Binary Search

I/p: arr[] = [10, 20, 30, 40, 50, 60] (0 1 2 3 4 5)
x = 20
o/p: 1

I/p: arr[] = [5, 15, 25] (0 1 2)
x = 25
o/p: 2

I/p: arr[] = [5, 10, 15, 25, 35] (0 1 2 3 4)
x = 20
o/p: -1

I/p: arr[] = [10, 15] (0 1)
x = 20
o/p: -1

I/p: arr[] = [10, 15] (0 1)
x = 5
o/p: -1

I/p: arr[] = [10, 10] (0 1)
x = 10
o/p: 0 OR 1

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

low = 0, high = 6, x = 60

Steps:
compute mid = ⌊(low+high)/2⌋
case 1 : (arr[mid] == x) → return mid
case 2 : (arr[mid] > x) → Repeat: high = mid-1
case 3 : (arr[mid] < x) → Repeat: low = mid+1

T.C = O(logn)
A.S = O(1)

```
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;
}
```

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

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

Recursive Binary Search:

Binary Search functions on C++ STL

⇒ binary_search() } used directly on a list or container
⇒ upper_bound()
⇒ lower_bound()

① binary_search():

↳ Syntax: binary_search(start_ptr, end_ptr, element)

start_ptr → starting iterator
end_ptr → ending iterator

↳ Returns True (if element is present)
false (otherwise)

on vectors:

```
vector<int> vec;
binary_search(vec.begin(), vec.end(), element);
```

on arrays:

```
int arr[n];
binary_search(arr, arr+n, element);
```

② upper_bound():

↳ To find upper bound of element present in container.
↳ finds the location of an element just greater than a given element in a container.

Syntax: upper_bound(first_itr, last_itr, ele)

Return Value: Returns an iterator pointing to the element just greater than ele.

```
vector<int> v{10,20,30,40,50};
upper_bound(v.begin(), v.end(), 35); // index 3
```

③ lower_bound():

↳ To find lower bound of element present in container.
↳ Syntax: lower_bound(first_itr, last_itr, ele);

Return Value: Return and iterator pointing to the lower bound of the element ele. That is if ele exists in the container, it returns an iterator pointing to ele otherwise it returns an iterator pointing to the element just greater than ele

Recursive Binary Search:-

```
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 bs(arr, low, mid-1, x);
    else
        return bs(arr, mid+1, high, x);
```

T.C = O(logn)

AS = O(logn)

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

low → 0, high → 6

x = 20

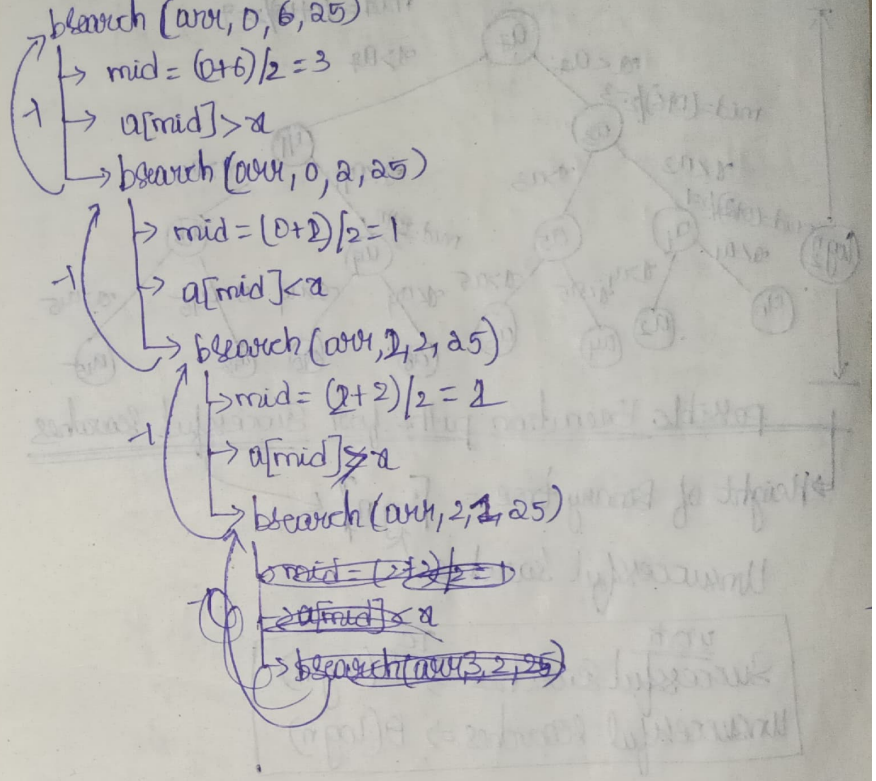
bsearch(arr, 0, 6, 20)
- mid = (0+6)/2 = 3
- arr[mid] > x
- bsearch(arr, 0, 2, 20)
  - mid = (0+2)/2 = 1
  - arr[mid] == x, return mid.

