Binary Search

---*---

→ Applicable only on sorted arrays. In some cases it is applicable on non-sorted arrays too; but that would be studied later.

generic code:

```
int binary Search (int* arr, int target) {

        int start = 0;
        ind end = arr. length -1;
         int mid;

        while ( start <= end) {
                  ────→ mid = start + (end-start)/2;
                  if (arr[mid] == target) {
                           return mid;
                  }
                 *else if (arr[mid] > target){
                           end = mid-1;
                  }
                  else start = mid+1 *
        }
        return -1.
}
```

(start + end)/2 is not used to protect the code from integer overflow.

→ In this code, it is assumed that array is st. in ascending order; for descending st. array the condition would change like this.

```
* else if (arr[mid] > target){
              start = mid+1;
  }
  else    end = mid-1; *
```

first and last occurence of a given target :

- first Occurence

```
           0   1   2   3   4   5   6   7   8   9   10
eg.  i/p : 2 , 4,  4,  5 , 6,  7,  7,  7,  9,  11, 11

     target = 4

     o/p :  1 (idx)
```

↳ slight change:

```
if (mid == target){
      ans = mid;
      end = mid -1;
}
else if (arr [mid] > target){
      end = mid - 1;
}
else start = mid +1;
```

- Last occurence

```
if (arr [mid] == target){
      ans = mid;
      start = mid + 1;
}
```

count of elements in sorted array:

```
return (last_occurence (arr, target) -
        first_occurce (arr, target) + 1);
```

Number of times a sorted array is rotated:

eg. i/p: 5 6 7 8 9 0 1 2 3 4

0 1 2 3 4 5 6 7 8 9

o/p: 5

explanation:

original sorted array:

0 1 2 3 4 5 6 7 8 9

1st rot. : 9 0 1 2 3 4 5 6 7 8

2nd rot. : 8 9 0 1 2 3 4 5 6 7

3rd rot. : 7 8 9 0 1 2 3 4 5 6

4th rot: 6 7 8 9 0 1 2 3 4 5

5th rot: 5 6 7 8 9 0 1 2 3 4

= given array

hence: output = 5 rotations.

observations: idx of min element = no. of rot.

Soln → find min element and return its index → O(N)

↳ linear complexity.

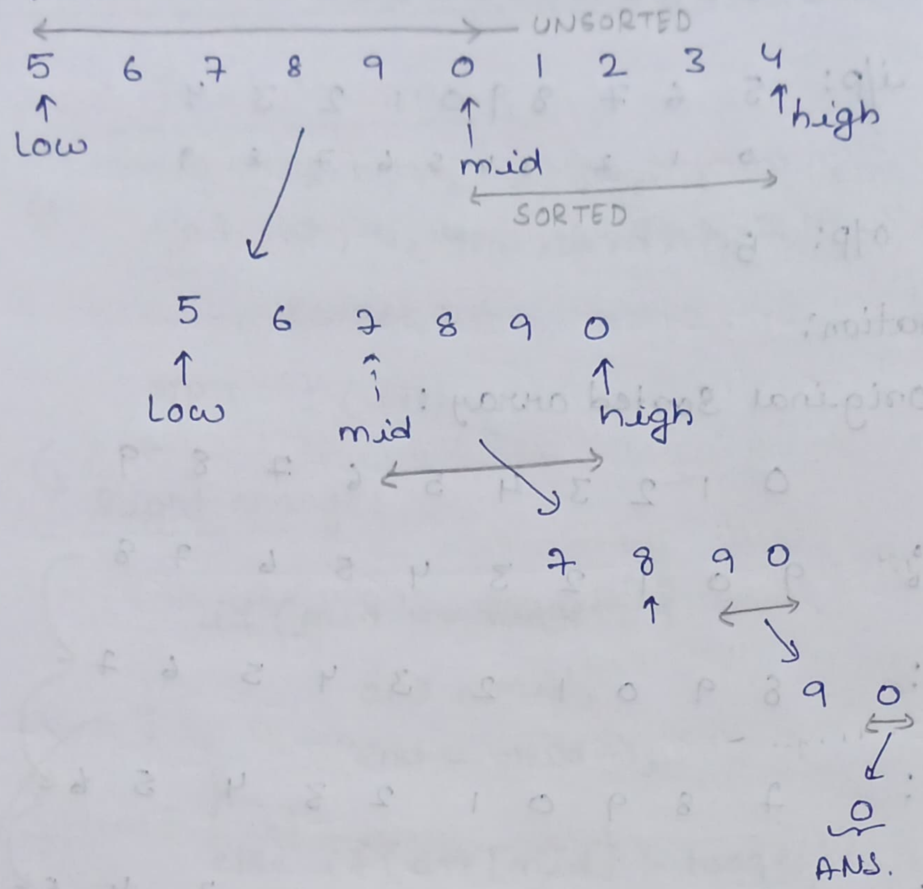
optimization:

given array: $\underbrace{5\ 6\ 7\ 8\ 9}_{\text{SORTED ARRAY}}\ \underbrace{0\ 1\ 2\ 3\ 4}_{\text{SORTED ARRAY}}$

↑ low ↑ mid ↑ high

in each iteration find the mid + move towards the un-sorted array to get the result : → log (N) → logarithmic complexity.

explanation:



code:

```
while (start < end){
        mid= start + (end - start) /2;
        if(arr[start] <= arr [mid]){
              start= mid+1;
        }
        else{
              end = mid-1;
        }
}
return start;
```

Note: The writtem code will give ERRORS / WRONG ANSWERS on multiple edge cases.

better code:

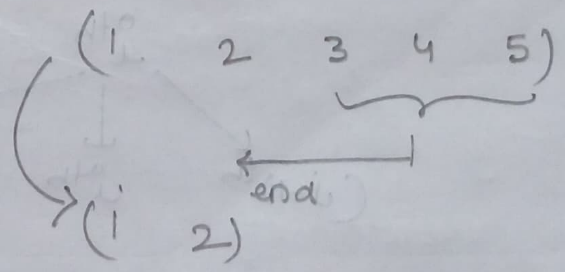
```
while (start < end){
        if (end == start +1) {
                return (arr[start] <=
                            arr[end]) ?
                        start : end;
        }

        if (arr[mid] <= arr[end]){
                end = mid;
        }
        else start = mid +1;

}
```

when are only two elements left for comparision, then it would become difficult to differ b/w start & mid.

comparing right condition first as we want to move left side as of our priority.

egde case :

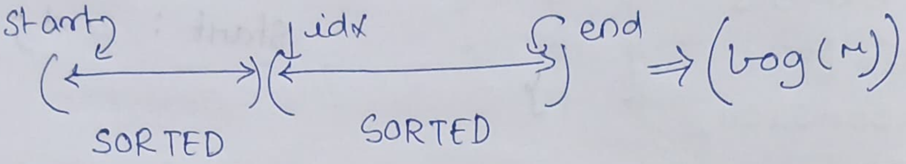
SORTED ARRAY



Find the element in a rotated sorted array:

→ find the index of the minimum element (no. of rotations)

→ determine in which range the target element lies. (from start → idx - 1 or to idx → end)

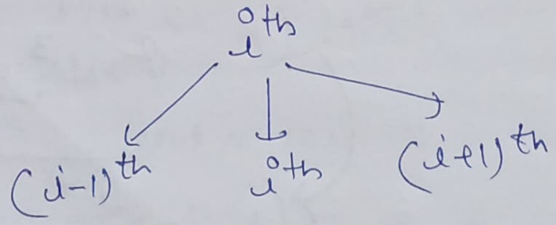


→ apply Binary Search on that specific range. ⇒ (log (N))

Searching in a nearly Sorted array:

Nearly Sorted Array :

any element to be present at ith position could be present at (i-1)th, ith, or (i+1)th position.



