⇒ Binary Search ||

~~1st~~ first occurence in binary search

```
if (arr[mid] == x){
    ans = mid;
    high = mid - 1; // find in left        1 2 3 4 4 4 5 6
    continue;                                     x = 4
}
                                                o/p = 3
if (arr[mid] < x) {
      low = mid+1;
  }
~~if (~~ else {
      high = mid - 1;
   }
}
return ans;
```

→ given an sorted array of duplicates, find the last occurence of x ?

// same code as before only change high to low.

```
if (arr[mid] == x){
    ans = mid;
    low = mid + 1;      // find more in right side
    continue;
}
```

= same;

⇒ Recursive Solution of binary search

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

⇒ lower bound of x

↳ first element >= x is the lower bound of x.

arr[] = {1, 3, 5, 7, 9, 10}

x = 7 , then lb = 7

x = 8 , then lb = 9

x = 11 , lb ✗

```
int lb = lower_bound (arr+0, arr+n, x) - arr;
```

arr+0 ↓ beginning iterator

arr+n ↓ end iterator

// it will give the index of lower bound of x.

a	a+1	a+2	a+3	a+4	a+5	a+6
1	2	3	4	5	6	7

x = 4

if a+3 is the address of 4, ~~then~~ by subtracting 'a' from it, we will get 3, i.e the index of '4'.

// if lower bound of any no. doesn't exist, then the pointer will point to nth index. (last index) ~~where~~ ~~that~~ is which is out of bounds of array.

// user ~~&~~ defined funcn for lower bound using binary search

```
int lb (int arr[], int n, int x) {
        int ans = n;          // assume that there doesn't exist any
                                  lower bound.

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

⇒ Upper bound

↳ |first element > x|

arr [ ]= {1, 3, 4, 6, 6, 7, 9}

x= 6 , ub= 5
x= 4 , ub= 3
x= 5 , ub= 3
x= 9 ub = 7
x= 10 ub = 7

} indexes

if 'ub' not found then nth index is returned, i.e size of array.

```
int ub (int arr[], int n, int x){
    int u= upper_bound(arr, arr+n, x) - arr;
    return u;
}
```

↓ inbuilt funcn

```
// user defined function
int ub( int arr[], int n, int x) {

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

⇒ Given ~~x~~, find the no. of occurences of x:-

x = 7
arr [ ] = { 1, 3, 5, 7, 7, 7, 10}

1st method   Recursion

```
int c = 0; →global
int binary (int a[], int l, int h; int x) {
        if (l <= h) {
            int mid = (l+h)/2;
            if (a[mid] == x) {  c++; }
            else if (a [mid] < x) {
                    l = mid + 1;
            }
            else {    h = mid - 1; }
        }
        binary (a, l, mid - 1, x);
        binary (a, mid+1, h, x);
    }
    return c;
}
```

2nd method → Using first occurence & last occurence

first occurence of '7' = ~~4~~ 3
last " " " = 5

∴ no. of '7' := last - first + 1
= 5 - 3 + 1 = 3.

3rd method → Using lower bound & upper bound

lb of 7 = 3
ub of 7 = 6

```
if (~~lb~~ a[lb] != x) {return 0;}   //i.e x doesn't exist
else {return (ub - lb);}
```

Q:- find integer square root of given n ?

n = 26 , then $\sqrt{26} = 5$

Constraints :→ 1 < = N < = INT_MAX

Biggest i such that
i * i ≤ N

we can clearly say that square root of any n will always lie b/w 1 to n. , ~~bcz~~ we can't take 1 to n/2 , bcz for n = 1, n/2 = 0 & 0 can't be square root of n.

