

基础组-字符串哈希和字典树-讲义

1 字符串哈希的实现

字符串哈希的实现有很多，这里展示我所熟悉的几个，不一定是最好的实现。

1.1 单个字符串的哈希

```
1 int str_hash(const std::string &s, int base, int mod) {
2     // 计算字符串哈希值
3     int res = 0;
4     for (char c : s) {
5         res = (1LL * res * base % mod + c) % mod;
6     }
7     return res;
8 }
```

这样求得整个字符串的哈希只能将整个字符串映射到一个整数。如果涉及子串的判断，我们需要使用前缀和的思想。

1.2 前缀和

鉴于之前没有讲过前缀和的知识，我们先看一下前缀和是什么。

可以参考：

- [OI-wiki:prefix-sum](#)
- [【深讲1.例1】求区间和](#)

前缀和的一大应用即是求区间和。在字符串哈希的计算中使用到了前缀和算法，这里简要介绍其思想：

- Question: 对于一个给定数列 A ，如何在 $O(1)$ 时间内求得 $\sum_{i=L}^R A_i$ ？

令 $S_i = \sum_{j=1}^i a_j$ ，则我们可以利用递推的思想 $O(N)$ 预处理出数组 S ，即 $S_i = S_{i-1} + A_i$ 。那么显然答案即为 $S_R - S_{L-1}$ ，这样我们就可以 $O(1)$ 静态查询区间和。

对于其他的一些运算，比如乘法 \times ，只需要满足有逆元和结合律，就可以使用这种思想求出区间运算的结果。

C++ 标准库中实现了前缀和函数 `std::partial_sum`，定义于头文件 `<numeric>` 中。

1.3 自然溢出哈希

相对简单的一种哈希写法，甚至没有必要封装，缺点是有概率被卡（正确率比双模数低）。

```
1 using ULL = unsigned long long;
2
3 struct strHashImpl2 { // 自然溢出哈希
4     // 使用 ULL 即是对 2**64 取模
5     ULL h[N], rh[N]; // 正向哈希值(前缀和), 反向哈希值(后缀和)
6     ULL p[N]; // p[i] 表示 base 的 i 次方, 也就是每一个字符的权值
7     int base; // 基数
8     int n; // 字符串长度
9
10    strHashImpl2(int base) : base(base) {
11        // 初始化: 计算 base 的 i 次方
```

```

12     p[0] = 1;
13     for (int i = 1; i < N; ++ i) {
14         p[i] = p[i - 1] * base;
15     }
16 }
17
18 void init(const std::string &s) { // 下标从 0 开始
19     // 初始化: 计算正向哈希值
20     n = s.size();
21     h[1] = s[0];
22     for (int i = 0; i < n; ++ i) {
23         h[i + 1] = h[i] * base + s[i];
24     }
25
26     // 初始化: 计算反向哈希值
27     rh[n + 1] = 0;
28     for (int i = n; i > 0; -- i) {
29         rh[i] = rh[i + 1] * base + s[i - 1];
30     }
31 }
32
33 ULL get(int l, int r) { // 获取 [l, r] 的哈希值
34     return h[r + 1] - p[r + 1] * h[l];
35 }
36
37 ULL rget(int l, int r) { // 获取 [l, r] 的反向哈希值
38     return rh[l] - p[r + 1] * rh[r + 1];
39 }
40
41 bool palindromic(int l, int r) { // 判断 [l, r] 是否为回文串
42     return get(l, r) == rget(l, r);
43 }
44
45 bool equal(int l1, int r1, int l2, int r2) { // 判断 [l1, r1] 和 [l2, r2] 是否相等
46     return get(l1, r1) == get(l2, r2);
47 }
48 };

```

1.4 单模数哈希【模数在 10^9 级别】

双哈希在大多数题目下都可以忽略错误率。

```

1  using LL = long long;
2  using PII = std::pair<int, int>;
3
4  constexpr int N = 1e5 + 10;
5  constexpr int mod1 = 1e9 + 7, mod2 = 1e9 + 9;
6  constexpr int base1 = 131, base2 = 233;
7
8  struct strHashImpl { // 单模数哈希
9      // h: 正向哈希值(前缀和), rh: 反向哈希值(后缀和)
10     // rh 用于判断回文串, 如果 h[l, r] == rh[l, r], 则 [l, r] 是回文串
11     // 如果不需要判断回文串, 可以不用 rh
12     int h[N], rh[N];
13     int p[N]; // p[i] 表示 base 的 i 次方, 也就是每一个字符的权值
14     int base, mod; // base: 基数, mod: 模数

