27/02/2024

Page No.
Date

Day 1 of DSA

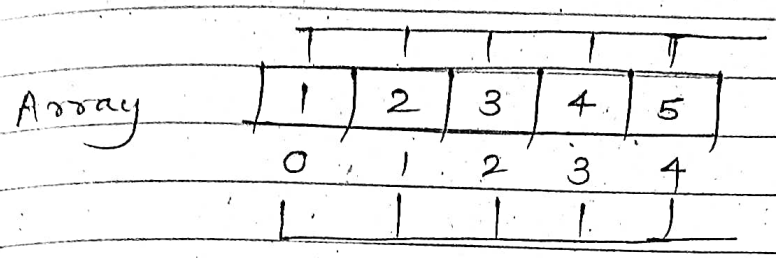
* Tasks :

Check Box	# Array.
☑	- Basic operations, Insertion, deletion, Searching.
☑	- Simple Manipulations: Reversal, rotation
☐	- Implement basic array operations - insertion, deletion, searching.
☐	- Solve simple array manipulation problems!

- Raj Kashid

Array :-

collection of items of same data type stored at contiguous memory locations.



- Basic Terms of Array :-

- Array Index - Array elements identified by index
  Index start from 0.

- Array element - Elements are items stored in an array.

- Array Length - length is determined by no. of elements it can contain

- Representation of Array :-

```
int arr[5];
char arr[10];
Float arr[15];
```

Page No.
Date

Static Array :-

Name

```
int array [10] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 }
data                                              Elements
Type
```

Dynamic Array :-

```
int * array = new int[5];
```

* Types of Array :-

① one-dimensional Arrays (only Row)

Elements	1	2	3	4	5
Index	0	1	2	3	4

② Two-dimensional Arrays (Row & column)

Col → 0 1 2

Row ↓				
0	0	1	2
1	3	4	5
2	6	7	8

③ Three-dimensional Arrays :- (2-Dimensional Arrays)

No. of 2D Arrays ↓

arr[1] [row] [col]

Col → 0 1 2

Row ↓			
0	1	2	3
1	4	5	6
2	7	8	9

* Types of Array operations :-

- Traversal - Traverse through array elements.
- Insertion - Insert elements
- Deletion - Delete elements
- Searching - Search an element
- Sorting - Maintain order of elements

} Of An Array

* Subarrays, Subsequences and Subsets in Array :-

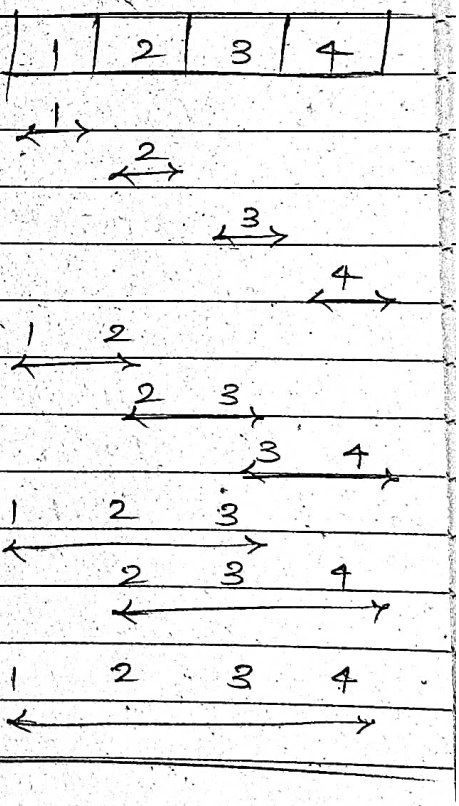
① Subarray - Contiguous part of array.

