**Problem 1 follows…**  
**Description:** Problem 1 is designed to demonstrate my knowledge of how different sorting algorithms work, in addition to what their memory and performance characteristics are.

**Answers to Questions:**

1. How does a Selection sort work? Provide a detailed description.

A selection sort arranges data in a prescribed sequence, such as lowest value must equal the lowest element. It is an in-place sorting mechanism because it swaps an unsorted portion of the data structure until all data is in sorted order. It does not change the size of the data structure. A data structure is broken into sorted and unsorted sections. It is only considered "sorted" until the first unsorted data. Then the rest of the data structure is considered "unsorted". Swapping the lowest element with the first element of the unsorted section of the data structure based on a prescribed sequence will eventually sort the data structure. Each time unsorted data gets swapped, the length of sorted data increases by 1 and the length of unsorted data decreases by 1. This shrinks the unsorted section of the data structure until the entire data structure is sorted.

2. Does it require extra memory other than for the array elements? If so, why?

The selection sort does not require extra memory other than for the array elements. It does not change the data structure's size. It simply swaps unsorted data with sorted data until the data structure is sorted. This can be completed using two loops, a comparison of values and a swap. None of these require dynamically allocating memory.

3. If an array had 10 elements to be sorted, how many passes would it take with this approach?

If 10 elements need to be sorted, then 9 passes must be made. The number of passes needed can be found by calculating n-1 where n is the number of elements. In this case n-1 would be 10-1 or 9 passes. Each element must be checked with the rest of the array to check if unsorted data belongs at that element or not. At the end of checking every value with one element, it may or may not swap a value then it moves onto the next element. Once n-1 elements have been swapped, there will no longer be unsorted data to swap because the end of the data structure will have been reached. At the end of this, the data structure will have been properly sorted.

4. How does an Insertion Sort work? Provide a detailed description.

An insertion sort is like a selection sort in that it checks the data structure to swap unsorted data based on a prescribed sequence. However, an insertion sort does not sort an existing number of elements. Instead, each time a sort occurs a new element is added, increasing the data structure's size by 1 to accommodate the element.

5. Does it require extra memory? If so, why?

An insertion sort requires additional memory if a new element is being added. If not, it is in-place and does not require extra memory. This is because a sort which requires a new element to get added into the data structure needs memory to be allocated for it.

6. If an array had 10 elements to be sorted, how many passes would it take with this approach?

If 10 elements need to be added and sorted, then 9 passes must be made. The number of passes needed can be found by calculating n-1 where n is the number of elements. In this case n-1 would be 10-1 or 9 passes. The first element to be added does not need to be compared to any value (because no other values exist in the data structure) and therefore simply gets added without a pass through the array. Each new element must be checked with the rest of the array to check if it belongs before or after an element based on a prescribed sequence. At the end of checking values with the new element, it may or may not swap the new value until it is in order then it moves onto the next newly added element (if any).

7. How does a Heap Sort work? Provide a detailed description.

A heap sort arranges data in a heap (an almost complete binary tree) by reordering array elements to an array-based binary tree. The array-based binary tree will either be a max-heap or a min-heap based on whether the root is a maximum or minimum respectively. It starts with the last non-leaf node and swaps it with its child's value based on a prescribed sequence. This repeats, moving right to left and up the tree, until the top is reached. When a swap occurs, the subtree must be revisited and checked again to conclude that the prescribed sequence is met (i.e. the max/min is moving closer to the root of the tree). This assigns the max/min to the root, but the array still might not be sorted. Sorting now entails moving the root to the last element and reforming the heap except for the last item (shrinking the heap size by numArrayElements-i, where i is the number of current iterations through this loop). This repeats until the array size has been reduced to 1 and the sort is complete.

8. What are Heapify, siftUp, siftDown? What are they used for? Why does Heapify processing start at ListLength/2?

Heapify means arranging data in an almost complete binary tree following a prescribed sequence. This sequence might be a max-heap where the maximum remains at the root or a min-heap where the minimum remains at the root. The binary tree must be filled at each level except possibly the lowest level. Each node must have zero, one or two children. Heapify processing starts at ListLength/2 because it represents the number of non-leaf nodes (ones with children). It starts at this value to go through the non-leaf nodes and compare its value with the values of its children. It works its way right to left and up the tree. It makes its time complexity linear, O(N), in the worst case.

SiftUp and SiftDown iterate across parent nodes from right to left and up the tree. They compare the parent's value with its children's values, swapping them if the child is larger/smaller depending on whether the heap should be a max-heap/min-heap respectively. SiftUp swaps larger values, SiftDown swaps smaller values. Once this is complete, the max or min will be the root.

9. Does it require extra memory beyond the nodes it holds? If so, why?

A heap sort does not require extra memory beyond the nodes it holds. It simply swaps parent/child values based on a prescribed sequence to move values throughout the already allocated binary tree. If a swap occurs, then the subtree must be revisited and checked again to make sure the max/min is moving closer to the root of the tree. At the end, the tree will be sorted just by using SiftUp and SiftDown functions that do not need new memory to be allocated.

10. How does a Quick Sort work? Provide a detailed description.

Quick sort uses an element in the array as a pivot point, for example the middle of the array. It is used to compare elements that remain on its left side with elements that remain on its right side. If one element equals or is greater/less than the pivot point, it gets moved to its respective side (left or right). The rearranging is done when all elements greater than the pivot are to one side and all elements less than the pivot point are to the other side. Then all that is needed is a recursive sort of the left and right sides. After both sides are sorted, the entire array will be sorted.

11. Does it require extra memory other than for the array elements? If so, why?

The quick sort does not require extra memory other than for the array elements. It simply picks a pivot point and swaps values until its left and right sides contain values greater/less than the pivot point. Then each side is recursively sorted through swapping values.

12. How does a Merge Sort work? Provide a detailed description.

Merge sort splits an array into two halves, n/2. Then each half is recursively split in half until the size becomes just 1 element. Then two arrays are merged in sorted order, until all separated elements have been merged back into one full-sized array.

13. Does it require extra memory other than for the array elements? If so, why?

Merge sort requires additional memory to store the splitting of arrays. However, an in-place merge sort could be constructed after creating a merge sort using pointers. However, its time complexity will be O(n^2) while a standard merge sort's complexity is O(n Log n).

