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 a sum of 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

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

Brute Force Approach
Algorithm / Intuition

In the question, it is clearly stated that for each case the picked indices i.e. i, j, and k must be distinct. This means [arr[1], arr[1], arr[2]] is not a valid triplet and also remember [arr[1], arr[0], arr[2]] and [arr[0], arr[1], arr[2]] will be considered the same.

Intuition:

This approach is pretty straightforward. Here, we will check all possible triplets using 3 loops and among them, we will consider the ones whose sum is equal to the given target i.e. 0. And before considering them as our answer we need to sort the triplets in ascending order so that we can consider only the unique ones.

Algorithm:

The steps are as follows:

1. First, we will declare a set data structure as we want unique triplets.
2. Then we will use the first loop(say i) that will run from 0 to n-1.
3. Inside it, there will be the second loop(say j) that will run from i+1 to n-1.
4. Then there will be the third loop(say k) that runs from j+1 to n-1.
5. Now, inside these 3 nested loops, we will check the sum i.e. arr[i]+arr[j]+arr[k], and if it is equal to the target i.e. 0 we will sort this triplet and insert it in the set data structure.
6. Finally, we will return the list of triplets stored in the set data structure.

Code

```
import java.util.*;

public class tUf {
    public static List<List<Integer>> triplet(int n, int[] arr) {
        Set<List<Integer>> st = new HashSet<>();

        // check all possible triplets:
        for (int i = 0; i < n; i++) {
            for (int j = i + 1; j < n; j++) {
                for (int k = j + 1; k < n; k++) {
                    if (arr[i] + arr[j] + arr[k] == 0) {
                        List<Integer> temp = Arrays.asList(
                              arr[i], arr[j], arr[k]
                        );
                        temp.sort(null);
                        st.add(temp);
                    }
                }
            }
        }

        // store the set elements in the answer:
        List<List<Integer>> ans = new ArrayList<>(st);
        return ans;
    }

    public static void main(String[] args) {
        int[] arr = { -1, 0, 1, 2, -1, -4};
        int n = arr.length;
        List<List<Integer>> ans = triplet(n, arr);
        for (List<Integer> it : ans) {
            System.out.print("[");
            for (Integer i : it) {
                System.out.print(i + " ");
            }
            System.out.print("] ");
        }
        System.out.println();
    }
}
```

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

Complexity Analysis

Time Complexity: O(N3 * log(no. of unique triplets)), where N = size of the array.
Reason: Here, we are mainly using 3 nested loops. And inserting triplets into the set takes O(log(no. of unique triplets)) time complexity. But we are not considering the time complexity of sorting as we are just sorting 3 elements every time.

Space Complexity: O(2 * no. of the unique triplets) as we are using a set data structure and a list to store the triplets.

Better Approach
Algorithm / Intuition

Intuition:

In the previous approach, we utilized 3 loops, but now our goal is to reduce it to 2 loops. To achieve this, we need to find a way to calculate arr[k] since we intend to eliminate the third loop (k loop). To calculate arr[k], we can derive a formula as follows:

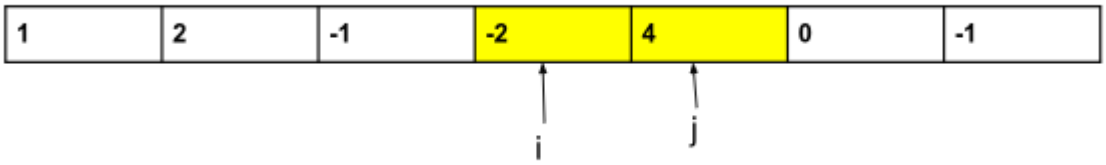
arr[k] = target - (arr[i]+arr[j]+arr[k]) = 0-(arr[i]+arr[j]+arr[k]) = -(arr[i]+arr[j]+arr[k])

So, we will first calculate arr[i] and arr[j] using 2 loops and for the third one i.e. arr[k] we will not use another loop and instead we will look up the value 0-(arr[i]+arr[j]+arr[k]) in the set data structure. Thus we can remove the third loop from the algorithm.

For implementing the search operation of the third element, we will store all the elements between the indices i and j in a HashSet and then we will search for the third element in the HashSet.

