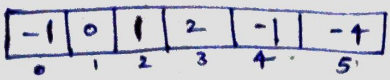
3 Sum Problem :-



target = 0
(It may or may not be 0)

⊙ Brute Force :-

i	j	k	i j k	Triplet
i constant	j constant (i+1)	(j+1) k ↓	(0, 1, 2)	(-1, 0, 1)
			(0, 1, 3)	(-1, 0, 2)
			(0, 1, 4)	(-1, 0, -1)
			(0, 1, 5)	(-1, 0, -4)
i constant	j+1 constant	k ↓	(0, 2, 3)	(-1, 1, 2)
			(0, 2, 4)	(-1, 1, -1)
			(0, 2, 5)	(-1, 1, -4)
i constant	j+2 constant	k ↓	(0, 3, 4)	(-1, 2, -1)
			(0, 3, 5)	(-1, 2, -4)
i constant	j+3	k ↓	(0, 4, 5)	(-1, -1, -4)
i = i+1 constant	j constant (i+1)	(j+1) k ↓	(1, 2, 3)	(-1, 1, 2)
			(1, 2, 4)	(-1, 1, -1)
			(1, 2, 5)	(-1, 1, -4)
i constant	j+1 constant	k ↓	(1, 3, 4)	(0, 2, -1)
			(1, 3, 5)	(0, 2, -4)

! So, on continue

```
for(i = 0 ⟶ n-1) {
    for(j = i+1 ⟶ n-1) {
        for(k = j+1 ⟶ n-1) {
            ≡≡≡
        }
    }
}
```

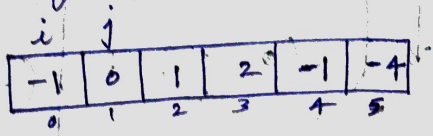
So, $T = O(n^3)$

Because of Hashset $S = O(n)$

[Each triplet we store in sorted order in HashSet. HashSet will contain unique triplets. Finally, all triplets are taken in a list and returned.]

⊙ Better Approach :—

Using HashMap & HashSet



target = 0

We have to search :—

target − (arr[i] + arr[j])

in HashMap

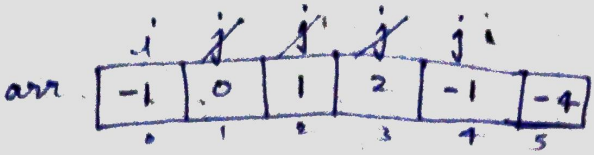
HashMap is used to store the elements which are already visited.

Each of the triplets we store in sorted order in Hashset. Hashset will contain unique triplets finally.

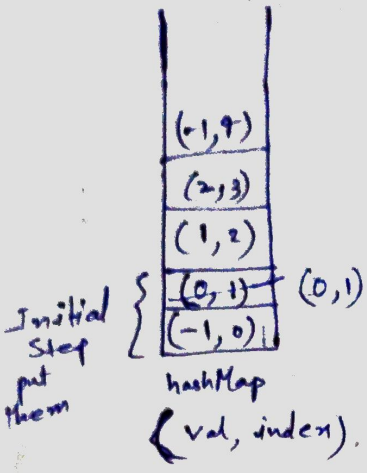
And finally, all unique triplets are taken in a list and returned

Unlike, 2 Sum, we have to take care of an edge case here.

1.) We have to make sure that we don't pick the same element pointed by i and j because, in a triplet all three elements should be different (index-wise)



Target = 0



i=0, j=1

searchKey = target - (arr[i] + arr[j])

= 0 - (-1+0)

= 1

Is 1 present in hashMap?

↳ No; so enter (arr[j], j) in hashMap i.e., (0,1)

---

i=0, j=2

SearchKey = 0 - (-1+1)

= 0 - (0)

= 0

Is 0 present in hMap?

↳ Yes ⇒ Is index of 0 == (i or j)

⇓

No

⇓

We got a triplet (Sort triplet and store in HashSet)

(-1, 0, 1) triplet = (arr[i], arr[j], searchKey)

⇓

Then add (arr[j], j) into the hashMap i.e., (1,2)

---

i=0, j=3

SearchKey = 0 - (-1+2) = -1

Is -1 present in hMap?

↳ Yes ⇒ Is index of -1 == (i or j)

⇓

Yes

⇓

Then add (arr[j], j) into hashMap i.e., (2,3)

i=0, j=1

searchKey = 0 - (-1 - 1) = 2

Is 2 present in hMap ?

⇓

Yes

⇓

Is index of 2 == (i or j)

⇓

No

⇓

We got a triplet

(-1, -1, 2)

(arr[i], arr[j], searchKey)

we will sort the triplet and store in HashSet

⇓

Then add (arr[j], j) to hashMap i.e., (-1, 1)

And So, on we will continue.

```
for (int i = 0 ⟶ n-1) {
    for (int j = i+1 ⟶ n-1) {
        hashMap.put(arr[j], j);
    }
}
```

$T = O(n^2) * O(\text{hasMap\_search} + \text{HashMap\_Put} + \text{HashSet\_Add}) + O(n)$

Here, $O(n^2)$ is for two nested loops.

$O(n)$ is for adding all triplets from hashSet to List.

$O($hashMap_search + HashMap_put + HashSet_Add$)$

- hashMap_search → $O(1)$ Average Case
- HashMap_put → $O(1)$ Average Case
- HashSet_Add → $O(1)$ Average Case

So, $T = O(n^2)$

$S = O(n) + O(n) = O(n)$

- $O(n)$ → HashMap
- $O(n)$ → HashSet

⊙ **Best Approach :—**

target = 0

-1	0	1	2	-1	-4	0	0	0	1	1	-1	-4
0	1	2	3	4	5	6	7	8	9	10	11	12

⇒ Sort the array

-4	-4	-1	-1	-1	0	0	0	0	1	1	1	2
0	1	2	3	4	5	6	7	8	9	10	11	12
i	j											K

```
i=0
while(i<n){
    j=i+1
    K=n-1;
    newTarget = target - arr[i];
    [ Same logic as 2 sum ]

    temp=arr[i];
    while(temp==arr[i]){
        i++;
        if(i>=n){
            break;
        }
    }
}
```

$T = O(n \log n) + O(n^2)$

Sort — Main logic

$T = O(n^2)$ $\quad$ $S = O(1)$

Note:- Better approach and Best Approach both has $T = O(n^2)$ but still better approach can give TLE in some compilers as For hasMap in coarst case time complexity can be $O(n)$. It is rare but still can happen.