Statement

↳ Given an integer array, nums, find and return all unique triplets [nums[i], nums[j], nums[k]], such that $i \neq j$, $i \neq k$, and $j \neq k$ and nums[i] + nums[j] + nums[k] == Ø.

Approach

↳ Preprocessing:

- Sort the array nums in ascending order
- Initialize an empty array result to store unique triplets
- Store the length of nums in n

↳ Iterate over array from index i = Ø to n-2

- Break the loop if nums[i] > Ø. Since array is sorted, no triplet can sum to Ø after that.
- Continue only if i == Ø or nums[i] != nums[i-1] to ensure the current number is either the first element or not a duplicate of the previous element.
  - Initialize two pointers: low = i+1 and high = n-1
  - Loop as long as low < high
    - ↳ Calculate the sum: nums[i] + nums[low] + nums[high]
    - ↳ If sum is less than Ø, move low pointer forward
    - ↳ If sum is greater than Ø, move high pointer backward
    - ↳ If sum is equal to Ø, add [nums[i], nums[low], nums[high]] to the result
    - ↳ Skip duplicates when moving low and high
      - Increment low while nums[low] == nums[low-1]
      - Decrement high while nums[high] == nums[high+1]

↳ Return result after iterating through the whole array.

Visualization

↳ Find all triplets that sum to Ø

i) Given array

-4	-1	2	2	-1	Ø

ii) Sort the array

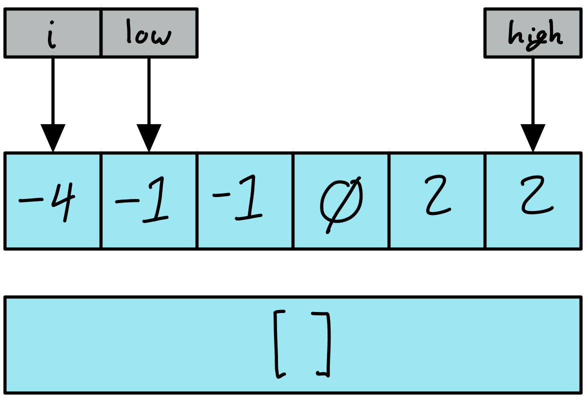
-4	-1	-1	Ø	2	2

iii) Create empty array to store unique triplets

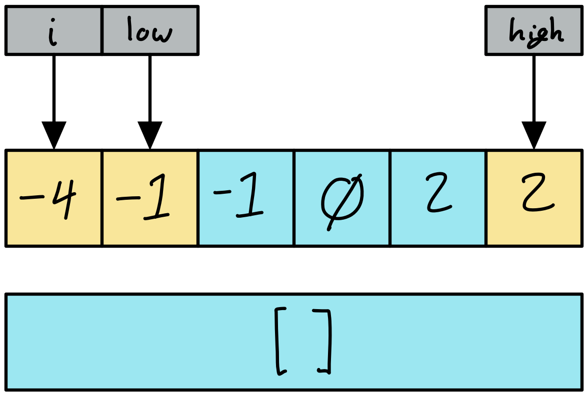
-4	-1	-1	Ø	2	2

[ ]

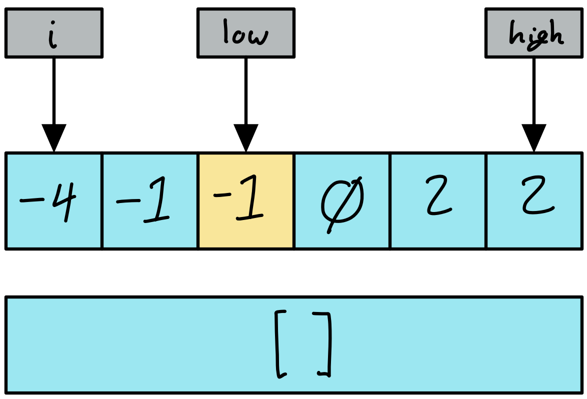
iv) Start iterating through the nums array using i pointer. Then initialize low with the element next to i and high with the last element of nums.



