1 Welcome to Computer Programming

Before you start working on your assignment, here is a kind reminder. Please adhere to the following rules to avoid any potential penalties:

1. Minimize the Use of Generative AI (not mandatory, but its use may hinder your learning experience)
2. Avoid Plagiarism: While discussing ideas with classmates is encouraged, copying their code is not allowed.
3. Try Your Best but Not Too Much: You don't need to achieve a perfect score. Challenge yourself but understand that perfection isn't necessary.

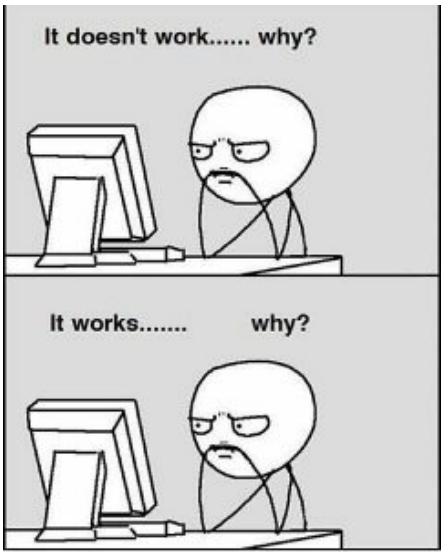
Please note that Problem 6 submitted will be checked for plagiarism. If your work is suspected of being copied, you may get zero point for this problem.

2 Grading

Each problem only if your submission achieves AC (Accepted), you can receive the full score, otherwise you will get 0 points. Your score will be compared against the following score chart:

Number of correct answers	0	1	2	3	4	5	6
Your assignment score	0	30	50	65	80	90	100

Table 1: Score Table



3 Problems

Problem 1. *Score*

There is an objective test result such as "OOXXOXXOOO". An 'O' means a correct answer of a problem and an 'X' means a wrong answer. The score of each problem of this test is calculated by itself and its just previous consecutive 'O's only when the answer is correct. For example, the score of the 10th problem is 3 that is obtained by itself and its two previous consecutive 'O's. Therefore, the score of "OOXXOXXOOO" is 10 which is calculated by "1+2+0+0+1+0+0+1+2+3". You are to write a program calculating the scores of test results.

Input Description

The input begins with an integer T ($1 \leq T \leq 1000$), representing the number of test cases. Each test case consists of:

- A single line containing a string of length L ($0 \leq L \leq 80$)
- The string contains only 'O' and 'X' characters
- No spaces between characters

Output Description

For each test case, output a single integer representing the total score according to the scoring rules.

Sample Input	Sample Output
5	10
OOXXOXXOOO	9
OOXXOOXXOO	7
OXOXOXOXOXOXOX	55
OOOOOOOOOO	30
OOOOXOOOOXOOOOX	

Table 2: Sample I/O

Reference Answer

```c
#include <stdio.h>
#include <string.h>

int main()
{
    int t, i, j, count, len, sum;
    char str[81];
    while (scanf("%d", &t) == 1)
    {
        for (i = 0; i < t; i++)
        {
            scanf("%s", str);
            len = strlen(str);
            count = 0;
            sum = 0;
            for (j = 0; j < len; j++)
            {
                if (str[j] == 'O')
                    count++;
                else
                    count = 0;
                sum = sum + (count * 1);
            }
            printf("%d\n", sum);
        }
        return 0;
    }
}
```

Problem 2. *Sort Array*

Given an array of integers nums, sort the array in ascending order and return it. I highly recommend implementing quick sort or merge sort, otherwise, you might get a TLE (Time Limit Exceeded).

Input Description

The input consists of an array of integers nums where:

- $1 \leq nums.length <= 5 * 10^6$
- $-5 \times 10^7 \leq nums[i] \leq 5 * 10^7$

Output Description

Print sorted array.

Sample Input	Sample Output
4 5 2 3 1	1 2 3 5
6 5 1 1 2 0 0	0 0 1 1 2 5

Table 3: Sample I/O

Example 1:

- Input: nums = [5,2,3,1]
- Output: [1,2,3,5]
- Explanation: After sorting the array, the positions of some numbers are not changed (for example, 2 and 3), while the positions of other numbers are changed (for example, 1 and 5).

Example 2:

- Input: nums = [5,1,1,2,0,0]
- Output: [0,0,1,1,2,5]
- Explanation: Note that the values of nums are not necessarily unique.

