Istituzioni di Algebra e Geometria

Andrea Agostini 1996124
Simone Bonanni 1992505
Giacomo Gneri 2025964
Marta Graziano 2024185
Elena Sbordoni 2000180

Febbraio 2025

Indice

1 Geometria: analisi dati

In questa parte del corso si mira a studiare la distribuzione di una grande quantità di dati per ricavarne importanti informazioni (per esempio quale opzione tra tante è ritenuta la migliore, come evolve una malattia in seguito alla somministrazione di un medicinale etc.).
Di seguito sono riportati i principali concetti matematici astratti e codici che serviranno per la creazione finale del **barcode**, uno strumento in grado di rappresentare graficamente l'andamento temporale di alcune caratteristiche topologiche dei dati presi in considerazione.

1.1 Complessi simpliciali

Sia X un generico insieme (per esempio una moltitudine di dati) ed $S \subset P(X)$, dove $P(X)$ rappresenta l'insieme delle parti di X.

Definizioni : La coppia (X, S) si dice **complesso simpliciale** se valgono le seguenti proprietà:

1. $\tau \in S \;\Rightarrow\; \sigma \in S \;\; \forall \sigma \subset \tau$
2. $\tau \in P(X)$, $|\tau| = 1 \;\Rightarrow\; \tau \in S$

Gli elementi di X sono chiamati **vertici** e, più in generale, gli elementi di S sono chiamati **simplessi**. Un simplesso τ si dice **massimale** se

$$\tau \subset \sigma \;\Rightarrow\; \tau = \sigma \; .$$

Dall'insieme di simplessi massimali è possibile ricostruire l'insieme complesso simpliciale e viceversa, come mostrano questi codici.

Listing 1: AllToMax.c

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

// Definizione della struttura "simplex"
typedef struct simplex {
    int* vertices;           // Array dinamico contenente i vertici (ordinato)
    int position;            // Numero di vertici (dimensione + 1)
    struct simplex* next;    // Puntatore al simplesso successivo nella lista concatenata
} simplex;

// Definizione della struttura "SimplicialComplex"
typedef struct {
    simplex* simplices;      // Puntatore alla lista concatenata di simplessi
    int size;                // Numero di simplessi presenti
} SimplicialComplex;

// Funzione di confronto per qsort (ordinamento crescente)
int cmpInt(const void *a, const void *b) {
    int int_a = *(const int *)a;
    int int_b = *(const int *)b;
    return int_a - int_b;
}

// Funzione che verifica se il simplesso "sub" è un sottoinsieme del simplesso "sup"
// Entrambi gli array di vertici devono essere ordinati in ordine crescente.
bool isSubset(simplex* sub, simplex* sup) {
    int i = 0, j = 0;
    while (i < sub->position && j < sup->position) {
        if (sub->vertices[i] == sup->vertices[j]) {
            i++;
```

```c
            j++;
        } else if (sub->vertices[i] > sup->vertices[j]) {
            j++;
        } else { // sub->vertices[i] < sup->vertices[j]
            return false;
        }
    }
    return (i == sub->position);
}

// Funzione per aggiungere un simplesso alla struttura SimplicialComplex
void addSimplex(SimplicialComplex* complex, int* vertices, int numVertices) {
    simplex* newNode = (simplex*)malloc(sizeof(simplex));
    if(newNode == NULL) {
        fprintf(stderr, "Errore di allocazione della memoria.\n");
        exit(1);
    }
    newNode->vertices = vertices;
    newNode->position = numVertices;
    newNode->next = complex->simplices;
    complex->simplices = newNode;
    complex->size++;
}

int main(void) {
    int k;
    printf("Inserisci il numero massimo di dimensioni (k): ");
    if (scanf("%d", &k) != 1) {
        fprintf(stderr, "Input non valido.\n");
        return 1;
    }

    // Allocazione dinamica di un array di SimplicialComplex per dimensioni da 0 a k.
    SimplicialComplex* complexes = (SimplicialComplex*)malloc((k + 1) * sizeof(SimplicialComplex));
    if (complexes == NULL) {
        fprintf(stderr, "Errore di allocazione della memoria.\n");
        return 1;
    }

    int i, dim, s, j, supDim;
    // Inizializzazione dei complessi
    for (i = 0; i <= k; i++) {
        complexes[i].simplices = NULL;
        complexes[i].size = 0;
    }

    // Costruzione automatica degli 0-simplessi
    int numZeroSimplices;
    printf("Inserisci il numero di 0-simplessi: ");
    if (scanf("%d", &numZeroSimplices) != 1) {
        fprintf(stderr, "Input non valido.\n");
        free(complexes);
        return 1;
    }
    for (i = 0; i < numZeroSimplices; i++) {
        int* vertex = (int*)malloc(sizeof(int));
        if (vertex == NULL) {
            fprintf(stderr, "Errore di allocazione della memoria.\n");
            exit(1);
        }
        vertex[0] = i + 1;  // Gli 0-simplessi sono rappresentati da un singolo vertice
        addSimplex(&complexes[0], vertex, 1);
    }

    // Input e costruzione dei simplessi per dimensioni maggiori di 0.
    // Per ogni dimensione dim (da 1 a k), ogni simplesso ha (dim + 1) elementi.
    for (dim = 1; dim <= k; dim++) {
```

```
        int numSimplices;
        printf("Inserisci il numero di %d-simplessi: ", dim);
        if (scanf("%d", &numSimplices) != 1) {
            fprintf(stderr, "Input non valido.\n");
            exit(1);
        }
        for (s = 0; s < numSimplices; s++) {
            int m = dim + 1;
            int* vertices = (int*)malloc(m * sizeof(int));
            if (vertices == NULL) {
                fprintf(stderr, "Errore di allocazione della memoria.\n");
                exit(1);
            }
            printf("Inserisci gli elementi del %d-simplesso %d: ", dim, s + 1);
            for (j = 0; j < m; j++) {
                if (scanf("%d", &vertices[j]) != 1) {
                    fprintf(stderr, "Input non valido.\n");
                    exit(1);
                }
            }
            // Ordinamento degli elementi per il corretto funzionamento di isSubset
            qsort(vertices, m, sizeof(int), cmpInt);
            addSimplex(&complexes[dim], vertices, m);
        }
    }

    // Processo per trovare i simplessi massimali.
    // Per ogni simplesso di dimensione inferiore a k, si verifica se è contenuto in un simplesso di dimensione superiore.
    for (dim = k - 1; dim >= 0; dim--) {
        simplex* newList = NULL;
        int newSize = 0;
        simplex* current = complexes[dim].simplices;
        while (current != NULL) {
            bool isMaximal = true;
            for (supDim = dim + 1; supDim <= k && isMaximal; supDim++) {
                simplex* candidate = complexes[supDim].simplices;
                while (candidate != NULL && isMaximal) {
                    if (isSubset(current, candidate)) {
                        isMaximal = false;
                    }
                    candidate = candidate->next;
                }
            }
            simplex* nextNode = current->next;
            if (isMaximal) {
                current->next = newList;
                newList = current;
                newSize++;
            } else {
                free(current->vertices);
                free(current);
            }
            current = nextNode;
        }
        complexes[dim].simplices = newList;
        complexes[dim].size = newSize;
    }

    // Output dei complessi massimali
    printf("I complessi massimali sono:\n");
    for (dim = 0; dim <= k; dim++) {
        if (complexes[dim].simplices != NULL) {
            printf("%d-simplessi massimali:\n", dim);
            simplex* cur = complexes[dim].simplices;
            while (cur != NULL) {
                printf("{");
                for (j = 0; j < cur->position; j++) {
                    if (j > 0) {
                        printf(", ");
```

```c
                    }
                    printf("%d", cur->vertices[j]);
                }
                printf("}\n");
                cur = cur->next;
            }
        }
    }

    // Deallocazione della memoria utilizzata
    for (dim = 0; dim <= k; dim++) {
        simplex* cur = complexes[dim].simplices;
        while (cur != NULL) {
            simplex* next = cur->next;
            free(cur->vertices);
            free(cur);
            cur = next;
        }
    }
    free(complexes);

    return 0;
}
```

Listing 2: MaxToAll.c

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

typedef struct simplex {
    int* vertices;              // Array dinamico dei vertici (già ordinato)
    int position;               // Numero di vertici (dimensione + 1)
    struct simplex* next;       // Puntatore al simplesso successivo (lista
        concatenata)
} simplex;

typedef struct {
    simplex* simplices;         // Puntatore alla lista concatenata dei simplessi
    int size;                   // Numero di simplessi presenti
} SimplicialComplex;

// Funzione di confronto per qsort (ordinamento crescente)
int cmpInt(const void *a, const void *b) {
    int int_a = *(const int*)a;
    int int_b = *(const int*)b;
    return int_a - int_b;
}

// Aggiunge un simplesso alla struttura SimplicialComplex
void addSimplex(SimplicialComplex* complex, int* vertices, int numVertices) {
    simplex* newNode = (simplex*)malloc(sizeof(simplex));
    if(newNode == NULL) {
        fprintf(stderr, "Errore di allocazione della memoria.\n");
        exit(1);
    }
    newNode->vertices = vertices;
    newNode->position = numVertices;
    newNode->next = complex->simplices;
    complex->simplices = newNode;
    complex->size++;
}

// Confronta due simplessi (si assume che abbiano lo stesso numero di vertici)
int compareSimplex(const simplex* a, const simplex* b) {
    int i;
    if(a->position != b->position)
```

```c
        return a->position - b->position;
    for(i = 0; i < a->position; i++) {
        if(a->vertices[i] != b->vertices[i])
            return a->vertices[i] - b->vertices[i];
    }
    return 0;
}

// Rimuove i duplicati all'interno di un SimplicialComplex (lista concatenata)
void removeDuplicates(SimplicialComplex* complex) {
    simplex *current, *runner, *prevRunner, *temp;
    current = complex->simplices;
    while(current != NULL) {
        prevRunner = current;
        runner = current->next;
        while(runner != NULL) {
            if(compareSimplex(current, runner) == 0) {
                prevRunner->next = runner->next;
                temp = runner;
                runner = runner->next;
                free(temp->vertices);
                free(temp);
                complex->size--;
            } else {
                prevRunner = runner;
                runner = runner->next;
            }
        }
        current = current->next;
    }
}

// Genera tutti i sottoinsiemi non vuoti di un simplesso massimale e li aggiunge
// alla struttura dei simplessi "allComplexes" in base alla loro dimensione.
void generateSubsetsForSimplex(const simplex* maxSimplex, SimplicialComplex*
    allComplexes) {
    int n = maxSimplex->position;
    int total = 1 << n; // 2^n
    int i, j;
    for(i = 1; i < total; i++) {
        int count = 0;
        for(j = 0; j < n; j++) {
            if(i & (1 << j))
                count++;
        }
        int* subset = (int*)malloc(count * sizeof(int));
        if(subset == NULL) {
            fprintf(stderr, "Errore di allocazione della memoria.\n");
            exit(1);
        }
        int index = 0;
        for(j = 0; j < n; j++) {
            if(i & (1 << j)) {
                subset[index] = maxSimplex->vertices[j];
                index++;
            }
        }
        // La dimensione del simplesso è count-1 (numero di vertici - 1)
        int subsetDim = count - 1;
        addSimplex(&allComplexes[subsetDim], subset, count);
    }
}

// Libera tutta la memoria allocata per un SimplicialComplex
void freeSimplicialComplex(SimplicialComplex* complex) {
    simplex* cur = complex->simplices;
    simplex* next;
    while(cur != NULL) {
        next = cur->next;
        free(cur->vertices);
```

```
        free(cur);
        cur = next;
    }
    complex->simplices = NULL;
    complex->size = 0;
}

int main(void) {
    int k;
    printf("Inserisci il numero massimo di dimensioni (k): ");
    if(scanf("%d", &k) != 1) {
        fprintf(stderr, "Input non valido.\n");
        return 1;
    }

    // Allocazione dinamica di due array di SimplicialComplex (uno per i simplessi
        massimali e uno per tutti quelli generati)
    SimplicialComplex* maxSimplicialComplexes = (SimplicialComplex*)malloc((k + 1)
        * sizeof(SimplicialComplex));
    SimplicialComplex* allSimplicialComplexes = (SimplicialComplex*)malloc((k + 1)
        * sizeof(SimplicialComplex));
    if(maxSimplicialComplexes == NULL || allSimplicialComplexes == NULL) {
        fprintf(stderr, "Errore di allocazione della memoria.\n");
        exit(1);
    }

    int dim, i, j;
    // Inizializzazione delle strutture per ogni dimensione
    for(dim = 0; dim <= k; dim++) {
        maxSimplicialComplexes[dim].simplices = NULL;
        maxSimplicialComplexes[dim].size = 0;
        allSimplicialComplexes[dim].simplices = NULL;
        allSimplicialComplexes[dim].size = 0;
    }

    // Input dei simplessi massimali per ogni dimensione
    for(dim = 0; dim <= k; dim++) {
        int numMax;
        printf("Inserisci il numero di %d-simplessi massimali: ", dim);
        if(scanf("%d", &numMax) != 1) {
            fprintf(stderr, "Input non valido.\n");
            exit(1);
        }
        for(i = 0; i < numMax; i++) {
            int m = dim + 1;
            int* vertices = (int*)malloc(m * sizeof(int));
            if(vertices == NULL) {
                fprintf(stderr, "Errore di allocazione della memoria.\n");
                exit(1);
            }
            printf("Inserisci gli elementi del %d-simplesso massimale %d: ", dim, i
             + 1);
            for(j = 0; j < m; j++) {
                if(scanf("%d", &vertices[j]) != 1) {
                    fprintf(stderr, "Input non valido.\n");
                    exit(1);
                }
            }
            qsort(vertices, m, sizeof(int), cmpInt);
            addSimplex(&maxSimplicialComplexes[dim], vertices, m);
        }
    }

    // Per ogni simplesso massimale, genera tutti i suoi sottoinsiemi
    simplex* current;
    for(dim = 0; dim <= k; dim++) {
        current = maxSimplicialComplexes[dim].simplices;
        while(current != NULL) {
            generateSubsetsForSimplex(current, allSimplicialComplexes);
            current = current->next;
```

```
        }
    }

    // Rimuove eventuali duplicati per ciascuna dimensione
    for(dim = 0; dim <= k; dim++) {
        removeDuplicates(&allSimplicialComplexes[dim]);
    }

    // Output di tutti i simplessi generati per ogni dimensione
    printf("Tutti i simplessi generati sono:\n");
    for(dim = 0; dim <= k; dim++) {
        if(allSimplicialComplexes[dim].simplices != NULL) {
            printf("%d-simplessi generati:\n", dim);
            current = allSimplicialComplexes[dim].simplices;
            while(current != NULL) {
                printf("{");
                for(i = 0; i < current->position; i++) {
                    if(i > 0)
                        printf(", ");
                    printf("%d", current->vertices[i]);
                }
                printf("}\n");
                current = current->next;
            }
        }
    }

    // Deallocazione della memoria utilizzata
    for(dim = 0; dim <= k; dim++) {
        freeSimplicialComplex(&maxSimplicialComplexes[dim]);
        freeSimplicialComplex(&allSimplicialComplexes[dim]);
    }
    free(maxSimplicialComplexes);
    free(allSimplicialComplexes);

    return 0;
}
```

Un altro modo per creare un complesso simpliciale, molto utile per gli scopi del corso, è tramite una matrice quadrata booleana, chiamata matrice di adiacenza, di dimensioni pari al numero di vertici in cui l'elemento x_{ij} indica se il vertice i ed il vertice j sono "vicini" o meno. Partendo infatti dai vertici e dalla matrice si può ricostruire l'intero complesso simpliciale tramite una strategia adottata in questo codice.

Listing 3: Complesso da 1 simplessi.h

```
#include "Compl_Simpl.h"

SimplicialComplex* complex_from_adjacency_matrix_complete (int**, int);
SimplicialComplex* complex_from_adjacency_matrix_truncated (int**, int, int);
SimplicialComplex* complex_from_1_simplices_complete (SimplicialComplex*, int);
SimplicialComplex* complex_from_1_simplices_truncated (SimplicialComplex*, int, int
    );
void add_k_simplices_adjacency_matrix (SimplicialComplex*, int**, int, int);

int check_neighbor_adjacency_matrix (int**, int*, int, int);
int check_neighbor (SimplicialComplex*, int, int);


// Costruisce il complesso simpliciale completo a partire dagli 1-simplessi,
    utilizzando la matrice di adiacenza.
// I vertici sono nominati da 0 a n-1, come gli indici della matrice.
SimplicialComplex* complex_from_adjacency_matrix_complete (int** M, int n) {
```

```
    SimplicialComplex* complex = (SimplicialComplex*) malloc((n)*sizeof(SimplicialComplex));
    int i;

    // aggiungo i vertici
    complex[0].size=n;
    Simplex *current, *new_simplex; // current sarà un puntatore all'ultimo elemento della lista degli 0-simplessi, new_simplex servirà per creare quelli da aggiungere

    new_simplex = (Simplex*) malloc(sizeof(Simplex));
    new_simplex->next = NULL;
    new_simplex->position = 0;
    new_simplex->vertices = (int*) malloc(sizeof(int));
    new_simplex->vertices[0]=0;
    complex[0].simplices=new_simplex;
    current=new_simplex;

    for (i=1; i<n; i++) {
        new_simplex = (Simplex*) malloc(sizeof(Simplex));
        new_simplex->next = NULL;
        new_simplex->position = i;
        new_simplex->vertices = (int*) malloc(sizeof(int));
        new_simplex->vertices[0]=i;
        current->next=new_simplex;
        current=current->next;
    }

    // costruisco fino al livello n
    for (i=1; i<n; i++) {
        if (complex[i-1].simplices!=NULL) { // posso costruire i-simplessi solo se esistono degli (i-1)-simplessi
            add_k_simplices_adjacency_matrix(complex,M,n,i);
        }
        else {
            complex[i].simplices=NULL;
            complex[i].size=0;
        }
    }

    return complex;
}

// Costruisce il complesso simpliciale a partire dagli 1-simplessi troncato fino ai k-simplessi (inclusi), utilizzando la matrice di adiacenza.
// I vertici sono nominati da 0 a n-1, come gli indici della matrice.
// Utile ad esempio se serve calcolare il gruppo di omologia H_(k-1) e non serve l'intero complesso simpliciale.
SimplicialComplex* complex_from_adjacency_matrix_truncated (int** M, int n, int k) {
    SimplicialComplex* complex = (SimplicialComplex*) malloc((k+1)*sizeof(SimplicialComplex));
    int i;

    // aggiungo i vertici
    complex[0].size=n;
    Simplex *current, *new_simplex; // current sarà un puntatore all'ultimo elemento della lista degli 0-simplessi, new_simplex servirà per creare quelli da aggiungere

    new_simplex = (Simplex*) malloc(sizeof(Simplex));
    new_simplex->next = NULL;
    new_simplex->position = 0;
    new_simplex->vertices = (int*) malloc(sizeof(int));
    new_simplex->vertices[0]=0;
    complex[0].simplices=new_simplex;
    current=new_simplex;

    for (i=1; i<n; i++) {
        new_simplex = (Simplex*) malloc(sizeof(Simplex));
```

```
        new_simplex->next = NULL;
        new_simplex->position = i;
        new_simplex->vertices = (int*) malloc(sizeof(int));
        new_simplex->vertices[0]=i;
        current->next=new_simplex;
        current=current->next;
    }

    // costruisco fino al livello k
    for (i=1; i<=k; i++) {
        if (complex[i-1].simplices!=NULL) { // posso costruire i-simplessi solo se esistono degli (i-1)-simplessi
            add_k_simplices_adjacency_matrix(complex,M,n,i);
        }
        else {
            complex[i].simplices=NULL;
            complex[i].size=0;
        }
    }

    return complex;
}

// Costruisce il complesso simpliciale completo a partire dagli n vertici e dagli 1-simplessi.
// I vertici verranno rinominati a partire da 0 e con interi consecutivi (se vogliamo posso scrivere una funzione per farli ritornare ai nomi originali)
SimplicialComplex* complex_from_1_simplices_complete (SimplicialComplex* complex, int n) {
    int i,j;
    int** M = (int**) malloc(n*sizeof(int*));
    for (i=0; i<n; i++) M[i] = (int*) malloc(n*sizeof(int));
    // salvo per comodita' in un array i vertici (per ovviare a problemi di nomenclatura ed evitare di scorrere sempre la lista)
    int* vertices = (int*) malloc(n*sizeof(int));
    Simplex* current=complex[0].simplices;
    i=0;
    while (current!=NULL) {
        vertices[i]=current->vertices[0];
        i++;
        current=current->next;
    }

    // costruisco la matrice di adiacenza
    for (i=0; i<n; i++) {
        M[i][i]=1;
        for (j=i+1; j<n; j++) {
            // controllo se il vertice i-esimo e il j-esimo sono collegati da un 1-simplesso
            if (check_neighbor(complex,vertices[i],vertices[j])!=0) {
                M[i][j]=1; M[j][i]=1;
            }
            else {
                M[i][j]=0; M[j][i]=0;
            }
        }
    }

    return complex_from_adjacency_matrix_complete(M,n);
}

// Costruisce il complesso simpliciale a partire dagli n vertici e dagli 1-simplessi troncato fino ai k-simplessi (inclusi).
// I vertici verranno rinominati a partire da 0 e con interi consecutivi (se vogliamo posso scrivere una funzione per farli ritornare ai nomi originali)
SimplicialComplex* complex_from_1_simplices_truncated (SimplicialComplex* complex, int n, int k) {
    int i,j;
    int** M = (int**) malloc(n*sizeof(int*));
    for (i=0; i<n; i++) M[i] = (int*) malloc(n*sizeof(int));
```

```c
    // salvo per comodita' in un array i vertici (per ovviare a problemi di nomenclatura ed evitare di scorrere sempre la lista)
    int* vertices = (int*) malloc(n*sizeof(int));
    Simplex* current=complex[0].simplices;
    i=0;
    while (current!=NULL) {
        vertices[i]=current->vertices[0];
        i++;
        current=current->next;
    }

    // costruisco la matrice di adiacenza
    for (i=0; i<n; i++) {
        M[i][i]=1;
        for (j=i+1; j<n; j++) {
            // controllo se il vertice i-esimo e il j-esimo sono collegati da un 1-simplesso
            if (check_neighbor(complex,vertices[i],vertices[j])!=0) {
                M[i][j]=1; M[j][i]=1;
            }
            else {
                M[i][j]=0; M[j][i]=0;
            }
        }
    }

    return complex_from_adjacency_matrix_truncated(M,n,k);
}

// Aggiunge al complesso simpliciale (che contiene fino ai (k-1)-simplessi) i k-simplessi specificati dalla matrice di adiacenza M n*n.
// E' necessario che il complesso sia ordinato e k>0. M[i][j] sara' diversa da 0 se (i,j) e' nel complesso (denomino i vertici a partire da 0).
void add_k_simplices_adjacency_matrix (SimplicialComplex* complex, int** M, int n, int k) {
    // parto con 0 k-simplessi: la lista complex[k].simplices e' vuota
    complex[k].simplices=NULL;
    int num_k_simplices=0;
    int i,j;

    // ricerco i k-simplessi da aggiungere:

    Simplex* previous_simplex = complex[k-1].simplices; // scorrerà i (k-1)-simplessi
    Simplex *current = NULL, *new_simplex; // current sarà un puntatore all'ultimo elemento della lista dei k-simplessi, new_simplex servirà per creare quelli da aggiungere
    while (previous_simplex!=NULL) { // per costruire i k-simplessi devo partire dai (k-1)-simplessi

        // cerco nuovi vertici a partire da quelli successivi all'ultimo gia' presente (funziona se il complesso e' ordinato)
        for (j=previous_simplex->vertices[k-1]+1; j<n; j++) {
            if (check_neighbor_adjacency_matrix(M,previous_simplex->vertices,k,j)!=0) {
                // ho trovato il vicino j: costruisco il nuovo k-simplesso

                if (num_k_simplices==0) { // caso in cui aggiungo il primo k-simplesso alla lista
                    new_simplex = (Simplex*) malloc(sizeof(Simplex));
                    new_simplex->next = NULL;
                    new_simplex->position = num_k_simplices;
                    new_simplex->vertices = (int*) malloc((k+1)*sizeof(int));
                    for (i=0; i<k; i++) new_simplex->vertices[i]=previous_simplex->vertices[i];
                    new_simplex->vertices[k]=j;

                    complex[k].simplices=new_simplex;
                    current=new_simplex;
                }
```

```
                else { // caso in cui la lista dei k-simplessi e' non vuota
                    new_simplex = (Simplex*) malloc(sizeof(Simplex));
                    new_simplex->next = NULL;
                    new_simplex->position = num_k_simplices;
                    new_simplex->vertices = (int*) malloc((k+1)*sizeof(int));
                    for (i=0; i<k; i++) new_simplex->vertices[i]=previous_simplex->vertices[i];
                    new_simplex->vertices[k]=j;

                    current->next=new_simplex;
                    current=current->next;
                }
                num_k_simplices++;
            }
        }

        previous_simplex = previous_simplex->next;
    }

    complex[k].size=num_k_simplices;
    return;
}

// Controlla se il vertice di indice j e' "vicino" ai k vertici di indici rows, ossia se il minore di righe rows e colonna j e' tutto non nullo.
// Restituisce 0 se j non e' un vicino, 1 se lo e'.
int check_neighbor_adjacency_matrix (int** M, int* rows, int k, int j) {
    for (int i=0; i<k; i++) {
        if (M[rows[i]][j]==0) return 0;
    }
    return 1;
}

// Controlla se a e b sono "vicini", ossia se la coppia (a,b) appartiene agli 1-simplessi. Restituisce 0 se non sono vicini, 1 se lo sono.
int check_neighbor (SimplicialComplex* complex, int a, int b) {
    Simplex* current=complex[1].simplices;
    while (current!=NULL) { // scorro lungo gli 1-simplessi
        if ((current->vertices[0]==a && current->vertices[1]==b) || (current->vertices[0]==b && current->vertices[1]==a)) return 1;
        current=current->next;
    }
    return 0;
}
```

1.2 Complessi di catene e omologia simpliciale

Per studiare meglio i complessi simplicicali è utile introdurre una numerazione dei vertici al fine di poter ”percorrere” l'insieme in analisi, dove per convenzione il vero positivo è da un vertice minore ad uno maggiore. Questo permette di definire la funzione

$$\partial = \sum_{i=0}^{n} (-1)^i (x_0, ..., x_{i-1}, \hat{x}_i, x_{i+1}, ..., x_n)$$

dove con $\hat{x}_i$ si intende che l'elemento i-esimo del simplesso è ignorato. Viene dunque naturale introdurre il concetto di **combinazione lineari di k-simplessi** a coefficienti in un qualche anello commutativo A:

$$C_k((X,S),A) := \left\{ \sum a_i \sigma_i \mid a_i \in A \ , \ \sigma_i \in S_k \right\}$$

C_k è un A-modulo libero, nonchè un grubbo abeliano finitamente generato (S_k ne è una base), e ∂ è una funzione A-lineare da C_k a C_{k-1}, quindi può essere studiata attraverso una matrice.