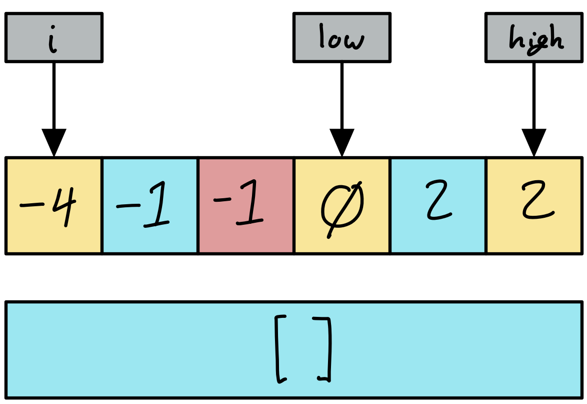
v) As nums[i] + nums[low] + nums[high] = -4 + (-1) + 2 = -3, which is less than Ø, we increment low.



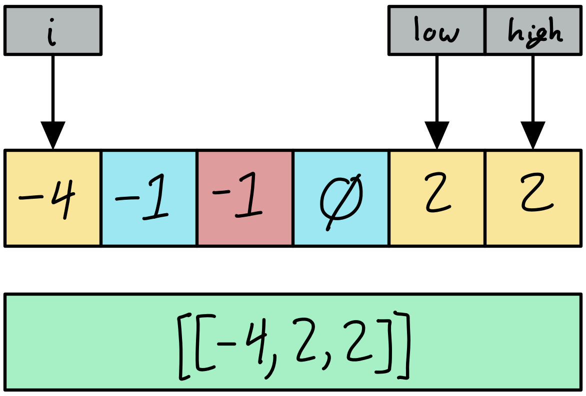
vi) As nums[low] is equal to nums[low-1], we skip this to avoid duplication and increment low again.



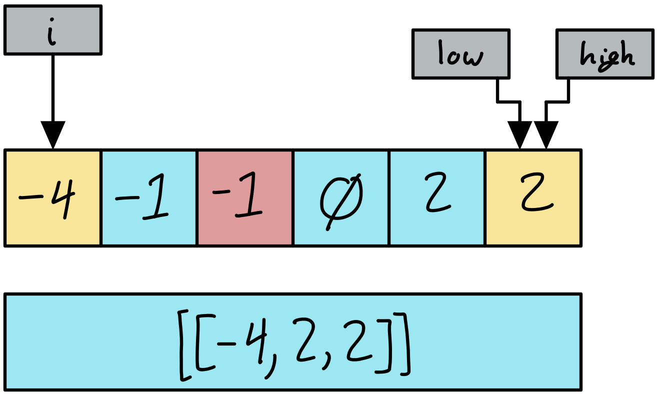
vii) As nums[i] + nums[low] + nums[high] = -4 + Ø + 2 = -2, which is less than Ø, increment low.



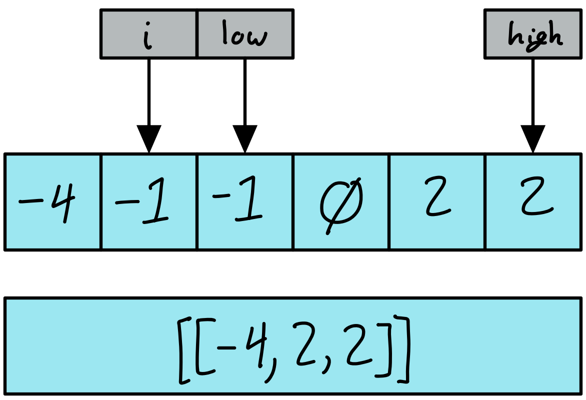
viii) As nums[i]+nums[low]+nums[high]=-4+2+2=∅, which is equal to ∅, add the triple to result and increment low.



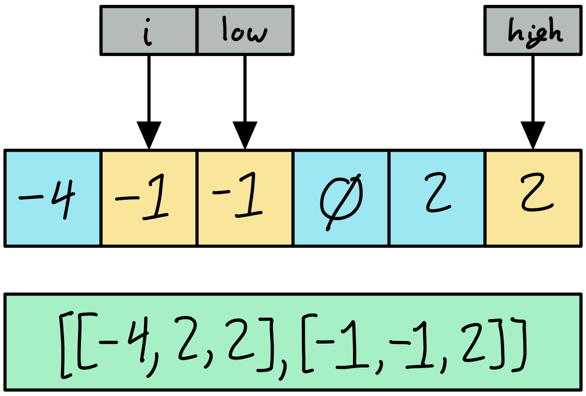
ix) Stop the two pointers iteration because low is not less than high anymore.



