

# 南京大学 ACM-ICPC 集训队代码模版库



Linux-4.15.0-65-generic-x86\_64-with-Ubuntu-18.04-bionic  
XeTeX 3.14159265-2.6-0.99998 (TeX Live 2017/Debian)  
CPython 2.7.15+  
2019-10-23 09:15:48.758386, build 0051

## Contents

<b>1 General</b>	<b>3</b>	5.4 Minimum spanning arborescence, slow	15
1.1 Code library checksum	3	5.5 Maximum flow (Dinic)	15
1.2 Makefile	3	5.6 Maximum cardinality bipartite matching (Hungarian)	16
1.3 .vimrc	3	5.7 Maximum matching of general graph (Edmond's blossom)	17
1.4 Stack	3	5.8 Minimum cost maximum flow	18
1.5 Template	3	5.9 Fast LCA	19
<b>2 Miscellaneous Algorithms</b>	<b>4</b>	5.10 Heavy-light decomposition	19
2.1 2-SAT	4	5.11 Centroid decomposition	20
2.2 Knuth's optimization	4	5.12 DSU on tree	20
2.3 Mo's algorithm	5	<b>6 Data Structures</b>	<b>21</b>
<b>3 String</b>	<b>5</b>	6.1 Fenwick tree (point update range query)	21
3.1 Knuth-Morris-Pratt algorithm	5	6.2 Fenwick tree (range update point query)	21
3.2 Manacher algorithm	6	6.3 Segment tree	22
3.3 Aho-corasick automaton	6	6.4 Treap	23
3.4 Trie	7	6.5 Link/cut tree	24
3.5 Suffix array	7	6.6 Balanced binary search tree from pb_ds	24
3.6 Rolling hash	8	6.7 Persistent segment tree, range k-th query	25
<b>4 Math</b>	<b>8</b>	6.8 Block list	26
4.1 Extended Euclidean algorithm and Chinese remainder theorem	8	6.9 Persistent block list	27
4.2 Linear basis	9	6.10 Sparse table, range minimum query	28
4.3 Gauss elimination over finite field	9	<b>7 Geometrics</b>	<b>28</b>
4.4 Berlekamp-Massey algorithm	10	7.1 2D geometric template	28
4.5 Fast Walsh-Hadamard transform	10	<b>8 Appendices</b>	<b>30</b>
4.6 Fast fourier transform	10	8.1 Number theory	30
4.7 Number theoretic transform	11	8.1.1 First primes	30
4.8 Sieve of Euler	11	8.1.2 Arbitrary length primes	30
4.9 Sieve of Euler (General)	12	8.1.3 $\sim 1 \times 10^9$	30
4.10 Miller-Rabin primality test	12	8.1.4 $\sim 1 \times 10^{18}$	30
4.11 Integer factorization (Pollard's rho)	13	8.2 Pell's equation	30
<b>5 Graph Theory</b>	<b>13</b>	8.3 Maximum number of divisors of $n$ -digit number	31
5.1 Strongly connected components	13	8.4 Burnside's lemma and Polya's enumeration theorem	31
5.2 Vertex biconnected components, cut vertex	13	8.5 Lagrange's interpolation	31
5.3 Minimum spanning arborescence, faster	14		

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

Solve  $ax + by = g = \gcd(a, b)$  w.r.t.  $x, y$ .

If  $(x_0, y_0)$  is an integer solution of  $ax + by = g = \gcd(x, y)$ , then every integer solution of it can be written as  $(x_0 + kb', y_0 - ka')$ , where  $a' = a/g$ ,  $b' = b/g$ , and  $k$  is arbitrary integer.

```

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);
    }
}

```



```

3cd3     ret = (ret + y * m * r[i]) % q;
95cf     }
2e47     return (q + ret) % q;
95cf     }

```

## 4.2 Linear basis

```

8b44     const int MAXD = 30;
03a6     struct linearbasis {
3558         ULL b[MAXD] = {};
427e
1566         bool insert(LL v) {
9b2b             for (int j = MAXD - 1; j >= 0; j--) {
de36                 if (!(v & (1ll << j))) continue;
ee78                 if (b[j] & v) b[j] ^= v;
037f                 else {
7836                     for (int k = 0; k < j; k++)
f0b4                         if (v & (1ll << k)) v ^= b[k];
b0aa                     for (int k = j + 1; k < MAXD; k++)
46c9                         if (b[k] & (1ll << j)) b[k] ^= v;
8295                     b[j] = v;
3361                     return true;
95cf                 }
95cf             }
438e             return false;
95cf         }
329b     };

```

## 4.3 Gauss elimination over finite field

```

b784     const LL p = 1000000007;
427e
2a2c     LL powmod(LL b, LL e) {
95a2         LL r = 1;
3e90         while (e) {
1783             if (e & 1) r = r * b % p;
5549             b = b * b % p;
16fc             e >>= 1;
95cf         }
547e         return r;
95cf     }

```

```

typedef vector<LL> VLL;
typedef vector<VLL> VLLL;

```

```

LL gauss(VLLL &a, VLLL &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])
            rep (k, n) if (!ipiv[k])
                if (pj == -1 || a[j][k] > a[pj][pk]) {
                    pj = j;
                    pk = k;
                }
        if (a[pj][pk] == 0) return 0;
        ipiv[pk]++;
        swap(a[pj], a[pk]);
        swap(b[pj], b[pk]);
        if (pj != pk) det = (p - det) % p;
        irow[i] = pj;
        icol[i] = pk;

