Array

⊗ Longest Subarray with Sum K :-

Given an array arr[] containing integers & an integer k, your task is to find the length of longest subarray where the sum of its elements is equal to the given value k. If there is no subarray with sum equal to k, return 0.

Ex:

I/P: arr = [10, 5, 2, 7, 1, -10]
k = 15

O/P: 6 → [5, 2, 7, 1], [10, 5], [10, 5, 2, 7, 1, -10]

Sol: We shall use similar approach of the problem "Number of subarrays whose sum = k". At each i, check if sum = k, if yes, then calculate result. Otherwise check if (sum - k) is there or not. We are using map to store the sum & corresponding index. If it exists, then calculate result based on that length. i.e i - mp[sum - k].

At the end to tackle negative value check if sum exists or not, if it doesn't exist then only update index; else keep the previous one, as this will yield longer length.

c++ code

```
int longestSubarray(vector<int>& nums, int k){
unordered_map<long long, int> mp;
int n = nums.size(), result = 0;
long long sum = 0;
for (int i=0; i<n; i++) {
sum += nums[i];
if(sum == k) result = max(result, i+1);
if(mp.find(sum-k) != mp.end())
        result = max(result, i - mp[sum-k]);
if (mp.find(sum) == mp.end())
        mp[sum] = i;
}
return result;
}
```

⊗ There is also 2 pointers approach which will work only for +ve numbers.

Longest consecutive sequence

Given an unsorted array of integer nums, return the length of the longest consecutive elements seq.

T.C: O(N).

Ex

I/P: 100, 4, 200, 1, 3, 2

O/P: 4 (1, 2, 3, 4.)

Solution: We will store all the array elements in a map or set.

Now, we shall try to find a sequence, we shall try to find, if x is the beginning of a sequence or not, we can do this by checking if (x-1) exists or not, if it exists then x cannot be the element of a starting index.

But, if x is the beginning we keep on checking if x+1 exists in the array. And each time we found x+1 in array we increment counter & compute result.

We will store in map or set to avoid iterating over duplicate elements also to find (x+1) in constant time.

C++ code

```
int longestConsecutive(vector<int>& nums) {
int n = nums.size(), res = 0;
unordered_map<int, int> mp;
for(int i=0; i<n; i++) mp[nums[i]]++;
for(auto it : mp) {
int j=0, curr=0, x = it.first;
if(mp.find(x-1) == mp.end()) {
while(mp.find(x+j) != mp.end) {
j++; curr++;
}
res = max(res, curr);
}
}
return res;
}
```

4Sum

Given an array nums of n integers, return an array of all the unique quadruplets [nums[a], nums[b], nums[c], nums[d]] such that,

(i) $0 <= a, b, c, d < n$

(ii) $a != b != c != d$

(iii) nums[a] + nums[b] + nums[c] + nums[d] == target.

Return ans in any order.

Sol :-

① Sort the array :- To avoid duplicacy & to use 2-pointer.

Also, helps to check sum & skip duplicates.

② 2 nested loops to fix the first two elements.

i → 0 . . . n-3
j → i+1 . . . n-2
} every time skip for duplicate elements i.e. i++ if (nums[i] == nums[i-1]).

③ 2 pointer technique for finding other 2 elements :-

(i) For each pairs (nums[i], nums[j]) it's a 2pointer problem where sum = target - (nums[i] + nums[j])

(ii) 2 pointer, k = j+1
l = n-1

The loop will run till (k < l)

(iii) Here also check for duplicate

Increment or decrement k, l.

(iv) if ans is found store it.

(v) Otherwise, if (nums[k] + nums[l]) < sum increment k else decrement l.