Reference Answer 1 - Quick Sort

```c
#include <stdio.h>
#include <stdlib.h>

int part(int nums[], int low, int high) {
    int random = low + rand() % (high - low + 1);
    int temp = nums[random];
    nums[random] = nums[high];
    nums[high] = temp;
    int base = nums[high];
    int i = low - 1;
    for (int j = low; j < high; j++) {
        if (nums[j] < base) {
            i++;
            temp = nums[i];
            nums[i] = nums[j];
            nums[j] = temp;
        }
    }
    temp = nums[i + 1];
    nums[i + 1] = nums[high];
    nums[high] = temp;
    return i + 1;
}
void quicksort(int nums[], int low, int high) {
    while (low < high) {
        int p = part(nums, low, high);
        if (p - low < high - p) {
            quicksort(nums, low, p - 1);
            low = p + 1;
        } else {
            quicksort(nums, p + 1, high);
            high = p - 1;
        }
    }
}
int main(void) {
    int N;
    scanf("%d", &N);
    int *nums = (int *)malloc(N * sizeof(int));
    for (int n = 0; n < N; n++) {
        scanf("%d", &nums[n]);
    }
    quicksort(nums, 0, N - 1);
    for (int n = 0; n < N; n++) {
        if (n > 0) printf(" ");
        printf("%d", nums[n]);
    }
    free(nums);
    return 0;
}
```

Reference Answer 2 - Merge Sort

```c
#include <stdio.h>
#include <stdlib.h>

#define MAX_SIZE 5000005

void merge(int arr[], int left, int mid, int right)
{
    int leftSize = mid - left + 1;
    int rightSize = right - mid;

    // Create temporary arrays
    int leftArr[leftSize], rightArr[rightSize];

    // Copy data to temporary arrays
    for (int i = 0; i < leftSize; i++)
        leftArr[i] = arr[left + i];
    for (int i = 0; i < rightSize; i++)
        rightArr[i] = arr[mid + 1 + i];

    // Merge the temporary arrays back into arr
    int i = 0, j = 0, k = left;
    while (i < leftSize && j < rightSize)
    {
        if (leftArr[i] <= rightArr[j])
        {
            arr[k++] = leftArr[i++];
        }
        else
        {
            arr[k++] = rightArr[j++];
        }
    }

    // Copy remaining elements
    while (i < leftSize)
        arr[k++] = leftArr[i++];
    while (j < rightSize)
        arr[k++] = rightArr[j++];
}

void mergeSort(int arr[], int left, int right)
{
    if (left < right)
    {
        int mid = left + (right - left) / 2;
        mergeSort(arr, left, mid);
        mergeSort(arr, mid + 1, right);
        merge(arr, left, mid, right);
    }
}

int arr[MAX_SIZE];
```

```
53
54 int main()
55 {
56     int size;
57     scanf("%d", &size);
58
59     // Input array elements
60     for (int i = 0; i < size; i++)
61         scanf("%d", &arr[i]);
62
63     // Sort array
64     mergeSort(arr, 0, size - 1);
65
66     // Print sorted array
67     for (int i = 0; i < size; i++)
68         printf("%d ", arr[i]);
69
70     return 0;
71 }
```

Problem 3. *Merge Two Sorted Lists Using Pointers*

Given two sorted linked lists, write a function that merges them into a single sorted linked list using only pointer manipulation. The merged list should be created by manipulating the pointers of the original lists without creating new nodes. I highly recommend implementing it with pointers, otherwise, you might get an MLE (Memory Limit Exceeded).

Input Description

The input consists of two lines:

- First line contains n_1 ($1 \leq n_1 \leq 10^6$) followed by n_1 integers representing the first sorted linked list
- Second line contains n_2 ($1 \leq n_2 \leq 10^6$) followed by n_2 integers representing the second sorted linked list

All integers in the lists are in the range $[-10^5, 10^5]$ and are given in non-decreasing order.

Output Description

Output a single line containing $(n_1 + n_2)$ integers representing the merged sorted list.