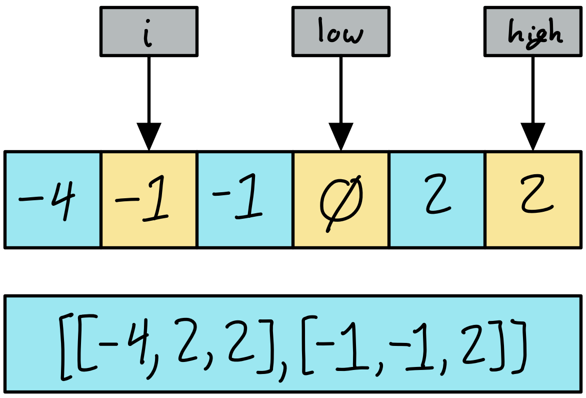
x) Move i to the next index. Initialize low with the element next to the i and high with the last element of nums.



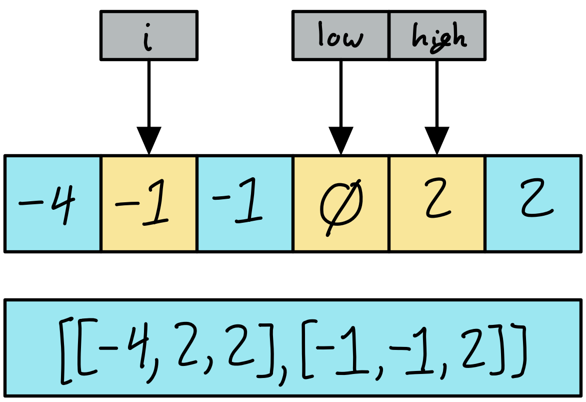
xi) As nums[i] + nums[low] + nums[high] = -1 + (-1) + 2 = 0, add it to result and increment low.



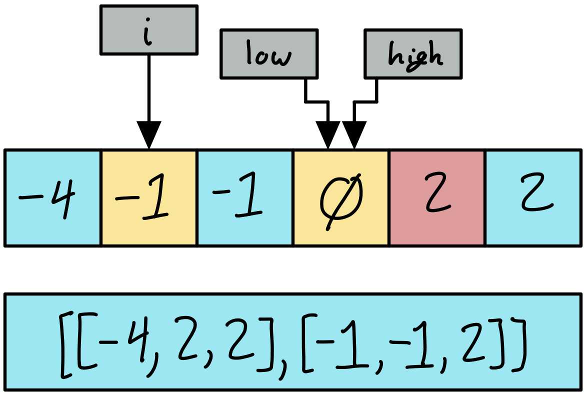
xii) As nums[i] + nums[low] + nums[high] = -1 + 0 + 2 = 1, which is greater than 0, decrement high.



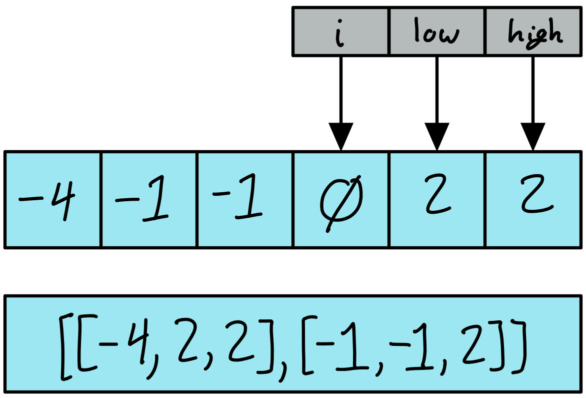
xiii) As nums[high] equals nums[high+1], decrement high to skip this value in order to avoid duplication.



