字符串匹配专题

字符串匹配算法（String matching algorithms）是字符串问题下的一个**搜索问题**，因此它本质是一个搜索问题，不过和我们前面讲的 BFS，DFS， 回溯不太一样，这些侧重的是对搜索树进行搜索，即状态空间是非线性的。而本章的状态空间是线性的。

什么是字符串匹配

字符串匹配用来在一长字符串或文章中，找出其是否包含某一个或多个字符串以及其位置的算法。该算法的应用非常广泛，比如：生物基因匹配、信息检索（比如编辑器的 ctrl + f 搜索功能）等。

用数学语言描述如下:

假设T是一个长度为n的文本串，P是长度为m的模式串。如果有$0 <= \mathrm{s} < \mathrm{n} - \mathrm{m}$，使得$\mathrm{T}[\mathrm{s}, \mathrm{s}+1, ..., \mathrm{s}+\mathrm{m}-1]$等于$\mathrm{P}$，则称$\mathrm{P}$在$\mathrm{T}$中出现且位移为$\mathrm{s}$。

比如 T 为 "lucifer"， "lu"（开始位置为 0）， "cifer"（开始位置为 2） 在 T 中出现过。而 "xifa"， "hello" 等没有在 T 中出现过。

字符串匹配常见算法

暴力(Brute Force)

在日常编码生活中，我们肯定会遇到类似的问题，大部分数据量其实**不大**，串长也**较短**，因此自然会想到**窗口大小固定的滑动窗口**找出所有子串并依次和模式串按字母顺序依次比对看是否匹配并记录。

> 这里的窗口大小就是模式串的长度

Brute Force 有时候也简称为 BF。其核心思路非常朴素。该算法可以抽象为以下几步：

0. 非法情况处理，如模式串长度大于待匹配串等（防御性编程）

1. 初始化大小为模式串长的滑窗

2. 固定当前窗口，将当前窗口的子串与模式串匹配

2.1. 若匹配成功，则记录相关信息，如该位置下标

2.2. 否则窗口向后移动一格

伪代码

```
n = length[T]
m = length[P]

for s = 0 to n - m
  do if T[s..s+m] == P
    save info
```

时间复杂度$O(n * m)$, 空间复杂度$O(1)$

这种暴力算法对于数据规模小，串短的问题已经足够了，但是很多场景下数据规模很大，那暴力算法就显得捉襟见肘了，毕竟时间复杂度摆在那里，响应时间过长。

我们稍加分析其实不难发现，我们在每次窗口后移一位进行匹配的时候，实际上是把**上一个窗口的所有状态信息全部都丢掉不要了**，这会造成信息的浪费，那么都有哪些常见且优秀的解决方案呢?

核心其实还是前面讲滑动窗口提到的**变化仅仅是窗口边缘，窗口中间不变**。这就是 RK 算法的本质。

Rabin-Karp 算法(RK)

这个方法是本节最最推荐掌握的一个方法。其他方法可以不掌握，但这个解法务必掌握。

核心思路

RK 算法主要是对T中每个长度为m的子字符串$T[s..s+m]$进行 hash 运算，生成 hash 值$h1$，对P进行 hash 运算，生成 hash 值$h2$, 比对$h1$和$h2$，如果两个 hash 值(不考虑冲突)相等，则判断P在T中出现，且位移为s。

该算法需要对 P 计算一次哈希，并对 T 计算 n - m + 1 次哈希。而计算哈希的复杂度和**被计算哈希的字符串**长度线性相关，每步 hash 的时间复杂度为$O(m)$。 在这里被计算哈希的字符串长度为 m，因此**如果仅仅使用哈希，而不对算法进行任何其他优化的话**总的时间复杂度和前面提到的暴力法一样，最后的整体复杂度和 BF 一样都为$O(m * n)$。

RK 算法妙在滑动窗口的时候，设计了一个适合的哈希函数，有效保留了上一个状态的部分信息，这样**第一次**计算子串 hash 值时间复杂度为$O(m)$，而**后续就可以达到**$O(1)$。RK 算法最终的时间复杂度就降为$O(m + n)$。

