UNIT 4

- **Searching:**
  - Linear Search
  - Binary Search (Iterative and recursive method)
  - **Interpolation Search**

- **Sorting Sorts:**
  - Selection Sort
  - Bubble Sort
  - Insertion Sort
  - Merge Sort
  - Quick Sort
  - Bucket Sort
  - Radix Sort

UNIT 4

Learning Objective :

- Searching and sorting are two of the most common operations in computer science.
- Searching refers to finding the position of a value in a collection of values. Sorting refers to arranging data in a certain order.
- The two commonly used orders are numerical order and alphabetical order. In this chapter, we will discuss the different techniques of searching and sorting arrays of numbers or characters.

Introduction to SEARCHING

- Searching means to find whether a particular value is present in an array or not.
- If the value is present in the array, then searching is said to be successful and the searching process gives the location of that value in the array.
- However, if the value is not present in the array, the searching process displays an appropriate message and in this case searching is said to be unsuccessful.
- There are two popular methods for searching the array elements: ***linear search*** and ***binary search***.
- The algorithm that should be used depends entirely on how the values are organized in the array. For example, if the elements of the array are arranged in ascending order, then binary search should be used, as it is more efficient for sorted lists in terms of complexity. We will discuss all these methods in detail in this section.

Linear Search : Algorithm

```
LINEAR_SEARCH(A, N, VAL)
Step 1: [INITIALIZE] SET POS = -1
Step 2: [INITIALIZE] SET I = 1
Step 3:     Repeat Step 4 while I<=N
Step 4:             IF A[I] = VAL
                            SET POS = I
                            PRINT POS
                            Go to Step 6
                    [END OF IF]
                     SET I = I + 1
             [END OF LOOP]
Step 5: IF POS = -1
        PRINT "VALUE IS NOT PRESENT
        IN THE ARRAY"
        [END OF IF]
Step 6: EXIT
```

Linear Search

- Linear search, also called as sequential search, is a very simple method used for searching an array for a particular value.
- It works by comparing the value to be searched with every element of the array one by one in a sequence until a match is found.
- Linear search is mostly used to search an unordered list of elements (array in which data elements are not sorted).
- For example, if an array A[] is declared and initialized as,

  ```
  int A[] = {10, 8, 2, 7, 3, 4, 9, 1, 6, 5};
  ```

- and the value to be searched is VAL = 7, then searching means to find whether the value '7' is present in the array or not. If yes, then it returns the position of its occurrence. Here, POS = 3 (index starting from 0).

Complexity of Linear Search Algorithm

- Linear search executes in O(n) time
- where n is the number of elements in the array.
- Obviously, the best case of linear search is when VAL is equal to the first element of the array. In this case, only one comparison will be made.
- Likewise, the worst case will happen when either VAL is not present in the array or it is equal to the last element of the array. In both the cases, n comparisons will have to be made.
- However, the performance of the linear search algorithm can be improved by using a sorted array.

Linear Search : Code

```c
int main() {
    int arr[] = {12, 45, 7, 23, 56, 19, 32};
    int size = sizeof(arr) / sizeof(arr[0]);
    int target = 19;

    int result = linearSearch(arr, size, target);

    if (result != -1) {
        printf("Element found at index %d\n", result);
    } else {
        printf("Element not found in the array\n");
    }

    return 0;
}
```

```c
#include <stdio.h>

int linearSearch(int arr[], int size, int target) {
    for (int i = 0; i < size; i++) {
        if (arr[i] == target) {
            return i; // Return the index where the target is found
        }
    }
    return -1; // Return -1 if the target is not found
}
```

Linear Search: Complexity Analysis

- Linear search executes in O(n) time, where n is the number of elements in the array.
- Obviously, the best case of linear search is when VAL is equal to the first element of the array. In this case, only one comparison will be made.
- Likewise, the worst case will happen when either VAL is not present in the array or it is equal to the last element of the array. In both the cases, n comparisons will have to be made.
- However, the performance of the linear search algorithm can be improved by using a sorted array.

Linear Search: Complexity Analysis

- Worst case time complexity: **O(N)**
- Average case time complexity: **O(N)**
- Best case time complexity: **O(1)**
- Space complexity: **O(1)**

Binary Search (Interval search)

- Search a sorted array by repeatedly dividing the search interval in half.
- Begin with an interval covering the whole array.
- If the value of the search key is less than the item in the middle of the interval, narrow the interval to the lower half.
- Otherwise narrow it to the upper half. Repeatedly check until the value is found or the interval is empty.

Binary Search (Example 1):

Binary Search (Example 2):

- Binary search is a searching algorithm that works efficiently with a sorted list.
- The mechanism of binary search can be better understood by an analogy of a telephone directory.
- When we are searching for a particular name in a directory,
- we first open the directory from the middle and
- then decide whether to look for the name in the first part of the directory
- or in the second part of the directory.
- Again, we open some page in the middle
- and the whole process is repeated until we finally find the right name.

Binary Search (Example 3):

- Take another analogy. How do we find words in a dictionary?
- We first open the dictionary somewhere in the middle.
- Then, we compare the first word on that page with the desired word whose meaning we are looking for.
- If the desired word comes before the word on the page, we look in the first half of the dictionary,
- else we look in the second half. Again, we open a page in the first half of the dictionary and compare the first word on that page with the desired word
- and repeat the same procedure until we finally get the word.
- The same mechanism is applied in the binary search.