```

```

        LL c = powmod(a[pk][pk], p - 2);
        det = det * a[pk][pk] % p;
        a[pk][pk] = 1;
        rep (j, n) a[pk][j] = a[pk][j] * c % p;
        rep (j, m) b[pk][j] = b[pk][j] * c % p;
        rep (j, n) if (j != pk) {
            c = a[j][pk];
            a[j][pk] = 0;
            rep (k, n) a[j][k] = (a[j][k] + p - a[pk][k] * c % p) % p;
            rep (k, m) b[j][k] = (b[j][k] + p - b[pk][k] * c % p) % p;
        }
    }

```

```

    for (int j = n - 1; j >= 0; j--) if (irow[j] != icol[j]) {
        for (int k = 0; k < n; k++) swap(a[k][irow[j]], a[k][icol[j]]);
    }
    return det;
}

```

427e  
c130  
42ac  
427e  
2c62  
561b  
a25e  
2976  
427e  
be8e  
d2b5  
6b4a  
e582  
6112  
a905  
657b  
95cf  
d480  
0305  
8dad  
aad8  
be4d  
d080  
f156  
427e  
4ecd  
865b  
c36a  
dd36  
1b23  
f8f3  
e97f  
c449  
820b  
f039  
95cf  
95cf  
427e  
37e1  
50dc  
95cf  
f27f  
95cf

## 4.4 Berlekamp-Massey algorithm

Call `berlekamp()` with input sequence  $(x_0, x_1, \dots, x_{n-1})$ . Return a vector of coefficients  $(c_0 = 1, c_1, \dots, c_{m-1})$  with minimum  $m$ , such that  $\sum_{i=0}^m c_i x_{j-i} = 0$  for all possible  $j$ .

```

6e50 LL mod = 1000000007;
97db vector<LL> berlekamp(const vector<LL>& a) {
8904     vector<LL> p = {1}, r = {1};
075b     LL dif = 1;
8bc9     rep (i, a.size()) {
1b35         LL u = 0;
bd0b         rep (j, p.size()) u = (u + p[j] * a[i-j]) % mod;
eae9         if (u == 0) {
b14c             r.insert(r.begin(), 0);
8e2e         } else {
0c78             auto op = p;
02f6             p.resize(max(p.size(), r.size() + 1));
0a2e             LL idif = powmod(dif, mod - 2);
9b57             rep (j, r.size())
dacc                 p[j+1] = (p[j+1] - r[j] * idif % mod * u % mod + mod) % mod;
bcd1             dif = u; r = op;
95cf         }
95cf     }
e149     return p;
95cf }
```

## 4.5 Fast Walsh-Hadamard transform

```

061e void fwt(int* a, int n){
5595     for (int d = 1; d < n; d <= 1)
05f2         for (int i = 0; i < n; i += d << 1)
b833             rep (j, d){
7796                 int x = a[i+j], y = a[i+j+d];
427e                 // a[i+j] = x+y, a[i+j+d] = x-y; // xor
427e                 // a[i+j] = x+y; // and
427e                 // a[i+j+d] = x+y; // or
95cf             }
95cf }
427e
4db1 void ifwt(int* a, int n){
5595     for (int d = 1; d < n; d <= 1)
05f2         for (int i = 0; i < n; i += d << 1)
b833             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
    }
}

void conv(int* a, int* b, int n){
    fwt(a, n);
    fwt(b, n);
    rep(i, n) a[i] *= b[i];
    ifwt(a, n);
}
```

7796  
427e  
427e  
427e  
95cf  
95cf  
427e  
2ab6  
950a  
e427  
8a42  
430f  
95cf

## 4.6 Fast fourier transform

```

const int NMAX = 1<<20;

typedef complex<double> cplx;

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

4e09  
427e  
3fbf  
427e  
abd1  
12af  
c47c  
27d7  
9827

```

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

427e  
1442  
e209  
b393  
7ba3  
1908  
a166  
95cf  
95cf

```

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

427e  
b941  
a215  
ac6e  
2969  
b3cf  
c24f  
fe06

```

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 }
427e
bdc0 void conv(cplx* a, cplx* b){
6497     fft(a); fft(b);
12a5     rep (i, N) a[i] *= b[i];
f84e     ifft(a);
95cf }
329b };

```

## 4.7 Number theoretic transform

```

4ab9 const int NMAX = 1<<21;
427e
427e // 998244353 = 7*17*2^23+1, G = 3
fb9a const int P = 1004535809, G = 3; // = 479*2^21+1
427e
87ab struct NTT{
c47c     int rev[NMAX];
0eda     LL omega[NMAX], oinv[NMAX];
81af     int g, g_inv; // g: g_n = G^((P-1)/n)
9827     int K, N;
427e
2a2c     LL powmod(LL b, LL e){
95a2         LL r = 1;
3e90         while (e){
6624             if (e&1) r = r * b % P;
489e             b = b * b % P;
16fc             e >>= 1;
95cf         }
547e         return r;
95cf     }
427e
f420 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;
    }
}

void _ntt(LL* a, LL* w){
    rep (i, N) if (i < rev[i]) swap(a[i], a[rev[i]]);
    for (int l = 2; l <= N; l *= 2){
        int m = l/2;
        for (LL* p = a; p != a + N; p += l)
            rep (k, m){
                LL t = w[N/l*k] * p[k+m] % P;
                p[k+m] = (p[k] - t + P) % P;
                p[k] = (p[k] + t) % P;
            }
    }

    void ntt(LL* a){_ntt(a, omega);}
    void intt(LL* a){
        LL inv = powmod(N, P-2);
        _ntt(a, oinv);
        rep (i, N) a[i] = a[i] * inv % P;
    }

    void conv(LL* a, LL* b){
        ntt(a); ntt(b);
        rep (i, N) a[i] = a[i] * b[i] % P;
        intt(a);
    }
};

```

## 4.8 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;
5f51             if (i % prime[j] == 0) break;
95cf         }
95cf     }
95cf }