Now, think of a hypothetical array , 1 to 26.
~~now~~ calculate mid = (1+26)/2 = 13 , since 13×13 > 26
∴ high = mid - 1 = 12,
~~now~~ again follow the same procedure & update the answer variable till you get the largest i such that $i^2 <= n$

```
int sqrt(int n){
  int low = 1,  high = n;
   int ans = 1;
  while (low <= high){
      ~~if~~ int  mid = (low + high)/2;
        if (~~a*mid~~ mid * mid <= n){
              ans = mid;
              low = mid + 1;
        }
        else { high = mid - 1; }
  }
  return ans;
}
```

Q:- find integer cube root. of $n=26$

~~Simply~~ same procedure as before, just update $(mid)^2 \le n$ to $(mid)^3 \le n$, i.e, if (mid * mid * mid <= n)

⇒ overflow condition

int low, high;

if ~~on~~ the low & high is int_max, i.e 2147483647, then:-

when we do, $mid = \underbrace{(low+high)}_{\text{This portion will be overflow although the mid will not overflow, but the ans may be wrong.}}/2$

so, to avoid this, we can write

$$\boxed{mid = low + (high - low)/2} \Rightarrow \frac{2low + high - low}{2} = \left(\frac{low + high}{2}\right)$$

or another way is ~~that~~, take larger data type such as long or long long.

⇒ find minimum in rotated sorted array.

Given the sorted array ~~of~~ of nums of unique elements, find the min element.

eg → [3, 4, 5, 1, 2]   o/p - 1

Approach

```
  0    1   2   3   4    5   6   7
[ 7    8   9   1   2    4   5   6 ]
  ↑            ↑                ↑
 low          mid              high
```

$mid = \frac{0+7}{2} = 3$

since a[mid] < 6, i.e, high

So, I can clearly say that the min. can't be available from mid + 1 to high. , But the 'mid' ~~can~~ might be ~~my~~ the minimum.

~~so if a[mid] < 1~~

So, we will modify (high = mid) instead of high = mid − 1 in the classical approach of Binary Search

```
                         0   1   2   3
now, the array becomes :- [ 7   8   9   1 ]
                           ↑           ↑
                          low         high
```

$mid = \frac{0+3}{2} = 1$

now, a[mid] = 8 > 1

here, we can clearly say that 8 is greater than 1 so, the answer can never lie on the left side upto 8, the array will be eliminated.

∴ (low = mid + 1)

```
[ 9, 1 ]     mid = 9
  ↑  ↑
 low high
```

∵ 9 > 1

∴ low = mid + 1

now, mid = high = low = 1.

∴ ans ⇒ a [low].

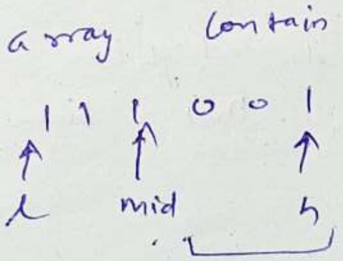
→ In any binary Search, a[low] will always be the answer.

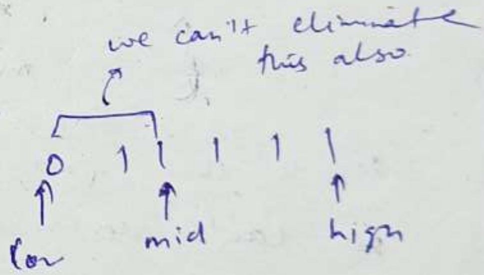
// code

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

// if the array contain duplicates - or

e.g →





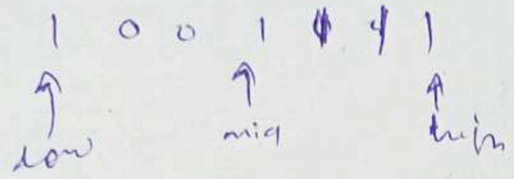
if I eliminate this whole right subarray, as arr[mid] >= arr[h] then, it might be possible that any smaller element is there in b/w mid & high.

so, here a modification is that, if arr[mid] = arr[high] then decrement the high to only one element left side.

ie, if arr[mid] = arr[high], { h - - ; }

remaining all code will be same.

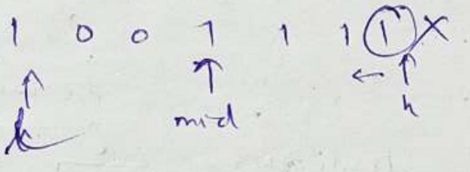
if the conditi is somewhat like this :-



here, low, mid & high all are equal, so what to compare & how to decide.