Binary Search : Algorithm

```
BINARY_SEARCH(A, lower_bound, upper_bound, VAL)
Step 1: [INITIALIZE] SET BEG = lower_bound
        END = upper_bound, POS = - 1
Step 2: Repeat Steps 3 and 4 while BEG <= END
Step 3:            SET MID = (BEG + END)/2
Step 4:            IF A[MID] = VAL
                         SET POS = MID
                         PRINT POS
                         Go to Step 6
                   ELSE IF A[MID] > VAL
                         SET END = MID - 1
                   ELSE
                         SET BEG = MID + 1
                   [END OF IF]
        [END OF LOOP]
Step 5: IF POS = -1
            PRINT "VALUE IS NOT PRESENT IN THE ARRAY"
        [END OF IF]
Step 6: EXIT
```

Binary Search : Code

```c
#include <stdio.h>

int binarySearch(int arr[], int size, int target) {
    int left = 0;
    int right = size - 1;

    while (left <= right) {
        int mid = left + (right - left) / 2;

        if (arr[mid] == target) {
            return mid; // Return the index where the target is found
        } else if (arr[mid] < target) {
            left = mid + 1;
        } else {
            right = mid - 1;
        }
    }

    return -1; // Return -1 if the target is not found
}
```

```c
int main() {
    int arr[] = {7, 12, 19, 23, 32, 45, 56};
    int size = sizeof(arr) / sizeof(arr[0]);
    int target = 32;

    int result = binarySearch(arr, size, target);

    if (result != -1) {
        printf("Element found at index %d\n", result);
    } else {
        printf("Element not found in the array\n");
    }

    return 0;
}
```

Binary Search: Complexity Analysis

- Worst case time complexity: **O(lon2 (n))**
- Average case time complexity: **O(lon2 (n))**
- Best case time complexity: **O(1)**
- Space complexity: **O(1)**

Binary Search: Calculating Time complexity

At each iteration, the array is divided by half. So let's say the length of array at any iteration is **n**

- At Iteration 1,
  - Length of array = **n**
- At Iteration 2,
  - Length of array = **n/2**
- At Iteration 3,
  - Length of array = **(n/2)/2= n/2^2**
- Therefore, after Iteration k
  - Length of array = **n/2^k**
- Also, we know that after
  - After k divisions, the length of array becomes **1**
  - Therefore, Length of array **= n/2^k = 1 => n = 2^k**
  - Applying log function on both sides: **=> log (n) = log (2^k) => log (n) = k* log (2)**
  - Therefore, **=> k = log (n)**

Interpolation Search

- Interpolation search is another searching algorithm that works efficiently on uniformly distributed datasets.
- It is an improvement over binary search, as it adapts its search position based on the distribution of the data.
- Interpolation search works best on sorted datasets where the values are uniformly distributed.

Interpolation Search : Algorithm

```
INTERPOLATION_SEARCH (A, lower_bound, upper_bound, VAL)

Step 1: [INITIALIZE] SET LOW = lower_bound,
        HIGH = upper_bound, POS = -1
Step 2:    Repeat Steps 3 to 4 while LOW <= HIGH
Step 3:         SET MID = LOW + (HIGH - LOW) ×
                ((VAL - A[LOW]) / (A[HIGH] - A[LOW]))
Step 4:         IF VAL = A[MID]
                   POS = MID
                   PRINT POS
                   Go to Step 6
                 ELSE IF VAL < A[MID]
                    SET HIGH = MID - 1
                 ELSE
                    SET LOW = MID + 1
                [END OF IF]
        [END OF LOOP]
Step 5: IF POS = -1
              PRINT "VALUE IS NOT PRESENT IN THE ARRAY"
        [END OF IF]
Step 6: EXIT
```

Interpolation Search : Complexity

- Complexity of Interpolation Search Algorithm When n elements of a list to be sorted are uniformly distributed (average case), interpolation search makes about **log(log n)** comparisons.
- However, in the worst case, that is when the elements increase exponentially, the algorithm can make up to **O(n)** comparisons.

Introduction to Sorting

- Sorting means arranging the elements of an array so that they are placed in some relevant order which may be either ascending or descending.
- That is, if A is an array, then the elements of A are arranged in a sorted order (ascending order) in such a way that A[0] < A[1] < A[2] < ..... < A[N].
- For example, if we have an array that is declared and initialized as

  int A[] = {21, 34, 11, 9, 1, 0, 22}; Then the sorted array (ascending order) can be given as: A[] = {0, 1, 9, 11, 21, 22, 34 };

Introduction to Sorting

- A sorting algorithm is defined as an algorithm that puts the elements of a list in a certain order, which can be either numerical order, lexicographical order, or any user-defined order.
- Efficient sorting algorithms are widely used to optimize the use of other algorithms like search and merge algorithms which require sorted lists to work correctly.
- There are two types of sorting:
  - ***Internal sorting*** which deals with sorting the data stored in the computer's memory
  - ***External sorting*** which deals with sorting the data stored in files. External sorting is applied when there is voluminous data that cannot be stored in the memory.

Classification of Sorting Algorithms

- Stable vs Unstable Sorting
- Internal vs External Sort
- In-Place vs not-In-Place Sorting
- Comparison based Sorting
- Counting based Sorting

There are many Well known sorting algorithms, such as:

- Selection Sort
- Bubble Sort
- Insertion Sort
- Merge Sort
- Quick Sort
- Bucket Sort
- Radix Sort

Selection Sort

- We find the smallest element from the unsorted sub list and swap it with the element at the beginning of the unsorted data.
- After each selection and swapping, the imaginary wall between the two sub lists move one element ahead, increasing the number of sorted elements and decreasing the number of unsorted ones
- Each time we move one element from the unsorted sub list to the sorted sub list, we say that we have completed a sort pass.
- A list of n elements requires n-1 passes to completely rearrange the data.

Algorithms of Selection Sort

- SELECTION SORT(A[], N)
  - Step 1: Repeat Steps 2 and 3 for i = 1 to N-1
  - Step 2: Call Smallest (A[], i, N, pos)
  - Step 3: SwapA[i] with A[pos]

  [End of Loop]
  - Step 4: Exit

- Smallest (A[], i, N, POS)
  - Step 1: [Initialize] set Small= A[i]
  - Step 2: [Initialize] Set pos = i
  - Step 3: Repeat for J = i+1 to N -1

  if small > A[J]