```

```

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

```

f87a  
20cc  
9985  
3f93  
8e2e  
cc91  
6322  
95cf  
6191  
d614  
95cf  
5f51  
95cf  
95cf  
95cf

## 4.9 Sieve of Euler (General)

```

b62e namespace sieve {
6589     constexpr int MAXN = 10000007;
e982     bool p[MAXN]; // true if not prime
6ae8     int prime[MAXN], sz;
cbf7     int pval[MAXN], pcnt[MAXN];
6030     int f[MAXN];
427e
76f6     void exec(int N = MAXN) {
9628         p[0] = p[1] = 1;
427e
8a8a         pval[1] = 1;
bdda         pcnt[1] = 0;
c6b9         f[1] = 1;
427e
a643         for (int i = 2; i < N; i++) {
01d6             if (!p[i]) {
b2b2                 prime[sz++] = i;
37d9                 for (LL j = i; j < N; j *= i) {
758c                     int b = j / i;
81fd                     pval[j] = i * pval[b];
e0f3                     pcnt[j] = pcnt[b] + 1;
a96c                     f[j] = _____; // f[j] = f(i^pcnt[j])
95cf                 }
95cf             }
34c0         }
for (int j = 0; i * prime[j] < N; j++) {

```

## 4.10 Miller-Rabin primality test

The array `a[]` (excluding sentinel, 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 $n < 2^{64}$ .

```

bool test(LL n){
    if (n < 3) return n==2;
    // ! The array a[] should be modified if the range of x changes.
    const LL a[] = {2LL, 7LL, 61LL, LLONG_MAX};
    LL r = 0, d = n-1, x;
    while (~d & 1) d >>= 1, r++;
    for (int i=0; a[i] < n; i++){
        x = powmod(a[i], d, n); // ! powmod must use for 64bit mulmod
        if (x == 1 || x == n-1) goto next;
        rep (i, r) {
            x = mulmod(x, x, n);
            if (x == n-1) goto next;
        }
        return false;
    }
next:;
}
return true;

```

f16f  
59f2  
427e  
3f11  
c320  
f410  
2975  
ece1  
7f99  
e257  
d7ff  
8d2e  
95cf  
438e  
d490  
95cf  
3361

95cf }

## 4.11 Integer factorization (Pollard's rho)

```

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;
d3e5     if (~n&1) return 2;
3c69     while (d == n){
0964         x = y = 2;
4753         d = 1;
5952         c = rand() % (n - 1) + 1;
9e5b         while (d == 1){
33d5             x = (mulmod(x, x, n) + c) % n;
e1bf             y = (mulmod(y, y, n) + c) % n;
e1bf             y = (mulmod(y, y, n) + c) % n;
a313             d = gcd(x>y ? x-y : y-x, n);
95cf         }
95cf     }
5d89     return d;
95cf }
```

## 5 Graph Theory

### 5.1 Strongly connected components

#### Usage:

dfs(u)	Run dfs(u) for each unlabelled vertex.
scc[i]	The vertices of the <i>i</i> -th scc.
sccid[u]	The index of the scc that contains <i>u</i> .
contract()	Compute the contracted graph.

```

0f42 const int MAXN = 100005;
35b8 int n, m;
0b32 vector<int> adj[MAXN];
18e4 int dfn[MAXN], low[MAXN], idx;
589d int sccid[MAXN], sccn;
ac27 vector<int> scc[MAXN];
427e
d714 void dfs(int u) {
```

```

static stack<int> s;
dfn[u] = low[u] = ++idx;
s.push(u);
for (int v : adj[u]) {
    if (!dfn[v]) {
        dfs(v);
        low[u] = min(low[u], low[v]);
    } else if (!sccid[v]) {
        low[u] = min(low[u], dfn[v]);
    }
}
if (dfn[u] == low[u]) {
    sccn++;
    do {
        sccid[s.top()] = sccn;
        scc[sccn].push_back(s.top());
        s.pop();
    } while (scc[sccn].back() != u);
}
}

vector<int> adjc[MAXN];
void contract() {
    Rep (u, n) for (int v : adj[u]) if (sccid[u] != sccid[v])
        adjc[sccid[u]].push_back(sccid[v]);
}
```

```

56b7
9891
80f6
18f6
3c64
5f3c
a19f
50c8
769a
95cf
95cf
4804
660f
a69f
8c0c
c8c7
c2f4
8b07
95cf
95cf
427e
1f52
364d
7cbf
426e
95cf
```

### 5.2 Vertex biconnected components, cut vertex

A component root *u* is a cut vertex iff the size of bccin[*u*] is at least 2; for any other vertice *u*, it is a cut vertex iff bccin[*u*] is nonempty.

#### Usage:

dfs(u)	Run dfs(u) for each connected component.
bcc[i]	The edges of the <i>i</i> -th biconnected components, numbered from 0. If the bcc is a simple cycle, the edges are sorted in order.
bccin[u]	The indices of biconnected components reachable from vertex <i>u</i> .

```

const int MAXN = 100005;
int n, m;
vector<int> adj[MAXN];
int dfn[MAXN], low[MAXN], idx = 0;
```

```

0f42
35b8
0b32
0a8f
```

```

05d2 vector<int> bccin[MAXN];
2eab vector<vector<pair<int, int>>> bcc;
3eed stack<pair<int, int>> st;
427e
6576 void dfs(int u, int p = 0) {
9891     dfn[u] = low[u] = ++idx;
18f6     for (int v : adj[u]) {
3c64         if (!dfn[v]) {
c600             st.emplace(u, v);
e2f7             dfs(v, u);
a19f             low[u] = min(low[u], low[v]);
9cb7             if (low[v] >= dfn[u]) {
a0e8                 bccin[u].push_back(bcc.size());
7dc7                 vector<pair<int, int>> cur;
a69f                 do {
bfe3                     cur.push_back(st.top());
b439                     st.pop();
5f33                 } while (cur.back() != make_pair(u, v));
b854                 reverse(range(cur));
0c6c                 bcc.push_back(move(cur));
95cf             }
dddc         } else if (dfn[v] < dfn[u] and v != p) {
c600             st.emplace(u, v);
769a             low[u] = min(low[u], dfn[v]);
95cf         }
95cf     }
95cf }

```

### 5.3 Minimum spanning arborescence, faster

All vertices are 1-based. Clear the fields when reuse the struct.