这就是**有效利用了前面计算的窗口信息，而不是全盘计算。这不就是滑动窗口的精髓么？**。

该方法可以抽象为以下几步：

0. 非法情况处理，如模式串长度大于待匹配串等（防御性编程）

1. 计算出模式串 hash 值

2. 初始化大小为模式串长的滑窗并计算出 hash 值，判断当前 hash 值是否和模式串 hash 值相等。

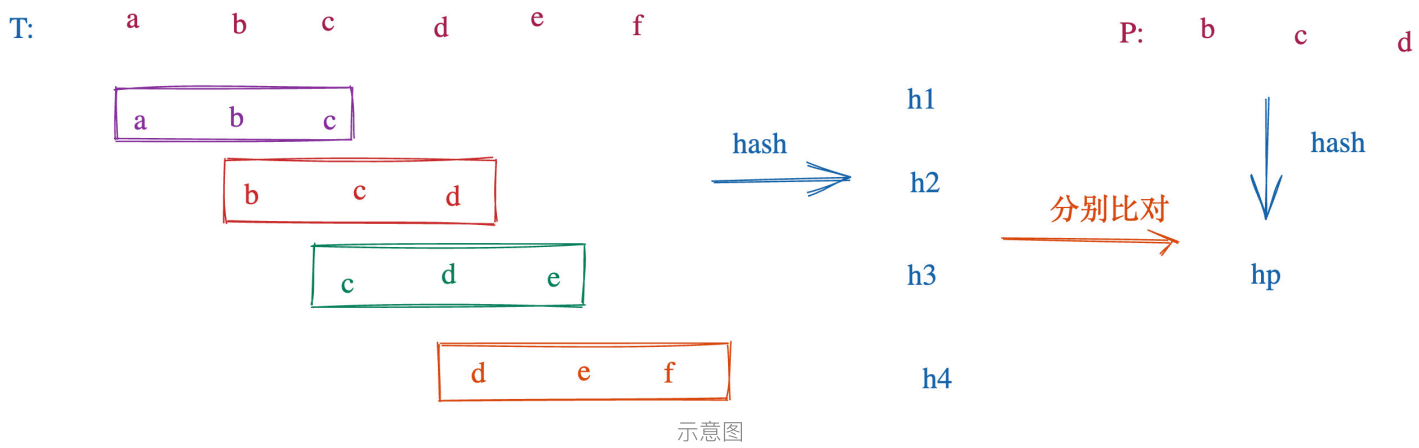
2.1. 若相等，则记录相关信息，如该位置下标

2.2. 否则窗口向后移动一格，并再次计算 hash 值（此处利用上个状态可直接一步计算）

伪代码

```
n = length[T]
m = length[P]
hp = hash(P)

for s = 0 to n - m
  hs = hash(T[s..s+m])
  if hp == hs and double check is right
    save info
```



示意图

我们这里选取的哈希函数为：$f(P) = P$ 表示的10进制值。

> 接下来的题目中，哈希函数可能会有所不同，比如 26 进制值来计算仅含 26 个英文字母的字符串的哈希。

假设P 和T 全由d个字符组成的（也就是说字符集大小为 d），则我们可以选择d进制表示P 和T，再将d进制转为10进制便于计算。

为了简化说明，我们更特殊地假设P 和T 全由[0-9]10 个数字组成。

P 的 10 进制为：

```
f(P) = P[0] * 10 ^ (m - 1) + P[1] * 10 ^ (m - 2) ... + P[m-1]*10^0
```

接下来，我们代入上面的哈希函数计算 T 的所有子串的哈希值。

$T[s..s+m]$的 10 进制为:

```
f(T[s..s + m]) = T[s] * 10 ^ (m - 1) + T[s + 1] * 10 ^ (m - 2) ... + T[s + m - 1]
```

相应地，下一个字符串$T[s+1..s+m]$的哈希值可以根据前面计算好的$T[s..s+m-1]$的哈希值推导出来。即**加上窗口中新增加的字符的哈希值贡献，再减去窗口中移除的字符的哈希值贡献**

该哈希算法是将字符串的每一个位的字符转换成对应的数字，再根据一定的权重（这里是 10）相乘得到一个数值。

具体过程如下：

```
f(T[s+1..s + m]) = T[s+1] * 10 ^ (m - 1) + T[s + 2] * 10 ^ (m - 2) ... + T[s + m]
= (f(T[s..s + m]) - T[s] * 10 ^ (m - 1)) * 10 + T[s + m ]
```

这样就把以上 `hp === hs` 的哈希比较转化为正常的 10 进制比较。

到目前为止，以上假设我们回避的一个问题是如果$f(P)$或者$f(T)$计算的 10 进制过大，导致**运算溢出**怎么办？比如我们用 32 位整形存储哈希值，那么如果计算的哈希值超过了 32 位整形所能表示的上限怎么办？

即使在 Python 等支持大数的语言，我们不会遇到溢出问题。这种算法同样有问题，因为这种算法性能会随着计算的哈希值变大而降低。

如何解决溢出和性能问题呢？

这里我们通过选择一个比较大的素数q, 计算后的 10 进制数对q**取模后**再进行比较。 由于我们使用对 q 取模，因此理论上来必然会有冲突的情况，并且冲突的平均个数 n/q。