In such case, we can't determine ~~how to elimate~~ which portion to eliminate.

• In these types of cases, one thing we can surely say that the arr[high] ~~to~~ can never be the answer.



so, in else part, we will do high-- & nothing ~~we~~ can't do other than that, bcz we are not sure.

Time complexity

* if the array is [1, 1, 1, 1, 1, 1] in such case we every time do high-- ending up in O(n) time.

* But the average case time will be O(logn)

```
// code :-
  while (low < high) {
      int mid = (low + high)/2;
      if (arr[mid] < arr[high]) { high = mid; }
      else if (arr[mid] > arr[high]) { low = mid + 1; }
      else { high --; }
  }
  return arr[high]; or arr[low]   // both will be same.
```

→ Binary Search

Q:- Search in Rotated Sorted Array

A rotated sorted array is given find 'x' in it.

```
[ • [4, 5, 6, [7, 8, 0, 1, 2]]          x = 1
    ↑         ↑              ↑
    l        mid             h
```

I can clearly say that at least one ~~either~~ of two parts of the array is definitely sorted. we will check which part is sorted & if 'x' lies in that part, the other part will be eliminated, or Else that particular part will be eliminated.

"Binary search is a game of elimination"

here, 4, 5, 6, 7 is sorted, since x=1 does not lie b/w 4 to 7, i.e, we will shift low to mid+1. & follow the same procedure further

```
// code :-
int  l = 0,   h = n - 1;
while ( l <= h) {
    int  mid = (l + h)/2;
    if (arr [mid] == x)  return mid;

    // if the left part is sorted
    if ( arr [low] <= arr [mid]) {
        if (x >= arr [low] && x <= arr[mid]) {
            high = mid - 1;
        }
        else {
            low = mid + 1;
        }
}
```

// or the right part is sorted.

```
else {
    if (x >= arr[mid] && x <= arr[high]) {
        low = mid + 1;
    }
    else {
        high = mid - 1;
    }
  }
 }
 return -1;
}
```

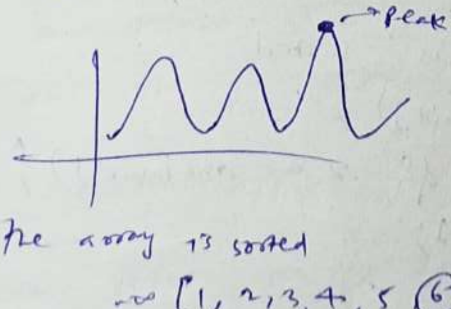
Q: Find peak element

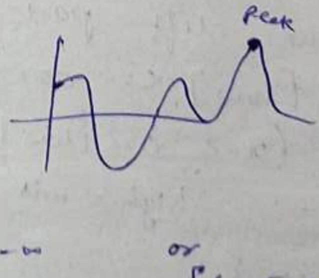
A peak element is an element that is strictly greater than its neighbours.

Given an array return the index of peak element. (any of the peak element)

Imagine nums[-1] = nums[n] = -∞ & there are ~~no duplicates~~ no consecutive duplicates

⇒ It is guaranteed that there is always a peak for any random array

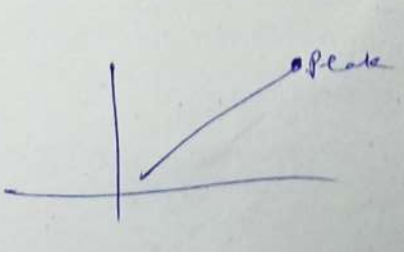


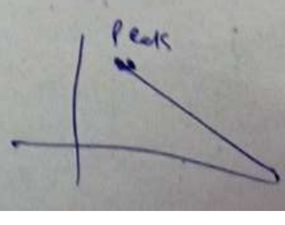


if the array is sorted

-∞ [1, 2, 3, 4, 5 6] -∞ or [6 5 4 3 2 1]

peak





E.g → 1 2 3 4 3 2 1

↑ low ↑ mid ↑ high

if mid > than its ~~a~~ adjacent element then

can ~~defin they~~

there may be a peak on the side of adjacent element, but I can definitely say that, there is a peak on the other side of mid.

here, ~~the~~ ⇒ mid > mid + 1, then I can definitely say that there is a peak in left side of peak.

proof :- if mid > mid + 1

then for left hand side

$\underbrace{1\ 2\ 3}\ 4 \leftarrow$

if all the elements on left side are smaller than mid, then mid itself is the peak.

or otherwise if greater, then anywhere there will be a peak

$\boxed{9}$ 8 7 6 3 $\boxed{5}$ 4

peak

now, I can eliminate the right part, ie, high = mid.