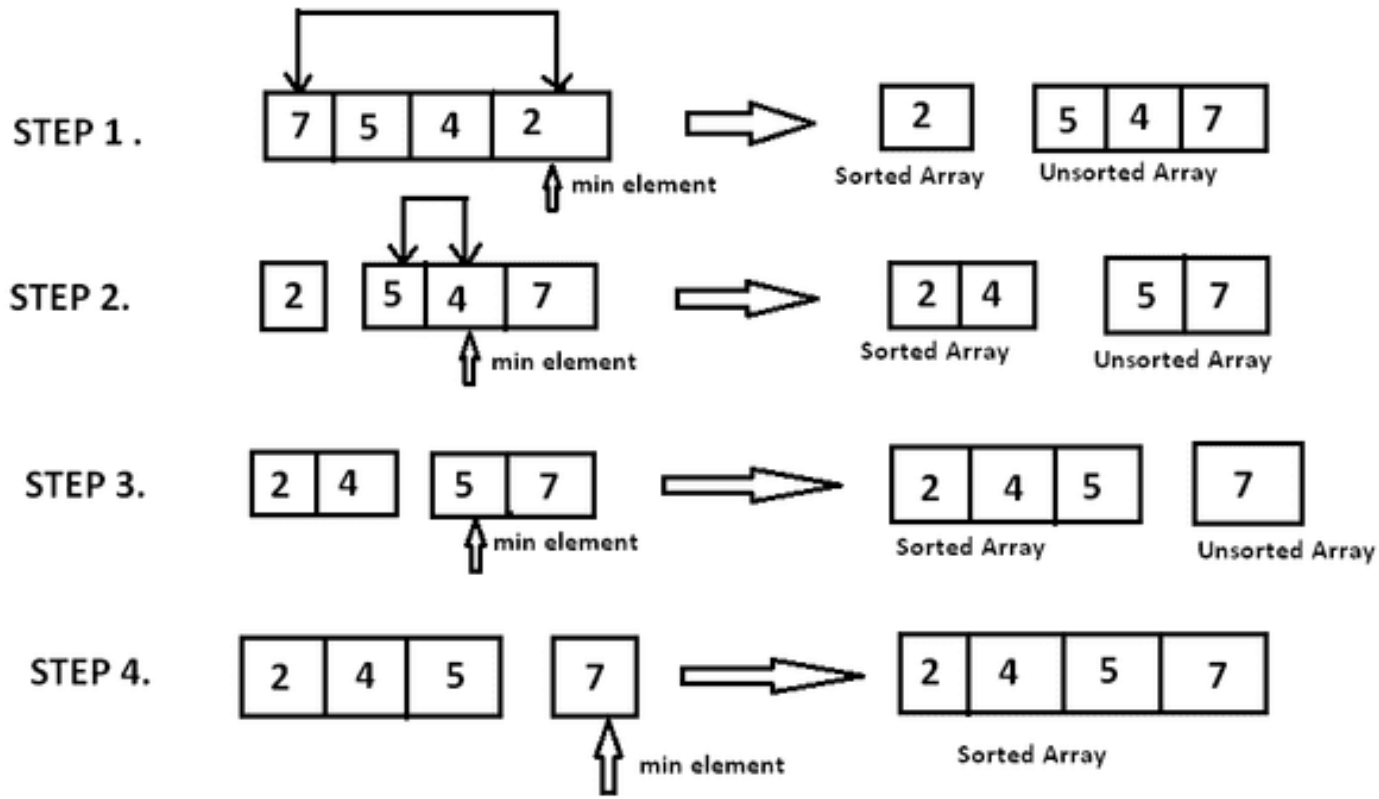
  Set small = A[J]

  Set pos = J

  [End of if]

  [End of Loop]
  - Step 4: Return pos

Working of Selection Sort :



Analysis of Selection Sort

- Best-case: O(n^2)
  - ✓ Array is already sorted in ascending order
- Worst-case: O(n^2)
  - ✓ Worst case occurs when array is reverse sorted.
- Average-case: O(n^2)
  - ✓ We have to look at all possible initial data organizations