取模的作用是防止溢出以及提供性能。但是有冲突的可能，$f(P)$并不能代表$f(P) == f(T[s])$。因此任何的$f(P)$都需要额外进行再次验证, 这里我们通过检测$P == T[s..s+m]$来完成。

从这里有可以看出，如果我们的代码冲突很多，那么效率和暴力法相差无几。极端情况，每次都哈希冲突，那么算法效率和暴力解一样，如果算法计算哈希值的开销，那么效率会比暴力法更差。

因此设计一个好的哈希算法，减少冲突是关键。如果你是参加面试或者比赛则可以尝试不同的模数来**调参**，直到通过所有的测试用例。不过这种做法非常不建议在工程中使用，即使在面试中，也希望你能先和面试官进行沟通之后再做决定。

常见的比较大的素数为 ${10**9 + 7}，为了提高性能可以选取更小的素数。

题目推荐：

- 5803. 最长公共子路径

总的来说，RK 算法就是：哈希函数 + 滑动窗口。使用哈希函数使得我们后续可以在 $O(1)$ 的时间计算字符串的哈希值。

扩展：

- 如果匹配 T 的任意一段字符串。我们可以用结合前缀和的技巧来完成。`prefix[i] = prefix[i-1] * base + s[i]`，这样子串 s[i:j] 的前缀就是 `prefix[j] - prefix[i-1] * base ** (j - i + 1)`。 推荐题目 6036. 构造字符串的总得分和

KMP

> 建议先观看文末扩展部分的视频后再来读讲义。

KMP 本质上是个预处理 + dp。

- 预处理指的是经过这样的处理一个模式串会生成一个 **唯一的 next 数组**，从而可以去匹配任意的主串。

> 唯一指的是给定模式串，生成的 next 数组就是确定的。和主串是没有任何关系的。

- dp 指的是建立 next 数组的部分使用到了动态规划的算法。

而匹配的过程，暴力（BF）算法中主串会有很多回溯，使用 KMP 可以**避免主串回溯，而只回溯模式串**。 形象地看就是模式串不停地在对齐主串。

核心思路

首先我们定义模式串的前缀函数 f(i) 为 模式串 P 中 P[1...i]相同前缀后缀的最大长度。对 P[1...m]中的每个 i，(i > 0 && i <= m), 用一个数组 next 记录。 KMP 算法由 Knuth，Morris 和 Pratt 三个大佬联合发明，KMP 算法名字由三个大佬名字首位字符组成。

首先我们定义模式串的函数$\mathbf{f(i)}$为模式串$\mathbf{P}$ 中$\mathbf{P[:i]}$相同前缀后缀的最大长度。对$\mathbf{P}$ 中的每个$\mathbf{i}$的信息，用一个数组$\mathbf{next}$统一记录。 KMP 算法在每次失配后，会根据上一次的比对信息跳转到相应的$\mathbf{s}$处，借助的就是上述的$\mathbf{next}$数组。推导过程可以参考 [从头到尾彻底理解 KMP](), 个人觉得这篇讲的非常透彻，这里就不班门弄斧了。该方法可以抽象为以下几步：

- 非法情况处理，如模式串长度大于待匹配串等（防御性编程）
- 计算出模式串的 next 数组。
- 开始从待匹配串开始进行匹配
- 若匹配成功，则记录相关信息；若失配，则按 next 数组回退到上一个待匹配状态继续进行匹配

以下是计算$\mathbf{next}$数组的伪代码

```
get_next(P):
  m  = P.length
  使得 next 为长度为m的数组
  next[1] = 0
  k = 0
  for i = 2 to m
    while(k > 0 并且 P[k+1] != P[i])
      k = next[k]
```

```
    if P[k+1] == P[i]
      k = k + 1
    next[i] = k
  return next
```

以下是 KMP 的伪代码

```
KMP(T, P)
  n = T.length
  m = P.length
  next = getNext(P)
  q = 0
  for let i = 1 to n:
    while(q > 0 并且 P[q + 1] !== T[i])
      q = next[q]
    if P[q + 1] == T[i]
      q = q + 1
    if (q == m)
      找到匹配位移 s = i
```

相关专题

最长公共前缀

前置知识：动态规划

实际上我们也可以用最长公共前缀（LCP） 来解决这个问题。

lcp(i,j) 表示字符串 s 从下标 i 开始的后缀和从下标 j 开始的后缀之间的最长公共前缀，那么转移方程为：

```
if s[i] == s[j]: lcp(i, j) = lcp(i+1, j+1) + 1
else: lcp(i,j) = 0
```

注意到状态 lcp(i,j) 依赖于状态 lcp(i+1,j+1)，因此我们需要倒序枚举。

完整求解 lcp 的 Python 代码如下：

```
lcp = [[0] * (n + 1) for _ in range(n + 1)]  # lcp[i][j] 表示 s[i:] 和 s[j:] 的最长公共前缀
for i in range(n - 1, -1, -1):
    for j in range(n - 1, i, -1):
        if s[i] == s[j]:
            lcp[i][j] = lcp[i + 1][j + 1] + 1
```