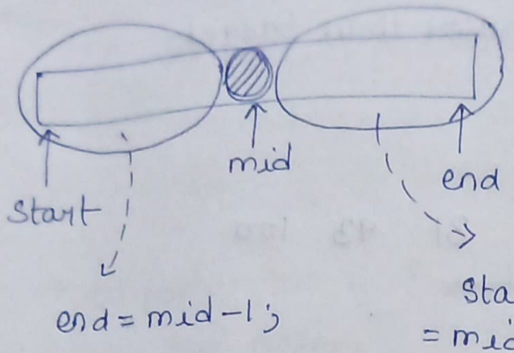
eg. 50 100 300 200 400.

original Array:	50	100	200	300	400
	0	1	2	3	4

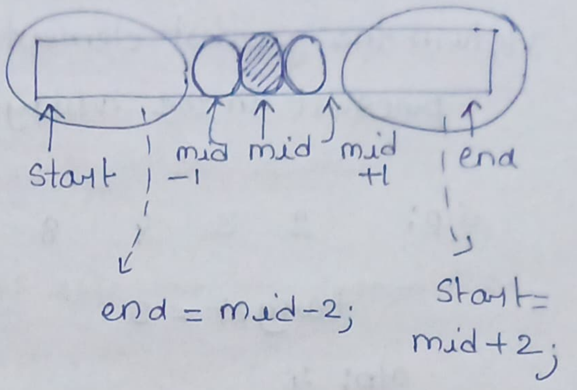
2th element → 3rd element

3rd " → 2th "

Sorted Array



Nearly Sorted Array

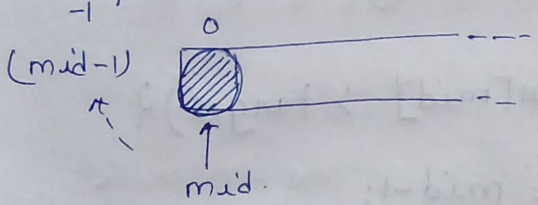


we will check for all three values, at the mid, at (mid-1) and at (mid+1).

```
if (arr[mid] == target) return mid;
else if (arr[mid-1] == target) return mid-1;
else if (arr[mid+1] == target) return
                                   mid+1;
```

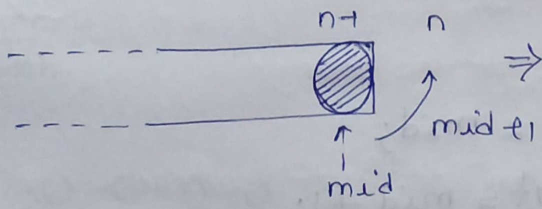
edge case:

①



⇒ Segmentation fault.

②



⇒ Segmentation fault.

```
if ((mid-1) ≥ start) - - - -
if ((mid+1) ≤ end) - - -
```

floor of an element in a sorted array:

return the greatest element smaller than target present in the array.

i/p: 2 3 4 8 11 31 93 100

target = 5

o/p: 4

code:

```
ans = -1;
while ( start <= mid) {
        mid = start + (end - start)/2;
        if (arr [mid] == target) {
                return mid;
        }
      * else if (arr[mid] > target) {
                end = mid-1;
        }
        else {
                ans = mid;
                start = mid +1;
        } *
}
return ans;
```

ciel of the element in a sorted array:

```
* else if( arr[mid] > target) {
          ans = mid;
          end = mid-1;
  }
  else    start = mid + 1; *
```

Next Alphabetical Element:

i/p: [a, b, c, f, h, i]

Key = f

o/p : h

→ to print the next alphabet after the key present in the array.

→ SOLUTION: Same as ceil problem just instead of comparing the value, will compare the ASCII weight of the alphabets given.

