

南京大学 ACM-ICPC 集训队代码模版库



OS: Linux-4.15.0-45-generic-x86_64-with-Ubuntu-18.04-bionic
python: 2.7.15rc1
2019-02-24 23:23:44.509359, build 0016

Contents

1 General	3	5 Graph Theory	13
1.1 Code library checksum	3	5.1 Strongly connected component	13
1.2 Makefile	3	5.2 Vertex biconnected component	14
1.3 .vimrc	3	5.3 Cut vertices	15
1.4 Stack	3	5.4 Minimum spanning arborescence (Chu-Liu)	15
1.5 Template	3	5.5 Maximum flow (Dinic)	16
2 Miscellaneous Algorithms	4	5.6 Maximum cardinality bipartite matching (Hungarian)	17
2.1 2-SAT	4	5.7 Maximum matching of general graph (Edmond's blossom)	17
2.2 Knuth's optimization	4	5.8 Minimum cost maximum flow	18
2.3 Mo's algorithm	5	5.9 Global minimum cut (Stoer-Wagner)	19
3 String	5	5.10 Fast LCA	20
3.1 Knuth-Morris-Pratt algorithm	5	5.11 Heavy-light decomposition	20
3.2 Manacher algorithm	6	5.12 Centroid decomposition	21
3.3 Aho-corasick automaton	6	5.13 DSU on tree	21
3.4 Trie	7	6 Data Structures	22
3.5 Suffix array	7	6.1 Fenwick tree (point update range query)	22
3.6 Rolling hash	8	6.2 Fenwick tree (range update point query)	22
4 Math	8	6.3 Segment tree	23
4.1 Extended Euclidean algorithm and Chinese remainder theorem	8	6.4 Treap	24
4.2 Matrix powermod	9	6.5 Link/cut tree	25
4.3 Linear basis	9	6.6 Balanced binary search tree from pb_ds	26
4.4 Gauss elimination over finite field	9	6.7 Persistent segment tree, range k-th query	26
4.5 Berlekamp-Massey algorithm	10	6.8 Block list	27
4.6 Fast Walsh-Hadamard transform	10	6.9 Persistent block list	28
4.7 Fast fourier transform	11	6.10 Sparse table, range extremum query	29
4.8 Number theoretic transform	11	7 Geometrics	29
4.9 Sieve of Euler	12	7.1 2D geometric template	29
4.10 Sieve of Euler (General)	12	8 Appendices	31
4.11 Miller-Rabin primality test	13	8.1 Primes	31
4.12 Pollard's rho algorithm	13	8.1.1 First primes	31
		8.1.2 Arbitrary length primes	31
		8.1.3 $\sim 1 \times 10^9$	31
		8.1.4 $\sim 1 \times 10^{18}$	31
		8.2 Pell's equation	31
		8.3 Burnside's lemma and Polya's enumeration theorem	32
		8.4 Lagrange's interpolation	32

1 General

1.1 Code library checksum

```
ab14 #!/usr/bin/python3
c502 import re, sys, hashlib
427e
f7db for line in sys.stdin.read().strip().split("\n") :
ddf5     print(hashlib.md5(re.sub(r'\s|//[.]*', '', line).encode('utf8')).hexdigest()
        [-4:], line)
```

1.2 Makefile

```
dab2 .PHONY : run
427e
207e $(t) : $(t).cpp
2d16     g++ --std=c++14 -Wall -D__LOCAL_DEBUG__ -fsanitize=undefined -fsanitize=
        address -ggdb -pipe -o $@ $<
427e
5f25 run : $(t)
bf3e     ./$$(t) < $$(t).in
```

1.3 .vimrc

```
914c set nocompatible
733d syntax on
6bbc colorscheme slate
7db5 set number
b0e3 set cursorline
061b set shiftwidth=2
8011 set softtabstop=2
a66d set tabstop=2
d23a set expandtab
5245 set magic
740c set smartindent
bee8 set backspace=indent,eol,start
815d set cmdheight=1
0a40 set laststatus=2
1c67 set whichwrap=b,s,<,>,[,]
```

1.4 Stack

```
const int STK_SZ = 2000000;
char STK[STK_SZ * sizeof(void)];
void *STK_BAK;

#if defined(__i386__)
#define SP "%esp"
#elif defined(__x86_64__)
#define SP "%rsp"
#endif

int main() {
    asm volatile("movl SP, %0; movl %1, SP: "=g"(STK_BAK):"g"(STK+sizeof(STK)):");
    ;

    // main program

    asm volatile("movl %0, SP: "=g"(STK_BAK));
    return 0;
}
```

1.5 Template

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

#ifdef __LOCAL_DEBUG__
# define _debug(fmt, ...) fprintf(stderr, "[%s] " fmt "\n", \
    __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) begin(x), end(x)
typedef long long LL;
typedef unsigned long long ULL;
```

2 Miscellaneous Algorithms

2.1 2-SAT

```

0f42 const int MAXN = 100005;
03a9 struct twoSAT{
5c83     int n;
8f72     vector<int> G[MAXN*2];
d060     bool mark[MAXN*2];
b42d     int S[MAXN*2], c;
427e
d34f     void init(int n){
b985         this->n = n;
f9ec         for (int i=0; i<n*2; i++) G[i].clear();
0609         memset(mark, 0, sizeof(mark));
95cf     }
427e
3bd5     bool dfs(int x){
bd70         if (mark[x^1]) return false;
c96a         if (mark[x]) return true;
fd23         mark[x] = true;
4bea         S[c++] = x;
1ce6         for (int i=0; i<G[x].size(); i++)
d942             if (!dfs(G[x][i])) return false;
3361         return true;
95cf     }
427e
5894     void add_clause(int x, bool xval, int y, bool yval){
6afe         x = x * 2 + xval;
e680         y = y * 2 + yval;
81cc         G[x^1].push_back(y);
6835         G[y^1].push_back(x);
95cf     }
427e
d0cb     bool solve() {
7c39         for (int i=0; i<n*2; i+=2){
e63f             if (!mark[i] && !mark[i+1]){
88fb                 c = 0;
f4b9                 if (!dfs(i)){
3f03                     while (c > 0) mark[S[--c]] = false;
86c5                     if (!dfs(i+1)) return false;
95cf                 }
95cf             }

```

```

    }
    return true;
}

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

```

95cf
3361
95cf
427e
5f0a
329b

2.2 Knuth's optimization

```

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;
                }
            }
        }
    }
};

```

5c83
d77c
427e
b7ec
0bc7
0423
8f5e
9488
95cf
be8e
95b5
aa0f
95cf
ec08
88b8
d3da
9824
a24a
f933
90d2
9bd0
26b5
e6af
9c88
95cf
95cf
95cf
95cf
329b

2.3 Mo's algorithm

All intervals are closed on both sides. When running functions `enter()` and `leave()`, the global `l` and `r` has not changed yet.

Usage:

```
add_query(id, l, r)    Add id-th query [l, r].
run()                 Run Mo's algorithm.
init()                TODO. Initialize the range [l, r].
yield(id)             TODO. Yield answer for id-th query.
enter(o)              TODO. Add o-th element.
leave(o)              TODO. Remove o-th element.
```

```
5194 constexpr int BLOCK_SZ = 300;
427e
3ec4 struct query { int l, r, id; };
d26a vector<query> queries;
427e
1e30 void add_query(int id, int l, int r) {
54c9     queries.push_back(query{l, r, id});
95cf }
427e
9f6b int l, r;
427e
427e // ----- functions to implement -----
62b4 inline void init();
50e1 inline void yield(int id);
b20d inline void enter(int o);
13af inline void leave(int o);
427e
37f0 void run() {
ab0b     if (queries.empty()) return;
8508     sort(range(queries), [](query lhs, query rhs) {
c7f8         int lb = lhs.l / BLOCK_SZ, rb = rhs.l / BLOCK_SZ;
03e7         if (lb != rb) return lb < rb;
0780         return lhs.r < rhs.r;
b251     });
6196     l = queries[0].l;
9644     r = queries[0].r;
07e2     init();
5bc9     for (query q : queries) {
7bc7         while (l > q.l) enter(l - 1), l--;
d646         while (r < q.r) enter(r + 1), r++;
13f0         while (l < q.l) leave(l), l++;
e1c6         while (r > q.r) leave(r), r--;
```

```
        yield(q.id);
    }
}
```

```
82f5
95cf
95cf
```

3 String

3.1 Knuth-Morris-Pratt algorithm

```
const int SIZE = 10005;

struct kmp_matcher {
    char p[SIZE];
    int fail[SIZE];
    int len;

    void construct(const char* needle) {
        len = strlen(p);
        strcpy(p, needle);
        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* haystack) { // must be called after construct
        const char* t = haystack;
        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);
        }
    }
};
```

```
2836
427e
d02b
2d81
9847
57b7
427e
60cf
aaa1
3a87
3dd4
d8a8
147f
3c79
4643
95cf
95cf
427e
c464
427e
95cf
427e
2daf
700f
8482
8fd0
be8e
4e19
b5d5
f024
95cf
95cf
329b
```

3.2 Manacher algorithm

```

81d4 struct Manacher {
cd09     int Len;
9255     vector<int> lc;
b301     string s;
427e
ec07     void work() {
c033         lc[1] = 1;
6bef         int k = 1;
427e
491f         for (int i = 2; i <= Len; i++) {
7957             int p = k + lc[k] - 1;
5e04             if (i <= p) {
24a1                 lc[i] = min(lc[2 * k - i], p - i + 1);
8e2e             } else {
e0e5                 lc[i] = 1;
95cf             }
74ff             while (s[i + lc[i]] == s[i - lc[i]]) lc[i]++;
2b9a             if (i + lc[i] > k + lc[k]) k = i;
95cf         }
95cf     }
427e
bfd5     void init(const char *tt) {
aaaf         int len = strlen(tt);
f701         s.resize(len * 2 + 10);
7045         lc.resize(len * 2 + 10);
8e13         s[0] = '*';
ae54         s[1] = '#';
1321         for (int i = 0; i < len; i++) {
e995             s[i * 2 + 2] = tt[i];
69fd             s[i * 2 + 1] = '#';
95cf         }
43fd         s[len * 2 + 1] = '#';
75d1         s[len * 2 + 2] = '\0';
61f7         Len = len * 2 + 2;
3e7a         work();
95cf     }
427e
b194     pair<int, int> maxpal(int l, int r) {
901a         int center = l + r + 1;
ffb2         int rad = lc[center] / 2;
ab54         int rmid = (l + r + 1) / 2;

```

```

    int r1 = rmid - rad, rr = rmid + rad - 1;
    if ((r ^ 1) & 1) {
    } else rr++;
    return {max(l, r1), min(r, rr)};
}
};

```

```

17e4
3908
69f3
69dc
95cf
329b

```

3.3 Aho-corasick automaton

```

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) {

```

```

a1ad
9143
daca
427e
8690
93d2
a7a6
ce3c
b1c6
a506
3e14
f689
95cf
95cf
cc78
31f0
15dd
ce3c
ab59
0ef5
9d58
b333
95cf
3e14
b3ff
d2ea
c275
654c
95cf
95cf
95cf
427e
7752

```

