Data Structures Lab 6

Course: Data Structures (CL2001) **Semester:** Fall 2023
Instructor: Sameer Faisal **T.A:** N/A

Note:

- Lab manual cover following below Advance sorting algorithms
  {Quick Sort, Merge Sort, Radix Sort, Linear Searching, Binary Searching, Interpolation Search}
- Maintain discipline during the lab.
- Just raise hand if you have any problem.
- Completing all tasks of each lab is compulsory.
- Get your lab checked at the end of the session.

Quick Sort:

Quick Sort Algorithm is a Divide & Conquer algorithm. It divides input array in two partitions, calls itself for the two partitions (recursively) and performs in-place sorting while doing so. A separate partition () function is used for performing this in-place sorting at every iteration.

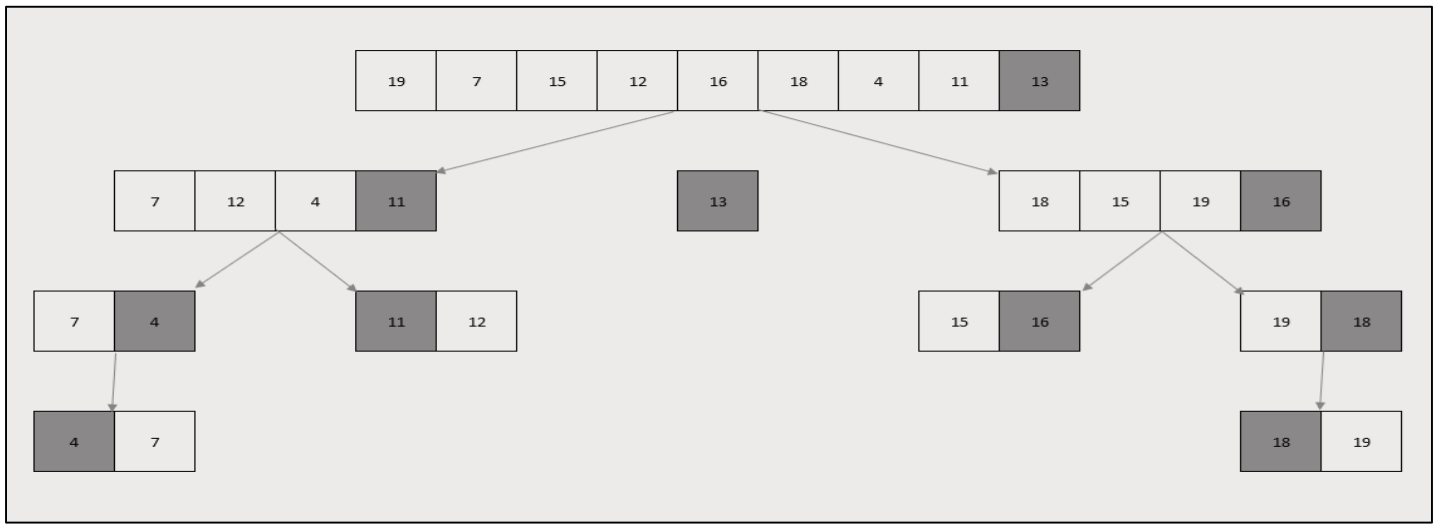
There are 2 Phases in the Quick Sort Algorithm.

Division Phase – Divide the array into 2 halves by finding the pivot point to perform the partition of the array.

