Searching

- Binary Search (Iterative) :-

Only work if array is sorted.

I/P :- arr[] = { 10, 20, 30, 40, 50, 60 }
(indices: 0, 1, 2, 3, 4, 5)
x = 20

O/P :- 1

I/P :- arr[] = { 5, 15, 25 }
(indices: 0, 1, 2)
x = 25

O/P :- 2

I/P :- arr[] = { 5, 10, 15, 25, 35 }
x = 20

O/P :- -1

* Worst case if item is not present in the array.

* Idea :-

10	20	30	40	50	60	70
0	1	2	3	4	5	6
↑						↑
low						High

Steps :-

Compute mid = [(low + high)/2]

Return mid ← case 1 : (arr[mid] == x)

Repeat : high = mid - 1 ← case 2 : (arr[mid] > x)

Repeat : low = mid + 1 ← case 3 : (arr[mid] < x)

⇒
```
int BSearch (int arr[], int n, int x)
{
    int low = 0, high = n-1;
    while (low <= high)
    {
        int mid = (low + high)/2;
        if (arr[mid] == x)
            return mid;
        else if (arr[mid] > x)
            high = mid-1;
        else
            low = mid+1;
    }
    return -1;
}
```

(log n)

(*) Implementation :-

10	20	30	40	50	60
0	1	2	3	4	5

low ↑ (0) — High ↑ (5)

x = 25

(I) mid = (0+5)/2 = 2
Since arr[mid] > x, high = mid-1 = 1

(II) mid = (0+1)/2 = 0
Since arr[mid] < x, low = mid+1 = 1

(III) mid = (1+1)/2 = 1
Since arr[mid] < x, low = mid+1 = 2

return -1

- Binary Search (Recursive)

⇒
```
int BSearch (int arr[], int low, int high, int x)
{
    if (low > high) return -1;
    int mid = (low + high)/2;
    if (arr[mid] == x) return mid;
    else if (arr[mid] > x)
        return BSearch (arr, low, mid-1, x);
    else
        return BSearch (arr, mid+1, high, x);
```

0	1	2	3	4	5	6
10	20	30	40	50	60	70

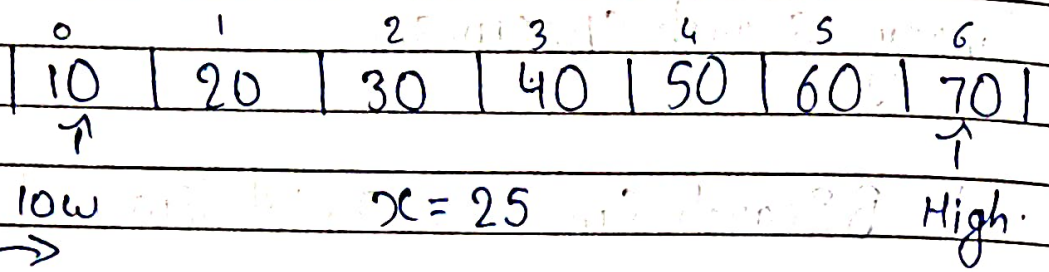
↑ low — x = 20 — ↑ High

BSearch (arr, 0, 6, 20)

→ mid = (0+6)/2 = 3
→ arr[mid] > x
→ BSearch (arr, 0, 2, 20)

→ mid = (0+2)/2 = 1
→ arr[mid] == x, return mid.

⊛ implemetation:-



-1 → BSearch (arr, 0, 6, 25)

→ mid = (0+6)/2 = 3
→ arr[mid] > x
→ BSearch (arr, 0, 2, 25)

-1 → mid = (0+2)/2 = 1
→ arr[mid] < x
→ BSearch (arr, 2, 2, 25)

