```
#include <fstream>

ifstream in("datos.txt");
auto cinbuf = cin.rdbuf(in.rdbuf());

cin.rdbuf(cinbuf);

#include <sstream>
string s;      //Puede ser necesario un getline extra
getline(cin, s); //para saltar de la linea anterior
stringstream ss(s);
ss >> num;

// PRECISION EN DOUBLE

cout << fixed << setprecision(4) << num;

// DIJKSTRA
using ii = pair<int, int>;
using vi = vector<int>;
using vvi = vector<vi>;
vvi adjList;
void dijkstra(int s, vi& dist) {
  dist.assign(adjList.size(), INF);
  dist[s] = 0;
  priority_queue<ii, vii, greater<ii>> pq;
  pq.push({0, s});
  while (!pq.empty()) {
    ii front = pq.top(); pq.pop();
    int d = front.first, u = front.second;
    if (d > dist[u]) continue;
    for (auto a : adjList[u]) {
      if (dist[u] + a.first < dist[a.second]) {
        dist[a.second] = dist[u] + a.first;
        pq.push({dist[a.second], a.second});
      }
    }
  }
}

int V; vvi adjMat, camino;
void floyd() {
  for (int k = 0; k < V; k++)
    for (int i = 0; i < V; i++)
      for (int j = 0; j < V; j++)
        if (adjMat[i][k] + adjMat[k][j] < adjMat[i][j]) {
          adjMat[i][j] = adjMat[i][k] + adjMat[k][j];
          camino[i][j] = k;
        }
}
void reconstruct(int src, int dst) {
  if (src == dst) return;
  int k = camino[src][dst];
  if (k == dst) {
    cout << ' ' << dst; return;
  }
  reconstruct(src, k);
  reconstruct(k, dst);
}
adjMat = vvi(V, vi(V, INF));
camino = vvi(V, vi(V));
//Valores iniciales:
adjMat[u][v] = distancia;
camino[u][v] = v;

// BUSCA CICLOS EN GRAFO DIRIGIDO

int V; vvi grafo; vi estado; //Tam V, no_visto
int const NO_VISTO = 0, TOCADO = 1, HUNDIDO = 2;
bool dfs(int u) {
  estado[u] = TOCADO;
  bool b = false;
  for (int i = 0; i < grafo[u].size(); ++i) {
    int v = grafo[u][i];
    if (estado[v] == TOCADO)
      b = true;
    else if ((estado[v] == NO_VISTO) && dfs(v))
      b = true;
  }
  estado[u] = HUNDIDO;
  return b;
}

// BUSCA PUENTES / PTS ARTICULACION

int V; vvi grafo;/*vi cuenta(V,0)*/
int hora; vi horaVertice, alcanzable;
void dfs(int u, int uParent) {
  horaVertice[u] = alcanzable[u] = hora; hora++;
  for (int i = 0; i < grafo[u].size(); ++i) {
    int v = grafo[u][i];
    if (v == uParent) continue; // BackEdge con el padre
    if (horaVertice[v] == 0) { // No visitado
      dfs(v, u);
      if (alcanzable[v] >/*>=*/ horaVertice[u]) {
        // La arista u-v es un puente / cuenta[u]++
      }          //<algorithm>
      alcanzable[u] = min(alcanzable[u], alcanzable[v]);
    }
    else // BackEdge
      alcanzable[u] = min(alcanzable[u], horaVertice[v]);
  }
}
hora = 1;
horaVertice = vi(V, 0); alcanzable = vi(V, -1);
for (int i = 0; i < V; ++i) {
  if (!horaVertice[i]) {
    //cuenta[i]=-1
    dfs(i, 0);
  }
}
```

```
//          TSP

int V; vvi adjMat, memo; // tabla de DP
int tsp(int pos, int visitados) {
  if (visitados == (1 << V) - 1)//hemos visto todos
    return adjMat[pos][0]; // volvemos al origen
  if (memo[pos][visitados] != -1)
    return memo[pos][visitados];
  int res = 1000000000; // INF
  for (int i = 1; i < V; ++i)
    if (!( visitados & (1 << i))) //no hemos visitado i
      res=min(res,adjMat[pos][i]+tsp(i,visitados
        | (1 << i)));
