Department of Software Engineering
Mehran University of Engineering and Technology, Jamshoro

Course: SWE224 – Data Structure & Algorithms			
Instructor	Mariam Memon	**Practical/Lab No.**	09 & 10
Date		**CLOs**	3
Signature		**Assessment Score**	02

Topic	**Implementation of Sorting Algorithms**
Objectives	Apply merge sort, bubble, quick & insertion sort on a list of items

Lab Discussion: Theoretical concepts and Procedural steps

Sorting

Sorting data means arranging it in a certain order, often in an array-like data structure. You can use various ordering criteria, common ones being sorting numbers from least to greatest or vice-versa, or sorting strings lexicographically.

There are various sorting algorithms, and they're not all equally efficient. We'll be analyzing their time complexity in order to compare them and see which ones perform the best.

We will discuss the following algorithms:

- Bubble Sort
- Insertion Sort
- Merge Sort
- Quicksort

Bubble Sort

Bubble sort works by swapping adjacent elements if they're not in the desired order. This process repeats from the beginning of the array until all elements are in order.

Here are the steps for sorting an array of numbers from least to greatest:

- **4 2** 1 5 3: The first two elements are in the wrong order, so we swap them.
- 2 **4 1** 5 3: The second two elements are in the wrong order too, so we swap.
- 2 1 **4 5** 3: These two are in the right order, 4 < 5, so we leave them alone.

- 2 1 4 **5 3**: Another swap.
- 2 1 4 3 5: Here's the resulting array after one iteration.

Because at least one swap occurred during the first pass (there were actually three), we need to go through the whole array again and repeat the same process.

By repeating this process, until no more swaps are made, we'll have a sorted array.

The reason this algorithm is called Bubble Sort is because the numbers kind of "bubble up" to the "surface." If you go through our example again, following a particular number (4 is a great example), you'll see it slowly moving to the right during the process.

All numbers move to their respective places bit by bit, left to right, like bubbles slowly rising from a body of water.

If you'd like to read a detailed, dedicated article for Bubble Sort, we've got you covered!

Algorithm:

1. Repeat steps 2-3 for i = n-1 down to 1
2. Repeat step 3 for j = 0 up to j = i-1
3. if $a_j > a_i$, swap them

Time Complexity: O(n^2).

Insertion Sort

The idea behind Insertion Sort is dividing the array into the *sorted* and *unsorted* subarrays.

The sorted part is of length 1 at the beginning and is corresponding to the first (left-most) element in the array. We iterate through the array and during each iteration, we expand the sorted portion of the array by one element.

Upon expanding, we place the new element into its proper place within the sorted subarray. We do this by shifting all of the elements to the right until we encounter the first element we don't have to shift.

For example, if in the following array the bolded part is sorted in an ascending order, the following happens:

- **3 5 7 8** 4 2 1 9 6: We take 4 and remember that that's what we need to insert. Since 8 > 4, we shift.

- **3 5 7 x 8** 2 1 9 6: Where the value of x is not of crucial importance, since it will be overwritten immediately (either by 4 if it's its appropriate place or by 7 if we shift). Since 7 > 4, we shift.
- **3 5 x 7 8** 2 1 9 6
- **3 x 5 7 8** 2 1 9 6
- **3 4 5 7 8** 2 1 9 6

After this process, the sorted portion was expanded by one element, we now have five rather than four elements. Each iteration does this and by the end we'll have the whole array sorted.

Algorithm

1. Do steps 2-5 for i=1 up to n-1
2. Hold the element a_i in a temporary space
3. Locate the least index j<=i for $a_j >= a_i$
4. Shift the sub-sequence $\{a_j...a_{i-1}\}$ up one position, into $\{a_{j+1}...a_i\}$
5. copy the held value of a_i to a_j

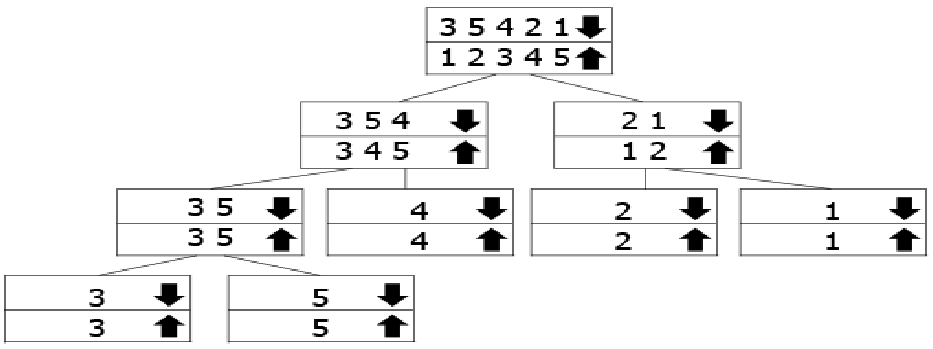
Time Complexity: O(n^2)

Merge Sort

Merge Sort uses recursion to solve the problem of sorting more efficiently than algorithms previously presented, and in particular it uses a divide and conquer approach.

Using both of these concepts, we'll break the whole array down into two subarrays and then:

1. Sort the left half of the array (recursively)
2. Sort the right half of the array (recursively)
3. Merge the solutions



This tree is meant to represent how the recursive calls work. The arrays marked with the down arrow are the ones we call the function for, while we're merging the up arrow ones going back up. So you follow the down arrow to the bottom of the tree, and then go back up and merge.

In our example, we have the array `3 5 3 2 1`, so we divide it into `3 5 4` and `2 1`. To sort them, we further divide them into their components. Once we've reached the bottom, we start merging up and sorting them as we go.

Algorithm

```
1. If the sequence has fewer than two elements, return
2. Let $a_m$ be the middle element of the sequence
3. Sort the sub-sequence of elements that precede $a_m$
4. Sort the sub-sequence of all the other elements
5. Merge the two sorted sub-sequences
```

Time Complexity: O(nlog n).

Quicksort

Quicksort is another Divide and Conquer algorithm. It picks one element of an array as the pivot and sorts all of the other elements around it, for example smaller elements to the left, and larger to the right.

This guarantees that the pivot is in its proper place after the process. Then the algorithm recursively does the same for the left and right portions of the array.

Algorithm

1. If the sequence has fewer than 2 elements, return
2. Using the first element a_p as the pivot, partition the sequence $[a_p,\ldots,a_{q-1}]$ into three sub-sequences {X,Y,Z}, where every element in the sub-sequence X is bounded above by a_p, every element in the sub-sequence Y is bounded below by a_p and Y is a singleton sub-sequence Y = $\{a_p\}$
3. Sort the sub-sequence X
4. Sort the sub-sequence Y

Time Complexity: O(n^2) and average case of O(nlog n)

One of the reasons it is preferred to Merge Sort is that it doesn't take any extra space, all of the sorting is done in-place, and there's no expensive allocation and deallocation calls.

Lab Tasks

Task 1: Implement all the sorting algorithms discussed.

Task 2: Compare computation time for all the algorithms on an array of 100000 items added into the array in descending order.

Task 3: Change your algorithm implementations to sort items in descending order instead of ascending.

Lab Rubrics for Evaluation

Rubrics	**Proficient**	**Adequate**	**Poor**
Accuracy	Completely accurate implementation of algorithms **(1)**	Somewhat accurate implementations **(0.5)**	Incorrect implementations **(0.0)**
Troubleshooting	Easily traced and corrected faults **(1)**	Traced and corrected faults with some guidance **(0.5)**	No troubleshooting skills **(0.0)**