Why we are not inserting all the array elements in the HashSet and then searching for the third element:

Let's understand this intuition using an example. Assume the given array is {1, 2, -1, -2, 4, 0, -1} and the target = 0. Now, we will notice a situation like the following:



Here, arr[i] = -2 and arr[j] = 4.
Therefore, arr[k] = -(arr[i]+arr[j]) = -(-2+4) = -2

- If all the elements were in the set data structure while searching for -2, we would again pick the element at index 3, that is currently pointed by i.
- Hence, the triplet will be [arr[i], arr[j], arr[i]]. And this is an invalid triplet. That is why we cannot insert all the elements into the set data structure.

Algorithm:

The steps are as follows:

1. First, we will declare a set data structure as we want unique triplets.
2. Then we will use the first loop(say i) that will run from 0 to n-1.
3. Inside it, there will be the second loop(say j) that will run from i+1 to n-1.
4. Before the second loop, we will declare another HashSet to store the array elements as we intend to search for the third element using this HashSet.
5. Inside the second loop, we will calculate the value of the third element i.e. -(arr[i]+arr[j]).
6. If the third element exists in the HashSet, we will sort these 3 values i.e. arr[i], arr[j], and the third element, and insert it in the set data structure declared in step 1.

7. After that, we will insert the j-th element i.e. arr[j] in the HashSet as we only want to insert those array elements that are in between indices i and j.
8. Finally, we will return a list of triplets stored in the set data structure.

Code

```
import java.util.*;

public class tUf {
    public static List<List<Integer>> triplet(int n, int[] arr) {
        Set<List<Integer>> st = new HashSet<>();

        for (int i = 0; i < n; i++) {
            Set<Integer> hashset = new HashSet<>();
            for (int j = i + 1; j < n; j++) {
                //Calculate the 3rd element:
                int third = -(arr[i] + arr[j]);

                //Find the element in the set:
                if (hashset.contains(third)) {
                    List<Integer> temp = Arrays.asList(arr[i], arr[j], third);
                    temp.sort(null);
                    st.add(temp);
                }
                hashset.add(arr[j]);
            }
        }

        // store the set elements in the answer:
        List<List<Integer>> ans = new ArrayList<>(st);
        return ans;
    }

    public static void main(String[] args) {
        int[] arr = { -1, 0, 1, 2, -1, -4};
        int n = arr.length;
        List<List<Integer>> ans = triplet(n, arr);
        for (List<Integer> it : ans) {
            System.out.print("[");
            for (Integer i : it) {
                System.out.print(i + " ");
            }
            System.out.print("] ");
        }
        System.out.println();
    }
}
```

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

Complexity Analysis

Time Complexity: O(N2 * log(no. of unique triplets)), where N = size of the array.
Reason: Here, we are mainly using 3 nested loops. And inserting triplets into the set takes O(log(no. of unique triplets)) time complexity. But we are not considering the time complexity of sorting as we are just sorting 3 elements every time.

Space Complexity: O(2 * no. of the unique triplets) + O(N) as we are using a set data structure and a list to store the triplets and extra O(N) for storing the array elements in another set.

Optimal Approach
Algorithm / Intuition

In this approach, we intend to get rid of two things i.e. the HashSet we were using for the look-up operation and the set data structure used to store the unique triplets.

So, We will first sort the array. Then, we will fix a pointer i, and the rest 2 pointers j and k will be moving.

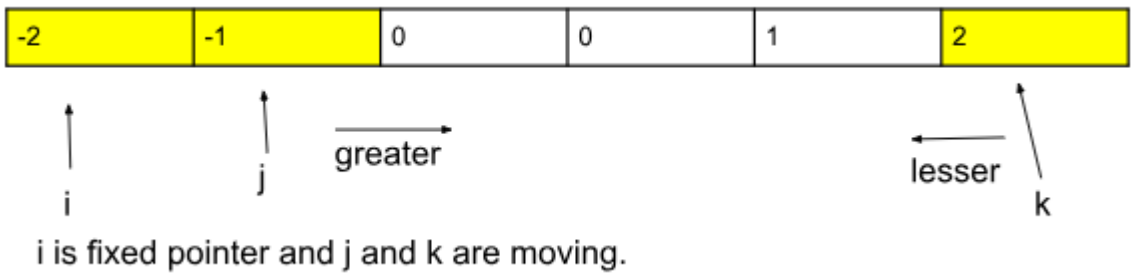
Now, we need to first understand what the HashSet and the set were doing to make our algorithm work without them. So, the set data structure was basically storing the unique triplets in sorted order and the HashSet was used to search for the third element.

