DIPLOMA WALLAH

(Your **One Stop Hub** For Diploma Resources)

DATA STRUCTURES

WITH PYTHON

Complete Notes Based on Full Syllabus

- Diploma Engineering

4th Semester

www.diplomawallah.in

Notes prepared by **Sangam**

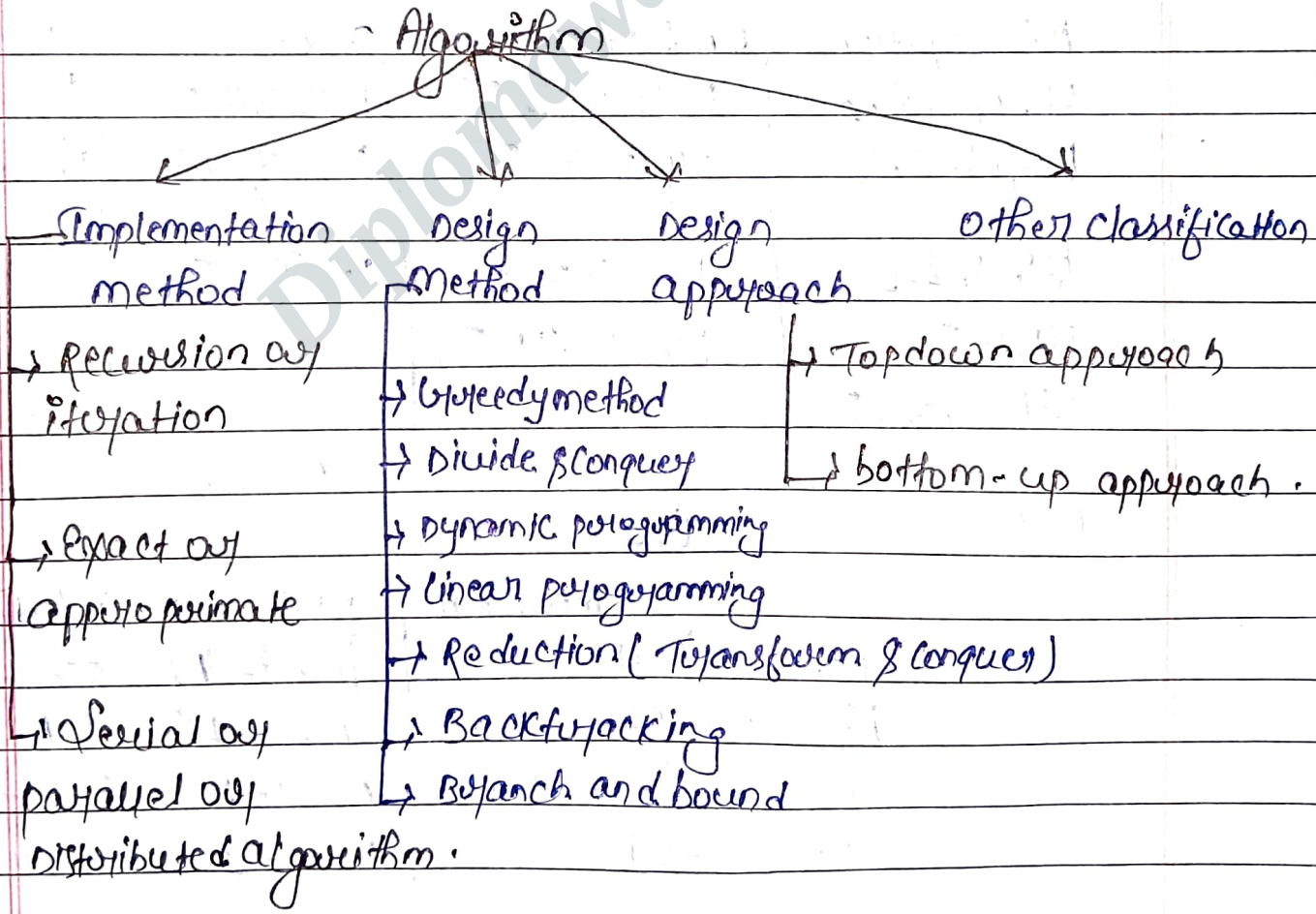
Sangam

Date
Page

unit - 03

Algorithm: — An algorithm is a procedure to solve a particular problem in a finite number of steps for a finite sized input.