#### Usage:

add\_edge(u, v, w)      Add an edge from *u* to *v* with weight *w*.  
run(n, rt)              Compute the total weight of MSA rooted at *rt*. If not  
                             exist, return LLONG\_MIN.

**Time Complexity:**  $O(|E| \log^2 |V|)$

```

5ece const int MAXN = 300005;
2fef typedef pair<LL, int> pii;
1495 struct MDST {
01b2     priority_queue<pii, vector<pii>, greater<pii>> heap[MAXN];
321d     LL shift[MAXN];

```

```

int fa[MAXN], vis[MAXN];

int find(int x) { return fa[x] == x ? x : fa[x] = find(fa[x]); }

void unite(int x, int y) {
    x = find(x); y = find(y); fa[y] = x; if (x == y) return;
    if (heap[x].size() < heap[y].size()) {
        swap(heap[x], heap[y]);
        swap(shift[x], shift[y]);
    }
    while (heap[y].size()) {
        auto p = heap[y].top(); heap[y].pop();
        heap[x].emplace(p.first - shift[y] + shift[x], p.second);
    }
}

void add_edge(int u, int v, LL w) { heap[v].emplace(w, u); }

LL run(int n, int rt) {
    LL ans = 0;
    iota(fa, fa + n + 1, 0);
    Rep (i, n) if (find(i) != find(rt)) {
        int u = find(i);
        stack<int, vector<int>> s;
        while (find(u) != find(rt)) {
            if (vis[u]) while (s.top() != u) {
                vis[s.top()] = 0; unite(u, s.top()); s.pop();
            } else { vis[u] = 1; s.push(u); }
            while (heap[u].size()) {
                ans += heap[u].top().first - shift[u];
                shift[u] = heap[u].top().first;
                if (find(heap[u].top().second) != u) break;
                heap[u].pop();
            }
            if (heap[u].empty()) return LLONG_MIN;
            u = find(heap[u].top().second);
        }
        while (s.size()) { vis[s.top()] = 0; unite(rt, s.top()); s.pop(); }
    }
    return ans;
}
};

```

## 5.4 Minimum spanning arborescence, slow

All vertices are 1-based. Clear the fields when reuse the struct.

### Usage:

`init(n)` Initialize the structure with  $n$  vertices, indexed from 1.  
`add_edge(u, v, w)` Add an edge from  $u$  to  $v$  with weight  $w$ .  
`run(n, rt)` Compute the total weight of MSA rooted at  $rt$ . If not exist, return `LLONG_MIN`.

**Time Complexity:**  $O(|V|^2)$

```

1495 struct MDST {
3d02     int V;
d48e     LL heap[MAXN][MAXN];
321d     LL shift[MAXN];
fc06     int fa[MAXN], vis[MAXN];

427e
d34f     void init(int n) {
34cc         V = n;
3295         Rep (i, n) Rep (j, n) heap[i][j] = LLONG_MAX / 2;
95cf     }

427e
38dd     int find(int x) { return fa[x] == x ? x : fa[x] = find(fa[x]); }
427e
29b0     void unite(int x, int y) {
0c14         x = find(x); y = find(y); fa[y] = x; if (x == y) return ;
6506         Rep (i, V) heap[x][i] = min(heap[x][i], heap[y][i] - shift[y] + shift[x]
95cf     ];
427e     }

f09c     void add_edge(int u, int v, LL w) { heap[v][u] = min(heap[v][u], w); }
427e
a526     LL run(int n, int rt) {
34cc         V = n;
f7ff         LL ans = 0;
81f2         iota(fa, fa + n + 1, 0);
19b3         Rep (i, n) if (find(i) != find(rt)) {
a7b1             int u = find(i);
010e             stack<int, vector<int>> s;
eff5             while (find(u) != find(rt)) {
0dda                 if (vis[u]) while (s.top() != u) {
c593                     vis[s.top()] = 0; unite(u, s.top()); s.pop();
83c4                 } else { vis[u] = 1; s.push(u); }
427e
6e45         Rep (i, V) if (find(i) == u) heap[u][i] = LLONG_MAX / 2;

```

```

auto ptr = min_element(heap[u] + 1, heap[u] + V + 1);
if (*ptr == LLONG_MAX / 2) return LLONG_MIN;
ans += *ptr - shift[u];
shift[u] = *ptr;

u = ptr - heap[u];
}
while (s.size()) { vis[s.top()] = 0; unite(rt, s.top()); s.pop(); }
}
return ans;
};

```

427e  
02cd  
9ea0  
4e38  
d5c6  
427e  
4264  
95cf  
2d46  
95cf  
4206  
95cf  
329b