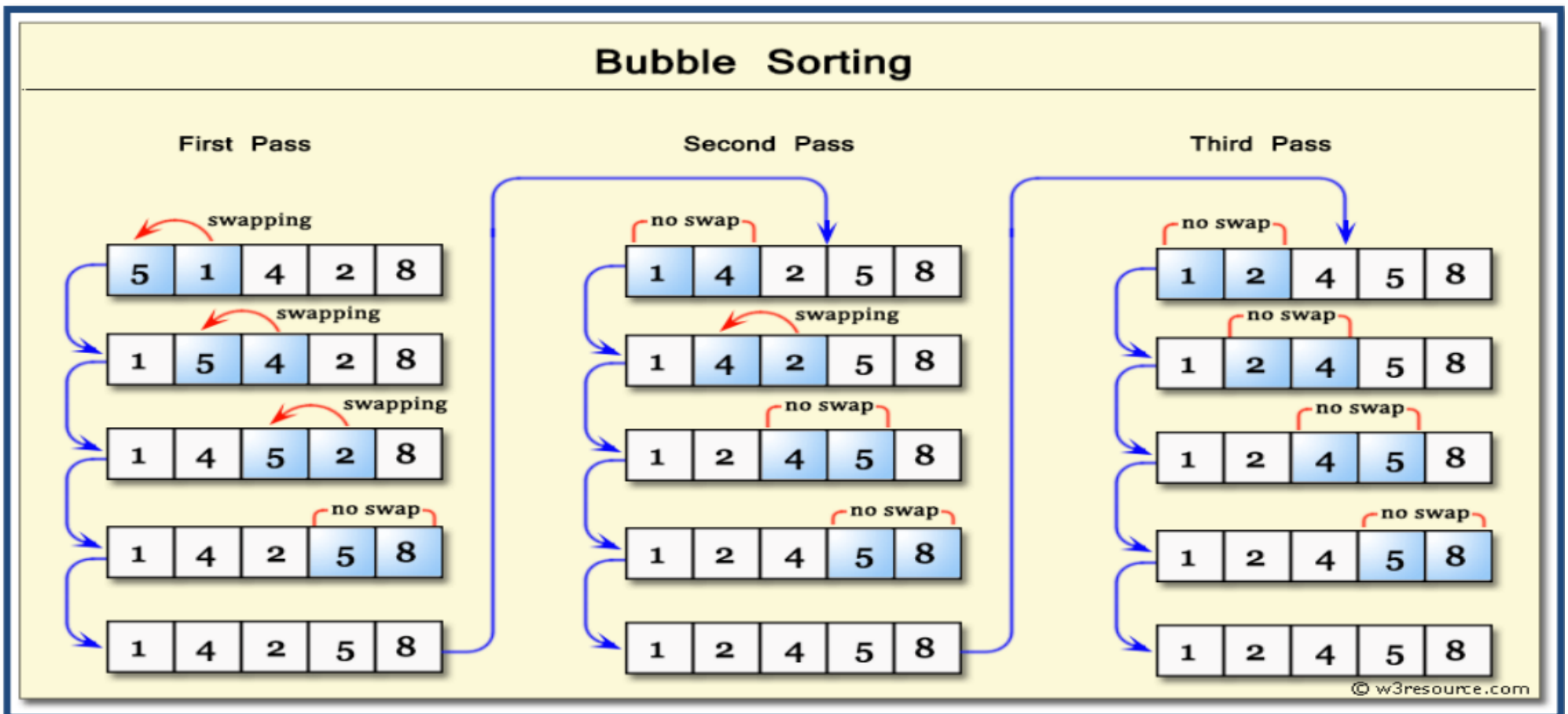
Bubble Sort

- The smallest element is bubbled from the unsorted list and moved to the sorted sub list.
- After that, the wall moves one element ahead, increasing the number of sorted elements and decreasing the number of unsorted ones.
- Each time an element moves from the unsorted part to the sorted part one sort pass is completed.
- Given a list of n elements, bubble sort requires up to n-1 passes to sort the data

Algorithms of Bubble Sort

- Step 1: Repeat Step 2 For i = 0 to N-1
- Step 2: Repeat For J = i + 1 to N - I
- Step 3: IF A[J] > A[i]
  - Swap A[J] and A[i]
  - [End of inner loop]
  - [end of outer loop]
- Step 4: Exit

Working of Bubble Sort :

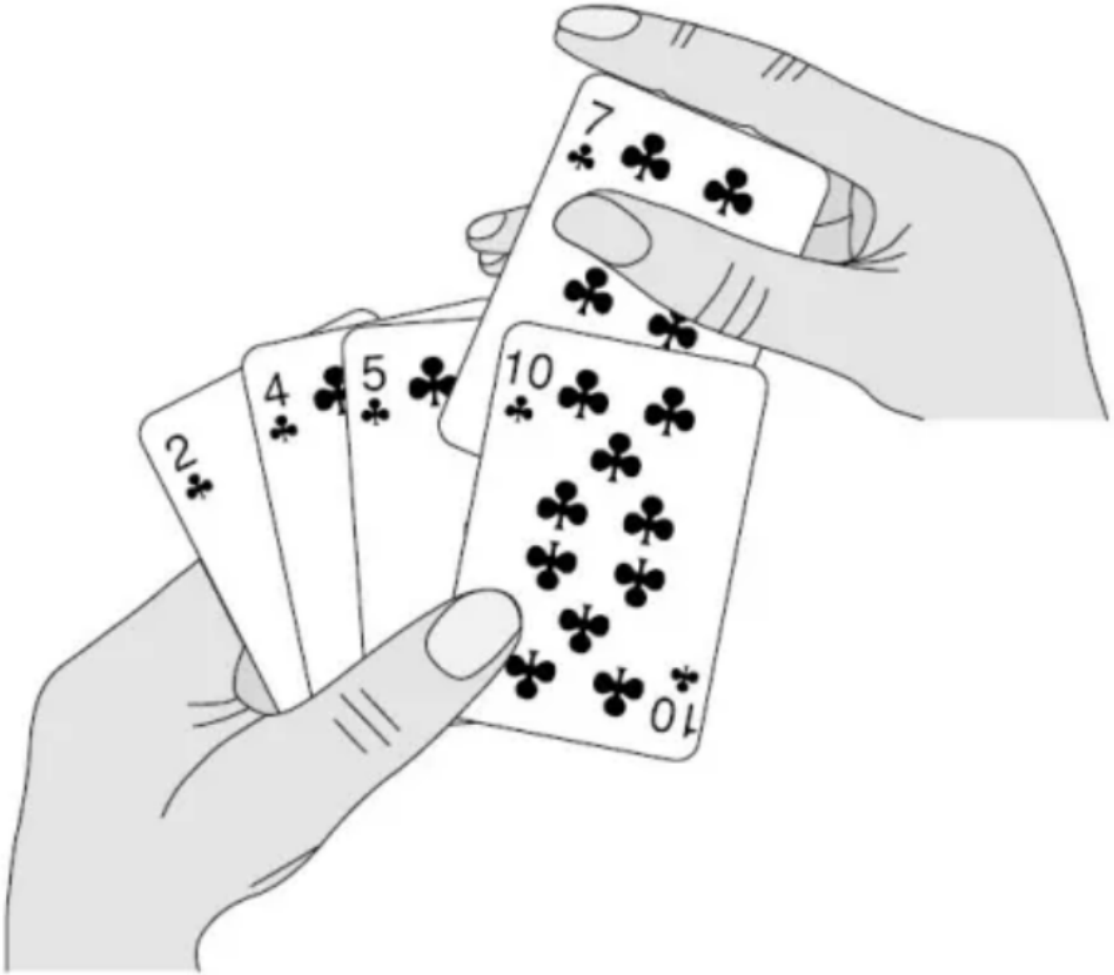


Analysis of Bubble Sort

- Best-case: O(n)
  - ✓ Array is already sorted in ascending order
- Worst-case: O(n^2)
  - ✓ Worst case occurs when array is reverse sorted.
- Average-case: O(n^2)
  - ✓ We have to look at all possible initial data organizations

Insertion Sort

- Insertion sort is the simple sorting algorithm which is commonly used in the daily lives while ordering a deck of cards.
- In each pass, the first element of the unsorted part is picked up, transferred to the sorted sub list, and inserted at the appropriate place.
- A list of n elements will take at most n-1 passes to sort the data.