C++ code1

```
vector<vector<int>> 4Sum(vector<int>& nums,
                         int target) {
vector<vector<int>> ans;
int n= nums.size();
if(n<4) return ans;
for(int i=0; i<n-3; i++) {
if(i>0 && nums[i]==nums[i-1]) continue;
 for(int j=i+1; j<n-2; j++) {
  if(j>i+1 && nums[j]==nums[j-1]) continue;
   int k= j+1, l=n-1;
long long sum=(long long) target-(nums[i]+
                                  num[j]);
  while(k<l) {
if(nums[k]+nums[l]==sum) {
ans.push_back({nums[i], nums[j], nums[k],
                                  nums[l]});
while(k<l && nums[k]==nums[k+1]) k++;
while(k<l && nums[l]==nums[l-1]) l--;
   k++;
   l--;
} else if (nums[k]+nums[l]<sum) {
   k++;
 } else {
   l--;
 }
}
}
 return ans;
}
```

Binary Search

① Floor in a Sorted Array

Given a sorted array arr[] with distinct elements and an integer K, find the index (0-based) of the largest element in arr[] that is less than or equal to K. The element is called "floor" of K. If such an element doesn't exist, return -1.

Ex: [1, 2, 8, 10, 11, 12, 19] K=0 ⇒ -1

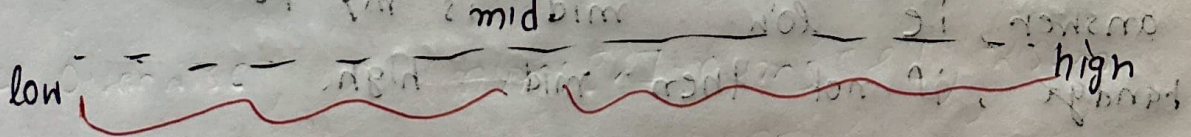
" K=5 ⇒ 1 (arr[1]=2)

Solution: In Binary search ⇒

(i) search space must be sorted

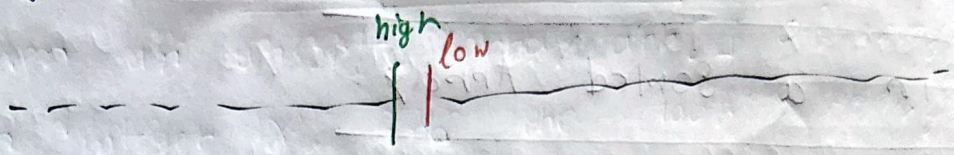
(ii) There will be a range of possible answers among which one will be final answer

(iii) Key factor is we need to shrink the search space gradually.



(iv) At each space we shall try to delete one of the halves of ~~the~~ based on some given condition.

(v) At the end



low will cross high.

(vi) Based on condition one of the range low → mid or mid → high will be the range of possible answers.

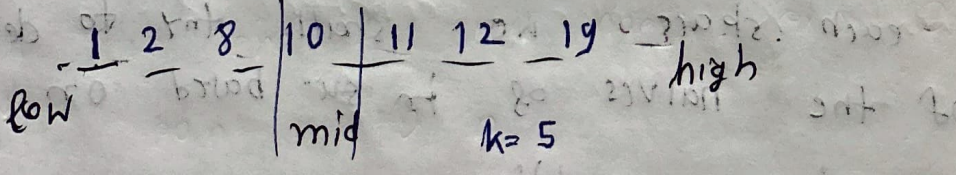
(vii) low & high will initially be either possible - non possible / non possible - possible

(viii) At the end they will be to the opposite side. suppose at start if low was in not-possible zone & it reaches possible zone after search it will be the answer; same for high.

We need to identify in which region low & high exists.

lower bound

Here we can check if arr[mid] <= k, if true which means mid might be my possible answer, i.e low ... mid is my possible range, if not then mid ... high.



Here, arr[mid] > k, now, mid+1 ... high

will also be >k so, our answer will be in low → mid-1, so high = mid-1.

at the end high will point to the answer.

