MODULE 2

SORTING ALGORITHMS

Sorting Techniques

- Sorting is a technique to rearrange the elements of a list in ascending or descending order, which can be numerical, lexicographical, or any user-defined order. Sorting is a process through which the data is arranged in ascending or descending order.
- Sorting can be classified in two types; Internal Sorts:- This method uses only the primary memory during sorting process. All data items are held in main memory and no secondary memory is required this sorting process. If all the data that is to be sorted can be accommodated at a time in memory is called internal sorting. There is a limitation for internal sorts; they can only process relatively small lists due to memory constraints.
- There are 3 types of internal sorts
- (i) SELECTION SORT :- Ex:- Selection sort algorithm, Heap Sort algorithm
  (ii)INSERTION SORT :- Ex:- Insertion sort algorithm, Shell Sort algorithm
  ((iii)EXCHANGE SORT :- Ex:- Bubble Sort Algorithm, Quick sort algorithm

External Sorts:- Sorting large amount of data requires external or secondary memory. This process uses external memory such as HDD, to store the data which is not fit into the main memory. So, primary memory holds the currently being sorted data only. All external sorts are based on process of merging. Different parts of data are sorted separately and merged together.

Ex:- Merge Sort

EXCHANGE SORT

- BUBBLE SORT:In bubble sort method the list is divided into two sub-lists sorted and unsorted. The smallest element is bubbled from unsorted sub-list. After moving the smallest element the imaginary wall moves one element ahead. The bubble sort was originally written to bubble up the highest element in the list.
- This method is easy to understand but time consuming. In this type, two successive elements are compared and swapping is done. Thus, step-by-step entire array elements are checked. Given a list of ‘n’ elements the bubble sort requires up to n-1 passes to sort the data.
-

Algorithm for Bubble Sort: Bubble_Sort (A [ ] , N)

Step 1 : Repeat For P = 1 to N – 1 Begin

Step 2 : Repeat For J = 1 to N – P Begin

Step 3 : If (A [J] < A [J – 1])

Swap (A [J] , A [J – 1]) End For

End For

Step 4 : Exit

}

Time Complexity of Bubble Sort :

The complexity of sorting algorithm is depends upon the number of comparisons that are made. Total comparisons in Bubble sort is: $n (n – 1) / 2 \approx n^2 – n$

Best case : $O (n^2)$

Average case : $O (n^2)$

Worst case : $O (n^2)$

How does Bubble Sort Work?

Input: *arr[] = {5, 1, 4, 2, 8}*

First Pass:

- *Bubble sort starts with very first two elements, comparing them to check which one is greater.*
  - *(**5 1** 4 2 8) –> (**1 5** 4 2 8), Here, algorithm compares the first two elements, and swaps since 5 > 1.*
  - *(1 **5 4** 2 8) –> (1 **4 5** 2 8), Swap since 5 > 4*
  - *(1 4 **5 2** 8) –> (1 4 **2 5** 8), Swap since 5 > 2*
  - *(1 4 2 **5 8**) –> (1 4 2 **5 8**), Now, since these elements are already in order (8 > 5), algorithm does not swap them.*

- *Now, during second iteration it should look like this:*
  - *(**1 4** 2 5 8) -> (**1 4** 2 5 8)*
  - *(1 **4 2** 5 8) -> (1 **2 4** 5 8), Swap since 4 > 2*
  - *(1 2 **4 5** 8) -> (1 2 **4 5** 8)*
  - *(1 2 4 **5 8**) -> (1 2 4 **5 8**)*

Third Pass:

- *Now, the array is already sorted, but our algorithm does not know if it is completed.*
- *The algorithm needs one **whole** pass without **any** swap to know it is sorted.*
  - *(**1 2** 4 5 8) -> (**1 2** 4 5 8)*
  - *(1 **2 4** 5 8) -> (1 **2 4** 5 8)*
  - *(1 2 **4 5** 8) -> (1 2 **4 5** 8)*
  - *(1 2 4 **5 8**) -> (1 2 4 **5 8**)*

- QUICK SORT
- It is a faster and highly efficient sorting algorithm. This algorithm follows the divide and conquer approach. Divide and conquer is a technique of breaking down the algorithms into subproblems, then solving the subproblems, and combining the results back together to solve the original problem.
- **Divide:** In Divide, first pick a pivot element. After that, partition or rearrange the array into two sub-arrays such that each element in the left sub-array is less than or equal to the pivot element and each element in the right sub-array is larger than the pivot element.
- **Conquer:** Recursively, sort two subarrays with Quicksort.
- **Combine:** Combine the already sorted array.