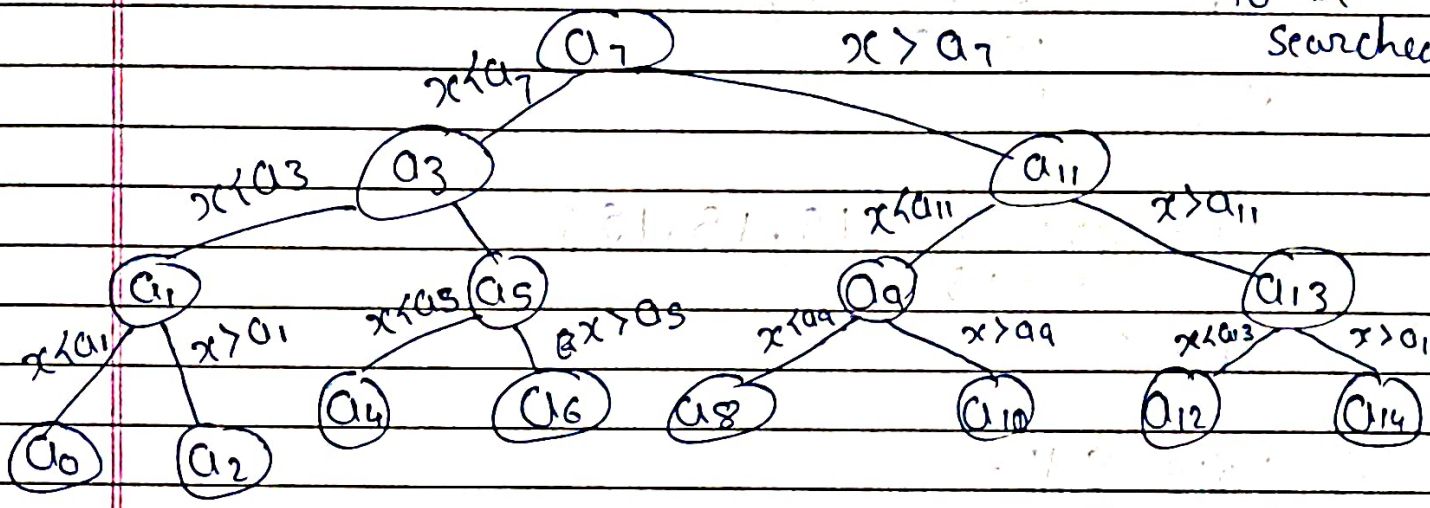
-1 → mid = (2+2)/2 = 2
→ arr[mid] > x
→ BSearch (arr, 2, 1, 25)

condition
Break (low < high)

- **Analysis of Binary Search:-**

a_0	a_1	.....	a_6	a_7	a_8	.....	a_{14}

$x \rightarrow$ Element to be searched



(*) All Possible Execution Paths for Successful Search

Three cases

(1) $x == a_{mid}$
return mid

(2) $x < a_{mid}$
high = mid - 1.

(3) $x > a_{mid}$
low = mid + 1.

- **Index of first Occurrence :- (Array is Sorted)**

I/P :- arr[] = { 1, 10, 10, 10, 20, 20, 40 }
(0, 1, 2, 3, 4, 5, 6)
x = 20

O/P :- 4

I/P :- arr[] = { 10, 20, 30 }
x = 15

O/P :- -1

I/P :- arr[] = { 15, 15, 15 }
x = 15

O/P :- 0

(*) **Naive Sol^n :-**

```
int firstOccurrence (int arr[], int n, int x)
{
    for (int i=0; i<n; i++)
    { if (arr[i] == x)
        return i; }

    return -1;
}
```

O(n) Time
O(1) Aux Space

I/P :- arr[] = { 5, 10, 10, 15, 15 }
x = 15.

O/P :- 3

⊛ Efficient soln:- (Recursive Binary search)

```
int firstOcc(int arr[], int low, int high, int x)
{
    if (low > high)
        return -1;
    int mid = (low + high)/2;
    if (x > arr[mid])
        return firstOcc(arr, mid+1, high, x);
    else if (x < arr[mid])
        return firstOcc(arr, low, mid-1, x);
    else
    { if (mid == 0 || arr[mid-1] != arr[mid])
        return mid;
      else
        return firstOcc(arr, low, mid-1, x);
    }
}
```

Normal Binary search

x = 20

{5, 10, 10, 15, 20, 20, 20}

low = 0, high = 6

firstOcc(arr, 0, 6, 20)

→ mid = 3

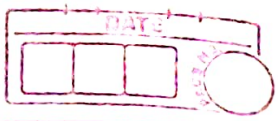
→ firstOcc(arr, 4, 6, 20

→ mid = 5

→ first occ(arr, 4, 4)

→ mid = 4

→ Return 4.