Listing 4: Compl Simpl.h

```c
#pragma once
// MioFile.h
#ifndef Complesso_Simplciale
    #define Complesso_simpliciale

    #include <stdbool.h>
    #include <stdio.h>
    #include <stdlib.h>
    #include <math.h>   // Per la funzione pow

    /*Il complesso simpliciale è formato da un vettore di liste di vettori
    Ogni cella del primo vettore conterrà i rispettivi simplessi cella 0 gli 0-simpelessi cella 1 gli 1-simplessi etc..
    a loro volta gli n-simplessi sono organizzati in liste di vettori così da distinguere i vertici che compongono il rispettivo simplesso.*/


    //Definzione delle strutture
    //DEfinzione degli n-simplessi
    typedef struct Simplex {
        int* vertices;  // Array di vertici del simplicio
        int position;   // Ordine del simplicio nella base
        struct Simplex* next;
    } Simplex;

    // Definizione di un complesso simpliciale
    typedef struct {
        Simplex* simplices;  // Array di simplici
        int size;            // Numero degli n-simplessi esenziale per costruire le matrici di bordo e non scorrere tutta la lista
    } SimplicialComplex;


    //Definizione delle funzioni
    Simplex* createSimplex(int, int);
    Simplex* createSimplex_file(int, int, FILE*);
    SimplicialComplex* readComplex(int);
    SimplicialComplex* readComplex_file(char *);
    void printComplex(SimplicialComplex*, int);
    int* creasubs(int*, int, int);
    bool equal(int*, int*, int);
    bool isIn(Simplex*, Simplex*, int);
    bool isSimplicial(SimplicialComplex*, int);
        int** edge_Matrix(SimplicialComplex*, int);
    int base_number(Simplex*, int*, int);
        int matrix_rank(int**, int, int);

    //Esplicitare le funzioni

    #pragma region gestione dei simplessi e dei complessi simpliciali

        // funzione per leggere un complesso da input quindi sappiamo la misura del complesso più grande = size
        SimplicialComplex* readComplex(int size){
                SimplicialComplex* complex = (SimplicialComplex*)malloc(size * sizeof(SimplicialComplex));
                int n = 0;

                //itero i vari n-simplessi su tutta la lunghezza del vettore
                for (int i = 0; i < size; i++) {
                        printf("Inserisci il numero di %d-simplessi: ", i);
                        scanf("%d", &n);
                complex[i].size = n;
                        complex[i].simplices = createSimplex(i, n);
                }
                return complex;
        }

                SimplicialComplex* readComplex_file(char* file) {
```

```
                FILE* f = fopen(file, "rt");
                if (!f) {
                        printf("Errore nell'apertura del file\n");
                        return NULL;
                }
    else
                        printf("File aperto correttamente\n");
                int size = 0;
                fscanf(f, "%d", &size);
                printf("size = %d\n", size);
                SimplicialComplex* complex = (SimplicialComplex*)malloc(size * sizeof(SimplicialComplex));
                int n = 0;
                for (int i = 0; i < size; i++) {
                        fscanf(f, "%d", &n);
                        complex[i].size = n;
                        complex[i].simplices = createSimplex_file(i, n, f);
                }
                fclose(f);
                return complex;
        }

// funzione per stampare un simplesso
void printComplex(SimplicialComplex* complex, int size){
        Simplex* app;
                for (int i = 0; i < size; i++) {
                                printf("%d-simplessi\n", i);
                                printf("size = %d\n", complex[i].size);
                        app = complex[i].simplices;
            if (app) {
                printf("{ ");
                while (app) {
                    printf("( ");
                    for (int j = 0; j <= i; j++)
                        printf("%d ", app->vertices[j]);
                    printf(")");
                    app = app->next;
                    if (app)
                        printf(", ");
                }
                printf(" }\n");
            }
                }
        return;
}

// funzione per creare un sottoinsieme di un simplesso togliendo il vertice i-esimo
int* creasubs(int* v, int i, int n) {
    int* s = (int*)malloc((n) * sizeof(int));
    int k = 0;
        //porto il cilco fino a n perchè il vertice i-esimo non deve essere inserito quindi inserisco solo n-1 vertici
    for (int j = 0; j <= n; j++) {
        if (j != i) {
            s[k] = v[j];
            k++;
        }
    }
    return s;
}

//controlla se due vettori sono uguali
bool equal(int* v, int* s, int n) {
    for (int i = 0; i < n; i++) {
        if (v[i] != s[i])
            return false;
    }
    return true;
}
```

```c

//Funzione per controllare se i sottoinsiemi dell' n simplesso sono contenuti nell' n-1 simplesso
bool isIn(Simplex* s, Simplex* s1, int sizeN) {
    int* v = (int*)malloc(sizeN * sizeof(int));
    Simplex* Nsimp = s, * Nmosimp = s1;
    bool isin = false;
        //fino a quando non ho terminato la lista degli n-simplessi
    while (Nsimp) {
        for (int i = 0; i < sizeN + 1; i++) {
                        //creo i sottoinsiemi del n-simplesso
            v = creasubs(Nsimp->vertices, i, sizeN+1);
            isin = false;
                        //fino a quando non ho terminato la lista degli n-1-simplessi oppure ho trovato il sottoinsieme
            while (Nmosimp && !isin) {
                                //confronto i due vettori
                if (equal(Nmosimp->vertices, v, sizeN)) {
                    isin = true;
                }
                else
                    isin = false;
                Nmosimp = Nmosimp->next;
            }
            if (!isin)
                return false;
            Nmosimp = s1;
        }
        Nsimp = Nsimp->next;
    }
    return true;
}

// Funzione per controllare se il complesso è simpliciale
bool isSimplicial(SimplicialComplex* sc, int size) {
        //faccio il controllo per ogni n-simplesso partendo dall'ultimo
    for (int i = size - 1; i > 0; i--)
        if (!isIn(sc[i].simplices, sc[i - 1].simplices, i))
            return false;
    return true;
}

// Funzione per creare un simplesso sapendo prima le dimensioni
Simplex* createSimplex(int size, int n) {
    Simplex* simplex, * head, * app;      // Definiamo la testa del simplicio + due simplex ausiliari
    simplex = (Simplex*)malloc(sizeof(Simplex));
        //mi salvo la testa del simplesso e la riempio
    head = simplex;
    simplex->next = NULL;
    simplex->vertices = (int*)malloc((size + 1) * sizeof(int));
    printf("Inserisci i %d-simplessi scrivendo i vertici in ordine crescente separati da uno spazio es: (1 2 3)\n", size);
    for (int j = 0; j < size + 1; j++) {
        scanf("%d", &simplex->vertices[j]);
    }
        simplex->position = 0;
        //riempio i restanti n-1 simplessi utilizzando un simplex ausiliario
    for (int i = 1; i < n; i++) {
        app = (Simplex*)malloc(sizeof(Simplex));
        app->next = NULL;
        app->vertices = (int*)malloc((size + 1) * sizeof(int));
        for (int j = 0; j < size + 1; j++) {
            scanf("%d", &app->vertices[j]);
        }
                app->position = i;
        simplex->next = app;
        simplex = simplex->next;
    }
```

```
        return head;
    }

    //stessa funzione di sopra ma leggo da file
    Simplex* createSimplex_file(int size, int n, FILE* f) {
        Simplex* simplex, * head, * app;
        simplex = (Simplex*)malloc(sizeof(Simplex));
        head = simplex;
        simplex->next = NULL;
        simplex->vertices = (int*)malloc((size + 1) * sizeof(int));
        for (int j = 0; j < size + 1; j++) {
            fscanf(f, "%d", &simplex->vertices[j]);
        }
        simplex->position = 0;
        for (int i = 1; i < n; i++) {
            app = (Simplex*)malloc(sizeof(Simplex));
            app->next = NULL;
            app->vertices = (int*)malloc((size + 1) * sizeof(int));
            for (int j = 0; j < size + 1; j++) {
                fscanf(f, "%d", &app->vertices[j]);
            }
            app->position = i;
            simplex->next = app;
            simplex = simplex->next;
        }
        return head;
    }

#pragma endregion


#pragma region edge_Matrix

            // Funzione per calcolare la matrice di bordo
            int** edge_Matrix(SimplicialComplex* sc, int n) {
        int i = 0, j = 0, k = -1, row = sc[n - 1].size, col = sc[n].size;
        int** matrix;
        matrix = (int**)calloc(row,sizeof(int*));
                    Simplex* Nsimp, * Nmosimp;
                    int* v = (int*)malloc(n * sizeof(int));
                    Nsimp = sc[n].simplices;
                    Nmosimp = sc[n - 1].simplices;
                    //itero i vari n-simplessi
        for (int i = 0; i < row; i++)
            matrix[i] = (int*)calloc(col, sizeof(int));
            for (int i = 0; i < sc[n].size; i++) {
                                //itero i vari n-1-simplessi
                                //creo i sotto inisiemi di ciascun simplesso e vedo che posizione ha nella base
                for (int j = 0; j <= n; j++) {
                                        v = creasubs(Nsimp->vertices, j, n);
                                        k = base_number(Nmosimp, v, n);
                                        //printf("k = %d\n", k);
                    if (k != -1) {
                        matrix[k][i] = pow(-1, j);
                    }
                }
                                Nsimp = Nsimp->next;
                        }
                    return matrix;
            }

            int base_number(Simplex* sc, int* v, int n) {
                    Simplex* app = sc;
                    while (app) {
                            if (equal(app->vertices, v, n))
                                    return app->position;
                            app = app->next;
                    }
                    return -1;
```

```
                }
    #pragma endregion

    #pragma region matrix_rank
        void swap_rows(int** matrix, int row1, int row2, int cols) {
            for (int i = 0; i < cols; i++) {
                double temp = matrix[row1][i];
                matrix[row1][i] = matrix[row2][i];
                matrix[row2][i] = temp;
            }
        }

                /*
        // Funzione per calcolare il rango di una matrice
                int matrix_rank(int** matrix, int rows, int cols) {
            int rank = cols;  // Inizialmente assumiamo il rango massimo possibile

            for (int row = 0; row < rank; row++) {
                // Controlliamo se l'elemento diagonale è diverso da zero
                if (fabs(matrix[row][row]) > 0) {
                    // Eliminiamo gli elementi sotto l'elemento pivot
                    for (int i = 0; i < rows; i++) {
                        if (i != row) {
                            double factor = matrix[i][row] / matrix[row][row];
                            for (int j = 0; j < cols; j++) {
                                matrix[i][j] -= factor * matrix[row][j];
                            }
                        }
                    }
                }
                else {
                    // Se il pivot è zero, cerchiamo una riga non nulla da
                        scambiare
                    int reduce = 1;
                    for (int i = row + 1; i < rows; i++) {
                        if (fabs(matrix[i][row]) > 0) {
                            swap_rows(matrix, row, i, cols);
                            reduce = 0;
                            break;
                        }
                    }

                    if (reduce) {
                        // Se non troviamo una riga valida, riduciamo il rango
                        rank--;

                        // Spostiamo l'ultima colonna a sinistra
                        for (int i = 0; i < rows; i++) {
                            matrix[i][row] = matrix[i][rank];
                        }
                    }
                    row--;
                }
            }

            return rank;
                }*/
#pragma endregion

#endif
```

Un' altra proprietà fondamentale di ∂ è che

$$\partial^2 = 0 \ ,$$

da cui si ottiene che $Im(\partial : C_{k+1} \to C_k) \subset Ker(\partial : C_k \to C_{k-1})$, quindi ha senso introdurre il concetto di **omologia del complesso simpliciale**:

$$H_n(C) := \frac{Ker(\partial : C_k \to C_{k-1})}{Im(\partial : C_{k+1} \to C_k)}$$

In particolare nel corso è stato dimostrato che l'omologia è invariante per riordinamento dei vertici (un morfismo di insiemi simpliciali ne induce uno anche sui rispettivi C_k) e

$$H_0(X, \mathbb{R}) \cong \mathbb{R}^{numero\ di\ componenti\ connesse}$$

Intuitivamente H_1 rappresenta il numero di triangoli con interno vuoto (ovvero tre 1-simplessi che non formano un 2-simpesso), H_2 il numero di tetraedri con interno vuoto e così via, il che fornisce un'idea geometrica e spaziale sempre più accurata di come i dati siano distribuiti.

Listing 5: Omologie.c

```c
#include "..\Include\Compl_Simpl.h"
#include "..\Include\Omo.h"

// Funzione per calcolare l'omologia di un complesso simpliciale
int main() {

        int size = 0, n = 2;
        int** matrix1 = NULL, ** matrix2 = NULL;
        //mi serve per stampare il complesso simpliciale
        printf("Inserisci la grandezza del simplesso piu' grande nel complesso: ");
        scanf("%d", &size);

        //Leggo il simplesso da un file e lo stmpo
        SimplicialComplex* complex = readComplex_file("simplicial_complex.txt");
        printComplex(complex, 3);

        //controllo sia un complesso simpliciale
        if (!isSimplicial(complex, size)) {
                printf("Errore il complesso non e' simpliciale inserire un nuovo simplesso\n");
                return 1;
        }

        //chiedo di quale numero voglio calcolare h
        printf("Inserisci il numero n di cui vuoi calcolare l'omologia H_n \n (n >= 0) = ");
        scanf("%d", &n);

        n++;

        //calcolo le matrici di bordo dei simplessi n e n-1
        //per comodità di notazione ho aggiunto 1 ad n
        if (n == 1) {
                matrix2 = NULL;
                matrix1 = edge_Matrix(complex, n);
        }
        else if (n == size) {
                matrix1 = NULL;
                matrix2 = edge_Matrix(complex, n - 1);
        }
        else if (n >= size + 1) {
                matrix1 = NULL;
                matrix2 = NULL;
        }
        else {
                matrix1 = edge_Matrix(complex, n);
                matrix2 = edge_Matrix(complex, n - 1);
        }

        //calcolo la dimensione dell'omologia più veloce che decomporla in z-moduli
        int h = dim_omologia(matrix1, complex[n - 1].size, complex[n].size, matrix2, complex[n - 2].size, complex[n - 1].size);
```

```
        printf("Il gruppo di omologia H_%d del complesso simpliciale inserito ha dimensione = %d\n", n - 1, h);

        //esempio di calcolo delll'imologia decomposta in z-moduli
        int** h1 = NULL;
        int** M = NULL;
        M = input_null(M, 3, 2);
        int** M2 = NULL;
        M2 = input_null(M2, 1, 3);

        M[0][0] = 8;
        M[0][1] = -15;
        M[1][0] = -15;
        M[1][1] = 29;
        M[2][0] = -6;
        M[2][1] = 13;

        M2[0][0] = -9;
        M2[0][1] = -6;
        M2[0][2] = 3;

        h1 = omologia(M, 3, 2, M2, 1, 3);


        return;
}
```

Listing 6: Omo.h

```
#pragma once
#ifndef Omo
        #define Omo
        #include "..\Include\Matrix.h"

    // Questo file include funzioni per la manipolazione delle omologie

        //Definizioni funzioni
        int** omologia(int**, int, int, int**, int, int);
        int compare(const void*, const void*);

        int dim_omologia(int**, int, int, int**, int, int);
        bool is_Complex(int**, int, int, int**, int, int);

        //Implementazione

        //semplcie funzione per comparare due interi
        int compare(const void* a, const void* b) {
                return (*(int*)a - *(int*)b);
        }

        //funzione per controllare se due matrici formano un complesso ovvero se la loro moltiplicazione è nulla
        //proprietà delle matrici di bordo
        bool is_Complex(int** matrix1, int row1, int col1, int** matrix2, int row2, int col2) {
                if (col2 != row1) {
                        return false;
                }
                int** temp = mul_matrix(matrix2, row2, col2, matrix1, row1, col1);
                //print_matrix(temp, row2, col1);
                for (int i = 0; i < row2; i++) {
                        for (int j = 0; j < col1; j++) {
                                if (temp[i][j] != 0) {
                                        return false;
                                }
                        }
                }
```

```
        return true;
    }


    //funzione per calcolare la dimensione dell'omologia n richiesta passo in input solo le due matrici di bordo
    int dim_omologia(int** matrix1, int row1, int col1, int** matrix2, int row2, int col2) {
        int h = 0, ck = 0, rk = 0, rk1 = 0;

        //controlliamo che le due matrici diano un complesso
        if (matrix1 && matrix2) {
            if (!is_Complex(matrix1, row1, col1, matrix2, row2, col2)) {
                return -1;
            }
        }

        int** D = NULL;
        int** S = NULL;
        int** T = NULL;

        //calcoliamo la dimensione dell'omologia come h = ck - rk - rk1 dove ck è il numero di n simplessi
        //rk è il rango della matrice di bordo n+1-esima e rk1 è il rango della matrice di bordo n-esima
        //controlliamo se esistono le matrici e quindi ne calcolo il rango utilizzando la matrice di smith
        if (matrix2) {

            D = input_null(D, row2, col2);
            S = input_id(S, row2);
            T = input_id(T, col2);

            for (int i = 0; i < row2; i++) {
                for (int j = 0; j < col2; j++) {
                    D[i][j] = matrix2[i][j];
                }
            }


            ck = col2;

            SmithNormalForm(D, S, T, row2, col2);
            rk = rank_matrix_diag(D, row2, col2);
            printf("rk = %d\n", rk);

            free(D);
            free(S);
            free(T);
        }
        else
            rk = 0;

        if (matrix1) {
            D = input_null(D, row1, col1);
            S = input_id(S, row1);
            T = input_id(T, col1);

            for (int i = 0; i < row1; i++) {
                for (int j = 0; j < col1; j++) {
                    D[i][j] = matrix1[i][j];
                }
            }

            ck = row1;

            SmithNormalForm(D, S, T, row1, col1);
            rk1 = rank_matrix_diag(D, row1, col1);
            printf("rk1 = %d\n", rk1);
```

```
				free(D);
				free(S);
				free(T);
			}
			else
				rk1 = 0;


			printf("ck = %d\n", ck);
			h = ck - rk - rk1;

			return h;
		}


		//Come rappresentare l'omologia in z-moduli
		int** omologia(int** matrix1, int row1, int col1, int** matrix2, int row2,
		    int col2) {
			int** h = NULL;

			//controlliamo che le due matrici diano un compleso
			if (!is_Complex(matrix1, row1, col1, matrix2, row2, col2)) {
				printf("Errore: le matrici non formano un complesso\n");
				return NULL;
			}

			int** D = NULL;
			D = input_null(D, row2, col2);
			int** S = NULL;
			S = input_id(S, row2);
			int** T = NULL;
			T = input_id(T, col2);


			for (int i = 0; i < row2; i++) {
				for (int j = 0; j < col2; j++) {
					D[i][j] = matrix2[i][j];
				}
			}

			//calcoliamo la forma di smith della matrice An
			SmithNormalForm(D, S, T, row2, col2);

			//troviamo il rango della matrice ed estraiamo una base del nucleo
			    ovvero
			//le ultime r colonne della matrice T (SAT = D) che salvo in T2
			int r = rank_matrix_diag(D, row2, col2);

			int** T2 = NULL;
			T2 = input_null(T2, col2, col2-r);


			for (int i = 0; i < col2; i++) {
				for (int j = 0; j < col2 - r; j++) {
					T2[i][j] = T[i][j + r];

				}
			}

			//troviamo la forma di smith della base del nucleo appena trovata
			//per risolvere Bx = An+1 negli interi moltiplicando per l'inverso
			    di ogni matrice che gia ho
			//e fermandomi al rango della matrice B

			int** S1 = NULL;
			S1 = input_id(S1, col2);
			int** T1 = NULL;
			T1 = input_id(T1, col2 - r);
			int** D1 = NULL;
```

```c
        D1 = input_null(D1, col2, col2 - r);

        for (int i = 0; i < col2; i++) {
                for (int j = 0; j < col2 - r; j++) {
                        D1[i][j] = T2[i][j];
                }
        }

        SmithNormalForm(D1, S1, T1, col2, col2 - r);
        int** temp = NULL;

        //mi fermo a col2-r righe per s1*A
        temp = mul_matrix(S1, col2 - r, col2, matrix1, row1, col1);

        //D^-1 * S1 * A
        for (int i = 0; i < col2 - r; i++)
        {
                for (int j = 0; j < col1; j++)
                {
                        temp[i][j] = temp[i][j] / D1[i][i];
                }
        }

        //T1 * D^-1 * S1 * A=x
        temp = mul_matrix(T1, col2 - r, col2 - r, temp, col2 - r, col1);


        free(S1);
        free(T1);
        free(D1);

        //ora decompongo in z-moduli il risultato x=temp sempre attaverso
            smith e la matrice diagonale

        S1 = NULL;
        S1 = input_id(S1, col2 - r);
        T1 = NULL;
        T1 = input_id(T1, col2 - r);
        D1 = NULL;
        D1 = input_null(D1, col2 - r, col2 - r);

        for (int i = 0; i < col2 - r; i++) {
                for (int j = 0; j < col2 - r; j++) {
                        D1[i][j] = temp[i][j];
                }
        }

        SmithNormalForm(D1, S1, T1, col2 - r, col2 - r);
        h = (int*)calloc(col2 - r - 1, sizeof(int));
        for (int i = 0; i < col2 - r ; i++) {
                h[i] = D1[i][i];
        }

        //printiamo il risultato ottenuto ovvero i valori di h
        printf("\nDecomposizione di h\n");
        printf("Z_%d", h[0]);
        for (int i = 1; i < col2 - r; i++) {
                printf(" + ");
                printf("Z_%d", h[i]);
        }

        /* non sempre trovo un minore invertibile negli itneri quindi
            questo codice non va bene
        int** B = NULL;
        int** B_inv = NULL;
        int** C = NULL;
        B = input_null(B, col2 - r, col2 - r);
        B_inv = input_null(B_inv, col2 - r, col2 - r);
        C = input_null(C, col2 - r, col1);

```

```
		//cerco un minore invertibile di T2
		//uso gauss per mettere la matrice in forma diagonale
		//poi prendo gli indici con elemnti non nulli e li salvo per estrarne un minore
		int* index = (int*)calloc(col2 - r, sizeof(int));
		gauss_rectangular(T2, col2, col2 - r);
		printf("Matrice T2 gauss\n");
		print_matrix(T2, col2, col2 - r);
		for (int i = 0; i < col2 - r; i++) {
			int k = 0;
			for (k = 0; k < col2; k++) {
				if (T2[k][i] != 0) {
					index[i] = k;

					for (int j = i; j < col2 - r; j++) T2[k][j] = 0;
					break;
				}
			}
		}
		//ordino gli indici per prendere le righe corrispondenti
		//sia su B che su A
		qsort(index, col2 - r, sizeof(int), compare);


		for (int i = 0; i < col2 - r; i++) {
			printf("i = %d \n", index[i]);
			int k = index[i];
			for (int j = 0; j < col2 - r; j++) {
				B[i][j] = T[k][j + r];
			}

			for( int j = 0; j < col1; j++){
				C[i][j] = matrix1[k][j];
			}
		}

		int iter = -1;

		//cerco il minore invertibile di B negli interi condizione sul determinante
		while (det_matrix_triangular(B, col2 - r) != 1 && det_matrix_triangular(B, col2 - r) != - 1) {
			iter++;
			if (iter == col2)
			{
				printf("Matrice non invertibile\n");
				return NULL;
			}
			else
			{

				while(T[iter][col2 - 1] == 0 && iter < col2)
					iter++;
			}

			for (int j = 0; j < col2 - r; j++) {
				B[col2 - r - 1][j] = T[iter][j + r];
			}

			for (int j = 0; j < col1; j++) {
				C[col2 - r - 1][j] = matrix1[iter][j];
			}
		}



		//inverto la matrice
		B_inv = invert_matrix_integer(B, col2 - r);

```

```
                printf("Matrice B\n");
                print_matrix(B, col2 - r, col2 - r);
                printf("Matrice B_inv\n");
                print_matrix(B_inv, col2 - r, col2 - r);

                int** temp = mul_matrix(B_inv, col2 - r, col2 - r, C, col2 - r,
                    col2 - r);


                printf("Matrice molt\n");
                print_matrix(temp, col2 - r, col2 - r);



                printf("Matrice C\n");
                print_matrix(C, col2 - r, col1);
                */


                free(D);
                free(S);
                free(T);
                free(S1);
                free(T1);
                free(D1);
                free(T2);

                return h;
        }


#endif
```

1.3 Forma normale di Smith

Teorema: Sia $A \in \mathbb{Z}^{n\times m}$. Allora esistono S, T matrici invertibili in $\mathbb{Z}$ tali che

$$A = SDT \ , \ D = \begin{pmatrix} d_1 & 0 & 0 & ... & 0 \\ 0 & d_2 & 0 & ... & 0 \\ ... \\ 0 & 0 & ... & d_k & 0 \\ 0 & 0 & ... & 0 & 0 \\ 0 & 0 & ... & 0 & 0 \end{pmatrix}$$

con d_i determinati univocamente e tali che $d_i | d_{i+1}$.
Un possibile algoritmo per determinare le due matrici è il seguente:

Listing 7: Smith.h

```
#pragma once
#ifndef Smith
#define Smith

int** input_sm (int**, int**, int, int);
int** input_id (int**, int);
int** input_null (int**, int, int);
void printMatrix (int**, int, int, int);
bool nullMatrix (int**, int, int);
int** traspMatrix (int**, int**, int, int);
void switchZeros (int**, int**, int**, int**, int, int, int);
void switchSigns (int**, int**, int, int, int, int);
```

```c
void swapRows (int**, int, int, int);
void addRows (int**, int, int, int, int);
void minUp (int**, int**, int, int, int, int);
void div_rem (int**, int**, int, int, int, int);
bool nullVector (int**, int, int);
void Smith_fst (int**, int**, int**, int**, int**, int, int, int);
int my_min (int, int);
bool bool_dk (int**, int, int);
void SmithNormalForm (int**, int**, int**, int, int);
void multiplyMatrices(int**, int**, int**, int, int, int);
void Smith_D (int**, int, int);
int checkDiagZeros (int**, int);
void Smith_fst5mat (int**, int**, int**, int**, int**, int**, int**, int**, int, int, int);
void SmithNormalForm5mat (int**, int**, int**, int**, int**, int, int);
void minUp5mat (int**, int**, int**, int, int, int, int);
void div_rem5mat (int**, int**, int**, int, int, int, int);
void switchSigns5mat (int**, int**, int**, int, int, int, int);
void switchZeros5mat (int**, int**, int**, int**, int**, int, int, int);

// inizializzazione matrice i cui elementi sono scelti dall'utente
int** input_sm (int **mat, int **temp, int m, int n) {
        mat=calloc(m, sizeof(int*));
        for (int i=0; i<m; i++) {
                mat[i]=calloc(n, sizeof(int));
        }
        printf("Elementi della matrice:\n");
        for (int i=0; i<m; i++) {
                for (int j=0; j<n; j++) {
                        scanf("%d", &mat[i][j]);
                        temp[i][j]=mat[i][j];
                }
        }
        return mat;
}

// inizializzazione matrice identità
int** input_id (int **mat, int m) {
        mat=calloc(m, sizeof(int *));
        for (int i=0; i<m; i++) {
                mat[i]=calloc(m, sizeof(int));
        }
        for (int i=0; i<m; i++) {
                mat[i][i]=1;
        }
        return mat;
}

// inizializzazione matrice nulla
int** input_null(int **mat, int m, int n) {
        mat=calloc(m, sizeof(int*));
        for (int i=0; i<m; i++) {
                mat[i]=calloc(n, sizeof(int));
        }
        return mat;
}

// stampare la matrice
void printMatrix (int **mat, int m, int n, int t) {
        for (int i=0; i<m; i++) {
                for (int j=0; j<n; j++) {
                        if (mat[i][j]<0) {
                                printf("%d ", mat[i][j]);
                        } else {
                                printf(" %d ", mat[i][j]);
                        }
                }
                printf("\n");
                if (i!=(m-1)) {
                        for (int l=0; l<t; l++) {
```

```
                                printf(" ");
                        }
                }
        }
        return;
}

// verificare che la matrice non sia nulla
bool nullMatrix (int **mat, int m, int n) {
        for (int i=0; i<m; i++) {
                for (int j=0; j<n; j++) {
                        if (mat[i][j]!=0) {
                                return false;
                        }
                }
        }
        return true;
}

// trasporre una matrice
int** traspMatrix (int **mat, int **trasp, int m, int n) {
        for (int i=0; i<n; i++) {
                for (int j=0; j<m; j++) {
                        trasp[i][j]=mat[j][i];
                }
        }
        return trasp;
}

// spostare un'eventuale colonna nulla in fondo alla matrice
void switchZeros (int **A, int **T, int **At, int **Tt, int m, int n, int k) {
        bool var=true;
        for (int i=k; i<m; i++) {
                if (A[i][k]!=0) {
                        var=false;
                        break;
                }
        }
        if (var) {
                At=traspMatrix(A, At, m, n); // swapRows lavora sulle righe, quindi per invertire le colonne bisogna lavorare sulla trasposta
                Tt=traspMatrix(T, Tt, n, n); // invertire le colonne è un' operazione che lavora su T
                for (int j=k; j<(n-1); j++) {
                        swapRows(At, j, j+1, m); // inverte le righe j e j+1 di una matrice
                        swapRows(Tt, j, j+1, n);
                }
                A=traspMatrix(At, A, n, m); // trasposta di trasposta per tornare nella forma originaria
                T=traspMatrix(Tt, T, n, n);
        }
        return;
}

// invertire i segni di una riga della matrice
void switchSigns (int **A, int **mat, int c, int m, int n, int k) {
        for (int i=k; i<m; i++) {
                if (A[i][k]<0) {
                        for (int j=k; j<n; j++) {
                                A[i][j]=-A[i][j]; // matrice A
                        }
                        for (int l=0; l<c; l++) {
                                mat[i][l]=-mat[i][l]; // matrice S o T a seconda dei casi
                        }
                }
        }
        return;
}
```

```

// scambaire le righe row1 e row2 di una matrice
void swapRows (int **mat, int row1, int row2, int n) {
	int temp;
	for (int i=0; i<n; i++) {
		temp=mat[row1][i];
		mat[row1][i]=mat[row2][i];
		mat[row2][i]=temp;
	}
	return;
}

// aggiungere alla riga row2 un multiplo (mult) della riga row1
void addRows (int **mat, int row1, int row2, int mult, int n) {
	for (int j=0; j<n; j++) {
		mat[row2][j]=mat[row2][j]-mult*mat[row1][j];
	}
	return;
}

// portare il minimo della colonna k di una matrice in posizione [k][k]
void minUp (int **A, int **mat, int c, int m, int n, int k) {
	int min=A[k][k], r_min=k;
	for(int i=k; i<m; i++) {
		if (A[i][k]<min && A[i][k]!=0 || min==0) {
			min=A[i][k]; // individuare il minimo
			r_min=i; // individuare la riga del minimo
		}
	}
	swapRows(A, k, r_min, n); // invertire la riga k e r_min della matrice per portare il minimo in alto
	swapRows(mat, k, r_min, c); // stesso lavoro anche su S o T a seconda dei casi
	return;
}

// divisione con i resti
void div_rem (int **A, int **mat, int c, int m, int n, int k) {
	int quot=0;
	for (int i=k+1; i<m; i++) {
		quot=A[i][k]/A[k][k]; // individuare il quziente della divisione
		addRows(A, k, i, quot, n); // modificare le righe di A (inserendo i resti)
		addRows(mat, k, i, quot, c); // stesso lavoro su S o T a seconda dei casi
	}
	return;
}

// verificare se la colonna è della forma [0, ..., 0, d, 0, ..., 0] con d in posizione k
bool nullVector (int **mat, int m, int k) {
	for (int i=0; i<m; i++) {
		if (i!=k && mat[i][k]!=0) {
			return false;
		}
	}
	return true;
}

// prima forma di Smith che porta a un matrice diagonale ma i cui elementi in diagonale non necessariamente sono multipli
void Smith_fst (int **A, int **S, int **T, int **At, int **Tt, int m, int n, int k) {
	while(!nullVector(A, m, k) || !nullVector(traspMatrix(A, At, m, n), n, k)) { //lavoro sulla riga e colonna k fino a che entrambe non siano contemporaneamente della forma [d, 0, ..., 0]
		switchZeros(A, T, At, Tt, m, n, k); // per prima cosa sposto l'eventuale colonna nulla
		while (!nullVector(A, m, k)) { // comincio a lavorare sulle colonne
```

```
            switchSigns(A, S, m ,m, n, k); // cambio i segni della colonna k
            while (!nullVector(A, m, k)) { // finchè la colonna non è della forma corretta applico a ripetizione le seguenti funzioni
                minUp(A, S, m, m, n, k); // porto il minimo in alto
                div_rem(A, S, m, m, n, k); // divisione con i resti
            }
        }
        // per poter lavorare sulle righe traspongo la matrice e ripeto lo stesso procedimento fatto per la colonna
        At=traspMatrix(A, At, m, n);
        Tt=traspMatrix(T, Tt, n, n);
        while (!nullVector(At, n, k)) {
            switchSigns(At, Tt, n, n, m, k);
            while (!nullVector(At, n, k)) {
                minUp(At, Tt, n, n, m, k);
                div_rem(At, Tt, n, n, m, k);
            }
        }
        A=traspMatrix(At, A, n, m);
        T=traspMatrix(Tt, T, n, n);
    }
    switchSigns(A, S, m, m, n, k); // cambio il segno dell'ultimo elemento in diagonale nel caso sia negativo
    return;
}

// individuo il min tra due interi
int my_min (int m, int n) {
    if (m<n) {
        return m;
    } else {
        return n;
    }
}

// verifico che d(k) divida tutti gli elementi diagonali successivi, ossia d(k)|d(k+1), d(k)|d(k+2), ... , d(k)|d(min(m, n))
bool bool_dk (int **A, int r, int k) { // r=min(m, n) e k è l'indice dell'elemento diagonale che sto verificando
    int quot=0, rem=0;
    for (int i=k+1; i<r; i++) { // scorro i da k+1 ad r
        rem=A[i][i]%A[k][k];
        if(rem!=0) { // se il resto della divisione non è zero allora esco dalla verifica con false perchè d(k) non divide d(i) per almeno un indice i>k
            return false;
        }
    }
    return true;
}

// verifico che la diagonale non abbia zeri fuori posto
int checkDiagZeros (int **A, int r) {
    for (int i=0; i<r; i++) {
        if (A[i][i]==0) { // appena trovo un elemento diagonale nullo verifico i successivi
            for (int j=i; j<r; j++) {
                if (A[j][j]!=0) { // appena ne trovo uno dopo non nullo ritorno la riga i in cui ho uno zero fuori posto (la verifica è fallita)
                    return i;
                }
            }
            return r; // se invece a fine del ciclo sui successivi sono tutti nulli ritorno r, senza bisogno di proseguire con il ciclo for iniziale tanto i successivi sono già stati verificati (la verifica ha avuto successo)
```

```
			}
		}
		return r;
}

// forma normale di Smith
void SmithNormalForm (int **A, int **S, int **T, int m, int n) {
		int **At=NULL;
		At=input_null(At, n, m);
		int **Tt=NULL;
		Tt=input_null(Tt, n, n);

		int r=my_min(m, n);

		for (int k=0; k!=r; k++) {
			Smith_fst(A, S, T, At, Tt, m, n, k); // prima raggiungo la forma diagonale
		}

		// verifico che la forma diagonale raggiunta abbia gli zeri tutti in fondo altrimenti riapplico smith_fst dopo aver fatto delle modifiche
		int err=checkDiagZeros(A, r);
		if (err!=r) { // se il check è fallito, ossia ci sono degli zeri fuori posto
			for (int j=err+1; j<r; j++) { // sommo alla riga err (con lo zero fuori posto) le righe successive j
				addRows(A, j, err, -1, n);
				addRows(S, j, err, -1, m);
			}

			for (int k=err; k<r; k++) {
				Smith_fst(A, S, T, At, Tt, m, n, k); // prima raggiungo la forma diagonale
			}
		}

		// una volta raggiunta una forma diagonale con gli zeri in fondo verifico che la forma sia effettivamente di Smith
		for (int it=0; it<r; it++) {
			if (A[it][it]==0) { // se l'elemento diagonale è nullo lo sono anche i successivi (per quanto fatto da checkDiagZeros e la riapplicazione di Smith_fst)
				break; // quindi posso uscire dal for perchè la verifica è completa
			}
			if (!bool_dk(A, r, it)) { // nel caso non sia di Smith
				for (int i=it+1; i<r; i++) { // sommo alla riga it tutte le righe successive i
					addRows(A, i, it, -1, n);
					addRows(S, i, it, -1, m);
				}
				for (int iter=it; iter<r; iter++) {
					Smith_fst(A, S, T, At, Tt, m, n, iter); // e riapplico Smith (dalla riga it in poi) così da ottenere in alto il mcm dei d_i
				}
			}
		}
		switchSigns(A, S, m, m, n, r-1); // cambio eventualmente il segno dell'ultimo elemento in diagonale
		return;
}

// moltiplicare due matrici: C=A*B
void multiplyMatrices(int **A, int **B, int **C, int m, int n, int p) {
		// Itera sulle righe di A
		for (int i=0; i<m; i++) {
			// Itera sulle colonne di B
			for (int j=0; j<p; j++) {
				C[i][j] = 0;
```

```
                        // Calcola il prodotto scalare tra la riga i di A e la colonna j di B
                        for (int k=0; k<n; k++) {
                                C[i][j]=C[i][j]+A[i][k]*B[k][j];
                        }
                }
        }
}

// forma normale di Smith senza ricordare le matrici S e T
void Smith_D (int **A, int m, int n) {
        int **S=NULL;
        S=input_id(S, m);
        int **T=NULL;
        T=input_id(T, n);
        int **At=NULL;
        At=input_null(At, n, m);
        int **Tt=NULL;
        Tt=input_null(Tt, n, n);

        SmithNormalForm(A, S, T, m, n);
        return;
}

// questa funzione produce la forma di Smith in maniera totalmente analoga alla funzione SmithNormalForm, l'unica differenza è che produce anche le matrici S_inv e T_inv tali per cui S_inv*D*T_inv=A (di seguito andiamo quindi a commentare solo le differenze rispetto alla funzione precedente)
void SmithNormalForm5mat (int **A, int **S, int **T, int **S_inv, int **T_inv, int m, int n) {
        int **At=NULL;
        At=input_null(At, n, m);
        int **Tt=NULL;
        Tt=input_null(Tt, n, n);
        int **SinvT=NULL;
        SinvT=input_null(SinvT, m, m);

        int r=my_min(m, n);

        for (int k=0; k<r; k++) {
                Smith_fst5mat(A, S, T, S_inv, T_inv, At, Tt, SinvT, m, n, k);
        }

        int err=checkDiagZeros(A, r);
        if (err!=r) {
                SinvT=traspMatrix(S_inv, SinvT, m, m);
                for (int j=err+1; j<r; j++) {
                        addRows(A, j, err, -1, n);
                        addRows(S, j, err, -1, m);
                        addRows(SinvT, err, j, 1, m); // presa S1 e S2, mentre S=S2*S1, S_inv=S1_inv*S2_inv quindi così come per le T per moltiplicare a destra usavo la trasposta lo stesso faccio con S_inv (tenendo conto che l'inversa fa i passaggi al contrario, quindi anzichè -1 metto 1 e inverto it e i)
                }
                S_inv=traspMatrix(SinvT, S_inv, m, m);
                for (int k=err; k<r; k++) {
                        Smith_fst5mat(A, S, T, S_inv, T_inv, At, Tt, SinvT, m, n, k);
                }
        }

        for (int it=0; it<r; it++) {
                if (A[it][it]==0) {
                        break;
                }
                if (!bool_dk(A, r, it)) {
                        SinvT=traspMatrix(S_inv, SinvT, m, m);
                        for (int i=it+1; i<r; i++) {
```

```
				addRows(A, i, it, -1, n);
				addRows(S, i, it, -1, m);
				addRows(SinvT, it, i, 1, m);
			}
			S_inv=traspMatrix(SinvT, S_inv, m, m);
			for (int iter=it; iter!=r; iter++) {
				Smith_fst5mat(A, S, T, S_inv, T_inv, At, Tt, SinvT, m, n, iter);
			}
		}
	}
	SinvT=traspMatrix(S_inv, SinvT, m, m);
	switchSigns5mat(A, S, SinvT, m, m, n, r-1);
	S_inv=traspMatrix(SinvT, S_inv, m, m);
	return;
}

// come nel caso precedente, questa funzione è analoga a Smith_fst con l'aggiunta di S_inv e T_inv
void Smith_fst5mat (int **A, int **S, int **T, int **S_inv, int **T_inv, int **At, int **Tt, int **SinvT, int m, int n, int k) {
	while(!nullVector(A, m, k) || !nullVector(traspMatrix(A, At, m, n), n, k)) {
		switchZeros5mat(A, T, T_inv, At, Tt, m, n, k);
		SinvT=traspMatrix(S_inv, SinvT, m, m);
		while (!nullVector(A, m, k)) {
			switchSigns5mat(A, S, SinvT, m, m, n, k);
			while (!nullVector(A, m, k)) {
				minUp5mat(A, S, SinvT, m, m, n, k);
				div_rem5mat(A, S, SinvT, m, m, n, k);
			}
		}
		S_inv=traspMatrix(SinvT, S_inv, m, m);
		At=traspMatrix(A, At, m, n);
		Tt=traspMatrix(T, Tt, n, n);
		while (!nullVector(At, n, k)) {
			switchSigns5mat(At, Tt, T_inv, n, n, m, k);
			while (!nullVector(At, n, k)) {
				minUp5mat(At, Tt, T_inv, n, n, m, k);
				div_rem5mat(At, Tt, T_inv, n, n, m, k);
			}
		}
		A=traspMatrix(At, A, n, m);
		T=traspMatrix(Tt, T, n, n);
	}
	SinvT=traspMatrix(S_inv, SinvT, m, m);
	switchSigns5mat(A, S, SinvT, m, m, n, k);
	S_inv=traspMatrix(SinvT, S_inv, m, m);
	return;
}

// funzione analoga a switchZeros con l'aggiunta di S_inv e T_inv
void switchZeros5mat (int **A, int **T, int **T_inv, int **At, int **Tt, int m, int n, int k) {
	bool var=true;
	for (int i=k; i<m; i++) {
		if (A[i][k]!=0) {
			var=false;
			break;
		}
	}
	if (var) {
		At=traspMatrix(A, At, m, n);
		Tt=traspMatrix(T, Tt, n, n);
		for (int j=k; j<(n-1); j++) {
			swapRows(At, j, j+1, m);
			swapRows(Tt, j, j+1, n);
			swapRows(T_inv, j, j+1, n);
		}
		A=traspMatrix(At, A, n, m);
```

```
                T=traspMatrix(Tt, T, n, n);
        }
        return;
}

// funzione analoga a switchSigns con l'aggiunta di S_inv e T_inv
void switchSigns5mat (int **A, int **mat, int **mat2, int c, int m, int n, int k) {
        for (int i=k; i<m; i++) {
                if (A[i][k]<0) {
                        for (int j=k; j<n; j++) {
                                A[i][j]=-A[i][j];
                        }
                        for (int l=0; l<c; l++) {
                                mat[i][l]=-mat[i][l];
                                mat2[i][l]=-mat2[i][l]; // matrice S_inv o T_inv a seconda dei casi
                        }
                }
        }
        return;
}

// funzione analoga a minUp con l'aggiunta di S_inv e T_inv
void minUp5mat (int **A, int **mat, int **mat2, int c, int m, int n, int k) {
        int min=A[k][k], r_min=k;
        for(int i=k; i<m; i++) {
                if (A[i][k]<min && A[i][k]!=0 || min==0) {
                        min=A[i][k];
                        r_min=i;
                }
        }
        swapRows(A, k, r_min, n);
        swapRows(mat, k, r_min, c);
        swapRows(mat2, k, r_min, c); // stesso lavoro anche su S_inv o T_inv a seconda dei casi

        return;
}

// funzione analoga a div_rem con l'aggiunta di S_inv e T_inv
void div_rem5mat (int **A, int **mat, int **mat2, int c, int m, int n, int k) {
        int quot=0;
        for (int i=k+1; i<m; i++) {
                quot=A[i][k]/A[k][k];
                addRows(A, k, i, quot, n);
                addRows(mat, k, i, quot, c);
                addRows(mat2, i, k, -quot, c); // stesso lavoro su S_inv o T_inv a seconda dei casi (l'inversa fa i passaggi al contrario)
        }
        return;
}

#endif
```

Il vantaggio di questa scomposizione è che determina l'omologia del complesso simpliciale associato anche se come anello commutativo si prende $\mathbb{Z}$:

$$H_n \cong \mathbb{Z}^r \oplus \mathbb{Z}/(d_1) \ \oplus \ \dots \ \oplus \mathbb{Z}/(d_k) \ ,$$

dove r è un numero da determinare e gli ideali che quozientano sono generati dagli elementi diagonali non nulli della matrice D della forma di Smith.

1.4 Categorie, funtori e moduli di persistenza

Nello studio dei dati che ci siamo posti ad inizio corso molto spesso bisogna tenere conto che le informazioni sono variabili nel tempo (aumentano i dati, si modificano nel tempo etc.). Per studiare questa variante più formalmente si introducono i concetti di **categoria**, ovvero un insieme di oggetti (nel nostro caso specifico complessi simpliciali, complessi di moduli o spazi vettoriali), e di **funtore**, una applicazione $F : C \to D$ tra due categorie con le seguenti proprietà:

1. $\forall f : X \to Y$ in C $F(f) : F(X) \to F(Y)$ in D
2. $F(Id_X) = Id_{F(X)}$
3. $F(f \circ g) = F(f) \circ F(g)$

Nota: H_k è un funtore dalla categoria dei complessi di moduli alla categoria dei moduli.

Come categoria di partenza consideriamo ora un insieme di tempi $\{t_0, t_1, ..., t_n\}$ con l'ordinamento totale determinato dal $\leq$ e come categoria di arrivo l'insieme degli spazi vettoriali; una struttura del genere si definisce **modulo di persistenza**. Questo permette di creare la seguente catena di relazioni:

$$(\{t_0, ..., t_n\}, \leq) \to \textit{insiemi simpliciali} \xrightarrow{C_\cdot(\cdot\ ,\ \mathbb{Q})} \textit{complessi di moduli} \xrightarrow{H_\cdot} \textit{spazi vettoriali}\ ,$$

ovvero come variano nel tempo i dati forniti.
Poichè gli spazi vettoriali sono univocamente determinati (a meno di isomorfismi) dalla loro dimensione è nuovamente possibile passare alle matrici associate. Sfruttando tutto ciò che stato detto finora concludiamo mostrando un algoritmo che crea il **barcode** di una omologia partendo da un complesso simpliciale determinato dalla matrice di adiacenza variabile lungo l'insieme ordinato di tempi $\{t_0 \leq t_1 \leq\ ...\ \leq t_n\}$. L'output rappresenta, tramite linee di diversa lunghezza, il tempo in cui l'omologia presa in analisi persiste (nel caso di H_0 mostreranno per quanto tempo ogni vertice resta una componente connessa prima di far parte di un 1-simplesso) sfruttando le matrici β e μ, dove

$$\beta_{i,j} = rg(\varphi_{i \to j})\ ,$$

$$\mu_{i,j} = \beta_{i,j+1} + \beta_{i-1,j} - \beta_{i,j} - \beta_{i-1,j-1}\ .$$

Listing 8: Barcode.h

```
#pragma once
#ifndef Barcode
        #define Barcode
        #include "Complesso_da_1_simplessi.h"
        #include "Smith.h"
        #include "Matrix.h"

        /*Come creiamo il barcode
        come prima cosa creiamo il modulo di persistenza ovvero la collezione dell' evoluzione dei nostri complessi simplciali
        a questo punto ci chiediamo di quale H vogliamo calcolare il barcode e calcoliamo le rispettive matrici di bordo
        poi calcoliamo una base del nucleo
        applichiamo la trasformazione da H a H+t
        calcoliamo il rango di questa applicazione fusa con la matrice di bordo
        e sottraiamo il rango della matrice di bordo il risultato è l'elemento per il resto dobbiamo avere che j >= i e calcoliamo tutti gli altri elementi*/


        //Definizione modulo di persistenza

        /*Definisco l'evoluzione delle matrici in base al cambiamento delle distanze
        ovvero mi salvo una lista di matrici di zero e 1 che sono le matrici di adiacenza */
        typedef struct Matrici_Persistenza {
```

```
                double l_min;
                double l_max;
                int size;
                int** matrix_d;
                struct Matrici_Persistenza* next;
                struct Matrici_Persistenza* prev;
        } matrici_persistenza;

        // Mi creo una lista di complessi simpliciali che rappresentano l'evoluzione
        typedef struct Modulo_Persistenza {
                double l_min;
                double l_max;
                int size;
                SimplicialComplex* sc;
                struct Modulo_Persistenza* next;
                struct Modulo_Persistenza* prev;
        }M_P;

        //matrici_persistenza* create_M_P(void);
        M_P* create_Modulo_Persistenza(matrici_persistenza*, int);
        int** matrix_phi_l1_to_l2(Simplex*, int, Simplex*, int, int);
        int beta_i_j(M_P*, double, double, int, int);
        int** beta_matrix(int, int, double, M_P*, int, int);
        int** mu_matrix(int**, int);
        double** distance_matrix(int**, int);
        int** input_point(int*);
        int** zero_one_matrix(double**, int, double);
        double* find_lambda_value(double**, int);
        matrici_persistenza* create_matrix_persistenza(double**, int, double*);
        int compare_double(const void*, const void*);
        void print1(M_P*, int);

        /* Esempio non da input
        matrici_persistenza* create_M_P(void) {
                matrici_persistenza* mp1 = (matrici_persistenza*)malloc(sizeof(matrici_persistenza));
                matrici_persistenza* mp2 = (matrici_persistenza*)malloc(sizeof(matrici_persistenza));
                matrici_persistenza* mp3 = (matrici_persistenza*)malloc(sizeof(matrici_persistenza));
                matrici_persistenza* mp4 = (matrici_persistenza*)malloc(sizeof(matrici_persistenza));
                mp1->l_min = 0.5;
                mp1->l_max = 1;
                mp2->l_min = 1.5;
                mp2->l_max = 10;
                mp3->l_min = 10.5;
                mp3->l_max = 10.5;
                mp4->l_min = 11;
                mp4->l_max = 11; // per segnalare la fine
                mp1->size = 3;
                mp2->size = 3;
                mp3->size = 3;
                mp4->size = 3;
                int** m1 = NULL, ** m2 = NULL, ** m3 = NULL, ** m4 = NULL;
                m1 = input_null(m1, 3, 3);
                m2 = input_null(m2, 3, 3);
                m3 = input_null(m3, 3, 3);
                m4 = input_null(m4, 3, 3);
                m1[0][0] = 1; m1[0][1] = 0; m1[0][2] = 0;
                m1[1][0] = 0; m1[1][1] = 1; m1[1][2] = 0;
                m1[2][0] = 0; m1[2][1] = 0; m1[2][2] = 1;
                m2[0][0] = 1; m2[0][1] = 1; m2[0][2] = 0;
                m2[1][0] = 1; m2[1][1] = 1; m2[1][2] = 0;
                m2[2][0] = 0; m2[2][1] = 0; m2[2][2] = 1;
                m3[0][0] = 1; m3[0][1] = 1; m3[0][2] = 0;
                m3[1][0] = 1; m3[1][1] = 1; m3[1][2] = 1;
                m3[2][0] = 0; m3[2][1] = 1; m3[2][2] = 1;
                m4[0][0] = 1; m4[0][1] = 1; m4[0][2] = 1;
```

```
		m4[1][0] = 1; m4[1][1] = 1; m4[1][2] = 1;
		m4[2][0] = 1; m4[2][1] = 1; m4[2][2] = 1;
		mp1->matrix_d = m1;
		mp2->matrix_d = m2;
		mp3->matrix_d = m3;
		mp4->matrix_d = m4;
		mp1->next = mp2;
		mp2->next = mp3;
		mp3->next = mp4;
		mp4->next = NULL;
		mp1->prev = NULL;
		mp2->prev = mp1;
		mp3->prev = mp2;
		mp4->prev = mp3;
		return mp1;
	}*/

	/*Data la successione di matrici creo da quelle i complessi simpliciali
	salvandomi ogni volta il lambda min e max in cui quel complesso non varia
	utilizzo la fomra troncata per risparmiare memoria e per alcuni accorgimenti detti all'utente*/
	M_P* create_Modulo_Persistenza(matrici_persistenza* mp, int max) {
		M_P* mod_p = NULL, * app = NULL, * my_new = NULL;
		int size = mp->size;
		mod_p = (M_P*)malloc(sizeof(M_P));
		mod_p->l_min = mp->l_min;
		mod_p->l_max = mp->l_max;
		mod_p->size = size;
		mod_p->sc = complex_from_adjacency_matrix_truncated(mp->matrix_d, size, max);
		mod_p->prev = NULL;
		app = mod_p;

		while (mp->next) {
			mp = mp->next;
			size = mp->size;
			my_new = (M_P*)malloc(sizeof(M_P));
			my_new->l_min = mp->l_min;
			my_new->l_max = mp->l_max;
			my_new->sc = complex_from_adjacency_matrix_truncated(mp->matrix_d, size, max);

			my_new->prev = mod_p;
			my_new->next = NULL;
			mod_p->next = my_new;
			mod_p = my_new;
		}

		return app;
	}

	//Mi costruisco la matrice di cambio di base da s1 a s2 evoluzione dei simplessi
	int** matrix_phi_l1_to_l2(Simplex* s1, int sizeb1, Simplex* s2, int sizeb2, int size) {
		int** matrix = NULL;
		matrix = input_null(matrix, sizeb2, sizeb1);
		Simplex* current = s1;
		int col = 0;
		//scorro il primo simplesso che è contenuto nel secondo per costruzione e mi salvo il numero
		//corrispondente alla posizione e creo la matrice di cambiamento di base
		while (current)
		{
			int* v = current->vertices;
			int k = base_number(s2, v, size);
			if (k != -1) {
				matrix[k][col] = 1;
			}
```

```
            current = current->next;
            col++;
        }

        return matrix;
    }

    //calcolo l'elemento ij della matrice beta sempre i > j
    //n è il grado del complesso i e j l'evoluzione al tempo i e j
    //h l'omologia cercata
    int beta_i_j(M_P* mp, double i, double j, int h, int n) {
        SimplicialComplex* sc1 = NULL, * sc2 = NULL;
        M_P* app = mp;

        if (i < app->l_min)
            return 0;
        if(h > n)
            return 0;
        //come prima cosa trovo a quale complessi simplicale appartengono le evoluzioni i e j
        while (app && !sc2) {
            if (app->l_min <= i && i <= app->l_max) {
                sc1 = app->sc;
            }

            if (app->l_min <= j && j <= app->l_max) {
                sc2 = app->sc;

            }
            app = app->next;
        }
        //se trovo il secondo anche il primo esiste per costruzione altrimenti torno 0
        if (!sc2)
            return 0;

        //se il primo ha dimensione nulla torno zero perchè vuole dire che non ci sono evoluzioni
        if (sc1[h].size == 0)
            return 0;

        //printf("sc1[h].size = %d\n", sc1[h].size);

        //se tutto è andato bene calcolo il rango dell'omologia tramite phi
        //per prima cosa trovo la matrice di cambio di base
        int** phi_i_j = matrix_phi_l1_to_l2(sc1[h].simplices, sc1[h].size, sc2[h].simplices, sc2[h].size, h + 1);
        //print_matrix(phi_i_j, sc2[h].size, sc1[h].size);

        int** edge1 = NULL;
        int** kernel_edge1 = NULL;
        int colonne_base_rango = 0;
        //mi devo calcolare la matrice di bordo del complesso simpliciale i
        // tra i simplessi h e h -1

        //quindi se h = 0 la matrice è nulla altrimenti la calcolo e calcolo una base del nucleo
        if (h == 0) {
            //edge1 = input_id(edge1, sc1[h].size);
            edge1 = input_null(edge1, sc1[h].size, sc1[h].size);
            //print_matrix(edge1, sc1[h].size, sc1[h].size);
            kernel_edge1 = input_id(kernel_edge1, sc1[h].size);
            colonne_base_rango = sc1[h].size;
        }
        else {
            edge1 = edge_Matrix(sc1, h);
            kernel_edge1 = kernel_base(edge1, sc1[h - 1].size, sc1[h].size, &colonne_base_rango);
        }

```

```
			//se il nucleo è nullo bij = 0 perchè non ho simplessi da dove sono partito
			if (!kernel_edge1)
				return 0;

			//altrimenti applico il cambiamento di base al nucleo
			int** matrix_j = NULL;
			matrix_j = mul_matrix(phi_i_j, sc2[h].size, sc1[h].size, kernel_edge1, sc1[h].size, colonne_base_rango);

			//non resta che applicare f e calolcare i rispettivi ranghi
			int** edge2 = NULL;
			int rank2 = 0, row = 0, col = 0;

			//se h è il massimo allora la matrice di bordo di j è nulla quindi f lo è
			// oppure se è nullo lo spazio di partenza
			//altrimenti la calcolo
			if (h == n)
			{
				edge2 = input_null(edge2, sc2[h].size, sc2[h].size);
				row = sc2[h].size;
				col = sc2[h].size;
				rank2 = 0;
			}
			else if (!sc2[h + 1].size || sc2[h + 1].size == 0) {

				edge2 = input_null(edge2, sc2[h].size, sc2[h].size);
				row = sc2[h].size;
				col = sc2[h].size;
				//rank2 = rank_matrix(edge2, sc2[h].size, sc2[h + 1].size + 1);
				rank2 = 0;

			}
			else {
				edge2 = edge_Matrix(sc2, h + 1);
				rank2 = rank_matrix(edge2, sc2[h].size, sc2[h + 1].size);
				row = sc2[h].size;
				col = sc2[h + 1].size;
			}

			//applico f mettendo le due matrici una di seguito all'altra
			int** matrix_f = NULL;
			matrix_f = link2matrix_same_row(matrix_j, sc2[h].size, colonne_base_rango, edge2, row, col);

			//ne calcolo il rango
			int rank1 = rank_matrix(matrix_f, sc2[h].size, col + colonne_base_rango);

			//il rango dell'omologia è la differenza tra il rango della matrice con f applicata
			//e il rango di f
			int b = rank1 - rank2;

			return b;

		}

		//stampo il modulo di persistenza
		void print1(M_P* a, int n) {
			M_P* mp = a;
			while (mp) {
				printComplex(mp->sc, n);
				mp = mp->next;
			}
		}

		//calcolo la matrice beta elemento a elemnto ma solo con j >= i
```

```
        // k è il grado del complesso e h è l'omologia richiesta
        int** beta_matrix(int min, int max, double passo, M_P* mp, int h, int k) {
                int** matrix = NULL;
                //numero di passi da calcolare in base al lambda massimo e minimo
                int n = ((max - min) / passo) + 1;
                matrix = input_null(matrix, n + 1, n + 1);
                int i = 0;
                int j = 0;

                for (i = 0; i <= n; i++) {
                        for (j = i; j <= n; j++) {
                                matrix[i][j] = beta_i_j(mp, min + i * passo, min +
                                    j * passo, h, k);
                        }
                }
                return matrix;
        }

        //mu è una combinazione lineare delle entrate di beta
        int** mu_matrix(int** beta, int n) {
                int i = 0, j = 0;
                int** matrix = NULL;
                matrix = input_null(matrix, n , n);
                for (i = 1; i < n; i++)
                        for (j = i + 1; j < n; j++)
                                matrix[i][j] = beta[i][j - 1] - beta[i][j] + beta[i
                                     - 1][j] - beta[i - 1][j - 1];
                return matrix;
        }

        //calcolo la matrice delle distanze tra i punti nel piano cartesiano
        double** distance_matrix(int** point, int n) {
                int i = 0, j = 0;
                double** matrix = NULL;
                matrix = (double**)calloc(n, sizeof(double*));
                for (i = 0; i < n; i++) {
                        matrix[i] = (double*)calloc(n, sizeof(double));
                }
                for (i = 0; i < n; i++) {
                        for (j = i; j < n; j++) {
                                matrix[i][j] = sqrt(pow(point[i][0] - point[j][0],
                                    2) + pow(point[i][1] - point[j][1], 2));
                                matrix[j][i] = matrix[i][j];
                        }
                }
                return matrix;
        }

        //richiede i punti da input
        int** input_point(int *k) {
                int** matrix = NULL;
                int n = 0;
                printf("Inserisci il numero di punti: n = ");
                scanf("%d", &n);
                *k = n;
                matrix = input_null(matrix, n, 2);
                int i = 0;
                for (i = 0; i < n; i++) {
                        printf("Inserisci le coordinate del punto %d: ", i + 1);
                        scanf("%d %d", &matrix[i][0], &matrix[i][1]);
                }

        return matrix;
        }

        //calcola le matrici 0-1 partendo dalla matrice delle distanze e un valore
            lambda
        int** zero_one_matrix(double** matrix, int n, double lambda) {
                int** m = NULL;
                m = input_null(m, n, n);
```

```
		int i = 0, j = 0;
		for (i = 0; i < n; i++) {
			for (j = i; j < n; j++) {
				if (matrix[i][j] < lambda) {
					m[i][j] = 1;
					m[j][i] = 1;
				}
				else
				{
					m[i][j] = 0;
					m[j][i] = 0;
				}
			}
		}

		return m;
	}

	//data la matrice trovo i valori di lambda estremeali
	//per cui in ogni intervallo trovato non cambia il complesso
	double* find_lambda_value(double** matrix, int n) {
		int size = n * (n - 1) / 2;
		double* lambda = (double*)calloc(size, sizeof(double));
		int i = 0, j = 0, k = 0;

		//mi salvo tutti gli elementi sopra la diagonale della matrice delle distanze
		for (i = 0; i < n; i++)
			for (j = i + 1; j < n; j++) {
				lambda[k] = matrix[i][j];
				k++;
			}



		double l = 0;
		double app = 0;
		int iter = 0;

		//ordino le distanze
		qsort(lambda, size, sizeof(double), compare_double);



		//creo un vettore con gli estremi degli intervalli andando a leggere le distanze
		while (iter < size) {
			l += 0.5;

			if (lambda[iter] == app) {
				lambda[iter] = 0;
				iter++;
				l -= 0.5;
				continue;
			}
			else if(l > lambda[iter])
			{
				app = lambda[iter];
				lambda[iter] = l;
				iter++;
			}
		}

		/*printf("lambda\n");
		for (int i = 0; i < size; i++)
			printf("%lf ", lambda[i]);*/
		return lambda;
	}

	//data la matrice e gli intervallo creo la successione di amtrici di
```

```c
adiacenza
matrici_persistenza* create_matrix_persistenza(double** matrix, int n, double* l) {
	matrici_persistenza* mp = NULL, * app = NULL, * my_new = NULL;
	int size = (n*(n-1)/2);
	printf("size = %d", size);
	double* lambda = find_lambda_value(matrix, n);
	printf("lambda\n");
	for (int i = 0; i < size; i++)
		printf("%lf ", lambda[i]);
	//dentro il vettore ci sono zeri che salto
	int i = 0;
	while (lambda[i] == 0)
		i++;

	//creo quando lambda è non nullo la matrice di adiacenza
	//e mi salvo l'intervallo
	//per comodità divido il perimo e l'ultimo elemnto della lista
	//poichè hanno valori di minimo e massimo predefiniti
	mp = (matrici_persistenza*)malloc(sizeof(matrici_persistenza));
	mp->l_min = 0.5;
	mp->l_max = lambda[i]-0.5;
	printf("\n\n\n\n Il codice inzia qui");
	printf("l_max = %lf", mp->l_max);
	printf("l_min = %lf", mp->l_min);
	mp->size = n;
	mp->matrix_d = zero_one_matrix(matrix, n, mp->l_max);
	printf("\n\n\n");
	print_matrix(mp->matrix_d, n, n);

	mp->prev = NULL;
	mp->next = NULL;
	app = mp;
	i++;

	while (i < size)
	{
		my_new = (matrici_persistenza*)malloc(sizeof(matrici_persistenza));
		my_new->l_min = mp->l_max + 0.5;
		while (lambda[i] == 0 && i < size)
		{
				i++;
		}
		if (i < size) {
			my_new->l_max = lambda[i] - 0.5;
			printf("l_max = %lf\n", my_new->l_max);
			my_new->size = n;
			my_new->matrix_d = zero_one_matrix(matrix, n, my_new->l_max);
			printf("\n\n\n");
			print_matrix(my_new->matrix_d, n, n);
			my_new->prev = mp;
			my_new->next = NULL;
			mp->next = my_new;
			mp = my_new;
			i++;
			printf("l_min = %lf\n", my_new->l_min);
			printf("l_max = %lf\n", my_new->l_max);
		}

	}

	my_new = (matrici_persistenza*)malloc(sizeof(matrici_persistenza));
	my_new->l_min = mp->l_max + 0.5;

	my_new->l_max = my_new->l_min;
	my_new->size = n;
	printf("l_max = %lf\n", my_new->l_min);
	my_new->matrix_d = zero_one_matrix(matrix, n, my_new->l_max);
```

```
				printf("Matrice n");
				print_matrix(my_new->matrix_d, n, n);
				my_new->prev = mp;
				my_new->next = NULL;
				mp->next = my_new;
				mp = my_new;
				l[0] = mp->l_max;

				return app;
		}

		//funzione per comparare due double usata successivamente
		int compare_double(const void* a, const void* b) {
				double da = *(const double*)a;
				double db = *(const double*)b;

				if (da < db) return -1;
				if (da > db) return 1;
				return 0;
		}

#endif
```

