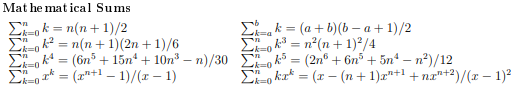
**Combinatorics**

****

****

**Catalan Numbers: is the number of: properly nested sequences of 𝑛 pairs of parentheses; rooted ordered binary trees with 𝑛 + 1 leaves; triangulations of a convex (𝑛 + 2)-gon**

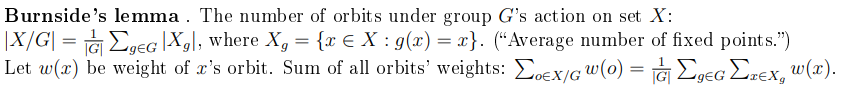
****

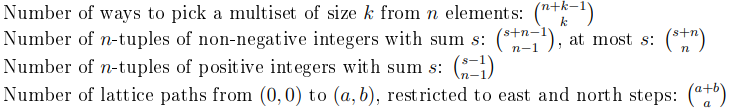
**Derangements:**

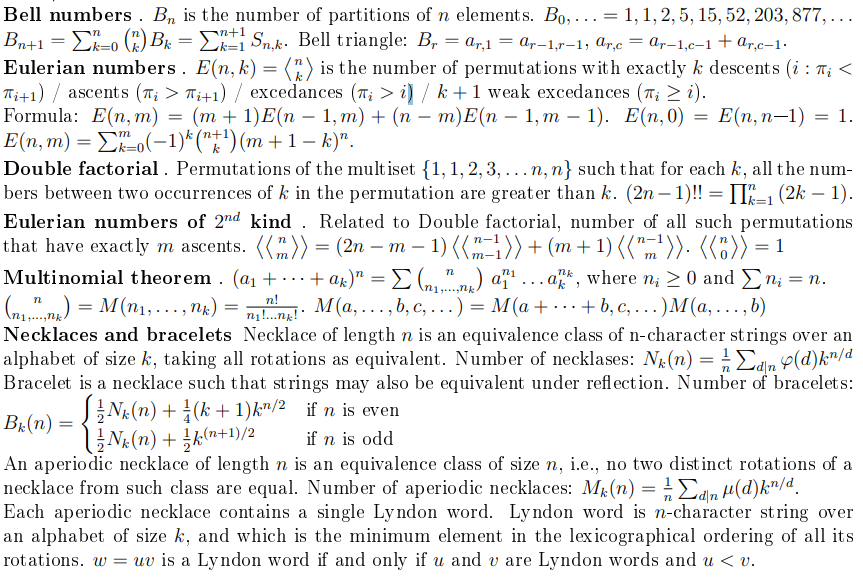
****

**Stirling numbers of 1 𝑠𝑡 kind . 𝑠𝑛,𝑘 is (−1)𝑛−𝑘 times the number of permutations of 𝑛 elements with exactly 𝑘 permutation cycles. [︀ 𝑛 𝑘 ]︀ = |𝑠𝑛,𝑘| = |𝑠𝑛−1,𝑘−1| + (𝑛 − 1)|𝑠𝑛−1,𝑘| 𝑠(0, 0) = 1 and 𝑠(𝑛, 0) = 𝑠(0, 𝑛) = 0.**

**Stirling numbers of 2 𝑛𝑑 kind . 𝑆𝑛,𝑘 = {︀𝑛 𝑘}︀ is the number of ways to partition a set of 𝑛 elements into exactly 𝑘 non-empty subsets. {︀𝑛 𝑘}︀ = 𝑆𝑛,𝑘 = 𝑆𝑛−1,𝑘−1 + 𝑘𝑆𝑛−1,𝑘; 𝑆𝑛,1 = 𝑆𝑛,𝑛 = 1. {︀𝑛 𝑘}︀ = 1/𝑘! ∑︀𝑘 𝑗=0(−1)𝑗 (︀𝑘 𝑗)︀ (𝑘 − 𝑗) 𝑛 .**

****

****

****

gcd(fm,fn)=fgcd(m,n)*.*

*If*fm|fn*then*m|n*.*

**Next Combination of K elements (universe of size N)**

**bool next\_combination(vector<int>& a, int n) {**

**int k = (int)a.size();**

**for (int i = k - 1; i >= 0; i--) {**

**if (a[i] < n - k + i + 1) {**

**a[i]++;**

**for (int j = i + 1; j < k; j++)**

**a[j] = a[j - 1] + 1;**

**return true;**

**}**

**}**

**return false;**

**}**

**Data Structures**

**2D BIT:**

**struct FenwickTree2D {**

**vector<vector<int>> bit;**

**int n, m;**

**// init(...) { ... }**

**int sum(int x, int y) {**

**int ret = 0;**

**for (int i = x; i >= 0; i = (i & (i + 1)) - 1)**

**for (int j = y; j >= 0; j = (j & (j + 1)) - 1)**

**ret += bit[i][j];**

**return ret;**

**}**

**void add(int x, int y, int delta) {**

**for (int i = x; i < n; i = i | (i + 1))**

**for (int j = y; j < m; j = j | (j + 1))**

**bit[i][j] += delta;**

**}**

**};**

**Range BIT:**

**def add(b, idx, x):**

**while idx <= N:**

**b[idx] += x**

**idx += idx & -idx**

**def range\_add(l,r,x):**

**add(B1, l, x)**

**add(B1, r+1, -x)**

**add(B2, l, x\*(l-1))**

**add(B2, r+1, -x\*r)**

**def sum(b, idx):**

**total = 0**

**while idx > 0:**

**total += b[idx]**

**idx -= idx & -idx**

**return total**

**def prefix\_sum(idx):**

**return sum(B1, idx)\*idx - sum(B2, idx)**

**def range\_sum(l, r):**

**return sum(r) - sum(l-1)**

**Indexed MultiSet:**

**#include <bits/stdc++.h>**

**#include <ext/pb\_ds/assoc\_container.hpp>**

**using namespace std;**

**using namespace \_\_gnu\_pbds;**

**typedef long long ll;**

**typedef tree<pair<ll, int>, null\_type, less\_equal<pair<ll, int>>, rb\_tree\_tag, tree\_order\_statistics\_node\_update> indexed\_set;**