```

043e     if (j) {
427e         // ! add codes for having found word with tag[j]
4a96         found(pos, last[j]);
95cf     }
95cf }
427e
9785 void find(const char* text) { // must be called after construct()
80a4     int p = 0, c, len = strlen(text);
9c94     rep(i, len) {
b3db         c = id(text[i]);
f119         p = tr[p][c];
f08e         if (tag[p])
389b             found(i, p);
1e67         else if (last[p])
299e             found(i, last[p]);
95cf     }
95cf }
329b };

```

3.4 Trie

```

e6f1 const int MAXN = 12000;
dd87 const int CHARN = 26;
427e
8ff5 inline int id(char c) { return c - 'a'; }
427e
a281 struct Trie {
5c83     int n;
f4f5     int tr[MAXN][CHARN]; // Trie tree, 0 denotes fail
35a5     int tag[MAXN];
427e
4fee     Trie() {
3ccc         memset(tr[0], 0, sizeof(tr[0]));
4d52         tag[0] = 0;
46bf         n = 1;
95cf     }
427e
427e     // tag should not be 0
30b0 void add(const char* s, int t) {
d50a     int p = 0, c, len = strlen(s);
9c94     rep(i, len) {
3140         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];
}
};

```

```

d6c8
26dd
2e5c
73bb
95cf
f119
95cf
35ef
95cf
427e
427e
427e
216c
d50a
9c94
3140
f339
f119
95cf
840e
95cf
329b

```

3.5 Suffix array

The character immediately after the end of the string **MUST** be set to the **UNIQUE SMALLEST** element.

Usage:

<code>s[]</code>	the source string
<code>sa[i]</code>	the index of starting position of i -th suffix
<code>rk[i]</code>	the number of suffixes less than the suffix starting from i
<code>h[i]</code>	the longest common prefix between the i -th and $(i-1)$ -th lexicographically smallest suffixes
<code>n</code>	size of source string
<code>m</code>	size of character set

```

void radix_sort(int x[], int y[], int sa[], int n, int m) {
    static int cnt[1000005]; // size > max(n, m)
    fill(cnt, cnt + m, 0);
    rep(i, n) cnt[x[y[i]]]++;
    partial_sum(cnt, cnt + m, cnt);
    for (int i = n - 1; i >= 0; i--) sa[--cnt[x[y[i]]]] = y[i];
}

```

```

de09
ec00
6066
93b7
9154
acac
95cf

```

```

427e void suffix_array(int s[], int sa[], int rk[], int n, int m) {
c939     static int y[1000005]; // size > n
a69a     copy(s, s + n, rk);
7306     iota(y, y + n, 0);
afb6     radix_sort(rk, y, sa, n, m);
7b42     for (int j = 1, p = 0; j <= n; j <= 1, m = p, p = 0) {
c8c2         for (int i = n - j; i < n; i++) y[p++] = i;
8c3a         rep (i, n) if (sa[i] >= j) y[p++] = sa[i] - j;
9323         radix_sort(rk, y, sa, n, m + 1);
9e9d         swap_ranges(rk, rk + n, y);
ae41         rk[sa[0]] = p = 1;
ffd2         for (int i = 1; i < n; i++)
445e             rk[sa[i]] = ((y[sa[i]] == y[sa[i-1]] and y[sa[i]+j] == y[sa[i-1]+j])
f8dc                 ? p : ++p);
02f0         if (p == n) break;
95cf     }
97d9     rep (i, n) rk[sa[i]] = i;
95cf }
427e
1715 void calc_height(int s[], int sa[], int rk[], int h[], int n) {
c41f     int k = 0;
f313     h[0] = 0;
be8e     rep (i, n) {
0883         k = max(k - 1, 0);
527d         if (rk[i]) while (s[i+k] == s[sa[rk[i]-1]+k]) ++k;
56b7         h[rk[i]] = k;
95cf     }
95cf }

```

3.6 Rolling hash

PLEASE call `init_hash()` in `int main()`!

Usage:

`build(str)` Construct the hasher with given string.
`operator()(l, r)` Get hash value of substring $[l, r)$.

```

1e42 const LL mod = 1006658951440146419, g = 967;
9f60 const int MAXN = 200005;
0291 LL pg[MAXN];
427e
dfe7 inline LL mul(LL x, LL y) { return __int128_t(x) * y % mod; }
427e

```

```

void init_hash() { // must be called in `int main()`
    pg[0] = 1;
    for (int i = 1; i < MAXN; i++) pg[i] = mul(pg[i-1], g);
}

struct hasher {
    LL val[MAXN];

    void build(const char *str) { // assume lower-case letter only
        for (int i = 0; str[i]; i++)
            val[i+1] = (mul(val[i], g) + str[i]) % mod;
    }

    LL operator() (int l, int r) { // [l, r)
        return (val[r] - mul(val[l], pg[r-l]) + mod) % mod;
    }
};

```

4 Math

4.1 Extended Euclidean algorithm and Chinese remainder theorem

```

void exgcd(LL a, LL b, LL &g, LL &x, LL &y) {
    if (!b) g = a, x = 1, y = 0;
    else {
        exgcd(b, a % b, g, y, x);
        y -= x * (a / b);
    }
}

LL crt(LL r[], LL p[], int n) {
    LL q = 1, ret = 0;
    rep (i, n) q *= p[i];
    rep (i, n) {
        LL m = q / p[i];
        LL d, x, y;
        exgcd(p[i], m, d, x, y);
        ret = (ret + y * m * r[i]) % q;
    }
    return (q + ret) % q;
}

```


4.2 Matrix powermod

```

44b4 const int MAXN = 105;
92df const LL modular = 1000000007;
5c83 int n; // order of matrices
427e
8864 struct matrix{
3180     LL m[MAXN][MAXN];
427e
43c5     void operator *=(matrix& a){
e735         static LL t[MAXN][MAXN];
34d7         Rep (i, n){
4c11             Rep (j, n){
ee1e                 t[i][j] = 0;
c4a7                 Rep (k, n){
fc9f                     t[i][j] += (m[i][k] * a.m[k][j]) % modular;
199e                     t[i][j] %= modular;
95cf                 }
95cf             }
95cf         }
dad4         memcpy(m, t, sizeof(t));
95cf     }
329b };
427e
63d8 matrix r;
3ec2 void m_powmod(matrix& b, LL e){
83f0     memset(r.m, 0, sizeof(r.m));
a7c3     Rep(i, n)
de64         r.m[i][i] = 1;
3e90     while (e){
5a0e         if (e & 1) r *= b;
35c5         b *= b;
16fc         e >>= 1;
95cf     }
95cf }

```

4.3 Linear basis

```

8b44 const int MAXD = 30;
03a6 struct linearbasis {
3558     ULL b[MAXD] = {};
427e

```

```

bool insert(LL v) {
    for (int j = MAXD - 1; j >= 0; j--) {
        if (!(v & (1ll << j))) continue;
        if (b[j]) v ^= b[j]
        else {
            for (int k = 0; k < j; k++)
                if (v & (1ll << k)) v ^= b[k];
            for (int k = j + 1; k < MAXD; k++)
                if (b[k] & (1ll << j)) b[k] ^= v;
            b[j] = v;
            return true;
        }
    }
    return false;
}

```

4.4 Gauss elimination over finite field

```

const LL p = 1000000007;

LL powmod(LL b, LL e) {
    LL r = 1;
    while (e) {
        if (e & 1) r = r * b % p;
        b = b * b % p;
        e >>= 1;
    }
    return r;
}

typedef vector<LL> VLL;
typedef vector<VLL> WLL;

```

```

LL gauss(WLL &a, WLL &b) {
    const int n = a.size(), m = b[0].size();
    vector<int> irow(n), icol(n), ipiv(n);
    LL det = 1;

    rep (i, n) {
        int pj = -1, pk = -1;
        rep (j, n) if (!ipiv[j])

```

1566
9b2b
de36
ee78
037f
7836
f0b4
b0aa
46c9
8295
3361
95cf
95cf
438e
95cf
329b

b784
427e
2a2c
95a2
3e90
1783
5549
16fc
95cf
547e
95cf
427e
c130
42ac
427e
2c62
561b
a25e
2976
427e
be8e
d2b5
6b4a

```

e582     rep (k, n) if (!ipiv[k])
6112         if (pj == -1 || a[j][k] > a[pj][pk]) {
a905             pj = j;
657b             pk = k;
95cf         }
d480     if (a[pj][pk] == 0) return 0;
0305     ipiv[pk]++;
8dad     swap(a[pj], a[pk]);
aad8     swap(b[pj], b[pk]);
be4d     if (pj != pk) det = (p - det) % p;
d080     irow[i] = pj;
f156     icol[i] = pk;
427e
4ecd     LL c = powmod(a[pk][pk], p - 2);
865b     det = det * a[pk][pk] % p;
c36a     a[pk][pk] = 1;
dd36     rep (j, n) a[pk][j] = a[pk][j] * c % p;
1b23     rep (j, m) b[pk][j] = b[pk][j] * c % p;
f8f3     rep (j, n) if (j != pk) {
e97f         c = a[j][pk];
c449         a[j][pk] = 0;
820b         rep (k, n) a[j][k] = (a[j][k] + p - a[pk][k] * c % p) % p;
f039         rep (k, m) b[j][k] = (b[j][k] + p - b[pk][k] * c % p) % p;
95cf     }
95cf }
427e
37e1     for (int j = n - 1; j >= 0; j--) if (irow[j] != icol[j]) {
50dc         for (int k = 0; k < n; k++) swap(a[k][irow[j]], a[k][icol[j]]);
95cf     }
f27f     return det;
95cf }

```

4.5 Berlekamp-Massey algorithm

```

d790 vector<int> berlekamp(const vector<int>& a) {
4166     vector<int> p = {1}, r = {1};
baed     int dif = 1;
8bc9     rep (i, a.size()) {
3e58         int u = 0;
ac8e         rep (j, p.size())
a488             u = (u + 111 * p[j] * a[i-j]) % mod;
eae9         if (u == 0) {

```

```

        r.insert(r.begin(), 0);
    } else {
        auto op = p;
        p.resize(max(p.size(), r.size() + 1));
        int idif = inv(dif);
        rep (j, r.size())
            p[j+1] =
                (p[j+1] - 111 * r[j] * idif % mod * u % mod + mod) % mod;
        dif = u;
        r = op;
    }
}
return p;
}

```

```

b14c
8e2e
0c78
02f6
786b
9b57
793c
1836
644c
bc58
95cf
95cf
e149
95cf

```

4.6 Fast Walsh-Hadamard transform

```

void fwt(int* a, int n){
    for (int d = 1; d < n; d <= 1)
        for (int i = 0; i < n; i += d < 1)
            rep (j, d){
                int x = a[i+j], y = a[i+j+d];
                // a[i+j] = x+y, a[i+j+d] = x-y;    // xor
                // a[i+j] = x+y;                    // and
                // a[i+j+d] = x+y;                    // or
            }
}

```