[1, 2, 3, 4]

↑ l ↑ mid ↑ h

here, mid > mid - 1 or mid < mid+1 then definitely there will be a peak on right side of mid.

so, set low = mid + 1.

⇒ [3, 4]

low ↗ mid ↑ high

mid < mid + 1

⇒ there will be peak on right side, ⇒ [4] → will be the peak.

1/1 code

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

Q:- Minimum size subarray sum :-

Given an array of +ve integers & a integer target, return the minimal length of contiguous sub array of which the sum is greater than or equal to target. If not, return 0.

~~eg~~ eg → [2, 3, 1, 2, 4, 3]   target = 7

O/P = 2

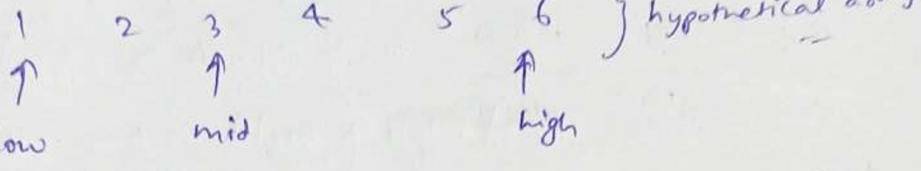
⟹ we ~~can~~ know that the length of the minm size ~~to~~ subarray can be :- 1, 2, 3, ---- n

So we will apply [binary search on the answer]

i.e, ~~how~~ what minm length is optimal.

e.g ⇒ [2, 3, 1, 2, 4, 3]

the answers can be:-



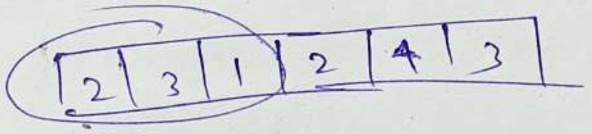
I will use the technique, that if at length = mid, the ~~sum~~ ~~of~~ max^m sum of subarray is >= 3, then check for ~~mid~~ the length lesser than mid, BCZ, if sum >= target at 'mid' only, then it will ~~also~~ definitely satisfy at length > mid.

But if not, ~~then~~ check for length > mid.

