DESIGN

Pre-Lab pt1:

1) Since there are 7 elements, there will be a total of 6 rounds of swapping because, by the end of the 6th round of sorting, there will be no 2 distinct elements left to be compared.
2) In the first pass, it will make 6 comparisons. At the end of the first pass, the last element would be the largest of the array. Then on the second run, it will only go up to the second to last element which means a total of 5 comparisons with the same result at the end of the run. This loop repeats until there are no more comparisons that can be made. So, in worst case scenario there would be 6+5+4+3+2+1 = 21 swaps.

Pre-Lab pt2:

1) The worst time complexity for shell sort is when the array is in descending order or when the size of the gaps are relatively small compared to the size of the array since that would need more swaps to finish.
2) To improve runtime, a more efficient gap sequence can be used to generate the gap sizes. Instead of the gap sequences(in the lab pdf) of floor(5*n/11), a more efficient algorithm could be floor(3*n/4)

Pre-Lab pt3:

1) Since the worst case would be if the array is ordered(ascending or descending) and if the pivot was a min or max values, choosing the pivot through ways like taking the median of the first, last and middle element or using the element of the middle index will minimize the possibility of the worse case occurring even for arrays that're ordered. Website source: https://www.geeksforgeeks.org/when-does-the-worst-case-of-quicksort-occur/

Pre-Lab pt4:

1) When combined, the binary search algorithm takes less time to figure out the where an element of the array should be. Compared to insertion sort algorithm, where an element is swapped until it's in the right position, binary insertion sort uses the fact the array is correctly ordered up till the element to be checked and just performs binary search to place it in the right position. The binary search algorithm is quicker in terms of time complexity because of the divided search approach compared to merely just checking

each and all elements to determine the right position(which is what insertion sort does). So adding binary search makes the average and worst case time complexity of insertion sort better.

Pre-Lab pt5:

1)Pass in comparison variable to swapping functions that check if elements should be swapped. The function returns true or false to determine if elements should be swapped and increments comparison counter regardless. The function is called in each sorting algorithm to see if elements need to be swapped. A counter variable for number of moves will be passed in each sort function. The number of moves made(counts every time a condition of comparison between elements is met) will be incremented by 3 if swapping function returns true.

Main program flow: Program determines the sorts to run, the size of the array to be sorted, the number of sorted elements to be printed and the random seed based on the flags passed in. Boolean and integer variables are used to store the algorithms that are to be run and nonoption arguments passed in with flags “-n”, “-p” “-r” respectively.

Functions for creating and deleting array:

```
def createArray(input length) {
  New_array = allocated memory of size(length * size of uint32_t).
  return array
}

def deleteArray(uint32_t array[]) {
  Free memory allocated to array[]
  Array = points nowhere(NULL)
 }
```

Function for generating 30 bits of random values.

```
def setrandomGen(input array_length, array[], random_seed) {
  Set random seed by passing in random_seed
  Mask_number = 0x3FFFFFFF
  For loop iterating over size of array:
    generate random_number with rand()
    array[iterator] = Random_number AND Mask_number
 }
```

```
Function to output result from the sorting algorithms desired
def PrintElements(input: to_bubble, to_binary, to_quick, to_shell, comparison_counter,
swap_counter, array_length, array[], numPrintElements, random_seed) {
 If to_binary is true:
    Run binarysort(array, comparison_counter, swap_counter, array_length)
    For loop iterating till numPrintElements:
      Print array[iterator]
      new line after 7 elements
    Reset swap_counter and comparison_counter back to zero.
    run setrandomGen(input: array_length, array[], random_seed) //set random array back again
If to_quick is true:
    Run quicksort(array,0, array_length-1, comparison_counter, swap_counter, array_len)
    For loop iterating till numPrintElements:
      Print array[iterator]
      new line after 7 elements
    Reset swap_counter and comparison_counter back to zero.
    run setrandomGen(input: array_length, array[], random_seed)//set random array back again
If to_shell is true:
    Run shellsort(array, comparison_counter, swap_counter, array_length)
    For loop iterating till numPrintElements:
      Print array[iterator]
      new line after 7 elements
    Reset swap_counter and comparison_counter back to zero.
    run setrandomGen(input: array_length, array[], random_seed)//set random array back again
If to_bubble is true:
    Run bubblesort(array, comparison_counter, swap_counter, array_length
    For loop iterating till numPrintElements:
      Print array[iterator]
      new line after 7 elements
}

Main function flow:
int main() {
  Read options from command line using getopt() to bool and int variables for storage(using
switch statement)
  Run createArray
  Pass to PrintElement function
  deleteArray to free memory
  Return 0;
}
```

Pseudocode for swapping function(general form)

```
def swap(comparison_index, value_for_comparison, comparison_counter, array[]) {
  Swap = false
  if array[comparison_index] >= or <= value_for_comparison:
    Swap = true
  comparison_counter += 1
  Return Swap
}
```

For each sorting algorithim, when swap is called AND returns TRUE, then the swap_counter is incremented by 3.

Pseudocode from sorting algorithms based on and provided by Darrell Long in assignment PDF.