字符串

Ac自动机

```
struct AhoCorasick {
    static constexpr int ALPHABET = 26;
    struct Node {
        int len;
        int link;
        int top;
        int val;
        int d;
        std::array<int, ALPHABET> next;
        Node() : len{}, link{}, next{}, top{}, val {-1}, d{} {}
    };

    std::vector<Node> t;

    AhoCorasick() {
        init();
    }

    void init() {
        t.assign(2, Node());
        t[0].next.fill(1);
        t[0].len = -1;
    }

    int newNode() {
        t.emplace_back();
        return t.size() - 1;
    }

    int add(const std::vector<int> &a) {
        int p = 1;
        for (auto x : a) {
            if (t[p].next[x] == 0) {
                t[p].next[x] = newNode();
                t[t[p].next[x]].len = t[p].len + 1;
            }
            p = t[p].next[x];
        }
        apply (t[p].val);
        return p;
    }

    int add(const std::string &a, char offset = 'a') {
        std::vector<int> b(a.size());
        for (int i = 0; i < a.size(); i++) {
            b[i] = a[i] - offset;
        }
        return add(b);
    }
```

```
    void work() {
        std::queue<int> q;
        q.push(1);

        while (!q.empty()) {
            int x = q.front();
            q.pop();

            t[x].top = t[link(x)].val >= 0 ? link(x) : top(link(x));

            for (int i = 0; i < ALPHABET; i++) {
                if (t[x].next[i] == 0) {
                    t[x].next[i] = t[t[x].link].next[i];
                } else {
                    t[t[x].next[i]].link = t[t[x].link].next[i];
                    t[t[t[x].link].next[i]].d += 1;
                    q.push(t[x].next[i]);
                }
            }
        }
    }

    int next(int p, int x) {
        return t[p].next[x];
    }

    int next(int p, char c, char offset = 'a') {
        return next(p, c - 'a');
    }

    int link(int p) {
        return t[p].link;
    }

    int len(int p) {
        return t[p].len;
    }

    int& val(int p) {
        return t[p].val;
    }

    int top (int p) {
        return t[p].top;
    }

    int size() {
        return t.size();
    }

    int& d ( int p ) {
        return t[p].d;
    }

    void apply (auto& val) {
```

```
        val = 0 ;
    }
};
```

字符串哈希

```
std::mt19937 rng(std::chrono::steady_clock::now().time_since_epoch().count());

bool isprime(int n) {
    if (n <= 1) return false;
    for (int i = 2; i * i <= n; i++)
        if (n % i == 0)
            return false;
    return true;
}
int findPrime(int n) {
    while (!isprime(n))
        n++;
    return n;
}

template<int N>
struct StringHash {
    static array<int, N> mod;
    static array<int, N> base;
    vector<array<int, N>> p, h;
    StringHash() = default;
    StringHash(const string& s) {
        int n = s.size();
        p.resize(n);
        h.resize(n);
        fill(p[0].begin(), p[0].end(), 1);
        for (int i = 0; i < n; i++)
        for (int j = 0; j < N; j++) {
            p[i][j] = 1ll * (i == 0 ? 1ll : p[i - 1][j]) * base[j] % mod[j];
            h[i][j] = (1ll * (i == 0 ? 0ll : h[i - 1][j]) * base[j] + s[i]) %
mod[j];
        }
    }
    array<int, N> query(int l, int r) {
        assert(r >= l - 1);
        array<int, N> ans{};
        if (l > r) return {0, 0};
        for (int i = 0; i < N; i++) {
            ans[i] = (h[r][i] - 1ll * (l == 0 ? 0ll : h[l - 1][i]) * (r - l + 1
== 0 ? 1ll : p[r - l][i]) % mod[i] + mod[i]) % mod[i];
        }
        return ans;
    }
};

constexpr int HN = 2;
template<>
array<int, 2> StringHash<HN>::mod =
    {findPrime(rng() % 900000000 + 100000000),
```

```
    findPrime(rng() % 900000000 + 100000000)};
template<>
array<int, 2> StringHash<HN>::base {13331, 131};
using Hashing = StringHash<HN>;
```

