Hashing

→ Two sum problem -

Given an array of integers nums & an integer target return the indices of 2 nos. such that they add up to target. You may assume that each input would have exactly one solution & you may not use the same element twice. Return the answer in any order.

Soln

i/p: 2, 6, 5, 8, 11 & target = 14
(0, 1, 2, 3, 4)
↳ not sorted

ans = <1, 3>

optimize

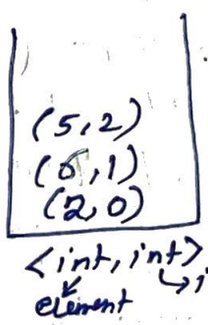
0	1	2	3	4
2	6	5	8	11

① ∴ 14 − 2 = 12 ↳ not exist in hash table ∴ put 2 in map along with index

② 14 − 6 = 8 ↳ ✗ ∴ put in map

③ 14 − 5 = 9 ↳ ✗ ∴ put in map

④ 14 − 8 = 6 ↳ exist in map and it at index 1 (from map we have) ∴ (1, 3) ↳ ans



$TC = O(n * 1)$ ↳ unordered map ↳ WC = O(n) when # collisions is more (rare)

$SC = O(n)$ ↳ as all values can be in map

PTO

```
vector<int> twosum (vector<int>& nums, int target)
{   int n=nums.size();
    vector<int> ans;
    unordered_map<int,int> m;
    for(int i=0;i<n;i++)
    {
        if(m.find(target-nums[i])!=m.end())
        {
            ans.push_back(m[target-nums[i]]);
            ans.push_back(i);
            return ans;
        }
        m[nums[i]]=i;
    }
    return ans;
}
```

→ **4 Sum -**

Given an array nums of n integers & an integer target, are there elements a, b, c, d in nums such that a+b+c+d = target?
Find all unique quadruplets in array which gives the sum of target.
Note - the solution set must not contain duplicate quadruplets.

Sol^n -

i/p: [1, 0, -1, 0, -2, 2] & target = 0

o/p = [[-1, 0, 0, 1], [-2, -1, 1, 2], [-2, 0, 0, 2]]

Approach

optimal

PTO

i/p: 4, 3, 3, 4, 4, 2, 1, 2, 11 & target

↓ sort

1, 1, 1, 2, 2, 3, 3, 4, 4, 4

We will use two pointers i & j
& two variables for two sum l & r

(since sorted)

①

0	1	2	3	4	5	6	7	8	9
1	1	1	2	2	3	3	4	4	4
i	j	~~l~~	t	t	t	t r	l r	~~r~~	~~r~~

rem = 9 - a[i] - a[j]
= 9 - 1 - 1
= 7 → look from l to r for 7 using (2 sum)

- 5 < 7 ↑ l
- 2 + 4 < 7 ↑ l
  & since at ind = 3 → 2 is there no need to check for 2 at 4 ∴ go till we get diff element
- 3 + 4 == 7
  ∴ we get 1st ans (1, 1, 3, 4) ✓
- Already computed for 3 & 4, so go for diff element now l > r
  ∴ stop

② now, i = 0
j = 1 → has val = 1
∴ go for element having val diff than 1

0	1	2	3	4	5	6	7	8	9
1	1	1	2	2	3	3	4	4	4
i	j	j	j	t	t r	~~r l~~	~~r~~	~~r~~	~~r~~

rem = 9 - a[i] - a[j]
= 9 - 1 - 2
= 6
↳ look for 6 using 2 sum

- 2 + 4 == 6
  ∴ we get 2nd ans (1, 2, 2, 4) ✓
  now, move l & r for diff value
- 3 + 3 == 6
  ∴ we get 3rd ans (1, 2, 3, 3) ✓
  now, move l & r for diff value
  now l > r
  ∴ stop

as asked for unique quadruplets

Note: When 'j' is moved → avoid duplicates
Similarly 'i' is moved → " "
Same for 'l' & 'r'

Similarly do this

here, $i=0$, $j=i+1$, $l=j+1$, $r=n-1$ } initially

$TC = O(n^3) + O(n \log n)$ → to sort

$SC = O(m)$ → to store ans, say #ans = m

```
vector<vector<int>> fourSum(vector<int>& num, int target)
{
    int n = num.size();
    vector<vector<int>> ans;
    sort(num.begin(), num.end());
    for(int i=0; i<n; i++)
    {
        if(i>0 && num[i]==num[i-1])
            continue;
        for(int j=i+1; j<n; j++)
        {
            if(j>i+1 && num[j]==num[j-1])
                continue;
            int k=j+1;
            int l=n-1;
            while(k<l)
            {
                long long sum = (ll)num[i] + (ll)num[j] + (ll)num[k] + (ll)num[l];
                if(sum==target)
                {
                    vector<int> temp = {num[i], num[j], num[k], num[l]};
                    ans.push_back(temp);
                    k++;
                    l--;
                    while(k<l && num[k]==num[k-1])
                        ++k;
                    while(k<l && num[l]==num[l+1])
                        --l;
                }
                else if(sum<target)
                    ++k;
                else
                    --l;
            } // end of while()
        }
    }
    return ans;
}
```

agar integer overflow karaha hai use long long

3 Sum -

Given an array nums of n integers, are there elements a, b, c in nums such that a+b+c = 0? Find all unique triplets in the array which gives the sum of zero.

