字符串哈希

定义

我们定义一个把字符串映射到整数的函数 f，这个 f 称为是 Hash 函数。

我们希望这个函数 f 可以方便地帮我们判断两个字符串是否相等。

Hash 的思想

Hash 的核心思想在于，将输入映射到一个值域较小、可以方便比较的范围。

> **Warning**
>
> 这里的「值域较小」在不同情况下意义不同。
>
> 在 哈希表 中，值域需要小到能够接受线性的空间与时间复杂度。
>
> 在字符串哈希中，值域需要小到能够快速比较（10^9、10^{18} 都是可以快速比较的）。
>
> 同时，为了降低哈希冲突率，值域也不能太小。

性质

具体来说，哈希函数最重要的性质可以概括为下面两条：

1. 在 Hash 函数值不一样的时候，两个字符串一定不一样；
2. 在 Hash 函数值一样的时候，两个字符串不一定一样（但有大概率一样，且我们当然希望它们总是一样的）。

   我们将 Hash 函数值一样但原字符串不一样的现象称为哈希碰撞。

解释

我们需要关注的是什么？

时间复杂度和 Hash 的准确率。

通常我们采用的是多项式 Hash 的方法，对于一个长度为 l 的字符串 s 来说，我们可以这样定义多项式 Hash 函数：$f(s) = \sum_{i=1}^{l} s[i] \times b^{l-i} \pmod M$。例如，对于字符串 xyz，其哈希函数值为 $xb^2 + yb + z$。

特别要说明的是，也有很多人使用的是另一种 Hash 函数的定义，即 $f(s) = \sum_{i=1}^{l} s[i] \times b^{i-1} \pmod M$，这种定义下，同样的字符串 xyz 的哈希值就变为了 $x + yb + zb^2$ 了。

显然，上面这两种哈希函数的定义函数都是可行的，但二者在之后会讲到的计算子串哈希值时所用的计算式是不同的，因此千万注意 **不要弄混了这两种不同的 Hash 方式**。

由于前者的 Hash 定义计算更简便、使用人数更多、且可以类比为一个 b 进制数来帮助理解，所以本文下面所将要讨论的都是使用 $f(s) = \sum_{i=1}^{l} s[i] \times b^{l-i} \pmod M$ 来定义的 Hash 函数。

还有，有时为了方便和扩大模数，我们在 C++ 中我们会使用 `unsigned long long` 来定义 Hash 函数的结果。由于 C++ 的特性，我们相当于把模数 M 定为 2^{64}，也是一个不错的选择。

准确率会在后面讨论。

Hash 的错误率分析

Hash 冲突

Hash 冲突是指两个不同的字符串映射到相同的 Hash 值。

我们设 Hash 的取值空间（所有可能出现的字符串的数量）为 d，计算次数（要计算的字符串数量）为 n。

则 Hash 冲突的概率为：

$$p(n,d) = 1 - \frac{d!}{d^n\,(d-n)!} \approx 1 - \exp\left(-\frac{n(n-1)}{2d}\right)$$

证明

当 Hash 中每个值生成概率相同时，Hash 不冲突的概率为：

$$\overline{p}(n,d)=1\cdot\left(1-\frac{1}{d}\right)\cdot\left(1-\frac{2}{d}\right)\cdots\left(1-\frac{n-1}{d}\right)$$

化简得到：

$$\begin{aligned}\overline{p}(n,d)&=\frac{d}{d}\cdot\frac{d-1}{d}\cdot\frac{d-2}{d}\cdots\frac{d-n+1}{d}\\&=\frac{d\cdot(d-1)\cdot(d-2)\cdots(d-n+1)}{d^n}\\&=\frac{d!}{d^n\,(d-n)!}\end{aligned}$$

则 Hash 冲突的概率为：

$$p(n,d)=1-\frac{d!}{d^n\,(d-n)!}$$

这个公式还是太复杂了，我们进一步化简。

根据泰勒公式：

$$\exp(x)=\sum_{k=0}^{\infty}\frac{x^k}{k!}=1+x+\frac{x^2}{2}+\frac{x^3}{6}+\frac{x^4}{24}+\cdots$$

