

# Find Pairs Sum



## Problem statement

You are given an integer array 'ARR' of size 'N' and an integer 'S'. Your task is to return the list of all pairs of elements such that each sum of elements of each pair equals 'S'.

Note:

Each pair should be sorted i.e the first value should be less than or equals to the second value.

Return the list of pairs sorted in non-decreasing order of their first value. In case if two pairs have the same first value, the pair with a smaller second value should come first.

Detailed explanation ( Input/output format, Notes, Images )

Constraints:

$1 \leq N \leq 10^3$

$-10^5 \leq \text{ARR}[i] \leq 10^5$

$-2 * 10^5 \leq S \leq 2 * 10^5$

Time Limit: 1 sec

Sample Input 1:

5 5

1 2 3 4 5

Sample Output 1:

1 4

2 3

Explanation For Sample Output 1:

Here,  $1 + 4 = 5$

$2 + 3 = 5$

Hence the output will be, (1,4) , (2,3).

Sample Input 2:

5 0

2 -3 3 3 -2

Sample Output 2:

-3 3

-3 3

-2 2. Give java code using both hashmap and two pointers as well



Here is the Java code to solve the problem using **two approaches**:

1. Using HashMap
2. Using Two Pointers

Both approaches are implemented below:

## Using HashMap Approach

```
java

import java.util.*;

public class FindPairsUsingHashMap {
    public static List<List<Integer>> findPairsWithSumHashMap(int[] arr, int S) {
        List<List<Integer>> result = new ArrayList<>();
        Map<Integer, Integer> map = new HashMap<>();

        for (int num : arr) {
            int complement = S - num;

            // Check if the complement exists in the map
            if (map.containsKey(complement) && map.get(complement) > 0) {
                result.add(Arrays.asList(Math.min(num, complement), Math.max(num, complement)));
                map.put(complement, map.get(complement) - 1); // Decrease count of complement
            } else {
                map.put(num, map.getOrDefault(num, 0) + 1); // Add number to the map
            }
        }

        // Sort the pairs as per the requirements
        result.sort((a, b) -> a.get(0).equals(b.get(0)) ? a.get(1) - b.get(1) : a.get(0) - b.get(0));
        return result;
    }

    public static void main(String[] args) {
        int[] arr = {2, -3, 3, 3, -2};
        int S = 0;
        List<List<Integer>> pairs = findPairsWithSumHashMap(arr, S);
        for (List<Integer> pair : pairs) {
            System.out.println(pair.get(0) + " " + pair.get(1));
        }
    }
}
```

## Using Two Pointers Approach

```
java

import java.util.*;

public class FindPairsUsingTwoPointers {
    public static List<List<Integer>> findPairsWithSumTwoPointers(int[] arr, int S) {
        List<List<Integer>> result = new ArrayList<>();

        // Sort the array
        Arrays.sort(arr);

        int left = 0, right = arr.length - 1;

        while (left < right) {
            int sum = arr[left] + arr[right];

            if (sum == S) {
                result.add(Arrays.asList(arr[left], arr[right]));
                left++;
                right--;
            } else if (sum < S) {
                left++;
            } else {
                right--;
            }
        }

        return result;
    }
}
```

```

        right--;
    }
}

return result;
}

public static void main(String[] args) {
    int[] arr = {2, -3, 3, 3, -2};
    int S = 0;
    List<List<Integer>> pairs = findPairsWithSumTwoPointers(arr, S);
    for (List<Integer> pair : pairs) {
        System.out.println(pair.get(0) + " " + pair.get(1));
    }
}
}

```

## Explanation of Both Approaches:

### HashMap Approach

1. Iterate through the array and calculate the complement of each element (`S - current\_element`).
2. Check if the complement exists in the `HashMap` with a non-zero frequency.
3. If it exists, add the pair `(min(num, complement), max(num, complement))` to the result and reduce the frequency of the complement in the map.
4. Otherwise, add the current element to the map with its frequency.
5. Finally, sort the result list to meet the constraints.

### Two Pointers Approach

1. First, sort the array.
2. Use two pointers: one starting at the beginning (`left`) and the other at the end (`right`).
3. Calculate the sum of the elements at the two pointers:
  - If the sum equals `S`, add the pair to the result and move both pointers inward.
  - If the sum is less than `S`, increment the `left` pointer.
  - If the sum is greater than `S`, decrement the `right` pointer.
4. Continue until the pointers meet.

