Design and Analysis of Algorithms (CSC-314)

B.Sc. CSIT

Unit-2: Iterative Algorithms

- A program is call iterative when there is a loop (or repetition).
- Example: Program to find the factorial of a number
- Time complexity of iteration can be found by finding the number of cycles being repeated inside the loop.

Unit-2: Iterative Algorithms

Euclidean algorithm:

- The Euclidean algorithm is one of the oldest numerical algorithms still to be in common use.
- It solves the problem of computing the greatest common divisor (gcd) of two positive integers.

Unit-2: Iterative Algorithms

Euclidean algorithm by subtraction

- The original version of Euclid's algorithm is based on subtraction: we recursively subtract the smaller number from the larger.

Greatest common divisor by subtraction.

  - 1 def gcd(a, b):
  - 2 if a == b:
  - 3 return a
  - 4 if a > b:
  - 5 gcd(a - b, b)
  - 6 else:
  - 7 gcd(a, b - a)

- Let's estimate this algorithm's time complexity (based on n = a+b). The number of steps can be linear, for e.g. gcd(x, 1), so the time complexity is O(n).
- This is the worst-case complexity, because the value x + y decreases with every step.

Unit-2: Iterative Algorithms

Euclidean algorithm by division

- Let’s start by understanding the algorithm and then go on to prove its correctness.
- For two, given numbers a and b, such that a ≥b:
  - if b=a, then gcd(a, b) = b,
  - otherwise gcd(a, b) = gcd(b, a mod b).
- **Greatest common divisor by dividing.**
  - 1 def gcd(a, b):
  - 2 if a % b == 0:
  - 3 return b
  - 4 else:
  - 5 return gcd(b, a % b)

Unit-2: Iterative Algorithms

Euclidean algorithm by division

Denote by (a_i, b_i) pairs of values a and b, for which the above algorithm performs i steps. Then $b_i \geqslant Fib_{i-1}$ (where Fib_i is the i-th Fibonacci number). Inductive proof:

1. for one step, $b_1 = 0$,
2. for two steps, $b \geqslant 1$,
3. for more steps, $(a_{k+1}, b_{k+1}) \rightarrow (a_k, b_k) \rightarrow (a_{k-1}, b_{k-1})$, then $a_k = b_{k+1}, a_{k-1} = b_k$, $b_{k-1} = a_k \bmod b_k$, so $a_k = q \cdot b_k + b_{k-1}$ for some $q \geqslant 1$, so $b_{k+1} \geqslant b_k + b_{k-1}$.

Fibonacci numbers can be approximated by:

$$Fib_n \approx \frac{\left(\frac{1+\sqrt{5}}{2}\right)^n}{\sqrt{5}} \quad (12.1)$$

Thus, the time complexity is logarithmic based on the sum of a and b — $O(\log(a+b))$.

Unit-2: Iterative Algorithms

You are given two positive numbers M and N. The task is to print greatest common divisor of M'th and N'th Fibonacci Numbers.

- The first few Fibonacci Numbers are 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, ……..
- Note that 0 is considered as 0'th Fibonacci Number.
- **Example 1:**
  - Input : M = 3, N = 6
  - Output : 2
  - Fib(3) = 2, Fib(6) = 8
  - GCD of above two numbers is 2
- **Example 2:**
  - Input : M = 8, N = 12
  - Output : 3
  - Fib(8) = 21, Fib(12) = 144
  - GCD of above two numbers is 3

Unit-2: Iterative Algorithms

- For Fibonacci Numbers do at your own.

  We have covered sufficient examples in chapter one.

Unit-2: Iterative Algorithms

Sequential Search and its analysis

- Linear / sequential search is a very simple search algorithm. In this type of search, a sequential search is made over all items one by one.
- Every item is checked and if a match is found then that particular item is returned, otherwise the search continues till the end of the data collection.