当 x 为一个极小值时，$\exp(x)$ 趋近于 $1+x$。

将它带入 Hash 不冲突的原始公式：

$$\overline{p}(n,d)\approx 1\cdot\exp\left(-\frac{1}{d}\right)\cdot\exp\left(-\frac{2}{d}\right)\cdots\exp\left(-\frac{n-1}{d}\right)$$

化简：

$$\begin{aligned}\overline{p}(n,d)&\approx\exp\left(-\frac{1}{d}-\frac{2}{d}-\cdots-\frac{n-1}{d}\right)\\&=\exp\left(-\frac{n(n-1)}{2d}\right)\end{aligned}$$

则 Hash 冲突的概率为：

$$p(n,d)\approx 1-\exp\left(-\frac{n(n-1)}{2d}\right)$$

卡大模数 Hash

注意到这个公式：

$$p(n,d)\approx 1-\exp\left(-\frac{n(n-1)}{2d}\right)$$

为了卡掉 Hash，我们要满足一下条件：

1. d 要大于模数。
2. $1-p(d,n)$ 要尽量小。

举个例子：

若字符集为 **大小写字母和数字**，模数为 10^9+7 时：

$\log_{62} 10^9+7 \approx 6$

$p(10^6, 62^6) \approx 0.9$

所以对于这个范围，我们随机生成 10^6 个长度为 6 的字符串，它们 Hash 值相同的概率高达 90% 。

卡自然溢出 Hash

这种 Hash 由于模数太大，用上面的方法卡不了，所以我们需要另一种方法。

首先，这种 Hash 是形如 $f(s)=\sum_{i=1}^{l} s[i] \times b^{l-i}$，我们根据 b 来分类讨论。

b 为偶数

此时 $f(s)=s_1 \cdot b^l + s_2 \cdot b^{l-1} + \cdots + s_l \cdot b \pmod M$，其中 M 为 2^{64}。

容易发现若 $l \geq 64$，$s_i \cdot b^l \equiv 0 \pmod M$。

所以我们只要构造形如：

`aaa...a`

`baa...a`

且长度大于 64 的字符串就能冲突。

b 为奇数

定义 $!s_i$ 为把 s_i 中所有字符反转。

例：

$s_i = abaab$

$!s_i = babba$

即把 `a` 变成 `b`，把 `b` 变成 `a`。

再定义 $hash_i$ 为 s_i 的 Hash 值，$!hash_i$ 为 $!s_i$ 的 Hash 值。

不断构造 $s_i = s_{i-1} + !s_{i-1}$。

s_{12} 和 $!s_{12}$ 就是我们要的两个字符串。

推导

首先，有：

$$\begin{aligned} hash_i &= hash_{i-1} \cdot base^{2^{i-2}} + !hash_{i-1} \\ !hash_i &= !hash_{i-1} \cdot base^{2^{i-2}} + hash_{i-1} \end{aligned}$$

尝试相减：

$$\begin{aligned} & hash_i - !hash_i \\ = \ & hash_{i-1} \cdot base^{2^{i-2}} + !hash_{i-1} - (!hash_{i-1} \cdot base^{2^{i-2}} + hash_{i-1}) \\ = \ & (hash_{i-1} - !hash_{i-1}) \cdot (base^{2^{i-2}} - 1) \end{aligned}$$

发现出现了 2^i，但是原式太复杂，尝试换元：

设：

$$\begin{aligned} f_i &= hash_i - !hash_i \\ g_i &= base^{2^{i-2}} - 1 \end{aligned}$$

根据原式得：

$$\begin{aligned} f_i &= f_{i-1} \cdot g_i \\ &= f_1 \cdot g_1 \cdot g_2 \cdots g_{i-1} \end{aligned}$$

因为 $base^{2^{i-2}}$ 一定是奇数，所以 g_i 一定是偶数。

所以：

$$2^{i-1} | f_i$$

但这样太大了，$i - 1 \geq 64$ 才能卡掉，继续化简：

$$g_i = base^{2^{i-2}} - 1 = (base^{2^{i-3}} - 1) \cdot (base^{2^{i-3}} + 1)$$

即 g_i 为 $g_{i-1} \cdot c \ (c \equiv 0 \pmod 2)$ 的形式。

所以 $2|s_1$，$4|s_2$...，即

$$\begin{aligned} & 2^i && | g_i \\ & 2^1 \cdot 2^2 \cdot 2^3 \cdots 2^{i-1} && | f_i \\ & 2^{i(i-1)/2} && | f_i \end{aligned}$$

