ARRAY REPRESENTATION

Static Vs. Dynamic Array :-

```
Void main ()
{
    int A[5];
```

For C++ :- `int n;`
`Cin >> n;`
`int B[n];`

Not included in this code

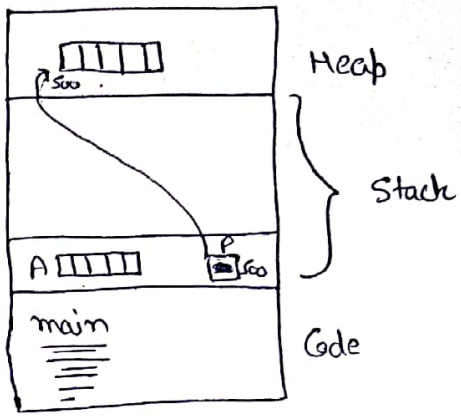
```
    int *P;
```

In C++ → `P = new int [5];`

In C → `P = (int*) malloc (5 * sizeof(int));`

In C++ → `delete [] P;` // method for Deallocation

In C → `free (P);`



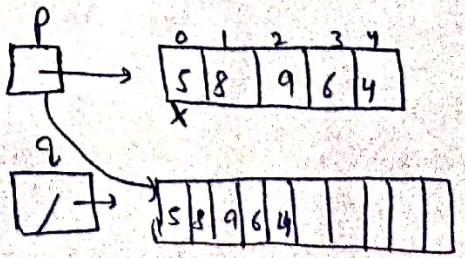
Note :- In C language, the size of an array has to be decided during Compile time but in C++, it will be decided during run time also and both these arrays create in stack.

Once the Array of any size is created it can't be resized, but it is possible in another way with some alternative in heap only not in stack.

```
# include <stdio.h>
# include <stdlib.h>
int main ()
{
    int A[5] = { 2, 4, 7, 8, 10 }
    int * p;
    int i;
    p = (int *) malloc (5 * sizeof(int)); // Allocate memory in heap.
    p[0] = 3;
    p[1] = 5;
    p[2] = 7;
    p[3] = 9;
    p[4] = 11;
    for (i=0; i<5; i++)
    {
        printf ("%d\n", A[i]);
    }
    for (i = 0; i<5 ; i++)
    {
        printf ("%d\n", p[i]);
    }
}
```

- **Increasing Array Size** :- Take another pointer of required size

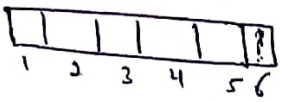
```
int main()
{   int *p = new int [5];
    int *q = new int [10];
    for (i=0 ; i< 5; i++)
    {   q[i] = p[i];
        delete []p;
```



```
    P=q;
  }
}
  q=NULL;
```

Note :- Why the array size cant be grown?

A- Bcz Memory for the Array should be Contigous.



It means, if we want to increase to 6, than we can't bcz we don't know that this address is contained by some value or not. So we take another larger array & shift that.

Code :-

```
int main ()
{
    int *P; int *Q;
    int i;
    P = (int *) malloc (5* sizeof (int));
    P[0] = 3; P[1] = 5; P[2] = 7; P[3] = 9; P[4] = 11;
    q = (int *) malloc (10* sizeof (int));
    for (i=0; i<5; i++)
        {q[i] = P[i];
        }
        free (P);
        P = q;
        q = NULL;
        for (i=0; i<5; i++)
            printf ("%d \n", P[i]); // 3,5,7,9,11
}
```

Code for increasing Size of array.

If we take for(i=0;i<10;i++) it will take some garbage values but array size increase.

✱ Representation of 2-D Array :-

① Simple Declaration :- `int A[3][4] = { {1,2,3,4}, {2,9,6,8}, {4,5,6,8} }`

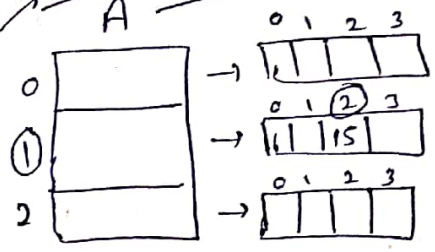
② Array of Pointers :-

(Array of Array of Arrays)

```
int * A[3];
A[0] = new int[4];
A[1] = new int[4];
A[2] = new int[4];
```

Array of size 4

A → it will present in stack



`A[1][2] = 15;`

We will create memory in ~~[redacted]~~ Heap

③ Double Pointers | Pointers to Pointers :-

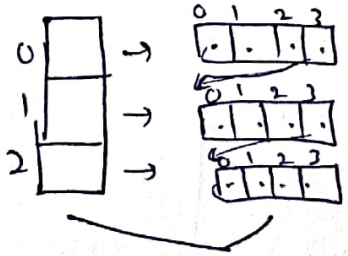
```
int ** A;
A = new int*[3]
```

A → it will in stack

↓

Array of Pointers of type integer

```
A[0] = new int[4];
A[1] = new int[4];
A[2] = new int[4];
```



it will created in heap

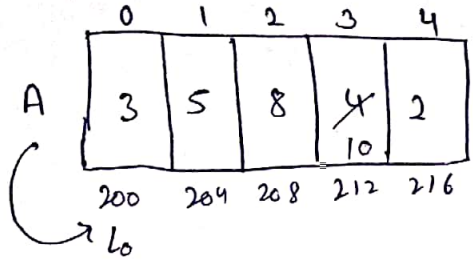
For Accessing 2-D Array :- Two nested for loops are used.

```
for (i=0; i<3; i++)
  { for (j=0; j<4; j++)
      {
      }
  }
```

```
int main ( )
{
    int A [3] [4] = { {1,2,3,4}, {2,4,6,8}, {1,3,5,7} };

    int * B [3];
    int i, j;
      B [0] = (int *) malloc (4 * Size of (int));
      B [1] = (int *) malloc (4 * Size of (int));
      B [2] = (int *) malloc (4 * Size of (int));

    int * * C;

      C = (int * *) malloc (3 * Size of (int *));
      C[0] = (int *) malloc (4 * Size of (int));
      C[1] = (int *) malloc (4 * Size of (int));
      C[2] = (int *) malloc (4 * Size of (int));

    for (i=0; i < 3; i++)
      {
        for (j = 0; j<4; j++)
          { printf ("%d", A[i] [j]);
          }
          printf ("\n");
      }

for (i=0; i<3; i++)
  { for (j=0; j <4; j++)
    {
      printf ("%d", B[i][j]);
    }
    printf ("\n");
  }

for (i=0; i<3; i++)
  {
    for (j=0; j<4; j++)
      {
        printf ("%d", C[i][j]);
      }
      printf ("\n");
  }
}  → // main
```

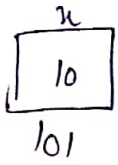
Array Representation By Compiler :- (1) 1-D ARRAY

int A[5] = {3, 5, 8, 4, 2}



Basically, Machine Code should prefer the location with their addresses as not by name, so Compiler Convert name to address.

int x;



To Access, A[3] = 10;

Addr(A[3]) = 200 + 3 * 4 → each integer takes 4 bytes
= 212

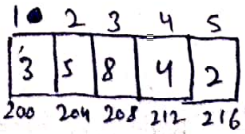
→ Lo + 3 * Size of integer

Addr(A[i]) = Lo + i * w → Size of Data type

Base Address, Index

This is formula for Converting any index to a address, To obtain address our program knows base address and this is a relative address based on base address.

Note :- In some Compiler, index starts from 1- , in this case



Addr(A[3]) = 200 + (3-1) * 4
= 208

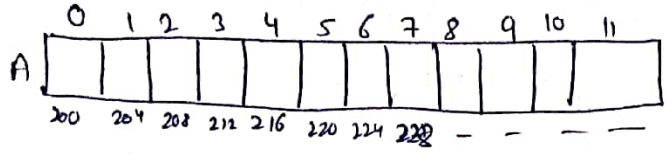
Addr(A[i]) ⇒ Lo + (i-1) * w

In C/C++ we have index starting from 0, bcz index starting from 1 has additional one arithmetic operator and then it will be really slow and affects the time taken by Program

2) 2-D ARRAY :-

During Execution actual memory of 2-D Array is linear

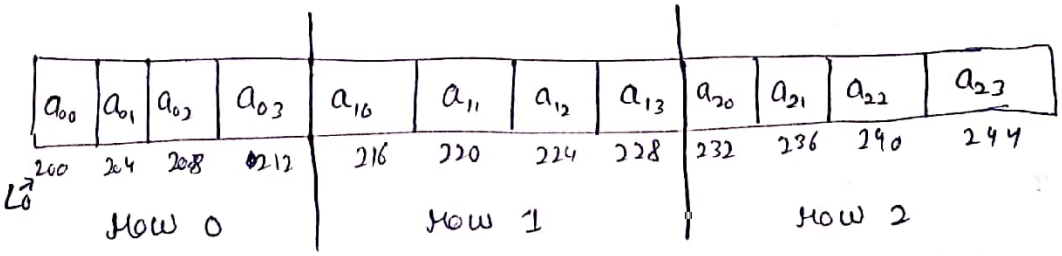
int A[3][4];



A	0	1	2	3
0	a_{00}	a_{01}	a_{02}	a_{03}
1	a_{10}	a_{11}	a_{12}	a_{13}
2	a_{20}	a_{21}	a_{22}	a_{23}

(i) Row - Major Mapping
(ii) Colum - Major Mapping
} → Representation way of 2-D Array

⇒ **Row Major Mapping :-**



To Access A[i][j] = 10;

Addr (A[i][j]) = 200 + [1 * 4 + 2] * 4 = 224

Addr (A[2][3]) = 200 + [2 * 4 + 3] * 4 = 244

Addr (A[i][j]) = L_0 + [i * n + j] * w → Size of integer

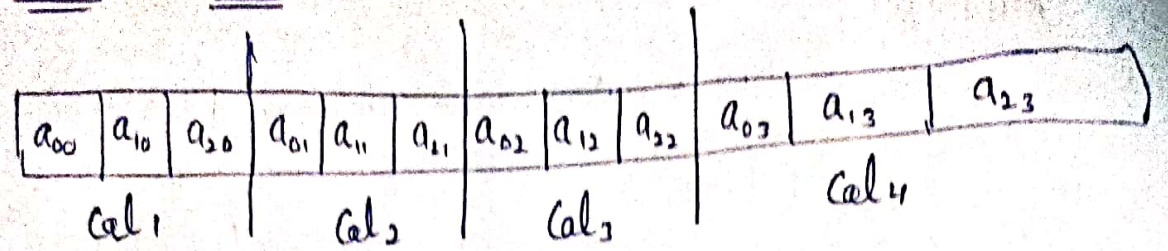
- L_0 ↓ Base Address
- i ↓ Leaving no. of rows
- n ↓ No. of Columns
- j ↓ Elements move further

If a language allows index start from '1' then formula will be

Addr (A[i][j]) = L_0 + [(i-1) * n + (j-1)] * w

but in this there are two additional operators, that's why we use index starting from '0' in C/C++ bcz due to more arithmetic operators program will take more time.

