数据结构与算法

Data Structure and Algorithm

极夜酱

目录

Chapter 1 模式匹配

1.1 BF

1.1.1 BF（Brute Force）

在一个字符串中查找另一个字符串的操作称为模式匹配。其中，被查找的字符串被称为文本串，需要查找的子串称为模式串。

查找子串的最简单的算法就是采用暴力匹配的方式。暴力匹配的基本思想就是逐个比较相应位置的字符。

假设文本串为 S，模式串为 P：

1. 如果当前字符匹配成功（即 S[i] == P[j]），则 i++，j++，继续匹配下一个字符。
2. 如果失配（即 S[i] != P[j]），令 i = i - j + 1，j = 0，相当于每次匹配失败时，i 回溯，j 被置为 0。

例如当文本串 S = "BBC ABCDAB ABCDABCDABDE"，P = "ABCDABD"。

S[0] 为 B，P[0] 为 A，不匹配（情形 2）。

0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**	**20**	**21**	**22**
B	B	C		A	B	C	D	A	B		A	B	C	D	A	B	C	D	A	B	D	E
A	B	C	D	A	B	D																

然后再判断 S[1] 和 P[0] 是否匹配，相当于模式串向右移动一位。S[1] 与 P[0] 还是不匹配（情形 2）。

0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**	**20**	**21**	**22**
B	B	C		A	B	C	D	A	B		A	B	C	D	A	B	C	D	A	B	D	E
	A	B	C	D	A	B	D															

然后再判断 S[2] 和 P[0] 是否匹配，模式串再向右移动一位。直到 S[4] 与 P[0] 匹配成功（情形 1）。

0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**	**20**	**21**	**22**
B	B	C		A	B	C	D	A	B		A	B	C	D	A	B	C	D	A	B	D	E
				A	B	C	D	A	B	D												

此时 i = 5，j = 1，接下来判断 S[5] 与 P[1] 是否匹配。S[5] 与 P[1] 匹配成功（情形 1）。可得 i = 6，j = 2。

0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**	**20**	**21**	**22**
B	B	C		A	B	C	D	A	B		A	B	C	D	A	B	C	D	A	B	D	E
				A	B	C	D	A	B	D												

直到 S[10] 为空格，P[6] 为 D，不匹配（情形 2）。

0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**	**20**	**21**	**22**
B	B	C		A	B	C	D	A	B		A	B	C	D	A	B	C	D	A	B	D	E
				A	B	C	D	A	B	D												

此时 i = 5，j = 0，相当于判断 S[5] 跟 P[0] 是否匹配。

0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**	**20**	**21**	**22**
B	B	C		A	B	C	D	A	B		A	B	C	D	A	B	C	D	A	B	D	E
					A	B	C	D	A	B	D											

至此可以看出，按照暴力匹配算法的思路。同理，直到找到匹配的字符串或文本串遍历结束退出。

BF 暴力匹配在最坏情况下时间复杂度是 $O(mn)$。

BF

```
1 def brute_force(s, p):
```

```
    s_len = len(s)
    p_len = len(p)
    i = 0
    j = 0

    while i < s_len and j < p_len:
        if s[i] == p[j]:
            i += 1
            j += 1
        else:
            i = i - j + 1
            j = 0

    if j == p_len:
        return i - j
    else:
        return -1
```

1.2 Sunday

1.2.1 Sunday

Sunday 是从前往后匹配的算法，在匹配失败时重点关注的是文本串中参加匹配的最末位字符的下一位字符。如果该字符没有在模式串中出现则直接跳过，移动位数为模式串长度 + 1，否则移动位数为模式串长度 - 该字符最右出现的下标。

Sunday 算法巧妙的地方在于它发现匹配失败之后可以直接考察文本串中参加匹配的最末尾字符的下一个字符。

例如当文本串 S = "bcaitsnaxzfinihao"，P = "nihao"。

S[0] 为 b，P[0] 为 n，匹配失败。关注文本串中参加匹配的最末位字符的下一位字符 s，该字符并没在模式串中出现，因此将模式串向右移动模式串长度 + 1，即 5 + 1 = 6。

0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**
b	c	a	i	t	s	n	a	x	z	f	i	n	i	h	a	o
n	i	h	a	o												

S[7] 为 a，P[1] 为 i，匹配失败。关注文本串中参加匹配的最末位字符的下一位字符 i，该字符在模式串中最右出现出现下标为 1，因此将模式串向右移动模式串长度 - 最右下标，即 5 - 1 = 4。

0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**
b	c	a	i	t	s	n	a	x	z	f	i	n	i	h	a	o
						n	i	h	a	o						

0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**
b	c	a	i	t	s	n	a	x	z	f	i	n	i	h	a	o
										n	i	h	a	o		

S[10] 为 f，P[0] 为 n，匹配失败。关注文本串中参加匹配的最末位字符的下一位字符 a，该字符在模式串中最右出现出现下标为 3，因此将模式串向右移动模式串长度 - 最右下标，即 5 - 3 = 2。

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
b	c	a	i	t	s	n	a	x	z	f	i	n	i	h	a	o
										n	i	h	a	o		