7
2
4
5
10
10

Algorithms of Insertion Sort

- Step 1: Repeat Steps 2 to 5 for i = 1 to N-1
- Step 2: Set Temp = A[i]
- Step 3: Set J = i - 1
- Step 4: Repeat while Temp <=A[J]

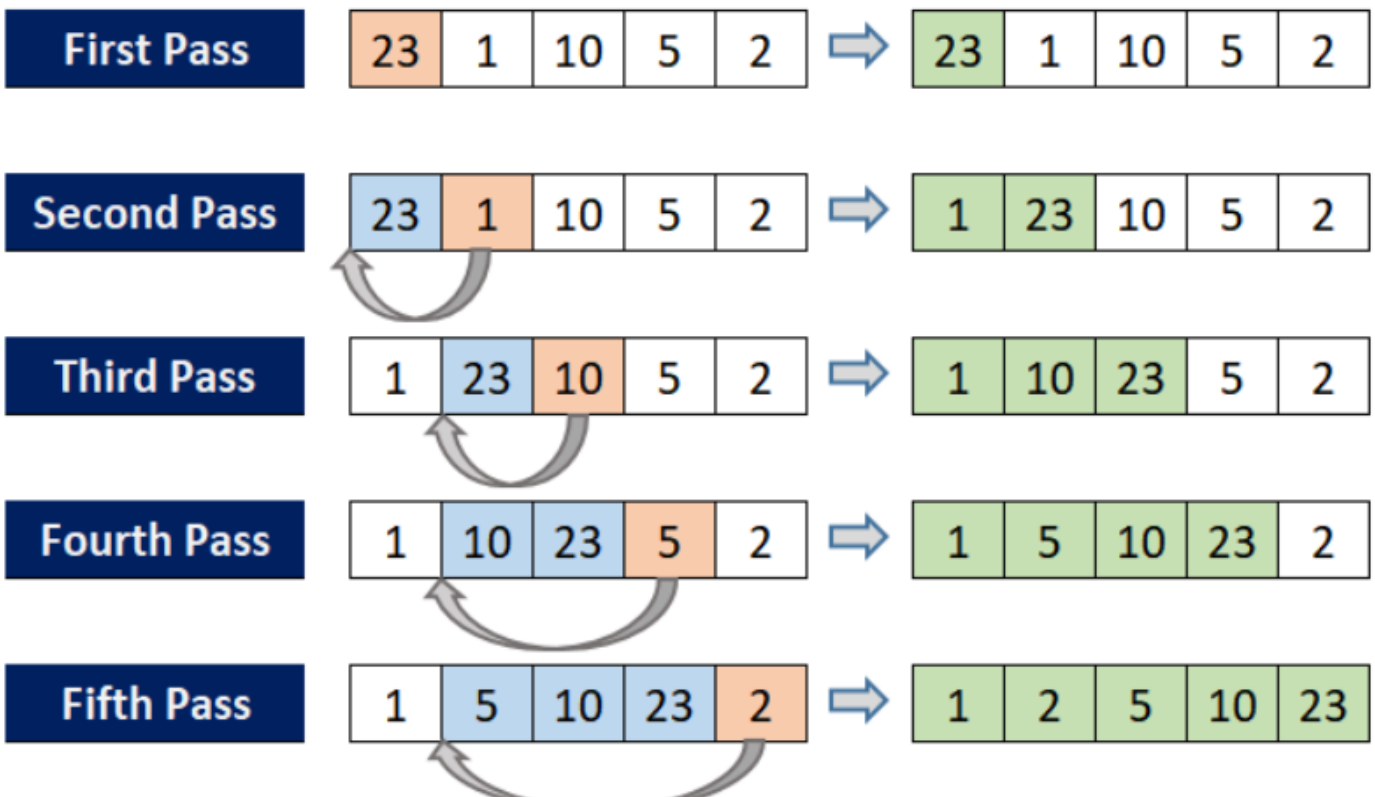
  Set A[J + 1] = A[J]

  Set J = J - 1

  [End of inner Loop]
- Step 5: Set A[J + 1] = Temp

  [End of loop]
- ✓Step 6: Exit

Working of Insertion Sort :