⇒ Column - Major Mapping :-



To Access A[1][2] = 10;

Addr (A[1][2]) = 200 + [2 * 3 + 1] * 4 = 228

Addr (A[1][2]) = 200 + [3 * 3 + 1] * 4 = 240

Addr (A[i][j]) = L_0 + [j * m + i] * w

Note :- In Row major Addr (A[i][j]) = L_0 + [i * n + j] * w

In Column major Addr (A[i][j]) = L_0 + [j * m + i] * w

Both the formulas have same no. of arithmetic operations so, Compiler can use any of them but mainly C/C++ uses Row major formula.

③ For 4-D ARRAY :- A[d1][d2][d3][d4];

(i) Row Major :-

$$Addr(A[i_1][i_2][i_3][i_4]) = L_0 + [i_1*d_2*d_3*d_4 + i_2*d_3*d_4 + i_3*d_4 + i_4]*\omega$$

(ii) Column :- $$Addr(A[i_1][i_2][i_3][i_4]) = L_0 + [i_4*d_1*d_2*d_3 + i_3*d_1*d_2 + i_2*d_1 + i_1]*\omega$$

④ For N-D ARRAY :-

General formula Row Major :- $$L_0 + \sum_{p=1}^{n} \left[i_p * \prod_{q=p+1}^{n} d_q \right] * \omega$$

Here, Σ → for Sum

Π → for Product

General formula Column major :- $$L_0 + \sum_{p=n}^{1} \left[i_p * \prod_{q=n-1}^{1} d_q \right] * \omega$$

- Analysis on Row Major formula :- Now, we know, In 4-D Array

$$Addr(A[i_1][i_2][i_3][i_4]) = L_0 + [i_1*d_2*d_3*d_4 + i_2*d_3*d_4 + i_3*d_4 + i_4]*\omega$$

(3) (2) (1)

Now, if we see there are no. of multiplication as :-

In 4D ARRAY → 3 + 2 + 1

5D " → 4 + 3 + 2 + 1

nD " → (n-1) + (n-2) + ... + 3 + 2 + 1

$$= \frac{n(n-1)}{2} \Rightarrow O(n^2)$$

So, The time Consuming is so much. Now, to Consume less time, We can Reduce no. of multiplications by Horner's Rule (i.e. By taking Common).

Now, we can write it as

$$\text{Addr}(A[i_1][i_2][i_3][i_4]) = L_0 + [i_1 * d_2 * d_3 * d_4 + i_2 * d_3 * d_4 + i_3 * d_4$$

$$= L_0 + [i_4 + i_3 * d_4 + i_2 * d_3 * d_4 + i_1 * d_2 * d_3 * d_4] * \omega$$

$$= L_0 + [i_4 + d_4 * [i_3 + i_2 * d_3 + i_1 * d_2 * d_3]] * \omega$$

$$= L_0 + [i_4 + d_4 * [\quad * i_3 + [d_3 * [i_2 + i_1 * d_2]]]] * \omega$$

So, there are only 3 multiplications for 4-D ARRAY

For 5D → 4

6D → 5

For nD → (n-1) = O(n)

So, by Horner's Rule, we can reduce no. of multiplications and hence time.

⑤ For 3-D ARRAY :-

int A[l][m][n];

Row major :- $\text{Addr}(A[i][j][k]) = L_0 + [i * m * n + j * n + k] * \omega$

Column major :- $\text{Addr}(A[i][j][k]) = L_0 + [k * l * m + j * l + i] * \omega$

Practise

Q-① :- Let A be a two-Dimensional array declared as follows:

A : array [1...10] [1....15] of integer;

Assuming that each integer takes one memory location. The array is stored in row-major order and the first element of the array is stored at location 100, what is the address of A[i][j]?

A-① :-

$$\text{Addr}(A[i][j]) = L_0 + ((i-1) * 15 + (j-1))$$

$$= 100 + (15i - 15 + j - 1)$$

$$= 100 + (15i + j - 16)$$

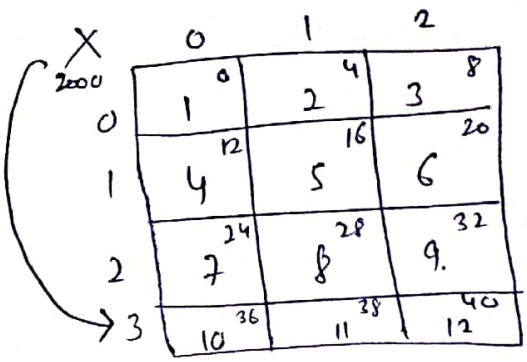
$$\boxed{\text{Addr}(A[i][j]) = 15i + j + 84}$$

Q-② :- What is the output of the following C code? Assume that the Address of x is 2000 (in decimal) and an integer requires four bytes memory.

```
int main()
{ Unsigned int x[4][3] = {{1,2,3}, {4,5,6}, {7,8,9}, {10,11,12}};
  printf("%u %u %u", x+3, *(x+3) * (x+2) +3);
}
```

A-(2) :- `int x[4][3] = { {1,2,3}, {4,5,6}, {7,8,9}, {10,11,12} }`

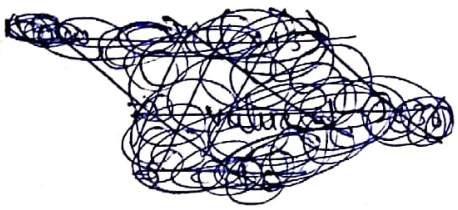
Size of (int) = 4



Now, X+3

⇒ 2000 + 36

= 2036



Now, * (X+3)

⇒ 2036

Now, * (X+2) + 3

⇒ 2024 + 4*3

⇒ 2036 [value = 10]

if we want to Access 8, then

* (* (X + 2) + 1)

↓ ↓

Row Column

for → (* (*+2) + 1) → Address of Data

for → * (X+2)

↓

beginning address of Row 2

V.Imp "For value we should have 2 *".

Q-3 :- If A and B are two matrices, for multiplying two matrices which if the following representation efficient.

(i) A in Row-major and B in Column-major

(ii) " " Column-major " " " Row-major

(iii) Both A & B in Row major

(iv) Independent of Representation.

Q-4 :- If an 3D Array is declared in C Language as follows

? x [?] [?] [?]

Find data type & Dimensions of Array.

If Compiler performs following intermediate operations for finding Address of any location x [i] [j] [k].

$t0 = i * 1024$ — (1)

$t1 = j * 32$ — (2)

$t2 = k * 4$ — (3)

$t3 = t0 + t1$ — (4)

$t4 = t3 + t2$ — (5)

$t5 = x[t4]$ — (6)

(Assume int takes 2 bytes, float takes 4 bytes)

A - (5) :-

When the Compiler performs, it follows Row-major

Now, Addr (x [i] [j] [k]) = Lo + [i * m * n + j * n + k] * w

= Lo + i * m * n * w + j * n * w + k * w

Type x [l] [m] [n]

Now, Comparing with (3), we get ; k * w = k * 4

$\therefore w = 4$

Now, with (2), j * n * w = j * 32

n * w = 32

$\therefore n = 8$

Now, with ①, $m * n * w = 1024$

$m * 32 = 1024$

$$\boxed{m = 32}$$

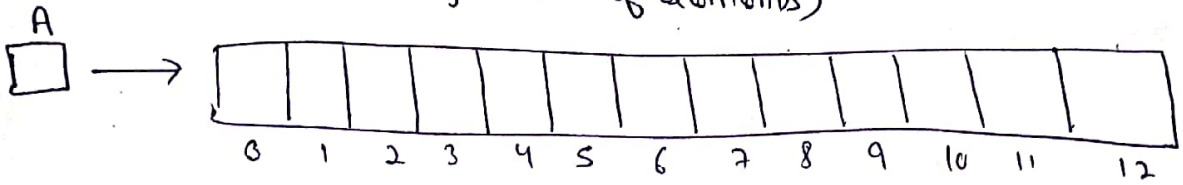
if $w = 4$, Data type = float

$$\boxed{\therefore ?x\,[1][m][n] = \text{float}\ x\,[12][32][8]}$$

ARRAY ADT

Data Representation and Set of operations on Array is known as Array ADT (i.e. Array as an Abstract Data type).

Data :-
1. Array space
2. Size
3. Length (No. of elements)



1. `int A[10];` → mention size first
2. `int * A;` → Array of required size
   `A = new int [size];`

- Now, the length = 0, if there are some elements in an array then length can be determined according to it.

Set of operations :- Display(), Add(x) | Append(x), Insert(index, x), Delete(index), Search(x), Get(index), Set(index, x), Max() | Min(), Reverse(), Shift() | Rotate()

★ Display operation :- Demo - for ADT

```
Struct Array
{
    int *A;
    int size;
    int length;
};

int main()
{
    int n, i
    Struct Array arr;
    printf("Enter the size of an array : \n");
    scanf("%d", &arr.size);
    arr.A = (int *) malloc (arr.size * sizeof(int));
    arr.length = 0;
    printf("Enter no. of elements : \n");
    scanf("%d", &n);
    for(i=0; i<n; i++)
    {
        scanf("%d", &arr.A[i]);
    }
    arr.length = n;
    Void Display (struct Array arr);
    Display(arr);
}
```

```
Void Display (struct Array arr)
{
    int i;
    printf ("Enter Elements are \n");
    for (i=0 ; i< arr. length ; i++)
        printf ("%d", arr. A[i]);
}
```

Note :- "Here *A only points to first element in an array in heap."
We can see this during debugging.