*

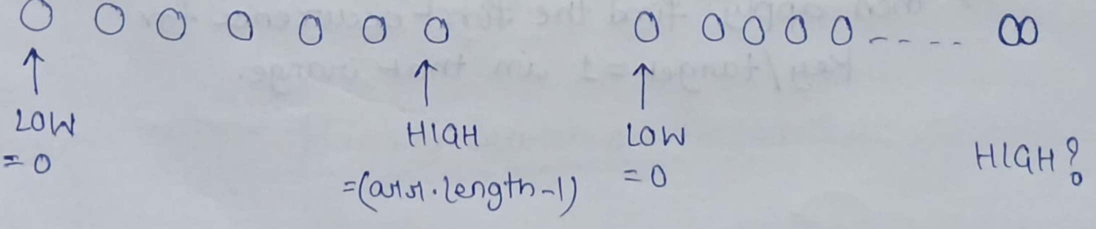
find the position of an element in an infinite large array : Popular Interview question

- - - . * * * * * * - - - (a) - - - * * * - -

to return the position of a given element.

Binary Search

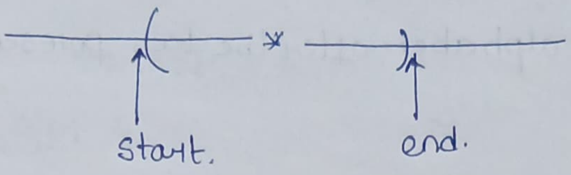
Modified Binary Search



```
        0  1
eg.     1  2  3  4  5  6  7  8  9  10  - - - - -  ∞
        ↑  ↑
        L  H  → initially                      (Key = 7)

    while (arr[H] < key) {
          H *= 2;  } → L = H;
    }
```

→ initialy we are setting the low at 0 amd high at 1.

→ them we are finding the range in which the key could join.



⇒ Log(N)

→ then apply binary search on that particular range.

⇒ log(N)

find the index of first 1 in the binary sorted array.

(contains only 0 & 1)

* the array is infinite:

Solution:

↳ the solution for search in the infinite large array

↳ to get the range.

↳ then apply find the first occurence for key/target = 1 in that range.

Minimum difference element in a sorted Array:

given a key, find the element in a given sorted array such that diff. b/w the key & the element will get mimimized.

eg. 4, 6, 10 key = 7

-7 -7 . -7

---

abs: 3 ① 3

↳ minimum → o/p: 6

Soln: we will get the minimum d/f with the help of ceil & floor value of 7.

Note: if 7 (key) is present in the array the min d/f = 0, hence return key.

else key is not present then return the min (abs (ceil - 7), abs(floor - 7))

↑
key.

↳ complexity: (log N + log N)

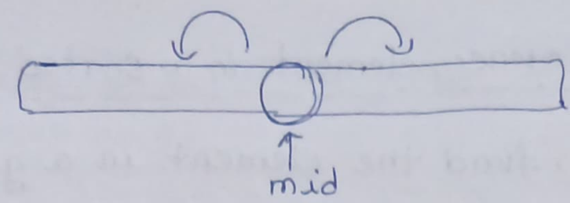
↑ find ceil ↑ find floor

Binary Search on Answers

Till now we have applied Binary Search on sorted array but now we will try to apply BS on an unsorted array.

if(arr[mid]==key) ⇒ (CRITERIA)

↑ we have to develop.



> this time we have to develop the criteria.

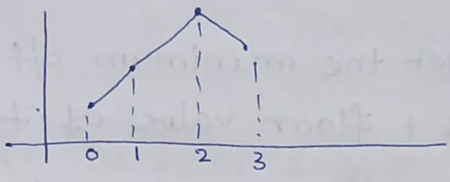
eg.

Peak Element:

i/p → arr:

0	1	2	3
5	10	20	15

o/p → index of the peak element. ⇒ 2.