## Complexity:

Approach	Time Complexity	Space Complexity
HashMap	$O(N)$	$O(N)$
Two Pointers	$O(N \log N)$	$O(1)$

Both approaches produce the same output, but the choice depends on the use case and constraints.



dry run hashmap method solution with arr[]={1,2,3,4,5} & s=5



Let's dry-run the **HashMap method** step-by-step for the given input:

``arr = {1, 2, 3, 4, 5}`` and ``S = 5``.

## Initial Setup

- ``map = {}`` (HashMap to store numbers and their frequencies)
  - ``result = []`` (List to store the resulting pairs)
- 

## Step-by-Step Execution

### Iteration 1 (num = 1):

- Complement =  $S - num = 5 - 1 = 4$
  - Check if ``4`` exists in ``map`` → **No**.
  - Add ``1`` to ``map`` → ``map = {1: 1}``.
- 

### Iteration 2 (num = 2):

- Complement =  $S - num = 5 - 2 = 3$
  - Check if ``3`` exists in ``map`` → **No**.
  - Add ``2`` to ``map`` → ``map = {1: 1, 2: 1}``.
- 

### Iteration 3 (num = 3):

- Complement =  $S - num = 5 - 3 = 2$
  - Check if ``2`` exists in ``map`` → **Yes**, and its frequency is ``1``.
  - Add the pair `(2, 3)` to ``result`` → ``result = [[2, 3]]``.
  - Decrease the frequency of ``2`` in ``map`` → ``map = {1: 1, 2: 0}``.
- 

### Iteration 4 (num = 4):

- Complement =  $S - num = 5 - 4 = 1$
  - Check if ``1`` exists in ``map`` → **Yes**, and its frequency is ``1``.
  - Add the pair `(1, 4)` to ``result`` → ``result = [[2, 3], [1, 4]]``.
  - Decrease the frequency of ``1`` in ``map`` → ``map = {1: 0, 2: 0}``.
- 

### Iteration 5 (num = 5):

- Complement =  $S - num = 5 - 5 = 0$
  - Check if ``0`` exists in ``map`` → **No**.
  - Add ``5`` to ``map`` → ``map = {1: 0, 2: 0, 5: 1}``.
- 

## Final Result

- ``result = [[2, 3], [1, 4]]``
- Sort the pairs → After sorting: ``result = [[1, 4], [2, 3]]``.

## Output

The final list of pairs is:

```
1 4
2 3
```



which approach is best?



## Comparison of the Two Approaches:

### 1. HashMap Approach

- **Time Complexity:**  $O(N)$ 
  - Single pass to iterate over the array.
  - HashMap operations (insertion and lookup) are  $O(1)$  on average.
- **Space Complexity:**  $O(N)$ 
  - Requires additional space to store elements and their frequencies in the HashMap.

### 2. Two Pointers Approach

- **Time Complexity:**  $O(N \log N)$ 
  - Sorting the array takes  $O(N \log N)$ .
  - Finding pairs with two pointers takes  $O(N)$ .
- **Space Complexity:**  $O(1)$ 
  - Does not use any additional data structures beyond a few variables.

## Which Approach is Best?

The best approach depends on the constraints and the context:

### When to Use HashMap:

- If the array is large and unsorted, and sorting it would take too much time.
- If the array contains duplicate values and you want to handle them efficiently.
- HashMap ensures a consistent  $O(N)$  time complexity regardless of sorting.

### When to Use Two Pointers:

- If minimizing space complexity is a priority.
- If the array is already sorted (or can be sorted easily without much overhead).

- Two Pointers have an edge when  $N$  is relatively small and sorting  $O(N \log N)$  is not a major bottleneck.
- 

### Verdict:

- For larger arrays ( $N$  is closer to  $10^3$ ), **HashMap** is generally more efficient due to its linear  $O(N)$  complexity.
- For smaller arrays or scenarios where space usage is critical, **Two Pointers** is preferred due to its  $O(1)$  space requirement.

### Practical Recommendation:

Use **HashMap** if the array is unsorted and contains a mix of large values and duplicates. Use **Two Pointers** if the array is sorted or if you need to prioritize space efficiency.