Unit-2: Iterative Algorithms

Sequential Search and its analysis

- Sequential search, or linear search, is a search algorithm implemented on lists.
- It is one of the most intuitive approaches to search: simply look at all entries in order until the element is found.
- Given a target value, the algorithm iterates through every entry on the list and compares it to the target. If they match then it is a successful search and the algorithm returns true.
- If the end of the list is reached and no match was found, it is an unsuccessful search and the algorithm returns false

Unit-2: Iterative Algorithms

Given a list L of length n with the ith element denoted L_i, and a target value denoted T:

for i from 0 to n−1:

if L_i=T:

return i

return −1

- The basic and dominant operation of sequential search (and search algorithms in general) is comparison. Thus we can measure the running time of this algorithm by counting the number of comparisons it makes given a list of size n. i.e. O(n).
- The algorithm is iterative, meaning the only space needed is the single variable that keeps track of the index of the current element being checked. As such, sequential search always has a constant spatial complexity O(1).

Unit-2: Iterative Algorithms

Sorting Algorithms:

- A Sorting Algorithm is used to rearrange a given array or list elements according to a comparison operator on the elements.
- The comparison operator is used to decide the new order of element in the respective data structure.
- For example: The below list of characters is sorted in increasing order of their ASCII values. That is, the character with lesser ASCII value will be placed first than the character with higher ASCII value.

geeksforgeeks =====> eeeefggkkorss

Input Output

Unit-2: Iterative Algorithms

Sorting Algorithms:

- Sorting Algorithms are methods of reorganizing a large number of items into some specific order such as highest to lowest, or vice-versa, or even in some alphabetical order.
- These algorithms take an input list, processes it (i.e, performs some operations on it) and produce the sorted list.
- The most common example we experience every day is sorting clothes or other items on an e-commerce website either by lowest-price to highest, or list by popularity, or some other order.
- Some Examples of sorting algorithms are:
  - Bubble Sort,
  - Selection Sort and
  - Insertion Sort

Unit-2: Iterative Algorithms

Sorting Algorithms:

- Sorting Algorithms are methods of reorganizing a large number of items into some specific order such as highest to lowest, or vice-versa, or even in some alphabetical order.
- These algorithms take an input list, processes it (i.e, performs some operations on it) and produce the sorted list.
- The most common example we experience every day is sorting clothes or other items on an e-commerce website either by lowest-price to highest, or list by popularity, or some other order.
- Some Examples of sorting algorithms are:
  - Bubble Sort,
  - Selection Sort and
  - Insertion Sort

Unit-2: Iterative Algorithms

Bubble Sort:

- Bubble sort, also referred to as comparison sort, is a simple sorting algorithm that repeatedly goes through the list, compares adjacent elements and swaps them if they are in the wrong order.
- This is the most simplest algorithm and inefficient at the same time.
- Yet, it is very much necessary to learn about it as it represents the basic foundations of sorting.
- Understand the working of Bubble sort
  - Bubble sort is mainly used in educational purposes for helping students understand the foundations of sorting.
  - This is used to identify whether the list is already sorted. When the list is already sorted (which is the best-case scenario), the complexity of bubble sort is only O(n).
  - In real life, bubble sort can be visualised when people in a queue wanting to be standing in a height wise sorted manner swap their positions among themselves until everyone is standing based on increasing order of heights.

Unit-2: Iterative Algorithms

Bubble Sort:

Example:

First Pass:

(**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.

Second Pass:

(**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**)

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.

Third Pass:

(**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**)

Unit-2: Iterative Algorithms

Bubble Sort:

```
void swap(int *xp, int *yp)
{
        int temp = *xp;
        *xp = *yp;
        *yp = temp;
}

void bubbleSort(int arr[], int n)
{
int i, j;
bool swapped;
for (i = 0; i < n-1; i++)
{
        swapped = false;
        for (j = 0; j < n-i-1; j++)
        {
                if (arr[j] > arr[j+1])
                {
                swap(&arr[j], &arr[j+1]);
                swapped = true;
                }
        }
        // IF no two elements were swapped by inner loop, then break
        if (swapped == false)
                break;
}
}
```

Unit-2: Iterative Algorithms

Bubble Sort:

Complexity Analysis

Time Complexity of Bubble sort

- **Best case scenario:** The best case scenario occurs when the array is already sorted. In this case, no swapping will happen in the first iteration (The `swapped` variable will be false). So, when this happens, we break from the loop after the very first iteration. Hence, time complexity in the best case scenario is `O(n)` because it has to traverse through all the elements once.
- **Worst case and Average case scenario:** In Bubble Sort, `n-1` comparisons are done in the 1st pass, `n-2` in 2nd pass, `n-3` in 3rd pass and so on. So, the total number of comparisons will be:

```
Sum = (n-1) + (n-2) + (n-3) + ..... + 3 + 2 + 1
Sum = n(n-1)/2
```

Hence, the time complexity is of the order $\mathbf{n^2}$ or $\mathbf{O(n^2)}$.