```

061e
5595
05f2
b833
7796
427e
427e
427e
95cf
95cf
427e
4db1

```

```

void ifwt(int* a, int n){
    for (int d = 1; d < n; d <= 1)
        for (int i = 0; i < n; i += d < 1)
            rep (j, d){
                int x = a[i+j], y = a[i+j+d];
                // a[i+j] = (x+y)/2, a[i+j+d] = (x-y)/2;    // xor
                // a[i+j] = x-y;                                // and
                // a[i+j+d] = y-x;                                // or
            }
}

```

```

5595
05f2
b833
7796
427e
427e
427e
95cf
95cf
427e
2ab6

```

```

void conv(int* a, int* b, int n){
    fwt(a, n);
    fwt(b, n);

```

```

950a
e427

```

```

8a42     rep(i, n) a[i] *= b[i];
430f     ifwt(a, n);
95cf }

```

4.7 Fast fourier transform

```

4e09 const int NMAX = 1<<20;
427e
3fbf typedef complex<double> cplx;
427e
abd1 const double PI = 2*acos(0.0);
12af struct FFT{
c47c     int rev[NMAX];
27d7     cplx omega[NMAX], oinv[NMAX];
9827     int K, N;
427e
1442     FFT(int k){
e209         K = k; N = 1 << k;
b393         rep (i, N){
7ba3             rev[i] = (rev[i>>1]>>1) | ((i&1)<<(K-1));
1908             omega[i] = polar(1.0, 2.0 * PI / N * i);
a166             oinv[i] = conj(omega[i]);
95cf         }
95cf     }
427e
b941     void dft(cplx* a, cplx* w){
a215         rep (i, N) if (i < rev[i]) swap(a[i], a[rev[i]]);
ac6e         for (int l = 2; l <= N; l *= 2){
2969             int m = l/2;
b3cf             for (cplx* p = a; p != a + N; p += l)
c24f                 rep (k, m){
fe06                     cplx t = w[N/l*k] * p[k+m];
ecbf                     p[k+m] = p[k] - t; p[k] += t;
95cf                 }
95cf             }
95cf         }
427e
617b     void fft(cplx* a){dft(a, omega);}
a123     void ifft(cplx* a){
3b2f         dft(a, oinv);
57fc         rep (i, N) a[i] /= N;
95cf     }

```

```

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

```

```

427e
bdc0
6497
12a5
f84e
95cf
329b

```

4.8 Number theoretic transform

```

const int NMAX = 1<<21;

// 998244353 = 7*17*2^23+1, G = 3
const int P = 1004535809, G = 3; // = 479*2^21+1

struct NTT{
    int rev[NMAX];
    LL omega[NMAX], oinv[NMAX];
    int g, g_inv; // g: g_n = G^((P-1)/n)
    int K, N;

    LL powmod(LL b, LL e){
        LL r = 1;
        while (e){
            if (e&1) r = r * b % P;
            b = b * b % P;
            e >>= 1;
        }
        return r;
    }

    NTT(int k){
        K = k; N = 1 << k;
        g = powmod(G, (P-1)/N);
        g_inv = powmod(g, N-1);
        omega[0] = oinv[0] = 1;
        rep (i, N){
            rev[i] = (rev[i>>1]>>1) | ((i&1)<<(K-1));
            if (i){
                omega[i] = omega[i-1] * g % P;
                oinv[i] = oinv[i-1] * g_inv % P;
            }
        }
    }
}

```

```

4ab9
427e
427e
fb9a
427e
87ab
c47c
0eda
81af
9827
427e
2a2c
95a2
3e90
6624
489e
16fc
95cf
547e
95cf
427e
f420
e209
7652
4b3a
e04f
b393
7ba3
ad4f
8d8b
9e14
95cf

```

```

95cf    }
95cf    }
427e
9668    void _ntt(LL* a, LL* w){
a215        rep (i, N) if (i < rev[i]) swap(a[i], a[rev[i]]);
ac6e        for (int l = 2; l <= N; l *= 2){
2969            int m = l/2;
7a1d            for (LL* p = a; p != a + N; p += l)
c24f                rep (k, m){
0ad3                    LL t = w[N/l*k] * p[k+m] % P;
6209                    p[k+m] = (p[k] - t + P) % P;
fa1b                    p[k] = (p[k] + t) % P;
95cf                }
95cf            }
95cf        }
427e
92ea    void ntt(LL* a){_ntt(a, omega);}
5daf    void intt(LL* a){
1f2a        LL inv = powmod(N, P-2);
9910        _ntt(a, oinv);
a873        rep (i, N) a[i] = a[i] * inv % P;
95cf    }
427e
3a5b    void conv(LL* a, LL* b){
ad16        ntt(a); ntt(b);
e49e        rep (i, N) a[i] = a[i] * b[i] % P;
5748        intt(a);
95cf    }
329b    };

```

4.9 Sieve of Euler

```

cfc3    const int MAXX = 1e7+5;
5861    bool p[MAXX];
73ae    int prime[MAXX], sz;
427e
9bc6    void sieve(){
9628        p[0] = p[1] = 1;
1ec8        for (int i = 2; i < MAXX; i++){
bf28            if (!p[i]) prime[sz++] = i;
e82c            for (int j = 0; j < sz && i*prime[j] < MAXX; j++){
b6a9                p[i*prime[j]] = 1;

```

```

        if (i % prime[j] == 0) break;
    }
}

```

4.10 Sieve of Euler (General)

```

namespace sieve {
constexpr int MAXN = 10000007;
bool p[MAXN]; // true if not prime
int prime[MAXN], sz;
int pval[MAXN], pcnt[MAXN];
int f[MAXN];

void exec(int N = MAXN) {
    p[0] = p[1] = 1;

    pval[1] = 1;
    pcnt[1] = 0;
    f[1] = 1;

    for (int i = 2; i < N; i++) {
        if (!p[i]) {
            prime[sz++] = i;
            for (LL j = i; j < N; j *= i) {
                int b = j / i;
                pval[j] = i * pval[b];
                pcnt[j] = pcnt[b] + 1;
                f[j] = _____; // f[j] = f(i^pcnt[j])
            }
        }
        for (int j = 0; i * prime[j] < N; j++) {
            int x = i * prime[j]; p[x] = 1;
            if (i % prime[j] == 0) {
                pval[x] = pval[i] * prime[j];
                pcnt[x] = pcnt[i] + 1;
            } else {
                pval[x] = prime[j];
                pcnt[x] = 1;
            }
            if (x != pval[x]) {
                f[x] = f[x / pval[x]] * f[pval[x]]

```

5f51
95cf
95cf
95cf

b62e
6589
e982
6ae8
cbf7
6030
427e
76f6
9628
427e
8a8a
bdda
c6b9
427e
a643
01d6
b2b2
37d9
758c
81fd
e0f3
a96c
95cf
95cf
34c0
f87a
20cc
9985
3f93
8e2e
cc91
6322
95cf
6191
d614

```

95cf      }
5f51      if (i % prime[j] == 0) break;
95cf    }
95cf  }
95cf  }
95cf  }

```

4.11 Miller-Rabin primality test

The array `a[]` (excluding `senitel`, i.e. `LLONG_MAX`) should be

```

{2}                when  $n < 2,047$ .
{2, 7, 61}          when  $n < 4,759,123,141 (2^{32})$ .
{2, 3, 5, 7, 11}    when  $n < 2.1 \times 10^{12}$ .
{2, 325, 9375, 28178, 450775, 9780504, 1795265022} when  $n < 2^{64}$ .

```

```

f16f bool test(LL n){
59f2   if (n < 3) return n==2;
427e   // ! The array a[] should be modified if the range of x changes.
3f11   const LL a[] = {2LL, 7LL, 61LL, LLONG_MAX};
c320   LL r = 0, d = n-1, x;
f410   while (~d & 1) d >>= 1, r++;
2975   for (int i=0; a[i] < n; i++){
ece1     x = powmod(a[i], d, n); // ! powmod must use for 64bit mulmod
7f99     if (x == 1 || x == n-1) goto next;
e257     rep (i, r) {
d7ff       x = mulmod(x, x, n);
8d2e       if (x == n-1) goto next;
95cf     }
438e     return false;
d490 next:;
95cf   }
3361   return true;
95cf }

```

4.12 Pollard's rho algorithm

```

2e6b ULL gcd(ULL a, ULL b) {return b ? gcd(b, a % b) : a;}
427e
54a5 ULL PollardRho(ULL n){
45eb   ULL c, x, y, d = n;

```

```

if (~n&1) return 2;
while (d == n){
  x = y = 2;
  d = 1;
  c = rand() % (n - 1) + 1;
  while (d == 1){
    x = (mulmod(x, x, n) + c) % n;
    y = (mulmod(y, y, n) + c) % n;
    y = (mulmod(y, y, n) + c) % n;
    d = gcd(x>y ? x-y : y-x, n);
  }
}
return d;
}

```

```

d3e5
3c69
0964
4753
5952
9e5b
33d5
e1bf
e1bf
a313
95cf
95cf
5d89
95cf

```

5 Graph Theory

5.1 Strongly connected component

```

const int MAXV = 100005;

struct graph{
  vector<int> adj[MAXV];
  stack<int> s;
  int V; // number of vertices
  int pre[MAXV], lnk[MAXV], scc[MAXV];
  int time, sccn;

  void add_edge(int u, int v){
    adj[u].push_back(v);
  }

  void dfs(int u){
    pre[u] = lnk[u] = ++time;
    s.push(u);
    for (int v : adj[u]){
      if (!pre[v]){
        dfs(v);
        lnk[u] = min(lnk[u], lnk[v]);
      } else if (!scc[v]){
        lnk[u] = min(lnk[u], pre[v]);
      }
    }
  }
}

```

```

837c
427e
2ea0
88e3
9cad
3d02
8b6c
27ee
427e
bfab
c71a
95cf
427e
d714
7e41
80f6
18f6
173e
5f3c
002c
6068
d5df

```

```

95cf      }
95cf      }
8de2      if (lnk[u] == pre[u]){
660f          sccn++;
3c9e          int x;
a69f          do {
3834              x = s.top(); s.pop();
b0e9              scc[x] = sccn;
6757          } while (x != u);
95cf      }
95cf      }
427e
4c88      void find_scc(){
f4a2          time = sccn = 0;
8de7          memset(scc, 0, sizeof scc);
8c2f          memset(pre, 0, sizeof pre);
6901          Rep (i, V){
56d1              if (!pre[i]) dfs(i);
95cf          }
95cf      }
427e
27ce      vector<int> adjc[MAXV];
364d      void contract(){
1a1e          Rep (i, V)
21a2              rep (j, adj[i].size()){
b730                  if (scc[i] != scc[adj[i][j]])
b46e                      adjc[scc[i]].push_back(scc[adj[i][j]]);
95cf              }
95cf          }
329b      };

```

5.2 Vertex biconnected component

```

0f42      const int MAXN = 100005;
2ea0      struct graph {
33ae          int pre[MAXN], iscut[MAXN], bccno[MAXN], dfs_clock, bcc_cnt;
848f          vector<int> adj[MAXN], bcc[MAXN];
6b06          set<pair<int, int>> bcce[MAXN];
427e
76f7          stack<pair<int, int>> s;
427e
bfab          void add_edge(int u, int v) {

```

