
NANJING UNIVERSITY

# ACM-ICPC Codebook 0

# **Miscellaneous**

May 4, 2018

# Contents

# 1 General

## 1.1 vimrc

```
set nocompatible
syntax on
colorscheme slate
set number
set cursorline
set shiftwidth=2
set softtabstop=2
set tabstop=2
set expandtab
set magic
set smartindent
set backspace=indent,eol,start
set cmdheight=1
set laststatus=2
set statusline=\ %<%F[%1*%M%*%n%R%H]%=\ %y\ %0(%{&fileformat}\ %{&encoding}\ %c
  :%l/%L%)\
set whichwrap=b,s,<,>,[,]
```

## 1.2 bashrc

```
mkdir -p ~/.trash
alias rm=trash
trash()
{
  mv $@ ~/.trash/
}

cleartrash()
{
   \rm -rvf ~/.trash
   mkdir -p ~/.trash
}
```

## 1.3 runbash

```
if [ $# -ge 1 ]; then
  fn=$1
  echo ${fn} > .run.log
```

```
else
  fn=`cat .run.log`
fi

# cat $fn.cpp | xsel -ib

if g++ $fn.cpp -std=c++11 -D__LOCAL_DEBUG__ -Wall -O2 -g -o $fn; then
  echo "********** Compilation Success! ********** [$fn]"
  if [ $# -ge 2 ]; then
    time -f "\n%U user, %S system, %e real" ./$fn < $2
  else
    time -f "\n%U user, %S system, %e real" ./$fn
  fi
#  cat $fn.cpp | xsel -ib
else
  echo "********** Compilation Failed! ********** [$fn]"
fi
```

## 1.4 Template

```
#include <bits/stdc++.h>
using namespace std;

#ifdef __LOCAL_DEBUG__
# define _debug(fmt, ...) fprintf(stderr, "\033[94m%s:␣" fmt "\n\033[0m", \
    __func__, ##__VA_ARGS__)
#else
# define _debug(...) ((void) 0)
#endif
#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;
typedef unsigned long long ULL;

template <unsigned p>
struct Zp{
    unsigned x;
    Zp(unsigned x):x(x){}
    operator unsigned(){return x;}
    Zp operator ^ (ULL e) {
        Zp b=x, r=1;
        while (e) {
            if (e&1) r=r*b;
            b=b*b;
```

```
            e>>=1;
        }
        return r;
    }
    Zp operator + (Zp rhs) {return (x+rhs)%p;}
    Zp operator - (Zp rhs) {return (x+p-rhs)%p;}
    Zp operator * (Zp rhs) {return x*rhs%p;}
    Zp operator / (Zp rhs) {return Zp(x)*(rhs^(p-2));}
};

typedef Zp<1000000007> zp;

zp operator"" _ (ULL n){return n;}
```

# 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
}
```

```
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);
```

```
        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{
```

```
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          }
29      }
30  }
31
32  void found(int pos, int j){
33      if (j) {
34          // ! add codes for having found word with tag[j]
35          found(pos, last[j]);
36      }
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  };
```

# 3 Game Theory

## 3.1 Nim games

以下游戏中，不能动的算输。

### 3.1.1 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。

### 3.1.2 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。

### 3.1.3 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。

# 4 Others

## 4.1 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);
    }
};
```

## 4.2 2-SAT

**Usage:**

| | |
|---|---|
| `init(n)` | Initialize the structure with at most $n$ Boolean variables. |
| `add_clause(x, xval, y, yval)` | Add clause: $x = xval$ or $y = yval$. |
| `solve()` | Solve the 2-SAT problem. Return false if no solution. |
| `value(i)` | Return the value of $i$-th variable in some solution, if exists. |

**Time complexity:** $O(m + n)$.

```
const int MAXN = 100005;
struct twoSAT{
    int n;
    vector<int> G[MAXN*2];
    bool mark[MAXN*2];
    int S[MAXN*2], c;

    void init(int n){
        this->n = n;
        for (int i=0; i<n*2; i++) G[i].clear();
        memset(mark, 0, sizeof(mark));
    }

    bool dfs(int x){
        if (mark[x^1]) return false;
        if (mark[x]) return true;
        mark[x] = true;
        S[c++] = x;
        for (int i=0; i<G[x].size(); i++)
            if (!dfs(G[x][i])) return false;
        return true;
    }

    void add_clause(int x, bool xval, int y, bool yval){
        x = x * 2 + xval;
        y = y * 2 + yval;
        G[x^1].push_back(y);
        G[y^1].push_back(x);
    }

    bool solve() {
        for (int i=0; i<n*2; i+=2){
            if (!mark[i] && !mark[i+1]){
                c = 0;
                if (!dfs(i)){
                    while (c > 0) mark[S[--c]] = false;
```

```
                    if (!dfs(i+1)) return false;
                }
            }
        }
        return true;
    }

    inline bool value(unsigned i){return mark[2*i+1];}
};
```