Formula = $n*\frac{(n+1)}{2}$

$n = 4 = 4*\frac{(5)}{2}$

$= \frac{20}{2}$

Subarrays = 10



② Subsequence -
A sequence by removing 0 or more elements

Formula = $(2^n - 1)$

$n = 4 = (2^4 - 1)$

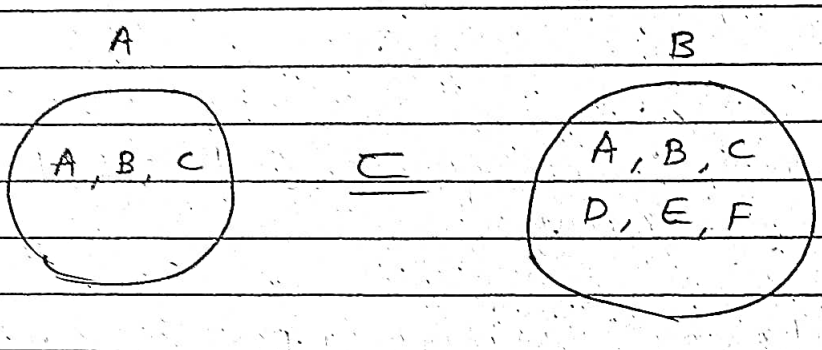
$16 - 1$

Subsequence = 15

(5) Subset –

A set has all its elements belonging to other sets.

$A \subset B$



* Basic operations in Array:-

(1) Array / searching:

operat^n of Finding a particular element or a group of elements in the array.

- Linear search — O(N) ⟶
- Binary search — $O(\log_2 N)$ ⟷|⟷
- Tertiary search. — $O(\log_3 N)$ ⟷|⟷|⟷

(2) Reverse an Array :-

Input - {1,2,3}
Output - {3,2,1}

(I) Array Reverse Using an Extra Array :-

- Create a New Array of same size as the original Array.
- Copy elements from the original array to the new Array in reverse order

* ~~Magic~~ Approach :-

```
                    New Array
                   ↗
int revArr[size];
For(int i=0; i<size; i++){

    revArr[i] = arr[size-i-1];
}
                    ↑
                original Array
```

arr = | 1 | 2 | 3 | size = 3
      0   1   2

1) i=0

revArr[0] = arr[3-0-1];
= arr[2]
revArr[0] = 3

revArr | 3 | | |
       0   1   2

2) i=1

revArr[1] = arr[3-1-1];
= arr[1]
revArr[1] = 2

| 3 | 2 | | |
0   1   2

3) $i = 2$

rev Arr[2] = arr[3-2-1];
= arr[0];
rev Arr[2] = 1

3	2	1
0	1	2

✓

Time complexity = O(n)
Space complexity = O(n).

(II) Array Reverse Using a Loop (In-place)

1) two pointers (start & end)
2) swap (start & end) pointers
3) move start ++ to the end & end -- to the start.

* Approach :-

```
while( start < end)
{
    int temp = arr[start];
    arr[start++] = arr[end];
    arr[end] = temp;
    start ++;
    end --;
}
```

Dry Run

Arr → | 1 | 2 | 3 |

0 1 2

S E

S = start

E = end

① · S = 0 & E = 2

temp = arr[0] = 1

arr[0] = arr[2] = 3

arr[2] = temp = 1

S++;

E--;

② S = 1 & E = 1

temp = arr[1] = 2

arr[1] = arr[1] = 2

arr[1] = temp = 2

S++;

E--;

③ · S = 2 & E = 0 loop breaks (S < E)

Time Complexity = O(n)
Space Complexity = O(1)

(III) Array Reverse Inbuilt Methods (Non In-place)

we have ready made Function in C++ called reverse

Add | algorithm | Header

Syntax :-

reverse(arr, arr + length);

& length = sizeof(arr) / sizeof(arr[0]);

Time complexity = O(n)
Space complexity = O(n)

(IV) Array Reverse Recursion (In-place or Non-In-place) :-

① Define a recursive function that takes array as input

② swap First & last elements

③ Recursively call the function with remaining sub arrays.

