串

子串与子序列

中文名称	常见英文名称	解释
子串	substring	连续的选择一段字符（可以全选、可以不选）组成的新字符串
子序列	subsequence	从左到右取出若干个字符（可以不取、可以全取、可以不连续）组成的新字符串

kmp

> 应用：
>
> 1. 在字符串中查找子串；
> 2. 最小周期：字符串长度-整个字符串的 border ；
> 3. 最小循环节：区别于周期，当字符串长度 $n \bmod (n - nxt[n]) = 0$ 时，等于最小周期，否则为 n 。

以最坏 $\mathcal{O}(N + M)$ 的时间计算 t 在 s 中出现的全部位置。

```cpp
std::vector<int> get_next(std::string& t) {
    std::vector<int> next(t.size());
    next[0] = -1;
    for (int i = 0, j = -1; i < (int)t.size();) {
        if (j == -1 || t[i] == t[j]) {
            ++i, ++j;
            next[i] = j;
        }
        else
            j = next[j];
    }
    return next;
}
```

```cpp
bool kmp(std::string& s, std::string& t) {
    if (t.length() > s.length())return false;
    auto next = get_next(t);

    for (int i = 0, j = 0; i < (int)s.size() && j < (int)t.size();) {
        if (j == -1 || s[i] == t[j]) {
            ++i, ++j;
        }
        else
            j = next[j];
        if (j == (int)t.size())return true;
    }
    return false;
}
```

zfunction

获取字符串 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，总复杂度 $\mathcal{O}(N)$。

```cpp
std::vector<int> z_function(std::string s) {
    int n = (int)s.length();
    std::vector<int> z(n);
    for (int i = 1, l = 0, r = 0; i < n; ++i) {
        if (i <= r && z[i - l] < r - i + 1) {
            z[i] = z[i - l];
        }
        else {
            z[i] = std::max(0, r - i + 1);
            while (i + z[i] < n && s[z[i]] == s[i + z[i]]) ++z[i];
        }
        if (i + z[i] - 1 > r) l = i, r = i + z[i] - 1;
    }
    return z;
}
```

最长公共子序列 LCS

求解两个串的最长公共子序列的长度。

小数据解

针对 10^3 以内的数据。

```cpp
const int N = 1e3 + 10;
char a[N], b[N];
int n, m, f[N][N];
void solve(){
    cin >> n >> m >> a + 1 >> b + 1;
    for (int i = 1; i <= n; i++)
        for (int j = 1; j <= m; j++){
            f[i][j] = max(f[i - 1][j], f[i][j - 1]);
            if (a[i] == b[j]) f[i][j] = max(f[i][j], f[i - 1][j - 1] + 1);
        }
    cout << f[n][m] << "\n";
}
int main(){
    solve();
    return 0;
}
```

大数据解

针对 10^5 以内的数据。

```cpp
const int INF = 0x7fffffff;
int n, a[maxn], b[maxn], f[maxn], p[maxn];
int main(){
    cin >> n;
```

```
    for (int i = 1; i <= n; i++){
        scanf("%d", &a[i]);
        p[a[i]] = i;  //将第二个序列中的元素映射到第一个中
    }
    for (int i = 1; i <= n; i++){
        scanf("%d", &b[i]);
        f[i] = INF;
    }
    int len = 0;
    f[0] = 0;
    for (int i = 1; i <= n; i++){
        if (p[b[i]] > f[len]) f[++len] = p[b[i]];
        else {
            int l = 0, r = len;
            while (l < r){
                int mid = (l + r) >> 1;
                if (f[mid] > p[b[i]]) r = mid;
                else l = mid + 1;
            }
            f[l] = min(f[l], p[b[i]]);
        }
    }
    cout << len << "\n";
    return 0;
}
```

