<https://leetcode.com/problems/3sum/>

Given an integer array nums, return all the triplets [nums[i], nums[j], nums[k]] such that i != j, i != k, and j != k, and nums[i] + nums[j] + nums[k] == 0.

Notice that the solution set must not contain duplicate triplets.

**Example 1:**

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

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

Explanation:

nums[0] + nums[1] + nums[2] = (-1) + 0 + 1 = 0.

nums[1] + nums[2] + nums[4] = 0 + 1 + (-1) = 0.

nums[0] + nums[3] + nums[4] = (-1) + 2 + (-1) = 0.

The distinct triplets are [-1,0,1] and [-1,-1,2].

Notice that the order of the output and the order of the triplets does not matter.

**Example 2:**

Input: nums = [0,1,1]

Output: []

Explanation: The only possible triplet does not sum up to 0.

**Example 3:**

Input: nums = [0,0,0]

Output: [[0,0,0]]

Explanation: The only possible triplet sums up to 0.

**Constraints:**

3 <= nums.length <= 3000

-10^5 <= nums[i] <= 10^5

**Attempt 1:2022-09-15**

**Solution 1:  Binary Search solution (360 min, too long to figure out how the skip duplicate mechanism work and how to apply Binary Search template)**

class Solution {

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

        Arrays.sort(nums);

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

        int len = nums.length;

        for(int i = 0; i < len; i++) {

            int twoSumTarget = -nums[i];

            // Since requirement as triple numbers sum equal to 0, and based

            // on sorted nums array, if first number of triple numbers already

            // larger than 0, there is no chance to build any triple combination

            // directly return empty list as result

            if(nums[i] > 0) {

                break;

            }

            // The tricky to avoid calculation on duplicate numbers

            // e.g

            // Input: [-1,0,1,2,-1,-4]

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

            // Expected: [[-1,-1,2],[-1,0,1]]

            // Initially we may worry about if adding skip duplicate elements mechanism

            // will cause a combination like [-1,-1,2] not able to find, but result is

            // this combination not missed. Let's say after sort the input is [-4,-1,-1,0,1,2],

            // the skip duplicate mechanism won't apply to the first time we encounter

            // an element which has duplicates, in another word, when we hold the first -1

            // and try to find its complementary two sum up as 1 in later section of input,

            // we will find only one combination if must include the only 2 as [-1,2], the

            // second -1 included in this combination, but remember how we come to a triple

            // combination as [-1,-1,2], it is when we try to calculate a complementary two

            // sum up for the first -1, not for the second -1, no skip duplicate mechanism

            // triggered yet. And if we come to the second -1, from the for loop we can see

            // it already truncate the input as [-1,0,1,2] and try to find a complementary

            // two sum up in later section of the truncated input after second -1 as [0,1,2],

            // there is no combination for this, and that's why we find only one triple

            // combination as [-1,-1,2] and even no skip duplicate mechanism added we won't

            // find another duplicate combination as [-1,-1,2]

            // ------------------------------------------------------------------

            // Then let's see how the skip duplicate mechanism works when comes to [-1,0,1].

            // For the first -1, we try to find a complementary two sum up as 1 in later

            // section as [-1,0,1,2], besides previously talked [-1,2] we find another one

            // as [0,1], the first triple combination as [-1,0,1], the '-1' is the first -1

            // in original sorted input. Then go ahead to the second -1, if no skip duplicate

            // mechanism, we will also try to find a complementary two sum up as 1 in later

            // section for the second -1 as [0,1,2], now we also able to find a two sum up

            // combination as [0,1], the second triple combination as [-1,0,1], the '-1' is

            // the second -1 in the original sorted input. We find two [-1,0,1] based on two

            // continuous -1 as initial element, to avoid duplicate combination the key point

            // is not holding same initial element, which means if we use first -1 to find its

            // complementary two sum up combination, then no need to use second -1 to find its

            // complementary again, the simple solution is when encounter an element same as

            // its previous one (since in sorted input), just skip it by continue the loop

            if(i > 0 && nums[i] == nums[i - 1]) {

                continue;

            }

            for(int j = i + 1; j < len; j++) {

                // Same tricky to avoid calculation on duplicate numbers apply again

                if(j > i + 1 && nums[j] == nums[j - 1]) {

                    continue;