Choosing the pivot

Picking a good pivot is necessary for the fast implementation of quicksort. However, it is typical to determine a good pivot. Some of the ways of choosing a pivot are as follows -

- Pivot can be random, i.e. select the random pivot from the given array.
- Pivot can either be the rightmost element of the leftmost element of the given array.
- Select median as the pivot element.
-

QUICK SORT NOTES (PROBLEM + PSEUDOCODE GIVEN IN THE CLASS)

INSERTION SORT

Algorithm

The simple steps of achieving the insertion sort are listed as follows -

Step 1 - If the element is the first element, assume that it is already sorted. Return 1.

Step2 - Pick the next element, and store it separately in a **key.**

Step3 - Now, compare the **key** with all elements in the sorted array.

Step 4 - If the element in the sorted array is smaller than the current element, then move to the next element. Else, shift greater elements in the array towards the right.

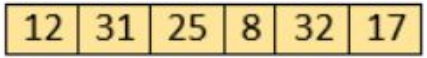
Step 5 - Insert the value.

Step 6 - Repeat until the array is sorted.

PSEUDOCODE

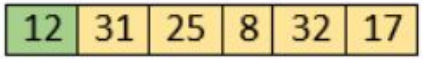
```
INSERTION-SORT(A)
1  for j = 2 to A.length
2      key = A[j]
3      // Insert A[j] into the sorted sequence A[1 .. j − 1].
4      i = j − 1
5      while i > 0 and A[i] > key
6          A[i + 1] = A[i]
7          i = i − 1
8      A[i + 1] = key
```

Let the elements of array are -

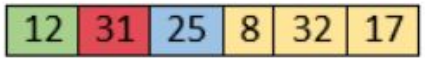


Initially, the first two elements are compared in insertion sort.

Here, 31 is greater than 12. That means both elements are already in ascending order. So, for now, 12 is stored in a sorted sub-array.



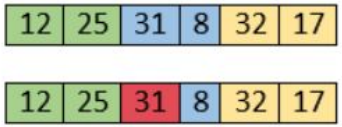
Now, move to the next two elements and compare them.



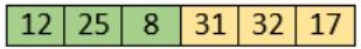
Here, 25 is smaller than 31. So, 31 is not at correct position. Now, swap 31 with 25. Along with swapping, insertion sort will also check it with all elements in the sorted array.

For now, the sorted array has only one element, i.e. 12. So, 25 is greater than 12. Hence, the sorted array remains sorted after swapping.

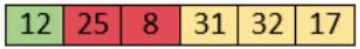
Now, two elements in the sorted array are 12 and 25. Move forward to the next elements that are 31 and 8.



Both 31 and 8 are not sorted. So, swap them.

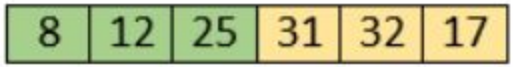


After swapping, elements 25 and 8 are unsorted.



So, swap them.

Now, elements 12 and 8 are unsorted.



Now, the sorted array has three items that are 8, 12 and 25. Move to the next items that are 31 and 32.

Hence, they are already sorted. Now, the sorted array includes 8, 12, 25 and 31.

Move to the next elements that are 32 and 17.

Insertion sort complexity

Now, let's see the time complexity of insertion sort in best case, average case, and in worst case. We will also see th space complexity of insertion sort.

1. Time Complexity

Case	Time Complexity
Best Case	$O(n)$
Average Case	$O(n^2)$
Worst Case	$O(n^2)$

SELECTION SORT

In selection sort, the smallest value among the unsorted elements of the array is selected in every pass and inserted to its appropriate position into the array. It is also the simplest algorithm. It is an in-place comparison sorting algorithm. In this algorithm, the array is divided into two parts, first is sorted part, and another one is the unsorted part. Initially, the sorted part of the array is empty, and unsorted part is the given array. Sorted part is placed at the left, while the unsorted part is placed at the right.