Line: if arr[mid] <= k, mid might be my answer & we got best possible answer from low → mid, so we eliminate & check mid+1 → high, at the end high will choose low & will point to largest possible answer.

c++ code

```
int findFloor(vector<int> &arr, int k) {
int n = arr.size(), low = 0, high = n-1;
int mid = (high + low) /2;
while (low <= high) {
    int mid = (high + low) /2;
    if (arr[mid] <= k) low = mid+1;
    else high = mid-1;
}
return high;
}
```

(*) If floor doesn't exist, high will go to -1.

Ceil the floor

Given an unsorted array arr[] of integers & an integer x, find the floor & ceiling of x in arr[].

Floor of x is the largest element which is smaller than or equal to x. Floor of x doesn't exist if x is smaller than smallest element of arr[].

Ceil of x is the smallest element which is greater than or equal to x. Ceil of x doesn't exist if x is greater than greatest element of arr[]. If doesn't exist return -1.

I/P: [5, 6, 8, 9, 6, 5, 5, 6] k=7 O/P => 8

Solution: Here we shall use the same approach, But search space & condition will change.

Earlier, we were trying to find,

low -- mid => so we returned high,
arr[mid] >= k,

Now, we need to check arr[mid] >= k
& low will be in range mid -- high
after search so, we return low,

At the end, if ceil doesn't exist low will point to n.

C++ code

```
int findCeil(vector<int>& arr, int k) {
int n = arr.size(), low=0, high=n-1;
int mid = (high+low)/2;
while (low<=high) {
  int mid = (high+low)/2;
  if(arr[mid]<k) low = mid+1;
  else high = mid-1;
}
return low;
}
```

Search Insert position

Given a sorted array of distinct integers & a target value return the index if target is found. If not, return the index where it would be if it were insented in sorted order.

Ex1 1 3 5 6 ⇒ 5

o/p ⇒ 2 1 3 5 5 6

target = 2 o/p ⇒ 1.

1 2 3 5 6

Soln This is same basically finding floor But with minor modification.

(i) Within floor function, if we find target nums[mid] == target ⇒ return mid.

(ii) From main function, if

lower_bound == -1, return 0

if ~~nums[index] >= target return target~~

else return index + 1;

c++ code

Find first & last position of element in sorted array

Given an array of integers nums sorted in non-decreasing order, find the starting & ending position of a given target value.

If target value is not find return [-1,-1]

Sol: we shall use the same code as floor & ceil, just need to check if low & high are equal to target or not.

C++ code:

```
vector<int> searchRange(vector<int>& nums, int target){
vector<int> ans(2);
sort(nums.begin(), nums.end());
int low= findFloor(nums, target);
int high= findCeil(nums, target);
if(low != -1 && nums[low] != target) low=-1;
if(high != -1 && nums[high] != target) high=-1;
ans[1]= low;
ans[0]=high;
return ans;
}
```

we are inter

Number of occurence.

Given a sorted array, return number of occurences of a given number target.

Ex: 1, 1, 2, 2, 2, 2, 3 target = 2

O/P: 4

Sol: We will use the prev prob of finding first & last occurence.

c++ code:

```
int countFreq(vector<int>& arr, int target){
vector<int> res = searchRange(arr, target);
if(res[0] != -1) return res[1] - res[0] + 1;
return 0;
}
```

Search in a Rotated sorted array.

I/P: 4 5 6 7 0 1 2 target = 4.
(Array elements are distinct)

Sol: Since the array is sorted we can apply binary search, At any point of time, one half of the array must be sorted i.e either low -- mid / mid -- high.

And we shall check in sorted part & will try to find out if element exists in that part or not.

If ~~in sorted part~~ i.e

Eg lets say, low -- mid is sorted.

if nums[low] ~~<~~ target < nums[mid]

we shall search in low -- mid-1 only,

i.e high = mid-1, otherwise we eliminate this half. i.e low = mid+1

similarly if we check for other half if that was sorted

C++ code

```
int search (vector <int> &nums, int target) {
int low =0, high = nums.size() -1;
while (low <= high) {
  int mid = (low + high) /2;
  if (nums [mid] = target) return mid;
if (nums [low] <= nums [mid]) { //Find sorted part
if (target < nums [mid] && target >= nums[low])
        high = mid - 1;
else    low = mid+1;
} else {
if (target > nums [mid] && target < nums [high])
        low = mid+1;
else    high = mid-1;
}}
  return -2;
}
```

Search in a Rotated Sorted Array II

Same question, only difference is we have duplicates now.