1. The in-place sorting happens in this partition process itself.

Recursion Phase –

2. Call Quick Sort on the left partition
3. Call Quick Sort on the right partition.

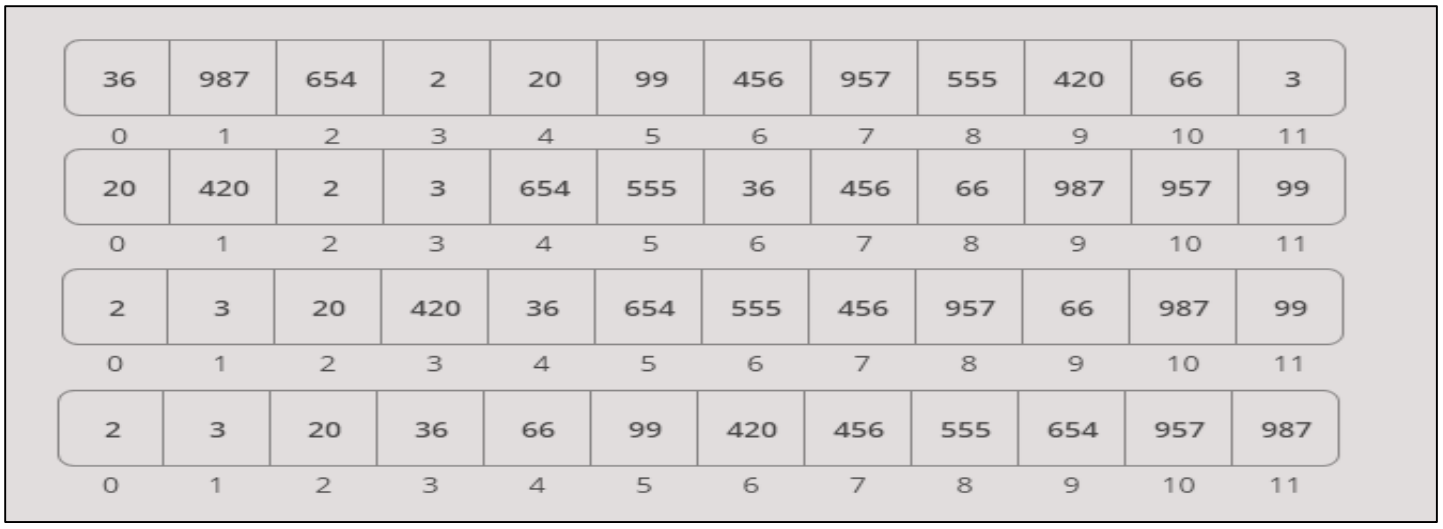


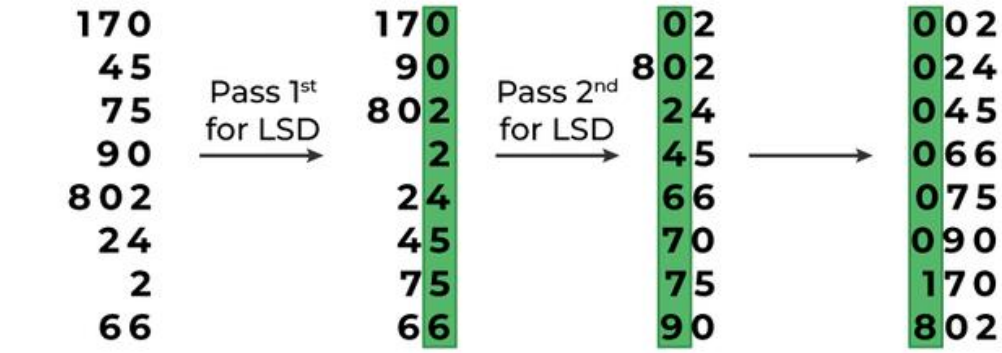
Task 1:

Given the array in the above-mentioned figure implement quick sort that simply chooses the middle element as the pivot and sort accordingly.

Radix Sort (Bucket Sort):
Radix sort is a non-comparative sorting algorithm. It avoids comparison by creating and distributing elements into buckets according to their radix. For elements with more than one significant digit, this bucketing process is repeated for each digit, while preserving the ordering of the prior step, until all digits have been considered. For this reason, radix sort has also been called bucket sort and digital sort.

- https://www.youtube.com/watch?v=XiuSW_mEn7g



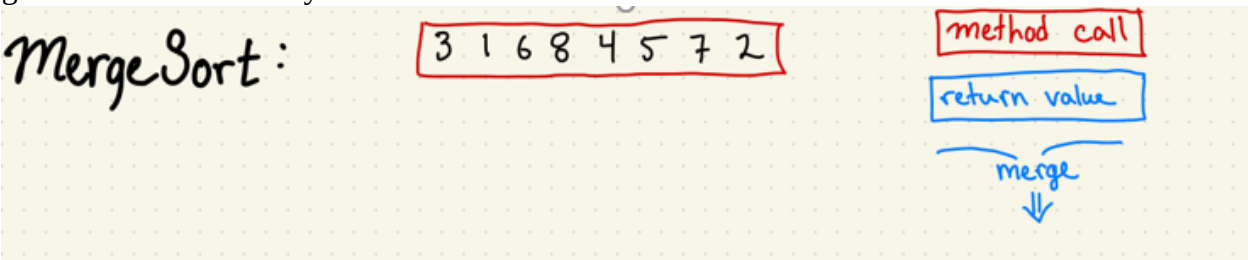


Ans.