In selection sort, the first smallest element is selected from the unsorted array and placed at the first position. After that second smallest element is selected and placed in the second position. The process continues until the array is entirely sorted.

The average and worst-case complexity of selection sort is **O(n^2)**, where **n** is the number of items. Due to this, it is not suitable for large data sets.

Working of Selection sort Algorithm

Now, let's see the working of the Selection sort Algorithm.

To understand the working of the Selection sort algorithm, let's take an unsorted array. It will be easier to under the Selection sort via an example.

Let the elements of array are -

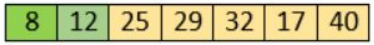
Now, for the first position in the sorted array, the entire array is to be scanned sequentially.

At present, **12** is stored at the first position, after searching the entire array, it is found that **8** is the smallest value.

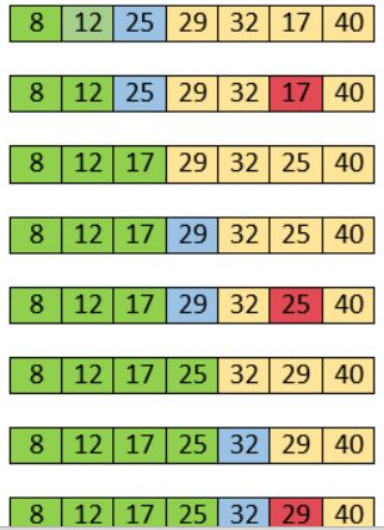
So, swap 12 with 8. After the first iteration, 8 will appear at the first position in the sorted array.

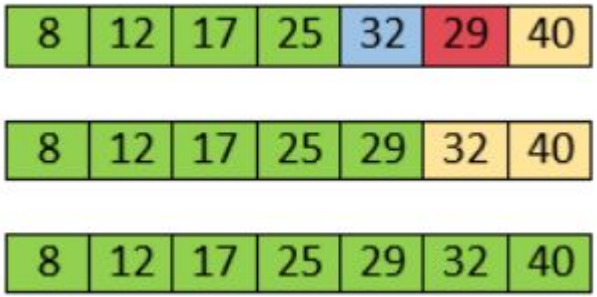
For the second position, where 29 is stored presently, we again sequentially scan the rest of the items of unsorted array. After scanning, we find that 12 is the second lowest element in the array that should be appeared at second position.

Now, swap 29 with 12. After the second iteration, 12 will appear at the second position in the sorted array. So, after two iterations, the two smallest values are placed at the beginning in a sorted way.



The same process is applied to the rest of the array elements. Now, we are showing a pictorial representation of the entire sorting process.





MERGE SORT

- Merge sort is similar to the quick sort algorithm as it uses the divide and conquer approach to sort the elements. It is one of the most popular and efficient sorting algorithm.
- It divides the given list into two equal halves, calls itself for the two halves and then merges the two sorted halves. We have to define the **merge()** function to perform the merging.
- The sub-lists are divided again and again into halves until the list cannot be divided further.
- Then we combine the pair of one element lists into two-element lists, sorting them in the process. The sorted two-element pairs is merged into the four-element lists, and so on until we get the sorted list.

Algorithm

In the following algorithm, **arr** is the given array, **beg** is the starting element, and **end** is the last element of the array.

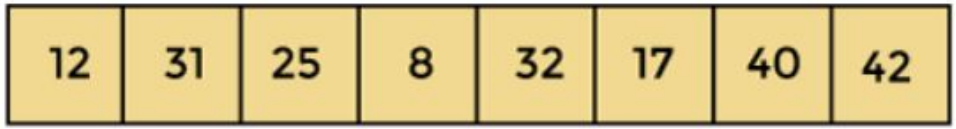
```
MERGE_SORT(arr, beg, end)

if beg < end
set mid = (beg + end)/2
MERGE_SORT(arr, beg, mid)
MERGE_SORT(arr, mid + 1, end)
MERGE (arr, beg, mid, end)
end of if

END MERGE_SORT
```

Working of Merge sort Algorithm

Now, let's see the working of merge sort Algorithm.

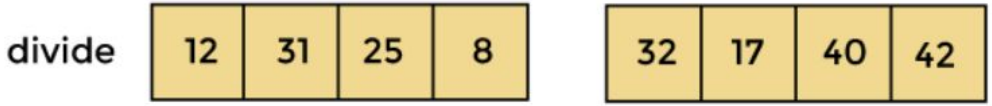
To understand the working of the merge sort algorithm, let's take an unsorted array. It will be easier to understand the merge sort via an example.