**// .clear(), .insert(), .erase(), .find\_by\_order(), .order\_of\_key()**

**2D Segment Tree**

**void build\_y(int vx, int lx, int rx, int vy, int ly, int ry) {**

**if (ly == ry) {**

**if (lx == rx)**

**t[vx][vy] = a[lx][ly];**

**else**

**t[vx][vy] = t[vx\*2][vy] + t[vx\*2+1][vy];**

**} else {**

**int my = (ly + ry) / 2;**

**build\_y(vx, lx, rx, vy\*2, ly, my);**

**build\_y(vx, lx, rx, vy\*2+1, my+1, ry);**

**t[vx][vy] = t[vx][vy\*2] + t[vx][vy\*2+1];**

**}**

**}**

**void build\_x(int vx, int lx, int rx) {**

**if (lx != rx) {**

**int mx = (lx + rx) / 2;**

**build\_x(vx\*2, lx, mx);**

**build\_x(vx\*2+1, mx+1, rx);**

**}**

**build\_y(vx, lx, rx, 1, 0, m-1);**

**}**

**int sum\_y(int vx, int vy, int tly, int try\_, int ly, int ry) {**

**if (ly > ry)**

**return 0;**

**if (ly == tly && try\_ == ry)**

**return t[vx][vy];**

**int tmy = (tly + try\_) / 2;**

**return sum\_y(vx, vy\*2, tly, tmy, ly, min(ry, tmy))**

**+ sum\_y(vx, vy\*2+1, tmy+1, try\_, max(ly, tmy+1), ry);**

**}**

**int sum\_x(int vx, int tlx, int trx, int lx, int rx, int ly, int ry) {**

**if (lx > rx)**

**return 0;**

**if (lx == tlx && trx == rx)**

**return sum\_y(vx, 1, 0, m-1, ly, ry);**

**int tmx = (tlx + trx) / 2;**

**return sum\_x(vx\*2, tlx, tmx, lx, min(rx, tmx), ly, ry)**

**+ sum\_x(vx\*2+1, tmx+1, trx, max(lx, tmx+1), rx, ly, ry);**

**}**

**void update\_y(int vx, int lx, int rx, int vy, int ly, int ry, int x, int y, int new\_val) {**

**if (ly == ry) {**

**if (lx == rx)**

**t[vx][vy] = new\_val;**

**else**

**t[vx][vy] = t[vx\*2][vy] + t[vx\*2+1][vy];**

**} else {**

**int my = (ly + ry) / 2;**

**if (y <= my)**

**update\_y(vx, lx, rx, vy\*2, ly, my, x, y, new\_val);**

**else**

**update\_y(vx, lx, rx, vy\*2+1, my+1, ry, x, y, new\_val);**

**t[vx][vy] = t[vx][vy\*2] + t[vx][vy\*2+1];**

**}**

**}**

**void update\_x(int vx, int lx, int rx, int x, int y, int new\_val) {**

**if (lx != rx) {**

**int mx = (lx + rx) / 2;**

**if (x <= mx)**

**update\_x(vx\*2, lx, mx, x, y, new\_val);**

**else**

**update\_x(vx\*2+1, mx+1, rx, x, y, new\_val);**

**}**

**update\_y(vx, lx, rx, 1, 0, m-1, x, y, new\_val);**

**}**

**Dynamic Programming**

**Divide and Conquer Optimization:**

**Recurrence: Optimal splitting point must be monotonic.**

****

**int n;**

**long long C(int i, int j);**

**vector<long long> dp\_before(n), dp\_cur(n);**

**// compute dp\_cur[l], ... dp\_cur[r] (inclusive)**

**void compute(int l, int r, int optl, int optr)**

**{**

**if (l > r)**

**return;**

**int mid = (l + r) >> 1;**

**pair<long long, int> best = {INF, -1};**

**for (int k = optl; k <= min(mid, optr); k++) {**

**best = min(best, {dp\_before[k] + C(k, mid), k});**

**}**

**dp\_cur[mid] = best.first;**

**int opt = best.second;**

**compute(l, mid - 1, optl, opt);**

**compute(mid + 1, r, opt, optr);**

**}**

**Convex Hull Optimization**

**Recurrence:**

****

**typedef int ftype;**

**typedef complex<ftype> point;**

**#define x real**

**#define y imag**

**ftype dot(point a, point b) {**

**return (conj(a) \* b).x();**

**}**

**ftype cross(point a, point b) {**

**return (conj(a) \* b).y();**

**}**

**vector<point> hull, vecs;**

**void add\_line(ftype k, ftype b) {**

**point nw = {k, b};**

**while(!vecs.empty() && dot(vecs.back(), nw - hull.back()) < 0) {**

**hull.pop\_back();**

**vecs.pop\_back();**

**}**

**if(!hull.empty()) {**

**vecs.push\_back(1i \* (nw - hull.back()));**

**}**

**hull.push\_back(nw);**

**}**

**int get(ftype x) {**

**point query = {x, 1};**

**auto it = lower\_bound(vecs.begin(), vecs.end(), query, [](point a, point b) {**

**return cross(a, b) > 0;**

**});**

**return dot(query, hull[it - vecs.begin()]);**

**}**

**Li Chao Tree**

**typedef int ftype;**

**typedef complex<ftype> point;**

**#define x real**

**#define y imag**

**ftype dot(point a, point b) {**

**return (conj(a) \* b).x();**

**}**

**ftype f(point a, ftype x) {**

**return dot(a, {x, 1});**

**}**

**const int maxn = 2e5;**

**point line[4 \* maxn];**

**void add\_line(point nw, int v = 1, int l = 0, int r = maxn) {**

**int m = (l + r) / 2;**

**bool lef = f(nw, l) < f(line[v], l);**

**bool mid = f(nw, m) < f(line[v], m);**

**if(mid) {**

**swap(line[v], nw);**

**}**

**if(r - l == 1) {**

**return;**

**} else if(lef != mid) {**

**add\_line(nw, 2 \* v, l, m);**

**} else {**

**add\_line(nw, 2 \* v + 1, m, r);**

**}**

**}**

**int get(int x, int v = 1, int l = 0, int r = maxn) {**

**int m = (l + r) / 2;**

**if(r - l == 1) {**

**return f(line[v], x);**

**} else if(x < m) {**

**return min(f(line[v], x), get(x, 2 \* v, l, m));**

**} else {**

**return min(f(line[v], x), get(x, 2 \* v + 1, m, r));**

**}**

**}**

**Knuth Optimization:**

**Recurrence:**

***It is applicable in the case where recurrence is in the form :  
dp[i][j] = mini < k < j{dp[i][k] + dp[k][j]} + C[i][j]  
  
The sufficient condition for applicability is:  
A[i, j - 1] ≤ A[i, j] ≤ A[i + 1, j]  
  
Where,***

***A[i][j] — the smallest k that gives optimal answer, for example in:***

***dp[i][j] = dp[i - 1][k] + C[k][j]***

***C[i][j] — given cost function***

**for** (int s = 0; s<=k; s++) //s - length(size) of substring

**for** (int l = 0; l+s<=k; l++) { //l - left point

int r = l + s; //r - right point

**if** (s < 2) {

res[l][r] = 0; //DP base - nothing to break

mid[l][r] = l; //mid is equal to left border

**continue**;

}

int mleft = mid[l][r-1]; //Knuth's trick: getting bounds on m

int mright = mid[l+1][r];

res[l][r] = 1000000000000000000LL;

**for** (int m = mleft; m<=mright; m++) { //iterating for m in the bounds only

int64 tres = res[l][m] + res[m][r] + (x[r]-x[l]);

**if** (res[l][r] > tres) { //relax current solution

res[l][r] = tres;

mid[l][r] = m;

}

}

}

int64 answer = res[0][k];

**LIS:**

#include <bits/stdc++.h>

#include <ext/pb\_ds/assoc\_container.hpp>

#include <ext/pb\_ds/tree\_policy.hpp>

using namespace std;

using namespace \_\_gnu\_pbds;

typedef long long ll;

typedef pair < int, int > pii;

typedef tree < ll, null\_type, less\_equal < ll > , rb\_tree\_tag, //less\_equal

tree\_order\_statistics\_node\_update > oset;

const int MAXN = 2e5 + 5;

const int MOD = 1e9 + 7;

const int MAXK = 2e3 + 2;#

define sz(x)(int)(x).size()# define f first

ll p[MAXN];

int main() {

ios::sync\_with\_stdio(0);

cin.tie(0);

cout.tie(0);

int n;

cin >> n;

vector < int > v, lis;

v.resize(n + 1);

for (int i = 1; i <= n; i++)

cin >> v[i];

for (int i = 1; i <= n; i++) {

auto e = lower\_bound(lis.begin(), lis.end(), v[i]);

if (e == lis.end()) {

lis.push\_back(v[i]);

p[sz(lis)] = v[i];

} else {

\* e = v[i];

e = lower\_bound(lis.begin(), lis.end(), v[i]);

p[e - lis.begin() + 1] = v[i];

}

}

// in case you want print LIS

// remember me i'm in CSES :D

ll ans = sz(lis);

cout << ans << '**\n**';

return 0;

vector < int > plis;

while (ans) {

plis.push\_back(p[ans]);

--ans;

}

reverse(plis.begin(), plis.end());

for (auto & k: plis)

cout << k << ' ';

}

**int lis(vector<int> const& a) {**

**int n = a.size();**

**const int INF = 1e9;**

**vector<int> d(n+1, INF);**

**d[0] = -INF;**

**for (int i = 0; i < n; i++) {**

**int j = upper\_bound(d.begin(), d.end(), a[i]) - d.begin();**

**if (d[j-1] < a[i] && a[i] < d[j])**

**d[j] = a[i];**

**}**

**int ans = 0;**

**for (int i = 0; i <= n; i++) {**

**if (d[i] < INF)**

**ans = i;**

**}**

**return ans;**

**}**

**Number Theory**

**Extended Euclidean Algorithm:**

**int gcd(int a, int b, int & x, int & y) {**

**if (a == 0) {**

**x = 0;**

**y = 1;**

**return b;**

**}**

**int x1, y1;**

**int d = gcd(b % a, a, x1, y1);**

**x = y1 - (b / a) \* x1;**

**y = x1;**

**return d;**

**}**

**Miller Rabin Primality Check:**

**using u64 = uint64\_t;**

**using u128 = \_\_uint128\_t;**

**u64 binpower(u64 base, u64 e, u64 mod) {**

**u64 result = 1;**

**base %= mod;**

**while (e) {**

**if (e & 1)**

**result = (u128)result \* base % mod;**

**base = (u128)base \* base % mod;**

**e >>= 1;**

**}**

**return result;**

**}**

**bool check\_composite(u64 n, u64 a, u64 d, int s) {**

**u64 x = binpower(a, d, n);**

**if (x == 1 || x == n - 1)**

**return false;**

**for (int r = 1; r < s; r++) {**

**x = (u128)x \* x % n;**

**if (x == n - 1)**

**return false;**

**}**

**return true;**

**}**

**bool MillerRabin(u64 n) { // returns true if n is prime, else returns false.**

**if (n < 2)**

**return false;**

**int r = 0;**

**u64 d = n - 1;**

**while ((d & 1) == 0) {**

**d >>= 1;**

**r++;**

**}**

**for (int a : {2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37}) {**

**if (n == a)**

**return true;**

**if (check\_composite(n, a, d, r))**

**return false;**

**}**

**return true;**

**}**

**Pollard Rho: Try multiple cycle starts x0 in case of failure.**

**long long mult(long long a, long long b, long long mod) {**

**return (\_\_int128)a \* b % mod;**

**}**

**long long f(long long x, long long c, long long mod) {**

**return (mult(x, x, mod) + c) % mod;**

**}**

**long long rho(long long n, long long x0=2, long long c=1) {**

**long long x = x0;**

**long long y = x0;**

**long long g = 1;**

**while (g == 1) {**

**x = f(x, c, n);**

**y = f(y, c, n);**

**y = f(y, c, n);**

**g = gcd(abs(x - y), n);**

**}**

**return g;**

**}**

**Euler Phi: a and m are coprime.**

** **

**Linear Congruences**

****

**Let g=gcd(a,n), i.e. the greatest common divisor of a and n (which in this case is greater than one). if b is not divisible by g, there is no solution.**

**Divide both sides of the equation by g (i.e. dividing a, b and n by g), we receive a new equation:**

****

**x′ will also be a solution of the original equation. It will not be the only solution. It can be shown that the original equation has exactly g solutions, and they will look like this:**