The algorithms can be classified in various way.



Byute force →

A byute force algorithm is a straightforward method that tries all possible solutions to solve a problem without using any optimization or shortcut.

① Bubble Sort :—

[4, 1, 5, 2, 3]

1st iteration	2nd iteration	3rd	4th
4,1,5,2,3	1 4 2 3 5	1 2 3 4 5	1 2 3 4 5
1,4,5,2,3	1 4 2 3 5	1 2 3 4 5	1 2 3 4 5
1,4,5,2,3	1 2 4 3 5	1 2 3 4 5	
1,4,2,5,3	1 2 3 4 5		
1,4,2,3,5			
i=0	i=1	i=2	i=3

	comparison	pattern to find out comparison
i=0	4	n-i-1 : 5-1 = 4
i=1	3	n-i-1 5-1-1 = 3
i=2	2	n-i-1 5-2-1 = 2
i=3	1	n-i-1 5-3-1 = 1

in code

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

Example python code :→ Space complexity = O(1)

```
def bubble_sort(arr):
    n = len(arr)
    for i in range(n):
        swapped = False
        for j in range(0, n-i-1):
            if arr[j] > arr[j+1]:
                arr[j], arr[j+1] =
                arr[j+1], arr[j]   # Swap
                swapped = True
# example usage:
arr = [5, 3, 8, 4, 2]
bubble sort(arr)
print("Sorted array :", arr)
```

- best case - O(n)
- worst case = O(n²)
- Average case = O(n²)

Time complexity - O(n²) - average case. if swap
If not swap then it is - O(n)

- Bubble Sort is the simplest sorting algorithm that works by repeatedly swapping the adjacent elements if they are in the wrong order.
- Bubble Sort algorithm, sorts an array by repeatedly comparing adjacent elements and swapping them if they are in the wrong order.

- The algorithm iterates through the array multiple times, with each pass pushing the largest unsorted element to its position at the end.

Advantages

1. Simple to understand and easy to implement.
2. No extra space required (in-place sorting).
3. Useful for small datasets.
4. Detect unsorted condition quickly.
5. Best case time complexity : O(n).

✗ Disadvantage.
✗ Inefficient for large.
✗ Too many swap.
✗ Not suitable for real-time system
✗ No significant practical use.

Algorithm

1. Start from the first element
2. Compare each adjacent pair.
   - If arr[j] > arr[j+1] (for ascending order), Swap them.
3. After the first pass, the largest element will be at the end.
4. Repeat the process for the remaining elements (n-1 passes).

Pseudocode

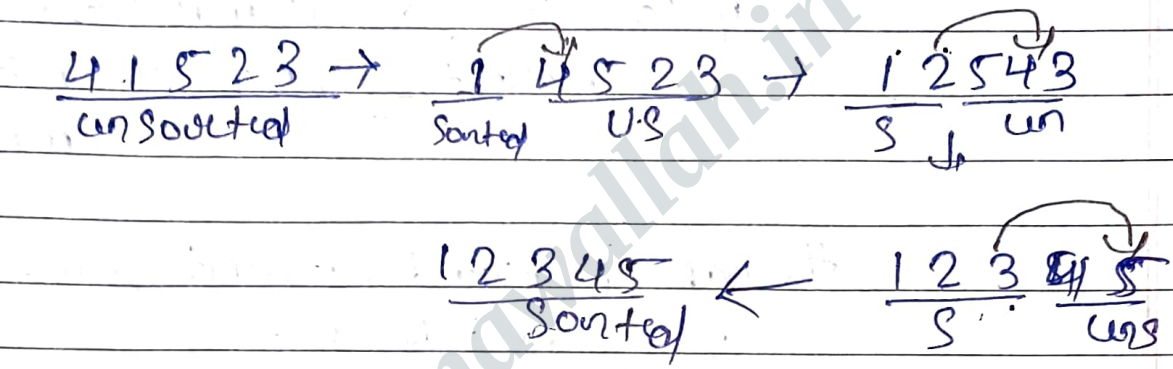
```
procedure bubbleSort(A)
    n = length(A)
    for i = 0 to n-1
        for j = 0 to n-i-2
            if A[j] > A[j+1]
                Swap A[j] and A[j+1]
```

