Design

Prelab

Part 1

1. At least 20 swaps.
2. Worst case you can expect n^2 comparisons

Part 2

1. The time complexity depends on the gap because the bigger the gap the longer the code has to run causing an increase in time complexity.
2. You can improve runtime by maybe instead of doing a for loop, doing a while and just calculating the next gap instead of yielding

Part 3

1. It isn't doomed because of the recursion because the recursion allows when the function is running to solve multiple numbers at a time while being in the middle of sorting another number

Part 4

1. The binary search allows the algorithm to find the correct location for the number to be sorted thereby narrowing down the number of comparisons.

Part 5

1. I plan to keep track of the number of moves and comparisons by doing it within the function including the print statement, keeping everything in the same scope, for this lab that is perfectly fine since when running a certain algorithm we also need to print it.

The design of the program can be split into five parts:

Bubble.c, .h (has the bubble sort algorithm)

Shell.c, .h (has the shell sort algorithm)

Quick.c, .h (has the quick sort algorithm)

Binary.c, .h (has the binary insertion sort)

Sorting.c (has the getopt code to allow for command-line options and creates the algorithm of the code, also has the print function for printing the sorted lists)

Bubble.c and bubble.h file.

The design for the bubble sort algorithm came from the lab manual where it gave the supported pseudocode for the algorithm. I had to add a variable to the function, length, which is the length of the array, c does not have a len function within the standard library. For swapping the numbers, I had to create a temporary variable, temp, since how they did it in the pseudo-code is not possible. Below is the pseudo-code give from the lab manual.

```
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
```

I also created the variables moves and comparisons, so whenever it compares the numbers, it increments comparisons, and every time they swap it increases moves by three since there is a temp variable used. At the end of the bubble_sort function, I added two print statement printing types of sort, and the number of elements, comparison, and moves.

Shell.c and shell.h file.

The design for the shell sort algorithm came from the lab manual where it gave the supported pseudocode for the algorithm. I had to add a variable to the function, length, which is the length of the array, c does not have a len function within the standard library. For swapping the numbers, I had to create a temporary variable, temp, since how they did it in the pseudo-code is not possible. Below is the pseudo-code given from the lab manual.

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

gap (pseudocode)

```
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
```

Shell Sort (pseudocode)

Since the gap function had a yield statement I changed it to next_gap and made the function that finds the next gap and calls it everytime when going through the while loop instead of the first for loop. I also created the variables moves and comparisons, so every time when it compares the numbers, it increments comparisons, and every time they swap it increases moves by three since there is a temp variable used. At the end of the shell_sort function I added two print statement printing types of sort, and the number of elements, comparison and moves.

Quick.c and quick.h

The design for the quick sort algorithm came from the lab manual where it gave the supported pseudocode for the algorithm. I had to add a variable to the function, length, which is the length of the array, c does not have a len function within the standard library. For swapping the numbers, I had to create a temporary variable, temp, since how they did it in the pseudo-code is not possible. Below is the pseudo-code given from the lab manual.

```
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
```

I also created the variables moves and comparisons, so every time when it compares the numbers, it increments comparisons, and every time they swap it increases moves by three since there is a temp variable used, what is different from previous algorithm is that the variables are at global scale because there is a recursion so this is to prevent the variable to be reinitialized every time quick_sort is called. At the end of the quick_sort function I added two print statement printing types of sort, and the number of elements, comparison and moves. To prevent the statistics to be printed every time the function is called, I added an if statement saying that if right is all the way at the end of the list and left is at the beginning of the list meaning all are sorted, then print the stats.

Binary.c and binary.h

The design for the binary insertion sort algorithm came from the lab manual where it gave the supported pseudocode for the algorithm. I had to add a variable to the function, length, which is the length of the array, c does not have a len function within the standard library. For swapping the numbers, I had to create a temporary variable, temp, since how they did it in the pseudo-code is not possible. Below is the pseudo-code given from the lab manual.

```
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
```

I also created the variables moves and comparisons, so whenever it compares the numbers, it increments comparisons, and every time they swap it increases moves by three since there is a temp variable used. At the end of the binary_insertion function, I added two print statement printing types of sort, and the number of elements, comparison, and moves.

Sortation.c

Design

Included all the sorting header files

Prints array

print_array(a, lenght)

Iterates through the array and prints the array in rows of 7

Makes a copy of the array, this is because for each of the sorting algorithms it has to start with the same unsorted array so made function that iterates through the array and sets one array equal to the other

copy_array(array, new_array, lenght)

For i in length array

New_array[i] = array[i]

Main function

Sets seed, first_elements, and seed to default

Set all getopt cases false

getopt code

While loop through the command line arguments

If p

Sets number give after p to first_elements

If r

Sets number give after p to seed

If n

Sets number give after p to size_array

If A

Sets all the algorithms ture

If b

Sets bubble true

If s

Sets shell true

If q

Sets quick true

If i

Sets insertion true

If no args is given

Sets all algorithms to true

If given number after n is less than one

Returns an error message

Allocates memory for array

Call srand(seed)

Creates random numbered lists usin rand() with bit mask of 1073741823

If the algorithm cases from getopt loop are set true do the following for the algorithms that are set true

Allocate space for array

Check if array is allocated

Call copy function with new array created and the random numbers array

Call the sorting algorithm that was set true with the new array which is a copy of the original array

Call the printing function of the sorted array

Free the new array created

Free the original array