****

**Discrete Log: a and m are coprime**

****

**int solve(int a, int b, int m) {**

**int n = (int) sqrt (m + .0) + 1;**

**int an = 1;**

**for (int i = 0; i < n; ++i)**

**an = (an \* a) % m;**

**map<int, int> vals;**

**for (int p = 1, cur = an; p <= n; ++p) {**

**if (!vals.count(cur))**

**vals[cur] = p;**

**cur = (cur \* an) % m;**

**}**

**for (int q = 0, cur = b; q <= n; ++q) {**

**if (vals.count(cur)) {**

**int ans = vals[cur] \* n - q;**

**return ans;**

**}**

**cur = (cur \* a) % m;**

**}**

**return -1;**

**}**

**Primitive Root: for all a and n are coprime**

****

**int powmod (int a, int b, int p) {**

**int res = 1;**

**while (b)**

**if (b & 1)**

**res = int (res \* 1ll \* a % p), --b;**

**else**

**a = int (a \* 1ll \* a % p), b >>= 1;**

**return res;**

**}**

**//Add calculation of phi to generalize**

**int generator (int p) {**

**vector<int> fact;**

**int phi = p-1, n = phi;**

**for (int i=2; i\*i<=n; ++i)**

**if (n % i == 0) {**

**fact.push\_back (i);**

**while (n % i == 0)**

**n /= i;**

**}**

**if (n > 1)**

**fact.push\_back (n);**

**for (int res=2; res<=p; ++res) {**

**bool ok = true;**

**for (size\_t i=0; i<fact.size() && ok; ++i)**

**ok &= powmod (res, phi / fact[i], p) != 1;**

**if (ok) return res;**

**}**

**return -1;**

**}**

**FFT**

**using cd = complex<double>;**

**const double PI = acos(-1);**

**void fft(vector<cd> & a, bool invert) {**

**int n = a.size();**

**if (n == 1)**

**return;**

**vector<cd> a0(n / 2), a1(n / 2);**

**for (int i = 0; 2 \* i < n; i++) {**

**a0[i] = a[2\*i];**

**a1[i] = a[2\*i+1];**

**}**

**fft(a0, invert);**

**fft(a1, invert);**

**double ang = 2 \* PI / n \* (invert ? -1 : 1);**

**cd w(1), wn(cos(ang), sin(ang));**

**for (int i = 0; 2 \* i < n; i++) {**

**a[i] = a0[i] + w \* a1[i];**

**a[i + n/2] = a0[i] - w \* a1[i];**

**if (invert) {**

**a[i] /= 2;**

**a[i + n/2] /= 2;**

**}**

**w \*= wn;**

**}**

**}**

**vector<int> multiply(vector<int> const& a, vector<int> const& b) {**

**vector<cd> fa(a.begin(), a.end()), fb(b.begin(), b.end());**

**int n = 1;**

**while (n < a.size() + b.size())**

**n <<= 1;**

**fa.resize(n);**

**fb.resize(n);**

**fft(fa, false);**

**fft(fb, false);**

**for (int i = 0; i < n; i++)**

**fa[i] \*= fb[i];**

**fft(fa, true);**

**vector<int> result(n);**

**for (int i = 0; i < n; i++)**

**result[i] = round(fa[i].real());**

**return result;**

**}**

**Pythagorean Triples: All relatively prime triples are given by: 𝑥 = 2𝑚𝑛, 𝑦 = 𝑚² − 𝑛², 𝑧 = 𝑚² + 𝑛² where 𝑚 > 𝑛, gcd(𝑚, 𝑛) = 1 and 𝑚 ̸≡ 𝑛 (mod 2). All other triples are multiples of these.**