  return memo[pos][visitados] = res;
}

adjMat = vvi(V, vi(V));
memo = vvi(V, vi(1 << V, -1));
cout << tsp(0, 1) << '\n';

// UNION FIND

struct UFDS {
  vector<int> p;
  int numSets;
  UFDS(int n) : p(n, 0), numSets(n) {
    for (int i = 0; i < n; ++i) p[i] = i;
  }
  int find(int x) {
    return (p[x] == x) ? x : p[x] = find(p[x]);
  }
  void merge(int x, int y) {
    int i = find(x), j = find(y);
    if (i == j) return;
    p[i] = j;
    --numSets;
  }
};

// MAX CARDINALITY BIPARTITE MATCHING
// MAX INDEPENDENT SET: MCBM + MIS = V
// MIN VERTEX COVER: MVC = MCBM
int M, N; vvi grafo; //Solo los primeros M vertices
vi match, vis;
int aug(int l) {
  if (vis[l]) return 0;
  vis[l] = 1;
  for (auto r : grafo[l])
    if (match[r] == -1 || aug(match[r])) {
      match[r] = l;
      return 1;
    }
  return 0;
}
int berge_mcbm() {
  int mcbm = 0;
  match.assign(N + M, -1);
  for (int l = 0; l < M; l++) {
    vis.assign(M, 0);
    mcbm += aug(l);
  }
  return mcbm;
}
```

```
// MAX FLOW

int V, sumidero;
vector<bool> visited; vvi grafo, adjMat;
int dfs(int u, int flow) {
  visited[u] = true;
  if (u == sumidero) return flow;
  for (int v : grafo[u]) {
    if (! visited[v] && adjMat[u][v] > 0) {
      int sol = dfs(v, min(flow, adjMat[u][v]));
      if (sol > 0) {
        adjMat[u][v] -= sol;
        adjMat[v][u] += sol;
        return sol; }
    }
  }
  return 0;
}
int maxFlow(int s, int t) {
  int ret = 0; sumidero = t;
  int flow = 0;
  do {
    visited = vector<bool>(V, false);
    flow = dfs(s, INF);
    ret += flow;
  } while (flow > 0);
  return ret;
}

// MATES
//Con criba d erastotenes hasta n, podemos decidir
// si un num es primo hasta n^2
using lli = long long int;
int gcd(int a, int b) {
  return b == 0 ? a : gcd(b, a % b);
}
// COEFICIENTES BEZOUT
int extendedEuclidRec(int a, int b, int& u, int& v) {
  if (!b) {u = 1; v = 0; return a;}
  int r = extendedEuclidRec(b, a % b, u, v);
  int uAux = v;
  int vAux = u - (a / b) * v;
  u = uAux;
  v = vAux;
  return r;
}
typedef pair<unsigned int, unsigned int> uu;
// <inicio ciclo, longitud ciclo>
uu floydCycleFinding(unsigned int x0) {
  int tortoise = f(x0), hare = f(f(x0));
  while (tortoise != hare) {
    tortoise = f(tortoise);
    hare = f(f(hare));
  }
  int mu = 0; hare = x0;
  while (tortoise != hare) {
    tortoise = f(tortoise);
    hare = f(hare); mu++;
  }
  int lambda = 1; hare = f(tortoise);
  while (tortoise != hare) {
    hare = f(hare); lambda++;
  }
  return uu(mu, lambda);
```

```
// GEOMETRIA

#include <iostream,vector,algorithm,
set,iomanip,cmath,random,tuple,cassert>
#include <iostream>
#include <vector>
#include<algorithm>
#include<set>
#include<iomanip>
#include <cmath>
#include<random>
#include<tuple>
#include<cassert>
using namespace std;
const double EPS = 1e-9;
const double PI = acos(-1);
using T = double;
struct pt {
  T x, y;
  pt operator+(pt p) const { return { x + p.x, y + p.y }; }
  pt operator-(pt p) const { return { x - p.x, y - p.y }; }
  pt operator*(T d) const { return { x * d, y * d }; }
  pt operator/(T d) const { return { x / d, y / d }; }
  bool operator==(pt o) const {return x == o.x && y == o.y;}