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

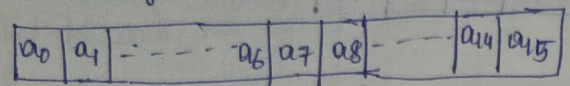
low → 0, high → 6

x = 25

bsearch(arr, 0, 6, 25)
- mid = (0+6)/2 = 3
- a[mid] > x
- bsearch(arr, 0, 2, 25)
  - mid = (0+2)/2 = 1
  - a[mid] < x
  - bsearch(arr, 2, 2, 25)
    - mid = (2+2)/2 = 2
    - a[mid] ≯ x
    - bsearch(arr, 2, 1, 25)
      - ~~mid = (2+1)/2 = 1~~
      - ~~a[mid] < x~~
      - ~~bsearch(arr, 3, 2, 25)~~

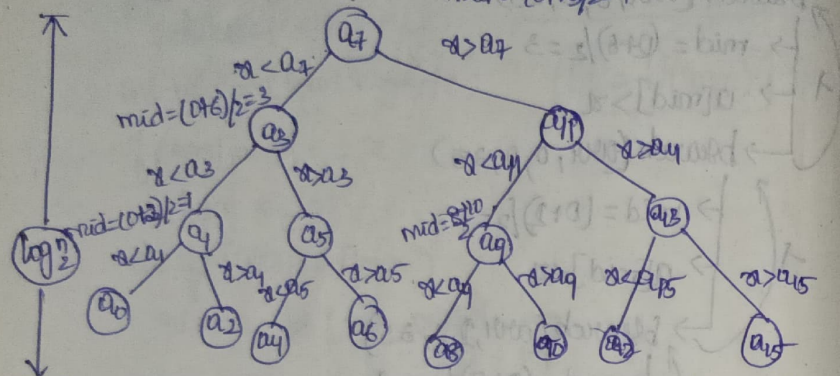


Analysis of Binary search:



X → Element to be searched

mid = (0+15)/2 = 7



possible Execution paths for Successful searches

→ Height of Binary tree = $\lceil \log n \rceil$

Unsuccessful searches

	work	TC
Successful searches		$\Theta(\log n)$
Unsuccessful searches		$\Theta(\log n)$

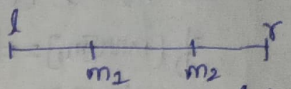
Ternary Search:

dividing the sorted array into 3 equal parts.

$$\text{Mid} = \begin{cases} mid1 = l + (r-l)/3 \\ mid2 = r - (r-l)/3 \end{cases}$$

$l = 0$, $r = n-1$

Iterative:




```
int ternarySearch(int arr[], int key, int l, int r)
{
  while (l<=r)
  {
    int m1 = (l+(r-l)/3);
    int m2 = (r-(r-l)/3);
    if (arr[m1] == key)
      return m1;
    if (arr[m2] == key)
      return m2;
    if (key < arr[m1])
      r = m1-1;
    else if (key > arr[m2])
      l = m2+1;
    else
    {
      l = m1+1;
      r = m2-1;
    }
  }
  return -1;
}
```