```

adj[u].push_back(v);
adj[v].push_back(u);
}

int dfs(int u, int fa) {
    int lowu = pre[u] = ++dfs_clock;
    int child = 0;
    for (int v : adj[u]) {
        if (!pre[v]) {
            s.push({u, v});
            child++;
            int lowv = dfs(v, u);
            lowu = min(lowu, lowv);
            if (lowv >= pre[u]) {
                iscut[u] = 1;
                bcc[bcc_cnt].clear();
                bcce[bcc_cnt].clear();
                while (1) {
                    int xu, xv;
                    tie(xu, xv) = s.top(); s.pop();
                    bcce[bcc_cnt].insert({min(xu, xv), max(xu, xv)});
                    if (bccno[xu] != bcc_cnt) {
                        bcc[bcc_cnt].push_back(xu);
                        bccno[xu] = bcc_cnt;
                    }
                    if (bccno[xv] != bcc_cnt) {
                        bcc[bcc_cnt].push_back(xv);
                        bccno[xv] = bcc_cnt;
                    }
                    if (xu == u && xv == v) break;
                }
                bcc_cnt++;
            }
        } else if (pre[v] < pre[u] && v != fa) {
            s.push({u, v});
            lowu = min(lowu, pre[v]);
        }
    }
    if (fa < 0 && child == 1) iscut[u] = 0;
    return lowu;
}

void find_bcc(int n) {
    memset(pre, 0, sizeof pre);

```

```

c71a
a717
95cf
427e
7d3c
9fe6
ec14
18f6
173e
e7f8
fddf
f851
189c
b687
6323
57eb
90b8
a147
a6a3
a0c3
0ef5
3db2
e0db
d27f
95cf
f357
752b
57c9
95cf
7096
95cf
03f5
95cf
7470
e7f8
f115
95cf
95cf
e104
1160
95cf
427e
17be
8c2f

```

```
e2d2     memset(iscut, 0, sizeof iscut);
40d3     memset(bccno, -1, sizeof bccno);
fae2     dfs_clock = bcc_cnt = 0;
5c63     rep (i, n) if (!pre[i]) dfs(i, -1);
95cf     }
329b     };
```

5.3 Cut vertices

If the graph is unconnected, the algorithm should be run on each component. One may run Rep (i, n)if (!dfn[i])tarjan(i, i) for unconnected graph.

Usage:

add_edge(u, v) Add an undirected edge (u, v).
tarjan(u, fa) Run Tarjan's algorithm on tree rooted at fa. Please call
 with identical u and fa.
cut[v] Whether v is a cut vertex.

```
9f60     const int MAXN = 200005;
0b32     vector<int> adj[MAXN];
18e4     int dfn[MAXN], low[MAXN], idx;
d39d     bool cut[MAXN];
427e
bfa8     void add_edge(int u, int v) {
c71a         adj[u].push_back(v);
a717         adj[v].push_back(u);
95cf     }
427e
50aa     void tarjan(int u, int fa) {
9891         dfn[u] = low[u] = ++idx;
ec14         int child = 0;
18f6         for (int v : adj[u]) {
3c64             if (!dfn[v]) {
9636                 tarjan(v, fa); low[u] = min(low[u], low[v]);
f368                 if (low[v] >= dfn[u] && u != fa) cut[u] = true;
7923                 child += u == fa;
95cf             }
769a             low[u] = min(low[u], dfn[v]);
95cf         }
7927         if (u == fa && child > 1) cut[u] = true;
95cf     }
```

5.4 Minimum spanning arborescence (Chu-Liu)

All vertices are 1-based.

Usage:

getans(n, root, edges) Compute the total size of MSA rooted at root.

Time Complexity: $O(|V||E|)$

```
struct edge {
    int u, v;
    LL w;
};

const int MAXN = 10005;
LL in[MAXN];
int pre[MAXN], vis[MAXN], id[MAXN];

LL getans(int n, int rt, vector<edge>& edges) {
    LL ans = 0;
    int cnt = 0;
    while (1) {
        Rep (i, n) in[i] = LLONG_MAX, id[i] = vis[i] = 0;
        for (auto e : edges) {
            if (e.u != e.v and e.w < in[e.v]) {
                pre[e.v] = e.u;
                in[e.v] = e.w;
            }
        }
        in[rt] = 0;
        Rep (i, n) {
            if (in[i] == LLONG_MAX) return -1;
            ans += in[i];
            int u;
            for (u = i; u != rt && vis[u] != i && !id[u]; u = pre[u])
                vis[u] = i;
            if (u != rt && !id[u]) {
                id[u] = ++cnt;
                for (int v = pre[u]; v != u; v = pre[v])
                    id[v] = cnt;
            }
        }
        if (!cnt) return ans;
        Rep (i, n) if (!id[i]) id[i] = ++cnt;
        for (auto& e : edges) {
```

```
bcf8
54f1
309c
329b
427e
f5a4
7124
1c1d
427e
5a43
f7ff
8abb
a147
641a
0705
073a
c1df
5fbc
95cf
95cf
3fdb
34d7
3c97
cf57
a763
4b0e
88a2
4b22
b66e
0443
5c22
95cf
95cf
91e9
5e22
7400
```

```

7750         LL laz = in[e.v];
97ae         e.u = id[e.u];
fae6         e.v = id[e.v];
bdd2         if (e.u != e.v) e.w -= laz;
95cf     }
6cc4         n = cnt; rt = id[rt]; cnt = 0;
95cf     }
95cf }

```

5.5 Maximum flow (Dinic)

Usage:

add_edge(u, v, c) Add an edge from u to v with capacity c .
max_flow(s, t) Compute maximum flow from s to t .

Time Complexity: For general graph, $O(V^2E)$; for network with unit capacity, $O(\min\{V^{2/3}, \sqrt{E}\}E)$; for bipartite network, $O(\sqrt{VE})$.

```

bcf8 struct edge{
60e2     int from, to;
5e6d     LL cap, flow;
329b };
427e
e2cd const int MAXN = 1005;
9062 struct Dinic {
4dbf     int n, m, s, t;
9f0c     vector<edge> edges;
b891     vector<int> G[MAXN];
bbb6     bool vis[MAXN];
b40a     int d[MAXN];
ddec     int cur[MAXN];

427e
5973     void add_edge(int from, int to, LL cap) {
7b55         edges.push_back(edge{from, to, cap, 0});
1db7         edges.push_back(edge{to, from, 0, 0});
fe77         m = edges.size();
dff5         G[from].push_back(m-2);
8f2d         G[to].push_back(m-1);
95cf     }
427e
1836     bool bfs() {
3b73         memset(vis, 0, sizeof(vis));
93d2         queue<int> q;
5d13         q.push(s);

```

```

vis[s] = 1;
d[s] = 0;
while (!q.empty()) {
    int x = q.front(); q.pop();
    for (int i = 0; i < G[x].size(); i++) {
        edge& e = edges[G[x][i]];
        if (!vis[e.to] && e.cap > e.flow) {
            vis[e.to] = 1;
            d[e.to] = d[x] + 1;
            q.push(e.to);
        }
    }
}
return vis[t];
}

LL dfs(int x, LL a) {
    if (x == t || a == 0) return a;
    LL flow = 0, f;
    for (int& i = cur[x]; i < G[x].size(); i++) {
        edge& e = edges[G[x][i]];
        if (d[x] + 1 == d[e.to] && (f = dfs(e.to, min(a, e.cap-e.flow))) > 0)
        {
            e.flow += f;
            edges[G[x][i]^1].flow -= f;
            flow += f;
            a -= f;
            if(a == 0) break;
        }
    }
    return flow;
}

LL max_flow(int s, int t) {
    this->s = s; this->t = t;
    LL flow = 0;
    while (bfs()) {
        memset(cur, 0, sizeof(cur));
        flow += dfs(s, LLONG_MAX);
    }
    return flow;
}

vector<int> min_cut() { // call this after maxflow

```



```

1df9     vector<int> ans;
df9a     for (int i = 0; i < edges.size(); i++) {
56d8         edge& e = edges[i];
46a2         if(vis[e.from] && !vis[e.to] && e.cap > 0) ans.push_back(i);
95cf     }
4206     return ans;
95cf }
329b };

```

5.6 Maximum cardinality bipartite matching (Hungarian)

```

302f #include <bits/stdc++.h>
421c using namespace std;
427e
0d6c #define rep(i, n) for (int i = 0; i < (n); i++)
cfe3 #define Rep(i, n) for (int i = 1; i <= (n); i++)
8843 #define range(x) (x).begin(), (x).end()
5cad typedef long long LL;
427e
84ee struct Hungarian{
fbf6     int nx, ny;
9ec6     vector<int> mx, my;
9d4c     vector<vector<int>> > e;
edec     vector<bool> mark;
427e
8324     void init(int nx, int ny){
c1d1         this->nx = nx;
f9c1         this->ny = ny;
ac92         mx.resize(nx); my.resize(ny);
3f11         e.clear(); e.resize(nx);
1023         mark.resize(nx);
95cf     }
427e
4589     inline void add(int a, int b){
486c         e[a].push_back(b);
95cf     }
427e
0c2b     bool augment(int i){
207c         if (!mark[i]) {
dae4             mark[i] = true;
6a1e             for (int j : e[i]){
0892                 if (my[j] == -1 || augment(my[j])){

```

```

        mx[i] = j; my[j] = i;
        return true;
    }
}
return false;
}

int match(){
    int ret = 0;
    fill(range(mx), -1);
    fill(range(my), -1);
    rep (i, nx){
        fill(range(mark), false);
        if (augment(i)) ret++;
    }
    return ret;
}
};

```

```

9ca3
3361
95cf
95cf
95cf
438e
95cf
427e
3fac
5b57
b0f1
b957
4ed1
13a5
cc89
95cf
ee0f
95cf
329b

```

5.7 Maximum matching of general graph (Edmond's blossom)

Usage:

init(n)	Initialize the template with n vertices, numbered from 1.
add_edge(u, v)	Add an undirected edge uv .
solve()	Find the maximum matching. Return the number of matched edges.
mate[]	The mate of a matched vertex. If it is not matched, then the value is 0.

Time Complexity: $O(|V|^3)$, but extremely fast in practice.