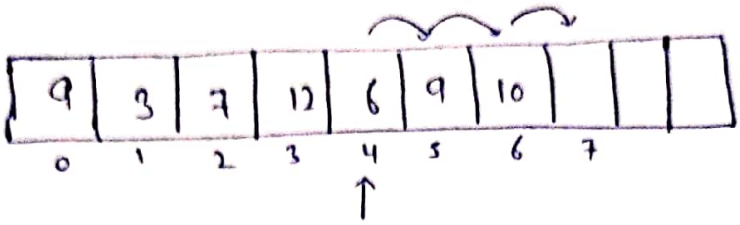
⭐ We can do this by creating Array of fixed size (i.e. not in heap)

```
Struct Array
{
    int A[20];
    int size;
    int length;
};
int main ()
{
    Struct Array arr = { {2,3,4,5,6}, 20, 5};
                          Declaration   size  length
    Display (arr);
}
Void Display (struct Array arr)
{ int i;
    printf ("Enter Elements are \n");
    for (i=0 ; i< arr. length ; i++)
        printf ("%d", arr. A[i]);
}
```

★ Add (x) | Append (x) :- Adding a new element at the end of an Array. (i.e. next free space). ★

```
Struct Array
{
  int A[10];
  int Size;
  int length;
};
int main ( )
{
   struct Array arr = { {2,3,4,5,6}, 10, 5};

    APPend (& arr, 10);
}  Display (arr);
Void APPend ( struct Array * arr , int x)
   {
     if ( arr → length < arr → Size)

         arr → A [arr → length ++] = x ;
   }
Void Display ( struct Array arr)
   { int i;
     printf (" Elements are : \n");
     for (i = 0 ; i < arr. length; i++)
        { printf ("%d", arr. a[i]);
   }
```

★ **Insertion in an Array** :- Inserting element at given index, in this elements are shifted to get free space where the element has to be inserted



Insert (4, 15) ⇒
index x

```
for (i = length ; i > index ; i--)
    A[i] = A[i-1]
A[index] = x;
length++;
```

Time Complexity :- Depends upon where or on what index we insert.

So, Minimum Time :- O(1)

Maximum Time :- O(n)

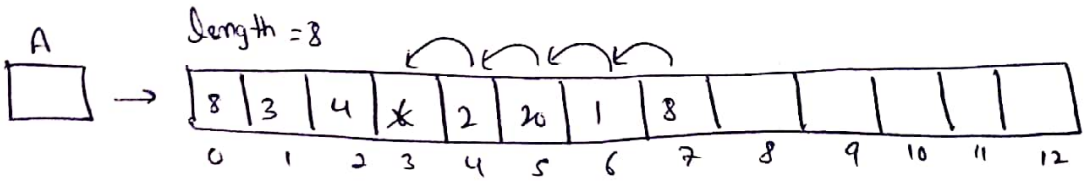
Code :-

```
Struct Array
{
    int A[10];
    int size;
    int length;
};
```

```
int main ( )
{
    Struct Array arr = {{2,3,4,5,6},10,5}
    Insert (&arr, 3, 10);
    Display (arr);
}

Void Insert (struct Array *arr, int index, int x)
{
    int i;
    if (index >= 0 && index <= arr -> length) // Condition
    {
        for (i = arr -> length ; i > index ; i--)
        {
            arr -> A[i] = arr -> A[i-1];
        }
        arr -> A[index] = x;
        arr -> length ++;
    }
}

Void Display ( struct Array arr)
{
    int i;
    printf (" Elements are : \n");
    for (i=0 ; i < arr.length ; i++)
    {
        printf ("%d", a[i]);
    }
}
```

Deletion in an Array :- Delete an element from an Array.

Now, when we delete an element from an Array, then that space will be vacant but we cant leave it vacant as otherwise we have to check Every time in an array, that the space b/w elements is vacant or not. So we will shift them.



Delete(3) ← index

```
x = A[index];
for (i=index ; i<length-1 ; i++)
  {
    A[i] = A[i+1];
  }
Length--;
```

Time Complexity :- Minimum :- O(1)

Maximum :- O(n)

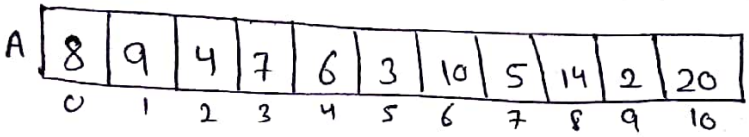
Code :-

```
Struct Array
{
  int A[10];
  int size;
  int length;
};
```

```
int main( )
{
    Struct Array arr = { { 2, 3, 4, 5, 6 }, 10, 15 }
    [redacted]
    Delete (&arr, 3);
    Display (arr);
}
int Delete ( Struct Array * arr , int index)
{
    int x = 0;
    int i;
    if ( index >= 0 && index < arr -> length)
    {
        x = arr -> A [index];
        for (i = index ; i < arr -> length - 1 ; i++)
        {
            arr -> A[i] = arr -> A[i+1];
        }
        arr -> length --;
        return x;
    }
    return 0;
}
Void Display (struct Array arr)
{
    int i;
    printf (" Elements are : \n");
    for (i = 0 ; i < arr.length ; i++)
    {
        printf ("%d", arr.a[i]);
    }
}
```

Searching in an Array :-

① Linear Searching :-



Succesful → key = 5

Unsuccesful → key = 12

```
for (i=0; i < length ; i++)
  { if (key == A[i])
      return i;
  }
Return -1; // if index element is not found.
```

Succesful → **Time Complexity** :- Minimum :- $O(1)$ → best Case

Maximum :- $O(n)$ → worst Case

Unsuccesful → **Time Complexity** :- Maximum :- $O(n)$

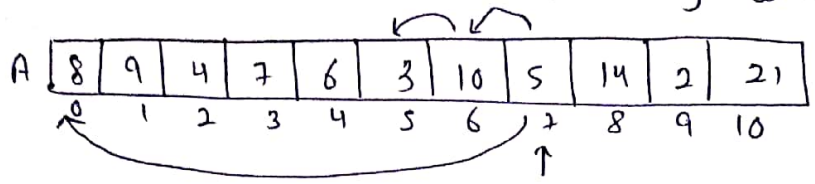
Average Case :- $\frac{\text{Sum of time taken in all possible cases}}{\text{Total no. of cases}}$

$$= \frac{1+2+3+\cdots+n}{n} = \frac{\frac{n(n+1)}{2}}{n} = \frac{n+1}{2}$$

$$= O(n)$$

Note :- Worst Case & Average case take same time approx. (thats why we check worst time).

- **Improvement in Linear Search :-** There is a possibility that we Search a key element again



1st method :-
We can move key element one step forward, this is called as Transposition. So we can search faster next time.

```
for (i=0; i<length; i++)
{ if (key == A[i])
  { swap (A[i], A[i-1]);
  } } return i-1;
}
```

2nd method :- keep key element at index '0' so that when we search it next time, it will take constant time, this is called as move to front / move to Head.

```
for (i=0; i<length; i++)
  { if (key == A[i])
    { swap (A[i], A[0]);
    } return 0;
  }
```

// For Improving Linear Search

① Code for Transposition :- Also for Head / front

```
Struct Array
{
    int A[10];
    int size;
    int length;
};

int main ()
{
    Struct Array arr = { {2,3,4,5,6}, 10, 5};
    int n = Linear Search (&arr, 05);
    Display (arr);
}

void swap (int *x, int *y)
{ int temp;
  temp = *x;
  *x = *y;
  *y = temp;
}

int Linear Search (struct Array *arr, int key)
{
    int i;
    for (i = 0; i < arr->length ; i++)
    {
        if (key == arr->A[i])
        { swap (&arr->A[i], &arr->A[i-1]);
          return i;
        }
    }
    return -1;
}

void Display (struct Array arr)
{ int i;
  printf (" Elements are : \n");
  for (i=0; i < arr.length; i++)
  {
      printf (" %d ", arr.A[i]);
  }
}
```

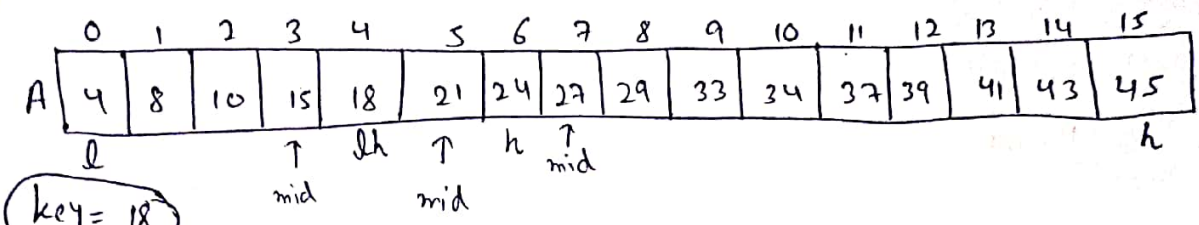
For Head / move to front this will be A[0];

Code :-

```
Struct Array
{
    int A[10];
    int size;
    int length;
};
int main()
{
    Struct Array arr = {{2,4,5,6,8}, 10, 5};
int m = Linear Search (arr, 4)
}
printf("found at index : %d", m);
int Linear Search (struct Array arr, int key)
{
    int i;
    for (i=0; i< arr.length; i++)
    {
        if (key == arr.A[i])
            return i;
    }
    return -1;    // element not found
}
```

② Binary Searching :- The very important Condition is that the Array/List must be Sorted.

Size = 16
Length = 16



key = 18

l	h	mid = $\lfloor \frac{l+h}{2} \rfloor$
0	15	7
	mid − 1	
0	6	3
4	6	5
4	4	4 = found

4 Comparisons

key = 34

l	h	mid
0	15	7
8	15	11
8	10	9
10	10	10 → found

4 Comparisons

key = 25 [Unsuccessful Search]

l	h	mid
0	15	7
0	6	3
4	6	5
6	6	6 → not found
7	6	X

4 Comparisons

Low has become greater than high, it means element is not in the list, So Search is Unsuccessful.