14. How does a Shell Sort work? Provide a detailed description.

A shell sort compares two values an interval apart. It swaps them if necessary, then moves onto the next two values. Then the interval is set to 1 and the two sets of numbers become sub-lists. Insertion sort is used to sort the final array using the interval value 1.

15. Does it require extra memory other than for the array elements? If so, why?

Shell sort does not require extra memory other than for the array elements. It is in-place and sorts by insertion, without any new elements being added. It uses an interval to swap values, then an interval of 1 to conduct insertion sorting.

16. List the speeds of the 6 sorts from fastest to slowest as expressed by their Big-O notation. Where sorts are roughly the same speed, they can be in any order. Justify your answer by identifying the source of your information.

The 6 sorts listed in fastest to slowest average time:

Heap

Merge

Quick

Shell

Insertion

Selection

Heap, Merge and Quick sort have an average time of O(n\*log(n)) and Shell is slower at O(n^1.25). Insertion and Selection are O(n^2). This information can be found from https://www.bigocheatsheet.com/

I chose to order by average time. While there were three different times to pick from when creating my list, I felt the best/worst time complexity to be better used in special circumstances to identify when one algorithm should be used over another. Those time complexities may prove advantageous in a specific application being used or developed.

**Problem 2 follows…**  
**Description:** Problem 2 is designed to demonstrate building an insertion sort. It consists of functions and definitions for the insertion sort including the InsertionSort function to arrange data, filling an array with random integer data, printing the array data, determining order of elements low to high and high to low and runTest to provide the routine for testing the sort.

**Program Code (SortADTs.h):**

#ifndef SORTADTS\_H\_INCLUDED

#define SORTADTS\_H\_INCLUDED

// the definition of what the UserData typedef consists of

#include "UserData.h"

// we use bool true and false

#include <stdbool.h>

// Comparer is a typedef that gives us a shorthand way of expressing the

// comparison function that will be called by these sort functions.

// it says that Comparer is a function that receives 2 UserData and

// returns a bool. The intent is to compare "first" to "second"

// and return a bool.

// If the comparer is supposed to support a low to high sort,

// first < second returns true.

// If the comparer is supposed to support a high to low sort,

// first > second returns true.

// NEVER code a <= or >= on the comparison functions used for the

// sort functions!!

//

typedef bool Comparer (UserData first, UserData second) ;

// InsertionSort is a function for sorting an array by continuously swapping an element

// with the adjacent element on its left if it is greater/less than that element. It

// decides whether to sort by increasing/decreasing from a given boolean function which

// decides if two elements are greater/less than one another. InsertionSort is also given

// the array of UserData integers to sort and the number of elements it contains, so that

// the loop knows when to stop sorting. It returns a boolean.

void InsertionSort (UserData list[], int ListSize, Comparer);

// SelectionSort is a function for sorting an array by finding the next lowest/greatest

// element and swapping it with the front element. This repeats itself, each time ignoring

// the previously placed elements at the front. It decides whether to sort by increasing/decreasing

// from a given boolean function which decides if two elements are greater/less than one another.

// SelectionSort is also given the array of UserData integers to sort and the number of elements it

// contains, so that the loop knows when to stop sorting. It returns nothing.

void SelectionSort (UserData list[], int ListSize, Comparer);

// HeapSort is a function for sorting an array after arranging it in a heap (binary tree). It starts

// with the last non-leaf and swaps its value with its child's value if it is greater/less than that

// element. This repeats right to left and up the tree until a maximum/minimum remains as the root.

// Then the root is moved to the last element and the heap is reformed except for the last item.

// This repeats until the tree is sorted. It decides whether to sort by increasing/decreasing from

// a given boolean function which decides if two elements are greater/less than one another and if

// the heap should be a max-heap or min-heap. HeapSort is also given the array of UserData integers

// to sort and the number of elements it contains, so that the loop knows when to stop sorting.

// It returns nothing.

void HeapSort (UserData list[], int ListSize, Comparer);

// MergeSort is a function for sorting an array by splitting an array in two halves. Then each half

// is recursively split in half until the array size becomes just 1 element. Then two arrays are

// merged in sorted order, until all separated elements have been merged back into one full-sized

// array. It decides whether to sort by increasing/decreasing from a given boolean function which

// decides if two elements are greater/less than one another. MergeSort is also given the array of

// UserData integers to sort and the number of elements it contains, so that the loop knows when

// to stop sorting. It returns nothing.

void MergeSort (UserData list[], int ListSize, Comparer);

// QuickSort is a function for sorting an array by taking a pivot point and moving elements greater

// to one side while moving elements less than to the other. Then a recursive sort of the left and

// right sides will leave the entire array sorted. It decides whether to sort by increasing/decreasing

// from a given boolean function which decides if two elements are greater/less than one another.

// QuickSort is also given the array of UserData integers to sort and the number of elements it contains,

// so that the loop knows when to stop sorting. It returns nothing.

void QuickSort (UserData list[], int ListSize, Comparer);

// ShellSort is a function for sorting an array by comparing two elements an interval apart. They get

// swapped if necessary, then the algorithm moves onto the next adjacent elements until all elements

// have been passed over. Then the interval is set to 1 and each element gets swapped with the adjacent

// element if it is greater/less than that element until it is in its final, sorted position. ShellSort

// is also given the array of UserData integers to sort and the number of elements it contains, so that

// the loop knows when to stop sorting. It returns nothing.

void ShellSort (UserData list[], int ListSize, Comparer);

#endif // SORTADTS\_H\_INCLUDED

**Program Output (3):**

Enter the maximum number of items to process (1-10) :3

Sort tests follow..

==================

Insertion low to high test follows:

Array initial contents are:-59 76 3

Sorted array contents are: -59 3 76

Insertion high to low test follows:

Array initial contents are:-59 76 3

Sorted array contents are: 76 3 -59

==================

**Program Output (10):**

Enter the maximum number of items to process (1-10) :10

Sort tests follow..