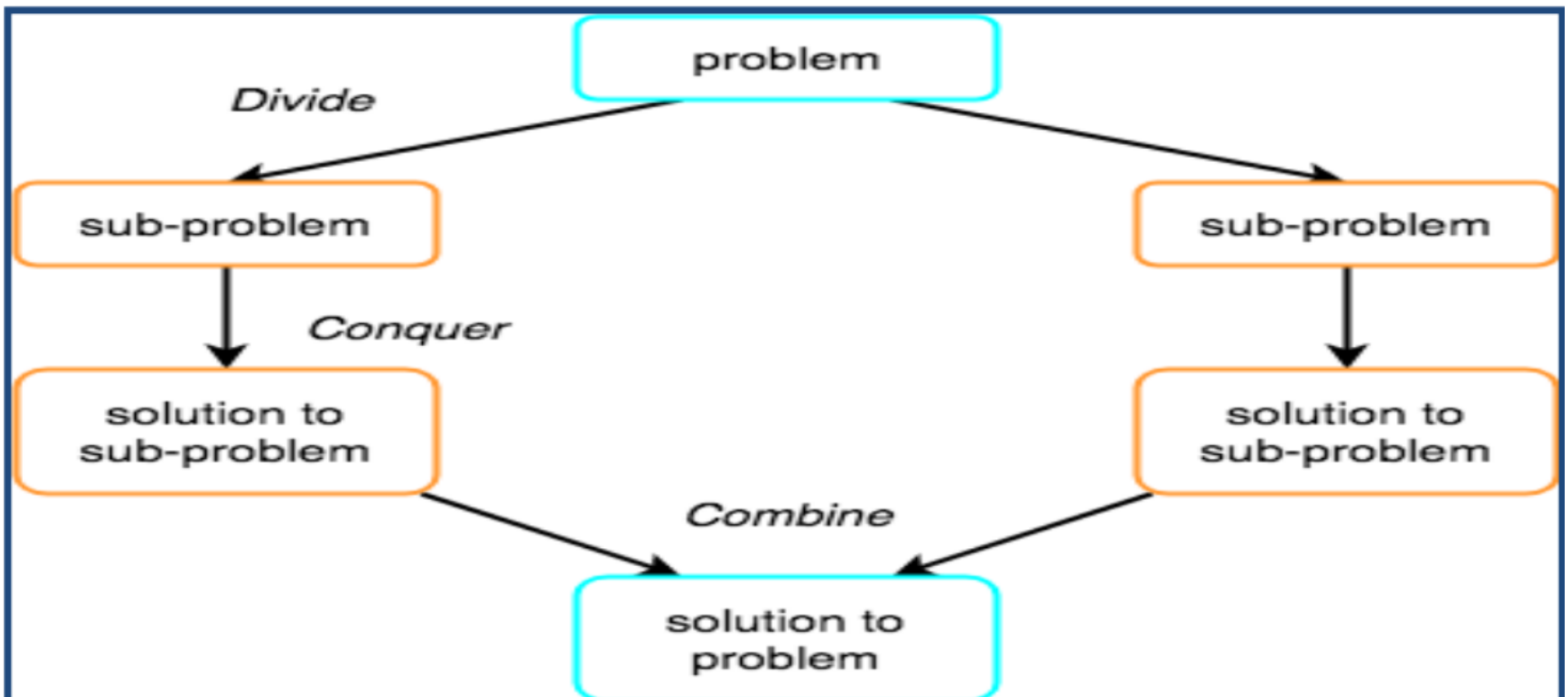
Analysis of Insertion Sort

- Best-case: O(n)
  - ✓ Array is already sorted in ascending order
- Worst-case: O(n^2)
  - ✓ Worst case occurs when array is reverse sorted.
- Average-case: O(n^2)
  - ✓ We have to look at all possible initial data organizations

Merge Sort

- Merge sort is a sorting technique based on divide and conquer technique.
- Before discussing merge sort, we need to understand what is the meaning of splitting and merging
- What do you mean by Divide and Conquer Technique? .

Divide and Conquer Technique



Algorithms of Merge Sort

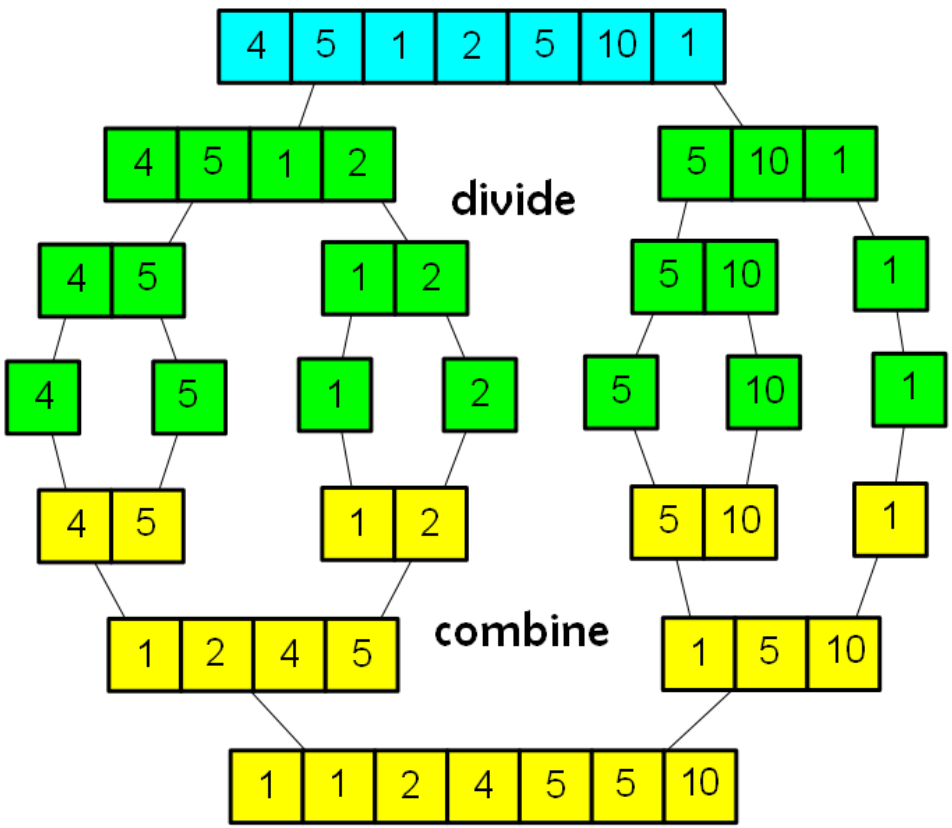
- Step 1: [check for recursive call]
  - if b<e then mid=(b+e)/2
  - mergesort(a, b, mid)
  - mergesort(a,mid+1,e)
  - **merge(a,b,mid,e)**
  - end if
- Step 2: [Finish]

Merging Procedure Algorithms

This algorithm merge two sorted array of size n/2 in to one sorted array a of size n.

✓Merge Algorithms
✓Step 1:[Initialization]
✓i=b, temp[100], i1=b, e1=mid, i2=mid+1,e2=e;
✓Step 2:[Merge until all elements of one array got merged]
✓Repeat step 3 while (i1<= e1 &&i2<=e2)
✓ Step 3:[Merging of sub arrays]
✓ if a[i1]< a[i2] then temp[i++]=a[i1] i1++
✓ else
✓ temp[i++]=a[i2]
✓ i2++
✓ end if
✓ Step 4:[Copy all elements from first sub array]
✓ repeat step 5 while(i1<=e1)

Working of Merge Sort :



