String

Created	@2025년 5월 19일 오후 10:02
Tags	Aho-Corasick KMP LCP Z

KMP
Z
LCP with Suffix Array
아호코라식
Trie

KMP

π[i] = j := str[0 ~ j] == str[i-j ~ i]

i.e. **i 번째 문자까지 봤을 때, "가장 긴 접두사이자 접미사"가 패턴 앞의 (j+1) 글자만큼 일치한다.**

→ 패턴과 문자열을 비교하는 중, 불일치가 발생하였을 때, **j = pi[j]** 로 돌아가면서 불필요한 비교연산 줄임

** j = π[j]인 이유? :

새로운 s[i]가 오기 전까지는 **[0 1 … j]** *[…]* **[(i-1-j) (i-j) … (i-1)]** 꼴이었음. (굵은 글씨 동일한 문자열)

s[i] ≠ s[j+1] 이기 때문에 기존의 j 길이 만큼의 접미사는 매칭 실패.

따라서 j 까지 보았을 때의 최대 매칭 길이인 π[j]로 j를 대체하여

[0 1 … j] *[…]* **[(i-1-j) (i-j) … (i-1)]** [i]

→ **[0 1 … π[j] … j]** *[…]* **[(i-1-j) (i-j) … (i-1-π[j]) … (i-1)]** [i]

→ **[0 1 … π[j]]** ***[(π[j]+1) … j]*** *[…]* ***[(i-1-j) … (i-2-π[j])]*** **[(i-1-π[j]) … (i-1)]** [i]

→ **[0 1 … π[j]]** *[…]* **[(i-1-π[j]) … (i-1)]** [i]

→ **[0 1 … j]** *[…]* **[(i-1-j) … (i-1)]** [i]

이 과정을 s[i] ≠ s[j+1] 인 동안 반복

만약 매칭을 유지하면서 s[i] == s[j+1] 에 도달하는 것이 성공했다면, 매칭 길이에 1 추가하여 저장.

여전히 매칭을 실패했다면 -1 (매칭 실패) 로 저장

pi를 구하는 것과 동일하게, KMP 에서도 비교 중 실패가 발생하면, 패턴의 접두사와 (현재까지 본) 문자열의 접미사+새롭게 추가된 문자 가 같아질 때까지 j=π[j] 과정 반복

BOJ 1786 - 찾기

BOJ 4354 - 문자열 제곱

BOJ 1305 - 광고

```
// Usage: vl pos = kmp(str, pattern);
// O(|s|+|p|)
// Note: return matched position (0-based)
// pi[i]=j := str[0...j]==str[(i-j)...i]

void calculate_pi(vl &pi, string &s) {
    pi[0]=-1;
    for (ll i=1,j=-1;i<s.size();i++) {
        while (j>=0 && s[i]!=s[j+1]) j=pi[j];
        if (s[i]==s[j+1]) pi[i]=++j;
        else pi[i]=-1;
    }
}

vl kmp(string& str, string& pattern) {
    vl pi(pattern.size()), ans;
    if (pattern.empty()) return ans;
    calculate_pi(pi, pattern);
    for (ll i=0,j=-1;i<str.size();i++) {
        while (j>=0 && str[i]!=pattern[j+1]) j=pi[j];
        if (str[i]==pattern[j+1]) {
            j++;
            if (j+1==pattern.size()) ans.push_back(i-j), j=pi[j];
        }
    }
    return ans;
}
```

Z

Z[i] = max{ j | 0≤j≤|str|−i && str[0 ... (j−1)] = str[i ... (i+j−1)]}

i.e. **i 에서 시작하는 부분문자열과 원래 문자열의 공통 접두사의 최대 길이**

[0 1 ... (r-l-1)] [...] **[l ... (r-1)]** [...]

현재 위와 같은 상태일 때, [l, r) 구간을 Z-box라 부르며, Z-box는 prefix와 일치한다.

i가 Z-box 바깥에 있으면, 처음부터 비교를 시작한다. Z-box 안에 있으면 불필요한 비교를 줄일 수 있다.