Ex: 4 5 5 6 7 0 1 2 3

or

~~4 4 4 5 6 7 0 2 3 4 4 4~~

low mid high

4 4 5 6 4 4 4 4 4 (target = 5)

Sol: Here we are not able to identify which portion is sorted (2nd array) cause nums[low] ≤ nums[mid]

also nums[high] ≥ nums[mid]

In such a situation we add an extra conditions

```
if( nums[low] == nums[mid]
    &&
  nums[mid] == nums[high] )
{
    low++;
    high--;
    continue;
}
```

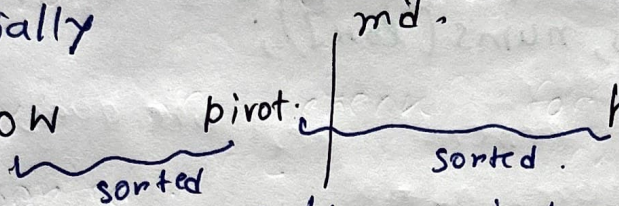
We continue because, if execute the rest code without ensuring, it may go to infinite loop.

Find minimum in Rotated sorted array (unique elements)

I/P: 6 7 8 9 1 2 3 4 5 O/P ⇒ 1

Sol: Here ~~also~~ we will follow a different approach.

Initially



we need pivot / peak index + 1,

we check. if low - - mid is sorted or not, if sorted we ~~store~~ check if low can be our ans or, then we are sure that we checked lowest among low - - - mid, so, we eliminate & move low = mid + 1.

Other wise, if mid - - high is sorted, we check the lowest in this half. i.e mid & simply eliminate this half. i.e high = mid - 1.

C++ code:

```
int findMin(vector<int>& nums) {
int low=0, high = nums.size() - 1, ans=INT_MAX;
while (low<= high) {
 int mid = (high+low)/2;
 if (nums[low]<= nums[mid]) {
    ans = min(ans, nums[low]);
    low = mid+1;
 } else {
    ans = min( ans, nums[mid]);
    high = mid-1;
 }}
   return ans;
}
```

Find out how many times the array is rotated

	0	1	2	3	4	5	6	7	8	9
I/P 1	6	7	8	9	10	2	3	4	5	6

O/P 1: 4

	0	1	2	3
I/P 2	6	9	2	4

O/P 2: 2

Sol: If we observe indexes, the index of minimum element in the array is the number of times if is rotated,

so we can use the previous code,

but while calculating ans, we shall store ~~minimum~~ index & will return it,

That is the only change

Single Element in a Sorted Array

I/P: 1 1 2 3 3 4 4 8 8
0 1 2 3 4 5 6 7 8

O/P: 2

I/P: 1 1 2 2 3 3 4 5 5 6 6
0 1 2 3 4 5 6 7 8 9 10

O/P: 4

⊗ Only 1 single element rest are duals,

Sol: Note: Before single element it was even-odd pair, after single elem odd → even

① Base case:

(i) if nums[mid] != nums[mid-1]
&& nums[mid] != nums[mid+1]

② if nums[mid] == nums[mid-1]

⇒ mid is not the unique element,

Determine if we need to go left or right.

if (mid-1) % 2 == 0, && arr[mid-1] == arr[mid]

* means it's an even-odd pair, so single element is in it's right side,

so, go to right i.e. low = mid+1

if not, that means, nums[mid] = nums[mid-1]
but pair is odd-even, then we need to go left, high = mid-1,
similarly we need to check for mid, mid+1, if they are equal.

Corner case: If single element is at 0 or n-1, then we need handle seperately.