Sample Input	Sample Output
4 1 3 5 7 4 2 4 6 8	1 2 3 4 5 6 7 8
3 1 2 3 3 1 2 3	1 1 2 2 3 3
2 -1 5 2 0 7	-1 0 5 7

Table 4: Sample I/O

Reference Answer

```c
#include <stdio.h>
#include <stdlib.h>

int arr1[1000000 - 5];
int arr2[1000000 - 5];

int main()
{
    int n, k;
    scanf("%d", &n);
    for (int i = 0; i < n; i++)
        scanf("%d", &arr1[i]);
    scanf("%d", &k);
    for (int i = 0; i < k; i++)
        scanf("%d", &arr2[i]);
    int i = 0, j = 0;
    while (i < n && j < k)
    {
        if (arr1[i] < arr2[j])
            printf("%d ", arr1[i++]);
        else
            printf("%d ", arr2[j++]);
    }
    while (i < n)
        printf("%d ", arr1[i++]);
    while (j < k)
        printf("%d ", arr2[j++]);
    printf("\n");
    return 0;
}
```

Problem 4. *Robot Motion Planning*

The world of robotics intersects various Computer Science disciplines, including artificial intelligence, algorithms, and engineering. This problem focuses on robot navigation in a bounded rectangular grid world.

Your task is to track a robot's position as it explores a pre-Columbian flat world. The robot moves according to specific commands while avoiding locations where previous robots have been lost.

A robot position consists of a grid coordinate (a pair of integers: x-coordinate followed by y- coordinate) and an orientation (N,S,E,W for north, south, east, and west). A robot instruction is a string of the letters 'L', 'R', and 'F' which represent, respectively, the instructions:

- Left: the robot turns left 90 degrees and remains on the current grid point.
- Right: the robot turns right 90 degrees and remains on the current grid point.
- Forward: the robot moves forward one grid point in the direction of the current orientation and mantains the same orientation.

The direction North corresponds to the direction from grid point (x, y) to grid point (x, y + 1). Since the grid is rectangular and bounded, a robot that moves "off" an edge of the grid is lost forever. However, lost robots leave a robot "scent" that prohibits future robots from dropping off the world at the same grid point. The scent is left at the last grid position the robot occupied before disappearing over the edge. An instruction to move "off" the world from a grid point from which a robot has been previously lost is simply ignored by the current robot.

Figure 1: Cry! Stanley

Input Description

The first line of input is the upper-right coordinates of the rectangular world, the lower-left coordinates are assumed to be 0,0. The remaining input consists of a sequence of robot positions and instructions (two lines per robot). A position consists of two integers specifying the initial coordinates of the robot and an orientation (N,S,E,W), all separated by white space on one line. A robot instruction is a string of the letters 'L', 'R', and 'F' on one line. Each robot is processed sequentially, i.e., finishes executing the robot instructions before the next robot begins execution.

You may assume that all initial robot positions are within the bounds of the specified grid. The maximum value for any coordinate is 50. All instruction strings will be less than 100 characters in length.

Output Description

For each robot position/instruction in the input, the output should indicate the final grid position and orientation of the robot. If a robot falls off the edge of the grid the word 'LOST' should be printed after the position and orientation.

Sample Input	**Sample Output**
5 3 1 1 E RFRFRFRF 3 2 N FRRFLLFFRRFLL 0 3 W LLFFFLFLFL	1 1 E 3 3 N LOST 2 3 S

Table 5: Sample I/O

Reference Answer

```c
#include <stdio.h>
#include <stdlib.h>

int main()
{
    int n, m, sx, sy;
    char D[2], cmd[1000], pre[100][100] = {};
    scanf("%d %d", &n, &m);
    while (scanf("%d %d %s", &sx, &sy, D) == 3)
    {
        scanf("%s", cmd);
        int d = D[0], flag = 0, i;
        for (i = 0; cmd[i]; i++)
        {
            if (cmd[i] == 'F')
            {
                switch (d)
                {
                case 'N':
                    sy++;
                    break;
                case 'E':
                    sx++;
                    break;
                case 'W':
                    sx--;
                    break;
                case 'S':
                    sy--;
                    break;
                }
            }
            else if (cmd[i] == 'R')
            {
                switch (d)
                {
                case 'N':
                    d = 'E';
                    break;
                case 'E':
                    d = 'S';
                    break;
                case 'W':
                    d = 'N';
                    break;
                case 'S':
                    d = 'W';
                    break;
                }
            }
            else
            {
```

```
                    switch (d)
                    {
                    case 'N':
                        d = 'W';
                        break;
                    case 'E':
                        d = 'N';
                        break;
                    case 'W':
                        d = 'S';
                        break;
                    case 'S':
                        d = 'E';
                        break;
                    }
                }
                if (sx < 0 || sy < 0 || sx > n || sy > m)
                {
                    switch (d)
                    {
                    case 'N':
                        sy--;
                        break;
                    case 'E':
                        sx--;
                        break;
                    case 'W':
                        sx++;
                        break;
                    case 'S':
                        sy++;
                        break;
                    }
                    if (pre[sx][sy] == 1)
                        continue;
                    flag = 1;
                    pre[sx][sy] = 1;
                    break;
                }
            }
            if (!flag)
                printf("%d %d %c\n", sx, sy, d);
            else
            {
                printf("%d %d %c LOST\n", sx, sy, d);
            }
        }
        return 0;
}
```

Problem 5. *Tower of Hanoi Recursive Solution*

Given three pegs (labeled as 1, 2, and 3) and n disks of different sizes initially stacked in ascending order on peg 1, write a program to print the sequence of moves to transfer all disks from peg 1 to peg 3.