==================

Insertion low to high test follows:

Array initial contents are:-59 76 3 69 -26 -54 -79 -88 -72 43

Sorted array contents are: -88 -79 -72 -59 -54 -26 3 43 69 76

Insertion high to low test follows:

Array initial contents are:-59 76 3 69 -26 -54 -79 -88 -72 43

Sorted array contents are: 76 69 43 3 -26 -54 -59 -72 -79 -88

==================

**Problem 3 follows…**  
**Description:** Problem 3 is designed to demonstrate my knowledge of how different sorting algorithms work by fully documenting them and their code. This includes insertion sort, selection sort, shell sort and merge sort.

**Program Code (InsertionSort.c):**

// the definition of what the UserData typedef consists of

#include "UserData.h"

// declaration of the sort ADT(s) to be called

#include "SortADTs.h"

// we use bool true and false

#include <stdbool.h>

// InsertionSort is responsible for swapping an observed element to its left

// adjacent spot until it is in its final spot. It takes an array to arrange

// UserData integers for based on a boolean which decides whether to arrange

// low to high or high to low. It also takes an integer in case a data

// element were to be added. It returns nothing.

void InsertionSort (UserData list[], int n, Comparer ComesFirst)

{

// h is index of observed UserData

int h;

// iterate over every element in the array

// h starts at 1 so that k can exist left of it, at the first index

// it increments until all data is iterated over

for (h = 1; h < n; h++)

{

// assign observed element's data to temporary UserData type variable

UserData key = list[h];

// k is index left of temporary UserData

int k = h -1;

// while both k exists in the array and the less/greater than condition

// is met, swap observed element with its left adjacent element

// repeat this until the condition is no longer met and k remains

// at final spot for observed element to be placed

while (k >= 0 && ComesFirst (key, list[k]))

{

list[k+1] = list[k];

--k;

}

// place observed element at the furthest left index

list[k+1] = key;

}

}

**Program Code (SelectionSort.c):**

// the definition of what the UserData typedef consists of

#include "UserData.h"

// declaration of the sort ADT(s) to be called

#include "SortADTs.h"

// we use bool true and false

#include <stdbool.h>

// Swap is responsible for switching the first index of the unsorted

// part of the array with the lowest/greatest value found towards the

// end of the array (also unsorted). It takes an array to swap data

// for and the indices of the two elements to switch data for.

// It returns nothing.

static void swap (UserData list[], int firstindex, int lastindex);

// getChoice is responsible for finding the lowest/greatest value

// in the unsorted part of the array. It compares all data

// in the unsorted part of the array in order to return the found

// lowest/greatest integer. It takes the list to iterate through,

// over two given indices which determine the unsorted part of the

// array. It compares all data values using a boolean function to

// dictate whether to compare low to high or high to low.

static int getChoice (UserData List[], int startindex, int lastIndex, Comparer ComesFirst);

// SelectionSort will rearrange elements in an array by finding the

// next lowest/greatest element and swaps it with the front element.

// This repeats, each time shrinking the unsorted part of the array.

// It takes an array of UserData integer elements to sort, the array's

// number of elements contained and a boolean function to determine

// how to arrange data, either from low to high or high to low.

// It returns nothing.

void SelectionSort (UserData list[], int ListLength, Comparer ComesFirst)

{

// array's index counter variable

int index;

// start at the first element in the array

// iterating over every element in the array

for (index = 0; index < ListLength-1; index++)

{

// call getChoice to find lowest/greatest value in unsorted

// part of the array and assign it to a placeholder variable.

int indexofchoice = getChoice(list, index, ListLength-1, ComesFirst);

// swap the lowest/greatest value (the placeholder variable)

// with the first index of the unsorted part of the array, essentially

// adding it to the sorted part of the array

swap (list, index, indexofchoice);

}

}

// Swap will replace one element's data with another element's

// data, given the array and two indices to do so. Swapping

// is given unsorted parts of the array and places the lowest/greatest

// into a new sorted part of the array.

void swap (UserData list[], int firstindex, int lastindex)

{

// assign first index of unsorted array value to a placeholder

UserData temp = list[firstindex];

// assign lowest/greatest value to first index of unsorted

// array

list[firstindex] = list[lastindex];

// swap lowest/greatest element that was just assigned

// to first index of unsorted array with the placeholder value

list[lastindex] = temp;

return;

}

// getChoice will determine if two elements are less than or greater than

// one another. If so, it will return true, otherwise return false. It

// requires the array and two indices in order to iterate through the array, plus

// the boolean function to compare based on low to high or high to low.

int getChoice (UserData list[], int startindex, int lastindex, Comparer ComesFirst)

{

// placeholder variable for index of starting

// position

int indexofchoice = startindex;

int loop;

// compare index's value with rightmost remaining indices to determine

// if its value is less/greater than the other

for (loop = startindex+1; loop <= lastindex; loop++)

{

// if the element to its right is less/greater than its left,

// set that lower/greater value to a placeholder variable

// and get ready to return that value

if (ComesFirst(list[loop], list[indexofchoice]) == true) indexofchoice = loop;

}

// return placeholder variable to return lowest/greatest element's index

return indexofchoice;

}

**Program Code (ShellSort.c):**

// the definition of what the UserData typedef consists of

#include "UserData.h"

// declaration of the sort ADT(s) to be called

#include "SortADTs.h"

// we use bool true and false

#include <stdbool.h>

// hsort is responsible for iterating through a given array in order to swap values

// based on the given boolean condition, either low to high or high to low. It is given

// the interval, or spanSize, from ShellSort function in order to properly compare

// two different values an interval apart.

// It returns nothing.

static void hsort(UserData list[], int ListSize, int spanSize, Comparer ComesFirst) ;

// ShellSort is responsible for swapping two elements an interval apart if

// the first is lower/greater than the other. It calls hsort to do the swapping

// and it creates the interval, or span size based on the given array. It also

// requires the number of elements in the array and a boolean function

// to determine if it is to sort values low to high or high to low.