**String Processing:**

**Suffix Array for Cyclic Shifts. Add ‘$’ for suffixes.**

**vector<int> sort\_cyclic\_shifts(string const& s) {**

**int n = s.size();**

**const int alphabet = 256;**

**vector<int> p(n), c(n), cnt(max(alphabet, n), 0);**

**for (int i = 0; i < n; i++)**

**cnt[s[i]]++;**

**for (int i = 1; i < alphabet; i++)**

**cnt[i] += cnt[i-1];**

**for (int i = 0; i < n; i++)**

**p[--cnt[s[i]]] = i;**

**c[p[0]] = 0;**

**int classes = 1;**

**for (int i = 1; i < n; i++) {**

**if (s[p[i]] != s[p[i-1]])**

**classes++;**

**c[p[i]] = classes - 1;**

**}**

**vector<int> pn(n), cn(n);**

**for (int h = 0; (1 << h) < n; ++h) {**

**for (int i = 0; i < n; i++) {**

**pn[i] = p[i] - (1 << h);**

**if (pn[i] < 0)**

**pn[i] += n;**

**}**

**fill(cnt.begin(), cnt.begin() + classes, 0);**

**for (int i = 0; i < n; i++)**

**cnt[c[pn[i]]]++;**

**for (int i = 1; i < classes; i++)**

**cnt[i] += cnt[i-1];**

**for (int i = n-1; i >= 0; i--)**

**p[--cnt[c[pn[i]]]] = pn[i];**

**cn[p[0]] = 0;**

**classes = 1;**

**for (int i = 1; i < n; i++) {**

**pair<int, int> cur = {c[p[i]], c[(p[i] + (1 << h)) % n]};**

**pair<int, int> prev = {c[p[i-1]], c[(p[i-1] + (1 << h)) % n]};**

**if (cur != prev)**

**++classes;**

**cn[p[i]] = classes - 1;**

**}**

**c.swap(cn);**

**}**

**return p;**

**}**

**Compare two substrings: Store Suffix Array Intermediate equivalence classes**

**int compare(int i, int j, int l, int k) {**

**pair<int, int> a = {c[k][i], c[k][(i+l-(1 << k))%n]};**

**pair<int, int> b = {c[k][j], c[k][(j+l-(1 << k))%n]};**

**return a == b ? 0 : a < b ? -1 : 1;**

**}**

**LCP:**

**vector<int> lcp\_construction(string const& s, vector<int> const& p) {**

**int n = s.size();**

**vector<int> rank(n, 0);**

**for (int i = 0; i < n; i++)**

**rank[p[i]] = i;**

**int k = 0;**

**vector<int> lcp(n-1, 0);**

**for (int i = 0; i < n; i++) {**

**if (rank[i] == n - 1) {**

**k = 0;**

**continue;**

**}**

**int j = p[rank[i] + 1];**

**while (i + k < n && j + k < n && s[i+k] == s[j+k])**

**k++;**

**lcp[rank[i]] = k;**

**if (k)**

**k--;**

**}**

**return lcp;**

**}**

**Prefix Function:**

**vector<int> prefix\_function(string s) {**

**int n = (int)s.length();**

**vector<int> pi(n);**

**for (int i = 1; i < n; i++) {**

**int j = pi[i-1];**

**while (j > 0 && s[i] != s[j])**

**j = pi[j-1];**

**if (s[i] == s[j])**

**j++;**

**pi[i] = j;**

**}**

**return pi;**

**}**

**// Counting**

**vector<int> ans(n + 1);**

**for (int i = 0; i < n; i++)**

**ans[pi[i]]++;**

**for (int i = n-1; i > 0; i--)**

**ans[pi[i-1]] += ans[i];**

**for (int i = 0; i <= n; i++)**

**ans[i]++;**

**// Compression: k = n – pi[n-1]. n%k == 0;**

**Rabin Karp:**

**vector<int> rabin\_karp(string const& s, string const& t) {**

**const int p = 31;**

**const int m = 1e9 + 9;**

**int S = s.size(), T = t.size();**

**vector<long long> p\_pow(max(S, T));**

**p\_pow[0] = 1;**

**for (int i = 1; i < (int)p\_pow.size(); i++)**

**p\_pow[i] = (p\_pow[i-1] \* p) % m;**

**vector<long long> h(T + 1, 0);**

**for (int i = 0; i < T; i++)**

**h[i+1] = (h[i] + (t[i] - 'a' + 1) \* p\_pow[i]) % m;**

**long long h\_s = 0;**

**for (int i = 0; i < S; i++)**

**h\_s = (h\_s + (s[i] - 'a' + 1) \* p\_pow[i]) % m;**

**vector<int> occurences;**

**for (int i = 0; i + S - 1 < T; i++) {**

**long long cur\_h = (h[i+S] + m - h[i]) % m;**

**if (cur\_h == h\_s \* p\_pow[i] % m)**

**occurences.push\_back(i);**

**}**

**return occurences;**

**}**

**Min Cycle:**

**string min\_cyclic\_string(string s) {**

**s += s;**

**int n = s.size();**

**int i = 0, ans = 0;**

**while (i < n / 2) {**

**ans = i;**

**int j = i + 1, k = i;**

**while (j < n && s[k] <= s[j]) {**

**if (s[k] < s[j])**

**k = i;**

**else**

**k++;**

**j++;**

**}**

**while (i <= k)**

**i += j - k;**

**}**

**return s.substr(ans, n / 2);**

**}**

**Palindromic tree:**

**// C++ program to demonstrate working of**

**// palindromic tree**

**#include "bits/stdc++.h"**

**using namespace std;**

**#define MAXN  1000**

**struct Node**

**{**

**// store start and end indexes of current**

**// Node inclusively**

**int start, end;**

**// stores length of substring**

**int length;**

**// stores insertion Node for all characters a-z**

**int insertEdg[26];**

**// stores the Maximum Palindromic Suffix Node for**

**// the current Node**

**int suffixEdg;**

**};**

**// two special dummy Nodes as explained above**

**Node root1, root2;**

**// stores Node information for constant time access**

**Node tree[MAXN];**

**// Keeps track the current Node while insertion**

**int currNode;**

**string s;**

**int ptr;**

**void insert(int idx)**

**{**

**//STEP 1//**

**/\* search for Node X such that s[idx] X S[idx]**

**is maximum palindrome ending at position idx**

**iterate down the suffix link of currNode to**

**find X \*/**

**int tmp = currNode;**

**while (true)**

**{**

**int curLength = tree[tmp].length;**

**if (idx - curLength >= 1 and s[idx] == s[idx-curLength-1])**

**break;**

**tmp = tree[tmp].suffixEdg;**

**}**

**/\* Now we have found X ....**

**\* X = string at Node tmp**

**\* Check : if s[idx] X s[idx] already exists or not\*/**

**if(tree[tmp].insertEdg[s[idx]-'a'] != 0)**

**{**

**// s[idx] X s[idx] already exists in the tree**

**currNode = tree[tmp].insertEdg[s[idx]-'a'];**

**return;**

**}**

**// creating new Node**

**ptr++;**

**// making new Node as child of X with**

**// weight as s[idx]**

**tree[tmp].insertEdg[s[idx]-'a'] = ptr;**

**// calculating length of new Node**

**tree[ptr].length = tree[tmp].length + 2;**

**// updating end point for new Node**

**tree[ptr].end = idx;**

**// updating the start for new Node**

**tree[ptr].start = idx - tree[ptr].length + 1;**

**//STEP 2//**

**/\* Setting the suffix edge for the newly created**

**Node tree[ptr]. Finding some String Y such that**

**s[idx] + Y + s[idx] is longest possible**

**palindromic suffix for newly created Node.\*/**

**tmp = tree[tmp].suffixEdg;**

**// making new Node as current Node**

**currNode = ptr;**

**if (tree[currNode].length == 1)**

**{**

**// if new palindrome's length is 1**

**// making its suffix link to be null string**

**tree[currNode].suffixEdg = 2;**

**return;**

**}**

**while (true)**

**{**

**int curLength = tree[tmp].length;**

**if (idx-curLength >= 1 and s[idx] == s[idx-curLength-1])**

**break;**

**tmp = tree[tmp].suffixEdg;**

**}**

**// Now we have found string Y**

**// linking current Nodes suffix link with s[idx]+Y+s[idx]**

**tree[currNode].suffixEdg = tree[tmp].insertEdg[s[idx]-'a'];**

**}**

**// driver program**

**int main()**

**{**

**// initializing the tree**

**root1.length = -1;**

**root1.suffixEdg = 1;**

**root2.length = 0;**

**root2.suffixEdg = 1;**

**tree[1] = root1;**

**tree[2] = root2;**

**ptr = 2;**

**currNode = 1;**

**// given string**

**s = "abcbab";**

**int l = s.length();**

**for (int i=0; i<l; i++)**

**insert(i);**

**// printing all of its distinct palindromic**

**// substring**

**cout << "All distinct palindromic substring for "**

**<< s << " : \n";**

**for (int i=3; i<=ptr; i++)**

**{**

**cout << i-2 << ") ";**

**for (int j=tree[i].start; j<=tree[i].end; j++)**

**cout << s[j];**

**cout << endl;**

**}**

**return 0;**

**}**

**Manacher:**

**vector<int> d1(n);**

**for (int i = 0, l = 0, r = -1; i < n; i++) {**

**int k = (i > r) ? 1 : min(d1[l + r - i], r - i + 1);**

**while (0 <= i - k && i + k < n && s[i - k] == s[i + k]) {**

**k++;**

**}**

**d1[i] = k--;**

**if (i + k > r) {**

**l = i - k;**

**r = i + k;**

**}**

**}**

**vector<int> d2(n);**

**for (int i = 0, l = 0, r = -1; i < n; i++) {**

**int k = (i > r) ? 