## 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,  $O(\sqrt{VE})$ .

```

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

const int MAXN = 1005;
struct Dinic {
    int n, m, s, t;
    vector<edge> edges;
    vector<int> G[MAXN];
    bool vis[MAXN];
    int d[MAXN];
    int cur[MAXN];

    void add_edge(int from, int to, LL cap) {
        edges.push_back(edge{from, to, cap, 0});
        edges.push_back(edge{to, from, 0, 0});
        m = edges.size();
        G[from].push_back(m-2);
        G[to].push_back(m-1);
    }
}

```

bcf8  
60e2  
5e6d  
329b  
427e  
e2cd  
9062  
4dbf  
9f0c  
b891  
bbb6  
b40a  
ddec  
427e  
5973  
7b55  
1db7  
fe77  
dff5  
8f2d  
95cf

```

427e bool bfs() {
1836     memset(vis, 0, sizeof(vis));
3b73     queue<int> q;
93d2     q.push(s);
5d13     vis[s] = 1;
2cd2     d[s] = 0;
721d     while (!q.empty()) {
cc78         int x = q.front(); q.pop();
66ba         for (int i = 0; i < G[x].size(); i++) {
3b61             edge& e = edges[G[x][i]];
b510             if (!vis[e.to] && e.cap > e.flow) {
cd72                 vis[e.to] = 1;
cf26                 d[e.to] = d[x] + 1;
ca93                 q.push(e.to);
95cf             }
95cf         }
95cf     }
b23b     return vis[t];
95cf }

427e LL dfs(int x, LL a) {
9252     if (x == t || a == 0) return a;
6904     LL flow = 0, f;
8bf9     for (int& i = cur[x]; i < G[x].size(); i++) {
f515         edge& e = edges[G[x][i]];
b510         if(d[x] + 1 == d[e.to] && (f = dfs(e.to, min(a, e.cap-e.flow))) > 0)
2374             {
1cce                 e.flow += f;
e16d                 edges[G[x][i]^1].flow -= f;
a74d                 flow += f;
23e5                 a -= f;
97ed                 if(a == 0) break;
95cf             }
95cf     }
84fb     return flow;
95cf }

427e LL max_flow(int s, int t) {
5bf2     this->s = s; this->t = t;
590d     LL flow = 0;
62e2     while (bfs()) {
ed58         memset(cur, 0, sizeof(cur));
f326         flow += dfs(s, LLONG_MAX);
fb3a

```

```

    }
    return flow;
}

vector<int> min_cut() { // call this after maxflow
    vector<int> ans;
    for (int i = 0; i < edges.size(); i++) {
        edge& e = edges[i];
        if(vis[e.from] && !vis[e.to] && e.cap > 0) ans.push_back(i);
    }
    return ans;
}
};

```

## 5.6 Maximum cardinality bipartite matching (Hungarian)

```

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

#define rep(i, n) for (int i = 0; i < (n); i++)
#define Rep(i, n) for (int i = 1; i <= (n); i++)
#define range(x) (x).begin(), (x).end()
typedef long long LL;

struct Hungarian{
    int nx, ny;
    vector<int> mx, my;
    vector<vector<int>> > e;
    vector<bool> mark;

    void init(int nx, int ny){
        this->nx = nx;
        this->ny = ny;
        mx.resize(nx); my.resize(ny);
        e.clear(); e.resize(nx);
        mark.resize(nx);
    }

    inline void add(int a, int b){
        e[a].push_back(b);
    }
}

```



```

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])){
9ca3                     mx[i] = j; my[j] = i;
3361                     return true;
95cf             }
95cf         }
95cf     }
438e     return false;
95cf }
427e
3fac     int match(){
5b57         int ret = 0;
b0f1         fill(range(mx), -1);
b957         fill(range(my), -1);
4ed1         rep (i, nx){
13a5             fill(range(mark), false);
cc89             if (augment(i)) ret++;
95cf         }
ee0f         return ret;
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.

```

c041     const int MAXN = 1024;
6ab1     struct Blossom {
0b32         vector<int> adj[MAXN];
93d2         queue<int> q;
5c83         int n;
0de2         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);
    fill(range(save), 0); fill(range(used), 0);
}

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

void rematch(int x, int y) {
    int m = mate[x]; mate[x] = y;
    if (mate[m] == x) {
        if (label[x] <= n) {
            mate[m] = label[x]; rematch(label[x], m);
        } else {
            int a = 1 + (label[x] - n - 1) / n;
            int b = 1 + (label[x] - n - 1) % n;
            rematch(a, b); rematch(b, a);
        }
    }
}

void traverse(int x) {
    Rep (i, n) save[i] = mate[i];
    rematch(x, x);
    Rep (i, n) {
        if (mate[i] != save[i]) used[i] ++;
        mate[i] = save[i];
    }
}

void relabel(int x, int y) {
    Rep (i, n) used[i] = 0;
    traverse(x); traverse(y);
    Rep (i, n) {
        if (used[i] == 1 and label[i] < 0) {
            label[i] = n + x + (y - 1) * n;
            q.push(i);
        }
    }
}

int solve() {
    Rep (i, n) {

```

427e  
 2186  
 3728  
 477d  
 bb35  
 95cf  
 427e  
 c2dd  
 427e  
 2a48  
 8af8  
 1aa4  
 f4ba  
 740a  
 8e2e  
 3341  
 2885  
 ef33  
 95cf  
 95cf  
 95cf  
 427e  
 8a50  
 43c0  
 2ef7  
 34d7  
 62c5  
 97ef  
 95cf  
 95cf  
 427e  
 8bf8  
 d101  
 c4ea  
 34d7  
 dee9  
 1c22  
 eb31  
 95cf  
 95cf  
 95cf  
 427e  
 a0ce  
 34d7