字符串哈希

双哈希封装

随机质数列表：1111111121、1211111123、1311111119。

```
const int N = 1 << 21;
static const int mod1 = 1E9 + 7, base1 = 127;
static const int mod2 = 1E9 + 9, base2 = 131;
using U = Zmod<mod1>;
using V = Zmod<mod2>;
vector<U> val1;
vector<V> val2;
void init(int n = N) {
    val1.resize(n + 1), val2.resize(n + 2);
    val1[0] = 1, val2[0] = 1;
    for (int i = 1; i <= n; i++) {
        val1[i] = val1[i - 1] * base1;
        val2[i] = val2[i - 1] * base2;
    }
}
struct String {
    vector<U> hash1;
    vector<V> hash2;
    string s;

    String(string s_) : s(s_), hash1{1}, hash2{1} {
        for (auto it : s) {
            hash1.push_back(hash1.back() * base1 + it);
```

```
            hash2.push_back(hash2.back() * base2 + it);
        }
    }
    pair<U, V> get() { // 输出整串的哈希值
        return {hash1.back(), hash2.back()};
    }
    pair<U, V> substring(int l, int r) { // 输出子串的哈希值
        if (l > r) swap(l, r);
        U ans1 = hash1[r + 1] - hash1[l] * val1[r - l + 1];
        V ans2 = hash2[r + 1] - hash2[l] * val2[r - l + 1];
        return {ans1, ans2};
    }
    pair<U, V> modify(int idx, char x) { // 修改 idx 位为 x
        int n = s.size() - 1;
        U ans1 = hash1.back() + val1[n - idx] * (x - s[idx]);
        V ans2 = hash2.back() + val2[n - idx] * (x - s[idx]);
        return {ans1, ans2};
    }
};
```

前后缀去重

`sample please ease` 去重后得到 `samplease`。

```
string compress(vector<string> in) { // 前后缀压缩
    vector<U> hash1{1};
    vector<V> hash2{1};
    string ans = "#";
    for (auto s : in) {
        s = "#" + s;
        int st = 0;
        U chk1 = 0;
        V chk2 = 0;
        for (int j = 1; j < s.size() && j < ans.size(); j++) {
            chk1 = chk1 * base1 + s[j];
            chk2 = chk2 * base2 + s[j];
            if ((hash1.back() == hash1[ans.size() - 1 - j] * val1[j] +
chk1) &&
                (hash2.back() == hash2[ans.size() - 1 - j] * val2[j] +
chk2)) {
                st = j;
            }
        }
        for (int j = st + 1; j < s.size(); j++) {
            ans += s[j];
            hash1.push_back(hash1.back() * base1 + s[j]);
            hash2.push_back(hash2.back() * base2 + s[j]);
        }
    }
    return ans.substr(1);
}
```

马拉车

```
struct Manachar {
    std::vector<int> d1, d2;
    Manachar(std::string s) {
        int n = s.length();
        d1.assign(n, 0);
        d2.assign(n, 0);
        for (int i = 0, l = 0, r = -1;i < n;++i) {
            int k = (i > r) ? 1 : std::min(d1[l + r - i], r - i + 1);
            while (i + k < n && i - k >= 0 && s[i + k] == s[i - k])k++;
            d1[i] = k--;
            if (i + k > r) {
                r = i + k;
                l = i - k;
            }
        }
        for (int i = 0, l = 0, r = -1;i < n;++i) {
            int k = (i > r) ? 0 : std::min(d2[l + r - i + 1], r - i + 1);
            while (i + k < n && i - k - 1 >= 0 && s[i + k] == s[i - k -
1])k++;
            d2[i] = k--;
            if (i + k > r) {
                r = i + k;
                l = i - k - 1;
            }
        }
    }
    bool check(int l, int r) {
        if (r < l)return false;
        int len = r - l + 1;
        if (len % 2) {
            return d1[l + len / 2] * 2 - 1 < len;
        }
        else {
            return d2[l + len / 2] * 2 < len;
        }
    }
};
```