- Peak element :- (i)th element > (i-1)th ethment

44 (i)th element > (i+1)th element

eg.2:

0	1	2	3	4	5	6
10	20	15	2	23	90	67

o/p: 1 or 5

eg 3: 1 2 3 4 5 x

↑ No element

o/p : 4

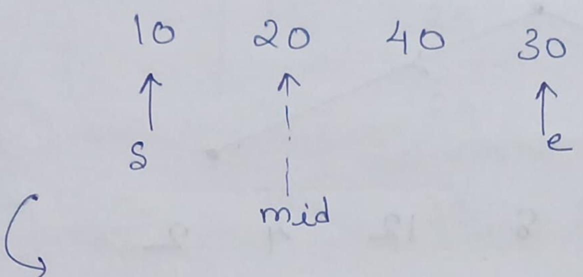
eg.4 x 5 4 3 2 1

↑ NO element

o/p : 0

Solution :-

↳ intution → we will always try to move to the greater element.



in this case we can see

arr [mid] > arr [mid -1] but

arr [mid] $\not>$ arr [mid +1]

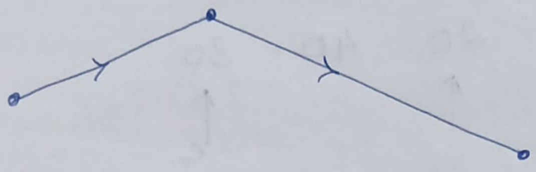
ie. arr [mid +1] > arr [mid] → then (start = mid +1).

∴ we changed the cretion as of :

```
if (arr[mid] == target)     →   if (arr[mid] >
return mid;                      arr[mid-1] && arr[mid]
                                 > arr[mid +1]).
                                 return mid;

else if (arr[mid] < target) →   else if (arr[mid] <
   start = mid+1;                arr[mid+1])
                                 Start = mid+1;

else                        →   else
   end = mid-1;                  end = mid-1;
```

Finding MAX element in a Bitonic array:

Bitonic Array:



eg. 1 3 8 12 4 2

this is the same question as of the finding the peak element.

Search in a Bitonic Array:

eg. i/p: 1 3 8 12 4 2

target : 4

approach: → find peak element (say the idex of peak element to be idx) ⇒ Log(N)

↳ apply BS in start → idx ⇒ log(N)

↳ apply BS (reverse order) in idx → end. ⇒ log(N)

over all complexity: log(N) + log(N) + log(N)

$= 3\log(N)$

$\boxed{= \log N^3}$

Search in a row-wise + column wise sorted matrix:

eg. arr[][]:

$$\begin{array}{cccc} 10 & 20 & 30 & 40 \\ 15 & 25 & 35 & 45 \\ 27 & 29 & 37 & 48 \\ 32 & 33 & 39 & 50 \end{array}$$

Sol^n:

Ptr (i, j)

$$\begin{array}{cccc} 10 & 20 & 30 & \boxed{40} \\ 15 & 25 & 35 & 45 \\ 27 & 29 & 37 & 48 \\ 32 & 33 & 39 & 50 \end{array}$$

developed criteria

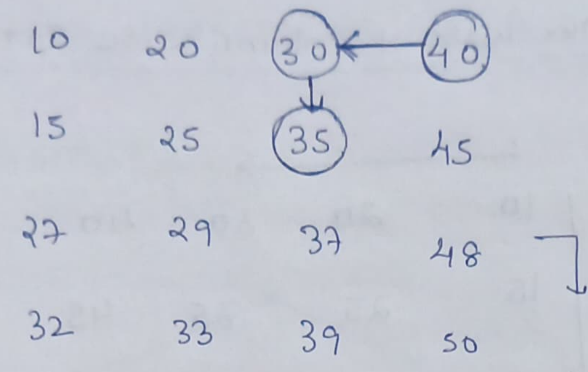
```
if (target < arr[ptr]) j--;
else if (target > arr[ptr]) i++;
else return ptr;
```

eg. target = 33.

