ইনডেক্সঃ

7. হ্যাশিং
    - ডাবল হ্যাশিং
    - ম্যাপের সাহায্যে স্ট্রিং হ্যাশিং
    - বার্নস্টাইন স্ট্রিং হ্যাশিং

কন্টেস্ট টেমপ্লেট

```
/*
USER: zobayer
TASK:
ALGO:
*/

//#pragma warning (disable: 4786)
//#pragma comment (linker, "/STACK:0x800000")
//#define _CRT_SECURE_NO_WARNINGS 1

#include <cassert>
#include <cctype>
#include <climits>
#include <cmath>
#include <cstdio>
#include <cstdlib>
#include <cstring>
#include <iostream>
#include <sstream>
#include <iomanip>
#include <string>
#include <vector>
#include <list>
#include <set>
#include <map>
#include <stack>
#include <queue>
#include <algorithm>
#include <iterator>
#include <utility>
using namespace std;

template< class T > T _abs(T n) { return (n < 0 ? -n : n); }
template< class T > T _max(T a, T b) { return (!(a < b) ? a : b); }
template< class T > T _min(T a, T b) { return (a < b ? a : b); }
template< class T > T sq(T x) { return x * x; }

#define ALL(p) p.begin(),p.end()
#define MP(x, y) make_pair(x, y)
#define SET(p) memset(p, -1, sizeof(p))
#define CLR(p) memset(p, 0, sizeof(p))
#define MEM(p, v) memset(p, v, sizeof(p))
#define CPY(d, s) memcpy(d, s, sizeof(s))
#define READ(f) freopen(f, "r", stdin)
#define WRITE(f) freopen(f, "w", stdout)
#define SZ(c) (int)c.size()
#define PB(x) push_back(x)
#define ff first
#define ss second
#define i64 long long
#define ld long double
#define pii pair< int, int >
#define psi pair< string, int >

const double EPS = 1e-9;
const int INF = 0x7f7f7f7f;
```

```
int main() {
    //READ("in.txt");
    //WRITE("out.txt");
    return 0;
}
```

ফাস্ট রিডার

```
/*
In gcc/g++ fread_unlocked() is even faster.
Can be made faster in non-object-oriented fashion with macros instead of functions.
In contest, who cares dynamic allocation?
*/

struct FastRead {
    char *buff, *ptr;
    FastRead(int size) {
        buff = new char[size];
        ptr = buff;
        fread(buff, 1, size, stdin);
    }
    ~FastRead() {
        delete[] buff;
    }
    int nextInt() {
        int ret = 0;
        while(*ptr < '0' || *ptr > '9') ptr++;
        do { ret = ret * 10 + *ptr++ - '0';
        } while(*ptr >= '0' && *ptr <= '9');
        return ret;
    }
};
```

টাইম ট্র্যাকার

```
/*
Usage: Just create an instance of it at the start of the block of
which you want to measure running time. Reliable up to millisecond
scale. You don't need to do anything else, even works with freopen.
*/

class TimeTracker {
    clock_t start, end;
public:
    TimeTracker() {
        start = clock();
    }
    ~TimeTracker() {
        end = clock();
        fprintf(stderr, "%.3lf s\n", (double)(end - start) / CLOCKS_PER_SEC);
    }
};
```

পপ কাউন্ট

```
/*
Counts number of set bits in an integer.
For 32 bit integer, delete 8 hex from const values and remove the line for 64 bits.
*/

typedef __int64 INT;

const INT b1  = 0x5555555555555555;
const INT b2  = 0x3333333333333333;
const INT b4  = 0x0f0f0f0f0f0f0f0f;
```

```
const INT b8  = 0x00ff00ff00ff00ff;
const INT b16 = 0x0000ffff0000ffff;
const INT b32 = 0x00000000ffffffff;

int popcount(INT x) {
    x = (x & b1 ) + (( x >>  1 ) & b1 ); // 2 bits
    x = (x & b2 ) + (( x >>  2 ) & b2 ); // 4 bits
    x = (x & b4 ) + (( x >>  4 ) & b4 ); // 8 bits
    x = (x & b8 ) + (( x >>  8 ) & b8 ); //16 bits
    x = (x & b16) + (( x >> 16 ) & b16); //32 bits
    x = (x & b32) + (( x >> 32 ) & b32); //64 bits
    return x;
}
```

হিস্টোগ্রাম প্রবলেম

```
/*
Finds largest rectangular area in a histogram in O(n)
*/

i64 calc(int *ht, int n) {
    i64 ret = 0;
    int top = 1, st[MAX], i;
    ht[0] = st[0] = ht[++n] = 0;
    for(i = 1; i <= n; i++) {
        while(top > 1 && ht[st[top-1]] >= ht[i]) {
            ret = _max(ret, (i64)ht[st[top-1]]*(i64)(i - st[top-2]-1));
            top--;
        }
        st[top++] = i;
    }
    return ret;
}
```

ইনফাইনাইট চেসবোর্ডে নাইট ডিস্ট্যান্স

```
/*
NK is the size of grid you want to precalculate
NK/2,NK/2 will be considered origin
Calculates minimum knight distance from 0,0 to x,y
*/

const int KN = 101;

i64 dk[KN][KN];
int dx[] = {-1, -1, 1, 1, -2, -2, 2, 2};
int dy[] = {-2, 2, -2, 2, -1, 1, -1, 1};

void precalc() {
    int x, y, x1, y1, i;
    queue< int > Q;
    memset(dk, 0x3f, sizeof dk);
    x = y = (KN >> 1);
    dk[x][y] = 0;
    Q.push(x); Q.push(y);
    while(!Q.empty()) {
        x = Q.front(); Q.pop();
        y = Q.front(); Q.pop();
        for(i = 0; i < 8; i++) {
            x1 = x + dx[i], y1 = y + dy[i];
            if(0 <= x1 && x1 < KN && 0 <= y1 && y1 < KN) {
                if(dk[x1][y1] > dk[x][y] + 1) {
                    dk[x1][y1] = dk[x][y] + 1;
                    Q.push(x1); Q.push(y1);
                }
            }
```

```
        }
    }
}

i64 knight(i64 x, i64 y) {
    i64 step, res = 0;
    if(x < y) swap(x, y);
    while((x<<1) > KN) {
        step = x / 2 / 2; res += step;
        x -= step * 2; y -= step;
        if(y < 0) y = ((y % 2) + 2) % 2;
        if(x < y) swap(x, y);
    }
    res += dk[x+(KN>>1)][y+(KN>>1)];
    return res;
}
```