Selection Sort

imagine two parts :

[Sorted, unsorted]

[4, 1, 5, 2, 3]



① Assume the first element is sorted and smallest.

② after that compare with all other indexes element. If the element is less than that then it swap its values.

Java code

```
eg. for(i=0; i<(n-1); i++){
        int smallestIndex = i;
        for(j=i+1; j<n; j++)
            if(arr[j] < arr[smallestIndex]){
                smallestIndex = j;
            }
        }
        swap(arr[i], arr[smallestIndex])
```

n(n-1)
2

Python

```
def selection_sort(arr):
    n = len(arr)
    for i in range(n-1):  # n-1 iterations
        smallest_index = i  # assume current index has the smallest value
        for j in range(i+1, n):
            if arr[j] < arr[smallest_idx]:
                smallest_index = j  # update if smaller value found.
        # Swap the found smallest value with the value at index i.
        arr[i], arr[smallest_idx] = arr[smallest_idx], arr[i]

# Example usage:
arr = [4,1,5,2,3]
selection_sort(arr)
print("Sorted array:", arr)
```

O/P →

Sorted array: [1,2,3,4,5]

• Selection Sort is a comparison based Sorting algorithm. It Sort an array by repeatedly selecting the smallest (or largest) element from the unsorted portion and swapping it with the first unsorted element.

This process will continue until the entire array is Sorted.:

Steps:.

(i) First we find the smallest elements and swap it with the first element. This way we get the smallest element at its correct position.

(ii) Then we find the smallest among the remaining elements (or second smallest) and swap it with the second elements.

(iii) we keep doing this until we get all elements moved to correct position.

ex

arr[] = [64, 25, 12, 22, 11]

swap

current element → minimum element

arr[] = [11, 25, 12, 22, 64]

arr[] = [11, 12, 25, 22, 64]

arr[] = [11, 12, 22, 25, 64]

Sorted

* Complexity analysis:-

Time complexity : $O(n^2)$
- one loop to select an element of array one by one. $O(n)$
- Another loop to execute and compare the element = $O(n)$

$\therefore$ $O(n) \cdot O(n)$

$O(n^2)$

Auxiliary space : $O(1)$, as the only extra memory used for temporary variable.

* Advantage
i) Simplicity and Easy to understand
ii) Fewer Swaps compared to Bubble Sort
iii) No extra Space Required (In-place Sort)
iv) Works well for Small list.
v) Does not require a stable Environment

* Disadvantage.
i) Time complexity is always $O(n^2)$ always perform selection sort $n(n-1)/2$.
ii) Not Suitable for large datasets.
iii) Wasted comparisons.
iv) poor Cache performance
v) Not adaptive.

What is Stable Sorting algorithm

A Stable sorting algorithm the relative order of equal elements is on the original input.

Original array: [40, 3, 2, 4b, 1]

ascending order, Stable Sort output [1, 2, 3, 4a, 4b]

unstable sort [1, 2, 3, 4b, 4a] (order of 4a and 4b is changed)

* Selection Sort not Stable by default because it swaps the minimum element with the current position, without checking whether its distributing the relative order of equal elements.

Step by step:
[5, 3, 8, 1, 2]
Step 1:
Find the smallest: 1
- Swap it with first number [5] → [1, 3, 8, 5, 2]

Step 2:
- look at [3, 8, 5, 2]
- Smallest is 2
- Swap with 3 → [1, 2, 8, 5, 3]

Step 3:
- look at [8, 5, 3]
- Smallest is 3
- Swap with 8 → [1, 2, 3, 5, 8]

Step 4:
- look at [5, 8]
- Smallest is 5 → already in place.