Listing 9: Grafico Barcode

```
import sys
import json
import matplotlib.pyplot as plt


"""print("Argomenti ricevuti:", sys.argv)
"""
def plot_barcode(matrix, lambda_min, h):
    """
    Data una matrice quadrata (lista di liste) che rappresenta
    il modulo di persistenza, estrae gli intervalli e disegna il barcode.

    Regole:
      - Per ogni riga non nulla (con almeno un valore non zero):
          * Si parte dalla riga (birth = numero della riga, 1-indexed).
          * Si individuano gli elementi non nulli, a partire dall'elemento più a destra
            e poi procedendo verso sinistra.
          * Per ciascun elemento non nullo, lâ€™intervallo è [birth, death] dove:
              - birth = numero della riga (1-indexed);
              - death = numero della colonna (1-indexed).
      - Ogni intervallo viene disegnato come segmento orizzontale e posizionato sulla "prima quota libera"
        (cioè, sul livello y più basso in cui non si sovrappone ad un altro intervallo).
    """
    n = len(matrix)  # dimensione della matrice
    intervals = []   # lista di intervalli: ciascuno è una tupla (birth, death)

    # Scorriamo le righe (i indici partono da 0; aggiungiamo 1 per ottenere la 1-indexazione)
    for i, row in enumerate(matrix):
        if any(val != 0 for val in row):
            # Troviamo gli indici degli elementi non nulli e li ordiniamo decrescentemente (da destra a sinistra)
            nonzero_indices = [j for j, val in enumerate(row) if val != 0]
            nonzero_indices.sort(reverse=True)

            birth = lambda_min + i*h   # riga corrente in 1-indexing
```

```
            for j in nonzero_indices:
                death = lambda_min + j*h  # colonna in 1-indexing
                num_segments = row[j]
                # Aggiungiamo tante istanze quanto il valore in cella
                for _ in range(num_segments):
                    intervals.append((birth, death))


    # Ordiniamo gli intervalli per birth (e per death in ordine decrescente se la riga è la stessa)
    intervals.sort(key=lambda x: (x[0], -x[1]))

    # Assegniamo la quota (livello) ad ogni intervallo
    levels = []  # levels[l] contiene il valore di death dell'ultimo intervallo assegnato a quel livello
    assigned_intervals = []  # lista di tuple (birth, death, livello)

    for birth, death in intervals:
        assigned_level = None
        # Cerchiamo il livello l più basso per cui l'intervallo corrente non si sovrappone
        for l, last_death in enumerate(levels):
            if last_death < birth:
                assigned_level = l
                break
        if assigned_level is None:
            assigned_level = len(levels)
            levels.append(0)
        levels[assigned_level] = death
        assigned_intervals.append((birth, death, assigned_level + 1))  # usiamo 1-indexazione per il livello

    # Disegniamo il barcode con matplotlib
    fig, ax = plt.subplots()
    for birth, death, level in assigned_intervals:
        ax.hlines(level, birth, death, colors='blue', lw=2)
        ax.plot([birth, death], [level, level], 'bo')

    ax.set_xlabel("x (indice)")
    ax.set_ylabel("Quota (livello)")
    ax.set_title("Barcode del modulo di persistenza")
    # Impostiamo i limiti dell'asse x:
    # da un po' sotto lambda_min fino a lambda_min + n*h (o anche oltre, se necessario)
    ax.set_xlim(lambda_min, lambda_min + (n-1) * h)

    # Impostiamo i tick delle ascisse: ogni riga corrisponde a un tick
    xticks = [lambda_min + i * h for i in range(n)]
    ax.set_xticks(xticks)
    ax.set_xticklabels([f"{x:.2f}" for x in xticks])
    max_level = max((lev for (_, _, lev) in assigned_intervals), default=1)
    ax.set_ylim(0, max_level + 1)

    plt.grid(axis='x', linestyle='--', alpha=0.6)
    plt.show()

def main():
    # Se viene passato un argomento, lo interpretiamo come una stringa JSON contenente la matrice
    if len(sys.argv) > 1:
        matrix_str = sys.argv[1]
        try:
            matrix = json.loads(matrix_str)
        except json.JSONDecodeError as e:
            print("Errore nella decodifica della matrice:", e)
            sys.exit(1)
        lambda_min = float(sys.argv[2])  # Valore minimo della scala delle ascisse
        h = float(sys.argv[3])  # Passo della scala delle ascisse
    else:
        # Matrice di default (se non viene passato alcun argomento)
```

```
        matrix = [
            [0, 0, 0, 0, 0],
            [0, 0, 0, 0, 0],
            [0, 0, 0, 1, 1],
            [0, 0, 0, 0, 0],
            [0, 0, 1, 1, 0]
        ]
    plot_barcode(matrix, lambda_min, h)

if __name__ == '__main__':
    main()
```

2 Algebra: crittografia

In questa parte del corso si analizzano i principali metodi crittografici sviluppati nel corso del tempo. Per capire a fondo se un codice rende sicuro o meno uno scambio di informazioni sono necessarie alcuni concetti algebrici che sono riportati nelle seguenti sezioni anche attraverso codici che implementano le definizioni astratte.

2.1 Ideali su $\mathbb{Z}$, MCD e identità di Bézout

Definizioni: Sia A un anello commutativo. $I \subset A$ si dice **ideale** se

1. $(I, +)$ *sottogruppo di* $(A, +)$
2. $i \in I \ , \ a \in A \Rightarrow ai \in I$

Dalla definizione si deduce che l'intersezione di ideali è ancora un ideale, è quindi ben definito, preso $S \subset A$, **l'ideale generato da S**:

$$< S > := \bigcap_{S \subset I} I \ , \quad I \ ideale$$

L'ideale generato da S può anche essere identificato con l'insieme delle combinazioni lineari degli elementi in S a coefficienti in A, in particolare se $d \in A$ allora $(d) := < \{d\} > = \{ad \mid a \in A\}$ si dice **ideale principale**.

Nel corso abbiamo dimostrato che $\mathbb{Z}$ è a ideali principali, ovvero ogni suo ideale è della forma (d) con $d \in \mathbb{Z}$, quindi se si considera un ideale della forma (a, b) esiste un altro numero d unico a meno del segno, che risulta essere il loro **MCD**, tale che $(a, b) = (d)$.
Per determinare il MCD si può applicare l'algoritmo euclideo delle divisioni successive, metodo applicato nel seguente codice.

Listing 10: MCD.h

```
#include <stdlib.h>
#include <gmp.h>

void mcd_euclide (mpz_t, mpz_t, mpz_t);
void mcd_binario (mpz_t, mpz_t, mpz_t);
void mcd_euclide_array (mpz_t, mpz_t*, int);
void mcd_binario_array (mpz_t, mpz_t*, int);
void mcd_euclide_concat (mpz_t, mpz_t*, int);
void mcd_binario_concat (mpz_t, mpz_t*, int);

// massimo comun divisore tra n e m, salvato in a, con algoritmo di Euclide. Il
    risultato è sempre non negativo.
void mcd_euclide (mpz_t a, mpz_t n, mpz_t m) {
    // creo nuove variabili per non modificare n e m, e le rendo positive per avere
        mcd>=0
    mpz_t x,y;
    mpz_inits(x,y,NULL);
    mpz_abs(x,n); mpz_abs(y,m);

    while (mpz_cmp_si(y,0)>0) {  // se y>0 continuo a iterare
        mpz_fdiv_r(x,x,y);  // x diventa il resto di x/y
        mpz_swap(x,y);
    }
    mpz_set(a,x);

    mpz_clears(x,y,NULL);
    return;
}

```

```c
// massimo comun divisore tra n e m, salvato in a, con algoritmo binario. Il risultato è sempre non negativo.
void mcd_binario (mpz_t a, mpz_t n, mpz_t m) {
    // casi base: uno dei due numeri è 0
    if (mpz_cmp_si(n,0)==0) {
        mpz_set(a,m);
        mpz_abs(a,a);
        return;
    }
    else if (mpz_cmp_si(m,0)==0) {
        mpz_set(a,n);
        mpz_abs(a,a);
        return;
    }

    // creo nuove variabili per non modificare n e m, e le rendo positive per avere mcd>=0
    mpz_t x,y;
    mpz_inits(x,y,NULL);
    mpz_abs(x,n); mpz_abs(y,m);

 /* ottimizzazione che ho trovato io facendo qualche test: se i due numeri hanno ordini di grandezza molto diversi l'algoritmo diventa
    inefficiente, quindi eseguo una sola passata dell'algoritmo euclideo facendo una prima divisione con resto. In questo modo i due
    numeri avranno ordine di grandezza simile al più piccolo, e l'algoritmo binario si velocizza molto. */
    if (mpz_cmp(x,y)>0) {
        mpz_fdiv_r(x,x,y);
        if (mpz_cmp_si(x,0)==0) {
            mpz_set(a,y);
            mpz_clears(x,y,NULL);
            return;
        }
    }
    else {
        mpz_fdiv_r(y,y,x);
        if (mpz_cmp_si(y,0)==0) {
            mpz_set(a,x);
            mpz_clears(x,y,NULL);
            return;
        }
    }

    // calcolo (in k) la più alta potenza di 2 che divide i due numeri
    unsigned long int i,j,k;
    i = mpz_scan1(x,0);
    j = mpz_scan1(y,0);
    if (i<j) k=i;
    else k=j;

    // rendo dispari i due numeri, avendo già salvato la più alta potenza di 2 che divide entrambi
    mpz_fdiv_q_2exp(x,x,i);
    mpz_fdiv_q_2exp(y,y,j);

    // ciclo lavorando sempre su numeri dispari, ed esco quando uno dei due è 0
    while (mpz_cmp_si(x,0)>0) {
        if (mpz_cmp(x,y)<0) mpz_swap(x,y); // in questo modo x>=y
        mpz_sub(x,x,y);  // eseguo la sottrazione x=x-y; ora x è pari
        if (mpz_cmp_si(x,0)==0) break;  // se provo a scannerizzare il primo 1 nella rappresentazione binaria di 0 ottengo -1, quindi devo uscire prima
        i = mpz_scan1(x,0);
        mpz_fdiv_q_2exp(x,x,i);  // tolgo tutte le potenze di 2 che dividono x: così torna dispari
    }

    // quando esco dal ciclo in y ci sarà mcd dei numeri resi dispari; dopo rimoltiplico per la giusta potenza di 2 calcolata prima
```

```
    mpz_set(a,y);
    mpz_mul_2exp(a,a,k);

    mpz_clears(x,y,NULL);
    return;
}

// massimo comun divisore degli n>0 interi dell'array "integers", salvato in a, con algoritmo di Euclide esteso. Il risultato è sempre non negativo.
void mcd_euclide_array (mpz_t a, mpz_t* integers, int n) {
    // caso banale: n=1
    if (n==1) {
        mpz_set(a,integers[0]);
        mpz_abs(a,a);
        return;
    }
    // creo nuove variabili per non modificare gli interi dati, e le rendo positive per avere mcd>=0
    int i;
    mpz_t* nums = malloc(n*sizeof(mpz_t));
    for (i=0; i<n; i++) {
        mpz_init_set(nums[i],integers[i]);
        mpz_abs(nums[i],nums[i]);
    }

    int k=n-1; i=0;
    // sposto gli zeri in fondo
    while (i<k) {
        while (mpz_cmp_si(nums[i],0)>0 && i<n) i++;
        if (i>=n) break;
        while (mpz_cmp_si(nums[k],0)==0 && k>=0) k--;
        if (k<0) break;
        if (i>=k) break;
        // ho individuato in posizione i un elemento ==0 e in posizione k un elemento >0: li scambio
        mpz_swap(nums[i],nums[k]);
        i++; k--;
    }
    k=n; // k indicherà l'indice dal quale gli elementi sono nulli (se ci sono zeri): nums[k]==0, nums[k-1]>0. (se non ce ne sono k=n)
    for (i=0; i<n; i++) {
        if (mpz_cmp_si(nums[i],0)==0) {
            k=i;
            break;
        }
    }
    // caso vettore tutto nullo: restituisco mcd=0
    if (k==0) {
        mpz_set_si(a,0);
        for (i=0; i<n; i++) mpz_clear(nums[i]);
        return;
    }

    // ciclo finché ho più di un elemento non nullo
    while (k>1) {
        // trovo il minimo elemento non nullo e lo porto in prima posizione
        int index_min=0;
        for (i=1; i<k; i++) {
            if (mpz_cmp(nums[index_min],nums[i])>0) index_min=i;
        }
        mpz_swap(nums[index_min],nums[0]);

        // divido per il minimo, calcolo i resti e sposto eventuali zeri ottenuti
        for (i=1; i<k; i++) {
            mpz_fdiv_r(nums[i],nums[i],nums[0]); // nums[i] diventa nums[i] mod nums[0], dove nums[0] è il minimo

            // se dopo la divisione ho ottenuto resto 0, porto il numero in fondo
            if (mpz_cmp_si(nums[i],0)==0) {
                mpz_swap(nums[i],nums[k-1]);
```

```
                k--; i--; // aggiorno gli indici
            }
        }
    }

    // quando esco dal ciclo ho k=1 ossia solo il primo elemento è non nullo: ho trovato mcd.
    mpz_set(a,nums[0]);
    for (i=0; i<n; i++) mpz_clear(nums[i]);
    return;
}

// massimo comun divisore degli n>0 interi dell'array "integers", salvato in a, con algoritmo binario esteso. Il risultato è sempre non negativo.
void mcd_binario_array (mpz_t a, mpz_t* integers, int n) {
    // caso banale: n=1
    if (n==1) {
        mpz_set(a,integers[0]);
        mpz_abs(a,a);
        return;
    }
    // creo nuove variabili per non modificare gli interi dati, e le rendo positive per avere mcd>=0
    int i;
    mpz_t* nums = malloc(n*sizeof(mpz_t));
    for (i=0; i<n; i++) {
        mpz_init_set(nums[i],integers[i]);
        mpz_abs(nums[i],nums[i]);
    }

    int k=n-1; i=0;
    // sposto gli zeri in fondo
    while (i<k) {
        while (mpz_cmp_si(nums[i],0)>0 && i<n) i++;
        if (i>=n) break;
        while (mpz_cmp_si(nums[k],0)==0 && k>=0) k--;
        if (k<0) break;
        if (i>=k) break;
        // ho individuato in posizione i un elemento ==0 e in posizione k un elemento >0: li scambio
        mpz_swap(nums[i],nums[k]);
        i++; k--;
    }
    k=n; // k indicherà l'indice dal quale gli elementi sono nulli (se ci sono zeri): nums[k]==0, nums[k-1]>0. (se non ce ne sono k=n)
    for (i=0; i<n; i++) {
        if (mpz_cmp_si(nums[i],0)==0) {
            k=i;
            break;
        }
    }
    // caso vettore tutto nullo: restituisco mcd=0
    if (k==0) {
        mpz_set_si(a,0);
        for (i=0; i<n; i++) mpz_clear(nums[i]);
        return;
    }

 /* ottimizzazione che ho trovato io facendo qualche test: se i numeri hanno ordini di grandezza molto diversi l'algoritmo diventa
    inefficiente, quindi eseguo una sola passata dell'algoritmo euclideo facendo una prima divisione con resto. In questo modo i
    numeri avranno ordine di grandezza simile al più piccolo, e l'algoritmo binario si velocizza molto. */
    int index_min=0;
    for (i=1; i<k; i++) {
        if (mpz_cmp(nums[index_min],nums[i])>0) index_min=i;
    }
    mpz_swap(nums[index_min],nums[0]);
    // divido per il minimo, calcolo i resti e sposto eventuali zeri ottenuti
```

```
    for (i=1; i<k; i++) {
        mpz_fdiv_r(nums[i],nums[i],nums[0]); // nums[i] diventa nums[i] mod nums[0], dove nums[0] è il minimo
        // se dopo la divisione ho ottenuto resto 0, porto il numero in fondo
        if (mpz_cmp_si(nums[i],0)==0) {
            mpz_swap(nums[i],nums[k-1]);
            k--; i--; // aggiorno gli indici
        }
    }

    // inizio (finalmente) con la parte di algoritmo binario: salvo la più alta potenza di 2 che divide tutti i numeri
    unsigned long int j,exp;
    exp=mpz_scan1(nums[0],0);
    mpz_fdiv_q_2exp(nums[0],nums[0],exp);
    for (i=1; i<k; i++) {
        j=mpz_scan1(nums[i],0);
        mpz_fdiv_q_2exp(nums[i],nums[i],j);
        if (j<exp) exp=j;
    }

    // ciclo finché ho più di un elemento non nullo; saranno sempre dispari
    while (k>1) {
        // trovo il minimo elemento non nullo e lo porto in prima posizione
        int index_min=0;
        for (i=1; i<k; i++) {
            if (mpz_cmp(nums[index_min],nums[i])>0) index_min=i;
        }
        mpz_swap(nums[index_min],nums[0]);

        // sottraggo il minimo, poi se ho ottenuto 0 lo sposto in fondo, altrimenti tolgo le potenze di 2
        for (i=1; i<k; i++) {
            mpz_sub(nums[i],nums[i],nums[0]);

            if (mpz_cmp_si(nums[i],0)==0) {
                mpz_swap(nums[i],nums[k-1]);
                k--; i--;
            }
            else {
                j=mpz_scan1(nums[i],0);
                mpz_fdiv_q_2exp(nums[i],nums[i],j);
            }
        }
    }

    // quando esco dal ciclo ho k=1 ossia solo il primo elemento è non nullo: ho trovato mcd, a patto di rimoltiplicare per la giusta potenza di 2
    mpz_set(a,nums[0]);
    mpz_mul_2exp(a,a,exp);
    for (i=0; i<n; i++) mpz_clear(nums[i]);
    return;
}

// massimo comun divisore degli n>0 interi dell'array "integers", salvato in a, ottenuto concatenando l'algoritmo euclideo a coppie
void mcd_euclide_concat (mpz_t a, mpz_t* integers, int n) {
    // caso banale: n=1
    if (n==1) {
        mpz_set(a,integers[0]);
        mpz_abs(a,a);
        return;
    }

    mcd_euclide(a,integers[0],integers[1]);
    for (int i=2; i<n; i++) {
        mcd_euclide(a,a,integers[i]);
    }

    return;
```

```
}

// massimo comun divisore degli n>0 interi dell'array "integers", salvato in a,
    ottenuto concatenando l'algoritmo binario a coppie
void mcd_binario_concat (mpz_t a, mpz_t* integers, int n) {
    // caso banale: n=1
    if (n==1) {
        mpz_set(a,integers[0]);
        mpz_abs(a,a);
        return;
    }

    mcd_binario(a,integers[0],integers[1]);
    for (int i=2; i<n; i++) {
        mcd_binario(a,a,integers[i]);
    }

    return;
}
```

Ripercorrendo all'indietro i passaggi dell'algoritmo euclideo si possono determinare due coefficienti interi x e y tali che $(a,b) = (d) \Rightarrow d = ax + by$, dove quest'ultima relazione prende il nome di **identità di Bèzout**, estremamente utile per determinare gli inversi degli elementi in $(\mathbb{Z}_{(n)})^*$.

Listing 11: Bezout.h

```
#include <stdlib.h>
#include <gmp.h>

void bezout (mpz_t, mpz_t, mpz_t, mpz_t, mpz_t);
void bezout_array (mpz_t, mpz_t*, mpz_t*, int);
int inv_mod (mpz_t, mpz_t, mpz_t);
int inv_mod_mcd (mpz_t, mpz_t, mpz_t, mpz_t);
void exp_mod (mpz_t, mpz_t, mpz_t, mpz_t);
void exp_mod_ui (mpz_t, mpz_t, unsigned int, mpz_t);

// funzioni ausiliarie per bezout:
void init_id_matrix (mpz_t**, int);
void swap_rows (mpz_t**, int, int, int);
void add_multiple_row (mpz_t**, int, mpz_t, int, int);

// Calcolo d=mcd(n,m) con algoritmo euclideo (scelgo d>=0), e anche x,y interi tali
     che x*n+y*m=d.
void bezout(mpz_t d, mpz_t x, mpz_t y, mpz_t n, mpz_t m) {
        int i,j;
        mpz_t a,b,q,temp1,temp2;
        mpz_inits(a,b,q,temp1,temp2,NULL);
        mpz_abs(a,n); mpz_abs(b,m); // inizializzo a,b positivi per avere d>=0; se
            necessario cambio segno a x,y alla fine
        // la matrice M (2*2) conterra' alla fine i coefficienti di bezout nella
            prima riga
        mpz_t** M=malloc(2*sizeof(mpz_t*));
        for (i=0; i<2; i++) M[i]=malloc(2*sizeof(mpz_t));
        mpz_init_set_si(M[0][0],1); mpz_init(M[0][1]); mpz_init(M[1][0]);
            mpz_init_set_si(M[1][1],1); // inizializzo M = identità

        // algoritmo di euclide
        while (mpz_cmp_si(b,0)>0) {  // se b>0 continuo a iterare
        mpz_fdiv_qr(q,a,a,b);  // a diventa il resto di a/b, q quoziente
        mpz_swap(a,b);

                // aggiorno matrice dei coefficienti: devo moltiplicare a sinistra
                    per [[0, 1]; [1, -q]] (caso particolare di solo due interi)
                mpz_set(temp1,M[1][0]); mpz_set(temp2,M[1][1]); // copio prima riga
```

```
                mpz_neg(q,q);
                mpz_mul(M[1][0],M[1][0],q); mpz_add(M[1][0],M[1][0],M[0][0]); // calcolato riga 2 colonna 1
                mpz_mul(M[1][1],M[1][1],q); mpz_add(M[1][1],M[1][1],M[0][1]); // calcolato riga 2 colonna 2
                mpz_set(M[0][0],temp1); mpz_set(M[0][1],temp2); // calcolata riga 1
    }
    mpz_set(d,a);  // calcolato mcd
        mpz_set(x,M[0][0]); mpz_set(y,M[0][1]);
        // aggiusto i coefficienti per eventuali cambi di segno
        if (mpz_cmp_si(n,0)<0) mpz_neg(x,x);
        if (mpz_cmp_si(m,0)<0) mpz_neg(y,y);

        for (i=0; i<2; i++) {
                for (j=0; j<2; j++) mpz_clear(M[i][j]);
        }
        mpz_clears(a,b,q,temp1,temp2,NULL);
        return;
}

// Calcolo d=mcd (d>=0) dell'array "integers" di n>0 interi, salvando in "coeff" i coefficienti tali che d=coeff[0]*integers[0]+...+coeff[n-1]*integers[n-1]
void bezout_array (mpz_t d, mpz_t* coeff, mpz_t* integers, int n) {
        // caso banale: n=1
    if (n==1) {
        mpz_set(d,integers[0]);
                if (mpz_cmp_si(d,0)<0) {
                        mpz_neg(d,d);
                        mpz_set_si(coeff[0],-1);
                }
                else mpz_set_si(coeff[0],1);
        return;
    }

        // creo nuove variabili per non modificare gli interi dati (positive per avere mcd>=0), e alloco la matrice necessaria per calcolare i coefficienti
    int i,j;
    mpz_t* nums = malloc(n*sizeof(mpz_t));
    for (i=0; i<n; i++) {
        mpz_init_set(nums[i],integers[i]);
        mpz_abs(nums[i],nums[i]);
    }
        mpz_t** M=malloc(n*sizeof(mpz_t*));
        for (i=0; i<n; i++) M[i]=malloc(n*sizeof(mpz_t));
        init_id_matrix(M,n);

        // algoritmo di euclide:

        int k=n-1; i=0;
    // sposto gli zeri in fondo
    while (i<k) {
        while (mpz_cmp_si(nums[i],0)>0 && i<n) i++;
        if (i>=n) break;
        while (mpz_cmp_si(nums[k],0)==0 && k>=0) k--;
        if (k<0) break;
        if (i>=k) break;
        // ho individuato in posizione i un elemento ==0 e in posizione k un elemento >0: li scambio, e tengo traccia nella matrice M
        mpz_swap(nums[i],nums[k]);
                swap_rows(M,n,i,k);
        i++; k--;
    }
        k=n; // k indicherà l'indice dal quale gli elementi sono nulli (se ci sono zeri): nums[k]==0, nums[k-1]>0. (se non ce ne sono k=n)
    for (i=0; i<n; i++) {
        if (mpz_cmp_si(nums[i],0)==0) {
            k=i;
            break;
        }
```

```
    }
    // caso vettore tutto nullo: restituisco d=0, coeff = array nullo.
    if (k==0) {
        mpz_set_si(d,0);
        for (i=0; i<n; i++) mpz_set_si(coeff[i],0);
                for (i=0; i<n; i++) mpz_clear(nums[i]);
                for (i=0; i<n; i++) for (j=0; j<n; j++) mpz_clear(M[i][j]);
        return;
    }

        mpz_t q; mpz_init(q);
        // ciclo finché ho più di un elemento non nullo
    while (k>1) {
        // trovo il minimo elemento non nullo e lo porto in prima posizione, tenendo traccia in M
        int index_min=0;
        for (i=1; i<k; i++) {
            if (mpz_cmp(nums[index_min],nums[i])>0) index_min=i;
        }
        mpz_swap(nums[index_min],nums[0]);
                swap_rows(M,n,index_min,0);

        // divido per il minimo, calcolo i resti e sposto eventuali zeri ottenuti
        for (i=1; i<k; i++) {
            mpz_fdiv_qr(q,nums[i],nums[i],nums[0]); // nums[i] diventa nums[i] mod nums[0], dove nums[0] è il minimo
                        mpz_neg(q,q);
                        add_multiple_row(M,n,q,0,i); // alla riga i tolgo q volte la prima riga (con indice 0)

            // se dopo la divisione ho ottenuto resto 0, porto il numero in fondo
            if (mpz_cmp_si(nums[i],0)==0) {
                mpz_swap(nums[i],nums[k-1]);
                                swap_rows(M,n,i,k-1);
                k--; i--; // aggiorno gli indici
            }
        }
    }

        // quando esco dal ciclo ho k=1 ossia solo il primo elemento è non nullo: ho trovato mcd, e nella prima riga di M ho i coefficienti cercati
    mpz_set(d,nums[0]);
        for (i=0; i<n; i++) {
                mpz_set(coeff[i],M[0][i]);
                if (mpz_cmp_si(integers[i],0)<0) mpz_neg(coeff[i],coeff[i]); // aggiusto i segni poiché all'inizio avevo reso tutto positivo
        }

        mpz_clear(q);
        for (i=0; i<n; i++) mpz_clear(nums[i]);
        for (i=0; i<n; i++) for (j=0; j<n; j++) mpz_clear(M[i][j]);
        return;
}

// Calcola in x l'inverso di a modulo n: se esiste restituisce 1, altrimenti 0. Scelgo x>=0, e suppongo n>0.
int inv_mod (mpz_t x, mpz_t a, mpz_t n) {
        mpz_t d,y;
        mpz_inits(d,y,NULL);

        bezout(d,x,y,a,n);
        mpz_fdiv_r(x,x,n); // rendo x tale che 0<x<n
        if (mpz_cmp_si(d,1)==0) { // caso mcd(a,n)=1: l'inverso esiste ed è x
                mpz_clears(d,y,NULL);
                return 1;
        }
        // caso mcd(a,n)!=1: l'inverso non esiste
        mpz_clears(d,y,NULL);
        return 0;
}
```

```c

// Come la funzione precedente, ma restituisco anche d=mcd(a,n)
int inv_mod_mcd (mpz_t d, mpz_t x, mpz_t a, mpz_t n) {
        mpz_t y;
        mpz_init(y);

        bezout(d,x,y,a,n);
        mpz_fdiv_r(x,x,n); // rendo x tale che 0<x<n
        if (mpz_cmp_si(d,1)==0) { // caso mcd(a,n)=1: l'inverso esiste ed è x
                mpz_clear(y);
                return 1;
        }
        // caso mcd(a,n)!=1: l'inverso non esiste
        mpz_clear(y);
        return 0;
}

// calcola in x la base b elevata alla potenza exp>=0 modulo n, con esponenziazione binaria
void exp_mod (mpz_t x, mpz_t b, mpz_t exp, mpz_t n) {
        int bit;
        mpz_t squares,e;
        mpz_init_set(squares,b);
        mpz_init_set(e,exp);
        mpz_set_si(x,1);

        while (mpz_cmp_si(e,0)>0) {
                bit=mpz_tstbit(e,0); // controllo l'ultima cifra della rappresentazione binaria di e
                if (bit==1) { // se bit è 1 moltiplico x per il quadrato (mod n), altrimenti lascio così
                        mpz_mul(x,x,squares);
                        mpz_fdiv_r(x,x,n);
                }
                mpz_mul(squares,squares,squares); // aggiorno il quadrato (mod n)
                mpz_fdiv_r(squares,squares,n);
                mpz_fdiv_q_2exp(e,e,1); // shifto di 1 i bit di e
        }

        mpz_clears(squares,e,NULL);
        return;
}

// come exp_mod sopra, ma l'esponente è di tipo unsigned int
void exp_mod_ui (mpz_t x, mpz_t b, unsigned int exp, mpz_t n) {
        mpz_t e;
        mpz_init_set_ui(e,exp);
        exp_mod(x,b,e,n);
        mpz_clear(e);

        return;
}

// inizializza M matrice identità n*n
void init_id_matrix (mpz_t** M, int n) {
        int i,j;
        for (i=0; i<n; i++) {
                for (j=0; j<n; j++) {
                        mpz_init(M[i][j]);
                }
        }
        for (i=0; i<n; i++) mpz_set_si(M[i][i],1);

        return;
}

// scambia le righe i e k della matrice M (n*n)
void swap_rows (mpz_t** M, int n, int i, int k) {
        int j;
        for (j=0; j<n; j++) mpz_swap(M[i][j],M[k][j]);
```

```c

        return;
}

// nella matrice M n*n, aggiungo alla riga k la riga i moltiplicata per mult
void add_multiple_row (mpz_t** M, int n, mpz_t mult, int i, int k) {
        int j;
        mpz_t temp;
        mpz_init(temp);
        for (j=0; j<n; j++) {
                mpz_mul(temp,M[i][j],mult);
                mpz_add(M[k][j],M[k][j],temp);
        }

        mpz_clear(temp);
        return;
}
```

2.2 Teorema cinese dei resti

Teorema (versione 1): Siano $a_1, a_2, ..., a_n \in \mathbb{Z}$ tali che $(a_i, a_j) = 1 \; \forall i \neq j$. Allora

$$\mathbb{Z}/(a_1, a_2, ..., a_n) \cong \mathbb{Z}/(a_1) \oplus \mathbb{Z}/(a_2) \oplus ... \oplus \mathbb{Z}/(a_n)$$

Teorema (versione 2): Siano $a_1, a_2, ..., a_n, \alpha_1, ..., \alpha_n \in \mathbb{Z}$ con $(a_i, a_j) = 1 \; \forall i \neq j$. Allora il sistema di congruenze $\{x \equiv \alpha_i \; mod \; a_i \mid i = 1, ..., n\}$ ammette un'unica soluzione $mod \; a_1 a_2 ... a_n$.