后缀数组

```
using i64 = long long;
struct SuffixArray {
    int n;
    std::vector<int> sa, rk, lc;
    SuffixArray(const std::string &s) {
        n = s.length();
        sa.resize(n);
        lc.resize(n - 1);
        rk.resize(n);
        std::iota(sa.begin(), sa.end(), 0);
        std::sort(sa.begin(), sa.end(), [&](int a, int b) {return s[a] < s[b];});
        rk[sa[0]] = 0;
        for (int i = 1; i < n; ++i)
            rk[sa[i]] = rk[sa[i - 1]] + (s[sa[i]] != s[sa[i - 1]]);
        int k = 1;
        std::vector<int> tmp, cnt(n);
        tmp.reserve(n);
        while (rk[sa[n - 1]] < n - 1) {
            tmp.clear();
            for (int i = 0; i < k; ++i)
                tmp.push_back(n - k + i);
            for (auto i : sa)
                if (i >= k)
                    tmp.push_back(i - k);
            std::fill(cnt.begin(), cnt.end(), 0);
            for (int i = 0; i < n; ++i)
                ++cnt[rk[i]];
            for (int i = 1; i < n; ++i)
                cnt[i] += cnt[i - 1];
            for (int i = n - 1; i >= 0; --i)
                sa[--cnt[rk[tmp[i]]]] = tmp[i];
            std::swap(rk, tmp);
            rk[sa[0]] = 0;
            for (int i = 1; i < n; ++i)
                rk[sa[i]] = rk[sa[i - 1]] + (tmp[sa[i - 1]] < tmp[sa[i]] || sa[i
- 1] + k == n || tmp[sa[i - 1] + k] < tmp[sa[i] + k]);
            k *= 2;
        }
        for (int i = 0, j = 0; i < n; ++i) {
            if (rk[i] == 0) {
                j = 0;
            } else {
                for (j -= j > 0; i + j < n && sa[rk[i] - 1] + j < n && s[i + j]
== s[sa[rk[i] - 1] + j]; )
                    ++j;
                lc[rk[i] - 1] = j;
            }
        }
```

```
    }
};
```

KMP

```
struct KMP{
    int n;
    std::vector<int> pi;
    std::vector<vector<int>> aut;

    KMP(const std::string &s) {
        n = (int)s.length();
        prefix_function(s);
        compute_automaton(s);
    }

    void prefix_function(string s) {
        pi.resize(n);
        for (int i = 1; i < n; i++) {
            int j = pi[i - 1];
            while (j > 0 && s[i] != s[j]) j = pi[j - 1];
            if (s[i] == s[j]) j++;
            pi[i] = j;
        }
    }

    void compute_automaton(string s) {
        aut.resize(n, vector<int>(26));
        for (int i = 0; i < n; i++) {
            for (int c = 0; c < 26; c++) {
            if (i > 0 && 'a' + c != s[i])
                aut[i][c] = aut[pi[i - 1]][c];
            else
                aut[i][c] = i + ('a' + c == s[i]);
            }
        }
    }
};
```

Trie

```
constexpr int max_size = 262144000;
uint8_t buf[max_size];
uint8_t *head = buf;

using u32 = uint32_t;

template <class T>
struct u32_p {
    u32 x;
    u32_p(u32 x = 0) : x(x) {}
```

```
    T *operator->() {
        return (T *)(buf + x);
    }
    operator bool() {
        return x;
    }
    operator u32() {
        return x;
    }
    bool operator==(u32_p rhs) const {
        return x == rhs.x;
    }
    static u32_p __new() {
        // assert(x < max_size);
        return (head += sizeof(T)) - buf;
    }
};

constexpr int N = 2e5;

struct node;
using Trie = u32_p<node>;

struct node {
    array<Trie, 2> ch{};
    int x; int sum;
};
```

