NANJING UNIVERSITY

ACM-ICPC Codebook 0

Miscellaneous

June 14, 2017

Contents

1 General

1.1 Template

```cpp
#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 Trie tree

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.

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

```
3
4   inline int id(char c){
5       return c - 'a';
6   }
7
8   struct Trie{
9       int n;
10      int tr[MAXN][CHARN]; // Trie tree, 0 denotes fail
11      int tag[MAXN];
12
13      Trie(){
14          memset(tr[0], 0, sizeof(tr[0]));
15          tag[0] = 0;
16          n = 1;
17      }
18
19      // tag should not be 0
20      void add(const char* s, int t){
21          int p = 0, c, len = strlen(s);
22          rep (i, len){
23              c = id(s[i]);
24              if (!tr[p][c]){
25                  memset(tr[n], 0, sizeof(tr[n]));
26                  tag[n] = 0;
27                  tr[p][c] = n++;
28              }
29              p = tr[p][c];
30          }
31          tag[p] = t;
32      }
33
34      // returns 0 if not found
35      // AC automaton does not need this function
36      int search(const char* s){
37          int p = 0, c, len = strlen(s);
38          rep (i, len){
39              c = id(s[i]);
40              if (!tr[p][c]) return 0;
41              p = tr[p][c];
42          }
43          return tag[p];
44      }
45  } trie;
```

2.2 Aho-Corasick automaton

Automaton for multi-pattern matching.

⚠ See the warnings of Trie tree.
⚠ 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:
2.1 Trie tree

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

```
1  struct AC : Trie{
2      int fail[MAXN];
3      int last[MAXN];
4
5      void construct(){
6          queue<int> q;
7          fail[0] = 0;
8          rep (c, CHARN){
9              if (int u = tr[0][c]){
10                 fail[u] = 0;
11                 q.push(u);
12                 last[u] = 0;
13             }
14         }
15         while (!q.empty()){
16             int r = q.front(); q.pop();
17             rep (c, CHARN){
18                 int u = tr[r][c];
19                 if (!u){
20                     tr[r][c] = tr[fail[r]][c];
21                     continue;
22                 }
23                 q.push(u);
24                 int v = fail[r];
25                 while (v && !tr[v][c]) v = fail[v];
26                 fail[u] = tr[v][c];
27                 last[u] = tag[fail[u]] ? fail[u] : last[fail[u]];
28             }
```

```
        }
    }

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

    void find(const char* text){
        int p = 0, c, len = strlen(text);
        rep (i, len){
            c = id(text[i]);
            p = tr[p][c];
            if (tag[p])
                found(i, p);
            else if (last[p])
                found(i, last[p]);
        }
    }
};
```