ট্রাই (প্রিফিক্স ট্রি)

```
/*
Basic trie, don't use it with large alphabet set or long length.
All operation has complexity O(length).
MAX is number of different items.
*/

struct trie {
    trie *next[MAX+1];
    trie() { for(int i=0; i<=MAX; i++) next[i] = NULL; }
};

void insert(trie *root, int *seq, int len) {
    trie *curr = root;
    for(int i = 0; i < len; i++) {
        if(!curr->next[seq[i]]) curr->next[seq[i]] = new trie;
        curr = curr->next[seq[i]];
    }
    if(!curr->next[MAX]) curr->next[MAX] = new trie;
}

bool found(trie *root, int *seq, int len) {
    trie *curr = root;
    for(int i = 0; i < len; i++) {
        if(!curr->next[seq[i]]) return false;
        curr = curr->next[seq[i]];
    }
    if(!curr->next[MAX]) return false;
    return true;
}
```

ট্রাই (স্ট্যাটিক অ্যারে)

```
/*
Trie implementation using array, faster and takes less memory.
Each node can contain arbitrary data as needed for solving the problem.
The ALPHABET, MAX and scale() may need tuning as necessary.
*/

const int ALPHABET = 26;
const int MAX = 100000;

// for mapping items form 0 to ALPHABET-1
#define scale(x) (x-'a')


struct TrieTree {
```

```
    int n, root;
    int next[MAX][ALPHABET];
    char data[MAX]; // there can be more data fields

    void init() {
        root = 0, n = 1; data[root] = 0;
        memset(next[root], -1, sizeof(next[root]));
    }

    void insert(char *s) {
        int curr = root, i, k;
        for(i = 0; s[i]; i++) {
            k = scale(s[i]);
            if(next[curr][k] == -1) {
                next[curr][k] = n;
                data[n] = s[i]; // optional
                memset(next[n], -1, sizeof(next[n]));
                n++;
            }
            curr = next[curr][k];
        }
        data[curr] = 0; // sentinel, optional
    }

    bool find(char *s) {
        int curr = root, i, k;
        for(i = 0; s[i]; i++) {
            k = scale(s[i]);
            if(next[curr][k] == -1) return false;
            curr = next[curr][k];
        }
        return (data[curr] == 0);
    }

} trieTree;
```

সাফিক্স অ্যারে (স্লো)

```
/*
Generates suffix array in O(n (lg n)^2). Finds LCP in O(lg n)
A[] is the target string with N length. base is the lowest character in A[].
S[i] holds the index of ith suffix when sorted lexicographically.
Change base as necessary and MAXLOG = log2(MAXLEN).
*/

int N, stp, P[MAXLOG][MAXLEN], S[MAXLEN];
char A[MAXLEN], base = 'a';
struct entry { int nr[2], p; } L[MAXLEN];

inline bool cmp(const entry &a, const entry &b) {
    return a.nr[0] == b.nr[0] ? (a.nr[1] < b.nr[1] ? 1 : 0) : (a.nr[0] < b.nr[0] ?
1 : 0);
}

void generateSuffix() {
    register int i, cnt;
    for(i=0; i<N; i++) P[0][i] = A[i]-base;
    for(stp = cnt = 1; cnt>>1 < N; stp++, cnt<<=1) {
        for(i=0; i<N; i++) {
            L[i].nr[0] = P[stp-1][i];
            L[i].nr[1] = i+cnt<N ? P[stp-1][i+cnt] : -1;
            L[i].p = i;
        }
        sort(L, L+N, cmp);
        for(i=0; i<N; i++) {
            if(i > 0 && L[i].nr[0] == L[i - 1].nr[0] && L[i].nr[1] == L[i - 1].nr[1])
```

```
P[stp][L[i].p] = P[stp][L[i - 1].p];
            else P[stp][L[i].p] = i;
        }
    }
    for(i=0; i<N; i++) S[P[stp-1][i]] = i;
}

int lcp(int x, int y) {
    int k, ret = 0;
    if(x == y) return N - x;
    for(k = stp - 1; k >= 0 && x < N && y < N; k --)
        if(P[k][x] == P[k][y])
            x += 1 << k, y += 1 << k, ret += 1 << k;
    return ret;
}
```