Algorithm :-

```
Algorithm BiSearch (l, h, key)
{
    while (l <= h)
    {
        mid = L (l+h)/2 J
        if (key == A[mid])
            Return mid;
        else if (key < A[mid])
            h = mid - 1;
        else
            l = mid + 1;
    }
    return -1 ; // Not found
}
```

Iterative Version ←

(This is better bcz Recursive function uses stack.)

```
Algorithm RBinary Search (l, h, key)
{
    if (l <= h)
    {
        mid = L(l+h)/2J;
        if (key == A[mid])
            Return mid;
        else if (key < A[mid])
            Return R Binary Search (l, mid-1, key);
        else
            Return R Binary Search (l, mid+1, key);
    }
    Return -1 ; // Not found
}
```

Recursive Version ←

↓

Tail Recursion

Code :-

// Iterative Version

```
Struct Array
{
    int A[10];
    int size;
    int length;
};

int main()
{
    Struct Array arr = { {2,3,4,5,6}, 10, 5};
    printf("index : %d", Binary Search(arr, 5));
}

int Bincary Search (struct Array arr, int key)
{ int l, mid, h;
  l = 0;
  h = arr.length -1;
  while (l <= h)
  {
      mid = floor ((l+h)/2);
      if (key == arr.a[mid])
          return mid;
      else if (key < arr.a[mid])
          h = mid-1;
      else
          l = mid +1;
  }
  return -1; // Not found
}
```

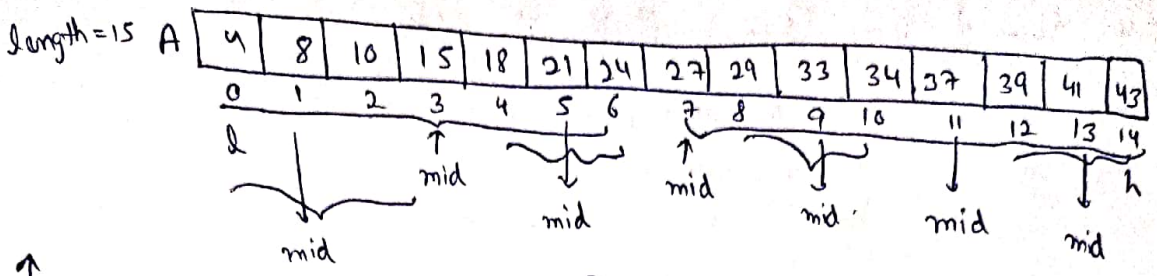
// Recursive function

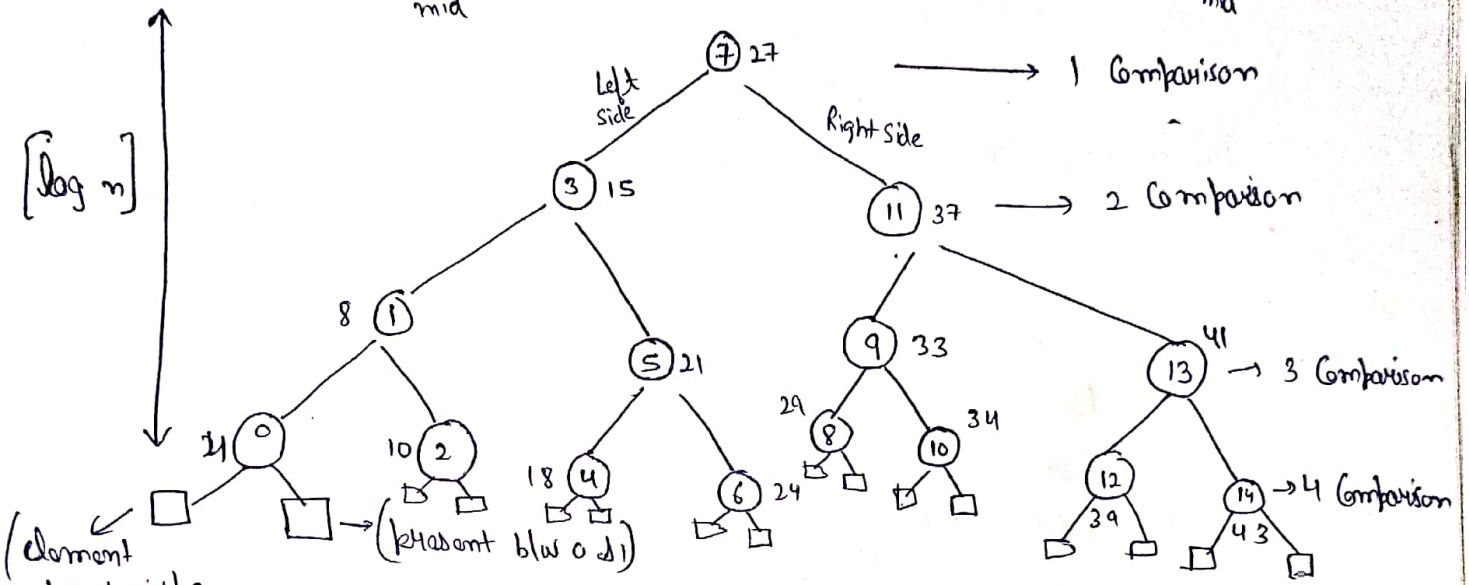
```
Struct Array
{
   int a[10];
   int size;
   int length;
};
int main()
{ Struct Array arr = { {2,3,4,8,9} ,10,5 };
   printf("index : %d", rbinary Search (arr, 0, arr.length-1, 9));
}
int rbinary Search (struct array arr, int l, int h, int key)
{
   int mid;
   if (l<=h)
   {
      mid= floor ((l+h)/2);
      if (key == arr.a[mid])
            return mid;
      else if (key < arr.a[mid])
         return rbinary Search (arr, l, mid-1, key);
      else
      return rbinary Search (arr, mid+1, h, key);
   }
   return -1; // Not found
}
```

Analysis of binary Search :- finding elements in how many Comparisons.

Size = 15

length = 15





So, no. of Comparisons are depending upon height of tree (i.e. $\log n$) and Time Complexity depends upon no. of Comparisons

Time Complexity :- Successful Search
- minimum → $O(1)$ → best Case
- maximum → $O(\log n)$ → worst Case

For Unsuccesful Search :- Always $O(\log n)$

Assume, $n = 16$, how binary Search works :-

$\frac{16}{2}$

$\frac{2}{2}$

$\frac{2}{2}$

$!$ = until it reaches one number.

For this :- $\frac{16}{2^4} = 1$

$2^4 = 16$

$4 = (\log_2 16)$

no. of Comparison ;

Correct formula

So, Time Complexity = Ceil $\leftarrow \lceil \log_2 (n+1) \rceil$

- **Average Case Analysis of Binary Search :-**

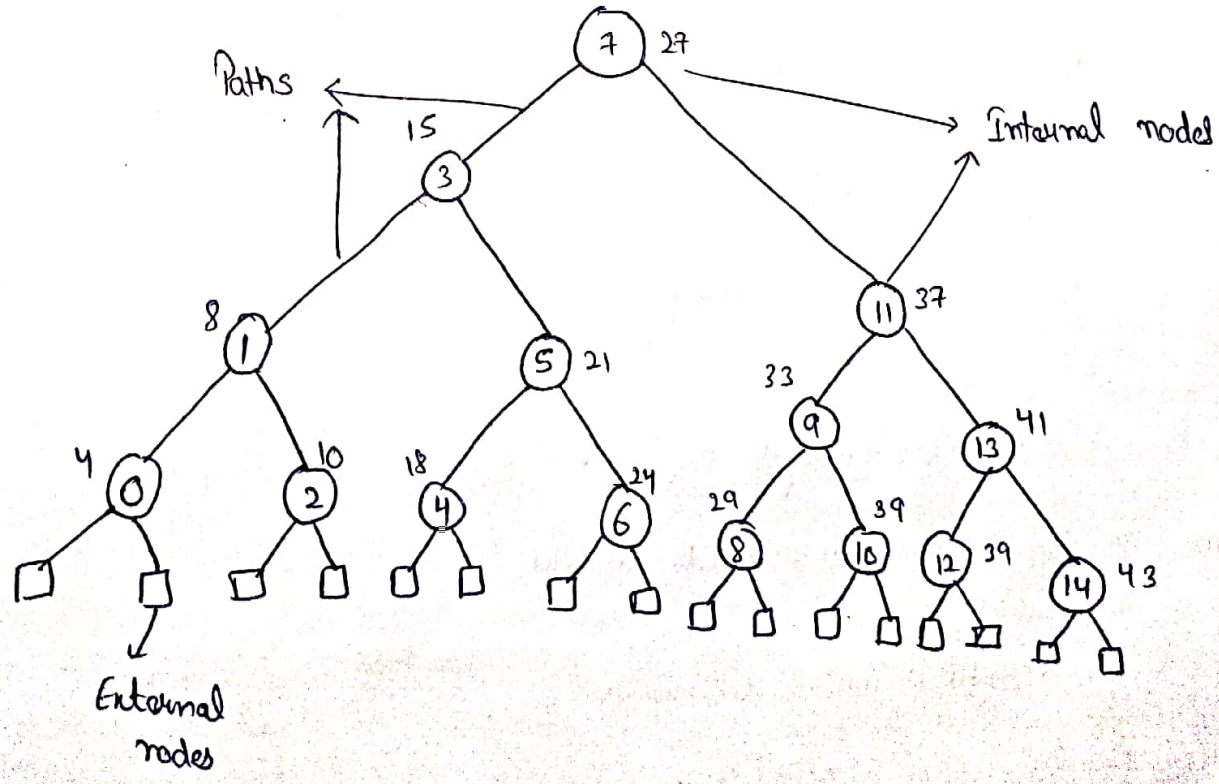
Now, Sum of time taken in all possible cases

$$1 + 1\times2 + 2\times4 + 3\times8$$
$$1 + 1\times2^1 + 2\times2^2 + 3\times2^3$$
$$= \sum_{i=1}^{\log n} i \times 2^i$$

So, Average Case :- $\frac{\sum_{i=1}^{\log n} i \times 2^i}{n} = \frac{\log n \times 2^{\log n}}{n} = \frac{\log n \times n^{\log_2 2}}{n}$

$= \log n$

So, Best Case :- $o(1)$
Worst Case :- $o(\log n)$
Average Case :- $o(\log n)$

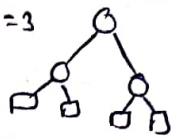


Let, $I \rightarrow$ Sum of the paths of all internal nodes which are representing successful search.

$E \rightarrow$ Sum of the paths of all external nodes which are representing unsuccessful search.