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
b	c	a	i	t	s	n	a	x	z	f	i	n	i	h	a	o
												n	i	h	a	o

此时，模式串匹配成功。

Sunday

```python
def sunday(s, p):
    s_len = len(s)
    p_len = len(p)
    i = 0
    j = 0
    result = 0

    while i < s_len and j < p_len:
        if s[i] == p[j]:
            i += 1
            j += 1
            continue

        idx = result + p_len
        if idx >= s_len:
            return -1

        k = p_len - 1
        while k >= 0 and s[idx] != p[k]:
```

```
            k -= 1

        i = result
        i += p_len - k
        result = i
        j = 0

        if result + p_len > s_len:
            return -1

    return result
```

1.3 RK

1.3.1 RK（Rabin-Karp）

RK 算法的命名由 Rabin 和 Karp 两位发明者的名字而来，它的实现方式有点与众不同。BF 算法只是简单粗暴地对两个字符串的所有字符依次比较，而 RK 算法比较的是两个字符串的哈希值。

RK 算法的基本思想就是将模式串 P 的哈希值跟文本串 S 中每一个长度为 |S| 的子串的哈希值进行比较。如果两个字符串哈希值不相同，则它们肯定不匹配。如果它们的哈希值相同，它们有可能匹配（因为可能存在哈希冲突）。

由于哈希函数有可能会产生哈希冲突，哈希值相等的两个字符串不一定相同。因此如果两个字符串的哈希值相等，就把这两个字符串本身进行一次比较即可。这种方法的前提是要控制冲突概率，达到可以接受的状态。

不过每次 hash 的时间复杂度为 $O(n)$，如果把全部子串都进行 hash，总的时间复杂度不是和 BF 算法一样，都是 $O(mn)$ 了吗?

其实对子串的 hash 计算并不是独立的，从第二个子串开始，每一次子串的 hash 都可以由上一次子串进行简单的加减得到（减去第一个字符的 hash，加上新字符的 hash）。因此通过设计特殊的哈希算法，只需扫描一遍文本串就能计算出所有子串的哈希值。期间最多需要比较 m - n + 1 个子串的哈希值。

相比于 BF 算法，RK 算法采用哈希值比较的方式，免去了许多无谓的字符比较，所以时间复杂度大大提高了。RK 算法的缺点在于哈希冲突，每一次哈希冲突的时候，RK 算法都要对子串和模式串进行逐个字符的比较。如果冲突太多了，RK 算法就退化成了 BF 算法。

RK

```
1 public static int rk(String s, String p) {
```

```
    int sLen = s.length();
    int pLen = p.length();

    int sHash = hash(s.substring(0, pLen)); // 文本串子串哈希值
    int pHash = hash(p);                    // 模式串哈希值

    for(int i = 0; i < sLen - pLen + 1; i++) {
        if(sHash == pHash) {
            if(match(s.substring(i, i + pLen), p)) {
                return i;
            }
        }
        if(i < sLen - pLen) {
            sHash = nextHash(s, sHash, i, pLen);
        }
    }

    return -1;
}

public static int hash(String s) {
    int hashCode = 0;
    for(int i = 0; i < s.length(); i++) {
        hashCode += s.charAt(i) - 'a';
    }
    return hashCode;
}

public static int nextHash(String s, int hash, int start, int n) {
    hash -= s.charAt(start) - 'a';
    hash += s.charAt(start + n) - 'a';
    return hash;
}

public static boolean match(String s, String p) {
    return s.equals(p);
}
```

1.4 BM

1.4.1 BM（Boyer-Moore）

BM 算法的名字取自于它的两位发明者，计算机科学家 Bob Boyer 和 J Strother Moore。

为了能减少比较，BM 算法制定了两条规则：

- 坏字符规则（bad character）
- 好后缀规则（good suffix）

1.4.2 坏字符规则

坏字符是指文本串与模式串不匹配的字符。

0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**	**20**
G	T	T	A	T	A	G	C	T	G	G	T	A	G	C	G	G	C	G	A	A
G	T	A	G	C	G	G	C	G												

咦？为什么坏字符不是主串中下标为 2 的字符 T 呢？那个位置不是先被检测的到吗？

BM 算法的检测顺序是从字符串的最右侧向最左侧进行的。当检测到第一个坏字符后，并没有必要让模式串一位一位向后挪动并比较。因为只有模式串与坏字符对齐的位置相同的情况下，两者才有匹配的可能。由于模式串的第 1 位字符也是 T，这样就可以直接把模式串中的 T 与文本串的坏字符对齐，进行下一轮比较。

0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**	**20**
G	T	T	A	T	A	G	C	T	G	G	T	A	G	C	G	G	C	G	A	A
							G	T	A	G	C	G	G	C	G					

坏字符的位置越靠右，下一轮模式串的挪动跨度就可能越长，节省的比较次数也就越多。这就是 BM 算法从右向左检测的好处。

接着，从右向左成功匹配 GCG，并遇到坏字符 A。

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
G	T	T	A	T	A	G	C	T	G	G	T	A	G	C	G	G	C	G	A	A
							G	T	A	G	C	G	G	C	G					