⊛ More efficient :- (Iterative Binary search)

arr[] = {5, 10, 10, 20, 20} x = 10
low = 0, high = 4

I^st Iteration :-
mid = 2
high = 1

II^nd Iteration :-
mid = 0
low = 1

III^rd Iteration
mid = 1
return 1.

⇒
```
int firstOcc (int arr[], int n, int x)
{ int low = 0, high = n-1;
  while (low <= high)
  { int mid = (low + high)/2;
    if (arr[mid] > x)
        high = mid-1;
    else if (arr[mid] < x)
        low = mid+1;
    else
        if (mid == 0 || arr[mid-1] != arr[mid])
            return mid;
        else
            high = mid-1;
  }
  return -1;
}
```

- Index of last Occurrence. (array is sorted)

I/P:- arr[] = {10, 15, 20, 20, 40, 40}
x = 20
O/P:- 3

I/P:- arr[] = {8, 10, 10, 12}
x = 7
O/P:- -1

⊛ Naive Soln:-

To Traverse the array from (n-1) till 0 & when x == arr[i] cout i.

⊛ Efficient Soln:- (Recursive Binary Search)

```
int lastOcc (int arr[], int low, int high, int x, int n)
  { if (low > high) return -1;
    int mid = (low + high)/2;
    if (arr[mid] > x)
          return lastOcc (arr, low, mid-1, n);
    else if (arr[mid] < x)
          return (lastOcc(arr, mid+1, high, n);
    else
      { if(mid == n-1 || arr[mid] != arr[mid+1])
          return mid;
        else
          return lastOcc (arr, mid+1, high, n);
      }
  }
```

{5, 10, 10, 10, 10, 20, 20}
0 1 2 3 4 5 6

x = 10

lastOcc(arr, 0, 6, 10)
→ mid = 3
→ lastOcc(arr, 4, 6, 10)
→ mid = 5
→ lastOcc(arr, 4, 4)
→ mid = 4
→ Return 4.

- Count Occurrence in Sorted array.

I/P:- arr[] = {10, 20, 20, 20, 30, 30}
x = 20
O/P:- 3

I/P:- arr[] = {10, 10, 10, 10, 10}
x = 10
O/P:- 5

I/P:- arr[] = {5, 8, 10}
x = 7
O/P:- 0

* Naive Sol^n :-

```
int COUNT (int arr[], int n, int x)
{
    int count = 0;
    for (int i = 0; i < n; i++)
    {
        if (arr[i] == x)
            { count++; }
    }
    cout << count;
    return 0;
}
```

* Idea is to use two Binary Search.

* Efficient Sol^n :-

```
int countOcc (int arr[], int n, int x)
    {
        int first = firstOcc(arr, n, x);
        if (first == -1)
            return 0;
        else
            return (lastOcc(arr, n, x) - first + 1);
    }
```

Time = O(logn)

* Basic idea is getting the difference of last occurrence & first occurrence & then add one to it.

- **Count 1s in a Sorted Binary Array:-**

I/P:- arr[] = {0, 0, 0, 1, 1, 1, 1}
O/P:- 4

I/P:- arr[] = {1, 1, 1, 1, 1}
O/P:- 5.

I/P:- arr[] = {0, 0, 0, 0}
O/P:- 0.