即当 $i = 12$ 时就可以使 $2^{64} | hash_i - !hash_i$ 达到要求。

例题

例题：BZOJ 3097 Hash Killer I

给定一个用 **自然溢出** 实现的 Hash，要求构造一个字符串来卡掉它。

例题：BZOJ 3097 Hash Killer II

给定一个用 **大模数** 实现的 Hash，要求构造一个字符串来卡掉它。

例题：洛谷 U461211 字符串 Hash（数据加强）

给定 n 个字符串，判断不同的字符串有多少个。

Hash 的改进

多值 Hash

看了上面这么多的卡法，当然也有解决办法。

多值 Hash，就是有多个 Hash 函数，每个 Hash 函数的模数不一样，这样就能解决 Hash 冲突的问题。

判断时只要有其中一个的 Hash 值不同，就认为两个字符串不同，若 Hash 值都相同，则认为两个字符串相同。

一般来说，双值 Hash 就够用了。

多次询问子串哈希

单次计算一个字符串的哈希值复杂度是 $O(n)$，其中 n 为串长，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。

一般采取的方法是对整个字符串先预处理出每个前缀的哈希值，将哈希值看成一个 b 进制的数对 M 取模的结果，这样的话每次就能快速求出子串的哈希了：

令 $f_i(s)$ 表示 $f(s[1..i])$，即原串长度为 i 的前缀的哈希值，那么按照定义有 $f_i(s) = s[1] \cdot b^{i-1} + s[2] \cdot b^{i-2} + \dots + s[i-1] \cdot b + s[i]$

现在，我们想要用类似前缀和的方式快速求出 $f(s[l..r])$，按照定义有字符串 $s[l..r]$ 的哈希值为 $f(s[l..r]) = s[l] \cdot b^{r-l} + s[l+1] \cdot b^{r-l-1} + \dots + s[r-1] \cdot b + s[r]$

对比观察上述两个式子，我们发现 $f(s[l..r]) = f_r(s) - f_{l-1}(s) \times b^{r-l+1}$ 成立（可以手动代入验证一下），因此我们用这个式子就可以快速得到子串的哈希值。其中 b^{r-l+1} 可以 $O(n)$ 的预处理出来

然后 $O(1)$ 的回答每次询问（当然也可以快速幂 $O(\log n)$ 的回答每次询问）。

实现

模数 Hash：

注：效率较低，实际使用中不推荐。

C++

```cpp
using std::string;

constexpr int M = 1e9 + 7;
constexpr int B = 233;

using ll = long long;

int get_hash(const string& s) {
  int res = 0;
  for (int i = 0; i < s.size(); ++i) {
    res = ((ll)res * B + s[i]) % M;
  }
  return res;
}

bool cmp(const string& s, const string& t) {
  return get_hash(s) == get_hash(t);
}
```

Python

```python
M = int(1e9 + 7)
B = 233


def get_hash(s):
    res = 0
    for char in s:
        res = (res * B + ord(char)) % M
    return res


def cmp(s, t):
    return get_hash(s) == get_hash(t)
```

双值 Hash：

C++

```cpp
using ull = unsigned long long;
ull base = 131;
ull mod1 = 212370440130137957, mod2 = 1e9 + 7;

ull get_hash1(std::string s) {
  int len = s.size();
  ull ans = 0;
  for (int i = 0; i < len; i++) ans = (ans * base + (ull)s[i]) % mod1;
  return ans;
}

ull get_hash2(std::string s) {
  int len = s.size();
  ull ans = 0;
  for (int i = 0; i < len; i++) ans = (ans * base + (ull)s[i]) % mod2;
  return ans;
}

bool cmp(const std::string s, const std::string t) {
  bool f1 = get_hash1(s) != get_hash1(t);
  bool f2 = get_hash2(s) != get_hash2(t);
  return f1 || f2;
}
```

