Despina Patronas
Assignment 5 – Sorting
Design Document

Pre-Lab Part 1:

1) Q: How many rounds of swapping to sort the numbers below

8,22,7,9,31,5,13 //Initial sort n (n=7 elements)

Round 1) //n
Swap 1: 8,7,22,9,31,5,13
Swap 2: 8,7,9,22,31,5,13
Swap 3: 8,7,9,22,5,31,13
Swap 4: 8,7,9,22,5,13,31 //31 is set in round 1

8,7,9,22,5,13,31
Round 2) //n-1
Swap 5 7,8,9,22,5,13 **//31**
Swap 6 7,8,9,5,22,13
Swap 7 7,8,9,5,13,22 //22 is set in round 2

7,8,9,5,13,22,31
Round 3) //n-2
Swap 8 7,8,5,9,13 **//22,31** //13 is set in round 3

7,8,5,9,13,22,31
Round 4) //n-3
Swap 9 7,5,8,9 **//13,22,31** //9 was set from previous round, after this round is excluded

7,5,8,9,13,22,31
Round 5) //n-4
Swap 10 5,7,8 **//9,13,22,31** //8 was set from previous round

5,7,8,9,13,22,31 //already sorted at this point. Final check of last 2 numbers finalizes sort
Round 6) //n-5
5,7 **//8,9,13,22,31** //5 and 7 set in place and now the bubble sort is complete

A:
7 elements took **6 rounds** and 10 swapping and 21 comparisons
Bubble sort will iterate through **n-1 rounds of comparisons based on the initial number of elements n**

2) Q: How many comparisons for the Worst Case Scenario of Bubble Sort?

Worst Case, n is numbers is sorted in reverse order so all numbers have to be checked and exchanged each round
A: In worst case scenario: swaps = comparisons = [n(n-1)]/2 //worst == O(n^2) time complexity

(note: in a best case scenario: 0 swaps and only 1 pass //best == O(n) time complexity

Pre-Lab Part 2:

Q1: The worst time complexity for shell sort depends on the size of the gap. Investigate why this is the case. How can you improve the time complexity of this sort by changing the gap Size?

A: To improve the time complexity of QuickSort, using a different gap size could reduce the number of moves and comparisons.

Q2: How would you improve the runtime of this sort without changing the gap size?

A: If you chose a pivot that is the median

Pre-Lab Part 3:

Q1: Quicksort with a worse case time complexity = O(N^2)
Investigate and explain why quicksort can perform better than its worse case scenario

A: Randomly picking a bad pivot can disparage the balance of the subarrays drastically, but luckily you can **choose your pivot to prevent the random worse case option altogether.**

Pre-Lab Part 4:

Q1: What is the effect of the binary search algorithm on the complexity when it is combined with the insertion sort algorithm?

A: Effectively lowers the amount of comparisons without changing the time

Source: Eugene (Wednesday: 11-11 Lab / Office Hours)

Pre-Lab Part 5

Q1: Explain how you plan to keep track of the number of moves and comps since each sort will reside within its own file.

A: I will be using a global variables to accomplish this task

CREDITS:

Pre-Lab part 4 explaination by :
Eugene Lab / Office Hours: Wednesday (11/11)

Implementation for the sorting algorithms Bubble, Binary Insertion, and Quick Sort Provided by:
Lab Manual Assignment 5 (Algorithm Pseudocodes)

Implementation for the sorting algorithm QuickSort provided by :
TextBook source: "the C Programming Language"
by Brian W. Kernighan and Dennis M. Ritchie
(refer to page 62 for pseudo code / implementation)

//implementation of sorts – BUBBLE SORT

Bubble.h

```
    //declare the global static variables to keep track of moves and comparisons
    Int B_moves = 0
    Int b_comps = 0

    //Declare prototype headers

    //print the array in columns of 7
    //length = elements in arr[]
    Void print(int arr[], int length, int print_len)

    //function to swap two elements of array (arr[]) indexes I & array element left to i
    //increments moves by 3
    Void swap ( int arr[], int i)

    //function to compare two elements of array indexes (i) & array element left to i
    //returns true if arr[i] < arr[i-1]
    //increments count by 1
    Bool comp(int arr[], int i)

    //performs the search then calls the print function
    Int bubble_sort(int arr[], int length, int print_len)

End Bubble.h
```