Space Complexity of Bubble sort

The space complexity for the algorithm is **O(1)**, because only a single additional memory space is required i.e. for temporary variable used for swapping.

Unit-2: Iterative Algorithms

Selection Sort

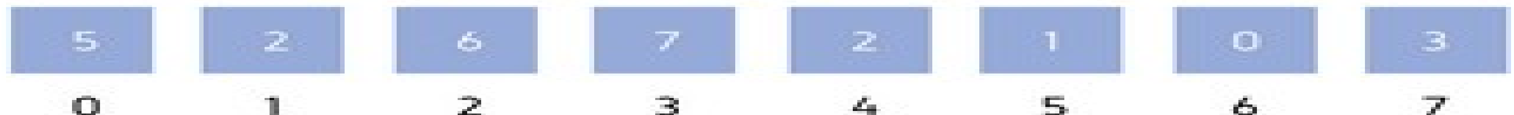
- Selection sort is a simple comparison-based sorting algorithm. It is in-place and needs no extra memory.
- The idea behind this algorithm is pretty simple. We divide the array into two parts: sorted and unsorted. The left part is sorted subarray and the right part is unsorted subarray. Initially, sorted subarray is empty and unsorted array is the complete given array.
- We perform the steps given below until the unsorted subarray becomes empty:
  - Pick the minimum element from the unsorted subarray.
  - Swap it with the leftmost element of the unsorted subarray.
  - Now the leftmost element of unsorted subarray becomes a part (rightmost) of sorted subarray and will not be a part of unsorted subarray.

Unit-2: Iterative Algorithms

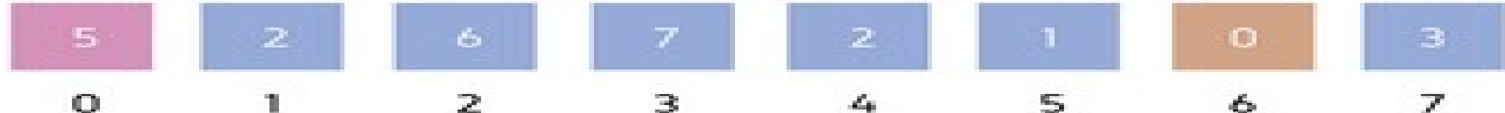
Selection Sort

A selection sort works as follows:

- Part of unsorted array
- Part of sorted array
- Leftmost element in unsorted array
- Minimum element in unsorted array



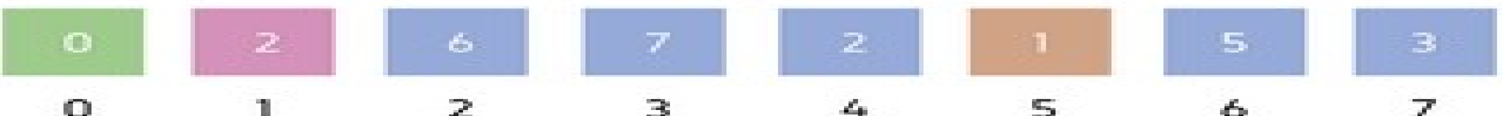
This is our initial array A = [5, 2, 6, 7, 2, 1, 0, 3]



Leftmost element of unsorted part = A[0]

Minimum element of unsorted part = A[6]

We will swap A[0] and A[6] then, make A[0] part of sorted subarray.



Leftmost element of unsorted part = A[1]

Minimum element of unsorted part = A[5]

We will swap A[1] and A[5] then, make A[1] part of sorted subarray.