⊛ **Naive Sol^{n}:-**

⊛ Basic idea is to find first 1 & then find the difference b/w that position & the size of array (i - n)

⊛ **Efficient Sol^{n}:-**

```
int countOnes(int arr[], int n)
{ int low = 0, high = n-1;
  while (low <= high)
  { int mid = (low + high)/2;
    if (arr[mid] == 0)
      low = mid + 1;
    else
    { if mid == 0 || arr[mid-1] != arr[mid]
        return (n - mid);
      else
        high = mid - 1;
    }
  } return 0;
}
```

O(log n) → time
O(1) → space

0 0 1 1 1 1 1

low = 0 , high = 6

I^st Iteration :-

mid = 3

high = 2

II^nd Iteration :-

mid = 1

low = 2

III^rd Iteration

mid = 2

return (7-2)

- **Square Root :-**

- I/P :- x = 4
  O/P :- 2
- I/P :- x = 14
  O/P :- 3
- I/P :- x = 25
  O/P :- 5

⊛ Naive Sol' :-

```
int sqRootFloor (int x)
{
    int i=1;
    while (i*i<=x)
        i++;
    return (i-1);
}
```

x = 9

i = 1

i = 2

i = 3

i = 4

result = (i-1) = 3.

Time :- $\theta(\sqrt{x})$

⊛ Efficient Solution:-

```
int SqRootFloor(int x)
{ int low=1, high=x, ans=-1;
  while (low <= high)
  { int mid = (low+high)/2;
    int mSq = mid*mid;
    if (mSq == x)
      return mid;
    else if (mSq > x)
      high = mid-1;
    else
    { low = mid+1;
      ans = mid; }
  }
  return ans;
}
```

x = 10

low = 1, high = 10

I^st Iteration
- mid = 5
- mSq = 25
- high = 4

II^nd Iteration
- mid = 2
- mSq = 4
- low = 3, ans = 2

III^rd Iteration
- mid = 3
- mSq = 9
- low = 4, ans = 3

IV^th Iteration
- mid = 4
- mSq = 16
- high = 3

• Search in Infinite Sized Array :- (Unbounded Binary Search)

I/P :- arr[] = {1, 10, 15, 20, 40, 50, 90, 100, 120, 500, ...}
x = 100.

O/P :- 7

I/P :- arr[] = {20, 40, 100, 300, ...}
x = 50.

O/P :- -1

⊛ Naive Sol^n :-

```
int Search (int arr[], int x)
  { int i=0;
    while (true)
    { if (arr[i] == x) return i;
      if (arr[i] > x) return -1;
        i++;
    }
  }
```

Time :- O(pos)
Space :- O(1)

⊛ Efficient Sol^n :-

```
int search (int arr[], int x)
{ if (arr[0] == x) return 0;
  int i=1;
  while (arr[i] < x)
      i = i * 2;
  if (arr[i] == x) return i;

  return binarySearch(int arr[]
  return binarySearch (arr, x, i/2+1, i-1);
  }
```

Time :- O(log(pos))

- Search in Sorted Rotated Array:-

1) I/P:- arr[] = {10, 20, 30, 40, 50}
x = 30
O/P:- 2

2) I/P:- arr[] = {100, 200, 300, 10, 20}
x = 40
O/P:- -1

(*) Sol^n:

```
int Search (int arr[], int n, int x)
{ int low = 0 ; int high = n-1;
  while (low <= high)
  { int mid = (low + high)/2;
    if (arr[mid] == x) return mid;
    if (arr[low] <= arr[mid])
    { if (x >= arr[low] && x < arr[mid])
        high = mid - 1;
      else
        low = mid + 1;
    }
    else
    { if (x > arr[mid] && x <= arr[high])
        low = mid + 1;
      else
        high = mid - 1;
    }
  }
  return -1;
}
```

Time :- log(n)
Space :- O(1)

arr[] = {10, 20, 40, 60, 5, 8}
x = 5
Initially: low = 0, high = 5

I^st Iteration:: mid = 2
low = 3

II^nd Iteration :: mid = 4
return 4

- Find a Peak Element:-

- An element is said to be peak when its neighbours elements is smaller than that element.

I/p:- arr[] = {5, 10, 20, 15, 7}
O/p:- 20

I/p :- arr[] = {10, 20, 15, 5, 23, 90}
o/p:- 20 or 90.

I/p:- arr[] = {80, 70, 90}
O/p :- 80 or 90.