// It returns nothing.

void ShellSort (UserData list[], int ListSize, Comparer ComesFirst)

{

// start by determining the sequence span

// the Knuth method (span = span\*3 + 1 until span > ListSize) will be used

int span\_size = 0;

int prev\_span\_size = 0;

// find the highest span number

while (span\_size < ListSize) {

// keep track of the highest span before we hit ListSize

prev\_span\_size = span\_size;

span\_size = 3 \* span\_size +1;

}

// The initial start span is the highest span for the ListSize

span\_size = prev\_span\_size;

// execute the shell sorting using the span

// after each shell sort, update the span and exit only after

// the span has gone below 1

while (span\_size > 0) {

hsort (list, ListSize, span\_size, ComesFirst);

// This update is the Knuth method that reduces the span number based on the

// previous Knuth number used

span\_size = (span\_size-1) / 3;

}

} //end ShellSort

// hsort will swap values an interval apart given an array, an interval between elements,

// and a boolean function to determine how to sort, i.e. low to high or high to low.

// It determines the element indices an interval apart

void hsort(UserData list[], int ListSize, int spanSize, Comparer ComesFirst)

{

int k;

for (k = spanSize; k < ListSize; k++) {

// determine the second element's index

// an interval apart to compare with the

// first element

int j = k - spanSize;

// set first's value to a placeholder

UserData key = list[k];

// iterate over the array until the interval becomes 1

// this loops as long as the second element exists

// and the boolean condition is met, i.e. first is

// lower/greater than the second

while (j >= 0 && ComesFirst (key, list[j])) {

// swap elements the desired interval apart

list[j + spanSize] = list[j];

j = j - spanSize;

}

// place lowest/greatest value in its final spot

list[j + spanSize] = key;

}

} //end hsort

**Program Code (MergeSort.c)**

// the definition of what the UserData typedef consists of

#include "UserData.h"

// declaration of the sort ADT(s) to be called

#include "SortADTs.h"

// we use bool true and false

#include <stdbool.h>

// we use rand to generate random numbers

#include <stdlib.h>

// we use printf

#include <stdio.h>

// merge will combine two arrays into one, in sorted order of low to high or high to low

// based on the given boolean function. It takes two UserData arrays and low and high values of

// those arrays in order to combine them in sorted order. It returns nothing.

static void merge(UserData A[], int lo, int mid, int hi, UserData T[], Comparer ComesFirst);

// mergeSort will sort the array by splitting it in halves, then recursively splitting those

// halves until the arrays consist of 1 element. They get joined together in sorted order

// until the whole array is sorted. It is given two UserData arrays and low and high values

// of the arrays used in conjunction with the boolean function to determine how to rearrange

// data elements when they are being merged back into a single array. It returns nothing.

static void mergeSort(UserData A[], int lo, int hi, UserData T[], Comparer ComesFirst);

// merge will combine two arrays into one, in sorted order of low to high or high to low

// based on the given boolean function. It takes two UserData arrays and low and high values of

// those arrays in order to combine them in sorted order. It returns nothing.

static void merge(UserData A[], int lo, int mid, int hi, UserData T[], Comparer ComesFirst);

// mergeSort is responsible for sorting an array by splitting the array into halves. Then

// each half is recursively split until the arrays consist of just 1 element. Each array

// is then merged in sorted order, until eventually the full-sized array remains. It takes

// two arrays of UserData integer values, two integer values for comparison and a boolean

// function to compare those two integer values based on a low to high or high to low sequence.

// It returns nothing.

void MergeSort (UserData list[], int ListSize, Comparer ComesFirst)

{

// allocate memory for temporary array to be used while splitting the

// actual array of unsorted data

UserData \*T = (UserData \*) malloc( sizeof (UserData) \* ListSize);

// check if memory has been dynamically allocated

if (T == NULL)

{

// no memory has been allocated, so print an error message

// and exit the program

printf ("Unable to allocate sufficient space for merge copy.. exiting\n");

exit (0);

}

// call mergeSort to sort the array by splitting it and

// merging it together in sorted order based on a boolean

// condition of low to high or high to low

mergeSort (list, 0, ListSize-1, T, ComesFirst);

// release allocated memory of the temporary array

// because the array will have been sorted and its use

// is no longer required in memory

free (T);

}

// mergeSort will sort a given array by splitting it in halves, then recursively splitting those

// halves until the arrays consist of 1 element. They get joined together in sorted order

// until the whole array is sorted. It knows how to insert array elements based on their low and

// high values and the check is done in the boolean function to determine which is less/greater than

// the other.

void mergeSort(UserData A[], int lo, int hi, UserData T[], Comparer ComesFirst)

{

if (lo < hi) //list contains at least 2 elements

{

int mid = (lo + hi) / 2; //get the mid-point subscript

mergeSort(A, lo, mid, T, ComesFirst); //sort first half

mergeSort(A, mid + 1, hi, T, ComesFirst); //sort second half

merge(A, lo, mid, hi, T, ComesFirst); //merge sorted halves

}

} //end mergeSort

// merge will take two UserData arrays and those arrays' low, middle and high values.

// it will combine arrays based on low, middle and high values, assuring the boolean

// condition to arrange based on low to high or high to low. It checks that the low

// and high values are on their respective sides of the middle value so assure

// the array is in sorted order.

void merge(UserData A[], int lo, int mid, int hi, UserData T[], Comparer ComesFirst)

{

// initialize variables for comparing low, middle and high values

int i = lo;

int j = mid + 1;

int k = lo;

// rearrange elements so that low and high values are moved

// to their respective sides, left and right of the middle value

while (i <= mid || j <= hi)

{

if (i > mid) T[k++] = A[j++]; //A[lo..mid] completely processed

else if (j > hi) T[k++] = A[i++]; //A[mid+1..hi] completely processed

else if (ComesFirst(A[i], A[j])) T[k++] = A[i++]; //neither part completed

else T[k++] = A[j++];

}

for (i = lo; i <= hi; i++) A[i] = T[i]; //copy merged elements from T to A

} //end merge

**Program Output (3):**

Enter the maximum number of items to process (1-10) :3

Sort tests follow..

