

Dwylkz's Algorithm Library

GuangZhou University
Team Kimady
Dwylkz(Wūdōng Yáng)

November 8, 2013

Contents

1	Perface	1
2	vimrc	1
3	main	2
4	makefile	2
5	numeric	2
5.1	High Precision Integer	2
5.2	Minimum Prime Factor Sieve	3
5.3	Contor coding.	4
5.4	Chinese Remind Theory	5
5.5	Base 2 Fast Fourier Transfrom	5
5.6	Triangle Diagonal Matrix Algorithm	6
6	pattern	6
6.1	KMP	6
6.2	Extend KMP	7
6.3	Manacher	7
6.4	Minimum Notation	7
6.5	AC automaton	8
6.6	Suffix Array	8
6.7	Suffix Automaton	9
7	data	10
7.1	RMQ Sparse Table	10
7.2	Splay	11
7.3	Treap	13
7.4	Link-Cut Tree	15
7.5	Functional Segment	17
7.6	Functional Trie	18
7.7	Lefist Tree	18
8	geometry	19
8.1	Float Compare Functions	19
8.2	2D point	19
8.3	Angle Sort	20
8.4	Graham Scan	20
9	graph	20
9.1	Graph	20
9.2	Shortest Path Algorithm	21
9.3	Bipartite Graph match	22
9.4	General Graph match	22
9.5	Dancing Link	24

9.6	Directed Minimum Spanning Tree	27
9.7	Spfa Cost Stream	28
9.8	KM Maximum perfect match	29
9.9	Doubling LCA	30
9.10	Shortest Augment Path	30
9.11	ZKW Cost Stream	31
10	graph test	32
10.1	Graph	32
10.2	Shortest Augment Path	33
10.3	Strong Connected Component	33
10.4	Heavy Light Division	34
10.5	Biconnected Component	35
10.6	Static Lowest Ancestor	35

1 Preface

The content enclosed by '<' and '>' beside template, is just a note which remind us to define something before we use this template.

2 vimrc

```
1 colorscheme desert
2 set langmenu=en_US.UTF-8
3 source $VIMRUNTIME/delmenu.vim
4 source $VIMRUNTIME/menu.vim
5 language messages en_US.UTF-8
6 syntax on
7 filetype on
8 filetype plugin indent on
9 set smartindent
10 set autochdir
11 set autoindent
12 set smartindent
13 set backspace=2
14 set columns=120
15 set foldmethod=syntax
16 set nohlsearch
17 set incsearch
18 set lines=40
19 set nocompatible
20 set noswapfile
21 set number
22 set shiftwidth=2
23 set tabstop=2
24 set expandtab
25
26 func Compile()
27     exec "w"
28     exec "make"
29 endfunc
30 func Debug()
31     exec "w"
32     exec "make_debug"
33 endfunc
34 func Run()
35     exec "w"
36     exec "make_run"
37 endfunc
38
39 map <F4> :tabp<CR>
40 map <F5> :tabn<CR>
41 map <C-a> ggVG
42 map <C-p> "+p
43 map_<F8>_<ESC>:clist<CR>
44 map_<C-F8>_<ESC>:call _Debug()<CR>
45 map_<F9>_<ESC>:call _Compile()<CR>
46 map_<C-F9>_<ESC>:call _Run()<CR>
47 map_<silent><F6>_:s#^#//_#g<CR>
48 map_<silent><F7>_:s#^//_###g<CR>
49 vmap_<C-y>_" +y
```

3 main

```
1 #include <cstdio>
2 #include <cstdlib>
3 #include <iostream>
4 using namespace std;
5
6 int main() {
7     #if 1
8         freopen("input.in", "r", stdin);
9     #endif
10    return 0;
11 }
```

4 makefile

```
1 main: main.cc
2     gcc main.cc -o main -lstdc++ -g
3 .PHONY: debug clean run
4 debug:
5     gdb main
6 run:
7     ./main
```

5 numeric

5.1 High Precision Integer

```
1 /* High-Precision-Integer
2  * */
3 struct int_t {
4     string d;
5     int_t(string _d = "0"): d(_d) {}
6     int_t(int _d) {
7         static char buff[20];
8         sprintf(buff, "%d", _d);
9         d = buff;
10    }
11    static void trans(string &s) {
12        for (int i = 0; i < s.length(); i++) s[i] += '0';
13    }
14    friend int_t &operator + (const int_t &lhs, const int_t &rhs) {
15        static int_t result;
16        const string &a = lhs.d, &b = rhs.d;
17        string &c = result.d;
18        int maxlen = max(a.length(), b.length())+1;
19        c.resize(maxlen);
20        fill(c.begin(), c.end(), 0);
21        for (int i = 0; i < maxlen-1; i++) {
22            int x = a.length() <= i? 0: a[a.length()-1-i]- '0',
23                y = b.length() <= i? 0: b[b.length()-1-i]- '0';
24            c[i] += x+y;
25            c[i+1] += c[i]/10;
26            c[i] %= 10;
27        }
28        if (!c[maxlen-1]) c.resize(maxlen-1);
```

```

29     reverse(c.begin(), c.end());
30     trans(c);
31     return result;
32 }
33 friend int_t &operator += (const int_t &lhs, const int_t &rhs) {
34     return lhs+rhs;
35 }
36 friend int_t &operator * (const int_t &lhs, const int_t &rhs) {
37     static int_t result;
38     const string &a = lhs.d, &b = rhs.d;
39     string &c = result.d;
40     int maxlen = a.length()+b.length();
41     c.resize(maxlen);
42     fill(c.begin(), c.end(), 0);
43     for (int i = 0; i < a.length(); i++) {
44         int x = a[a.length()-1-i]- '0';
45         for (int j = 0; j < b.length(); j++) {
46             int y = b[b.length()-1-j]- '0';
47             c[i+j] += x*y;
48             c[i+j+1] += c[i+j]/10;
49             c[i+j] %= 10;
50         }
51     }
52     for ( ; maxlen > 1 && !c[maxlen-1]; maxlen--) {}
53     c.resize(maxlen);
54     reverse(c.begin(), c.end());
55     trans(c);
56     return result;
57 }
58 friend int_t &operator *= (const int_t &lhs, const int_t &rhs) {
59     return lhs*rhs;
60 }
61 const char *show() {
62     return d.data();
63 }
64 };

```

5.2 Minimum Prime Factor Sieve

```

1  /* Minimum_Prime_Factor_Sieve
2   * N   : upper bound
3   * p[] : primes
4   * n   : primes number
5   * e[] : eular funtion
6   * d[] : divisors number
7   * f[] : minimum prime factor
8   * c[] : minimum prime factor's power
9   * m[] : mobius function
10  */
11 template<int N> struct sieve_t {
12     int b[N], p[N], n, e[N], d[N], f[N], c[N], m[N];
13     sieve_t() {
14         memset(this, 0, sizeof(sieve_t));
15         d[1] = m[1] = 1;
16         for (int i = 2; i < N; i++) {
17             if (!b[i]) {
18                 e[i] = i-1;
19                 c[i] = 1;
20                 d[i] = 2;

```

```

21     f[i] = i;
22     m[i] = -1;
23     p[n++] = i;
24 }
25 for (int j = 0; j < n && 1ll*i*p[j] < N; j++) {
26     int k = i*p[j];
27     b[k] = 1;
28     f[k] = p[j];
29     if (i%p[j]) {
30         e[k] = e[i]*(p[j]-1);
31         c[k] = 1;
32         d[k] = d[i]*(c[k]+1);
33         m[k] = m[i]*m[p[j]];
34     } else {
35         e[k] = e[i]*p[j];
36         c[k] = c[i]+1;
37         d[k] = d[i]/(c[i]+1)*(c[k]+1);
38         m[k] = 0;
39         break;
40     }
41 }
42 }
43 }
44 };

```

5.3 Contor coding.

```

1  /* Contor_coding.
2  * Notice that x in [1, l!] in array->integer mapping
3  * while x in [0, l!) in integer->array mapping. */
4  template<int N> struct contor_t {
5      int f[N];
6      contor_t() {
7          for (int i = f[0]= 1; i < N; i++)
8              f[i] = f[i - 1]*i;
9      }
10     void operator () (int l, int x, int *t) {
11         int id = 0, h[100] = {0};
12         x--;
13         for (int i = l-1; 0 <= i; i--) {
14             int rm = x/f[i], rank = 0;
15             for (int j = 1; j <= l; j++) {
16                 rank += !h[j];
17                 if (rank == rm+1) {
18                     t[id++] = j;
19                     h[j] = 1;
20                     break;
21                 }
22             }
23             x %= f[i];
24         }
25     }
26     int operator () (int l, int *t) {
27         int rv = 0;
28         for (int i = 0; i < l; i++) {
29             int cnt = 0;
30             for (int j = i+1; j < l; j++)
31                 if (t[j] < t[i]) cnt++;
32             rv += cnt*f[l-i-1];

```

```

33     }
34     return rv;
35 }
36 };