$$\begin{array}{cccc} 10 & 20 & 30 & \boxed{40} \\ 15 & 25 & 35 & 45 \\ 27 & 29 & 37 & 48 \\ 32 & 33 & 39 & 50 \end{array} \rightarrow \begin{array}{cccc} 10 & 20 & \boxed{30} & \leftarrow \boxed{40} \\ 15 & 25 & 35 & 45 \\ 27 & 29 & 37 & 48 \\ 32 & 33 & 39 & 50 \end{array}$$

target < 40   target > 30

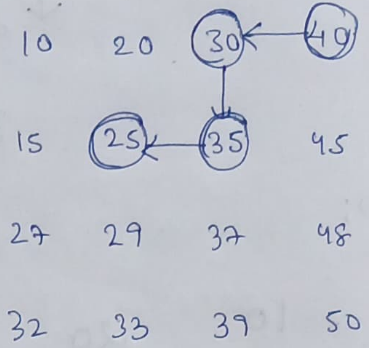
j--;   i++;



10	20	30	40
15	25	35	45
27	29	37	48
32	33	39	50

target < 35

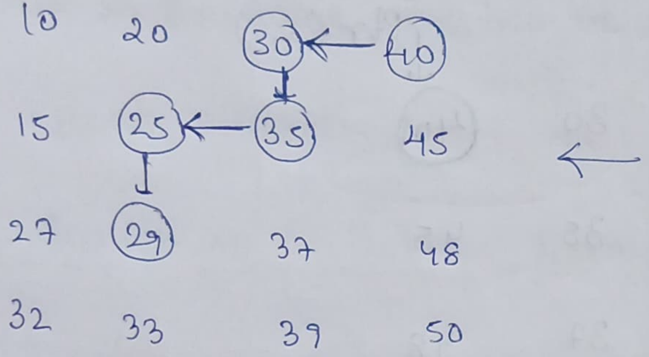
j--;



10	20	30	40
15	25	35	45
27	29	37	48
32	33	39	50

target > 25

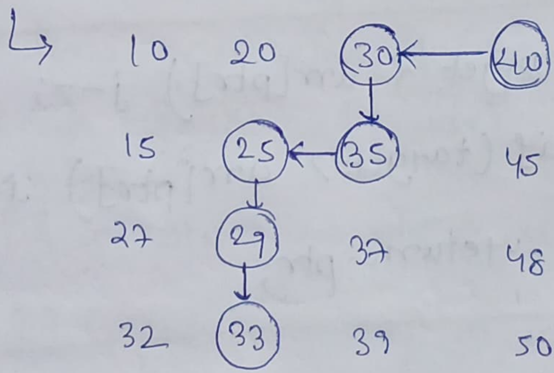
i++;



10	20	30	40
15	25	35	45
27	29	37	48
32	33	39	50

target > 29

i++.



10	20	30	40
15	25	35	45
27	29	37	48
32	33	39	50

target == 33

(found)

Allocate Minimum no. of pages: GOOGLE QUESTION

given n books with certain no. of pages in form of an array and some no. of students. Our task is to find the maximum no. of books given to a student is minimum.

Points to remember :-

① each student must get atleast one book.

② Book allocation will be in contigous order.

eg. n = 4 (books count)

arr [12, 34, 67, 90] (pages)

students = 2.

allocation could be in :-

Student 1	Student 2	(Max)
12	34 67 90	191
12 34	67 90	157
12 34 67	90	113

Min of all : 113.