Manacher

```
std::vector<int> manacher(std::string s) {
    std::string t = "#";
    for (auto c : s) {
        t += c;
        t += '#';
    }
    int n = t.size();
    std::vector<int> r(n);
    for (int i = 0, j = 0; i < n; i++) {
        if (2 * j - i >= 0 && j + r[j] > i) {
            r[i] = std::min(r[2 * j - i], j + r[j] - i);
        }
        while (i - r[i] >= 0 && i + r[i] < n && t[i - r[i]] == t[i + r[i]]) {
            r[i] += 1;
        }
        if (i + r[i] > j + r[j]) {
            j = i;
        }
    }
    return r;
}
```

Z函数

```
std::vector<int> zFunction(std::string s) {
    int n = s.size();
    std::vector<int> z(n + 1);
    z[0] = n;
    for (int i = 1, j = 1; i < n; i++) {
        z[i] = std::max(0ll, std::min(j + z[j] - i, z[i - j]));
        while (i + z[i] < n && s[z[i]] == s[i + z[i]]) {
            z[i]++;
        }
        if (i + z[i] > j + z[j]) {
            j = i;
        }
    }
    return z;
}
```

PAM

```
struct PAM {
    static constexpr int ALPHABET_SIZE = 28;
    struct Node {
        int len;
        int link;
        int cnt;
        std::array<int, ALPHABET_SIZE> next;
        Node() : len{}, link{}, cnt{}, next{} {}
    };
    std::vector<Node> t;
    int suff;
    std::string s;
    PAM() { init(); }
    void init() {
        t.assign(2, Node());
        t[0].len = -1;
        suff = 1;
        s.clear();
    }
    int newNode() {
        t.emplace_back();
        return t.size() - 1;
    }

    bool add(char c, char offset = 'a') {
        int pos = s.size();
        s += c;
        int let = c - offset;
        int cur = suff, curlen = 0;

        while (true) {
            curlen = t[cur].len;
            if (pos - 1 - curlen >= 0 && s[pos - 1 - curlen] == s[pos])
                break;
```

```
            cur = t[cur].link;
        }
        if (t[cur].next[let]) {
            suff = t[cur].next[let];
            return false;
        }

        int num = newNode();
        suff = num;
        t[num].len = t[cur].len + 2;
        t[cur].next[let] = num;

        if (t[num].len == 1) {
            t[num].link = 1;
            t[num].cnt = 1;
            return true;
        }

        while (true) {
            cur = t[cur].link;
            curlen = t[cur].len;
            if (pos - 1 - curlen >= 0 && s[pos - 1 - curlen] == s[pos]) {
                t[num].link = t[cur].next[let];
                break;
            }
        }

        t[num].cnt = 1 + t[t[num].link].cnt;

        return true;
    }
};

PAM pam;
// 应用：
// 1：求s本质不同回文串个数：自动机状态数
// 2：求所有回文子串分别出现次数:插入的时候cnt[last]++，然后查询的时候倒推
cnt[fail[i]]+=cnt[i]
// 3：以第i个位置为结尾的回文串个数，cnt[i]=cnt[fail[i]]+1，边加边查cnt[last]
```