```

5.4 Chinese Remind Theory

```

1  /* Chinese_Remind_Theory
2  * */
3  template<int N> struct crt_t {
4      vector<int> a, b;
5      int gcd(int a, int b, int &x, int &y) {
6          int d, tx, ty;
7          if (b == 0) {
8              x = 1;
9              y = 0;
10             return a;
11         }
12         d = gcd(b, a%b, tx, ty);
13         x = ty;
14         y = tx - (a/b)*ty;
15         return d;
16     }
17     int mle(int a, int b, int n) {
18         int d, x, y;
19         d = gcd(a, n, x, y);
20         if (b%d == 0) {
21             x = 1ll*x*b/d%n;
22             return x;
23         }
24         return 0;
25     }
26     int init() {
27         a.clear();
28         b.clear();
29     }
30     int operator () () {
31         int x = 0, n = 1, i, bi;
32         for (i = 0; i < b.size(); i++) n *= b[i];
33         for (i = 0; i < a.size(); i++) {
34             bi = mle(n/b[i], 1, b[i]);
35             x = (x+1ll*a[i]*bi*(n/b[i]))%n;
36         }
37         return x;
38     }
39 };

```

5.5 Base 2 Fast Fourier Transfrom

```

1  /* Base_2_Fast_Fourier_Transfrom
2  * (): transfrom
3  * []: inversion */
4  struct b2_fft_t {
5      typedef complex<double> cd_t;
6      typedef vector<cd_t> vcd_t;
7      vcd_t c;
8      void brc(vcd_t &x) {
9          int l;

```

```

10     for (l = 1; l < x.size(); l <= 1) {}
11     c.resize(l);
12     for (int i = 0; i < c.size(); i++) {
13         int to = 0;
14         for (int o = l>>1, t = i; o; o >>= 1, t >>= 1)
15             if (t&1) to += o;
16         c[to] = i < x.size()? x[i]: cd_t(0., 0.);
17     }
18 }
19 void fft(int on) {
20     double dpi = acos(-1.)*on;
21     for (int m = 1; m < c.size(); m <= 1) {
22         cd_t wn(cos(dpi/m), sin(dpi/m));
23         for (int j = 0; j < c.size(); j += m<<1) {
24             cd_t w = 1.;
25             for (int k = j; k < j+m; k++, w *= wn) {
26                 cd_t u = c[k], t = w*c[k+m];
27                 c[k] = u+t, c[k+m] = u-t;
28             }
29         }
30     }
31     if (~on) return;
32     for (int i = 0; i < c.size(); i++)
33         c[i] /= c.size()*1.;
34 }
35 void operator () (vcd_t &x) {
36     brc(x), fft(1), x = c;
37 }
38 void operator [] (vcd_t &x) {
39     brc(x), fft(-1), x = c;
40 }
41 };

```

5.6 Triangle Diagonal Matrix Algorithm

```

1  /* Triangle_Diagonal_Matrix_Algorithm
2  * */
3  template<class T> struct tdma_t {
4      void operator () (int n, T *a, T *b, T *c, T *d, T *x) {
5          for (int i = 0; i < n; i++) {
6              T tp = a[i]/b[i-1];
7              b[i] -= tp*c[i-1];
8              d[i] -= tp*d[i-1];
9          }
10         x[n-1] = d[n-1]/b[n-1];
11         for (int i = n-2; ~i; i--) x[i] = (d[i]-c[i]*x[i+1])/b[i];
12     }
13 };

```

6 pattern

6.1 KMP

```

1  /* KMP
2  * */
3  template<class T> struct kmp_t {
4      void get(T *p, int pl, int *f) {
5          for (int i = 0, j = f[0] = -1; i < pl; f[++i] = ++j)

```



```

6     for ( ; ~j && p[i] != p[j]; ) j = f[j];
7 }
8 void operator () (T *p, int pl, int *f) {
9     int i = 0, j = f[0] = -1;
10    for ( ; i < pl; i++, j++, f[i] = p[i] == p[j]? f[j]: j)
11        for ( ; ~j && p[i] != p[j]; ) j = f[j];
12 }
13 int operator () (T *s, int sl, T *p, int pl, int *f) {
14     int i = 0, j = 0;
15     for ( ; i < sl && j < pl; i++, j++)
16         for ( ; ~j && s[i] != p[j]; ) j = f[j];
17     return j;
18 }
19 };

```

6.2 Extend KMP

```

1  /* Extend_KMP
2  * */
3  template<class T> struct exkmp_t {
4      void operator () (T *p, int pl, int *g) {
5          g[g[1] = 0] = pl;
6          for (int i = 1, k = 1; i < pl; (k+g[k] < i+g[i]? k = i: 0), i++)
7              for (g[i] = min(g[i-k], max(k+g[k]-i, 0)); ; g[i]++)
8                  if (i+g[i] >= pl || p[i+g[i]] != p[g[i]]) break;
9      }
10     void operator () (T *s, int sl, int *f, T *p, int pl, int *g) {
11         for (int i = f[0] = 0, k = 0; i < sl; (k+f[k] < i+f[i]? k = i: 0), i++)
12             for (f[i] = min(g[i-k], max(k+f[k]-i, 0)); ; f[i]++)
13                 if (i+f[i] >= sl || f[i] >= pl || s[i+f[i]] != p[f[i]]) break;
14     }
15 };

```

6.3 Manacher

```

1  /* Manacher
2  * */
3  template<class T> struct mana_t {
4      void operator () (T *s, int &n, int *p) {
5          for (int i = n<<1; i >= 0; i--) s[i] = i&1? s[i>>1]: -1;
6          p[s[n = n<<1|1] = 0] = 1;
7          for (int i = p[1] = 2, k = 1; i < n; i++) {
8              p[i] = min(p[2*k-i], max(k+p[k]-i, 1));
9              for (; p[i] <= i && i+p[i] < n && s[i-p[i]] == s[i+p[i]]; ) p[i]++;
10             if (k+p[k] < i+p[i]) k = i;
11         }
12     }
13 };

```

6.4 Minimum Notation

```

1  /* Minimum_Notation
2  * */
3  template<class T, class C> struct mnn_t {
4      int operator () (T *s, int n) {
5          int i = 0, j = 1;
6          for (int k = 0; k < n; )

```

```

7     if (s[(i+k)%n] == s[(j+k)%n]) k++;
8     else if (C()(s[(i+k)%n], s[(j+k)%n])) j += k+1, k = 0;
9     else i += k+1, j = i+1, k = 0;
10    return i;
11  }
12 };

```

6.5 AC automaton

```

1  /* AC_automaton
2  * */
3  template<class T, int n, int m> struct aca_t {
4      struct node {
5          node *s[m], *p;
6          int ac;
7      } s[n], *top, *rt, *q[n];
8      void init() {
9          memset(top = s, 0, sizeof(s));
10         rt = top++;
11     }
12     void put(T *k, int l, int ac) {
13         node *x = rt;
14         for (int i = 0; i < l; i++) {
15             if (!x->s[k[i]]) x->s[k[i]] = top++;
16             x = x->s[k[i]];
17         }
18         x->ac = ac;
19     }
20     void link() {
21         int l = 0;
22         for (int i = 0; i < m; i++)
23             if (rt->s[i]) (q[l++] = rt->s[i])->p = rt;
24         else rt->s[i] = rt;
25         for (int h = 0; h < l; h++)
26             for (int i = 0; i < m; i++)
27                 if (q[h]->s[i]) {
28                     (q[l++] = q[h]->s[i])->p = q[h]->p->s[i];
29                     q[h]->s[i]->ac |= q[h]->p->ac;
30                 } else q[h]->s[i] = q[h]->p->s[i];
31     }
32     void tom(int mt[][n]) {
33         for (node *x = s; x < top; x++)
34             for (int i = 0; i < m; i++)
35                 if (!x->s[i]->ac) mt[x-s][x->s[i]-s] = 1;
36     }
37 };

```

6.6 Suffix Array

```

1  /* Suffix_Array
2  * Notice that the input array should end with 0 (s[s's length-1] = 0)
3  * and then invoke dc3, remember to expand N to 3 times of it. */
4  template<int N> struct sa_t {
5      int wa[N], wb[N], wv[N], ws[N], r[N];
6      void da(int *s, int n, int *sa, int m) {
7          #define da_F(c, a, b) for (int c = (a); i < (b); i++)
8          #define da_C(s, a, b, l) (s[a] == s[b] && s[a+1] == s[b+1])
9          #define da_R(x, y, z) da_F(i, 0, m) ws[i] = 0; da_F(i, 0, n) ws[x]++; \