```

```

15     int n; // 字符串长度
16
17     strHashImpl(int base, int mod) : base(base), mod(mod) {
18         // 初始化: 计算 base 的 i 次方
19         p[0] = 1;
20         for (int i = 1; i < N; ++ i) {
21             p[i] = 1LL * p[i - 1] * base % mod;
22         }
23     }
24
25     strHashImpl(PII arg) : strHashImpl(arg.first, arg.second) {} // 用于初始化多哈希
26
27     void init(const std::string &s) { // 下标从 0 开始
28         // 初始化: 计算正向哈希值
29         n = s.size();
30         h[1] = s[0];
31         for (int i = 0; i < n; ++ i) {
32             h[i] = (1LL * h[i - 1] * base % mod + s[i - 1]) % mod;
33         }
34
35         // 初始化: 计算反向哈希值
36         rh[n + 1] = 0;
37         for (int i = n; i; -- i) {
38             rh[i] = (1LL * rh[i + 1] * base % mod + s[i - 1]) % mod;
39         }
40     }
41
42     int get(int l, int r) { // 获取 [l, r] 的哈希值
43         return (h[r] - 1LL * h[l - 1] * p[r - l + 1] % mod + mod) % mod;
44     }
45
46     int rget(int l, int r) { // 获取 [l, r] 的反向哈希值
47         return (rh[l] - 1LL * rh[r + 1] * p[r - l + 1] % mod + mod) % mod;
48     }
49
50     bool palindromic(int l, int r) { // 判断 [l, r] 是否为回文串
51         return get(l, r) == rget(l, r);
52     }
53
54     bool equal(int l1, int r1, int l2, int r2) { // 判断 [l1, r1] 和 [l2, r2] 是否相等
55         return get(l1, r1) == get(l2, r2);
56     }
57 };

```

写两个单哈希分别判断就是双哈希，可以不看这里泛型的封装，一般最多只会用到双哈希。

```

1 // 实现一下泛型的 HashChecker
2 // 方便写多哈希(虽然最多只会用到两个哈希)
3 // 同时使用几个单哈希就是多哈希
4 template <PII ...Args> // Args: {base, mod}
5 struct Checker {
6     std::vector<strHashImpl> h;
7
8     Checker() : h({Args...}) {}
9
10    void init(const std::string &s) {

```

```

11     for (auto &hi : h) {
12         hi.init(s);
13     }
14 }
15
16 bool palindromic(int l, int r) { // 检查 [l, r] 是否为回文串
17     for (auto &hi : h) {
18         if (!hi.palindromic(l, r)) {
19             return false;
20         }
21     }
22     return true;
23 }
24
25 bool equal(int l1, int r1, int l2, int r2) { // 检查 [l1, r1] 和 [l2, r2] 是否相等
26     for (auto &hi : h) {
27         if (hi.get(l1, r1) != hi.get(l2, r2)) {
28             return false;
29         }
30     }
31     return true;
32 }
33 };

```

1.5 单模数哈希【模数在 10^{18} 级别】

如果模数在 10^{18} 次方级别，就可以不需要双哈希了，此时正确率已经足够高。

```

1  using i128 = __int128; // 当然如果模数够大的话，可以直接单模数哈希
2
3  struct strHashImpl3 { // 单模数哈希
4      LL h[N]; // 实现和之前一样，只是用了更大的模数
5      LL p[N];
6      LL base, mod;
7      int n;
8
9      strHashImpl3(LL base, LL mod) : base(base), mod(mod) {
10         p[0] = 1;
11         for (int i = 1; i < N; ++ i) {
12             p[i] = (i128)p[i - 1] * base % mod;
13         }
14     }
15
16     strHashImpl3() : strHashImpl3(base_, mod_) {}
17
18     void init(const std::string &s) {
19         n = s.size();
20         h[1] = s[0];
21         for (int i = 0; i < n; ++ i) {
22             h[i] = (i128)h[i - 1] * base % mod;
23             h[i] = ((i128)h[i] + s[i - 1]) % mod;
24         }
25     }
26
27     LL get(int l, int r) { // 获取 [l, r] 的哈希值
28         return (h[r] - (i128)h[l - 1] * p[r - l + 1] % mod + mod) % mod;

```

```

29     }
30
31     bool equal(int l1, int r1, int l2, int r2) { // 判断 [l1, r1] 和 [l2, r2] 是否相等
32         return get(l1, r1) == get(l2, r2);
33     }
34 };