Some Rules :

- Only one disk can be moved at a time
- A disk can only be placed on top of a larger disk or an empty peg
- Each move should be printed in the format: Move disk "disk number" from peg "source" to peg "destination"

Input Description

A single integer n ($1 \leq n \leq 10$) representing the number of disks.

Output Description

Print the sequence of moves required to solve the Tower of Hanoi puzzle. Each move should be printed on a new line.

Sample Input	Sample Output
3	Move disk 1 from peg 1 to peg 3 Move disk 2 from peg 1 to peg 2 Move disk 1 from peg 3 to peg 2 Move disk 3 from peg 1 to peg 3 Move disk 1 from peg 2 to peg 1 Move disk 2 from peg 2 to peg 3 Move disk 1 from peg 1 to peg 3

Table 6: Sample I/O

Note:

The solution should be implemented using recursion. For N disks, the program will make $2^N - 1$ moves.

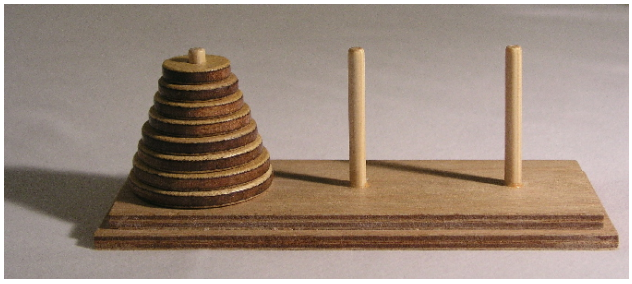


Figure 2: Tower of Hanoi

Reference Answer

```c
#include <stdio.h>

void solveHanoiTower(int numDisks, int sourcePeg, int auxiliaryPeg, int targetPeg)
{
    if (numDisks == 1)
    {
        printf("Move disk %d from peg %d to peg %d\n", numDisks, sourcePeg, targetPeg);
        return;
    }

    solveHanoiTower(numDisks - 1, sourcePeg, targetPeg, auxiliaryPeg);
    printf("Move disk %d from peg %d to peg %d\n", numDisks, sourcePeg, targetPeg);
    solveHanoiTower(numDisks - 1, auxiliaryPeg, sourcePeg, targetPeg);
}

int main()
{
    int numDisks;
    scanf("%d", &numDisks);
    solveHanoiTower(numDisks, 1, 2, 3);
    return 0;
}
```

Problem 6. *Drop out periodic function(D.O.P.F)*

The Drop Out Periodic Function (D.O.P.F) is a renowned mathematical function that elegantly describes the relationship between a college student's changing mindset and the duration of their university studies. This function was invented by ***Li, Wei-Cheng****, a senior student in the Department of Learning Science at National Taiwan Normal University. Thanks to his contribution, we can now fully comprehend this function.*

However, what if we were unaware of the function's exact form and only knew its inputs and outputs? Would it be possible to reconstruct any arbitrary function under these circumstances? For instance, if you were given several pairs of (x_i, y_i) *where* $f(x_i) = y_i$*, could you recreate the D.O.P.F function?*

We'll assume the function is an nth-degree polynomial, and you'll be provided with n+1 valid points. Your program should implement the Lagrange Interpolation Method to solve this problem, reconstructing the original function based on the given data points.

Input Description

The input data consists of $N + 1$ lines. N represents the number of subsequent data lines (where $2 \leq N \leq 10$), and it also indicates that the polynomial being simulated is of degree $N-1$. The following N lines represent N data points, each containing two integers separated by a space. The first integer represents X_i (where $0 < X_i < 10^6$), and the second integer represents Y_i (where $0 < Y_i < 10^6$). The function that guarantees the answer will be a polynomial with all integer coefficients.

