Busca Binária (Binary Search) e Variações

A busca binária é um algoritmo de busca eficiente que utiliza a estratégia de **"dividir para conquistar"**.

Como funciona (Passo a passo)

1. **Encontrar o Meio:** O algoritmo olha para o elemento no centro da lista.
2. **Comparar:**
   - Se o elemento central for o alvo, a busca termina.
   - Se o alvo for **menor**, descarta-se a metade direita.
   - Se o alvo for **maior**, descarta-se a metade esquerda.
3. **Repetir:** Até encontrar o valor ou a lista ficar vazia.

O Cálculo do Meio em Listas Ímpares

Em linguagens como Go, a divisão de inteiros descarta as casas decimais. Isso resolve o problema de listas com tamanhos ímpares.

Exemplo 1: Lista com 5 elementos (Ímpar)

Índices: `[0, 1, 2, 3, 4]` | `mid = 0 + (4 - 0) / 2` = **2**

Exemplo 2: Lista com 4 elementos (Par)

Índices: `[0, 1, 2, 3]` | `mid = 0 + (3 - 0) / 2` = 1.5 -> (Truncado para **1**)

Desafio de Entrevista: Busca em Array Rotacionado

Exemplo de Execução (Dry Run)

Array: `[4, 5, 6, 7, 0, 1, 2]` | **Alvo:** `0`

Iteração	low	high	mid	nums[mid]	Lógica Aplicada
1	0	6	3	7	Esquerda ordenada (`4 <= 7`). Alvo `0` não está entre `4` e `7` . Move `low` para `mid + 1` .
2	4	6	5	1	Direita ordenada (`1 <= 2`). Alvo `0` não está entre `1` e `2` . Move `high` para `mid - 1` .
3	4	4	4	0	`nums[mid] == target` . **Retorna índice 4.**

Por que funciona em Arrays NÃO rotacionados?

Este algoritmo é um "superconjunto" da busca binária tradicional. Se o array estiver perfeitamente ordenado (ex: `[0, 1, 2, 4, 5, 6, 7]`), a condição `nums[low] <= nums[mid]` será sempre verdadeira para a primeira metade, e o código cairá sempre na lógica de busca padrão. **Ele é universal para arrays ordenados.**

Implementação em Go

```
package main

import "fmt"

func SearchRotated(nums []int, target int) int {
        low := 0
        high := len(nums) - 1

        for low <= high {
                mid := low + (high-low)/2

                if nums[mid] == target {
                        return mid
                }

                // Se o array não estiver rotacionado, este IF sempre será verdac
                if nums[low] <= nums[mid] {
                        if target >= nums[low] && target < nums[mid] {
                                high = mid - 1
                        } else {
                                low = mid + 1
                        }
                } else {
                        if target > nums[mid] && target <= nums[high] {
                                low = mid + 1
                        } else {
                                high = mid - 1
                        }
                }
        }
        return -1
}

func main() {
        rotatedArray := []int{4, 5, 6, 7, 0, 1, 2}
        normalArray := []int{0, 1, 2, 4, 5, 6, 7}
        target := 0

        fmt.Printf("Rotacionado - Índice do %d: %d\n", target, SearchRotated(rotc
        fmt.Printf("Normal      - Índice do %d: %d\n", target, SearchRotated(norm
}
```

Pontos Chave para a Entrevista

1. **Universalidade:** O código lida com ambos os casos (rotacionado ou não).
2. **Integer Division:** Em Go, `5 / 2` resulta em `2`.