Final sorted list
[1, 2, 3, 5, 8]

Date
Page

Insertion Sort

[4, 1, 5, 2, 3]

A [prev]   A [prev + 1]

Iteration 1

4 | 1, 5, 2, 3
S | uns
↳ prev

current = 1
↳ considered or next element

↳ 1, 4, 5, 2, 3
shift to 1 and from current value the 1
shift to index 0.

Iteration 2

curr
1, 4, | 5, 2, 3
S | uns
↳ prev

current = 5

already sorted

Iteration 3

1, 4, 5, | 2, 3
S | uns
↳ prev

current = 2

loop. Arr [prev] > current
then, A [prev + 1] = A [prev]

(i) 1, 4, 5, 5, 3   Compare with 5
(ii) 1, 4, 4, 5, 3   Compare with 4
(iii) 1, 2, 4, 5, 3   Compare with 1

Iteration 4

1 2 4 5 3
sorted unsorted

current = 3
prev = 5

(i) 1 2 4 5 5 # compare with 5
(ii) 1 2 4 4 5 # compare with 4
1 2 3 4 5 # compare with 2.

#C++/java.

```
for(int i=1; i<n; i++)
    curr = i;
    prev = i-1;
    while(prev>=0 && A[prev]>curr)
    {
        A[prev+1] = A[prev]
        prev--;
    }
    A[prev+1] = curr;
}
```

started from 1 because we want to show i i.e first ele

4)0 * 4)
arr[0]: arr[1]
1 : 1

(1)

```
def insertion_sort(arr):
    n=len(arr)
    for i in range(1,n):
        curr = arr[i]
        prev = i-1
# move elements of arr[0,...i-1] that are greater than curr, to one position ahead.
        while prev>=0 and arr[prev]>curr:
            arr[prev+1] = arr[prev]
            prev = prev-1
```

⇐ arr[prev+1]=curr
← arr = [4,1,5,2,3]
← insertion_sort(arr)
← print("Sorted array:" arr)

(1,5)
curr = arr[1]
prev = 0 — index
prev >= 0 && arr[0] > 1
prev =

Algorithm

Step1: Start

Step2: Repeat Step 3 to 6 for i = 1 to n-1

Step3: set key = arr[i]

Step4: set prev = i-1

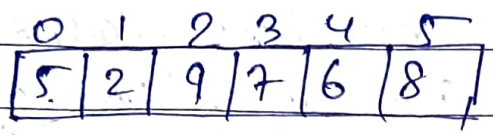
Step5: while prev > 0 and arr[Prev] > key

- Shift arr[Prev] to position i+1.
- Decrease prev = prev-1

Step6: place key at position prev+1

Step7: End

Linear Search / Sequential Search



@ 6 element

Search - 7

↳ it will be start from 0 index and starting searching till not found that number.

* Linear search is used to search an element in an array from start to end one by one. If the element is found, we store its position & terminate the loop.

If the number was found, we display the position along with message or if not found we display appropriate message.

Algorithm:-

Step 1:) step

Step 2: Input element & value that have to be search.

Step 3: loop from 0 to n-1:

if arr[i] == key

pos = i+1

flag = 1

Step 4:

Step 4: - If (flag == 1) {

print("found")

else: print("not found")

Code

```
def linear_search(arr, target):
    for i in range(len(arr)):
        if arr[i] == target:
            return i # return index if found
    return -1 # return -1 if not found

# Example usage
arr = [4,1,5,2,3]
search = 5

result = linear_search(arr, search)
if result != -1:
    print(f"Element {search} found at index {result}")
else:
    print(f"Element {target} not found in the list")
```

Time complexity:- O(1) if the element at the position starting.

worst case O(n) → if the element at the end or not present

Space Complexity:-

O(1) - no extra space used

Diploma Wallah.