[0 1 ... (r-l-1)] *[...]* **[l ... i ... (r-1)]** *[...]*

→ **[0 ... (i-l) ... (r-l-1)]** *[...]* **[l ... i ... (r-1)]** *[...]*

→ **[0 ... i′ ... (r′-1)]** *[...]* **[l ... i ... (r-1)]** *[...]*

→ **[0 ... (Z[i′]-1)] [...] [i′ ... (i′+Z[i′]-1)] [... (r′-1)]** *[...]* **[l ... (l+Z[i′]-1)] [...] [i ... (i+Z[i′]-1)]**

[... (r-1)] *[...]*

→ **[0 ... (Z[i′]-1)]** *[...]* **[i ... (i+Z[i′]-1)]** *[...]*

Z[i']가 r-i 보다 크다면, 일단 이미 알려진 최대 매칭 길이(r-i) 만큼 부여하고, 그 이후에 대해서는 직접 비교.

Z[i']가 기존 Z-box 안에 위치한다면 값 그대로 사용.

BOJ 13506 - 카멜레온 부분 문자열

```
// Z[i] : maximum common prefix length of &s[0] and &s[i]
// O(|s|)
void get_z(string& s, vl& Z) {
    ll n = s.size();
    Z.resize(n);
    Z[0] = n;
    for (ll i=1,l=0,r=0; i<n; i++) {
        if (i<r) Z[i] = min(r-i, Z[i-l]);
        while (i+Z[i]<n && s[Z[i]]==s[i+Z[i]]) Z[i]++;
        if (i+Z[i]>r) l=i, r=i+Z[i];
    }
}
```

LCP with Suffix Array

SA[i] = j := i번째로 작은 접미사의 시작 인덱스가 j

i.e. 모든 접미사를 정렬했을 때, i번째 접미사는 문자열 j번째 에서 시작하는 접미사이다.

rank[i]=j := SA[j]=i

i.e. 문자열 i에서 시작하는 접미사는 j번째 접미사이다.

LCP[i]=j := SA[i] 접미사와 SA[i-1] 접미사가 앞에서부터 j개 만큼 동일하다.

i.e. 모든 접미사를 정렬했을 때, i번째 접미사와 그 이전(i-1번째) 접미사의 최장 공통 접두사 길이가 j이다.

** LCP 구하는 게 O(N)인 이유?

if(cnt) cnt--; 코드 한 줄 덕분에 선형 시간 유지 가능.

예를 들어 banana 에서 anana와 우선순위 전단계 접미사 ana의 비교를 통해 cnt==3을 얻어냈다고 하자.

순서 상, anana의 다음 접미사인 nana와 그 우선순위 전단계 접미사를 비교해야 하는데,

우리는 이미 ana에서 앞 글자 하나를 뺀 na가 접미사에 포함되어 있음을 알고 있음.

따라서 처음부터 비교할 필요 없이 cnt==2 부터 비교하면 됨.

(nana의 우선순위 전 단계 접미사가 na임을 의미하지는 않음)

BOJ 1701 - Cubeditor

```
// O(Nlog^2N)
void getSuffixArray(string& str, vl& SA, vl& rank) {
    ll n = str.size();
    SA.resize(n), rank.resize(n);
    for (ll i=0; i<n; i++)
        SA[i] = i, rank[i] = str[i];
    for (ll L=1; L<n; L<<=1) {
        auto cmp = [&](ll a, ll b) {
            if (rank[a] != rank[b])     // rank는 L만큼의 비교 정보 담고 있음
                return rank[a] < rank[b];
            ll ra = a+L<n ? rank[a+L] : -1;
            ll rb = b+L<n ? rank[b+L] : -1;
            return ra < rb;
        };
        sort(SA.begin(), SA.end(), cmp);
        vl tmp(n, 0);   // tmp[SA[0]] = 0;
```

```
        for (ll i=1; i<n; i++)
            tmp[SA[i]] = tmp[SA[i-1]] + (cmp(SA[i-1], SA[i]) ? 1 : 0);
        rank = tmp;
        if (rank[SA[n-1]] == n-1) break;
    }
}
// O(n)
void getLCP (string& str, vl& LCP) {
    ll n = str.size(); LCP.resize(n);
    vl SA, rank; getSuffixArray(str, SA, rank);
    for (ll cnt=0, i=0; i<n; i++) {
        if (!rank[i]) { cnt = 0; continue; }
        ll j = SA[rank[i] - 1];
        while (i+cnt<n && j+cnt<n && str[i+cnt]==str[j+cnt]) cnt++;
        LCP[rank[i]] = cnt;
        if (cnt) cnt--;
    }
}
```

