Input: non-empty array of distinct integers
an integer representing the target sum

array = [7, 6, 4, -1, 1, 2]
targetSum = 16

Output: 2D array of quadruplets that sum to the target sum in no particular order

[[7,6,4,-1],[7,6,1,2]]

[7, 6, 4, -1, 1, 2]

```
// Avg: O(n^2) Time | O(n^2) space
// Worst: O(n^3) | O(n^2) space
function fourNumberSum(array, targetSum) {
  const allPairSums = {};
  const quadruplets = [];

  for (let i = 1; i < array.length - 1; i++) {
    for (let j = i + 1; j < array.length; j++) {
      const currentSum = array[i] + array[j];
      const difference = targetSum - currentSum;
      if (difference in allPairSums) {
        for (const pair of allPairSums[difference]) {
          quadruplets.push(pair.concat([array[i], array[j]]));
        }
      }
    }
    for (let k = 0; k < i; k++) {
      const currentSum = array[i] + array[k];
      if (currentSum in allPairSums) {
        allPairSums[currentSum].push([array[i], array[k]]);
      } else {
        allPairSums[currentSum] = [[array[i], array[k]]];
      }
    }
  }
  return quadruplets;
}
```

Notes: We do it this way in order to avoid counting duplicates

Idea:

1) Start by iterating through the array (1 → array.length - 1) — why? (bc nothing is in the hash table)
   - 1.1) At each iteration, loop through the numbers to the right, and including, the current number (currNum → array.length) — currNum non-inclusive
     - 1.1.1) At each loop, get the sum of the pair.
     - 1.1.2) Get the difference b/w target sum and the pair — non-inclusive bc we already have a pointer with i
     - 1.1.3) If this difference is not in the hash table, continue, otherwise loop through all the values at the key (difference) and push the curr index numbers + the pair onto the quadruplets array
   - 1.2) At each iteration, loop through the numbers to the left, and including, the number (0 → currNum) — non-inclusive
     - 1.2.1) At each loop, get the sum of the pair.
     - 1.2.2) If the sum of the pair is in our hash table, we append the new pair to the specific key otherwise we add it to our hash table with the sum as the key and the pairs as the value
2) return the quadruplets.

Time: $O(n^2)$ (where n is the # of elements in the input array) since at each element, we iterate through every other element in the input array

Worst case is $O(n^3)$:

e.g. imagine the targetSum is 0 and our array is [-4, -3, -2, -1, 1, 2, 3, 4]
We will have to iterate through our inner for loop $\approx n$ times (0: [-4,4], [-3,3], [-2,2], [-1,1]) making the time $O(n^3)$. (Note: this edge case is awkward and hard to justify as well)

Space: $O(n^2)$ since we could have none of the sum's overlap given us n^2 space (because of n^2 pairs)