সাফিক্স অ্যারে (n lg n)

```
/*
Suffix array implementation using bucket sorting + lcp.
Complexity O(n log n), str[] is the target string,
n is its length and suffix[i] contains i'th sorted suffix position.
*/

const int MAXN = 1 << 16;
const int MAXL = 16;

int n, stp, mv, suffix[MAXN], tmp[MAXN];
int sum[MAXN], cnt[MAXN], rank[MAXL][MAXN];
char str[MAXN];

inline bool equal(const int &u, const int &v){
    if(!stp) return str[u] == str[v];
    if(rank[stp-1][u] != rank[stp-1][v]) return false;
    int a = u + mv < n ? rank[stp-1][u+mv] : -1;
    int b = v + mv < n ? rank[stp-1][v+mv] : -1;
    return a == b;
}

void update(){
    int i, rnk;
    for(i = 0; i < n; i++) sum[i] = 0;
    for(i = rnk = 0; i < n; i++) {
        suffix[i] = tmp[i];
        if(i && !equal(suffix[i], suffix[i-1])) {
            rank[stp][suffix[i]] = ++rnk;
            sum[rnk+1] = sum[rnk];
        }
        else rank[stp][suffix[i]] = rnk;
        sum[rnk+1]++;
    }
}

void Sort() {
    int i;
    for(i = 0; i < n; i++) cnt[i] = 0;
    memset(tmp, -1, sizeof tmp);
    for(i = 0; i < mv; i++){
        int idx = rank[stp - 1][n - i - 1];
        int x = sum[idx];
        tmp[x + cnt[idx]] = n - i - 1;
        cnt[idx]++;
    }
    for(i = 0; i < n; i++){
        int idx = suffix[i] - mv;
        if(idx < 0)continue;
```

```
        idx = rank[stp-1][idx];
        int x = sum[idx];
        tmp[x + cnt[idx]] = suffix[i] - mv;
        cnt[idx]++;
    }
    update();
    return;
}

inline bool cmp(const int &a, const int &b){
    if(str[a]!=str[b]) return str[a]<str[b];
    return false;
}

void SortSuffix() {
    int i;
    for(i = 0; i < n; i++) tmp[i] = i;
    sort(tmp, tmp + n, cmp);
    stp = 0;
    update();
    ++stp;
    for(mv = 1; mv < n; mv <<= 1) {
        Sort();
        stp++;
    }
    stp--;
    for(i = 0; i <= stp; i++) rank[i][n] = -1;
}

inline int lcp(int u, int v) {
    if(u == v) return n - u;
    int ret, i;
    for(ret = 0, i = stp; i >= 0; i--) {
        if(rank[i][u] == rank[i][v]) {
            ret += 1<<i;
            u += 1<<i;
            v += 1<<i;
        }
    }
    return ret;
}
```

মিনিমাম এক্সপ্রেশন (ঝো-জুয়ান)

```
/*
Finds alphabetically first representation of a cyclic string in O(length)
*/

inline int minimumExpression(char *s) {
    int i, j, k, n, len, p, q;
    len = n = strlen(s), n <<= 1, i = 0, j = 1, k = 0;
    while(i + k < n && j + k < n) {
        p = i+k >= len ? s[i+k-len] : s[i+k];
        q = j+k >= len ? s[j+k-len] : s[j+k];
        if(p == q) k++;
        else if(p > q) { i = i+k+1; if(i <= j) i = j+1; k = 0; }
        else if(p < q) { j = j+k+1; if(j <= i) j = i+1; k = 0; }
    }
    return i < j ? i : j;
}
```

কে.এম.পি

```
/*
finds all occurance of pattern in buffer
buffer can be online
```

```
*/

char buffer[MAX], pattern[MAX];
int overlap[MAX];

void computeOverlap() {
    overlap[0] = -1;
    for(int i=0; pattern[i]; i++) {
        overlap[i+1] = overlap[i] + 1;
        while(overlap[i+1] > 0 && pattern[i] != pattern[overlap[i+1]-1]) {
            overlap[i+1] = overlap[overlap[i+1]-1] + 1;
        }
    }
}

vector< int > kmpMatcher(int m) {
    vector< int > V;
    for(int i = 0, j = 0; buffer[i]; i++) {
        while(true) {
            if(buffer[i] == pattern[j]) {
                j++;
                if(j == m) {
                    V.push_back(i-m+1);
                    j = overlap[j];
                }
                break;
            }
            else if(j == 0) break;
            else j = overlap[j];
        }
    }
    return V;
}
```

ম্যানাক্যার আলগোরিদম

```
/*
Manacher algorithm implementation.
Application, largest palindromic substring, largest palindromic suffix
*/

int lengths[MAX<<1];

int manacher(char *buff, int len) {
    int i, k, pallen, found, d, j, s, e;
    k = pallen = 0;
    for(i = 0; i < len; ) {
        if(i > pallen && buff[i-pallen-1] == buff[i]) {
            pallen += 2, i++;
            continue;
        }
        lengths[k++] = pallen;
        s = k - 2, e = s - pallen, found = 0;
        for(j = s; j > e; j--) {
            d = j - e - 1;
            if(lengths[j] == d) {
                pallen = d;
                found = 1;
                break;
            }
            lengths[k++] = (d < lengths[j]? d : lengths[j]);
        }
        if(!found) { pallen = 1; i++; }
    }
    lengths[k++] = pallen;
    return lengths[k-1];
}
```

জোসেফাস রিকারেন্স

```
/*
The first one is for K = 2 and the second one is general.
Note: first function returns 1 based index while second one is 0 based.
*/

int f(int n) {
    if(n == 1) return 1;
    return (f((n-(n&1))>>1)<<1) + ((n&1)?1:-1);
}

int f(int n, int k) {
    if(n == 1) return 0;
    return (f(n-1, k) + k)%n;
}
```

শ্যাঙ্ক'স বেবি স্টেপ জায়ান্ট স্টেপ

```
/*
Shanks baby step giant step - discrete logarithm algorithm
for the equation: b = a^x % p where a, b, p known, finds x
works only when p is an odd prime
*/

int shank(int a, int b, int p) {
    int i, j, m;
    long long c, aj, ami;
    map< long long, int > M;
    map< long long, int > :: iterator it;
    m = (int)ceil(sqrt((double)(p)));
    M.insert(make_pair(1, 0));
    for(j = 1, aj = 1; j < m; j++) {
        aj = (aj * a) % p;
        M.insert(make_pair(aj, j));
    }
    ami = modexp(modinv(a, p), m, p);
    for(c = b, i = 0; i < m; i++) {
        it = M.find(c);
        if(it != M.end()) return i * m + it->second;
        c = (c * ami) % p;
    }
    return 0;
}
```