Note: the solution set must ~~not~~ contain unique triplets.

i/p: nums = [-1, 0, 1, 2, -1, -4]

o/p = [[-1, -1, 2], [-1, 0, 1]]

Soln

i/p: [-1, -2, -2, -1, -1, 2, 0, 2, 0, 0, 2]

↓ sort

[-2, -2, -1, -1, -1, 0, 0, 0, 2, 2, 2]

we will use 2 pointer approach

Note $a+b+c = 0$ → target

keep 'a' constant

$\therefore b+c = -a$

↳ boils down to 2 sum problem

$TC = O(n^2) + O(n \log n)$ (for sort)

$SC = O(m)$ ↳ to store ans, say #ans = m

similarly as done in 4 sum

$i = 0$, $l = i+1$, $j = n-1$ } & jab kaam hojaye ↑ i, l, j to avoid duplicates

```
vector<vector<int>> threeSum(vector<int> &num)
{
    sort(num.begin(), num.end());
    vector<vector<int>> res;
    for(int i=0; i<num.size(); i++)
    {
        if(i>0 && num[i]==num[i-1]) continue;
        int l=i+1, r=num.size()-1;
        while(l<r) //two sum
        {
            int sum = num[l] + num[r] + num[i];
            if(sum<0)
                ++l;
            else if(sum>0)
                --r;
            else if(sum==0)
            {
                vector<int> temp={num[i], num[l], num[r]};
                res.push_back(temp);
                ++l; --r;
                while(l<r && num[l]==num[l-1]) ++l;
                while(l<r && num[r]==num[r+1]) --r;
            }
        }
        while(i+1<num.size() && num[i]==num[i+1]) ++i;
    }
    return res;
}
```

→ Longest Consecutive Sequence -

Given an unsorted array of integers, find the length of the longest consecutive elements sequence.

Your algorithm should run in O(n) time.

Soln

102, 4, 100, 1, 101, 3, 2, 2, 1, 1

↓ put all elements in unordered set (to avoid duplicates)

set{102, 4, 100, 1, 101, 3, 2, 1}

now, instead of iterating through array, iterate through set as it has no dup ∴ less elements

- 102 → is 101 (102-1) present in set? Yes
  ∴ ignore
- 4 → is 3 present? ↳ ignore
- 100 → is 99 present?
  No.
  ∴ count = 1 (for 100)
  curr = 100
  ↓ now iterate through set & check for consecutive nos.
  we get 100, 101, 102
  ∴ count = 3 & put in maxi = 3
- 1 → is 0 present? X
  ∴ count = 1
  curr = 1
  ↓ iterate through set & check for consecutive nos.
  ∴ we get 1, 2, 3, 4 & count = 4
  ∴ maxi = max(3, 4) = 4
- 101 → is 100 present? ↳

Similarly do this

$TC = O(n) + O(n) + O(n) = O(n + 2n)$

- $O(n)$ → to put elements in set
- $O(n)$ → # unique elements in set = n (say) & to iterate over it
- $O(n)$ → to check longest for consecutive sequence

$SC = O(n)$ → set

```
int longestConsec(vector<int> &nums)
{
    unordered_set<int> st;
    for(auto it : nums)
    {
        st.insert(it);
    }
    int maxi = 0;
    for(auto it : st)
    {
        if(st.count(it-1) == 0)
        {
            int count = 1;
            int curr = it;
            while(st.count(curr+1))
            {
                ++count;
                ++curr;
            }
            maxi = max(maxi, count);
        }
    }
    return maxi;
}
```

→ Largest subarray with zero sum -

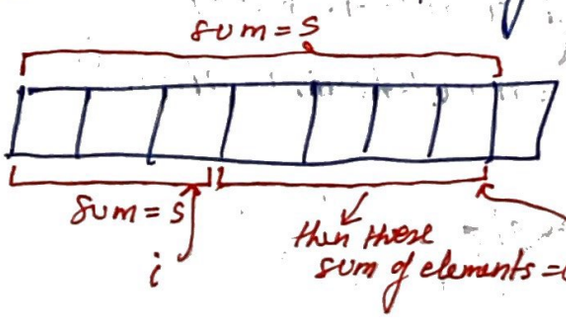
Given an array having both +ve & -ve nos., find the length of the largest subarray with sum = 0.

i/p : {15, -2, 2, -8, 1, 7, 10, 23}

o/p : 5 → as largest subarray is {-2, 2, -8, 1, 7}

Soln

Optimal We will use concept of prefix sum. i.e.



