基础组-字符串哈希和字典树-题解

1　狗头人图书馆

我们考虑求出每一个字符串的哈希值，排序之后去重即可。

这道题显然有其他做法，任意做法均可。放在这里只是作为哈希的一个简单应用。

```cpp
using LL = long long;
using PII = pair<int, int>;

constexpr int mod1 = 1e9 + 7; // 在这道题中 我们使用双模数去减少冲突概率
constexpr int mod2 = 1e9 + 9;
constexpr int base1 = 233;
constexpr int base2 = 131;

int Hash(string &s, int base, int mod) {
    // ... 请自行实现哈希的部分
    return HASH_OF_S;
}

int main() {
    int n;
    cin >> n;
    vector<string> a(n);
    // 用到了 C++ 中的 range_based_for
    // 感兴趣的可以了解:
    // https://zh.cppreference.com/w/cpp/language/range-for
    for (auto &s : a) cin >> s;
    vector<PII> vec; // 双模数当然需要用一个 pair 存啦
    for (auto &s : a) {
        // emplace_back 可以自动调用构造函数
        // 感兴趣可以了解一下:
        // https://zh.cppreference.com/w/cpp/container/vector/emplace_back
        vec.emplace_back(
            Hash(s, base1, mod1),
            Hash(s, base2, mod2)
        );
    }
    // 在排序这节中已经讲述了如何求得非重复元素的个数
    sort(vec.begin(), vec.end());
    vec.erase(unique(vec.begin(), vec.end()), vec.end());
    cout << vec.size() << '\n';
}
```

2　LESSON 5，这是最完美的黄金回旋！

最长回文子串问题。感兴趣的可以学习 Manacher 算法。

2.1 二分做法

对于一个回文中心来说，显然存在单调的性质：如果半径为 x 的回文串存在，那么半径为 $x-1$ 的回文串也存在。因此我们可以二分答案，然后判断是否存在半径为 x 的回文串（当然这里对于整个遍历整个字符串半径为 x 的子串亦可）。需要注意的是，如果回文串长度为奇数，那么中心是一个字符，如果回文串长度为偶数，那么中心就不存在了。

```
typedef long long LL;
const int N = 1000000 + 10;
const int base1 = 131;
const int base2 = 233;
const int mod1 = 1e9 + 7;
const int mod2 = 1e9 + 9;
char s[N];

// 需要注意的是在 C/C++ 中负数取模的结果为负数
// 我们不希望见到这种情况，所以我们让结果变为正数
// 可以参见 https://www.runoob.com/w3cnote/remainder-and-the-modulo.html
// 严格意义上 C/C++ 中的 % 表示的是取余而非取模
int momo(int x, int mod) { return (x % mod + mod) % mod; }
void chkmax(int &a, const int &b) { if (a < b) a = b; }

struct strhash {
// ... 省略哈希的实现
} h1(mod1, base1), h2(mod2, base2);

bool check_odd(int R, int n, int pos) {
    // 此时圆心刚好位于回文中心字符
    // pos 刚好表示字符位置
    // R 表示半径 n 传入字符串的长度
    int l = pos - R;
    int r = pos + R;
    if (0 < l && l <= r && r <= n) {
        return h1.check(l, r) && h2.check(l, r);
    }
    return false;
}

bool check_even(int R, int n, int pos) {
    // 此时圆心夹在中心字符之间
    // pos 表示圆心位于[pos,pos+1]字符之间
    int l = pos - R + 1;
    int r = pos + R;
    if (0 < l && l < r && r <= n) {
        return h1.check(l, r) && h2.check(l, r);
    }
    return false;
}

int solve() {
    int n;
    scanf("%d%s", &n, s + 1);
    h1.init(s, n);
    h2.init(s, n);
    int res = 1;
    // 最长奇数回文串
```

```
    for (int i = 1; i <= n; ++ i) {
        // 二分回文半径
        // 此处的回文半径定义为回文串长度除以2下取整
        int l = 0, r = std::min(i - 1, n - i) + 1;
        while (l < r) {
            int mid = l + r + 1 >> 1;
            if (check_odd(mid, n, i)) l = mid;
            else r = mid - 1;
        }
        chkmax(res, 2 * l + 1);
    }
    // 最长偶数回文串
    for (int i = 1; i < n; ++ i) {
        int l = 1, r = std::min(i - 1, n - i) + 1;
        while (l < r) {
            int mid = l + r + 1 >> 1;
            if (check_even(mid, n, i)) l = mid;
            else r = mid - 1;
        }
        chkmax(res, 2 * l);
    }
    return res;
}
```

2.2 $O(N)$ 做法

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 $\max_{i=1}^{n} R_i$。注意到 $R_i \le R_{i-1}+2$（可以思考一下这是为什么）。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。假设每次开始遍历的区间为 $[l,i]$，显然每一个位置最多只会被 l 扫描两次，因此时间复杂度为 $O(n)$。

```
int solve() {
    int n;
    scanf("%d", &n);
    scanf("%s", s + 1);
    h1.init(s, n); // 表示我们实现哈希的结构体实例
    h2.init(s, n);
    R[1] = 1; // 表示以 i 作为结尾的最长回文的长度
    R[2] = s[1] == s[2] ? 2 : 1;
    for (int i = 3; i <= n; ++ i) {
        R[i] = 1;
        // 每一次我们只需要暴力从 R[i-1]+2 开始判断即可
        // 考虑到每一次循环的起点都是非递减的
        // 从而时间复杂度为 O(N)
        for (int j = std::min(R[i - 1] + 2, i); j; -- j) {
            // j 表示遍历可能的长度
            // i 表示回文串的结尾
            // 从而计算出回文串的范围
            // 注意这里没有中心和半径的考量，从而无需对长度奇偶性进行讨论
            int l = i - j + 1;
            int r = i;
            if (h1.check(l, r) && h2.check(l, r)) { // 判断回文串
                R[i] = j;
                break;
            }
        }
    }
```

```
    int res = 1;
    for (int i = 1; i <= n; ++ i) {
        chkmax(res, R[i]);
    }
    return res;
}
```

