two sums

```
def two_sum(numbers, target):

    for i in range(len(numbers)):
      for x in range(i + 1, len(numbers)):
       if numbers[i] + numbers[x]  == target:
           return tuple(sorted([i, x]))
```

this has 2 iterations, i retrieves the index and elements up to the length of the numbers

x does the same but +1 up to the length of numbers so as to avoid numbers adding together

its returned as a tuple in ascending order

```
def two_sum(numbers, target):
    seen = set()
    for i, num in enumerate(numbers):
        complement = target - num
        if complement in seen:
            return [numbers.index(complement), i]
        seen.add(num)
    return None
```

This is a lot more optimised as its O(n)

It enumerates the list of numbers meaning it only iterates once

target - nums, if the result matches complement

the end result is complement and nums

Example

```
numbers = [2, 7, 11, 15]
target_sum = 9
result = two_sum(numbers, target_sum)
print(result)
```

1. Initialize an empty set `seen`.
2. Iterate through the `numbers` list using `enumerate` to get both the index `i` and the element `num` at each iteration.
3. Calculate the complement (`complement = target_sum - num`) for the current element.
4. Check if the complement is already in the `seen` set. If yes, return the indices of the pair that adds up to the target sum.
5. If the complement is not in `seen`, add the current element `num` to the `seen` set.
6. Repeat the process for each element in the `numbers` list.

Let's go through the example:

- For `numbers[0] = 2`, the complement is `9 - 2 = 7`. Since 7 is not in `seen`, add 2 to `seen`.
- For `numbers[1] = 7`, the complement is `9 - 7 = 2`. 2 is already in `seen`, so return the indices `[0, 1]`.
- The function returns `[0, 1]` as the result.

This example illustrates how the function efficiently finds the pair of elements in the `numbers` list that adds up to the target sum. The use of the `seen` set helps avoid redundant checks and ensures correct results.