```

```

10 da_F(i, 1, m) ws[i] += ws[i-1]; da_F(i, 0, n) sa[--ws[y]] = z;
11 int *x = wa, *y = wb;
12 da_R(x[i] = s[i], x[n-i-1], n-i-1);
13 for(int j = 1, p = 1; p < n; j *= 2, m = p) {
14     da_F(i, (p = 0, n-j), n) y[p++] = i;
15     da_F(i, 0, n) if(sa[i] >= j) y[p++] = sa[i]-j;
16     da_F(i, 0, n) wv[i] = x[y[i]];
17     da_R(wv[i], wv[n-i-1], y[n-i-1]);
18     da_F(i, (swap(x, y), x[sa[0]] = 0, p = 1), n)
19         x[sa[i]] = da_C(y, sa[i-1], sa[i], j)? p-1: p++;
20 }
21 }
22 int dc3_c12(int k, int *r, int a, int b, int *wv) {
23     if (k != 2) return r[a]<r[b] || r[a]==r[b] && wv[a+1]<wv[b+1];
24     return r[a]<r[b] || r[a]==r[b] && dc3_c12(1, r, a+1, b+1, wv);
25 }
26 void dc3(int *s, int n, int *sa, int m) {
27 #define dc3_H(x) ((x)/3+((x)%3 == 1? 0: tb))
28 #define dc3_G(x) ((x) < tb? (x)*3+1: ((x)-tb)*3+2)
29 #define dc3_c0(s, a, b) (s[a]==s[b] && s[a+1]==s[b+1] && s[a+2]==s[b+2])
30 #define dc3_F(c, a, b) for (int c = (a); c < (b); c++)
31 #define dc3_sort(s, a, b, n, m) dc3_F(i, 0, n) wv[i] = (s)[(a)[i]]; \
32     dc3_F(i, 0, m) ws[i] = 0; dc3_F(i, 0, n) ws[wv[i]]++; \
33     dc3_F(i, 1, m) ws[i] += ws[i-1]; \
34     dc3_F(i, 0, n) (b)[--ws[wv[n-i-1]]] = a[n-i-1];
35     int i, j, *rn = s+n, *san = sa+n, ta = 0, tb = (n+1)/3, tbc = 0, p;
36     dc3_F(i, s[n] = s[n+1] = 0, n) if(i%3) wa[tbc++] = i;
37     dc3_sort(s+2, wa, wb, tbc, m);
38     dc3_sort(s+1, wb, wa, tbc, m);
39     dc3_sort(s, wa, wb, tbc, m);
40     dc3_F(i, (rn[dc3_H(wb[0])] = 0, p = 1), tbc)
41         rn[dc3_H(wb[i])] = dc3_c0(s, wb[i-1], wb[i])? p-1: p++;
42     if(p < tbc) dc3(rn, tbc, san, p);
43     else dc3_F(i, 0, tbc) san[rn[i]] = i;
44     dc3_F(i, 0, tbc) if(san[i] < tb) wb[ta++] = san[i]*3;
45     if(n%3 == 1) wb[ta++] = n-1;
46     dc3_sort(s, wb, wa, ta, m);
47     dc3_F(i, 0, tbc) wv[wb[i] = dc3_G(san[i])] = i;
48     for(i = j = p = 0; i < ta && j < tbc; p++)
49         sa[p] = dc3_c12(wb[j]%3, s, wa[i], wb[j], wv)? wa[i++]:wb[j++];
50     for( ; i < ta; p++) sa[p] = wa[i++];
51     for( ; j < tbc; p++) sa[p] = wb[j++];
52 }
53 void ch(int *s, int n, int *sa, int *h) {
54     for (int i = 1; i < n; i++) r[sa[i]] = i;
55     for (int i = 0, j, k = 0; i < n-1; h[r[i++]] = k)
56         for (k? k--: 0, j = sa[r[i]-1]; s[i+k] == s[j+k]; k++);
57 }
58 void icats(int *b, int *l, char *s) {
59     static int delim = 'z'+1;
60     for (*l += strlen(s)+1; *s; s++) *b++ = *s;
61     *b++ = delim++;
62 }
63 };

```

6.7 Suffix Automaton

```

1 /* Suffix_Automaton
2  * */

```

```

3 template<int N, int M> struct sam_t {
4     static const int n = N*3;
5     struct node {
6         node *s[M], *p;
7         int l;
8         int range() {
9             return l-(p? l-p->l: 0);
10        }
11    } s[n], *top, *back;
12    node *make(int l) {
13        memset(top, 0, sizeof(node));
14        top->l = l;
15        return top++;
16    }
17    void init() {
18        top = s;
19        back = make(0);
20    }
21    void put(int k) {
22        node *x = make(back->l+1), *y = back;
23        for ( ; y && !y->s[k]; y = y->p) y->s[k] = x;
24        if (!y) x->p = s;
25        else {
26            node *w = y->s[k];
27            if (w->l == y->l+1) x->p = w;
28            else {
29                node *z = make(0);
30                *z = *w;
31                z->l = y->l+1;
32                x->p = w->p = z;
33                for ( ; y && y->s[k] == w; y = y->p) y->s[k] = z;
34            }
35        }
36    }
37 };

```

7 data

7.1 RMQ Sparse Table

```

1  /* RMQ-Sparse-Table
2   * */
3  template<int N> struct rmq_t {
4      int s[20][N], *k;
5      void operator () (int l, int *_k) {
6          k = _k;
7          for (int i = 0; i < l; i++) s[0][i] = i;
8          for (int i = 1; i < 20; i++)
9              if ((1<<i) <= l) for (int j = 0; j < l; j++)
10                  if (k[s[i-1][j]] < k[s[i-1][j+(1<<(i-1))]]) s[i][j] = s[i-1][j];
11                  else s[i][j] = s[i-1][j+(1<<(i-1))];
12      }
13      int operator () (int l, int r) {
14          if (l > r) swap(l, r);
15          int i = r-l+1, o = 1, j = 0;
16          for (int o = 1 ; o <= i; o <=<= 1) j++;
17          j--, r = r-(1<<j)+1;
18          return k[s[j][l]] < k[s[j][r]]? s[j][l]: s[j][r];
19      }

```

7.2 Splay

```

1  /* Splay
2   * */
3  struct splay_t {
4      struct node {
5          node *p, *s[2];
6          int size, key;
7          int sum, lsum, rsum, msum;
8          int cover_tag, reverse_tag;
9          int side() {
10             return p->s[1] == this;
11         }
12         int rank() {
13             return s[0]? 1+s[0]->size: 1;
14         }
15         node *set_cover(int _key) {
16             cover_tag = 1;
17             key = _key;
18             sum = size*key;
19             lsum = rsum = msum = max(sum, key);
20             return this;
21         }
22         node *set_reverse() {
23             reverse_tag ^= 1;
24             swap(s[0], s[1]);
25             swap(lsum, rsum);
26             return this;
27         }
28         node *push() {
29             for (int i = 0; i < 2; i++) {
30                 if (!s[i]) continue;
31                 if (cover_tag) s[i]->set_cover(key);
32                 if (reverse_tag) s[i]->set_reverse();
33             }
34             cover_tag = reverse_tag = 0;
35             return this;
36         }
37         node *merge_sum(node *x, node *y) {
38             if (!x || !y) return &(*this = x? *x: *y);
39             sum = x->sum+y->sum;
40             lsum = max(x->lsum, x->sum+y->lsum);
41             rsum = max(y->rsum, x->rsum+y->sum);
42             msum = max(x->msum, y->msum);
43             msum = max(msum, max(lsum, rsum));
44             msum = max(msum, x->rsum+y->lsum);
45             return this;
46         }
47         node *pull() {
48             size = 1;
49             sum = lsum = rsum = msum = key;
50             for (int i = 0; i < 2; i++) {
51                 if (!s[i]) continue;
52                 size += s[i]->size;
53             }
54             return merge_sum(node(*this).merge_sum(s[0], this), s[1]);
55     }

```

```

56 node *set(int b, node *x) {
57     if (push()->s[b] = x) x->p = this;
58     return pull();
59 }
60 node *get(int b) {
61     return push()->s[b];
62 }
63 node *cut(int b, node *&x) {
64     if (x = push()->s[b]) s[b]->p = 0;
65     s[b] = 0;
66     return pull();
67 }
68 node *spin() {
69     node *y = p->push();
70     int b = push()->side();
71     if (p = y->p) p->s[y->side()] = this;
72     if (y->s[b] = s[!b]) s[!b]->p = y;
73     return (s[!b] = y)->pull()->p = this;
74 }
75 node *fine(node *x = 0) {
76     for ( ; p != x; spin())
77         if (p->p != x)
78             if (side() == p->side()) p->spin();
79             else spin();
80     return pull();
81 }
82 node *pick(int k, node *y = 0) {
83     node *x = this;
84     for ( ; x->rank() != k; ) {
85         int b = x->rank() < k;
86         k -= b*x->rank();
87         x = x->get(b);
88     }
89     return x->fine(y);
90 }
91 };
92 node *give(node *t = 0) {
93     static node *top = 0;
94     static int size = 1;
95     if (t) t->s[1] = top, top = t;
96     else {
97         if (!top) {
98             top = new node[size <<= 1];
99             for (int i = 0; i < size - 1; i++)
100                 top[i].s[1] = top + i + 1;
101             top[size - 1].s[1] = 0;
102         }
103         t = top, top = top->s[1];
104     }
105     return t;
106 }
107 node *make(int key) {
108     node t = {0, {0}, 1, key, key, key, key, 0, 0};
109     return &(*give() = t);
110 }
111 void drop(node *&t) {
112     if (!t) return;
113     drop(t->s[0]), drop(t->s[1]);
114     give(t), t = 0;
115 }