Let the elements of array are -

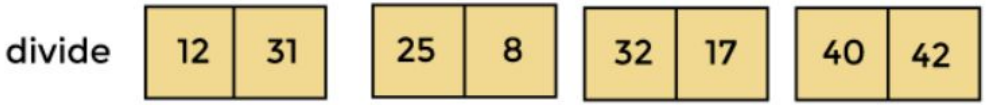


According to the merge sort, first divide the given array into two equal halves. Merge sort keeps dividing the list into equal parts until it cannot be further divided.

As there are eight elements in the given array, so it is divided into two arrays of size 4.



Now, again divide these two arrays into halves. As they are of size 4, so divide them into new arrays of size 2.

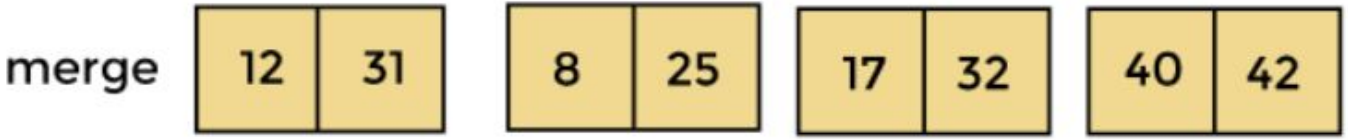


Now, again divide these arrays to get the atomic value that cannot be further divided.

Now, combine them in the same manner they were broken.

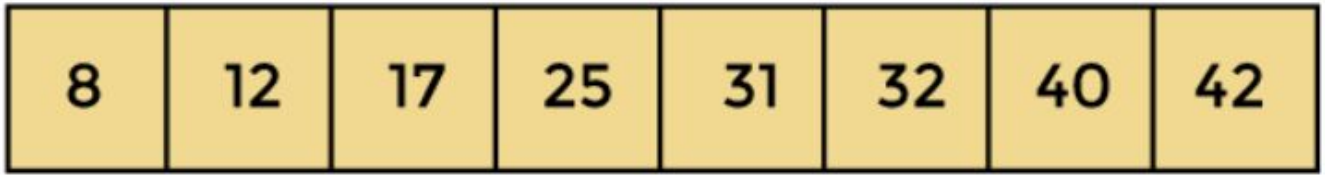
In combining, first compare the element of each array and then combine them into another array in sorted order.

So, first compare 12 and 31, both are in sorted positions. Then compare 25 and 8, and in the list of two values, put 8 first followed by 25. Then compare 32 and 17, sort them and put 17 first followed by 32. After that, compare 40 and 42, and place them sequentially.



In the next iteration of combining, now compare the arrays with two data values and merge them into an array o values in sorted order.

Now, there is a final merging of the arrays. After the final merging of above arrays, the array will look like -