  bool operator!=(pt o) const { return !(* this == o); }
  bool operator<(pt o) const {
    if (x == o.x) return y < o.y;
    return x < o.x;
  }
};
T distsq(pt v) { return v.x * v.x + v.y * v.y; }
double modulo(pt v) { return sqrt(distsq(v)); }

T dot(pt v, pt w) { return v.x * w.x + v.y * w.y; }
bool isPerp(pt v, pt w) { return dot(v, w) == 0; }
double angle(pt v, pt w) {
  double cosTheta = dot(v, w) / modulo(v) / modulo(w);
  return acos(max(-1.0, min(1.0, cosTheta)));
}
T cross(pt v, pt w) { return v.x * w.y - v.y * w.x; }
// +/0/-: c a la izquieda/contenido/derecha de a-b
T orient(pt a, pt b, pt c) { return cross(b - a, c - a); }
bool inAngle(pt a, pt b, pt c, pt p) {
  assert(orient(a, b, c) != 0);
  if (orient(a, b, c) < 0) swap(b, c);
  return orient(a, b, p) >= 0 && orient(a, c, p) <= 0;
}

double orientedAngle(pt a, pt b, pt c) {
  if (orient(a, b, c) >= 0)
    return angle(b - a, c - a);
  else
    return 2 * PI - angle(b - a, c - a);
}
// if strict , returns false when A is on the boundary
bool inPolygon(vector<pt> const& p, pt a, bool strict ) {
  int numCrossings = 0;
  for (int i = 0, n = int(p.size()) - 1; i < n; ++i) {
    if (onSegment(p[i], p[i + 1], a))
      return ! strict ;
    numCrossings += crossesRay(a, p[i], p[i + 1]);
  }
  return numCrossings & 1;//inside if odd n crossings
}

// ORDENACIÓN POLAR
bool half(pt p) { // true if in blue half
  assert(p.x != 0 || p.y != 0);
  // the argument of (0,0) is undefined
  return p.y > 0 || (p.y == 0 && p.x < 0);
}
void polarSort(vector<pt>& v) {
  sort(v.begin(), v.end(), [](pt v, pt w) {
    return make_tuple(half(v), 0, distsq(v)) <
    make_tuple(half(w), cross(v, w), distsq(w));});
}

pt translate(pt v, pt p) { return p + v; }
pt scale(pt c, double factor, pt p) {
  return c + (p - c) * factor;
}
// rotate p by a certain angle a contrareloj around origin
pt rotate(pt p, double a) {
return{p.x*cos(a) - p.y*sin(a),p.x*sin(a) + p.y*cos(a)};
}} // rotate 90 counterclockwise
pt perp(pt p) { return { -p.y, p.x }; }

struct line {
  pt v; T c;
  // v:vector director , c: punto
  line(pt v, T c) : v(v), c(c) {}
  // from equation ax + by = c
  line(T a, T b, T c) : v({ b,-a }), c(c) {}
  // from points p and q
  line(pt p, pt q) : v(q - p), c(cross(v, p)) {}
  T side(pt p) { return cross(v, p) - c; }
  double dist(pt p) { return abs(side(p)) / modulo(v); }
  line translate(pt t) { return { v, c + cross(v,t) }; }
  pt proj(pt p) { return p - perp(v) * side(p) / distsq(v); }
};

bool inter(line l1, line l2, pt& out) {
  T d = cross(l1.v, l2.v);
  if (d == 0) return false;
  out = (l2.v * l1.c - l1.v * l2.c) / d;
  return true;
}
// POLIGONOS, el primer y último puntos coinciden
double areaTriangle(pt a, pt b, pt c) {
  return abs(cross(b - a, c - a)) / 2.0;
}
double areaPolygon(vector<pt> const& p) {
  double area = 0.0;
  for (int i = 0, n = int(p.size()) - 1; i < n; ++i) {
    area += cross(p[i], p[i + 1]);
  }
  return abs(area) / 2.0;
}

bool isConvex(vector<pt> const& p) {
  bool hasPos = false, hasNeg = false;
  for (int i = 0, n = (int)p.size(); i < n; ++i) {
    int o = orient(p[i], p[(i + 1) % n], p[(i + 2) % n]);
    if (o > 0) hasPos = true;
    if (o < 0) hasNeg = true;