C++ code:

```
int singleNonDuplicate(vector<int>& nums) {
int low=0, high = nums.size()-1, ans;
if (high==0) return nums[high];
if (nums[low] != nums[low+1]) return nums[low];
if(nums[high] != nums[high-1]) return nums[high];
while (low <= high) {
    int mid = (low+high)/2;
if (nums[mid] != nums[mid-1]) && (nums[mid] != nums[mid+1])
    return nums[mid];
if (nums[mid] == nums[mid-1]) {
  if ((mid-1)%2==0) low = mid+1;
  else high = mid-1;
} else {
  if ((mid+1)%2==0) high = mid-1;
  else low = mid+1;
}}
return -1;
}
```

Find peak element

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

Given a 0-indexed integer array [nums], find a peak element, & return its index. If the array contains multiple peaks, return the index to any of the peaks.

Note: nums[-1] = nums[n] = -∞.

I/P1: 6 7 8 9 1 2 3 4

O/P2: ~~1 2 3 4 5 6 7 8~~

I/P2: 1 2 1 3 5 6 4

O/P2:

I/P3: 9 8 7 5 2 8

O/P1:

Sol: same approach like previous, handle first & last element separately & check from 1 - - n-2.

Base case: nums[mid] > nums[mid-1] && nums[mid] > nums[mid+1] ⇒ return mid

Recursion: if (nums[mid] > nums[mid-1])

means mid-1, mid, mid+1 is sorted

We need to find greater element so shift to low = mid+1

otherwise, i.e.

nums[mid-1] > nums[mid] > nums[mid+1]

we need to go left to find the greatest element.

c++ code:

```
int findPeakElement(vector<int>& nums) {
int low=0, high=nums.size()-1;
if(high==0)
if (nums[low] > nums[low+1]) return low;
if (nums[high] > nums[high-1]) return high;
low++, high--;
while(low<=high) {
int mid = (low+high)/2;
if (nums[mid] > nums[mid-1] &&
   nums[mid] > nums[mid+1]) return mid;
   if (nums[mid] > nums[mid-1]) low=mid+1;
   else high=mid-1;
}
return low;
}
```

Square root

Given a number n return floor($\sqrt{n}$)

I/P: 9 11

O/P: 3 3

Soln Here also, it is about search space.

We can be sure that answer is in range 1 to n/2+1

1 2 3 -- $\lfloor\sqrt{n}\rfloor$ $\lfloor\sqrt{n}+1\rfloor$ -- n/2 n/2+1

low high

mid

if (mid x mid) > n then we can be sure that answer can't be there from mid .. high, so, high = mid-1, else move low → mid+1,

At the end, low will cross high, so high will move to not possible range to possible. so at the end high will store the answer.

C++ code:

```
int floorSqrt(int n) {
long long low = 1, high = n/2+1;
while (low <= high) { // use long long because mid^2 can be larger than INT_MAX
long long mid = (low+high)/2;
if( pow(mid,2) > n) high = mid-1;
else low = mid+1;
}
return high;
}
```

Find nth root m

You Find $\sqrt[n]{m}$, if not found return -1.

Sol: We can find the same strategy, However there can be 2 problem,

① Overflow ⇒ To handle this even long long

② will be overflowed n^m, so, to handle this we will calculate $1 \times n \times n \cdots$ m times $\times n$ & we will stop whenever product crosses the target value & return a value to indicate that this is not possible,

at the end of calculation, if product == target, return a value to understand we found answer.

C++ code:

```
int power(int n, int m, int target) {
  long long prod = 1;
  for (int i=1; i<=m; i++) {
    prod = prod * n;
    if (prod > target) return 2;
  }
  if (prod == target) return 1;
  return 0;
}
int nthRoot(int n, int m) {
  long long low=1, high=m;
  while (low<=high) {
    long long mid = (low+high)/2;
    int ans = power(mid, n, m);
    if (ans == 1) return mid;
    if (ans == 2) high = mid-1;
    else low = mid+1;
  }
  return -1;
```