按照类似的方式，找到模式串的第 2 位字符 A，将它与文本串的坏字符对齐。

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
G	T	T	A	T	A	G	C	T	G	G	T	A	G	C	G	G	C	G	A	A
										G	T	A	G	C	G	G	C	G		

如果出现坏字符在模式串中不存在的情况，就直接把模式串挪到主串坏字符的下一位。

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
G	T	T	A	T	A	G	C	T	G	G	T	A	G	C	G	G	C	G	A	A
G	C	A	I	C	G	G	C	G												

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
G	T	T	A	T	A	G	C	T	G	G	T	A	G	C	G	G	C	G	A	A
									G	C	A	I	C	G	G	C	G			

1.4.3 好后缀规则

好后缀是指文本串与模式串当中相匹配的后缀。

例如对于这个例子，如果继续使用坏字符规则，模式串只能向后挪动一位。

0	1	2	3	4	5	6	7	8	9	10	11	12	13
C	T	G	G	G	C	G	A	G	C	G	G	A	A
G	C	G	A	G	C	G							

为了能真正减少比较次数，就需要使用好后缀规则。在第一轮比较中，文本串和模式串都有共同的后缀 GCG，这就是所谓的好后缀。如果模式串的其它位置也包含与 GCG 相同的子串，那么就可以挪动模式串，让这个子串与好后缀对齐。

0	1	2	3	4	5	6	7	8	9	10	11	12	13
C	T	G	G	G	C	G	A	G	C	G	G	A	A
G	C	G	A	G	C	G							

0	1	2	3	4	5	6	7	8	9	10	11	12	13
C	T	G	G	G	C	G	A	G	C	G	G	A	A
				G	C	G	A	G	C	G			

如果模式串中不存在其它与好后缀相同的片段，是不是可以直接把模式串挪到好后缀之后?

使不得! 这里还要判断一种特殊情况，模式串的前缀是否和好后缀的后缀相匹配，免得挪过头了。

0	1	2	3	4	5	6	7	8	9	10	11	12
T	G	G	G	C	G	A	G	C	G	G	A	A
C	G	A	G	C	G							

0	1	2	3	4	5	6	7	8	9	10	11	12
T	G	G	G	C	G	A	G	C	G	G	A	A
				C	G	A	G	C	G			

那应该什么时候用坏字符规则，什么时候用好后缀规则呢?

可以在每一轮字符比较之后，按照坏字符和好后缀规则分别计算相应的挪动距离，哪一种距离更长，就把模式串挪动对应的长度。比如坏字符可以让模式串在下一轮挪动 3 位，好后缀可以让模式串移动 5 位，那么就应该采用好后缀规则。

1.5 KMP

1.5.1 KMP（Knuth-Morris-Pratt）

KMP 算法是一个里程碑似的算法，它的出现宣告了人类找到了线性时间复杂度的字符串匹配算法。在此之后才出现了其它线性时间的字符串匹配算法，比如 BM 算法和 Sunday 算法。

KMP 算法由三位计算机科学家 D. E. Knuth、J. H. Morris 和 V. R. Pratt 提出，KMP 这个算法名字正是取自这三个人的姓氏首字母。

与 BM 算法类似，KMP 算法也在试图减少无谓的字符比较，但 KMP 算法把专注点放在了已匹配的前缀。

在每一次匹配过程中，其实可以判断出后续几次匹配是否会成功。算法的核心就是每次匹配过程中推断出后续完全不可能匹配成功的部分，从而减少比较的次数。

例如主串中已匹配末尾的 GTG 是最长可匹配后缀，模式串中开头的 GTG 是最长可匹配前缀。

0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**	**20**
G	T	G	T	G	A	G	C	T	G	G	T	G	T	G	T	G	C	F	A	A
G	T	G	T	G	C	F														

将最长可匹配后缀与最长可匹配前缀对齐。

0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**	**20**
G	T	G	T	G	A	G	C	T	G	G	T	G	T	G	T	G	C	F	A	A
		G	T	G	T	G	C	F												

主串中已匹配末尾的 G 是最长可匹配后缀，模式串中开头的 G 是最长可匹配前缀。

0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**	**20**
G	T	G	T	G	A	G	C	T	G	G	T	G	T	G	T	G	C	F	A	A
		G	T	G	T	G	C	F												

将最长可匹配后缀与最长可匹配前缀对齐。

0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**	**20**
G	T	G	T	G	A	G	C	T	G	G	T	G	T	G	T	G	C	F	A	A
				G	T	G	T	G	C	F										

KMP 算法的整体思路就是在已匹配的前缀当中寻找最长可匹配后缀子串和最长可匹配前缀子串，在下一轮直接把两者对齐，从而实现模式串的快速移动。

那么如何找到一个字符串前缀的最长可匹配后缀子串和最长可匹配前缀子串呢？难道在每一轮都要重新遍历吗？

要找到这两个子串没有必要每次都去遍历，可以事先缓存到一个集合当中，用的时候再去集合里面取。这个集合被称为 next 数组，如何生成 next 数组是 KMP 算法的最大难点。

1.5.2 next 数组