Nanjing University

ACM-ICPC Codebook 0

Miscellaneous

August 28, 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
}
```

```

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]);
  }
}

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]);
  }
```

```
  }
};
```

2.4 Manacher

Find maximum palindrome radii for all centers.

Usage:

`init(str)`	Run this algorithm on `str`.
`maxpar(l, r)`	Query maximal palindrome central region between $[l, r)$.

Time complexity: Linear in length of string.

```
struct Manacher {
  int Len;
  vector<int> lc;
  string s;

  void work() {
    lc[1] = 1;
    int k = 1;

    for (int i = 2; i <= Len; i++) {
      int p = k + lc[k] - 1;
      if (i <= p) {
        lc[i] = min(lc[2 * k - i], p - i + 1);
      } else {
        lc[i] = 1;
      }
      while (s[i + lc[i]] == s[i - lc[i]]) lc[i]++;
      if (i + lc[i] > k + lc[k]) k = i;
    }
  }

  void init(const char *tt) {
    int len = strlen(tt);
    s.resize(len * 2 + 10);
    lc.resize(len * 2 + 10);
    s[0] = '*';
    s[1] = '#';
    for (int i = 0; i < len; i++) {
      s[i * 2 + 2] = tt[i];
      s[i * 2 + 1] = '#';
    }
    s[len * 2 + 1] = '#';
    s[len * 2 + 2] = '\0';
```

```
    Len = len * 2 + 2;
    work();
  }

  pair<int, int> maxpal(int l, int r) {
    int center = l + r + 1;
    int rad = lc[center] / 2;
    int rmid = (l + r + 1) / 2;
    int rl = rmid - rad, rr = rmid + rad - 1;
    if ((r ^ l) & 1) {
    } else rr++;
    return {max(l, rl), min(r, rr)};
  }
};
```

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 Dynamic Programming Optimization

4.1 Knuth's Optimization

Knuth's optimization is applicable for the dynamic programming of the form

$$\mathrm{dp}[i][j] = \min_{i<k<j} \{\mathrm{dp}[i][k] + \mathrm{dp}[k][j]\} + C[i][j]$$

whenever $A[i][j-1] \leq A[i][j] \leq A[i+1][j]$.

A sufficient condition for Knuth's optimization is that C follows the monotonicity and quadrangle inequality:

monotonicity $C[a][d] \leq C[b][c], a \leq b \leq c \leq d$;

quadrangle inequality $C[a][c] + C[b][d] \leq C[a][d] + C[b][c], a \leq b \leq c \leq d$.

Usage:

`n`	the total length of the array (0-based)
`cost`	function C
`dp`	the result of dynamic programming
`dc`	decision point

Time complexity: $O(n^2)$.

```
int n;
int dp[256][256], dc[256][256];

template <typename T>
void compute(T cost) {
  for (int i = 0; i <= n; i++) {
    dp[i][i] = 0;
    dc[i][i] = i;
  }
  rep (i, n) {
    dp[i][i+1] = 0;
    dc[i][i+1] = i;
  }
  for (int len = 2; len <= n; len++) {
    for (int i = 0; i + len <= n; i++) {
      int j = i + len;
      int lbnd = dc[i][j-1], rbnd = dc[i+1][j];
      dp[i][j] = INT_MAX / 2;
      int c = cost(i, j);
      for (int k = lbnd; k <= rbnd; k++) {
```

```
        int res = dp[i][k] + dp[k][j] + c;
        if (res < dp[i][j]) {
          dp[i][j] = res;
          dc[i][j] = k;
        }
      }
    }
  }
};
```

5 Others

5.1 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);
    }
};
```

5.2 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];}
};
```