Bubble.c

```
    //this function will print the results of sorting factors: elements, moves, and comps and the sorted array itself
    // Note: this function exists in every sorting source file and does not change
    define print(int arr[], int length, int print_len)

        Display "bubble sort"
        Display "elements" and length
        Display "moves" and b_moves
        Display "compares" and b_comps

        Iterate through I =[0 , length)
            Print arr[i]                //print the index
            If ( I MOD 7 == 6 )         // if 6 elements have printed
                Print a new line        // generate columns of 6
            Increment i
        End iteration

    End print
```

```
//bubble sort simply swaps with the element next to it.
//takes in the initial element and swaps with the element next to it
//
define swap ( int arr[], int index)

        Int temp = arr[i]           //hold the intermediate step in temp variable
        Arr[i] = arr[i-1]           // this one just swaps with the element to the left
        Arr[i-1] = temp             //final move in the swap. Total moves = 3
        B_moves += 3                //increment the moves for bubble sort by 3

End swap


//bubble sort simply compares with the indexed element next to it
//similar methodology as the swap
//returns true of false on the compare
//true will trigger a swap in this case
//
define comp (int arr[], int I )
                                                                  //left > right
        If ( arr[i] < arr[i-1] )          // if value of element at index i-1 > value of element index I in array
                Increment B_comps         //add 1 to comp
                Return True
        otherwise
                Add 1 to comp             // even if it was not successful, add 1 to comp
                Return false

End bool comp


// bubble sort implementation
//compares every element
//swaps in sequence of array traversal
//
define bubble_sort( int arr[], int length, int print_len)

        Int j = 0;

        Iterate through I = [0, length-1)  //outer loop denotes the elements of the array to traverse each round
                Set j = length -1                 // j holds the end index

                While( j >I )                     //while index is less than the end
                        If (comp (arr, j)         //if arr[j-1] > arr[j] aka the next element is smaller
                                Swap(arr, j)      //swap the elements arr[j-1] <-> arr[j]
                        Endif
                Endwhile
                Decrement j               //largest element now in place, check one less element in the while loop
        End iteration
        Print(arr, length print_len)      // print the results of comps, moves, and array itself

END bubble_sort
```

//implementation of BINARY SORT

Binary.h

```
//declare the global static variables to keep track of moves and comparisons
Int i_moves = 0
Int i_comps = 0

//Declare prototype headers

//print the array in columns of 7
//length = elements in arr[]
Void print(int arr[], int length, int print_len)

//function to swap two elements of array (arr[]) indexes I & array element left of I
//increments moves by 3
Void swap ( int arr[], int i)

//function to compare two values fed into function
//returns true if val1 >= val2
//if called increments count by 1
Bool comp(int arr[], int i)

//performs the search then calls the print function
Int binary insertion sort (int arr[], int length, int print_len)
```

End Binary.h

Binary.c

```
//this function will print the results of sorting factors: elements, moves, and comps and the sorted array itself
// Note: this function exists in every sorting source file and does not change
Define print(int arr[], int length, int print_len)

    Display "binary insertion sort"
    Display "elements" and length
    Display "moves" and i_moves
    Display "compares" and i_comps

    Iterate through I =[0 , print_len)
        Print arr[i]                //print the index
        If ( I MOD 7 == 6 )         // if 6 elements have printed
            Print a new line        // generate columns of 6
        Increment i
    End iteration

End print
```

```
    //note this function is the same as the one used in bubblesort
    //takes in the initial element and swaps with the element next to it
    //
    define swap ( int arr[], int index)

        Int temp = arr[i]           //hold the intermediate step in temp variable
        Arr[i] = arr[i-1]           // this one just swaps with the element to the left
        Arr[i-1] = temp             //final move in the swap. Total moves = 3
        i_moves += 3                //increment the moves for bubble sort by 3

    End swap


    //function to compare two values (fed from arrays)

    define comp( int val1, int val2)

         if (val1 >= val2) {
            increment i_comps
            return True

        otherwise
            i_comps++;      //in either case, increment the comparisons
            return False
        endif

    end comp



    define binary_insertion( int arr, int length, int print_len)

        iterate from I [1, length)
            declare value = arr[i]
            declare left = 0
            declare right = I

            while left < right                      //the barrier
                mid = left + ((right – left) / 2)   //middle value floored
                if ( comp(value, arr[mid] )         //if arr[mid] < value of arr[i]
                    set left = mid +1               //move the left value
                otherwise
                    set right = mid                 // otherwise move the right value
                end if
            iterate from j [left, i]            //sort the elements that are in range left and current index
                swap (arr, j)                   // arr[j] <-> arr[j-1]
                decrement j
            end Iteration

            increment I                         //move onto the next element in array
        end iteration
        print (arr, length, print_len)          //print the array one sort is complete
    end binary_insertion
end binary.c
```

