字符串哈希

定义

设 str 是一个字符串， $f(str)$ 是对 str 的一个映射函数，其输入是一个字符串，输出是一个整数，那么我们就称我们的 f 是 Hash 函数

适用范围

我们都知道，给定两个字符串 s_1, s_2 ，判断它们是否**相同**所需的时间复杂度是 $O(|s|)$ 的，但如果我们选择合适的 Hash 函数，将其分别映射成两个**整数**，这样我们就可以 $O(1)$ 地判断两个字符串是否相同

在编程领域内，我们输入的字符串 str 一般没有过多限制，而输出的函数值 $f(str)$ 我们一般限制在 32 或 64 位整型范围之内（因为这样方便 $O(1)$ 比较和代码书写）

哈希函数的性质

我们假设某个哈希函数的值域范围为 $[0, 10^{18}]$ ，定义域为长度 $|s| \le 2 \times 10^5$ 的，小写英文字母的字符串，显然，这样的字符串有 $26^{2\times 10^5}$ 个，个数显然远大于我们的值域范围

因此，我们得到了哈希函数的两个性质：

- 在哈希函数值不一样时，两个字符串一定不一样

  正确性显然
- 在哈希函数值一样时，两个字符串**可能**一样

  例如我们选取 $f(str) = \sum s_i$ ，很显然， $f(\texttt{ab}) = f(\texttt{ba})$ ，但实际上的字符串并不相同

  在 $str_1 \ne str_2 \land f(str_1) = f(str_2)$ 时，我们就称发生了**哈希冲突**

哈希函数的选取

从上例可以看出，哈希函数的选取是非常关键的，一个好的哈希函数能够帮助我们显著地降低**哈希冲突**的概率

在实际应用时，我们一般采用**多项式哈希**的方法：

$$f(str) \equiv \sum_{i=1}^{len} s_i \times B^{l-i} \mod M$$

式子看起来很复杂，但实际上比想象得简单，以 $str = \texttt{abcd}, B = 233$ 为例：

$$f(str) = a \times 233^3 + b \times 233^2 + c \times 233 + d$$

因此，我们能很简单地写出计算某个字符串 str 的哈希函数的代码：

```cpp
int Hash(const stirng& str) {
    ll res = 0;
    for (auto ch: str) {
        res *= b; res += ch;
        res %= mod;
    }
    return res;
}
```

其中，b 和 M 我们一般选取**质数**，特别注意的是，M, b 要大于我们的 $\max(ch)$，若小于的话，考虑如下情况：

$$\begin{aligned}
&str_1 = \mathtt{y0z}, str_2 = \mathtt{xyz}, B = \mathtt{x} \\
&f(str_1) = y \times B^2 + z \\
&f(str_2) = x \times B^2 + y \times B + z = (x+1) \times B^2 + z = y \times B^2 + z \\
&f(str_1) = f(str_2)
\end{aligned}$$

这样产生哈希冲突的概率是很高的

哈希冲突率

假设我们的值域范围是 M ，并假设我们的哈希函数足够好，所有的字符串的哈希值都是**均匀分布**在我们的值域范围内的，随机选取 n 个不同的字符串，那么未出现碰撞的概率是：

$$P = \prod_{i=0}^{n-1} \frac{M-i}{M}$$

上述式子和生日悖论类似，第 i 项是是第 i 个字符串不与前面 $i-1$ 个字符串发生碰撞的概率

若 $M = 10^9 + 7, n = 10^6$ ，可以算出 $P \approx 1 \times 10^{-128}$ ，也就是说，大概率会发生冲突

那么如何避免情况发生呢?

我们可以同时采用两套不同的 M_1, M_2 ，这样如果 str_1 和 str_2 在 M_1 模数下发生冲突，那在 M_2 模数下仍然发生冲突的概率是 $\frac{1}{M_2}$ ，可以看出，这个概率是非常小的，我们称之为**双哈希**

```
pair<int, int> Hash(const stirng& str) {
    ll x1 = 0, x2 = 0;
    for (auto ch: str) {
        x1 *= b; x1 += ch;
        x2 *= b; x2 += ch;
        x1 %= M1; x2 %= M2;
    }
    return {x1, x2};
}
```

质数表

1e2	1e6	1e9	1e18
129	1000003	1000000007	1000000000000000003
233	1000033	1000000009	1000000000000000079
257	1000159	1000000103	1000000000000000201
331	1000409	1000000271	1000000000000000283

子串的哈希

单次计算一个字符串的哈希值复杂度是 $O(|s|)$，与暴力匹配没有区别，如果需要多次询问一个字符串的子串的哈希值，每次重新计算效率非常低下

令 $res[i]$ 为字符串 s 某个**前缀**（$s[1\ldots i]$）的哈希值，也就可以按以下方法得出：

```
void Hash(const stirng& str, vector<int>& res) {
    ll tmp = 0;
    for (int i = 0; i < str.length(); i++) {
        tmp *= B; tmp += str[i];
        tmp %= mod; res[i] = tmp;
    }
}
```

显然，$res[i]=\sum_{j=1}^{i}s[j]\times B^{i-j}$，写成多项式形式就是：

$$res[i]=s[1]\times B^{i-1}+s[2]\times B^{i-2}+\cdots+s[i-1]\times B+s[i]$$

和前缀和类似，如果我们想要某一个子串 $s[l\cdots r]$ 的哈希值，那么我们可以用 $res[r]-res[l-1]$ 得到

但是这样会有问题

$$\begin{aligned}
res[l-1] &= s[1]\times B^{l-2}+s[2]\times B^{l-3}+\cdots+s[l-2]\times B+s[l-1]\\
res[r] &= s[1]\times B^{r-1}+s[2]\times B^{r-2}+\cdots+s[r-1]\times B+s[r]\\
res[r]-res[l-1] &= s[1]\times(B^{r-1}-B^{l-2})+s[2]\times(B^{r-2}-B^{l-3})\cdots\\
&\quad+s[l]\times B^{r-l}+s[l+1]\times B^{r-l-1}\cdots+s[r-1]\times B+s[r]
\end{aligned}$$

注意到对于项 $s[1],s[2]\cdots s[l-1]$ 而言，其系数比值均为**固定**的 B^{r-l+1}，因此，我们可以用 $res[r]-B^{r-l+1}res[l-1]$ 得到我们子串 $s[l\cdots r]$ 的哈希值

双哈希模板

```
struct Hash{
    const static int M1 = 1572869, M2 = 3145769, B = 129;
    int x1, x2;
    Hash(int x1 = 0, int x2 = 0): x1(x1), x2(x2) {}
    Hash add(char ch){
        return {(x1 * B + ch) % M1, (x2 * B + ch) % M2};
    }
    bool operator==(const Hash& oth) const{
        return (x1 == oth.x1 && x2 == oth.x2);
    }
    bool operator!=(const Hash& oth) const{
        return !(x1 == oth.x1 && x2 == oth.x2);
    }
    Hash operator-(const Hash& oth) const{
        return {(x1 - oth.x1 + M1) % M1, (x2 - oth.x2 + M2) % M2};
    }
    Hash operator*(int y) const{
        return {x1 * y % M1, x2 * y % M2};
    }
```

```
    Hash operator*(const Hash& oth) const{
        return Hash((ll)x1 * oth.x1 % M1, (ll)x2 * oth.x2 % M2);
    }
};
```

Bonus: anti-hash