```

const int MAXN = 1024;
struct Blossom {

    vector<int> adj[MAXN];
    queue<int> q;
    int n;
    int label[MAXN], mate[MAXN], save[MAXN], used[MAXN];

    void init(int nv) {
        n = nv; for (auto& v : adj) v.clear();
        fill(range(label), 0); fill(range(mate), 0);

```

```

c041
6ab1
427e
0b32
93d2
5c83
0de2
427e
2186
3728
477d

```

```

bb35     fill(range(save), 0); fill(range(used), 0);
95cf     }
427e
c2dd     void add_edge(int u, int v) { adj[u].push_back(v); adj[v].push_back(u); }
427e
2a48     void rematch(int x, int y) {
8af8         int m = mate[x]; mate[x] = y;
1aa4         if (mate[m] == x) {
f4ba             if (label[x] <= n) {
740a                 mate[m] = label[x]; rematch(label[x], m);
8e2e             } else {
3341                 int a = 1 + (label[x] - n - 1) / n;
2885                 int b = 1 + (label[x] - n - 1) % n;
ef33                 rematch(a, b); rematch(b, a);
95cf             }
95cf         }
95cf     }
427e
8a50     void traverse(int x) {
43c0         Rep (i, n) save[i] = mate[i];
2ef7         rematch(x, x);
34d7         Rep (i, n) {
62c5             if (mate[i] != save[i]) used[i] ++;
97ef             mate[i] = save[i];
95cf         }
95cf     }
427e
8bf8     void relabel(int x, int y) {
d101         Rep (i, n) used[i] = 0;
c4ea         traverse(x); traverse(y);
34d7         Rep (i, n) {
dee9             if (used[i] == 1 and label[i] < 0) {
1c22                 label[i] = n + x + (y - 1) * n;
eb31                 q.push(i);
95cf             }
95cf         }
95cf     }
427e
a0ce     int solve() {
34d7         Rep (i, n) {
a073             if (mate[i]) continue;
1fc0             Rep (j, n) label[j] = -1;
7676             label[i] = 0; q = queue<int>(); q.push(i);
1c7d             while (q.size()) {

```

```

int x = q.front(); q.pop();
for (int y : adj[x]) {
    if (mate[y] == 0 and i != y) {
        mate[y] = x; rematch(x, y); q = queue<int>(); break;
    }
    if (label[y] >= 0) { relabel(x, y); continue; }
    if (label[mate[y]] < 0) {
        label[mate[y]] = x; q.push(mate[y]);
    }
}
}
}
int cnt = 0;
Rep (i, n) cnt += (mate[i] > i);
return cnt;
}
};

```

5.8 Minimum cost maximum flow

```

struct edge{
    int from, to;
    int cap, flow;
    LL cost;
};

const LL INF = LLONG_MAX / 2;
const int MAXN = 5005;
struct MCMF {
    int s, t, n, m;
    vector<edge> edges;
    vector<int> G[MAXN];
    bool inq[MAXN]; // queue
    LL d[MAXN]; // distance
    int p[MAXN]; // previous
    int a[MAXN]; // improvement

    void add_edge(int from, int to, int cap, LL cost) {
        edges.push_back(edge{from, to, cap, 0, cost});
        edges.push_back(edge{to, from, 0, 0, -cost});
        m = edges.size();
        G[from].push_back(m-2);

```

```

8f2d     G[to].push_back(m-1);
95cf     }
427e
3c52     bool spfa(){
93d2         queue<int> q;
8494         fill(d, d + MAXN, INF); d[s] = 0;
fd48         memset(inq, 0, sizeof(inq));
5e7c         q.push(s); inq[s] = true;
2dae         p[s] = 0; a[s] = INT_MAX;
cc78         while (!q.empty()){
b0aa             int u = q.front(); q.pop(); inq[u] = false;
3bba             for (int i : G[u]) {
56d8                 edge& e = edges[i];
3601                 if (e.cap > e.flow && d[e.to] > d[u] + e.cost){
55bc                     d[e.to] = d[u] + e.cost;
0bea                     p[e.to] = G[u][i];
8249                     a[e.to] = min(a[u], e.cap - e.flow);
e5d3                     if (!inq[e.to]) q.push(e.to), inq[e.to] = true;
95cf                 }
95cf             }
6d7c         return d[t] != INF;
95cf     }
427e
71a4     void augment(){
06f1         int u = t;
b19d         while (u != s){
db09             edges[p[u]].flow += a[t];
25a9             edges[p[u]^1].flow -= a[t];
e6c9             u = edges[p[u]].from;
95cf         }
95cf     }
427e
6e20     #ifndef GIVEN_FLOW
5972         bool min_cost(int s, int t, int f, LL& cost) {
590d             this->s = s; this->t = t;
21d4             int flow = 0;
23cb             cost = 0;
22dc             while (spfa()) {
bcd8                 augment();
a671                 if (flow + a[t] >= f){
b14d                     cost += (f - flow) * d[t]; flow = f;
3361                     return true;
8e2e                 } else {

```

```

flow += a[t]; cost += a[t] * d[t];
        }
    }
    return false;
}
#else
int min_cost(int s, int t, LL& cost) {
    this->s = s; this->t = t;
    int flow = 0;
    cost = 0;
    while (spfa()) {
        augment();
        flow += a[t]; cost += a[t] * d[t];
    }
    return flow;
}
#endif
};

```

2a83
95cf
95cf
438e
95cf
a8cb
f9a9
590d
21d4
23cb
22dc
bcd8
2a83
95cf
84fb
95cf
1937
329b

5.9 Global minimum cut (Stoer-Wagner)

Usage:

stoer(w) Compute the global minimum cut of the graph specified by the **symmetric** adjacent matrix w (0-based). Return the capacity of the cut and the indices of one part of the cut.

Time Complexity: $O(|V|^3)$

```

typedef vector<LL> VI;
typedef vector<VI> WVI;

pair<LL, VI> stoer(WVI &w) {
    int n = w.size();
    VI used(n), c, bestc;
    LL bestw = -1;

    for (int ph = n - 1; ph >= 0; ph--) {
        VI wt = w[0], added = used;
        int prev, last = 0;
        rep (i, ph) {
            prev = last;
            last = -1;
            for (int j = 1; j < n; j++)

```

f9d7
045e
427e
f012
66f7
4d98
329d
427e
cd21
ec6e
f20e
4b32
8bfc
0706
4942

```

c4b9         if (!added[j] && (last == -1 || wt[j] > wt[last]))
887d             last = j;
71bc     if (i == ph - 1) {
9cfa         rep (j, n) w[prev][j] += w[last][j];
1f25         rep (j, n) w[j][prev] = w[prev][j];
5613         used[last] = true;
8e11         c.push_back(last);
bb8e         if (bestw == -1 || wt[last] < bestw) {
bab6             bestc = c;
372e             bestw = wt[last];
95cf         }
8e2e     } else {
caeb         rep (j, n) wt[j] += w[last][j];
8b92         added[last] = true;
95cf     }
95cf }
95cf }
038c     return {bestw, bestc};
95cf }

```

5.10 Fast LCA

All indices of the tree are 1-based.

Usage:

preprocess(root) Initialize with tree rooted at root.
lca(u, v) Query the lowest common ancestor of u and v .

```

0e34 const int MAXN = 500005;
0b32 vector<int> adj[MAXN];
fccb int id[MAXN], nid;
1356 pair<int, int> st[MAXN << 1][33 - __builtin_clz(MAXN)];
427e
e16d void dfs(int u, int p, int d) {
0df2     st[id[u] = nid++][0] = {d, u};
18f6     for (int v : adj[u]) {
bd87         if (v == p) continue;
f58c         dfs(v, u, d + 1);
08ad         st[nid++][0] = {d, u};
95cf     }
95cf }
427e
3d1b void preprocess(int root) {
3269     nid = 0;

```

```

dfs(root, 0, 1);
int l = 31 - __builtin_clz(nid);
rep (j, l) rep (i, 1+nid-(1<<j))
    st[i][j+1] = min(st[i][j], st[i+(1<<j)][j]);
}

int lca(int u, int v) {
    tie(u, v) = minmax(id[u], id[v]);
    int k = 31 - __builtin_clz(v-u+1);
    return min(st[u][k], st[v-(1<<k)+1][k]).second;
}

```

5.11 Heavy-light decomposition

Time Complexity: The decomposition itself takes linear time. Each query takes $O(\log n)$ operations.

```

const int MAXN = 100005;
vector<int> adj[MAXN];
int sz[MAXN], top[MAXN], fa[MAXN], son[MAXN], depth[MAXN], id[MAXN];

void dfs1(int x, int dep, int par){
    depth[x] = dep;
    sz[x] = 1;
    fa[x] = par;
    int maxn = 0, s = 0;
    for (int c : adj[x]){
        if (c == par) continue;
        dfs1(c, dep + 1, x);
        sz[x] += sz[c];
        if (sz[c] > maxn){
            maxn = sz[c];
            s = c;
        }
    }
    son[x] = s;
}

int cid = 0;
void dfs2(int x, int t){
    top[x] = t;
    id[x] = ++cid;
    if (son[x]) dfs2(son[x], t);
}

```

```

c861     for (int c: adj[x]){
9881         if (c == fa[x]) continue;
5518         if (c == son[x]) continue;
13f9         else dfs2(c, c);
95cf     }
95cf }
427e
0f04 void decomp(int root){
9fa4     dfs1(root, 1, 0);
1c88     dfs2(root, root);
95cf }
427e
2c98 void query(int u, int v){
03a1     while (top[u] != top[v]){
45ec         if (depth[top[u]] < depth[top[v]]) swap(u, v);
427e         // id[top[u]] to id[u]
005b         u = fa[top[u]];
95cf     }
6083     if (depth[u] > depth[v]) swap(u, v);
427e     // id[u] to id[v]
95cf }

```

5.12 Centroid decomposition

Note that the centroid here is not the exact centroid of the graph. It only guarantees that the size of each subtree does not exceed half of that of the original tree. This is enough to guarantee the correct time complexity. All vertices are numbered from 1. Call `decomp(root)` to use.

Usage:

`decomp(u, p)` Decompose the tree rooted at u with parent p .

Time Complexity: The decomposition itself takes $O(n \log n)$ time.

```

1fb6 vector<int> adj[100005];
88e0 int sz[100005], sum;
427e
f93d void getsz(int u, int p) {
5b36     sz[u] = 1; sum++;
18f6     for (int v : adj[u]) {
bd87         if (v == p) continue;
e3cb         getsz(v, u);
8449         sz[u] += sz[v];
95cf     }
95cf }

```

```

int getcent(int u, int p) {
    for (int v : adj[u])
        if (v != p and sz[v] > sum / 2)
            return getcent(v, u);
    return u;
}

void decompose(int u) {
    sum = 0; getsz(u, 0);
    u = getcent(u, 0); // update u to the centroid

    for (int v : adj[u]) {
        // get answer for subtree v
    }
    // get answer for the whole tree
    // don't forget to count the centroid itself

    for (int v : adj[u]) { // divide and conquer
        adj[v].erase(find(range(adj[v]), u));
        decompose(v);
        adj[v].push_back(u); // restore deleted edge
    }
}

```

5.13 DSU on tree

This implementation avoids parallel existence of multiple data structures but requires that the data structure is invertible. To use this template, implement `merge`, `enter`, `leave` as needed; first call `decomp(root, 0)`, then call `work(root, 0, false)`. Labels of vertices start from 1.

Usage:

`decomp(u, p)` Decompose the tree u .
`work(u, p, keep)` Work for subtree u . When `keep` is set, information is not cleared.

Time Complexity: $O(n \log n)$ times the complexity for `merge`, `enter`, `leave`.