2	24	45	66	75	90	170	802

Task 2:
Given an array in the above figure of integers, sort the array in ascending as well as descending order and return it using radix sort

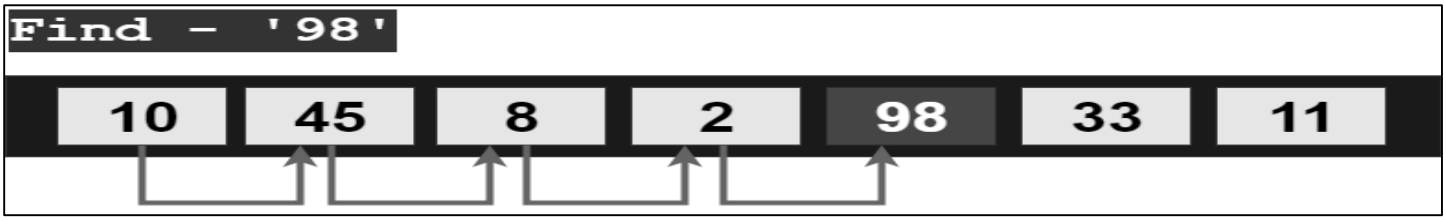
Merge Sort:
Merge sort is a recursive algorithm that continuously splits the array in half until it cannot be further divided i.e., the array has only one element left (an array with one element is always sorted). Then the sorted subarrays are merged into one sorted array.



Task 3

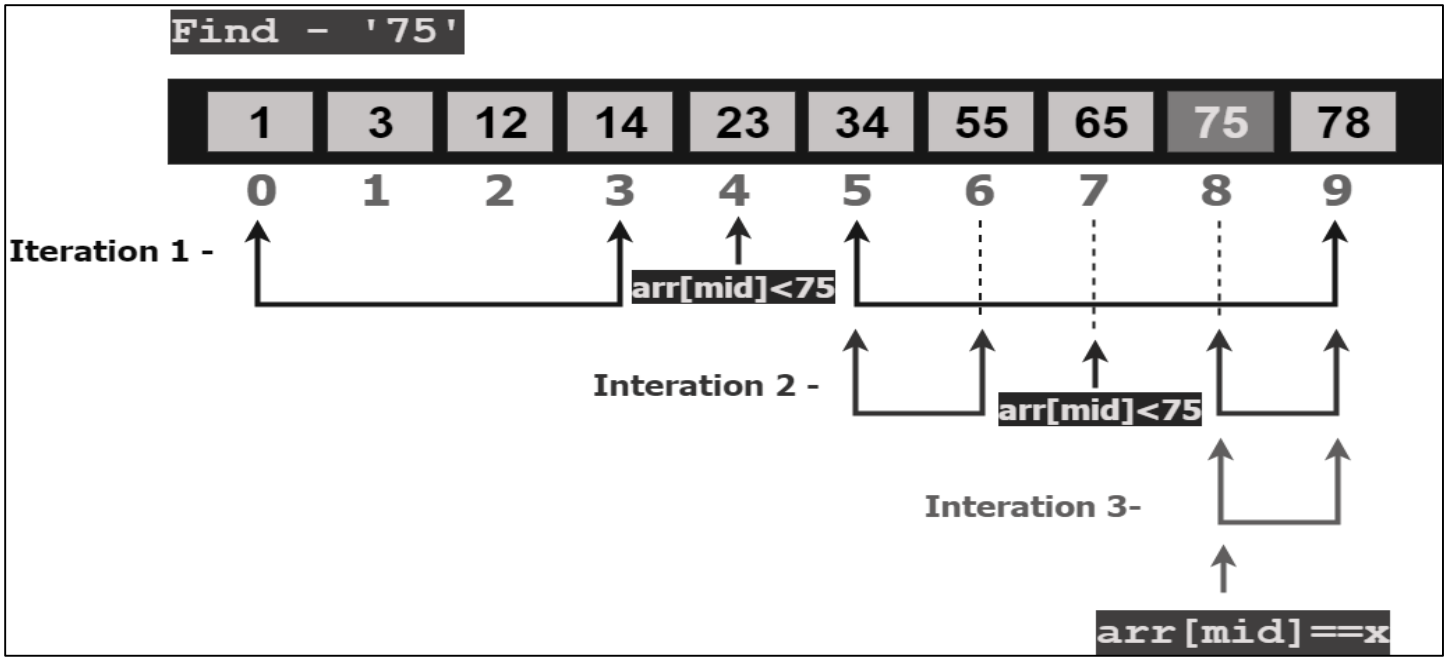
Given an array of integers in the above mentioned figure, sort the array in ascending as well as descending order and return it using Merge sort.

Linear Search
Linear search algorithm or sequential search is a method for finding an element within a list. It sequentially checks each element of the list until a match is found or the whole list has been searched.