```

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()) {
66ba         int x = q.front(); q.pop();
b98c         for (int y : adj[x]) {
c07f             if (mate[y] == 0 and i != y) {
7f36                 mate[y] = x; rematch(x, y); q = queue<int>(); break;
95cf             }
d315             if (label[y] >= 0) { relabel(x, y); continue; }
58ec             if (label[mate[y]] < 0) {
c9c4                 label[mate[y]] = x; q.push(mate[y]);
95cf             }
95cf         }
95cf     }
8abb     int cnt = 0;
b52f     Rep (i, n) cnt += (mate[i] > i);
6808     return cnt;
95cf }
329b };

```

## 5.8 Minimum cost maximum flow

```

bcf8 struct edge{
60e2     int from, to;
d698     int cap, flow;
32cc     LL cost;
329b };
427e
cc3e const LL INF = LLONG_MAX / 2;
2aa8 const int MAXN = 5005;
c6cb struct MCMF {
9ceb     int s, t, n, m;
9f0c     vector<edge> edges;
b891     vector<int> G[MAXN];
f74f     bool inq[MAXN]; // queue
8f67     LL d[MAXN];    // distance
9524     int p[MAXN];    // previous
b330     int a[MAXN];    // improvement
427e
f7f2     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);
G[to].push_back(m-1);
}

bool spfa(){
    queue<int> q;
    fill(d, d + MAXN, INF); d[s] = 0;
    memset(inq, 0, sizeof(inq));
    q.push(s); inq[s] = true;
    p[s] = 0; a[s] = INT_MAX;
    while (!q.empty()){
        int u = q.front(); q.pop(); inq[u] = false;
        for (int i : G[u]) {
            edge& e = edges[i];
            if (e.cap > e.flow && d[e.to] > d[u] + e.cost){
                d[e.to] = d[u] + e.cost;
                p[e.to] = G[u][i];
                a[e.to] = min(a[u], e.cap - e.flow);
                if (!inq[e.to]) q.push(e.to), inq[e.to] = true;
            }
        }
    }
    return d[t] != INF;
}

void augment(){
    int u = t;
    while (u != s){
        edges[p[u]].flow += a[t];
        edges[p[u]^1].flow -= a[t];
        u = edges[p[u]].from;
    }
}

#ifdef GIVEN_FLOW
bool min_cost(int s, int t, int f, LL& cost) {
    this->s = s; this->t = t;
    int flow = 0;
    cost = 0;
    while (spfa()) {
        augment();

```

24f0  
95f0  
fe77  
dff5  
8f2d  
95cf  
427e  
3c52  
93d2  
8494  
fd48  
5e7c  
2dae  
cc78  
b0aa  
3bba  
56d8  
3601  
55bc  
0bea  
8249  
e5d3  
95cf  
95cf  
95cf  
6d7c  
95cf  
427e  
71a4  
06f1  
b19d  
db09  
25a9  
e6c9  
95cf  
95cf  
427e  
6e20  
5972  
590d  
21d4  
23cb  
22dc  
bcd b

```

a671         if (flow + a[t] >= f){
b14d             cost += (f - flow) * d[t]; flow = f;
3361             return true;
8e2e         } else {
2a83             flow += a[t]; cost += a[t] * d[t];
95cf         }
95cf     }
438e     return false;
95cf }
a8cb #else
f9a9     int min_cost(int s, int t, LL& cost) {
590d         this->s = s; this->t = t;
21d4         int flow = 0;
23cb         cost = 0;
22dc         while (spfa()) {
bcd8             augment();
2a83             flow += a[t]; cost += a[t] * d[t];
95cf         }
84fb         return flow;
95cf     }
1937 #endif
329b };

```

## 5.9 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 }

```

```

void preprocess(int root) {
    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;
}

```

## 5.10 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){

```

```

8d96     top[x] = t;
d314     id[x] = ++cid;
c4a1     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 }

```

## 5.11 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);

```

```

        sz[u] += sz[v];
    }
}

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
    }
}

```

## 5.12 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];

```

```

8449
95cf
95cf
427e
67f9
d51f
76e4
18e3
81b0
95cf
427e
4662
618e
303c
427e
18f6
427e
95cf
427e
427e
427e
18f6
c375
fa6b
a717
95cf
95cf

```

```

1fb6
901d

```

```

427e void decomp(int u, int p) {
5559     sz[u] = 1;
50c0     for (int v : adj[u]) {
18f6         if (v == p) continue;
bd87         decomp(v, u);
a851         sz[u] += sz[v];
8449         if (sz[v] > sz[son[u]]) son[u] = v;
d28c     }
95cf }
95cf
427e template <typename T>
b7ec void trav(T fn, int u, int p) {
62f5     fn(u);
4412     for (int v : adj[u]) if (v != p) trav(fn, v, u);
30b3 }
95cf
427e #define for_light(v) for (int v : adj[u]) if (v != p and v != son[u])
7467 void work(int u, int p, bool keep) {
33ff     for_light(v) work(v, u, 0); // process light children
72a2
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) { 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  
456d  
427e  
63d0  
f7ff  
6770  
4206  
95cf  
427e  
f4bd  
968e  
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) { tr.resize(N = n + 5);}