→ key < a[m1] → r = m1-1

→ key > a[m2] → l = m2+1

T.C = $O(\log \frac{N}{3})$

```
int tsearch (int arr[], int key, int l, int r)
{
    if (l>r)
        return -1;
    int m1 = l+(r-l)/3;
    int m2 = l-(r-l)/3;
    if (arr[m1] == key)
        return m1;
    if (arr[m2] == key)
        return m2;
    if (arr[m1] > key)
        return tsearch (arr, key, l, m1-1);
    else if (arr[m2] < key)
        return tsearch (arr, key, m2+1, r);
    else
        return tsearch (arr, key, m1+1, m2-1);
}
```

Index of first occurrence:

I/P: arr[] = [1, 10, 10, 10, 20, 20, 40]
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: T.C = O(n) A.S = O(1)

```
int firstOccurrence (int arr[], int n, int x)
{
    for (i=0; i<n; i++)
    {
        if (arr[i] == x)
            return i;
    }
    return -1;
}
```

arr[] = [5, 10, 10, 15, 15]
x = 15
O/P: 3

Recursive Binary search: T.C = O(log n) A.S = O(log n)

T(n) = T(n/2) + θ(1)

```
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 (arr[mid] > x)
        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

→ firstOcc(arr, 4, 4, 20)

→ mid = 4

→ return 4.

Iterative Solution:-

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

x = 20
arr[] = [5, 10, 10, 15, 20, 20, 20]
low = 0 high = 6

1st: mid = 3, high = 2

2nd: mid = 1, low = 2

3rd: mid = 2, return 2

Index of last Occurrence:

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

I/P: arr[] = [5, 8, 8, 10, 10], x = 10
O/P: 4

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

Efficient Solution:

Recursive Binary search:

[5, 10, 10, 10, 10, 20, 20]

x = 10

lastOcc(arr, 0, 6, 10)

→ mid = 3

→ lastOcc(arr, 4, 6, 10)

→ mid = 5

→ lastOcc(arr, 4, 4)

→ mid = 4

→ return 4.

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

Iterative

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

0	1	2	3	4	5	6
5	10	10	10	10	20	20

x=10
low=0 high=6

1st iter: mid=3, low=4

2nd iter: mid=5, high=4

3rd iter: mid=4, return 4

T.C = O(logn)
A.S = O(1)

Count Occurrences in a Sorted Array

I/p: arr[] = [10,20,20,20,30,30], x=20
O/p: 3

I/p: arr[] = [5,8,10], x=7
O/p: 0

I/p: arr[] = [10,10,10,10,10], x=10
O/p: 5

By using B.S = O(logn+k) - but in wc it becomes O(n)

Naive approach: linear search ⇒ T.C = O(n), A.S = O(1)

Efficient approach: so

```
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);
}
```

T.C = O(logn)
A.S = O(1)

[10, 20, 20, 20, 30, 30] (indices 0 1 2 3 4 5)
3-1+1 = 3

Count 1's 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: T.C = O(n)
linear scan for 1 & subtract from n

efficient:

```
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]!=arr[mid-1] (or) arr[mid-1]==0)
        return (n-mid);
      else
        high=mid-1;
    }
  }
  return 0;
}
```

$T.C = O(\log n)$
$A.S = O(1)$

0	1	2	3	4	5	6
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 (⇒ floor of sqrt)
I/p: x=25 → o/p: 5

Naive Solution: T.C = $O(\sqrt{x})$, A.S = O(1)

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

x=9 : i=1, i=2, i=3, i=4, return (4-1) = 3

x=15 : i=1, i=2, i=3, i=4, return (4-1) = 3

efficient solution: T.C = $O(\log n)$, A.S = $O(1)$

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

x=10
low=1 high=10

I st iteration: mid=5, midSq=25, high=4

II nd iteration: mid=2, midSq=4, low=3, ans=2

III rd iteration: mid=3, midSq=9, low=4, ans=3

IV th iteration: mid=4, midSq=16, high=3

Search in an Infinite sorted array :-

I/p: arr[] = [1, 10, 15, 20, 40, 80, 90, 100, 120, 500]
x = 100

o/p: 7

I/p: arr[] = [20, 40, 100, 300. . . .]
x = 50

o/p: -1

Naive solution: T.C = O(position), A.S = O(1)

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

[unbounded b.s]

effecient solution:

```
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(arr, i/2+1, i-1);
}
```

arr[] = [1, 10, 15, 20, 40, 80, 100, 120, 300, 500, 1000 . . .]
x = 100
Initially: i = 1
After while loop: i = 8
Return binarySearch

T.C = ~~O(2 * pos)~~ O(log(2 * pos)) = O(log(pos))

Search in a Sorted Rotated array :

I/p: arr[] = [10, 20, 30, 40, 50, 8, 9] (Distinct elements)
x = 30

o/p: 2

I/p: arr[] = [100, 200, 300, 10, 20]
x = 40

o/p: -1

Naive solution: T.C = O(n), A.S = O(1)

```
int search [int arr[], int x, int n)
{ for (int i=0; i<n; i++)
    { if (arr[i] == x)
        return i;
    }
  return -1;
}
```

effecient approach (Binary search): T.C = O(log n), A.S = O(1)

[100, 200, 500, 1000, 2000, 10, 20]
x = 25

[100, 500, 10, 20, 30, 40, 50]

x = 40 — 100 > 20 — search

x = 500 => [100, 500, 10, 20, 30, ~~40~~, ~~50~~]

[100, ~~200~~, ~~300~~, 2100, 20, 30, 40]
x = 50

```
int search (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;
    }
    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;
}
```

T.C = O(Logn)
A.S = O(1)

=> Like normal Binary Search

arr[] = [10, 20, 40, 60, 4, 8]
n=6

Initially: low=0, high=5

I st iteration: mid=2, low=3

II nd iteration: mid=4, low=5

III rd iteration: mid=5, high=4

return -1.

arr[] = [10, 20, 40, 60, 4, 8]
n=5

Initially: low=0 high=5

I st iteration: mid=2, low=3

II nd iteration: mid=4

return 4.

Find a peak element:

=> Not Smaller than Neighbours

I/p: arr[] = [5, 10, 20, 15, 7]

O/p: 20

I/p: arr[] = [10, 20, 15, 5, 23, 90, 67]

O/p: 20 OR 90

I/p: arr[] = [80, 70, 90]

O/p: 80 OR 90

Naive solution:

T.C = O(n)
A.S = O(1)

[80 70 90]

```
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 solution:

[5, 20, 40, 30, 25, 50, 60]

mid = (0+6)/2 = 3

if (arr[mid] >= arr[mid])

{ // peak element lies on left half

}

else

{ // right half:

}

[5, 80, 40, 30, 25, 50, 60]

[120, 80, 40, 80, 50, 50, 60]

corner cases:

[10]

[10, 20]

[20, 10]

T.C = O(log n)
A.S = O(1)

```
int getPeak (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;
}
```

Something on left on mid

nothing on left of mid

Two pointer Approach:-

ex: find if there is a pair with sum x in a sorted Array.

I/p: arr[] = [5, 8, 12, 30]
x = 17
o/p: TRUE // pair is (5, 12)

I/p: arr[] = [3, 8, 13, 18]
x = 14
o/p: false.

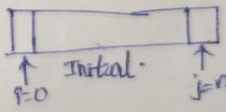
Naive Solution:

T.C = O(n²)
A.S = O(1)

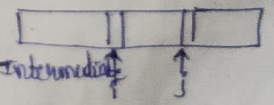
```
bool ispair(int arr[], int n, int x)
{ for (i=0; i<n; i++)
  { for (j=i+1; j<n; j++)
    { if (arr[i]+arr[j] == x)
        return true;
    }
  }
  return false;
}
```

Idea for efficient solution:-

let the given sum be x.



⇒ we move i and j toward each other




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

[2, 3, 8, 9, 11, 12, 20, 30] x = 23

i=0, j=7

(2+30) > 23 : i=0, j=6
(2+20) < 23 : i=0, j=5
(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.