(*) Naive Soln^:-

⇒
```
int getPeak (int arr[], int n)
{
    if (n==1) return arr[0];
    if (arr[0] >= arr[1]) return arr[0];
    if (arr[n-1] >= arr[n-2] return arr[n-1];

    for (int i=1; i<n-1; i++)
    { if (arr[i] >= arr[i-1] &&
          arr[i] >= arr[i+1])
          return arr[i];
    }
}
```

⊛ Efficient Sol^{n}:-

```
int getApeak (int arr[], int n)
{ int low=0, high=n-1;
  while (low<=high)
  { int mid = (low+high)/2;
    if((mid==0 || arr[mid-1] <= arr[mid]) &&
       (mid==n-1 || arr[mid+1] <= arr[mid]))
          return mid;
    if (mid>0 && arr[mid-1] >= arr[mid])
          high = mid-1;
    else
          low = mid+1;
  }
  return -1;
}
```

Time:- O(logn)
Space:- O(1)

- **Two Pointer Approach:-**

Find if there is a pair with sum x in a sorted array.

I/P:- arr[] = {2, 5, 8, 12, 30}
x = 17
O/P:- True

I/P:- arr[] = {3, 8, 13, 18}
x = 14
O/P:- False.

⊛ **Naive Sol^n:-**

```
bool isPair(int arr[], int n, int x)
  { for (int i=0; i<n-1; i++)
Time:- O(n²)    { for (int j=i+1; j<n; j++)
Space:- O(1)      { if (arr[i]+arr[j]==x)
                    { return true; }
                  }
                }
                return false;
            }
```

arr[] = {2, 3, 8, 11}, x = 14
i=0: j=1
j=2
j=3
i=1: j=2
j=3
return True (3, 11)

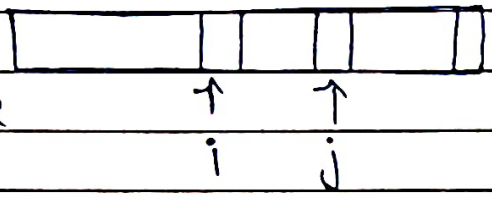
⊛ Idea for efficient Sol^n:-

The array is sorted.

if arr[i]+arr[j]==x
return true
else if(arr[i]+arr[j] > x)
j=j-1;
else
i=i+1;

Let the given sum be x

We move i & j toward each other.



0	1	2	3	4	5	6	7
2	4	8	9	11	12	20	30

x=23

(0 → 1st one, (n-1) → last one)

Comparing (1st one) with (last one) if that sum is greater that the given sum which is x (in this case) then we will ignore "30" (in this case) because all other element will in increasing fuction since array is sorted.

i=0, j=7
(2+30) > 23 : i=0 ; j=6
(2+20) < 23 : i=1 ; j=6
(4+20) > 23 : i=1 ; j=5
(4+12) < 23 : i=2 ; j=5
(8+12) < 23 : i=3 ; j=5
(9+12) < 23 : i=4 ; j=5
(11+12) == 23 : return true.

⊛ Two Pointer Techniques ⊛

```
bool isPair(int arr[], int n, int x)
{ int i=0; int j=n-1;
  while (i<j)
  { if (arr[i] + arr[j] == x)
      return true;
    else if (arr[i]+arr[j]<x)
      i++;
    else
      j--;
  }
  return false;
}
```

Time :- O(n)
Space : O(1)

arr = {2, 5, 8, 12, 30}
x = 17.
Initially :- i=0, j=4
(2+30) > 17 : j=3
(2+12) < 17 : i=1
(5+12) == 17 : return true.

- **Triplet in a Sorted Array :-**

I/P :- arr[] = {2, 3, 4, 8, 9, 20, 40}, x = 32
O/P :- True // Triplet is (4, 8, 20)

I/P :- arr[] = {1, 2, 5, 6}
x = 14
O/P :- false.