So, Relation b/w them is $\boxed{E = I + 2n}$

Eg :- $n = 3$



$I = 2$; $E = 8$

$\therefore E = 2 + 2(3)$

$E = 8$

No. of External nodes = No. of internal nodes + 1

$$\boxed{e = i + 1}$$

Now, Average Successful time for Search of n elements is

$$\boxed{A_s(n) = 1 + \left(\frac{I}{n}\right)} \rightarrow \text{Average path}$$

For Unsuccessful Search :- $\boxed{A_u(n) = \frac{E}{n+1}}$ Now, we know, all External nodes are at same height

So, we can take, $E = n \log n$

$$\therefore A_u(n) = \frac{n \log n}{n+1} = \log n \text{ (approx.)}$$

Now, $A_s(n) = 1 + \frac{I}{n}$

Also, $E = I + 2n$

$E - 2n = I$

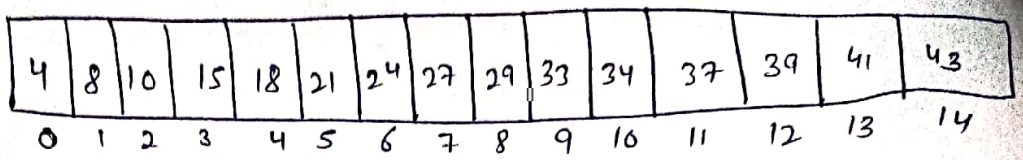
$$\therefore A_s(n) = 1 + \frac{E - 2n}{n}$$

$$A_s(n) = 1 + \frac{E}{n} - 2 \Rightarrow A_s(n) = \frac{E}{n} - 1 \Rightarrow A_s(n) = \frac{n \log n}{n} - 1$$

$$\boxed{A_s(n) = \log n \text{ (approx.)}}$$

★ Functions on Array :-

Size = 15
length = 15



① Get (index) :- Element on that index

```
if (index >=0 && index < length)
    Return A[index];
```

Time Complexity :- O(1)

---

② Set (index, x) → replace value at particular index

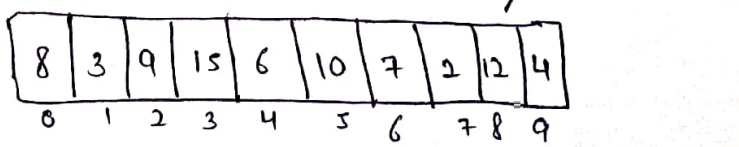
```
if (index >=0 && index < length)
    A[index] = x;
```

Time Complexity = O(1)

---

③

Size = 10
length = 10



max ( ) → maximum

```
max = A[0];
for(i=1; i<Length; i++)
{
    if (A[i] > max)
        max = A[i];
}
Return max;
```

max: 8 ~~9~~ 15

Time Complexity :- O(n)

④ Min()

min = 8 → 3 → 2

8	3	9	15	6	10	7	2	12	4

```
min = A[0];
for (i=0; i< length; i++)
    { if (A[i] < min)
        min = A[i];
    }
Return min;
}
```

Time Complexity :- O(n)

---

⑤ Sum()

```
int Total = 0;
for (i=0; i<length; i++)
    {
      Total = Total + A[i];
    }
Return Total;
}
```

Time Complexity :- O(n)

We can do this recursively, $Sum(arr, n) = \begin{cases} 0 & n<0 \\ Sum(arr, n-1) + A[n] & n >= 0 \end{cases}$

```
int Sum (arr, n)
  { if (n<0)
      Return 0;
    else
      Return Sum(arr, n-1) + A[n];
  }
```

---

⑥ Avg()

```
int Total = 0;
for (i=0; i< length; i++)
    { Total = Total + A[i];
    }
Return Total/n;
}
```

Time Complexity :- O(n)

Code :-

```
struct Array
{
    int A[10];
    int size;
    int length;
};
int Get (struct Array arr, int index)
{
    if (index >=0 && index < arr.length)
        return arr.A[index];
    else
        return -1;
}
Void Set (struct Array *arr ,int index, int x)
{
    if (index >=0 && index < arr->length)
        arr->A[index] = x;
}
int max (struct Array arr)
{
    int max = arr.A[0];
    int i;
    for (i=1; i<arr.length ; i++)
    {
        if (arr.A[i]> max)
            max = arr.A[i];
    }
    return max;
}
```

```
int min (struct Array arr)
{
    int min = arr.A[0];
    int i;
    for (i=1; i<arr.length; i++)
    {
        if (arr.A[i] < min)
            min = arr.A[i];
    }
    return min;
}

int Sum (struct Array arr)
{
    int s=0;
    int i;
    for (i=0; i<arr.length; i++)
    {
        s = s + arr.A[i];
    }
    return s;
}

float Avg (struct Array arr)
{
    return (float) Sum(arr)/arr.length;
}
```

```
Void Display (Struct Array arr)
    {
      int i;
      printf (" Elements are : \n");
      for (i=0 ; i< arr.length; i++)
          {
            printf ("%d", arr.A[i]);
          }
      }

int main ( )
    {
      Struct Array arr = {{2,3,9,16, 18, 21,28, 32, 35}, 10, 9}
      printf (" Element : %d \n", Get (arr, 5));
      Set (&arr, 3, 20);
      Display (arr);
      printf (" maximum : %d \n", max(arr));
      printf (" minimum : %d \n", min (arr));
      printf (" Total : %d \n", Sum (arr));
      printf (" Average : %f \n", Avg (arr));
    }
```

Reverse and Shift An Array :-

Size = 10

Length = 10

A

8	3	9	15	6	10	7	2	12	4
0	1	2	3	4	5	6	7	8	9
									i

Reverse :-

B

4	12	2	7	10	6	15	9	3	9
0	1	2	3	4	5	6	7	8	9
j									

1st method :- We will take an Auxiliary Array and we will Copy elements from Array A to Array B, then Copy back from Array B to Array A.

i.e. $A \rightarrow B$ n

$B \rightarrow A$ n

$2n$

So, Time Complexity :- $O(n)$

$A \rightarrow B$:-

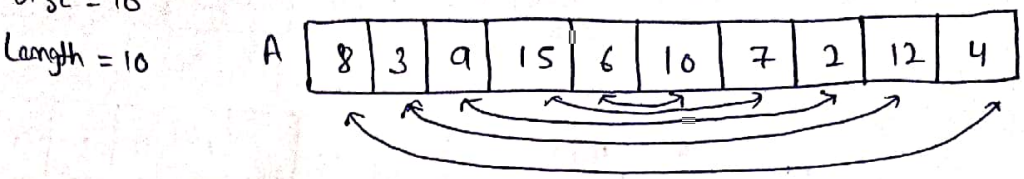
```
for (i = length - 1 , j = 0 ; i >= 0 ; i--, j++)
{
    B[j] = A[i];
}
```

$B \rightarrow A$:-

```
for ( i = 0 ; i < length ; i++)
{
    A[i] = B[i];
}
```

2nd Method :- In this we will swap the elements from both the end of an array.

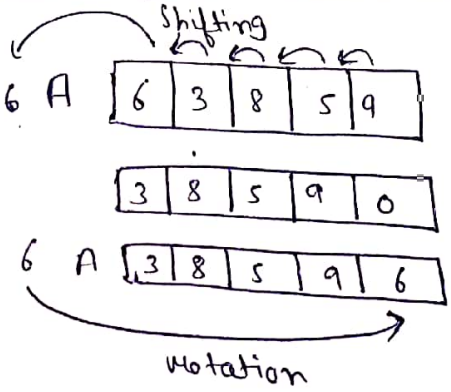
Size = 10
Length = 10



Time Complexity :- O(n)

```
for (i=0, j = length-1 ; i<j ; i++, j--)
{
    temp = A[i];
    A[i] = A[j];
    A[j] = temp;
}
```

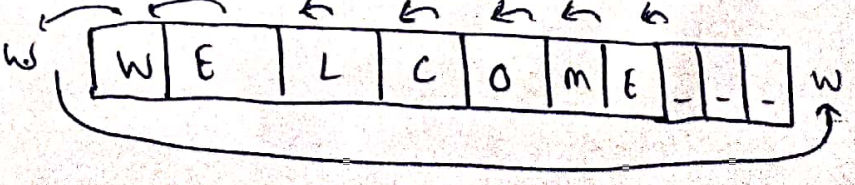
- Left Shift | Rotate :-



Time Complexity :- O(n)

- Right shift | Rotate :- Just Direction is opposite of Left shift | Rotate.

Example / Application :- In Display LED Boards



This is Commonly used in Electronics.

Code for 1st method & 2nd Method :-

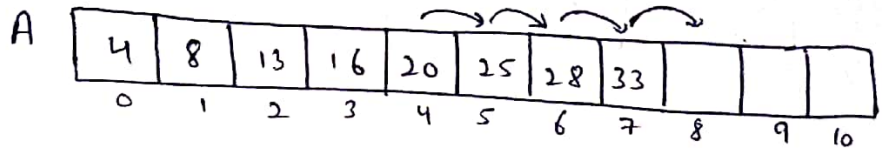
```
Struct Array
{
    int A[10];
    int size;
    int length;
};

int main()
{
    Struct Array arr = {{2,3,9,16,18,21,28,32,35},10,9};
    Reverse (&arr);
    Display (arr);
    Reverse2(&arr);
    Display (arr);
}

Void Swap (int *x, int *y)
{
    int temp = *x;
    *x = *y;
    *y = temp;
}

Void Display (struct Array arr)
{
    int i;
    printf (" Elements are :\n");
    for (i=0; i< arr.length ;i++)
    { printf (" %d", arr.A[i]);
    }
}
```

```
Void Reverse (struct Array * arr)
{
    int * B;
    int i, j;
    B = (int *) malloc (arr -> length * Size of (int));
    for (i = arr -> length - 1, j = 0 ; i >= 0 ; i--, j++)
        B[j] = arr -> A[i];
    for (i = 0 ; i < arr -> length ; i++)
        arr -> A[i] = B[i];
}
```

```
Void Reverse 2 (struct Array * arr)
{
    int i, j;
    for (i = 0, j = arr -> length - 1 ; i < j ; i++, j--)
        {
            Swap (& arr -> A[i], & arr -> A[j]);
        }
}
```

Sorting Array :-

(1) Inserting in sorted Array :-



To insert in sorted, we don't have to find out its position by checking all the members, we can simply check from the last and shift them.