아호코라식

```
// Usage: aho_corasick ac; ac.init(patterns); bool matched = ac.query(text);
// init: O(sum of |p|)
// query: O(|s|)

struct aho_corasick {
    static constexpr ll MAXN=100005, MAXC=26;
    vector<array<ll, MAXC>> trie;
    vl fail,term;
    ll piv=0;
    ll offset='A'; // 소문자라면 'a'로 바꿔야 함
    void init(vector<string> &v) {
        trie.assign(MAXN, array<ll,MAXC>{});
        fail.assign(MAXN, 0);
        term.assign(MAXN, 0);
        piv=0;
        for (auto &i: v) {
```

```
        ll p=0;
        for (auto &j: i) {
            ll k=j-offset;
            if (!trie[p][k]) trie[p][k]=++piv;
            p=trie[p][k];
        }
        term[p]=1;
    }
    queue<ll> que;
    for (ll i=0;i<MAXC;i++) if (trie[0][i]) que.push(trie[0][i]);
    while (!que.empty()) {
        ll x=que.front(); que.pop();
        for (ll i=0;i<MAXC;i++) if (trie[x][i]) {
            ll p=fail[x];
            while (p && !trie[p][i]) p=fail[p];
            p=trie[p][i];
            fail[trie[x][i]]=p;
            if (term[p]) term[trie[x][i]]=1;
            que.push(trie[x][i]);
        }
    }
}
bool query(string &s) {
    ll p=0;
    for (auto &i: s) {
        ll k=i-offset;
        while (p && !trie[p][k]) p=fail[p];
        p=trie[p][k]; if (term[p]) return 1;
    }
    return 0;
}
// 문자열 s에서 모든 매칭된 (패턴ID, 시작위치)를 반환
vector<pll> query_pos(const string& s, const vector<string>& patterns) {
    vector<pll> matches;
    ll p = 0;
    for (ll i = 0; i < (ll)s.size(); i++) {
        ll k = s[i] - offset;
        while (p && !trie[p][k]) p = fail[p];
```

```
            p = trie[p][k];
            // 이 노드에 매칭된 패턴들이 있으면
            for (ll id : out[p]) {
                ll start_idx = i - (ll)patterns[id].size() + 1;
                matches.emplace_back(id, start_idx);
            }
        }
        return matches;
    }
};
```

Trie

```
struct Trie {
    struct Node {
        map<char,ll> nxt;
        bool isEnd = false;
    };

    vector<Node> tree;

    Trie() { tree.emplace_back(); }

    void insert(const string& word) {
        ll cur = 0; // 루트
        for (auto& c : word) {
            if (!tree[cur].nxt.contains(c)) {
                tree[cur].nxt[c] = tree.size();
                tree.emplace_back();
            }
            cur = tree[cur].nxt[c];
        }
        tree[cur].isEnd = true;
    }

    bool search(const string& word) {
        ll cur = 0;
```

```
        for (auto& c : word) {
            if (!tree[cur].nxt.contains(c)) return false;
            cur = tree[cur].nxt[c];
        }
        return tree[cur].isEnd;
    }

    // 해당 prefix로 시작하는 단어 존재 여부
    bool startsWith(const string& prefix) {
        ll cur = 0;
        for (auto& c : prefix) {
            if (!tree[cur].nxt.contains(c)) return false;
            cur = tree[cur].nxt[c];
        }
        return true;
    }
};
```