```

```

116 void show(node *t) {
117     if (!t) return ;
118     t->push();
119     show(t->s[0]);
120     printf("%d", t->key);
121     show(t->s[1]);
122     t->pull();
123 }
124 };

```

7.3 Treap

```

1  /* Treap
2   * */
3  struct treap_t {
4      struct node {
5          node *s[2];
6          int size, weight;
7          int key, cover_tag, sum,
8              max_sum, lsum, rsum, reverse_tag;
9          int rank() {
10             return s[0]? s[0]->size+1: 1;
11         }
12         node *set_cover(int _key) {
13             key = _key;
14             cover_tag = 1;
15             sum = size*key;
16             max_sum = lsum = rsum = max(key, sum);
17             return this;
18         }
19         node *set_reverse() {
20             reverse_tag ^= 1;
21             swap(s[0], s[1]);
22             swap(lsum, rsum);
23             return this;
24         }
25         node *push() {
26             for (int i = 0; i < 2; i++) {
27                 if (!s[i]) continue;
28                 if (cover_tag) s[i]->set_cover(key);
29                 if (reverse_tag) s[i]->set_reverse();
30             }
31             cover_tag = reverse_tag = 0;
32             return this;
33         }
34         node *merge_sum(node *x, node *y) {
35             if (!x || !y) {
36                 x? *this = *x: *this = *y;
37                 return this;
38             }
39             sum = x->sum+y->sum;
40             lsum = max(x->lsum, x->sum+y->lsum);
41             rsum = max(y->rsum, x->rsum+y->sum);
42             max_sum = max(x->max_sum, y->max_sum);
43             max_sum = max(max_sum, max(lsum, rsum));
44             max_sum = max(max_sum, x->rsum+y->lsum);
45             return this;
46         }
47         node *pull() {

```

```

48     size = 1;
49     max_sum = sum = lsum = rsum = key;
50     for (int i = 0; i < 2; i++) {
51         if (!s[i]) continue;
52         size += s[i]->size;
53     }
54     return merge_sum(node(*this).merge_sum(s[0], this), s[1]);
55 }
56 };
57 node *give(node *t = 0) {
58     static node *top = 0;
59     static int size = 1;
60     if (t) t->s[1] = top, top = t;
61     else {
62         if (!top) {
63             top = new node[size <= 1];
64             for (int i = 0; i < size - 1; i++)
65                 top[i].s[1] = top + i + 1;
66             top[size - 1].s[1] = 0;
67         }
68         t = top, top = top->s[1];
69     }
70     return t;
71 }
72 node *make(int key) {
73     node t = {{0}, 1, rand()*rand(), key, 0, key, key, key, 0};
74     return &(*give() = t);
75 }
76 void drop(node *&t) {
77     if (!t) return;
78     drop(t->s[0]), drop(t->s[1]);
79     give(t), t = 0;
80 }
81 void merge(node *x, node *y, node *&t) {
82     if (!x || !y) t = x? x: y;
83     else if (x->weight < y->weight)
84         x->push(), merge(x->s[1], y, x->s[1]), t = x->pull();
85     else y->push(), merge(x, y->s[0], y->s[0]), t = y->pull();
86 }
87 void split(node *t, int k, node *&x, node *&y) {
88     if (!k) x = 0, y = t;
89     else if (t->size == k) x = t, y = 0;
90     else if (k < t->rank())
91         y = t->push(), split(t->s[0], k, x, y->s[0]), y->pull();
92     else x = t->push(), split(t->s[1], k - t->rank(), x->s[1], y), x->pull();
93 }
94 void slice(node *&t, int l = -1, int r = -1) {
95     static node *a, *b;
96     if (~l) split(t, l - 1, a, b), split(b, r - l + 1, t, b);
97     else merge(t, b, b), merge(a, b, t);
98 }
99 void show(node *t) {
100     if (!t) return;
101     t->push();
102     show(t->s[0]);
103     printf("%2d", t->key);
104     show(t->s[1]);
105     t->pull();
106 }
107 int ask_sum(node *t) {

```



```

108     return t? t->sum: 0;
109 }
110 int ask_max_sum(node *t) {
111     return t? t->max_sum: 0;
112 }
113 };

```

7.4 Link-Cut Tree

```

1  /* Link-Cut-Tree
2  * */
3  template<int N> struct lct_t {
4      struct node {
5          node *s[2], *p;
6          int sz, rev, w, mx, at;
7          node *sets(int b, node *x) {
8              if (s[b] == x) x->p = this;
9              return this;
10         }
11         bool root() {
12             return !p || !(p->s[0] == this || p->s[1] == this);
13         }
14         bool which() {
15             return p->s[1] == this;
16         }
17         node *set() {
18             swap(s[0], s[1]);
19             rev ^= 1;
20             return this;
21         }
22         node *cover(int d) {
23             w += d;
24             mx += d;
25             at += d;
26             return this;
27         }
28         node *push() {
29             if (at) {
30                 for (int i = 0; i < 2; i++)
31                     if (s[i]) s[i]->cover(at);
32                 at = 0;
33             }
34             if (rev) {
35                 for (int i = 0; i < 2; i++)
36                     if (s[i]) s[i]->set();
37                 rev = 0;
38             }
39             return this;
40         }
41         node *pull() {
42             sz = 1;
43             mx = w;
44             for (int i = 0; i < 2; i++)
45                 if (s[i]) {
46                     sz += s[i]->sz;
47                     mx = max(mx, s[i]->mx);
48                 }
49             return this;
50         }

```

```

51 node *spin() {
52     node *y = p->push();
53     int b = push()->which();
54     y->sets(b, s[!b])->pull();
55     if (y->root()) p = y->p;
56     else y->p->sets(y->which(), this);
57     return sets(!b, y);
58 }
59 node *splay() {
60     for ( ; !root(); )
61         if (p->root()) spin();
62         else {
63             if (which() == p->which()) p->spin();
64             else spin();
65             spin();
66         }
67     return pull();
68 }
69 node *end(int b) {
70     node *x = this;
71     for ( ; x->push()->s[b]; x = x->s[b]) ;
72     return x;
73 }
74 } lct[N], *top;
75 void init() {
76     top = lct;
77 }
78 node *make(int w) {
79     *top = (node){0, 0, 0, 1, 0, w, w};
80     return top++;
81 }
82 node *access(node *x, int o = 0, int d = 0) {
83     static node rv;
84     for (node *y = x, *z = 0; y; z = y, y = y->p) {
85         y->splay()->push();
86         if (!y->p) {
87             if (o == 1) {
88                 y->w += d;
89                 if (y->s[1]) y->s[1]->cover(d);
90                 if (z) z->cover(d);
91             } else if (o == 2) {
92                 int mx = y->w;
93                 if (y->s[1]) mx = max(mx, y->s[1]->mx);
94                 if (z) mx = max(mx, z->mx);
95                 rv.mx = mx;
96                 return &rv;
97             }
98         }
99         y->sets(1, z)->pull();
100     }
101     return x->splay();
102 }
103 node *join(node *x, node *y) {
104     return x->p = y;
105 }
106 node *cut(node *x) {
107     if (access(x)->s[0]) x->s[0]->p = 0;
108     x->s[0] = 0;
109     return x;
110 }

```

```

111 node *find(node *x) {
112     return access(x)->end(0);
113 }
114 node *rooting(node *x) {
115     return access(x)->set();
116 }
117 node *cover(node *x, node *y, int w) {
118     access(x);
119     access(y, 1, w);
120     return x;
121 }
122 int ask(node *x, node *y) {
123     access(x);
124     return access(y, 2)->mx;
125 }
126 };

```

7.5 Functional Segment

```

1  /* Functional.Segment
2  * */
3  template<int N> struct fs_t {
4      struct node {
5          int l, r, sm;
6          node *ls, *rs;
7          int m() {
8              return l+r>>1;
9          }
10     } s[N*20], *top;
11     void init() {
12         top = s;
13     }
14     node *phi(int l, int r) {
15         node *x = top++, t = {l, r, 0};
16         *x = t;
17         if (l < r) {
18             x->ls = phi(l, x->m());
19             x->rs = phi(x->m()+1, r);
20         }
21         return x;
22     }
23     node *put(int k, node *y) {
24         node *x = top++;
25         *x = *y;
26         x->sm++;
27         if (x->l < y->r) {
28             if (k <= x->m()) x->ls = put(k, y->ls);
29             else x->rs = put(k, y->rs);
30         }
31         return x;
32     }
33     int ask(int l, int r, node *x, node *y) {
34         int rv = 0;
35         if (l <= x->l && x->r <= r) rv = x->sm-y->sm;
36         else {
37             if (l <= x->m()) rv += ask(l, r, x->ls, y->ls);
38             if (x->m() < r) rv += ask(l, r, x->rs, y->rs);
39         }
40         return rv;