Sample Input	Sample Output
2 1 3 2 5 3 7	f(x) = 2x + 1
2 10 -20 20 -40 30 -60	f(x) = -2x

Table 7: Sample I/O

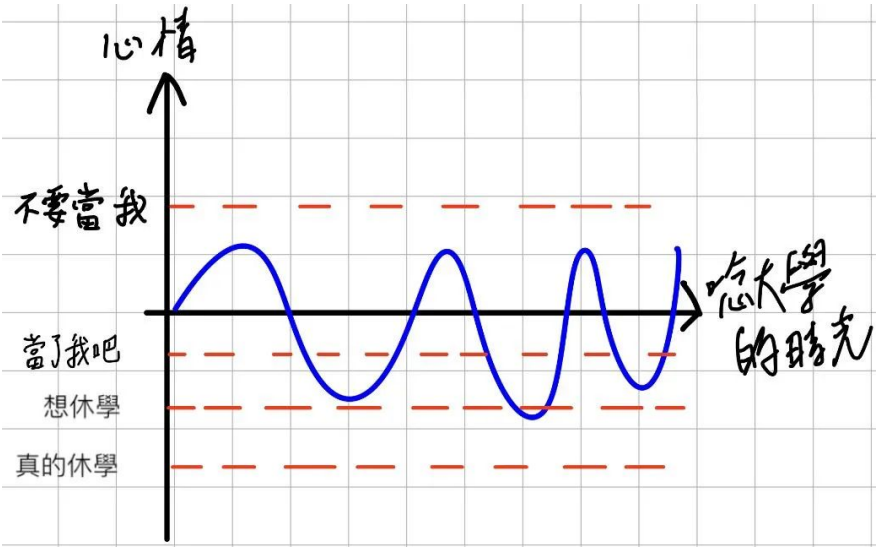


Figure 3: Drop out periodic function (This image is from Li,Wei-Cheng 's Thread)

Reference Answer 1 (Cite to : 41375001H)

```c
#include <math.h>
#include <stdio.h>

#define MAX_DEGREE 100

// Function prototypes
void readInputPoints(int N, double points[][2]);
void constructMatrix(int N, double points[][2], double M1[][2 * MAX_DEGREE + 2], double M2[]);
void findInverseMatrix(int N, double M1[][2 * MAX_DEGREE + 2]);
void calculateCoefficients(int N, double M1[][2 * MAX_DEGREE + 2], double M2[], int coeff[][1]);
void printPolynomial(int N, int coeff[][1]);
void printTerm(int power, int coefficient, int *isFirstTerm);

int main(void) {
    int N;
    scanf("%d", &N);

    // Declare arrays with fixed size
    double points[MAX_DEGREE + 1][2];
    double M1[MAX_DEGREE + 1][2 * MAX_DEGREE + 2];
    double M2[MAX_DEGREE + 1];
    int coeff[MAX_DEGREE + 1][1];

    // Read input points
    readInputPoints(N, points);

    // Process the polynomial interpolation
    constructMatrix(N, points, M1, M2);
    findInverseMatrix(N, M1);
    calculateCoefficients(N, M1, M2, coeff);

    // Print the resulting polynomial
    printPolynomial(N, coeff);

    return 0;
}

void readInputPoints(int N, double points[][2]) {
    for (int i = 0; i <= N; i++) {
        scanf("%lf%lf", &points[i][0], &points[i][1]);
    }
}

void constructMatrix(int N, double points[][2], double M1[][2 * MAX_DEGREE + 2], double M2[]) {
    for (int i = 0; i <= N; i++) {
        M1[i][0] = 1;
        M1[i][N + 1] = 0;

        // Fill the matrix row
```

```
        for (int j = 1; j <= N; j++) {
            M1[i][j] = M1[i][j - 1] * points[i][0];
            M1[i][N + 1 + j] = 0;
        }

        M1[i][N + 1 + i] = 1;
        M2[i] = points[i][1];
    }
}

void findInverseMatrix(int N, double M1[][2 * MAX_DEGREE + 2]) {
    for (int i = 0; i <= N; i++) {
        // Normalize current row
        double factor = M1[i][i];
        for (int k = i; k <= 2 * N + 1; k++) {
            M1[i][k] /= factor;
        }

        // Eliminate in other rows
        for (int j = 0; j <= N; j++) {
            if (j != i) {
                double c = M1[j][i];
                for (int l = i; l <= 2 * N + 1; l++) {
                    M1[j][l] -= c * M1[i][l];
                }
            }
        }
    }
}

void calculateCoefficients(int N, double M1[][2 * MAX_DEGREE + 2], double
    M2[], int coeff[][1]) {
    // Matrix multiplication to find coefficients
    for (int i = 0; i <= N; i++) {
        double temp = 0;
        for (int j = 0; j <= N; j++) {
            temp += M1[i][N + 1 + j] * M2[j];
        }
        coeff[i][0] = (int)round(temp);
    }
}

void printTerm(int power, int coefficient, int *isFirstTerm) {
    if (coefficient == 0) return;

    char sign = (*isFirstTerm) ? (coefficient < 0 ? '-' : '\0') :
                                 (coefficient < 0 ? " - " : " + ");
    int absCoeff = abs(coefficient);

    if (*isFirstTerm) {
        *isFirstTerm = 0;
    }

    // Print coefficient
```

```
    if (power > 0) {
        if (absCoeff == 1) {
            printf("%s", sign ? &sign : "");
        } else {
            printf("%s%d", sign ? &sign : "", absCoeff);
        }
    } else {
        printf("%s%d", sign ? &sign : "", absCoeff);
    }

    // Print variable and exponent
    if (power > 1) {
        printf("x^%d", power);
    } else if (power == 1) {
        printf("x");
    }
}

void printPolynomial(int N, int coeff[][1]) {
    printf("f(x) = ");
    int isFirstTerm = 1;

    // Print terms from highest to lowest degree
    for (int i = N; i >= 0; i--) {
        printTerm(i, coeff[i][0], &isFirstTerm);
    }
    printf("\n");
}
```