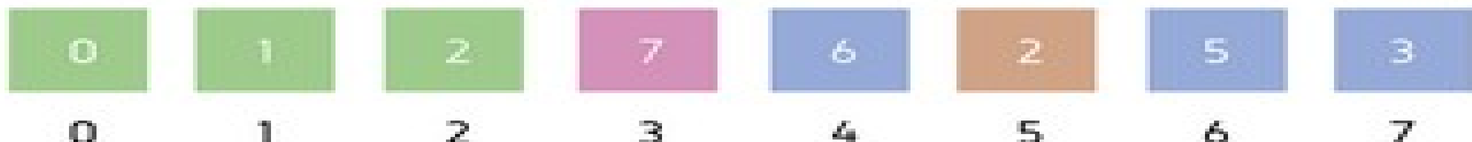
Unit-2: Iterative Algorithms

Selection Sort

Leftmost element of unsorted part = A[2]

Minimum element of unsorted part = A[4]

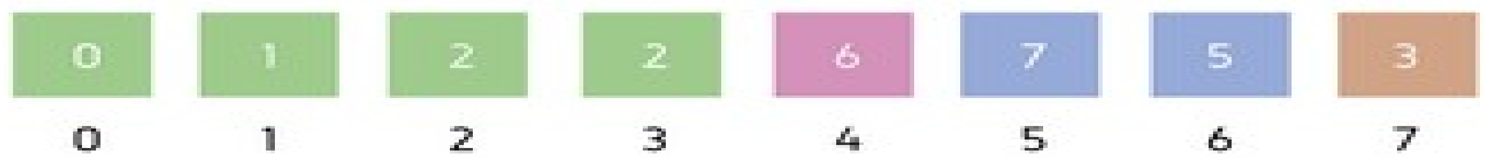
We will swap A[2] and A[4] then, make A[2] part of sorted subarray.



Leftmost element of unsorted part = A[3]

Minimum element of unsorted part = A[5]

We will swap A[3] and A[5] then, make A[3] part of sorted subarray.



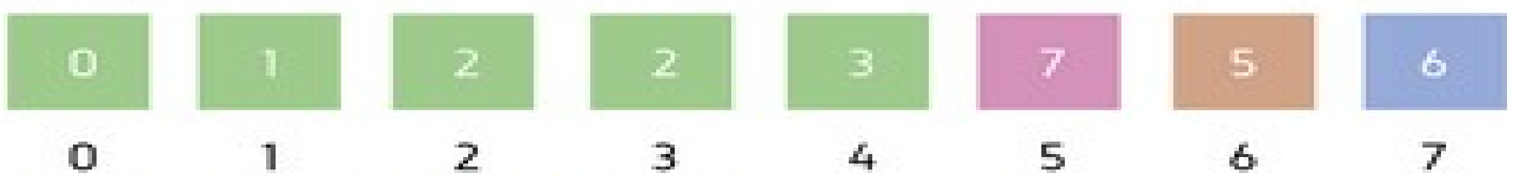
Leftmost element of unsorted part = A[4]

Minimum element of unsorted part = A[7]

We will swap A[4] and A[7] then, make A[4] part of sorted subarray.

Unit-2: Iterative Algorithms

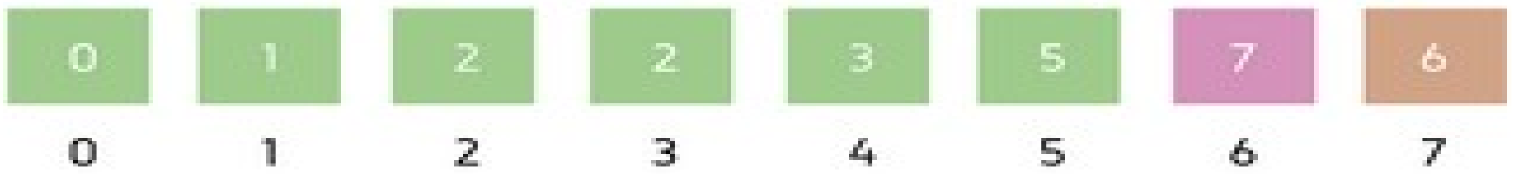
Selection Sort



Leftmost element of unsorted part = A[5]

Minimum element of unsorted part = A[6]

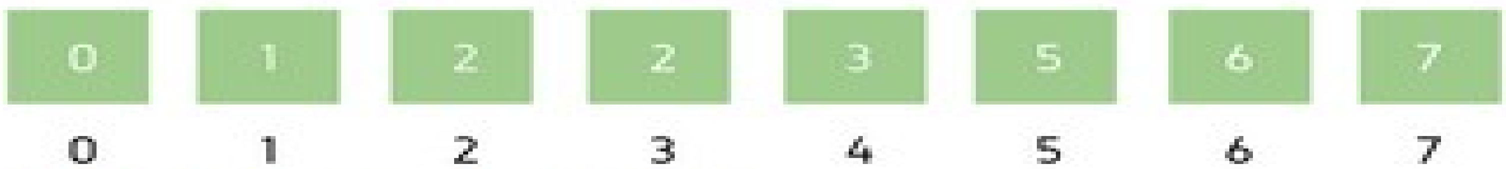
We will swap A[5] and A[6] then, make A[5] part of sorted subarray.