* Approach :

```
Void reverseArray(int arr[], int start,
                  int end) {

Base case → if (start >= end) return;
small calculat^ { int temp = arr[start]; arr[start] =
                  arr[end] = temp          arr[end];
Recursive call  { reverseArray(arr, start+1, end-1);
                }
```

Non-In-place = Using New things
In-place = Using Existing one

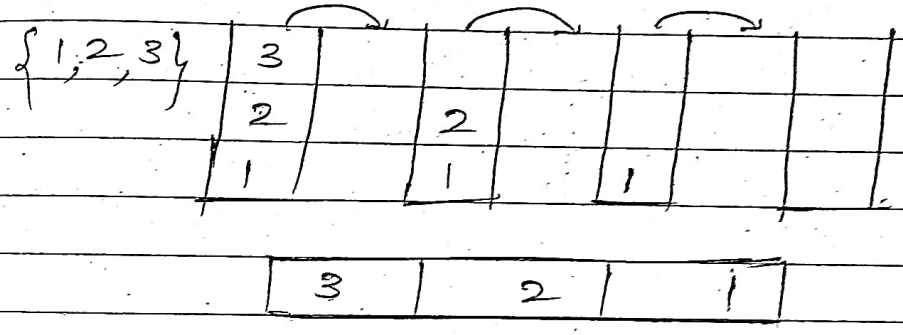
Time Complexity = O(n)

Space Complexity = O(n) - Non-In-place
O(log n) - In-place

(I) Array Reverse Stack. (Non-In-place)

(1) Push each element of the array onto a stack

(2) Pop elements from the stack to form reversed array.



* Approach :-

```
Stack<int> stack;

For(int i=0; i<size; i++){
    stack.push(arr[i]);
}

For(int i=0; i<size; i++){
    arr[i] = stack.top;
    stack.pop();
}
```

Time Complexity = O(n)
space Complexity = O(n)

Time complexity = O(N)
space complexity = O(N)

* Array left rotation by d positions :-

Approach 1

* Three Rules :-

(1) First store the elements from d to n-1 into the temp Array.

arr[] = { 1, 2, 3, 4, 5, 6, 7 }, d = 2

2 → 6

d → n-1

temp[] = [3, 4, 5, 6, 7]

(2) store First d elements of the original Array into the temp Array.

0 → d

temp[] = [3, 4, 5, 6, 7, 1, 2]

(3) Copy back the elements of the temp array into the original array.

arr ← temp

arr[] = temp[]

arr[] = [3, 4, 5, 6, 7, 1, 2]

- Approach :-

```
int temp[n]; int k=0;

(1) For(i=d → n)
    { temp[k] = arr[i];      d → n
      k++;
    }

(2) For( i=0 → d)
    { temp[k] = arr[i];      0 → d
      k++;
    }

(3) For( i=0 → n)
    {
      arr[i] = temp[i];      0 → n
    }
```

Time complexity : O(N*d)
Space complexity : O(1)

Approach 2

* Rotate one by one :-

1. Store the first element of the array in a temp variable
2. Shift the rest of the elements in the original array by one place
3. Update the last index of the array with temp variable.