∴ subarray length = $j - i$ → how recognize

we will use map & maxi = 0

i/p: 1, -1, 3, 2, -2, -8, 1, 7, 10 (indices 0 1 2 3 4 5 6 7 8)

- first put m[0] = -1 → i.e. sum = 0 occurs at ind = -1

sum = 0 (initially)

1 → present in map? ✗ ∴ put it in map w/ index

0 → sum == 0 ∴ till index 1 we get sum = 0

∴ maxi = max(0, i - [-1]) = 2

3 → put in map

5 → " " "

3 → present at ind = 2 (in map)

∴ maxi = max(2, i - 2) = max(2, 4 - 2) = 2

-5 → put in map

-4 → " " "

+7 → 3 → present in map at ind = 2

∴ maxi = max(2, i - 2) = max(2, 7 - 2) = 5

13 → now i > n

∴ we get maxi = 5 ans

↳ {2, -2, -8, 1, 7}

map<int, int>

prefix sum	index
(13, 8)	
(-4, 6)	
(-5, 5)	
(5, 3)	
(3, 2)	
(1, 0)	
(0, -1)	

[Note: when to check for sum == 0, always whenever you add a[i] to sum, check for s == 0]

Code-

```
int maxLen(int A[], int len)
{
    unordered_map<int,int> m;
    m[0] = -1;
    int maxi = 0, sum = 0;
    for(int i = 0; i < n; i++)
    {
        sum += A[i];
        if(m.find(sum) != m.end())
        {
            maxi = max(maxi, i - m[sum]);
        }
        else
        {
            m[sum] = i;
        }
    }
    return maxi;
}
```

TC = O(n × 1)
↳ unordered map
↳ WC = O(n) (rare)

SC = O(n)
↳ map

→ Count the number of subarrays having a given XOR.-

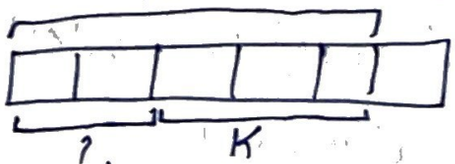
Given an array of integers arr[] & a number m, count the number of subarrays having XOR of their elements as m.

i/p: {4, 2, 2, 6, 4} & m = 6

o/p = 4
↳ subarrays having XOR of their elements as m = 6 are
{4,2}, {4,2,2,6,4}, {2,2,6}, {6}

Soln- approach

XR → prefix XOR of elements



$\therefore Y \wedge K = XR$

$\Rightarrow Y = XR \wedge K$

↳ if we fig. out how many Y's there in map, then we can find #subarrays having XOR = K

we will use map

<int, int>
↳ prefix XOR
↳ count (how many time it occurs)

Dry run

0 1 2 3 4
4, 2, 2, 6, 4 & K = 6

say, XR = 0 (initially) & cnt = 0

① XR = 0 ∧ 4 = 4 (≠ K)
Y = XR ∧ K = 4 ∧ 6 → not present in map
→ put in map

② XR = 4 ∧ 2 = 6 (== K) → get subarray {4, 2}
∴ ↑ cnt i.e. cnt = 0 + 1
Y = 6 ∧ 6 = 0 → not in map
∴ put (6, 1) in map

4 ∧ 2
100
010
110
= 6

③ XR = 6 ∧ 2 = 4 (≠ K)
Y = 4 ∧ 6 = 2 → not in map
∴ put 4 in map since, 4 already there
↑ it's count by 1

④ XR = 4 ∧ 6 = 2 (≠ K)
Y = 2 ∧ 6 = 4 → in map
∴ ↑ cnt by m[4] → we get 2 subarrays {2, 2, 6} & {6}
cnt = 1 + 2 = 3

⑤ XR = 2 ∧ 4 = 6 (== K)
∴ we get 1 subarray {4, 2, 2, 6, 4}
↑ cnt by 1
cnt = 3 + 1 = 4
& Y = 6 ∧ 6 = 0 → not in map
↳ now i > n

freq of 0 by 4 in map

```
int solin (vector<int> &A, int B)
{
    unordered_map<int,int> m;
    int cnt=0;
    int xorr=0;
    for(auto it:A)
    {
        xorr = xorr ^ it;
        if (xorr==B)
            ++cnt;
        if(m.find(xorr^B) != m.end())
        {
            cnt += m[xorr^B];
        }
        ++m[xorr];
    }
    return cnt;
}
```

TC = O(n x 1)
↳ unordered map
↳ O(n) WC (rare)

SC = O(n)