  }
  return !(hasPos && hasNeg);
}
```

```
// ENVOLVENTE CONVEXA
// para aceptar puntos colineales cambia a >=
// returns true if point r is on the left side of line pq
bool ccw(pt p, pt q, pt r) {
  return orient(p, q, r) > 0;
}
vector<pt> convexHull(vector<pt>& P) {
  int n = int(P.size()), k = 0;
  vector<pt> H(2 * n);
  sort(P.begin(), P.end());
  // build lower hull
  for (int i = 0; i < n; ++i) {
    while (k >= 2 && !ccw(H[k - 2], H[k - 1], P[i])) --k;
    H[k++] = P[i];
  }
  // build upper hull
  for (int i = n - 2, t = k + 1; i >= 0; --i) {
    while (k >= t && !ccw(H[k - 2], H[k - 1], P[i])) --k;
    H[k++] = P[i];
  }
  H.resize(k);
  return H;
}
// DIVISIÓN DE UN POLÍGONO
vector<pt> cutPolygon(pt a, pt b, vector<pt> const& P) {
  vector<pt> R; pt c;
  for (int i = 0; i < int(P.size()) - 1; i++) {
    double left1 = cross(b - a, P[i] - a),
      left2 = cross(b - a, P[i + 1] - a);
    if (left1 >= 0) R.push_back(P[i]);
    if (left1 * left2 < 0) {
      inter({ P[i], P[i + 1] }, { a,b }, c);
      R.push_back(c);
    }
  }
  if (!R.empty())
    R.push_back(R[0]); //Polig empiezan como acaban
  return R;
}
```

```
string T, P;
vi b; // back table
int n, m; // n = length of T, m = length of P
void kmpPreprocess() { // before calling kmpSearch
  b = vi(m + 1);
  int i = 0, j = -1; b[0] = -1;
  while (i < m) {
    while (j >= 0 && P[i] != P[j]) j = b[j];
    ++i; ++j;
    b[i] = j;
  }
}

void kmpSearch() {
  int i = 0, j = 0;
  while (i < n) {
    while (j >= 0 && T[i] != P[j])
      j = b[j]; // different reset j using b
    ++i; ++j; // same, advance both pointers
    if (j == m) {
      printf("P is found at index %d in T\n", i - j);
      j = b[j];
    }
  }
}
```

```
const int MAXN = 26;
class Trie {
  int prefixes;
  int words;
  std::vector<Trie*> child;
public:
  Trie() : prefixes(0), words(0), child(MAXN, nullptr) {}
  ~Trie() {
    for (int i = 0; i < MAXN; ++i)
      delete child[i];
  }
  void add(const char* s) {
    if (*s == '\0') ++words;
    else {
      Trie* t;
      if (child[*s - 'a'] == nullptr) {
        t = child[*s - 'a'] = new Trie();
        t->prefixes = 1;
      }
      else {
        t = child[*s - 'a'];
        t->prefixes++;
      }
      t->add(s + 1);
    }
  }
};
```

```
#include<cstring>
#define MAX_N 100010
std::string T;
int n;
int RA[MAX_N], tempRA[MAX_N], tempSA[MAX_N];
int SA[MAX_N]; //Posicion de comienzo de la subcadena
int c[MAX_N];

void countingSort(int k) {
  int i, sum, maxi = std::max(300, n); // up to 255 ASCII
  memset(c, 0, sizeof c);
  for (i = 0; i < n; ++i)
    ++c[i + k < n ? RA[i + k] : 0];
  for (i = sum = 0; i < maxi; ++i) {
    int t = c[i]; c[i] = sum; sum += t;
  }
  for (i = 0; i < n; ++i)
    tempSA[c[SA[i] + k < n ? RA[SA[i] + k] : 0]++] = SA[i];
  for (i = 0; i < n; ++i)
    SA[i] = tempSA[i];
}

void constructSA() {
  int i, k, r;
  for (i = 0; i < n; ++i) RA[i] = T[i];
  for (i = 0; i < n; ++i) SA[i] = i;
  for (k = 1; k < n; k <<= 1) {
    countingSort(k);
    countingSort(0);
    tempRA[SA[0]] = r = 0;
    for (i = 1; i < n; ++i)
      tempRA[SA[i]] =
      (RA[SA[i]] == RA[SA[i - 1]] &&
        RA[SA[i] + k] == RA[SA[i - 1] + k]) ?