Reference Answer 2 (Cite to : 41375007H)

```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <math.h>

#define EPSILON 1e-6

typedef long long ll;

// Function declarations
void findCoefficients(ll* coeff, ll* x, int k, int degree);
int isZero(double val);
void printTerm(ll num, int power);
void printPolynomial(double* result, int degree);

// Main function to perform polynomial interpolation
void interpolatePolynomial(ll* x, ll* y, int degree, double* result) {
    ll* coeff = malloc((degree + 1) * sizeof(ll));

    for (int i = 0; i <= degree; i++) {
        memset(coeff, 0, (degree + 1) * sizeof(ll));
        coeff[0] = 1;
        findCoefficients(coeff, x, i, degree);

        ll ratio = 1;
        for (int j = 0; j <= degree; j++) {
            if (i == j) continue;
            ratio *= (x[i] - x[j]);
        }

        for (int k = 0; k <= degree; k++) {
            result[k] += (double)y[i] * (double)coeff[k] / (double)ratio;
        }
    }

    free(coeff);
}

void findCoefficients(ll* coeff, ll* x, int k, int degree) {
    for (int i = 0; i <= degree; i++) {
        if (i == k) continue;
        for (int j = i; j >= 0; j--) {
            coeff[j + 1] = coeff[j] + coeff[j + 1];
            coeff[j] = coeff[j] * -x[i];
        }
    }
}

int isZero(double val) {
    return !(fabs(val - round(val)) < EPSILON && fabs(val) > EPSILON);
}

```

```
void printTerm(ll num, int power) {
    if (power == 1) {
        printf(num == 1 ? "x" : "%lldx", num);
    } else if (power == 0) {
        printf("%lld", num);
    } else {
        printf(num == 1 ? "x^%d" : "%lldx^%d", num, power);
    }
}

void printPolynomial(double* result, int degree) {
    printf("f(x) = ");
    int first = 1;
    int hasTerms = 0;

    for (int i = degree; i >= 0; i--) {
        if (isZero(result[i])) continue;

        hasTerms = 1;
        ll coefficient = (ll)round(result[i]);

        if (first) {
            first = 0;
            printTerm(coefficient, i);
        } else {
            if (coefficient > 0) {
                printf(" + ");
                printTerm(coefficient, i);
            } else {
                printf(" - ");
                printTerm(-coefficient, i);
            }
        }
    }

    if (!hasTerms) {
        printf("0");
    }
}

int main() {
    int degree;
    scanf("%d", &degree);

    ll* x = malloc((degree + 1) * sizeof(ll));
    ll* y = malloc((degree + 1) * sizeof(ll));
    double* result = calloc(degree + 1, sizeof(double));

    // Read input points
    for (int i = 0; i <= degree; i++) {
        scanf("%lld %lld", &x[i], &y[i]);
    }

    // Perform interpolation
```

```
    interpolatePolynomial(x, y, degree, result);

    // Print result
    printPolynomial(result, degree);

    // Free allocated memory
    free(x);
    free(y);
    free(result);

    return 0;
}
```