সেগমেন্টেড সিভ

```
/*
Generates primes within interval [a, b] when b - a <= 100000
and 1 <= a <= b <= 2147483647
*/
int base[MAX>>6], segment[RNG>>6], primes[LEN], prlen;

#define chkC(x,n) (x[n>>6]&(1<<((n>>1)&31)))
#define setC(x,n) (x[n>>6]|=(1<<((n>>1)&31)))

void sieve() {
    int i, j, k;
    for(i=3; i<LMT; i+=2) if(!chkC(base, i)) for(j=i*i, k=i<<1; j<MAX; j+=k)
setC(base, j);
    for(i=3, prlen=0; i<MAX; i+=2) if(!chkC(base, i)) primes[prlen++] = i;
}

int segmented_sieve(int a, int b) {
    int rt, i, k, cnt = (a<=2 && 2<=b)? 1 : 0;
```

```
    if(b<2) return 0;
    if(a<3) a = 3;
    if(a%2==0) a++;
    memset(segment, 0, sizeof segment);
    for(i=0, rt=(int)sqrt((double)b); i < prlen && primes[i] <= rt; i++) {
        unsigned j = primes[i] * ( (a+primes[i]-1) / primes[i] );
        if(j%2==0) j += primes[i];
        for(k=primes[i]<<1; j<=b; j+=k) if(j!=primes[i]) setC(segment, (j-a));
    }
    for(i=0; i<=b-a; i+=2) if(!chkC(segment, i)) cnt++;
    return cnt;
}
```

এক্সটেন্ডেড ইউক্লিড ও মডুলার ইনভার্স

```
/*
Implementation of extended euclid algorithm.
Modular inverse requires gcd(a, n) = 1.
*/

class Euclid {
public:
    i64 x, y, d;
    Euclid() {}
    Euclid(i64 _x, i64 _y, i64 _d) : x(_x), y(_y), d(_d) {}
};

Euclid egcd(i64 a, i64 b) {
    if(!b) return Euclid(1, 0, a);
    Euclid r = egcd(b, a % b);
    return Euclid(r.y, r.x - a / b * r.y, r.d);
}

i64 modinv(i64 a, i64 n) {
    Euclid t = egcd(a, n);
    if(t.d > 1) return 0;
    i64 r = t.x % n;
    return (r < 0 ? r + n : r);
}
```

কনভেক্স হাল (গ্রাহাম'স স্ক্যান)

```
/*
ConvexHull : Graham's Scan O(n lg n), integer implementation
P[]: holds all the points, C[]: holds points on the hull
np: number of points in P[], nc: number of points in C[]
to handle duplicate, call makeUnique() before calling convexHull()
call convexHull() if you have np >= 3
to remove co-linear points on hull, call compress() after convexHull()
*/

point P[MAX], C[MAX], P0;

inline int triArea2(const point &a, const point &b, const point &c) {
    return (a.x*(b.y-c.y) + b.x*(c.y-a.y) + c.x*(a.y-b.y));
}

inline int sqDist(const point &a, const point &b) {
    return ((a.x-b.x)*(a.x-b.x) + (a.y-b.y)*(a.y-b.y));
}

inline bool comp(const point &a, const point &b) {
    int d = triArea2(P0, a, b);
    if(d < 0) return false;
    if(!d && sqDist(P0, a) > sqDist(P0, b)) return false;
    return true;
```

```
}

inline bool normal(const point &a, const point &b) {
    return ((a.x==b.x) ? a.y < b.y : a.x < b.x);
}

inline bool issame(const point &a, const point &b) {
    return (a.x == b.x && a.y == b.y);
}

inline void makeUnique(int &np) {
    sort(&P[0], &P[np], normal);
    np = unique(&P[0], &P[np], issame) - P;
}

void convexHull(int &np, int &nc) {
    int i, j, pos = 0;
    for(i = 1; i < np; i++)
        if(P[i].y<P[pos].y || (P[i].y==P[pos].y && P[i].x<P[pos].x))
            pos = i;
    swap(P[0], P[pos]);
    P0 = P[0];
    sort(&P[1], &P[np], comp);
    for(i = 0; i < 3; i++) C[i] = P[i];
    for(i = j = 3; i < np; i++) {
        while(triArea2(C[j-2], C[j-1], P[i]) < 0) j--;
        C[j++] = P[i];
    }
    nc = j;
}

void compress(int &nc) {
    int i, j, d;
    C[nc] = C[0];
    for(i=j=1; i < nc; i++) {
        d = triArea2(C[j-1], C[i], C[i+1]);
        if(d || (!d && issame(C[j-1], C[i+1]))) C[j++] = C[i];
    }
    nc = j;
}
```

পয়েন্ট ইন কনভেক্স পলিগন

```
/*
C[] array of points of convex polygon in ccw order,
nc number of points in C, p target points.
returns true if p is inside C (including edge) or false otherwise.
complexity O(lg n)
*/

inline bool inConvexPoly(point *C, int nc, const point &p) {
    int st = 1, en = nc - 1, mid;
    while(en - st > 1) {
        mid = (st + en)>>1;
        if(triArea2(C[0], C[mid], p) < 0) en = mid;
        else st = mid;
    }
    if(triArea2(C[0], C[st], p) < 0) return false;
    if(triArea2(C[st], C[en], p) < 0) return false;
    if(triArea2(C[en], C[0], p) < 0) return false;
    return true;
}
```

পলিগন এরিয়া (2D)

```
/*
```

```
P[] holds the points, must be either in cw or ccw
function returns double of the area.
*/

inline int dArea(int np) {
    int area = 0;
    for(int i = 0; i < np; i++) {
        area += p[i].x*p[i+1].y - p[i].y*p[i+1].x;
    }
    return abs(area);
}
```

