Sorting

- Sorting is arranging in a proper way for easy access
  1. Selection Sort
  2. Bubble Sort
  3. Insertion Sort

1. Selection Sort

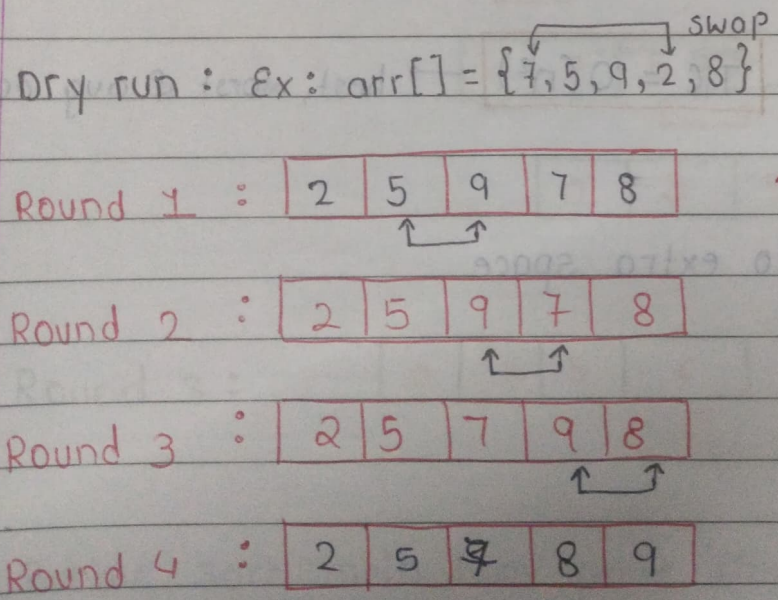
- It is a technique for arranging items in a specific order.
- Selection sort algorithm organizes elements by selecting smallest unsorted element and swapping it with first unsorted element.

Intitution -

- Search for ~~elem~~ smallest element in given array
- Put the obtained smallest element at 1st pos and swap.

Note - In case you want it in decreasing order find the largest element instead of the smallest element.

Dry run : Ex: arr[] = {7, 5, 9, 2, 8}



Page No.
Date : / /

Psuedocode :

```
void selection-sort (int arr[], int n) {
    for (int i = 0 ; i<n-1; i++) {
        int index = i;
        for (int j = i+1 ; j < n ; j++) {
            if (arr[j] < arr[index])
        }   index = j;
    swap (arr[i], arr[index]);
    }
}
for (int i = 0 ; i<n ; i++) {      // printing element
    cout << arr[i] << " ";
}
}
```

T.C :

Outer loop : 0 to n-1 times
Inner loop : i+1 to n

} using AP : $\frac{n(n+1)}{2} = \frac{n^2+n}{2}$

By T.C laws: $T.C = O(n^2)$ → best, worst & avg case

S.C :

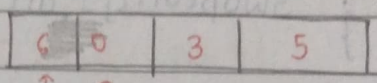
$O(1)$: No extra space

2. Bubble Sort

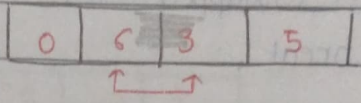
- In this we compare two value and swap it.
- In every pass one element goes at its right place. The last element goes at right place.
- We will need to (n-1) pass to complete this

Ex : arr[] = {6, 3, 0, 5}

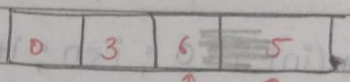
Round 1 : for i = 0



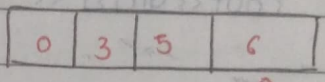
i = 1



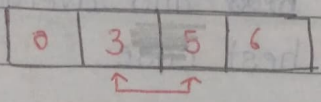
i = 2

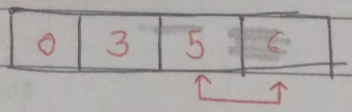


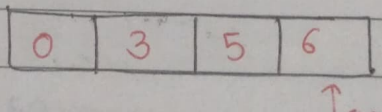
i = 3



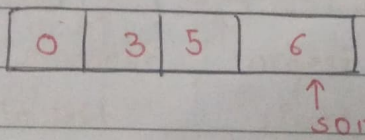
Round 2 :







Round 3 :



Bubble sort mein hum pehle first pos. Value lete uske baad adjacent element check karte agar woh usse chota hai to aage lete aur agar bada hai toh new value hold karte until we get sorted array.