```

当然我们需要先找到这么一个模数。这时我们朴素的判断素数的方法就行不通了，我们需要更高效的做法。

1.6 * Miller Rabin 素性检验

Miller Rabin 素性检验可以比较快的判断一个数是否是素数。

想了解的可以参考：

- [OI-Wiki:Miller Rabin](#)
- [0x11分钟学会 Miller Rabin 质数判定算法](#)

```

1  using i128 = __int128; // 当然如果模数够大的话，可以直接单模数哈希
2
3  LL fpow(LL a, LL b, LL mod) { // 快速幂 a^b % mod
4      LL res = 1;
5      while (b) {
6          if (b & 1) res = (i128)res * a % mod;
7          a = (i128)a * a % mod;
8          b >>= 1;
9      }
10     return res;
11 }
12
13 bool Miller_Rabin(LL x) { // Miller-Rabin 素数测试
14     // 时间复杂度 O(klogx)
15     // 有 2^(-k) 的概率判断错误
16     if (x == 2) return true;
17     if (x < 2 || !(x & 1)) return false;
18     LL u = x - 1, t = 0;
19     while (!(u & 1)) u >>= 1, ++ t;
20     for (LL a : {2, 325, 9375, 28178, 450775, 9780504, 1795265022}) {
21         LL v = fpow(a, u, x);
22         if (v == 1 || v == x - 1) continue;
23         for (int i = 1; i < t; ++ i) {
24             v = (i128)v * v % x;
25             if (v == x - 1) break;
26             if (v == 1) return false;
27         }
28         if (v != x - 1) return false;
29     }
30     return true;
31 }

```

所以我们可以较为快速的找到 $1e18$ 数量级的素数。

```

1  /**
2   * 打表可得>1e18的前10个素数为:
3   * 1000000000000000003,
4   * 1000000000000000009,
5   * 1000000000000000031,
6   * 1000000000000000079,
7   * 1000000000000000177,
8   * 1000000000000000183,
9   * 1000000000000000201,
10  * 1000000000000000283,
11  * 1000000000000000381,
12  * 1000000000000000387
13  */

```

2 字典树的实现

```

1  #include <bits/stdc++.h>
2
3  constexpr int N = 5e5 + 10; // 字典树节点数 (字符串总长度)
4  constexpr int M = 30; // 字符集大小
5  // 对于小写字母, 字符集大小为 26
6  // 对于二进制整数, 字符集大小为 2
7  // 字符集的定义是比较广泛的, 不要局限于传统意义上的字符
8
9  struct TrieTree {
10     // son[p][u] 表示节点 p 的第 u 个儿子
11     // 对应的字符是字符集中的第 u 个字符
12     // son[p][u] = 0 表示节点 p 的第 u 个儿子不存在
13     // 也就是说, 此时是字符串的末尾
14     // cnt[p] 表示以节点 p 结尾的字符串的数量
15     // idx 表示节点编号
16     // 我们使用 0 号节点作为根节点
17     int son[N][M], cnt[N], idx;
18
19     void insert(char str[]) { // 插入字符串
20         int p = 0;
21         for (int i = 0; str[i]; i++) {
22             int u = str[i] - 'a';
23             if (!son[p][u]) son[p][u] = ++ idx; // 如果儿子不存在, 创建新节点
24             p = son[p][u]; // 移动到下一个节点
25         }
26         cnt[p] ++; // 以节点 p 结尾的字符串数量加一
27     }
28
29     int query(char str[]) { // 查询字符串出现次数
30         int p = 0;
31         for (int i = 0; str[i]; i++) {
32             int u = str[i] - 'a';
33             if (!son[p][u]) return 0; // 如果没有完整匹配, 说明字符串不存在
34             p = son[p][u];
35         }
36         return cnt[p]; // 返回以节点 p 结尾的字符串数量
37     }
38 };

```

3 习题集提示

- T1 狗头人图书馆：考虑求出哈希值后去重
- T2 LESSON 5, 这是最完美的黄金回旋!：考虑枚举回文串中心，二分回文串半径
- T3 于是他错误的点名开始了：字典树模板
- T4 The XOR Largest Pair：参考 PPT
- T5 我是否在哪里见过你？你的名字是!：字符串匹配问题，考虑求出 T 的哈希值和 S 的哈希前缀和，枚举匹配的起点， $O(1)$ 判断