字典树 trie

基础封装

```
struct Trie {
    int ch[N][63], cnt[N], idx = 0;
    map<char, int> mp;
    void init() {
        LL id = 0;
        for (char c = 'a'; c <= 'z'; c++) mp[c] = ++id;
        for (char c = 'A'; c <= 'Z'; c++) mp[c] = ++id;
        for (char c = '0'; c <= '9'; c++) mp[c] = ++id;
    }
    void insert(string s) {
```

```
        int u = 0;
        for (int i = 0; i < s.size(); i++) {
            int v = mp[s[i]];
            if (!ch[u][v]) ch[u][v] = ++idx;
            u = ch[u][v];
            cnt[u]++;
        }
    }
    LL query(string s) {
        int u = 0;
        for (int i = 0; i < s.size(); i++) {
            int v = mp[s[i]];
            if (!ch[u][v]) return 0;
            u = ch[u][v];
        }
        return cnt[u];
    }
    void Clear() {
        for (int i = 0; i <= idx; i++) {
            cnt[i] = 0;
            for (int j = 0; j <= 62; j++) {
                ch[i][j] = 0;
            }
        }
        idx = 0;
    }
} trie;
```

01 字典树

```
struct Trie {
    int n, idx;
    vector<vector<int>> ch;
    Trie(int n) {
        this->n = n;
        idx = 0;
        ch.resize(30 * (n + 1), vector<int>(2));
    }
    void insert(int x) {
        int u = 0;
        for (int i = 30; ~i; i--) {
            int &v = ch[u][x >> i & 1];
            if (!v) v = ++idx;
            u = v;
        }
    }
    int query(int x) {
        int u = 0, res = 0;
        for (int i = 30; ~i; i--) {
            int v = x >> i & 1;
            if (ch[u][!v]) {
                res += (1 << i);
                u = ch[u][!v];
            } else {
                u = ch[u][v];
```

```
            }
        }
        return res;
    }
};
```

后缀数组 SA

以 $\mathcal{O}(N)$ 的复杂度求解。

```
struct SuffixArray {
    int n;
    vector<int> sa, rk, lc;
    SuffixArray(const string &s) {
        n = s.length();
        sa.resize(n);
        lc.resize(n - 1);
        rk.resize(n);
        iota(sa.begin(), sa.end(), 0);
        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;
        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);
                }
            }
            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];
            }
            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;
                continue;
            }
            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;
        }
    }
};
```

AC 自动机

定义 $|s_i|$ 是模板串的长度，$|S|$ 是文本串的长度，$|\Sigma|$ 是字符集的大小（常数，一般为 26），时间复杂度为 $\mathcal{O}(\sum |s_i| + |S|)$。

```
// Trie+Kmp，多模式串匹配
struct ACAutomaton {
    static constexpr int N = 1e6 + 10;
    int ch[N][26], fail[N], cntNodes;
    int cnt[N];
    ACAutomaton() {
        cntNodes = 1;
    }
    void insert(string s) {
        int u = 1;
        for (auto c : s) {
            int &v = ch[u][c - 'a'];
            if (!v) v = ++cntNodes;
            u = v;
        }
        cnt[u]++;
    }
    void build() {
        fill(ch[0], ch[0] + 26, 1);
        queue<int> q;
        q.push(1);
        while (!q.empty()) {
            int u = q.front();
            q.pop();
            for (int i = 0; i < 26; i++) {
                int &v = ch[u][i];
                if (!v)
                    v = ch[fail[u]][i];
                else {
                    fail[v] = ch[fail[u]][i];
                    q.push(v);
                }
```

```
            }
        }
    }
    LL query(string t) {
        LL ans = 0;
        int u = 1;
        for (auto c : t) {
            u = ch[u][c - 'a'];
            for (int v = u; v && ~cnt[v]; v = fail[v]) {
                ans += cnt[v];
                cnt[v] = -1;
            }
        }
        return ans;
    }
};
```