Psuedocode :

~~void b~~

```
void bubblesort (int arr[], int n) {
    for (int i = n-2 ; i>=0 ; i--) {
        bool swapped =0 ;
        for (int j= 0 ; j <=i ; j++ ) { if (arr[j]>arr[j+1]){
            swapped =1 ;
            swap(arr[j],arr[j+1]) ;
    } }
    if (swapped ==0)
    break;
}
    for(int i=0 ; i<n ; i++)
    cout<<arr[i] <<" ";
} }
```

T·C : $O(n^2)$ --- worst ~~best~~ and average case

$O(n)$ --- best case

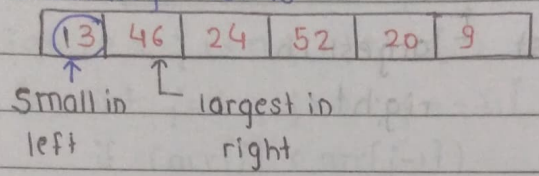
S·C : $O(1)$

3. Insertion Sort Algorithm

- Search an element in each iteration from the unsorted array (using loop).
- Place it in its corresponding position in the sorted array & shift the corresponding position remaining elements, accordingly (using an loop & swaping).
- The "inner loop" basically shifts the element using swapping.

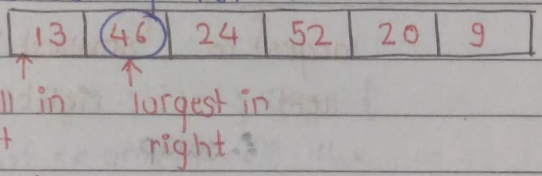
DRY RUN : arr = {13, 46, 24, 52, 20, 9}

Round 1 : Checking for 13



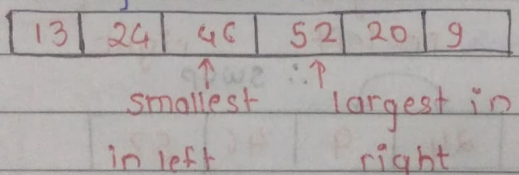
∴ no swap

Round 2 : Checking for 46



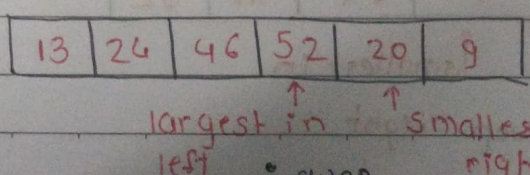
∴ no swap

Round 3 : Checking for 52

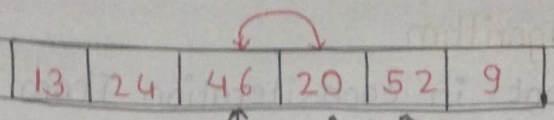


∴ no swap

Round 4 : Checking for 20

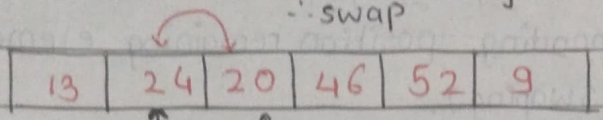


∴ swap



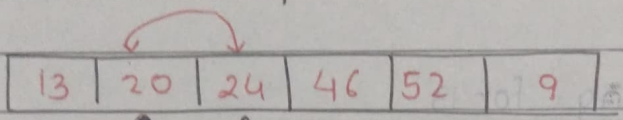
largest in left — smallest in right

∴ swap



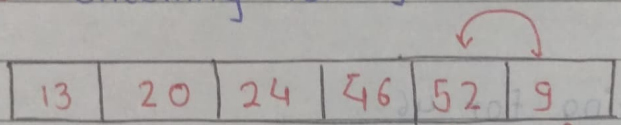
largest in left — smallest in right

∴ swap



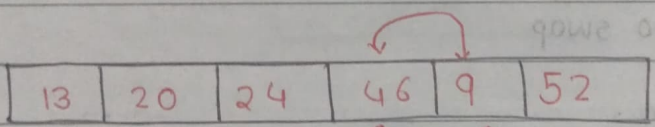
smallest in left — largest in right

Round 5 : checking for 9



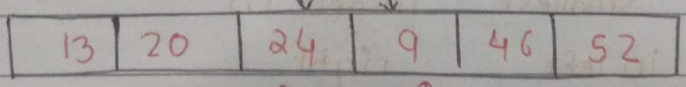
largest in left — smallest in right

∴ swap



