two sum

simple example

nums: 3, 2, 4 , target = 6

6 → (2, 4) → (1, 2)

2 + 4 = 6

indexes

x + y = 6

y = nums[i]

looping over nums

i = 0, nums[0] = 3, y = 6 − 3 = 3 does it exists in nums (on oth index?) NO

i = 1, nums[1] = 2, y = 6 − 2 = 4 does it ..... YES

indexes 1, 2 (2 + 4 = 6)

we do the search with map (O(1))

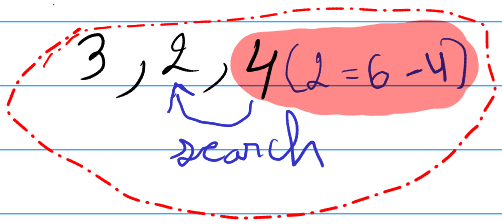
```
class Solution {
public:
    std::vector<int> twoSum(std::vector<int> &nums, int target) {
        map<int, int> numsIndexes;
        for (int i = 0; i < nums.size(); i++) {
            numsIndexes[nums[i]] = i;
        }
        for (int i = 0; i < nums.size(); i++) {
            int calculatedTarget = target - nums[i];
            if (numsIndexes.contains(calculatedTarget)) {
                int j = numsIndexes[calculatedTarget];
                if (i == j) {
                    continue;
                }
                return {i, j};
            }
        }
        return {};
    }
};
```

How to avoid the loop for storing in map?

3, 2, 4 — 2 + 4 = 6 (break condition in last example)

can also be 4 + 2 = 6

means that we can search for 2 when we attempt the 4 instead of searching for 4 at 2 (i+1) future



index 0: 3 → 6 - 3 = 3 not in map empty
store 3 in map

index 1: 2 → 6 - 2 = 4 not in map
store 2 in map, we have (3, 2)

index 2: 4 → 6 - 4 = 2 → exists in map
so 4 + 2 = 6 (1, 2)

```
class Solution {
public:
    std::vector<int> twoSum(std::vector<int> &nums, int target) {
        map<int, int> numsIndexes;
        for (int i = 0; i < nums.size(); i++) {
            int calculatedTarget = target - nums[i];
            if (numsIndexes.contains(calculatedTarget)) {
                int j = numsIndexes[calculatedTarget];
                if (i == j) {
                    continue;
                }
                return {j, i};
            } else {
                numsIndexes[nums[i]] = i;
            }
        }
        return {};
    }
};
```

Try to sole it without map usage, using greedy approach

increase → ← decrease

2 5 6 8 11 (sorted)

target = 14 (6 + 8) → (2, 3)

leftP → 2 + 11 = 13 ≠ 14 ← rightP

for each step we move the leftP or rightP means we control our desired target based on increase or decrease, ↑ search

2 + 11 = 13 < 14 increase (leftP++)

5 + 11 = 16 > 14 → decrease (rightP++)

5 + 8 = 13 < 14 => increase

6 + 8 = 14 == 14 finish

example 2 2, 3, 4 target = 6

4 + 2 = 6 == 6 finish

```
class Solution {
public:
    std::vector<int> twoSum(std::vector<int> &nums, int target) {
        // track values with its original index before sorting
        vector<pair<int, int> > indexedNums;
        for (int i = 0; i < nums.size(); i++) {
            indexedNums.push_back({nums[i], i});
        }
        sort(indexedNums.begin(), indexedNums.end());
        int leftP = 0;
        int rightP = nums.size() - 1;
        while (rightP > leftP) {
            int leftValue = indexedNums[rightP].first;
            int rightValue = indexedNums[leftP].first;
            int sum = leftValue + rightValue;
            if (sum > target) {
                rightP--;
            } else if (sum < target) {
                leftP++;
            } else {
                // nums[rightP] + nums[leftP] == target
                return {indexedNums[leftP].second, indexedNums[rightP].second};
            }
        }
        return {-1, -1};
    }
};
```