# 极客时间算法训练营
# 第十八课
# 字符串处理

李煜东

《算法竞赛进阶指南》作者

# 目录

# 本课的重要性

字符串类题目是面试一大考点

这类题目算法一般不太复杂，但灵活多变，实现细节较多，容易出错

在实际应用中也比较普遍

属于熟能生巧的一系列题目

无论是Online assessment还是Onsite面试，挂在这类题目上就太可惜了

推荐课后多花一点时间练习

# 字符串基础知识

# 字符串

C++:

- `#include<string>`
- `string x = "abbc";`

Python:

- `x = 'abbc'`
- `x = "abbc"`

Java:

- `String x = "abbc";`

# 遍历字符串

C++

- ```
  string s = "abbc";
  ```
- ```
  for (int i = 0; i < s.length(); i++) {
    cout << s[i];
  }
  ```
- ```
  for (char ch : s) {
    cout << ch;
  }
  ```

Python

- ```
  for ch in "abbc":
    print(ch)
  ```

# 遍历字符串

Java

- ```
  String x = "abbc";
  ```
- ```
  for (int i = 0; i < x.size(); ++i) {
    char ch = x.charAt(i);
  }
  ```
- ```
  for (char ch : x.toCharArray()) {
    System.out.println(ch);
  }
  ```
  ```

# 字符串比较

字符串的值是否相等?

C++ / Python:

- `x == y`

Java

- `String x = new String("abbc");`
- `String y = new String("abbc");`
- `x == y` — false
- `x.equals(y)` — true
- `x.equalsIgnoreCase(y)` — true

# 基础问题（Homework 5选2）

转换成小写字母

https://leetcode-cn.com/problems/to-lower-case/

最后一个单词的长度

https://leetcode-cn.com/problems/length-of-last-word/

宝石与石头

https://leetcode-cn.com/problems/jewels-and-stones/

字符串中的第一个唯一字符

https://leetcode-cn.com/problems/first-unique-character-in-a-string/

最长公共前缀

https://leetcode-cn.com/problems/longest-common-prefix/description/

# 字符串操作（Homework 5选2）

反转字符串

https://leetcode-cn.com/problems/reverse-string

https://leetcode-cn.com/problems/reverse-string-ii/

翻转字符串里的单词

https://leetcode-cn.com/problems/reverse-words-in-a-string/

https://leetcode-cn.com/problems/reverse-words-in-a-string-iii/

仅仅反转字母

https://leetcode-cn.com/problems/reverse-only-letters/

# Atoi

字符串转换整数 (atoi)

https://leetcode-cn.com/problems/string-to-integer-atoi/

# Rabin-Karp 字符串哈希算法

# Rabin-Karp 算法

Rabin-Karp 是一种基于 Hash 的高效的字符串搜索算法

问题：

给定长度为 n 的字符串 s（文本串），长度为 m 的字符串 t（模式串）

求 t 是否在 s 中出现过（t 是否为 s 的子串）

朴素：$O(nm)$

Rabin-Karp 算法：$O(n+m)$

思路：

计算 s 的每个长度为 m 的子串的 Hash 值（一个宽度为 m 的滑动窗口滑过 s）

检查是否与 t 的 Hash 值相等

# Rabin-Karp 算法

选用的 Hash 函数：

把字符串看作一个 b 进制数（一个多项式），计算它（在十进制下）对 p 取模的值

举例：

取 b = 131，p = 2^64

字符串 foobar 的 Hash 值为（a=1, b=2, f=6, o=15, r=18）