0 : min(d2[l + r - i + 1], r - i + 1);**

**while (0 <= i - k - 1 && i + k < n && s[i - k - 1] == s[i + k]) {**

**k++;**

**}**

**d2[i] = k--;**

**if (i + k > r) {**

**l = i - k - 1;**

**r = i + k;**

**}**

**}**

**SLE**

**// Last column contains RHS constants**

**int gauss (vector < vector<double> > a, vector<double> & ans) {**

**int n = (int) a.size();**

**int m = (int) a[0].size() - 1;**

**vector<int> where (m, -1);**

**for (int col=0, row=0; col<m && row<n; ++col) {**

**int sel = row;**

**for (int i=row; i<n; ++i)**

**if (abs (a[i][col]) > abs (a[sel][col]))**

**sel = i;**

**if (abs (a[sel][col]) < EPS)**

**continue;**

**for (int i=col; i<=m; ++i)**

**swap (a[sel][i], a[row][i]);**

**where[col] = row;**

**for (int i=0; i<n; ++i)**

**if (i != row) {**

**double c = a[i][col] / a[row][col];**

**for (int j=col; j<=m; ++j)**

**a[i][j] -= a[row][j] \* c;**

**}**

**++row;**

**}**

**ans.assign (m, 0);**

**for (int i=0; i<m; ++i)**

**if (where[i] != -1)**

**ans[i] = a[where[i]][m] / a[where[i]][i];**

**for (int i=0; i<n; ++i) {**

**double sum = 0;**

**for (int j=0; j<m; ++j)**

**sum += ans[j] \* a[i][j];**

**if (abs (sum - a[i][m]) > EPS)**

**return 0;**

**}**

**for (int i=0; i<m; ++i)**

**if (where[i] == -1)**

**return INF;**

**return 1;**

**}**

**Ternary Search:**

**double ternary\_search(double l, double r) {**

**double eps = 1e-9; //set the error limit here**

**while (r - l > eps) {**

**double m1 = l + (r - l) / 3;**

**double m2 = r - (r - l) / 3;**

**double f1 = f(m1); //evaluates the function at m1**

**double f2 = f(m2); //evaluates the function at m2**

**if (f1 < f2)**

**l = m1;**

**else**

**r = m2;**

**}**

**return f(l); //return the maximum of f(x) in [l, r]**

**}**

**Graph Theory**

**Bridges:**

**int n; // number of nodes**

**vector<vector<int>> adj; // adjacency list of graph**

**vector<bool> visited;**

**vector<int> tin, low;**

**int timer;**

**void dfs(int v, int p = -1) {**

**visited[v] = true;**

**tin[v] = low[v] = timer++;**

**for (int to : adj[v]) {**

**if (to == p) continue;**

**if (visited[to]) {**

**low[v] = min(low[v], tin[to]);**

**} else {**

**dfs(to, v);**

**low[v] = min(low[v], low[to]);**

**if (low[to] > tin[v])**

**IS\_BRIDGE(v, to);**

**}**

**}**

**}**

**void find\_bridges() {**

**timer = 0;**

**visited.assign(n, false);**

**tin.assign(n, -1);**

**low.assign(n, -1);**

**for (int i = 0; i < n; ++i) {**

**if (!visited[i])**

**dfs(i);**

**}**

**}**

**Articulations: Might fail on Root**

**int n; // number of nodes**

**vector<vector<int>> adj; // adjacency list of graph**

**vector<bool> visited;**

**vector<int> tin, low;**

**int timer;**

**void dfs(int v, int p = -1) {**

**visited[v] = true;**

**tin[v] = low[v] = timer++;**

**int children=0;**

**for (int to : adj[v]) {**

**if (to == p) continue;**

**if (visited[to]) {**

**low[v] = min(low[v], tin[to]);**

**} else {**

**dfs(to, v);**

**low[v] = min(low[v], low[to]);**

**if (low[to] >= tin[v] && p!=-1)**

**IS\_CUTPOINT(v);**

**++children;**

**}**

**}**

**if(p == -1 && children > 1)**

**IS\_CUTPOINT(v);**

**}**

**void find\_cutpoints() {**

**timer = 0;**

**visited.assign(n, false);**

**tin.assign(n, -1);**

**low.assign(n, -1);**

**for (int i = 0; i < n; ++i) {**

**if (!visited[i])**

**dfs (i);**

**}**

**}**

**Bellman Ford:**

**void solve()**

**{**

**vector<int> d (n, INF);**

**d[v] = 0;**

**vector<int> p (n - 1);**

**int x;**

**for (int i=0; i<n; ++i)**

**{**

**x = -1;**

**for (int j=0; j<m; ++j)**

**if (d[e[j].a] < INF)**

**if (d[e[j].b] > d[e[j].a] + e[j].cost)**

**{**

**d[e[j].b] = max (-INF, d[e[j].a] + e[j].cost);**

**p[e[j].b] = e[j].a;**

**x = e[j].b;**

**}**

**}**

**if (x == -1)**

**cout << "No negative cycle from " << v;**

**else**

**{**

**int y = x;**

**for (int i=0; i<n; ++i)**

**y = p[y];**

**vector<int> path;**

**for (int cur=y; ; cur=p[cur])**

**{**

**path.push\_back (cur);**

**if (cur == y && path.size() > 1)**

**break;**

**}**

**reverse (path.begin(), path.end());**

**cout << "Negative cycle: ";**

**for (size\_t i=0; i<path.size(); ++i)**

**cout << path[i] << ' ';**

**}**

**}**

**SPFA:**

**const int INF = 1000000000;**

**vector<vector<pair<int, int>>> adj;**

**bool spfa(int s, vector<int>& d) {**

**int n = adj.size();**

**d.assign(n, INF);**

**vector<int> cnt(n, 0);**

**vector<bool> inqueue(n, false);**

**queue<int> q;**

**d[s] = 0;**

**q.push(s);**

**inqueue[s] = true;**

**while (!q.empty()) {**

**int v = q.front();**

**q.pop();**

**inqueue[v] = false;**

**for (auto edge : adj[v]) {**

**int to = edge.first;**

**int len = edge.second;**

**if (d[v] + len < d[to]) {**

**d[to] = d[v] + len;**

**if (!inqueue[to]) {**

**q.push(to);**

**inqueue[to] = true;**

**cnt[to]++;**

**if (cnt[to] > n)**

**return false; // negative cycle**

**}**

**}**

**}**

**}**

**}**

**Paths of length K:**

****

**Shortest paths of a fixed length K:**

****

**Number of ways to connect a graph:**

****

**Code of Tree**

**vector<vector<int>> adj;**

**vector<int> parent;**

**void dfs(int v) {**

**for (int u : adj[v]) {**

**if (u != parent[v]) {**

**parent[u] = v;**

**dfs(u);**

**}**

**}**

**}**

**vector<int> pruefer\_code() {**

**int n = adj.size();**

**parent.resize(n);**

**parent[n-1] = -1;**

**dfs(n-1);**

**int ptr = -1;**

**vector<int> degree(n);**

**for (int i = 0; i < n; i++) {**

**degree[i] = adj[i].size();**

**if (degree[i] == 1 && ptr == -1)**

**ptr = i;**

**}**

**vector<int> code(n - 2);**

**int leaf = ptr;**

**for (int i = 0; i < n - 2; i++) {**

**int next = parent[leaf];**

**code[i] = next;**

**if (--degree[next] == 1 && next < ptr) {**

**leaf = next;**

**} else {**

**ptr++;**

**while (degree[ptr] != 1)**

**ptr++;**

**leaf = ptr;**

**}**

**}**

**return code;**

**}**

**Restoring Tree:**

**vector<pair<int, int>> pruefer\_decode(vector<int> const& code) {**

**int n = code.size() + 2;**

**vector<int> degree(n, 1);**

**for (int i : code)**

**degree[i]++;**

**int ptr = 0;**

**while (degree[ptr] != 1)**

**ptr++;**

**int leaf = ptr;**

**vector<pair<int, int>> edges;**

**for (int v : code) {**

**edges.emplace\_back(leaf, v);**

**if (--degree[v] == 1 && v < ptr) {**

**leaf = v;**

**} else {**

**ptr++;**

**while (degree[ptr] != 1)**

**ptr++;**

**leaf = ptr;**

**}**

**}**

**edges.emplace\_back(leaf, n-1);**

**return edges;**

**}**

**Finding Cycles:**

**int n;**

**vector<vector<int>> adj;**

**vector<char> color;**

**vector<int> parent;**

**int cycle\_start, cycle\_end;**

**bool dfs(int v) {**

**color[v] = 1;**

**for (int u : adj[v]) {**

**if (color[u] == 0) {**

**parent[u] = v;**

**if (dfs(u))**

**return true;**

**} else if (color[u] == 1) {**

**cycle\_end = v;**

**cycle\_start = u;**

**return true;**

**}**

**}**

**color[v] = 2;**

**return false;**

**}**

**void find\_cycle() {**

**color.assign(n, 0);**

**parent.assign(n, -1);**

**cycle\_start = -1;**

**for (int v = 0; v < n; v++) {**

**if (dfs(v))**

**break;**

**}**

**if (cycle\_start == -1) {**

**cout << "Acyclic" << endl;**

**} else {**

**vector<int> cycle;**

**cycle.push\_back(cycle\_start);**

**for (int v = cycle\_end; v != cycle\_start; v = parent[v])**

**cycle.push\_back(v);**

**cycle.push\_back(cycle\_start);**

**reverse(cycle.begin(), cycle.end());**

**cout << "Cycle found: ";**

**for (int v : cycle)**

**cout << v << " ";**

**cout << endl;**

**}**

**}**

**LCA:**

**struct LCA {**

**vector<int> height, euler, first, segtree;**

**vector<bool> visited;**

**int n;**

**LCA(vector<vector<int>> &adj, int root = 0) {**

**n = adj.size();**

**height.resize(n);**

**first.resize(n);**

**euler.reserve(n \* 2);**

**visited.assign(n, false);**

**dfs(adj, root);**

**int m = euler.size();**

**segtree.resize(m \* 4);**

**build(1, 0, m - 1);**

**}**

**void dfs(vector<vector<int>> &adj, int node, int h = 0) {**

**visited[node] = true;**

**height[node] = h;**

**first[node] = euler.size();**

**euler.push\_back(node);**

**for (auto to : adj[node]) {**

**if (!visited[to]) {**

**dfs(adj, to, h + 1);**

**euler.push\_back(node);**

**}**

**}**

**}**

**void build(int node, int b, int e) {**

**if (b == e) {**

**segtree[node] = euler[b];**

**} else {**

**int mid = (b + e) / 2;**

**build(node << 1, b, mid);**

**build(node << 1 | 1, mid + 1, e);**

**int l = segtree[node << 1], r = segtree[node << 1 | 1];**

**segtree[node] = (height[l] < height[r]) ? l : r;**

**}**

**}**

**int query(int node, int b, int e, int L, int R) {**

**if (b > R || e < L)**

**return -1;**

**if (b >= L && e <= R)**

**return segtree[node];**

**int mid = (b + e) >> 1;**

**int left = query(node << 1, b, mid, L, R);**

**int right = query(node << 1 | 1, mid + 1, e, L, R);**

**if (left == -1) return right;**

**if (right == -1) return left;**

**return height[left] < height[right] ? left : right;**

**}**

**int lca(int u, int v) {**

**int left = first[u], right = first[v];**

**if (left > right)**

**swap(left, right);**

**return query(1, 0, euler.size() - 1, left, right);**

**}**

**};**

**Flows**

**Edmond Karp:**

**int n;**

**vector<vector<int>> capacity;**

**vector<vector<int>> adj;**

**int bfs(int s, int t, vector<int>& parent) {**

**fill(parent.begin(), parent.end(), -1);**

**parent[s] = -2;**