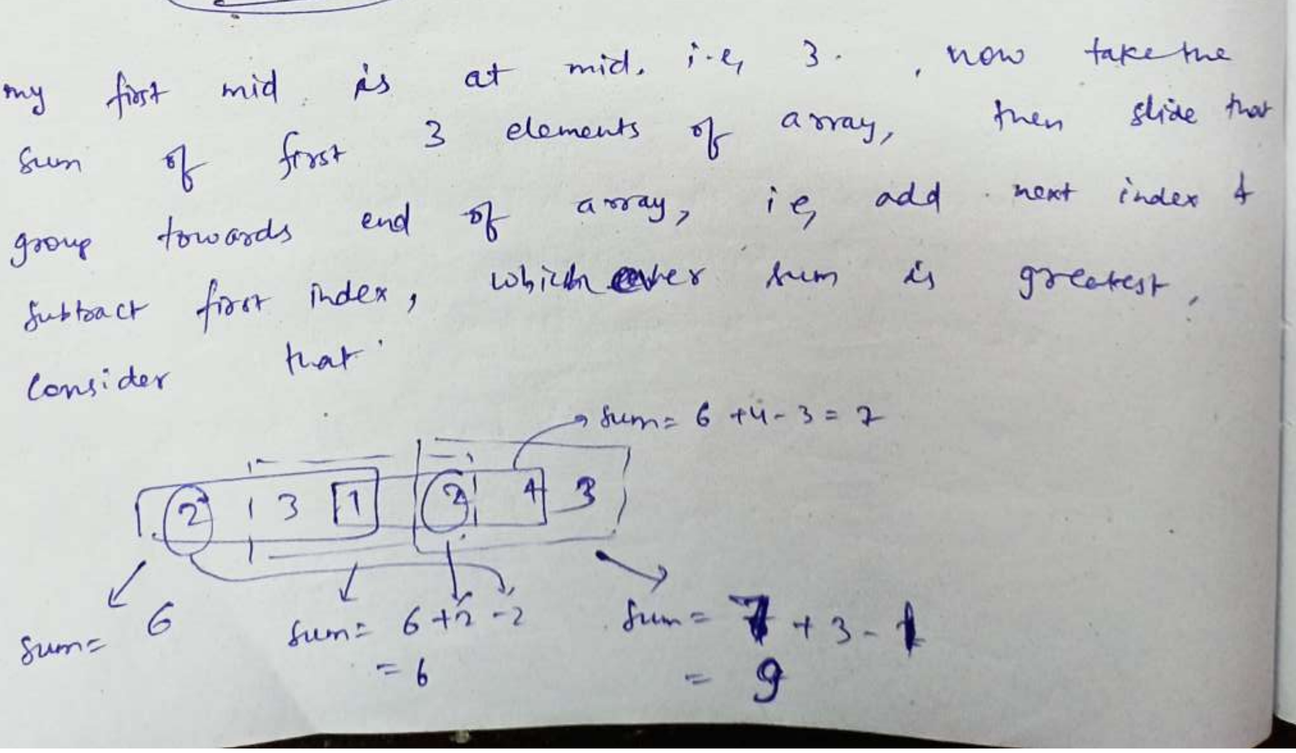
⇒ $\boxed{\text{Binary Search on Answers}}$

⇒ So, now to find max^m sum subarray at any length k ?

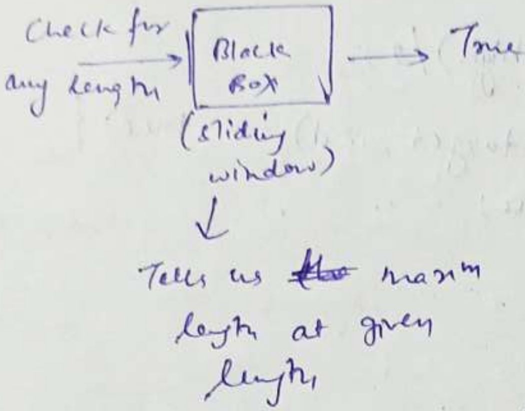
Use sliding window

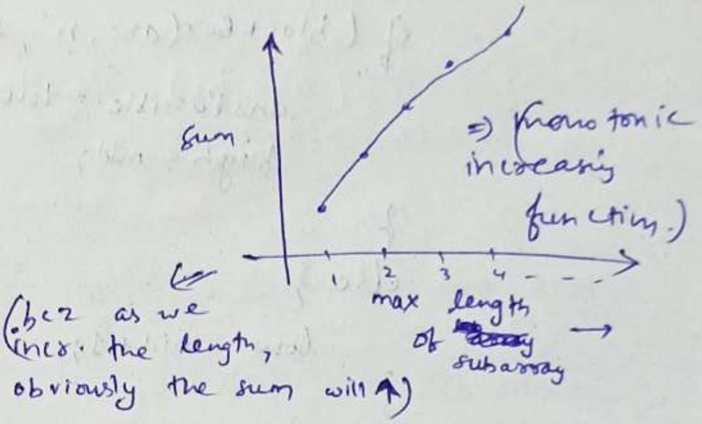


my first mid is at mid, i.e, 3. , now take the sum of first 3 elements of array, then slide that group towards end of array, i.e, add next index & subtract first index, whichever sum is greatest, consider that.



Since, the greatest sum is 9, which is >= 7,
so, make high = mid, & check for lesser ~~—~~ length.
And, by doing this, finally, we get our answer
as 2.