```

vector<int> adj[100005];
int sz[100005], son[100005];

void decomp(int u, int p) {
    sz[u] = 1;

```

```

18f6     for (int v : adj[u]) {
bd87         if (v == p) continue;
a851         decomp(v, u);
8449         sz[u] += sz[v];
d28c         if (sz[v] > sz[son[u]]) son[u] = v;
95cf     }
95cf }
427e
b7ec template <typename T>
62f5 void trav(T fn, int u, int p) {
4412     fn(u);
30b3     for (int v : adj[u]) if (v != p) trav(fn, v, u);
95cf }
427e
7467 #define for_light(v) for (int v : adj[u]) if (v != p and v != son[u])
33ff void work(int u, int p, bool keep) {
72a2     for_light(v) work(v, u, 0); // process light children
427e
427e     // process heavy child
427e     // current data structure contains info of heavy child
9866     if (son[u]) work(son[u], u, 1);
427e
18a9     auto merge = [u] (int c) { /* count contribution of c */ };
1ab0     auto enter = [] (int c) { /* add vertex c */ };
f241     auto leave = [] (int c) { /* remove vertex c */ };
427e
3d3b     for_light(v) {
74c6         trav(merge, v, u);
c13d         trav(enter, v, u);
95cf     }
427e
427e     // count answer for root and add it
427e     // Warning: special check may apply to root!
c54f     merge(u);
9dec     enter(u);
427e
427e     // Leave current tree
4e3e     if (!keep) trav(leave, u, p);
95cf }

```

6 Data Structures

6.1 Fenwick tree (point update range query)

```

struct bit_purq { // point update, range query
    int N;
    vector<LL> tr;

    void init(int n) { // fill the array with 0
        tr.resize(N = n + 5);
    }

    LL sum(int n) {
        LL ans = 0;
        while (n) {
            ans += tr[n];
            n &= n - 1;
        }
        return ans;
    }

    void add(int n, LL x){
        while (n < N) {
            tr[n] += x;
            n += n & -n;
        }
    }
};

```

9976
d7af
99ff
427e
d34f
1010
95cf
427e
63d0
f7ff
e290
0715
c0d4
95cf
4206
95cf
427e
f4bd
ad20
6c81
0af5
95cf
95cf
329b

6.2 Fenwick tree (range update point query)

```

struct bit_rupq{ // range update, point query
    int N;
    vector<LL> tr;

    void init(int n) { // fill the array with 0
        tr.resize(N = n + 5);
    }

    LL query(int n) {
        LL ans = 0;
        while (n < N) {

```

3d03
d7af
99ff
427e
d34f
1010
95cf
427e
38d4
f7ff
ad20

```

0715         ans += tr[n];
0af5         n += n & -n;
95cf     }
4206     return ans;
95cf }
427e
f4bd void add(int n, LL x) {
e290     while (n){
6c81         tr[n] += x;
c0d4         n &= n - 1;
95cf     }
95cf }
329b };

```

6.3 Segment tree

```

3942 LL p;
1ebb const int MAXN = 4 * 100006;
451a struct segtree {
27be     int l[MAXN], m[MAXN], r[MAXN];
4510     LL val[MAXN], tadd[MAXN], tmul[MAXN];
427e
ac35 #define lson (o<<1)
1294 #define rson (o<<1|1)
427e
1344 void pull(int o) {
bbe9     val[o] = (val[lson] + val[rson]) % p;
95cf }
427e
e4bc void push_add(int o, LL x) {
5dd6     val[o] = (val[o] + x * (r[o] - l[o])) % p;
6eff     tadd[o] = (tadd[o] + x) % p;
95cf }
427e
d658 void push_mul(int o, LL x) {
b82c     val[o] = val[o] * x % p;
aa86     tadd[o] = tadd[o] * x % p;
649f     tmul[o] = tmul[o] * x % p;
95cf }
427e
b149 void push(int o) {
3159     if (l[o] == m[o]) return;

```

```

if (tmul[o] != 1) {
    push_mul(lson, tmul[o]);
    push_mul(rson, tmul[o]);
    tmul[o] = 1;
}
if (tadd[o]) {
    push_add(lson, tadd[o]);
    push_add(rson, tadd[o]);
    tadd[o] = 0;
}
}

void build(int o, int ll, int rr) {
    int mm = (ll + rr) / 2;
    l[o] = ll; r[o] = rr; m[o] = mm;
    tmul[o] = 1;
    if (ll == mm) {
        scanf("%lld", val + o);
        val[o] %= p;
    } else {
        build(lson, ll, mm);
        build(rson, mm, rr);
        pull(o);
    }
}

void add(int o, int ll, int rr, LL x) {
    if (ll <= l[o] && r[o] <= rr) {
        push_add(o, x);
    } else {
        push(o);
        if (m[o] > ll) add(lson, ll, rr, x);
        if (m[o] < rr) add(rson, ll, rr, x);
        pull(o);
    }
}

void mul(int o, int ll, int rr, LL x) {
    if (ll <= l[o] && r[o] <= rr) {
        push_mul(o, x);
    } else {
        push(o);
        if (ll < m[o]) mul(lson, ll, rr, x);
        if (m[o] < rr) mul(rson, ll, rr, x);
    }
}

```

```

0a90
0f4a
045e
ac0a
95cf
1b82
9547
0e73
6234
95cf
95cf
427e
471c
0e87
9d27
ac0a
5c92
001f
e5b6
8e2e
7293
5e67
ba26
95cf
95cf
427e
4406
3c16
db32
8e2e
c4b0
4305
d5a6
ba26
95cf
95cf
427e
48cd
3c16
e7d0
8e2e
c4b0
d1ba
67f3

```

```

ba26     pull(o);
95cf     }
95cf     }
427e
0f62     LL query(int o, int ll, int rr) {
3c16         if (ll <= l[o] && r[o] <= rr) {
6dfe             return val[o];
8e2e         } else {
c4b0             push(o);
462a             if (rr <= m[o]) return query(lson, ll, rr);
5cca             if (ll >= m[o]) return query(rson, ll, rr);
bbf9             return query(lson, ll, rr) + query(rson, ll, rr);
95cf         }
95cf     }
4d99 } seg;

```

6.4 Treap

Self-balanced binary search tree which supports split and merge.

Usage:

push(x)	Push lazy tags to children.
pull(x)	Update statistics of node x .
Init(x, v)	Initialize node x with value v .
Add(x, v)	Apply addition to subtree x .
Reverse(x)	Apply reversion to subtree x .
Merge(x, y)	Merge trees rooted at x and y . Return the root of new tree.
Split(t, k, x, y)	Split out the left k elements of tree t . The roots of left part and right part are stored in x and y , respectively.
init(n)	Initialize the treap with array of size n .
work(op, l, r)	Range operation over $[l, r)$.

Time Complexity: Expected $O(\log n)$ per operation.

```

9f60     const int MAXN = 200005;
a7c5     mt19937 gen(time(NULL));
9542     struct Treap {
6d61         int ch[MAXN][2];
3948         int sz[MAXN], key[MAXN], val[MAXN];
5d9a         int add[MAXN], rev[MAXN];
2b1b         LL sum[MAXN] = {0};
a773         int maxv[MAXN] = {INT_MIN}, minv[MAXN] = {INT_MAX};
427e
a629     void Init(int x, int v) {

```

```

ch[x][0] = ch[x][1] = 0;
key[x] = gen(); val[x] = v; pull(x);
}

void pull(int x) {
    sz[x] = 1 + sz[ch[x][0]] + sz[ch[x][1]];
    sum[x] = val[x] + sum[ch[x][0]] + sum[ch[x][1]];
    maxv[x] = max({val[x], maxv[ch[x][0]], maxv[ch[x][1]]});
    minv[x] = min({val[x], minv[ch[x][0]], minv[ch[x][1]]});
}

void Add(int x, int a) {
    val[x] += a; add[x] += a;
    sum[x] += LL(sz[x]) * a; maxv[x] += a; minv[x] += a;
}

```

```

void Reverse(int x) {
    rev[x] ^= 1;
    swap(ch[x][0], ch[x][1]);
}

```

```

void push(int x) {
    for (int c : ch[x]) if (c) {
        Add(c, add[x]);
        if (rev[x]) Reverse(c);
    }
    add[x] = 0; rev[x] = 0;
}

```

```

int Merge(int x, int y) {
    if (!x || !y) return x | y;
    push(x); push(y);
    if (key[x] > key[y]) {
        ch[x][1] = Merge(ch[x][1], y); pull(x); return x;
    } else {
        ch[y][0] = Merge(x, ch[y][0]); pull(y); return y;
    }
}

```

```

void Split(int t, int k, int &x, int &y) {
    if (t == 0) { x = y = 0; return; }
    push(t);
    if (sz[ch[t][0]] < k) {
        x = t; Split(ch[t][1], k - sz[ch[t][0]] - 1, ch[t][1], y);

```

```

5a00
d8cd
95cf
427e
3bf9
e1c3
99f8
94e9
6bb9
95cf
427e
8c8e
a7b1
832a
95cf
427e
aaf6
52c6
7850
95cf
427e
1a53
5fe5
fd76
7a53
95cf
49ee
95cf
427e
9d2c
1b09
cd7e
bfffa
a3df
8e2e
bf9e
95cf
95cf
427e
dc7e
6303
f26b
3465
ffd8

```



```

8e2e         } else {
8a23             y = t; Split(ch[t][0], k, x, ch[t][0]);
95cf         }
89e3         if (x) pull(x); if (y) pull(y);
95cf     }
b1f4 } treap;
427e
24b6 int root;
427e
d34f void init(int n) {
34d7     Rep (i, n) {
7681         int x; scanf("%d", &x);
0ed8         treap.Init(i, x);
bcc8         root = (i == 1) ? 1 : treap.Merge(root, i);
95cf     }
95cf }
427e
d030 void work(int op, int l, int r) {
6639     int tl, tm, tr;
b6c4     treap.Split(root, l, tl, tm);
8de3     treap.Split(tm, r - 1, tm, tr);
3658     if (op == 1) {
c039         int x; scanf("%d", &x); treap.Add(tm, x);
1dcb     } else if (op == 2) {
ae78         treap.Reverse(tm);
581d     } else if (op == 3) {
e092         printf("%lld_%d_%d\n",
867f             treap.sum[tm], treap.minv[tm], treap.maxv[tm]);
95cf     }
6188     root = treap.Merge(treap.Merge(tl, tm), tr);
95cf }

```

6.5 Link/cut tree

Dynamic connectivity of undirected acyclic graph. Support single-vertex update, path aggregation and relative LCA query. Vertices are numbered from 1. Zero initialization is enough except for the statistic information.

Usage:

pull(x)	Update statistics of node x .
Root(u)	Get the root of tree where vertex u is in.
Link(u, v)	Link two unconnected trees.
Cut(u, v)	Cut an existent edge.
Query(u, v)	Path aggregation.
Update(u, x)	Single point modification.
LCA(u, v, root)	Get the lowest common ancestor of u and v in tree rooted at root.

Time Complexity: $O(\log n)$ per operation