**queue<pair<int, int>> q;**

**q.push({s, INF});**

**while (!q.empty()) {**

**int cur = q.front().first;**

**int flow = q.front().second;**

**q.pop();**

**for (int next : adj[cur]) {**

**if (parent[next] == -1 && capacity[cur][next]) {**

**parent[next] = cur;**

**int new\_flow = min(flow, capacity[cur][next]);**

**if (next == t)**

**return new\_flow;**

**q.push({next, new\_flow});**

**}**

**}**

**}**

**return 0;**

**}**

**int maxflow(int s, int t) {**

**int flow = 0;**

**vector<int> parent(n);**

**int new\_flow;**

**while (new\_flow = bfs(s, t, parent)) {**

**flow += new\_flow;**

**int cur = t;**

**while (cur != s) {**

**int prev = parent[cur];**

**capacity[prev][cur] -= new\_flow;**

**capacity[cur][prev] += new\_flow;**

**cur = prev;**

**}**

**}**

**return flow;**

**}**

**Dinic:**

**struct FlowEdge {**

**int v, u;**

**long long cap, flow = 0;**

**FlowEdge(int v, int u, long long cap) : v(v), u(u), cap(cap) {}**

**};**

**struct Dinic {**

**const long long flow\_inf = 1e18;**

**vector<FlowEdge> edges;**

**vector<vector<int>> adj;**

**int n, m = 0;**

**int s, t;**

**vector<int> level, ptr;**

**queue<int> q;**

**Dinic(int n, int s, int t) : n(n), s(s), t(t) {**

**adj.resize(n);**

**level.resize(n);**

**ptr.resize(n);**

**}**

**void add\_edge(int v, int u, long long cap) {**

**edges.emplace\_back(v, u, cap);**

**edges.emplace\_back(u, v, 0);**

**adj[v].push\_back(m);**

**adj[u].push\_back(m + 1);**

**m += 2;**

**}**

**bool bfs() {**

**while (!q.empty()) {**

**int v = q.front();**

**q.pop();**

**for (int id : adj[v]) {**

**if (edges[id].cap - edges[id].flow < 1)**

**continue;**

**if (level[edges[id].u] != -1)**

**continue;**

**level[edges[id].u] = level[v] + 1;**

**q.push(edges[id].u);**

**}**

**}**

**return level[t] != -1;**

**}**

**long long dfs(int v, long long pushed) {**

**if (pushed == 0)**

**return 0;**

**if (v == t)**

**return pushed;**

**for (int& cid = ptr[v]; cid < (int)adj[v].size(); cid++) {**

**int id = adj[v][cid];**

**int u = edges[id].u;**

**if (level[v] + 1 != level[u] || edges[id].cap - edges[id].flow < 1)**

**continue;**

**long long tr = dfs(u, min(pushed, edges[id].cap - edges[id].flow));**

**if (tr == 0)**

**continue;**

**edges[id].flow += tr;**

**edges[id ^ 1].flow -= tr;**

**return tr;**

**}**

**return 0;**

**}**

**long long flow() {**

**long long f = 0;**

**while (true) {**

**fill(level.begin(), level.end(), -1);**

**level[s] = 0;**

**q.push(s);**

**if (!bfs())**

**break;**

**fill(ptr.begin(), ptr.end(), 0);**

**while (long long pushed = dfs(s, flow\_inf)) {**

**f += pushed;**

**}**

**}**

**return f;**

**}**

**};**

**Min Cost Max Flow:**

**struct Edge**

**{**

**int from, to, capacity, cost;**

**};**

**vector<vector<int>> adj, cost, capacity;**

**const int INF = 1e9;**

**void shortest\_paths(int n, int v0, vector<int>& d, vector<int>& p) {**

**d.assign(n, INF);**

**d[v0] = 0;**

**vector<int> m(n, 2);**

**deque<int> q;**

**q.push\_back(v0);**

**p.assign(n, -1);**

**while (!q.empty()) {**

**int u = q.front();**

**q.pop\_front();**

**m[u] = 0;**

**for (int v : adj[u]) {**

**if (capacity[u][v] > 0 && d[v] > d[u] + cost[u][v]) {**

**d[v] = d[u] + cost[u][v];**

**p[v] = u;**

**if (m[v] == 2) {**

**m[v] = 1;**

**q.push\_back(v);**

**} else if (m[v] == 0) {**

**m[v] = 1;**

**q.push\_front(v);**

**}**

**}**

**}**

**}**

**}**

**int min\_cost\_flow(int N, vector<Edge> edges, int K, int s, int t) {**

**adj.assign(N, vector<int>());**

**cost.assign(N, vector<int>(N, 0));**

**capacity.assign(N, vector<int>(N, 0));**

**for (Edge e : edges) {**

**adj[e.from].push\_back(e.to);**

**adj[e.to].push\_back(e.from);**

**cost[e.from][e.to] = e.cost;**

**cost[e.to][e.from] = -e.cost;**

**capacity[e.from][e.to] = e.capacity;**

**}**

**int flow = 0;**

**int cost = 0;**

**vector<int> d, p;**

**while (flow < K) {**

**shortest\_paths(N, s, d, p);**

**if (d[t] == INF)**

**break;**

**// find max flow on that path**

**int f = K - flow;**

**int cur = t;**

**while (cur != s) {**

**f = min(f, capacity[p[cur]][cur]);**

**cur = p[cur];**

**}**

**// apply flow**

**flow += f;**

**cost += f \* d[t];**

**cur = t;**

**while (cur != s) {**

**capacity[p[cur]][cur] -= f;**

**capacity[cur][p[cur]] += f;**

**cur = p[cur];**

**}**

**}**

**if (flow < K)**

**return -1;**

**else**

**return cost;**

**}**

**Hopcroft Karp:**

**// C++ implementation of Hopcroft Karp algorithm for**

**// maximum matching**

**#include<bits/stdc++.h>**

**using namespace std;**

**#define NIL 0**

**#define INF INT\_MAX**

**// A class to represent Bipartite graph for Hopcroft**

**// Karp implementation**

**class BipGraph**

**{**

**// m and n are number of vertices on left**

**// and right sides of Bipartite Graph**

**int m, n;**

**// adj[u] stores adjacents of left side**

**// vertex 'u'. The value of u ranges from 1 to m.**

**// 0 is used for dummy vertex**

**list<int> \*adj;**

**// These are basically pointers to arrays needed**

**// for hopcroftKarp()**

**int \*pairU, \*pairV, \*dist;**

**public:**

**BipGraph(int m, int n); // Constructor**

**void addEdge(int u, int v); // To add edge**

**// Returns true if there is an augmenting path**

**bool bfs();**

**// Adds augmenting path if there is one beginning**

**// with u**

**bool dfs(int u);**

**// Returns size of maximum matcing**

**int hopcroftKarp();**

**};**

**// Returns size of maximum matching**

**int BipGraph::hopcroftKarp()**

**{**

**// pairU[u] stores pair of u in matching where u**

**// is a vertex on left side of Bipartite Graph.**

**// If u doesn't have any pair, then pairU[u] is NIL**

**pairU = new int[m+1];**

**// pairV[v] stores pair of v in matching. If v**

**// doesn't have any pair, then pairU[v] is NIL**

**pairV = new int[n+1];**

**// dist[u] stores distance of left side vertices**

**// dist[u] is one more than dist[u'] if u is next**

**// to u'in augmenting path**

**dist = new int[m+1];**

**// Initialize NIL as pair of all vertices**

**for (int u=0; u<m; u++)**

**pairU[u] = NIL;**

**for (int v=0; v<n; v++)**

**pairV[v] = NIL;**

**// Initialize result**

**int result = 0;**

**// Keep updating the result while there is an**

**// augmenting path.**

**while (bfs())**

**{**

**// Find a free vertex**

**for (int u=1; u<=m; u++)**

**// If current vertex is free and there is**

**// an augmenting path from current vertex**

**if (pairU[u]==NIL && dfs(u))**

**result++;**

**}**

**return result;**

**}**

**// Returns true if there is an augmenting path, else returns**

**// false**

**bool BipGraph::bfs()**

**{**

**queue<int> Q; //an integer queue**

**// First layer of vertices (set distance as 0)**

**for (int u=1; u<=m; u++)**

**{**

**// If this is a free vertex, add it to queue**

**if (pairU[u]==NIL)**

**{**

**// u is not matched**

**dist[u] = 0;**

**Q.push(u);**

**}**

**// Else set distance as infinite so that this vertex**

**// is considered next time**

**else dist[u] = INF;**

**}**

**// Initialize distance to NIL as infinite**

**dist[NIL] = INF;**

**// Q is going to contain vertices of left side only.**

**while (!Q.empty())**

**{**

**// Dequeue a vertex**

**int u = Q.front();**

**Q.pop();**

**// If this node is not NIL and can provide a shorter path to NIL**

**if (dist[u] < dist[NIL])**

**{**

**// Get all adjacent vertices of the dequeued vertex u**

**list<int>::iterator i;**

**for (i=adj[u].begin(); i!=adj[u].end(); ++i)**

**{**

**int v = \*i;**

**// If pair of v is not considered so far**

**// (v, pairV[V]) is not yet explored edge.**

**if (dist[pairV[v]] == INF)**

**{**

**// Consider the pair and add it to queue**

**dist[pairV[v]] = dist[u] + 1;**

**Q.push(pairV[v]);**

**}**

**}**

**}**

**}**

**// If we could come back to NIL using alternating path of distinct**

**// vertices then there is an augmenting path**

**return (dist[NIL] != INF);**

**}**

**// Returns true if there is an augmenting path beginning with free vertex u**

**bool BipGraph::dfs(int u)**

**{**

**if (u != NIL)**

**{**

**list<int>::iterator i;**

**for (i=adj[u].begin(); i!=adj[u].end(); ++i)**

**{**

**// Adjacent to u**

**int v = \*i;**

**// Follow the distances set by BFS**

**if (dist[pairV[v]] == dist[u]+1)**

**{**

**// If dfs for pair of v also returns**

**// true**

**if (dfs(pairV[v]) == true)**

**{**

**pairV[v] = u;**

**pairU[u] = v;**

**return true;**

**}**

**}**

**}**

**// If there is no augmenting path beginning with u.