字符串 1

张博凯

天津大学

2023 年 7 月 29 日

目录

目录

字符串哈希

Description

字符串哈希是一种将字符串映射成整数的方法。如果有一个高效的方法将每个字符串映射为一个整数，那么字符串匹配问题就变成了这两个字符串对应的整数是否相等的问题。

计算哈希值

字符串 $s_1 s_2 s_3 s_4 \ldots s_n$ 的哈希值为

$$(s_1 \times b^{n-1} + s_2 \times b^{n-2} + s_3 \times b^{n-3} + \ldots + s_n) \mod p$$

其中 p 为一大质数 (10^9 附近)，b 为一个大于 s_n 值域的整数。可以看作将字符串看成一个 b 进制大整数，再对大质数 p 取模。

字符串哈希

为何优越？

使用字符串哈希如何使字符串匹配更简便？
首先，由于 p 选取大质数，所以字符串的哈希值基本可以看作是随机均匀分布在 $[0, p-1]$ 之间。任意两个不同的字符串的哈希值相同的概率可以近似看作 $\frac{1}{p}$，错误的概率很小。
其次，子串的哈希值容易计算。给定一个字符串，每次询问一个子串的哈希值，我们只需提前预处理每个前缀的哈希值，然后利用这个式子：
$s_l \times b^{r-l} + s_{l+1} \times b^{r-l-1} + \ldots + s_r =$
$(s_1 \times b^{r-1} + s_2 \times b^{r-2} + \ldots + s_r) - (s_1 \times b^{l-2} + s_2 \times b^{l-3} + \ldots + s_{l-1}) \times b^{r-l+1}$
即可 $O(1)$ 得到子串的哈希值。

字符串哈希

例题

给定一个长度为 n 的字符串，q 次询问，每次给定两组 l, r ，求这两个子串的最长公共前缀。
$n \leq 10^5, q \leq 10^5$

字符串哈希

解答

注意到到最长公共前缀这个问题具有单调性，即若答案为长度为 l 的前缀，则长度小于 l 的前缀都相同，而长度大于 l 的前缀都不相同。那么二分答案。现在问题转化为每次给定两个字符串的两个前缀，判断是否相等。由于一个字符串的子串的前缀还是这个字符串的子串，所以可以 $O(1)$ 算出哈希值，$O(1)$ 判断。总时间复杂度为 $O(n + q\log n)$。考虑正确率，共进行 $q\log n$ 次哈希比较，出错的概率应该在 $1-(1-\frac{1}{10^9})^{1700000}$ 级别。计算出约为千分之二，可以接受。

字符串哈希

生日悖论

正如前面的例题，在多数情况下，字符串哈希的正确性都是能够接受的。但在生日悖论情形下，字符串哈希的正确性会受到挑战。
生日悖论：当屋子里有 23 个人时，存在两个人同一天生日的概率就会超过百分之五十。
更一般地，若随机数的值域为 n，则生成 $O(\sqrt{n})$ 个随机数，则会有超过一半的概率存在两个数相同。
于是，当你注意到，对于一个比较大的字符串集合，只要其中存在某两个字符串的哈希冲突了，就会导致答案出错，那么这道题目就属于生日悖论的范畴，哈希算法的正确率就会不能接受。

字符串哈希

更大的值域

在哈希的正确率不能接受时，除了寻找哈希以外的解法，还有一种方法是扩大哈希的值域，将哈希值的值域扩大为 10^{18} 甚至更大。这时一般有两种方法。

双模数哈希：采用两个模数，分别计算哈希值，仅当字符串的两个哈希值都对应相等时认为两个字符串相同。这种做法的依据是中国剩余定理。对于两个质数 p, q，如果知道 x 分别对 p, q 的余数，就能唯一推出 x 对 $p \times q$ 的余数。于是值域扩大为了 $p \times q$。

整数溢出哈希：哈希值使用 unsigned long long 存储，省略所有取模运算。这种方法简单省力，值域大，运算快。但唯一的缺点在于：出题人可以定向卡。存在一种构造字符串的方式，让这种哈希产生冲突，与选取的进制数无关。详见 hash killer。同样的，对于一般的哈希算法，如果出题人知道你的模数，也可以构造数据定向卡掉。出题人有时会卡掉常见的模数，如 $10^9 + 7, 998244353$。最好自己背一个不常见大质数防止这种情况出现。

目录

KMP 算法

Description

KMP 解决这样的问题：给定两字符串 s 和 t，$O(|s|+|t|)$ 求出 s 在 t 中的每个出现位置。

一些定义

- sub(l, r)：$s_l s_{l+1} \dots s_r$
- pre(s, i)：s 的长为 i 的前缀。
- suf(s, i)：s 的长为 i 的后缀。
- border：若 $0 \le r < |s|$，pre(s, r) = suf(s, r)，就称 pre(s, r) 是 s 的 border。
- next 数组：next[i] 表示 pre(s, i) 的最长 border 的长度。

KMP 算法

求 next 数组

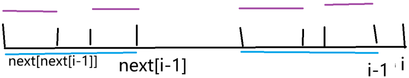
要解决模式串在主串中的匹配问题，我们首先要求模式串的 next 数组。在原始问题中，s 为模式串，则要求 s 的 next 数组。注意到，如果 next[i] $>$ 1，则 sub(1, next[i]) $=$ sub(i-next[i]$+$1, i)。由此可推出 sub(1,next[i]-1) $=$ sub(i-next[i], i-1)。那么 next[i]-1 一定是 next[i-1] 的一个备选，即 next[i-1] $\geq$ next[i]-1。那么 next[i] $\leq$ next[i-1]$+$1。如果 s[i] $=$ s[next[i-1]$+$1]，则 sub(1,next[i-1]$+$1) $=$ sub(i-next[i-1], i)。即 next[i] $=$ next[i-1]$+$1。否则，同理上述思路可以得到，next[i] $\leq$ next[next[i-1]]$+$1。循环往复地执行跳 next 的操作，直到匹配成功，找到 next[i] 的值。