Leftmost element of unsorted part = A[6]

Minimum element of unsorted part = A[7]

We will swap A[6] and A[7] then, make A[6] part of sorted subarray.



This is the final sorted array.

Unit-2: Iterative Algorithms

Selection Sort

```
SelectionSort(Arr[], arr_size):
        FOR i from 1 to arr_size:
            min_index = FindMinIndex(Arr, i, arr_size)

            IF i != min_index:
                swap(Arr[i], Arr[min_index])
            END of IF
        END of FOR
```

Suppose, there are 'n' elements in the array. Therefore, at worst case, there can be n iterations in FindMinIndex() for start = 1 and end = n. No auxiliary space used.

Total iterations = (n – 1) + (n – 2) + . . . + 1 = (n * (n – 1)) / 2 = (n2 – n) / 2

Therefore,

Time complexity: $O(n^2)$

Space complexity: O(1)

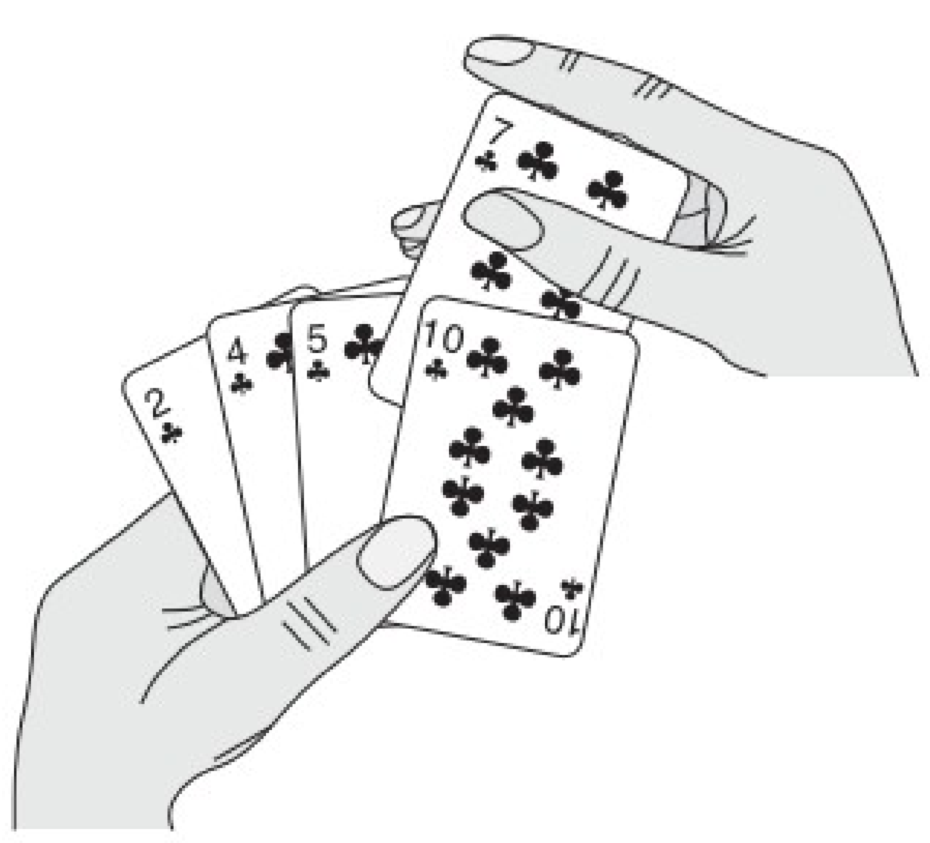
Unit-2: Iterative Algorithms

Insertion Sort:

- Insertion sort is the sorting mechanism where the sorted array is built having one item at a time.
- The array elements are compared with each other sequentially and then arranged simultaneously in some particular order.
- The analogy can be understood from the style we arrange a deck of cards. This sort works on the principle of inserting an element at a particular position, hence the name Insertion Sort.