Le due formulazioni sono equivalenti grazie all'identità di Bézout e la seconda è facilmente implementabile.

Listing 12: Sis Congruenze.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include "..\Include\Smith.h"
#include "..\Include\Bezout_old.h"

int** input_sys (int**, int);
void print_sys (int**, int);
int* input_vec_zeros (int*, int);
int* input_vec_one (int*, int);
void solve_sys (int**, int*, int*, int*, int*, int);
int adjust_sol (int, int);

int main() {
        int m;
        printf("Numero di equazioni nel sistema: ");
        scanf("%d", &m);
        printf("Dati del sistema (ad x=a mod b corrisponde l'input da tastiera 'a b
        '):\n");
        int **sys=NULL;
        sys=input_sys(sys, m);
        print_sys(sys, m);

        // vettori necessari per la risoluzione del sistema
        int *A=NULL; // sono i moduli delle singole equazioni
        A=input_vec_zeros(A, m);
        int *B=NULL; // B[i] è il prodotto di A[j] con j!=i
        B=input_vec_one(B, m);
        int *Alpha=NULL; // sono le congruenze delle singole equazioni
```

```
        Alpha=input_vec_zeros(Alpha, m);
        int *Gamma=NULL; // conterrà la prima riga di S dopo aver applicato Smith a
             B
        Gamma=input_vec_zeros(Gamma, m);
        solve_sys(sys, A, B, Alpha, Gamma, m);

        // troviamo la soluzione x
        int x=0;
        int mod=1;
        for (int i=0; i<m; i++) {
                x=x+Gamma[i]*B[i]*Alpha[i];
                mod=mod*A[i];
        }
        // aggiustiamo la soluzione x rendendola della forma x=a mod b con 0<=a<b
        x=adjust_sol(x, mod);
        printf ("Soluzione: x=%d mod %d\n", x, mod);

        return 0;
}

// inserire la matrice dei dati del sistema (matrice m*2)
int** input_sys (int** mat, int m) {
        mat=calloc(m, sizeof(int*));
        for (int i=0; i<m; i++) {
                mat[i]=calloc(2, sizeof(int));
        }
        for (int i=0; i<m; i++) {
                for (int j=0; j<2; j++) {
                        scanf("%d", &mat[i][j]);
                }
        }
        return mat;
}

// stampare il sistema
void print_sys (int** sys, int m) {
        printf("\nSistema: ");
        for (int i=0; i<m; i++) {
                printf("x=%d mod %d\n", sys[i][0], sys[i][1]);
                if (i!=(m-1)) {
                        printf("\t ");
                }
        }
        printf("\n");
        return;
}

// inizializzare un vettore nullo di m componenti
int* input_vec_zeros (int *vec, int m) {
        vec=calloc(m, sizeof(int));
        return vec;
}

// inizializzare un vettore di 1 di m componenti
int* input_vec_one (int* vec, int m) {
        vec=calloc(m, sizeof(int));
        for (int i=0; i<m; i++) {
                vec[i]=1;
        }
        return vec;
}

// creare i vettori che serviranno poi nella risoluzione del sistema
void solve_sys (int **sys, int *A, int *B, int *Alpha, int *Gamma, int m) {
        for (int i=0; i<m; i++) { // prese congruenze del tipo x=a mod alpha
                A[i]=sys[i][1]; // A è il vettore delle a
                for (int j=0; j<m; j++) {
                        if (j!=i) {
                                B[i]=B[i]*sys[j][1]; // la componente B[i] è il
                                    prodotto delle A[j] per j!=i
```

```
                        }
                }
                Alpha[i]=sys[i][0]; // Alpha è vettore delle alpha
        }

        int **D=NULL;
        D=input_null(D, m, 1);


        for (int i=0; i<m; i++) {
                D[i][0]=B[i]; // inizializziamo D a B
        }

        int **S=NULL;
        S=input_id(S, m);
        int **T=NULL;
        T=input_id(T, 1);
        bezout (m, B, D, S, T);

        for (int i=0; i<m; i++)  {
                Gamma[i]=S[0][i]; // Gamma corrisponde ai coefficienti di Bezout,
                    ossia è la prima riga della matrice S
        }
        return;
}

// aggiustiamo la soluzione x rendendola della forma x=a mod b con 0<=a<b
int adjust_sol (int x, int mod) {
        if (abs(x)>=mod) { // se abs(x)>=mod togliamo ad x i multipli di mod perchè
             a*x=x mod a
                int temp=0;
                temp=x/mod;
                x=x-temp*mod;
        }
        if (x<0) { // se x<0 prendiamo la sua classe equivalente positiva
                x=mod+x;
        }
        return x;
}
```

2.3 Test di primalità

2.3.1 Test di Fermat

Se $p \geq 3$ è un numero primo allora $|(\mathbb{Z}/(p))| = p - 1$ essendo un campo; conseguentemente, fissato $n \in \mathbb{Z}$, se esiste $a \in \{2, ..., n-2\}$ tale che

$$a^{n-1} \not\equiv 1 \; mod \; n$$

allora n è necessariamente un numero composto (in tal caso a si dice **testimone di Fermat**).
Nota: esistono dei numeri, detti **pseudoprimi di Fermat**, che non forniscono una prova di non-primalità per ogni scelta di a.

Listing 13: Fermat come fatto a lezione

```
#pragma once
#ifndef Fermat
#define Fermat

long int** matNull (long int**, int, int);
int* trasfBin (int*, long int, int);
void printVec (int*, int);
```

```c
long int** inputMatPot (long int**, int*, int, long int, long int);
long int potAmod (long int, long int);
long int congruenza_a (long int, long int, long int);
long int adjust_x (long int, long int);
long int change_a (long int, long int, long int);
bool fermat (long int, int);
long int MCD(long int, long int);

long int** matNull (long int **mat, int m, int n) {
        mat=calloc(m, sizeof(long int*));
        for (int i=0; i<m; i++) {
                mat[i]=calloc(n, sizeof(long int));
        }
        return mat;
}

int* trasfBin (int *vecBin, long int n, int l) {
        vecBin=calloc(l, sizeof(int));
        for (int i=l-1; i>=0; i--) {
                vecBin[i]=n%2;
                n=n/2;
        }
        return vecBin;
}

void printVec (int *vecBin, int l) {
        for (int i=0; i<l; i++) {
                printf("%d", vecBin[i]);
        }
        printf("\n");
        return;
}

long int** inputMatPot (long int **matPot, int *vecBin, int l, long int a, long int
    n) {
        matPot=matNull(matPot, l, 2);
        int j=l-1;
        for (int i=0; i<l; i++) {
                matPot[i][0]=vecBin[j]; //nella prima colonna scrivimao il numero
                    in base binaria (al contrario)
                j=j-1;
                if (i==0) { // nella seconda colonna mettiamo le potenze di a mod n
                     (a, a^2, (a^2)^2 ...)
                        matPot[i][1]=a;
                } else {
                        matPot[i][1]=potAmod(matPot[i-1][1], n);
                }
        }
        return matPot;
}

// calcola l'elevazione al quadrato di k mod mod
long int potAmod (long int k, long int mod) {
        long int x=k*k;
        x=adjust_x(x,mod);
        return x;
}

// calcola a^(n-1) mod n
long int congruenza_a (long int a, long int m, long int n) {
        int lBin=floor(log2(m))+1; // calcolo la lunghezza di n-1 (ossia m) in
            binario per poi inizializzare il vettore
        int *vecBin=NULL;
        vecBin=trasfBin(vecBin, m, lBin); //m sarà n-1, quindi trasformo n-1 in
            binario
        long int **matPot=NULL;
        matPot=inputMatPot(matPot, vecBin, lBin, a, n); // matPot è una matrice
            lBin*2, dove la prima colonna corrisponde ad n_binario e la seconda
            alle potenze di a (come meglio specificato nei commenti della funzione
            matPot)
```

```
	long int c=1; // inizializziamo c=a^(n-1) mod n
	for (int i=0; i<lBin; i++) { // modifichiamo c a dovere
		if (matPot[i][0]==1) { // se la cifra in binario è 1 moltiplico c per la potenza di a corrispondente, sennò non modifico c
			c=c*matPot[i][1];
		}
		c=adjust_x(c, n); // riporto c congruo modulo n
	}
	return c;
}

// prendo la classe di congruenza mod mod di x
long int adjust_x (long int x, long int mod) {
	if ((x>0 && x>=mod) || (x<0 && (-x)>=mod)) { // se abs(x)>=mod togliamo ad x i multipli di mod perchè a*x=x mod a (abs scrittto così perche abs lavora su int)
		long int temp=0;
		temp=x/mod;
		x=x-temp*mod;
	}
	long int ceilModMezzi=(mod/2)+1; // perchè ceil lavora su double
	if (x>ceilModMezzi) { // prendiamo la classe di equivalente di modulo minore
		x=x-mod;
	} else if (x<0 && (-x)>=ceilModMezzi) {
		x=mod+x;
	}
	return x;
}

// modifico in maniera casuale la base a
long int change_a (long int a, long int max, long int min) {
	srand(time(NULL));
	a=rand()%(max-min+1)+min; // prendiamo una base a scelta casualmente nell'intervallo [2, n-2]
	return a;
}

// studiamo la primalità dei numeri dispari
bool fermat (long int n, int it_max) {
	long int a=2, max=n-2, min=2;
	for (int it=1; it<=it_max; it++) {
		long int d=MCD(a, n); // calcolo il massimo comune divisore tra a e n
		if (d!=1) { // se è diverso da 1 n non è primo e d è un suo divisore
			printf("%ld non è un numero primo e %ld è un suo divisore\n", n, d);
			return false;
		} else { // altrimenti verifico se a^(n-1) é congruo o meno a 1 mod n, in caso affermativo n è uno pseudoprimo e continuo la verifica con un'altra base a, se non lo è n non è primo, ma in questo caso non ho informazioni sul suo divisore
			long int c=congruenza_a(a, n-1, n);
			if(c!=1) {
				printf("%ld non è un numero primo\n", n);
				return false;
			} else {
				a=change_a(a, max, min);
			}
		}
	}
	return true;
}

long int MCD(long int a, long int b) {
	long int temp=0;
	while (b!=0) {
		temp=b;
		b=a%b;
```

```
                a=temp;
        }
        return a;
}

#endif
```

Listing 14: Fermat.c

```
#include <stdio.h>
#include <stdbool.h>
#include <string.h>
#include <time.h>
#include <gmp.h>
#include "mcd.h"

bool fermat (mpz_t, mpz_t);
void RandNumber (mpz_t, mpz_t, gmp_randstate_t);
void exp_mod (mpz_t, mpz_t, mpz_t, mpz_t);

int main() {
        mpz_t n, r, it_max;
        mpz_inits(n, r, it_max, NULL);
        printf("n=");
        gmp_scanf("%Zd", &n);

        // 2 e 3 sono primi
        if (mpz_cmp_si(n, 2)==0 || mpz_cmp_si(n, 3)==0) {
                gmp_printf("%Zd è un numero primo\n", n);
                return 0;
        }

        // i numeri pari diversi da 2 ovviamente non sono primi
        mpz_fdiv_r_ui(r, n, 2);
        if (mpz_cmp_si(r, 0)==0) {
                gmp_printf("%Zd non è un numero primo e 2 è un suo divisore\n", n);
                return 0;
        }

        // studiamo la primalità dei numeri dispari
        mpz_set_ui(it_max, mpz_sizeinbase(n, 2));
        if (fermat(n, it_max)) {
                gmp_printf("%Zd è un numero primo\n", n);
        }

        mpz_clears(n, r, it_max, NULL);

        return 0;
}

// studiamo la primalità dei numeri dispari: la funzione ritorna true se il numero
    è primo, altrimenti ritorna false
bool fermat (mpz_t n, mpz_t it_max) {
        mpz_t a, it, d, c, exp;
        mpz_inits(a, it, d, c, exp, NULL);
        gmp_randstate_t state;
        gmp_randinit_mt(state);
        gmp_randseed_ui(state, time(NULL));
        mpz_set_si(it, 0);
        mpz_set_si(a, 2);

        while (mpz_cmp(it, it_max)<0) { // finchè it < it_max
                mcd_euclide(d, a, n); // calcolo il massimo comune divisore tra a e
                     n
```

```c
                if (mpz_cmp_si(d, 1)!=0) { // se d!=1, n non è primo e d è un suo divisore
                        gmp_printf("%Zd non è un numero primo e %Zd è un suo divisore\n", n, d);

                        mpz_clears(a, it, d, c, exp, NULL);
                        gmp_randclear(state);

                        return false;
                } else { // altrimenti verifico se a^(n-1) é congruo o meno a 1 mod n, in caso affermativo n è uno pseudoprimo e continuo la verifica con un'altra base a, se non lo è n non è primo, ma in questo caso non ho informazioni sul suo divisore
                        mpz_sub_ui(exp, n, 1); // exp = n - 1
                        exp_mod(c, a, exp, n); // calcoliamo a^(n-1) mod n
                        if (mpz_cmp_si(c, 1)!=0) {
                                gmp_printf("%Zd non è un numero primo\n", n);

                                mpz_clears(a, it, d, c, exp, NULL);
                                gmp_randclear(state);

                                return false;
                        } else {
                                RandNumber(a, n, state);
                                mpz_add_ui(it, it, 1); // it = it + 1
                        }
                }
        }

        // Pulizia della memoria
        mpz_clears(a, it, d, c, exp, NULL);
        gmp_randclear(state);

        return true;
}

void RandNumber(mpz_t a, mpz_t n, gmp_randstate_t state) {
    mpz_t range;
    mpz_init(range);

    mpz_sub_ui(range, n, 3);  // range = (n - 2) - 2 + 1 = n - 3

    // Genera un numero casuale in [0, range]
    mpz_urandomm(a, state, range);
    // Sposta il numero generato nell'intervallo [2, n-2]
    mpz_add_ui(a, a, 2);

    mpz_clear(range);
}

// calcola in x la base b elevata alla potenza exp>=0 modulo n (exp binario)
void exp_mod (mpz_t x, mpz_t b, mpz_t exp, mpz_t n) {
        mpz_t squares, e;
        mpz_init_set(squares, b);
        mpz_init_set(e, exp);
        mpz_set_si(x, 1);

        while (mpz_cmp_si(e, 0)>0) { // finchè e > 0
                if (mpz_tstbit(e, 0)) { // se bit è 1 moltiplico x per il quadrato (mod n), altrimenti lascio così
                        mpz_mul(x, x, squares);
                        mpz_mod(x, x, n);
                }
                mpz_mul(squares, squares, squares); // aggiorno il quadrato (mod n)
                mpz_mod(squares, squares, n); // squares mod n
                mpz_fdiv_q_2exp(e, e, 1); // shifto di 1 i bit di e
        }

        mpz_clears(squares, e, NULL);

```

```
        return;
}
```

2.3.2 Test di Eulero

Ripercorrendo la strategia nel test di Fermat ci si può accorgere che $a^{n-1} \equiv 1 \ mod \ n \Rightarrow a^{\frac{n-1}{2}} \in \{1, -1\} \ mod \ n$, quindi se esiste un a tale che

$$a^{\frac{n-1}{2}} \notin \{1, -1\} \ mod \ n$$

allora n è un numero composto.
Nota: questo test è più forte di quello di Fermat, ma esistono comunque degli pseudoprimi di Eulero.

Listing 15: Eulero.c

```c
#include <stdio.h>
#include <stdbool.h>
#include <string.h>
#include <time.h>
#include <gmp.h>
#include "mcd.h"

bool eulero (mpz_t, mpz_t);
void RandNumber (mpz_t, mpz_t, gmp_randstate_t);
void exp_mod (mpz_t, mpz_t, mpz_t, mpz_t);

int main() {
        mpz_t n, r, it_max;
        mpz_inits(n, r, it_max, NULL);
        printf("n=");
        gmp_scanf("%Zd", &n);

        // 2 e 3 sono primi
        if (mpz_cmp_si(n, 2)==0 || mpz_cmp_si(n, 3)==0) {
                gmp_printf("%Zd è un numero primo\n", n);
                return 0;
        }

        // i numeri pari diversi da 2 ovviamente non sono primi
        mpz_fdiv_r_ui(r, n, 2);
        if (mpz_cmp_si(r, 0)==0) {
                gmp_printf("%Zd non è un numero primo e 2 è un suo divisore\n", n);
                return 0;
        }

        // studiamo la primalità dei numeri dispari
        mpz_set_ui(it_max, mpz_sizeinbase(n, 2));
        if (eulero(n, it_max)) {
                gmp_printf("%Zd è un numero primo\n", n);
        }

        mpz_clears(n, r, it_max, NULL);

        return 0;
}

bool eulero (mpz_t n, mpz_t it_max) {
        mpz_t a, it, d, c, exp, n1;
        mpz_inits(a, it, d, c, exp, n1, NULL);
        gmp_randstate_t state;
        gmp_randinit_mt(state);
```

```c
        gmp_randseed_ui(state, time(NULL));
        mpz_set_si(it, 0);
        mpz_set_si(a, 2);

        while (mpz_cmp(it, it_max)<0) { // finchè it < it_max
                mcd_euclide(d, a, n); // calcolo il massimo comune divisore tra a e n
                if (mpz_cmp_si(d, 1)!=0) { // se d!=1, n non è primo e d è un suo divisore
                        gmp_printf("%Zd non è un numero primo e %Zd è un suo divisore\n", n, d);

                        mpz_clears(a, it, d, c, exp, n1, NULL);
                        gmp_randclear(state);

                        return false;
                } else { // verifico se a^((n-1)/2) é congruo o meno a più o meno 1 mod n, in caso affermativo n potrebbe essere primo quindi continuo la verifica con un'altra base a, altrimenti n non è primo
                        mpz_sub_ui(exp, n, 1);
                        mpz_fdiv_q_ui(exp, exp, 2); // exp = (n-1)/2
                        exp_mod(c, a, exp, n); // calcoliamo a^((n-1)/2) mod n
                        mpz_sub_ui(n1, n, 1); // la classe -1 corrisponde alla classe n-1
                        if (mpz_cmp_si(c, 1)!=0 && mpz_cmp(c, n1)!=0) {
                                gmp_printf("%Zd non è un numero primo\n", n);

                                mpz_clears(a, it, d, c, exp, n1, NULL);
                                gmp_randclear(state);

                                return false;
                        } else {
                                RandNumber(a, n, state);
                                mpz_add_ui(it, it, 1); // it = it + 1
                        }
                }
        }

        // Pulizia della memoria
        mpz_clears(a, it, d, c, exp, n1, NULL);
        gmp_randclear(state);

        return true;
}

void RandNumber(mpz_t a, mpz_t n, gmp_randstate_t state) {
    mpz_t range;
    mpz_init(range);

    mpz_sub_ui(range, n, 3);  // range = (n - 2) - 2 + 1 = n - 3

    // Genera un numero casuale in [0, range]
    mpz_urandomm(a, state, range);
    // Sposta il numero generato nell'intervallo [2, n-2]
    mpz_add_ui(a, a, 2);

    mpz_clear(range);
}

// calcola in x la base b elevata alla potenza exp>=0 modulo n (exp binario)
void exp_mod (mpz_t x, mpz_t b, mpz_t exp, mpz_t n) {
        mpz_t squares, e;
        mpz_init_set(squares, b);
        mpz_init_set(e, exp);
        mpz_set_si(x, 1);

        while (mpz_cmp_si(e, 0)>0) { // finchè e > 0
                if (mpz_tstbit(e, 0)) { // se bit è 1 moltiplico x per il quadrato (mod n), altrimenti lascio così
```

```
                    mpz_mul(x, x, squares);
                    mpz_mod(x, x, n);
                }
                mpz_mul(squares, squares, squares); // aggiorno il quadrato (mod n)
                mpz_mod(squares, squares, n); // squares mod n
                mpz_fdiv_q_2exp(e, e, 1); // shifto di 1 i bit di e
        }

        mpz_clears(squares, e, NULL);

        return;
}
```

2.3.3 Test di Solovay-Strassen

Con questo metodo si raffina la tecnica del test di Eulero sfruttando le proprietà del simbolo di Jacobi e del simbolo di Legendre. Nel corso infatti è stato dimostrato che se p è primo allora

$$a^{\frac{p-1}{2}} \equiv \left(\frac{a}{p}\right) \; mod \; p$$

Come per gli altri test, fissato un intero n, basta dunque trovare un a che falsifica la relazione sopra per ottenere la non-primalità.
Nota: per questo test non esistono pseudoprimi, infatti per ogni scelta di n almeno la metà degli $a \in \{2, ..., n-1\}$ fà da testimone.

Listing 16: Solovay-Strassen

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

// Funzione per calcolare il massimo comune divisore (MCD)
void mcd(mpz_t result, mpz_t a, mpz_t b) {
    mpz_gcd(result, a, b);
}

// Esponenziazione modulare binaria
type void moduloExponentiation(mpz_t result, mpz_t base, mpz_t exp, mpz_t mod) {
    mpz_powm(result, base, exp, mod);
}

// Calcolo del simbolo di Jacobi
int jacobiSymbol(mpz_t a, mpz_t n) {
    return mpz_jacobi(a, n);
}

// Test di Solovay-Strassen
int solovayStrassen(mpz_t n, int iterations) {
    if (mpz_cmp_ui(n, 2) < 0) return 0;
    if (mpz_cmp_ui(n, 2) == 0) return 1;
    if (mpz_even_p(n)) return 0;

    gmp_randstate_t state;
    gmp_randinit_mt(state);
    gmp_randseed_ui(state, rand());

    mpz_t a, n_minus1, exp, gcd, modExp;
    mpz_inits(a, n_minus1, exp, gcd, modExp, NULL);

    mpz_sub_ui(n_minus1, n, 1);
    mpz_fdiv_q_ui(exp, n_minus1, 2);
```

```c

    for (int i = 0; i < iterations; i++) {
        mpz_urandomm(a, state, n_minus1);
        mpz_add_ui(a, a, 2);

        mcd(gcd, a, n);
        if (mpz_cmp_ui(gcd, 1) != 0) return 0;

        moduloExponentiation(modExp, a, exp, n);
        int jacobi = jacobiSymbol(a, n);
        if (jacobi == 0 || mpz_cmp_ui(modExp, (jacobi + mpz_get_ui(n)) % mpz_get_ui
            (n)) != 0) return 0;
    }

    mpz_clears(a, n_minus1, exp, gcd, modExp, NULL);
    gmp_randclear(state);
    return 1;
}

// Main
int main() {
    mpz_t n;
    int iterations;

    mpz_init(n);

    printf("Inserisci il numero da testare: ");
    gmp_scanf("%Zd", n);

    printf("Inserisci il numero di iterazioni k: ");
    scanf("%d", &iterations);

    if (solovayStrassen(n, iterations)) {
        gmp_printf("%Zd e' molto probabilmente primo.\n", n);
    } else {
        gmp_printf("%Zd non e' primo.\n", n);
    }

    mpz_clear(n);
    return 0;
}
```

2.3.4 Test di Miller-Rabin

Sia $n = 2^h d + 1$ con d dispari e, fissato a, si consideri $\alpha = a^d$ e la successione

$$(\alpha, \alpha^2, ..., \alpha^{n-1})$$

Se $\alpha \neq 1$ (si pensi tutto *mod* n) ci sono 2 casi possibili:

Caso 1): nella successione non appare mai 1. In tal caso fallisce il test di Fermat ed n risulta composto.

Caso 2): esiste un primo elemento della successione che vale 1 (i successivi saranno necessariamente tutti 1). Allora, chiamando β l'elemento precedente al primo 1, necessariamente

$$\beta = -1$$

Se quindi si trova un a per cui $\beta \neq -1$ il test afferma che n non è primo.
Nota: i testimoni di Solovay-Strassen lo sono anche per questo test, inoltre almeno tre quarti dei possibili a sono testimoni di Miller-Rabin.

Listing 17: Miller-Rabin

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

// Calcolo MCD con algoritmo di Euclide
void mcd(mpz_t result, const mpz_t a, const mpz_t b) {
    mpz_gcd(result, a, b);
}

// Esponenziazione modulare binaria
void moduloExponentiation(mpz_t result, const mpz_t base, const mpz_t exp, const mpz_t mod) {
    mpz_powm(result, base, exp, mod);
}

// Funzione ausiliaria per verificare se n è composto
int isComposite(const mpz_t n, const mpz_t a, const mpz_t d, long h) {
    mpz_t x, prev, temp;
    mpz_inits(x, prev, temp, NULL);

    // x = a^d mod n
    moduloExponentiation(x, a, d, n);

    // Se x == 1 o x == n - 1, non posso concludere che sia composto
    mpz_sub_ui(temp, n, 1);
    if (mpz_cmp_ui(x, 1) == 0 || mpz_cmp(x, temp) == 0) {
        mpz_clears(x, prev, temp, NULL);
        return 0;
    }

    // Itero attraverso le squadrature
    for (long i = 1; i < h; i++) {
        mpz_set(prev, x);
        mpz_mul(x, x, x);
        mpz_mod(x, x, n);

        // Se x == 1 e prev != n - 1, allora è composto
        if (mpz_cmp_ui(x, 1) == 0) {
            if (mpz_cmp(prev, temp) != 0) {
                mpz_clears(x, prev, temp, NULL);
                return 1;
            } else {
                mpz_clears(x, prev, temp, NULL);
                return 0;
            }
        }

        // Se x == n - 1, esco dal ciclo
        if (mpz_cmp(x, temp) == 0) {
            mpz_clears(x, prev, temp, NULL);
            return 0;
        }
    }

    // Se dopo tutte le iterazioni x ? 1, allora è sicuramente composto
    if (mpz_cmp_ui(x, 1) != 0) {
        mpz_clears(x, prev, temp, NULL);
        return 1;
    }

    mpz_clears(x, prev, temp, NULL);
    return 1;
}

// Test di Miller-Rabin
int millerRabin(const mpz_t n, int iterations) {
    if (mpz_cmp_ui(n, 2) < 0) return 0;
    if (mpz_cmp_ui(n, 2) == 0 || mpz_cmp_ui(n, 3) == 0) return 1;
    if (mpz_even_p(n)) return 0;
```

```c

    mpz_t d, a, g, temp;
    mpz_inits(d, a, g, temp, NULL);

    // Scriviamo n - 1 come 2^h * d
    mpz_sub_ui(d, n, 1);
    long h = 0;
    while (mpz_even_p(d)) {
        mpz_fdiv_q_2exp(d, d, 1);
        h++;
    }

    gmp_randstate_t state;
    gmp_randinit_mt(state);

    for (int i = 0; i < iterations; i++) {
        // Genera un numero casuale a tra [2, n-2]
        mpz_sub_ui(temp, n, 4);
        mpz_urandomm(a, state, temp);
        mpz_add_ui(a, a, 2);

        // Controllo MCD
        mcd(g, a, n);
        if (mpz_cmp_ui(g, 1) > 0) {
            mpz_clears(d, a, g, temp, NULL);
            gmp_randclear(state);
            return 0;
        }

        if (isComposite(n, a, d, h)) {
            mpz_clears(d, a, g, temp, NULL);
            gmp_randclear(state);
            return 0;
        }
    }

    mpz_clears(d, a, g, temp, NULL);
    gmp_randclear(state);
    return 1;
}

int main() {
    mpz_t n;
    int iterations;

    mpz_init(n);

    printf("Inserisci un numero da verificare: ");
    gmp_scanf("%Zd", n);

    printf("Inserisci il numero di iterazioni del test: ");
    scanf("%d", &iterations);

    if (millerRabin(n, iterations)) {
        gmp_printf("%Zd è molto probabilmente primo.\n", n);
    } else {
        gmp_printf("%Zd è composto.\n", n);
    }

    mpz_clear(n);
    return 0;
}
```

2.4 Fattorizzazione di numeri composti

Una volta scoperto che n è un numero composto si può pensare di cercare un suo divisore proprio. Di seguito sono riportati i metodi visti nel corso.

2.4.1 Metodo rho di Pollard

Fissato un primo p ed un numero composto n si cercano due numeri $x, y \in \mathbb{Z}/(n)$ tali che $x \not\equiv y \; mod \; n$ e $x \equiv y \; mod \; p$. Se ciò avviene allora esiste $1 < d < n$ tale che

$$(n, x-y) = (d) \Rightarrow d|n$$

Un modo efficace per cercare x ed y è creare due successioni di valori tramite una funzione di partenza (per esempio $x^2 + x + 1$) e controllare la condizione sopra ad ogni iterazione, creando la caratteristica forma di rho. Nel seguente codice il metodo è implementato seguendo la strategia di Floyd, la quale permette di avere una convergenza più rapida:

Listing 18: Rho di Pollard.c

```c
#include <stdio.h>
#include <stdbool.h>
#include <gmp.h>
#include "..\Include\mcd.h"

bool rho_pollard (mpz_t, mpz_t);
void funz_f (mpz_t, mpz_t, mpz_t);
void funz_g (mpz_t, mpz_t, mpz_t);

int main () {
        mpz_t n, d, r;
        mpz_inits(n, d, r, NULL);
        printf("Inserire il numero non primo da fattorizzare: n = ");
        gmp_scanf("%Zd", n);

        mpz_fdiv_r_ui (r, n, 2);
        if (mpz_cmp_si(r,0)==0) { // se il numero è pari allora 2 è un divisore
            proprio
                gmp_printf("Un divisore di %Zd è d = 2\n", n);
        } else {
                if (rho_pollard(d, n)) {
                        gmp_printf("Un divisore di %Zd è d = %Zd\n", n, d);
                } else {
                        gmp_printf("Non è stato trovato un divisore di %Zd\n", n);
                }
        }

        // Pulizia della memoria
        mpz_clears(n, d, r, NULL);

        return 0;
}

// Fattorizza l'intero positivo n utilizzando l'algoritmo rho di Pollard:
// - true = d è un divisore proprio di n
// - false = non ha trovato un divisore di n
bool rho_pollard (mpz_t d, mpz_t n) {
        mpz_t x, y, f, g, x_new, it;
        mpz_inits(x, y, f, g, x_new, it, NULL);