∴ required allocation would be:

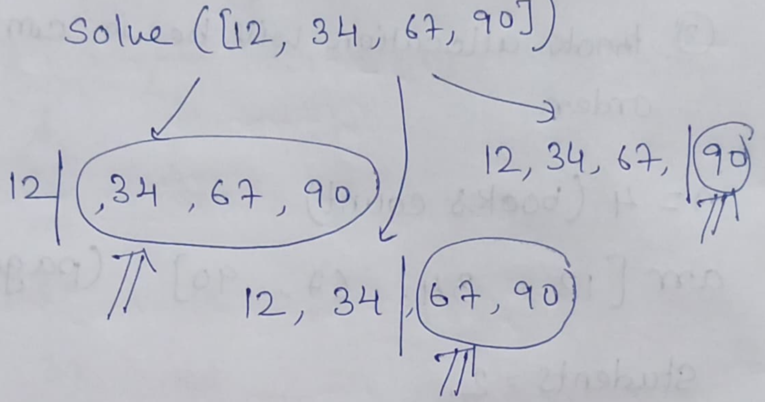
(12 34 67) (90)

Solⁿ :-

we can try applying recursion in this question; by checking where to partition the array.

eg. 12 34 67 90

like:



again apply recursion on the smaller arrays (⇈)

But this would take exponential time and even after applying DP that would take O(N²) space.

The most optimal Solⁿ we can get is by applying binary search.

But the array is not sorted then how ?

⇓

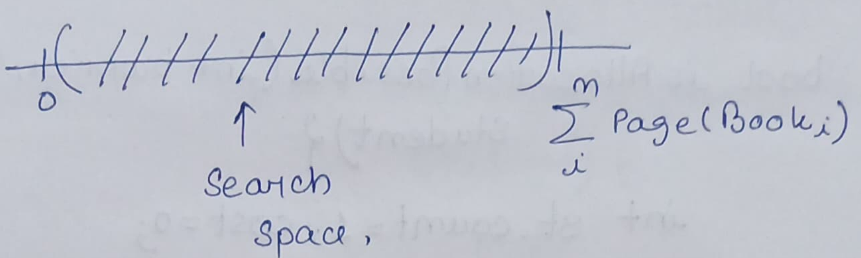
we can develop our own search space (array) that would be sorted and then we can get the answer.

what will be the least no. of pages we can allocate → zero (0) → students count = 0

Max no. of pages that we can allocate = $\sum_{i}^{n} \text{Pages}(\text{book}_i)$ = all pages.

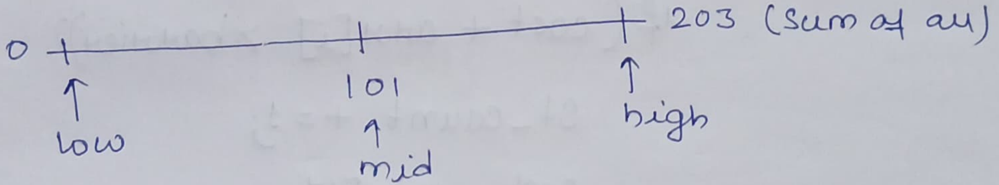
↳ student = 1.

Now, we know, that our answer would lie b/w those values.



Now we have developed our search space.

eg. 12 34 67 90



* { now try can i allocate max 101 pages to each student.

Yes

(high - 1)

↑

Minimization of the ans.

NO

start = ~~high~~ mid.

increasing the space so that we can allocate them.

* How to check.?

1 → 12 + 34 ← (checking if
2 → 67 ← the allocated
3 → 90 ← no. of pages should
not exceed
the max (101 eg).

as soon as they exceeds allocate to the new student.

↳ Code :-

```
bool is Allocation Possible (int barrier, int
                    Student){
    int st_count = 1, cost = 0;
    for (int i=0; i<n; i++){
          if (arr[i] > barrier) return
                         false;
          if (cost + arr[i] > barrier){
             St_count += 1;
             cost = arr[i];
          }
          else cost += arr[i];
    }
    return (st_count == student);
}
```

```
int solve(int arr[], int student){
    int high=0;
    for(int i=0; i<n; i++) high += arr[i
    int low=0; int res=-1;

    While(low <= high){
        mid = low + (high-low)/2;
        if(is AollocationPossible(mid,
                student){
            res = mid;
            end = mid-1;
        }
        else start = mid+1;
    }
    return res;
}.
```