T.C = O(n)
A.S = O(1)

```
bool isPair(int arr[], int n, int x)//two pointer technique
{
    int i=0, 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;
}
```

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 solution

```
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;
}
```

T.C = O(n^3)
S.C = O(1)

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

Idea for the effecient solution:

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 right side with sum (x - arr[i])

T.C = O(n²) S.C = O(1)

0	1	2	3	4	5	6
2	3	4	8	9	20	40

x = 32

2: search for (32-2) in arr[1:6]

3: search for (32-3) in arr[2:6]

we find a pair with sum 29, we return 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)
            j--;
        else
            i++;
    }
    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;
}
```

arr = [2, 5, 10, 15, 18] x = 33

i=0 : ispairSum(arr, 5, 31, 1)

i=1 : ispairSum(arr, 5, 28, 2)

return True.

triplet is (5, 10, 18)

Median of two sorted arrays (Hard)

I/P: a1[] = [10, 20, 30, 40, 50]

a2[] = [5, 15, 25, 35, 45]

O/P: 27.5

0	1	2	3	4	5	6	7	8	9
5	10	15	20	25	30	35	40	45	50

$\frac{25+30}{2} = \frac{55}{2} = 27.5$

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, 20, 30, 40, 50]

I/P: a1[] = [10, 20, 30, 40, 50, 60]

a2[] = [1, 2, 3, 4, 5]

O/P: 10.0

[1, 2, 3, 4, 5, 10, 20, 30, 40, 50, 60]

Naive : $O((n_1+n_2)*\log(n_1+n_2))$

n_1 : size of $a_1[]$
n_2 : size of $a_2[]$

1. create an array temp[] of size (n_1+n_2).
2. copy elements of $a_1[]$ and $a_2[]$ to temp[]
3. sort temp[]
4. if (n_1+n_2) is odd, then return middle of temp.
5. else return average of middle two elements.

$a_1[] = [10, 20, 30]$ $\quad a_2[] = [5, 15, 25]$

temp[] = [10, 20, 30, 5, 15, 25]

after sorting temp[] = [5, 10, 15, 20, 25, 30]

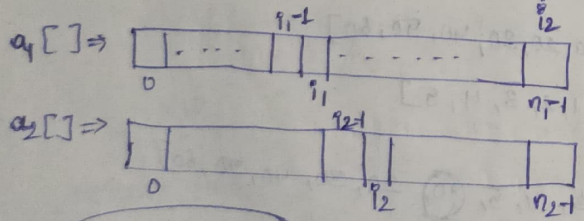
size is even

$\frac{15+20}{2} = \frac{35}{2} = 17.5$ ✓

Efficient: $O(\log n_1)$ where $(n_1 <= n_2)$

$n_1 = 5$ $\quad a_1[] = [10, 20, 30, 40, 50]$ $\quad \frac{0+5}{2} = 2$ ✓

$n_2 = 9$ $\quad a_2[] = [5, 15, 25, 35, 45, 55, 65, 75, 85]$



$a_1[0 \cdots i_1-1]$
$a_2[0 \cdots i_2-1]$
Left Half

$a_1[i_1 \cdots n_1-1]$
$a_2[i_2 \cdots n_2-1]$
Right Half

$i_2 = \left\lfloor \frac{n_1+n_2+1}{2} \right\rfloor - i_1$ $\quad \frac{5+9+1}{2} - 2 = 5$

If two arrays left half elements are <= Right half elements.

then take

$a_1[i_1]$ as min 1
$a_1[i_1-1]$ as max 1
$a_2[i_2]$ as min 2
$a_2[i_2-1]$ as max 2

If (n_1+n_2) is even $\Rightarrow$ find (max(max1, max2) + min(min1, min2))/2
is odd $\Rightarrow$ max(max1, max2)

$\Downarrow$

$n_1 = 5$ $\quad a_1[] = [10, 20, 30, 40, 50]$ $\quad \frac{0+5}{2} = 2$

$n_2 = 9$ $\quad a_2[] = [5, 15, 25, 27, 28, 55, 65, 75, 85]$

$i_2 = \left(\frac{5+9+1}{2}\right) - 2 = 5$ (i_2-1, i_2)

Two arrays left half elements <= Right half elements.

then.

$(\max(20, 28) + \min(30, 55))/2$

$(28+30)/2 = 58/2 = 29$

$\Rightarrow$ $a_1[] = [10, 20]$ $\quad i_1 = \frac{0+2}{2} = 1$ (begin, end) $\quad n_1 = \text{size of}(a_1)$

$a_2[] = [25]$ $\quad i_2 = \left(\frac{2+1+1}{2}\right) - 1 = 1$

if we take $a_1 = \text{size of}(a_1) - 1$

$i_1 = \frac{0+1}{2} = 0$

$i_2 = \left(\frac{2+1+1}{2}\right) - 0 = 2$

30 — Not present

begin: we do binary search for less sized array

for a1[] $\Rightarrow$ begin1 = 0
end1 = n_1

Code: $O(\log n_1)$ $(n_1 <= n_2)$