Binary Search
Binary search algorithm falls under the category of interval search algorithms. This algorithm is much more efficient compared to linear search algorithm. Binary search only works on sorted data structures. This algorithm repeatedly target the center of the sorted data structure & divide the search space into half till the match is found.
Algorithm:

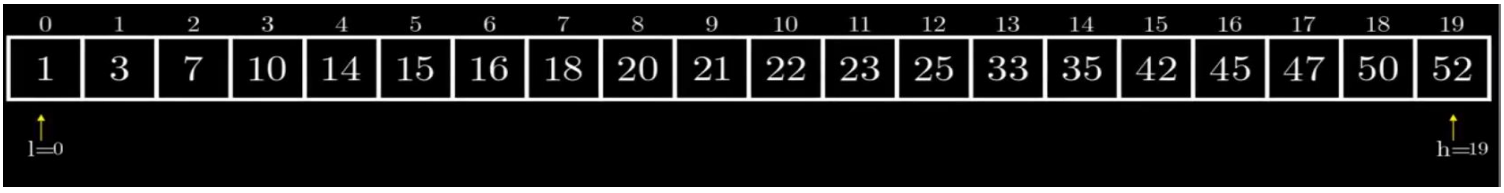
- ***Take input array, left, right & x***
- ***START LOOP – while(left greater than or equal to right)***
  - ***mid = left + (right-left)/2***
  - ***if(arr[mid]==x) then***
    - ***return m***
  - ***else if(arr[mid] less than x) then***
    - ***left = m + 1***
  - ***else***
    - ***right= mid – 1***
- ***END LOOP***
- ***return -1***

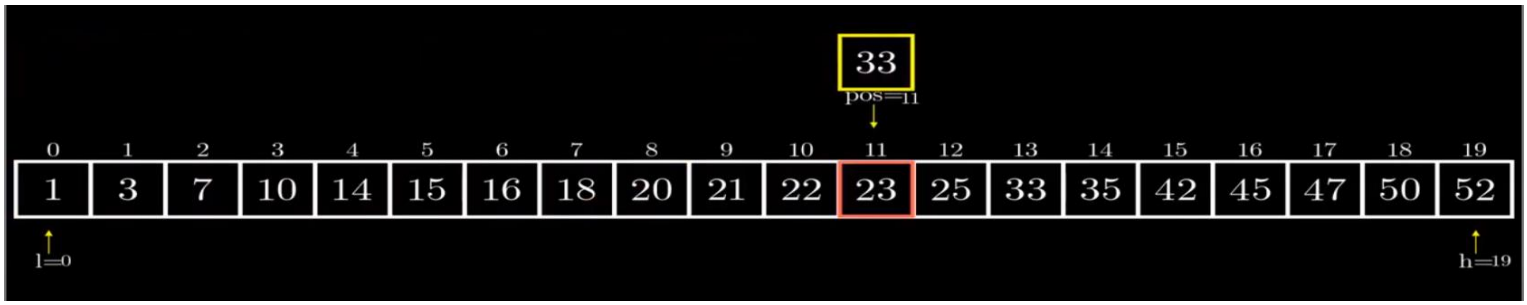


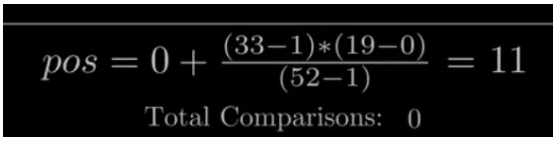
Task 4:
Implement the above array, sort it and find the value corresponding to to your last two digits of the roll number (if its not in the array add a value somewhere in between the array) and find it via binary search.

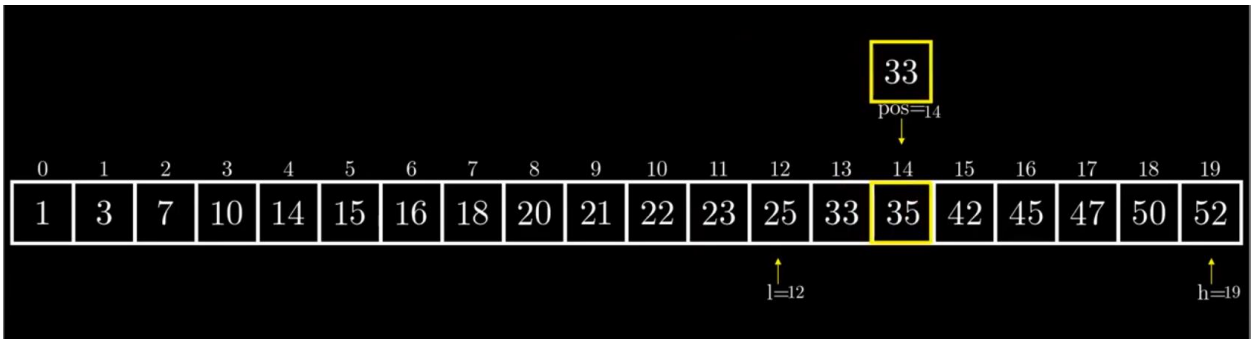
Interpolation Search
Interpolation search is an improvement over binary search. Binary Search always checks the value at middle index. But, interpolation search may check at different locations based on the value of element being searched. For interpolation search to work efficiently the array elements/data should be sorted and uniformly distributed.
Search key =33

