NANJING UNIVERSITY

ACM-ICPC Codebook 0

Miscellaneous

June 15, 2017

Contents

1 General

1.1 Template

```
#include <iostream>
#include <cstdio>
#include <cstring>
#include <climits>
#include <vector>
#include <stack>
#include <queue>
#include <string>
#include <algorithm>
using namespace std;

#define rep(i, n) for (int i = 0; i < (n); i++)
#define Rep(i, n) for (int i = 1; i <= (n); i++)
typedef long long LL;

int main(){

    return 0;
}
```

2 String

2.1 Knuth-Morris-Pratt algorithm

Single-pattern matching.

Usage:

`construct(p)`	Construct the failure table of pattern `p`.
`match(t, p)`	Match pattern `p` in text `t`.
`found(pos)`	Report the pattern found at `pos`.

Time complexity: $O(l)$.

```
const int SIZE = 10005;
int fail[SIZE];
int len;

void construct(const char* p){
    len = strlen(p);
```

```
    fail[0] = fail[1] = 0;
    for (int i = 1; i < len; i++) {
        int j = fail[i];
        while (j && p[i] != p[j]) j = fail[j];
        fail[i+1] = p[i] == p[j] ? j+1 : 0;
    }
}

inline void found(int pos){
    // ! add codes for having found at pos
}

void match(const char* t, const char* p){ // must be called after construct
    int n = strlen(t);
    int j = 0;
    rep (i, n){
        while (j && p[j] != t[i]) j = fail[j];
        if (p[j] == t[i]) j++;
        if (j == len) found(i - len + 1);
    }
}
```

2.2 Trie

Support insertion and search for a set of words.

⚠ If duplicate word exists, only the last one is preserved.
⚠ The tag must not be 0, which is considered as not being a word.

Usage:

`id(c)`	Covert character to its id.
`add(s, t)`	Add word s into Trie, where t is the tag attached to s.
`search(s)`	Search for word s. Return the tag attached to s if found; otherwise return 0.

Time complexity: $O(l|\Sigma|)$ for insertion, $O(l)$ for search.

```
const int MAXN = 12000;
const int CHARN = 26;

inline int id(char c){
    return c - 'a';
}

struct Trie{
    int n;
```

```
    int tr[MAXN][CHARN]; // Trie tree, 0 denotes fail
    int tag[MAXN];

    Trie(){
        memset(tr[0], 0, sizeof(tr[0]));
        tag[0] = 0;
        n = 1;
    }

    // tag should not be 0
    void add(const char* s, int t){
        int p = 0, c, len = strlen(s);
        rep (i, len){
            c = id(s[i]);
            if (!tr[p][c]){
                memset(tr[n], 0, sizeof(tr[n]));
                tag[n] = 0;
                tr[p][c] = n++;
            }
            p = tr[p][c];
        }
        tag[p] = t;
    }

    // returns 0 if not found
    // AC automaton does not need this function
    int search(const char* s){
        int p = 0, c, len = strlen(s);
        rep (i, len){
            c = id(s[i]);
            if (!tr[p][c]) return 0;
            p = tr[p][c];
        }
        return tag[p];
    }
};
```

2.3 Aho-Corasick automaton

Automaton for multi-pattern matching.

⚠ See the warnings of Trie.
⚠ If a word has too many suffixes, the automaton might run slow.

Usage:

`add(s, t)`	Add word s into Trie, where t is the tag attached to s.
`construct()`	Construct the automaton after all words added.
`find(text)`	Find words in text.
`found(pos, j)`	Report a word found in node j, the last character of which is at `pos`.

Requirement:
2.2 Trie

Time complexity: $O(l|\Sigma|)$ for insertion and construction, $O(l)$ for finding, provided the number of suffixes of a word is constant.

```
struct AC : Trie{
    int fail[MAXN];
    int last[MAXN];

    void construct(){
        queue<int> q;
        fail[0] = 0;
        rep (c, CHARN){
            if (int u = tr[0][c]){
                fail[u] = 0;
                q.push(u);
                last[u] = 0;
            }
        }
        while (!q.empty()){
            int r = q.front(); q.pop();
            rep (c, CHARN){
                int u = tr[r][c];
                if (!u){
                    tr[r][c] = tr[fail[r]][c];
                    continue;
                }
                q.push(u);
                int v = fail[r];
                while (v && !tr[v][c]) v = fail[v];
                fail[u] = tr[v][c];
                last[u] = tag[fail[u]] ? fail[u] : last[fail[u]];
            }
        }
    }

    void found(int pos, int j){
        if (j) {
            // ! add codes for having found word with tag[j]
            found(pos, last[j]);
        }
```

```
37      }
38
39      void find(const char* text){ // must be called after construct()
40          int p = 0, c, len = strlen(text);
41          rep (i, len){
42              c = id(text[i]);
43              p = tr[p][c];
44              if (tag[p])
45                  found(i, p);
46              else if (last[p])
47                  found(i, last[p]);
48          }
49      }
50  };
```