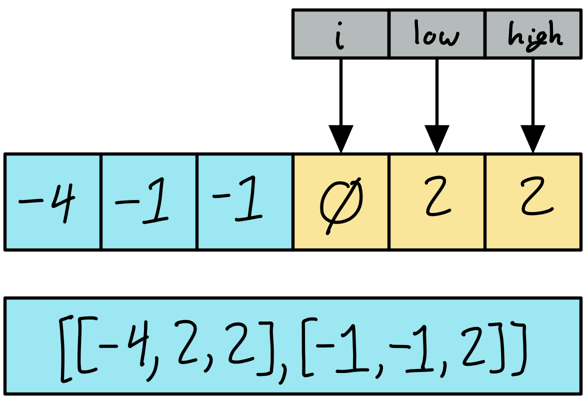
xiv) Stop iterating the two pointers because low >= high.



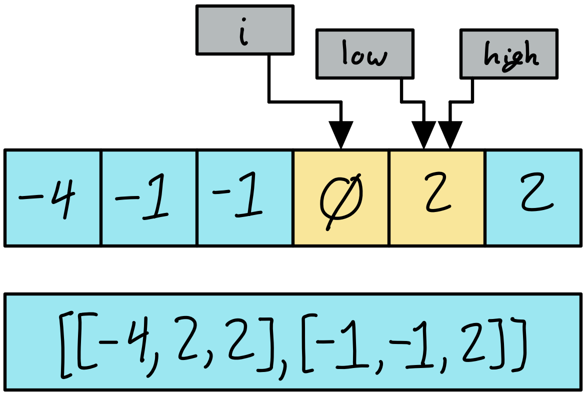
xv) Next, move i forward. As nums[2] equals nums[1], skip nums[2] and move to nums[3] to avoid duplicates. Then, initialize low with the element next to i and high with the last element of nums.



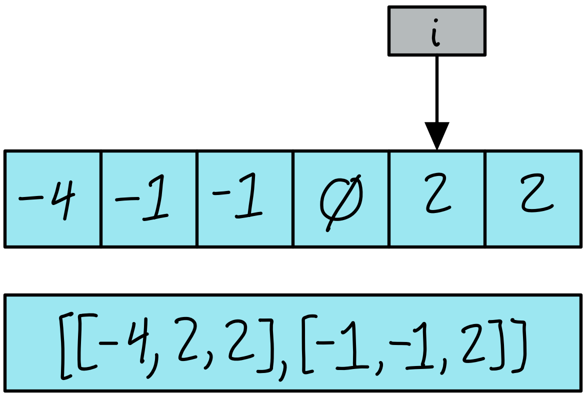
xvi) As nums[i] + nums[low] + nums[high] = 0 + 2 + 2 = 4, is greater than 0, decrement high.



xvii) Stop iterating the two pointers because low >= high. Then, increment i.



xviii) At this point i has reached the second-to-last element of the array. As a valid triplet requires three distinct elements, no further triplets can be formed with only two remaining. Therefore, stop iterating through the array and return the result array, which contains all the unique triplets that sum to zero.



Code

```
vector<vector<int>> threeSum(vector<int>& nums) {
    sort(nums.begin(), nums.end());

    vector<vector<int>> result;

    int n = nums.size();

    for (int i = 0; i < n-2; i++) {
        if (nums[i] > 0) {
            break;
        }

        if (i = 0 || nums[i] != nums[i-1]) {
            int low = i + 1, high = n-1;

            if (sum < 0) {
                low++;
            } else if (sum > 0 {
                high--;
            } else {
                result.push_back({nums[i], nums[low], nums[high]});

                low++;
                high--;

                while (low < high && nums[low] == nums[low-1]) {
                    low++;
                }
                while (low < high && nums[high] == nums[high+1]) {
                    high++;
                }
            }
        }
    }
    return result;
}
```

Time Complexity

↳ Let n represent the length of the nums array.

- Sorting takes $O(n \log n)$
- The nested iteration takes $O(n^2)$, where each nums[i] is paired with a two pointer traversal over the remaining elements of the array.

↳ Therefore, the overall time complexity is $O(n^2)$.

Space Complexity

↳ Apart from the space used by the built-in sorting algorithm, the algorithm's space complexity is constant $O(1)$.