- Repeat the above steps for the no. of left rotation

- Approach :-

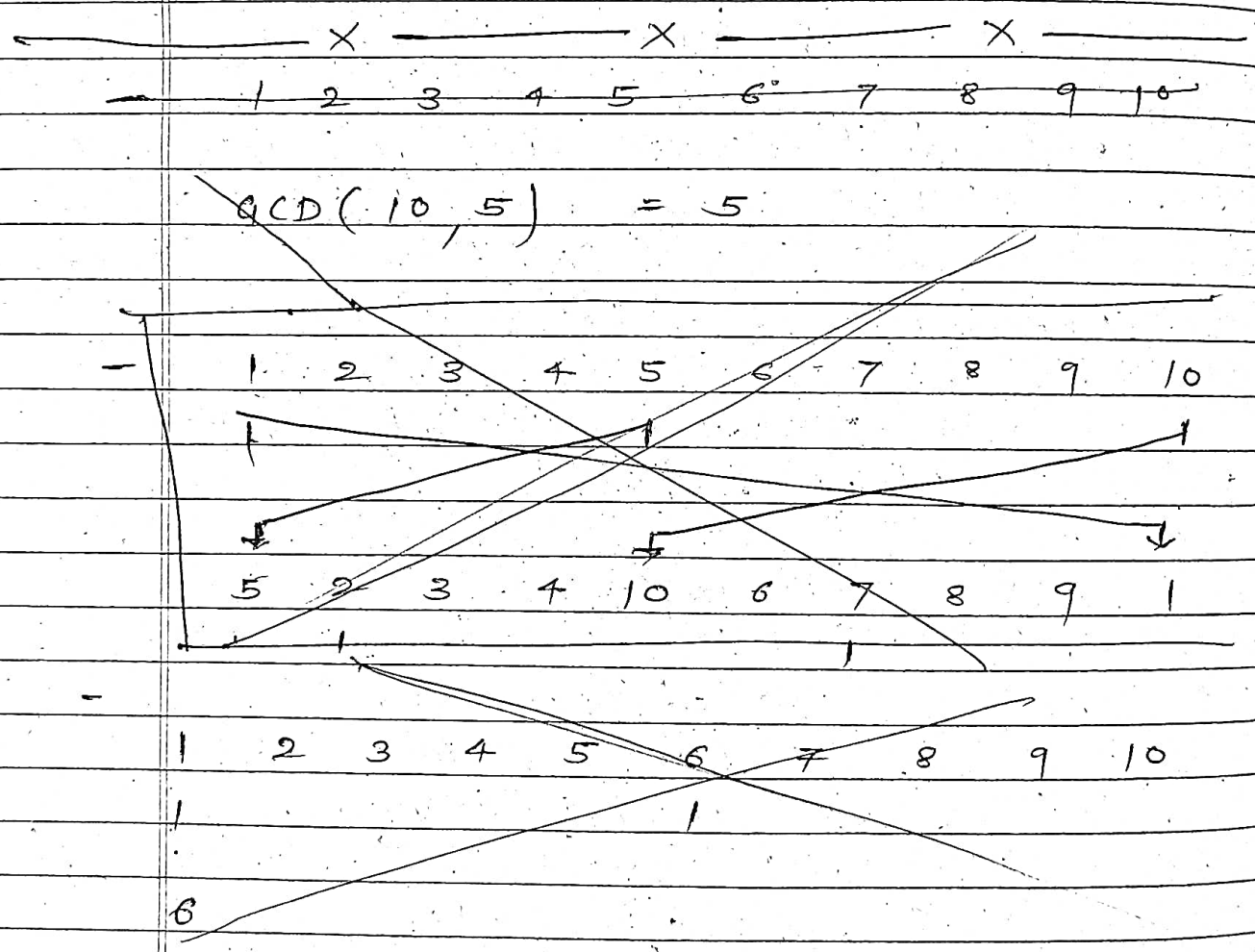
```
void Rotate(int arr[], int d, int n){
    int p=1;

    while( p<=d)
    {
        int last = arr[0];
        For( int=0; i<n-1; i++){
            arr[i] = arr[i+1];
        }
        arr[n-1] = last;
        p++;
    }
```

Approach 3

* A Juggling Algorithm :-

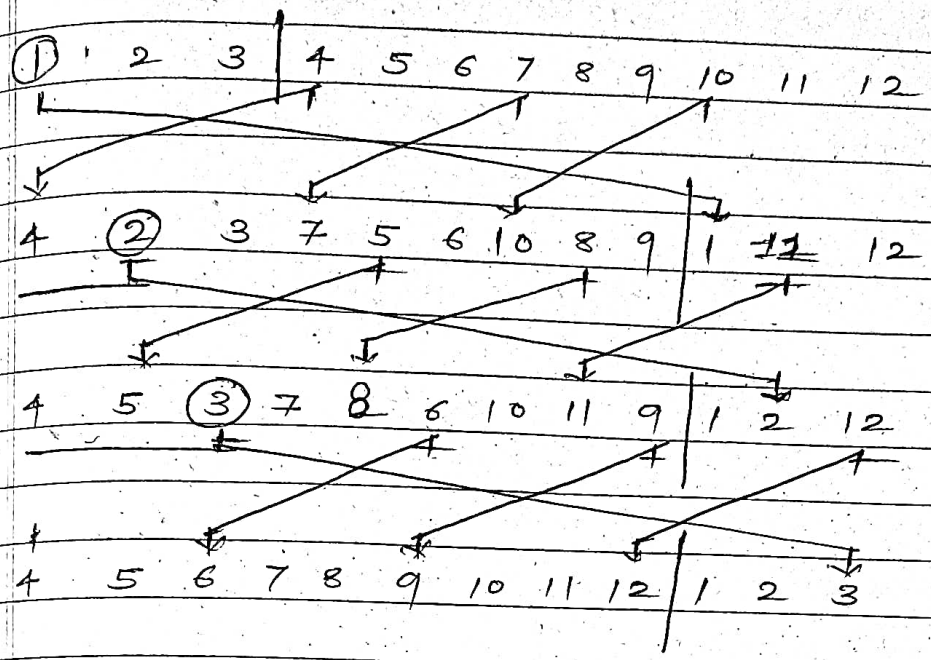
(1) Take a GCD of (array Length, d places)