Unit-2: Iterative Algorithms

Insertion Sort:



Unit-2: Iterative Algorithms

Insertion Sort:

- Insertion sort is a simple sorting algorithm that works similar to the way you sort playing cards in your hands. The array is virtually split into a sorted and an unsorted part. Values from the unsorted part are picked and placed at the correct position in the sorted part.
- Algorithm : To sort an array of size n in ascending order:

1: Iterate from arr[1] to arr[n] over the array.

2: Compare the current element (key) to its predecessor.

3: If the key element is smaller than its predecessor, compare it to the elements before. Move the greater elements one position up to make space for the swapped element.

Unit-2: Iterative Algorithms

Insertion Sort works as follows:

1. The first step involves the comparison of the element in question with its adjacent element.
2. And if at every comparison reveals that the element in question can be inserted at a particular position, then space is created for it by shifting the other elements one position to the right and inserting the element at the suitable position.
3. The above procedure is repeated until all the element in the array is at their apt position.

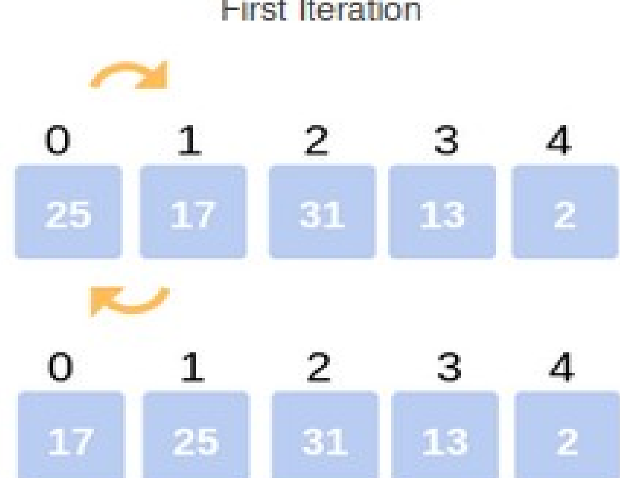
Let us now understand working with the following example:

Consider the following array: 25, 17, 31, 13, 2

First Iteration: Compare 25 with 17. The comparison shows 17< 25. Hence swap 17 and 25.

The array now looks like:

17, 25, 31, 13, 2



Insertion Sort:

Second Iteration: Begin with the second element (25), but it was already swapped on for the correct position, so we move ahead to the next element.

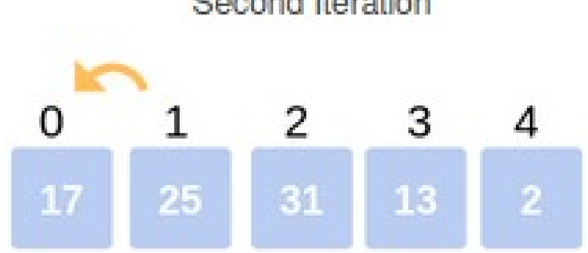
Now hold on to the third element (31) and compare with the ones preceding it.

Since 31> 25, no swapping takes place.

Also, 31> 17, no swapping takes place and 31 remains at its position.

The array after the Second iteration looks like:

17, 25, 31, 13, 2



Unit-2: Iterative Algorithms

Insertion Sort:

Third Iteration: Start the following Iteration with the fourth element (13), and compare it with its preceding elements.

Since 13< 31, we swap the two.

Array now becomes: 17, 25, 13, 31, 2.

But there still exist elements that we haven't yet compared with 13. Now the comparison takes place between 25 and 13. Since, 13 < 25, we swap the two.