//implementation of SHELL SORT

shell.h

```
//declare the global static variables to keep track of moves and comparisons
Int s_moves = 0
Int s_comps = 0

//Declare prototype headers

//print the array in columns of 7
//length = elements in arr[]
Void print(int arr[], int length, int print_len)

//function to swap two elements of array (arr[])
//j and gap are two variables indexes to be swapped
//returns  arr[j+gap] <-> arr[gap]
Void swap ( int arr[], int j, int gap)

//function to compare two elements of array indexes (i)
//returns true if arr[j+1] < arr[gap]
//if called increments count by 1
Bool comp(int arr[], int j, int gap)

//This implementation appears in:
//page 62 of "the C Programming Language"
//by Brian W. Kernighan and Dennis M. Ritchie
//function to do the shell sort of array (arr[])
//length = elements in arr[]Int
Shell_sort (int arr[], int length, int print_len)
```

end shell.h

shell.c

```
//this function will print the results of sorting factors: elements, moves, and comps and the sorted array itself
// Note: this function exists in every sorting source file and does not change
define print(int arr[], int length, int print_len)

    Display “shell  sort”
    Display “elements” and length
    Display “moves” and s_moves
    Display “compares” and s_comps

    Iterate through I =[0 , print_len)
        Print arr[i]                  //print the index
        If ( I MOD 7 == 6 )           // if 6 elements have printed
            Print a new line          // generate columns of 6
        Increment i
    End iteration

End print
```

```
    Define swap ( int arr[], int j, int gap )

        int temp = arr[j];              //temp to hold the intermediate step of swap
        arr[j] = arr[j+gap];            //swap
        arr[j+gap] = temp;              //swap
        s_moves += 3;                   //increment moves (3 per swap)

    end swap


    define comp ( int arr[], int j, int gap )

        if ( arr[ j+gap] < arr[j] )     //if array to the left of gap is larger then swap
            increment s_comps           //return true
            return true
        otherwise
            increment s_comps           //keep incrementing the comps
            return false
        endif
    end comp


    define shell_sort ( int arr[], int length, int print_len)

        int gap = 0         //the gap to create subarrays initialized

        iterate from [gap = 0, gap = length / 2)                    //iterate through the gaps
            iterate from I [gap, length)                            //step through elements in gap range
                iterate from j [ 0 && comp(j,gap), (I – gap) )      //comp the elements across gap
                                                            // if the elements to the left are larger, swap
                    swap (arr, j, gap)                      //swap the j and j+gap elements of array
                    decrement j by gap                      //j =j-gap
                end iteration
                increment I                 //increment the gap to the next element of array
            end iteration                   //keep comparing array values across the j range for gap
                                            //until values of index j+gap > j in array
        Gap = length / 2                //find the new gap value halved each time
        end iteration

        print( arr, length, print_len)          //print the results of sorts
    End shell_sort

End shell.c
```

Implementation of **quicksort**

Quick.h

```
//declare the global static variables to keep track of moves and comparisons
Int q_moves = 0
Int q_comps = 0

//Declare prototype headers

//print the array in columns of 7
//length = elements in arr[]
Void print(int arr[], int length, int print_len)

//function to swap two elements of array (arr[]) indexes
//low and hi are two variables indexes to be swapped
//returns  arr[low] <-> arr[hi]
Void swap ( int arr[], int low, int hi)

//function to compare two elements of array indexes (i)
//returns true if arr[val1] < arr[val2]
//if called increments count by 1
Bool comp(int arr[], int val1, int val2)

//function to perform quick sort
quick_sort (int arr[], int length)
```

end quick.h

Quick.c

```
//this function will print the results of sorting factors: elements, moves, and comps and the sorted array itself
```
// Note: this function exists in every sorting source file and does not change
```
define print(int arr[], int length)

    Display “quick sort”
    Display “elements” and length
    Display “moves” and s_moves
    Display “compares” and s_comps

    Iterate through I =[0 , length)
        Print arr[i]                  //print the index
        If ( I MOD 7 == 6 )           // if 6 elements have printed
            Print a new line          // generate columns of 6
        Increment i
    End iteration

End print

Define swap ( int arr[], int low, int hi )
    Int temp = arr[low]
    Arr[low] = arr[hi]
    Arr[hi] = temp
End swap
```

```
    Define comp( int arr[], int low, int hi )

        If ( val1 <= val2)
            Return true
            Q_comps++
        Otherwise return False
        Endif
        Q_comps++

    End comp


    Define partition ( int arr[], int left, int right )

        Int pivot = arr[left]        //first element to start
        Int low = left + 1           //the element directly right of pivot
        Int hi = right               //opposite side of left

        While (TRUE)                 //yes infinite loop

            //low and hi are keeping track of pos
            While ( low < = hi) && comp(pivot, arr[low]) )          // if value arr[low] >= the pivot
                Hi = hi -1                                          //decrement the hi on the right sub

            While ( low <= hi ) && comp(arr[low], pivot             //if value arr[low] < = the pivot
                Low = low +1                                        //increment the low on the left sub

            //low and hi are each in their respective subarrays

            //in this case arr[low] < = arr[hi]
            If ( comp(low, hi) )                        //compare the values of arr at current low and hi
                Swap ( arr, low, hi)
            Endif
                                   //if low > hi break in any case
            Otherwise break        //condition to break: ( low > hi || pivot > arr[low] ) subarray
        Endwhile                   //                    or :   ( low > hi || arr[low] > pivot ) subarray

        // arr[left] <-> arr[hi]
        Swap( arr, left, hi)
        Return hi                  //returns a new position for index to be used in recursive call
    End partition

    //recursive calls to quicksort to change partition
    Define quicksort ( int arr[], int left, int right )

        If ( left < right)
            Index = Partition ( arr, left, right)          //create a new index based on partition return
            Quicksort (arr, left, index -1 )                //sort based on new right = index -1
            Quicksort (arr, index -1, right )                   //sort based on new left = index -1
        End if

    End quicksort
End quick.c
```