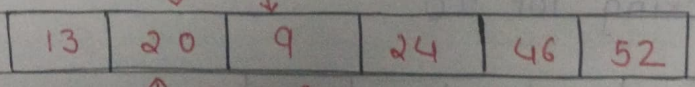
largest in left — smallest in right

∴ swap



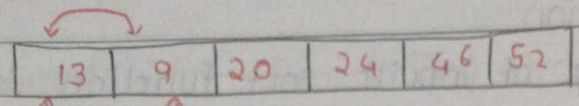
largest in left — smallest in right

∴ swap



largest in left — smallest in right

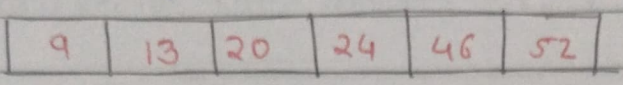
∴ swap



largest in left

smallest in right

∴ swap



---- sorted

Psuedocode :

```
void insertionSort (int arr[],int n){
    for (int i=1 ; i<n ; i++) {
        for (int j=1 ; j>0 ; j--) {
            if (arr[j] < arr[j-1])
                swap (arr[j] , arr[j-1]);
            else
                break;
    } }
    for (int i=0 ; i<n ; i++) {
        cout << arr[i] << " ";
    }
}
```

T.C : $O(n^2)$ --- two loops (2 for loops)

S.C : $O(1)$

4. Inbuilt Sort Function

- Time complexity of inbuilt sort function is O(nlogn) <<< $O(n^2)$
- Inbuilt function of sort is present in STL

Psuedocode :

```
int main () {
std::vector <int> numbers = {5,2,8,1,9};
        //ascending order
//Type-I  std::sort (numbers.begin(), numbers.end());
 for (int num : numbers) {
 }      cout << num << " ";


 ← std::sort (numbers +2 , numbers +n);
  for (int i = 0 ;
   int n = numbers.size();
//Type-II  Std :: sort (numbers +2 , numbers +n);
   for (int i=0 ; i<n; i++) {
    }     cout << numbers [i] << " ";

 }      return 0;
```

Type-II will sort the array which lies in range of 2 to n-1.

Psuedocode : For Descending Order

```
int main() {
    std::vector <int> numbers = {5,2,8,1,9};

    std :: sort (numbers.begin(), numbers.end(), std::greater <int> ());

    for (int num =0 ; num < n ; num++) {
            std :: cout << num << " ";
    }
}
```