Analysis of Merge Sort

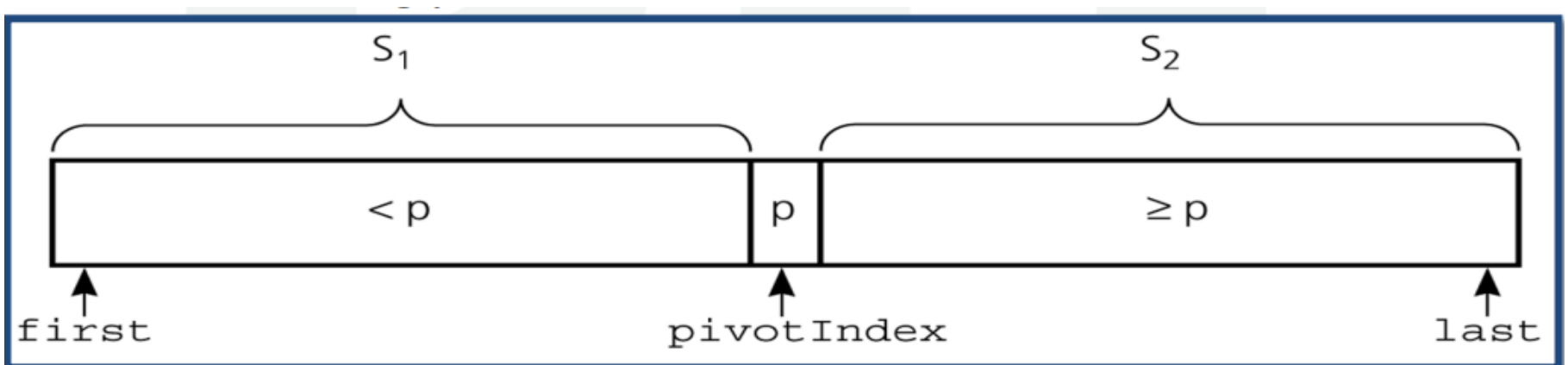
- Best-case: O(nlogn)
  - ✓ Array is already sorted in ascending order
- Worst-case: O(nlogn)
  - ✓ Worst case occurs when array is reverse sorted.
- Average-case: O(nlogn)
  - ✓ We have to look at all possible initial data organizations

Quick Sort

- Like merge sort, Quicksort is also based on the divide and conquer paradigm.
- But it uses this technique in a somewhat opposite manner, as all the hard work is done before the recursive calls.
- The quick-sort algorithm consists of the following three steps
  - **Divide**: Partition the list.
  - **Recursion**: Recursively sort the sub lists separately.
  - **Conquer**: Put the sorted sub lists together

Partition Procedure

- Partition means places the pivot in its correct place position within the array.
- Arranging the array elements around the pivot p generates two smaller sorting problems



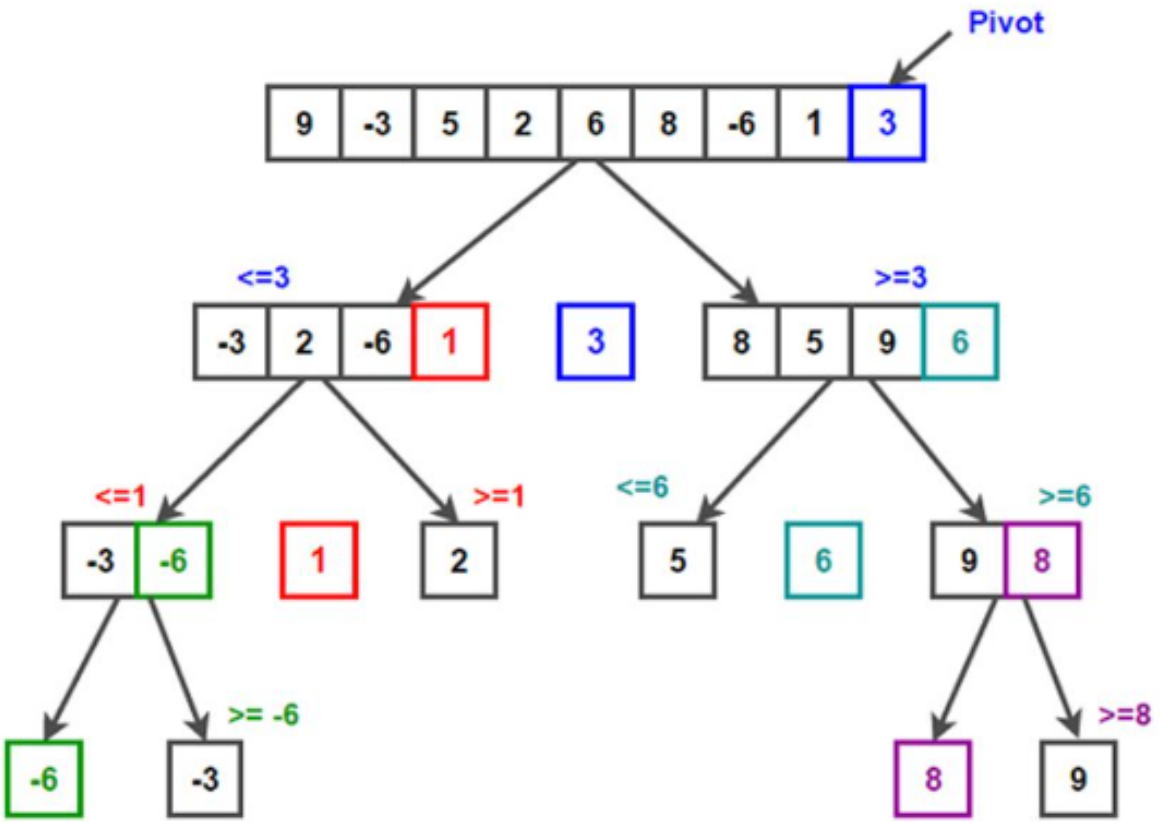
Partition Algorithms

- Choose a pivot element from the array.
- Initialize two pointers, left and right, pointing to the first and last elements of the array respectively.
- Repeat the following steps until left is less than or equal to right:
  - Move the left pointer to the right until an element greater than or equal to the pivot is found.
  - Move the right pointer to the left until an element less than or equal to the pivot is found.
  - If left is still less than or equal to right, swap the elements at positions left and right.
- Swap the pivot element with the element at the right pointer. This places the pivot element in its final sorted position.
- Return the index of the pivot element.

Quick Sort Algorithms

- Choose a pivot element from the array. This can be any element, but often the middle element is chosen.
- Partition the array into two sub-arrays:
  - Elements less than the pivot in one sub-array.
  - Elements greater than the pivot in another sub-array.
- Recursively apply the Quick Sort algorithm to the sub-arrays generated in the previous step.
- Combine the sorted sub-arrays and the pivot in the order: left sub-array, pivot, right sub-array.

Working of Quick Sort :



Analysis of Quick Sort

- Best-case: O(nlogn)
  - ✓ Array is already sorted in ascending order
- Worst-case: O(nlogn)
  - ✓ Worst case occurs when array is reverse sorted.
- Average-case: O(nlogn)
  - ✓ We have to look at all possible initial data organizations

Bucket Sort

- Bucket Sort is a sorting algorithm that works by distributing the elements of an array into a number of buckets and
- then sorting the elements within each bucket individually using another sorting algorithm (often insertion sort).
- After sorting each bucket, the sorted elements are concatenated to obtain the final sorted array.