回文自动机 PAM（回文树）

> 应用：
>
> 1. 本质不同的回文串个数：$idx-2$；
> 2. 回文子串出现次数。

对于一个字符串 s，它的本质不同回文子串个数最多只有 $|s|$ 个，那么，构造 s 的回文树的时间复杂度是 $\mathcal{O}(|s|)$。

回文自动机的核心思想是，它以一种高度压缩的形式存储了给定字符串的**所有本质不同的回文子串**的信息。它的构造过程非常高效，可以在线性的时间复杂度（O(n)）内完成。

主要用途

1. **统计所有本质不同的回文子串数量**：
   - 这是回文自动机最基本的功能。其结构中的节点数量（除两个初始节点外）就直接对应着本质不同的回文子串个数。
2. **计算每个回文子串的出现次数**：
   - 可以在构建自动机的同时或之后，高效地计算出每个本质不同的回文子串在原字符串中出现了多少次。这对于解决需要统计回文串频率的问题至关重要。
3. **查找最长回文子串**：
   - 自动机中的每个节点都代表一个回文串，因此最长的回文子串对应着长度最长的那个节点，可以轻松获取。
4. **计算以每个字符结尾的最长回文子串**：
   - 在线构建回文自动机的过程中，可以实时知道当字符串增长到第 `i` 个字符时，以该字符结尾的最长回文子串是什么。
5. **解决双重回文串问题（palindromic factorization）**：
   - 例如，询问一个字符串可以被划分成多少个回文串的组合，或者找出最小的回文划分数量。回文自动机可以通过其 `fail` 指针链（指向当前回文串的最长回文后缀）来高效解决这类动态规划问题。
6. **不同回文子串的总长度**：
   - 遍历自动机的所有节点，将每个节点代表的回文串长度相加即可。

核心优势

- **高效性**：时空复杂度均为线性（O(n)，其中 n 为字符串长度）。
- **在线处理**：支持在线算法，即可以一个一个地添加字符并随时更新和查询回文子串的信息。
- **结构精巧**：它的 `fail` 指针构成了所谓的“回文树”结构，这棵树深刻地揭示了字符串中所有回文子串之间的后缀关系，是解决复杂回文问题的关键。

回文自动机是解决字符串中“回文子串”相关问题的终极利器。当题目涉及到统计、查找、或基于回文子串进行复杂计算时，它通常是最高效、最直接的解决方案。

```cpp
struct PalindromeAutomaton {
    constexpr static int N = 5e5 + 10;
    int tr[N][26], fail[N], len[N];
    int cntNodes, last;
    int dep[N]; //记录深度
    int cnt[N]; //记录出现次数
    std::string s;
    PalindromeAutomaton(std::string s) {
        memset(tr, 0, sizeof tr);
        memset(fail, 0, sizeof fail);
        memset(dep, 0, sizeof dep);
        memset(cnt, 0, sizeof cnt);
        len[0] = 0, fail[0] = 1;
        len[1] = -1, fail[1] = 0;
        cntNodes = 1;
        last = 0;
        this->s = s;
    }
    void insert(char c, int i) {
        int u = get_fail(last, i);
        if (!tr[u][c - 'a']) {
            int v = ++cntNodes;
            fail[v] = tr[get_fail(fail[u], i)][c - 'a'];
            tr[u][c - 'a'] = v;
            len[v] = len[u] + 2;
            dep[v] = dep[fail[v]] + 1;
        }
        last = tr[u][c - 'a'];
        cnt[last] += 1;
    }
    void countAll() {
        for (int i = cntNodes;i >= 0;i--)
            cnt[fail[i]] += cnt[i];
    }
    int get_fail(int u, int i) {
        while (i - len[u] - 1 <= -1 || s[i - len[u] - 1] != s[i]) {
            u = fail[u];
        }
        return u;
    }
};
```