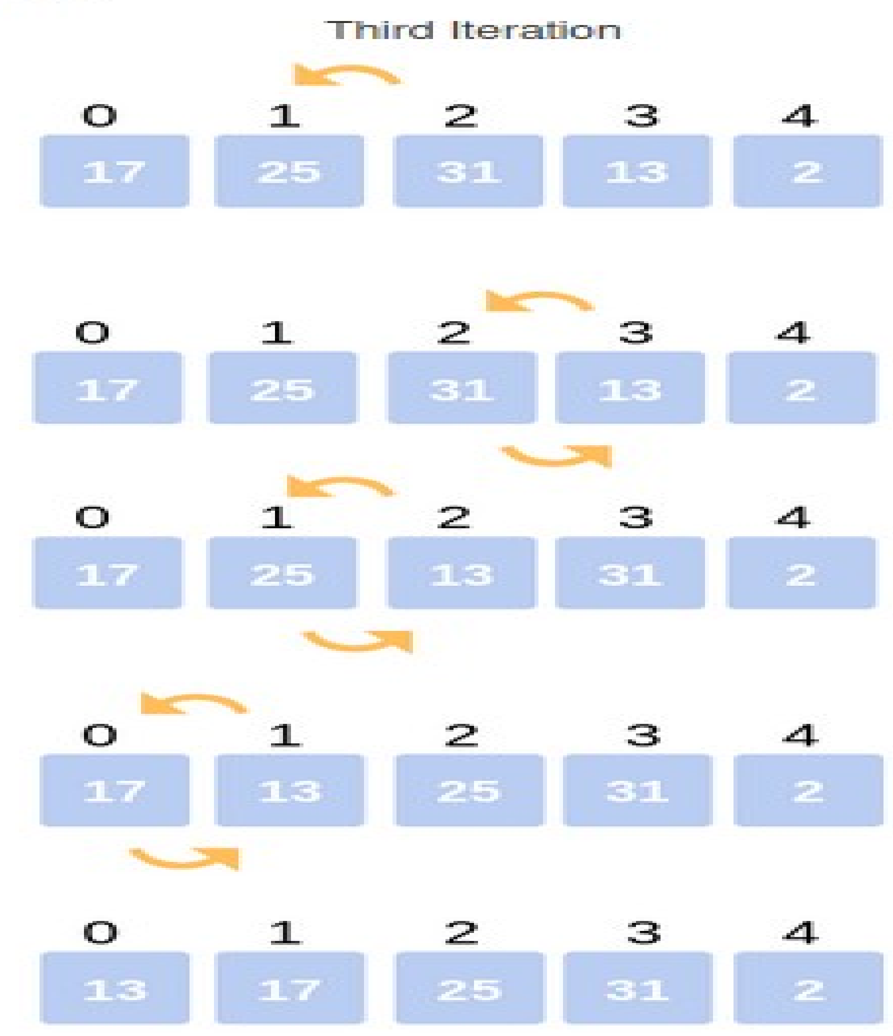
The array becomes **17, 13, 25, 31, 2**.

The last comparison for the iteration is now between 17 and 13. Since 13 < 17, we swap the two.

The array now becomes **13, 17, 25, 31, 2**.

Unit-2: Iterative Algorithms

Insertion Sort:



Unit-2: Iterative Algorithms

Insertion Sort:

Fourth Iteration: The last iteration calls for the comparison of the last element (2), with all the preceding elements and make the appropriate swapping between elements.

Since, 2< 31. Swap 2 and 31.

Array now becomes: 13, 17, 25, 2, 31.

Compare 2 with 25, 17, 13.

Since, 2< 25. Swap 25 and 2.

13, 17, 2, 25, 31.

Compare 2 with 17 and 13.

Since, 2<17. Swap 2 and 17.

Array now becomes:

13, 2, 17, 25, 31.

The last comparison for the Iteration is to compare 2 with 13.

Since 2< 13. Swap 2 and 13.