        // Assegniamo i valori iniziali x, y e it
        mpz_set_si(x, 1); // poniamo x = 1
        mpz_set_si(y, 1); // poniamo y = 1
        mpz_set_si(it, 0); // poniamo it = 0

        while (mpz_cmp(it, n)<0) { // facciamo massimo n iterazioni
                // Calcoliamo f(x) e g(x)
```

```
                funz_f(f, x, n);
                funz_g(g, y, n);

                // calcoliamo f(x)-g(x) mod n
                mpz_sub(x_new, g, f); // x_new = g - f
                mpz_mod(x_new, x_new, n); // x_new = x_new mod n
                mcd_euclide(d, x_new, n); // d = MCD(x_new, n)

                if (mpz_cmp_si(d, 1)!=0 && mpz_cmp(d, n)!=0) { // se d è diverso da 1 ho trovato un divisore di n
                        return true;
                }

                mpz_set(x, f); // la nuova x è f(x) calcolato prima
                mpz_set(y, g); // la nuova y è g(y) calcolato prima
                mpz_add_ui(it, it, 1); // it = it + 1
        }

        // Pulizia della memoria
        mpz_clears(x, y, f, g, x_new, it, NULL);

        return false;
}

void funz_f (mpz_t result, mpz_t x, mpz_t n) {
        mpz_t temp;
        mpz_init(temp);

        mpz_mul(temp, x, x);  // temp = x * x
        mpz_add_ui(result, temp, 1);  // result = temp + 1

        mpz_mod (result, result, n); // calcolo result mod n

        // Pulizia memoria
        mpz_clear(temp);
}

void funz_g (mpz_t result, mpz_t x, mpz_t n) {
        mpz_t temp, temp2, temp4;
        mpz_inits(temp, temp2, temp4, NULL);

        mpz_mul(temp2, x, x);  // temp2 = x * x (calcolo x^2)
        mpz_mul(temp4, temp2, temp2);  // temp4 = temp2 * temp2 (calcolo x^4)
        mpz_mul_ui(temp2, temp2, 2);  // temp2 = temp2 * 2 (calcolo 2*x^2)
        mpz_add(temp, temp4, temp2);  // temp = temp4 + temp2 (calcolo x^4+2*x^2)
        mpz_add_ui(result, temp, 2);  // result = temp + 1 (calcolo x^4+2*x^2+1)

        mpz_mod (result, result, n); // calcolo result mod n

        // Pulizia memoria
        mpz_clears(temp, temp2, temp4, NULL);
}
```

2.4.2 Basi di primi di Pomerance

La chiave di questo metodo è il fatto che scomporre un numero dispari equivale a scriverlo come differenza di quadrati: se $n = a^2 - b^2$ allora n ha come divisori $a + b$ e $a - b$. Se invece $n = d * e$ allora posso risolvere il sistema lineare

$$a + b = d \ , \ a - b = e$$

il quale ammette come soluzione un'unica coppia (a, b).
Si scelgano dunque $a, b \in \{0, ..., n - 1\}$ distinti, allora $(a - b, n) \in \{1, d\}$ con d divisore proprio di n. Bisogna quindi scegliere i due valori affinchè risolvano

$$a^2 - b^2 \equiv 0 \ mod \ n$$

in modo non banale. Per fare ciò si seleziona una base di primi arbitrariamente lunga e dei valori i cui quadrati sono "piccoli" modulo n, ottenibili attraverso lo sviluppo in frazione continua (facilmente implementabile nel caso di radici di interi.

Listing 19: Basi primi Pomerance.h

```
#include <stdlib.h>
#include <stdbool.h>
#include <gmp.h>
#include "mcd.h"

// DISCLAIMER: questa versione a volte non funziona correttamente: per certi input sembra restituire come divisore 1 anziche' un divisore proprio;
// sarebbero necessarie ulteriori indagini.

int basi_primi (mpz_t, mpz_t, mpz_t, unsigned long int);

// funzioni ausiliarie
mpz_t* cont_frac_sqrt_mod(mpz_t, mpz_t, mpz_t, mpz_t, mpz_t, mpz_t, unsigned long int);
void cont_frac_next_term(mpz_t, mpz_t, mpz_t, mpz_t, mpz_t, mpz_t, mpz_t);
mpz_t* squares_mod(mpz_t*, unsigned long int, mpz_t);
void riduci_mod_min(mpz_t, mpz_t);
unsigned long int check_null_column(bool*, int**, unsigned long int, unsigned long int);
void Gauss (bool**, bool**, unsigned long int, unsigned long int);
unsigned long int find_null_rows (bool**, unsigned long int, unsigned long int, unsigned long int);
void get_B (mpz_t, mpz_t, bool**, mpz_t*, int*, unsigned long int, unsigned long int);
void get_A (mpz_t, mpz_t, mpz_t*, int**, unsigned long int, bool**, int*, unsigned long int);

// Fattorizza l'intero positivo n utilizzando l'algoritmo delle basi di primi di Pomerance: se riesce restituisce 1 e salva in
// d un divisore proprio di n, altrimenti restituisce 0. Bound rappresenta il massimo valore che possono assumere i primi nella base,
// iter il numero di iterazioni eseguite dall'algoritmo.
int basi_primi (mpz_t d, mpz_t n, mpz_t bound, unsigned long int iter){
        int i,j;

    // controllo che n sia dispari
    mpz_t r;
    mpz_init(r);
    mpz_fdiv_r_ui(r,n,2);
    if (mpz_cmp_si(r,0)==0) {
        mpz_set_ui(d,2);
        mpz_clear(r);
        return 1;
    }
    mpz_clear(r);

    // controllo che n non sia un quadrato (serve per produrre i numeratori dello sviluppo in frazione continua di sqrt(n))
    mpz_t sqrt_n,temp;
    mpz_inits(sqrt_n,temp,NULL);
    mpz_sqrt(sqrt_n,n); // parte intera della radice di n
    mpz_mul(temp,sqrt_n,sqrt_n);
    if (mpz_cmp(temp,n)==0) {
        mpz_set(d,sqrt_n);
        mpz_clears(sqrt_n,temp,NULL);
        return 1;
    }
    mpz_clears(sqrt_n,temp,NULL);

        // Costruisco una base formata da -1, 2 e tutti i numeri dispari da 3 fino a bound (non saranno tutti primi; e' una scelta semplice ma non ottimale)
```

```c
    unsigned long int base_length = mpz_get_ui(bound);
    base_length=(base_length-1)/2 +2;
    mpz_t *base = (mpz_t*)malloc(base_length*sizeof(mpz_t));
    mpz_init_set_si(base[0],-1);
    mpz_init_set_si(base[1],2);
    for (i=2; i<base_length; i++) {
        mpz_init_set_si(base[i],2*i-1);
    }

    // la matrice M ((l+1)*l, con l=base_length) conterra', nell'entrata M[i][j], l'esponente massimo exp tale che base[j]^exp divide b[i]
    int** M=(int**)malloc((base_length+1)*sizeof(int*));
    for (i=0; i<base_length+1; i++) M[i]=(int*)malloc(base_length*sizeof(int));
    for (i=0; i<base_length+1; i++) for (j=0; j<base_length; j++) M[i][j]=0; // inizializzo a 0

    // dati iniziali delle relazioni ricorsive per lo sviluppo in frazione continua di radice di n
    mpz_t a,beta,gamma,bmeno1,bmeno2;
    mpz_inits(a,beta,gamma,bmeno1,bmeno2,NULL);
    mpz_sqrt(a,n);
    mpz_neg(beta,a);
    mpz_set_ui(gamma,1);
    mpz_set_ui(bmeno1,1);
    mpz_set_ui(bmeno2,0);

    unsigned long int it=0;
    while (it<iter) {
        // costruisco i b_i con quadrato piccolo mod n: ne voglio base_length+1 cosi' da assicurarmi almeno una combinazione lineare di righe
        // nulla nella matrice che costruiro' dopo
        mpz_t* b=cont_frac_sqrt_mod(n,bmeno1,bmeno2,a,beta,gamma,base_length+1);
        mpz_t* squares=squares_mod(b,base_length+1,n);

        // calcolo la matrice degli esponenti massimi dei primi che dividono ciascun quadrato
        for (i=0; i<base_length+1; i++) {
            // primo termine (corrisponde a -1 nella base)
            if (mpz_cmp_si(squares[i],0)<0) {
                M[i][0]=1;
                mpz_neg(squares[i],squares[i]);
            }

            // secondo termine (corrisponde a 2 nella base)
            while (mpz_divisible_ui_p(squares[i],2)) {
                mpz_divexact_ui(squares[i],squares[i],2);
                M[i][1]++;
            }

            // termini rimanenti (corrispondono ai dispari (2*j -1) nella base)
            for (j=2; j<base_length; j++) {
                while (mpz_divisible_ui_p(squares[i],2*j-1)) {
                    mpz_divexact_ui(squares[i],squares[i],2*j-1);
                    M[i][j]++;
                }
            }

            // controllo se il quadrato e' stato completamente scomposto nella base; in caso contrario lo sostituisco con un nuovo b
            // (e il suo quadrato) e ritento la scomposizione
            if (mpz_cmp_si(squares[i],1)!=0) {
                cont_frac_next_term(b[i],n,bmeno1,bmeno2,a,beta,gamma);
                mpz_mul(squares[i],b[i],b[i]);
                riduci_mod_min(squares[i],n);
                i--; // attenzione: in questo modo il ciclo for su i potrebbe non terminare: aggiungiamo una qualche condizione?
            }
        }

        // conto le colonne nulle poiche' possono essere ignorate nell'eliminazione
```

```
    di Gauss, facendo risparmiare spazio e tempo
    unsigned long int num_null_cols;
    bool* is_col_zeros=(bool*)malloc(base_length*sizeof(bool));
    num_null_cols = check_null_column(is_col_zeros,M,base_length+1,base_length);
    unsigned long int num_cols_M_mod2 = base_length-num_null_cols;
    // creo un vettore di "differenze cumulative", lungo come il numero di colonne di M_mod2, che tiene conto di quante colonne sono state
    // rimosse fino a quel punto: l'idea e' che, se sto guardando la colonna j di M_mod2, questa corrisponde alla colonna (j+cumul_diff[j]) di M,
    // e cioe' all'elemento della base di indice (j+cumul_diff[j]).
    int* cumul_diff=(int*)malloc(num_cols_M_mod2*sizeof(int));
    int contatore=0;
    i=0; j=0;
    while (j<num_cols_M_mod2) {
        while (is_col_zeros[i]) {
            contatore++;
            i++;
            if (i>base_length) break;
        }
        cumul_diff[j]=contatore;
        i++; j++;
    }
    // creo la matrice booleana M_mod2 che contiene i coefficienti di M ridotti modulo 2, escluse le colonne nulle (true=1, false=0).
    // sara' una matrice di dimensioni (base_length+1)*(base_length-num_null_cols).
    bool** M_mod2=(bool**)malloc((base_length+1)*sizeof(bool*));
    for (i=0; i<base_length+1; i++) M_mod2[i]=(bool*)malloc(num_cols_M_mod2*sizeof(bool));
    for (i=0; i<base_length+1; i++) {
        for (j=0; j<num_cols_M_mod2; j++) {
            M_mod2[i][j]=M[i][j+cumul_diff[j]] % 2;
        }
    }

    // costruisco la matrice identità da affiancare a M_mod2 prima di applicare Gauss
    bool** Id=(bool**)malloc((base_length+1)*sizeof(bool*));
    for (i=0; i<base_length+1; i++) Id[i]=(bool*)malloc((base_length+1)*sizeof(bool));
    for (i=0; i<base_length+1; i++) for (j=0; j<base_length+1; j++) Id[i][j]=false; // inizializzo a 0
    for (i=0; i<base_length+1; i++) Id[i][i]=true; // pongo la diagonale uguale ad 1

    // applico Gauss e calcolo A e B
    Gauss(M_mod2, Id, base_length+1, num_cols_M_mod2);
    unsigned long int index=0;
    unsigned long int null_row=0;
    mpz_t A, B, menoB, ApiuB;
        mpz_inits(A, B, menoB, ApiuB, NULL);
    while (index<(base_length+1)) {
                null_row=find_null_rows(M_mod2, base_length+1, num_cols_M_mod2, index);
                get_B(B, n, Id, b, cumul_diff, null_row, (base_length+1));
                    get_A(A, n, base, M, base_length, Id, cumul_diff, null_row);
        mpz_sub(menoB,n,B);
        mpz_fdiv_r(menoB,menoB,n);

                if (mpz_cmp(A, B)==0 || mpz_cmp(A, menoB)==0) { // se A=B oppure A=-B mod n, l'algoritmo è fallito
                    if (index!=base_length) { // se non abbiamo esaurito le righe riproviamo con un'altra combinazione
                            index++;
                    } else { // altrimenti ricominciamo da capo
                            break; // in questo modo rientriamo nel ciclo while (it<iter)
                    }
```

```
                    } else { // se invece se A!=B oppure A!=-B mod n concludo l'algoritmo calcolando d=MCD(A+B, n)
                                mpz_add(ApiuB, A, B);
                                mcd_euclide(d, ApiuB, n);

/*                  if (mpz_cmp_si(d,1)==0) {//// sarebbe da rimuovere; e' solo per evitare che l'algoritmo (sbagliato) restituisca 1
                        index++;
                        continue;
                    }
*/

                // pulizia memoria
                                mpz_clears(A, B, menoB, ApiuB, NULL);
                free(is_col_zeros); free(cumul_diff);
                for (i=0; i<base_length+1; i++) free(M_mod2[i]); free(M_mod2);
                for (i=0; i<base_length+1; i++) free(Id[i]); free(Id);
                for (i=0; i<base_length+1; i++) mpz_clear(b[i]); free(b);
                for (i=0; i<base_length+1; i++) mpz_clear(squares[i]); free(squares);
                for (i=0; i<base_length+1; i++) free(M[i]); free(M);
                for (i=0; i<base_length; i++) mpz_clear(base[i]); free(base);
                mpz_clears(a,beta,gamma,bmeno1,bmeno2,NULL);
                                return 1;
                }
                }
        // pulizia memoria
                mpz_clears(A, B, menoB, ApiuB, NULL);
        free(is_col_zeros); free(cumul_diff);
        for (i=0; i<base_length+1; i++) free(M_mod2[i]); free(M_mod2);
        for (i=0; i<base_length+1; i++) free(Id[i]); free(Id);
        for (i=0; i<base_length+1; i++) mpz_clear(b[i]); free(b);
        for (i=0; i<base_length+1; i++) mpz_clear(squares[i]); free(squares);
        it++;
    }

    // se sono arrivato qui ho completato tutte le iterazioni senza trovare un divisore: fallimento.

    for (i=0; i<base_length+1; i++) free(M[i]); free(M);
    for (i=0; i<base_length; i++) mpz_clear(base[i]); free(base);
    mpz_clears(a,beta,gamma,bmeno1,bmeno2,NULL);
    return 0;
}

// Restituisce k>0 numeratori dell'approssimazione in frazione continua di sqrt(n) modulo n, espressi con valore assoluto minimo, a partire
// dai due numeratori precedenti e dai corrispondenti valori di a, beta, gamma (che vengono modificati durante l'esecuzione).
mpz_t* cont_frac_sqrt_mod(mpz_t n, mpz_t bmeno1, mpz_t bmeno2, mpz_t a, mpz_t beta, mpz_t gamma, unsigned long int k) {
    int i;
    mpz_t sqrt_n,temp;
    mpz_inits(sqrt_n,temp,NULL);
    mpz_sqrt(sqrt_n,n);
    mpz_t* b=(mpz_t*)malloc(k*sizeof(mpz_t));
    for (i=0;i<k;i++) mpz_init(b[i]);

    // esplicito i primi due termini (poiché utilizzano dati iniziali specifici su b_-1 e b_-2)

    mpz_mul(temp,a,bmeno1);
    mpz_add(b[0],temp,bmeno2); // aggiornato b[0]
    mpz_mul(temp,beta,beta);
    mpz_sub(temp,n,temp);
    mpz_fdiv_q(gamma,temp,gamma); // aggiornato gamma
    mpz_sub(temp,sqrt_n,beta);
    mpz_fdiv_q(a,temp,gamma); // aggiornato a
    mpz_mul(temp,a,gamma);
    mpz_add(temp,temp,beta);
```

```
    mpz_neg(beta,temp); // aggiornato beta
    riduci_mod_min(b[0],n);

    if (k==1) {
        mpz_set(bmeno2,bmeno1);
        mpz_set(bmeno1,b[0]);
        return b;
    }

    mpz_mul(temp,a,b[0]);
    mpz_add(b[1],temp,bmeno1); // aggiornato b[1]
    mpz_mul(temp,beta,beta);
    mpz_sub(temp,n,temp);
    mpz_fdiv_q(gamma,temp,gamma); // aggiornato gamma
    mpz_sub(temp,sqrt_n,beta);
    mpz_fdiv_q(a,temp,gamma); // aggiornato a
    mpz_mul(temp,a,gamma);
    mpz_add(temp,temp,beta);
    mpz_neg(beta,temp); // aggiornato beta
    riduci_mod_min(b[1],n);

    if (k==2) {
        mpz_set(bmeno2,b[0]);
        mpz_set(bmeno1,b[1]);
        return b;
    }

    for (i=2; i<k; i++) {
        mpz_mul(temp,a,b[i-1]);
        mpz_add(b[i],temp,b[i-2]); // aggiornato b[i]
        mpz_mul(temp,beta,beta);
        mpz_sub(temp,n,temp);
        mpz_fdiv_q(gamma,temp,gamma); // aggiornato gamma
        mpz_sub(temp,sqrt_n,beta);
        mpz_fdiv_q(a,temp,gamma); // aggiornato a
        mpz_mul(temp,a,gamma);
        mpz_add(temp,temp,beta);
        mpz_neg(beta,temp); // aggiornato beta
        riduci_mod_min(b[i],n);
    }

    mpz_set(bmeno2,b[k-2]);
    mpz_set(bmeno1,b[k-1]);

    mpz_clears(sqrt_n,temp,NULL);
    return b;
}

// salva in b il termine successivo dello sviluppo in frazione continua di sqrt(n) modulo n, espresso con valore assoluto minimo, a partire
// dai due numeratori precedenti e dai corrispondenti valori di a, beta, gamma (che vengono modificati durante l'esecuzione).
void cont_frac_next_term(mpz_t b, mpz_t n, mpz_t bmeno1, mpz_t bmeno2, mpz_t a, mpz_t beta, mpz_t gamma) {
    mpz_t sqrt_n,temp;
    mpz_inits(sqrt_n,temp,NULL);
    mpz_sqrt(sqrt_n,n);

    mpz_mul(temp,a,bmeno1);
    mpz_add(b,temp,bmeno2); // aggiornato b
    mpz_mul(temp,beta,beta);
    mpz_sub(temp,n,temp);
    mpz_fdiv_q(gamma,temp,gamma); // aggiornato gamma
    mpz_sub(temp,sqrt_n,beta);
    mpz_fdiv_q(a,temp,gamma); // aggiornato a
    mpz_mul(temp,a,gamma);
    mpz_add(temp,temp,beta);
    mpz_neg(beta,temp); // aggiornato beta
    riduci_mod_min(b,n);

```

```
    mpz_set(bmeno2,bmeno1);
    mpz_set(bmeno1,b); // aggiornati i termini precedenti

    mpz_clears(sqrt_n,temp,NULL);
    return;
}

// restituisce un array con i quadrati degli elementi nell'array in input (lungo k) modulo n, espressi con valore assoluto minimo.
mpz_t* squares_mod(mpz_t* array, unsigned long int k, mpz_t n) {
    int i;
    mpz_t* squares=(mpz_t*)malloc(k*sizeof(mpz_t));
    for (i=0;i<k;i++){
        mpz_init(squares[i]);
        mpz_mul(squares[i],array[i],array[i]);
        riduci_mod_min(squares[i],n);
    }

    return squares;
}

// riduce a modulo n esprimendolo con valore assoluto minimo (potra' essere positivo o negativo, ma avra' abs(a)<= n/2).
void riduci_mod_min(mpz_t a, mpz_t n){
    mpz_t n_mezzi;
    mpz_init(n_mezzi);
    mpz_fdiv_q_2exp(n_mezzi,n,1);

    mpz_fdiv_r(a,a,n);
    if (mpz_cmp(a,n_mezzi)>0) mpz_sub(a,a,n); // se a>n/2, allora a-->a-n

    mpz_clear(n_mezzi);
    return;
}

// per una matrice m*n, salva in is_zero[j] true se la colonna j di M e' tutta nulla, false altrimenti; restituisce il numero di colonne nulle.
unsigned long int check_null_column(bool* is_zero, int** M, unsigned long int m, unsigned long int n) {
    int i,j;
    unsigned long int num_null_columns=0;
    for (j=0; j<n; j++) {
        is_zero[j]=true;
        num_null_columns++;
        for (i=0; i<m; i++) {
            if (M[i][j]!=0) {
                is_zero[j]=false;
                num_null_columns--;
                break;
            }
        }
    }
    return num_null_columns;
}

//eliminazione di Gauss applicata alla matrice M_mod2 i cui passaggi vengono ripetuti anche su una matrice identità di dimensione pari al numero di righe di M_mod2
void Gauss(bool **M, bool **Id, unsigned long int rows, unsigned long int cols) {
    int row=0;
    for (int col=0; col<cols && row<rows; col++) {
        // Trova il pivot nella colonna corrente
        int pivot=-1;
        for (int i=row; i<rows; i++) {
            if (M[i][col]) {
                pivot=i;
                break;
            }
        }

```

```
        // Se non troviamo un pivot, passiamo alla colonna successiva
        if (pivot==-1) {
                continue;
        }

        // Scambia la riga pivot con la riga attuale
        if (pivot!=row) {
            bool *temp=M[pivot];
            M[pivot]=M[row];
            M[row]=temp;

            temp=Id[pivot];
            Id[pivot]=Id[row];
            Id[row]=temp;
        }

        // Elimina i valori nelle altre righe (comando XOR)
        for (int i=0; i<rows; i++) {
            if (i!=row && M[i][col]) {
                for (int j=0; j<cols; j++) {
                    M[i][j] ^= M[row][j];
//                    Id[i][j] ^= Id[row][j];
                }
                for (int j=0; j<rows; j++) { // forse devo fare cosi'? Il fatto e' che Id ha piu' colonne di M
                    Id[i][j] ^= Id[row][j];
                }
            }
        }

        // Passa alla prossima riga
        row++;
    }
    return;
}

// troviamo la prima riga nulla di M_mod2 dopo aver applicato Gauss, l'indice indica da quela riga cominciare a fare la ricerca
unsigned long int find_null_rows (bool** mat, unsigned long int rows, unsigned long int cols, unsigned long int index) {
        unsigned long int null_row=rows; // inizializzo ad un intero che non può assumere
        for (int i=index; i<rows; i++) {
                for (int j=0; j<cols; j++) {
                        if (mat[i][j]==true) { // appena incontro un 1 interrompo il ciclo sulle colonne
                                break;
                        } else if (j==cols-1 && mat[i][j]==false) { // se ho controllato tutte le colonne modifico null_row
                                null_row=i;
                        }
                }
                if (null_row!=rows) { // se ho modificato null_row ho trovato una riga nulla, posso interrompere la ricerca
                        break;
                }
        }
        return null_row;
}

// B equivale al prodotto dei vari b[i] con i indice trovato con gauss, ossia i=j+cumul_diff[j] se Id[null_row][j]==true
void get_B (mpz_t B, mpz_t n, bool** Id, mpz_t* b, int* cumul_diff, unsigned long int null_row, unsigned long int cols) {
        mpz_set_si(B, 1);
        for (int j=0; j<cols; j++){
                if (Id[null_row][j]==true) {
//                        mpz_mul(B, B, b[j+cumul_diff[j]]);
            mpz_mul(B, B, b[j]); // i vari 1 nella matrice identita' si trovano sulla colonna j se devo considerare la riga j della matrice M,
```

```
                                        // ossia il j-esimo elemento dei b; giusto? Anche
                                            perche' get_B viene chiamata con cols=
                                            base_length+1, e b e'
                                        // lungo base_length+1, cosi' come la dimensione
                                            di Id

                                mpz_fdiv_r(B,B,n);
                        }
        }
        return;
}

void get_A (mpz_t A, mpz_t n, mpz_t* base, int** M, unsigned long int base_length,
    bool** Id, int* cumul_diff, unsigned long int null_row) {
        mpz_set_si(A, 1);
        // calcolo exp_tot, ossia sommo le righe i di M (i sono quelli ricavati da
            gauss)
        int* exp_tot=(int*)malloc(base_length*sizeof(int));
        for (int i=0; i<base_length; i++) exp_tot[i]=0; // inizializzo a zero
        for (int i=0; i<(base_length+1); i++) { // scorriamo la riga null_row di Id
            , se nella colonna i l'elemento è 1 (ossia true)
                if (Id[null_row][i]==true) { // allora consideriamo la riga i+
                    cumul_diff[i] di M
                        for (int j=0; j<base_length; j++) {
//                              exp_tot[j]=exp_tot[j]+M[i+cumul_diff[i]][j];
                exp_tot[j]=exp_tot[j]+M[i][j]; // ragionamento analogo a quello per
                     get_B: la matrice M ha dimensione (base_length+1 ) *
                    base_length
                        }
                }
        }

        // ora calcoliamo A=base[i]^(exp_tot[i]/2)
        mpz_t temp;
        mpz_inits(temp, NULL);
        for (int i=0; i<base_length; i++) {
                mpz_powm_ui(temp, base[i], exp_tot[i]/2, n);
                mpz_mul(A, A, temp);
                mpz_fdiv_r(A,A,n);
        }

        mpz_clears(temp, NULL);
        free(exp_tot);

        return;
}
```

2.4.3 Metodo $p-1$ di Pollard

Un numero $n = p_1^{e_1}...p_k^{e_k}$ è detto ***b*-liscio**, se esiste un intero b tale che

$$p_i^{e_i} \leq b \ \forall i \in \{1, ..., k\}$$

Il codice che segue trova un divisore di n scommettendo sul fatto che almeno un gruppo ciclico Gp generato da p fattore primo di n sia tale che la cardinalità di G_P sia b-liscia n.

Listing 20: Pmeno1 Pollard.c

```
#include <stdio.h>
#include <stdbool.h>
```

```c
#include <string.h>
#include <math.h>
#include <gmp.h>
#include "..\Include\mcd.h"

bool p1_pollard (mpz_t, mpz_t);
void exp_mod (mpz_t, mpz_t, mpz_t, mpz_t);

int main () {
	mpz_t n, d, r;
	mpz_inits(n, d, r, NULL);
	printf("Inserire il numero non primo da fattorizzare: n = ");
	gmp_scanf("%Zd", n);

	mpz_fdiv_r_ui (r, n, 2);
	if (mpz_cmp_si(r,0)==0) { // se il numero è pari allora 2 è un divisore proprio
		gmp_printf("Un divisore di %Zd è d = 2\n", n);
	} else {
		if (p1_pollard(d, n)) {
			gmp_printf("Un divisore di %Zd è d = %Zd\n", n, d);
		} else {
			gmp_printf("Non è stato trovato un divisore di %Zd\n", n);
		}
	}

	// Pulizia della memoria
	mpz_clears(n, d, r, NULL);

	return 0;
}

bool p1_pollard (mpz_t d, mpz_t n) {
	mpz_t b, b0, b1, a, m, y, it;
	mpz_inits(b, b0, b1, a, m, y, it, NULL);
	mpz_set_si(a, 2);
	mpz_set_si(m, 1);
	mpz_set_si(b, 2);
	int primes[18] = {2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61};

	while (mpz_cmp(b, n)<0) {
		// cerco m tc m sia la produttoria su i di p_i^e_i con p_i primi e p_i^e_i < b
		// tuttavia essendo la base di primi limitata, da 61 in poi tutti per tutti i primi è come se prendessimo e_i=0
		for (int i=0; i<18; i++) {
			if (mpz_cmp_si(b, primes[i])<0) { // appena b > primes[i] interrompo il for
				break;
			} else { // finchè b < primes(i)
				int e;
				e=pow(primes[i], strlen(mpz_get_str(NULL,primes[i], b))-1);
				mpz_mul_ui(m, m, e);  // m = m * e
			}
		}

		exp_mod(y, a, m, n);

		mpz_sub_ui(y, y, 1);

		mcd_euclide(d, y, n); // d = MCD(a^m-1, n)
		if (mpz_cmp_si(d, 1)!=0 && (mpz_cmp(d, n)!=0)) {
			return true;
		}

		mpz_add_ui(b, b, 1); // it = it + 1
	}

```

```
        mpz_clears(b, b0, b1, a, m, y, it, NULL);

        return false;
}

// calcola in x la base b elevata alla potenza exp>=0 modulo n (exp binario)
void exp_mod (mpz_t x, mpz_t b, mpz_t exp, mpz_t n) {
        mpz_t squares, e;
        mpz_init_set(squares, b);
        mpz_init_set(e, exp);
        mpz_set_si(x, 1);

        while (mpz_cmp_si(e, 0)>0) { // finchè e > 0
                if (mpz_tstbit(e, 0)) { // se bit è 1 moltiplico x per il quadrato
                    (mod n), altrimenti lascio così
                        mpz_mul(x, x, squares);
                        mpz_mod(x, x, n);
                }
                mpz_mul(squares, squares, squares); // aggiorno il quadrato (mod n)
                mpz_mod(squares, squares, n); // squares mod n
                mpz_fdiv_q_2exp(e, e, 1); // shifto di 1 i bit di e
        }

        mpz_clears(squares, e, NULL);

        return;
}
```

2.4.4 Algoritmo di Lenstra

Si basa sulla costruzione di una curva ellittica $y^2 = x^3 + ax + b$ a coefficienti nel campo $\mathbb{F}_p$ con p primo scelto in una lista predefinita. L'insieme di queste curve ha una struttura di gruppo naturale che può essere sfruttata per trovare un divisore di n.

Listing 21: Lenstra.h

```
#include <string.h>
#include <time.h>
#include "mcd.h"
#include "Bezout.h"

// Numero di primi da utilizzare nella fattorizzazione di n: deve essere compreso
    tra 1 e 18 (poiché ho salvato solo una lista dei primi 18 numeri primi)
#define NUM_PRIMES 4

int lenstra (mpz_t, mpz_t);
int lenstra_iter (mpz_t, mpz_t);

// funzioni ausiliarie per algoritmo di Lenstra
int double_elliptic (mpz_t, mpz_t, mpz_t, mpz_t, mpz_t, mpz_t, mpz_t);
int sum_elliptic (mpz_t, mpz_t, mpz_t, mpz_t, mpz_t, mpz_t, mpz_t, mpz_t);
int multiply_elliptic (mpz_t, mpz_t, mpz_t, mpz_t, mpz_t, int, mpz_t, mpz_t);
void hasse (mpz_t, mpz_t);
void create_exp_primes (unsigned int*, int*, mpz_t);
void calcola_b (mpz_t, mpz_t, mpz_t, mpz_t, mpz_t);
void calcola_delta (mpz_t, mpz_t, mpz_t, mpz_t);

