Part 1

1. It would take going through a list 6 times in order to sort all the numbers completely.
2. For every element that is to be sorted, bubble sort will do n - 1 comparisons

Part 2

1. For a small list of elements shell sort is able to efficiently break the list into smaller subsections and compute each subsection. The problem with shell sort is that as the list gets larger, so do the gaps, which can "produce various complexity between O(n) and O(n2)" (codingeek.com). The most efficient, according to wikipedia, is the gap sequence 1, 4, 10, 23, 57, 132, 301, 701. Which cannot be generated, so you must know the upper bound of the data to be sorted.
2. If I were to try and increase the runtime I would generate my list within my shell sort in order to avoid using a second function.

Part 3

1. There are two reasons why quicksort is often cited as better. The first is that the average runtime for quicksort is actually just O(nlogn), which is much faster than its counterparts. This means that in practice, quicksort often wins out in efficiency. The second reason is that it works well with

virtual-memory environments due to its use of in-place sorting. (stackexchange.com).

Part 4

1. When combined with the binary search algorithm, the insertion sorts time complexity of O(n) will drop down to O(nlogn). This is due to the improvement in efficiency of the binary search algorithm. Together the binary Insertion sort's worst case is O(nlogn).

Part 5

1. Two of the sorts seem to have an easy way of printing the values as part of the sort, while the other two are recursive which won't allow for that strategy. For those two I will likely define a global variable and use that to keep track.

Pseudo-Code

The code for my sort implementations are all based off of the pseudo code provided in the lab document, and the header files are simple so I will provide the flow of my program and pseudo-code for the sorting.c file.

sorting.c

```
#include "binary.h"
#include "bubble.h"
#include "quick.h"
#include "shell.h"
#include <getopt.h>
#include <math.h>
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>
#define OPTIONS "Aiqsbp:r:n:"

Function printN(NumPrint, arr):
  For i in range (1, NumPrint):
    print("%13u", a[i - 1])
    If (i % 7 == 0 && i!=0):
        printf("\n")

  printf("\n")
  return


Function copyAr(int *a, int *b, int size):
  For i in range (0,size):
     a[i] = b[i];


Main(argc, **argv):
   bubble = false;
   print = false;
```

```
shell = false;
quick = false;
binary = false;
NumPrint = 100;
Seed = 8222022;
size = 100;
d = 0;
while((d = elements in getopt that match OPTIONS) != -1):
        Switch (d):
            case 'A':
                    binary = true;
                    quick = true;
                    shell = true;
                    bubble = true;
                    break;

            case 'i':
                    binary = true;
                    break;

            case 'q':
                    quick = true;
                    Break;

            case 's':
                    shell = true;
                    break;

            case 'b':
                    bubble = true;
                    break;

            case 'p':
                    print = true;
                    NumPrint = given value;
                    break;

            case 'r':
                    Seed = given value;
                    break;

            case 'n':
                    size = given value;
                    break;
```

```
If (Size == 0):
        Handle error

if (NumPrint > size):
        Handle Error

myArray = []
srand(Seed)

For i in range(1,size+1):
        myArray[i-1] = rand() & 0x3FFFFFFF




If binary:
        Print("Binary Header")
        b  =  copyA(myArray)
        binary_sort( b, size )
        Print(size, movesB, comparesB)
        PrintN(NumPrint, b)
        free(b)
If quick:
        Print("quick Header")
        b  =  copyA(myArray)
        quick_sort( b, size )
        Print(size, movesQ, comparesQ)
        PrintN(NumPrint, b)
        free(b)
If shell:
        b  =  copyA(myArray)
        quick_sort( b, size )
        PrintN(NumPrint, b)
        free(b)
If bubble:
        b  =  copyA(myArray)
        quick_sort( b, size )
        PrintN(NumPrint, b)
        free(b)

free(myArray)
```

The basic layout of my main file is ***including libraries -> defining bools and vars -> parsing command line -> changing bools/values depending on user input -> error checking ->***

-> Generating main array -> performing each sort on a copy of main array I prefer this layout because it allows me to segment my work into easily readable chunks. The only variables not included in main is movesB, comparesB, movesQ and comparesQ. These are how I kept the values for moves and compares for the quick and binary sorts, they are easily defined in header files of each respective sort.

For the Flow of my program, after the command line is read and the main array is generated 4 “If” statements layout my sorts. If the user asks for a specific sort, then the if statement will create a copy of the main Array holding all the randomized values, and put it through the sort. To do this, the sorting.c file only ever calls the header files of each sort, never the .c implementation itself. After the sort is completed, the array will be printed along with the moves and compares, then the copy will be cleared from memory and the next if statement will repeat this process.