(2) Find the gap between two elements & move it.



$k = 3$

$GCD(12, 3) \rightarrow 3$.



Time complexity.	: O(N)
Space complexity	: O(1)

Page No.
Date

* Print array after it is right rotated.

Input = {1, 3, 5, 7, 9}, k = 2

Output = {7, 9, 1, 3, 5}

Input = {1, 2, 3, 4, 5}, k = 4.

Output = {2, 3, 4, 5, 1}

* Approach 1:- By 2 formulaes.

```
k = k % n;

For (int i=0; i<n; i++){

    iF (i<k)
    { cout<< a[n+i-k] << " ";
    }
    else
    {
      cout<<( a[i-k]) << " ";
    }
}
```

Dry Run

arr[] = { 1, 2, 3, 4 }, k = 2
0 1 2 3

① i = 0

0 < 2 = a[4 + 0 - 2] [n + i - k]

a[2] = 3

② i = 1

1 < 2 = a[4 + 1 - 2] [n + i - k]

a[3] = 3 4

③ i = 2

2 < 2 ✗

↓

a[2 - 2] = a[0] = 3 4 1

[i - k]

④ i = 3

3 < 2 ✗

↓

a[3 - 2] = a[1] = 3 4 1 2

[i - k]

Time Complexity = O(n)
space Complexity = O(1)

Approach 2):- Reversing the Array.

- Three Rules:-

0 k n

1 2 | 3 4

1) Reverse last k elements

1 2 | 4 3

2) Reverse First k elements

2 1 | 4 3

3) Reverse Entire Array.

3 4 1 2 ✓

```
int k = k % n;
int i, j;

For( i = n-k , j = n-1 ; i<j; i++, j--){

step 1:-    int temp = arr[i];
            arr[i] = arr[j]
            arr[j] = temp;
        }

        For( i = 0, j = n-k-1; i<j; i++, j--){
            int temp = arr[i];
step 2:-    arr[i] = arr[j];
            arr[j] = temp;
        }
```

```
For (int i = 0 ; j = n-1 ; i<j ; i++ ; j--) {
    int temp = arr[i];
    arr[i] = arr[j]
    arr[j] = temp;
}
```

Time complexity = O(N)
Space complexity = O(1)

Approach 3 :- Recursive Approach

```
K = K % n;
        1st part.
reverse ( arr, arr + n - k);
        2nd part
reverse ( arr + n - k, arr + n);
        Entire array.
reverse ( arr, arr + n);
```

```
void rotateArray (int arr[], int n, int k) {
    if (k == 0) { return; }

    int temp = arr[n-1];
    For ( int i = n-1 ; i > 0 ; i--) {
        arr[i] = arr[i-1];
    }
    arr[0] = temp;
    rotate Array ( arr, n, k-1);
}
```

1	2	3	4
0	1	2	3

$k = 2$

$n = 4$

Recursive Method :-

1)

temp = ~~4~~ ~~3~~ arr[3]

For (

① ~~i = 4~~ 3

arr[~~4~~ 3] = arr[~~3~~ 2] ;

1 2 3 3

② i = 2

arr[2] = arr[1]