// code

```
bool blackBox( int arr[], int n, int target, int k){
    // check if there exists a subarray of size k which
       has a sum >= target

    // first find the first k size subarray sum
    int sum = 0;
    for(int i=0; i<k; i++) { sum += arr[i]; }
    int maxi = sum;
    int l=0, r=k-1;
    // move the slider
    while (r != n-1) {
        sum -= arr [l];
        l++;

        sum += arr [r];
        r--;
        maxi = max(maxi, sum);
    }
    return (maxi >= target);
}
```

```
// Find the min length of subarray such that sum >= target
int findMinLength(int arr[], int n, int target){
    int low = 1,   high = n;
    bool ansPossible = n;
    while (low < high) {
        int mid = (low + high)/2;
        if (blackBox(arr, n, target, mid) == true) {
            ansPossible = true;
            high = mid;
        }
        else {
            low = mid + 1;
        }
    }
    if (ansPossible == true)  return low;
    return 0;
}
```

Complexity → $O(n \log n)$

n → for black Box i.e, sliding window

$\log n$ → for binary Search

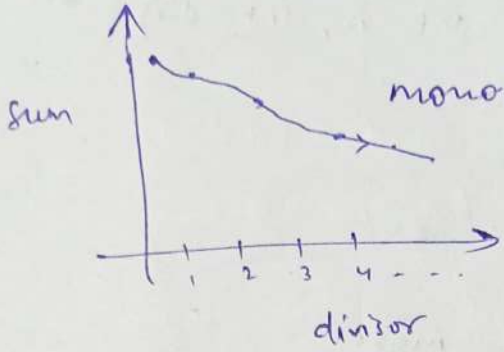
Q2:- Given an array of integers, & an integer threshold, choose a +ve integer 'divisor', divide the array by it & sum the divisor's result. find the smaller divisor such that the result is <= threshold.

result of division is taken in ceil. like $7/2 = 3$ ...

e.g → arr ⇒ [1, 2, 5, 9] , threshold = 6

o/p = 5   bcz, $\frac{1}{5} + \frac{2}{5} + \frac{5}{5} + \frac{9}{5} = 5 <= 6$

⇒ Binary search on Answer



(As we increase the ~~diviso~~ divisor, obviously the sum will decrease)

e.g → [1, 2, 5, 9]

now, the min^m divisor = 1, where we get the max^m answer.

and, obviously the max^m divisor will be the max^m element, which will convert all the elements to 1 & sum will be n. Any divisor greater than max^m element of array will produce the same o/p. = n.

now, think of a hypothetical array 1 to max-element, then find mid, if the mid is able to produce the sum <= threshold, then try check for any other smaller divisor, bcz if mid is able, then (mid+1) is obviously able.

And if mid can't produce a sum <= m, then find mid for (mid+1) to high, i.e, set low = mid+1.

My answer will always be 'high', bcz I have to return the max^m no. which is the smallest divisor.

// Code

```
int   findSumAfterDiv (int arr[], int n, int div) {
    int sum = 0;
    for (int i=0; i<n; i++) {
        sum += (arr[i]/div);

        // for ceiling add 1 more.
        if(arr[i] % div != 0) {
            sum += 1;
        }
    }
    return sum;
}

int   findMinDivisor (int arr[], int n, int threshold){
    int low= 1 , high = *max_element(arr, arr+n);
    int ans = high;
    while (low <= high) {
        int mid = (low + high) /2;
        // mid is giving <= threshold but I am
        looking for even smaller , hence do a search
        on the left.

        if(findSumAfterDiv(arr, n, mid) <= threshold) {
            ans = mid;
            high= mid-1;
        }
        else {
            low = mid +1;
        }
    }
    return high;
}
```

Complexity=>
n× log(max)

Q:- Split Array Largest Sum (Hard)

⇒ Given an array nums which consists of non-negative integers & an integer 'm', you can split the array into 'm' non-empty continuous subarrays. Write an algo to minimize the largest sum among these m subarrays.

e.g → [7, 2, 5, 10, 8]   m = 2

o/p = 18   ⇒ divide the array into [7, 2, 5] & [10, 8]
~~min~~ max^{m} sum is 18 which is minimized.

