Question: https://leetcode.com/problems/3sum/

So here we have to find 3 sum.

The naive approach would be running 3 loops in nested format to generate all possible combinations.

Time Complexity:O(n^3)

Now if we look carefully then if we traverse the array from 0 to n and set target as target-nums[i] Now we need to find 2 more array elements whose sum will be equal to target-nums[i]; so now it is similar to 2 sum problem, if recall two sum problem then there were 2 approaches one was using memorization and the other was to sort the array and then use 2 pointer approach.

Since we will be using nested loops the time complexity would be O(n^2) in both the approaches, so regarding time complexity there is no advantage in choosing any of the 2 approaches.

Now lets see is there any advantage over space complexity or not.

Now if we are using the memorization approach then n-3 times we will create hashmap to find 2 sum, but whereas in two pointer approach no such space will be consumed, and once the array is sorted it is also easy to eliminate the duplicate elements.

Code:  
class Solution {

public List<List<Integer>> threeSum(int[] nums) {

List<List<Integer>> res = new ArrayList<>();

if(nums.length<3){

return res;

}

Arrays.sort(nums);

for(int i=0; i<nums.length; i++){

int s=i+1, e=nums.length-1, target=-nums[i];

HashMap<Integer, Integer> mem = new HashMap<Integer, Integer>();

//2 pointer approach

while(s<e){

if(nums[s]+nums[e]==target){

List<Integer> sub = new ArrayList<Integer>();

sub.add(nums[i]);

sub.add(nums[s]);

sub.add(nums[e]);

res.add(sub);

while(s<e && nums[s]==nums[s+1]) s++;

while(s<e && nums[e]==nums[e-1]) e--;

s++;

e--;

}

else if(nums[s]+nums[e]<target) s++;

else if(nums[s]+nums[e]>target) e--;

}

//eliminating duplicate element

while(i<nums.length-1 && nums[i]==nums[i+1]) i++;

}

return res;

}

}

Github Link :<https://lnkd.in/ecwtJeaz>