```

const int MAXN = 1000005;
struct LCT {
    int fa[MAXN], ch[MAXN][2], val[MAXN], sum[MAXN];
    bool rev[MAXN];

    bool isroot(int x) { return ch[fa[x]][0] == x || ch[fa[x]][1] == x; }
    void pull(int x) { sum[x] = val[x] ^ sum[ch[x][0]] ^ sum[ch[x][1]]; }
    void reverse(int x) { swap(ch[x][0], ch[x][1]); rev[x] ^= 1; }
    void push(int x) {
        if (rev[x]) rep (i, 2) if (ch[x][i]) reverse(ch[x][i]); rev[x] = 0;
    }
    void rotate(int x) {
        int y = fa[x], z = fa[y], k = ch[y][1] == x, w = ch[x][!k];
        if (isroot(y)) ch[z][ch[z][1] == y] = x;
        ch[x][!k] = y; ch[y][k] = w; if (w) fa[w] = y;
        fa[y] = x; fa[x] = z; pull(y);
    }
    void pushall(int x) { if (isroot(x)) pushall(fa[x]); push(x); }
    void splay(int x) {
        int y = x, z = 0;
        for (pushall(y); isroot(x); rotate(x)) {
            y = fa[x]; z = fa[y];
            if (isroot(y)) rotate((ch[y][0] == x) ^ (ch[z][0] == y) ? x : y);
        }
        pull(x);
    }
    void access(int x) {
        int z = x;
        for (int y = 0; x; x = fa[y = x]) { splay(x); ch[x][1] = y; pull(x); }
        splay(z);
    }
    void chroot(int x) { access(x); reverse(x); }
    void split(int x, int y) { chroot(x); access(y); }
}

```

2e73
ca06
6a6d
c6e1
427e
eba3
f19f
1c4d
1a53
89a0
95cf
425f
51af
e1fe
1e6f
6d09
95cf
52c6
f69c
d095
c494
ceef
4449
95cf
78a0
95cf
6229
1548
8854
7afd
95cf
a067
126d
427e

```

d87a     int Root(int x) {
f4f1         for (access(x); ch[x][0]; x = ch[x][0]) push(x);
0d77         splay(x); return x;
95cf     }
9e46     void Link(int u, int v) { chroot(u); fa[u] = v; }
7c10     void Cut(int u, int v) { split(u, v); fa[u] = ch[v][0] = 0; pull(v); }
0691     int Query(int u, int v) { split(u, v); return sum[v]; }
a999     void Update(int u, int x) { splay(u); val[u] = x; }
1f42     int LCA(int x, int y, int root) {
6cb2         chroot(root); access(x); splay(y);
02e5         while (fa[y]) splay(y = fa[y]);
c218         return y;
95cf     }
329b };

```

6.6 Balanced binary search tree from pb_ds

```

0475 #include <ext/pb_ds/assoc_container.hpp>
332d using namespace __gnu_pbds;
427e
43a7 tree<int, null_type, less<int>, rb_tree_tag, tree_order_statistics_node_update>
    rkt;
427e // null_tree_node_update
427e
427e // SAMPLE USAGE
190e rkt.insert(x);           // insert element
05d4 rkt.erase(x);          // erase element
add5 rkt.order_of_key(x);    // obtain the number of elements less than x
b064 rkt.find_by_order(i);    // iterator to i-th (numbered from 0) smallest element
c103 rkt.lower_bound(x);
4ff4 rkt.upper_bound(x);
b19b rkt.join(rkt2);          // merge tree (only if their ranges do not intersect)
cb47 rkt.split(x, rkt2);       // split all elements greater than x to rkt2

```

6.7 Persistent segment tree, range k-th query

```

f1a7 struct node {
2ff6     static int n, pos;
427e
7cec     int value;
70e2     node *left, *right;

```

```

void* operator new(size_t size);

```

```

static node* Build(int l, int r) {
    node* a = new node;
    if (r > l + 1) {
        int mid = (l + r) / 2;
        a->left = Build(l, mid);
        a->right = Build(mid, r);
    } else {
        a->value = 0;
    }
    return a;
}

```

```

static node* init(int size) {
    n = size;
    pos = 0;
    return Build(0, n);
}

```

```

static int Query(node* lt, node *rt, int l, int r, int k) {
    if (r == l + 1) return l;
    int mid = (l + r) / 2;
    if (rt->left->value - lt->left->value < k) {
        k -= rt->left->value - lt->left->value;
        return Query(lt->right, rt->right, mid, r, k);
    } else {
        return Query(lt->left, rt->left, l, mid, k);
    }
}

```

```

static int query(node* lt, node *rt, int k) {
    return Query(lt, rt, 0, n, k);
}

```

```

node *Inc(int l, int r, int pos) const {
    node* a = new node(*this);
    if (r > l + 1) {
        int mid = (l + r) / 2;
        if (pos < mid)
            a->left = left->Inc(l, mid, pos);
        else
            a->right = right->Inc(mid, r, pos);
    }
}

```

427e
20b0
427e
3dc0
b6c5
ce96
181e
3ba2
8aaf
8e2e
bfc4
95cf
5ffd
95cf
427e
5a45
2c46
7ee3
be52
95cf
427e
93c0
d30c
181e
cb5a
8edb
2412
8e2e
0119
95cf
95cf
427e
c9ad
9e27
95cf
427e
b19c
5794
ce96
181e
203d
f44a
649a
1024

```

95cf    }
2b3e    a->value++;
5ffd    return a;
95cf    }
427e
e80f    node *inc(int index) {
c246        return Inc(0, n, index);
95cf    }
865a    } nodes[8000000];
427e
99ce    int node::n, node::pos;
1987    inline void* node::operator new(size_t size) {
bb3c        return nodes + (pos++);
95cf    }

```

6.8 Block list

All indices are 0-based. All ranges are left-closed right-open.

Usage:

block::fix()	Apply tags to the current block.
Init(l, r)	Range initializer.
Reverse(l, r)	Reverse the range.
Add(l, r, x)	Add x to the range.
Query(l, r)	Range aggregation.

```

fd9e    const int BLOCK = 800;
76b3    typedef vector<int> vi;
427e
a771    struct block {
8fbc        vi data;
e3b5        LL sum; int minv, maxv;
41db        int add; bool rev;
427e
d7eb        block(vi&& vec) : data(move(vec)),
1f0c            sum(accumulate(range(data), 0ll)),
8216            minv(*min_element(range(data))),
527d            maxv(*max_element(range(data))),
6437            add(0), rev(0) { }
427e
b919        void fix() {
0694            if (rev) reverse(range(data));          rev = 0;
0527            if (add) for (int& x : data) x += add;    add = 0;
95cf        }

```

```

427e    void merge(block& another) {
8bc4        fix(); another.fix();
b895        vi temp(move(data));
f516        temp.insert(temp.end(), range(another.data));
d02c        *this = block(move(temp));
88ea    }
95cf
427e    block split(int pos) {
42e8        fix();
3e79        block result(vi(data.begin() + pos, data.end()));
ccab        data.resize(pos); *this = block(move(data));
861a        return result;
56b0    }
95cf
329b    };
427e
2a18    typedef list<block>::iterator lit;
427e
ce14    struct blocklist {
5540        list<block> blk;
427e
7b8e        void maintain() {
3131            lit it = blk.begin();
4628            while (it != blk.end() && next(it) != blk.end()) {
852d                lit it2 = it;
188c                while (next(it2) != blk.end() &&
3600                    it2->data.size() + next(it2)->data.size() <= BLOCK) {
93e1                    it2->merge(*next(it2));
e1fa                    blk.erase(next(it2));
95cf                }
5771                ++it;
95cf            }
95cf        }
427e
b7b3        lit split(int pos) {
2273            for (lit it = blk.begin(); ; it++) {
5502                if (pos == 0) return it;
8e85                while (it->data.size() > pos)
2099                    blk.insert(next(it), it->split(pos));
a5a1                pos -= it->data.size();
427e            }
95cf        }
95cf    }
427e

```

```

1c7b void Init(int *l, int *r) {
9919     for (int *cur = l; cur < r; cur += BLOCK)
8950         blk.emplace_back(vi(cur, min(cur + BLOCK, r)));
95cf }
427e
a22f void Reverse(int l, int r) {
997b     lit it = split(l), it2 = split(r);
dfd0     reverse(it, it2);
8f89     while (it != it2) {
6a06         it->rev ^= 1;
5283         it++;
95cf     }
b204     maintain();
95cf }
427e
3cce void Add(int l, int r, int x) {
997b     lit it = split(l), it2 = split(r);
8f89     while (it != it2) {
e927         it->sum += LL(x) * it->data.size();
03d3         it->minv += x; it->maxv += x;
4511         it->add += x; it++;
95cf     }
b204     maintain();
95cf }
427e
3ad3 void Query(int l, int r) {
997b     lit it = split(l), it2 = split(r);
c33d     LL sum = 0; int minv = INT_MAX, maxv = INT_MIN;
8f89     while (it != it2) {
e472         sum += it->sum;
72c4         minv = min(minv, it->minv);
e1c4         maxv = max(maxv, it->maxv);
5283         it++;
95cf     }
b204     maintain();
8792     printf("%lld_%d_%d\n", sum, minv, maxv);
95cf }
958e } lst;

```

6.9 Persistent block list

Block list that supports persistence. All indices are 0-based. All ranges are left-closed right-open. `std::shared_ptr` is used to ease memory management. One should modify

the constructor of `block` to maintain extra information. Here we use this policy that the size of each block does not exceed `BLOCK`, while the sum of sizes of two adjacent blocks does not less than `BLOCK`.

When some operation that breaks block list property, please call `maintain` in time to restore the property.

Usage:

<code>maintain()</code>	Maintain the block list property.
<code>split(pos)</code>	Split the block list at position <code>pos</code> . Returns an iterator to a block starting at <code>pos</code> .
<code>sum(l, r)</code>	An example function of list traversal between $[l, r)$.

Time Complexity: When `BLOCK` is properly selected, the time complexity is $O(\sqrt{n})$ per operation.

```

constexpr int BLOCK = 800;
typedef vector<int> vi;
typedef shared_ptr<vi> pvi;
typedef shared_ptr<const vi> pcvi;

struct block {
    pcvi data;
    LL sum;

    // add information to maintain
    block(pcvi ptr) :
        data(ptr),
        sum(accumulate(ptr->begin(), ptr->end(), 0ll))
    { }

    void merge(const block& another) {
        pvi temp = make_shared<vi>(data->begin(), data->end());
        temp->insert(temp->end(), another.data->begin(), another.data->end());
        *this = block(temp);
    }

    block split(int pos) {
        block result(make_shared<vi>(data->begin() + pos, data->end()));
        *this = block(make_shared<vi>(data->begin(), data->begin() + pos));
        return result;
    }
};

```

```
typedef list<block>::iterator lit;
```

```

ce14 struct blocklist {
5540     list<block> blk;
427e
7b8e     void maintain() {
3131         lit it = blk.begin();
5e44         while (it != blk.end() and next(it) != blk.end()) {
852d             lit it2 = it;
0b03             while (next(it2) != blk.end() and
029f                 it2->data->size() + next(it2)->data->size() <= BLOCK) {
93e1                 it2->merge(*next(it2));
e1fa                 blk.erase(next(it2));
95cf             }
5771             ++it;
95cf         }
95cf     }
427e
b7b3     lit split(int pos) {
2273         for (lit it = blk.begin(); ; it++) {
5502             if (pos == 0) return it;
d480             while (it->data->size() > pos) {
2099                 blk.insert(next(it), it->split(pos));
95cf             }
a1c8             pos -= it->data->size();
95cf         }
95cf     }
427e
fd38     LL sum(int l, int r) { // traverse
48b4         lit it1 = split(l), it2 = split(r);
ac09         LL res = 0;
9f1d         while (it1 != it2) {
8284             res += it1->sum;
61fd             it1++;
95cf         }
b204         maintain();
244d         return res;
95cf     }
329b };

```

6.10 Sparse table, range extremum query

The array is 0-based and the range is closed.