5. Counting Sort Algorithm

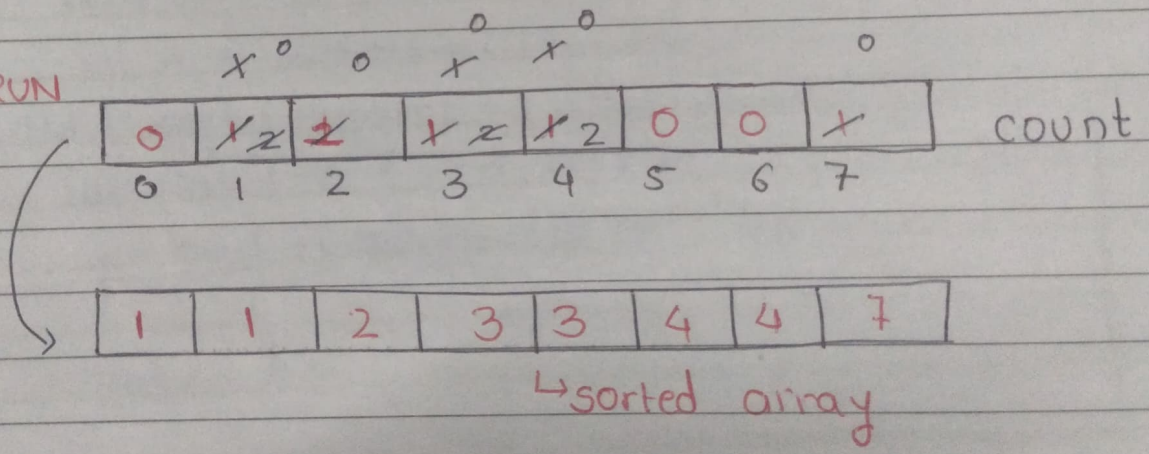
- We use this method only when the range btw minimum no. & greatest no. is lesser. No. of data doesn't matter but range should be lesser.
- Counting sort is used for positive integers.
- **T.C : O(n + range)**
  - ↑ if range is smaller then T.C can be O(n). i.e. linear time complexity

Ex : {1, 4, 1, 3, 2, 4, 3, 7}

Approach

- We first store the count or frequency of every no. in a count array.
- Then we use another array for sort and we reduce the frequency until it becomes one.

DRY RUN



Psuedocode -

```
void countingSort (std:: vector <int> &arr) {
    int largest = INT_MIN;
    for (int i=0 ; i< arr.size() ; i++) {
        largest = std::max (largest, arr[i]);
    }
    // for counting frequencies
    std:: vector <int> count (largest + 1, 0);
    for (int i = 0 ; i < arr.size() ; i++) {
        count [arr[i]] ++ ;
    }
    // for sorting ; int j=0;
    for (int i =0 ; i < count.size() ; i++) {
        while (count [i] > 0) {
            arr[j] = i ;
            j++ ;
            count [i] -- ;
        }
    }
}
```

Homework

1) Use insertion sort algorithm to ~~start~~ sort the array of integers in decreasing order.

Ans,

```
for (int i=1 ; i<n ; i++) {
    for (int j=i ; j>0 ; j++ ) {
        if (arr[j] > arr[j-1])
            swap(arr[j], arr[j-1]);
        else
            break;
} }
```

2) Insertion sort algorithm to sort the array of integers in increasing order if we start from the last element of the array.

```
for (int i=n-1 ; i>=0 ; i--) {
    for (int j=i ; j<n ; j++) {
        if (arr[j] < arr[j-1])
            swap(arr[j], arr[j-1]);
        else
            break;
} }
```

6. Merge Sort Algorithm

It is a classic example of "divide and conquer" strategy for sorting an array.

Key Concepts :

1. Divide and Conquer : The array is recursively divided into two halves until each half contains only one element. These smaller arrays are then merged in sorted array.

2. Merge Functions : This function takes two started halves and merges them into a single sorted array. It uses a temporary array ~~the~~ to store the result in sorted ~~array~~ order ensuring that the merged result is sorted.

Approach :

1. mergeSort
   - Divide the array into two halves until each sub-array contains one element.
   - Recursively call mergeSort on each halves until reaching the base case (low >= high)

2. merge
   - Use two pointers, left and right, to traverse sorted halves.
   - Compare elements at the pointers and add the smaller element to a temporary array.
   - Once all elements from either half are ~~neede~~ added, add any remaining elements from the other half
   - Copy the contents of the temporary array back into the original array segment.