⊛ **Naive Sol^n :-**

Time :- $O(n^3)$
Space :- O(1)

```
bool isTriplet(int arr[], int n, int x)
{ for (int i=0; i<n-2; i++)
    for (int j=i+1; j<n-1; j++)
      for (int k=j+1; k<n; k++)
        if (arr[i]+arr[j]+arr[k] == x)
          return true;
  return false;
}
```

arr = [2, 3, 5, 6, 15]
x = 20
i=0 : j=1
k=2
k=3
k=4
return True.

⊛ Idea for efficient Sol^n:-

Hint:- Use Pair Sum Problem as a Subroutine.

① Traverse the array from left to right
② For every element arr[i], check if there is a pair on righ side with sum (x-arr[i])

[2, 3, 4, 8, 9, 20, 40], x=32

2: Search for (32-2) in arr[1:6] } => range (1 to 6)
3: Search for (32-3) in arr[2:6] } => range (2 to 6)
We find a pair with sum 29, we retun true.

=>

```
bool isPair (int arr[], int n, int x, int &i)
{ int i=&i, j=n-1;
   while (i<j)
   { if (arr[i]+arr[j]==x)
        return true;
     else if (arr[i]+arr[j]<x) {i++;}
     else { j--; }
   }
   return false;
}
bool isTriplet (int arr[], int n, int x)
{ for (int i=0; i<n-2; i++)
   {if (isPair (arr, x-arr[i], i+1))
      {return true;} return false;}
   }
}
```

Time:- O(n²)
O(1) (Space)

- **Median of Two Sorted Array:-**

I/P:- arr[] = {10, 20, 30, 40, 50}
arr1[] = {5, 15, 25, 35, 45}

O/P:- 27.5 // {5, 10, 15, 20, 25, 30, 35, 40, 45, 50}

I/P:- a1[] = {1, 2, 3, 4, 5, 6}
a2[] = {10, 20, 30, 40, 50}

O/P:- 6.0 // {1, 2, 3, 4, 5, 6, 10, 2, 30, 40, 50}

⊛ Naive Sol":- O((n1+n2) * log(n1+n2)) → Time.

① Create a array temp[] of size (n1+n2)
② Copy elements of a1[] & a2[] to temp[]
③ Sort temp[]
④ If (n1+n2) is odd, then return middle of temp.
⑤ Else return average of middle two elements.

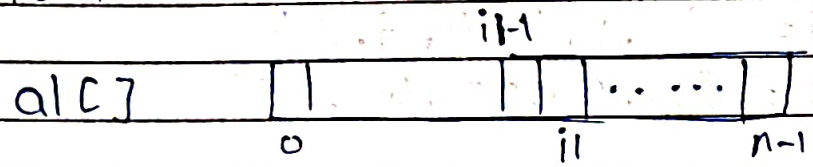
a1[] = {10, 20, 30}, a2 = {5, 15, 25}
temp[] = {10, 20, 30, 5, 15, 25}
After Sorting.
temp[] = {5, 10, 15, 20, 25, 30}
Take average of (15 & 20) // (15+20)/2.
as an output.

⊛ Efficient Sol:- $O(\log n1)$ where $n1 <= n2$

$n1 = 5$, $a1[] = \{10, 20, 30, 40, 50\}$
$i1$

$n2 = 9$, $a2[] = \{5, 15, 25, 35, 45, 55, 65, 75, 85\}$
$i2$

$n1 + n2 = 14$



a1[0 ... (i1-1)]
a2[0 .... (i2-1)]

Left - Half

a1[i1 .... n1-1]
a2[i2 ..... n2-1]

Right - Half

$$i2 = \left\lceil \frac{n1 + n2 + 1}{2} \right\rceil \quad i1 = 7 - 2 = 5$$

(Use Binary search)

⇒
```
double getMed(int a1[], int a2[], int n1, int n2)
{ int begin1 = 0, end1 = n1
  while (begin1 <= end1)
  { int i1 = (begin1 + end1) / 2;
    int i2 = (n1 + n2 + 1) / 2 - i1;
    int min1 = (i1 != n1) ? INT_MAX : a1[i1];

    int max1 = (i1 == 0) ? INT_MIN : a1[i1-1];

    int min2 = (i2 == n2) ? INT_MAX : a2[i2];
    int max2 = (i2 == 0) ? INT_MIN : a2[i2-1];

  if (max1 <= min2 && max2 <= min1)
  { if ((n1+n2) % 2 == 0)
      return ((double) max(max1, max2) +
                  min(min1, min2)) / 2;
    else
      return (double) max(max1, max2);
  }
  else if (max1 > min2) end1 = i1-1;
  else begin1 = i1+1;
} }
```

min1 : Minimum element in right side of a1
max1 : Maximum element in left side of a1
min2 : Minimum element in right side of a2.
max2 : Maximum element in left side of a2.