```
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;
    }
}
```

[max1 min1 ↓ ↓ i1-1 i1]

[max2 min2 ↑ ↑ i2-1 i2]

min1 : Minimum element in right side of a_1.

max1 : Maximum element in left side of a_1.

min2 : Minimum element in right side of a_2.

max2 : Maximum element in left side of a_2.

ex:

$a_1[] = [30, 40, 50, 60]$ (0, 1, 2, 3)

$a_2[] = [5, 6, 7, 8, 9]$ (0, 1, 2, 3, 4)

begin1 = 0 end1 = 4

$i_1 = 2$

$i_2 = (4+5+1)/2 - 2 = 3$

min1 = 50 min2 = 8

max1 = 40 max2 = 7

IInd Iteration:

begin1 = 0 end1 = 1

$i_1 = \frac{0+1}{2} = 0$ [30, 40, 50, 60]

$i_2 = \left(\frac{4+5+1}{2}\right) - 0 = 5$ [5, 6, 7, 8, 9] (0, 1, 2, 3, 4)

max1 = $-\infty$ min1 = 30

max2 = 9 min2 = ∞

res = max(max1, max2)

res = max($-\infty$, 9) = 9.

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)
* All the elements from 0 to max(arr) are present

therefore $0 \leq max(arr) \leq n-2$

Repeating element:

* O(n) Time
* O(1) Aux Space
* No modifications to original array.

[0, 2, 1, 3, 2, 2]

Super Naive solution:

T.C = $O(n^2)$
A.S = O(1)

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

Naive solution:

T.C = O(nlogn)
A.S = O(1)

1) Sort the Array: [0, 1, 2, 2, 2, 3]

2)
```
for (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 → (n-1)

visited[] = [F, F, F, F, F, F,]

2)
```
for(i=0; i<n; i++)
{
    if(visited[arr[i]])
        return arr[i];
    visited[arr[i]] = true;
}
```

deleted

we can save space bcz array elements are $\leq n-2$
so we create size of n-1

Repeating Element O(n) Time, O(1) space & No changes to Original array:

Ip: arr[] = [1, 3, 2, 4, 3, 3]

Op: 3

Ip: arr[] = [1, 1]

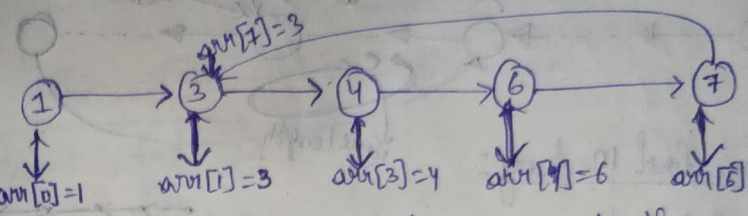
Op: 1

Ip: arr[] = [3, 4, 9, 1, 2, 4, 4]

