Day 11 : Sorting & Searching

linear search → O(n)
binary search → O(log n)

Binary search & complex variations of that.

** (i) implement binary search → (for given range)
↓
(important to not mess this up)
↓
(should be very clear)

(ii) ternary search (array divided into 3 parts)
(neglect 1/3 array, to search in 2/3 array)

*) practical scenarios regarding binary search

↓ finding an element in a sorted array.

i) searching contact no:s in a phone directory.

ii) searching meaning of words in a dictionary

iii) git bisect

↓ by different people

When a lot of commits are made to a project, and after certain stage the team realises, that code / project is broken, error finding is done using binary search.

↓

At different commits, checking compilation or running and finding the point where the code broke

iv) helps searching for answers.

binary search pseudo code (in a given range of an array)

```
int binary_search(vector<int> A, int x, int l,
                  int r)
{
    int low = l;
    int high = r;
    while (low <= high)
    {
        int mid = low + (high - low) / 2;
        if (A[mid] == x) return mid;
        else if (A[mid] > x) high = mid - 1;
        else low = mid + 1;
    }
    return -1;
}
```

Q. Searching in a rotated array → a sorted array is rotated some no. of times

we don't know how many times

↓

(find x in rotated sorted array)

↓

given no duplicates in the array.

*

i) find pivot element (greatest element of the array)

ii) That allows us to split the array into two parts on which binary search can be implemented / called separately & max of each can be returned.

since one or both should give -1, other will be more.

pivot can also be found using binary search

↓

can be used to return the smallest or greatest element in a rotated sorted array as well.

↓

```
int pivot(vector<int> &A) {   (test on non-rotated array)
    int low = 0;
    int high = A.size() - 1;
    while (low <= high) {
        int mid = low + (high - low)/2;
        if (mid+1 == A.size() || A[mid+1] < A[mid])
        {
            return mid;
        }
        else if (A[low] <= A[mid])   → if the low element is less than mid element, then we search in the right side
        {
            low = mid+1;
        }
        else {
            high = mid-1;
        }
    }
}
```

by comparing with low, we can predict in which half the pivot is likely to be in.

(only when low = mid)

→ otherwise we search for the pivot on the left side

* application of binary search

duplicate elements? → won't work since an ambiguity can be caused in direction to undertake in case of duplicate presence.

(linear search problem)

1 5 5 5 5

↓

(5) 1 (5) (5) 5 → ambiguity in what decision to take at some index

Q. An array which has unique elements which first ~~increase~~ decreases & then increases. find the x around which this phenomenon occurs.

0	1	2	3	4	5	6	7
5	3	2	4	6	8	9	10

find mid → if the mid-1 & mid+1 are in increasing then search on left else search on right

if mid-1 and mid+1 are both larger than mid, then mid is the answer.

0	1	2	3	4	5	6	7
6	5	4	3	2	1	3	4

↓

we can even search in this array. since the array can be divided into two sorted arrays.

* Binary search for answers
  * sqrt(x)
  * modular exponentiation

$\log_2 N = x$

↓

need to find $\log_2 N$, so now we know that N can take values from 0 to N, so we binary search in that range. for x.

↓

(operating on floats as well)

$N < (32)$ → 32 bits used to represent numbers in C++

↓

(*) when we know the range of possible values, and we are able to recognise if our right answer is more or less than a given mid, then we can employ binary search in such cases.

EPS (machine epsilon) → floats in C.S. are not exactly accurate, so we need to define the workable accuracy around which we can work

↓

EPS is that accuracy.

↓

(f1 > f2 - EPS && f1 < f2 + EPS)

to check for f1 == f2

↓

in other ways abs(f1 - f2) < EPS

↓

∞ loops can occur due to float & doubles.

Q. In an unsorted immutable array, we need to find the kth smallest element in O(1) constant space.

(approach 1)

↓ iterate array k times to find the next higher element till the kth smallest element isn't reach

↓ O(kn)

↓ k can be of the order of $\frac{n}{2}$.

$\therefore O(n^2)$

↓ no duplicates

i) find mn, mx

ii) find mid

iii) elements smaller than mid = k-1

k or more than k

[low, mid-1]

k-2 or less

[mid+1, high]

check if mid is present in array, if not return next greater element?

i) when no duplicates

ii) when duplicates are present

$O(n \log N)$ — N: range of values; n: no. of elements in the array

< mid	< k
> mid	(n-k)

(NO duplicates)

1	5	2	15	18	9	11

mid = 3 5

3rd highest element

< mid = 2 → k

> mid = 5 → (k-1)

n-k = 7 - 2 = 5

then move higher up the range

n-k = 7 - 2 = 5 ✓

mid = 5

k_1 = < mid = 2

k_2 = > mid = 4

if not present then $k_1 + k_2 = n$

and we want $k_1 + k_2 = n-1$

↓ since the presence is guaranteed by knowing that one element is actually equal to the needed element

for duplicates :

<mid < k && >mid <= n-k
then return mid.

<mid < k && >mid > n-k
then [mid+1, high]

else [low, mid-1]

} try out cases

Q. Painter's partition problem

given boards of certain lengths and painters whose unit length painting speed is given.

B0 : 10
B1 : 12
:
BN : x

P0 → 2
P1 → 1
P2 → 10
:
PM → Y

constraints : (i) painters donot share boards to be painted
(no partial job by one painter)

(ii) painter can paint only contiguous boards
(any no from 0 to N, given other constraints are satisfied)

(iii) painters will pick boards to paint in order.

minimum time in which entire board set can be painted by the given set of painters.

case 2 : when M painters are available all take same time.

↓

answer → (0 ∞)

↓

we need to check possibility of being able to complete the work in the mid time

↓

for the mid time, we check if the req. painters <= k, if yes then return true, else return false

↓

if true, → we then search for a tighter bound.

[low, mid-1]

else → we search for higher bound [mid+1, high]

mid

B0 → (B0*1)

mid*(B0*1)

B1 → (B1*1)

if (B1*1) > mid-(B0*1)

↓

similarly keep moving.

```
bool isPossible(vector<int> B, int k, int mid, int T)
{
    int numPainters = 1; int sum = 0;
    for(i=0; i<B.size(); i++){
        sum += B[i]*T;
        if(sum > mid){
            numPainters++;
            sum = 0;
            i--;
        }
    }
    if(numPainters > k) return false;
}
if(numPainters <= k) return true;
}
```

* include a while loop in to take case of num-painters. (more intuitive) ←

* callside

```
int ans = -1 // to take care of previous ans
int low = 0;
int high = ∞;
while (low <= high){
    int mid = low + (high-low)/2;
    if (possible(mid)){
        ans = mid
        high = mid-1;
    }
    else{
        low = mid+1;
    }
}
return ans;
```