Sol → [7, 2, 5, 10, 8]

for m = 2, there can be 4 options for splitting the array.

take the max

[7]   [2, 5, 10, 8] ⇒ (7 , 25) ⇒ 25
[7, 2]   [5, 10, 8] ⇒ (9, 23) ⇒ 23
[7, 2, 5]   [10, 8] ⇒ (14, 18) ⇒ 18
[7, 2, 5, 10]   [8] ⇒ (24, 8) ⇒ 24

→ min = 18
Ans

Here, we have to find the min^{m} of the max^{m} sum of subarray splitted in m parts.

The worst case can be when the array of size n is to divided in n ways, then for each subarray there will be 1 element. and this is the only 1 way.

[7]. [2] [5] [10] [8] ⇒ max sum = 10

which is the minimum bcz, there is no other way possible to split array in n parts.

And the best case can be when m = 1, ~~the array~~ There is only 1 way, i.e, the whole array is the required subarray.

So, max - sum = [7 + 2 + 5 + 10 + 8] = 32

Now, I can definitely say that my answer always lie b/w 10 & 32., i.e, the range of answers & hence I will apply Binary search on [10, 32], i.e, for any array, my range will be [max_elem, sum of array].

* If You find the range of answers, half game is over.

10 11 12 13 - - - - - - - 32

↑ low ↑ mid = 21 ↑ high

for any 'mid', I can say that this the minimum of max^m sum which can be obtained by splitting the array into 'm' parts & each part having at most sum = mid.

~~If any part~~

So, iterate from beginning of the array, & push those elements in the subarray upto which at most sum is mid; then go to next subarray.

If no. of subarrays for a particular mid is > k, then any no. less than mid, will definitely not serve the purpose.

s.o, set low = mid + 1.

But if at any mid, no. of subarrays <= K, then mid might be the answer, but can we find any other minimum sum lesser than mid. If found, then report that otherwise the current value stored in answer. , i.e, set high = mid.

// code

```
countSubarrayAtLimit (int arr[], int n, int limit, int m);
int minMaxSumSubarray ( int arr[], int n, int m) {
    int low = * max_element (arr, arr+n);
    int high = 0;
    for (int i=0; i<n; i++) {
            high + = arr[i];
          // sum of the array.
    }
    int ans = high;      // Since, we have report the maxm
                            sum, so ans is always high.

    while (low <= high) {
        int mid = (low + high)/2;
        if( countSubarrayAt Limit(arr, n, mid, m) == false) {
              low = mid + 1;
        }
        else {
              ans = mid;
        }
    }
    return ans;
}
```

```
bool
~~int~~  countSubarrayAtLimit(int arr[], int n, int limit, int m){
      int  count = 1;   // count the no. of subarrays required
      int  sum = 0;

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

           if(arr[i] > limit)  return false;
           // If any element is greater than limit, then
              it cannot be part of the subarray.

           if(sum + arr[i] > limit) {
                 count++;
                 sum = arr[i];     // start a new subarray
           }                          from here

           else {
                sum += arr[i];
           }
      }

      return (count <= m);
}
```

Complexity

$$\boxed{n * \log(\underbrace{sum - max + 1}_{});}$$

no. of elements ~~distance~~ b/w sum & max

⇒ Q:- Divide Chocolate

You have one chocolate bar that consists of some chunks. Each chunk has its own sweetness given by the array sweetness.

You want to share the chocolate with your 'k' friends so, you cut ~~to~~ it into (k+1) pieces using k cuts, each piece consists of some consecutive chunks.

You will eat the piece of `minimum total sweetness' and give the other pieces to your friends. find the "maximum total sweetness" of the piece you can get by cutting the chocolate bar optimally

e.g → sweetness = [1, 2, 3, 4, 5, 6, 7, 8, 9], k = 5

o/p = 6

You can divide the chocolate to [1,2,3], [4,5], [6], [7], [8], [9].