```

```

41     }
42 };

```

7.6 Functional Trie

```

1  /* Functional_Trie
2  */
3  template<int N, int D> struct ftrie_t {
4      struct node {
5          node *s[2];
6          int c[2];
7      } s[D*N+D], *top, *phi;
8      void init() {
9          top = s;
10         phi = top++;
11         phi->c[0] = phi->c[1] = 0;
12         phi->s[0] = phi->s[1] = phi;
13     }
14     node *put(int k, node *y, int d = D) {
15         if (!d) return 0;
16         node *x = top++;
17         *x = *y;
18         int i = k>>(d-1)&1;
19         x->c[i]++;
20         x->s[i] = put(k, y->s[i], d-1);
21         return x;
22     }
23     int ask(int k, node *x, node *y, int d = D) {
24         if (!d) return 0;
25         int i = k>>(d-1)&1;
26         if (x->c[!i]-y->c[!i])
27             return (1<<d-1)+ask(k, x->s[!i], y->s[!i], d-1);
28         return ask(k, x->s[i], y->s[i], d-1);
29     }
30 };

```

7.7 Lefist Tree

```

1  /* Lefist_Tree
2  */
3  template<int N> struct lefist_t {
4      struct node {
5          node *l, *r;
6          int k, d;
7      } s[N], *top;
8      void init() {
9          top = s;
10     }
11     node *make(int k) {
12         node *x = top++, t = {0, 0, k, 0};
13         *x = t;
14         return x;
15     }
16     node *merge(node *x, node *y) {
17         if (!x) return y;
18         if (!y) return x;
19         if (x->k < y->k) swap(x, y);
20         x->r = merge(x->r, y);

```

```

21     if (!x->l || x->r && x->l->d < x->r->d) swap(x->l, x->r);
22     if (x->r) x->d = x->r->d+1;
23     return x;
24 }
25 node *drop(node *x) {
26     return merge(x->l, x->r);
27 }
28 };

```

8 geometry

8.1 Float Compare Functions

```

1  /* Float_Compare_Functions
2  * */
3  struct fc_t {
4      double eps;
5      fc_t() {
6          eps = 1e-8;
7      }
8      bool e(double lhs, double rhs) {
9          return abs(lhs-rhs) < eps;
10     }
11     bool l(double lhs, double rhs) {
12         return lhs+eps < rhs;
13     }
14     bool g(double lhs, double rhs) {
15         return lhs-eps > rhs;
16     }
17 } fc;

```

8.2 2D point

```

1  /* 2D_point
2  * */
3  struct pt_t {
4      double x, y;
5      pt_t(double _x = 0, double _y = 0) {
6          x = _x, y = _y;
7      }
8      double operator [] (int b) {
9          return b? b < 2? abs(x)+abs(y): x*x+y*y: sqrt(x*x+y*y);
10     }
11     friend pt_t operator + (const pt_t &lhs, const pt_t &rhs) {
12         return pt_t(lhs.x+rhs.x, lhs.y+rhs.y);
13     }
14     friend pt_t operator - (const pt_t &lhs, const pt_t &rhs) {
15         return pt_t(lhs.x-rhs.x, lhs.y-rhs.y);
16     }
17     friend double operator * (const pt_t &lhs, const pt_t &rhs) {
18         return lhs.x*rhs.x+lhs.y*rhs.y;
19     }
20     friend double operator % (const pt_t &lhs, const pt_t &rhs) {
21         return lhs.x*rhs.y-lhs.y*rhs.x;
22     }
23     pt_t &input() {
24         scanf("%lf%lf", &x, &y);
25         return *this;

```

```

26     }
27 };

```

8.3 Angle Sort

```

1  /* Angle_Sort
2  * */
3  struct asort_t {
4      bool cmpl(pt_t lhs, pt_t rhs) {
5          return fc.l(lhs.y, rhs.y) || (fc.e(lhs.y, rhs.y) && fc.l(lhs.x, rhs.x));
6      }
7      static pt_t o;
8      static bool cmp(pt_t lhs, pt_t rhs) {
9          double c = (lhs-o)%(rhs-o);
10         if (!fc.e(c, 0.0)) return fc.g(c, 0.0);
11         return fc.g((lhs-o)[1], (rhs-o)[1]);
12     }
13     void operator () (vector<pt_t> &p) {
14         int mn = 0;
15         for (int i = 0; i < p.size(); i++)
16             if (cmpl(p[i], p[mn])) mn = i;
17         swap(p[0], p[mn]);
18         o = p[0];
19         sort(p.begin()+1, p.end(), cmp);
20     }
21 } asort;
22 pt_t asort_t::o;

```

8.4 Graham Scan

```

1  /* Graham_Scan
2  * */
3  struct graham_t {
4      vector<pt_t> p;
5      double l;
6      graham_t(vector<pt_t> &ps) {
7          asort(p = ps);
8          vector<pt_t> s(p.begin(), p.begin()+2);
9          ps.clear();
10         for (int i = 2; i < p.size(); i++) {
11             for (; fc.g((s[s.size()-2]-s.back())%(p[i]-s.back()), 0.0); )
12                 ps.push_back(s.back()), s.pop_back();
13             s.push_back(p[i]);
14         }
15         p = s;
16         for (int i = 1 = 0; i < p.size(); i++)
17             l += (p[(i+1)%p.size()-p[i]][0];
18     }
19 };

```

9 graph

9.1 Graph

```

1  /* Graph
2  * */

```

```

3 template<int N> struct graph_t {
4     struct edge_t {
5         int v, to;
6     };
7     vector<edge_t> E;
8     int L[N];
9     void init() {
10         E.clear();
11         memset(L, -1, sizeof(L));
12     }
13     void add(int u, int v) {
14         edge_t t = {v, L[u]};
15         L[u] = E.size();
16         E.push_back(t);
17     }
18 };

```

9.2 Shortest Path Algorithm

```

1  /* Shortest_Path_Algorithm
2  * */
3  template<class edge_t, int N> struct spfa_t {
4      int d[N], b[N], c[N], s[N], mx[N];
5      int operator () (vector<edge_t> &E, int *L, int n, int u) {
6          memset(d, 0x7f, sizeof(d));
7          memset(b, 0, sizeof(b));
8          memset(c, 0, sizeof(c));
9          d[s[s[0] = 1] = u] = 0;
10         b[u] = c[u] = 1;
11         for ( ; s[0]; ) {
12             b[u = s[s[0]--]] = 0;
13             for (int e = L[u]; ~e; e = E[e].to) {
14                 int v = E[e].v, w = E[e].w;
15                 if (d[v]-w > d[u]) {
16                     d[v] = d[u]+w;
17                     if (!b[v]) {
18                         if ((c[v] += b[v] = 1) > n) return 0;
19                         s[++s[0]] = v;
20                     }
21                 }
22             }
23         }
24         return 1;
25     }
26     struct node {
27         int u, w;
28         node (int _u = 0, int _w = 0): u(_u), w(_w) {}
29         friend bool operator < (const node &lhs, const node &rhs) {
30             return lhs.w > rhs.w;
31         }
32     };
33     void operator () (vector<edge_t> &E, int *L, int u) {
34         memset(d, 0x7f, sizeof(d));
35         memset(b, 0, sizeof(b));
36         priority_queue<node> q;
37         for (q.push(node(u, d[u] = 0)); q.size(); ) {
38             u = q.top().u, q.pop();
39             if (b[u]++) continue;
40             for (int e = L[u]; ~e; e = E[e].to) {

```

```

41         int v = E[e].v, w = E[e].w;
42         if (b[u] && d[v]-w > d[u])
43             q.push(node(v, d[v] = d[u]+w));
44     }
45 }
46 }
47 };

```

9.3 Bipartite Graph match

```

1  /* Bipartite_Graph_match
2  * */
3  template<class edge_t, int N> struct bgm_t {
4      int vis[N], pre[N], lma[N], rma[N];
5      bool bfs(vector<edge_t> &E, int *L, int u) {
6          vector<int> q(1, u);
7          memset(vis, 0, sizeof(vis));
8          memset(pre, -1, sizeof(pre));
9          for (int h = 0; h < q.size(); h++) {
10             u = q[h];
11             for (int e = L[u]; ~e; e = E[e].to) {
12                 int v = E[e].v;
13                 if (!vis[v]) {
14                     vis[v] = 1;
15                     if (rma[v] == -1) {
16                         for (; ~u; ) {
17                             rma[v] = u;
18                             swap(v, lma[u]);
19                             u = pre[u];
20                         }
21                         return 1;
22                     } else {
23                         pre[rma[v]] = u;
24                         q.push_back(rma[v]);
25                     }
26                 }
27             }
28         }
29         return 0;
30     }
31     int operator () (vector<edge_t> &E, int *L, int V) {
32         int mmat = 0;
33         memset(lma, -1, sizeof(lma));
34         memset(rma, -1, sizeof(rma));
35         for (int u = 0; u < V; u++)
36             mmat += bfs(E, L, u);
37         return mmat;
38     }
39 };