$$(6 * 131^5 + 15 * 131^4 + 15 * 131^3 + 2 * 131^2 + 1 * 131 + 18) \bmod 2^{64}$$

选取的 b 和 p 的值决定了 Hash 函数的质量

根据经验， b = 131, 13331 等，p 为大质数，冲突概率极小

Hash 值相等时可以再比对一下两个字符串，避免 Hash 碰撞问题

# Rabin-Karp 算法

如何快速计算一个子串的 Hash 值?

s = "foobar"

先计算 6 个前缀子串的 Hash 值，$O(n)$:

$$H[i] = \text{Hash}(s[0 \ldots i-1]) = (H[i-1] * b + s[i-1]) \bmod p$$

计算子串 oba 的的 Hash 值:

相当于 b 进制下两个数做减法（$H[5] - H[2] * b^3$）

$$\begin{array}{r} \text{fooba} \\ -\ \underline{\text{fo000}} \\ \text{oba} \end{array}$$

$\text{Hash}(s[l \ldots r]) = \left(H[r+1] - H[l] * b^{r-l+1}\right) \bmod p$，$O(1)$

# 实战

实现 strStr()

https://leetcode-cn.com/problems/implement-strstr/

重复叠加字符串匹配（Homework）

https://leetcode-cn.com/problems/repeated-string-match/

# Rabin-Karp 算法的广泛适用性

由于 Rabin-Karp 算法 O(n) 预处理 + O(1) 求出任意子串哈希值的特性

配合二分查找、二分答案等技巧

可以作为字符串匹配、回文等一系列问题的次优解（可能比最优解多一个 log 的时间复杂度）

可以说几乎是一个万金油算法

# 字符串子串、回文、同构

# 回文串系列问题

验证回文串

https://leetcode-cn.com/problems/valid-palindrome/

验证回文字符串 II

https://leetcode-cn.com/problems/valid-palindrome-ii/

贪心+验证

最长回文子串

https://leetcode-cn.com/problems/longest-palindromic-substring/

中间向两边扩张 $O(n^2)$

加入二分 + Rabin-Karp优化，O(nlogn)

# 同构/异位词系列问题（Homework）

同构字符串

https://leetcode-cn.com/problems/isomorphic-strings/

有效的字母异位词

https://leetcode-cn.com/problems/valid-anagram/

字母异位词分组

https://leetcode-cn.com/problems/group-anagrams/

找到字符串中所有字母异位词

https://leetcode-cn.com/problems/find-all-anagrams-in-a-string/

# 字符串 + 动态规划

正则表达式匹配

https://leetcode-cn.com/problems/regular-expression-matching/

通配符匹配（Homework）

https://leetcode-cn.com/problems/wildcard-matching/

不同的子序列

https://leetcode-cn.com/problems/distinct-subsequences/

状态表示都比较类似（子问题也都比较明显）

$f[i][j]$ 表示 s 串的前 i 个字符和 t 串的前 j 个字符配，决策就是考虑末尾字符 i 和 j 怎么用

# 字符串模式匹配

# 字符串匹配

文本串 s

模式串 t

找出 t 在 s 中所有出现的位置

Brute-force：O(nm)

Rabin-Karp：O(n+m)

KMP：O(n+m)

# Rabin-Karp 字符串匹配

使用 Rabin-Karp 解决字符串匹配问题的思路是：

1. 计算长度为 m 的模式串 t 的 hash 值 hash_pattern，O(m)
2. 计算长度为 n 的文本串 s 中每个长度为 m 的子串的 hash 值，共需要计算 n-m+1 次
3. 比较每个子串和模式串的 hash 值，如果 hash 值不同，必然不匹配
4. 如果 hash 值相同，还需要使用朴素算法再次判断

在 hash 选取较好的情况下，可以做到 O(n+m)

# KMP 模式匹配算法

思考：朴素算法为什么慢？

尝试构造一组使朴素算法达到 O(nm) 的数据