Sol → Similar to the previous questions.

here, we have to find the max^m of min^m value of (k + 1) subarrays.

now, the array [1, 2, - - - 9] ~~to~~ is to be divided in 6 subarrays, i.e, k+1

It can be done in many ways.

(1,2,3) (4,5) (6) (7) (8) (9) ⇒ min^m sum = 6

(1, 2) (3, 4,5) (6) (7) (8) (9) ⇒ min^m sum = 3

(1) (2, 3) (4,5,6) (7) (8) (9) ⇒ min^m sum = 1

- - - etc.

And the max^m of these min^m sums is $\textcircled{6}$ ..

That's what we have to find.

In the worst case, we have n subarrays, each having only one element. In this case, the element having the minimum value is the minimum sum.

In another case, where there is only 1 subarray, i.e, the whole array, the minimum sum will be the sum of complete sum.

So, the range of answers is:-

[min elem of array — sum of array]

since, there can't be more divisions lesser than 1 element per subarray

now, in this e.g, the range is :-

1 2 3 4 — — — — — 45
↑ low ↑ high
↑ mid = 23

I will say that the mid is the ~~maximum~~ ~~minimum~~ min^m. sum and then check whether I get ' ~~more~~ more than or equal to (K+1) subarrays ~~havin~~ considering this mid,

If yes, then store that mid in a variable ans, and check for 'sum' greater than mid; bcz we have to find the max^m. of min^m sum.

∴ , set low = mid + 1.

And if, at mid, we can't find K+1 or greater no. of subarrays, then for any value > mid, we can't find any answer.

So, set high = mid - 1.

~~Since, sum equal to mid,~~

Since, our min^m sum is mid, So, take the values upto that index where sum of subarray is >= mid, bcz if mid is minimum, then all other sum of subarrays should be greater than mid.

[1, 2, 3, — — — — 45]

mid = 23

at (23), the no. of subarrays having sum >= 23 are:—

[(1, 2, 3, 4, 5, 6, 7), (8, 9)]

sum = 28 sum = 17 X

only 1 subarray. which is < k+1

so, now shift high = mid - 1.

ie, high = 23 - 1 = 22

[1, 2, — — — 22]

mid = 11

at mid = 11, the no. of subarrays are:-

[(1, 2, 3, 4, 5), (6, 7), (8, 9)]

15 13 17

3 subarrays which is < 6 (i.e, k+1)

hence, high = mid - 1 = 11 - 1 = 10

[1, 2, - — — 10]

↑

mid = 5

at mid = 5, no. of subarrays are:-

[1, 2, 3, 4, 5 6 7 8 9]

= 6 subarrays which is ≤ K+1

since, mid = 5 satisfy our result, can we check for some greater mid which will also satisfy.

ans = 5

set low = mid + 1 = 5 + 1 = 6.

[6, 7, 8, 9, 10]

↑

mid

at mid = 8 :-

[1, 2, 3, 4, 5, 6 . 7, 8, 9]

4 subarrays ✗ ..

set high = 8 - 1 = 7

[6, 7]

↑

mid.

no. of subarrays at '6' = 6 satisfied

ans = 6

set low = 6 + 1 = 7

[7] ← high

↑ ↑

low mid

at mid = 7, no. of subarrays = 5

not satisfied ✗

$\boxed{\text{Hence, our answer is 6}}$

// code

```
bool   canGet More Than _kSubarrays(int arr[], int n, int limit,
                                                  int k)
{
    int  count = 0;
    int  sum = 0;
    for( int  i=0;  i<n; i++) {
         sum += arr [i];

         if (sum >= limit) {
              count ++;
              sum = 0;
         }
    }
    return  (count  > k);
}
int find Max Chocolates(int  arr [ ],  int n,  int  k) {
      ~~int low=1~~, int high = accumulate (arr,  arr+n, 0);
      int  low =  *min_element(arr,  arr+n);
    while (low <= high) {
          int  mid =  (low + high)/2 ;
          if (can Get More Than_ k Subarrays(arr, n,  mid, k)){

                ans = mid;
                low = mid +1;
          }
          else {
                high = mid - 1;
          }
    }
    return ans;
}
```