0x30 字符串

0x31 KMP

- 时间复杂度 $O(n+m)$

```
struct KMP
{
    int n;
    string p;
    vector<int> ne;
    KMP(string &s) : n(s.size() - 1), p(s), ne(n + 1)
    {
        for (int i = 2, j = 0; i <= n; i ++ )
        {
            while (j && p[i] != p[j + 1]) j = ne[j];
            if (p[i] == p[j + 1]) j ++ ;
            ne[i] = j;
        }
    }

    int boarder(int x) { return ne[x]; }
    /* 求所有在s串中的start_pos，如果first_only设置为true，则只返回第一个位置 */
    vector<int> match(string &s, bool first_only = false)
    {
        vector<int> start_pos;
        for (int i = 1, j = 0; i < s.size(); i ++ )
        {
            while (j && s[i] != p[j + 1]) j = ne[j];
            if (s[i] == p[j + 1]) j ++ ;
            if (j == n)
            {
                start_pos.push_back(i - j + 1);
                if (first_only) return start_pos;
                j = ne[j];
            }
        }
        return start_pos;
    }
    /* 循环周期 形如 acaca 中 ac 是一个合法周期 */
    vector<int> periodic()
    {
        vector<int> ret;
        int now = n;
        while (now)
        {
            now = ne[now];
            ret.push_back(n - now);
        }
        return ret;
```

```
    }
    /* 循环节 形如 acac 中ac、acac是循环节，aca不是 */
    vector<int> periodic_loop()
    {
        vector<int> ret;
        for (int x : periodic())
        {
            if (n % x == 0) ret.push_back(x);
        }
        return ret;
    }
    int min_periodic_loop() { return periodic_loop()[0]; }
};
```

0x32 Z函数（扩展KMP）

定义

对于个长度为 n 的字符串 S。定义函数 $z[i]$ 表示 S 和 $S[i,n-1]$（即以 $S[i]$ 开头的后缀）的最长公共前缀（LCP）的长度。z 被称为 S 的 Z 函数（扩展 KMP）。**特殊地**，$z[0]=0$。

- 时间复杂度 $O(n)$

```
// z[i] = LCP(T[i,lent],T)
// extend[i] = LCP(S[i,lens],T)
struct Z
{
    int n;
    string p;
    vector<int> z;
    Z(string &s) : n(s.size() - 1), p(s), z(z_algorithm(p)) {}

    vector<int> z_algorithm(string &s)
    {
        vector<int> extend(s.size());
        extend[0] = 0;
        for (int i = 1, st = 0, ed = 0; i < s.size(); i ++ )
        {
            extend[i] = i <= ed ? min(z[i - st + 1], ed - i + 1) : 0;
            while (i + extend[i] < s.size() && extend[i] < n && s[i + extend[i]]
== p[extend[i] + 1])
            {
                extend[i] ++ ;
            }
            if (i + extend[i] - 1 >= ed && i != 1)
            {
                st = i;
                ed = i + extend[i] - 1;
            }
        }
        return extend;
```

```
    }
};
```

0x33 字符串哈希

哈希

- 时间复杂度 $O(n)$

```
using Hashv = unsigned long long;
const Hashv base = 233;

struct StringHash
{
    const int n;
    vector<Hashv> h1, h2, power;
    StringHash(string &s) : n(s.size() - 1), h1(n + 2), h2(n + 2), power(n + 2)
    {
        for (int i = 1; i <= n; i ++ ) h1[i] = h1[i - 1] * base + s[i];
        for (int i = n; i >= 1; i -- ) h2[i] = h2[i + 1] * base + s[i];
        power[0] = 1;
        for (int i = 1; i <= n; i ++ ) power[i] = power[i - 1] * base;
    }

    Hashv order(int l, int r)
    {
        return h1[r] - h1[l - 1] * power[r - l + 1];
    }
    Hashv reorder(int l, int r)
    {
        return h2[l] - h2[r + 1] * power[r - l + 1];
    }
};
```

双模数哈希

```
typedef unsigned long long ULL;

ULL Prime_Pool[] = {1998585857ul,23333333333ul};
ULL Seed_Pool[] = {911,146527,19260817,91815541};
ULL Mod_Pool[] = {29123,998244353,1000000009,4294967291ull};

constexpr int P1 = 1e9 + 7, P2 = 1e9 + 9;
struct Hashv
{
    int h1, h2;
    Hashv(int base1 = 0, int base2 = 0) : h1(base1), h2(base2) {}

    ULL val() { return 1ll * h1 * P2 + h2; }
```

```
    Hashv &operator*=(const Hashv &rhs)
    {
        h1 = (ULL)h1 * rhs.h1 % P1;
        h2 = (ULL)h2 * rhs.h2 % P2;
        return *this;
    }
    Hashv &operator+=(const Hashv &rhs)
    {
        h1 += rhs.h1; if (h1 >= P1) h1 -= P1;
        h2 += rhs.h2; if (h2 >= P2) h2 -= P2;
        return *this;
    }
    Hashv &operator-=(const Hashv &rhs)
    {
        h1 -= rhs.h1; if (h1 < 0) h1 += P1;
        h2 -= rhs.h2; if (h2 < 0) h2 += P2;
        return *this;
    }
    friend Hashv operator*(const Hashv &lhs, const Hashv &rhs)
    {
        Hashv res = lhs;
        res *= rhs;
        return res;
    }
    friend Hashv operator+(const Hashv &lhs, const Hashv &rhs)
    {
        Hashv res = lhs;
        res += rhs;
        return res;
    }
    friend Hashv operator-(const Hashv &lhs, const Hashv &rhs)
    {
        Hashv res = lhs;
        res -= rhs;
        return res;
    }
    friend bool operator==(const Hashv &lhs, const Hashv &rhs)
    {
        return lhs.h1 == rhs.h1 && lhs.h2 == rhs.h2;
    }
    friend bool operator<(const Hashv &lhs, const Hashv &rhs)
    {
        return lhs.h1 != rhs.h1 ? lhs.h1 < rhs.h1 : lhs.h2 < rhs.h2;
    }
};

const Hashv base(131, 233);
// using Hashv = unsigned long long;
// const Hashv base = 233;

struct StringHash
{
    const int n;
    vector<Hashv> h1, h2, power;
```

```
    StringHash(string &s) : n(s.size() - 1), h1(n + 2), h2(n + 2), power(n + 2)
    {
        for (int i = 1; i <= n; i ++ ) h1[i] = h1[i - 1] * base + s[i];
        for (int i = n; i >= 1; i -- ) h2[i] = h2[i + 1] * base + s[i];
        power[0] = Hashv(1, 1);
        // power[0] = 1;
        for (int i = 1; i <= n; i ++ ) power[i] = power[i - 1] * base;
    }

    Hashv order(int l, int r)
    {
        return h1[r] - h1[l - 1] * power[r - l + 1];
    }
    Hashv reorder(int l, int r)
    {
        return h2[l] - h2[r + 1] * power[r - l + 1];
    }
};
```