      r : ++r;
    for (i = 0; i < n; ++i)
      RA[i] = tempRA[i];
    if (RA[SA[n - 1]] == n - 1) break;
  }
}
```

```
int LCP[MAX_N];
void computeLCP() {
  int Phi[MAX_N];
  int PLCP[MAX_N];
  int i, L;
  Phi[SA[0]] = -1;
  for (i = 1; i < n; ++i)
    Phi[SA[i]] = SA[i - 1];
  for (i = L = 0; i < n; ++i) {
    if (Phi[i] == -1) { PLCP[i] = 0; continue; }
    while (T[i + L] == T[Phi[i] + L]) ++L;
    PLCP[i] = L;
    L = std::max(L - 1, 0);
  }
  for (i = 0; i < n; ++i)
    LCP[i] = PLCP[SA[i]];
}
```

```
// NO USAR EL INDICE 0 !!
```

```
class FenwickTree {
private:
  vector<int> ft;
public:
  FenwickTree(int n) { ft.assign(n + 1, 0); }
  int getSum(int b) {
    int ret = 0;
    while (b) {
      ret += ft[b]; //OPERADOR
      b -= (b & -b);
    }
    return ret;
  }
  void add(int pos, int val) {
    while (pos < ft.size()) {
      ft[pos] += val; //OPERADOR
      pos += (pos & -pos);
    }
  }
  int getSum(int a, int b) {
    return getSum(b) - getSum(a - 1);} //OPERADOR
  int getValue(int pos) {
    return getSum(pos) - getSum(pos - 1);} //OPERADOR
  void setValue(int pos, int val) {
    add(pos, val - getValue(pos));} //OPERADOR
};
```

```
int idx; //Siguiente entrada en euler y prof
int euler[2 * MAX_V - 1];
int prof[2 * MAX_V - 1];//Prof. del nodo en euler[]
int first[MAX_V];//Primera aparicion del nodo i en euler[]
void eulerTour(int u, int parent, int d) { // d = depth
  first[u] = idx; euler[idx] = u; prof[idx] = d; ++idx;
  for (int i = 0; i < adj[u].size(); ++i) {
    int v = adj[u][i];
    if (v == parent) continue;
    eulerTour(v, u, d + 1);
    euler[idx] = u; prof[idx] = d; ++idx;
  }
}
int lca(int u, int v) {
  return euler[st.query(first[u], first[v])];
}
```

```
class SegmentTree {
  vector<int> st;
  int tam;
public:

  SegmentTree(int maxN) {
    st.reserve(4 * maxN + 10);
  }
  int query(int a, int b) {
    return query(1, 0, tam - 1, a, b);
  }
  int query(int vertex, int L, int R, int i, int j) {
    if (i > R || j < L) {
      return 0;
    }
    if (L >= i && R <= j)
      // Segmento completamente dentro de la consulta
      return st[vertex];
    int mitad = (L + R) / 2;
    return query(2 * vertex, L, mitad, i, j) + //OPERADOR
      query(2 * vertex + 1, mitad + 1, R, i, j);
  }

  void update(int pos, int newVal) {
    update(1, 0, tam - 1, pos, newVal);
  }
  void update(int vertex, int l, int r, int pos, int newVal) {
    if ((pos < l) || (r < pos)) return;
    if (l == r) {
      st[vertex] = newVal;
      return;
    }
    int m = (l + r) / 2;
    if ((l <= pos) && (pos <= m))
      update(2 * vertex, l, m, pos, newVal);
    else
      update(2 * vertex + 1, m + 1, r, pos, newVal);
    st[vertex] = st[2 * vertex] + st[2 * vertex + 1]; //OPERADO
  }

  void build(vector<int> values, int n) {
    tam = n;
    build(values, 1, 0, n - 1);
  }
  void build(vector<int> values, int p, int l, int r) {
    if (l == r) {
      st[p] = values[l];
      return;
    }
    int m = (l + r) / 2;
    build(values, 2 * p, l, m);
    build(values, 2 * p + 1, m + 1, r);
    st[p] = st[2 * p] + st[2 * p + 1]; //OPERADOR
  }
};
```

```
class SegmentTree {
```

```
class SegmentTree { //CON range-update
```

```
  vector<int> st;
  vector<int> lazy;
  int tam; // Numero de hojas que manejamos

  void setLazyUpdate(int vertex, int value) {
    // Mezclamos
    // Importante +=: el nodo podria tener
    // otras operaciones pendientes anteriores
    lazy[vertex] += value; //modi
  }

  void pushLazyUpdate(int vertex, int L, int R) {
    st[vertex] += (R - L + 1) * lazy[vertex]; //modi
    if (L != R) {
      // Tenemos hijos
      int m = (L + R) / 2;
      setLazyUpdate(2 * vertex, lazy[vertex]);
      setLazyUpdate(2 * vertex + 1, lazy[vertex]);
    }
    lazy[vertex] = 0; //modi
  }

public:
  // Tamano maximo que podremos guardar
  // (numero de hojas).