Python

```python
def get_hash1(s: str) -> int:
    base = 131
    mod1 = 212370440130137957
    ans = 0
    for char in s:
        ans = (ans * base + ord(char)) % mod1
    return ans


def get_hash2(s: str) -> int:
    base = 131
    mod2 = 1000000007
    ans = 0
    for char in s:
        ans = (ans * base + ord(char)) % mod2
    return ans


def cmp(s: str, t: str) -> bool:
    f1 = get_hash1(s) != get_hash1(t)
    f2 = get_hash2(s) != get_hash2(t)
    return f1 or f2
```

Hash 的应用

字符串匹配

求出模式串的哈希值后，求出文本串每个长度为模式串长度的子串的哈希值，分别与模式串的哈希值比较即可。

允许 k 次失配的字符串匹配

问题：给定长为 n 的源串 s，以及长度为 m 的模式串 p，要求查找源串中有多少子串与模式串配。s' 与 s 匹配，当且仅当 s' 与 s 长度相同，且最多有 k 个位置字符不同。其中 $1 \le n, m \le 10^6$，$0 \le k \le 5$。

这道题无法使用 KMP 解决，但是可以通过哈希 + 二分来解决。

枚举所有可能匹配的子串，假设现在枚举的子串为 s'，通过哈希 + 二分可以快速找到 s' 与 p 第一个不同的位置。之后将 s' 与 p 在这个失配位置及之前的部分删除掉，继续查找下一个失配位置。这样的过程最多发生 k 次。

总的时间复杂度为 $O(m + kn\log_2 m)$。

最长回文子串

二分答案，判断是否可行时枚举回文中心（对称轴），哈希判断两侧是否相等。需要分别预处理正着和倒着的哈希值。时间复杂度 $O(n \log n)$。

这个问题可以使用 manacher 算法 在 $O(n)$ 的时间内解决。

通过哈希同样可以 $O(n)$ 解决这个问题，具体方法就是记 R_i 表示以 i 作为结尾的最长回文的长度，那么答案就是 $\max_{i=1}^{n} R_i$。考虑到 $R_i \le R_{i-1} + 2$，因此我们只需要暴力从 $R_{i-1} + 2$ 开始递减，直到找到第一个回文即可。记变量 z 表示当前枚举的 R_i，初始时为 0，则 z 在每次 i 增大的时候都会增大 2，之后每次暴力循环都会减少 1，故暴力循环最多发生 $2n$ 次，总的时间复杂度为 $O(n)$。

最长公共子字符串

问题：给定 m 个总长不超过 n 的非空字符串，查找所有字符串的最长公共子字符串，如果有多个，任意输出其中一个。其中 $1 \le m, n \le 10^6$。

很显然如果存在长度为 k 的最长公共子字符串，那么 $k - 1$ 的公共子字符串也必定存在。因此我们可以二分最长公共子字符串的长度。假设现在的长度为 k，`check(k)` 的逻辑为我们将所有所有字符串的长度为 k 的子串分别进行哈希，将哈希值放入 n 个哈希表中存储。之后求交集即可。

时间复杂度为 $O(m + n \log n)$。

确定字符串中不同子字符串的数量

问题：给定长为 n 的字符串，仅由小写英文字母组成，查找该字符串中不同子串的数量。

为了解决这个问题，我们遍历了所有长度为 $l = 1, \cdots, n$ 的子串。对于每个长度为 l，我们将其 Hash 值乘以相同的 b 的幂次方，并存入一个数组中。数组中不同元素的数量等于字符串中长度不同的子串的数量，并此数字将添加到最终答案中。

为了方便起见，我们将使用 $h[i]$ 作为 Hash 的前缀字符，并定义 $h[0] = 0$。

参考代码

```cpp
int count_unique_substrings(string const& s) {
  int n = s.size();

  constexpr static int b = 31;
  constexpr static int m = 1e9 + 9;
  vector<long long> b_pow(n);
  b_pow[0] = 1;
  for (int i = 1; i < n; i++) b_pow[i] = (b_pow[i - 1] * b) % m;

  vector<long long> h(n + 1, 0);
  for (int i = 0; i < n; i++)
    h[i + 1] = (h[i] + (s[i] - 'a' + 1) * b_pow[i]) % m;

  int cnt = 0;
  for (int l = 1; l <= n; l++) {
    set<long long> hs;
    for (int i = 0; i <= n - l; i++) {
      long long cur_h = (h[i + l] + m - h[i]) % m;
      cur_h = (cur_h * b_pow[n - i - 1]) % m;
      hs.insert(cur_h);
    }
    cnt += hs.size();
  }
  return cnt;
}
```