{10, 20, 30} {5, 15, 25, 35, 45}
↑ i1 ↑ i2

i1 = 1, i2 = (3+5+1)/2 - 1 = 3
min1 = 20, max1 = 10
min2 = 35, max2 = 25

- **Repeating Element :-**

I/P :- arr[] = {0, 2, 1, 3, 2, 2}
O/P :- 2

I/P :- arr[] = {1, 2, 3, 0, 3, 4, 5}
O/P :- 3

I/P :- arr[] = {0, 0}
O/P :- 0

⊛ Array Size, $n \geq 2$
⊛ Only One Element Repeats (Any number of times)
⊛ Also the element from 0 to max(arr) are present.

Therefore $0 \leq max(arr) \leq n-2$.

⊛ Constraints :- O(n) Time.
O(1) Aux Space
No modification to original Array
arr[] = {0, 2, 1, 3, 2, 2}

⊛ **Super naive Sol^n :-** $O(n^2)$ Time & O(1) space.

```
for (int i=0; i<n-1; i++)
    for (int j=i+1; j<n; j++)
        if (arr[i] == arr[j])
            return arr[i];
```

(*) Naive Sol":- O(nlogn) Time & O(1) Aux Space.

1) Sort the Array: arr[] = {0, 1, 2, 2, 2, 3}
2) for (int i=0; i<n-1; i++)
    if (arr[i] == arr[i+1])
        return arr[i];

(*) Efficient Approach:- O(n) time & O(n) space.
arr[] = {0, 2, 1, 3, 2, 2}

1) Create a boolean array of size n.
bool visited[] = {f, f, f, f, f, f}
2) for (int i=0; i<n; i++)
  { if (visited(arr[i]))
      return arr[i];
    visited(arr[i]) = True;
  }

(*) Efficient Approach:- O(n) Time, O(1) space & No changes to original array:-

I/P:- arr[] = {1, 3, 2, 4, 3, 3}
O/P:- 3

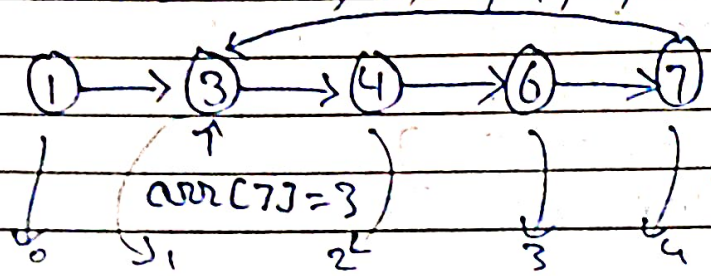
I/P:- arr[] = {1, 1}
O/P:- 1

I/P:- arr[] = {3, 5, 4, 1, 2, 4, 4}
O/P:- 4

* All elements from 1 to max(arr) are present
* 1 ≤ max(arr) ≤ n-1.

Idea:-

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



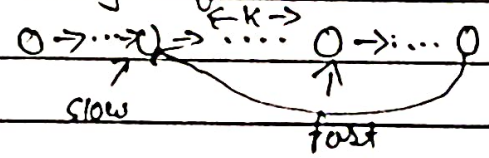
The first element of cycle will always be the repeating element.