```
db63 const int MAXN = 100007;
```

```

int a[MAXN];
int st[MAXN][32 - __builtin_clz(MAXN)];

inline int ext(int x, int y){return x>y?x:y;} // ! max

void init(int n){
    int l = 31 - __builtin_clz(n);
    rep (i, n) st[i][0] = a[i];
    rep (j, l)
        rep (i, 1+n-(1<<j))
            st[i][j+1] = ext(st[i][j], st[i+(1<<j)][j]);
}

int rmq(int l, int r){
    int k = 31 - __builtin_clz(r-l+1);
    return ext(st[l][k], st[r-(1<<k)+1][k]);
}

```

7 Geometrics

7.1 2D geometric template

```

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

typedef int T;
typedef struct pt {
    T x, y;
    T operator , (pt a) { return x*a.x + y*a.y; } // inner product
    T operator * (pt a) { return x*a.y - y*a.x; } // outer product
    pt operator + (pt a) { return {x+a.x, y+a.y}; }
    pt operator - (pt a) { return {x-a.x, y-a.y}; }

    pt operator * (T k) { return {x*k, y*k}; }
    pt operator - () { return {-x, -y}; }
} vec;

typedef pair<pt, pt> seg;

bool ptOnSeg(pt& p, seg& s){
    vec v1 = s.first - p, v2 = s.second - p;

```

b330
69ae
427e
8041
427e
d34f
ce01
cf75
b811
6937
082a
95cf
427e
c863
92f5
baa2
95cf

302f
421c
427e
4553
c0ae
7a9d
ffaa
3ec7
221a
8b34
427e
368b
90f4
ba8c
427e
0ea6
427e
8d6e
ce77

```

de97     return (v1, v2) <= 0 && v1 * v2 == 0;
95cf }
427e
427e // 0 not on segment
427e // 1 on segment except vertices
427e // 2 on vertices
8421 int ptOnSeg2(pt& p, seg& s){
ce77     vec v1 = s.first - p, v2 = s.second - p;
70ca     T ip = (v1, v2);
8b14     if (v1 * v2 != 0 || ip > 0) return 0;
0847     return (v1, v2) ? 1 : 2;
95cf }
427e
427e // if two orthogonal rectangles do not touch, return true
72bb inline bool nIntRectRect(seg a, seg b){
f9ac     return min(a.first.x, a.second.x) > max(b.first.x, b.second.x) ||
f486         min(a.first.y, a.second.y) > max(b.first.y, b.second.y) ||
39ce         min(b.first.x, b.second.x) > max(a.first.x, a.second.x) ||
80c7         min(b.first.y, b.second.y) > max(a.first.y, a.second.y);
95cf }
427e
427e // >0 in order
427e // <0 out of order
427e // =0 not standard
7538 inline double rotOrder(vec a, vec b, vec c){return double(a*b)*(b*c);}
427e
31ed inline bool intersect(seg a, seg b){
427e     // ! if (nIntRectRect(a, b)) return false; // if commented, assume that a
        and b are non-collinear
cb52     return rotOrder(b.first-a.first, a.second-a.first, b.second-a.first) >= 0 &&
059e         rotOrder(a.first-b.first, b.second-b.first, a.second-b.first) >= 0;
95cf }
427e
427e // 0 not intersect
427e // 1 standard intersection
427e // 2 vertex-line intersection
427e // 3 vertex-vertex intersection
427e // 4 collinear and have common point(s)
4d19 int intersect2(seg& a, seg& b){
5dc4     if (nIntRectRect(a, b)) return 0;
42c0     vec va = a.second - a.first, vb = b.second - b.first;
2096     double j1 = rotOrder(b.first-a.first, va, b.second-a.first),
72fe         j2 = rotOrder(a.first-b.first, vb, a.second-b.first);
5ac6     if (j1 < 0 || j2 < 0) return 0;

```

```

if (j1 != 0 && j2 != 0) return 1;
if (j1 == 0 && j2 == 0){
    if (va * vb == 0) return 4; else return 3;
} else return 2;
}

template <typename Tp = T>
inline pt getIntersection(pt P, vec v, pt Q, vec w){
    static_assert(is_same<Tp, double>::value, "must_be_double!");
    return P + v * (w*(P-Q)/(v*w));
}

// -1 outside the polygon
// 0 on the border of the polygon
// 1 inside the polygon
int ptOnPoly(pt p, pt* poly, int n){
    int wn = 0;
    for (int i = 0; i < n; i++) {
        T k, d1 = poly[i].y - p.y, d2 = poly[(i+1)%n].y - p.y;
        if (k = (poly[(i+1)%n] - poly[i])*(p - poly[i])){
            if (k > 0 && d1 <= 0 && d2 > 0) wn++;
            if (k < 0 && d2 <= 0 && d1 > 0) wn--;
        } else return 0;
    }
    return wn ? 1 : -1;
}

istream& operator >> (istream& lhs, pt& rhs){
    lhs >> rhs.x >> rhs.y;
    return lhs;
}

istream& operator >> (istream& lhs, seg& rhs){
    lhs >> rhs.first >> rhs.second;
    return lhs;
}

```

9400
83db
6b0c
fb17
95cf
427e
2c68
5894
6850
7c9a
95cf
427e
427e
427e
427e
cbdd
5fb4
1294
427e
3cae
b957
8c40
3c4d
aad3
95cf
0a5f
95cf
427e
d4a3
fa86
331a
95cf
427e
07ae
5cab
331a
95cf

8 Appendices

8.1 Primes

8.1.1 First primes

p	$g(p)$	p	$g(p)$	p	$g(p)$	p	$g(p)$	p	$g(p)$
2	1	3	2	5	2	7	3	11	2
13	2	17	3	19	2	23	5	29	2
31	3	37	2	41	6	43	3	47	5
53	2	59	2	61	2	67	2	71	7
73	5	79	3	83	2	89	3	97	5
101	2	103	5	107	2	109	6	113	3
127	3	131	2	137	3	139	2	149	2
151	6	157	5	163	2	167	5	173	2
179	2	181	2	191	19	193	5	197	2
199	3	211	2	223	3	227	2	229	6

8.1.2 Arbitrary length primes

$\lg p$	p	$g(p)$	p	$g(p)$
3	967	5	1031	14
4	9859	2	10273	10
5	96331	10	102931	3
6	958543	6	1031137	5
7	9594539	2	10169651	2
8	96243449	3	103211039	7
9	980483981	2	1042484357	2
10	9858935453	2	10261276009	7
11	95748666809	3	101759940101	2
12	950781833849	3	1012797784423	5
13	9739822952371	7	10037217092377	7
14	96181051140397	5	104974966380359	11
15	981030138360889	13	1029038416465403	2
16	9655206098080843	3	10116299875820773	2
17	97687777921994419	3	101506415998163437	2

8.1.3 $\sim 1 \times 10^9$

p	$g(p)$	p	$g(p)$	p	$g(p)$
954854573	3	967607731	2	973215833	3
975831713	3	978949117	2	980766497	3
983879921	3	985918807	3	986608921	29
991136977	5	991752599	13	997137961	11
1003911991	3	1009775293	2	1012423549	6
1021000537	5	1023976897	7	1024153643	2
1037027287	3	1038812881	11	1044754639	3
1045125617	3	1047411427	3	1047753349	6

8.1.4 $\sim 1 \times 10^{18}$

p	$g(p)$	p	$g(p)$
951970612352230049	3	963284339889659609	3
967495386904694119	3	969751761517096213	2
983238274281901499	2	984647442475101409	23
989286107138674069	11	1002507954383424641	3
1006658951440146419	2	1020152326159075903	3
1034876265966119449	7	1042753851435034019	2
1043609016597371563	2	1045571042176595707	2
1048364250160580293	2	1049495624119026949	2

8.2 Pell's equation

$x^2 - ny^2 = 1$, where n is a positive nonsquare integer.

Let (x_0, y_0) be the smallest positive solution of the equation, then the k -th solution is:

$$\begin{pmatrix} x_k \\ y_k \end{pmatrix} = \begin{pmatrix} x_0 & ny_0 \\ y_0 & x_0 \end{pmatrix}^k \begin{pmatrix} x_0 \\ y_0 \end{pmatrix}$$

Some smallest solutions to Pell's equation:

n	2	3	5	6	7	8	10	11	12	13	14	15	17	18	19	20
x	3	2	9	5	8	3	19	10	7	649	15	4	33	17	170	9
y	2	1	4	2	3	1	6	3	2	180	4	1	8	4	39	2

8.3 Burnside's lemma and Polya's enumeration theorem

The Burnside's lemma says that

$$|X/G| = \frac{1}{|G|} \sum_{g \in G} |X^g|$$

where G is a group acting on X , X^g is the set of elements in X that are fixed by g , i.e. $X^g = \{x \in X : gx = x\}$.

The unweighted version of Pólya enumeration theorem says that

$$|Y^X/G| = \frac{1}{|G|} \sum_{g \in G} m^{c_g}$$

where $m = |X|$ is the number of colors, c_g is the number of the cycles of permutation g .

8.4 Lagrange's interpolation

For sample points $(x_0, y_0), \dots, (x_k, y_k)$, define

$$l_j(x) = \prod_{0 \leq m \leq k, m \neq j} \frac{x - x_m}{x_j - x_m}$$

then the Lagrange polynomial is

$$L(x) = \sum_{j=0}^k y_j l_j(x).$$

To use the script below, type two lines

```
x0 x1 x2 ... xn
y0 y1 y2 ... yn
```

the script will print the fractional coefficient of the polynomial in ascending exponent order.

```
#!/usr/bin/python2
from fractions import *

def polyadd(a, b) : return map(lambda x, y : (x or 0) + (y or 0), a, b)

def polymul(a, b) :
    p = [0] * (len(a)+len(b)-1)
    for e1, c1 in enumerate(a) :
        for e2, c2 in enumerate(b) :
            p[e1 + e2] += c1 * c2
    return p

x, y = [map(Fraction, raw_input().split()) for _ in 0,0]
n = len(x)
lj = [reduce(polymul, [[-x[m]/(x[j]-x[m]), 1/(x[j]-x[m])]
    for m in range(n) if m != j]) for j in range(n)]
print ' '.join(map(str, reduce(polyadd,
    map(lambda a, b : [x * a for x in b], y, lj))))
```

6dc9
4b2b
427e
bbbe
427e
796b
83e4
f697
156c
dfce
5849
427e
f06d
e80a
a649
9dfa
46f9
d754