3 于是他错误的点名开始了

字典树裸题，只需要对于是否存在，是否查找过讨论即可。如果查找过，我们打上一个标记即可。

```
const int M = 5e5 + 10;
int son[M][30], cnt[M], idx;
char str[55];

void insert(char str[]) {
    int p = 0;
    for (int i = 0; str[i]; i++) {
        int u = str[i] - 'a';
        if (!son[p][u]) son[p][u] = ++ idx;
        p = son[p][u];
    }
    cnt[p] ++;
}

int query(char str[]) {
    int p = 0;
    for (int i = 0; str[i]; i++) {
        int u = str[i] - 'a';
        if (!son[p][u]) return 0; // 如果没有节点可走即为没有出现的情况
        p = son[p][u];
    }
    int res = cnt[p];
    if (res == 0) return 0; // 请注意如果 cnt 为 0 也是没有出现的情况
    cnt[p] = -1; // 如果 cnt 非 0 那么我们在一次查询过后打上标记即可
    // 也就是说
    // res=0  => 找不到
    // res>0  => 第一次找到
    // res=-1 => 多次查找
    return res;
}
```

4 The XOR Largest Pair

我们可以通过：`x >> i & 1` 求出一个十进制整数对应位置上的二进制数位，这表示 x 第 $i, 0 \le i \le 31$ 位上的数码。当然如果是 `long long` 或者其他类型，也是同样道理，只是位数有所不同。

```
constexpr int N = 1e5 + 5, M = 35;
constexpr int NM = N * M;

struct trie {
    int son[NM][2], idx = 0;

    void insert(int x) {
        int p = 0;
        for (int i = 30; i >= 0; -- i) {
```

```
            int t = (x >> i) & 1; // t 就表示 x 的当前位的数码
            if (!son[p][t]) son[p][t] = ++ idx;
            p = son[p][t];
        }
    }

    int query(int x) {
        int p = 0, res = 0;
        for (int i = 30; i >= 0; -- i) {
            int t = (x >> i) & 1;
            // t 非 0 即为 1
            // 而 t ^ 1 为对于最后一位取反
            // 从而 t=0 ^1 => 1; t=1 ^1 = 0
            if (son[p][t ^ 1]) { // 我们从高到低位贪心查找，找与当前位相反的数字
                res |= 1 << i; // 如果存在那么就对当前位答案有 1 的贡献
                // 位运算可以参考：
                // https://www.runoob.com/w3cnote/bit-operation.html
                p = son[p][t ^ 1];
            } else {
                p = son[p][t];
            }
        }
        return res;
    }
} tr;

void solve() {
    int n;
    std::cin >> n;
    std::vector<int> a(n);
    for (int &x : a) std::cin >> x;

    int res = 0;
    for (int i = 0; i < n; ++ i) {
        // 由于异或的交换律
        // 所以我们只需要每次插入一个数 再查找一次即可
        tr.insert(a[i]);
        res = std::max(res, tr.query(a[i]));
    }

    std::cout << res << '\n';
}
```

5 我是否在哪里见过你？你的名字是！

字符串匹配裸题。感兴趣的同学可以了解前缀函数和 KMP 算法。我们只需要求出 s 和 t 的哈希值，然后在 s 中枚举 t 的起点，然后用 $O(1)$ 的时间复杂度来判断是否匹配即可。

```
using LL = long long;

constexpr int N = 1e6 + 10;
constexpr int mod1 = 1e9 + 7; // 在这道题中 我们使用双模数去减少冲突概率
constexpr int mod2 = 1e9 + 9;
constexpr int base1 = 233;
constexpr int base2 = 131;

```

```cpp
struct strhash {
    // ... 省略哈希的实现
} h0(base1, mod1), h1(base2, mod2);

void solve() {
    int n, m;
    std::cin >> n >> m;

    std::string S, T;
    std::cin >> S >> T;

    h0.init(S, n);
    h1.init(S, n);

    std::vector<int> thash(2, 0); // 计算 t 对应的哈希值
    for (int i = 0; i < 2; ++ i) {
        for (int j = 0; j < m; ++ j) {
            thash[i] = (LL)thash[i] * base % mod[i];
            thash[i] = ((LL)thash[i] + T[j]) % mod[i];
        }
    }
    // lambda 表达式表示检测 s[l,r] 和 t 是否相等
    auto check = [&](int l, int r) -> bool {
        return h0.get(l, r) == thash[0] && h1.get(l, r) == thash[1];
    };
    // 暴力遍历每一个匹配的起点即可
    std::vector<int> ans;
    for (int i = 0; i + m - 1 < n; ++ i)
        if (check(i + 1, i + m))
            ans.push_back(i + 1);
    // 请注意答案有可能是 0 也就是 T 没有在 S 中出现过
    std::cout << ans.size() << '\n';
    for (int i = 0; i < ans.size(); ++ i) {
        std::cout << ans[i] << " \n"[i == ans.size() - 1];
    }
}
```

CONGRATULATIONS
ON YOUR COMPLETION
OF THE SUMMER SCHOOL
BASIC GROUP CONTENT LEARNING!
以上，恭喜你完成暑期学校基础组的内容！
感谢大家的聆听！THX 4 listening!