==================

Insertion low to high test follows:

Array initial contents are:-59 76 3

Sorted array contents are: -59 3 76

Insertion high to low test follows:

Array initial contents are:-59 76 3

Sorted array contents are: 76 3 -59

==================

SelectionSort low to high test follows:

Array initial contents are:-59 76 3

Sorted array contents are: -59 3 76

SelectionSort high to low test follows:

Array initial contents are:-59 76 3

Sorted array contents are: 76 3 -59

==================

HeapSort low to high test follows:

Array initial contents are:-59 76 3

Sorted array contents are: -59 3 76

HeapSort high to low test follows:

Array initial contents are:-59 76 3

Sorted array contents are: 76 3 -59

==================

QuickSort low to high test follows:

Array initial contents are:-59 76 3

Sorted array contents are: -59 3 76

QuickSort high to low test follows:

Array initial contents are:-59 76 3

Sorted array contents are: 76 3 -59

==================

MergeSort low to high test follows:

Array initial contents are:-59 76 3

Sorted array contents are: -59 3 76

MergeSort high to low test follows:

Array initial contents are:-59 76 3

Sorted array contents are: 76 3 -59

==================

ShellSort low to high test follows:

Array initial contents are:-59 76 3

Sorted array contents are: -59 3 76

ShellSort high to low test follows:

Array initial contents are:-59 76 3

Sorted array contents are: 76 3 -59

==================

**Program Output (10):**

Enter the maximum number of items to process (1-10) :10

Sort tests follow..

==================

Insertion low to high test follows:

Array initial contents are:-59 76 3 69 -26 -54 -79 -88 -72 43

Sorted array contents are: -88 -79 -72 -59 -54 -26 3 43 69 76

Insertion high to low test follows:

Array initial contents are:-59 76 3 69 -26 -54 -79 -88 -72 43

Sorted array contents are: 76 69 43 3 -26 -54 -59 -72 -79 -88

==================

SelectionSort low to high test follows:

Array initial contents are:-59 76 3 69 -26 -54 -79 -88 -72 43

Sorted array contents are: -88 -79 -72 -59 -54 -26 3 43 69 76

SelectionSort high to low test follows:

Array initial contents are:-59 76 3 69 -26 -54 -79 -88 -72 43

Sorted array contents are: 76 69 43 3 -26 -54 -59 -72 -79 -88

==================

HeapSort low to high test follows:

Array initial contents are:-59 76 3 69 -26 -54 -79 -88 -72 43

Sorted array contents are: -88 -79 -72 -59 -54 -26 3 43 69 76

HeapSort high to low test follows:

Array initial contents are:-59 76 3 69 -26 -54 -79 -88 -72 43

Sorted array contents are: 76 69 43 3 -26 -54 -59 -72 -79 -88

==================

QuickSort low to high test follows:

Array initial contents are:-59 76 3 69 -26 -54 -79 -88 -72 43

Sorted array contents are: -88 -79 -72 -59 -54 -26 3 43 69 76

QuickSort high to low test follows:

Array initial contents are:-59 76 3 69 -26 -54 -79 -88 -72 43

Sorted array contents are: 76 69 43 3 -26 -54 -59 -72 -79 -88

==================

MergeSort low to high test follows:

Array initial contents are:-59 76 3 69 -26 -54 -79 -88 -72 43

Sorted array contents are: -88 -79 -72 -59 -54 -26 3 43 69 76

MergeSort high to low test follows:

Array initial contents are:-59 76 3 69 -26 -54 -79 -88 -72 43

Sorted array contents are: 76 69 43 3 -26 -54 -59 -72 -79 -88

==================

ShellSort low to high test follows:

Array initial contents are:-59 76 3 69 -26 -54 -79 -88 -72 43

Sorted array contents are: -88 -79 -72 -59 -54 -26 3 43 69 76

ShellSort high to low test follows:

Array initial contents are:-59 76 3 69 -26 -54 -79 -88 -72 43

Sorted array contents are: 76 69 43 3 -26 -54 -59 -72 -79 -88

==================

**Problem 4 follows…**  
**Description:** Problem 4 is designed to demonstrate different sorting algorithms with different kinds of data other than integers. These new datatypes include char arrays, characters and floats. It consists of functions and definitions including InsertionSort, SelectionSort, HeapSort, QuickSort, MergeSort and ShellSort. It also includes filling an array with data read from a data file, printing the array data, determining order of elements low to high and high to low and runTest to provide the routine for testing the sort.

**Program Code (SortTest.c):**

// This problem demonstrates how a sort ABT can be called to

// sort the caller's data in whatever order the caller wants

//

// To use a sort ADT, the caller provides the ADT of the underlying

// data and the comparison routine to be used when the sort ADT wants

// to determine a sort order.

// Demonstration of all 6 sorts will be done. Each one will show

// the starting data and the sort results. In this demonstration,

// first names, last names, degrees and GPAs will be sorted.

// we use printf

#include <stdio.h>

// we use rand to generate random numbers

#include <stdlib.h>

// we use bool true and false

#include <stdbool.h>

// we use strcpy from string.h

#include <string.h>

// we use assert from assert.h

#include <assert.h>

// the definition of what the UserData typedef consists of

#include "UserData.h"

// declaration of the sort ADT(s) to be called

#include "SortADTs.h"

/////////////// local function declarations follow ///////////////////

// ProvideStudentData is the routine called to fill the data array with test

// data. It will fill up to a user defined number of UserData items

// with char, char array and float data which is read from a file

// it returns

// the actual number filled in

// list is the array to fill

// MaxListSize is the number of items that will fit in the table

static int ProvideStudentData (UserData list[], int MaxListSize);

// PrintArray will print out a message and the data in the array provided

// msg is the character array containing a message to log

// D is the array with the data to print

// size is the number of items in the array

static void PrintArray (char\* msg, UserData D[], int size);

// LowerFirstName is callable by a sort ADT, returning true if P1.firstName < P2.firstName, otherwise false

static bool LowerFirstName (UserData P1, UserData P2);

// HigherLastName is callable by a sort ADT, returning true if P1.lastName > P2.lastName, otherwise false

static bool HigherLastName (UserData P1, UserData P2);

// LowestDegreeFirst is callable by a sort ADT, returning true if P1.degree < P2.degree, otherwise false

static bool LowestDegreeFirst (UserData P1, UserData P2);

// HighestDegreeFirst is callable by a sort ADT, returning true if P1.degree > P2.degree, otherwise false

static bool HighestDegreeFirst (UserData P1, UserData P2);

// LowestGPAFirst is callable by a sort ADT, returning true if P1.GPA < P2.GPA, otherwise false

static bool LowestGPAFirst (UserData P1, UserData P2);

// HighestGPAFirst is callable by a sort ADT, returning true if P1.GPA > P2.GPA, otherwise false

static bool HighestGPAFirst (UserData P1, UserData P2);

// TheSortFunction typedef gives us a shorthand way of expressing a

// sort function to call. Such a function will need to provide the list to sort,

// the list size and a comparison routine to support the sort

typedef void (\*TheSortFunction) (UserData list[], int ListSize, Comparer ComesFirst);

// Runtest will print out the name of a test and the data, both unsorted and sorted.

