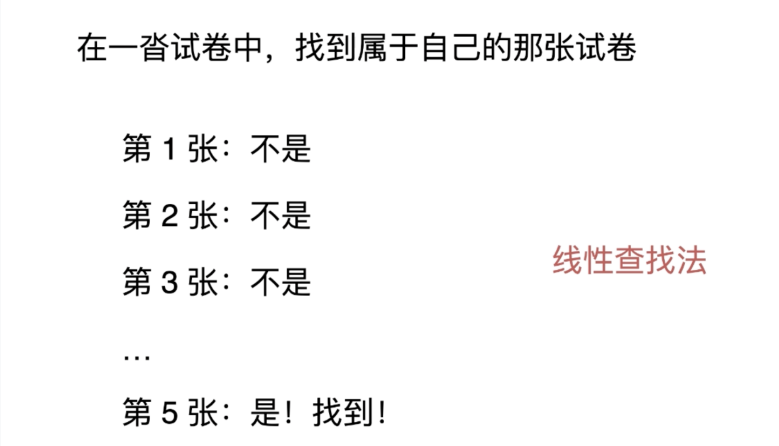
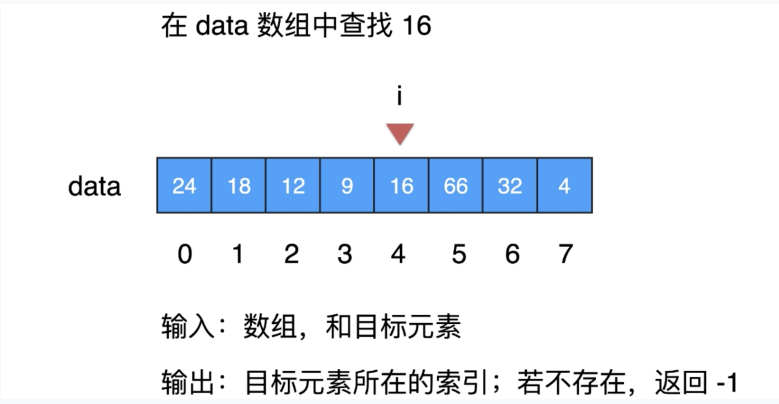
一.线性查找

- 什么是算法
  - 一系列解决问题的清晰、可执行的指令
  - 五大特性(了解)
    - 有限性：纵使执行1亿年，有期限就算有限性
    - 确定性：指令不会产生二义性。(比如生成随机数，这是符合确定性的，其指令是确定的)
    - 可行性：找出最大的质数，不可行（因为质数是无穷多个的）
    - 输入
    - 输出
- 什么是线性查找法
  - 生活：一沓试卷，找到属于自己的试卷。（一个个的顺序去找）

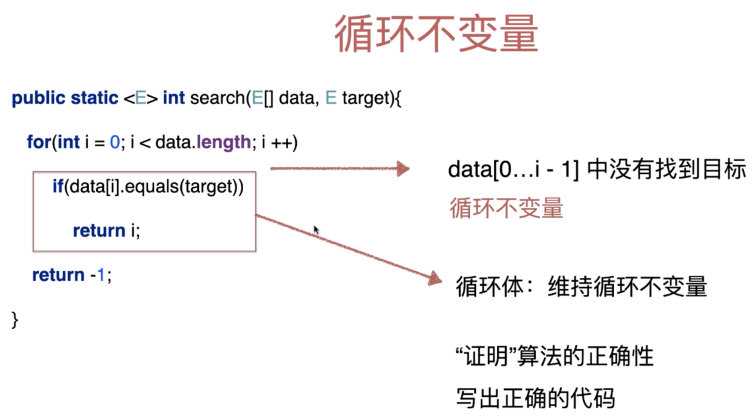


  - 计算机中：在数组中顺序查找某元素



- 实现线性查找
  - 泛型
    - 泛型，就是为了container能容纳不同的类型，不只是int、string等单一类型，这是强类型语言面临的共同问题。
    - Go中，使用interface{}，不能完整的实现功能。因为interface会比较 类型&值，两者都相等才是true

    - Go解决：
        - 定义接口 Comparable，定义方法 Equals()
        - 要比较的类，实现Comparable，Equals()内部定义如何比较
        - 比较的地方，调用Equals()，类似于java的Equals()
            - 【题外话】java为基本的数据类型，int、string等都实现好了Equals()，Go就需要自己实现了
    - reference：https://www.cnblogs.com/apocelipes/p/13832224.html
- 循环不变量
    - 就是for循环的条件，在每次for循环中，其要满足的条件都是不变的。循环体，就是要维持循环不变量