That is why, we will first sort the entire array, and then **to get the unique triplets,** we will simply skip the duplicate numbers while moving the pointers.

How to skip duplicate numbers:
As the entire array is sorted, the duplicate numbers will be in consecutive places. So, while moving a pointer, we will check the current element and the adjacent element. Until they become different, we will move the pointer by 1 place. We will follow this process for all 3 pointers. Thus, we can easily skip the duplicate elements while moving the pointers.

Now, we can also remove the HashSet as we have two moving pointers i.e. j and k that will find the appropriate value of arr[j] and arr[k]. So, we do not need that HashSet anymore for the look-up operations.

The process will look like the following:



Among the 3 pointers, 1 will be fixed and 2 will be moving. In each iteration, we will check if the sum i.e.arr[i]+arr[j]+arr[k] is equal to the target i.e. 0.

- **If the sum is greater**, then we need lesser elements and so we will decrease the value of k(i.e. k--).
- **If the sum is lesser than the target**, we need a bigger value and so we will increase the value of j (i.e. j++).
- **If the sum is equal to the target,** we will simply insert the triplet i.e. arr[i], arr[j], arr[k], into our answer and move the pointers j and k skipping the duplicate elements.

Approach:

The steps are as follows:

1. First, we will sort the entire array.

2. We will use a loop(say i) that will run from 0 to n-1. This i will represent the fixed pointer. In each iteration, this value will be fixed for all different values of the rest of the 2 pointers. Inside the loop, we will first check if the current and the previous element is the same and if it is we will do nothing and continue to the next value of i.
3. After that, there will be 2 moving pointers i.e. j(starts from i+1) and k(starts from the last index). The pointer j will move forward and the pointer k will move backward until they cross each other while the value of i will be fixed.
    1. Now we will check the sum i.e. arr[i]+arr[j]+arr[k].
    2. **If the sum is greater**, then we need lesser elements and so we will decrease the value of k(i.e. k--).
    3. **If the sum is lesser than the target**, we need a bigger value and so we will increase the value of j (i.e. j++).
    4. **If the sum is equal to the target,** we will simply insert the triplet i.e. arr[i], arr[j], arr[k] into our answer and move the pointers j and k skipping the duplicate elements(*i.e. by checking the adjacent elements while moving the pointers*).
4. Finally, we will have a list of unique triplets.

Code

```
import java.util.*;

public class tUf {
    public static List<List<Integer>> triplet(int n, int[] arr) {
        List<List<Integer>> ans = new ArrayList<>();
        Arrays.sort(arr);

        for (int i = 0; i < n; i++) {
            //remove duplicates:
            if (i != 0 && arr[i] == arr[i - 1]) continue;

            //moving 2 pointers:
            int j = i + 1;
            int k = n - 1;
            while (j < k) {
                int sum = arr[i] + arr[j] + arr[k];
                if (sum < 0) j++;
                else if (sum > 0) k--;
                else {
                    List<Integer> temp = Arrays.asList(arr[i], arr[j], arr[k]);
                    ans.add(temp);
                    j++; k--;
                    //skip the duplicates:
                    while (j < k && arr[j] == arr[j - 1]) j++;
                    while (j < k && arr[k] == arr[k + 1]) k--;
                }
            }
        }
        return ans;
    }

    public static void main(String[] args) {
        int[] arr = { -1, 0, 1, 2, -1, -4};
        int n = arr.length;
        List<List<Integer>> ans = triplet(n, arr);
        for (List<Integer> it : ans) {
            System.out.print("[");
            for (Integer i : it) {
                System.out.print(i + " ");
            }
            System.out.print("] ");
        }
        System.out.println();
    }
}
```

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

Complexity Analysis

Time Complexity: O(NlogN)+O(N2), where N = size of the array.
Reason: The pointer i, is running for approximately N times. And both the pointers j and k combined can run for approximately N times including the operation of skipping duplicates. So the total time complexity will be O(N2).

Space Complexity: O(no. of quadruplets), ***This space is only used to store the answer. We are not using any extra space to solve this problem.*** So, from that perspective, space complexity can be written as O(1).