// It will call whatever sort routine is provided and pass the comparison routine

// to be used to the sort ADT

// nameofSort is a char array containing the name of the sort being done

// masterArray is an array containing the unsorted data

// scratchArray is an array that will be filled with the unsorted data and

// changed to become sorted by the called sort routine

// arraySize is the number of items to be sorted

// TheSort is the function pointer to the sort function being executed

// SortOrder is the comparison function that the sort routine will call

void RunTest (char\* nameOfSort,

UserData masterArray[],

UserData scratchArray[],

int arraySize,

TheSortFunction TheSort,

Comparer SortOrder );

/////////////// local function definitions follow ///////////////////

// ProvideStudentData is a function the user writes that fills the List

// with data that can be used if the UserData is char, char array and float.

// The user is responsible for filling the list with data, up to the MaxListSize

// that has been allocated by performTest and returning the actual number of items

// filled in.

int ProvideStudentData (UserData list[], int MaxListSize)

{

// initialize variables for reading data from a file

int count;

int i = 0;

int NewListSize;

char fileName[] = "StudentData.txt";

char firstNameTemp[21];

char lastNameTemp[21];

char degreeTemp;

float GPATemp;

// attempt to open the file

FILE \* inputFile = fopen(fileName, "r");

// exit if the file did not open

assert(inputFile != NULL);

// the file opened, so proceed and process its contents

// read records from the file, printing out each record

do {

//try to read in a record

count = fscanf (inputFile, "%s%s %c%f", firstNameTemp, lastNameTemp, &degreeTemp, &GPATemp);

//write data to array if enough records are read

if (count == 4) {

//create data

UserData D;

//set that data's firstName, lastName, degree and GPA, strcpy is used to write to an array

strcpy(D.firstName, firstNameTemp);

strcpy(D.lastName, lastNameTemp);

D.degree = degreeTemp;

D.GPA = GPATemp;

//add data to the array

list[i] = D;

}

//if 4 records are not read, exit the loop

if (count != 4) {

break;

}

//otherwise increment a counter value and continue the do while loop

i++;

} while ((count == 4) && (i < MaxListSize));

//stop reading the file

fclose (inputFile);

// return the number of data items read from the file.

NewListSize = i;

return NewListSize;

}

// PrintArray takes a message to display to the user which describes the

// name of the test to be run. It takes an array of UserData that includes

// char, char arrays and float values

// PrintArray is responsible for displaying each element's

// first and last name, degree and GPA. It is given the array's number

// of elements so that it knows when to stop displaying data and

// it returns nothing.

void PrintArray (char\* msg, UserData D[], int size)

{

printf ("%s", msg);

for (int i = 0; i < size; i++)

printf ("\n\t%s %s %c %.2f", D[i].firstName, D[i].lastName, D[i].degree, D[i].GPA);

printf ("\n");

}

// RunTest is a utility that does the following:

// 1. It copies the master data provided into a scratch array

// 2. It prints out the data to be sorted by calling PrintArray

// 3. It calls the sort routine through the function pointer provided,

// passing the function pointer of the comparison function to be

// used by the sort routine

// 4. It prints out the sorted data

//

// Inputs:

// nameOfSort is a char string that contains the test being done

// masterArray is an array of the test data to use

// scratchArray is an array of the same size that data will be copied into

// and sorted

// arraySize is the number of items in the masterArray to be sorted

// TheSortToRun is a function pointer to the sort routine that will be called

// whenever the sort routine wants to compare one item to another

// SortOrder is a function pointer to the comparison function that will be called

//

void RunTest (char\* nameOfSort,

UserData masterArray[],

UserData scratchArray[],

int arraySize,

TheSortFunction TheSortToRun,

Comparer SortOrder )

{

// copy the master data into the scratch area to be used by the sort

for (int i=0; i < arraySize; i++)

scratchArray[i] = masterArray[i];

// print out the original data

printf ("%s test follows:\n", nameOfSort);

PrintArray((" Array initial contents are:"), scratchArray, arraySize);

// call the requested sort ADT, giving it the function to use for comparisons

TheSortToRun (scratchArray, arraySize, SortOrder);

// print out the sort results

PrintArray(("\n Sorted array contents are: "), scratchArray, arraySize);

return;

}

// LowerFirstName is a function to check that one UserData's

// firstName member is earlier in the alphabet than another UserData's

// firstName. It returns true if the first is sooner in the alphabet

// and false if it is further along in the alphabet.

bool LowerFirstName (UserData P1, UserData P2)

{

//strcmp is used to compare strings

// < 0 and the first comes earlier in the alphabet

// = 0 and they are the same

// > 0 and the first comes further along in the alphabet

return (strcmp(P1.firstName, P2.firstName) < 0);

}

// HigherLastName is a function to check that one UserData's

// firstName member is later in the alphabet than another UserData's

// firstName. It returns true if the first is further along in

// the alphabet and false if it comes sooner in the alphabet.

bool HigherLastName (UserData P1, UserData P2)

{

//strcmp is used to compare strings

// < 0 and the first comes earlier in the alphabet

// = 0 and they are the same

// > 0 and the first comes further along in the alphabet

return (strcmp(P1.lastName, P2.lastName) > 0);

}

// LowestDegreeFirst is a function to check that one UserData's

// degree member is less than another UserData's degree. It returns

// true if the first is less than the second and false if it

// is greater than the second.

bool LowestDegreeFirst (UserData P1, UserData P2)

{

// Degree values in Lowest to Highest Order

char expectedChar[4] = {'?', 'U', 'M', 'D'};

// temp variables to hold indices of P1 and P2 once

// they equal an element in expectedChar[] array

int P1IndexTemp;

int P2IndexTemp;