**

**dist[u] = INF;**

**return false;**

**}**

**return true;**

**}**

**// Constructor**

**BipGraph::BipGraph(int m, int n)**

**{**

**this->m = m;**

**this->n = n;**

**adj = new list<int>[m+1];**

**}**

**// To add edge from u to v and v to u**

**void BipGraph::addEdge(int u, int v)**

**{**

**adj[u].push\_back(v); // Add u to v’s list.**

**}**

**// Driver Program**

**int main()**

**{**

**BipGraph g(4, 4);**

**g.addEdge(1, 2);**

**g.addEdge(1, 3);**

**g.addEdge(2, 1);**

**g.addEdge(3, 2);**

**g.addEdge(4, 2);**

**g.addEdge(4, 4);**

**cout << "Size of maximum matching is " << g.hopcroftKarp();**

**return 0;**

**}**

**HLD**

**vector<int> parent, depth, heavy, head, pos;**

**int cur\_pos;**

**int dfs(int v, vector<vector<int>> const& adj) {**

**int size = 1;**

**int max\_c\_size = 0;**

**for (int c : adj[v]) {**

**if (c != parent[v]) {**

**parent[c] = v, depth[c] = depth[v] + 1;**

**int c\_size = dfs(c, adj);**

**size += c\_size;**

**if (c\_size > max\_c\_size)**

**max\_c\_size = c\_size, heavy[v] = c;**

**}**

**}**

**return size;**

**}**

**int decompose(int v, int h, vector<vector<int>> const& adj) {**

**head[v] = h, pos[v] = cur\_pos++;**

**if (heavy[v] != -1)**

**decompose(heavy[v], h, adj);**

**for (int c : adj[v]) {**

**if (c != parent[v] && c != heavy[v])**

**decompose(c, c, adj);**

**}**

**}**

**void init(vector<vector<int>> const& adj) {**

**int n = adj.size();**

**parent = vector<int>(n);**

**depth = vector<int>(n);**

**heavy = vector<int>(n, -1);**

**head = vector<int>(n);**

**pos = vector<int>(n);**

**cur\_pos = 0;**

**dfs(0, adj);**

**decompose(0, 0, adj);**

**}**

**Sample Query**

**int query(int a, int b) {**

**int res = 0;**

**for (; head[a] != head[b]; b = parent[head[b]]) {**

**if (depth[head[a]] > depth[head[b]])**

**swap(a, b);**

**int cur\_heavy\_path\_max = segment\_tree\_query(pos[head[b]], pos[b]);**

**res = max(res, cur\_heavy\_path\_max);**

**}**

**if (depth[a] > depth[b])**

**swap(a, b);**

**int last\_heavy\_path\_max = segment\_tree\_query(pos[a], pos[b]);**

**res = max(res, last\_heavy\_path\_max);**

**return res;**

**}**

**Centroid Decomposition**

**#pragma GCC optimize("O2")**

**#include <bits/stdc++.h>**

**#define ull unsigned long long**

**#define ll long long**

**#define FAST ios::sync\_with\_stdio(false);cin.tie(0);**

**ll inf = 1LL << 60;**

**ll mod = 1000000007;**

**using namespace std;**

**const int maxn = 100005;**

**vector<vector<int> > g(maxn);**

**vector<int> sizes(maxn, 1), ans(maxn, -1);**

**int dfs(int cur, int p = 0){**

**for(int i=0 ; i<g[cur].size() ; i++){**

**if(p == g[cur][i]) continue;**

**sizes[cur] += dfs(g[cur][i], cur);**

**}**

**return sizes[cur];**

**}**

**void work(int cur, int v){**

**sizes[cur] -= sizes[v];**

**sizes[v] += sizes[cur];**

**}**

**void build(int cur, int par = 0){**

**for(int i=0 ; i<g[cur].size() ; i++){**

**int v = g[cur][i];**

**if(sizes[v]\*2>sizes[cur]){**

**work(cur, v);**

**build(v, par);**

**return;**

**}**

**}**

**sizes[cur] = 0;**

**ans[cur] = ans[par]+1;**

**for(int i=0 ; i<g[cur].size() ; i++){**

**if(sizes[g[cur][i]]) build(g[cur][i], cur);**

**}**

**}**

**Geometry**

**Graham Scan:**

**struct pt {**

**double x, y;**

**};**

**bool cmp(pt a, pt b) {**

**return a.x < b.x || (a.x == b.x && a.y < b.y);**

**}**

**bool cw(pt a, pt b, pt c) {**

**return a.x\*(b.y-c.y)+b.x\*(c.y-a.y)+c.x\*(a.y-b.y) < 0;**

**}**

**bool ccw(pt a, pt b, pt c) {**

**return a.x\*(b.y-c.y)+b.x\*(c.y-a.y)+c.x\*(a.y-b.y) > 0;**

**}**

**void convex\_hull(vector<pt>& a) {**

**if (a.size() == 1)**

**return;**

**sort(a.begin(), a.end(), &cmp);**

**pt p1 = a[0], p2 = a.back();**

**vector<pt> up, down;**

**up.push\_back(p1);**

**down.push\_back(p1);**

**for (int i = 1; i < (int)a.size(); i++) {**

**if (i == a.size() - 1 || cw(p1, a[i], p2)) {**

**while (up.size() >= 2 && !cw(up[up.size()-2], up[up.size()-1], a[i]))**

**up.pop\_back();**

**up.push\_back(a[i]);**

**}**

**if (i == a.size() - 1 || ccw(p1, a[i], p2)) {**

**while(down.size() >= 2 && !ccw(down[down.size()-2], down[down.size()-1], a[i]))**

**down.pop\_back();**

**down.push\_back(a[i]);**

**}**

**}**

**a.clear();**

**for (int i = 0; i < (int)up.size(); i++)**

**a.push\_back(up[i]);**

**for (int i = down.size() - 2; i > 0; i--)**

**a.push\_back(down[i]);**

**}**

**Polygon Area:**

**double area(const vector<point>& fig) {**

**double res = 0;**

**for (unsigned i = 0; i < fig.size(); i++) {**

**point p = i ? fig[i - 1] : fig.back();**

**point q = fig[i];**

**res += (p.x - q.x) \* (p.y + q.y);**

**}**

**return fabs(res) / 2;**

**}**

**Segments Intersect Check:**

**struct pt {**

**long long x, y;**

**pt() {}**

**pt(long long \_x, long long \_y) : x(\_x), y(\_y) {}**

**pt operator-(const pt& p) const { return pt(x - p.x, y - p.y); }**

**long long cross(const pt& p) const { return x \* p.y - y \* p.x; }**

**long long cross(const pt& a, const pt& b) const { return (a - \*this).cross(b - \*this); }**

**};**

**int sgn(const long long& x) { return x >= 0 ? x ? 1 : 0 : -1; }**

**bool inter1(long long a, long long b, long long c, long long d) {**

**if (a > b)**

**swap(a, b);**

**if (c > d)**

**swap(c, d);**

**return max(a, c) <= min(b, d);**

**}**

**bool check\_inter(const pt& a, const pt& b, const pt& c, const pt& d) {**

**if (c.cross(a, d) == 0 && c.cross(b, d) == 0)**

**return inter1(a.x, b.x, c.x, d.x) && inter1(a.y, b.y, c.y, d.y);**

**return sgn(a.cross(b, c)) != sgn(a.cross(b, d)) &&**

**sgn(c.cross(d, a)) != sgn(c.cross(d, b));**

**}**

**Find Intersection of segments:**

**const double EPS = 1E-9;**

**struct pt {**

**double x, y;**

**bool operator<(const pt& p) const**

**{**

**return x < p.x - EPS || (abs(x - p.x) < EPS && y < p.y - EPS);**

**}**

**};**

**struct line {**

**double a, b, c;**

**line() {}**

**line(pt p, pt q)**

**{**

**a = p.y - q.y;**

**b = q.x - p.x;**

**c = -a \* p.x - b \* p.y;**

**norm();**

**}**

**void norm()**

**{**

**double z = sqrt(a \* a + b \* b);**

**if (abs(z) > EPS)**

**a /= z, b /= z, c /= z;**

**}**

**double dist(pt p) const { return a \* p.x + b \* p.y + c; }**

**};**

**double det(double a, double b, double c, double d)**

**{**

**return a \* d - b \* c;**

**}**

**inline bool betw(double l, double r, double x)**

**{**

**return min(l, r) <= x + EPS && x <= max(l, r) + EPS;**

**}**

**inline bool intersect\_1d(double a, double b, double c, double d)**

**{**

**if (a > b)**

**swap(a, b);**

**if (c > d)**

**swap(c, d);**

**return max(a, c) <= min(b, d) + EPS;**

**}**

**bool intersect(pt a, pt b, pt c, pt d, pt& left, pt& right)**

**{**

**if (!intersect\_1d(a.x, b.x, c.x, d.x) || !intersect\_1d(a.y, b.y, c.y, d.y))**

**return false;**

**line m(a, b);**

**line n(c, d);**

**double zn = det(m.a, m.b, n.a, n.b);**

**if (abs(zn) < EPS) {**

**if (abs(m.dist(c)) > EPS || abs(n.dist(a)) > EPS)**

**return false;**

**if (b < a)**

**swap(a, b);**

**if (d < c)**

**swap(c, d);**

**left = max(a, c);**

**right = min(b, d);**

**return true;**

**} else {**

**left.x = right.x = -det(m.c, m.b, n.c, n.b) / zn;**

**left.y = right.y = -det(m.a, m.c, n.a, n.c) / zn;**

**return betw(a.x, b.x, left.x) && betw(a.y, b.y, left.y) &&**

**betw(c.x, d.x, left.x) && betw(c.y, d.y, left.y);**

**}**

**}**

**Circle Line intersection:**

**double r, a, b, c; // given as input**

**double x0 = -a\*c/(a\*a+b\*b), y0 = -b\*c/(a\*a+b\*b);**

**if (c\*c > r\*r\*(a\*a+b\*b)+EPS)**

**puts ("no points");**

**else if (abs (c\*c - r\*r\*(a\*a+b\*b)) < EPS) {**

**puts ("1 point");**

**cout << x0 << ' ' << y0 << '\n';**

**}**

**else {**

**double d = r\*r - c\*c/(a\*a+b\*b);**

**double mult = sqrt (d / (a\*a+b\*b));**

**double ax, ay, bx, by;**

**ax = x0 + b \* mult;**

**bx = x0 - b \* mult;**

**ay = y0 - a \* mult;**

**by = y0 + a \* mult;**

**puts ("2 points");**

**cout << ax << ' ' << ay << '\n' << bx << ' ' << by << '\n';**

**}**

**Check if point inside convex polygon:**

**struct pt{**

**long long x, y;**

**pt(){}**

**pt(long long \_x, long long \_y):x(\_x), y(\_y){}**

**pt operator+(const pt & p) const { return pt(x + p.x, y + p.y); }**

**pt operator-(const pt & p) const { return pt(x - p.x, y - p.y); }**

**long long cross(const pt & p) const { return x \* p.y - y \* p.x; }**

**long long dot(const pt & p) const { return x \* p.x + y \* p.y; }**

**long long cross(const pt & a, const pt & b) const { return (a - \*this).cross(b - \*this); }**

**long long dot(const pt & a, const pt & b) const { return (a - \*this).dot(b - \*this); }**

**long long sqrLen() const { return this->dot(\*this); }**

**};**

**bool lexComp(const pt & l, const pt & r){**

**return l.x < r.x || (l.x == r.x && l.y < r.y);**

**}**

**int sgn(long long val){**

**return val > 0 ? 