后缀自动机 SAM

定义 $|\Sigma|$ 是字符集的大小，复杂度为 $\mathcal{O}(N \log |\Sigma|)$ 。

后缀自动机（Suffix Automaton, SAM）虽然也能用于匹配，但它的强大之处在于对**单个字符串**的**所有子串**进行深度分析。除去AC自动机也能做的简单子串匹配外，后缀自动机还能高效完成以下任务：

1. **统计不同子串的数量**：
    - 后缀自动机可以在线性时间内计算出一个字符串中本质不同的子串有多少个。这是AC自动机完全无法做到的。
2. **计算最长公共子串（LCS）**：
    - 通过构建两个或多个字符串的广义后缀自动机，可以高效地找到它们的最长公共子串。这虽然也涉及多字符串，但其解决问题的角度和AC自动机完全不同。
3. **查找字典序第k小子串**：
    - 后缀自动机的图结构天然支持按字典序遍历，因此可以高效地找出所有不同子串中，按字典序排序后的第k个。
4. **计算任意子串的出现次数**：
    - 对于给定的字符串S的任意一个子串P，后缀自动机可以快速计算出P在S中出现了多少次。而AC自动机只能计算“词典中”的串的出现次数。
5. **寻找最小循环移位**：
    - 这是一个经典应用，可以利用后缀自动机在线性时间内找到一个字符串的最小字典序循环移位。

如果把AC自动机看作是“**在一篇文章里找特定的几个关键词**”的专家，那么后缀自动机就是“**给你一篇文章，然后问关于这篇文章任何片段（子串）的任何刁钻问题**”的全能专家。它的应用深度和广度远超多模式匹配。

```cpp
// 有向无环图
struct SuffixAutomaton {
    static constexpr int N = 1e6;
    struct node {
        int len, link, nxt[26];
        int siz;
    } t[N << 1];
    int cntNodes;
    SuffixAutomaton() {
        cntNodes = 1;
        fill(t[0].nxt, t[0].nxt + 26, 1);
        t[0].len = -1;
    }
    int extend(int p, int c) {
        if (t[p].nxt[c]) {
            int q = t[p].nxt[c];
            if (t[q].len == t[p].len + 1) {
                return q;
            }
            int r = ++cntNodes;
            t[r].siz = 0;
            t[r].len = t[p].len + 1;
            t[r].link = t[q].link;
            copy(t[q].nxt, t[q].nxt + 26, t[r].nxt);
```

```
            t[q].link = r;
            while (t[p].nxt[c] == q) {
                t[p].nxt[c] = r;
                p = t[p].link;
            }
            return r;
        }
        int cur = ++cntNodes;
        t[cur].len = t[p].len + 1;
        t[cur].siz = 1;
        while (!t[p].nxt[c]) {
            t[p].nxt[c] = cur;
            p = t[p].link;
        }
        t[cur].link = extend(p, c);
        return cur;
    }
};
```

子序列自动机

对于给定的长度为 n 的主串 s，以 $\mathcal{O}(n)$ 的时间复杂度预处理、$\mathcal{O}(m+\log \text{size}{:}s)$ 的复杂度判定长度为 m 的询问串是否是主串的子序列。

好的，我们来谈谈子序列自动机（Subsequence Automaton）。

相比于后缀自动机（SAM）和回文自动机（PAM），子序列自动机在结构上要简单得多，但它同样是处理特定字符串问题的有效工具。它的主要应用领域集中在与**子序列**相关的匹配和统计问题上。

什么是子序列自动机?

对于一个给定的字符串 S（长度为 n），它的子序列自动机是一个能够识别 S 所有子序列的自动机。其构造非常直观和简单：

它通常被实现为一个二维数组 `next[i][c]`，表示在字符串的第 `i` 个位置之后（不包括 `i`），字符 `c` 第一次出现的位置。这个数组可以在 O(n×|Σ|) 的时间内预处理出来，其中 |Σ| 是字符集的大小（例如，对于小写字母是26）。