Complexity Analysis :

- Time Complexity = O(N log N)
- Space Complexity : O(N)

Psuedocode :

```
void merge (in vector <int> &arr, int low, int mid, int high) {
    vector <int> temp ;
    int left = low;
    int right = mid + 1;

    while (left <= mid && right <= high) {
        if (arr[left] <= arr[right]) {
            temp.push_back (arr [left]);
            left ++ ;
        }
        else {
            temp.push_back (arr [right]);
            right ++;
        }
    }

    while (right <= high) {
        temp.push_back (arr [right]);
        right ++;
    }

    while (left <= mid) {
        temp.push_back (arr [left]);
        left ++;
    }

    for (int i = low ; i <= high ; i++) {
        arr [i] = temp [i - low] ;
    }
}
void mergeSort (vector <int> & arr , int low , int high) {
    if (low >= high) return ;
    int mid = (low + high) / 2 ;
    mergeSort (arr, low, mid) ;
    mergeSort (arr, mid+1 , high) ;
    merge (arr, low, mid, high) ;
}
```

7. Quick Sort Algorithm

It is another powerful sorting technique based on divide-and-conquer approach. It sorts by selecting a "pivot" element, then partitioning the array into subarrays that are recursively sorted around the pivot.

Key Concepts :

1. Pivot selection - Choose an element as pivot. This could be -
   - The first element
   - The last element
   - The median element
   - A randomly choosen element

Approach :

1. partition function :
   - ~~starting~~ selects the pivot as the first element of the array segment (arr[low]).
   - Two pointers, i (starting at low) and j (starting at high), traverse the array segment to reorder elements.
   - i moves right, stopping when it finds an element larger than the pivot. j moves left, stopping when it finds an element smaller than the pivot.
   - When i < j, elements at i and j are swapped to move smaller elements left of the pivot and larger elements right.
   - After i and j meet, the pivot is placed at arr[j], ensuring it is in its correct position.

2. qs (Quick Sort Recursive Helper)
   - Recursively calls partition to sort elements on the left and right of the pivot.
   - Recursion continues until the array is fully partioned

and sorted.

3. Quicksort Wrapper
   - Calls qs to sort the entire array, starting from index 0 to n-1.

Psuedocode :

```
int partition(vector <int> &arr, int low, int high) {
    int pivot = arr[low];
    int i = low;
    int j = high;

    while (i<j) {
      while (arr[i] <= pivot && i<=high-1) {
          i++;
      }

      while (arr[i] > pivot && j>=low+1) {
          j--;
      }

      if (i <j) swap(arr[i], arr[j]); // swap elements to keep
                                         smaller on left
    }
swap(arr[low], arr[j]); // Place pivot in its correct sorted
    return j; // returns pivot index                position
}


void qs(vector <int> &arr, int low, int high) {
    if (low < high) {
        int pIndex = partition(arr, low, high);
        qs(vec arr, low, pIndex-1);      // left part sort
        qs(arr, pIndex+1, high);         // right part sort
    }
}
```

```
vector <int> quickSort (vector <int> arr) {
    qs(arr, 0, arr.size() - 1);
    return arr;
}
```

Space and Time Complexity

Case	Time Complexity	Space Complexity
Best Case	$O(N \cdot \log N)$	$O(\log N)$
Average Case	$O(N \cdot \log N)$	$O(\log N)$
Worst Case	$O(N^2)$	$O(N)$

8. Radix Sort

- A non-comparative sorting algorithm that sorts integers or strings by processing digits or characters, starting from the least significant to the most significant digit (LSD Radix Sort) or vice versa (MSD Radix sort).

- Use Case - Best suited for sorting integers or fixed-size keys, especially with a large dataset and small range of digits

Key Concept

1. Digit - by - Digit Sorting :

- Sort elements based on individual digits/characters, starting from the least or most significant place value.
- A stable sorting algorithm (eg. counting sort) is used to sort at each digite level.