Op: 4

* All Elements from 1 to max(arr) are present.
* $1 \leq \max(arr) \leq n-1$

arr[] = [1, 3, 2, 4, 6, 5, 7, 3]
0 1 2 3 4 5 6 7

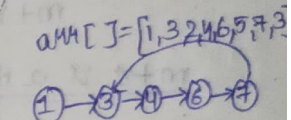


The first element of cycle will always be the repeating element

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



phase-I

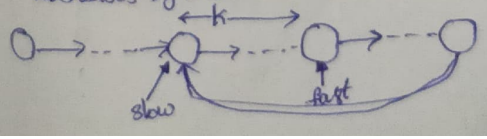
slow=1, fast=1
slow=3, fast=4
slow=4, fast=7
slow=6, fast=4
slow=7, fast=7

phase-II

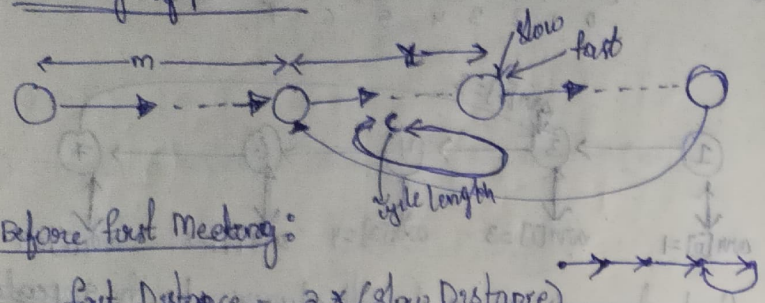
slow=1, fast=7
slow=3, fast=5

Working of phase-I

1) fast will enter the cycle first (or at same time)
2) In every iteration, distance increases by one



Working of phase-2 :



Before fast Meeting :

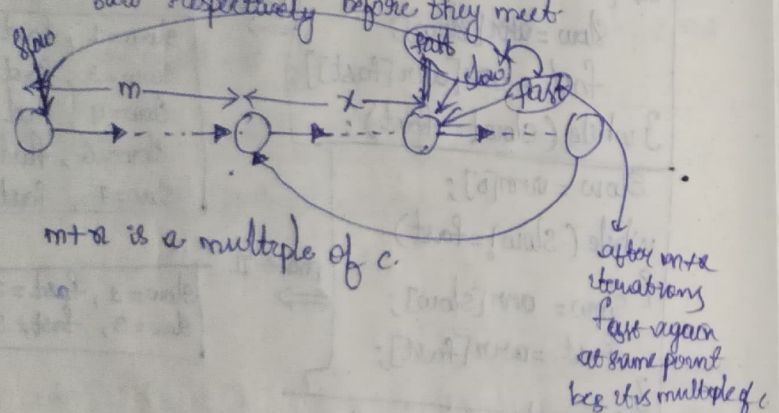
fast Distance = 2 x (slow Distance)

$m + x + c \times i = 2 \times (m + x + c \times j)$

$m + x = c \times (i - 2j)$

m+x is a multiple of c

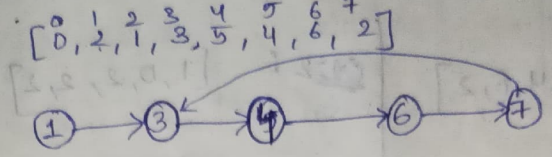
i, j ⇒ Number of cycle iterations made by fast and slow respectively before they meet



m+x is a multiple of c.

after m+x iterations fast again at same point beg the multiple of c

Solution for arrays having elements from 0.