//finally sorting.c

Sorting.c

```
    DEFINE BITMASK 0x3FFFFFFF      //decimal for (2^30) -1   //this is for the random numbers to be <= mask

    //helper functions
    //function to reinitialize the array for reuse

    Define reset_Array ( int arr, int length )

        Iterate through I [0,length)
            Arr[i] = 0
            Increment i
        End iteration

    End reset_Array

    //fill the array with values form rand seed
    Define fill_array ( int arr[], int length, int seed)

        Call Srand(seed)                        //reinstate the seed for replicating array values
        Iterate through I [0, length)
            Arr[i] = rand() & BITMASK           //value of I form rand ( <= (2^30) -1 )
            Increment I;
        End iteration

    End fill_array

    Define reset_array( int arr[], int length)           //function to recalibrate the array
        iterate from I [0, length)
            set arr[i] = 0

        End iteration

    End reset_array
```

```
Define main

    //Declare necessary variables
    Int c = 0
    Int length = 100          //default
    Int seed = 8222022        //default

    //Booleans for all the sorts used above
    //ex:
    Bool bubble
    //Boolean for all option optarg

    Bool all
    //don't forget to initialize all bools to 0

    //optarg input variables initialized
    Char get_len = NULL
    Char get_print = NULL
    Char get_seed = NULL

    While( ( c = getopt (argc, argv, 'Absqipr' ) != -1 )

        Case defined by c
            //ALL the sorts
            case 'A'
                set all = bubble = shell = quick = ibinary = 1
                exit case
            //bubble sort
            case 'b'
                set bubble = 1
                exit case
            //shell sort
            case 's'
                set shell = 1
                exit case
            //quick sort
            case 'q'
                set quick = 1
                exit case
            //binary insert search
            case 'i'
                set ibinary = 1
                exit case

            //print the first n elements of the array
            case 'p'
                get_print = optarg
                print_len = get_print     //perform the appropriate conversions if needed
                endif
                exit case
                //set seed
                case 'r'
                        get_seed = optarg
                        seed = ( get_seed)      //perform approp. type conversion if needed
                        exit case

                //set length of array (elements)
                Case ‘n’
                        Get_len = optarg        //perform approp. type conversion if needed
                        Length = get_len        //may perform some user bounds checking here as well
                        Exit case
        end c case
    end while

    //error check the getopt
    If (argc == 1 )
        Return -1
    Endif

    Int * arr = allocate ( length, sizeof (int) )       //allocate the array on the heap with size length

    Fill_array (arr, length, seed)                      //feed seed into function and populate alloc arrray

    //if getopts are triggered start the sorting
    //if all is triggered all will run in sequence
    //all printing for sorting occurs within the sorting files except QUICKSORT

    if (bubble) {
        bubble_sort(arr, length, print_len) //perform and print BUBBLE SORT
        reset_array(arr, length)             //reset and refill with same rand seed if all was triggered
        fill_array(ar, length, seed)        //fill with same seed

    if (shell) {
        shell_sort(arr, length, print_len) //perform and print SHELL SORT
        reset_array(arr, length)             //reset and refill with same rand seed if all was triggered
        fill_array(ar, length, seed)        //fill with same seed

    if (quick) {
        quick_sort(arr, 0, length-1)        //perform QUICK SORT
        printq( arr, length)                //print outside of the function
        reset_array(arr, length)             //reset and refill with same rand seed if all was triggered
        fill_array(ar, length, seed)        //fill with same seed

    if (ibinary) {
        binary_insertion(arr, length, print_len) //perform and print BIN_INSERTION SORT
        reset_array(arr, length)             //reset and refill with same rand seed if all was triggered
        fill_array(ar, length, seed)        //fill with same seed

    free(arr)                               //** free heap of allocated array

    return 0;
  end main
end sorting.c
```