例题

CF1200E Compress Words

给你若干个字符串，答案串初始为空。第 i 步将第 i 个字符串加到答案串的后面，但是尽量地去掉重复部分（即去掉一个最长的、是原答案串的后缀、也是第 i 个串的前缀的字符串），求最后得到的字符串。

字符串个数不超过 10^5，总长不超过 10^6。

题解

每次需要求最长的、是原答案串的后缀、也是第 i 个串的前缀的字符串。枚举这个串的长度，哈希比较即可。

当然，这道题也可以使用 KMP 算法 解决。

参考代码

```cpp
#include <cassert>
#include <cstring>
#include <iostream>
#include <vector>
using namespace std;

constexpr int L = 1e6 + 5;
constexpr int HASH_CNT = 2;

int hashBase[HASH_CNT] = {29, 31};
int hashMod[HASH_CNT] = {int(1e9 + 9), 998244353};

struct StringWithHash {
  char s[L];
  int ls;
  int hsh[HASH_CNT][L];
  int pwMod[HASH_CNT][L];

  void init() {  // 初始化
    ls = 0;
    for (int i = 0; i < HASH_CNT; ++i) {
      hsh[i][0] = 0;
      pwMod[i][0] = 1;
    }
  }

  StringWithHash() { init(); }

  void extend(char c) {
    s[++ls] = c;                          // 记录字符数和每一个
字符
    for (int i = 0; i < HASH_CNT; ++i) {  // 双哈希的预处理
      pwMod[i][ls] =
          1ll * pwMod[i][ls - 1] * hashBase[i] % hashMod[i];
// 得到b^ls
      hsh[i][ls] = (1ll * hsh[i][ls - 1] * hashBase[i] + c) %
hashMod[i];
    }
  }

  vector<int> getHash(int l, int r) {  // 得到哈希值
    vector<int> res(HASH_CNT, 0);
    for (int i = 0; i < HASH_CNT; ++i) {
      int t =
          (hsh[i][r] - 1ll * hsh[i][l - 1] * pwMod[i][r - l +
1]) % hashMod[i];
      t = (t + hashMod[i]) % hashMod[i];
      res[i] = t;
    }
```

```cpp
      return res;
    }
};

bool equal(const vector<int> &h1, const vector<int> &h2) {
  assert(h1.size() == h2.size());
  for (unsigned i = 0; i < h1.size(); i++)
    if (h1[i] != h2[i]) return false;
  return true;
}

int n;
StringWithHash s, t;
char str[L];

void work() {
  int len = strlen(str);  // 取字符串长度
  t.init();
  for (int j = 0; j < len; ++j) t.extend(str[j]);
  int d = 0;
  for (int j = min(len, s.ls); j >= 1; --j) {
    if (equal(t.getHash(1, j), s.getHash(s.ls - j + 1, s.ls))) {  // 比较哈希值
      d = j;
      break;
    }
  }
  for (int j = d; j < len; ++j) s.extend(str[j]);  // 更新答案数组
}

int main() {
  cin.tie(nullptr)->sync_with_stdio(false);
  cin >> n;
  for (int i = 1; i <= n; ++i) {
    cin >> str;
    work();
  }
  cout << s.s + 1 << '\n';
  return 0;
}
```

ShuYuMo2003, iamtwz, ksyx, mgt, taodaling, Yanjun-Zhao, algoriiiiithm, c-forrest, Chrogeek, GldHkkowo, Haohu Shen, HeRaNO, ImpleLee, kenlig, Marcythm, Menci, Molmin, runnableAir, ScrapW, shawlleyw, sshwy, szdytom, tfia, wangchong, wendajiang, xglight, xiangmy21, zyouxam