Two Sum на Go: ищем индексы двух чисел с заданной суммой

Есть задачи, которые встречаются почти на каждом техническом собеседовании — и Two Sum, пожалуй, самая частая из них. Разберём её пошагово: сначала простое решение «в лоб», потом оптимальное через `map`, с трассировкой каждого шага.

Практическая польза тут не только в подготовке к собеседованию. Похожая логика встречается и в реальных задачах — например, при поиске пары транзакций с нужной суммой в финансовом сервисе или при сопоставлении данных из двух источников.

Это первая задача из списка LeetCode, которую я разбираю на Go — дальше будут следующие, разного уровня сложности.

Тема: Массивы, Хэш-таблицы (map)
Уровень сложности: 🟢 лёгкий

Задача

Дан слайс целых чисел `nums` и целое число `target`. Нужно вернуть индексы двух чисел так, чтобы их сумма равнялась `target`.

Можно рассчитывать, что у каждого входа есть ровно одно решение, и один и тот же элемент нельзя использовать дважды. Порядок индексов в ответе не важен.

Пример 1

Входные параметры: `nums = [2, 7, 11, 15], target = 9`
Выходные параметры: `[0, 1]`
Объяснение: `nums[0] + nums[1] == 9`, поэтому возвращаем `[0, 1]`.

Пример 2

Входные параметры: `nums = [3, 2, 4], target = 6`
Выходные параметры: `[1, 2]`

Пример 3

Входные параметры: `nums = [3, 3], target = 6`
Выходные параметры: `[0, 1]`

Ограничения

- `2 ≤ len(nums) ≤ 10^4`
- `-10^9 ≤ nums[i] ≤ 10^9`
- `-10^9 ≤ target ≤ 10^9`
- Существует ровно один верный ответ

Дополнительный вопрос: сможете придумать решение со сложностью меньше `O(n²)`?

Алгоритм простого перебора

Самый очевидный вариант — пройтись циклом по каждому элементу `x` и проверить, есть ли где-то дальше в слайсе значение `target - x`.

```go
func twoSum(nums []int, target int) []int {
    for i := 0; i < len(nums)-1; i++ {
        for j := i + 1; j < len(nums); j++ {
            if nums[i]+nums[j] == target {
                return []int{i, j}
            }
        }
    }
    return nil
}
```

Трассировка

Разберём вызов `twoSum([3, 2, 4], 6)` по шагам.

Инициализация: `nums = [3, 2, 4]`

Шаг	Что происходит
i = 0, j = 1	`nums[0] + nums[1] = 3 + 2 = 5`, не равно `target = 6`
i = 0, j = 2	`nums[0] + nums[2] = 3 + 4 = 7`, не равно `target = 6`
i = 1, j = 2	`nums[1] + nums[2] = 2 + 4 = 6`, равно `target = 6` → возвращаем `[1, 2]`

Функция возвращает `[1, 2]`: элементы с индексами 1 и 2 в сумме дают 6.

Сложность

Временная сложность: `O(n²)`. Для каждого элемента перебираем оставшуюся часть слайса в поисках дополнения — это `O(n)` действий на каждый из `n` элементов.

Пространственная сложность: `O(1)`. Дополнительная память не растёт вместе с размером входных данных.

В задаче спрашивают про решение быстрее `O(n²)` — переходим к нему.

Алгоритм с использованием map

В Go для этого прекрасно подходит `map` — она даёт поиск за амортизированное `O(1)` и позволяет сократить время работы с `O(n²)` до `O(n)`, пожертвовав дополнительной памятью.

Идея простая: проходим по слайсу один раз и для каждого элемента проверяем, встречалось ли раньше число, дополняющее его до `target`. Если да — пара найдена. Если нет — запоминаем текущий элемент в `map` и идём дальше.

```
func twoSum(nums []int, target int) []int {
    seen := make(map[int]int)

    for i, v := range nums {
        if j, ok := seen[target-v]; ok {
            return []int{j, i}
        }
        seen[v] = i
    }

    return nil
}
```

Трассировка

Разберём тот же вызов `twoSum([3, 2, 4], 6)`.

Инициализация: `nums = [3, 2, 4]`, `seen = map[int]int{}`

Шаг	Что происходит
i = 0, v = 3	`target - v = 3`, такого ключа в `seen` нет → записываем `seen[3] = 0`
i = 1, v = 2	`target - v = 4`, такого ключа в `seen` нет → записываем `seen[2] = 1`
i = 2, v = 4	`target - v = 2`, `seen[2] = 1` уже есть → возвращаем `[1, 2]`

Функция возвращает `[1, 2]` — тот же результат, что и в первом варианте, но за один проход по слайсу вместо вложенных циклов.

Сложность

Временная сложность: `O(n)`. Проходим по слайсу из `n` элементов ровно один раз, каждая операция с `map` занимает амортизированное `O(1)`.

Пространственная сложность: `O(n)`. В худшем случае `map` хранит почти все элементы слайса.

Вывод

У обоих решений есть своё место. Перебор проще написать, и он не требует дополнительной памяти — подойдёт, если слайс небольшой или память в приоритете. `Map` выигрывает на больших объёмах данных за счёт дополнительного расхода памяти.

На собеседовании обычно ждут, что вы начнёте с перебора, а затем сами предложите оптимизацию через `map` — именно так эта задача чаще всего и разбирается.

#Go #Golang #Algorithms #LeetCode #lessgo

Понравился разбор? Подписывайся в социальных сетях, чтобы не пропустить новые задачи:

Telegram: https://t.me/lssgo
VK: https://vk.com/lssgo
GitHub: https://github.com/slemeshaev
Сетка: https://set.ki/a6DBfpg
Max: https://max.ru/channel_lssgo

@LSSGO