ক্লোজেস্ট পেয়ার

```
/*
closestPair(Point *X, Point *Y, int n);
X contains the points sorted by x co-ordinate,
Y contains the points sorted by y co-ordinate,
One additional item in Point structure is needed, the original index.
*/

typedef long long i64;
typedef struct { int x, y, i; } Point;

int flag[MAX];

inline i64 sq(const i64 &x) {
    return x*x;
}

inline i64 sqdist(const Point &a, const Point &b) {
    return sq(a.x-b.x) + sq(a.y-b.y);
}

inline i64 closestPair(Point *X, Point *Y, int n) {
    if(n == 1) return INF;
    if(n == 2) return sqdist(X[0], X[1]);

    int i, j, k, n1, n2, ns, m = n >> 1;
    Point Xm = X[m-1], *XL, *XR, *YL, *YR, *YS;
    i64 lt, rt, dd, tmp;

    XL = new Point[m], YL = new Point[m];
    XR = new Point[m+1], YR = new Point[m+1];
    YS = new Point[n];

    for(i = 0; i < m; i++) XL[i] = X[i], flag[X[i].i] = 0;
    for(; i < n; i++) XR[i - m] = X[i], flag[X[i].i] = 1;
    for(i = n2 = n1 = 0; i < n; i++) {
        if(!flag[Y[i].i]) YL[n1++] = Y[i];
        else YR[n2++] = Y[i];
    }

    lt = closestPair(XL, YL, n1);
    rt = closestPair(XR, YR, n2);
    dd = min(lt, rt);

    for(i = ns = 0; i < n; i++)
        if(sq(Y[i].x - Xm.x) < dd)
            YS[ns++] = Y[i];
    for(j = 0; j < ns; j++)
        for(k = j + 1; k < ns && sq(YS[k].y - YS[j].y) < dd; k++)
            dd = min(dd, sqdist(YS[j], YS[k]));

    delete[] XL; delete[] XR;
```

```
    delete[] YL; delete[] YR;
    delete[] YS;

    return dd;
}
```

সার্কেল ইন্টারসেকশন এরিয়া

```
/*
This code assumes the circle center and radius to be integer.
Change this when necessary.
*/

inline double commonArea(const Circle &a, const Circle &b) {
    int dsq = sqDist(a.c, b.c);
    double d = sqrt((double)dsq);
    if(sq(a.r + b.r) <= dsq) return 0;
    if(a.r >= b.r && sq(a.r-b.r) >= dsq) return pi * b.r * b.r;
    if(a.r <= b.r && sq(b.r-a.r) >= dsq) return pi * a.r * a.r;
    double angleA = 2.0 * acos((a.r * a.r + dsq - b.r * b.r) / (2.0 * a.r * d));
    double angleB = 2.0 * acos((b.r * b.r + dsq - a.r * a.r) / (2.0 * b.r * d));
    return 0.5 * (a.r * a.r * (angleA - sin(angleA)) + b.r * b.r * (angleB -
sin(angleB)));
}
```

সেগমেন্ট ইন্টারসেকশন (2D)

```
/*
Segment intersection in 2D integer space.
P1, p2 makes first segment, p3, p4 makes the second segment
*/

inline bool intersect(const Point &p1, const Point &p2, const Point &p3, const Point
&p4) {
    i64 d1, d2, d3, d4;
    d1 = direction(p3, p4, p1);
    d2 = direction(p3, p4, p2);
    d3 = direction(p1, p2, p3);
    d4 = direction(p1, p2, p4);
    if(((d1 < 0 && d2 > 0) || (d1 > 0 && d2 < 0)) && ((d3 < 0 && d4 > 0) || (d3 > 0
&& d4 < 0))) return true;
    if(!d3 && onsegment(p1, p2, p3)) return true;
    if(!d4 && onsegment(p1, p2, p4)) return true;
    if(!d1 && onsegment(p3, p4, p1)) return true;
    if(!d2 && onsegment(p3, p4, p2)) return true;
    return false;
}
```

টেটরাহেড্রন ফরমুলা

```
/*
Some tetrahedron formulas
*/

inline double volume(double u, double v, double w, double U, double V, double W) {
    double u1,v1,w1;
    u1 = v * v + w * w - U * U;
    v1 = w * w + u * u - V * V;
    w1 = u * u + v * v - W * W;
    return sqrt(4.0*u*u*v*v*w*w - u*u*u1*u1 - v*v*v1*v1 - w*w*w1*w1 + u1*v1*w1) /
12.0;
}

inline double surface(double a, double b, double c) {
    return sqrt((a + b + c) * (-a + b + c) * (a - b + c) * (a + b - c)) / 4.0;
```

```
}

inline double insphere(double WX, double WY, double WZ, double XY, double XZ, double YZ)
{
    double sur, rad;
    sur = surface(WX, WY, XY) + surface(WX, XZ, WZ) + surface(WY, YZ, WZ) +
surface(XY, XZ, YZ);
    rad = volume(WX, WY, WZ, YZ, XZ, XY) * 3.0 / sur;
    return rad;
}
```

স্ট্রংলি কানেকটেড কম্পোনেন্ট (টারজান)

```
/*
SCC (Tarjan) in O(|v| + |e|)
Input:
G[] is a input directed graph with n nodes in range [1,n]
Output:
Component[i] holds the component id to which node i belongs
components: total number of components in the graph
*/

int Stack[MAX], top;
int Index[MAX], Lowlink[MAX], Onstack[MAX];
int Component[MAX];
int idx, components;
vector< int > G[MAX];

void tarjan(int u) {
    int v, i;
    Index[u] = Lowlink[u] = idx++;
    Stack[top++] = u;
    Onstack[u] = 1;
    for(i = 0; i < SZ(G[u]); i++) {
        v = G[u][i];
        if(Index[v]==-1) {
            tarjan(v);
            Lowlink[u] = min(Lowlink[u], Lowlink[v]);
        }
        else if(Onstack[v]) Lowlink[u] = min(Lowlink[u], Index[v]);
    }
    if(Lowlink[u] == Index[u]) {
        components++;
        do {
            v = Stack[--top];
            Onstack[v] = 0;
            Component[v] = components;
        } while(u != v);
    }
}

void findSCC(int n) {
    components = top = idx = 0;
    SET(Index); CLR(Onstack); MEM(Lowlink, 0x3f);
    for(int i = 1; i <= n; i++) if(Index[i]==-1) tarjan(i);
}
```