```

9.4 General Graph match

```

1  /* General_Graph_match
2  * */
3  template<int N> struct blossom_t {
4      deque<int> Q;
5      int n;
6      bool g[N][N], inque[N], inblossom[N];

```



```

7  int match[N], pre[N], base[N];
8  int findancestor(int u, int v){
9      bool inpath[N]={false};
10     while(1){
11         u=base[u];
12         inpath[u]=true;
13         if(match[u]==-1)break;
14         u=pre[match[u]];
15     }
16     while(1){
17         v=base[v];
18         if(inpath[v])return v;
19         v=pre[match[v]];
20     }
21 }
22 void reset(int u, int anc){
23     while(u!=anc){
24         int v=match[u];
25         inblossom[base[u]]=1;
26         inblossom[base[v]]=1;
27         v=pre[v];
28         if(base[v]!=anc)pre[v]=match[u];
29         u=v;
30     }
31 }
32 void contract(int u, int v, int n){
33     int anc=findancestor(u, v);
34     //SET(inblossom, 0);
35     memset(inblossom, 0, sizeof(inblossom));
36     reset(u, anc); reset(v, anc);
37     if(base[u]!=anc)pre[u]=v;
38     if(base[v]!=anc)pre[v]=u;
39     for(int i=1; i<=n; i++){
40         if(inblossom[base[i]]){
41             base[i]=anc;
42             if(!inque[i]){
43                 Q.push_back(i);
44                 inque[i]=1;
45             }
46         }
47     }
48     bool dfs(int S, int n){
49         for(int i=0; i<=n; i++)pre[i]=-1, inque[i]=0, base[i]=i;
50         Q.clear(); Q.push_back(S); inque[S]=1;
51         while(!Q.empty()){
52             int u=Q.front(); Q.pop_front();
53             for(int v=1; v<=n; v++){
54                 if(g[u][v]&&base[v]!=base[u]&&match[u]!=v){
55                     if(v==S || (match[v]!=-1&&pre[match[v]]!=-1))contract(u, v, n);
56                     else if(pre[v]==-1){
57                         pre[v]=u;
58                         if(match[v]!=-1)Q.push_back(match[v]), inque[match[v]]=1;
59                     }
60                     else{
61                         u=v;
62                         while(u!=-1){
63                             v=pre[u];
64                             int w=match[v];
65                             match[u]=v;
66                             match[v]=u;
67                             u=w;

```

```

67         }
68         return true;
69     }
70 }
71 }
72 }
73 }
74 return false;
75 }
76 void init(int n) {
77     this->n = n; memset(match, -1, sizeof(match));
78     memset(g, 0, sizeof(g));
79 }
80 void addEdge(int a, int b) {
81     ++a;
82     ++b;
83     g[a][b] = g[b][a] = 1;
84 }
85 int gao() {
86     int ans = 0;
87     for (int i = 1; i <= n; ++i) {
88         if (match[i] == -1 && dfs(i, n)) {
89             ++ans;
90         }
91     }
92     return ans;
93 }
94 };

```

9.5 Dancing Link

```

1  /* Dancing_Link
2  * */
3  template<int N, int M> struct dancing {
4  #define dfor(c, a, b) for (int c = a[b]; c != b; c = a[c])
5      static const int row_size = N, column_size = M,
6                      total_size = row_size * column_size;
7      typedef int row[row_size],
8                column[column_size],
9                total[total_size];
10     total l, r, u, d, in_column;
11     column s;
12     int index, current_row, row_head;
13     void init(int n)
14     {
15         index = ++n;
16         for (int i = 0; i < n; i++) {
17             l[i] = (i - 1 + n) % n;
18             r[i] = (i + 1) % n;
19             u[i] = d[i] = i;
20         }
21         current_row = 0;
22         memset(s, 0, sizeof(s));
23     }
24     void push(int i, int j)
25     {
26         i++; j++;
27         if (current_row < i) {
28             row_head = l[index] = r[index] = index;

```

```

29     current_row = i;
30 }
31 l[index] = l[row_head]; r[index] = row_head;
32 r[l[row_head]] = index; l[row_head] = index;
33 u[index] = u[j]; d[index] = j;
34 d[u[j]] = index; u[j] = index;
35 s[j]++;
36 in_column[index++] = j;
37 }
38 void exactly_remove(int c)
39 {
40     l[r[c]] = l[c];
41     r[l[c]] = r[c];
42     dfor(i, d, c) {
43         dfor(j, r, i) {
44             u[d[j]] = u[j];
45             d[u[j]] = d[j];
46             s[in_column[j]]--;
47         }
48     }
49 }
50 void exactly_resume(int c)
51 {
52     dfor(i, u, c) {
53         dfor(j, l, i) {
54             s[in_column[j]]++;
55             d[u[j]] = u[d[j]] = j;
56         }
57     }
58     r[l[c]] = l[r[c]] = c;
59 }
60 bool exactly_dance(int step = 0)
61 {
62     if (!r[0]) {
63         return 1;
64     }
65     int x = r[0];
66     dfor(i, r, 0) {
67         if (s[i] < s[x]) {
68             x = i;
69         }
70     }
71     exactly_remove(x);
72     dfor(i, d, x) {
73         dfor(j, r, i) {
74             exactly_remove(in_column[j]);
75         }
76         if (exactly_dance(step + 1)) {
77             return 1;
78         }
79         dfor(j, l, i) {
80             exactly_resume(in_column[j]);
81         }
82     }
83     exactly_resume(x);
84     return 0;
85 }
86 int limit;
87 void remove(int c)
88 {

```

```

89     dfor(i, d, c) {
90         l[r[i]] = l[i];
91         r[l[i]] = r[i];
92     }
93 }
94 void resume(int c)
95 {
96     dfor(i, u, c) {
97         r[l[i]] = l[r[i]] = i;
98     }
99 }
100 bool dance(int step = 0)
101 {
102     if (limit <= step + heuristic()) {
103         return 0;
104     }
105     if (!r[0]) {
106         limit = min(limit, step);
107         return 1;
108     }
109     int x = r[0];
110     dfor(i, r, 0) {
111         if (s[i] < s[x]) {
112             x = i;
113         }
114     }
115     dfor(i, d, x) {
116         remove(i);
117         dfor(j, r, i) {
118             remove(j);
119         }
120         if (dance(step + 1)) {
121             return 1;
122         }
123         dfor(j, l, i) {
124             resume(j);
125         }
126         resume(i);
127     }
128     return 0;
129 }
130 int heuristic()
131 {
132     int rv = 0;
133     column visit = {0};
134     dfor(c, r, 0) {
135         if (!visit[c]) {
136             rv++;
137             visit[c] = 1;
138             dfor(i, d, c) {
139                 dfor(j, r, i) {
140                     visit[in_column[j]] = 1;
141                 }
142             }
143         }
144     }
145     return rv;
146 }
147 int dfs()
148 {

```

```

149     for (limit = heuristic(); !dance(); limit++) {}
150     return limit;
151 }
152 #undef dfor
153 };

```

9.6 Directed Minimum Spanning Tree

```

1  /* Directed_Minimum_Spanning_Tree
2  * */
3  template<int N> struct dmst_t {
4      struct edge_t {
5          int u, v, w;
6      };
7      vector<edge_t> E;
8      static const int inf = 0x7f7f7f7f;
9      int n, ine[N], pre[N], id[N], vis[N];
10     void init(int _n) {
11         n = _n;
12         E.clear();
13     }
14     void add(int u, int v, int w) {
15         edge_t t = {u, v, w};
16         E.push_back(t);
17     }
18     int operator () (int rt) {
19         int i, u, v, w, tn = n+1, index, rv = 0;
20         for ( ; ; ) {
21             fill(ine, ine+tn, inf);
22             for (i = 0; i < E.size(); i++) {
23                 u = E[i].u; v = E[i].v; w = E[i].w;
24                 if (u != v && w < ine[v]) {
25                     pre[v] = u;
26                     ine[v] = w;
27                 }
28             }
29             for (u = 0; u < tn; u++) {
30                 if (u == rt) continue;
31                 if (ine[u] == inf)
32                     return -1;
33             }
34             index = 0;
35             fill(id, id + tn, -1);
36             fill(vis, vis + tn, -1);
37             ine[rt] = 0;
38             for (u = 0; u < tn; u++) {
39                 rv += ine[v = u];
40                 for ( ; v != rt && vis[v] != u && id[v] == -1; ) {
41                     vis[v] = u;
42                     v = pre[v];
43                 }
44                 if (v != rt && id[v] == -1) {
45                     for (i = pre[v]; i != v; i = pre[i]) id[i] = index;
46                     id[v] = index++;
47                 }
48             }
49             if (index == 0) break;
50             for (u = 0; u < tn; u++)
51                 if (id[u] == -1) id[u] = index++;