Insert :- 18

```
x = 18;
i = length - 1;
while (A[i] > x)
  {
    A[i+1] = A[i];
      i--;
  }
A[i+1] = x;
```

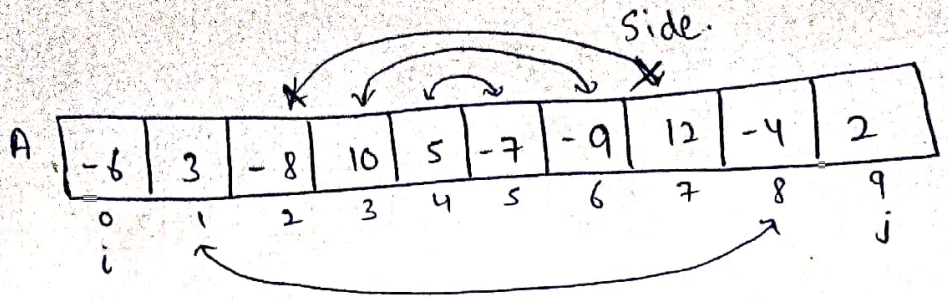
(2) Checking whether the list is sorted or not :-

Check the current no. with next no., if it is smaller than the next no. and this procedure continued until it reaches last index than the list will be sorted if not than it is not sorted.

```
Algorithm is sorted (arr, n) → length
{
  for (i=0 ; i<n-1 ; i++)
    {
      if (A[i] > A[i+1])
          return -1; // Not Sorted
    }
  return 1; // sorted
}
```

Time Complexity :- O(n)

③ -ve on left side | +ve on right side :- Bring all -ve no's on left side and +ve on right side.




```
i=0;
j=length-1;
while(i<j)
 {
   while (A[i]<0)
      { i++;
      }
   while (A[j]≥0)
        { j--;
        }
   if(i<j)
      { swap(A[i], A[j]);
      }
 }
```

Time Complexity :- O(n)

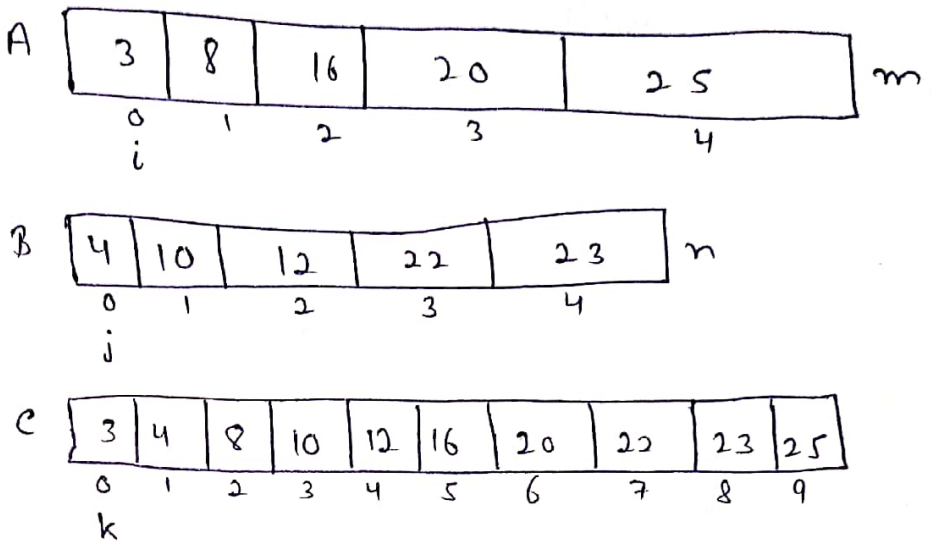
Code :-

```
Struct Array
{
    int A[10];
    int size;
    int length;
};

Void InsertSort (struct Array *arr, int x)
{
   int i = arr->length-1;
   if (arr->length == arr->size)
       return;
   while (arr->A[i] > x && i>=0)
       {
          arr->A[i+1] = arr->A[i];
          i--;
       }
   arr->A[i+1] = x;
   arr->length++;
}
int isSorted (struct Array arr)
{ int i;
   for(i=0; i<arr.length-1; i++)
      { if (arr.A[i] > arr.A[i+1])
              return -1; //Not Sorted
      }
      return 1; // Sorted
}
Void Swap (int *x, int *y)
   { int temp = *x;
      *x = *y;
   }  *y = temp;
```

```
void Rearrange (struct Array * arr1)
{
  int i, j;
  i=0;
  j = arr -> length -1;
  while (i< j)
    {
      while (arr1 -> A[i] <0)
        i++;
      while (arr1 ->A[j] >=0)
        j--;
      if (i<j)
        swap (&arr1 -> A[i] , &arr1 -> A[j];
    }
}
void Display (struct Array arr)
{
  int i;
  printf (" Elements are : \n");
  for (i=0; i< arr. length; i++)
      printf ("%d", arr. A[i]);
}
int main ()
{
  struct Array arr = {{ 2, 3, 5, 10, 15 },10, 5}
  struct Array arr1 = {{ 2,-3, 5, 10, -15, -7 }, 10, 6 }
  void Display (struct Array arr);
  Insert sort ( &arr, 9);
  Display (arr);
  printf ("%d \n", is sorted(arr));
  Void Rearrange (struct Array * arr1);
  Rearrange (& arr1);
  Display (arr1);
}
```

Merging Arrays :-

It can only be done on sorted Arrays. Now we have to combine two array to merge them in third sorted array.




```
i=0;
j=0;
k=0;
while (i<m && j<n)
 { if (A[i] < B[j])
      C[k++] = A[i++];
   else
      C[k++] = B[j++];
 }
for (; i<m; i++)
    C[k++] = A[i];
for (; j<n; j++)
    C[k++] = B[j];
```

Time Complexity :- O(m+n)

↓

Theta

Code :-

```
Struct Array
{
    int A[10];
    int size;
    int length;
};
int main()
{
    Struct Array arr1 = {{2,4,6,10,15,25},10,5};
    Struct Array arr2 = {{3,4,17,18,20},10,5};
Struct Array *merge(Struct Array *arr1, Struct Array *arr2);
    arr3 = merge(&arr1, &arr2);
Void Display(Struct Array arr);
    Display(*arr3);
}
Void Display(Struct Array arr)
{
    int i;
    printf("Elements are: \n");
    for(i=0; i<arr.length; i++)
        printf("%d", arr.a[i]);
}
```

```
Struct Array * merge (struct Array* arr1, struct Array * arr2)
{
    int i, j, k;
    i = j = k = 0;
    Struct Array * arr3 = (struct Array *) malloc (size of (struct array));
    while (i < arr1->length && j < arr2 -> length)
    {
        if (arr1 -> A[i] < arr2 -> A[j])
            arr3 -> A[k++] = arr1 -> A[i++];
        else
            arr3 -> A[k++] = arr2 -> A[j++];
    }
    for ( ; i < arr1 -> length ; i++)
        arr3 -> A [k++] = arr1 -> A[i];
    for ( ; j < arr2 -> length ; j++)
        arr3 -> A[k++] = arr2 -> A[j];
    arr3 -> length = arr1 -> length + arr2 -> length;
    arr3 -> size = 10;
    Return arr3;
}
```

★ **Set operations on Array** :- Mostly they are binary operations (i.e. Require two Arrays)

① **Union** :-

(a) If Array is Unsorted :-

- Copy elements of first Array in third Array.
- Compare all the elements of second Array with elements already present in third Array, if it is present then leave else copy that one.

A

3	5	10	4	6

B

12	4	7	2	5

C

3	5	10	4	6	12	7	2			

Time Complexity :

Copy A[] → $n + (n * n)$ → Compare B[.] and then Copy

$\Rightarrow n + n*n$ [Assume, $m = n$]

$\Rightarrow n + n^2$

$\Rightarrow O(n^2)$ → Time Consuming

(b) If Array is Sorted :- We will use Merging process in such a way that if $A[i] = B[j]$, Copy any one & move i, j & k.

A

3	4	5	6	10

i

B

2	4	5	7	12

j

C

2	3	4	5	6	7	10	12	

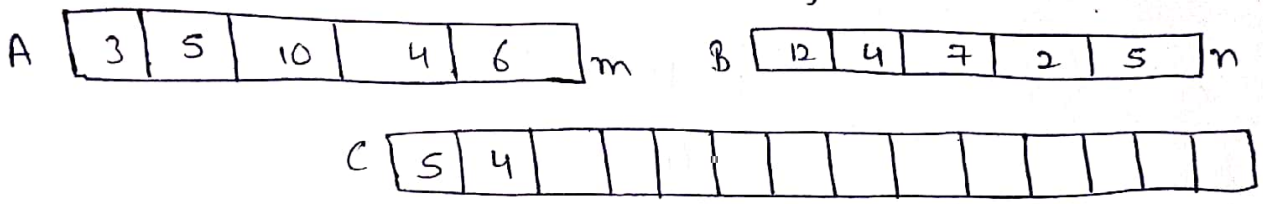
k

Time Complexity :-

$O(m+n)$ → bcz of merging

Intersection :- We have to Copy Common elements in such a way.

(a) If Array in unsorted :- We have to Compare all the elements of A[] with B[] one by one then Copy elements if they are same in third Array.