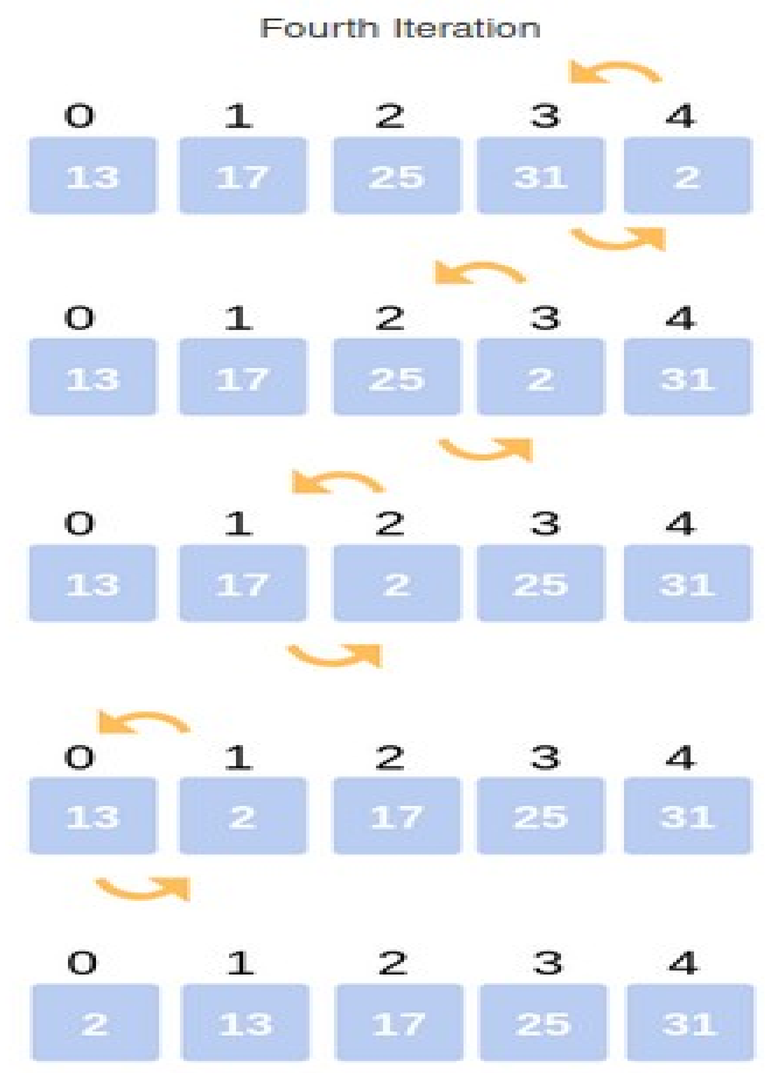
The array now becomes:

2, 13, 17, 25, 31.

This is the final array after all the corresponding iterations and swapping of elements.

Unit-2: Iterative Algorithms

Insertion Sort:



Unit-2: Iterative Algorithms

Insertion Sort:

Pseudocode

```
INSERTION-SORT(A)
   for i = 1 to n
        key ← A [i]
        j ← i – 1
         while j > = 0 and A[j] > key
                A[j+1] ← A[j]
                j ← j – 1
        End while
        A[j+1] ← key
  End for
```

Unit-2: Iterative Algorithms

Insertion Sort:

Time Complexity Analysis:

Even though insertion sort is efficient, still, if we provide an already sorted array to the insertion sort algorithm, it will still execute the outer for loop, thereby requiring n steps to sort an already sorted array of n elements, which makes its best case time complexity a linear function of n.

Wherein for an unsorted array, it takes for an element to compare with all the other elements which mean every n element compared with all other n elements. Thus, making it for n x n, i.e., n2 comparisons. One can also take a look at other sorting algorithms such as *Merge sort, Quick Sort, Selection Sort*, etc. and understand their complexities.

Worst Case Time Complexity [Big-O]: $O(n^2)$

Best Case Time Complexity [Big-omega]: O(n)

Average Time Complexity [Big-theta]: $O(n^2)$

Unit-2: Iterative Algorithms

Time Complexities of Sorting Algorithms:

Algorithm	Best	Average	Worst
Quick Sort	Ω(n log(n))	Θ(n log(n))	O(n^2)
Bubble Sort	Ω(n)	Θ(n^2)	O(n^2)
Merge Sort	Ω(n log(n))	Θ(n log(n))	O(n log(n))
Insertion Sort	Ω(n)	Θ(n^2)	O(n^2)
Selection Sort	Ω(n^2)	Θ(n^2)	O(n^2)
Heap Sort	Ω(n log(n))	Θ(n log(n))	O(n log(n))
Radix Sort	Ω(nk)	Θ(nk)	O(nk)
Bucket Sort	Ω(n+k)	Θ(n+k)	O(n^2)

Unit-2: Iterative Algorithms

Assignment:

- Discuss Binary Euclidean algorithm to find GCD and also mention its complexity.
-