আর্টিকুলেশন পয়েন্ট

```
/*
G[][]: undirected graph
cut[v] is true if node v is an articulation point / cut-vertex
*/

vector< int > G[MAX];
int low[MAX], vis[MAX], used[MAX], cut[MAX], dfstime;
```

```
void dfs(int u, int par = -1) {
    int i, v, child = 0;
    used[u] = 1;
    vis[u] = low[u] = ++dfstime;
    for(i = 0; i < G[u].size(); i++) {
        v = G[u][i];
        if(v == par) continue;
        if(used[v]) low[u] = min(low[u], vis[v]);
        else {
            child++;
            dfs(v, u);
            low[u] = min(low[u], low[v]);
            if(low[v] >= vis[u]) cut[u] = 1;
        }
    }
    if(par == -1) cut[u] = (child > 1);
}
```

ব্রিজ

```
/*
G[][]: undirected graph
finds all the bridges in a connected graph and
adds those edges to the Bridges[] vector
*/

vector< int > G[MAX];
vector< pair< int, int > > Bridges;
int low[MAX], vis[MAX], used[MAX], dfstime;

void dfs(int u, int par) {
    int i, v;
    used[u] = 1;
    vis[u] = low[u] = ++dfstime;
    for(i = 0; i < G[u].size(); i++) {
        v = G[u][i];
        if(v == par) continue;
        if(used[v]) low[u] = min(low[u], vis[v]);
        else {
            dfs(v, u);
            low[u] = min(low[u], low[v]);
            if(low[v] > vis[u]) Bridges.push_back(make_pair(u, v));
        }
    }
}
```

বাই কানেকটেড কম্পোনেন্ট

```
/*
G[][]: undirected graph
Separates bi-connected component by edges.
*/

vector< int > G[MAX];
stack< pii > S;
int dfstime;
int low[MAX], vis[MAX], used[MAX];

void dfs(int u, int par) {
    int v, i, sz = G[u].size();
    pii e, curr;
    used[u] = 1;
    vis[u] = low[u] = ++dfstime;
    for(i = 0; i < sz; i++) {
        v = G[u][i];
```

```
        if(v == par) continue;
        if(!used[v]) {
            S.push(pii(u, v));
            dfs(v, u);
            if(low[v] >= vis[u]) {
                // new component
                curr = pii(u, v);
                do {
                    e = S.top(); S.pop();
                    // e is an edge in current bcc
                } while(e != curr);
            }
            low[u] = min(low[u], low[v]);
        }
        else if(vis[v] < vis[u]) {
            S.push(pii(u, v));
            low[u] = min(low[u], vis[v]);
        }
    }
}
```

স্টেবল ম্যারিজ

```
/*
INPUT:
m: number of man, n: number of woman (must be at least as large as m)
L[i][]: the list of women in order of decreasing preference of man i
R[j][i]: the attractiveness of i to j.
OUTPUTS:
L2R[]: the mate of man i (always between 0 and n-1)
R2L[]: the mate of woman j (or -1 if single)
man priority
*/

int m, n, L[MAXM][MAXW], R[MAXW][MAXM], L2R[MAXM], R2L[MAXW], p[MAXM];

void stableMarriage() {
    int i, man, wom, hubby;
    SET(R2L); CLR(p);
    for(i = 0; i < m; i++ ) {
        man = i;
        while(man >= 0) {
            while(true) {
                wom = L[man][p[man]++];
                if(R2L[wom] < 0 || R[wom][man] > R[wom][R2L[wom]]) break;
            }
            hubby = R2L[wom];
            R2L[L2R[man] = wom] = man;
            man = hubby;
        }
    }
}
```

ম্যাক্স ফ্লো (ডিনিক)

```
/*
max flow (dinitz algorithm)
works on undirected graph
can have loops, multiple edges, cycles
*/

int src, snk, nNode, nEdge;
int Q[MAXN], fin[MAXN], pro[MAXN], dist[MAXN];
int flow[MAXE], cap[MAXE], next[MAXE], to[MAXE];

inline void init(int _src, int _snk, int _n) {
```

```
    src = _src, snk = _snk, nNode = _n, nEdge = 0;
    SET(fin);
}

inline void add(int u, int v, int _cap) {
    to[nEdge] = v, cap[nEdge] = _cap, flow[nEdge] = 0;
    next[nEdge] = fin[u], fin[u] = nEdge++;
    to[nEdge] = u, cap[nEdge] = _cap, flow[nEdge] = 0;
    next[nEdge] = fin[v], fin[v] = nEdge++;
}

bool bfs() {
    int st, en, i, u, v;
    SET(dist);
    dist[src] = st = en = 0;
    Q[en++] = src;
    while(st < en) {
        u = Q[st++];
        for(i=fin[u]; i>=0; i=next[i]) {
            v = to[i];
            if(flow[i] < cap[i] && dist[v]==-1) {
                dist[v] = dist[u]+1;
                Q[en++] = v;
            }
        }
    }
    return dist[snk]!=-1;
}

int dfs(int u, int fl) {
    if(u==snk) return fl;
    for(int &e=pro[u], v, df; e>=0; e=next[e]) {
        v = to[e];
        if(flow[e] < cap[e] && dist[v]==dist[u]+1) {
            df = dfs(v, min(cap[e]-flow[e], fl));
            if(df>0) {
                flow[e] += df;
                flow[e^1] -= df;
                return df;
            }
        }
    }
    return 0;
}

i64 dinitz() {
    i64 ret = 0;
    int df;
    while(bfs()) {
        for(int i=1; i<=nNode; i++) pro[i] = fin[i];
        while(true) {
            df = dfs(src, INF);
            if(df) ret += (i64)df;
            else break;
        }
    }
    return ret;
}
```

