KMP 与 AC 自动机

KMP 算法

KMP 是经典的线性时间复杂度的字符串匹配算法。

称要被匹配的串为模板串，而要在其中寻找模板串的串为文本串。KMP 的核心思想是：先对模板串进行预处理，使其产生某些可以利用的信息，然后在匹配时高效率匹配。这些“可以利用的信息”就是下面要介绍的 `next` 数组。

设原始串为 $s[0...len)$，`next[i]` 表示 $s[0...i)$ 最长的**前缀配后缀的长度** -1（之所以要减去 1 是为了方便移动，如果不减去 1 就最好设原始串为 $s[1...len]$）。

这样，如果在匹配到 $s[i]$ 的时候失败了，由于这时，$s[0...i)$ 都匹配成功了，而为了避免这些匹配成功的部分被浪费，就可以直接令当前匹配位置回到 `next[i]`，重新尝试匹配。

下面这个图给出了一个很好的解释：

> 文本串：YZZYZY...
>
> 模板串：YZZYZZ... 在模板串下标为 5 处失败
>
> ⬇（向着 `next[5] = 1` 移动）
>
> 文本串：YZZYZY...
>
> 模板串：　　YZZ... 从模板串下标为 2 处重新开始

（补充求出 `next` 的过程）

```
void build_fail(char *pat, int len, int *fail){
    fail[0] = -1;
    for(int i = 1, j = -1; i < len; ++i){
        while(j > -1 && pat[i] != pat[j + 1])
            j = fail[j];
        if(pat[i] == pat[j + 1])
            fail[i] = ++j;
        else
            fail[i] = -1;
    }
}
void KMP_match(char *text, int l1, char *pat, int l2, int *fail){
    for(int i = 0, j = -1; i < l1; ++i){
        while(j > -1 && text[i] != pat[j + 1])
            j = fail[j];
        if(text[i] == pat[j + 1])
            j++;
        if(j == l2 - 1){
            //do something...
        }
    }
}
```

扩展 KMP 算法（Z 算法）

定义 Z 函数：对于串 $S[0...n), T[0...m)$，$z[i]$ 表示 $S[i...n)$ 和 T 的最长公共前缀。我们希望有一个算法，能够高效求出每一个位置上的 Z 函数。这个算法就是扩展 KMP 算法，或 Z 算法。

之所以称这个算法为扩展 KMP 算法，是因为如果 $\exists i, z[i] = m$，那么就表明 T 在 S 中出现过。而这和 KMP 算法的目标是一样的。

要求 Z 函数，$O(n^2)$ 的暴力是容易实现的。我们考虑像 KMP 算法一样利用一些有用的信息。先假设我们已经求出了 $z[0...i)$，现在我们要求 $z[i]$。类比 Manacher 算法的思想，我们考虑某一个位置 $p = \arg\max(i + z[i] - 1)$。它的直观意义是和 T 相匹配时延伸到最右的，S 的某个后缀的起始位置。同样，我们讨论 p 和 i 的关系：（方便起见，下面设 $r = p + z[p] - 1$）

1. $i > r$。这时，我们似乎不能得到任何的有效信息，于是只有通过暴力匹配计算 $z[i]$。
2. $i \le r$。这时，通过匹配关系，我们可以得出：$S[p...r] = T[0...r-p]$。那么 $S[i...r] = T[i-p...r-p]$。这件事似乎并没有说明什么，因此我们不妨先考虑一个特殊情况：$S = T$。假设我们已经求出来了这种情况下的 Z 函数数组，并设其为 next。（这里和 KMP 中的在定义上稍微有些不同）

   回到一般情况，我们可以用上述 next 数组方便求解。考虑 $next[i-p]$，设其为 l。则：

   1. $l \le r - i$。那么直接令 $z[i] = l$。因为如果 $z[i] > l$，那么至少存在 $l' > l$，使得 $S[i...i+l') = T[0...l') = T[i-p...i-p+l')$。而根据 next 数组的定义，这表明 $next[i-p] > l$，矛盾。
   2. $l > r - i$。那么先令 $z[i] = l$，而在 $S[r]$ 之后的情况就完全不清楚了，所以这时候直接从 $S[r+1]$ 开始暴力匹配，并且更新 p 和 r。

在 $S = T$ 时，我们可以使用之前已经得到了的 next 数组求解剩余部分。这样，我们可以先对 T 自身做一次扩展 KMP（相当于预处理），然后再对 S 做一次。

和对 Manacher 算法分析时间复杂度的过程类似，可以证明两种情况下的时间复杂度都是线性的。所以整个扩展 KMP 的时间复杂度是线性的。

下面给出一个简单的实现：

```
void exKMP(char s[], char t[], int n, int m, int nxt[], int z[]){
    nxt[0] = m;
    int j = 0, p = 1;
    while (j + 1 < m && t[j] == t[j + 1])
        ++j;
    nxt[1] = j;
    for (int i = 2; i < m; ++i) {
        int r = p + nxt[p] - 1, l = nxt[i - p];
        if (l <= r - i) nxt[i] = l;
        else {
            j = max(0, r - i + 1);
            while (j + i < m && t[j] == t[j + i])
                ++j;
            nxt[i] = j, p = i;
        }
    }

    j = 0, p = 0;
    while (j < n && j < m && s[j] == t[j])
        ++j;
    z[0] = j;
    for (int i = 1; i < n; ++i){
        int r = p + z[p] - 1, l = nxt[i - p];
        if (l <= r - i) z[i] = l;
        else {
```

```
            j = max(0, r - i + 1);
            while (j + i < n && j < m && t[j] == s[j + i])
                ++j;
            z[i] = j, p = i;
        }
    }
}
```

上述代码将 $i > r$ 和 $l > r - i$ 的情况做了整合。