Linked list Algorithm (find first Node).

```
⇒ int findRepeating (int arr[], int n)
  { int slow = arr[0], fast = arr[0];
    do {
        slow = arr[slow];
        fast = arr[arr[fast]];
       }
       while (slow != fast)
    slow = arr[0];
    while (slow != fast)
      { slow = arr[slow];
        fast = arr[fast];
      }
      return slow;
  }
```

arr[] = {1,3,2,4,6,5,7}
# Phase I:-
slow=1 , fast=1
slow=3 , fast=4
slow=4 , fast=7
slow=6 , fast=4
slow=7 , fast=7
# Phase II:-
slow=1 , fast=7
slow=3 , fast=3

Working of Phase-I



1) fast will enter the cycle first (on at some time)

2)

- ⊛ Allocate minimum Pages (Naive Method)

I/P :- arr[] = {10, 20, 30, 40} 2
K = 2 ⇒ Divide in two students.
O/P :- 60 // (10,30) & (40,20) max of both o/p :- 60

I/P :- arr[] = {10, 20, 30}
K = 1
O/P :- 60

I/P :- arr[] = {10, 5, 30, 1, 2, 5, 10, 10}.
K = 3
O/P :- 30 // (10, 5) (30) (1, 2, 5, 10, 10)

⊛ Minimize the maximum pages allocated.
⊛ Only contiguous pages can be allocated.

Minimum possible value of maximum pages read by a student.

* Naive Recursive Sol^n :-

$x_0 | x_1 | x_2 | x_3 | x_4 | \cdots | x_{i-1} | x_i | \cdots | x_{n-1} | x_n$

We need to choose (K-1) cuts out of (n-1) cuts shown above.
Total way :- ${}^{n-1}C_{K-1}$

Example :- [10, 20, 30, 40]
K = 2
We need to choose 1 cut out of 3 cut

Pages should we contigeous no middle element should be skipped.

```
int minPages(int arr[], int n, int K)
{ if (K==1)
    return sum(arr, 0, n-1)
  if (n==1)
    return arr[0];
  int res = INF;
  for (int i=1; i<n; i++)
    res = min(res, max(minPages(arr, i, K-1),
                       sum(arr, i, n-1)));
  return res;
}
int sum(int arr[], int b, int e)
{ int s = 0;
  for (int i=b; i<=e; i++)
    s = s + arr[i];
  return s;
}
```

- Allocate minimum Pages (Binary Search)

[10, 20, 10, 30] K=2

Sum of all pages = 10+20+10+30 = 70

Answer will be in range [30, 70]

Maximum value ↑ ↑ Sum

→ $x = \frac{30+70}{2} = 50$

res = 50

→ $x = \frac{30+49}{2} = 39$ ∵ 39 is not a fissible region therefore all below that 38, 37, ... all will also not considered, so we will check another half from 50-70

changed the range from 40-44

$x = \frac{40+49}{2} = 44$, res = 44.

range [40-43] $x = \frac{40+43}{2} = 41$, res = 41

$x = \frac{40+40}{2} = 40$, res = 40

[40-40]

⇒ Code:-

```
int minPages(int arr[], int n, int K)
{ int sum=0, mx=0;
  for(int i=0; i<n; i++)
    { sum += arr[i];
      mx = max(mx, arr[i]);
    }
int low = mx, high = sum, res=0;
while(low<=high)
 { int mid = (low+high)/2;
 if(isFeasible(arr,n,K,mid))
   { res = mid;      // If feasible go to
     high = mid-1;   // the left half.
   }
 else
     low = mid+1; // Else go to right half.
 }
 return res;
}
```

```
bool isFeasible (int arr[], int n, int k,
                 int ans)
{ int req = 1, sum = 0;
  for (int i=0; i<n; i++)
  { if (num + arr[i] > ans)
    { req++;
      sum = arr[i];
    }
    else
      sum += arr[i];
  }
  return (req <= k);
}
```

O(nlog(sum-mx))

arr = [10, 5, 20], k = 2

[20,35] mid = $\frac{20+35}{2}$ = 27, res = 27

[20-26] mid = $\frac{20+26}{2}$ = 23, res = 23

[20-22] mid = $\frac{20+22}{2}$ = 21, res = 21

[20-20] mid = $\frac{20+20}{2}$ = 20, res = 20