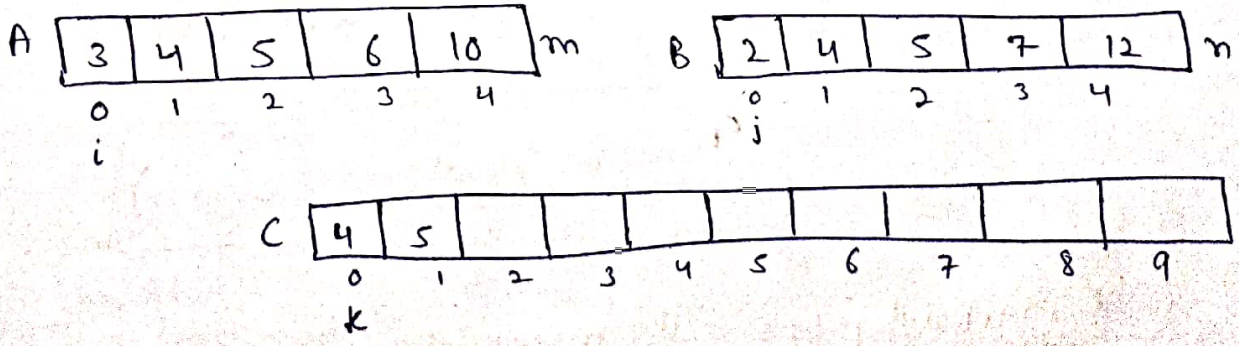


Time Complexity :- $n * m$

$= n * n$ [Assume, $m = n$]

$= O(n^2)$

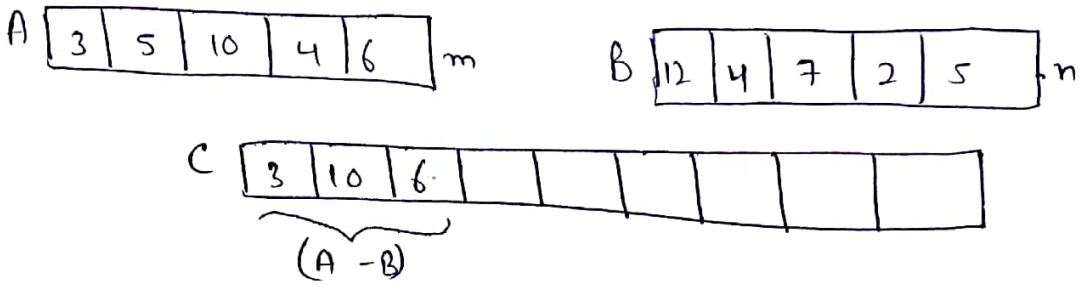
(b) If Array is Sorted :- In this we again use merging process in such a way that don't Copy the element if they are small / big when Comparing and move i & j, if they are equal then Copy & move i, j & k.



"This is not Merging Actually, But the process is Similar to merging."

(3) **Difference** :- A[] - B[] = It means we want all elements A which are not in B. or we can subtract common elements.

(a) **If Array is Sorted** :-



Every element of A[] is compared with every element of B[] if it is not in B[], then copy. otherwise don't copy it.

Time Complexity :- $m * n = n * n = O(n^2)$

(optional)

(b) **If Array is unsorted** :- we will again use merging process in such a way that, Compare elements of A[] if it is greater/smaller A[i], if it is same then move i & j, also at every step move i & j.

Time Complexity :- $O(m+n)$

(4) **Set Membership** :- It means an elements belongs to that set or not, we can do this by simply comparing that element with all elements of the set, if it is there, then it will be a member of that set.

Code :-

```
Struct Array
{
    int A[10];
    int Size;
    int length;
};

int main()
{
    Struct Array arr1 = { {2,9,21,28,35}, 10, 5};
    Struct Array arr2 = { {9,13,21,27,30}, 10, 5};
    Struct Array *arr3;
    Struct Array* Union (Struct Array *arr1, Struct Array *arr2);
    Struct Array * Intersection (Struct Array *arr1, Struct Array *arr2);
    Struct Array * Difference (Struct Array *arr1, Struct Array *arr2);
    Void Display (Struct Array arr);
    arr 3 = Union (&arr1, &arr2);
    Display (*arr3);
    arr 3 = Intersection (&arr1, &arr2);
    Display (*arr 3);
    arr 3 = Difference (&arr1, &arr2);
    Display (*arr 3);
}
```

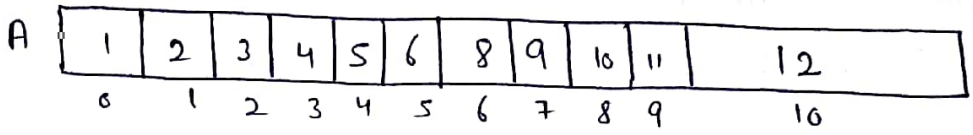
```
Struct Array * union( Struct Array * arr1, Struct Array * arr2)
{
    int i, j, k ;
    i = j = k = 0;
    Struct Array * arr3 = (Struct Array *) malloc( sizeof (Struct Array));
    while ( i < arr1 -> length && j < arr2 -> length)
    {
        if (arr1 -> A[i] < arr2 -> A[j])
            arr3 -> A[k++] = arr1 -> A[i++];
        else if (arr2 -> A[j] < arr1 -> A[i])
            arr3 -> A[k++] = arr2 -> A[i++];
        else
        {
            arr3 -> A[k++] = arr1 -> A[i++];
            j++;
        }
    }
    for( ; i < arr1 -> length; i++)
        arr3 -> A[k++] = arr1 -> A[i];
    for ( ; j < arr2 -> length; j++)
        arr3 -> A[k++] = arr2 -> A[j];
    arr3 -> length = k;
    arr3 -> size = 10;
    return arr3;
}
```

```
Struct Array * Intersection (struct Array * arr1, struct Array * arr2)
{
    int i, j, k;
    i = j = k = 0;

    Struct Array * arr3 = (struct Array *) malloc (size of (struct Array));
    while (i < arr1 -> length && j < arr2 -> length)
    {
        if (arr1 -> A[i] < arr2 -> A[j])
            i++;
        else if (arr2 -> A[j] < arr1 -> A[i])
            j++;
        else if (arr1 -> A[i] == arr3 -> A[i])
        {
            arr3 -> A[k++] = arr1 -> A[i++];
            j++;
        }
    }
    arr3 -> length = k;
    arr3 -> size = 10;
    return arr3;
}

Struct Array * Difference (struct Array * arr1, Struct Array * arr2)
{
    int i, j, k;
    i = j = k = 0;
```

```
Struct Array * arr3 = (struct Array *) malloc (size of (struct Array));
while ( i < arr1 -> length && j < arr2 -> length)
{
    if (arr1 -> A[i] < arr2 -> A[j])
        arr3 -> A[k++] = arr1 -> A[i++];
    else if (arr2 -> A[j] < arr1 -> A[i]).
        j++;
    else
    { i++;
      j++;
    }
for ( ; i< arr1 -> length ; i++)
    arr3 -> A[k++] = arr1 -> A[i];
arr3 -> length = k;
arr3 -> size = 10;
return arr3;
}
```

Note :- Code for Menu Driven Program is in V.S Code.
Containing 374 lines

Student challenge - 1 :- Finding Single Missing Element in an Array. (Sorted Array).



- For Sum of first 'n' natural numbers, starting from 1

```
int main()
{
  int i, sum=0, s, n;
  printf("Enter the value of n: \n");
  scanf("%d", &n);
  int a[n];
  for(i=0; i<n; i++)
  {
    printf("Enter value at a[%d]: \n", i);
    scanf("%d", &a[i]);
  }
  for(i=0; i<n; i++)
  {
    sum = sum + a[i];
  }
  s = (n+1) * (n+2) / 2;
  printf("Missing element: %d \n", s-sum);
}
```

Time Complexity
O(n)

No. Starting from 1 (i.e eg :- 6, 7, 8 ...) :-

```
int main()
{
    int i, n;
    printf("Enter the value of n: \n");
    scanf("%d", &n);
    int a[n];
    for(i=0; i<n; i++)
    {
        printf("Enter the value at a[%d]: \n", i);
        scanf("%d", &a[i]);
    }
    int diff = a[0];
    for(i=0; i<n; i++)
    {
        if(a[i] - i != diff)
        {
            printf("missing element : %d \n", i+diff);
            break;
        }
    }
}
```

Time Complexity
O(n)

Student Challenge - 2 :- Write a program to find Multiple Missing elements in an array. (Sorted Array).

```
int main()
{
int i, n;
printf ("Enter the value of n: \n");
scanf ("%d", &n);
int a[n];
for (i=0 ; i<n; i++)
```

```
{
  printf (" Enter value at a[%d] :\n", i);
  scanf ("%d", &a[i]);
}
int diff = a[0];
for (i=0 ; i<n ; i++)
{ if (a[i]-i != diff)
    {
      while (diff < a[i]-i)
        {
          printf ("missing element : %d \n" i+diff);
          diff++;
        }
    }
}
}
```

Time Complexity
O(n)

Student challange - 3 :- Write a program for finding missing elements in unsorted Array.

```
int main()
{
  int i, n;
  printf ("Enter the value of n: \n");
  Scanf ("%d", &n);
  int A[n];
  for (i=0; i<n; i++)
   {
     printf ("Enter the Value at a[%d]: \n", i);
   } Scanf ("%d", &a[i]);
  int max = a[0];
  for (i=0; i<n; i++)
    { if (max < a[i])
          max = a[i];
    }
  int h[max+1];
  for (i=0; i<max; i++)
    {
      h[i] = 0;
    }
  for (i=0; i<n; i++)
    {
      h[a[i]]++;
    }
  for (i=1; i<=max; i++)
    {
      if (h[i] == 0)
        printf ("Missing Element: %d \n", i);
    }
}
```

Time Complexity
O(n)

Student Challange - 4 :- Write a program for finding duplicates and Count no. of duplicated in sorted Array.

```
int main()
{
    int i, n, j;
    printf("Enter the value of n: \n");
    scanf("%d", &n);
    int a[n];
    for(i=0; i<n; i++)
    {
      printf("Enter the value at a[%d]: \n", i);
      scanf("%d", &a[i]);
    }
    int last duplicate = 0, Count = 0;
    for(i=0; i<n; i++)
    {
      if(a[i] == a[i+1] && a[i] != last duplicate)
        {
          printf("duplicated element: %d \n", a[i]);
          last duplicate = a[i];
        }
    }
    for(i=0; i<n-1; i++)
      {
        if(a[i] == a[i+1])
          {
            j = i+1;
            while(a[j] == a[i])
              j++;
            printf("%d is appearing %d times \n", a[i], j-i);
            i = j-1;
          }
      }
}
```

Time Complexity
O(n)

Student challange - 5 :- Write a program to find duplicates in Sorted Array with alternative method.

```
int main()
{
int n, i
printf("Enter value of n: \n");
scanf("%d", &n);
int A[n];
printf("Enter value at A[%d] : \n", i);
scanf("%d", &A[i]);
int max = A[0];
for(i=0; i<n; i++)
 {
   if(max < A[i])
 }     max = A[i];
int h[max+1];
   for(i=0; i<max+1; i++)
   {
    h[i]=0;
   }
for(i=0; i<n; i++)
   {
    h[A[i]]++;
   }
for(i=0; i<=max; i++)
   {
     if(h[i]>1)
     {
      printf("Duplicated element : %d \n", i);
     }printf("%d appearing %d times \n", i, H[i]);
}    }
```

Time Complexity
O(n)

In Case, if We Consider Space, then don't use hash table bcz, hash table Reluired more space.