    - 写每一次for循环，我们都要明白循环不变量是什么，定义清楚循环不变量，循环体维护循环不变量，有利于写出正确代码。
- 复杂度分析
    - 目的：表示算法的性能
    - 通常：通常看最差的情况(target在data的末尾)，是算法运行的上界(最差就这个上限了，不可能比他还差了)

```go
func searchWithEquals(data []Comparable, target Comparable) int {
    for i := 0; i < len(data); i++ {
        if data[i].Equals(target) {
            return i
        }
    }
    return -1
}
```

    - 背景
        - 如何知道我这个算法执行了多少指令？转化成汇编？汇编也是不够的，汇编背后对应的是机器指令？机器指令？也是不够的，不同CPU架构，对机器指令有各自的优化
        - 纵使知道了又多少指令，那么，需要多长时间？不同指令、不同CPU，执行时间是不一样的。
    - 解决：好在，计算机世界很多时候需要个大概，把以上问题都化简
        - 不需要知道有多少实际的指令
        - 不需要知道实际执行多长时间

复杂度分析：表示算法的性能

```
public static <E> int search(E[] data, E target){
  for(int i = 0; i < data.length; i ++)
    if(data[i].equals(target))
      return i;
  return -1;
}
```

通常看最差的情况

算法运行的上界

n = data.length

T = n？ T = 2n？ T = 3n？ T = 4n？

T = 5n？ T = 5n + 2？ 单位：ms？

- 表示：**O(n)，表示算法性能与data数据大小n的关系**

  复杂度分析：表示算法的性能

  算法运行的上界

  T = 5n + 2？　　T = c1 * n + c2

  O(n)　　常数不重要

  复杂度描述的是随着数据规模 n 的增大，

  算法性能的变化趋势

  - 通常看最坏的情况，**也就是无穷大的情况，所以常数不重要**。
  - 【题外话】如果A算法运行时间是10000n，B算法运行时间是$2n^2$，那么是A算法好还是B算法好呢？
    - 答案，B算法运行好，因为算法我们通常考虑最坏的情况，也就是n无限大，n越大，肯定B越好

    T1 = 10000n　　T2 =$2n^2$　　$10000n < 2n^2$

    O(n)　　<　　O(n^2)　　n > 5000

    n_0 = 5000

    存在某一临界点 n_0, 当 n >= n_0, T1 < T2

- 常见算法复杂度
  - 常见复杂度
    - 明确n是谁，左边n代表数组的一个维度，复杂度是O(n^2)，右边a代表数组的一个维度，复杂度是O(a^2)

    遍历一个 n*n 的二维数组　O(n^2)

    ```
    for(int i = 0; i < n; i ++)
      for(int j = 0; j < n; j ++)
        // 遍历到 A[i][j]
    ```

    遍历一个 a*a 的二维数组　O(n)

    a*a = n

    ```
    for(int i = 0; i < a; i ++)
      for(int j = 0; j < a; j ++)
        // 遍历到 A[i][j]
    ```

    明确 n 是谁。
  - log复杂度不关心底是多少，因为底也是常数(log2n，结果表示n除以2多少次可以除完)

数字 n 的二进制位数 $O(logn)$

```
while(n){
    n % 2 // n 的二进制中的一位
    n /= 2;
}
```

- log2N，表示N被2除完需要多少次
- log10N，表示N被10除完需要多少次
- 【题外话】
  - 不能数循环的个数判断复杂度，这里n=n/2，其数据规模是 log2N，就是O(logn)

- O(√n)

数字 n 的所有约数 $O(n)$

```
for(int i = 1; i <= n; i ++)
    if(n % i == 0)
        // i 是 n 的一个约数
```

$O(\sqrt{n})$

```
for(int i = 1; i * i <= n; i ++)
    if(n % i == 0)
        // i 和 n / i 是 n 的两个约数
```

  - 这里不是O(logn),log是以某个常数为底，只不过常数可以忽略，意思是N被常数除以多少次可以除完
  - 这里优化后，i*i，不是常数，与$O(n^2)$相反，是O(√n)
- **O(2^n)，非常大的复杂度，n超过20，基本就不行了**
- **O(n!)，全排列公式，比O(2^n)还要大，2^n表示n个2相乘，n!除了最后1个数是1，其他都要大于等于2**
- O(1)，与n大小无关

- 总结时刻
  - **$O(1)<O(logn)<O(\sqrt{n})<O(n)<O(n*logn)<O(n^2)<O(2^n)<O(n!)$**
  - $logn<n$，所以$nlogn<n^2$
  - $\sqrt{n}<logn$，比如n=1000，$logn\approx10$，$\sqrt{n}\approx35$