NANJING UNIVERSITY

ACM-ICPC Codebook 0

Miscellaneous

August 21, 2017

Contents

1 General

1.1 Template

```
#include <bits/stdc++.h>
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++)
#define range(x) (x).begin(), (x).end()
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;
    }
}
```

```
14
15 inline void found(int pos){
16     // ! add codes for having found at pos
17 }
18
19 void match(const char* t, const char* p){ // must be called after construct
20     int n = strlen(t);
21     int j = 0;
22     rep (i, n){
23         while (j && p[j] != t[i]) j = fail[j];
24         if (p[j] == t[i]) j++;
25         if (j == len) found(i - len + 1);
26     }
27 }
```

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.

```
1  const int MAXN = 12000;
2  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; n = 1;
16     }
```

```
17
18      // tag should not be 0
19      void add(const char* s, int t){
20          int p = 0, c, len = strlen(s);
21          rep (i, len){
22              c = id(s[i]);
23              if (!tr[p][c]){
24                  memset(tr[n], 0, sizeof(tr[n]));
25                  tag[n] = 0;
26                  tr[p][c] = n++;
27              }
28              p = tr[p][c];
29          }
30          tag[p] = t;
31      }
32
33      // returns 0 if not found
34      // AC automaton does not need this function
35      int search(const char* s){
36          int p = 0, c, len = strlen(s);
37          rep (i, len){
38              c = id(s[i]);
39              if (!tr[p][c]) return 0;
40              p = tr[p][c];
41          }
42          return tag[p];
43      }
44  };
```

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]);
        }
    }

    void find(const char* text){ // must be called after construct()
        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]);
        }
    }
};
```

3 Mathematical Analysis

3.1 Fast Fourier transform

⚠ The size of the sequence must be some power of 2.
⚠ When performing convolution, the size of the sequence should be doubled. To compute k, one may call `32-__builtin_clz(a+b-1)`, where a and b are the lengths of two sequences.

Usage:

`FFT(k)`	Initialize the structure with maximum sequence length 2^k.
`fft(a)`	Perform Fourier transform on sequence a.
`ifft(a)`	Perform inverse Fourier transform on sequence a.
`conv(a, b)`	Convolve sequence a with b.

Time complexity: $O(n \log n)$ for `fft`, `ifft` and `conv`.

```
const int NMAX = 1<<20;
typedef complex<double> cplx;
const double PI = 2*acos(0.0);
struct FFT{
    int rev[NMAX];
    cplx omega[NMAX], oinv[NMAX];
    int K, N;

    FFT(int k){
        K = k; N = 1 << k;
        rep (i, N){
            rev[i] = (rev[i>>1]>>1) | ((i&1)<<(K-1));
            omega[i] = polar(1.0, 2.0 * PI / N * i);
            oinv[i] = conj(omega[i]);
        }
    }

    void dft(cplx* a, cplx* w){
        rep (i, N) if (i < rev[i]) swap(a[i], a[rev[i]]);
        for (int l = 2; l <= N; l *= 2){
            int m = l/2;
            for (cplx* p = a; p != a + N; p += l)
```

```
                rep (k, m){
                    cplx t = w[N/l*k] * p[k+m];
                    p[k+m] = p[k] - t; p[k] += t;
                }
        }
    }

    void fft(cplx* a){dft(a, omega);}
    void ifft(cplx* a){
        dft(a, oinv);
        rep (i, N) a[i] /= N;
    }

    void conv(cplx* a, cplx* b){
        fft(a); fft(b);
        rep (i, N) a[i] *= b[i];
        ifft(a);
    }
};
```

4 Game Theory

以下游戏中，不能动的算输。

4.1 Nim game

4.1.1 Bash game

有 n 个石子，每人最多拿 m 个，最少拿 1 个。$n \bmod (m+1) \neq 0$ 时先手必胜。

4.1.2 Fibonacci nim

有 n 个石子，第一轮可以拿不超过 n 个石子。此后，每次拿的石子数不超过前一次的 2 倍。当 n 是斐波那契数时先手必胜。

4.1.3 Wythoff's game

有 2 堆石子，分别有 a,b 个（$a \leq b$），每人可以从一堆中拿任意多个，或从两堆中拿相同多个。当 $a = \lfloor (b-a)\frac{\sqrt{5}+1}{2} \rfloor$ 时先手必败。