Bucket Sort algorithm works:

- Determine the range of values in the input array and divide it into a suitable number of equally-sized buckets.
- Traverse through the input array and place each element into the appropriate bucket based on its value. The mapping from elements to buckets can be done using a simple formula:
  - **bucket_index = (value - min_value) / bucket_size**
  - where value is the element's value, min_value is the minimum value in the array, and bucket_size is the size of each bucket.
- Sort each bucket individually using a sorting algorithm. Commonly used algorithms are **Insertion Sort** or another sorting algorithm with good performance for small arrays.
- Concatenate the sorted buckets in order to obtain the final sorted array.

Bucket Sort algorithm (Example):

- Consider an array **arr[] = {22, 72, 62, 32, 82, 142}**

- **Range = (maximum-minimum) / number of elements**
- So, here the range will be given as: **Range = (142 – 22)/6 = 20**

- Thus, the range of each bucket in bucket sort will be: **20** So, the buckets will be as:
- **20-40; 40-60; 60-80; 80-100; 100-120; 120-140; 140-160**

Bucket Sort algorithm (Example):

- ***Bucket index = floor((a[i]-min)/range)***

  For 22, bucketindex = (22-22)/20 = 0

  For 72, bucketindex = (72-22)/20 = 2.5

  For 62, bucketindex = (62-22)/20 = 2

  For 32, bucketindex = (32-22)/20 = 0.5

  For 82, bucketindex = (82-22)/20 = 3

  For 142, bucketindex = (142-22)/20 = 6

Bucket Sort algorithm (Example):

- ***Elements can be inserted into the bucket as:***

  0 -> 22 -> 32

  1

  2 -> 72 -> 62 (72 will be inserted before 62 as it appears first in the list).

  3 -> 82

  4

  5

  6 -> 142

Bucket Sort algorithm (Example):

- *Now sort the elements in each bucket using the insertion sort.*
  0 -> 22 -> 32
  1
  2 -> 62 -> 72
  3 -> 82
  4
  5
  6 -> 142
- *Now gather them together.*
  arr[] = {22, 32, 62, 72, 82, 142}

Bucket Sort use case and complexities:

- Bucket Sort works best when the input elements are uniformly distributed across the range. It's most efficient when the range of input values is not significantly larger than the number of elements.

- Bucket Sort has an average time complexity of **O(n + k)**, where n is the number of elements and k is the number of buckets. However, the time complexity can degrade to **O(n^2)** if the buckets are not well-distributed or if the number of elements in some buckets is very large.

Radix Sort

- Radix Sort is a sorting algorithm that works by sorting the input numbers digit by digit.
- It sorts the numbers by processing individual digits from the least significant digit (LSD) to the most significant digit (MSD) or vice versa.

Radix Sort Algorithm:

- Determine the maximum number of digits among all the numbers in the input array. Let this value be `max_digits`.
- For each digit position from the least significant digit (LSD) to the most significant digit (MSD), do the following:
  - Create `10` buckets (0 to 9) to represent each digit value.
  - Traverse through the input array and place each number into the appropriate bucket based on the digit at the current position.
  - Concatenate the numbers from all the buckets back into the original array.
- After processing all digit positions, the array will be sorted.

Radix Sort (Note)*:

- Radix Sort can be applied in two ways:
  - LSD Radix Sort (starting from the rightmost digit) and
  - MSD Radix Sort (starting from the leftmost digit).
- The algorithm above describes LSD Radix Sort, which is more commonly used.

Radix Sort algorithm (Example):

Array: [170, 45, 75, 90, 802, 24, 2, 66]

1. Find the maximum number of digits: `max_digits = 3` (for number `802`).

2. Perform the sorting process for each digit position:

- *Bucket 0: [170, 90]*
- *Bucket 1: [801, 2]*
- *Bucket 2: [802]*
- *Bucket 3: []*
- *Bucket 4: [24, 45]*
- *Bucket 5: [75]*
- *Bucket 6: [66]*
- *Bucket 7: []*
- *Bucket 8: []*
- *Bucket 9: []*

3. Concatenate the numbers from all the buckets to get the sorted array: [2, 24, 45, 66, 75, 90, 170, 801, 802]

Radix Sort algorithm (Example):

*b. **Digit 2:***

- Bucket 0: [801, 802, 24, 45, 75, 66]

- Bucket 1: []

- Bucket 2: [2]

- Bucket 3: []

- Bucket 4: []

- Bucket 5: []

- Bucket 6: []

- Bucket 7: []

- Bucket 8: [170, 90]

- Bucket 9: []

Radix Sort algorithm (Example):

*c. **Digit 3 (MSD):***

- *Bucket 0: [2, 24, 45, 66, 75, 90, 170]*
- *Bucket 1: []*
- *Bucket 2: []*
- *Bucket 3: []*
- *Bucket 4: []*
- *Bucket 5: []*
- *Bucket 6: []*
- *Bucket 7: []*
- *Bucket 8: [801, 802]*
- *Bucket 9: []*

Radix **Sort use case and complexities:**

- Radix Sort has a time complexity of O(n+k),
  - where n is the number of elements and k is the number of digits in the maximum number.
- It can be very efficient for sorting large arrays of numbers with a small number of digits.

Sorting Algorithm	Time Complexity	Space Complexity	Stability	In-Place	Counting Based	Comparison Based
Selection Sort	O(n^2)	O(1)	Unstable	Yes	No	Yes
Insertion Sort	O(n^2)	O(1)	Stable	Yes	No	Yes
Bubble Sort	O(n^2)	O(1)	Stable	Yes	No	Yes
Quick Sort	O(n^2) avg	O(log n)	Unstable	Yes	No	Yes
Merge Sort	O(n log n)	O(n)	Stable	No	No	Yes
Radix Sort	O(nk)	O(n + k)	Stable	Yes	Yes	No
Bucket Sort	O(n^2) avg	O(n)	Stable	Yes	Yes	Yes