    LL query(int n) {
        LL ans = 0;
        while (n < N) { ans += tr[n]; n += n & -n; }
        return ans;
    }

    void add(int n, LL x) {
        while (n) { tr[n] += x; n &= n - 1; }
    }
};

```

3d03  
d7af  
99ff  
427e  
456d  
427e  
38d4  
f7ff  
3667  
4206  
95cf  
427e  
f4bd  
0a2b  
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;
0a90     if (tmul[o] != 1) {
0f4a         push_mul(lson, tmul[o]);
045e         push_mul(rson, tmul[o]);
ac0a         tmul[o] = 1;
95cf     }
1b82     if (tadd[o]) {
9547         push_add(lson, tadd[o]);
0e73         push_add(rson, tadd[o]);
6234         tadd[o] = 0;
95cf     }
95cf }
427e
471c void build(int o, int ll, int rr) {
0e87     int mm = (ll + rr) / 2;
9d27     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);
        pull(o);
    }
}

LL query(int o, int ll, int rr) {
    if (ll <= l[o] && r[o] <= rr) {
        return val[o];
    } else {
        push(o);
        if (rr <= m[o]) return query(lson, ll, rr);
        if (ll >= m[o]) return query(rson, ll, rr);
        return query(lson, ll, rr) + query(rson, ll, rr);
    }
}
} seg;

```

```

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
95cf
95cf
427e
0f62
3c16
6dfe
8e2e
c4b0
462a
5cca
bbf9
95cf
95cf
4d99

```

## 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) {
5a00         ch[x][0] = ch[x][1] = 0;
d8cd         key[x] = gen(); val[x] = v; pull(x);
95cf     }
427e
3bf9     void pull(int x) {
e1c3         sz[x] = 1 + sz[ch[x][0]] + sz[ch[x][1]];
99f8         sum[x] = val[x] + sum[ch[x][0]] + sum[ch[x][1]];
94e9         maxv[x] = max({val[x], maxv[ch[x][0]], maxv[ch[x][1]]});
6bb9         minv[x] = min({val[x], minv[ch[x][0]], minv[ch[x][1]]});
95cf     }
427e
8c8e     void Add(int x, int a) {
a7b1         val[x] += a; add[x] += a;
832a         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);
    } else {
        y = t; Split(ch[t][0], k, x, ch[t][0]);
    }
    if (x) pull(x); if (y) pull(y);
}

} treap;

int root;

void init(int n) {
    Rep(i, n) {
        int x; scanf("%d", &x);
        treap.Init(i, x);
        root = (i == 1) ? 1 : treap.Merge(root, i);
    }
}

```

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  
8a23  
95cf  
89e3  
95cf  
b1f4  
427e  
24b6  
427e  
d34f  
34d7  
7681  
0ed8  
bcc8

```

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:

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

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

```

2e73 const int MAXN = 1000005;
ca06 struct LCT {
6a6d     int fa[MAXN], ch[MAXN][2], val[MAXN], sum[MAXN];
c6e1     bool rev[MAXN];
427e
eba3     bool isroot(int x) { return ch[fa[x]][0] == x || ch[fa[x]][1] == x; }
f19f     void pull(int x) { sum[x] = val[x] ^ sum[ch[x][0]] ^ sum[ch[x][1]]; }
1c4d     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); }

int Root(int x) {
    for (access(x); ch[x][0]; x = ch[x][0]) push(x);
    splay(x); return x;
}
void Link(int u, int v) { chroot(u); fa[u] = v; }
void Cut(int u, int v) { split(u, v); fa[u] = ch[v][0] = 0; pull(v); }
int Query(int u, int v) { split(u, v); return sum[v]; }
void Update(int u, int x) { splay(u); val[u] = x; }
int LCA(int x, int y, int root) {
    chroot(root); access(x); splay(y);
    while (fa[y]) splay(y = fa[y]);
    return y;
}
};

```

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  
f4f1  
0d77  
95cf  
9e46  
7c10  
0691  
a999  
1f42  
6cb2  
02e5  
c218  
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;
427e
20b0     void* operator new(size_t size);
427e
3dc0     static node* Build(int l, int r) {
b6c5         node* a = new node;
ce96         if (r > l + 1) {
181e             int mid = (l + r) / 2;
3ba2             a->left = Build(l, mid);
8aaf             a->right = Build(mid, r);
8e2e         } else {
bfc4             a->value = 0;
95cf         }
5ffd         return a;
95cf     }
427e
5a45     static node* init(int size) {
2c46         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);
    }
    a->value++;
    return a;
}

node *inc(int index) {
    return Inc(0, n, index);
}
} nodes[8000000];

int node::n, node::pos;
inline void* node::operator new(size_t size) {
    return nodes + (pos++);
}

```

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  
5ffd  
95cf  
427e  
e80f  
c246  
95cf  
865a  
427e  
99ce  
1987  
bb3c  
95cf

## 6.8 Block list

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

### Usage:

<code>block::fix()</code>	Apply tags to the current block.
<code>Init(l, r)</code>	Range initializer.
<code>Reverse(l, r)</code>	Reverse the range.
<code>Add(l, r, x)</code>	Add $x$ to the range.
<code>Query(l, r)</code>	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
8bc4     void merge(block& another) {
b895         fix(); another.fix();
f516         vi temp(move(data));
d02c         temp.insert(temp.end(), range(another.data));
88ea         *this = block(move(temp));
95cf     }
427e
42e8     block split(int pos) {
3e79         fix();
ccab         block result(vi(data.begin() + pos, data.end()));
861a         data.resize(pos); *this = block(move(data));
56b0         return result;
95cf     }
329b };
427e
```