举例：

对于字符串 S = "banana"

next[0]['b'] 是 1 (第一个 'b' 的位置)

next[1]['n'] 是 3 (位置1 'a' 之后，下一个 'n' 在位置3)

next[3]['n'] 是 5 (位置3 'n' 之后，下一个 'n' 在位置5)

主要用途

子序列自动机的主要用途可以归结为以下几点：

1. **判断一个字符串是否为子序列（子序列匹配）**：
    - 这是最核心和最常见的用途。给定一个模式串 T，要判断它是否是主串 S 的子序列，只需利用预处理好的 `next` 数组进行贪心匹配。从位置0开始，依次为 T 的每个字符在 S 中寻找下一个最近的匹配位置。这个过程的效率极高，时间复杂度为 O(|T|)。

2. **解决“公共子序列”相关问题**：
    - 虽然寻找“最长公共子序列”（LCS）通常使用动态规划，但在某些特定场景下，子序列自动机可以提供不同的解题思路。
    - 例如，在多个字符串上构建各自的子序列自动机，然后通过在这些自动机上同步转移（类似DP），可以用来寻找多个字符串的“最短的公共超序列”（Shortest Common Supersequence）或解决其他相关的公共子序列变种问题。
3. **计算不同子序列的数量**：
    - 可以通过在子序列自动机上进行动态规划（DP）来计算一个字符串本质不同的子序列有多少个。DP状态通常定义为 `dp[i]` 表示从位置 `i` 开始的子序列个数。
4. **寻找字典序第k小子序列**：
    - 与计算数量类似，通过在自动机上进行DP，预先计算出从每个位置出发能产生多少不同的子序列，然后就可以按位确定第k小的子序列应该选择哪个字符作为开头，并跳转到相应的位置。

- **结构简单**：相比SAM和PAM，它的概念和实现都非常简单，就是一个`next`数组。
- **构建快速**：预处理速度很快，尤其适用于字符集较小的情况。
- **匹配高效**：对于子序列匹配问题，查询效率是线性的，与主串长度无关。

子序列自动机是专门用于高效处理字符串“子序列”相关问题的简单数据结构。当题目需要反复、快速地判断一个或多个字符串是否为某个主串的子序列，或者需要对子序列进行统计和计数时，它就是非常有用的工具。

自动离散化、自动类型匹配封装

```cpp
template<class T> struct SequenceAutomaton {
    vector<T> alls;
    vector<vector<int>> ver;

    SequenceAutomaton(auto in) {
        for (auto &i : in) {
            alls.push_back(i);
        }
        sort(alls.begin(), alls.end());
        alls.erase(unique(alls.begin(), alls.end()), alls.end());

        ver.resize(alls.size() + 1);
        for (int i = 0; i < in.size(); i++) {
            ver[get(in[i])].push_back(i + 1);
        }
    }
    bool count(T x) {
        return binary_search(alls.begin(), alls.end(), x);
    }
    int get(T x) {
        return lower_bound(alls.begin(), alls.end(), x) - alls.begin();
    }
    bool contains(auto in) {
        int at = 0;
        for (auto &i : in) {
            if (!count(i)) {
                return false;
```

```
            }

            auto j = get(i);
            auto it = lower_bound(ver[j].begin(), ver[j].end(), at + 1);
            if (it == ver[j].end()) {
                return false;
            }
            at = *it;
        }
        return true;
    }
};
```

朴素封装

原时间复杂度中的 $\text{size}:s$ 需要手动设置。类型需要手动设置。

```
struct SequenceAutomaton {
    vector<vector<int>> ver;

    SequenceAutomaton(vector<int> &in, int size) : ver(size + 1) {
        for (int i = 0; i < in.size(); i++) {
            ver[in[i]].push_back(i + 1);
        }
    }
    bool contains(vector<int> &in) {
        int at = 0;
        for (auto &i : in) {
            auto it = lower_bound(ver[i].begin(), ver[i].end(), at + 1);
            if (it == ver[i].end()) {
                return false;
            }
            at = *it;
        }
        return true;
    }
};
```

/END/