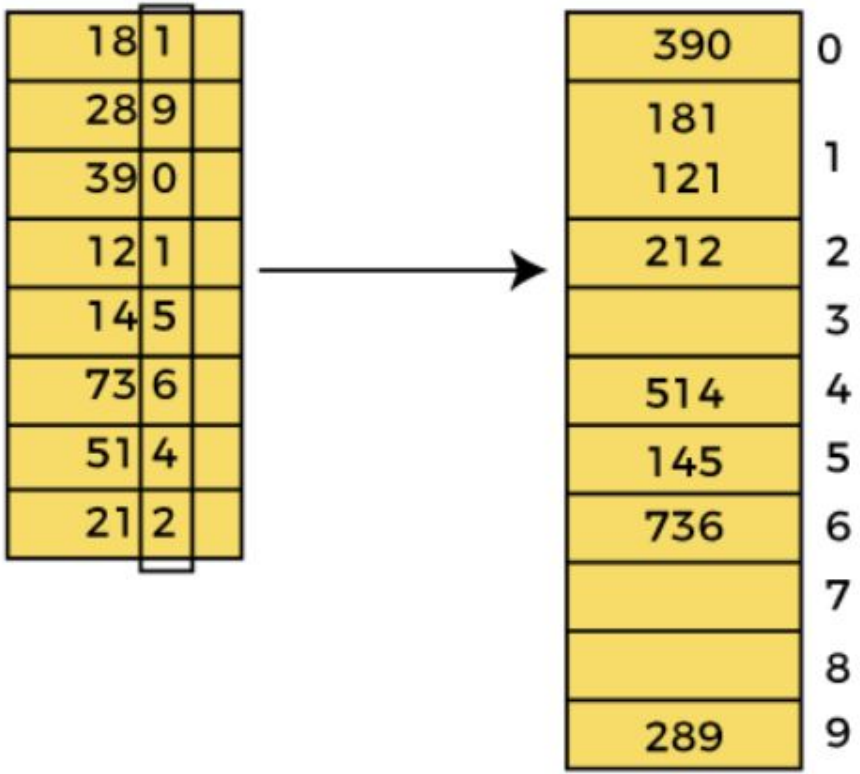
RADIX SORT

- Radix Sort is a linear sorting algorithm.
- Radix Sort's time complexity of O(nd), where n is the size of the array and d is the number of digits in the largest number.
- It is not an in-place sorting algorithm because it requires extra space.
- Radix Sort is a stable sort because it maintains the relative order of elements with equal values.
- Radix sort algorithm may be slower than other sorting algorithms such as merge sort and Quicksort if the operations are inefficient. These operations include sub-inset lists and delete functions, and the process of isolating the desired digits.

- The Radix sort algorithm works by ordering each digit from least significant to most significant.
- In base 10, radix sort would sort by the digits in the one's place, then the ten's place, and so on.
- To sort the values in each digit place, Radix sort employs counting sort as a subroutine.
- This means that for a three-digit number in base 10, counting sort will be used to sort the 1st, 10th, and 100th places, resulting in a completely sorted list. Here's a rundown of the counting sort algorithm.

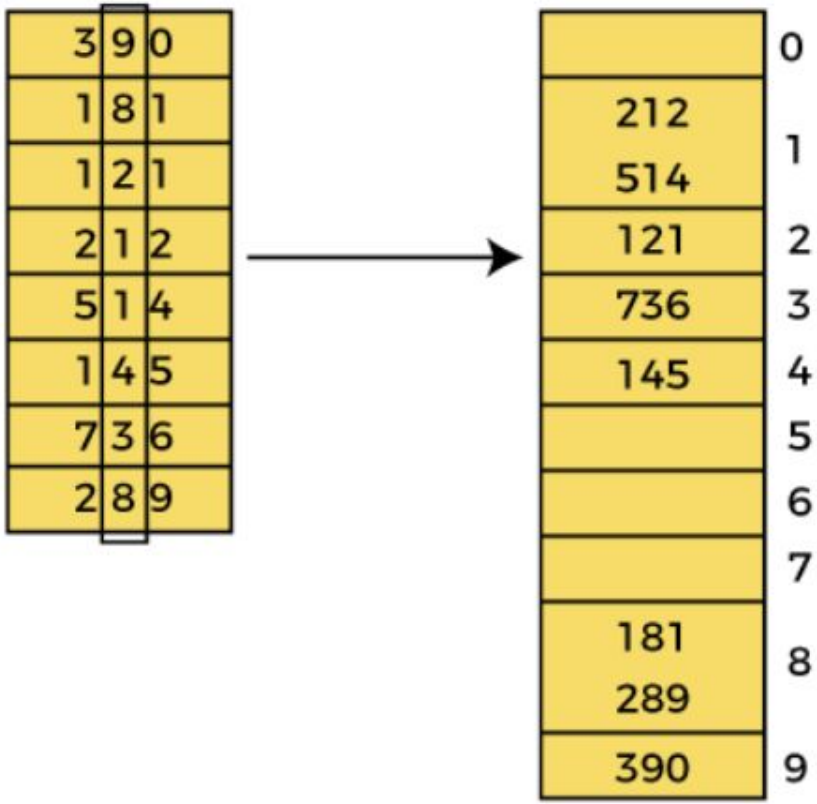
Pass 1:

In the first pass, the list is sorted on the basis of the digits at 0's place.



Pass 2:

In this pass, the list is sorted on the basis of the next significant digits (i.e., digits at 10^{th} place).



Pass 3:

In this pass, the list is sorted on the basis of the next significant digits (i.e., digits at 100^{th} place).