  // Antes de las consultas, se necesita rellenar
  // con los datos iniciales usando build().
  SegmentTree(int maxN) {
    st.assign(4 * maxN + 10, 0);
    lazy.assign(4 * maxN + 10, 0); //modi
  }

  int query(int a, int b) {
    return query(1, 0, tam - 1, a, b);
  }
  int query(int vertex, int L, int R, int i, int j) {
    pushLazyUpdate(vertex, L, R);

    if (i > R || j < L) {
      return 0;
    }
    if (L >= i && R <= j)
      // Segmento completamente dentro de la consulta
      return st[vertex];
    int mitad = (L + R) / 2;
    return query(2 * vertex, L, mitad, i, j) +
      query(2 * vertex + 1, mitad + 1, R, i, j); //modi
  }

  void update(int pos, int newVal) {
    update(1, 0, tam - 1, pos, newVal);
  }
  void update(int vertex, int l, int r,
    int pos, int newVal) {
    if ((pos < l) || (r < pos)) return;
    if (l == r) {
      st[vertex] = newVal;
      return;
    }
    int m = (l + r) / 2;
    if ((l <= pos) && (pos <= m))
      update(2 * vertex, l, m, pos, newVal);
    else
      update(2 * vertex + 1, m + 1, r, pos, newVal);
    st[vertex] = st[2 * vertex] + st[2 * vertex + 1]; //modi
  }

  void updateRange(int a, int b, int op) {
    updateRange(1, 0, tam - 1, a, b, op);
  }
  void updateRange(int vertex, int L, int R,
    int a, int b, int op) {
    // Resolvemos posibles operaciones pendientes
    pushLazyUpdate(vertex, L, R);
    if ((b < L) || (R < a)) return;
    // ¡Intervalo afectado por completo?
    if ((a <= L) && (R <= b)) {
      // Nos aplicamos la operacion y propagamos la
      // pereza a los hijos. Para evitar copiar/pegar,
      // lo hacemos aplicandonos la pereza, y luego
      // resolviendola
      setLazyUpdate(vertex, op);
      pushLazyUpdate(vertex, L, R);
      return;
    }
    // Intervalo no afectado por completo. No podemos
      // ser perezosos. Aplicamos la operacion en
      // los hijos
    int m = (L + R) / 2;
    updateRange(2 * vertex, L, m, a, b, op); //ESENCIAL, o al me
    updateRange(2 * vertex + 1, m + 1, R, a, b, op); //ESENCIAL,
    // Combinamos
    st[vertex] = st[2 * vertex] + st[2 * vertex + 1]; //modi
  }

  void build(vector<int> const& values, int n) {
    tam = n;
    build(values, 1, 0, n - 1);
  }
  void build(vector<int> const& values, int p, int l, int r) {
    if (l == r) {
      st[p] = values[l];
      return;
    }
    int m = (l + r) / 2;
    build(values, 2 * p, l, m);
    build(values, 2 * p + 1, m + 1, r);
    st[p] = st[2 * p] + st[2 * p + 1]; //modi
  }
};
```