Student challang-6 :- Write a program to find Duplicates & no. of Duplicates in Unsorted Arrays.

```
int main()
{
  int i, n, count, i;
  printf ("Enter the value of n : \n");
  scanf ("%d", &n);
  int a[n];
  for (i=0 ; i<n; i++)
  {
    printf ("Enter the value at a[%d] : \n", i);
    scanf ("%d", &a[i]);
  }
for (i=0 ; i<n-1 ; i++)
  {
    count = 1;
    if (a[i] != -1)
    {
      for (j=i+1 ; j<n ; j++)
      {
        if (a[i] == a[j])
        {
          count++;
          a[j] = -1;
        }
      }
      if (count > 1)
      {
        printf ("Duplicated element : %d\n", a[i]);
        printf ("%d appearing %d times \n", a[i], count);
      }
    }
  }
}
```

Time Complexity
$O(n^2)$

So, it is time Consuming.

2nd Method Using Hash table :-

```
int main()
{
    int i, n, count, j;
    printf("Enter the value of n: \n");
    scanf("%d", &n);
    int a[n];
    for(i=0; i<n; i++)
    {
        printf("Enter the value at a[%d] : \n", i);
        scanf("%d", &a[i]);
    }
    int max = a[0];
    for(i=0; i<n; i++)
    {
        if(max < a[i])
        {
            max = a[i];
        }
    int h[max+1];
    for(i=0; i<max+1; i++)
        h[i] = 0;
    for(i=0; i<n; i++)
        h[a[i]]++;
    for(i=0; i<=max; i++)
    {
        if(h[i]>1)
        {
            printf("Duplicated Element: %d \n", i);
            printf("%d appearing %d times \n", i, h[i]);
        }
    }
}
```

Time Complexity
O(n).

but in case, if we consider space then, we can't use hash table, bcz it requires more space.

Student-Challenge 7 :- Write a program for finding a pair with sum k ($a+b=k$). in Unsorted Array.

```
int main()
{
    int i, n, j, k;
    printf("Enter the Value of Sum we want: \n");
    scanf("%d", &k);
    printf("Enter the Value of n: \n");
    scanf("%d", &n);
    int A[n];
    for(i=0; i<n; i++)
    {
        printf("Enter the Value at A[%d]: \n", i);
        scanf("%d", &A[i]);
    }
    for(i=0; i<n-1; i++)
    {
        for(j=i+1; j<n; j++)
        {
            if(A[i]+A[j] == k)
            {
                printf("%d + %d = %d \n", A[i], A[j], k);
            }
        }
    }
}
```

Time Complexity
$O(n^2)$

FASTER METHOD :- Using Hash table

```
int main()
{
    int i, n, j, k;
    printf("Enter the value of Sum you want : \n");
    scanf("%d", &k);
    printf("Enter the value of n : \n");
    scanf("%d", &n);
    int A[n];
    for(i=0; i<n; i++)
    {
      printf("Enter value at A[%d] : \n", i);
    }  scanf("%d", &A[i]);
    int max = A[0];
    for(i=0; i<n; i++)
    { if (max < A[i])
    }      max = A[i];
    int h[max+1];
    for(i=0; i< max+1; i++)
        h[i] = 0;
    for(i=0; i<n; i++)
      { if (k-A[i] < 0)
            Continue;
        else if (h[k-A[i]] != 0) // element is already found
            printf("%d + %d = %d\n", A[i], k-A[i], k);
        h[A[i]]++;
      }
}
```

Time Complexity

O(n)

But in Case, if we Consider Space, then we Can't use hash table

Student - challenge-8 :- Write a program to find Sum of as k (a+b=k) in Sorted Array.

```
int main ( )
{
    int i, n, j, k;
    printf(" Enter the value of Sum you want : \n");
    Scanf (" %d ", &k);
    printf (" Enter the value of N: \n");
    Scanf ("%d", &n);
    int A[n];
    for (i=0 ; i<n; i++)
      { printf (" Enter the value at A[%d]: \n", i);
        scanf (" %d", & A[i]);
      }
    i=0, j=n-1;
    while (i<j)
    { if (A[i] + A[i] = = k)
        {
          printf (" %d + %d = %d\n", A[i], A[j], k);
          i++;
          j--;
        }
      elseif (A[i] + A[j] < k)
          i++;
      else
          j--;
    }
}
```

Time Complexity
O(n)

Student challenge - 9 :- Write a program to find maximum and minimum in single scan

```
int main()
{
  int i, n;
  printf("Enter the value of n : \n");
  scanf("%d", &n);
  int a[n];
  for(i=0; i<n; i++)
  {
    printf("Enter value at A[%d] : \n", i);
    scanf("%d", &A[i]);
  }
  int max = a[0];
  int min = a[0];
  for(i=0; i<n; i++)
  {
    if (a[i] < min) ------(i)
      min = a[i];
    else if (a[i] > max) —(ii)
      max = a[i];
  }
  printf("Maximum value : %d\n", max);
  printf("Minimum value : %d", min);
}
```

- if the list is in descending order (i.e. 10, 8, 3, 2, 1) then it will enter only into (i) Condition
- but if the list is in Ascending order then it will enter into (i) & (ii) Condition.
- So, in Ascending order, no. of Comparisons = n-1 (Best Case)
- So, in Descending order, no. of Comparisons = 2(n-1) (worst Case)

Time Complexity :- O(n) in both Cases

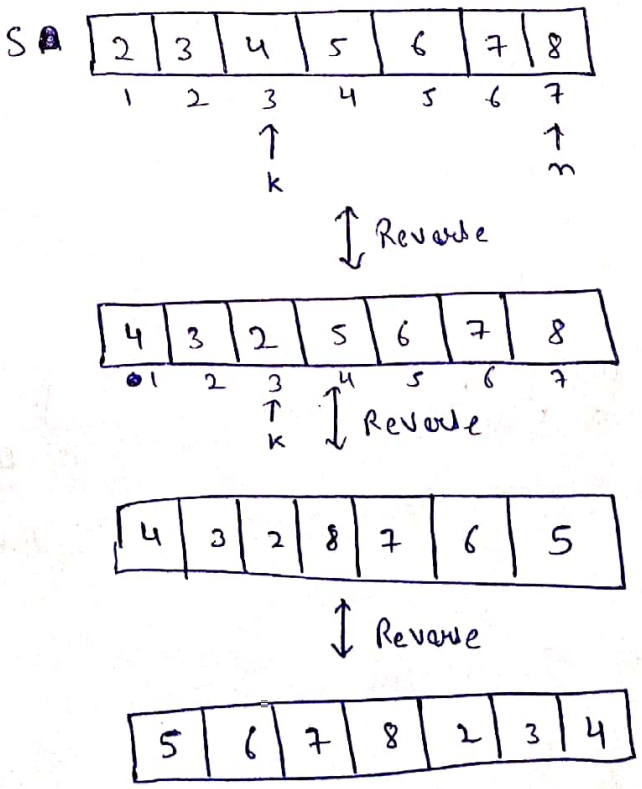
PRACTISE

Question-1 :- Suppose you are given an array $S[1....n]$ and a procedure reverse (S, i, j) which reverse the order of elements in between position i & j (both inclusive).

What does the following the sequence do, where $1 \leq k \leq n$.

```
reverse (S, 1, k);
reverse (S, k+1, n);
reverse (S, 1, n);
```

Answer-1 :-



(a) Rotate S left by k position

(b) Leaves S unchanged

(c) Reverse all elements of S

(d) none of the above.

Question-2

A Program 'P' reads in 500 integers in the range (0, 100) representing the scores of 500 students. If prints the frequency of each score above 50, what would be the best way for 'P' to store the frequencies

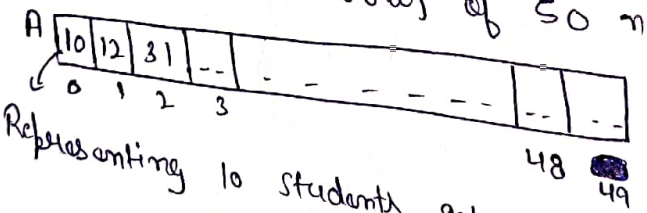
Answer-2

:- Total elements :- 500

Range :- 0-100

frequency of elements > 50

- Now, if the range is '0-100' for 500 students then, there are possibilities that many students get same marks.
- Now, we want frequencies > 50, so, the range will become automatically 50-100.
- Hence, we required an array of 50 numbers.



Representing 10 students got 51 marks (i.e. frequency)

~~(a)~~ An array of 50 Numbers.

(b) An array of 100 Numbers.

(c) An array of 500 Number.

(d) A dynamically allocated array of 550 Numbers.

Question-3 :- Let a be array containing n integers in increasing order. The following algorithm determines whether there are two distinct numbers in the array whose difference is a specified number S>0.

```
i=0; j=1;
while (j<n) {
    if (E) j++;
    else if (a[j]-a[i]==S) break;
    else i++;
}
if (j<n)
    printf ("yes");
```

E = a[j] - a[i] < S

Answer-3 :- Let, S=10

A

2	4	8	9	13	16	19	23	25
0	1	2	3	4	5	6	7	8
i	j							

We have to find E, in this Question, that is Condition for j++;

So, we know in sorted Array, if we want to move j, then a[j] - a[i] < S (must).

So, $\boxed{E = a[j] - a[i] < S}$

Question - 4 What is the time taken to find any element from an array of elements, which is neither maximum nor minimum.

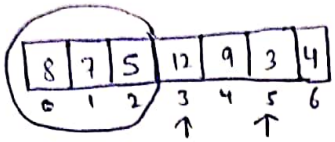
✓ (a) $O(1)$

(b) $O(n)$

(c) $O(n^2)$

(d) $O(\log n)$

Answer - 4 :-



"Simply take first 3 elements, there may be possibility that one is max. & one is min., so, remaining element may be neither minimum nor max. So, it takes constant time."

Question - 5 :- What is the time taken to find the second largest element in a Array.

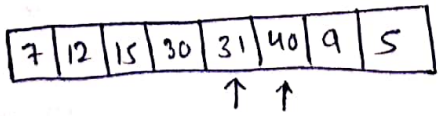
✓ (a) $O(n)$

(b) $O(n \log n)$

(c) $O(n^2)$

(d) $O(n^2 \log n)$

Answer - 5 :-



we have to scan whole array for finding first maximum, then we have to compare it for finding second max., so, it will take time of degree n. (i.e. $O(n)$).