// Fattorizza l'intero positivo n utilizzando l'algoritmo di Lenstra basato su
    curve ellittiche: se riesce restituisce 1 e salva in
```

```
// d un divisore proprio di n, altrimenti restituisce 0. Utilizza coefficienti e punti sulle curve ellittiche generati casualmente, ed esegue
// un numero di iterazioni fissato all'interno della funzione (attualmente 2^30).
int lenstra (mpz_t d, mpz_t n) {

    // salvo una lista di primi da utilizzare nel tentativo di fattorizzare n: scelgo i primi 18 poiché la funzione mpz_get_str, che uso
    // per calcolare il logaritmo base p che serve per la stima di Hasse, accetta come secondo argomento solo numeri da 2 a 62. Teoricamente si
    // può aumentare, ma bisogna trovare un modo alternativo per calcolare la stima, e in ogni caso solitamente bastano pochi primi.
    int primes[18] = {2,3,5,7,11,13,17,19,23,29,31,37,41,43,47,53,59,61};

    mpz_t r;
    mpz_init(r);
    // faccio due prime divisioni test: nell'algoritmo serve n dispari, e sul libro di Koblitz chiede che i primi che dividano n siano >3.
    mpz_fdiv_r_ui(r,n,2);
    if (mpz_cmp_si(r,0)==0) {
        mpz_set_ui(d,2);
        mpz_clear(r);
        return 1;
    }
    mpz_fdiv_r_ui(r,n,3);
    if (mpz_cmp_si(r,0)==0) {
        mpz_set_ui(d,3);
        mpz_clear(r);
        return 1;
    }
    mpz_clear(r);

    mpz_t hasse_n,x_P,y_P,a,b,delta,iter,maxiter; // x_P,y_P coordinate del punto P sulla curva ellittica y^2=x^3+ax+b
    mpz_inits(hasse_n,x_P,y_P,a,b,delta,iter,maxiter,NULL);
    unsigned int exp_primes[NUM_PRIMES];
    hasse(hasse_n,n);
    create_exp_primes(exp_primes,primes,hasse_n); // inizializzo gli esponenti a cui elevare ciascun primo
    mpz_clear(hasse_n);

    gmp_randstate_t randstate;
    gmp_randinit_default(randstate);
    unsigned long int t = time(NULL);
    gmp_randseed_ui(randstate, t);

    // inizio l'algoritmo di Lenstra

    mpz_set_ui(maxiter,1);
    mpz_mul_2exp(maxiter,maxiter,30); // testo 2^30 iterazioni

    while (mpz_cmp(iter,maxiter)<0) {

        mpz_urandomm(x_P,randstate,n);
        mpz_urandomm(y_P,randstate,n);
        mpz_urandomm(a,randstate,n);

        calcola_b(b,x_P,y_P,a,n); // calcolo b affinché P sia sulla curva ellittica
        calcola_delta(delta,a,b,n); // calcolo discriminante della curva ellittica (mod n)

        if (mpz_cmp_si(delta,0)==0) goto retry; // se delta=0 cambio a e riprovo

        mcd_binario(d,delta,n);

        if (mpz_cmp_si(d,1)!=0) { // se mcd(delta,n) diverso da 1, ho trovato un divisore proprio (ho escluso mcd=n nella condizione precedente)
            gmp_randclear(randstate);
            mpz_clears(x_P,y_P,a,b,delta,iter,maxiter,NULL);
            return 1;
        }
```

```
        // inizio i calcoli P --> m*P

        int i,j,outcome;
        for (i=0; i<exp_primes[0]; i++) { // calcolo P--> 2^(e_2)*P

            outcome=double_elliptic(d,x_P,y_P,x_P,y_P,a,n);
            if (outcome==-1) goto retry; // se non posso raddoppiare cambio parametri e riprovo
            if (outcome==0) { // successo: ho trovato in d un divisore proprio di n
                gmp_randclear(randstate);
                mpz_clears(x_P,y_P,a,b,delta,iter,maxiter,NULL);
                return 1;
            }
        }

        // calcolo le potenze con i primi della lista maggiori di 2
        for (j=1; j<NUM_PRIMES; j++) {
            for (i=0; i<exp_primes[j]; i++) { // calcolo P-->k^(e_k)*P, con k primo

                outcome=multiply_elliptic(d,x_P,y_P,x_P,y_P,primes[j],a,n);
                if (outcome==-1) goto retry; // se non posso moltiplicare cambio parametri e riprovo
                if (outcome==0) { // successo: ho trovato in d un divisore proprio di n
                    gmp_randclear(randstate);
                    mpz_clears(x_P,y_P,a,b,delta,iter,maxiter,NULL);
                    return 1;
                }
            }
        }

        // se arrivo qui ho calcolato mP senza trovare divisori: cambio parametri e riprovo

        retry: mpz_add_ui(iter,iter,1); // aggiorno l'indice di iterazione e cambio parametri
    }

    // se arrivo qui non ho trovato divisori di n nel numero di iterazioni fissate: fallimento

    gmp_randclear(randstate);
    mpz_clears(x_P,y_P,a,b,delta,iter,maxiter,NULL);
    return 0;
}

// Fattorizza l'intero positivo n utilizzando l'algoritmo di Lenstra basato su curve ellittiche: se riesce restituisce 1 e salva in
// d un divisore proprio di n, altrimenti restituisce 0. Versione iterativa: considera in ordine tutti i possibili punti e coefficienti
// per le curve ellittiche.
int lenstra_iter (mpz_t d, mpz_t n) {

    // salvo una lista di primi da utilizzare nel tentativo di fattorizzare n: scelgo i primi 18 poiché la funzione mpz_get_str, che uso
    // per calcolare il logaritmo base p che serve per la stima di Hasse, accetta come secondo argomento solo numeri da 2 a 62. Teoricamente si
    // può aumentare, ma bisogna trovare un modo alternativo per calcolare la stima, e in ogni caso solitamente bastano pochi primi.
    int primes[18] = {2,3,5,7,11,13,17,19,23,29,31,37,41,43,47,53,59,61};

    mpz_t r;
    mpz_init(r);
    // faccio due prime divisioni test: nell'algoritmo serve n dispari, e sul libro di Koblitz chiede che i primi che dividano n siano >3.
    mpz_fdiv_r_ui(r,n,2);
    if (mpz_cmp_si(r,0)==0) {
        mpz_set_ui(d,2);
        mpz_clear(r);
```

```
        return 1;
    }
    mpz_fdiv_r_ui(r,n,3);
    if (mpz_cmp_si(r,0)==0) {
        mpz_set_ui(d,3);
        mpz_clear(r);
        return 1;
    }
    mpz_clear(r);

    mpz_t hasse_n,x_P,y_P,x_iter,y_iter,a,b,delta; // x_P,y_P coordinate del punto P sulla curva ellittica y^2=x^3+ax+b
    mpz_inits(hasse_n,x_P,y_P,x_iter,y_iter,a,b,delta,NULL);
    unsigned int exp_primes[NUM_PRIMES];
    hasse(hasse_n,n);
    create_exp_primes(exp_primes,primes,hasse_n); // inizializzo gli esponenti a cui elevare ciascun primo
    mpz_clear(hasse_n);

    // inizio l'algoritmo di Lenstra

    // (ciclo su x_iter e y_iter poiché nel corso dell'algoritmo x_P e y_P vengono modificati)
    // ciclo sulla componente x del punto P da 0 a n-1
    while (mpz_cmp(x_iter,n)<0) {
        mpz_set_ui(y_iter,1);

        // ciclo sulla componente y del punto P da 1 a n-1 (parte da 1 altrimenti il primo raddoppio non è definito)
        while (mpz_cmp(y_iter,n)<0) {
            mpz_set_ui(a,0);

            // ciclo sulla scelta di a tra 0 e n-1
            while (mpz_cmp(a,n)<0) {

                mpz_set(x_P,x_iter);
                mpz_set(y_P,y_iter);

                calcola_b(b,x_P,y_P,a,n); // calcolo b affinché P sia sulla curva ellittica
                calcola_delta(delta,a,b,n); // calcolo discriminante della curva ellittica (mod n)

                if (mpz_cmp_si(delta,0)==0) goto retry; // se delta=0 cambio a e riprovo

                mcd_binario(d,delta,n);

                if (mpz_cmp_si(d,1)!=0) { // se mcd(delta,n) diverso da 1, ho trovato un divisore proprio (ho escluso mcd=n nella condizione precedente)
                    mpz_clears(x_P,y_P,x_iter,y_iter,a,b,delta,NULL);
                    return 1;
                }

                // inizio i calcoli P --> m*P

                int i,j,outcome;
                for (i=0; i<exp_primes[0]; i++) { // calcolo P--> 2^(e_2)*P

                    outcome=double_elliptic(d,x_P,y_P,x_P,y_P,a,n);
                    if (outcome==-1) goto retry; // se non posso raddoppiare cambio a e riprovo
                    if (outcome==0) {
                        mpz_clears(x_P,y_P,x_iter,y_iter,a,b,delta,NULL);
                        return 1; // successo: ho trovato in d un divisore proprio di n
                    }
                }

```

```
                // calcolo le potenze con i primi della lista maggiori di 2
                for (j=1; j<NUM_PRIMES; j++) {
                    for (i=0; i<exp_primes[j]; i++) { // calcolo P-->k^(e_k)*P, con k primo

                        outcome=multiply_elliptic(d,x_P,y_P,x_P,y_P,primes[j],a,n);
                        if (outcome==-1) goto retry; // se non posso moltiplicare cambio a e riprovo
                        if (outcome==0) {
                            mpz_clears(x_P,y_P,x_iter,y_iter,a,b,delta,NULL);
                            return 1; // successo: ho trovato in d un divisore proprio di n
                        }
                    }
                }

                // se arrivo qui ho calcolato mP senza trovare divisori: cambio a e riprovo

                retry: mpz_add_ui(a,a,1); // sommo 1 ad a e riprovo
            }

            mpz_add_ui(y_iter,y_iter,1); // sommo 1 a y e riprovo
        }

        mpz_add_ui(x_iter,x_iter,1); // sommo 1 a x e riprovo
    }

    // se arrivo qui non ho trovato divisori di n per alcun P e alcun a: restituisco 0

    mpz_clears(x_P,y_P,x_iter,y_iter,a,b,delta,NULL);
    return 0;
}

// tenta di calcolare P-->2P sulla curva ellittica; per farlo serve calcolare l'inverso di 2*y_P mod n, quindi si hanno i casi: se mcd(2y_P,n)=n allora
// restituisce -1 (dovrò cambiare curva); se mcd(2y_P,n)=1 il calcolo può essere portato a termine e restituisco 1; altrimenti ho trovato un divisore
// proprio d di n, e restituisco 0.
int double_elliptic (mpz_t d, mpz_t x_2P, mpz_t y_2P, mpz_t x_P, mpz_t y_P, mpz_t a, mpz_t n) {
    mpz_t y_P2_inv,temp1;
    mpz_inits(y_P2_inv,temp1,NULL);
    mpz_mul_si(temp1,y_P,2);
    inv_mod_mcd(d,y_P2_inv,temp1,n);
    if (mpz_cmp(d,n)==0) { // fallimento: restituisco -1
        mpz_clears(y_P2_inv,temp1,NULL);
        return -1;
    }
    if (mpz_cmp_si(d,1)!=0) { // successo: restituisco 0
        mpz_clears(y_P2_inv,temp1,NULL);
        return 0;
    }
    // calcolo 2P: vedi Koblitz per le formule
    mpz_t temp2,x_P_copy,y_P_copy; // creo copie per non modificare x_P e y_P nei calcoli
    mpz_inits(temp2,x_P_copy,y_P_copy,NULL);
    mpz_set(x_P_copy,x_P); mpz_set(y_P_copy,y_P);
    mpz_mul(temp1,x_P_copy,x_P_copy);
    mpz_mul_si(temp1,temp1,3);
    mpz_add(temp1,temp1,a);
    mpz_fdiv_r(temp1,temp1,n);
    mpz_mul(temp1,temp1,y_P2_inv);
    mpz_fdiv_r(temp1,temp1,n);
    mpz_mul(x_2P,temp1,temp1);
    mpz_fdiv_r(x_2P,x_2P,n);
    mpz_mul_si(temp2,x_P_copy,2);
    mpz_sub(x_2P,x_2P,temp2);
    mpz_fdiv_r(x_2P,x_2P,n);
```

```c
    mpz_sub(temp2,x_P_copy,x_2P);
    mpz_mul(y_2P,temp1,temp2);
    mpz_sub(y_2P,y_2P,y_P_copy);
    mpz_fdiv_r(y_2P,y_2P,n);

    mpz_clears(y_P2_inv,temp1,temp2,x_P_copy,y_P_copy,NULL);
    return 1;
}

// tenta di calcolare P+Q sulla curva ellittica; per farlo serve calcolare l'inverso di x_Q-x_P mod n, quindi si hanno i casi: se mcd(x_Q-x_P,n)=n allora
// restituisce -1 (dovrò cambiare curva); se mcd(x_Q-x_P,n)=1 il calcolo può essere portato a termine e restituisco 1; altrimenti ho trovato un divisore
// proprio d di n, e restituisco 0.
int sum_elliptic (mpz_t d, mpz_t x_sum, mpz_t y_sum, mpz_t x_P, mpz_t y_P, mpz_t x_Q, mpz_t y_Q, mpz_t n) {
    mpz_t denom,temp1;
    mpz_inits(denom,temp1,NULL);
    mpz_sub(temp1,x_Q,x_P);
    inv_mod_mcd(d,denom,temp1,n);
    if (mpz_cmp(d,n)==0) { // fallimento: restituisco -1
        mpz_clears(denom,temp1,NULL);
        return -1;
    }
    if (mpz_cmp_si(d,1)!=0) { // successo: restituisco 0
        mpz_clears(denom,temp1,NULL);
        return 0;
    }
    // calcolo P+Q: vedi Koblitz per le formule
    mpz_t temp2,x_P_copy,y_P_copy,x_Q_copy,y_Q_copy; // creo copie per non modificare le coordinate dei punti nei calcoli
    mpz_inits(temp2,x_P_copy,y_P_copy,x_Q_copy,y_Q_copy,NULL);
    mpz_set(x_P_copy,x_P); mpz_set(y_P_copy,y_P); mpz_set(x_Q_copy,x_Q); mpz_set(y_Q_copy,y_Q);
    mpz_sub(temp1,y_Q_copy,y_P_copy);
    mpz_mul(temp1,temp1,denom);
    mpz_fdiv_r(temp1,temp1,n);
    mpz_mul(x_sum,temp1,temp1);
    mpz_sub(x_sum,x_sum,x_P_copy);
    mpz_sub(x_sum,x_sum,x_Q_copy);
    mpz_fdiv_r(x_sum,x_sum,n);
    mpz_sub(temp2,x_P_copy,x_sum);
    mpz_mul(y_sum,temp1,temp2);
    mpz_sub(y_sum,y_sum,y_P_copy);
    mpz_fdiv_r(y_sum,y_sum,n);

    mpz_clears(denom,temp1,temp2,x_P_copy,y_P_copy,x_Q_copy,y_Q_copy,NULL);
    return 1;
}

// tenta di calcolare P-->kP, con k>2 intero, utilizzando raddoppi ripetuti (analogo a esponenziazione binaria). Nel farlo deve utilizzare le formule
// per raddoppio e per somma su curve ellittiche: se falliscono restituisco -1 e dovrò cambiare curva; se trovo un divisore proprio d di n nel processo
// restituisco 0; altrimenti porto a termine il calcolo e restituisco 1.
int multiply_elliptic (mpz_t d, mpz_t x_kP, mpz_t y_kP, mpz_t x_P, mpz_t y_P, int k, mpz_t a, mpz_t n) {
    int outcome;
    mpz_t temp1,temp2;
    mpz_inits(temp1,temp2,NULL);
    mpz_set(temp1,x_P); mpz_set(temp2,y_P); // temp conterranno man mano le coordinate dei raddoppi successivi di P
    mpz_set_si(x_kP,0); mpz_set_si(y_kP,0);

    while (k>0) {
        if (k&1==1) { // controllo che l'ultimo bit di k sia un 1; se lo è sommo, altrimenti lascio così

            outcome=sum_elliptic(d,x_kP,y_kP,x_kP,y_kP,temp1,temp2,n);
            if (outcome==-1) {
```

```
                mpz_clears(temp1,temp2,NULL);
                return -1; // somma non riuscita: devo cambiare curva
            }
            if (outcome==0) {
                mpz_clears(temp1,temp2,NULL);
                return 0; // successo: ho trovato il divisore d
            }
            // altrimenti ho portato a termine la somma con successo, e proseguo i calcoli
        }
        // aggiorno il raddoppio
        outcome=double_elliptic(d,temp1,temp2,temp1,temp2,a,n);
        if (outcome==-1) {
            mpz_clears(temp1,temp2,NULL);
            return -1; // raddoppio non riuscito: devo cambiare curva
        }
        if (outcome==0) {
            mpz_clears(temp1,temp2,NULL);
            return 0; // successo: ho trovato il divisore d
        }
        // altrimenti ho portato a termine il raddoppio con successo, e proseguo i calcoli

        k>>=1; // shifto di 1 i bit di k
    }
    // se esco dal ciclo ho portato a termine la moltiplicazione P-->kP con successo

    mpz_clears(temp1,temp2,NULL);
    return 1;
}

// calcola in x la stima di Hasse per n: hasse(n)= (radice_quarta(n) + 1)^2
void hasse (mpz_t x, mpz_t n) {
    mpz_root(x,n,4);
    mpz_add_ui(x,x,2); // sommo 2 poiché mpz_root restituisce la radice troncata alla parte intera
    mpz_mul(x,x,x);

    return;
}

// calcola gli esponenti a cui deve essere elevato ciascun primo: exp=parte intera di log_(base p) (hasse_n)
void create_exp_primes (unsigned int* exponents, int* primes, mpz_t hasse_n) {
    int i;
    for (i=0; i<NUM_PRIMES; i++) {
        // trucco per calcolare il logaritmo: mpz_get_str restituisce una stringa contenente la rappresentazione di hasse_n in base
        // primes[i]; per avere la parte intera del logaritmo base p basta quindi calcolare la lunghezza della stringa meno 1
        exponents[i]=strlen(mpz_get_str(NULL,primes[i],hasse_n))-1;
    }

    return;
}

// calcola il termine noto b in Z/(n) tale che il punto (x,y) sia sulla curva ellittica y^2=x^3+ax+b, ossia b=y^2-x^3-ax mod n
void calcola_b (mpz_t b, mpz_t x, mpz_t y, mpz_t a, mpz_t n) {
    mpz_t temp;
    mpz_init(temp);
    mpz_mul(b,y,y);
    mpz_fdiv_r(b,b,n);
    exp_mod_ui(temp,x,3,n);
    mpz_sub(b,b,temp);
    mpz_mul(temp,a,x);
    mpz_sub(b,b,temp);
    mpz_fdiv_r(b,b,n);

```

```c
    mpz_clear(temp);
    return;
}

// calcola il discriminante della curva ellittica: delta= 27*a^3 +4*b^2 mod n
void calcola_delta (mpz_t delta, mpz_t a, mpz_t b, mpz_t n) {
    mpz_t temp;
    mpz_init(temp);
    mpz_mul_ui(delta,a,3);
    exp_mod_ui(delta,delta,3,n);
    mpz_mul_ui(temp,b,2);
    mpz_mul(temp,temp,temp);
    mpz_add(delta,delta,temp);
    mpz_fdiv_r(delta,delta,n);

    mpz_clear(temp);
    return;
}
```

2.5 Logaritmo discreto

Se G è un gruppo ciclico allora, fissati un generatore g ed un elemento h, ci si può chiedere se è possibile determinare in modo efficiente l'elemento $t$ale che

$$g^x = h$$

Nel corso sono stati analizzati tre possibili strategie, riportate nei seguenti algoritmi.

Listing 22: Log discreto babystep giant step.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#pragma warning(disable:4146)
#include <gmp.h>

// Struttura per memorizzare un baby step: (valore, esponente)
typedef struct {
    mpz_t value;            // memorizza a^j mod m
    unsigned long exponent; // esponente j
} Pair;

// Funzione di confronto per qsort e bsearch (ordinamento in base a value)
int cmp_pair(const void* a, const void* b) {
    const Pair* pa = (const Pair*)a;
    const Pair* pb = (const Pair*)b;
    return mpz_cmp(pa->value, pb->value);
}

/*
 * Funzione babyStepGiantStep:
 *  - result: (mpz_t) verra' impostato con la soluzione x, se trovata.
 *  - a, b, m: valori in mpz_t che definiscono l'equazione a^x = b mod m.
 *
 * Restituisce 1 se la soluzione e' trovata, 0 altrimenti.
 */
int babyStepGiantStep(mpz_t result, const mpz_t a, const mpz_t b, const mpz_t m) {
    // Calcoliamo order = m - 1 (nel caso in cui m sia primo)
    mpz_t order;
    mpz_init(order);
```

```c
    mpz_sub_ui(order, m, 1);

    // Calcoliamo n = ceil(sqrt(order))
    double order_d = mpz_get_d(order);
    double n_d = sqrt(order_d);
    unsigned long n = (unsigned long)ceil(n_d);

    // Allocazione dell'array per i baby step (di dimensione n)
    Pair* baby = malloc(n * sizeof(Pair));
    if (!baby) {
        fprintf(stderr, "Errore di allocazione della memoria.\n");
        mpz_clear(order);
        return 0;
    }

    // Inizializziamo i baby step: baby[j].value = a^j mod m, per j = 0,..., n-1.
    mpz_t cur;
    mpz_init_set_ui(cur, 1); // a^0 mod m = 1
    for (unsigned long j = 0; j < n; j++) {
        mpz_init(baby[j].value);
        mpz_set(baby[j].value, cur);
        baby[j].exponent = j;
        // cur = cur * a mod m
        mpz_mul(cur, cur, a);
        mpz_mod(cur, cur, m);
    }
    mpz_clear(cur);

    // Ordiniamo la tabella dei baby step
    qsort(baby, n, sizeof(Pair), cmp_pair);

    // Calcoliamo a^n mod m
    mpz_t a_n;
    mpz_init(a_n);
    mpz_powm_ui(a_n, a, n, m);

    // Calcoliamo factor = (a^n)^{-1} mod m
    mpz_t factor;
    mpz_init(factor);
    if (mpz_invert(factor, a_n, m) == 0) {
        fprintf(stderr, "Inversione modulo fallita.\n");
        for (unsigned long j = 0; j < n; j++) {
            mpz_clear(baby[j].value);
        }
        free(baby);
        mpz_clear(a_n);
        mpz_clear(factor);
        mpz_clear(order);
        return 0;
    }
    mpz_clear(a_n);

    // Impostiamo gamma = b
    mpz_t gamma;
    mpz_init_set(gamma, b);

    // Loop sui giant step: per ogni i = 0,..., n, cerchiamo gamma nei baby step
    for (unsigned long i = 0; i <= n; i++) {
        // Prepariamo una chiave temporanea per la ricerca
        Pair key;
        mpz_init(key.value);
        mpz_set(key.value, gamma);
        key.exponent = 0; // non rilevante

        Pair* found = bsearch(&key, baby, n, sizeof(Pair), cmp_pair);
        mpz_clear(key.value);

        if (found != NULL) {
            // Soluzione trovata: x = i * n + found->exponent
            unsigned long x = i * n + found->exponent;
```

```c
            mpz_set_ui(result, x);

            // Liberiamo la memoria allocata per i baby step
            for (unsigned long j = 0; j < n; j++) {
                mpz_clear(baby[j].value);
            }
            free(baby);
            mpz_clear(gamma);
            mpz_clear(factor);
            mpz_clear(order);
            return 1;
        }
        // Aggiorniamo gamma: gamma = gamma * factor mod m
        mpz_mul(gamma, gamma, factor);
        mpz_mod(gamma, gamma, m);
    }

    // Nessuna soluzione trovata: liberiamo la memoria e restituiamo 0.
    for (unsigned long j = 0; j < n; j++) {
        mpz_clear(baby[j].value);
    }
    free(baby);
    mpz_clear(gamma);
    mpz_clear(factor);
    mpz_clear(order);
    return 0;
}

int main(void) {
    // Inizializziamo le variabili GMP
    mpz_t a, b, m, result;
    mpz_inits(a, b, m, result, NULL);

    // Lettura degli input
    gmp_printf("Inserisci a: ");
    gmp_scanf("%Zd", a);
    gmp_printf("Inserisci b: ");
    gmp_scanf("%Zd", b);
    gmp_printf("Inserisci m: ");
    gmp_scanf("%Zd", m);

    // Calcoliamo il logaritmo discreto
    if (babyStepGiantStep(result, a, b, m)) {
        gmp_printf("Soluzione trovata: x = %Zd\n", result);
    }
    else {
        gmp_printf("Nessuna soluzione trovata per l'equazione %Zd^x = %Zd (mod %Zd).\n", a, b, m);
    }

    // Pulizia
    mpz_clears(a, b, m, result, NULL);
    return 0;
}
```

Listing 23: Log discreto PHS.c

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

```c
// Funzione per calcolare il MCD con algoritmo di Euclide
// usa la libreria GMP (mpz_gcd)
// Algoritmo di Euclide esteso per trovare l'inverso modulo
void mod_inverse(mpz_t result, mpz_t a, mpz_t m) {
    mpz_invert(result, a, m);
}

// Esponenziazione modulare binaria
// calcola base^exp % mod, e utilizza la funzione GMP mpz_pow
void mod_exp(mpz_t result, mpz_t base, mpz_t exp, mpz_t mod) {
    mpz_t b, e;
    mpz_init_set(b, base);
    mpz_init_set(e, exp);
    mpz_set_ui(result, 1);

    while (mpz_cmp_ui(e, 0) > 0) {
        if (mpz_odd_p(e))
            mpz_mul(result, result, b), mpz_mod(result, result, mod);
        mpz_fdiv_q_2exp(e, e, 1);
        mpz_mul(b, b, b), mpz_mod(b, b, mod);
    }

    mpz_clear(b);
    mpz_clear(e);
}

// Algoritmo di Baby-Step Giant-Step per trovare il logaritmo discreto modulo un primo q
// L'algoritmo è una tecnica di ricerca per trovare x tale che: g^x = h (mod p)
// dove g è la base, h è il valore di cui vogliamo trovare il logaritmo, e p è un modulo primo.
// L'algoritmo divide il problema in due parti:
// La baby-step (precalcolo) crea una tabella con i valori di g^(j*m) % p per j da 0 a m, dove m è la radice quadrata di q (approssimato).
// La giant-step calcola successivamente i valori h * g^(-i*m) % p e cerca se uno di questi è nella tabella.
// Se trova una corrispondenza, restituisce la soluzione come la somma degli indici i e j. Se non trova nulla, restituisce 0.
void baby_giant_step(mpz_t result, mpz_t g, mpz_t h, mpz_t p) {
    mpz_t m, val, g_m, inv_g_m, temp;
    mpz_init(m);
    mpz_sqrt(m, p);
    mpz_add_ui(m, m, 1);

    mpz_init_set_ui(val, 1);
    mpz_init(temp);

    // Creazione della tabella dei baby steps
    size_t m_ui = mpz_get_ui(m);
    mpz_t *table = malloc(m_ui * sizeof(mpz_t));
    for (size_t i = 0; i < m_ui; i++) {
        mpz_init_set(table[i], val);
        mpz_mul(val, val, g);
        mpz_mod(val, val, p);
    }

    // Calcola g^(-m) mod p
    mpz_init(g_m);
    mpz_invert(g_m, g, p);
    mpz_init(inv_g_m);
    mod_exp(inv_g_m, g_m, m, p);

    mpz_set(val, h);
    for (size_t i = 0; i < m_ui; i++) {
        for (size_t j = 0; j < m_ui; j++) {
            if (mpz_cmp(table[j], val) == 0) {
                mpz_set_ui(result, i * m_ui + j);
                goto cleanup;
            }
```

```
        }
        mpz_mul(val, val, inv_g_m);
        mpz_mod(val, val, p);
    }
    mpz_set_ui(result, 0);

cleanup:
    for (size_t i = 0; i < m_ui; i++) {
        mpz_clear(table[i]);
    }
    free(table);
    mpz_clears(m, val, g_m, inv_g_m, temp, NULL);
}

// Risoluzione del logaritmo discreto modulo q^e usando il metodo di Pohlig-Hellman
// approccio efficiente quando il modulo è fattorizzato come un prodotto di potenze di primi piccoli
// Questa funzione sfrutta la fattorizzazione del modulo p come un prodotto di potenze di primi.
// La funzione esegue i seguenti passi:
// Calcola q^e (dove q è un primo e e è l'esponente corrispondente alla potenza di q).
// Calcola i valori di g_i e h_i come potenze di g e h modulo p, utilizzando q^e.
// Risolve il logaritmo discreto di h_i rispetto a g_i modulo q^e usando l'algoritmo di Baby-Step Giant-Step.
// Restituisce il risultato parziale del logaritmo discreto modulo q^e.
// Funzione per Pohlig-Hellman per un fattore q^e
void pohlig_hellman(mpz_t result, mpz_t g, mpz_t h, mpz_t p, mpz_t q, int exp) {
    mpz_t q_e, exponent, g_i, h_i;
    mpz_inits(q_e, exponent, g_i, h_i, NULL);

    // q_e = q^exp
    mpz_pow_ui(q_e, q, exp);
    // Calcola l'esponente: (p-1) / q^exp
    mpz_sub_ui(exponent, p, 1);
    mpz_divexact(exponent, exponent, q_e);
    // g_i = g^( (p-1)/q^exp ) mod p, h_i = h^( (p-1)/q^exp ) mod p
    mod_exp(g_i, g, exponent, p);
    mod_exp(h_i, h, exponent, p);

    baby_giant_step(result, g_i, h_i, p);

    mpz_clears(q_e, exponent, g_i, h_i, NULL);
}

// Teorema Cinese del Resto per combinare i risultati modulo q_i^e_i
// questa funzione combina i risultati del logaritmo discreto calcolati per moduli diversi, ottenuti tramite il metodo di Pohlig-Hellman.
// Il teorema permette di risolvere un sistema di congruenze, restituendo un risultato finale modulo M, dove M è il prodotto di tutti i moduli.
// La formula del CRT è la seguente:
// x = sum_i{xi*Mi·inv(Mi)} (mod M)
// dove M_i è il prodotto dei moduli escluso l' i-esimo, e inv(M_i) è l'inverso di M_i modulo il modulo corrente.
void chinese_remainder(mpz_t result, mpz_t *x, mpz_t *m, size_t len) {
    mpz_t M, Mi, inv, sum;
    mpz_init_set_ui(M, 1);
    for (size_t i = 0; i < len; i++) {
        mpz_mul(M, M, m[i]);
    }

    mpz_init_set_ui(sum, 0);
    for (size_t i = 0; i < len; i++) {
        mpz_init(Mi);
        mpz_divexact(Mi, M, m[i]);
        mpz_init(inv);
        mod_inverse(inv, Mi, m[i]);
        mpz_mul(Mi, Mi, inv);
        mpz_mul(Mi, Mi, x[i]);
        mpz_add(sum, sum, Mi);
```

```c
        mpz_clear(Mi);
        mpz_clear(inv);
    }
    mpz_mod(result, sum, M);

    mpz_clears(M, sum, NULL);
}

// Algoritmo di Pohlig-Hellman completo
// Funzione principale per il logaritmo discreto
// utilizza tutte le precedenti per risolvere il logaritmo discreto in modo efficiente.
// Per ogni primo q_i e il corrispondente esponente e_i, calcola il logaritmo discreto modulo q_i^e_i utilizzando il metodo di Pohlig-Hellman.
// Combina i risultati ottenuti usando il Teorema Cinese del Resto.
void pohlig_hellman_algorithm(mpz_t result, mpz_t g, mpz_t h, mpz_t p, mpz_t *q, int *e, size_t len) {
    mpz_t *x = malloc(len * sizeof(mpz_t));
    mpz_t *moduli = malloc(len * sizeof(mpz_t));

    for (size_t i = 0; i < len; i++) {
        mpz_init(moduli[i]);
        mpz_pow_ui(moduli[i], q[i], e[i]);  // moduli[i] = q[i]^(e[i])
        mpz_init(x[i]);
        pohlig_hellman(x[i], g, h, p, q[i], e[i]);
    }

    chinese_remainder(result, x, moduli, len);

    for (size_t i = 0; i < len; i++) {
        mpz_clear(x[i]);
        mpz_clear(moduli[i]);
    }
    free(x);
    free(moduli);
}

// Main
// Legge i valori di g (base), h (target), e p (modulo).
// Fa inserire all'utente la fattorizzazione di p, con una lista di primi q con i loro esponenti e .
// Chiama pohligHellmanAlgorithm() per calcolare il logaritmo discreto.
// Stampa il risultato finale.
int main() {
    mpz_t g, h, p, result;
    mpz_inits(g, h, p, result, NULL);

    printf("Inserisci base g, valore h e modulo p (p primo):\n");
    gmp_scanf("%Zd %Zd %Zd", g, h, p);

    // Calcoliamo l'ordine del gruppo: p - 1
    mpz_t order;
    mpz_init(order);
    mpz_sub_ui(order, p, 1);

    int num_factors;
    printf("Inserisci il numero di fattori distinti nella fattorizzazione di (p-1): ");
    scanf("%d", &num_factors);

    // Allochiamo dinamicamente gli array per i fattori e gli esponenti
    mpz_t *q = malloc(num_factors * sizeof(mpz_t));
    int *exp_array = malloc(num_factors * sizeof(int));

    for (int i = 0; i < num_factors; i++) {
        mpz_init(q[i]);
        printf("Fattore primo #%d: ", i + 1);
        gmp_scanf("%Zd", q[i]);
        printf("Esponente per questo fattore: ");
        scanf("%d", &exp_array[i]);
```

```c
    }

    pohlig_hellman_algorithm(result, g, h, p, q, exp_array, num_factors);
    gmp_printf("Logaritmo discreto x trovato: %Zd\n", result);

    // Pulizia della memoria GMP
    mpz_clear(order);
    mpz_clears(g, h, p, result, NULL);
    for (int i = 0; i < num_factors; i++) {
        mpz_clear(q[i]);
    }
    free(q);
    free(exp_array);

    return 0;
}
```

