3 Sum : Find triplets that add up to a zero

Problem Statement: Given an array of N integers, your task is to find unique triplets that add up to give sum zero. In short, you need to return *an array of all the unique* triplets [arr[a], arr[b], arr[c]] such that i!=j, j!=k, k!=i, and their sum is equal to zero.

Examples:

Example 1:

Input: nums = [-1,0,1,2,-1,-4]

Output: [[-1,-1,2],[-1,0,1]]

Explanation: Out of all possible unique triplets possible, [-1,-1,2] and [-1,0,1] satisfy the condition of summing up to zero with i!=j!=k

Example 2:

Input: nums=[-1,0,1,0]

Output: Output: [[-1,0,1],[-1,1,0]]

Explanation: Out of all possible unique triplets possible, [-1,0,1] and [-1,1,0] satisfy the condition of summing up to zero with i!=j!=k

Solution:

Disclaimer. *Don't jump directly to the solution, try it out yourself first.*

Solution 1 (Brute Force):

Intuition: While starting out, our aim must be to come up with a working solution first. Thus, given the problem, the most intuitive solution would be using three-pointers and checking for each possible triplet, whether we can satisfy the condition or not.

Approach: We keep three-pointers i,j, and k. For every triplet we find the sum of A[i]+A[j]+A[k]. If this sum is equal to zero, we've found one of the triplets. We add it to our data structure and continue with the rest.

Code:

```
static ArrayList < ArrayList < Integer >> threeSum(int nums[]) {
        ArrayList < ArrayList < Integer >> ans = new ArrayList < > ();

        int i, j, k;
        for (i = 0; i < nums.length - 2; i++) {
            for (j = i + 1; j < nums.length - 1; j++) {
                for (k = j + 1; k < nums.length; k++) {
                    ArrayList < Integer > temp = new ArrayList < > ();
                    if (nums[i] + nums[j] + nums[k] == 0) {
                        temp.add(nums[i]);
                        temp.add(nums[j]);
                        temp.add(nums[k]);
                    }
                    if (temp.size() != 0)
                        ans.add(temp);
                }
            }
        }

        return ans;
    }
```

Time Complexity : O(n^3) // 3 loops

Space Complexity : O(3*k) // k is the no.of triplets

Approach 2**:** This is a Hashing-based solution.

- **Approach:** This approach uses extra space but is simpler than the two-pointers approach. Run two loops outer loop from start to end and inner loop from i+1 to end. Create a hashmap or set to store the elements in between i+1 to j-1. So if the given sum is x, check if there is a number in the set which is equal to x – arr[i] – arr[j]. If yes print the triplet.

- **Algorithm:**
    1. Traverse the array from start to end. (loop counter i)
    2. Create a HashMap or set to store unique pairs.
    3. Run another loop from i+1 to end of the array. (loop counter j)
    4. If there is an element in the set which is equal to x- arr[i] – arr[j], then print the triplet (arr[i], arr[j], x-arr[i]-arr[j]) and break
    5. Insert the jth element in the set.
- **Implementation:**

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

        int n=nums.length;

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

        for (int i = 0; i < n - 1; i++)

        {

            HashSet<Integer> s = new HashSet<Integer>();

            for (int j = i + 1; j < n; j++)

            {

                int x = -(nums[i] + nums[j]);

                if (s.contains(x))

                {
```

```
                    List<Integer> temp=new ArrayList();
                    temp.add(nums[i]);
                    temp.add(nums[j]);
                    temp.add(x);
                    Collections.sort(temp);
                    res.add(temp);
                }
                else
                {
                    s.add(nums[j]);
                }
            }
        }
        return new ArrayList(res);
    }
```

Complexity Analysis:

- **Time Complexity:** O(n^2).
  Since two nested loops are required, so the time complexity is O(n^2).
- **Auxiliary Space:** O(n).
  Since a hashmap is required, so the space complexity is linear.

Approach 3: (Optimized Approach):

Intuition : Can we do something better?

I think yes! In our intuitive approach, we were considering all possible triplets. But do we really need to do that? I say we fixed two pointers i and j and came out with a combination of [-1,1,1]

which doesn't satisfy our condition. However, we still check for the next combination of say [-1,1,2] which is worthless. (Assuming the k pointer was pointing to 2).

Approach:

We could make use of the fact that we just need to return the triplets and thus could possibly sort the array. This would help us use a modified two-pointer approach to this problem.

Eg) Input : [-1,0,1,2,-1,-4]

After sorting, our array is : [-4,-1,-1,0,1,2]

Notice, we are fixing the i pointer and then applying the traditional 2 pointer approach to check whether the sum of three numbers equals zero. If the sum is less than zero, it indicates our sum is probably too less and we need to increment our j pointer to get a larger sum. On the other hand, if our sum is more than zero, it indicates our sum is probably too large and we need to decrement the k pointer to reduce the sum and bring it closer to zero.

Code:

```
static ArrayList < ArrayList < Integer >> threeSum(int[] num) {
        Arrays.sort(num);
        ArrayList < ArrayList < Integer >> res = new ArrayList < > ();
        for (int i = 0; i < num.length - 2; i++) {
            if (i == 0 || (i > 0 && num[i] != num[i - 1])) {
                int lo = i + 1, hi = num.length - 1, sum = 0 - num[i];
                while (lo < hi) {
                    if (num[lo] + num[hi] == sum) {
                        ArrayList < Integer > temp = new ArrayList < > ();
                        temp.add(num[i]);
                        temp.add(num[lo]);
                        temp.add(num[hi]);
                        res.add(temp);
```

```
                    while (lo < hi && num[lo] == num[lo + 1]) lo++;
                    while (lo < hi && num[hi] == num[hi - 1]) hi--;
                    lo++;
                    hi--;
                } else if (num[lo] + num[hi] < sum) lo++;
                else hi--;
            }
        }
    }
    return res;
}
```

Time Complexity : O(N^2)

Space Complexity : O(M)