```

```

52     for (i = 0; i < E.size(); i++) {
53         v = E[i].v;
54         E[i].u = id[E[i].u];
55         E[i].v = id[E[i].v];
56         if (E[i].u != E[i].v) E[i].w -= ine[v];
57     }
58     tn = index;
59     rt = id[rt];
60 }
61 return rv;
62 }
63 };

```

9.7 Spfa Cost Stream

```

1  /* Spfa_Cost_Stream
2  * */
3  template<class edge_t, int N> struct ek_t {
4      vector<edge_t> E;
5      static const int inf = 0x7f7f7f7f;
6      int n, *L, src, snk, dis[N], ra[N], inq[N];
7      int spfa(int u) {
8          vector<int> q(1, u);
9          memset(dis, 0x3f, sizeof(int)*n);
10         memset(ra, -1, sizeof(int)*n);
11         memset(inq, 0, sizeof(int)*n);
12         dis[u] = 0;
13         inq[u] = 1;
14         for (int h = 0; h < q.size(); h++) {
15             u = q[h], inq[u] = 0;
16             for (int e = L[u]; ~e; e = E[e].to) {
17                 int v = E[e].v, w = E[e].w, c = E[e].c;
18                 if (w && dis[v] > dis[u]+c) {
19                     dis[v] = dis[u]+c;
20                     ra[v] = e^1;
21                     if (inq[v]) continue;
22                     inq[v] = 1;
23                     q.push_back(v);
24                 }
25             }
26         }
27         return ra[snk] != -1;
28     }
29     int operator () (vector<edge_t> _E, int *_L, int _n, int _src, int _snk) {
30         E = _E, L = _L, n = _n;
31         src = _src, snk = _snk;
32         int mmf = 0;
33         for (; spfa(src); ) {
34             int mf = inf;
35             for (int e = ra[snk]; ~e; e = ra[E[e].v])
36                 mf = min(mf, E[e^1].w);
37             for (int e = ra[snk]; ~e; e = ra[E[e].v])
38                 E[e].w += mf, E[e^1].w -= mf;
39             mmf += dis[snk]*mf;
40         }
41         return mmf;
42     }
43 };

```

9.8 KM Maximum perfect match

```

1  /* KM_Maximum_perfect_match
2  * Notice that we could use this, when left side has the same amount
3  * as right side. (perfect match)
4  * If the situation above doesn't be hold, Cost-Flow algorithm is recommended.
5  * */
6  template<class edge_t, int N> struct km_t {
7      vector<edge_t> E;
8      static const int inf = 0x7f7f7f7f;
9      typedef int kmia_t[N];
10     kmia_t mat, lta, rta, sla, lvi, rvi;
11     int n, *L;
12     int dfs(int u) {
13         lvi[u] = 1;
14         for (int e = L[u]; ~e; e = E[e].to) {
15             int v = E[e].v, w = E[e].w;
16             if (!rvi[v]) {
17                 int t = lta[u] + rta[v] - w;
18                 if (!t) {
19                     rvi[v] = 1;
20                     if (mat[v] == -1 || dfs(mat[v])) {
21                         mat[v] = u;
22                         return 1;
23                     }
24                 } else if (t < sla[v]) sla[v] = t;
25             }
26         }
27         return 0;
28     }
29     int operator () (vector<edge_t> &E, int _L[N], int _n) {
30         E = _E, L = _L, n = _n;
31         memset(lta, 0, sizeof(lta));
32         memset(rta, 0, sizeof(rta));
33         memset(mat, -1, sizeof(mat));
34         for (int u = 0; u < n; u++)
35             for (int e = L[u]; ~e; e = E[e].to)
36                 if (lta[u] < E[e].w) lta[u] = E[e].w;
37         for (int u = 0; u < n; u++) {
38             for (int e = L[u]; ~e; e = E[e].to) sla[E[e].v] = inf;
39             for ( ; ; ) {
40                 memset(lvi, 0, sizeof(lvi));
41                 memset(rvi, 0, sizeof(rvi));
42                 if (dfs(u)) break;
43                 int mn = inf;
44                 for (int v = 0; v < n; v++)
45                     if (!rvi[v]) mn = min(mn, sla[v]);
46                 for (int v = 0; v < n; v++) {
47                     if (lvi[v]) lta[v] -= mn;
48                     if (rvi[v]) rta[v] += mn;
49                     else sla[v] -= mn;
50                 }
51             }
52         }
53         int rv = 0;
54         for (int v = 0; v < n; v++) if (~mat[v])
55             for (int e = L[mat[v]]; ~e; e = E[e].to)
56                 if (E[e].v == v) {
57                     rv += E[e].w;
58                     break;

```

```

59     }
60     return rv;
61 }
62 };

```

9.9 Doubling LCA

```

1  /* Doubling_LCA
2  * */
3  template<class edge_t, int N> struct lca_t {
4      static const int M = 16;
5      int d[N], a[N][M], p[1<<M];
6      void operator () (vector<edge_t> E, int *L, int u) {
7          vector<int> q(1, u);
8          memset(a, -1, sizeof(a));
9          for (int h = d[u] = 0; h < q.size(); h++) {
10             u = q[h];
11             for (int i = 1; i < M; i++)
12                 if (~a[u][i-1]) a[u][i] = a[a[u][i-1]][i-1];
13             for (int e = L[u]; ~e; e = E[e].to) {
14                 int v = E[e].v;
15                 if (v == a[u][0]) continue;
16                 d[v] = d[u] + 1;
17                 a[v][0] = u;
18                 q.push_back(v);
19             }
20         }
21         for (int i = 0; i < M; i++) p[1<<i] = i;
22     }
23     int skip(int u, int x) {
24         for (; x; x -= -x&x) u = a[u][p[-x&x]];
25         return u;
26     }
27     int operator () (int u, int v) {
28         if (d[u] < d[v]) swap(u, v);
29         u = skip(u, d[u]-d[v]);
30         if (u == v) return u;
31         for (int i = M-1; ~i && a[u][0] != a[v][0]; i--)
32             if (~a[u][i] && a[u][i] != a[v][i])
33                 u = a[u][i], v = a[v][i];
34         return a[u][0];
35     }
36 };

```

9.10 Shortest Augment Path

```

1  /* Shortest_Augment_Path
2  * */
3  template<class edge_t, int N> struct sap_t {
4      int dis[N], gap[N], _L[N], se[N];
5      int operator () (vector<edge_t> &E, int *L, int V, int src, int snk) {
6          int mxl = 0, te = 0;
7          memcpy(_L, L, sizeof(L));
8          memset(dis, -1, sizeof(dis));
9          memset(gap, 0, sizeof(gap));
10         gap[dis[snk] = 0] = 1;
11         vector<int> q(1, snk);
12         for (int h = 0; h < q.size(); h++)

```



```

13     for (int i = L[q[h]]; i != -1; i = E[i].to)
14         if (E[i].w && dis[E[i].v] < 0) {
15             gap[dis[E[i].v] = dis[q[h]]+1]++;
16             q.push_back(E[i].v);
17         }
18     for (int u = src; dis[src] < V; ) {
19         for (int &i = _L[u]; i != -1; i = E[i].to)
20             if (E[i].w && dis[u] == dis[E[i].v] + 1) break;
21         if (_L[u] != -1) {
22             u = E[se[te++]] = _L[u].v;
23             if (u == snk) {
24                 int _i = 0, mf = 0x7fffffff;
25                 for (int i = 0; i < te; i++)
26                     if (E[se[i]].w < mf) {
27                         mf = E[se[i]].w;
28                         _i = i;
29                     }
30                 for (int i = 0; i < te; i++) {
31                     E[se[i]].w -= mf;
32                     E[se[i]^1].w += mf;
33                 }
34                 mxf += mf;
35                 u = E[se[te = _i]^1].v;
36             }
37             continue;
38         }
39         int md = V;
40         _L[u] = -1;
41         for (int i = L[u]; i != -1; i = E[i].to)
42             if (E[i].w && dis[E[i].v] < md) {
43                 md = dis[E[i].v];
44                 _L[u] = i;
45             }
46         if (!--gap[dis[u]]) break;
47         gap[dis[u] = md+1]++;
48         if (u != src) u = E[se[te--1]^1].v;
49     }
50     return mxf;
51 }
52 };

```

9.11 ZKW Cost Stream

```

1  /* ZKW_Cost_Stream
2   * */
3  template<class edge_t, int N> struct zkw_t {
4      vector<edge_t> E;
5      static const int inf = 0x7f7f7f7f;
6      int n, src, snk, mc, mf, dis[N], *L;
7      int ap(int u, int f) {
8          if (u == snk) {
9              mc += dis*f;
10             mf += f;
11             return f;
12         }
13         vis[u] = 1;
14         int rf = f;
15         for (int e = L[u]; e > -1; e = E[e].to)
16             if (!vis[E[e].v] && E[e].w && !E[e].c) {

