Common Assumption: array sorted. (necessary?).

1. Classic. find an element in a sorted array.

```
// mid = left + (right - left) / 2;
update left / right based on comparison.
if  a[mid] < target
    left = mid + 1 !
```

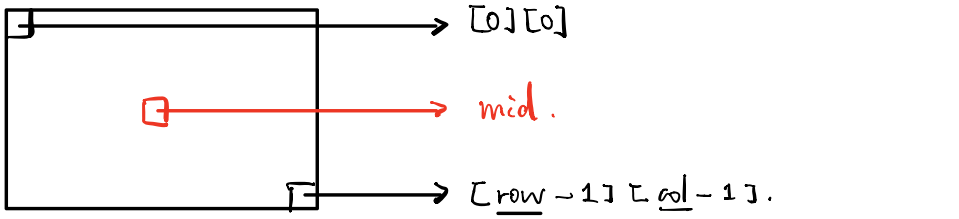
If narrow down to 2 elements.
left won't be updated. as (right - left) = 1.
left + (right - left) / 2 = left (mid)
will be INFINITE LOOP.

O(log(n))

2. Classic Search in 2D space.

Given a 2D matrix. Sorted each row. ascending.

! convert to 1D mathmetically.



left = 0; right = row * col - 1.

Convert mid to row & col.

```
row = mid / arr[0].length;
col = mid % arr[0].length;
```

O(log(m.n))

Principles: !!!.

#1. Must guarantee that search space decreases after iteration.

#2. Must guarantee that target cannot be ruled out accidentally when we update left / right.

3. Closest Element to Target.

rule out until have only 2 elements left.

compare left & right to decide which one is closer.

! while (right - left > 1).

! principle #2: target cannot be ruled out.

left = mid ; (NOT mid +/- 1).

right = mid ;

Post Processing: Use Math.abs. to compare.

(Target might be negative OR smaller/greater than both left & right).

4. First Target. (1st ocurrance of an element).

narrow down to 2 elements.

! if mid = target. update right = mid ;

otherwise everything else stays the same. ! (cannot merge mid > t & mid == t into one case)

Post-processing: compare left & right, then decide. edge case: 3 elements

! possible that target is neither left / right.

if (arr[left] == target) return left ;

else if right == target. return right;

else return -1;

5. last Target.

narrow down to 2 elements.

! instead of updating right. when mid == target.

update left = mid;

otherwise everything else stays the same.

Post-processing: compare left & right.

! first check right. if right == target, return right.

then check left. if left == target, return left.

else return -1;

Related: Total Ocurrance in a sorted array.

Same pre-processing — narrow down to 2. (1st ocurrance).

! post-processing: counting starting from arr[left] to the end.

6. Closest k Elements. new int[k];

narrow down to (2) elements. O(log(n))

! when updating:

if mid > target right = mid.

else left = mid.

! such that: narrow down to **Exactly** 2 elements

(unless array has **only** 1 element)

! so left != right

Post-processing: move k times O(k).

for k iterations.

if right is out of bound (right ≥ arr.length) OR

left is in bound (left ≥ 0) AND

abs diff (left) ≤ abs diff (right)

add left.

update left => left - 1.

else add right, right + 1.

Analysis: Time: O(log(n) + k).

if k is very large k → n.

O(log(n) + n) → O(n).

7. Smallest Element that's Larger than Target.

3 cases: element is smaller

element is equal sss...seee...eLLL...

element is larger. need to locate this.

if mid = target. left = mid

mid < target. left = mid.

(Not mid + 1. arr[mid + 1] might be larger than target)

mid > target. right = mid

(NOT mid-1, arr[mid-1] might be smaller than target).

Post-processing: CANNOT only check right

1st: check if right > target

2nd: check left > target, if yes return left
else return right.

3rd: otherwise return -1.

OR:

1st: check left > target, return left [larger, larger]

2nd: elif right > target, return right [equal, larger]
[smaller, larger]

3rd: return -1; [smaller, smaller]
or equal ...

L R

8. K-th smallest in 2 Sorted arr. (202)

① K iterations.

top arr1: [i → ...]

bot arr2: [j → ...]

move the smaller one & compare.

2 iter

3

0 4 1

if bottom is out of bound OR
top is in bound && top is smaller.
advance top ++;
else bot ++;

Post-processing: if one of the index is out of bound.

② binary + 9 TODO.

10. Binary Search w/ Unknown Size

Determine if a number is in this arr.

if arr[i] = null, then size is less than index.

Jump out : twice.

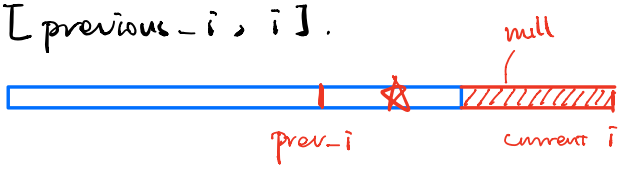
while arr[i] != null. && arr[i] < target.

double i.

Fall into cases either:

1: arr[i] is null. & target is in between.

[previous_i, i].



2: arr[i] is NOT null & target is in between

[previous i, i]

Post-Processing: Jump-in. (binary Search).

check right. arr[right] == null OR right > target.

right = mid - 1;

else if left < mid. (left = previous i. must NOT be null)

left = mid + 1;

else return mid;

Analysis:

	Jump Out	Jump In
10 times	$\log_{10}(n)$ Faster	$\log_2(10n)$
2 times	$\log_2(n)$	$\log_2(2n)$ Faster

Threshold n?

$$\log_a b = \frac{\log_c b}{\log_c a}$$

$$\log_{10}(n) + \log_2(10n) - (\log_2(n) + \log_2(2n))$$

$$= \log_{10}(n) - \log_2(n) + \log_2(10n) - \log_2(2n)$$

$$= \frac{\log_2(n)}{\log_2(10)} - \log_2(n) + \log_2(10n) - \log_2(2n)$$

$$(1/\log_2(10) - 1)\log_2(n) + \log_2(10) - \log_2(2) = \log_2(5)$$

$$= \left(\frac{1}{\log(10)/\log(2)} - 1\right) \cdot \frac{\log(n)}{\log(2)} + \frac{\log(5)}{\log(2)}$$

Set to 0 $= (\log(2) - 1) \cdot \log(n)/\log(2) + \log(5)/\log(2)$

$$\Rightarrow (\log(2) - 1) \cdot \log(n) + \log(5) = 0.$$

$$\Rightarrow (\log(2) - \log(10)) \cdot \log(n) = -\log(5)$$

$$\Rightarrow \quad \log\left(\frac{2}{10}\right) \cdot \log(n) \simeq \log(5^{-})$$

$$\Rightarrow \quad \log(5^{-}) \cdot \log(n) = \log(5^{-})$$

$$\Rightarrow \quad \log(n) = 1$$

$$\boxed{n = 10} \; !$$

negative

left < right

log(n) > right

Therefore, if $n > 10$, 10 times is faster.

if $n < 10$, 2 times is faster.

Tord Quicksort.

0	1	2	3	4	5	6	7	8	9	10	11
1.	2 .	2	2	4	5	8	13	13	13	13	16

↓

k = 3

i = 1

1 4 6

j = 1 x 2

2 3 8