1 : (val == 0 ? 0 : -1);**

**}**

**vector<pt> seq;**

**int n;**

**bool pointInTriangle(pt a, pt b, pt c, pt point){**

**long long s1 = abs(a.cross(b, c));**

**long long s2 = abs(point.cross(a, b)) + abs(point.cross(b, c)) + abs(point.cross(c, a));**

**return s1 == s2;**

**}**

**void prepare(vector<pt> & points){**

**n = points.size();**

**int pos = 0;**

**for(int i = 1; i < n; i++){**

**if(lexComp(points[i], points[pos]))**

**pos = i;**

**}**

**rotate(points.begin(), points.begin() + pos, points.end());**

**n--;**

**seq.resize(n);**

**for(int i = 0; i < n; i++)**

**seq[i] = points[i + 1] - points[0];**

**}**

**bool pointInConvexPolygon(pt point){**

**if(seq[0].cross(point) != 0 && sgn(seq[0].cross(point)) != sgn(seq[0].cross(seq[n - 1])))**

**return false;**

**if(seq[n - 1].cross(point) != 0 && sgn(seq[n - 1].cross(point)) != sgn(seq[n - 1].cross(seq[0])))**

**return false;**

**if(seq[0].cross(point) == 0)**

**return seq[0].sqrLen() >= point.sqrLen();**

**int l = 0, r = n - 1;**

**while(r - l > 1){**

**int mid = (l + r)/2;**

**int pos = mid;**

**if(seq[pos].cross(point) >= 0)l = mid;**

**else r = mid;**

**}**

**int pos = l;**

**return pointInTriangle(seq[pos], seq[pos + 1], pt(0, 0), point);**

**}**

**Closest Pair of Points:**

**struct pt {**

**int x, y, id;**

**};**

**struct cmp\_x {**

**bool operator()(const pt & a, const pt & b) const {**

**return a.x < b.x || (a.x == b.x && a.y < b.y);**

**}**

**};**

**struct cmp\_y {**

**bool operator()(const pt & a, const pt & b) const {**

**return a.y < b.y;**

**}**

**};**

**int n;**

**vector<pt> a;**

**double mindist;**

**pair<int, int> best\_pair;**

**void upd\_ans(const pt & a, const pt & b) {**

**double dist = sqrt((a.x - b.x)\*(a.x - b.x) + (a.y - b.y)\*(a.y - b.y));**

**if (dist < mindist) {**

**mindist = dist;**

**best\_pair = {a.id, b.id};**

**}**

**}**

**vector<pt> t;**

**void rec(int l, int r) {**

**if (r - l <= 3) {**

**for (int i = l; i < r; ++i) {**

**for (int j = i + 1; j < r; ++j) {**

**upd\_ans(a[i], a[j]);**

**}**

**}**

**sort(a.begin() + l, a.begin() + r, cmp\_y());**

**return;**

**}**

**int m = (l + r) >> 1;**

**int midx = a[m].x;**

**rec(l, m);**

**rec(m, r);**

**merge(a.begin() + l, a.begin() + m, a.begin() + m, a.begin() + r, t.begin(), cmp\_y());**

**copy(t.begin(), t.begin() + r - l, a.begin() + l);**

**int tsz = 0;**

**for (int i = l; i < r; ++i) {**

**if (abs(a[i].x - midx) < mindist) {**

**for (int j = tsz - 1; j >= 0 && a[i].y - t[j].y < mindist; --j)**

**upd\_ans(a[i], t[j]);**

**t[tsz++] = a[i];**

**}**

**}**

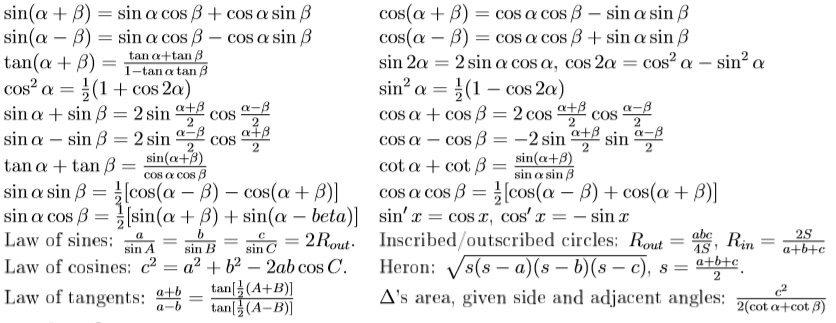
**}**

**/\*t.resize(n);**

**sort(a.begin(), a.end(), cmp\_x());**

**mindist = 1E20;**

**rec(0, n);\*/**

****

**Game Theory**

**Nim: To calculate the Grundy value of a given state you need to:**

***-Get all possible transitions from this state***

***-Each transition can lead to a sum of independent games (one game in the degenerate case). Calculate the Grundy value for each independent game and xor-sum them. Of course xor does nothing if there is just one game.***

***-After we calculated Grundy values for each transition we find the state's value as the mex of these numbers.***

***-If the value is zero, then the current state is losing, otherwise it is winning.***

**In comparison to the previous section, we take into account the fact that there can be transitions to combined games. We consider them a Nim with pile sizes equal to the independent games' Grundy values. We can xor-sum them just like usual Nim according to Bouton's theorem.**

**Example:**

**The rules. Consider a checkered strip of size 1×n. In one move, the player must put one cross, but it is forbidden to put two crosses next to each other (in adjacent cells). As usual, the player without a valid move loses.**

**The solution. When a player puts a cross in any cell, we can think of the strip being split into two independent parts: to the left of the cross and to the right of it. In this case, the cell with a cross, as well as its left and right neighbours are destroyed — nothing more can be put in them. Therefore, if we number the cells from 1 to n then putting the cross in position 1<i<n breaks the strip into two strips of length i−2 and n−i−1 i.e. we go to the sum of games i−2 and n−i−1. For the edge case of the cross being marked on position 1 or n, we go to the game n−2.**

**Thus, the Grundy value g(n) has the form:**

**g(n)=mex({g(n−2)}∪{g(i−2)⊕g(n−i−1)∣2≤i≤n−1}).**

**So we've got a O(n²) solution.**

**In fact, g(n) has a period of length 34 starting with n=52**

**Josephus**

**int josephus(int n, int k) {**

**if (n == 1)**

**return 0;**

**if (k == 1)**

**return n-1;**

**if (k > n)**

**return (joseph(n-1, k) + k) % n;**

**int cnt = n / k;**

**int res = joseph(n - cnt, k);**

**res -= n % k;**

**if (res < 0)**

**res += n;**

**else**

**res += res / (k - 1);**

**return res;**

**}**

**Stable Marriage**

**// C++ program for stable marriage problem**

**#include <iostream>**

**#include <string.h>**

**#include <stdio.h>**

**using namespace std;**

**// Number of Men or Women**

**#define N  4**

**// This function returns true if woman 'w' prefers man 'm1' over man 'm'**

**bool wPrefersM1OverM(int prefer[2\*N][N], int w, int m, int m1)**

**{**

**// Check if w prefers m over her current engagment m1**

**for (int i = 0; i < N; i++)**

**{**

**// If m1 comes before m in lisr of w, then w prefers her**

**// cirrent engagement, don't do anything**

**if (prefer[w][i] == m1)**

**return true;**

**// If m cmes before m1 in w's list, then free her current**

**// engagement and engage her with m**

**if (prefer[w][i] == m)**

**return false;**

**}**

**}**

**// Prints stable matching for N boys and N girls. Boys are numbered as 0 to**

**// N-1. Girls are numbereed as N to 2N-1.**

**void stableMarriage(int prefer[2\*N][N])**

**{**

**// Stores partner of women. This is our output array that**

**// stores paing information.  The value of wPartner[i]**

**// indicates the partner assigned to woman N+i.  Note that**

**// the woman numbers between N and 2\*N-1. The value -1**

**// indicates that (N+i)'th woman is free**

**int wPartner[N];**

**// An array to store availability of men.  If mFree[i] is**

**// false, then man 'i' is free, otherwise engaged.**

**bool mFree[N];**

**// Initialize all men and women as free**

**memset(wPartner, -1, sizeof(wPartner));**

**memset(mFree, false, sizeof(mFree));**

**int freeCount = N;**

**// While there are free men**

**while (freeCount > 0)**

**{**

**// Pick the first free man (we could pick any)**

**int m;**

**for (m = 0; m < N; m++)**

**if (mFree[m] == false)**

**break;**

**// One by one go to all women according to m's preferences.**

**// Here m is the picked free man**

**for (int i = 0; i < N && mFree[m] == false; i++)**

**{**

**int w = prefer[m][i];**

**// The woman of preference is free, w and m become**

**// partners (Note that the partnership maybe changed**

**// later). So we can say they are engaged not married**

**if (wPartner[w-N] == -1)**

**{**

**wPartner[w-N] = m;**

**mFree[m] = true;**

**freeCount--;**

**}**

**else  // If w is not free**

**{**

**// Find current engagement of w**

**int m1 = wPartner[w-N];**

**// If w prefers m over her current engagement m1,**

**// then break the engagement between w and m1 and**

**// engage m with w.**

**if (wPrefersM1OverM(prefer, w, m, m1) == false)**

**{**

**wPartner[w-N] = m;**

**mFree[m] = true;**

**mFree[m1] = false;**

**}**

**} // End of Else**

**} // End of the for loop that goes to all women in m's list**

**} // End of main while loop**

**// Print the solution**

**cout << "Woman   Man" << endl;**

**for (int i = 0; i < N; i++)**

**cout << " " << i+N << "\t" << wPartner[i] << endl;**

**}**

**// Driver program to test above functions**

**int main()**

**{**

**int prefer[2\*N][N] = { {7, 5, 6, 4},**

**{5, 4, 6, 7},**

**{4, 5, 6, 7},**

**{4, 5, 6, 7},**

**{0, 1, 2, 3},**

**{0, 1, 2, 3},**

**{0, 1, 2, 3},**

**{0, 1, 2, 3},**

**};**

**stableMarriage(prefer);**

**return 0;**

**}**