```

```

17         int df = ap(E[e].v, min(rf, E[e].w));
18         E[e].w -= df;
19         E[e^1].w += df;
20         rf -= df;
21         if (!rf) return f;
22     }
23     return f-rf;
24 }
25 int ml() {
26     int md = inf;
27     for (int u = 0; u < n; u++) if (vis[u])
28         for (int e = L[u]; ~e; e = E[e].to)
29             if (!vis[E[e].v] && E[e].w)
30                 md = min(md, E[e].c);
31     if (md == inf) return 0;
32     for (int u = 0; u < n; u++) if (vis[u])
33         for (int e = L[u]; ~e; e = E[e].to) {
34             E[e].c -= md;
35             E[e^1].c += md;
36         }
37     dis += md;
38     return 1;
39 }
40 int operator () (vector<edge_t> &E, int *L, int _n, int _src, int _snk) {
41     E = _E, L = _L, n = _n;
42     src = _src, snk = _snk;
43     mf = mc = dis = 0;
44     for ( ; ; ) {
45         for ( ; ; ) {
46             memset(vis, 0, sizeof vis);
47             if (!ap(src, inf)) break;
48         }
49         if (!ml()) break;
50     }
51     return mc;
52 }
53 };

```

10 graph test

10.1 Graph

```

1  /* Graph
2  * */
3  struct graph_t {
4      struct edge_t {
5          int v, to;
6      };
7      vector<edge_t> e;
8      vector<int> h;
9      edge_t &operator [] (int x) {
10         return e[x];
11     }
12     int &operator () (int x) {
13         return h[x];
14     }
15     int size() {
16         return h.size();
17     }

```

```

18 void init(int n) {
19     e.clear(), h.resize(n);
20     fill(h.begin(), h.end(), -1);
21 }
22 void add(int u, int v) {
23     edge_t t = {v, h[u]};
24     h[u] = e.size();
25     e.push_back(t);
26 }
27 void badd(int u, int v) {
28     add(u, v), add(v, u);
29 }
30 };

```

10.2 Shortest Augment Path

```

1  /* Shortest_Augment_Path
2  * */
3  template<class graph_t> struct sap_t {
4      vector<int> dis, gap;
5      int dfs(graph_t &g, int src, int snk, int u, int f = ~1u>>1) {
6          if (u == snk) return f;
7          int rf = f, md = g.size() - 1;
8          for (int e = g(u); ~e; e = g[e].to) {
9              int v = g[e].v, w = g[e].w;
10             if (!w) continue;
11             md = min(md, dis[v]);
12             if (dis[u] != dis[v] + 1) continue;
13             int df = dfs(g, src, snk, v, min(w, f));
14             g[e].w -= df, g[e ^ 1].w += df;
15             if (gap[src] == g.size() || !(rf == df)) return f;
16         }
17         if (!--gap[dis[u]]) gap[src] = g.size();
18         else gap[dis[u] = md + 1]++;
19         return f - rf;
20     }
21     int operator () (graph_t &g, int src, int snk) {
22         dis.clear(), gap.clear();
23         for (int i = g.size() << 1; i; i--)
24             dis.push_back(-1), gap.push_back(0);
25         vector<int> q(gap[dis[snk] = 0] = 1, snk);
26         for (int h = 0; h < q.size(); h++)
27             for (int e = g(q[h]); ~e; e = g[e].to)
28                 if (g[e ^ 1].w && !~dis[g[e].v])
29                     gap[dis[g[e].v] = dis[q[h]] + 1]++, q.push_back(g[e].v);
30         for (int i = 0; i < g.size(); i++)
31             if (!~dis[i]) gap[dis[i] = 0]++;
32         int result = 0;
33         for (; gap[src] < g.size(); ) result += dfs(g, src, snk, src);
34         return result;
35     }
36 };

```

10.3 Strong Connected Component

```

1  /* Strong_Connected_Component
2  * */
3  template<class graph_t> struct scc_t {

```

```

4  int time, cc;
5  vector<int> dfn, low, in, pushed, st;
6  void dfs(graph_t &g, int u) {
7      st.push_back(u), pushed[u] = 1;
8      dfn[u] = low[u] = time++;
9      for (int e = g(u); ~e; e = g[e].to) {
10         int v = g[e].v;
11         if (!~dfn[v]) dfs(g, v), low[u] = min(low[u], low[v]);
12         else if (pushed[v]) low[u] = min(low[u], dfn[v]);
13     }
14     if (dfn[u] == low[u]) {
15         for ( ; ; ) {
16             in[u = st.back()] = cc;
17             st.pop_back(), pushed[u] = 0;
18             if (dfn[u] == low[u]) break;
19         }
20         cc++;
21     }
22 }
23 void operator () (graph_t &g) {
24     dfn.clear(), low.clear(), in.clear(), pushed.clear(), st.clear();
25     for (int i = 0; i < g.size(); i++)
26         dfn.push_back(-1), low.push_back(-1), in.push_back(-1), pushed.push_back(0);
27     for (int u = time = cc = 0; u < g.size(); u++)
28         if (!~dfn[u]) dfs(g, u);
29 }
30 };

```

10.4 Heavy Light Division

```

1  /* Heavy_Light_Division
2  * */
3  template<class graph_t, int N> struct hld_t {
4      typedef int ai_t[N];
5      ai_t d, sz, hb, fa, cl, in, id;
6      void link(int h) {
7          cl[h] = 1, in[h] = h, id[h] = 0;
8          for (int v = h; ~hb[v]; )
9              in[v = hb[v]] = h, id[v] = cl[h]++;
10     }
11     void go(graph_t &g, int u, int p = -1, int l = 0) {
12         d[u] = 1, sz[u] = 1, hb[u] = -1, fa[u] = p;
13         for (int e = g(u); ~e; e = g[e].to) {
14             int v = g[e].v;
15             if (v == p) continue;
16             go(g, v, u, l+1);
17             sz[u] += sz[v];
18             if (!~hb[u] || sz[hb[u]] < sz[v]) hb[u] = v;
19         }
20         for (int e = g(u); ~e; e = g[e].to)
21             if (g[e].v != p && g[e].v != hb[u]) link(g[e].v);
22         if (!~p) link(u);
23     }
24     void make(int *w, int n) {
25     }
26     int ask(int u, int v) {
27         int result;
28         for ( ; in[u]^in[v]; u = fa[in[u]]) {
29             if (d[in[u]] < d[in[v]]) swap(u, v);

```

```

30     }
31     if (id[u] > id[v]) swap(u, v);
32     return result;
33 }
34 };

```

10.5 Biconnected Component

```

1  /* Biconnected_Component
2  * */
3  struct bcc_t {
4      int time, cc, cut[N], dfn[N], low[N];
5      vector<int> in, st;
6      void dfs(graph_t &g, int u, int p = -1) {
7          int branch = 0;
8          dfn[u] = low[u] = time++;
9          for (int e = g(u); ~e; e = g[e].to) {
10             int v = g[e].v;
11             if (e == p || dfn[v] >= dfn[u]) continue;
12             st.push_back(e);
13             if (~dfn[v]) low[u] = min(low[u], dfn[v]);
14             else {
15                 branch++;
16                 dfs(g, v, e ^ 1);
17                 low[u] = min(low[u], low[v]);
18                 if (dfn[u] > low[v]) continue;
19                 for (cut[u] = 1; ; ) {
20                     int t = st.back();
21                     st.pop_back();
22                     in[t] = in[t ^ 1] = cc;
23                     if (t == e) break;
24                 }
25                 cc++;
26             }
27         }
28         if (!p && cut[u] && branch < 2) cut[u] = 0;
29     }
30     void operator () (graph_t &g) {
31         in.resize(g.e.size());
32         for (int u = 0; u < g.size(); u++)
33             dfn[u] = low[u] = -1, cut[u] = 0;
34         st.clear();
35         for (int u = time = cc = 0; u < g.size(); u++)
36             if (!dfn[u]) dfs(g, u);
37     }
38 };

```

10.6 Static Lowest Ancestor

```

1  /* Static_Lowest_Ancestor
2  * */
3  struct slca_t {
4      graph_t q;
5      vector<int> r, f, c, d;
6      void init(int n) {
7          q.init(n), f.resize(n), c.resize(n);
8          for (int i = 0; i < n; i++)
9              f[i] = i, c[i] = 0;

```

```

10     }
11     void add(int u, int v) {
12         r.push_back(-1);
13         q.badd(u, v);
14     }
15     int find(int x) {
16         if (x != f[x]) f[x] = find(f[x]);
17         return f[x];
18     }
19     void go(graph_t &g, int u, int p = -1) {
20         for (int e = g(u); ~e; e = g[e].to) {
21             if (e == p) continue;
22             go(g, g[e].v, e^1);
23             f[find(g[e].v)] = u;
24         }
25         c[u] = 1;
26         for (int e = q(u); ~e; e = q[e].to) {
27             if (!c[q[e].v]) continue;
28             r[e>>1] = find(q[e].v);
29         }
30     }
31 };

```