মিন কস্ট ম্যাক্স ফ্লো (বেলম্যান ফোর্ড)

```
/*
min cost flow (bellman ford)
works only on directed graphs
handles multiple edges, cycles, loops
*/
```

```
int src, snk, nNode, nEdge;
int fin[MAXN], pre[MAXN], dist[MAXN];
int cap[MAXE], cost[MAXE], next[MAXE], to[MAXE], from[MAXE];

inline void init(int _src, int _snk, int nodes) {
    SET(fin);
    nNode = nodes, nEdge = 0;
    src = _src, snk = _snk;
}

inline void addEdge(int u, int v, int _cap, int _cost) {
    from[nEdge] = u, to[nEdge] = v, cap[nEdge] = _cap, cost[nEdge] = _cost;
    next[nEdge] = fin[u], fin[u] = nEdge++;
    from[nEdge] = v, to[nEdge] = u, cap[nEdge] = 0, cost[nEdge] = -(_cost);
    next[nEdge] = fin[v], fin[v] = nEdge++;
}

bool bellman() {
    int iter, u, v, i;
    bool flag = true;
    MEM(dist, 0x7f);
    SET(pre);
    dist[src] = 0;
    for(iter = 1; iter < nNode && flag; iter++) {
        flag = false;
        for(u = 0; u < nNode; u++) {
            for(i = fin[u]; i >= 0; i = next[i]) {
                v = to[i];
                if(cap[i] && dist[v] > dist[u] + cost[i]) {
                    dist[v] = dist[u] + cost[i];
                    pre[v] = i;
                    flag = true;
                }
            }
        }
    }
    return (dist[snk] < INF);
}

int mcmf(int &fcost) {
    int netflow, i, bot, u;
    netflow = fcost = 0;
    while(bellman()) {
        bot = INF;
        for(u = pre[snk]; u >= 0; u = pre[from[u]]) bot = min(bot, cap[u]);
        for(u = pre[snk]; u >= 0; u = pre[from[u]]) {
            cap[u] -= bot;
            cap[u^1] += bot;
            fcost += bot * cost[u];
        }
        netflow += bot;
    }
    return netflow;
}
```

ম্যাক্সিমাম বাইপারটাইট ম্যাচিং (ডি.এফ.এস.)

```
/*
G[] is the left-side graph, must be bipartite
match(n): n is the number of nodes in left-side set
and returns the maximum possible matching.
Left[] anf Right[] ar assigned with corresponding matches
*/

vector < int > G[MAX];
```

```
bool visited[MAX];
int Left[MAX], Right[MAX];

bool dfs(int u) {
    if(visited[u]) return false;
    visited[u] = true;
    int len = G[u].size(), i, v;
    for(i=0; i<len; i++) {
        v = G[u][i];
        if(Right[v]==-1) {
            Right[v] = u, Left[u] = v;
            return true;
        }
    }
    for(i=0; i<len; i++) {
        v = G[u][i];
        if(dfs(Right[v])) {
            Right[v] = u, Left[u] = v;
            return true;
        }
    }
    return false;
}

int match(int n) {
    int i, ret = 0;
    bool done;
    SET(Left); SET(Right);
    do {
        done = true; CLR(visited);
        for(i=0; i<n; i++) {
            if(Left[i]==-1 && dfs(i)) {
                done = false;
            }
        }
    } while(!done);
    for(i=0; i<n; i++) ret += (Left[i]!=-1);
    return ret;
}
```

ম্যাক্সিমাম বাইপারটাইট ম্যাচিং (হপক্রফট কার্প)

```
/*
n: number of nodes on left side, nodes are numbered 1 to n
m: number of nodes on right side, nodes are numbered n+1 to n+m
G = NIL[0] ? G1[G[1---n]] ? G2[G[n+1---n+m]]
*/

bool bfs() {
    int i, u, v, len;
    queue< int > Q;
    for(i=1; i<=n; i++) {
        if(match[i]==NIL) {
            dist[i] = 0;
            Q.push(i);
        }
        else dist[i] = INF;
    }
    dist[NIL] = INF;
    while(!Q.empty()) {
        u = Q.front(); Q.pop();
        if(u!=NIL) {
            len = G[u].size();
            for(i=0; i<len; i++) {
                v = G[u][i];
                if(dist[match[v]]==INF) {
                    dist[match[v]] = dist[u] + 1;
```

```
                    Q.push(match[v]);
                }
            }
        }
    }
    return (dist[NIL]!=INF);
}

bool dfs(int u) {
    int i, v, len;
    if(u!=NIL) {
        len = G[u].size();
        for(i=0; i<len; i++) {
            v = G[u][i];
            if(dist[match[v]]==dist[u]+1) {
                if(dfs(match[v])) {
                    match[v] = u;
                    match[u] = v;
                    return true;
                }
            }
        }
        dist[u] = INF;
        return false;
    }
    return true;
}

int hopcroft_karp() {
    int matching = 0, i;
    CLR(match);
    while(bfs())
        for(i=1; i<=n; i++)
            if(match[i]==NIL && dfs(i))
                matching++;
    return matching;
}
```

হেভী লাইট ডিকম্পজিশন

```
/*
Rough code for heavy light decomposition. Input graph must be a tree.
Also includes the LCA method.
What to do with the chain is problem dependent.
*/

vector< int > G[MAX];
int cost[MAX], lvl[MAX], parent[MAX];
int head[MAX], cnext[MAX], chainid[MAX], chainpos[MAX];
int nchain, temp[MAX];

int dfs(int u, int p, int d) {
    int i, v, sz = G[u].size(), tmp, mx, id, tot, hd, k;
    lvl[u] = d, mx = 0, id = u, tot = 1;
    for(i = 0; i < sz; i++) {
        v = G[u][i];
        if(v != p) {
            parent[v] = u;
            tmp = dfs(v, u, d + 1);
            tot += tmp;
            if(tmp > mx) {
                mx = tmp;
                id = v;
            }
        }
    }
    if(tot == 1) cnext[u] = -1;
```

```
    else cnext[u] = id;
    for(i = 0; i < sz; i++) {
        v = G[u][i];
        if(v != p && v != id) {
            for(hd = v, k = 0; v != -1; v = cnext[v], k++) {
                head[v] = hd;
                temp[k] = cost[v];
                chainpos[v] = k;
                chainid[v] = nchain;
            }
            // buff is the current chain of size k
            nchain++;
        }
    }
    return tot;
}

void hld(int v) {
    int hd, k;
    nchain = 0; lvl[0] = -1;
    dfs(v, 0, 0);
    for(hd = v, k = 0; v != -1; v = cnext[v], k++) {
        head[v] = hd;
        temp[k] = cost[v];
        chainpos[v] = k;
        chainid[v] = nchain;
    }
    // buff is the current chain of size k
    nchain++;
}

int lca(int a, int b) {
    while(chainid[a] != chainid[b]) {
        if(lvl[head[a]] < lvl[head[b]]) b = parent[head[b]];
        else a = parent[head[a]];
    }
    return (lvl[a] < lvl[b]) ? a : b;
}
```