SAM

```
struct SAM {
    static constexpr int ALPHABET_SIZE = 26;
    struct Node {
        int len;
        int link;
        std::array<int, ALPHABET_SIZE> next;
        Node() : len{}, link{}, next{} {}
    };
    std::vector<Node> t;
    SAM() { init(); }
    void init() {
        t.assign(2, Node());
```

```
        t[0].next.fill(1);
        t[0].len = -1;
    }
    int newNode() {
        t.emplace_back();
        return t.size() - 1;
    }
    int extend(int p, int c) {
        if (t[p].next[c]) {
            int q = t[p].next[c];
            if (t[q].len == t[p].len + 1) {
                return q;
            }
            int r = newNode();
            t[r].len = t[p].len + 1;
            t[r].link = t[q].link;
            t[r].next = t[q].next;
            t[q].link = r;
            while (t[p].next[c] == q) {
                t[p].next[c] = r;
                p = t[p].link;
            }
            return r;
        }
        int cur = newNode();
        t[cur].len = t[p].len + 1;
        while (!t[p].next[c]) {
            t[p].next[c] = cur;
            p = t[p].link;
        }
        t[cur].link = extend(p, c);
        return cur;
    }
    // int extend(int p, char c, char offset = 'a') {
    //     return extend(p, c - offset);
    // }

    int next(int p, int x) { return t[p].next[x]; }

    // int next(int p, char c, char offset = 'a') { return next(p, c - 'a'); }

    int link(int p) { return t[p].link; }

    int len(int p) { return t[p].len; }

    int size() { return t.size(); }

    string lcs(const string& s, char offset = 'a') {
        int p = 1, l = 0;
        int pos = 0, len = 0;
        int cnt = 0;
        for (auto i : s) {
            while (p != 1 && (next(p, i - offset) == 0)) {
                p = link(p);
                l = t[p].len;
            }
```

```
            if (next(p, i - offset)) {
                p = next(p, i - offset);
                l++;
            }
            if (l > len) {
                len = l;
                pos = cnt;
            }
            cnt++;
        }
        return s.substr(pos - len + 1, len);
    };
};

// 应用：
// 1: 检查字符串是否出现
// 给一个文本串 T 和多个模式串 P ，我们要检查字符串 P 是否作为 T
// 的一个子串出现。 我们在
// O(T)
// 的时间内对文本串 T 构造后缀自动机。为了检查模式串 P  是否在 T
// 中出现，我们沿转移（边）从 t0
// 开始根据 P 的字符进行转移。如果在某个点无法转移下去，则模式串 P 不是 T
// 的一个子串。如果我们能够这样处理完整个字符串 P  ，那么模式串在 T  中出现过。
// 对于每个字符串 P  ，算法的时间复杂度为 O(P)
// 此外，这个算法还找到了模式串 P  在文本串中出现的最大前缀长度。

// 2: 出现次数
// 对于一个给定的文本串T ，有多组询问，每组询问给一个模式串P,
// 回答模式串 P 在字符串 T 中作为子串出现了多少次。
// 对文本串 T 构造后缀自动机。
// 接下来做预处理：对于自动机中的每个状态v ，预处理cnt_v
// 使之等于endpos(v) 集合的大小。事实上，对应同一状态 v 的所有子串在文本串 T
// 中的出现次数相同，这相当于集合 endpos 中的位置数。
// 然而我们不能明确的构造集合 endpos ，因此我们只考虑它们的大小cnt
// 为了计算这些值，我们进行以下操作。对于每个状态，
// 如果它不是通过复制创建的（且它不是初始状态t0)，
// 我们将它的 cnt 初始化为 1。然后我们按它们的长度len降序遍历所有状态，
// 并将当前的 cnt_v 的值加到后缀链接指向的状态上，即：
//  cnt_link(v) += cnt_v
// 最后回答询问只需要查找值cnt_t ,其中 t 为模式串对应的状态，
// 如果该模式串不存在答案就为 0。单次查询的时间复杂度为O(P),预处理复杂度O(|T|)

// 3: LCS
// 对S构造后缀自动机,处理T串
```

子序列自动机

```
auto get_nxt(string s) {
    int n = (int)s.size() - 1;
    vector<vector<int>> nxt(n + 2, vector<int>(26, n + 1));
    for (int i = n; i >= 0; i--) {
        for (int j = 0; j < 26; j++) {
            if (i == n)
```

```
                nxt[i][j] = n + 1;
            else
                nxt[i][j] = nxt[i + 1][j];
        }
        if (i != n)
            nxt[i][s[i + 1] - 'a'] = i + 1;
    }
    return nxt;
}

auto jump(string s, vector<vector<int>> &nxt) {
    int now = 0;
    for (int i = 0; i < s.size(); i++) {
        now = nxt[now][s[i] - 'a'];
    }
    return now;
}
```