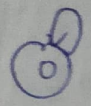
slow = 1 , fast = 1
slow = 3 , fast = 7
slow = 4 , fast = 7
slow = 6 , fast = 4
slow = 7 , fast = 7

slow = 1 , fast = 7
slow = 3 , fast = 3

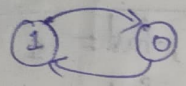
```
int findRepeating(int arr[], int n)
{
  int slow = arr[0]+1, fast = arr[0]+1;
  do
  {
    slow = arr[slow]+1;
    fast = arr[arr[fast]]+1;
  } while (slow != fast);
  slow = arr[0]+1;
  while (slow != fast)
  { fast = arr[fast]+1;
    slow = arr[slow]+1;
  }
  return slow-1;
}
```

why do we increase slow and fast by 1?

ex-1: [0,2,1,3,5,4,6,2]



ex-2: [1,0,2,2,2]



Allocate Minimum No. of pages:

I/p: arr[] = [10,20,30,40] ⇒ 0, 100
10 90
30 70
60 40
k=2

O/p: 60

I/p: arr[] = [10,20,30]
k=1

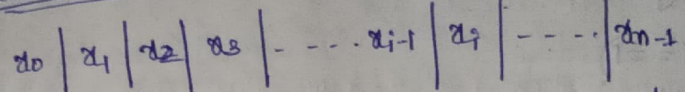
O/p: 60

I/p: arr[] = [10,5,30,1,2,5,10,10]
k=3

O/p: 30

* Minimize the maximum pages allocated
* only contiguous pages can be allocated

Naive Recursive solution:-

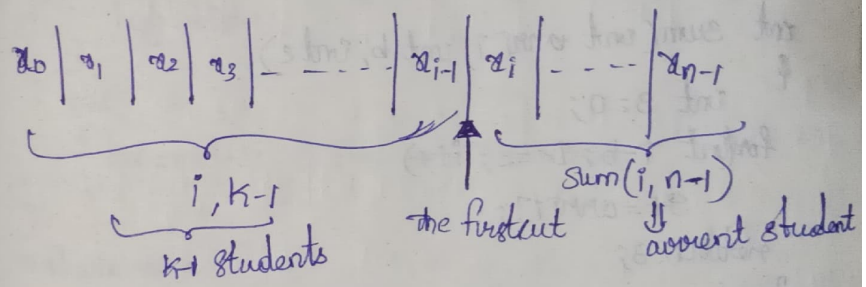


we need to choose (k-1) cuts out of (n-1) cuts shown above

Total ways = $^{n-1}C_{k-1}$

ex: [10, 20, 30, 40]
k = 2

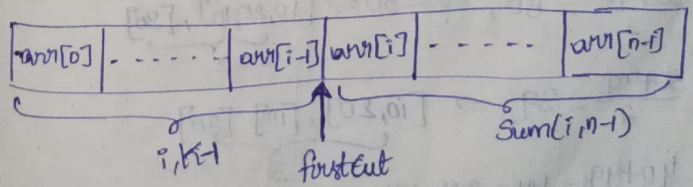
we need to choose 1 cut out of 3 cuts



$mp(n,k) = \min_{i=1}^{n-1}(\max(mp(i,k-1), Sum(arr, i, n-1)))$

$mp(1,k) = arr[0]$ (1 = posts, k = students)

$mp(n,1) = Sum(arr, 0, n-1)$



```
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 = INT_MAX;
    for (int i=1; i<n; i++)
        res = min(res, max(minpages(arr, i, k-1), sum(arr, i, n)));
    return res;
}

int sum(int arr[], int b, int e)
{
    int s=0;
    for(int i=b; i<=e; i++)
        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] (30 = max value, 70 = sum)

mid = (30+70)/2 = 50, res=50 ⇒ [10,20,10], [30] (2)

mid = (30+49)/2 = 39, ⇒ [10,20], [10] [30] (3)

mid = (40+49)/2 = 44, res=44

mid = (40+43)/2 = 41, res=41

mid = (40+40)/2 = 40, res=40

```
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 the left half
            high=mid-1;
        }
        else
        {
            low=mid+1;      // Else goto 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 (sum+arr[i] > ans)
        {
            req++;
            sum=arr[i];
        }
        else
            sum+=arr[i];
    }
    return (req<=k)
}
```

T.C = O(n * log(sum-mx))

O(n * log(sum))

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

mid = (20+35)/2 = 27, res = 27

mid = (20+26)/2 = 23, res = 23

mid = (20+21)/2 = 21, res = 21

mid = (20+20)/2 = 20, res = 20