题目推荐：

- [6195. 对字母串可执行的最大删除数]

PS: 这道题 n^3 的暴力 dp 可以直接莽出来，和使用 lcp 优化时间竟然差不多。 这其实是测试用例太水的原因。附下代码：

```python
class Solution:
    def deleteString(self, s: str) -> int:
        n = len(s)
        lcp = [[0] * (n + 1) for _ in range(n + 1)]  # lcp[i][j] 表示 s[i:] 和 s[j:] 的最长公共前缀
        for i in range(n - 1, -1, -1):
            for j in range(n - 1, i, -1):
                if s[i] == s[j]:
                    lcp[i][j] = lcp[i + 1][j + 1] + 1
        @cache
        def dp(pos):
            if pos == n: return 0
            ans = 1
            for i in range(pos+1, n):
                if lcp[pos][i] >= i - pos:
                # 不使用 lcp 优化就注释掉上面的代码， 然后打开下面的代码即可
                # if s[pos:i] == s[i:i+i-pos]:
                    ans = max(ans, 1 + dp(i))
            return ans
        return dp(0)
```

总结

字符串匹配本质就是求一个模式串是否在主串中出现过，以及出现的具体位置。我们本章讲解了字符串匹配的三种常见方法：暴力法，RK 算法和 KMP 算法。

其中

- 暴力法简单直接
- RK 算法就是前面滑动窗口专题的应用，核心在于哈希函数的选择
- KMP 则是前面动态规划的应用，核心在于 next 数组的生成。

虽然本章是讲字符串的匹配，但是你可以将其进行简单的扩展，以实现知识的迁移。

比如给你两个数组 A 和 B，其中 A 为 [1,2,3,4,5] ，B 为 [2,8,9,10]。让你在 A 中 找到 B 的最长相似数组。其中相似数组指的是；如果数组 A[i], A[i+1] , ... A[j] 与 B[p], B[p+1] , ... B[q]，满足 A[i] == B[p] + x，A[i+1] == B[p+1] + x, ... A[j] == B[q] + x，其中 x 为任意数字。

这个题目可以将 A 和 B 做一个简单的变换，变换之后就可以看成是字符串匹配，即让你在 A 中找 B。

那么如何变化呢？答案是相邻项做差。

```
for i in range(1, len(A)):
  A[i] -= A[i-1]
  B[i] -= B[i-1]
```

经过这样的处理 A 和 B 就变为了：

```
A = [1,1,1,1,1]
B = [2,1,1,1]
```

可以看出除了首项外，A 和 B 相似数组的值是相同的。

我们先不看 A 和 B 的首项，这样的话我们只要在 A 中找到 [1,1,1] 就行了，最后在考虑首项即可。

此时我们就可以使用本章的方法来解决，比如 RK 算法（选择一个合适的哈希函数即可）。

总结

本节讲述了三种字符串匹配算法，分别为暴力匹配，RK（哈希匹配）和 KMP。

- 暴力解法必须要理解，但不推荐在做题中使用，除非数据量足够小。
- 最最推荐掌握的是 RK（哈希匹配），不管是思想还是复杂度都非常优秀。
- KMP 作为拔高，有能力的同学可以掌握一下。如果仍然学有余力可以尝试下后面的扩展内容。

扩展

推荐加练

- **792. 匹配子序列的单词数** 这道题虽然不是求子串，而是子序列，算法也不能套用当前章。但是却和本章内容有关，大家可以结合练习。另外还可以顺便复习一下哈希表。 强烈推荐 🔥🔥🔥🔥

> 这道题能使用前缀树解决么？为什么？

其他推荐题目：

- https://leetcode-cn.com/problems/longest-duplicate-substring/
- https://leetcode-cn.com/problems/shortest-palindrome/
- https://leetcode-cn.com/problems/maximum-length-of-repeated-subarray/

- https://leetcode-cn.com/problems/longest-chunked-palindrome-decomposition/
- https://leetcode-cn.com/problems/distinct-echo-substrings/
- https://leetcode-cn.com/problems/longest-happy-prefix/

基于有限自动机的字符串匹配

这个算法仅限了解即可，这里不做展开，感兴趣可以参考 **Finite Automata algorithm for Pattern Searching**

Z 函数（扩展 KMP）

- **oi-wiki** 相关题目 **6036. 构造字符串的总得分和**

参考

- **油管 KMP 讲解**(需翻墙)
- **维基百科**
- **从头到尾彻底理解 KMP**
- **Finite Automata algorithm for Pattern Searching**

上一页 下一页