                }

                int target = twoSumTarget - nums[j];

                int lo = j + 1;

                int hi = len - 1;

                while(lo <= hi) {

                    int mid = lo + (hi - lo) / 2;

                    if(nums[mid] == target) {

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

                        solution.add(nums[i]);

                        solution.add(nums[j]);

                        solution.add(nums[mid]);

                        result.add(solution);

                        // Usually it will be a return to stop the while loop

                        // in a standard Binary Search, but here since we only

                        // leverage the Binary Search and no need to return for

                        // now, just add a solution and break out the loop, if

                        // no break out will cause forever loop

                        break;

                    } else if(nums[mid] > target) {

                        hi = mid - 1;

                    } else {

                        lo = mid + 1;

                    }

                }

            }

        }

        return result;

    }

}

Space Complexity: O(1)

Time Complexity: O(n^2logn)

Two for loop cost O(n^2), inside inner for loop binary search for target cost O(logn)

**Wrong attempt version before Binary Search Solution:**

**Wrong version 1: Don't remove duplicates from input, only skip duplicates in Binary Search**

class Solution {

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

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

        Arrays.sort(nums);

        // Wrong solution because of removing duplicate elements and result

        // into missing solution.

        // e.g

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

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

        // Actual Output: [[-1,0,1]]

        // Because remove of duplicate '-1' will result into no [-1,-1,2] solution

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

            if(i == 0 || i > 0 && nums[i] != nums[i - 1]) {

                tmp.add(nums[i]);

            }

        }

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

        int size = tmp.size();

        for(int i = 0; i < size; i++) {

            int twoSumTarget = -tmp.get(i);

            for(int j = i + 1; j < size; j++) {

                int target = twoSumTarget - tmp.get(j);

                int lo = j + 1;

                int hi = size - 1;

                while(lo <= hi) {

                    int mid = lo + (hi - lo) / 2;

                    if(tmp.get(mid) == target) {

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

                        solution.add(tmp.get(i));

                        solution.add(tmp.get(j));

                        solution.add(tmp.get(mid));

                        result.add(solution);

                        // Usually it will be a return to stop the while loop

                        // in a standard Binary Search, but here since we only

                        // leverage the Binary Search and no need to return for

                        // now, just add a solution and break out the loop, if

                        // no break out will cause forever loop

                        break;

                    } else if(tmp.get(mid) > target) {

                        hi = mid - 1;

                    } else {

                        lo = mid + 1;

                    }

                }

            }

        }

        return result;

    }

}

**Wrong version 2: No skip duplicates mechanism in Binary Search**

class Solution {

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

        Arrays.sort(nums);

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

        int len = nums.length;

        for(int i = 0; i < len; i++) {

            int twoSumTarget = -nums[i];

            for(int j = i + 1; j < len; j++) {

                int target = twoSumTarget - nums[j];

                int lo = j + 1;

                int hi = len - 1;

                while(lo <= hi) {

                    int mid = lo + (hi - lo) / 2;

                    if(nums[mid] == target) {

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

                        solution.add(nums[i]);

                        solution.add(nums[j]);

                        solution.add(nums[mid]);

                        result.add(solution);

                        // Usually it will be a return to stop the while loop

                        // in a standard Binary Search, but here since we only

                        // leverage the Binary Search and no need to return for

                        // now, just add a solution and break out the loop, if

                        // no break out will cause forever loop

                        break;

                    } else if(nums[mid] > target) {

                        hi = mid - 1;

                    } else {

                        lo = mid + 1;

                    }

                }

            }

        }

        return result;

    }

}

Wrong Answer

Details

Input

[-1,0,1,2,-1,-4]

Output

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

Expected

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

**Binary Search solution refer to:**

<https://leetcode.com/problems/3sum/discuss/2352346/C%2B%2B-or-Easy-or-Sorting-or-Binary-Search>

class Solution {

public:

    vector<vector<int>> threeSum(vector<int>& nums) {

        sort(nums.begin(),nums.end());

        vector<vector<int>> ans;

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

            if(nums[i] > 0)

                break;

            if(i > 0 && nums[i] == nums[i-1])

                continue;

            int target = -nums[i];

            for(int j=i+1;j<nums.size();j++){

                if(j > i+1 && nums[j] == nums[j-1])

                    continue;

                int l = j+1, r = nums.size()-1;

                int val = target-nums[j];

                bool ok = false;

                while(l <= r){

                    int mid = (l+r)/2;

                    if(nums[mid] < val)

