SC → O(N)

4/07/21 (Hashing)

Q9. 2 sum

my approach → brute force approach

TC → $O(n^2)$

SC → $O(1)$

Approach 2 use Hash Table

at any index check whether any element's complement exists or not, if it exists, then return index of hash table along with current index.

TC → $O(N^2)$ (generally o

SC → $O(N)$

data structure

unorderd-map<int, int> mp.

Q20. 4 sum

→ given an array and integer target

→ find all unique quadruplets such that each one's sum == target

Approach 1.

Sorting → 3 ptr + Binary search

target

4	3	3	4	4	2	1	2	1	1
1	1	1	2	2	3	3	4	4	4
i	j	k							

i = 0

j = i + 1

k = j + 1

find rem 9 − 3 = 6 using binary search.

if not found then increment value of k.

perform binary search on right half

1 3 4

~~when k gets finished up iterating array~~
~~j value~~ increments and $k = j+1$

Date: Page:

Time complexity
$= O(N^3 \log N)$
$O(1)$

Approach 2/
→ sort the given array
→ use 2 pts $i=0, j=1$, left $= j+1$ right $= n-1$
~~compcheck~~ left + right & remaining.
→ jump over duplicates

TC → $O(n^3)$
SC → $O(1)$

5/7/21 Q2 Longest common subsequence.
→ given unsorted array
→ find longest length of sequence elements

Approach 1
→ sort the array and then linearly iterate over the array to find sequence
* not to Nestest sol

$T = O(N \log N) + O(N)$

Approach 2
→ linearly iterate over array and store values in hash set
→ we check whether $a[i]-1$ exist in hash set, if it doesn't then we ~~check~~ increase the value of number and check whether it exists or not
→ we stop until we dont find any incrementing value

$TC = O(N^2)$
$SC = O(N)$

Date Page

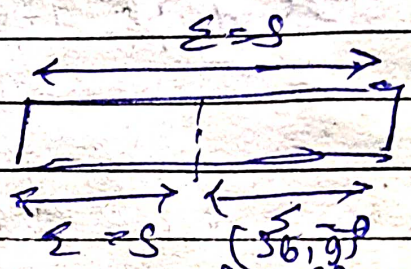
Q22. Largest subarray with sum 0.

Approach 1 → Naive solution

→ generate all possible subarray and count number of subarray with sum = 0.

$O(N^3) \rightarrow O(N^2)$

Approach 2 (Hashing)



→ declare hashmap <K, V>

S = 0.

→ Start linear traversal

1	-1	3	2	-2	-8	1	7	10	23

(36, 9)
(13, 8)
(-4, 6)
(-5, 5)
(5, 3)
(3, 2)
(1, 0)

S = 1 - 1 + 3 + 2 - 2 - 8 + 1 + 7

= 0 = 3 = 5 = 3 = -5 = -4 = 3

max = 2

already in set

max = 2, (4 - 2)

= 2

max = 2, (7 - 2)

Ans → = 5

TC = O(N log N)

SC = O(N)

Q23. Count subarrays with xor as K

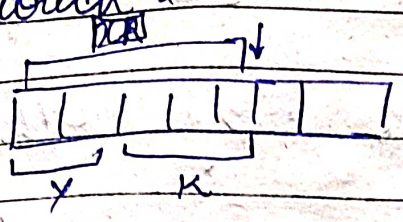
Approach 1 Brute force approach

TC $O(N^3) \rightarrow O(N^2)$

SC $O(1)$

→ given an array of integers and m, and we need to find the count of ~~number~~ subarrays ~~having all its elements m as XOR result~~

Approach 2.



$Y \wedge K = XR$

$Y = XR \wedge K$

find number of y

Dry Run

XOR = 0 (pre, cnt)

cnt = 0 1 3 4

[4, 2, 2, 6, 4] K = 6

(2,1)
(6,1)
(4,2)

freq.

XOR = 0^4 = 4^2 = 6^2 = 4^6 = 2.

① does XOR gives 6? if not then add it to hashmap

② if it gives, then increment cnt value.

Y = 6^6 = 0 → check if 0 prefix sum appears or not, if it appears then increment value of cnt

Y = 4^6 = 2 → does 2 exist in hashmap.

Y = 2^6 = 4

Ans. 4

TC → O(N log N)

code →

```
int sol(vector<int> A, int B) {
  map<int, int> freq;
  int cnt = 0.
  int xorr = 0;
  for (auto it : A) {
      xorr = xorr ^ it;
      if (xorr == B) cnt++;
      if (freq.find(xorr ^ B) != freq.end())
        cnt += freq[xorr ^ B];
      freq[xorr] += 1; }
return cnt.
```

Q2. Longest substring without repeating characters.

Date : Page :

Approach 1 Naive Approach. TC → $O(N^3)$
SC → $O(N)$

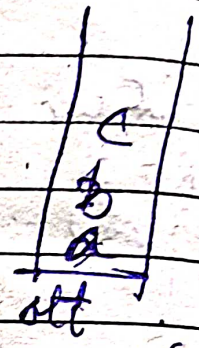
Approach 2

take l ptr & r ptr along with set and len = ~~0~~ ~~1~~ ~~0~~ 3

l

a b c a a b c d b a

r



[l-r] has no rep. char → push it in set & update ans.

→ if any char already present, then we remove the char at l index and increment value untill range has no repeating char.

TC = $O(2n)$
SC = $O(N)$

Approach 3.

Same as approach 2 but we are reduce travel time of l.

→ take map instead of set

Fractional knapsack

value [] = {60, 100, 120} N = 3 W = 50
wt [] = {10, 20, 30}

→ cannot carry all items in bag.

→ find maximum value including any fractional weight can be taken

greedy approach

→ find value per weight for each item

→ {6, 5, 4}

Date : Page :

→ sort items wrt. value per weight

→ if there's lesser weight left, then we pick fractional element

TC → N log N SC → O(1)

C++ code

```
struct item{
    int val;
    int wt; }

bool comp (item a, item b){
  double r1 = (double) a.val / (double) a.wt;
  double r2 = (double) b.val / (double) b.wt;
  return r1 > r2;

double fns. (int w, item arr [ ], int n)
 {
  sort (arr, arr + n, comp);
  int cw = 0;
  double finalwt = 0.0;
  for (int i = 0; i < n; i++){
    if (curWeight + arr [i].weight <= w){
      curWeight += arr [i].wt;
      finalvalue += arr [i].val;
    }
    else {
      int remain = w - curWeight;
      finalvalue += (arr[i].val / (double) a[i].weight)
                    * (double) remain.
      break; }
  return finalvalue;
 }
```