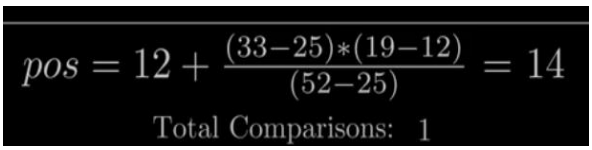
$$Position = startindex + \frac{(element - Arr[startindex]) * (end\ value\ index - start\ value\ index)}{Arr[endindex] - Arr[startindex]}$$

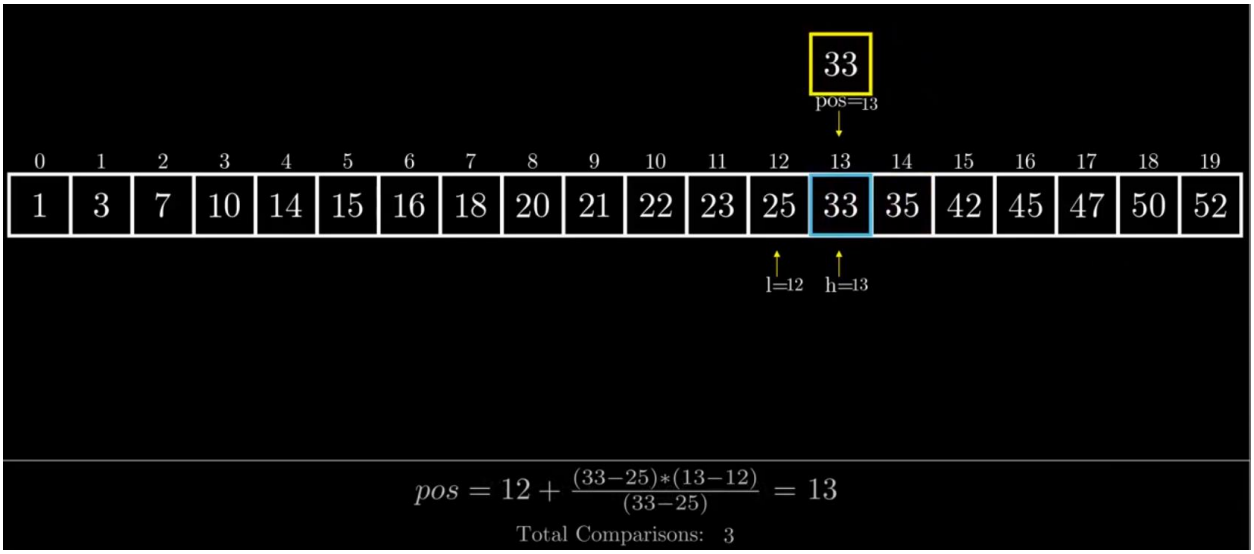












Algorithm

1. **start = 0 & end = n-1**
2. **calculate position to start searching at using formula:**

$$Position = startindex + \frac{(element - Arr[startindex]) * (end\ value\ index - start\ value\ index)}{Arr[endindex] - Arr[startindex]}$$

3. **If A [pos] == Element, element found at index pos.**

4. **Otherwise if element > A[pos] we make start = pos + 1**
5. **Else if element < A[pos] we make end = pos -1**
6. **Do steps 2,3, 4, 5, While : start <= end && element >= A[start] && element =< A[end]**
   - **Start <= end - that is until we have elements in the sub-array.**
   - **Element >= A[start] - element we are looking for is greater than or equal to the starting element of sub-array we are looking in.**
   - **Element =< A[end] - element we are looking for is less than or equal to the last element of sub-array we are looking in.**

Tasks 5:

Write code for interpolation and show the iterations for uniformly distributed array.

Note: An array is considered as uniformly distributed when the difference between the elements is equal or almost same. Example 1: 1,2,3,4,5,6 (Difference is 1)