                        l = mid+1;

                    else if(nums[mid] > val)

                        r = mid-1;

                    else{

                        ok = true;

                        break;

                    }

                }

                if(ok){

                    vector<int> triplet = {nums[i],nums[j],val};

                    ans.push\_back(triplet);

                }

            }

        }

        return ans;

    }

};

**Solution 2:  Two Pointers solution (20 min)**

class Solution {

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

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

        Arrays.sort(nums);

        int len = nums.length;

        for(int i = 0; i < len; i++) {

            if(nums[i] > 0) {

                break;

            }

            // Skip duplicate if same element seen as nums[i - 1] when

            // iterate from second time onwards

            // If no this line, will result into:

            // Your input: [-1,0,1,2,-1,-4]

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

            // Expected: [[-1,-1,2],[-1,0,1]]

            if(i > 0 && nums[i] == nums[i - 1]) {

                continue;

            }

            int target = -nums[i];

            int lo = i + 1;

            int hi = len - 1;

            // Why not while(lo <= hi) ?

            // e.g

            // Your input: [-1,0,1,2,-1,-4]

            // Output: [[-4,2,2],[-1,-1,2],[-1,0,1]]

            // Expected: [[-1,-1,2],[-1,0,1]]

            // -----------------------------

            // nums=[-1,0,1,2,-1,-4] sort -> nums=[-4,-1,-1,0,1,2]

            // -----------------------------

            // Round 1:

            // i=0,target=4

            // lo=1,hi=5

            // sum=nums[1]+nums[5]=1 < target -> lo++=2

            // lo=2,hi=5

            // sum=nums[2]+nums[5]=1 < target -> lo++=3

            // lo=3,hi=5

            // sum=nums[3]+nums[5]=2 < target -> lo++=4

            // lo=4,hi=5

            // sum=nums[4]+nums[5]=3 < target -> lo++=5

            // lo=5,hi=5

            // if while loop condition as while(lo <= hi), then lo=5, hi=5 is allowed,

            // which will result into sum=nums[5]+nums[5]=4 == target, get combination as

            // [-4,2,2], which is a wrong result so while loop condition should not

            // include '=', strictly '<' to avoid duplicately use same element again

            while(lo < hi) {

                int sum = nums[lo] + nums[hi];

                if(sum == target) {

                    result.add(Arrays.asList(nums[i], nums[lo], nums[hi]));

                    lo++;

                    hi--;

// Skip duplicate

                    while(lo < hi && nums[lo] == nums[lo - 1]) {

                        lo++;

                    }

// Skip duplicate

                    while(lo < hi && nums[hi] == nums[hi + 1]) {

                        hi--;

                    }

                } else if(sum > target) {

                    hi--;

                } else {

                    lo++;

                }

            }

        }

        return result;

    }

}

Time Complexity:O(n^k−1), or O(n^2) for 3Sum. We have k - 2 loops, and twoSum is O(n).

Note that for k > 2, sorting the array does not change the overall time complexity.

Space Complexity:O(n). We need O(k) space for the recursion. k can be the same as nn in the worst case for the generalized algorithm.

Note that, for the purpose of complexity analysis, we ignore the memory required for the output.

**Two Pointers solution refer to:**

<https://leetcode.com/problems/3sum/discuss/7631/Simple-Java-Solution-Without-using-HashSet>

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

    Arrays.sort(nums);

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

    for(int i = 0; i < nums.length-2; i++) {

        if(i > 0 && (nums[i] == nums[i-1])) continue; // avoid duplicates

        for(int j = i+1, k = nums.length-1; j<k;) {

            if(nums[i] + nums[j] + nums[k] == 0) {

                list.add(Arrays.asList(nums[i],nums[j],nums[k]));

                j++;k--;

                while((j < k) && (nums[j] == nums[j-1]))j++;// avoid duplicates

                while((j < k) && (nums[k] == nums[k+1]))k--;// avoid duplicates

            }else if(nums[i] + nums[j] + nums[k] > 0) k--;

            else j++;

        }

    }

    return list;

}

**Refer to**

[L18.P3.9.4Sum(Ref.L15)](note://C61608B7F28E422BA14F37ABF6331F2A)