2. Place Value Sorting :

- Each digit is processed in ascending order or descending order of significance.
- The final array is obtained by concatenating the sorted array after all digits are processed.

3. Variations

- LSD Radix sort - Starts from the least significant digit (eg : Unit Place)
- MSD Radix sort - Starts from the most significant digit (eg. highest Place)

How Radix Sort Works

Given an array : [170, 45, 75, 90, 802, 24, 2, 66]

Step 1 : Identify the largest number to determine the number of digits (e.g - 802 has 3 digits).

Step 2 : Sort the based on the units place (least significant digit).

- Result - [170, 90, 802, 2, 24, 45, 75, 66]

Step 3 : Sort the array based on tens place

- Result - [802, 2, 24, 45, 66, 170, 75, 90]

Step 4 : Sort the array based on hundreds place (most significant digit).

- Result - [2, 24, 45, 66, 75, 90, 170, 802]

Algorithm

1. Find Maximum Number : Determine the maximum element to know the no. of digits.

2. Iterate Over Digits :
   - Start with the least significant digit
   - Perform Couting sort on digit's value at each iteration

3. Repeat for Each Place Value until all digits are processed

Time Complexity

$O(d \times (n + b))$

- d : No. of digits in largest no.
- n : No. of elements
- b : Base of the numeral system (10 for decimal)

Space complexit

$O(n + b)$

temporary arrays

Advantages -

- Linear Time Sorting - For datasets with fixed size digits
- Stable Sorting - Preserves the relative order of elements with equal keys.

Disadvantages

- Requires additional space for temporary arrays
- Not ideal for datasets with varying size of keys of floating point numbers

Psuedocode

```
int getMax (int arr[], int n) {
    int max = arr[0];
    for (int i = 1 ; i < n ; i++)
        if (arr[i] > max)
            max = arr[i];
    return max;
}

void countSort(int arr[], int n, int exp) {
    int output[n],
    int i, count[10] = {0} ;

    for (i = 0 ; i < n ; i++)
        count[ (arr[i] / exp) % 10 ]++ ;

    for (i = 1 ; i < 10 ; i++)
    count [i] += count [i-1];

    for (i = n-1 ; i >= 0 ; i--) {
        output[count [(arr [i] / exp) % 10] - 1] = arr [i] ;
    count [(arr[i] / exp) % 10]-- ;
}
```

```
    for (i=0 ; i<n ; i++)
        arr[i] = output[i];
}

void radixSort (int arr[], int n) {
    int m = getMax(arr, n);

    for (int exp = 1 ; m/exp >0 ; exp *= 10)
        countSort(arr, n, exp);
}
```

Page No.
Date: / /

9. Bucket Sort

Bucket Sort is a sorting algorithm that divides an input array into several smaller groups or "buckets", sort these buckets individually, & then merges them to produce the sorted array. It works well when the input data is uniformly distributed over a known range

How Bucket Sort Works

1. Creating Buckets - Divide the input range into equal-sized intervals, represented by buckets.

2. Distributing Elements - Insert each element from input array into the appropriate bucket.
   - Use formula : bucket-index = floor (n * array[i]),
     n → total no. of buckets

3. Sorting Buckets - Sort elements inside each bucket. Typically insertion sort is used for smaller buckets.

4. Concatenating Buckets - Combine all sorted buckets to get the final sorted array.

Key Features

- Input characteristics - Bucket sort is most efficient when the input data is uniformly distributed over a range.

- Sorting within Buckets - While insertion sort is common, any stable algorithm (Ex: Quick, ~~So~~ Merge) can be applied

- Output - The array is sorted in ascending order

Time Complexity

1. Best Case - $O(n+k)$ - k : buckets
2. Worst Case - $O(n^2)$ - all element in one bucket, resulting in single sorting operation
3. Average Case - $O(n + n\log(k))$ - elements are reasonably distributed

Space Complexity

$O(n+k)$ n: no. of elements ; k: no. of buckets

Example

Input : [0.78, 0.17, 0.39, 0.26, 0.72, 0.94, 0.21, 0.12, 0.23, 0.68]