1 2 2 3

③ i = 1

arr[1] = arr[0]

1 1 2 3

~~④ i = 0~~

arr[0] = 4

4	1	2	3
0	1	2	3

2) rotateArray (arr, 4, 1)

temp = arr[3] = 3

For {

① i = 3

arr[3] = arr[2]

4 1 2 2

Page No.
Date

① i = 2

arr[2] = arr[1]

4 1 1 2

② i = 1

arr[1] = arr[0] 4 4 1 2

arr[0] = temp

3 4 1 2

3) rotateArray(arr, 4, 0)

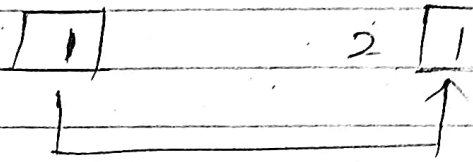
if (k == 0) return;

Time complexity : O(N)
space complexity : O(1)

* Search, Insert and Delete in an Unsorted Array:-

Search operation:-

Key



Array.

searching can be performed by linear traversal.

```
For (int i=0; i<n; i++) {
    if (arr[i] == key) {
        return i;
    }
}
return -1;
```

Time complexity = O(n)
Space complexity = O(1)

Insert At the end.

```
if (n >= capacity)
    return n;
```

Index given by User

arr size

```
arr[n] = key;
return (n+1);
```

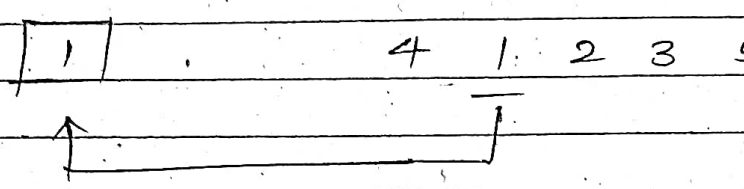
Time complexity : O(1)
Space complexity : O(1)

Insert At any position:-

```
For( int i = n-1; i >= pos ; i--){
    arr[i+1] = arr[i];
}
arr[pos] = x;
```

Time complexity : O(N)
space complexity : O(1)

Delete operation :-



4 2 3 5

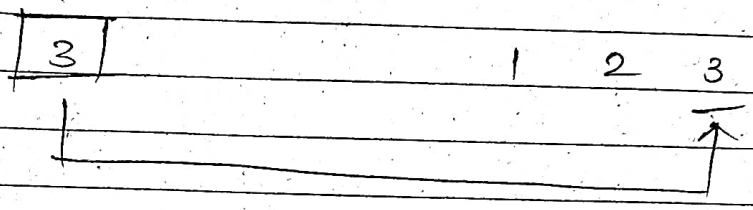
```
int pos = FindElement(arr, n, key);
                Find element index
                By Using linear search

IF(pos==-1)
    Cout << "Element Not Found"
    return n;
}

For(int i=pos; i<n-1; i++){
    arr[i] = arr[i+1];
}                    It will override the pos.
return n-1;
}
```

Time complexity : O(N)
Space complexity : O(1)

* Search, Insert and Delete in an Sorted Array

(1) Search -

We Used Binary search.




```
int binarysearch (int arr[], int low, int high, int key)
{       if(high < low)
            return -1

    int mid = (low + high) / 2;
    if (key == arr[mid])
        return mid;
    if( key > arr[mid]){
        return binarySearch( arr, (mid+1), high,
                                    key);
    }
    return binarySearch( arr, low, (mid-1), key);
}
```

Time complexity : O(log(n))
space complexity : O(log(n))

(2) Insert:-

20 26 Break
40 26

```
For(i=n-1; (i>=0 && arr[i] >key); i--)
{
    arr[i+1] = arr[i];
}

arr[i+1] = key;

return (n+1);
```

Time Complexity: O(N)
Space Complexity: O(1)

(3) Delete:

Just Do

```
int pos = Element's index;
For(int i = pos; i < n-1; i++){
    arr[i] = arr[i+1];
}

return n-1;
```

Time Complexity: O(N)
Space complexity: O(1)