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

struct blocklist {
    list<block> blk;

    void maintain() {
        lit it = blk.begin();
        while (it != blk.end() && next(it) != blk.end()) {
            lit it2 = it;
            while (next(it2) != blk.end() &&
                it2->data.size() + next(it2)->data.size() <= BLOCK) {
                it2->merge(*next(it2));
                blk.erase(next(it2));
            }
            ++it;
        }
    }

    lit split(int pos) {
        for (lit it = blk.begin(); ; it++) {
            if (pos == 0) return it;
            while (it->data.size() > pos)
                blk.insert(next(it), it->split(pos));
            pos -= it->data.size();
        }
    }

    void Init(int *l, int *r) {
        for (int *cur = l; cur < r; cur += BLOCK)
            blk.emplace_back(vi(cur, min(cur + BLOCK, r)));
    }

    void Reverse(int l, int r) {
        lit it = split(l), it2 = split(r);
        reverse(it, it2);
        while (it != it2) {
            it->rev ^= 1;
            it++;
        }
        maintain();
    }

    void Add(int l, int r, int x) {
```

```
2a18
427e
ce14
5540
427e
7b8e
3131
4628
852d
188c
3600
93e1
e1fa
95cf
5771
95cf
95cf
427e
b7b3
2273
5502
8e85
2099
a5a1
427e
95cf
95cf
427e
1c7b
9919
8950
95cf
427e
a22f
997b
dfd0
8f89
6a06
5283
95cf
b204
95cf
427e
3cce
```

```

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.

```

a19e constexpr int BLOCK = 800;
76b3 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;

struct blocklist {
    list<block> blk;

    void maintain() {
        lit it = blk.begin();
        while (it != blk.end() and next(it) != blk.end()) {
            lit it2 = it;
            while (next(it2) != blk.end() and
                it2->data->size() + next(it2)->data->size() <= BLOCK) {
                it2->merge(*next(it2));
                blk.erase(next(it2));
            }
            ++it;
        }
    }
};

```

0563  
013b  
427e  
a771  
2989  
8fd0  
427e  
427e  
a613  
24b5  
0cf0  
e93b  
427e  
5c0f  
0b18  
ac21  
6467  
95cf  
427e  
42e8  
dac1  
01db  
56b0  
95cf  
329b  
427e  
2a18  
427e  
ce14  
5540  
427e  
7b8e  
3131  
5e44  
852d  
0b03  
029f  
93e1  
e1fa  
95cf  
5771  
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 minimum query

The array is 0-based and the range is left-closed right-open.

```

db63     const int MAXN = 100007;
cefd     int a[MAXN], st[MAXN][30];
427e
d34f     void init(int n){
c73d         int l = log2(n);
cf75         rep (i, n) st[i][0] = a[i];
426b         rep (j, l) rep (i, 1+n-(1<<j))
1131             st[i][j+1] = min(st[i][j], st[i+(1<<j)][j]);
95cf     }
427e
c863     int rmq(int l, int r){
f089         int k = log2(r - l);
6117         return min(st[l][k], st[r-(1<<k)][k]);
95cf     }

```

## 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;
    return (v1, v2) <= 0 && v1 * v2 == 0;
}

// 0 not on segment
// 1 on segment except vertices
// 2 on vertices
int ptOnSeg2(pt& p, seg& s){
    vec v1 = s.first - p, v2 = s.second - p;
    T ip = (v1, v2);
    if (v1 * v2 != 0 || ip > 0) return 0;
    return (v1, v2) ? 1 : 2;
}

// if two orthogonal rectangles do not touch, return true
inline bool nIntRectRect(seg a, seg b){
    return min(a.first.x, a.second.x) > max(b.first.x, b.second.x) ||
           min(a.first.y, a.second.y) > max(b.first.y, b.second.y) ||
           min(b.first.x, b.second.x) > max(a.first.x, a.second.x) ||
           min(b.first.y, b.second.y) > max(a.first.y, a.second.y);
}

```

302f  
421c  
427e  
4553  
c0ae  
7a9d  
ffaa  
3ec7  
221a  
8b34  
427e  
368b  
90f4  
ba8c  
427e  
0ea6  
427e  
8d6e  
ce77  
de97  
95cf  
427e  
427e  
427e  
427e  
8421  
ce77  
70ca  
8b14  
0847  
95cf  
427e  
427e  
72bb  
f9ac  
f486  
39ce  
80c7  
95cf

```

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;
9400     if (j1 != 0 && j2 != 0) return 1;
83db     if (j1 == 0 && j2 == 0){
6b0c         if (va * vb == 0) return 4; else return 3;
fb17     } else return 2;
95cf }
427e
2c68 template <typename Tp = T>
5894 inline pt getIntersection(pt P, vec v, pt Q, vec w){
6850     static_assert(is_same<Tp, double>::value, "must_be_double!");
7c9a     return P + v * (w*(P-Q)/(v*w));
95cf }
427e
427e // -1 outside the polygon
427e // 0 on the border of the polygon
427e // 1 inside the polygon
cbdd int ptOnPoly(pt p, pt* poly, int n){
5fb4     int wn = 0;
1294     for (int i = 0; i < n; i++) {
427e
3cae         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;
}
}
```

b957  
8c40  
3c4d  
aad3  
95cf  
0a5f  
95cf  
427e  
d4a3  
fa86  
331a  
95cf  
427e  
07ae  
5cab  
331a  
95cf

## 8 Appendices

### 8.1 Number theory

#### 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 Maximum number of divisors of $n$ -digit number

$d$	max. #	first such number
1	4	6
2	12	60
3	32	840
4	64	7560
5	128	83160
6	240	720720
7	448	8648640
8	768	73513440
9	1344	735134400
10	2304	6983776800
11	4032	97772875200
12	6720	963761198400
13	10752	9316358251200
14	17280	97821761637600
15	26880	866421317361600
16	41472	8086598962041600
17	64512	74801040398884800
18	103680	897612484786617600

### 8.4 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 Polya 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  $g$ .

### 8.5 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 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 '\n'.join(map(str, map(sum, zip(*map(
    lambda a, b : [x*a for x in b], y, lj)))))
```

```
6dc9
4b2b
427e
796b
83e4
f697
156c
dfce
5849
427e
f06d
e80a
a649
9dfa
3cae
7c0d
```