Steps

1. Create 10 buckets
2. Distribute Elements :
   ex : $0.78 \times 10 = 7.8$ → bucket index 7
3. Sort elements within bucket
4. Conatenate buckets

Output : [0.12, 0.17, 0.21, 0.23, 0.26, 0.39, 0.68, 0.72, 0.78, 0.94]

Psuedocode

```
void bucketInsertionSort(vector<float>& bucket) {
    for (int i = 1; i < bucket.size(); ++i) {
        float key = bucket[i];
        int j = i - 1;
        while (j >= 0 && bucket[j] > key) {
            bucket[j+1] = bucket[j];
            j--;
        }
    } bucket[j+1] = key;
}
```

```
void bucketSort(float arr[], int n) {
    vector<float> b[n];
    for (int i = 0; i < n; i++) {
        int bi = n * arr[i];
        b[bi].push_back(arr[i]);
    }

    for (int i = 0; i < n; i++) {
        insertionSort(b[i]);
    }

    int index = 0;
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < b[i].size(); j++) {
            arr[index++] = b[i][j];
        }
    }
}
```

Applications

- Sorting floating-point numbers/decimals
- Distributing data for parallel processing
- certain data visualization & histogram-based algorithms

10. Cyclic Sort

Introduction

- Cycle Sort : An in-place, unstable sorting algorithm designed for minimizing memory writes.

- Key Idea : Divide ~~into~~ array into "cycles" & rotate elements within these cycles to their correct positions.

Algorithm steps

1. Start with an unsorted array of size n.
2. Initialize a cycle start position (variable cycleStart) at index 0.
3. For each element :
   - Compare it with all elements to its right.
   - Count elements smaller than the current element to find its correct position.
4. If an element is already in correct position / duplicates are encountered, move to next
5. Place each element in its correct position within cycle using a series of swap
6. Repeat for all cycles until the array is sorted

Advantages

- Memory Efficiency : Use minimal memory ; works in-place with a constant space complexity of O(1).
- Optimal Writes : Each element is written only once, reducing the no. of memory writes

Disadvantages

Time Complexity → $O(N^2)$ - nested loop : Worst Case
→ $O(N^2)$: Average Case

Unstable does not preserve relative order of duplicate elements

Psuedocode -

```
void cycleSort (int arr[], int n) {
  int writes = 0;
  for (int cycle-start = 0; cycle-start < n-1; cycle-start ++) {
     int item = arr [cycle-start];
     int pos = cycle-start;
     for (int i = cycle-start + 1; i < n; i++)
        if (arr[i] < item) pos++;

    if (pos == cycle-start) continue;
    while (item == arr[pos]) pos++;
    if (pos != cycle-start) {
        swap (item, arr[pos]);
        writes ++;
    }

    while (pos != cycle-start) {
     pos = cycle-start;
     for (int i = cycle-start + 1; i < n; i++)
       if (arr[i] < item) pos++;

    while (item == arr [pos]) poss ++;
    if (item != arr[pos]) {
        swap (item, arr[pos]);
        writes ++;
    }
  }
}
```

Input : {1, 8, 3, 9, 10, 10, 2, 4}

Output : {1, 2, 3, 4, 8, 9, 10, 10}

Alternate Approach

- Cyclic Sort

  Applicable when array elements are in range [1, N] or [0, N]

  - Correct position : index = value - 1 for [1, N]
  - Time complexity : O(N)
  - Space complexity : O(1)

Psuedocode

```
void cyclicSort (int arr[] ,int n) {
    int i = 0;
    while (i< n) {
      int correct = arr[i] - 1;
      if (arr[i] != arr[correct]) {
          swap (arr[i] , arr[correct]);
      } else {
          i++;
      }
    }
}
```

Summary

Advantages : Minimal memory ~~re~~ writes, in-place

Disadvantages : Not commonly used due to high $O(N^2)$ time complexity

Use Case : Specialized applications like low-write-memory sorting.