টারজান'স অফলাইন এল.সি.এ

```
/*
Tarjan's offline LCA algorithm. For each pair of node's in P {u, v, qid},
it finds the LCA of the nodes in the rooted tree G (no edge to back to the parent.
The array ans holds the result for queries in orders defined by qid.
*/

void lca(int u) {
    int v, i, sz;
    make_set(u);
    ancestor[find_set(u)] = u;
    sz = G[u].size();
    for(i = 0; i < sz; i++) {
        v = G[u][i];
        lca(v);
        union_set(u, v);
        ancestor[find_set(u)] = u;
    }
    color[u] = 1;
    sz = P[u].size();
    for(i = 0; i < sz; i++) {
        v = P[u][i].first;
        if(color[v]) ans[P[u][i].second] = ancestor[find_set(v)];
    }
}
```

বাইনারি সার্চ

```
/*
st and en defines the range [st, en) in a[]
in every range, start is inclusive and end is exclusive
*/

// returns index of first element not less than val
int lowerbound(int *a, int st, int en, int val) {
    int idx, cnt, stp;
    cnt = en - st;
    while(cnt > 0) {
        stp = cnt >> 1; idx = st + stp;
        if(a[idx] < val) st = ++idx, cnt -= stp+1;
        else cnt = stp;
    }
    return st;
}

// returns index of first element greater than val
int upperbound(int *a, int st, int en, int val) {
    int idx, cnt, stp;
    cnt = en - st;
    while(cnt > 0) {
        stp = cnt >> 1; idx = st + stp;
        if(a[idx] <= val) st = ++idx, cnt -= stp+1;
        else cnt = stp;
    }
    return st;
}

// returns true if items is found, otherwise false
bool binarysearch(int *a, int st, int en, int val) {
    int lb = lowerbound(a, st, en, val);
    return (lb != en && a[lb] == val);
}

// returns the interval where each element is equal to val
pair< int, int > equalrange(int *a, int st, int en, int val) {
    int ub = upperbound(a, st, en, val);
    int lb = lowerbound(a, st, en, val);
    return pair< int, int >(lb, ub);
}
```

ডিসজয়েন্ট সেট

```
/*
disjoint set data-structure
implements union by rank and path compression
*/

struct DisjointSet {
    int *root, *rank, n;
    DisjointSet(int sz) {
        root = new int[sz+1];
        rank = new int[sz+1];
        n = sz;
    }
    ~DisjointSet() {
        delete[] root;
        delete[] rank;
    }
    void init() {
        for(int i = 1; i <= n; i++) {
            root[i] = i;
            rank[i] = 0;
```

```
        }
    }
    int find(int u) {
        if(u != root[u]) root[u] = find(root[u]);
        return root[u];
    }
    void merge(int u, int v) {
        int pu = find(u);
        int pv = find(v);
        if(rank[pu] > rank[pv]) root[pv] = pu;
        else root[pu] = pv;
        if(rank[pu]==rank[pv]) rank[pv]++;
    }
};
```

2D বি.আই.টি

```
/*
update and query function for 2D bit.
MAX is the maximum possible value.
bit[][] holds the 2D binary indexed tree
*/

void update(int x, int y, int v) {
    int y1;
    while(x <= MAX) {
        y1 = y;
        while(y1 <= MAX) {
            bit[x][y1] += v;
            y1 += (y1 & -y1);
        }
        x += (x & -x);
    }
}

int readsum(int x, int y) {
    int v = 0, y1;
    while(x > 0) {
        y1 = y;
        while(y1 > 0) {
            v += bit[x][y1];
            y1 -= (y1 & -y1);
        }
        x -= (x & -x);
    }
    return v;
}
```

ডাবল হ্যাশিং

```
/*
M > N and should be close, better both be primes.
M should be as much large as possible, not exceeding array size.
HKEY is the Hash function, change it if necessary.
*/

#define NIL -1
#define M 1021
#define N 1019
#define HKEY(x,i) ((x)%M+(i)*(1+(x)%N))%M

int a[M+1];

inline int hash(int key) {
    int i = 0, j;
    do {
```

```
        j = HKEY(key, i);
        if(a[j]==NIL) { a[j] = key; return j; }
        i++;
    } while(i < M);
    return -1;
}

inline int find(int key) {
    int i = 0, j;
    do {
        j = HKEY(key, i);
        if(a[j]==key) return j;
        i++;
    } while(a[j]!=NIL && i < M);
    return -1;
}
```

ম্যাপের সাহায্যে স্ট্রিং হ্যাশিং

```
/*
hash() takes the string and next free index as parameter.
if string is found in map, it returns its index.
else, it inserts in current free index and updates the free index.
*/

inline int hash(char *s, int &n) {
    int ret; it = M.find(s);
    if(it == M.end()) M.insert(pair< string, int > (s, ret = n++));
    else ret = it->second;
    return ret;
}
```

বার্নস্টাইন স্ট্রিং হ্যাশিং

```
/*
u64 stands for unsigned long long type. Must use unsigned type.
hash = hash * 33 ^ c, reliable hash
*/
u64 hash(const char* s) {
    u64 hash = 0, c;
    while((c = *s++)) hash = ((hash << 5) + hash) ^ c;
    return hash;
}
```