```
/* Determinante de una matriz */

bool used[50];
double det(double matrix[50][50], int n) {
    double res = 1;
    for(int i = 0; i < n; i++) { int p;
        for(p = 0; p < n; p++)
            if(!used[p] && fabs(matrix[p][i]) > EPS) break;
        if (p >= n) return 0;
        res *= matrix[p][i]; used[p] = true;
        double z = 1 / matrix[p][i];
        for (int j = 0; j < n; j++) matrix[p][j] *= z;
        for (int j = 0; j < n; ++j)
            if (j != p) { z = matrix[j][i];
                for (int k = 0; k < n; ++k)
                    matrix[j][k] -= z * matrix[p][k];
            }
    }
    return res;
}

/* Invertir una matriz */

// A -> Invertir, B -> Identidad
bool invert(double **A, double **B, int N) {
    int i, j, k, jmax; double tmp;
    for(i = 1; i <= N; i++) {
        jmax = i;
        for(j = i+1; j <= N; j++)
            if(fabs(A[j][i]) > fabs(A[jmax][i])) jmax = j;
        for(j = 1; j <= N; j++) {
            swap(A[i][j], A[jmax][j]); swap(B[i][j], B[jmax][j]);
        }
        if(fabs(A[i][i]) < EPS)
            return false;
        tmp = A[i][i]; // Normalize row i
        for(j = 1; j <= N; j++) {
            A[i][j] /= tmp; B[i][j] /= tmp;
        }
        // Eliminate non-zero values in column i
        for(j = 1; j <= N; j++) {
            if(i == j) continue;
            tmp = A[j][i];
            for(k = 1; k <= N; k++) {
                A[j][k] -= A[i][k]*tmp; B[j][k] -= B[i][k]*tmp;
            }
        }
    }
    return true;
}
```

```
/* GCD, EGCD, INV, CRT */

typedef pair<int, int> pii;
int gcd(int a, int b) {
    if(b == 0) return a;
    return gcd(b, a%b);
}
// gcd(a,m) = 1 ssi 1 = a.x + m.y
pii egcd(int a, int b) {
    if(b == 0) return make_pair(1, 0);
    else { pii RES = egcd(b, a%b);
        return make_pair(RES.second, RES.first - RES.second*(a/b));
    }
}
int inv(int n, int m) { pii EGCD = egcd(n, m);
    return ((EGCD.first % m) + m)%m;
}
// x ≡ aᵢ mod nᵢ para i = 1,...,k con gcd(nᵢ ,nⱼ ) = 1, ∀i ≠ j
// existe un único x mod N = n₁...nₖ
int crt(int n[], int a[], int k) {
    int i, tmp, MOD, RES; MOD = 1;
    for(i = 0; i < k; i++) MOD *= n[i];
    RES = 0;
    for(i = 0; i < k; i++) { tmp = MOD/n[i];
        tmp *= inv(tmp, n[i]); RES += (tmp*a[i]) % MOD;
    }
    return RES%MOD;
}

/* Expresiones Matemáticas */

// imput = "33*(244+675)*72"; pos = 0;
// respuesta -> expr();
string imput; int pos;
int sign(char s) { return (s == '+') ? 1 : -1; }
int term(); int factor();
int expr() {
    int res = term();
    while(imput[pos] == '+' || imput[pos] == '-'){
        int signo = sign(imput[pos++]); res += signo*expr();
    }
    return res;
}
int term() {
    int res = factor();
    while(imput[pos] == '*' || imput[pos] == '/'){
        if(imput[pos++] == '*') res *= term();
        else res /= term();
    }
    return res;
}
```

```
int factor() { int res;
    if(imput[pos] == '('){
        pos++; res = expr(); pos++;
    }
    else {
        char cad[10]; int p = 0;
        for(int i = pos; i < imput.size(); i++){
            if(!isdigit(imput[i])) break;
            else cad[p++] = imput[i];
            pos++;
        }
        sscanf(cad, "%d", &res);
    }
    return res;
}

/* Ordenar puntos CW */

struct pt { int x, y;
    pt(int xx = 0, int yy = 0){ x = xx, y = yy; };
};
bool operator<(const pt &p1, const pt &p2){
    double a1 = atan2(p1.y, p1.x), a2 = atan2(p2.y, p2.x);
    if(a1 != a2) return a1 < a2;
    else return p1.x*p1.x + p1.y*p1.y < p2.x*p2.x + p2.y*p2.y;
}

/* To Roman */

string fill( char c, int n ) { string s;
    while(n--) s += c;
    return s;
}
// Converts an integer in the range [1, 4000) to a lower case Roman numeral
string toRoman(int n) {
    if(n < 4) return fill('I', n);
    if(n < 6) return fill('I', 5 - n) + "V";
    if(n < 9) return string("V") + fill('I', n - 5);
    if(n < 11) return fill('I', 10 - n) + "X";
    if(n < 40) return fill('X', n/10) + toRoman(n % 10);
    if(n < 60) return fill('X', 5 - n/10) + 'L' + toRoman(n % 10);
    if(n < 90) return string("L") + fill('X', n/10 - 5) + toRoman(n % 10);
    if(n < 110) return fill('X', 10 - n/10) + "C" + toRoman(n % 10);
    if(n < 400) return fill('C', n/100) + toRoman(n % 100);
    if(n < 600) return fill('C', 5 - n/100) + 'D' + toRoman(n % 100);
    if(n < 900) return string("D") + fill('C', n/100 - 5) + toRoman(n % 100);
    if(n < 1100) return fill('C', 10 - n/100) + "M" + toRoman(n % 100);
    if(n < 4000) return fill('M', n/1000) + toRoman(n % 1000);
    return "?";
}
```

```
/* TSP Cilco */

// tsp(0, (i << N) - 1)
// donde i es la ciudad inicial y N es el numero de ciudades
// dist -> distancias para cada una de las ciudades
int dist[20][20];
int dpTsp[20][1<<20];

int numberOfTrailingZeros(int a){
    return __builtin_ctz((unsigned int)a);
}
int bitCount(int a){
    return __builtin_popcount((unsigned int)a);
}
static int tsp(int current, int mask) {
    if(dpTsp[current][mask] != 0)
        return dpTsp[current][mask];
    if(bitCount(mask) == 1) {
        return dpTsp[current][mask] = dist[0][numberOfTrailingZeros(mask)];
    }
    int maskT = mask;
    int j = 0;
    int best = MAX_VALUE;
    int nextMask = mask & (~(1 << current));
    while(maskT != 0) {
        if((maskT & 1) == 1 && j != current)
            best = min(best, dist[current][j] + tsp(j, nextMask));
        j++; maskT >>= 1;
    }
    return dpTsp[current][mask] = best;
}

/* Fenwick Tree */

struct Fenwick_Tree {
    vector<int> data;
    Fenwick_Tree(int N):data(N, 0){}
    inline int lobit(int x){
        return x & -x;
    }
    int query(int x){
        int sum = 0;
        for(; x >= 0; x -= lobit(x + 1))
            sum += data[x]; return sum;
    }
    void update(int x, int val){
        for(; x < data.size(); x += lobit(x + 1))
            data[x] += val;
    }
};
```