// find the index that P1 Degree equals

// in expectedChar[] array

for (int i = 0; i < 4; i++) {

if (expectedChar[i] == P1.degree)

P1IndexTemp = i;

}

// find the index that P2 Degree equals

// in expectedChar[] array

for (int j = 0; j < 4; j++) {

if (expectedChar[j] == P2.degree)

P2IndexTemp = j;

}

// compare indices since their values in

// expectedChar[] array are sorted in Lowest

// to Highest Order

if (P1IndexTemp < P2IndexTemp) {

return true;

} else {

return false;

}

}

// HighestDegreeFirst is a function to check that one UserData's

// degree member is greater than another UserData's degree. It returns

// true if the first is greater than the second and false if it

// is less than the second.

bool HighestDegreeFirst (UserData P1, UserData P2)

{

// Degree values in Lowest to Highest Order

char expectedChar[4] = {'?', 'U', 'M', 'D'};

// temp variables to hold indices of P1 and P2 once

// they equal an element in expectedChar[] array

int P1IndexTemp;

int P2IndexTemp;

// find the index that P1 Degree equals

// in expectedChar[] array

for (int i = 0; i < 4; i++) {

if (expectedChar[i] == P1.degree)

P1IndexTemp = i;

}

// find the index that P2 Degree equals

// in expectedChar[] array

for (int j = 0; j < 4; j++) {

if (expectedChar[j] == P2.degree)

P2IndexTemp = j;

}

// compare indices since their values in

// expectedChar[] array are sorted in Lowest

// to Highest Order

if (P1IndexTemp > P2IndexTemp) {

return true;

} else {

return false;

}

}

// LowestGPAFirst is a function to check that one UserData's

// GPA member is less than another UserData's GPA. It returns

// true if the first is less than the second and false if it

// is greater than the second.

bool LowestGPAFirst (UserData P1, UserData P2)

{

return (P1.GPA < P2.GPA);

}

// HighestGPAFirst is a function to check that one UserData's

// GPA member is greater than another UserData's GPA. It returns

// true if the first is greater than the second and false if it

// is less than the second.

bool HighestGPAFirst (UserData P1, UserData P2)

{

return (P1.GPA > P2.GPA);

}

// Here is a test main that demonstrates direct sort calls to all 6 of

// the sorts we are studying for a UserData

//

// To just run a sort, you need:

// - an array of your UserData, filled in with your data

// - a scratch array of the same size that the sort will use

// - its size

// - the sort routine

// - the comparison function to be called by the sort routine

// Just call RunTest to run the test for you!

int main()

{

// allocate space for the test data and a scratch array to be used when sorting

UserData masterArray [10], scratchArray [10];

// get the size of the array to sort

int TestArrayMaxSize;

printf ("Enter the maximum number of items to process (1-10) :");

int result = scanf ("%d", &TestArrayMaxSize);

// abort if we don't have a valid size

if (( result != 1) || (TestArrayMaxSize < 1) || (TestArrayMaxSize > 10))

{

printf ("Bad maximum number of items to process... exiting\n");

exit(1);

}

// fill the masterArray with test data

int numTestItems = ProvideStudentData(masterArray, TestArrayMaxSize);

// run the tests

printf ("Sort tests follow:\n");

printf ("==================\n");

RunTest ("Insertion Lower First Name first", masterArray, scratchArray, numTestItems, InsertionSort, LowerFirstName);

printf ("==================\n");

RunTest ("SelectionSort Higher Last Name first", masterArray, scratchArray, numTestItems, SelectionSort, HigherLastName);

printf ("==================\n");

RunTest ("HeapSort Highest Degree first", masterArray, scratchArray, numTestItems, HeapSort, HighestDegreeFirst);

printf ("==================\n");

RunTest ("QuickSort Lowest Degree first", masterArray, scratchArray, numTestItems, QuickSort, LowestDegreeFirst);

printf ("==================\n");

RunTest ("MergeSort Highest GPA first", masterArray, scratchArray, numTestItems, MergeSort, HighestGPAFirst);

printf ("==================\n");

RunTest ("ShellSort Lowest GPA first", masterArray, scratchArray, numTestItems, ShellSort, LowestGPAFirst);

printf ("==================\n");

return (0);

} //end main

**Data File:**  
Tom Jones U 3.6

Mary Kerry D 3.7

Paula Smith U 3.6

Henry Little M 3.3

Jane Doe M 2.7

Bill Glass D 2.9

Tom K ? 2.0

**Program Output (3):**

Enter the maximum number of items to process (1-10) :3

Sort tests follow:

==================

Insertion Lower First Name first test follows:

Array initial contents are:

Tom Jones U 3.60

Mary Kerry D 3.70

Paula Smith U 3.60

Sorted array contents are:

Mary Kerry D 3.70

Paula Smith U 3.60

Tom Jones U 3.60

==================

SelectionSort Higher Last Name first test follows:

Array initial contents are:

Tom Jones U 3.60

Mary Kerry D 3.70

Paula Smith U 3.60

Sorted array contents are:

Paula Smith U 3.60

Mary Kerry D 3.70

Tom Jones U 3.60

==================

HeapSort Highest Degree first test follows:

Array initial contents are:

Tom Jones U 3.60

Mary Kerry D 3.70

Paula Smith U 3.60

Sorted array contents are:

Mary Kerry D 3.70

Tom Jones U 3.60

Paula Smith U 3.60

==================

QuickSort Lowest Degree first test follows:

Array initial contents are:

Tom Jones U 3.60

Mary Kerry D 3.70

Paula Smith U 3.60

Sorted array contents are:

Paula Smith U 3.60

Tom Jones U 3.60

Mary Kerry D 3.70

==================

MergeSort Highest GPA first test follows:

Array initial contents are:

Tom Jones U 3.60

Mary Kerry D 3.70

Paula Smith U 3.60

Sorted array contents are:

Mary Kerry D 3.70

Paula Smith U 3.60

Tom Jones U 3.60

==================

ShellSort Lowest GPA first test follows:

Array initial contents are:

Tom Jones U 3.60

Mary Kerry D 3.70

Paula Smith U 3.60

Sorted array contents are:

Tom Jones U 3.60

Paula Smith U 3.60

Mary Kerry D 3.70

==================

**Program Output (10):**

Enter the maximum number of items to process (1-10) :10

Sort tests follow:

==================

Insertion Lower First Name first test follows:

Array initial contents are:

Tom Jones U 3.60

Mary Kerry D 3.70

Paula Smith U 3.60

Henry Little M 3.30

Jane Doe M 2.70

Bill Glass D 2.90

Tom K ? 2.00

Sorted array contents are:

Bill Glass D 2.90

Henry Little M 3.30

Jane Doe M 2.70

Mary Kerry D 3.70

Paula Smith U 3.60

Tom Jones U 3.60

Tom K ? 2.00

==================

SelectionSort Higher Last Name first test follows:

Array initial contents are:

Tom Jones U 3.60

Mary Kerry D 3.70

Paula Smith U 3.60

Henry Little M 3.30

Jane Doe M 2.70

Bill Glass D 2.90

Tom K ? 2.00

Sorted array contents are:

Paula Smith U 3.60

Henry Little M 3.30

Mary Kerry D 3.70

Tom K ? 2.00

Tom Jones U 3.60

Bill Glass D 2.90

Jane Doe M 2.70

==================

HeapSort Highest Degree first test follows:

Array initial contents are:

Tom Jones U 3.60

Mary Kerry D 3.70

Paula Smith U 3.60

Henry Little M 3.30

Jane Doe M 2.70

Bill Glass D 2.90

Tom K ? 2.00

Sorted array contents are:

Bill Glass D 2.90

Mary Kerry D 3.70

Henry Little M 3.30

Jane Doe M 2.70

Tom Jones U 3.60

Paula Smith U 3.60

Tom K ? 2.00

==================

QuickSort Lowest Degree first test follows:

Array initial contents are:

Tom Jones U 3.60

Mary Kerry D 3.70

Paula Smith U 3.60

Henry Little M 3.30

Jane Doe M 2.70

Bill Glass D 2.90

Tom K ? 2.00

Sorted array contents are:

Tom K ? 2.00

Paula Smith U 3.60

Tom Jones U 3.60

Jane Doe M 2.70

Henry Little M 3.30

Bill Glass D 2.90

Mary Kerry D 3.70

==================

MergeSort Highest GPA first test follows:

Array initial contents are:

Tom Jones U 3.60

Mary Kerry D 3.70

Paula Smith U 3.60

Henry Little M 3.30

Jane Doe M 2.70

Bill Glass D 2.90

Tom K ? 2.00

Sorted array contents are:

Mary Kerry D 3.70

Paula Smith U 3.60

Tom Jones U 3.60

Henry Little M 3.30

Bill Glass D 2.90

Jane Doe M 2.70

Tom K ? 2.00

==================

ShellSort Lowest GPA first test follows:

Array initial contents are:

Tom Jones U 3.60

Mary Kerry D 3.70

Paula Smith U 3.60

Henry Little M 3.30

Jane Doe M 2.70

Bill Glass D 2.90

Tom K ? 2.00

Sorted array contents are:

Tom K ? 2.00

Jane Doe M 2.70

Bill Glass D 2.90

Henry Little M 3.30

Tom Jones U 3.60

Paula Smith U 3.60

Mary Kerry D 3.70

==================

**Problem 5 (Extra Credit):** Problem 5 is designed to demonstrate my knowledge of Big O notation of each of the 6 sorts. It includes a table identifying which Big O notation describes the sort performances (N, N^2, NlogN).

|  |  |  |
| --- | --- | --- |
| Sort Technique | Observed # of compares in test | # of items sorted |
| **Shell Sort** | 5601426 | 500000 |
| Shell Sort | 455719 | 50000 |
| Shell Sort | 35084 | 5000 |
| Shell Sort | 2457 | 500 |
| Shell Sort | 142 | 50 |
| **Merge Sort** | 4692496 | 500000 |
| Merge Sort | 382512 | 50000 |
| Merge Sort | 29804 | 5000 |
| Merge Sort | 2216 | 500 |
| Merge Sort | 133 | 50 |
| **Heap Sort** | 16951434 | 500000 |
| Heap Sort | 1368937 | 50000 |
| Heap Sort | 103624 | 5000 |
| Heap Sort | 6978 | 500 |
| Heap Sort | 378 | 50 |
| **Quick Sort** | 10384990 | 500000 |
| Quick Sort | 865022 | 50000 |
| Quick Sort | 69606 | 5000 |
| Quick Sort | 5430 | 500 |
| Quick Sort | 364 | 50 |
| **Insertion Sort** | 499999 | 500000 |
| Insertion Sort | 49999 | 50000 |
| Insertion Sort | 4999 | 5000 |
| Insertion Sort | 499 | 500 |
| Insertion Sort | 49 | 50 |
| **Selection Sort** | 1.25E+11 | 500000 |
| Selection Sort | 1249975000 | 50000 |
| Selection Sort | 12497500 | 5000 |
| Selection Sort | 124750 | 500 |
| Selection Sort | 1225 | 50 |

**Explanation:** Big O Notation helps predict running time as the number of items to sort increases. I started filling in the table by comparing the number of compares with the number of items sorted. Insertion Sort is the fastest (least number of compares), while Selection Sort is the slowest (greatest number of compares). Insertion Sort only requires n-1 iterations at its best time. Selection Sort requires n^2 iterations, the most out of any of the 6 sorts. Then I compared the number of compares with the number of items sorted among the Shell, Merge, Heap and Quick Sorts. I expected Merge Sort to be the fastest among these sorts and Heap Sort the slowest. Merge Sort is typically better with large lists than shell sort. Heap Sort is the slowest of Shell, Merge and Quick Sort. Then I was left with Shell and Quick Sort. I decided Shell Sort is faster than Quick Sort. Quick Sort is can result in many compares if the pivot point is the left-most element. The sorted order, from highest to lowest number of compares, for each sort using the highest number of items to sort is below:

1. Selection Sort: 1.25E+11
2. Heap Sort: 16,951,434
3. Quick Sort: 10,384,990
4. Shell Sort: 5,601,426
5. Merge Sort: 4,692,496