
1. The binary search algorithm will reduced the number of insertions as it finds the correct location for inserting as you don't need to look at the whole array, decreasing the time complexity.

PRE-LAB Part 5:
1. For every sort but quicksort, I'll implement local variables that keep track of the numbers of moves and comparisons each sort makes and have them print if. For quicksort, as it is recursive, I'll use extern variables and then print them in sorting.c before printing the sorted array.

Pseudocode:

Bubble Sort

```
def Bubble_Sort(arr):
  for i in range(len(arr) - 1):
    j = len(arr) - 1
    while j > i:
      if arr[j] < arr[j - 1]:
        arr[j], arr[j - 1] = arr[j - 1], arr[j]
      j -= 1
  return
```

Shell Sort

```
def gap(n):
  while n > 1:
    n = 1 if n <= 2 else 5 * n // 11
    yield n

def Shell_Sort(arr):
  for step in gap(len(arr)):
    for i in range(step, len(arr)):
      for j in range(i, step - 1, -step):
        if arr[j] < arr[j - step]:
          arr[j], arr[j - step] = arr[j - step], arr[j]
  return
```

Pseudocode Cont.:

Quick Sort

```
def Partition(arr, left, right):
  pivot = arr[left]
  lo = left + 1
  hi = right

  while True:
    while lo <= hi and arr[hi] >= pivot:
      hi -= 1

    while lo <= hi and arr[lo] <= pivot:
      lo += 1

    if lo <= hi:
      arr[lo], arr[hi] = arr[hi], arr[lo]
    else:
      break

  arr[left], arr[hi] = arr[hi], arr[left]
  return hi
```

```
def Quick_Sort(arr, left, right):
  if left < right:
    index = Partition(arr, left, right)
    Quick_Sort(arr, left, index - 1)
    Quick_Sort(arr, index + 1, right)
  return
```

Binary Insertion Sort

```
def Binary_Insertion_Sort(arr):
  for i in range(1, len(arr)):
    value = arr[i]
    left = 0
    right = i

    while left < right:
      mid = left + ((right - left) // 2)

      if value >= arr[mid]:
        left = mid + 1
      else:
        right = mid

    for j in range(i, left, -1):
      arr[j - 1], arr[j] = arr[j], arr[j - 1]

  return
```

```
create_rand_arr(seed, array length)
  set srand to seed
  create random array using calloc with length as
    array length and size of each item uint32_t
  check if calloc failed

  for i in range of array length
    set a temp variable to a random number
    set converted variable to temp masked to make it 30 bits
    add converted variable to random array[i]

  return random array
```

```
print array(array, array length)
  for i in range 1 to array length
    print array[i - 1] in format
  if printing 7th number of a row
    print a newline and start next row
```

```
int main(command line arguments)
  while getopt parses through arguments
    if -A, run all sorting algorithms
    if -b, run bubble sort
    if -s, run shell sort
    if -q, run quicksort
    if -i, run binary insertion sort
    if -p, read number after and set it to
      the number of elements printed from
      the sorted array
    if -r, read number after and set it to
      seed for srand
    if -n, read number after and set it to
       array size

for each sort, create a random array using
create_rand_arr

when running quicksort, print number of elements,
moves, and compares right after
```

Design Process:

Sorts were relatively easy to implement given the pseudocode provided by lab manual. Shell sort, however, was slightly complicated as for the gaps function, you cannot yield in C like you can in Python. I decided for the gaps function to instead produce an array with each gap, causing me to include it in the shell sort function as I couldn't return an array with it. I also initialized the gaps array to be a large number so it would be able to hold a large number of gaps up to a certain point; it requires more than 1000 gaps to break shell sort, however.

I had each sort calculate moves and compares and then print out the number of them along with elements at the end of the function. This was not possible with Quicksort as it is a recursive function, so I decided to use extern variables declared in the header file and calculated in the c file. Afterwards, in the main function in sorting.c, before printing the Quicksorted array I would print the elements and the number of moves and compares specific to quicksort.

Sources:

All sort functions were derived from Asgn 5 lab manual.