```
s = "aaaaaaabaaac"
t = "aaaac"
```

```
aaaaaaabaaac
aaaac
 aaaac
  aaaac
   aaaac
```

每次失败前都比较了 m-1 次字符a，浪费时间

可以优化的地方：蓝色部分，我们能不能提前知道这部分是否相等呢？

# KMP 模式匹配算法

KMP 算法先对模式串 t 进行自匹配，求出一个数组 $next$

假设字符串下标从 1 开始

$next[i]$ 表示“$t$ 中以 $i$ 结尾的非前缀子串”与“$t$ 的前缀”能够匹配的最长长度，即：

$$next[i] = \max\{j\}，其中\ j < i\ 并且\ t[i-j+1 \sim i] = t[1 \sim j]$$

当不存在这样的 $j$ 时，令 $next[i] = 0$

$$\begin{array}{rl} & 1\ 2\ 3\ 4\ 5\ 6\ 7\ 8\ 9 \\ & \qquad\qquad\qquad\quad i \\ A = & a\ b\ \underline{a\ b\ a\ b\ a}\ a\ c \\ & \quad\ \underline{a\ b\ a\ b\ a}\ b\ a\ a\ c \\ & \qquad\qquad\qquad\quad j \\ & 1\ 2\ 3\ 4\ 5\ 6\ 7\ 8\ 9 \end{array}$$

$$next[7] = 5$$

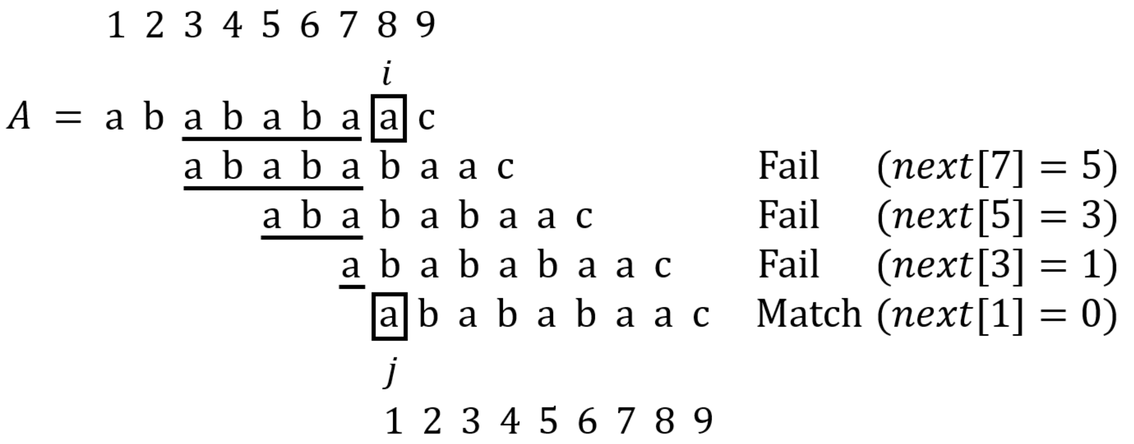
# KMP 模式匹配算法

数组 $next$ 的意义是什么？

顾名思义，$next$ 告诉我们下一个该比较什么

当求出 $next[i]=j$ 时

- 如果 $t[i+1]=t[j+1]$，自然 $next[i+1]=j+1$ —— 多匹配一个字符
- 如果 $t[i+1]\neq t[j+1]$，下一个可以直接比较 $t[i+1]$ 和 $t[next[j]+1]$，其他的可以跳过



# KMP 模式匹配算法

为什么？

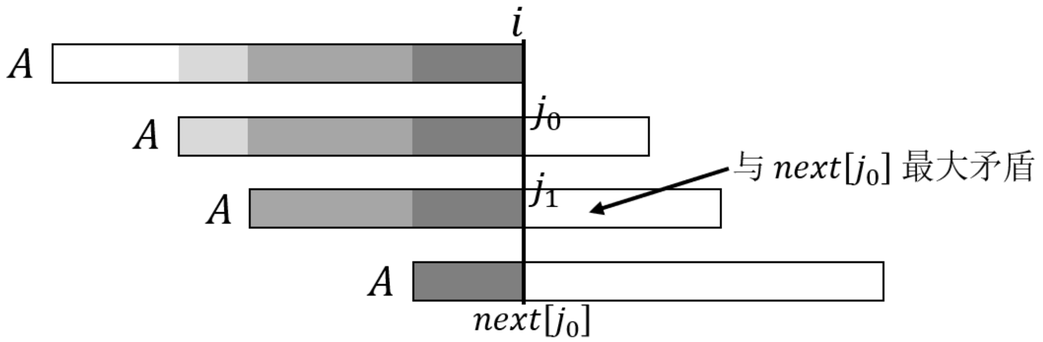
如果 $next[i] = j_0$（$1 \sim j_0$ 可以跟“$i$ 为结尾”匹配）

那么 $< j_0$ 的位置中，最大的能跟“$i$ 为结尾”匹配的就是 $next[j_0]$

不可能存在 $next[j_0] < j_1 < j_0$ 能跟 $i$ 匹配

可以用反证法证明

换言之，$next$ 告诉我们：如果匹配失败了，下一步跳到哪里继续配



# KMP 模式匹配算法

```
next[1] = 0;
for (int i = 2, j = 0; i <= m; i++) {  // 模式串t自匹配
    while (j > 0 && t[i] != t[j+1]) j = next[j];
    if (t[i] == t[j+1]) j++;
    next[i] = j;
}

for (int i = 1, j = 0; i <= n; i++) {  // 与文本串s匹配，过程相似
    while (j > 0 && (j == m || s[i] != t[j+1])) j = next[j];
    if (s[i] == t[j+1]) j++;
    f[i] = j;
    // if (f[i] == m)，此时就是t在s中的某一次出现
}
```

# KMP 模式匹配算法

时间复杂度?

看似是两重循环

在上面代码的 while 循环中，$j$ 的值不断减小

减小次数 ≤ 每层 for 循环开始时 $j$ 的值 - while 循环结束时 $j$ 的值

而在每层for循环中，$j$ 的值至多增加 1

因为 $j$ 始终非负，所以在整个计算过程中，$j$ 减小的幅度总和不会超过 $j$ 增加的幅度总和

故 $j$ 的总变化次数至多为 $2(N+M)$

整个算法的时间复杂度为 $O(N+M)$

# THANKS