Listing 24: Log discreto Rho Pollard.c

```c
#include <stdio.h>
#include <stdbool.h>
#include <gmp.h>
#include "..\Include\mcd.h"
#include "..\Include\Bezout.h"

void rho_pollard (mpz_t, mpz_t, mpz_t, mpz_t);
void funz (mpz_t, mpz_t, mpz_t, mpz_t, mpz_t, mpz_t, mpz_t);

int main() {
        mpz_t x, g, h, p;
        mpz_inits(x, g, h, p, NULL);
        printf("Risolvere l'equazione g^x=h  nel gruppo (Z/(p))* (p primo) con i
            seguenti dati:\n");
        printf("g = ");
        gmp_scanf("%Zd", g);
        printf("h = ");
        gmp_scanf("%Zd", h);
        printf("p = ");
        gmp_scanf("%Zd", p);

        rho_pollard(x, g, h, p);
        gmp_printf("La soluzione del log discreto è x = %Zd\n", x);

        mpz_clears(x, g, h, p, NULL);
        return 0;
}

void rho_pollard (mpz_t x, mpz_t g, mpz_t h, mpz_t p) {
        mpz_t gamma, a, b, lambda, alpha, beta, p1, temp_a, temp_b, invb, d, c,
            invc, pd, ad, temp, it;
        mpz_inits(gamma, a, b, lambda, alpha, beta, p1, temp_a, temp_b, invb, d, c,
             invc, pd, ad, temp, it, NULL);

        mpz_set_si(gamma, 1); // poniamo gamma = 1
        mpz_set_si(a, 0); // poniamo a = 0
        mpz_set_si(b, 0); // poniamo b = 0

        mpz_set_si(lambda, 1); // poniamo lambda = 1
        mpz_set_si(alpha, 0); // poniamo alpha = 0
        mpz_set_si(beta, 0); // poniamo beta = 0

```

```c
	mpz_sub_ui(p1, p, 1); // p1 = p - 1;
	mpz_set_si(it, 0);

	bool test=false;

	while (mpz_cmp(it, p)<0) {
		funz(gamma, a, b, p, p1, g, h);
		funz(lambda, alpha, beta, p, p1, g, h);
		funz(lambda, alpha, beta, p, p1, g, h); // si muove al doppio della velocità

		if (mpz_cmp(gamma, lambda)==0) {
			mpz_sub(temp_b, beta, b);
			mpz_sub(temp_a, a, alpha);
			mcd_euclide(d, temp_b, p1);
			if (mpz_cmp_si(d, 1)==0) {
				inv_mod(invb, temp_b, p1); // calcolo l'inverso di temp_b mod p1
				mpz_mul(x, invb, temp_a);
				mpz_mod(x, x, p1);
				break;
			} else {
				mpz_fdiv_q(pd, p1, d); // calcolo p2 = p1 / d
				mpz_fdiv_q(c, temp_b, d);
				mpz_fdiv_q(ad, temp_a, d);
				inv_mod(invc, c, pd); // calcolo l'inverso di c mod p2
				mpz_mul(x, invc, ad);
				mpz_mod(x, x, pd); // unica soluzione mod p1/d (ossia pd)

				// studio a mano le d soluzioni mod p1
				mpz_t i;
				mpz_inits(i, NULL);
				mpz_set_si(i, 0);
				while (mpz_cmp(i, d)<0) {
					if (mpz_cmp_si(i, 0)!=0) {
						mpz_add(x, x, d); // x = x + d
					}
					mpz_powm(temp, g, x, p);
					if (mpz_cmp(temp, h)==0) {
						test=true;
						break;
					}
					mpz_add_ui(i, i, 1);
				}
				mpz_clears(i, NULL);
			}
		}
		if (test==true) {
			break;
		}

		mpz_add_ui(it, it, 1);
	}

	mpz_clears(gamma, a, b, lambda, alpha, beta, p1, temp_a, temp_b, invb, d, c, invc, pd, ad, temp, it, NULL);
	return;
}

void funz (mpz_t x, mpz_t a, mpz_t b, mpz_t p, mpz_t p1, mpz_t g, mpz_t h) {
	mpz_t temp;
	mpz_inits(temp, NULL);
	// suddividiamo il campo in 3 sottoinsiemi di uguale cardinalità con le classi di resto mod 3
	mpz_fdiv_r_ui(temp, x, 3); // temp = x % 3

	if (mpz_cmp_si(temp, 0)==0) {
		mpz_mul(x, x, x); // x = x * x
```

```
                mpz_mod(x, x, p); // riporto x mod p
                mpz_mul_ui(a, a, 2); // a = a * 2
                mpz_mod(a, a, p1); // riporto a mod p-1
                mpz_mul_ui(b, b, 2); // b = b * 2
                mpz_mod(b, b, p1); // riporto b mod p-1
        } else if (mpz_cmp_si(temp, 1)==0) {
                mpz_mul(x, x, g); // x = x * g
                mpz_mod(x, x, p); // riporto x mod p
                mpz_add_ui(a, a, 1); // a = a + 1
                mpz_mod(a, a, p1); // riporto a mod p-1
                // b resta invariato
        } else {
                mpz_mul(x, x, h); // x = x * g
                mpz_mod(x, x, p); // riporto x mod p
                // a resta invariato;
                mpz_add_ui(b, b, 1); // a = a + 1
                mpz_mod(b, b, p1); // riporto a mod p-1
        }

        mpz_clears(temp, NULL);
        return;
}
```

2.6 Sistema a chiave pubblica RSA

La base dei sistemi crittografici a chiave pubblica è il seguente: due persone A e B vogliono scambiarsi un messaggio in modo da evitare che un osservatore esterno E possa intercettarlo e decifrarlo facilmente. Il metodo attualmente usato è quello RSA, il quale trae la sua forza dal fatto che fattorizzare un intero con un numero di cifre elevato (come si può verificare dai codici mostrati in precedenza) è molto laborioso.

Fasi dello scambio:
1) A sceglie due primi p e q molto grandi, calcola $n = pq$, $\varphi(n) = (p-1)(q-1)$, sceglie $e \in (\mathbb{Z}/(\varphi(n))^*$, calcola d tale che $[e][d] = 1$ e rende pubblici n ed e.
2) B sceglie il messaggio $[x] \in \mathbb{Z}/(n)$ con una conversione di dominio pubblico, calcola $[y] = [x^e]\ mod\ n$ e rende pubblico $[y]$.
3) A riceve $[y]$ e ricava $[x^e]^d = [x^{ed}] = [x]$, ottenendo il messaggio inviato da B (per esempio una chiave privata per futuri messaggi).

In questi passaggi E per risalire al messaggio di B non può far altro che scomporre n o calcolare manualmente $\varphi(n)$, azioni equivalenti e quindi ugualmente impegnative.

3 Codici ausiliari

3.1 Radici in Zp

Listing 25: Radici Modulo.h

```
#include <time.h>
#include "Bezout.h"

int radice_mod (mpz_t, mpz_t, mpz_t);

// calcolo in x, se esiste, la radice quadrata di a modulo p, con p primo. Se
    esiste restituisce 1, altrimenti 0.
int radice_mod (mpz_t x, mpz_t a, mpz_t p) {
    mpz_t a_modp;
    mpz_init(a_modp);
    mpz_fdiv_r(a_modp,a,p);

    // caso p=2:
    if (mpz_cmp_si(p,2)==0) {
        if (mpz_cmp_si(a_modp,0)==0) mpz_set_si(x,0); // se a==0 mod 2 la radice è
            0
        else mpz_set_si(x,1); // se a==1 mod 2 la radice è 1
        mpz_clear(a_modp);
        return 1;
    }
    // caso a==0 mod p: la radice è 0
    if (mpz_cmp_si(a_modp,0)==0) {
        mpz_set_si(x,0);
        mpz_clear(a_modp);
        return 1;
    }

    // controllo che a sia un quadrato mod p: se il simbolo di jacobi è diverso da
        1, a non è un quadrato
    int j = mpz_jacobi(a_modp,p);
    if (j!=1) {
        mpz_clear(a_modp);
        return 0;
    }

    // per numeri casuali:
    gmp_randstate_t randstate;
    gmp_randinit_default(randstate);
    unsigned long int t = time(NULL);
    gmp_randseed_ui(randstate, t);

    unsigned long int h;
    mpz_t d,alpha,beta,a_inv,exp,temp;
    mpz_inits(d,alpha,beta,a_inv,temp,NULL);
    inv_mod(a_inv,a,p);

    // scrivo p-1 = 2^h *d, con d dispari
    mpz_sub_ui(d,p,1);
    h = mpz_scan1(d,0);
    mpz_fdiv_q_2exp(d,d,h);

    // cerco un non-quadrato
    do {
        mpz_urandomm(beta,randstate,p); // genero un numero casuale tra 0 e p-1
    } while (mpz_jacobi(beta,p)!=-1);

    exp_mod(beta,beta,d,p); // inizializzo beta=(non-quadrato)^d
    // inizializzo x=a^((d+1)/2), ossia x^2*a^-1=a^d radice 2^(h-1)-esima di 1
    mpz_add_ui(temp,d,1);
    mpz_fdiv_q_2exp(temp,temp,1);
    exp_mod(x,a_modp,temp,p);
```

```
    // inizializzo alpha=x^2*a^-1
    exp_mod(alpha,a_modp,d,p);

    h--;
    mpz_init_set_si(exp,1);
    mpz_mul_2exp(exp,exp,h);

/*  Parto con x=a^((d+1)/2) che so essere tale che alpha=x^2*a^-1 è una radice 2^(h
    -1)-esima di 1 (dove p-1=2^h *d); voglio arrivare a h=1, ossia
    con alpha radice 1-esima di 1 ==> alpha=1, da cui x radice di a (la condizione
        nel ciclo è h>0 e non h>1 poiché ho già diminuito h di 1).
    Ad ogni iterazione so che alpha è radice 2^(h-1)-esima di 1, e testo se alpha è
         anche radice 2^(h-2)-esima di 1 (qui exp=2^(h-1)): se lo è
    proseguo lasciando x e alpha invariati; se invece non lo è, devo correggere x
        che diventa x*beta, dove beta sarà radice 2^h-esima primitiva
    di 1, e aggiornare alpha di conseguenza*/

    while (h>0) {
        // aggiorno h e exp
        h--;
        mpz_fdiv_q_2exp(exp,exp,1);
        exp_mod(temp,alpha,exp,p);
        // testo su alpha
        if (mpz_cmp_si(temp,1)!=0) { // test fallito: aggiorno x e alpha
            mpz_mul(x,x,beta);
            mpz_fdiv_r(x,x,p);

            mpz_mul(alpha,x,x);
            mpz_mul(alpha,alpha,a_inv);
            mpz_fdiv_r(alpha,alpha,p);
        }

        mpz_mul(beta,beta,beta); // aggiorno beta che diventa il suo quadrato
        mpz_fdiv_r(beta,beta,p);
    }
    // quando esco h=0 ossia alpha=x^2*a^-1 è radice 1-esima di 1, quindi x radice
        di a

    mpz_clears(a_modp,d,alpha,beta,a_inv,exp,temp,NULL);
    gmp_randclear(randstate);
    return 1;
}
```

3.2 Sistemi di interi

Listing 26: Sis interi.c

```
#include "..\Include\Matrix.h"

//risolvo un sistema di interi con n incognite e smith

int main() {

        int n = 0;
        printf("Quante incognite ha il sistema? n = ");
        scanf("%d", &n);

        //richiedo i dati per la matrice e il termine noto
        int** matrix = NULL;
        matrix = input_null(matrix, n, n);
```

```c

        for (int i = 0; i < n; i++) {
                for (int j = 0; j < n; j++) {
                        printf("Inserisci l'elemento %d %d della matrice: ", i, j);
                        scanf("%d", &matrix[i][j]);
                }
        }

        int** B = NULL;
        B = input_null(B, n, 1);
        for (int i = 0; i < n; i++) {
                printf("Inserisci il termine noto %d: ", i);
                scanf("%d", &B[i][0]);
        }

        //calcolo la forma normale di Smith della matrice
        int** S = NULL;
        S = input_id(S, n);
        int** S_inv = NULL;
        S_inv = input_id(S_inv, n);
        int** T = NULL;
        T = input_id(T, n);
        int** T_inv = NULL;
        T_inv = input_id(T_inv, n);
        int** D = NULL;
        D = input_null(D, n, n);

        for (int i = 0; i < n; i++) {
                for (int j = 0; j < n; j++) {
                        D[i][j] = matrix[i][j];
                }
        }

        //risolvo Ax = B con la forma normale di Smith

        SmithNormalForm5mat(D, S, T, S_inv, T_inv, n, n);

        //SDT x = B

        //Y = DT^-1 B
        int** Y = NULL;
        Y = mul_matrix(S, n, n, B, n, 1);


        for (int i = 0; i < n; i++) {
                for (int j = 0; j < 1; j++) {
                        if (D[i][i] == 0) {
                                Y[i][j] = 0;
                        }
                        else if (Y[i][j] % D[i][i] != 0) {
                                printf("Il sistema non ha soluzione negli interi\n"
                                    );
                                return 1;
                        }
                        else {
                                Y[i][j] = Y[i][j] / D[i][i];
                        }
                }
        }

        print_matrix(Y, n, 1);

        //X = TY
        int** X = NULL;
        X = mul_matrix(T, n, n, Y, n, 1);

        printf("La soluzione del sistema con matrice A :\n");
        print_matrix(matrix, n, n);
        printf("e termine noto B : \n");
        print_matrix(B, n, 1);
```

```
        printf("e' X:\n");
        print_matrix(X, n, 1);

        return 0;
}
```

3.3 Frazioni Continue

Listing 27: Frazioni continue.h

```
#pragma one
#ifndef Create_Factor
#define Create_Factor

#include <stdio.h>
#include <stdlib.h>
#include <math.h>

int* cont_frac_th(int, int);
int* cont_frac_impl(int, int);

int* cont_frac_th(int r, int n)
{
        int* a = NULL;
        long double x = 0, y = 0, z = 0;
        a = (int*)calloc(n, sizeof(int));
        if(!a)
        {
                printf("Errore di allocazione della memoria");
                return NULL;
        }

        x = sqrtl(r);

        int i = 0;
        while (i < n)
        {
                a[i] = floorl(x);
                y = x - a[i];
                z = 1 / y;
                x = z;
                i++;
        }
        return a;
}

int* cont_frac_impl(int r, int n)
{
        int* a = NULL;
        a = (int*)calloc(n, sizeof(int));
        if(!a)
        {
                printf("Errore di allocazione della memoria");
                return NULL;
        }

        long double x = sqrtl(r);
        a[0] = floorl(x);
        int i = 1, beta = -a[0], gamma = 1, app = 0;

```

```
        while (i < n)
        {
                app = (r - beta * beta) / gamma;
                gamma = app;
                app = 0;
                a[i] = floorl((a[0] - beta) / gamma);
                app = -(a[i] * gamma + beta);
                beta = app;
                app = 0;
                i++;
        }
        return a;
}


#endif
```

3.4 Funzioni ausiliarie

Listing 28: Matrix.h

```
#pragma once
#ifndef Matrix
        #define Matrix

    #include <stdlib.h>
    #include <stdio.h>
    #include <stdbool.h>
    #include "..\Include\Smith.h"

        // Funzioni per la manipolazione delle matrici
        int** mul_matrix(int**, int, int, int**, int, int);
        void print_matrix(int**, int, int);
        void gauss(float**, int);
        int det_matrix_triangular(int**, int);
    int rank_matrix_diag(int**, int, int);
        int** kernel_base(int**, int, int, int*);
        int** link2matrix_same_row(int**, int, int, int**, int, int);
        int rank_matrix(int**, int, int);
        char* matrix_to_json(int**, int); // Funzione per convertire una matrice in
             una stringa JSON per passarla a python


    //Funzione per calcolare la moltiplicazione tra due matrici
        int** mul_matrix(int** matrix1, int row1, int col1, int** matrix2, int row2
            , int col2) {
                if (col1 != row2) {
                        return NULL;
                }
                int** result = NULL;
                result = (int**)calloc(row1, sizeof(int*));
                for (int i = 0; i < row1; i++) {
                        result[i] = (int*)calloc(col2, sizeof(int));
                }
                for (int i = 0; i < row1; i++) {
                        for (int j = 0; j < col2; j++) {
                                for (int k = 0; k < col1; k++) {
                                        result[i][j] += matrix1[i][k] * matrix2[k][
                                            j];
                                }
```

```
                    }
                }
                return result;
        }

    //Funzione per stampare una matrice
    void print_matrix(int** matrix, int row, int col) {
            for (int i = 0; i < row; i++) {
                    for (int j = 0; j < col; j++) {
                            printf("%d ", matrix[i][j]);
                    }
                    printf("\n");
            }
            return;
    }



//Funzione per calcolare l'inversa di una matrice intera
int** invert_matrix_integer(int** matrix, int size) {
    int** A = NULL;
    A = input_null(A, size, size);
    int** inv = NULL;
    inv = input_null(inv, size, size);
    if (!A || !inv) {
        free(A);
        free(inv);
        return NULL;
    }

    // Copia della matrice originale in A e inizializzazione della matrice identità in inv
    for (int i = 0; i < size; i++) {
        for (int j = 0; j < size; j++) {
            A[i][j] = matrix[i][j];
            inv[i][j] = (i == j) ? 1 : 0;
        }
    }


    // Eliminazione di Gauss-Jordan
    for (int i = 0; i < size; i++) {
        // Se il pivot A[i][i] è 0, cerca una riga sottostante da scambiare
        if (A[i][i] == 0) {
            int swapRow = i + 1;
            while (swapRow < size && A[swapRow][i] == 0)
                swapRow++;
            if (swapRow == size) {  // Matrice singolare
                free(A);
                free(inv);
                return NULL;
            }
            swapRows(A, i, swapRow, size);
            swapRows(inv, i, swapRow, size);
        }

        // Per ottenere l'inversa intera, il pivot deve essere 1 o -1
        if (A[i][i] != 1 && A[i][i] != -1) {
            // La matrice non è unimodulare: non possiamo ottenere un'inversa intera
            free(A);
            free(inv);
            return NULL;
        }

        // Se il pivot è -1, moltiplica l'intera riga per -1 per renderlo 1
        if (A[i][i] == -1) {
            for (int j = 0; j < size; j++) {
                A[i][j] = -A[i][j];
                inv[i][j] = -inv[i][j];
```

```c
            }
        }

        // Elimina tutti gli altri elementi nella colonna i
        for (int k = 0; k < size; k++) {
            if (k != i && A[k][i] != 0) {
                int factor = A[k][i];  // in una matrice unimodulare dovrebbe essere più o meno 1
                for (int j = 0; j < size; j++) {
                    A[k][j] -= factor * A[i][j];
                    inv[k][j] -= factor * inv[i][j];
                }
            }
        }
    }

    free(A);
    return inv;
}

//Funzioni per calcolare l'eliminazione di gauss
void gauss(float** matrice, int n) {
    for (int i = 0; i < n; i++) {
        // Pivot
        if (matrice[i][i] == 0) {
            for (int k = i + 1; k < n; k++) {
                if (matrice[k][i] != 0) {
                    // Scambia righe
                    for (int j = 0; j < n; j++) {
                        float temp = matrice[i][j];
                        matrice[i][j] = matrice[k][j];
                        matrice[k][j] = temp;
                    }
                    break;
                }
            }
        }

        // Normalizza la riga pivot
        float pivot = matrice[i][i];
        for (int j = 0; j < n; j++) {
            matrice[i][j] /= pivot;
        }

        // Eliminazione verso il basso
        for (int k = i + 1; k < n; k++) {
            float coeff = matrice[k][i];
            for (int j = 0; j < n; j++) {
                matrice[k][j] -= coeff * matrice[i][j];
            }
        }
    }
}

//Funzione per calcolare Gauss di una matrice rettangolare senza scambiare le righe quindi non viene con le righe in ordine
    void gauss_rectangular(int** matrice, int row, int col) {
            int r = 0;
            for (int i = 0; i < col; i++) {

                    // Pivot
        if (matrice[i][i] == 0) {
            for (int k = i + 1; k < row; k++) {
                if (matrice[k][i] != 0) {
                    printf("matrice[%d][%d] = %d\n", k, i, matrice[k][i]);
                    // Non scambio le righe perchè mi serve sapere quali sono le righe non nulle
                    r = k;
                    break;
                }
```

```
                else
                    return;
            }
        }
        else r = i;
                    // Normalizza la riga pivot
                    int pivot = matrice[r][i];
                    printf("pivot = %d\n", pivot);


                    // Eliminazione verso il basso
                    for (int k = 0; k < row; k++) {
                            if (k == r) continue;
                            double coeff = (double)matrice[k][i]/pivot;
            printf("matrice[%d][%d] = %d\n", k, i, matrice[k][i]);
                            printf("coeff = %lf\n", coeff);
                            for (int j = i; j < col; j++) {
                                    matrice[k][j] -= coeff * matrice[r][j];
                            }
                    }
            }
    }

    //Funzione per calcolare il determinante di una matrice triangolare
    int det_matrix_triangular(int** matrice, int n) {
            int det = 1;
            for (int i = 0; i < n; i++) {
                    det *= matrice[i][i];
            }
            return det;
    }

    //Funzione per calcolare il rango di una matrice diagonale
int rank_matrix_diag(int** matrix, int row, int col) {
    int r = 0;
    for (int i = 0; i < my_min(row, col); i++) {
        if (matrix[i][i] != 0)
            r++;
    }
    return r;
}

    //Funzione per trovare una base del kernel di una matrice
int** kernel_base(int** M, int r, int c, int *n) {
    int** D = NULL;
    int** S = NULL;
    int** T = NULL;

    D = input_null(D, r, c);
    S = input_id(S, r);
    T = input_id(T, c);

    for (int i = 0; i < r; i++) {
        for (int j = 0; j < c; j++) {
            D[i][j] = M[i][j];
        }
    }


    SmithNormalForm(D, S, T, r, c);
    int rk = rank_matrix_diag(D, r, c);

    //calcolo la forma normale di smith e trovo il rango tramite la matrice diagonale
            //la base sarà data ultime rk colonne di T

            int** B = NULL;
            if (c - rk == 0) {
                    free(D);
                    free(S);
```

```
                free(T);
                return B;
            }
    B = input_null(B, c, c - rk);
        for (int i = 0; i < c; i++) {
            for (int j = 0; j < c - rk; j++) {
                B[i][j] = T[i][j + rk];
            }
        }

        //print_matrix(B, c, c - rk);

        free(D);
        free(S);
        free(T);
        *n = c - rk;
        return B;
  }


    //Funzione per concatenare due matrici con lo stesso numero di righe
    int** link2matrix_same_row(int** matrix1, int row1, int col1, int** matrix2, int row2, int col2) {
        int** result = NULL;
        if (row1 != row2) {
            printf("Errore: Le matrici non hanno lo stesso numero di righe.\n");
            return NULL;
        }
        result = input_null(result, row1, col1 + col2);
        for (int i = 0; i < row1; i++) {
            for (int j = 0; j < col1; j++) {
                result[i][j] = matrix1[i][j];
            }
            for (int j = 0; j < col2; j++) {
                result[i][j + col1] = matrix2[i][j];
            }
        }
        return result;
    }

    //Funzione per calcolare il rango di una matrice tramite smith
  //portiamo la amtrice in forma diagonale e calcoliamo il rango della matrice diagonale molto più veloce
    int rank_matrix(int** M, int r, int c) {
    int** D = NULL;
    int** S = NULL;
    int** T = NULL;

    D = input_null(D, r, c);
    S = input_id(S, r);
    T = input_id(T, c);

    for (int i = 0; i < r; i++) {
        for (int j = 0; j < c; j++) {
            D[i][j] = M[i][j];
        }
    }



    SmithNormalForm(D, S, T, r, c);
    int rk = rank_matrix_diag(D, r, c);

    free(D);
        free(S);
        free(T);

        return rk;
    }
```

```c

    char* matrix_to_json(int** matrix, int n) {
        // Calcoliamo una dimensione massima per il buffer.
        // Per ogni numero, assumiamo al massimo 12 caratteri (inclusi segno, cifra
             e separatore).
        int buffer_size = n * (n * 12 + 2) + 2;
        char* buffer = (char*)malloc(buffer_size);
        if (!buffer) {
            perror("Errore nell'allocazione della memoria");
            exit(EXIT_FAILURE);
        }
        int pos = 0;
        pos += snprintf(buffer + pos, buffer_size - pos, "[");
        for (int i = 0; i < n; i++) {
            pos += snprintf(buffer + pos, buffer_size - pos, "[");
            for (int j = 0; j < n; j++) {
                pos += snprintf(buffer + pos, buffer_size - pos, "%d", matrix[i][j
                    ]);
                if (j < n - 1) {
                    pos += snprintf(buffer + pos, buffer_size - pos, ",");
                }
            }
            pos += snprintf(buffer + pos, buffer_size - pos, "]");
            if (i < n - 1) {
                pos += snprintf(buffer + pos, buffer_size - pos, ",");
            }
        }
        pos += snprintf(buffer + pos, buffer_size - pos, "]");
        return buffer;
    }

#endif
```

Listing 29: Fattorizzazione.h

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

// definizione della struttura dei fattori: l'obiettivo e' costruire una lista di
    fattori in modo da poter scrivere un intero
// come prodotto di primi (in ordine), ciascuno con la rispettiva potenza.
typedef struct Factor {
    mpz_t prime;
    unsigned int exponent;
    struct Factor* next;
} Factor;

Factor* add_factor(Factor*, mpz_t, unsigned int);
Factor* cons(Factor*, mpz_t, unsigned int);
void print_factors(Factor*);

// aggiunge in testa alla lista il primo p elevato alla potenza exp
Factor* cons(Factor* factor, mpz_t p, unsigned int exp) {
    Factor* new_factor = (Factor*)malloc(sizeof(Factor));
    mpz_init_set(new_factor->prime,p);
    new_factor->exponent=exp;
    new_factor->next=factor;
    return new_factor;
}

// aggiunge alla lista di fattori "factor", in modo ordinato, il primo p elevato
    alla potenza exp
Factor* add_factor(Factor* factor, mpz_t p, unsigned int exp) {
    // caso in cui sono arrivato a fine lista, oppure ho trovato il primo numero
        primo presente nella lista maggiore di p: aggiungo un fattore
    if (factor==NULL || mpz_cmp(factor->prime,p)>0) {
        return cons(factor,p,exp);
```

```c
    }

    // caso in cui ho trovato nella lista un primo identico a p: sommo l'esponente
    if (mpz_cmp(factor->prime,p)==0) {
        factor->exponent=factor->exponent+exp;
        return factor;
    }

    // proseguo la ricerca in modo ricorsivo
    factor->next=add_factor(factor->next,p,exp);
    // riaggancio i puntatori al ritorno della ricorsione
    return factor;
}

// stampa la lista dei fattori
void print_factors(Factor* factor) {
    if (factor==NULL) return;

    gmp_printf("%Zd^(%u)",factor->prime,factor->exponent);
    factor=factor->next;
    while (factor!=NULL) {
        gmp_printf(" * %Zd^(%u)",factor->prime,factor->exponent);
        factor=factor->next;
    }

    return;
}
```