代码

```
next[1] = 0;
for(int i = 2, j = 0; i <= n; i++){
    while(j && s[i] != s[j + 1]) j = next[j];
    if(s[i] == s[j + 1]) j++;
    next[i] = j;
}
```

KMP 算法

图解



时间复杂度分析

注意到这个算法有两层循环，容易错误分析成 $O(n^2)$。但是注意这里指针 j 的移动。每执行一次内层的 while 循环，j 的值至少减少 1。然而，j 总共只加了 n 次 1，它又不能减到 0 以下，于是总共最多只执行了 n 次 while 循环，这个算法的时间复杂度为 $O(n)$。

KMP 算法

使用 KMP 算法来求解字符串匹配

一个通常的做法是，求出模式串的 next 数组后，再做一次和上面的算法几乎一样的操作，来得到 s 在 t 中的每个出现位置。这其中的思想也是几乎一样的。这里直接看代码。

```
for(int i = 1, j = 0; i <= m; i++){
    while(j && (j == n || t[i] != s[j + 1])) j = next[j];
    if(t[i] == s[j + 1]) j++;
    if(j == n) printf("%d\n", i - n + 1);
}
```

另一种方式是，直接把 s 和 t 拼成一个新的字符串，即 $s\#t$。这里井号代表任意不在字符集中的字符。计算这个字符串的 next 数组，所有 $\text{next}[\text{i}] = \text{len}(\text{s})$ 的位置都是 t 中出现 s 的位置。这个做法也非常显而易见。仔细考虑会发现，上面提到的两种做法其实本质相同。

KMP 算法

一些题目

给定一个字符串 s，统计 s 的每个前缀在 s 中的出现次数。
给定一个字符串 s，求最少在末尾添加几个字符，使其可以由另一个字符串复制拼接至少两次得到。（有一个整周期）

目录

manacher 算法

Description

manacher 算法解决这样的问题：给定一个字符串 s，求 s 每个位置的最长回文半径。即对每个 $i, i = 2, 3, ..., 2n$，求最长的子串 $sub(l, r)$ 是回文串且 $l + r = i$。

哈希

首先容易发现可以使用二分哈希 $O(n \log n)$ 解决这个问题。事实上很多字符串问题都可以使用哈希解决。虽然它可能不够快，但它仍是非常重要的方法，一定要掌握。

manacher 算法

预处理

首先，对于偶数长度的回文串，不存在回文中心。为了方便，我们在字符串中间插入特殊字符，变为 $\#s_1\#s_2\#s_3\ldots\#s_n\#$。这样，偶数长度的回文串的回文中心变成了 $\#$。解决了这个问题。

manacher 算法

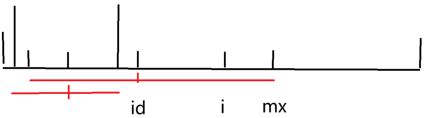
算法主体

```
string s;
cin >> s;
string t = "$#";
for(int i = 0; i < s.size(); i++){
    t += s[i];
    t += "#";
}
vector<int> p(t.size(), 0);
int mx = 0, id = 0, resLen = 0, resCenter = 0;
//mx 为当前最长回文串的右边界，id为当前最长回文串的中心
//p[i]为以i为中心的回文串的半径
for(int i = 1; i < t.size(); i++){
    p[i] = mx > i ? min(p[2 * id - i], mx - i) : 1;
    while(t[i + p[i]] == t[i - p[i]]) p[i]++;
    if(mx < i + p[i]){
        mx = i + p[i];
        id = i;
    }
}
```

核心思想为，如果有一个较长的已知的回文串 a 包含了当前这个需要求解的回文中心 i，那么 i 有一个关于 a 的回文中心对称的对应位置 j，由于 j 已经求解过，所以信息可以直接利用。

manacher 算法

图解



时间复杂度分析

每执行一次内层的 while 循环，mx 的值都会增加，而它最多增加到字符串的长度。所以 while 最多执行 $O(n)$ 次。总的时间复杂度为 $O(n)$。

目录

扩展 KMP（Z 函数）

Description

扩展 KMP 解决这样的问题：给定字符串 s，对 s 的每个后缀求出它与 s 的最长公共前缀。

算法思路

这个算法的思路与 manacher 有些类似。也是记录一个最靠右的匹配位置，然后求解新的位置时想办法利用之前已经求解过的信息。

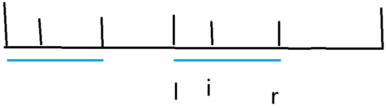
扩展 KMP（Z 函数）

代码实现

```cpp
string s;
cin >> s;
vector<int> z(s.size(), 0);
int l = 0, r = 0;
//z[i]表示s[i]与s[0]的最长公共前缀
//l,r表示s[l, r]是与前缀匹配的r最大的子串
for(int i = 1; i < s.size(); i++){
    if(i > r){
        l = r = i;
        while(r < s.size() && s[r] == s[r - l]) r++;
        z[i] = r - l;
        r--;
    }else{
        int k = i - l;
        if(z[k] < r - i + 1){
            z[i] = z[k];
        }else{
            l = i;
            while(r < s.size() && s[r] == s[r - l]) r++;
            z[i] = r - l;
            r--;
        }
    }
}
```

扩展 KMP（Z 函数）

图解



时间复杂度分析

同样的，每执行一次 while 循环则 r 加一。这决定了 while 循环执行的次数为 $O(n)$ 级别。

目录

Thank you! Any Question?